Express C

User's Guide

Version 3.0

© *ParaSoft* Corporation, 1988, 1989, 1990

All brand and product names are trademarks or registered trademarks of their respective holders.

Copyright © 1988, 1989, 1990
ParaSoft Corporation
2500, E.Foothill Blvd.
Pasadena, CA 91107

Printed in the U.S.A

Table of Contents

Chapter 5: Multitasking 156

Executing multiple processes on individual processors

Chapter 6: Parallel Graphics 174

A simple, portable, parallel graphics system: *Plotix*

Chapter 7: Debugging Parallel Programs 202

Using `ndb`, a source level debugger for parallel programs

© *ParaSoft* Corporation, 1988, 1989, 1990

Overview

What is *Express* and what kind of systems can be built with it?

1 Overview

A major barrier to the effective use of parallel machines is the bewildering array of hardware and software configurations. Every machine is different from all the others - often radically so. There is an enormous proliferation of new computer languages, programming methodologies and operating systems with no accepted standards.

One of the great difficulties for most people in making the transition from sequential to parallel computing is the enormous initial time investment in learning the basics and getting the first program running. It is often found that once this hurdle has been overcome parallel processing can become easy and even, dare one say it, satisfying.

What comprises the initial barrier which makes the entry into parallel processing so hard? While each parallel machine makes it's own claims to be just like something else; "UNIX like operating system", "VAX like architecture", "Compatible with MS-DOS", the cumulative effect of all the small (and sometimes not so small) variations in operating systems, programming models, development tools, etc. makes an overwhelming difference.

Into this chaotic, stumbling environment we introduce *Express*.

> What can it do?
>
> How can it help?
>
> What makes it different or better than anything else?

Why is Express different?

The purpose of this document is to explain some of the things that make *Express* different from other approaches to parallel computing. Before going into specific details, however, one might distinguish *Express* from other systems by saying that it is *solely* concerned with parallel processing. It has nothing to say about operating systems or languages or hardware at all. It is a system that evolved directly from the experiences of people who wrote parallel programs and while advanced concepts of software engineers are included in the system, no abstruse theory motivated its design; merely the desire to produce applications which effectively harness the power of parallel processors.

Simple to learn

Because of its concrete (some might say limited) goals *Express* is very simple to learn and use. Getting started under *Express* is very straightforward - hopefully this will enable more people to get involved in parallel processing which, in turn, will result in the availability of more and better software. Eventually it might even be possible to realize the goals set years ago when parallel processing was first held up as the programming paradigm of the future.

2 What *Express* is NOT

In order to understand how *Express* differs from other parallel processing "solutions" it is probably most important to understand what *Express* doesn't do. In a later section we can then explain just what is available in a much less confusing way.

2.1 An Operating System

The most disheartening thing that can possibly happen to a newcomer to parallel processing is to open the manual and read

"Boot the new, parallel operating system MAGIC_OS"

There you are, seated at your familiar terminal, with your nicely personalized environment under your fingertips and you are told to throw it all away. The manual probably contains some phrase in it which claims that `MAGIC_OS` is "just like UNIX" or similar but you would be wise not to be fooled. At this point you may be faced with learning a whole new set of commands, a new editor, new backup procedures ..... the list is endless. Furthermore you probably have particular software that ran on your old system that you won't be able to use now without some time consuming switching around of operating systems. You are probably going to end up in an environment with no "tools" in which you are expected to generate the world's fastest, and most parallel piece of software.

This is not to decry the efforts of many people who are developing extremely powerful and new operating systems for both conventional and parallel processors. On the contrary, such effort is to be encouraged since it will certainly pave the way to better solutions, one day. On the other hand it is extremely painful for the user to have to make the transition to a new operating system at the same time as the transition to parallel programming.

Leaves existing operating system intact

Express avoids this situation by making absolutely no statements about operating systems. All that is provided are the basic tools to access the parallel processing system at hand *from the native operating system that you started with.*

This point may seem rather subtle at first but may be clarified by considering the evolution of a simple PC based computer, shown schematically in Figure 1. Originally one starts with a normal PC running, for the sake of argument, MS-DOS. Such users are notorious "collectors" of software and tools. Each DOS system has its own little quirks designed to make life easier for its user. Software for many different purposes is probably involved; data-base, spreadsheet, games etc.... all the trappings that make programming pleasant.

In order to achieve greater speeds we now purchase a piece of parallel hardware such as a transputer board. Typically, little software is supplied with such systems so we are now left with a potentially very powerful computer but few tools to actually obtain real speedups.

To progress further one has to bring in software. One option is to acquire `MAGIC_OS` and run it on both the host and the parallel processing hardware. After loading up the new software you are thrown into a situation like that in the bottom right of Figure 1. MS-DOS, with all its familiar commands has disappeared and been replaced by `MAGIC_OS` which also resides on the parallel machine. The mysterious parallel computer has become slightly more usable - it can at least communicate with programs running on the host, although it may be difficult to tell that this is so since the host is now a completely different machine from the PC it began as.

Express adds to the existing commands set of the host

The alternative option is *Express*, shown in the lower left part of Figure 1. In this case the host environment remains just as it was. If you like to use `VEDIT` under MS-DOS then you still can. You develop all *Express* programs within the native environment of the host machine to which a few commands are added to control, debug and analyze programs running in parallel. Only when the parallel codes run do you get involved with the parallel machine. Even then the degree to which you are affected is under your control - if you wish to have half your program run under DOS on the PC while the other half computes in parallel, *Express* will do it for you. Finally the new hardware is revealed as the powerful

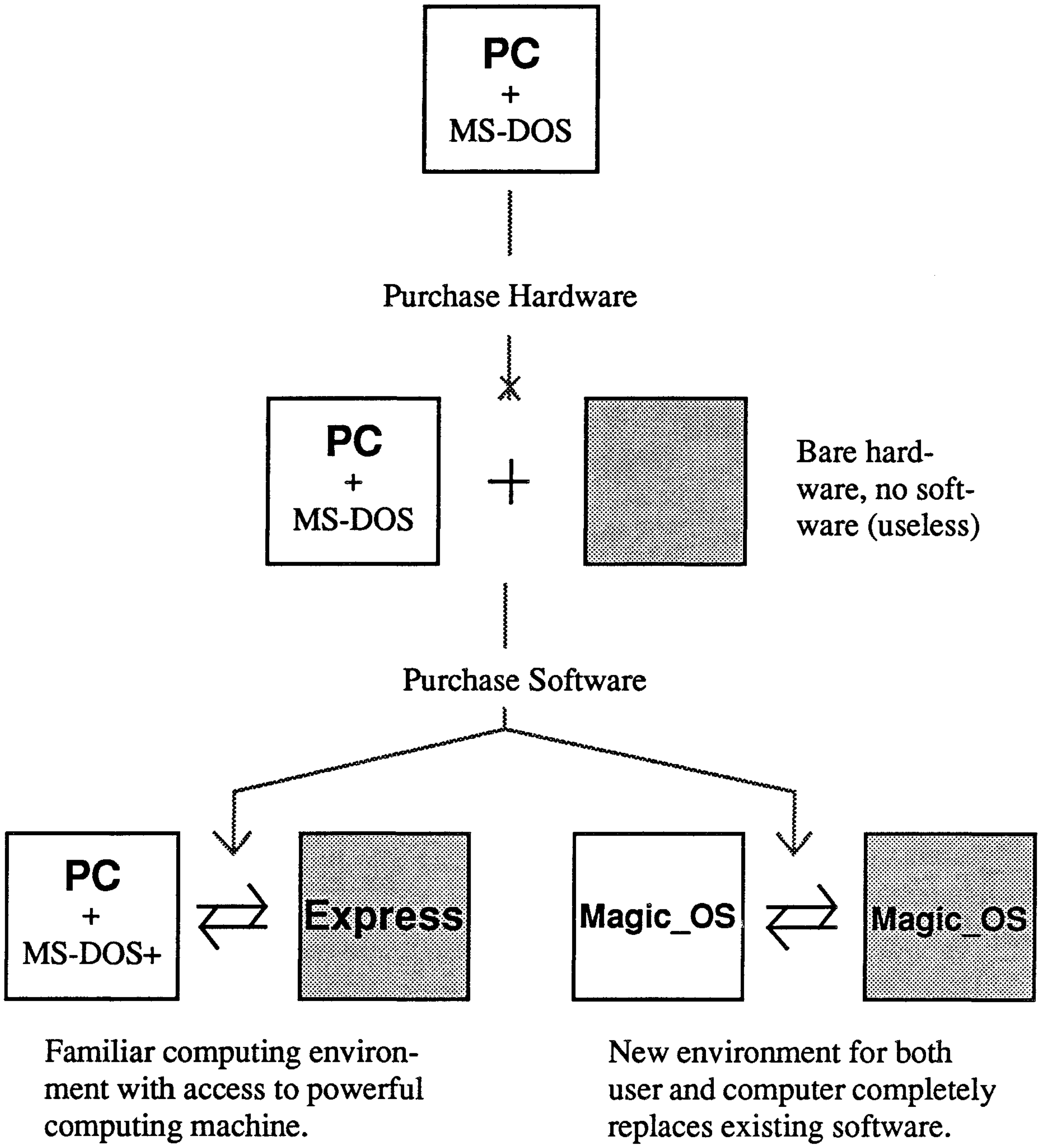

PC
+
MS-DOS
Purchase Hardware
PC
+
MS-DOS
Bare hard-ware, no soft-ware (useless)
Purchase Software
PC
+
MS-DOS+
Express
Magic_OS
Magic_OS
Familiar computing environ-ment with access to powerful computing machine.
New environment for both user and computer completely replaces existing software.

Figure 1. The evolution of a personal computer into a parallel computer

resource it was supposed to be.

Note that this discussion is not specific to MS-DOS. Users of UNIX, VMS and Macintosh are just as loth to leave their customized environments.

2.2 A Language

"Parallel" languages

One of the other areas which has undergone much study in recent years is the search for the "perfect" language which will make parallel programming totally transparent to everybody. This search rivals that for the Golden Fleece or the Holy Grail in terms of its immense value to the world at large; were such a language found life would indeed be easy. Unfortunately, however, similar comments apply in this case as in the last. Since no perfect parallel programming language seems to be available yet why not continue to use old-fashioned languages. After all, vast amounts of code have been written in C and Fortran - why not use it as the basis for development?

Dusty decks

It is surely unreasonable to expect someone with 350,000 lines of code written in Fortran 66 to convert it into some extremely clever parallel language which must be debugged afresh.

Again it becomes a question of how much transition one wants to go through in order to get involved in parallel programming. Just as it is much easier to work in a familiar operating system/environment it is most often best to start from an existing sequential program. In this way the changes necessary to achieve parallelism can be monitored more easily and comparisons are much simpler.

In response to the question of languages *Express* again replies with a stubborn "No comment"; language compilers are indeed provided with *Express* and some of them do in fact have "parallel" extensions but the extent to which these are used is entirely up to the user. If you have a program in Fortran 66 then *Express* will quite happily allow you to parallelize and execute it. If you like OCCAM or concurrent Prolog (and a compiler is available) *Express* will let you use that too!

2.3 The Ultimate Solution

As has already been hinted several times, *Express* is not going to make parallel programming completely automatic. It will not take an existing piece of code and run it N times faster on N processors. On the other hand it DOES provide the tools which allow you a good shot at this sort of performance. By allowing users the freedom to work within their own personal environments with whatever languages are appropriate, the number of "new" features which must be understood before parallel processing can begin is minimized. As a consequence, results are much easier to attain and understand.

Express includes advanced technology and research

On the other hand, *Express* is happy to "incorporate" features from advanced operating systems and languages wherever possible. This allows advanced users more scope in achieving better performance and also provides a solid basis for future research. *Express* also evolves as hardware changes. The system runs on a wide variety of architectures and will continue to expand. It's goal, however, is always to allow maximum performance with the minimum of interference.

3 What is *Express*?

The previous section explained at some length some of the things which *Express* does not do. In this section we will discuss some of the basic concepts underlying what *Express* actually does.

3.1 An "Operating System"

Earlier it was pointed out that *Express* was not an operating system in the conventional sense since the user was able to pursue the quest for performance with all the tools of their native environment be that MS-DOS, UNIX, VMS, Macintosh or any other. In a stricter sense, however, *Express* must be called an operating system since it provides the basic operational functionality for the creation of parallel programs. In an early chapter of the *Express* manuals you will indeed find an instruction to

Express uses conventional compilers from other companies

"Load the *Express* operating system"

The important point to note, however, is that this has absolutely no impact on your familiar working environment. Consistent with *Express*' philosophy it concerns only the parallel processor itself. The *Express* kernel merely provides the basic functionality needed by parallel programs - the ability to communicate, share data, read files, do graphics, get debugged, analyze performance etc. Furthermore, it does so in a totally transparent manner.

The Express kernel and its functions

Many other systems use the word 'transparent' in their documentation and this has come to mean something quite specific - in parallel processing it generally means that no special precautions need to be taken by the parallel program *other than those implicit in the system in use*. In the *Express* context, however, we mean something more - the facilities offered to the parallel program look just like they would to a program running on the host. You don't have to be concerned with a "new operating" system on the parallel machine just as it was of no concern on the host machine. Indeed a programmer familiar with, say, the VMS way of doing things can use the mind-set that the parallel program is running in a VMS environment. It can then access files and use the operating system just as a VMS program might.

Making the parallel computer behave as though it were its host

This, therefore, is the reason for the quotes in the statement that *Express* is an "operating system". In the strictest sense of the phrase it is indeed a distributed operating system for a parallel computer. However it is extremely unobtrusive and has the ability to adapt itself to the requirements of the users of whatever target operating system seems best.

Notice in all of the above that we keep talking about parallel programs *using* the services of the operating system. This is another area in which confusion has arisen. The phrase "operating system" has connotations quite outside the original scope of the phrase. Often included in the concept of what an operating system does is the ability to handle terminals, list directories, print files etc. In the *Express* world these things are all dealt with by whatever operating system is already available. If you are in a UNIX environment then typing '`ls`' at your terminal lists files and '`print`' will print them out in both VMS and MS-DOS. *Express* provides none of these facilities - it merely lets you use whatever was originally there.

Express provides services that are needed to run applications on the parallel computer and as a result can often quite happily co-exist with other "operating systems". Obvious cases are such distributed operating systems as `Mach` or `Helios`. Both are complete in the sense that they allow the user to log in directly to a node of a parallel machine, list and print files etc. In such an environment a user of *Express* would see the basic utilities of the underlying `Mach` or `Helios` while simultaneously being able to take advantage of the parallel processing features and tools provided by *Express*.

Co-existence with "paralleloperating systems"

3.2 A Parallel Processing Toolkit

A parallel processing "toolkit"

In the previous section we explained how *Express* must, strictly speaking be classed as an "Operating System" albeit a highly non-obtrusive one. At the level seen by the user *Express* is best classified as a set of tools and utilities designed for parallel processing. Basically the tools that make up the system are

1. Low level communication primitives for sending messages between processors, peripherals and other system components. This level provides for simple node addressed message passing with a sophisticated "typing" mechanism to differentiate between incoming messages.
2. High level message passing routine which perform a wide variety of common parallel processing tasks including broadcasts, global averaging, global min/max, data redistribution, etc.
3. An automatic "domain decomposition" library which can map problems from the physical domain in which they are naturally expressed to the underlying topology of the parallel computer hardware. This software allows programs to be completely independent of the hardware on which they run and also allows trivial scaling from one to many hundreds of processors by the modification of a run-time parameter.
4. A transparent I/O system. This allows any node in the machine access to the operating system facilities which would normally be available to the host processor. Several "modes" of operation tailored to the particular problem at hand.
5. A parallel graphics system. Device independent, run-time configurable graphics is available to all processors. Both low level vector graphics primitives and higher level packages (contouring, 3-D) are available for a variety of output devices.
6. A totally integrated multitasking system which allows both local and remote task generation through the message system.
7. NDB - a source level debugger for parallel programs. Similar syntax to the popular SUN utility, `dbx`. Allows direct interactive access to program execution on the parallel machine. Breakpoints, stack tracing, printing variables etc. Powerful additions to standard utilities explicitly for parallel processing.
8. PM - A graphical system for evaluating and enhancing the performance of parallel programs. Three styles of profiling are possible covering the entire range of potential bottlenecks. This system allows you to analyze such things as subroutine usage, communication overheads, load balancing, interprocessor timing differences etc., etc.

While this list serves to enumerate the various utilities included in the *Express* package it gives little insight into the functionality and simplicity of the system. The tools have evolved over five years of research into *parallel applications* and represent the wishes/ requirements of a large user community. One particular example might serve to illustrate the point.

Consider a fairly simple problem: calculating the average intensity of the pixels which make up a large X-ray image, in parallel. An obvious approach to this problem is to divide up the image into lots of small areas and allot one to each processor in the system. Each processor then adds up the pixels in its region, combines its results with the other nodes and prints out the result.

A simple problem and its Express solution

Although this seems to be quite a simple problem, getting it right can be quite tricky. You have to figure out how much of the picture is to be given to each node, get the data there and then collect and sum up the results. Furthermore, it would be nice if the code could be structured in such a way that it evolved with the hardware - if you buy four times as many nodes it would be good if the code could run immediately on the larger number of processors.

Express provides facilities for performing all these tasks.

Automatic "domain" decomposition

1. A utility that generates runtime parameters allowing programs to adapt, at runtime to their environment; number of nodes etc.
2. A tool that automatically maps the large image into smaller pieces suitable for distributing to the individual processors. Automatically calculates the size of each piece and its position in the "bigger" picture.
3. An interface to the I/O system that lets the parallel program read the image data directly from a disk file anywhere in the system, with each node getting the correct piece of the input data.
4. A "combine" function that makes the global accumulation of data such as the average intensity quite trivial.

Each of these tools is built upon lower level utilities that are also available to the programmer. Where *Express* differs from other systems, however, is that it does not stop at providing only the low level tools with the disclaimer that "All higher utilities can be generated by the user". While this may be technically correct, *Express* goes all the way by actually providing the extra functionality.

While we typically find that significant applications can be parallelized with only a few subroutine calls the *Express* library is itself quite large. In order to help users find good starting places from which they can build *Express* is supplied with a library of example codes which demonstrate various parallel processing techniques and the utilities needed to make them run efficiently on a distributed machine.

Only a few routines required for real applications

4 How to Run Parallel Programs

The fundamental goal of parallel processing is to achieve computing performance greater than that currently available. The procedure by which this is to be accomplished, however, is not well specified. *Express*, however, allows you maximum flexibility in designing and implementing parallel applications. Before even addressing the problems of "parallel computing" itself we should illustrate the means by which one can attack the problem. In order to demonstrate a couple of the possibilities consider a simple application sketched out in Figure 2.

Sample Application

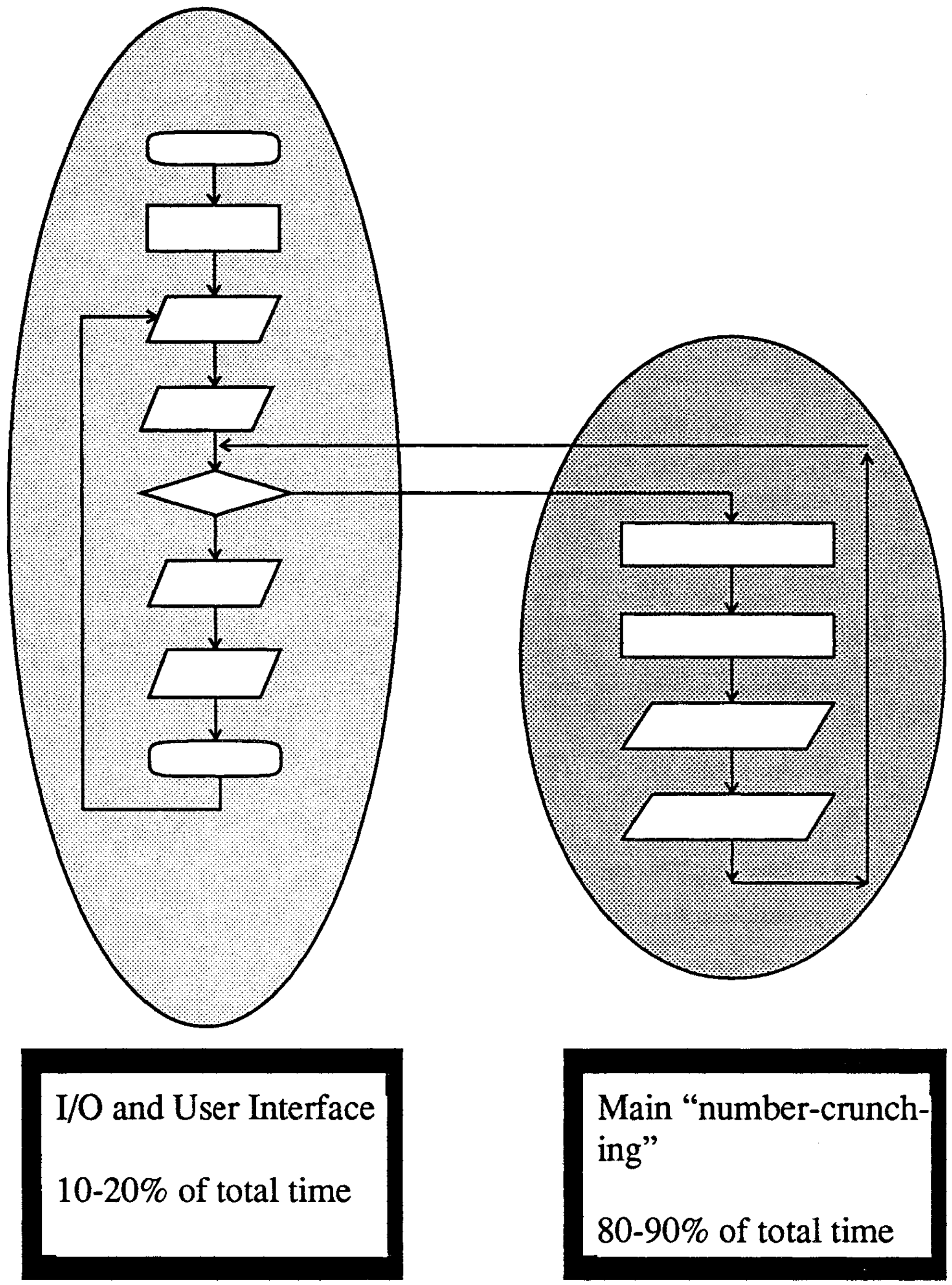


Figure 2. Schematic of typical application code.

Programming models

There are two major components of the program. On the left is the "user interface" consisting of the I/O and graphics-bound pieces of the system. In a data base system this might be the parser that deals with user inquiries and the terminal control system for data entry while in a mechanical engineering problem one might have to read the details of a large structure from a disk file. To the right is the compute-bound phase of the code. To use the two examples from above one might need to search, sort and collect statistics from a large data-base, or calculate the stress-strain relationships of a large bridge. While these examples carry little detail they serve to illustrate two important categories of computing activity in an application.

The most common approach to solving a problem such as these is shown in Figure 3 - the

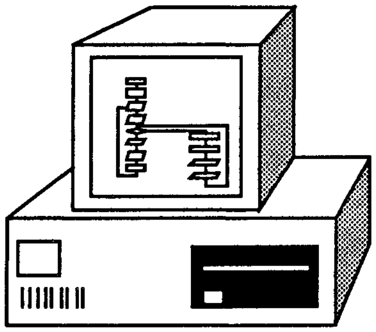


Figure 3. Solution of problem on a single processor

entire program runs on a single computer. This is "sequential" computing and is limited by the speed of the particular computer system in use. Its advantages, on the other hand, are manifest;

Sequential computing

1. You do all your programming in a convenient environment with easy access to a wide variety of programming tools developed over many years.
2. The program is easy to debug using whatever high level tools are available on the host computer
3. Program flow is simple to understand resulting in easy diagnosis of bottlenecks.
4. When finally debugged the resulting code is probably reasonably portable and can be executed on a wide variety of uni-processor systems with various cost/performance payoffs.

Let us now move into the misty world of parallel computing. Suppose that one has purchased some parallel processing system. How best to take advantage of its power? A simple scenario is presented in Figure 4.

Using the conventional system tools available to us in the previous scenario we identify the slowest parts of our application which would, therefore, benefit most from parallelization. With this information in hand we divide up the program into two pieces, one for the host machine and one for the parallel processor. The host deals with the "once-only" aspects of the code such as initialization, cleaning up and any other non-time critical tasks while the hard work is "farmed out" to the parallel processor.

"Host-Node" programming model

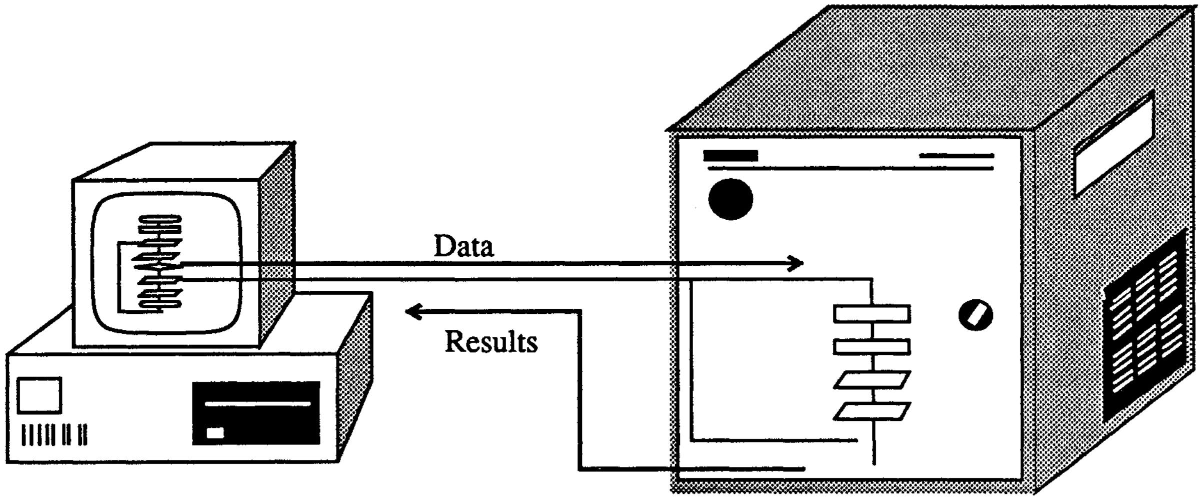


Figure 4. Problem decomposition for host and nodes

What does this scenario have to offer?

1. Significant parts of the original code remain intact and need not even be recompiled since they will run on the host machine. This enables a developer to maintain a complex product but still offer a parallel processing version for enhanced performance.

2. The compute intensive parts of the problem have been given to the "subroutine engine" - the parallel processor. The code to be executed there can be optimized for the parallel processor configuration in use, independent of the program running on the host.

This picture is identical in concept to the idea of "floating point co-processors" such as those found in most high performance systems today. While the host performs whatever work it is best capable of doing, the really hard stuff is sent off to another processor better matched to its requirements. Both processors continue at their own pace and get together to exchange data and results whenever necessary.

In *Express* we refer to this style of computation as "host-node" since both types of computer are involved. One of the most important problems with this style is that one typically has to maintain two programs, one for the host and another for the nodes. These are usually to be compiled in totally different manners and may be quite hard to "glue" back together whenever the program is to be run on a conventional sequential computer. In many cases the requirement to maintain two versions of the same code for parallel and sequential use is too great for many developers.

This style of programming is well suited to a wide class of applications. In particular it provides an easy way to get going on the parallel system by minimizing the amount of material which must be re-worked for the new machine. On the other hand it may have certain disadvantages in the I/O area - each invocation of the subroutine farm may require that large amounts of data be transmitted to the parallel machine which could, in fact, have been generated internally.

Another programming style supported by *Express*, therefore, is shown in Figure 5. In this

The Cubix programming model

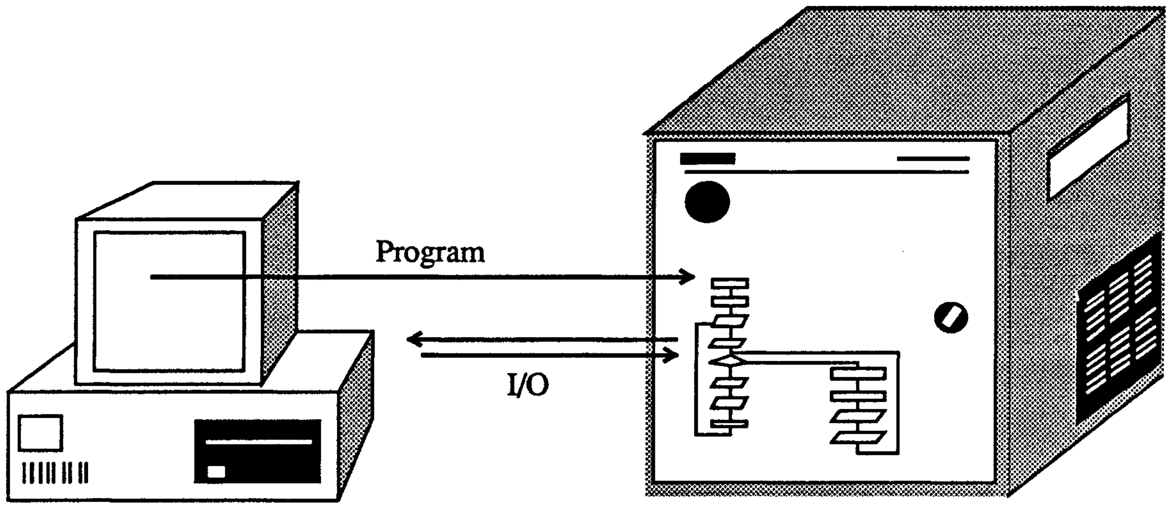


Figure 5. Program decomposed for nodes only

model the entire application is executed on the parallel system. A set of generic server processes are available to handle the I/O, system and graphical needs of the program. An important feature of this style is that the entire application is maintained the same way - you don't have to keep two compilers in mind etc.

Under *Express* this programming style is referred to as the *Cubix* model after the generic server which executes on the host.

Obviously the three figures represent various degrees of the same thing, from purely sequential programming to a style where the entire application runs on the parallel machine. The important point to note is that *Express* enforces no particular style but rather lets you adapt your strategy to the requirements of an application. Among the points which may influence the decision are

Express supports all programming models

- Does the application require too much memory to run entirely on the parallel machine?
- How tightly coupled to the host computer does the application need to be - are there particular I/O devices that need rapid response?
- How much machine specific code resides in the application? If one has devoted significant effort in, for example, machine coding particular parts of the application then these might best still run on the host while other parts of the program run in parallel.
- How important is ease of maintenance?

A very important feature of both styles of programming is that *Express* does not affect the environment of the host machine. This is (obviously) the feature that allows parts of your code to run intact, but is also important since it allows you to develop the code with familiar tools. Editors, disks etc. are all available in the usual manner. Furthermore *Express* provides you with the advanced debugging and performance monitoring facilities that allow you to actually DO effective parallel processing.

An important feature of parallel programs written in *Express* is their machine independence. This leads to two very important conclusions.

Parallel programs written in *Express* can run on machines of any size without software changes. This means, for example, that a code developed on a development system with only one or two processors will execute transparently on 200 nodes with no modifications. The only difference to the user is the hundredfold increase in speed!

Further, parallel programs written in *Express* are independent of the topology of the underlying hardware system. This allows *Express* applications to run on a wide variety of machines; transputer networks, hypercubes, shared memory architectures etc. It also opens the way to "network optimization" - on transputer systems, for example, one can adapt the interprocessor network to the particular program being executed. The advantage of *Express* in this regard is that this optimization can be done *after* the program is running. Normally one would implement and debug the algorithm on some general topology and then, using the performance evaluation software contained in *Express*, modify the network to achieve the best performance. It is not necessary to make the network design decision in advance.

5 Where can I use *Express*?

In the previous sections we discussed a few of the features of *Express* as they pertain to developing parallel programming strategies and getting started with parallel programming. Another important area is that of system design - actually putting together a parallel processing system for your application.

Hardware configurations

The simplest types of system are similar to that shown in Figure 4. One has a single workstation attached to some sort of parallel processing system. This "entry level" system is ideal for dedicated programmers or embedded systems. At the next higher level, however, are bigger systems designed for use by multiple users. After all, parallel programming resources are not so inexpensive that they needn't be shared.

Space-sharing: multiple users

In a timesharing operating system environment such as UNIX or VMS, for example, a simple solution to this problem is shown in Figure 6. Several terminals or workstations can be connected to a single host machine which is itself connected to the parallel computer system. Under *Express* this type of environment can be managed quite simply - users are allocated processors according to the needs of their applications. Each can run in whatever mode is appropriate and the system allocates resources dynamically.

Many hosts - one parallel computer

This system provides simple multi-user access to the power of the parallel computer by taking advantage of the operating system available on the host computer. This is entirely consistent with the *Express* concept under which one uses the capabilities of the host computer rather than replacing them with another totally disjoint operating system.

Under single user systems such as MS-DOS or Macintosh one cannot, however, operate a sharing system in this manner. Instead *Express* supports the "workbench" concept shown in Figure 7. Several user systems are connected to the parallel computing resource *directly*

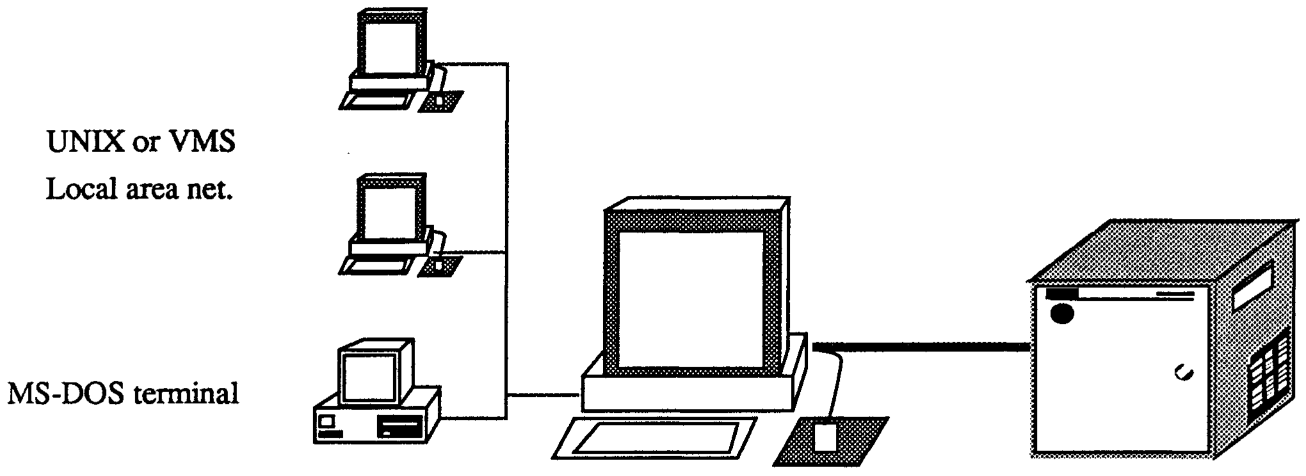


Figure 6. Multi-user access under timesharing operating system

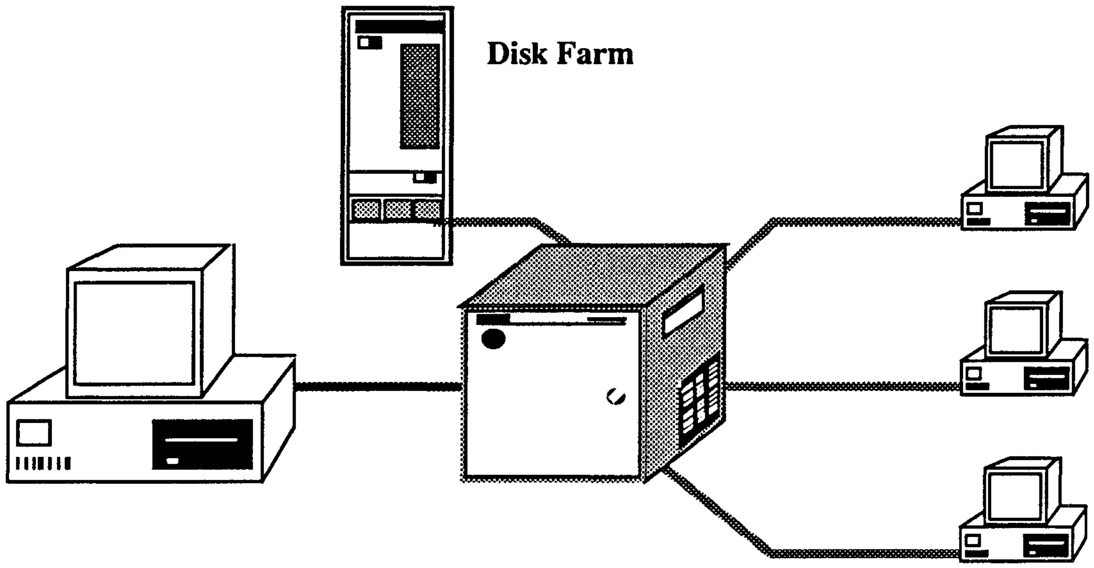


Figure 7. *Express* **Workbench**

rather than through any single host machine. Again *Express* controls access to the resource in a dynamic manner allocating nodes as requested. Such a system might, for example, be used in large data-base project; one machine might control overall system functions such as backing up and starting the system while others are used as data-entry and inquiry sites. Alternatively each could be responsible for an entirely different application.

Disk farms

An important variation of this theme is also indicated in this figure - the possibility of adding additional peripheral devices to the system. In the data-base case just mentioned one might very well require additional disk space. Such devices may be attached anywhere within the system; *Express* is able to locate such peripherals on the basis of system configuration information. Furthermore one of the modes of operation of the various servers allows programs running on the nodes to access information and/or system resources located anywhere within the network. This means, for example, that a program might read input from one file system, send output to another and direct graphical information to special purpose hardware.

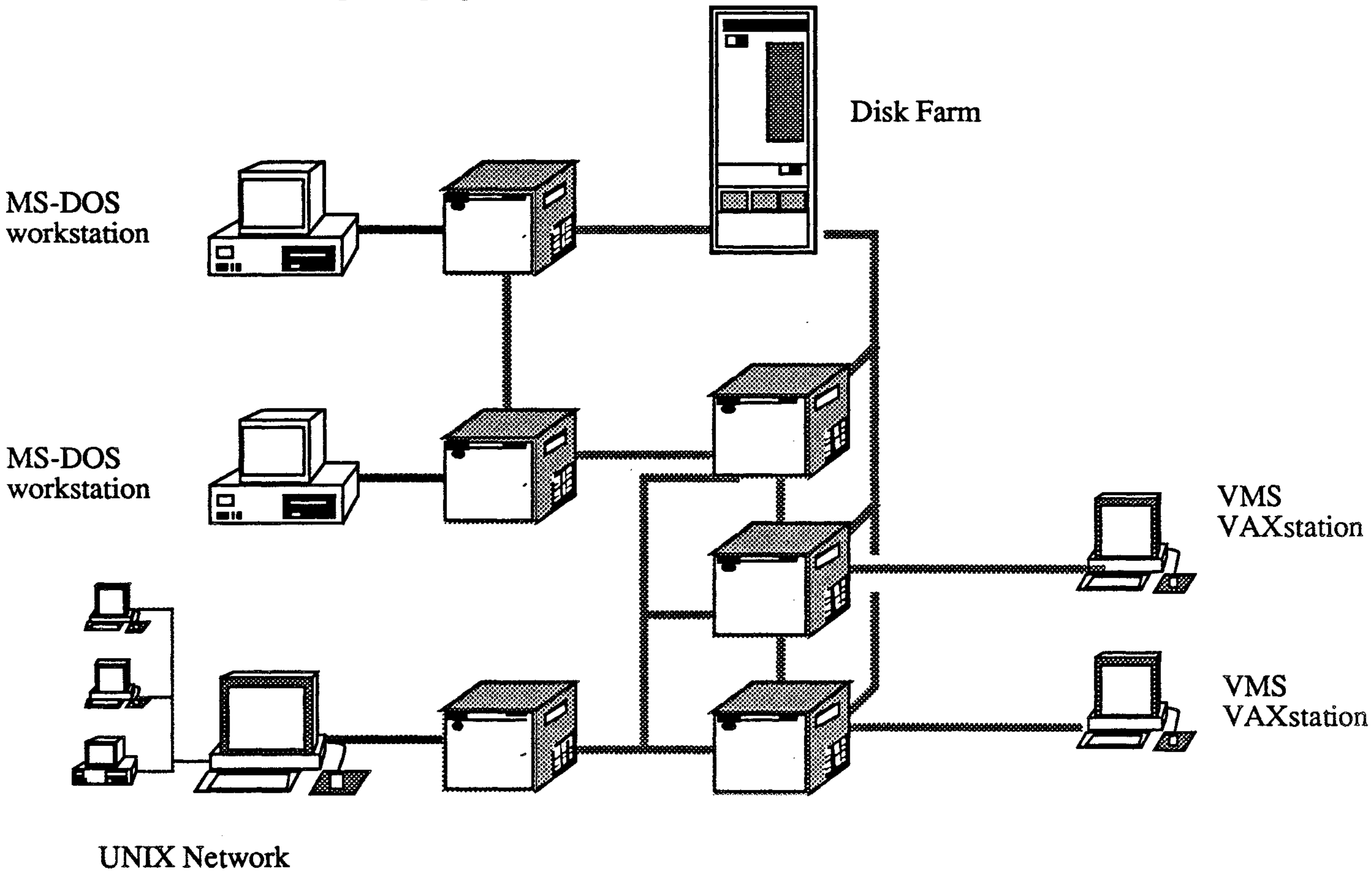


Figure 8. Heterogeneous parallel processing net

The network need not be homogeneous. In figure 8 several types of computer are attached to a centralized parallel processing system. Many types of processor are included which each have access to the parallel computing resources as well as the associated peripherals.

6 Conclusions

Express is a system designed solely to facilitate the process of building and executing parallel programs. Its features include

- Cooperates with existing operating systems to allow users the facility to develop, debug and offer applications running in familiar, highly developed environments.
- Supports a wide variety of programming paradigms; applications may run completely on the parallel machine or pieces may continue to execute on the

host computer. Much of the original code can be left intact.

- The deterministic nature of the parallel processing model allows us to use intuition gained on sequential computers while writing and debugging parallel programs.
- Able to take advantage of hardware/software developments and evolve as new technologies appear.
- Large set of "primitive" operations allow both high-level and low-level use. Designed by users to meet their own needs.
- Semi-automatic decomposition system allows many applications to be parallelized with little effort.
- Guarantees scalability - No recoding necessary to take advantage of more processors.
- Offers the developer the assurance of portability - programs developed under can be executed on many different parallel computer systems.
- High degree of reconfigurability allows multiple users to simultaneously take advantage of parallel processing facilities.
- Support for multiple peripheral devices allows for the construction of specialized networks tailored to application requirements.
- Supports both static and dynamic load balancing of data decomposed problems.

An *Express* tutorial

An introduction to parallel processing with *Express*

1 Introduction

Why parallel computing?

Parallel computing is the technique of using more than one processor at a time to solve a computational problem. This can be anything from controlling a robot arm to updating a spread-sheet or calculating the aerodynamic properties of a new automotive design.

In general we can imagine three reasons for doing parallel processing:

The need for speed

- Speed
  This usually the most important motivation. The original concept of parallel processing was to increase the execution speed of existing and new programs by executing parts of them on different processors. If for some reason the parallel code does not execute faster than its sequential counterpart the whole process of parallelization can be considered a failure.

Big problems require more memory than is available on current supercomputers

- Memory requirements.
  The problems being tackled today in both scientific and business sectors are characterized by their large size. One important limit often reached on conventional supercomputers is the amount of physical memory available. Large scale parallel computers offer the advantage of virtually unlimited memory resources allowing us to attempt problems which would ordinarily be beyond the scope of sequential machines.

Parallel computers use cheaper components than supercomputers

- Cost effectiveness.
  The developments of the last decade in VLSI and other technologies have resulted in dramatic improvements in both the cost and performance of low-priced microprocessors. The so-called "cost-performance ratio" of high-end systems has not improved nearly as much. The result is that purely on the basis of computing power per dollar, CPU's like the Inmos T800 Transputer, the Intel 80*x*86 and i860 and the NCUBE processors are markedly superior in this regard to current mainframes and supercomputers. The crucial question is how to turn the superior cost-performance ratio into sheer performance by simultaneously utilizing multiple processors. This is the goal of parallel computing.

A definition of a parallel computer

An assortment of computer hardware manufacturers have taken up the challenge of building parallel hardware, ranging in size from two to several thousand processors. University and government research groups have demonstrated that these systems can be used to solve real problems in science and engineering in a cost-effective and efficient manner. *ParaSoft* Corporation supplies a software environment for many of these systems which is uniform and easy to use.

The purpose of this chapter is to clarify some of the basic issues in parallel computing, and to de-mystify some of its secrets. It is also a tutorial on the use of *Express*. It is divided into two parts. The first is a low level introductory tutorial which introduces some of the basic concepts in programs which have no real substance. The second part covers more sophisticated examples of *Express* programs which each provide a basis for real applications in both scientific and business fields.

2 Overview

Hardware and software issues

As is generally the case in computer engineering, parallel computing can be divided into two parts: hardware and software. Hardware designers are concerned, above and beyond their usual concerns, with how to connect the processors to one another and to memory. System software designers generally try to hide the hardware decisions from application developers, with the least possible performance penalty, while application software designers are concerned with splitting problems into pieces, *decomposition*, to make effective use of the parallel processor. In the following sections we will consider some general features of parallel hardware, and how an application developer can use the *ParaSoft* data-parallel programming environment, *Express*, to develop his own parallel programs.

2.1 Hardware

Parallel processing hardware

Parallel hardware comprises, by definition, more than one central processing unit or CPU. On the other hand, there is more to a piece of parallel hardware than a collection of CPUs. Otherwise any home with a microwave oven and a programmable VCR might be considered a parallel computer. The extra ingredient, that allows the processors to work together on a single problem, is a communication medium through which data, such as intermediate results, can be communicated.

Shared memory systems

In some systems the processors share a single bank of memory, as in Figure 1. This type of architecture has the appealing property of allowing almost unlimited communication between processors, but this generality is not without its price. It is all too easy to devise incorrect and unreliable communication strategies using such a general mechanism. In addition, it is difficult (and expensive) to design the memory so it is fast enough to keep up with the demands of a large number of processors. Despite these issues *ParaSoft*'s *Express* can easily run on such shared memory systems.

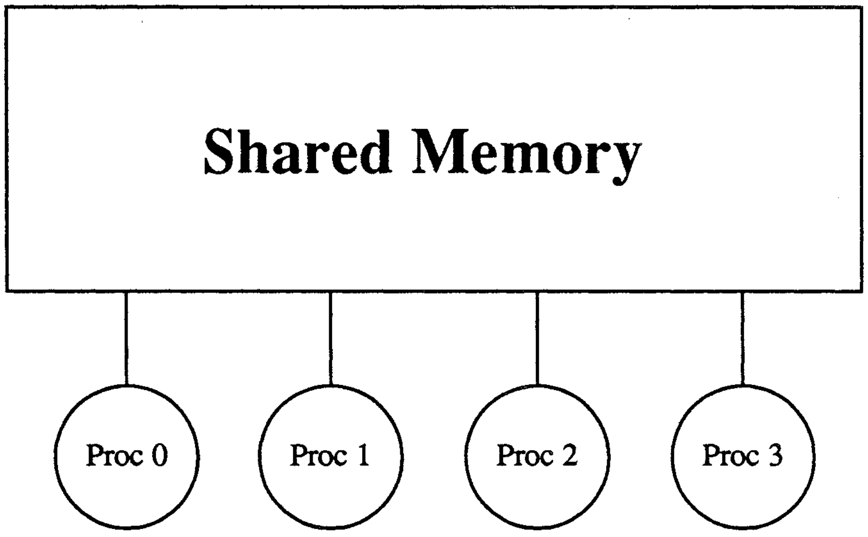


Figure 1. A shared memory architecture.

Another general class of parallel architectures, is shown in Figure 2.This architecture is known as a "distributed memory" system because each processor has its own private memory store, which cannot be accessed directly by any other processor. Only one processor has direct access to any given piece of memory. For the processors to coordinate their efforts in this type of system they must communicate directly by actively sending and receiving information across "links".

Distributed memory machines

Designs of this type are often distinguished by the topology of the interconnection system, e.g., hypercube, two-dimensional torus, etc. These terms refer to the way the individual processors are connected up with one another. It is impractical, with any more than a handful of processors, to connect each and every processor directly to all of the others. Instead, hardware designers provide a few direct connections (typically four to eleven) on each processor, and the ensembles are hooked up in some regular fashion, e.g. as an *n*-dimensional hypercube.

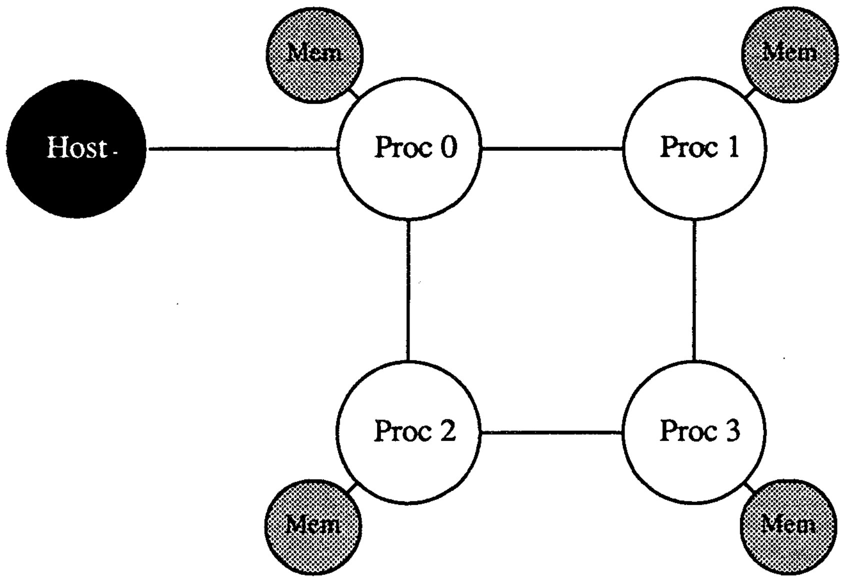


Figure 2. A distributed memory architecture

The types of machines so far described are denoted MIMD (Multiple Instruction Multiple Data) architectures since each processor executes its own instructions and operates on its own data independently of the other nodes. This is not to say, of course, that any given node may or may not have to cooperate with others in the network but this is algorithm dependent rather than being imposed by any hardware model.

MIMD machines

The last class of parallel computers are the SIMD (Single Instruction Multiple Data) machines. This architecture uses of many processors which execute exactly the same instruction (or no instruction at all) at the same time with each processor operating on its own data. Currently *Express* does not run on this type of machine.

SIMD machines

For the rest of this tutorial we will restrict our attention to MIMD machines on many of which *Express* currently runs.

The role of the host processor

An important feature of typical hardware designs is the so-called "host". This processor provides the environment seen by programmers and users of the system, i.e., the operating systems, editors, graphics devices, printers, disks, network services and other features that make up a modern computer system. The host computer is usually a personal computer, workstation, or a mini-computer. The environment is that provided by DOS, Unix, Macintosh, VMS, or whatever operating system is appropriate for the particular host, with a few low-level extensions to allow access to the parallel processor. The nature of these extensions determines the software support for a given parallel computer system and also, to some extent, what types of programs may be written on that machine.

2.2 Software

While parallel computer hardware is available in a large number of forms the associated software is much more varied.

Locking and semaphores for shared memory

Shared memory architectures offer sophisticated "locking" and "semaphore" operations and are currently supporting compilers which automatically parallelize certain program features.

Advanced languages for SIMD machines

Programming SIMD machines typically involves either new languages designed for very fine-grain parallelism or else modifications of existing languages such as *Lisp or C* used on the Connection Machine.

The flexibility of distributed memory MIMD machines

Distributed memory MIMD machines, as might be deduced from their extreme flexibility, support nearly all programming models. This is one of the reasons that they are believed to promise the most in terms of ultimate system performance. Unfortunately it also means that the potential user must choose between a wide variety of software systems.

Fortunately, the *Express* system provides a simple choice.

Express a simple alternative

When programming with *Express* one uses a system comprised of

- A set of compilers for conventional high level languages such as C and FORTRAN
- A library of system calls which provide parallel processing primitives at all levels of sophistication from low level message passing to automatic decomposition and parallelization tools.
- A set of sophisticated support tools including parallel debugging and performance analysis systems and an automatic parallelization tool.

The goal of *Express* is to make programming a parallel computer as much like programming a normal sequential computer as possible. This allows one to use all the knowledge developed over years of programming in a new environment, minimizing the amount of material and/or technique that must be learned from scratch.

Hiding the details of the parallel computer's hardware

One way in which this is done is to hide the details of the parallel computer in use. If you are more than a little confused by the discussion of the previous section which showed some of the types of parallel computer you need not worry since the *Express* model of a parallel computer is that shown in Figure 3. The host and all the nodes are connected to one another through the *Express* system. At the user level no account need be taken of the clever and/or complex ways in which the hardware has been constructed - using the

Express model we can program as though every node were connected to every other.

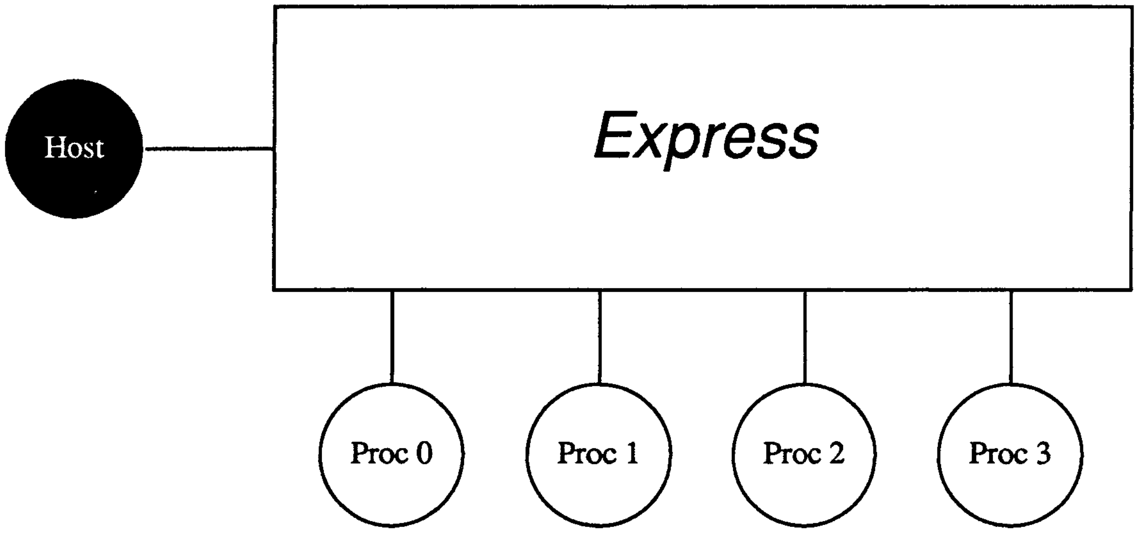


Figure 3. A parallel computer system viewed through *Express*

The Express run-time library

Express also offers an enormous range of parallel processing routines in its runtime library in the hope that whatever operation you may wish to perform has already been supplied. This obviously reduces the amount of "parallel" code that you have to write. Unfortunately, it also increases the apparent complexity of the *Express* system by making the manuals very thick! One of the purposes of this tutorial, therefore, is to point out some of the most commonly used routines and to point you to the manual pages which you will need most.

We should obviously note that there are many other parallel processing models and software systems than *Express*. Each offers its own type of parallelism and encourages different programming styles and thought patterns.We believe, however, that *Express* offers the simplest approach to parallel processing without compromising performance.

2.3 Message Passing

The above section might have left you with the impression that *Express* encompassed *all* parallel processing paradigms. This is partially true - you can write almost every kind of parallel algorithm using the tools provided by *Express*. The one concept that unifies all of the *Express* system, however, is that of "message passing".

A "message passing" programming model

To understand what message passing is all about let us consider a simple model of a bank. The are several tellers at their stations and a single line of customers waiting to be served. Each teller is currently working on a transaction for a client.

This is a good example of parallel processing. Each teller works on transactions for a single client independently of the other tellers. Consider, however, what happens when the teller is done with his customer. Several possibilities exist:

- The teller can do nothing.

- The teller can run round the counter, attract the attention of the next customer and escort them to their station to begin a new transaction.
- Either by voice or some other sign the teller can inform the next customer in line that they are free. The customer walks up to the teller's window and beings his transaction.

No communication usually means that no useful work gets done

The first possibility represents the situation in a parallel processing system when none of the nodes communicates with any of the others - nothing gets done. Eventually, of course, the bank's manager may come round to check on his tellers and, noting that one or more are idle, arrange service for some new customers. This, of course, means that he will have to communicate with both the tellers and the customers. In general we can observe that it is a rare parallel program that can function with absolutely no communication between processors.

The second and third possibilities (and any others that eventually serve all the customers) exhibit a standard feature of parallel processing - how to inform the system that more work can be done by a certain processing unit - in this case the teller wanting another customer.

The shared memory approach leads to bottlenecks and wasted work

The second solution, above, is characteristic of the solution adopted on shared memory machines - i.e., those with an architecture similar to that shown in Figure 1. We assume that the queue of customers is placed in the shared memory and each teller is represented by a node of the machine. As soon as a teller completes a task he goes to the queue in shared memory and fetches the next unit of work, removing it from the queue.

A "message passing" approach

The third solution in the above list is typical of that used on distributed memory machines like that shown in Figure 2. We store the queue of customers in one of the nodes (or even the host) and let the nodes do the teller's work. When a teller is finished he communicates this fact with the machine that contains the queue of customers. The node holding the queue removes the first in line from his list and sends it back to the free teller.

The communication occurring in this last example is what "message passing" is all about. The situation with the two nodes representing the teller and the processor maintaining the customer list is shown in Figure 4. The node on the right sends out a message containing the information "I'm free" to the node managing the customer list. Similarly the managing node sends back a piece of information describing the next customer in line, what transactions he wishes to perform, etc.

The physical structure of a message

Physically a message is a stream of bytes copied from one processor's memory.The teller, for example, could create a character string in memory containing the text "I'm free" and then send this to the managing processor. Similarly the customer queue is probably maintained as some sort of linked data structure. The managing node calculates which part of this structure is required by the teller node and sends it back, re-organizing its internal data structures as necessary to reflect the new "first customer".

Comparing the efficiency of possible parallel solutions

In the above discussion of possible solutions to the problem of assigning the next piece of work it may be observed that the "message passing" solution is the one that gets the job done fastest by minimizing the amount of overhead imposed on the teller in getting the attention of the next customer. This is not coincidental.

The overheads in using shared memory machines tend to be hidden somewhat by the fact

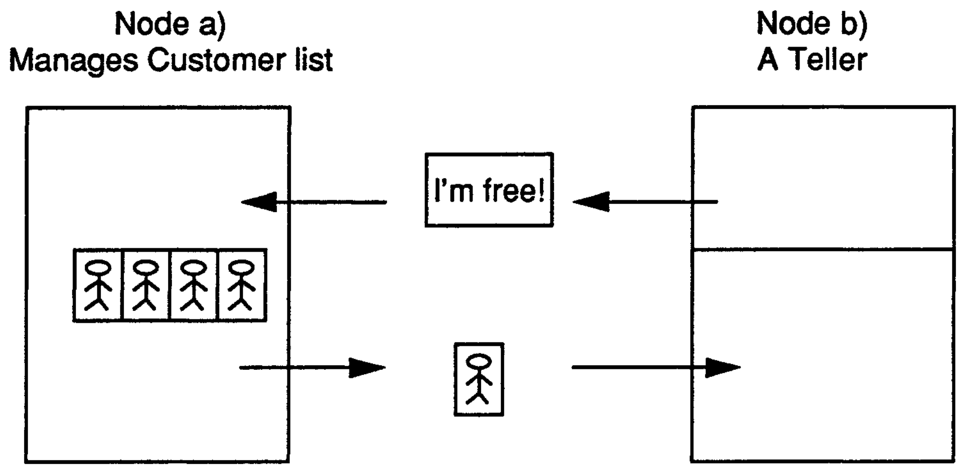


Figure 4. Messages in the banking system

that a piece of code that references memory may, in fact, involve many operations including locking and unlocking semaphores and resolving conflicts with other processors that wish to use the same memory. The problems are made worse by the availability of high-speed caches on advanced architecture machines which means that further decisions have to be made as to whether the shared data can be cached or not.

The distributed memory algorithm, however, is remarkably simple and has the benefit that its overheads can be easily assessed by evaluating the inter node communication speed.

Despite its message passing structure, Express can execute on all types of parallel processors

It is primarily for this reason that *Express* adopts the message passing style of parallel processing. A further reason is that we can easily implement the message passing programming style on shared memory architectures while the converse is very difficult.

It is important to note that, as discussed above, parallel algorithms in *Express* can be evaluated by considering the size and frequency of the internode message traffic. This allows us to make good *a priori* predictions of the performance of our algorithms and also to choose effectively between several possible implementations of the same program.

2.4 *Express* Programming Models.

How many programs should I write?

The first question that crosses the mind of someone contemplating programming a parallel computer is "How do you keep track of which processor is doing what?" Although it is possible to devise more complicated scenarios, the simplest, and most common, organization of *Express* programs is for each and every processor to run exactly the same program. This is not to say that they have the same data, or even that they are executing the same instructions, but only that the compiled program is identical in each processor. Once the program is loaded, the individual processors can, and almost always do, distinguish themselves and start working on different aspects of the problem. For example, in the banking program discussed above one node would probably be distinguished by managing the customer queue.

Another possibility is that different processors execute completely different programs or multiple tasks. This programming model is also supported by *Express*, but because it is more complicated we will not use it in this tutorial. If you wish to learn more about this style of programming you should read about the `expload` function in the *Express* Reference Manual and about the '`-f`' switch in the *Cubix* section of the manual.

Who does the I/O?

Similarly we will concentrate most on the *Cubix* programming model in which we only write code to run on the nodes of the parallel computer. The host processor is taken care of by a "universal host program" which just does what the nodes tell it. This style of programming is the easiest to use and all but one of the examples will be written this way. The sole exception is used to show the trade-offs between this model of computation and that in which we write code for both host and node processors and have the two communicate using the *Express* functions.

Running sequential programs on parallel computers

The whole idea behind this style of *Express* programming is to write one sequential code which can then execute in parallel. In this programming model only one program is needed for the parallel machine. Furthermore this same program will usually run on *any* number of processors and even on different types of parallel processors! This programming model is probably the only one that makes any sense when we imagine programming machines with thousands of nodes. If we had to build a separate program for each one we would rapidly lose our ability to control such a large project.

In this model the advantage obtained from the parallelism occurs when different sets of data are loaded into different processors. Because the data is distributed, each processor has less work to do and the whole program runs faster. In an ideal world the program would run N times faster when running on N processors. In reality this speedup is rarely obtained since the processors usually need to communicate with each other (as in the banking example) and may need to interact with the outside world. Both of these activities reduce the "speed-up" obtained.

The programming model used in this tutorial

The programming model we will be describing in this tutorial, therefore, takes the following form:

- A single program is written and compiled.
- This program is loaded into one or more parallel computer nodes.
- The program begins to execute in each node. For the most part the nodes operate independently on their own data.
- Whenever a node requires more data or whenever its data needs to be updated in some way messages are sent.

The most important benefit of this programming model is that the underlying code is basically the same as would be the case if it were executing on a conventional sequential computer. We can use all our normal intuition about programs when writing, developing and debugging the code. For this reason we can offer the following general piece of advice when writing parallel programs with *Express*:

If you don't know what to do in a particular situation, do what you would do in a sequential program.

It normally works!

3 An Introduction to *Express*

As far as using the *Express* tools and utilities the situation is basically as shown in Figure 5.. The system consists of a host with the parallel machine attached. The host is used

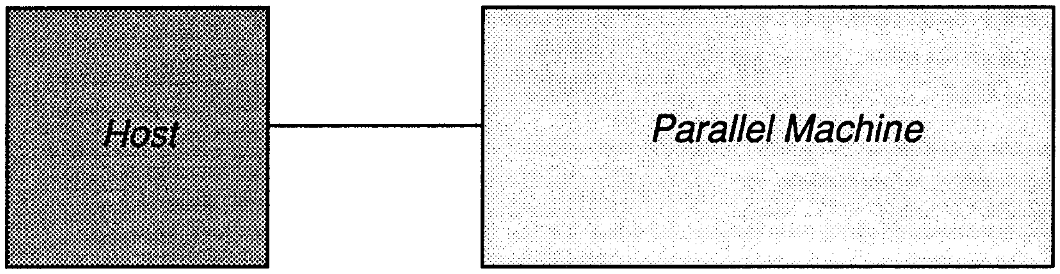


Figure 5. Express world

for program development and is where the various compilers, editors, debuggers etc. are run. This machine executes one of the standard operating systems mentioned before: DOS, UNIX, VMS, Macintosh, etc. The nodes of the parallel machine run the *Express* kernel. An extremely important command, therefore, is the one which loads the *Express* kernel into the parallel computer. This command is `exinit`.

The operating system of the host is unchanged when using Express

This command has to be executed correctly before accessing the parallel machine. Before `exinit` can be executed, however, *Express* must be installed and configured to run on your particular hardware. For information on how to install and configure *Express* refer to the Introductory Guide to *Express* for your system.

Make sure that your hardware is installed and configured correctly before starting this tutorial

Once *Express* has been installed and configured you should load the system by executing the command

```
exinit
```

Normally you do this by typing the above name at the command prompt. In some versions of *Express*, particularly those that run in windowing environments such as MicroSoft Windows or the Macintosh other approaches may be necessary. See the Introductory Guide for more details.

Everything should now be ready for you to write and execute *Express* programs.

The exercises

3.1 The Exercises

The rest of this section is written in the following style. Each exercise is introduced by the following symbol

The general format of the exercises is that the purpose is explained together with the techniques which are being exhibited. This is followed by a description of the technical things that the program should do together with an indication of the manual pages containing the necessary information. This material should suffice for you to write the indicated program yourself. In any case the text of a working program is shown which can, if you wish, be copied onto you machine. We also discuss the execution of the program and any special features of its operation.

Use your Introductory Guide for more information

The comments about the compilation are deliberately vague in this text. All but one of the examples is a simple *Cubix* program and you should be able to find out the details of the compilation process by reading the section "Compiling a first program" of your Introductory Guide. Similarly we will not mention the initialization process again. If your programs have bugs, however, you may need to re-initialize the *Express* kernel with the `exinit` command. This can be done at any time.

Exercise 1. Hello World.

The objective of this exercise is to write a program to print the immortal string "`Hello World`" from the nodes of the parallel processor. This program will execute on any number of nodes and will exhibit some of the features of the *parallel* I/O system built into *Express*.

In order to write this program we can use our intuition about sequential programs as suggested in the preceding comments and just write the code as we normally would for a sequential computer.

No additional manual pages need be consulted for this exercise.

The following is probably similar to the code you should use.

```
#include <stdio.h>

main()
{
    printf("Hello world.\n");
    exit(0);
}
```

Program 1. Code which prints the immortal text "Hello world".

Some rough notes about compiling and linking this program

To make things a little more concrete let's assume that you have written this program in a file called `ex1.c` with some standard editor or word processor. To compile this code for use on the parallel system we have to execute one of the compilers. Furthermore we must tell the system that the program being compiled is to be used in the *Cubix* programming model and should be linked with the appropriate libraries.

While the command to do this varies from system to system a typical version would be

```
        tcc -o ex1 ex1.c -lcubix
```

.. and a few about running it

To execute this program we should again consult our introductory guide - the section called "Running Programs: Cubix or Not?" contains instructions on how to run *Cubix* programs. In general, however, we can execute this program on a single node by typing a command similar to

```
        cubix -n1 ex1
```

In this command `cubix` is the name of *Express* I/O server which will load the program into the parallel machine and start it running. It also performs the I/O and system services requested by the nodes. The '`-n1`' switch indicates how many nodes should be used.

If you run this program you should see, printed on the screen, the text

```
        Hello world.
```

Only one line of output, even when run on 200 nodes

Try running the program on different numbers of processors by changing the value after the '`-n`' switch in the above command. Notice how the string "`Hello world`" appears only once however many nodes we use.

While this program hasn't demonstrated any great parallelism so far it has illustrated some very important points about *Express* and the parallel I/O model.

Running sequential programs in parallel

One of the most powerful features of *Express* is that this program, and virtually any other sequential program that would run on the host processor, can be run on the parallel processor even though it contains system calls that operate on the file system and terminals attached to the host processor. *Express* allows you to execute all of the system calls of the host operating system directly from a parallel program. In addition to the system-call library of the host processor, *Express* provides an implementation of the ANSI C "Standard Library", including features like portable I/O constructs, data conversions, and string handling. Taken together, these features provide an environment for the parallel program that is an extension of the programming environment on the host processor.

While this is an extremely important observation the `ex1` program actually shows more - the parallel features of the *Express* I/O system.

Why there is only one line of output - the Express I/O modes

If you ran the `ex1` program on multiple processors then you probably noticed that the output of the program did not depend on the number of processors which took part in executing the program. One of the most important features of the *Express* I/O system is that files are always in one of three "modes": `single`, `multiple` or `asynchronous`.

These I/O modes are tailored to the general observation that the parallel computer system generally has several computing nodes but only one or two hosts. As a result we encounter three common situations.

Single mode

1. Each processor prints the same message to the host at the same time. This is very common when a program starts and asks for parameters. In this case each program will usually want the same parameters and there is no reason

to see the same prompt from all processors. In this case *Express* prints the message from the first processor, synchronizes all other processors and checks that the message is really the same in all processors. (It aborts if not.) This means that a `printf` statement in this mode is a barrier. It is executed in a "*loosely synchronous*" fashion - each processor waits until all other processors execute this call. This is called "*single*" mode and is the default mode for all files when *Express* programs start. This is why only one line of output is generated from the `ex1` example irrespective of the number of nodes used.

Multiple mode

2. Each processor wants to send different data but all processors wish to contribute. In this case each processor puts data in its internal buffer which is not output until an `fflush` call is executed.This call forces the processors to empty their I/O buffers in order of increasing processor number. This means that data from processor number 0 appears first followed by data from node 1, node 2, etc. Again the `fflush` system call is a barrier requiring a "*loosely synchronous*" call. On the other hand individual "writes" to a file in this mode can be made at will. This mode is called "*multiple*" and is generally used to dump data to files or displays in an orderly fashion.

Asynchronous mode

3. Each processor sends data totally independently. In this mode I/O statements can be executed on different processors at any time and cause output to be sent to the outside world whenever executed. This mode is called "*asynchronous*" and is mostly used for reporting errors. It is difficult to use for backing up/restoring data because the unpredictable order in which data is placed in a file makes it difficult to restore.

As we mentioned before every file in *Express* is in one of these modes.While the above discussion has been mostly concerned with output the modes also apply to input and general system calls. For example single mode input means that the data read on the host will be automatically broadcast to all processors. It is important to note, however, that the I/O mode is associated with each file, rather than the system as a whole. This means that you can have one file in "*multi*" mode while the others are still in "*single*" mode.

I/O modes can also be changed after a file is open. While the default is to open a file in "single" mode the following system calls can be used to modify its behavior

`fsingl(fp)`	sets the indicated file to "*single*" mode.
`fmulti(fp)`	sets the indicated file to "*multiple*" mode.
`fasync(fp)`	sets the indicated file to "*asynchronous*" mode.

Exercise 2. A Parallel Hello World.

If you make no effort to request some kind of parallel behavior, the result of running a program under *Express* will be indistinguishable from running the same program on the host. This leads us to our next example, in which we begin to explore parallel programming.

The objective of the next program is to master the *Express* I/O modes and learn how processors can distinguish each other.

If you wish to write this program yourself it should do the following:

- Read a value from `stdin` in "*single*" mode and print it, also in "*single*" mode.
- Find the unique "processor number" assigned to each node and print it, in "*multi*" mode.
- Read a set of distinct values, one per processor, and print them.

To complete this exercise you will need to understand the various file modes discussed in connection with the previous exercise. More details can be found in the manuals pages for `fmulti`, `fsingl` and `fasync`.

Getting runtime information about the system - `exparam`

To find out how processors can identify themselves we need the `exparam` system call described on the manual page of the same name. This routine requires a single argument which is a pointer to a structure of type `nodenv`, defined in the standard header file `express.h`. It assigns values to various fields in the structure of which the most interesting are `procnum` and `nprocs`. The former is the "processor number" of the node making the call and is used to number the nodes 0, 1, 2, etc. The latter is the total number of nodes taking part in the current execution of the program.

Sample code for this exercise is shown below.

```
#include "express.h"
#include <stdio.h>

struct nodenv env;

main()
{
     int val;
/*
 * First get enviroment information so that we can
 * learn our processor number.
 */
     exparam(&env);
/*
 * Ask for a number - stdout is in singl mode.
 */
     printf("Enter 1 number\n");
/*
 * Now read it - stdin is in singl mode.
 */
     scanf("%d", &val);
/*
 * Print it - stdout is still in singl mode.
 */
```

```
        printf("The number is %d\n",val);
   /*
    * Now ask for a different number for each processor.
    * stdout is still in singl mode.
    */
        printf("Enter %d numbers\n",env.nprocs);
   /*
    * Switch stdin into multi mode and to get
    * a different number in each node.
    */
        fmulti(stdin);
        scanf("%d",&val);
   /*
    * Switch stdout to multi-mode and print value and
    * processor id.
    */
        fmulti(stdout);
        printf("I am processor %d, my value is %d\n",
            env.procnum,val);
   /*
    * Do not forget to flush - stdout is in multi-mode.
    */
        fflush(stdout);
   /*
    * exit
    */
        exit(0);
   }
```

Program 2. Code which reads and prints numbers from different processors.

If you compile and run this program in the same manner as described for the first exercise you will notice an important difference - the "*multi*" mode I/O requests operate in a different way in each node.

Learning more about the I/O system by omitting parts of this code

If you wish there are several important things that can be done with this code to exhibit important features of the I/O modes.

- Try leaving out the call to `fmulti(stdout)` before printing the last message. `cubix` will abort with a message that you have violated the "*loosely synchronous*" constraint. This is because you would have attempted to print out non-identical strings in "*single*" mode. This is a very common error.
- Try leaving out the call to `fflush(stdout)` at the end of the program. Notice how it still runs correctly. This is because the call to `exit` at the end of the code implicitly flushes all open files. Verify that some form of "flush" is necessary by replacing the call to `exit` with a `scanf`. Now the program stops

and waits for your input *before* printing the identifying information from the previous `printf` which is still buffered inside the nodes.

This example illustrates a very important point - we can run sequential programs on parallel computers but need to make small modifications in order to extract the parallel behavior. One of the advantages of *Express* is that the library of utilities available to the programmer is sufficiently large as to make the task fairly straightforward.

Exercise 3. Matrix by Vector Multiplication.

Although it may seem that little has been learned about parallel processing so far the I/O modes in *Express* are powerful enough to allow us to implement a simple matrix-vector multiplication routine.

We will multiply a matrix with *N* rows and *M* columns by a vector with *M* entries. This program will operate on vectors of any size, *M*, and runs on *N* processors. For now this is a real restriction - we cannot use this code to multiply matrixes of order 100 unless we have 100 nodes. Nonetheless this code is quite instructive and when we have learned about the semi-automatic decomposing tools in a later exercise it will be a simple matter to relax the restrictions of this code.

Using the I/O system to distribute data among processors

To solve this problem we need to have some idea of how the data will be distributed among the processors. The process of assigning values to nodes is known as "decomposition" and is of central importance in designing and implementing a parallel algorithm. So important is this issue, in fact, that *Express* provides a library of routines for performing commonly occurring decompositions automatically.

In this case we will distribute the matrix and vector data so that each processor has one row of the matrix and all the vector entries. The resulting decomposition is shown in Figure 5.

The important question in this exercise is how to actually achieve this data distribution - i.e., how can we get the data to the node which is supposed to read it?

The simplest way is by using different I/O modes.

Because the vector is to be duplicated in every node we can read it in "*single*" mode. The matrix, however, need to be distributed cyclically, the first to node 0, the next to node 1, etc. This is exactly the way we would naturally get the data if read in "*multi*" mode.

Once the data is in place we can perform local operations to calculate individual entries in the matrix-vector product which can then be printed in "*multi*" mode.

The basic program outline is, therefore:

- Find how many processors are in use and use this value as *N*, the number of rows in the matrix.
- Prompt for the value, *M*, the length of the vector.
- Read the vector from `stdin` in single mode, automatically generating a copy in each node.
- Read the matrix elements from `stdin` in "*multi*" mode, automatically

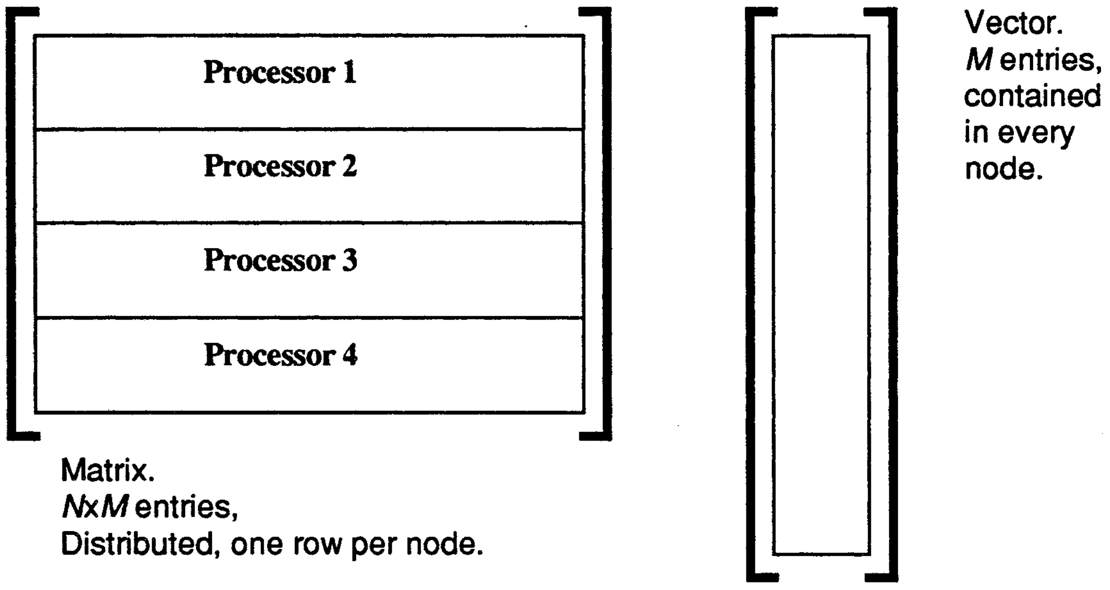


Figure 5. Data distribution for matrix vector multiplication

distributing every N^{th} value to a single processor.

- Perform the local matrix-vector multiplication and send the results to `stdout` in "*multi*" mode.

Since this code uses only those features of *Express* which we have already learned no new manual pages need be consulted to write this code.

Sample code is shown below.

```
#include "express.h"
#include <stdio.h>
#define MAXVEC 20   /* Size of biggest vector */

struct nodenv env;

float vector[MAXVEC], matrix[MAXVEC], value;

main()
{
     int i,size;
/*
 * Get environment
 */
     exparam(&env);
/*
 * First get size of the vector
 */
     printf("Enter size of the vector\n");
/*
 * Now read it.
 */
     scanf("%d", &size);
     if(size >= MAXVEC){
         printf("Vector too big, max is %d\n",MAXVEC);
         exit(1);
     }
/*
 * Ask for vector
 */
     printf("Enter vector numbers\n");
/*
 * Read vector
 */
     for(i=0;i<size;i++) scanf("%f",&vector[i]);
/*
 * Ask for matrix
 */
     printf("Enter %d numbers\n",env.nprocs * size);
/*
 * Switch to multi mode on stdin and read it
 */
     fmulti(stdin);
     for(i=0;i<size;i++)scanf("%f",&matrix[i]);
/*
 * Now data is in processors. Perform multiplication.
 */
     value = 0.0;
     for(i=0;i<size;i++){
         value = value + matrix[i]*vector[i];
     }
/*
 * Switch stdout to multi and print value and
 * processor id.
 */
     fmulti(stdout);
     printf("I am processor %d, my value is %f\n",
         env.procnum,value);
/*
 * Do not forget to flush
 */
     fflush(stdout);
/*
```

```
    * exit
    */
        exit(0);
}
```

Program 3. Code which multiplies matrix by vector, in parallel.

The above code demonstrates an important part of *Express* parallel programming model: the same sequential program is executed by all processors and parallelism is achieved by distributing data and having each node work on a fraction of the whole problem.

Exercise 4. A Parallel Sum.

When writing programs that use the data distribution techniques, just shown, we need to extract some property of a set of data that is distributed across a number of processors. A good example might be an average or maximum of a set of values. Since this type of operation is so common *Express* has a function in its runtime library which generalizes this concept for parallel processors.

"Global" operations greatly simplify common programming tasks

This function, `excombine`, is just one of a set of "global" functions which apply commonly occurring parallel operations to data sets distributed among processors. You might be interested in reading the manual pages for `excombine`, `exconcat`, `exbroadcast` and `exchange` in the Reference Manual for more details and examples of their use.

In this exercise we will demonstrate the use of the `excombine` function by assuming that we need to calculate the global sum of a set of values distributed among the processors.

The basic steps required by the program are

- Read in a data set and distribute it among the nodes.
- Execute `excombine` to make the sum global.
- Print the results.

This function is the first explicit "message passing" routine that we will use. In the previous exercises we managed to create parallel programs merely by exploiting the capabilities of the I/O system. Now we must use the actual routines that implement "message passing".

The arguments to `excombine`

The `excombine` function has seven arguments:

- A pointer to a buffer containing data to be "combined". In our case this will be the local sum which we wish to make global. `Excombine` will overwrite the data pointed to by this argument by the global value obtained by combining with the other nodes.
- A pointer to a function which combines individual data items. This function should be written by the user and is called internally by *Express* to combine elements of the array pointed to by the first argument. In our case this function will add values together.

- The size of an individual data item.
- The number of items to be combined. excombine allows many values to be operated on in a single function call. In our case we wish to combine only our local sums, a single item.
- The number of processors contributing to the global operation. In our case we wish all the nodes to combine their sums so we use the special value ALLNODES defined in the header file express.h.
- A list of processors. This argument is only used if we wish to combine results from less than all the nodes, In our case we can safely make this argument the NULL pointer.
- A pointer to a message type. All *Express* messages carry types and all the message passing functions require this parameter. In general this argument is used to prevent confusion between overlapping messages but since our program will only be sending a single message its value is irrelevant. We could choose any value between 0 and 16383 (inclusive).

The function pointed to by the second argument to excombine is very important since it indicates exactly what operation will be performed when excombine is called.

The user supplied function is the key to the operation of excombine

This function, which must be supplied by the user, will be called with three arguments. The first two are pointers to "items" in the same sense as passed to the original call to excombine. These can be anything from simple integers to complex structures or arrays. The user function must apply some combining operation to these two items, overwriting that pointed to by the first argument. The third argument supplied to the user routine is the "size" value passed as the third argument to excombine.

A final important detail concerns the value which should be returned by the user routine - if this value is different from 0 the system assumes that some sort of error has occurred and aborts the excombine operation.

This discussion has probably made the use of excombine seem very complicated. In practice, however, it is very straightforward as the code for this exercise shows.

```
#include "express.h"
#include <stdio.h>

struct nodenv env;

main()
{
    int val, add();
    int type = 100;
/*
 * First get environment information - processor id.
 */
    exparam(&env);
/*
 * Ask for different number for each processor
 */
     printf("Enter %d numbers\n",env.nprocs);
/*
 * Switch stdin into multi mode and get numbers.
 */
     fmulti(stdin);
     scanf("%d",&val);
/*
 * Call excombine to perform summation
 */
     excombine(&val, add, sizeof(val), 1, ALLNODES,
         (int *)0, &type);
/*
 * Switch stdout to multi and print value and
 * processor id.
 */
     fmulti(stdout);
     printf("I am processor %d, my value is %d\n",
         env.procnum,val);
/*
 * Do not forget to flush
 */
     fflush(stdout);
/*
 * exit
 */
     exit(0);
}

/*
 * Function used for addition
 */
int add(p1,p2,size)
int *p1, *p2, size;
{
/*
 * Perform sumation and save the result in local value
 */
     *p1 += *p2;
     return(0);
}
```

Program 4. Parallel sum

Note that we only combine a single value from each node in this example. In the next exercise we will extend this example to find the average of a large data set on the disk of the host.

An extremely important fact about this program is that no reference has been made to the underlying hardware topology. As a result this program runs on any type of parallel computer with any hardware configuration. This behavior is typical of the high level utilities supplied with *Express* - most common parallel processing operations have corresponding routines in the runtime library making life much simpler for the developer of parallel programs.

This program runs on any parallel architecture and any number of processors

Exercise 5. Averaging the contents of a file.

In this exercise we extend the model of the previous example by performing an average of a set of values contained in a file. While previous exercises have contained some assumptions about where to find the data, how many objects were to be read in, etc. we work hard in this exercise to eliminate any such assumptions and produce a robust piece of parallel code. Of course, the resulting code will still share with its predecessors the ability to execute on any number of processors, independent of the hardware topology or architecture.

In this exercise we will make use of C's conventional command line argument mechanism to pass to our program the name of a file which should be examined. Then, using the techniques of the previous exercise we will read data values from this file and add them to a local sum. Once complete we use `excombine` to add up the local sums and also the total number of items read from the file and use these quantities to build the global average. Note that we make no assumptions about the number of items in the file being exactly divisible by the number of nodes and as a result we cannot make any assumptions about how many items contributed to the local sum on any node - we must use `excombine` to add up these numbers too.

Passing command line arguments to node programs

The basic program outline should be

- Using the command line arguments, open the data-base file containing the items to be read.
- Read single items from the file distributing them individually to the nodes using "*multi*" mode. Take care to deal correctly with the end-of-file condition.
- Use `excombine` to find out how many items have been read.
- Use `excombine` to find the "global" total of the items read and use this to compute the average.
- Print the global average.

The only really new issues in this example concern the passing of command line arguments to node programs. You may wish to read the manual pages for the `exload` functions and also for the `cubix` command to learn how this is done.

If you are unfamiliar with the mechanisms used to detect end-of-file conditions in standard

The Express manuals don't discuss standard C practice

file I/O you may also wish to read about `fscanf` in any good C reference. (This information is not contained in the *Express* manuals since it is standard practice.) The basic method used is to monitor the value returned from `fscanf`. Traditionally this routine returns the number of items read from the input file. Since we are trying to read a single item we can look for any behavior different from this and treat it as the end-of-file condition.

Finally, the method used to pass command line arguments to node programs should be learned by consulting your Introductory guide. In general any text following the name of the program to be loaded in the `cubix` command line will be passed to the node program as `argc` and `argv`. Thus the command

```
cubix -n4 ex5 data.dat
```

normally causes the program `ex5` to be loaded into 4 nodes and passed the following parameters

```
argc = 2
argv[0] = "ex5"
argv[1] = "data.dat"
```

Note that the windowing versions of *Express* work slightly differently - they have spaces in their input dialog boxes in which arguments can be inserted.

```
#include <stdio.h>
#include "express.h"

int type = 100;
     /* Could equally well be any non-negative value */

/*
 * The next two functions are used in calls to excombine
 */
add_int(p1, p2, size)
int *p1, *p2, size;
{
     *p1 += *p2;
     return 0;
}

add_dbl(p1, p2, size)
double *p1, *p2;
int size;
{
     *p1 += *p2;
     return 0;
}
/*
 * Command line arguments can be passed to node programs
 * through the conventional argc, argv mechanism.
 */
main(argc, argv)
int argc;
char *argv[];
{
     FILE *fp;
     double newval, total = 0.;
     int nitems;

     if((fp = fopen(argv[1], "r")) == (FILE *)NULL) {
         perror(argv[1]);
         exit(1);
     }
     fmulti(fp);
/*
 * Read values from the file until fscanf fails.
 */
     nitems = 0;
     while(fscanf(fp, "%lf", &newval) == 1) {
         total += newval;
         nitems++;
     }
/*
 * Now we have the local sum. Find out how many items
 * were read from the file by combining the "nitems"
 * variables.
 */
     excombine(&nitems, add_int, sizeof(nitems), 1,
                 ALLNODES, (int *)0, &type);
/*
 * Similarly get the global total of items read.
 */
     excombine(&total, add_dbl, sizeof(total), 1,
                 ALLNODES, (int *)0, &type);
/*
 * Compute the average and print it out.
 */
     printf("Average: %f\n",total/(double)nitems);
     exit(0);
}
```

Program 5. Global sum of values stored in a disk file

Again it is important to note that this program runs on any number of nodes irrespective of the type of parallel machine in use. The use of the *Express* system calls guarantees hardware independence. Furthermore, the power of the `excombine` function should now be apparent. In this example we were able to use it for two different purposes by merely supplying pointers to two different two-line functions.

Exercise 6. A Host - Node Program.

The restrictions of the Cubix programming model

Up to now all the programs presented have used the *Express* I/O server *Cubix*. As can be seen this type of programming is quite straightforward but it has some restrictions. The most serious of these is that it only provides basic operating system facilities to the node programs - it cannot and does not support all the different types of graphics libraries or every type of data-base interface. Further, by placing all the code for your parallel programs in the nodes of the machine you lose direct control of the host. If your program needs to have direct, low level, access to peripheral devices, for example, another programming model may be more appropriate.

The alternative "host-node" programming model

Express provides for these cases by allowing you to use the "host-node" programming model. In this type of program you extract the compute-intensive aspects of your application and execute them on the parallel computer nodes. The interface or control portions of your code remain on the host computer. The interface between these two program is provided by *Express* function calls which allow data to be transferred between host and nodes as though the host computer were just an additional node in the parallel computer network. In this way an existing piece of code can be maintained almost completely intact - only a small portion is extracted and parallelized.

In this exercise we will construct a "host-node" version of Exercise 4. This will show us some of the features of this programming model and will also illustrate some of the drawbacks associated with this model.

The basic idea of the system is that the host program will allocate a number of nodes in the parallel machine and download the separately compiled node program. It then reads the values to be summed and sends them to the nodes. Finally the host reads back the sum and prints it out. The node program merely waits for values to appear from the host, adds them up and sends back the sum. All communication between host and nodes and among the nodes is done with *Express* system calls.

The basic outline for the host program is, therefore

- Prompt the user to enter the number of nodes to use.
- Allocate this many nodes.
- Download the node program.
- Read data values from the terminal and send one to each node with the *Express* `exwrite` routine.
- Read the sum from the nodes with `exread` and print it.
- De-allocate the nodes.

Allocating and de-allocating nodes; loading programs

The new routines whose manual pages you may wish to examine at this point are `exopen` and `exload` which are responsible for the second and third items above - allocating nodes and loading programs respectively. The routine `exclose` performs the node de-allocation procedure and has its own manual page. The basic node to node communication routines are described in the manual pages for `exread` and `exwrite` and the issue of sending messages to host processors is discussed in the *Express* section of the User's Guide.

The node program must perform the following steps

- Identify processor numbers by calling `exparam`.
- Read values from the host with `exread`.
- Calculate the global sum with `excombine`.
- Node 0 sends back the sum with `exwrite`.

Notice that in the last step only node 0 sends back the reply. This is a typical technique when using the "global" communication functions in this way - if every node were to send the sum back to the host we would have to read many superfluous messages all containing the same information. To prevent this we merely pick out one node to send the message.

The most basic communication routines: `exread` *and* `exwrite`

The new communication routines in this exercise, `exread` and `exwrite` both expect four arguments:

- A pointer to a region of memory containing the data to be sent (`exwrite`) or into which the incoming data should be placed (`exread`).
- The number of bytes to be transmitted (`exwrite`) or the *maximum* number of bytes to be placed in memory (`exread`).
- A pointer to an integer variable containing the processor number of the node to which communication is being performed. To communicate with the host processor we use the special variable `HOST` defined in the header file `express.h`.
- A pointer to an integer variable containing the "message type" to be used for this communication. This value can be any positive number less than 16384 and is used to differentiate between overlapping communication requests. Since this exercise has no such communication we need not worry. Nevertheless we follow the general practice of assigning one message type to the input phase of the program and another to the output messages.

Communicating with a host

You now have all the basic information required about the communication routines required to program this exercise. Unfortunately there is a hidden "catch".

The problems of "host-node" programming

The most tricky problem arising from the "host-node" computation model concerns the fact that the host processor and the node CPU's are rarely the same type of microprocessor. Consider, for example, a transputer system (INMOS) with a Sun workstation or Macintosh host (Motorola), or an NCUBE machine (proprietary chip) with a Sun workstation host. In these cases we have to be concerned with the bit-structure of the quantities that we communicate between the host and node processors.

Different numbers of bytes for variable types such as `int`

The simplest issue to deal with is that of "word length". An integer on the host, for example, is often 16-bits while the nodes most commonly use 32-bit integers. This problem is simply dealt with by choosing data types of matching lengths. Typically floating point numbers already match so the only issue is to use "`long`" variables for communicating integer values since these are normally 32-bits on all machines.

Byte ordering within variables

The second issue concerns the order of the bytes within a word. Machines such as PC's store the least significant byte of a word at the lowest memory address while Motorola microprocessors such as are found in Sun workstations store the most significant byte at the lowest address. The consequence of this is that "byte-swapping" must be performed when communicating data between microprocessors with opposite byte ordering. To help in this task *Express* provides a set of library routines (`_ex_swab`, `_ex_swaw`, `_ex_swad`) to perform these functions in both host and node processors. Note that "byte-swapping" twice results in the same result as not swapping at all so if you are working on a system that requires byte swapping be careful to do it either in the host or in the nodes but not both!

Structure alignment

The last problem is the most complex and deals with structure alignment. The exact layout of the individual bytes in a structure are normally of no interest to C programmers. Unfortunately, if we intend to communicate structures between host and node processors we have to take precautions that both machines lay them out in the same way. Otherwise data that is valid in one node will be garbage in the other. This issue must really be resolved on a case by case basis although we can observe that using only "`long`" integers and floating point values in structures is generally safe since all quantities are then multiples of 32-bits in length.

Cubix has similar problems if we use binary I/O rather than ASCII

Up to now *Cubix* has taken care of all these problems for us because we have been careful to use "formatted" (i.e., ASCII) I/O. In this case all the internal transformations are performed automatically by *Express*. If we had opted to use binary I/O in any of the examples we would have had to face this problem because *Express* does not attempt to "byte-swap" binary data.

I/O from node programs

A last, but VERY IMPORTANT point about the "host-node" programming model is that the node program should not do any I/O. This means that system calls such as `open`, `close`, `read`, `printf`, `scanf` should be restricted to the host program. All I/O to the nodes should actually be performed in the host and the data sent, as messages, to the node program. This restriction is caused by the fact that the `cubix` I/O process which we previously used to run our programs will not normally be running when we execute our "host-node" program and I/O requests cannot be serviced. If your host has a real "multitasking" operating system such as UNIX it is actually possible to have both the `cubix` I/O process *and* your user "host" program run at once. In this case you can have the best of both worlds - parallel I/O from every node *and* the host program. The technique involved in setting up such a system is described on the manual page for `exshare` and is also discussed in the chapter "Multi-host systems" in the User's Guide.

At this point you have all the information necessary to write the "host-node" program. Unfortunately the complications do not end here. The procedures necessary to compile and link this code are also different from those so far encountered.

Compiling the node part of "host-node" programs

For the node program the process is quite similar to that already used and you should be able to find the details in the Introductory Guide. Basically the only change is to omit the switch which previously linked in the *Cubix* libraries. Thus a command which was previously

```
        tcc -o foo foo.c -lcubix
```

would probably become

```
        tcc -o foo foo.c
```

in which no "library" switches are given. This is a very important point - if you accidentally link in the *Cubix* libraries you will generate a program that won't run.

Compiling the host part of "host-node" programs

Compiling the host program is fairly straightforward. Essentially you use the conventional compiler/linker that would be used to generate any other program running on the host but link in special libraries containing the *Express* interface routines. The Introductory Guide has all the details.

Finally we execute the host program just as we would any other program running on the host computer.

In this section we actually have two program pieces to present since the host and node parts of the program will be compiled and linked separately. Since the node program is quite similar to those already presented we show it first.

```
/*
 * NODE PROGRAM for global addition of a set of values
 * sent from the host processor.
 */
#include "express.h"
#define INT32   long

struct nodenv env;

main()
{
     INT32 val;
     int add();
     int type = 101,src = HOST, dest = HOST;
/*
 * First get enviroment information - processor id.
 */
     exparam(&env);
/*
 * Read data from the host
 */
     exread(&val,sizeof(val),&src,&type);
/*
 * Call excombine to perform summation
 */
     type = 100;
     excombine(&val, add, sizeof(val), 1, ALLNODES,
          (int *)0, &type);
/*
 * Send result to the host if node 0
 */
        if(env.procnum == 0){
            type = 102;
            exwrite(&val,sizeof(val),&dest, &type);
        };
/*
 * exit
 */
        exit(0);
}

/*
 * Function used for addition
 */
int add(p1,p2,size)
int *p1, *p2, size;
{
        *p1 += *p2;
        return 0;
}
```

Program 6. Node program for parallel sum

Replacing I/O statements with communication calls - a common scheme in "host-node" programs

Most of this code should look fairly familiar. The new features are the calls to `exread` and `exwrite` which replace the calls to `scanf` and `printf` in the *Cubix* version of this code. They are now responsible for getting the data to be combined from the host and sending back the sum.

The other interesting point is the use of the `INT32` macro to define the types of the variables used for communication between host and nodes. As indicated previously we must carefully match the length of the data objects used in the two processors and so we choose the "`long`" type for this exercise.

The code for the host program has a fair amount of superficial complexity associated with the (de)allocation of the processors which will perform the calculations for us. It is shown below.

```
/*
 * HOST PROGRAM for calculating the sum of a set of
 * values by sending them to the nodes.
 */
#include "express.h"
#include <stdio.h>
#define INT32 long
```

```
char *dev = "/dev/transputer";
char *node = "node";

main()
{
     INT32 val;
     int src = 0, nprocs,i, fd;
     int type = 101;
/*
 * First get number of processors
 */
     printf("\nHow many processors do you want? ");
     scanf("%d",&nprocs);
/*
 * Get access to parallel machine
 */
     if((fd = exopen(dev, nprocs, DONTCARE)) < 0){
         printf("Failed to access %d nodes\n",nprocs);
         exit(1);
     }
/*
 * Load node program to parallel machine
 */
     if(exload(fd, node) < 0){
         printf("Failed to load program %s\n",node);
         exit(2);
     }
/*
 * Ask for different number for each processor
 * and send to processors
 */
     printf("Enter %d numbers\n",nprocs);
     for(i=0;i<nprocs;i++){
         scanf("%ld",&val);
/*
 * Swap bytes if necessary. Asume INT32 is 4 bytes
 */
#ifdef SWAP
         _ex_swaw(&val, &val, sizeof(val));
#endif
         exwrite(&val, sizeof(val), &i, &type);
     }
/*
 * Read data from processor 0
 */
     type = 102;
```

```
        exread(&val,sizeof(val),&src, &type);
/*
 * Swap bytes if necessary. Asume INT is 4 bytes
 */
#ifdef SWAP
        _ex_swaw(&val, &val, sizeof(val));
#endif
/*
 * Print the result
 */
        printf("The value is %ld\n",val);
/*
 * Do not forget to release nodes
 */
        exclose(fd);
/*
 * exit
 */
        exit(0);
}
```

Program 7. Host program for parallel sum

Specifying the name of the hardware device to use

Important features of the host program are the allocation of the processors with the `exopen` system call and the loading of the node program with `exload`. The first argument to `exopen` indicates the name of the parallel computer device which you wish to use. This name depends on what sort of hardware you have available - the manual page for `exopen` contains a list of currently supported devices and their names.The second argument to `exload` is the name of the node program and must correspond to that used when compiling the node code. In this case we have assumed that the program to be loaded has been called "`node`".

Notice that we again use the macro definition of `INT32` to ensure that data objects communicated between processors will have equal length.

Using the preprocessor to enable the byte swapping

The other important feature of the host program is contained in the lines surrounded by the preprocessor directives

```
#ifdef SWAP
 _ex_swaw(....
#endif
```

A more complex strategy for byte swapping

As promised earlier these lines contain "byte-swapping" function calls to transform the data representation of the host to that of the nodes. We use the preprocessor definition feature so that we can turn the "byte-swapping" on or off when compiling the code. This enables us to use the same source code on machines which require "byte-swapping" and also on those that don't by using different '`-D`' options when compiling.

A slightly trickier alternative which is to arrange for the host and node programs to exchange a 32-bit value with a predetermined bit pattern when the code starts up. (`0x12345678` is a good candidate.) From this one can often deduce whether or not "byte-swapping" is necessary allowing decisions to be made at run-time. The advantage of this scenario is that you don't have to even think about preprocessor definitions when compiling for a new machine.

An informative exercise results from omitting the byte swapping in one of the cases where it is necessary - the resulting "garbage" printed at the end of the program is quite characteristic and you will quite readily recognize it in future occurrences.

This exercise has probably convinced you that "host-node" programming is too difficult for real work and you should stick to the *Cubix* model. In common with many contrived examples, however, this exercise has demonstrated almost every feature and complexity of the "host-node" programming style without adding any real substance in the form of a real program that accomplished some real task. Furthermore the additional lines of code which allocate, load and de-allocate the parallel computer nodes are virtually identical in every "host-node" application and can be simply copied from one to another. As a result we believe that the "host-node" computation model can easily be used in real applications and often offers substantial benefits in terms of the amount of code that can be reused - often without even recompiling it.

In the next exercise we return to the *Cubix* model of computation to demonstrate the semi-automatic decomposition system contained within *Express*. These tools provide the basis of a huge number of successful parallel codes and their study is an important part of learning about *Express*.

Exercise 7. The "Ring" program. Automatic decomposing tools.

So far in this tutorial we have written parallel programs that used the fundamental I/O modes and the "global" communication routine `excombine` to achieve parallelism. While we have already written quite important parallel algorithms we must, sooner or later, address the issue of interprocessor communication.

Specifying processor numbers when using Express routines

The basic message passing routines `exread` and `exwrite` that were introduced in the last section had, as their third argument, a pointer to an integer value containing the processor number of the node with which we wished to communicate. In the "host-node" program just presented the situation was really rather simple - the nodes only communicated with the `HOST` processor while the host dealt with each node in turn. The assignment of the node argument was quite straightforward, and could furthermore be made in a manner that did not depend on the topology of the parallel hardware in use.

The reader might be wondering how, in general, we are going to continue this practice as the underlying problem becomes more complex.

A problem with a "logical topology"

In this section we will begin to resolve this issue by considering the following problem:

Each node is required to take a simple message and circulate it, in the most efficient manner possible, through each of the other processors in our network. The result should be that every node has seen the message originating in every other node.

Lest the reader think that this is a frivolous exercise we might point out that this algorithm is the basis for a large fraction of the interesting physical simulations - gravity, melting, etc. are all processes controlled by the need for such an algorithm.

A idealized parallel algorithm which doesn't worry about the hardware

A particularly elegant solution to this problem requires thinking of arranging the processors, logically, in a "ring" as shown in Figure 6. Each processor is assigned a

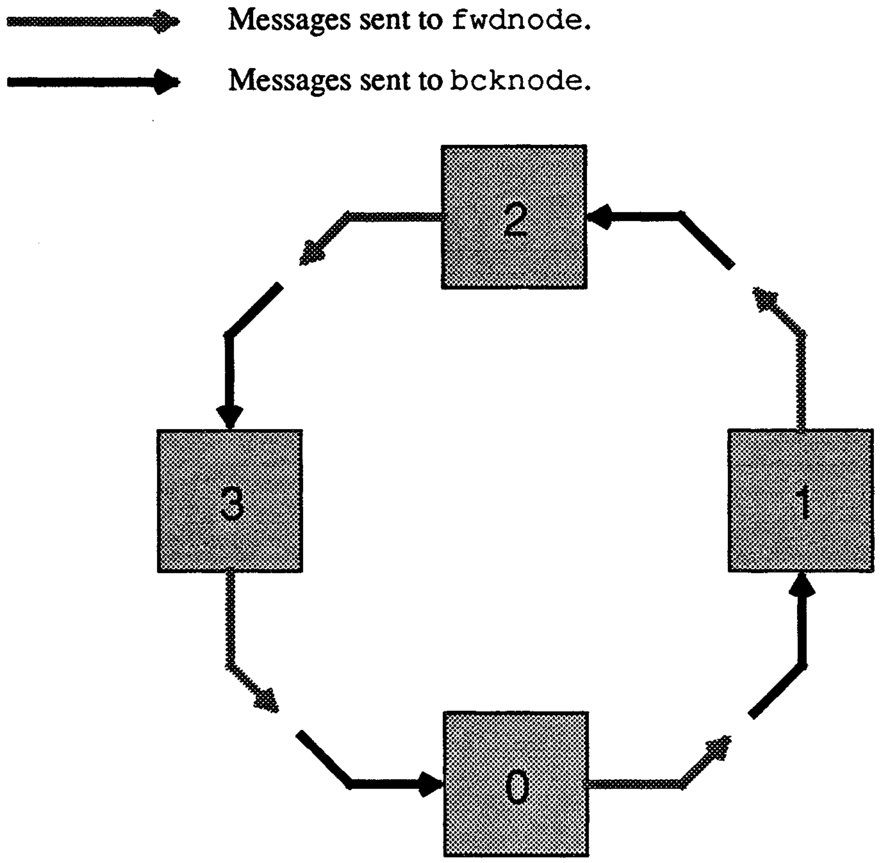


Figure 6. Communication around a ring of processors

"forward" and "backward" neighbor with whom it communicates. The basic algorithm, therefore, is the following

- Send the message originating from this node to our "forward" neighbor.
- Repeat N-1 times the process of reading the message from our "backward" neighbor and forwarding it to our "forward" neighbor.

A minute's thought should be enough to convince the reader that this process does, in fact, result in every node seeing the messages originating in every other node as required by the problem statement. Note, however, that the nodes do not necessarily have to be connected in the ring pattern by the hardware - we only imagine them to be so for the construction of our algorithm.

Assigning processor number for the "ring" topology

The question, therefore, is the assignment of the "forward" and "backward" processor numbers which will be required in the *Express* system calls required to send and forward messages. One simple solution is to use the numbers assigned by the `exparam` function - if we are assigned processor number *P* we can then have *P*+1 as our "forward" neighbor and *P*-1 as the "backward" node. (Modulo the number of nodes, of course - processor 0 cannot have node -1 as its "backward" neighbor!) This technique also removes any topology dependence from the algorithm - as long as the processors are numbered consecutively from 0 everything will work correctly.

Comparing the outcome on different types of hardware

An important issue, however, concerns the efficiency of this approach. To evaluate this we really must address the issue of hardware connectivity although our eventual solution will not depend on it. The method described above works reasonably well on a hardware system that is fully connected as shown in Figure 6. In this type of network every node can

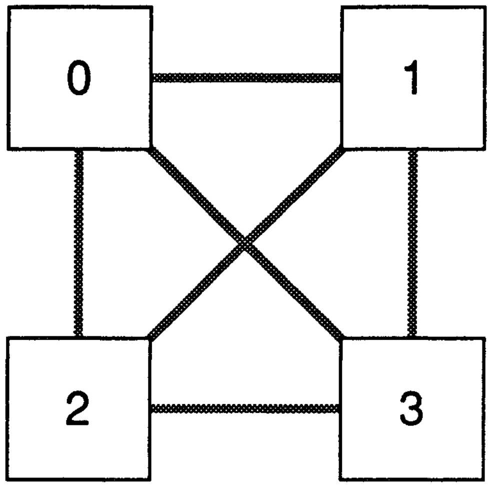


Figure 7. A fully connected parallel processing network

communicate directly with all others and the mapping we have envisaged will succeed.

A rather different outcome would be seen, however, if we were to run our program on a network of the form shown in Figure 6. This network is a square array connecting each node to only two others. It is very common in practical hardware implementations. The problem with the mapping described above is that no direct connection exists between node 3 and its "forward" neighbor, node 0. Similarly node 2 is unconnected to its "backward" neighbor, node 1. In practice this means that messages will travel much more slowly between the unconnected nodes than the connected ones since *Express* has much more work to do in forwarding messages. As a result the "ring" communication program will execute more slowly on the "square" network than the fully connected one. On some types of hardware it might execute as much as twice more slowly.

Forwarding between nodes takes longer than direct communication

The solution to this problem is, however, quite straightforward. On the "square" network we should re-assign the "forward" and "backward" node numbers to take advantage of the topology of the hardware. In particular the assignments

Node 0	Forward 1, backward 2
Node 1	Forward 3, backward 0

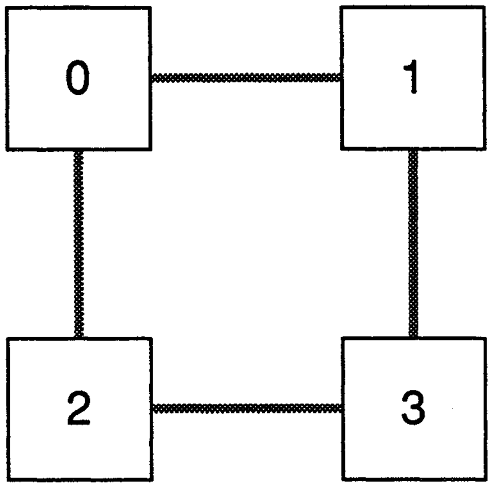


Figure 8. A partially connected parallel processing network

Node 2	Forward 0, backward 3
Node 3	Forward 2, backward 1

lead to a communication pattern in which nodes only communicate with other nodes to which they have direct hardware connections. As a result the program runs just as fast as it did on the fully connected network.

Making the program smart enough to reconfigure itself

The important issue, of course, is how our programs should deal with this without having to hard-wire processor numbers into the source code.

The solution is to use *Express*' `exgrid` utility.

This is a set of tools designed to map problems such as the one we have described onto the underlying hardware topology in an efficient, portable manner. In circumstances such as we have been describing in this section its use can save us from complex coding to perform optimal "mappings" and many hours of "debugging" when one person alters the interprocessor connections without telling other workers. It also helps with mapping multi-dimensional problems as we will see in the next section.

Setting up "logical" topologies with `exgridinit`

The function which is used to initialize the decomposition tools is `exgridinit`. (This and other functions are described on the `exgrid` page in the Reference manual - you may want to read this for more information.) The arguments passed to `exgridinit` tell the system the "dimensionality" of the logical system being used and the number of processors to assign to each "logical" dimension.

The use of the word "logical" in the above discussion may cause confusion. In fact it is usually obvious in any practical system what the appropriate dimension should be. In an image analysis problem, for example, the dimension will be 2 and we need to tell `exgridinit` how many processors to assign to the horizontal and vertical axes of our images. In a structural analysis system the dimensionality will be either 2 or 3 depending on how many real-world dimensions the system can handle. Again `exgridinit` will expect to be told how many nodes to assign to each physical dimension. In our current example we are interested in arranging the processors in a "ring" which is one dimensional. We also tell `exgridinit` to use all the processors in the ring scheme.

Assigning processors to logical dimensions

Note that it is sometimes difficult to decide how many processors should be assigned to a given dimension, especially if you wish to run your program on many differently sized machines. For this reason a function `exgridsplit` is provided to perform the assignment for you. This function can be used to generate the parameters that will be passed to `exgridinit`.

Once `exgridinit` has been called the mapping between hardware and the user's "logical" topology is defined. To find out the processor numbers which should be passed to the communication routines we now use the `exgridnode` routine. This expects three arguments as follows

- The processor number at which we originate the message. This will normally be the value returned in the `procnum` field of the structure returned by `exparam`.
- The dimension of the "logical" mapping along which we wish to send the message.
- The "distance" along this axis measured in units of "nodes". Positive and negative values cause travel in opposite directions.

Assigning processor numbers in a portable, efficient manner

To illustrate the simplicity of this method we can assign the "forward" and "backward" processor numbers with the statements

```
backnode = exgridnode(env.procnum, 0, -1);
nextnode = exgridnode(env.procnum, 0, 1);
```

i.e., we travel one processor in the positive and negative directions along "dimension 0" of our ring.

The beauty of this system is that we can maintain the independence of the program from the topology or architecture of the hardware while keeping its performance as high as possible. Furthermore we never need to know any of the processor numbers involved in message transactions - we just take the values returned by `exgridnode` and pass them to the communication routines without ever having to interpret their exact values.

Sending and receiving messages simultaneously with `exchange`

Most of the hard work in building the "ring" program can now be done automatically. A further optimization, however, can be made if we consider the way in which messages are passed around the ring. It is a matter of little concern whether the message being sent to the "forward" node goes before or after that being received from the "backward" node - the two messages can go simultaneously as far as our algorithm is concerned. This allows us to replace the calls to `exwrite` and `exread` which we might have imagined using with a single call to `exchange`. This routine essentially performs the combined actions of a read/write pair but allows possible hardware optimizations for increased efficiency. It also allows us to simplify the modifications to the code required for parallel processing.

The arguments to this function, which is described on the `exchange` manual page are essentially the four arguments to `exread` followed by the four arguments to `exwrite`. It is important to note, however, that the sensible interpretations of conflicting arguments are made - if the data to be sent has the same address as that being received then `exchange` makes sure that the transmitted message has sent the old data before any new information overwrites it.

The outline of our "ring" program should, therefore, be

- Use `exparam` to find out how many processors are in use and which processor number is assigned to each node.
- Call `exgridinit` to initialize the "ring" topology.
- Call `exgridnode` twice to find our "forward" and "backward" neighbors.
- Loop over the number of processors using `exchange` to both send and receive messages.
- Print a message to show the transit of the messages around the "ring".

The ring code can be compiled and run as a standard *Cubix* program as described in the previous sections.

Sample code for this exercise is shown below

```
#include "express.h"
#include <stdio.h>

struct nodenv env;

main()
{
     int val;
     int type = 100, nextnode, backnode, i;
/*
 * First get environment information - processor id.
 */
     exparam(&env);
     val = env.procnum;
/*
 * Call exgridinit to initialise topology
 */
     exgridinit(1, &(env.nprocs));
/*
 * Call exgridnode to find where to send and from
 * to read
 */
     nextnode = exgridnode(env.procnum, 0, 1);
     backnode = exgridnode(env.procnum, 0, -1);
/*
 * Switch stdout to multi to print value as it
 * moves around the ring.
 */
```

```
        fmulti(stdout);
/*
 * Set the ring loop.
 */
        for(i=0;i<env.nprocs;i++){
/*
 *     Call exchange to send
 */
               exchange(&val, sizeof(val), &backnode, &type,
                      &val, sizeof(val), &nextnode, &type);
/*
 * Print the value and processor number
 */
               printf("Processor %d has value %d\n",
                      env.procnum,val);
/*
 * Do not forget to flush
 */
               fflush(stdout);
        }
/*
 * exit
 */
        exit(0);
}
```

Program 8. Ring program

The beauty of this program is that it runs on *any* number of nodes regardless of the way in which the underlying hardware is connected. Furthermore the user is never concerned with the actual values returned by the `exgridnode` routine - all programming is done in the "logical" configuration assigned by `exgrid`.

In this case the whole concept of interprocessor communication has been replaced by communication between "data domains" - the algorithm requires that a processor access a value contained in a data region other than its own. As a result it uses the `exgrid` utilities to return a "magic token" which enables that processor to acquire the data it requires. No knowledge of "processors" ever enters into this thinking - only the data domain as decomposed by `exgrid` is relevant.

Think of communication in Express as between data domains rather than individual nodes

While this exercise has presented the `exgrid` functions in a simple, and somewhat abstract, one dimensional example the next exercise should provide more "feel" for the tools by mapping a two dimensional problem and adding graphics.

Exercise 8. Two dimensional decomposition with graphics

In the previous exercise we learned how to apply the automatic decomposing tools to a one dimensional problem. In this exercise we will use the same tools for a two dimensional topology typical of that used in computer graphics, image analysis, electromagnetism, fluid flow, cellular automata, expert systems and a whole host of other applications. To bring out the important features of this type of decomposition we will use the *Express* parallel graphics system, *Plotix*.

Extending to two dimensions

The goal of this program is to decompose a two-dimensional data set among the processors and display, on our monitor, a two-dimensional array of colored regions. Each region will represent the portion of the initial data set assigned to an individual processor and will be assigned a unique color.

All the parallel processing elements of this program are already to hand. Following the discussion of the previous exercise we will use the `exgrid` tools to assign processors to the two dimensional topology. Since we may want to run this program on odd numbers of nodes we will use the `exgridsplit` function to assign processors to the two-dimensions of the problem. The actual division of the input array between the processors is performed by the `exgridsize` function. This requires, as input, a processor number and the global size of the data set to be decomposed. It returns to us the starting index and local size of the piece of the data set which should be distributed to the indicated processor.

Elementary graphics

To display the colored regions on the screen we will use the features of the parallel graphics library, *Plotix*. For our purposes only a few routines are required and their manual pages should be consulted for details

`openpl`	Initializes the display device and assigns a buffer for use with graphical objects.
`vport`	Assigns a region of the display surface to a node.
`box`	Draws a rectangular region in a given color.
`closepl`	Deactivates the display device and returns to normal modes.
`usendplot`	Flushes the graphical objects to the display device.

Buffering in the graphics system

Each of these functions, except the last, might have been expected on the grounds of previous experience with graphical systems. The last function is required because *Plotix*, in common with *Cubix*, involves the concept of internal "buffering" for the sake of efficiency. If we drew every object on the display as soon as a node created it *Plotix* would be an *extremely* slow system because it would have to send a small message to the host processor for every single graphical item. Instead it uses the buffer allocated when `openpl` is called to store up information for later display when one of the `sendplot` routines is used. In this case many short messages are combined into one large one as the system is extremely efficient.

The mapping between flushing modes in Cubix and Plotix

As may be apparent, this concept is identical to that used in *Cubix* and, as might be expected there are three flushing commands corresponding to the three *Cubix* I/O modes

`sendplot`	Assumes that all nodes wish to draw the same objects. Forces a synchronization and sends data from the graphics buffer on node 0. This corresponds to the *Cubix* "*single*" mode.
`usendplot`	Each node sends data to the display, in order of increasing processor

number. This call matches the idea of the *Cubix* "*multi*" mode in that all processors are forced to synchronize.

`asendplot` Any node may flush its data independently to the host with this call. No internode synchronization is required and data may appear in an unpredictable order. Corresponds to the "*async*" mode of *Cubix*.

Note that one important difference between the *Plotix* flushing modes and those of *Cubix* is that the latter occasionally flushes buffers automatically, for example, when they are full and in "*single*" mode. *Plotix* never flushes automatically - if the graphics buffer fills up data at the beginning is overwritten by new data - the buffer is treated in a "circular" fashion.

Coordinate systems in Plotix

This discussion would allow us to write a simple program that draws squares on the display but we need to address the issue of coordinate systems in order to pass the correct values to the `box` routine.

After the call to `openpl` the whole display surface is mapped to the range 0.0 to 1.0 in each direction. Together with the information returned from the call to `exgridsize` we could now display the regions on the display. Instead we opt to use the `vport` function to re-map the surface of the display device which each processor is allowed to use. By default each node can address the entire view surface. The arguments to the `vport` function define a rectangular region, as a fraction of the whole, into which the calling node can draw. Using the data returned by the `exgridsize` call we can, therefore, redefine the active portions of the display and then have each node draw a "unit box" in its viewport.

The outline of the program is, therefore, as follows:

- Call `exparam` to identify processor numbers and the number of nodes participating.
- Partition the nodes by calling `exgridsplit`.
- Initialize the two dimensional mapping with `exgridinit`.
- Decompose the two-dimensional array with `exgridsize`.
- Start the graphics by calling `openpl`
- Use the data returned by `exgridsize` to call `vport`.
- Call `box` to draw a colored rectangular region on the display.
- Call `usendplot` to flush a different sub-image from each node.

Modifying the compilation process to link the Plotix library

Since the resulting program will need to be linked with the *Plotix* libraries as well as the usual *Cubix* library we will need to modify slightly the procedure used when compiling/linking this code. Normally this is just a matter of changing the `-lcubix` or `-kcubix` flags to `-lplotix` or `-kplotix` respectively. The manual page for the appropriate compiler and the Introductory Guides also offer suggestions.

Running with a graphics device

To run this program you will also have to modify the normal procedure. In most cases you have to indicate a display type for graphical output by specifying a '`-T`' switch to the `cubix` command. To execute on an IBM compatible PC, for example, you need the switch '`-Tbgi`' while Sun workstations use '`-Tsun`' for Sunview and '`-TX`' for Xwindows. The *Plotix* chapter of the User's Guide contains details of the switches for the supported graphics devices.

The following is sample code for this exercise.

```
/*
 * Code demonstrating the two dimensional decomposition
 * of an array using the exgrid primitives and PLOTIX.
 */
#include "express.h"
#include <stdio.h>
/*
 * Size of array to be decomposed
 */
#define SSIZE 100

int global[2] = {SSIZE,SSIZE};
int start[2],size[2],procs[2];

struct nodenv env;

main()
{
     float x0,y0,x1,y1;
/*
 * First get environment information - processor id, etc.
 */
     exparam(&env);
/*
 * Call exgridsplit & exgridinit to initialise topology
 */
     exgridsplit(env.nprocs, 2, procs);
     exgridinit(2, procs);
/*
 * Call exgridsize to organize data distribution
 */
     exgridsize(env.procnum, global, size, start);
/*
 * Calculate the corners of this processor's
 * window, as fractions of the whole display.
 */
     x0 = (float)start[0]/SSIZE;
     y0 = (float)start[1]/SSIZE;
     x1 = x0 + (float)size[0]/SSIZE;
     y1 = y0 + (float)size[1]/SSIZE;
/*
 * Open graphics device. If fails, exit cleanly
```

```
     */
         if(openpl(DONTCARE, (FILE *)NULL) < 0) {
             fprintf(stderr,
                 "Failed to initialize graphics\n");
             exit(0);
         }
    /*
     * Call vport map each processor's piece of the
     * decomposed data to the display surface.
     */
         vport(x0, y0, x1, y1);
    /*
     * Draw the box using a color based on our processor
     * number and flush it to the display
     */
         box(0.0, 0.0, 1.0, 1.0, 2+env.procnum, 1);
         usendplot();
    /*
     * Close graphics device and exit.
     */
         closepl();
         exit(0);
    }
```

Program 9. The Box Program

It is informative to execute this program on a variety of numbers of nodes and watch the various decompositions automatically appear on the display. Even though we didn't actually do anything with the data we decomposed it should by now be apparent that basic operations are trivially built into the system by using routines such as `exgridnode`.

Modifying the decomposition scheme using `exgrid`

One other, very important point is that we can easily modify important parameters associated with this decomposition. If, for example, we wished to examine the possibilities of one dimensional decompositions for a certain algorithm we merely change the 2 in the calls to `exgridsplit` and `exgridinit` to 1 and *Express* does all the rest. We could similarly extend this decomposition to three dimensions, although the graphics would then need some attention.

3.2 Summary

At this point the basic introduction to *Express* is complete. The information so far presented should be enough to build a wide variety of real applications despite the fact that the codes we have looked at so far have been mere "toys". The next few sections of this document extend the ideas presented up to this point by showing more complete and/or familiar algorithms which may be useful as the basis for complete programs.

A list of the most commonly used Express routines

At this point it is probably useful to summarize the *Express* system calls that have been used in constructing the programs developed so far. Despite the large number of routines in the runtime library we have so far built 8 quite useful programs each of which exhibits useful features using only the following routines:

`fsingl, fmulti`
: Switches a file between "*single*" and "*multi*" modes for I/O.

`exparam`
: Used to find crucial runtime parameters such as processor number and the number of nodes working on the same problem

`excombine`
: Used to "globalize" distributed data - i.e., generate a single result from a set of data which has been decomposed among processors.

`exgrid`
: A library of routines which collectively allow sophisticated decomposition strategies without user intervention and which allows the programmer to be completely free of the underlying parallel computer's architecture and topology.

`exchange`
: A routine which allows us to "simultaneously" send and receive data.

`exread, exwrite`
: Basic message passing functions which allow us to send and receive messages from other processors.

`exopen, exload, exclose`
: Control operations performed only by "host" programs to allocate, load and deallocate groups of nodes in the parallel computer.

`_ex_swaw`
: A byte swapping routine used only when the host and nodes of the architecture in use have different byte ordering properties.

Using only a few Express routines gets us a long way

Everything else we have used has been purely sequential code as might be found in any sequential programming system! This is one of the reasons that we believe that programming in *Express* is just like programming sequential computers - you use all your old code and merely add a few calls to the runtime library to exploit the parallel computer.

It is probably true that well over half of all parallel programs can be built using only these functions and we encourage you to study their manual pages when building your own programs.

4 Advanced Applications

Now that we have seen some of the basic operations of *Express* in a variety of applications we can start to build "real" programs. We use the term real in quotes here because it is obviously impossible to give the text of any commercial quality application in this document.

The techniques used, however, are no different from those already seen in the previous exercises - we have already written the bulk of the extra code that must be added to an application to make it run in parallel under *Express*.

Using the existing techniques to build "real" programs

This section begins by showing the *Express* solution to the Mandelbrot Set. This is not a particularly interesting problem in its own right but it shows the steps necessary to take an existing sequential program and run it in parallel. You might use this as a guideline to your own parallelization strategies.

Exercise 9. Porting existing codes: A Mandelbrot program

So far we have written all our example codes from scratch using, as a guide, the instructions laid out in each exercise. In this case we will first construct a purely sequential program and then examine the techniques used to parallelize it under *Express*.

The particular example which will be used is that of a program to generate and display a portion of the ubiquitous Mandelbrot set. For those unfamiliar with this topic the following is a brief discussion of the algorithm. More advanced discussions can be found in the numerous articles and books on the subject.

A brief introduction to the Mandelbrot set

Briefly, each point of the complex plane, *c*, is assigned an integer value by considering the sequence of values:

$$z_0 = 0$$

$$z_{n+1} = z_n^2 + c$$

The integer value assigned to the point *c* is the smallest value of n for which $|z_n|$ is greater than some arbitrary constant.

To display the Mandelbrot set on a computer screen, a portion of the complex plane is identified with the screen. Hence, each pixel in the screen corresponds to a different value of c. The color of the pixel is chosen from a "palette" according to the value of n. Exactly how the palette is constructed is a matter of artistic, rather than mathematical, judgment.

To render an image of the Mandelbrot set, one must specify the region of the complex plane that is desired. A very simple sequential program to display a portion of the set is shown in Program 10.

Use of Plotix in a sequential Mandelbrot code

First the user is asked to indicate the desired region of the complex plane. It then opens the graphical output device using the *Plotix* subroutine `openpl`, and determines the resolution of the device with the subroutine `aspect`. Once the resolution is known, it allocates an array, called `screen`, of the appropriate size and calls the subroutine `mbrot_fill` to determine n_c for each pixel on the screen. Finally, it calls `mbrot_display` which uses the *Plotix* `marker` and `color` functions to display individual pixels. As we shall see, when we use *Express* to parallelize this program, the subroutines `mbrot_fill` and `mbrot_display` will not have to change. The only changes necessary are a few additional subroutine calls within `main` to automatically decompose the data.

```
/*
```

Now that a sequential program to render the Mandlebrot set is in hand, we can consider how to parallelize it.

Parallelizing Mandelbrot automatically

As we have already seen *Express* provides a number of routines, collectively called the `exgrid` library, which are very useful for a decomposition such as this. The idea is to simply assign approximately equal subsets of the pixels to each processor. By following the generally encouraged practice of writing the `main` program in a "modular" style, we find that all we need to change are some of the arguments to the subroutines, `mbrot_fill` and `mbrot_display`. The subroutines themselves, and hence the bulk of the actual program remain completely unchanged by parallelization. In fact, the changes we are about to make are essentially ignorant of the fact that we are dealing with a program to render the Mandlebrot set. Any program with a similar two-dimensional structure can be treated in exactly the same way.

Domain decomposition with `exgrid`

To begin we initialize the `exgrid` library by calling `exgridinit`. The function `exgridsize` can be used to tell us how much of the array representing the screen should be calculated in each processor. This function fills two auxiliary arrays with values that describe the decomposition of the array `screen`. These values, which are not necessarily the same in all processors, are kept in the variables `y_start`, `x_start`, `ht_local` and `wdth_local`. The starting values in x and y, and the extent of the local array in width and height allow us to calculate the corners of the limited "local" region of the complex plane for which each node is responsible. The values that are returned by `exgridsize` are different in each processor, and the regions assigned to each processor form a non-overlapping array that covers the entire screen, as in Figure 9.

Next we observe that the arguments to `mbrot_fill` specify the coordinates and dimensions of interest in the complex plane. To ensure that each processor evaluates only pixels within its region, all we need to do is change these arguments to correspond to the coordinates and dimensions of the processor's region, rather than the entire screen. This is the purpose of the ten or so lines of arithmetic that appears before the call to `mbrot_fill` in Program 11.. Finally, the *Plotix* function `vport` can be used to map the graphical output from each processor onto the exact portion of the graphical device that corresponds to the region computed in `mbrot_fill`. By using `vport` we eliminate the need to change the `mbrot_display` subroutine in any way.

Notice how we are using almost exactly the same techniques in this program as were used in Exercise 8. This is typical of *Express* programs - the overall structure is quite similar from one program to the next. The parallel version of the Mandelbrot program is shown below

```
    #include <stdio.h>
    #include "express.h"

    main()
    {
        double center_x, center_y, cmplx_wdth, cmplx_ht;
```

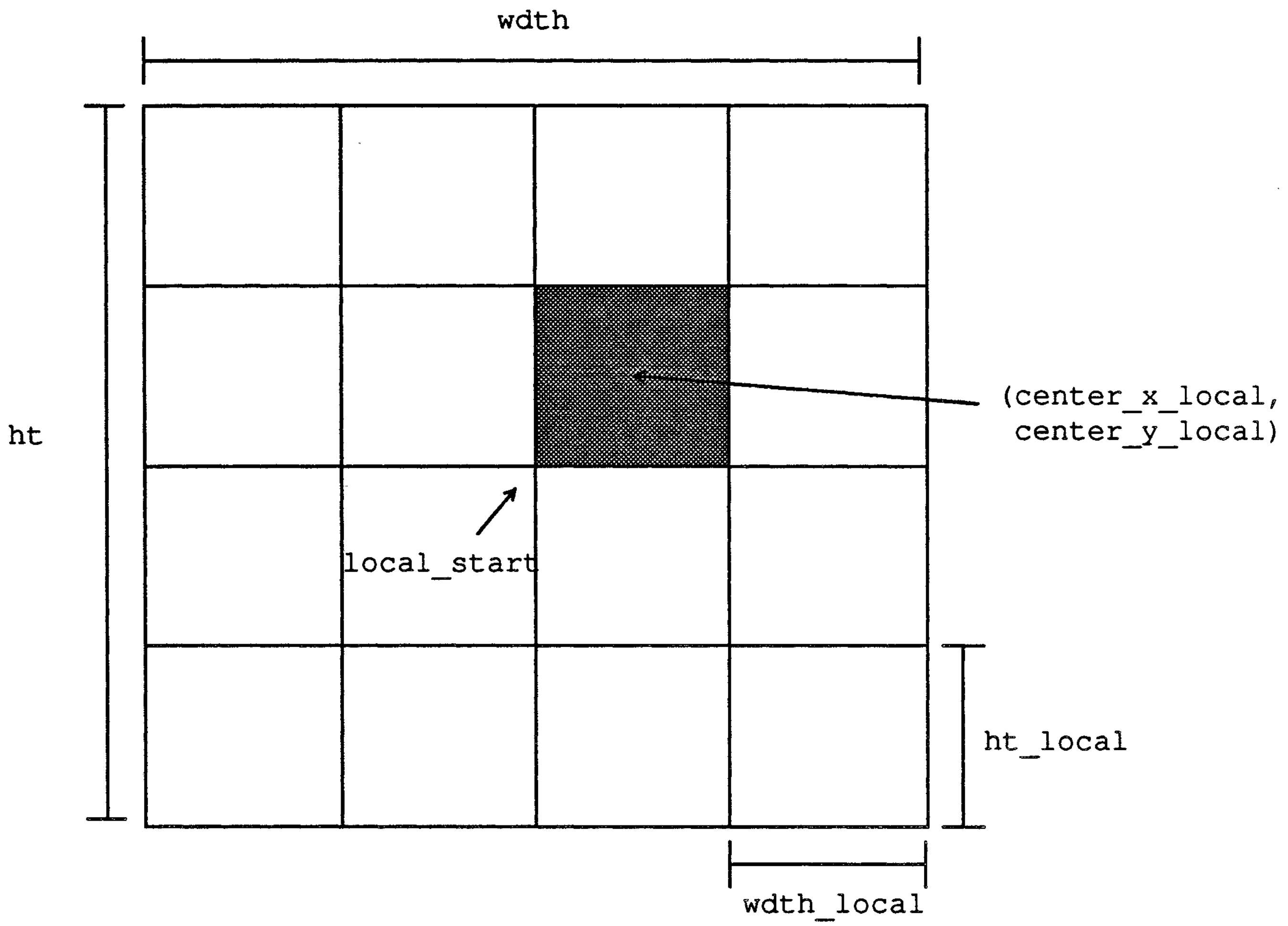


Figure 9. A domain decomposition for the screen for displaying the Mandlebrot set, as used in Program 11.

```
double center_x_local, center_y_local;
double ht_fraction, wdth_fraction;
double x_start_f, y_start_f;
int x_start, y_start;
int *screen;
double ht_f, wdth_f;
int ht, wdth;
int ht_local, wdth_local;
struct nodenv nenv;
int global[2];
int local_start[2], local_sz[2], nprocs[2];

printf("Enter the real and imag. components ");
printf("of the center of the screen ");
scanf("%lf%lf", &center_x, &center_y);
```

```
     printf("Enter the horizontal dimension of the");
     printf("screen in the complex plane ");
     scanf("%lf", &cmplx_wdth);
/*
 * Open the graphical device and determine it's
 * resolution
 */
     openpl(32768, (FILE *)0);
     aspect(&wdth_f, &ht_f);
     cmplx_ht = cmplx_wdth * ht_f/wdth_f;
     ht = (int)ht_f;
     wdth = (int)wdth_f;
/*
 * Call exparam, exgridinit and exgridsize to
 * determine what portion of the screen is to be
 * calculated in this processor.
 */
     exparam(&nenv);
     exgridsplit(nenv.nprocs, 2, nprocs);
     exgridinit(2, nprocs);
     global[0] = ht;
     global[1] = wdth;
     exgridsize(nenv.procnum, global, local_sz,
                                  local_start);
/*
 * Allocate the array representing the screen
 */
     ht_local = local_sz[0];
     wdth_local = local_sz[1];
     screen = (int *)calloc(ht_local * wdth_local,
                      sizeof(int));
/*
 * Determine the coordinates of the center controlled
 * by this processor.
 */
     y_start = local_start[0];
     x_start = local_start[1];
     ht_fraction = (double)ht_local/ht_f;
     wdth_fraction = (double)wdth_local/wdth_f;
     x_start_f = (double)x_start/wdth_f;
     y_start_f = (double)y_start/ht_f;
     center_x_local = center_x + cmplx_wdth*
            (x_start_f + (wdth_fraction - 1.)/2.);
     center_y_local = center_y + cmplx_ht *
            (y_start_f + (ht_fraction - 1.)/2.);
/*
```

```
 * Fill in the screen
 */
     mbrot_fill(screen, wdth_local, ht_local,
                center_x_local, center_y_local,
                cmplx_wdth * wdth_fraction,
                cmplx_ht * ht_fraction);
/*
 * Set up the graphics coordinate system to correspond
 * with pixels
 */
     space(0.,0.,(double)wdth_local,(double)ht_local);
/*
 * Set up a viewport on the device so this processor
 * maps into the right place
 */
     vport(x_start_f, y_start_f,
           x_start_f+wdth_fraction,
           y_start_f+ht_fraction);
/*
 * Finally, render the image on the device.
 */
     mbrot_display(screen, wdth_local, ht_local);
     closepl();
     exit(0);
}
```

Program 11. A parallel version of the program to display a representation of the Mandlebrot set using *Plotix*.

Exercise 10. Matrix transposition

Matrix transposition

The example we will consider in this section is the transposition of a matrix. We assume that the matrix has been decomposed among the processors in a manner similar to that used for the array `screen` in the Mandelbrot example, and that the individual elements have been filled in, perhaps by repeated calls to `scanf`, as in the "sum" example. Furthermore, for simplicity, we assume that the matrix is square, that the number of processors is a perfect square and that the matrix fits "evenly" into the processors. Relaxing one or more of these restrictions is an instructive exercise for the reader. The matrix is decomposed in exactly the same way as was the screen in Figure 9. As a result the code to perform this decomposition would look very similar to that of Program 11.

The subroutine `transpose` is shown in Program 12. It is called with arguments specifying the input and the output arrays and the number of rows (and columns) that are stored locally within each processor.

The program first places the transpose of the local input array into the output array. Then, it identifies the processor in which these values belong. This is done with the exgridcoord and exgridproc functions, which convert a processor number into indices into an array of processors, and vice versa. Note that the eventual home of the values is coincidentally the source of the values which belong in the current processor, so the source and the destination arguments to exchange are identical. Finally, the communication function exchange is called to send the data to its destination and receive the data from the source.

```
      #include "express.h"

      transpose(inarray, outarray, nrows_local)
      int *inarray, *outarray;
      int nrows_local;
      {
          int i, j, me[2], him[2];
          struct nodenv nenv;
          int type = 1234; /* Any value is ok here */
          int his_procnum, len;

          exparam(&nenv);
     /*
      * First transpose the part of the matrix on this
      * processor.
      */
          for(i=0; i<nrows_local; i++){
              for(j=0; j<nrows_local; j++){
                  outarray[j+i*nrows_local] =
                       inarray[i+j*nrows_local];
              }
          }
     /*
      * Figure out where I am in the big picture.
      */
          exgridcoord(nenv.procnum, me);
     /*
      * Now exchange my horizontal and vertical coordinates
      * to determine where my transpose is in the big
      * picture
      */
          him[0] = me[1];
          him[1] = me[0];
     /*
      * Finally, figure out the processor number of my
      * transpose
      */
          his_procnum = exgridproc(him);
     /*
```

```
 * Now exchange the data
 */
    len = nrows_local * nrows_local * sizeof(int);
    exchange(outarray, len, &his_procnum, &type,
             outarray, len, &his_procnum, &type);
}
```

Program 12. A subroutine to transpose a matrix that is decomposed by the `exgrid` library.

Exercise 11. A Spread-sheet Program

The final example of a parallel program we will consider is a spread-sheet program similar to any of the popular ones that run on personal computers. Again, the domain decomposition is as in Figure 9. Each processor assumes responsibility for a portion of the entire spread-sheet. Obviously, a spread-sheet program is much too complicated to consider in all its detail. Most commercial spread-sheets include features like highly sophisticated input languages and graphical interfaces. In regard to these features, we simply point out that a parallel computer is no harder to program than a sequential one. With the *Cubix* system, any spread-sheet program written in a high-level language that runs on the host can be recompiled (if the source code is available) and run on the parallel processor. Alternatively the host interface can be left intact and the computationally difficult updates can be passed to the parallel computer by using the "host-node" programming model.

Porting an existing system - the options

The time-consuming piece of a spread-sheet program is the update phase, when potentially thousands of cells must be modified to reflect new information. Fortunately, this is the piece that is amenable to parallel computation by domain decomposition.

Modifying this strategy is an easy exercise.

To simplify memory allocation we program the decomposition as a 4x4 array (16 processors) directly into the code. Furthermore, we assume that a given cell can depend only on other cells in the same row. The extension to intra-column dependencies is essentially accomplished by replacing "row" by "col" and vice versa in Program 14..

```
#include "express.h"
/*
 * Fix the decomposition so it has 16 processors in a
 * 4x4 array. This could be computed at run-time based
 * on the number of processors available, at the
 * expense of a few calls to the memory allocation
 * function calloc() and the function exgridsize()
 */
#define N_ROWS_PROC 4
#define N_COLS_PROC 4

int nprocs[2] = {N_COLS_PROC, N_ROWS_PROC};
int nrows, ncols;
int nrows_local, ncols_local;
int row_partners[N_COLS_PROC];
struct cell *spread_sheet;

decompose()
     /* Call this after the total number of rows and
      * columns, nrows and ncols, have been set
      */
{
     int size_global[2], size_local[2], start[2];
     int me[2], partner[2];
     struct nodenv nenv;

     exparam(&nenv);
     exgridinit(2, nprocs);
     size[0] = ncols;
     size[1] = nrows;
     exgridsize(nenv.procnum, size_global,
                           size_local, start);
     ncols_local = size_local[0];
     nrows_local = size_local[1];
     spread_sheet = (struct cell *)calloc(
            nrows_local * ncols_local,
            sizeof(struct cell));

     exgridcoord(nenv.procnum, me);

     for(col=0; col<N_COLS_PROC; col++){
         partner[0] = col;
         partner[1] = me[1];
         row_partners[col] = exgridproc(partner);
     }
}
```

Program 13. A routine to set up some external variables for use in a spread-sheet.

We set up a decomposition of the spread-sheet cells that is essentially identical to that in Figure 9. The parameters, both "local" and "global", are stored in external variables, where they can be accessed by other subroutines. In addition to setting the number of rows and columns of cells, processors, etc., an array is set up, `row_partners`, which stores the `procnum`'s of all the other processors in the given processor's row. This information is obtained with the `exgridproc` function, just as in Program 12..

The `update_rows` routine uses the external variables set up in `decompose` to collect entire rows, one row at a time, into each and every processor. With this data, the subroutine update_cell_from_row is expected to perform any necessary modifications to the values in spread_sheet. The communication subroutine used in this program is exconcat, which concatenates data sent by a collection of nodes. The collection is specified by the array row_partners which was filled by Program 13..

exconcat is another of the "global" communication routines

```
     struct cell whole_row[N_COLS_PROC];

     update_rows()
     {
          int row, col;
          int type = 100;
          struct cell *this_row;

          for(row=0; row<nrows_local; row++){
              this_row = &spread_sheet[row*ncols_local];
              exconcat(this_row,
                       ncols_local * sizeof(struct cell),
                       whole_row, sizeof(whole_row),
                       NULLPTR, N_COLS_PROC, row_partners,
                       &type);
              for(col=0; col<ncols_local; col++){
                  update_cell_from_row(spread_sheet,
                                       row, col, whole_row);
              }
          }
     }
```

Program 14. A subroutine to perform all intra-row updates.

5 For more information.

This tutorial has only scratched the surface of parallel computing and the features of the *Express* system. To begin with, there is no substitute for learning-by-doing. With the information in this tutorial, you should be able to write some simple programs of your own.

Other Express tools: NDB, PM, etc.

Unless you are a very rare and special individual, your first few attempts at writing your own parallel programs will contain bugs. Along with writing your first parallel program, you should read the chapter describing "ndb: A Source Level Debugger for Parallel Computers", and use ndb to help you find the bugs.

Once your program is running, you may wish to measure its performance. *ParaSoft*'s profiling system allows you to measure execution times, communication and "system" overheads and to trace detailed "events" within a running program. It is equipped with a graphical interface to facilitate interpretation of the large amount of data that is generated when profiling a parallel system. For complete documentation on the system, refer to "*PM*: A Profiling System for Parallel Computers."

To find out more about *ParaSoft*'s general purpose I/O system, refer to "*Cubix*: Programming Parallel Computers Without Programming Hosts."

To find out about the extensive parallel graphics package, *Plotix*, refer to "*Plotix*: A Graphical System for Parallel Computers." *Plotix* is a 2-D graphics system containing input as well as output primitives.

The extensive communication library in *Express* is described in "*Express*: A Communication Environment for Parallel Computers". This includes information on how to write your own host program (only necessary if *Cubix* does not satisfy your needs), mechanisms for loading different programs into different processors, asynchronous programming styles and many other topics.

Considerable work has taken place in the last five years identifying problems in science and engineering that are amenable to parallel computation. The book "Solving Problems on Concurrent Processors" by G. Fox, *et. al.*, published by Prentice-Hall surveys some of this work. It is an excellent place to look for analysis of a wide variety of problems and decompositions.

Express

A portable, efficient communication system for parallel computers ... and much more

1 Introduction

When coding an application on a conventional sequential computer one is often faced with a choice between several different implementation techniques based on the wide range of tools available. In typical parallel environments this has not been the case since the facilities offered to the application developer have either been limited in scope or completely non-existent. *Express* is a communication environment or Operating System designed to offer a wide range of implementation strategies to both system and application designers. In particular it has been motivated by application requirements rather than any intrinsic operating system concepts which is one of the reasons it is usually referred to as an "environment" rather than an operating system.

Sequential computing tools

As already stated a goal of *Express* is to meet the needs of application codes by offering utilities at all levels of complexity from low level message passing primitives to automatic data-level decomposers and a corresponding communication interface that is totally independent of the underlying hardware connectivity. Which system is appropriate for a particular application can only be decided by considering the needs of that application.

Helping develop commercial applications

Among the questions that might be important are

Issues in deciding on parallelization strategies

- Do I need completely non-blocking message passing or do my problems exhibit some sort of synchronization that can usefully be oriented to the communication structure?
- Does my application require totally asynchronous message processing, e.g. an interrupt handling capability?
- Can I usefully use techniques such as "double-buffering" to enhance my I/O bandwidth?
- What are the I/O requirements of my algorithms?
- Do I need to write a program for the host computer or can I just have a program running in the parallel machine?

Note that most of these questions contain references to applications. This is an obvious point - the nature of the application should determine the software model to be used. Until fairly recently, however, the answers to many of these questions were decided by the hardware - different implementations weren't flexible enough to support the different possibilities. The more recent machines, coupled with advancing software systems have allowed developers to once again relate these fundamental decisions to their algorithms rather than someone else's idea of an "operating system".

Express is a software package designed to meet the needs of applications. Its fundamental design is rather different from more conventional "operating systems". Rather than starting from the hardware and building a communication system etc. *Express* began with applications, considered their requirements and built up a system to fulfill them.

Express is conceptually a multi-layered system. At the lowest level is support for allocating processors, loading programs and asynchronous message passing. These facilities are available to the user, or can be ignored totally as befits a particular application. At a higher level, and logically distinct, are the utilities designed to automatically decompose problems with regular structure. These routines form the basis for an extremely elegant model of computation in which the underlying topology of the hardware can be completely ignored. Along with this level are routines that interface programs running on the host computer, if any, to the programs running in parallel. At the highest level is a complete I/O system allowing parallel programs uniform access to the operating system facilities of the host. This level makes for the easiest computing - the user simply adds communication calls to a working sequential program in order to parallelize it and writes no program for the host computer. The I/O system operates in several modes allowing either synchronous or asynchronous operation according to the needs of the application.

Each of the levels described above is logically distinct building only on those below it. As a result we are able to port the system to a wide variety of hardware/software systems taking a "top-down" approach in which the higher levels are built upon existing lower layers whether they be implemented in hardware, software, within *Express* or some other native operating system.

The structure of this chapter is as follows. Section 2 describes some fundamental issues including booting the *Express* kernel, programming models and initializing software in various high level languages. Section 3 describes the fundamental processes of allocating nodes and downloading programs from the host system. Section 4 describes the message passing support provided by the *Express* kernel and is divided into sections discussing the blocking and non-blocking subsystems separately. Also discussed in this section are the concepts and restrictions which surround the "processor number" and "type" concepts which are central to *Express*. Section 5 introduces the concepts of automatic decomposition and the utilities that take advantage of them and section 6 describes the I/O subsystem for file/terminal I/O and other operating system functions. Section 7 contains details of a hardware specific set of communication functions which may be of interest to users with heavy communication needs. Section 8 presents a complete example *Express* program and discusses the other example codes supplied with the system.

This chapter is arranged such that low level concepts are introduced first building to more sophisticated systems. This has the advantage of being quite logical but the disadvantage of perhaps giving too much information. One of the advantages of the multi-layered nature of *Express* is that one can use the upper layers with no knowledge of the lower ones. As a result programmers may choose to adopt a programming style in which certain sections of this chapter are completely irrelevant. Good examples include most scientific programs. These codes can usually be developed using *only* the techniques described in sections 2, 5 and 6 - the material of sections 3 and 4 is unnecessary. Since one typically needs to understand the whole picture before deciding which programming style to adopt we suggest that the reader at least skim through sections 2 and 3 even if the decision has been made to program at a higher level.

DON'T PANIC!

This document may seem a little daunting at first. There are many routines and lots of arguments. However users with particular problems may need to use only a small portion of the system described here. In particular we have found that many scientific applications can be parallelized with only a couple of the calls described in Section 5 and the I/O system of Section 6. Furthermore, the I/O system is essentially self-explanatory in most cases making the manual rather superfluous. One of the services that *ParaSoft* is happy to offer its customers is consultation in the needs of various applications. Please feel free to call us with any enquiries about the facilities contained within *Express*. We would also like to hear your suggestions/comments about the system. In particular, we would like to enhance the system at the user application level by providing whatever tools for automatic parallelization can be invented. Please send us your suggestions!

2 Fundamentals

Before you read this...

Before you can make any progress with *Express* there are certain fundamental tasks which must be performed; system configuration and software initialization. The former task can be quite tricky so most of the details are left to another document. The latter process depends rather heavily on the language in which your application is being written.

2.1 System Configuration; Booting *Express*

Starting Express

We can divide parallel processing systems up into two types; configurable and non-configurable. In the latter category are included machines with fixed hardware interconnects such as hypercubes and mesh machines. These systems need no configuration by the user - *Express* can be booted as described in the next paragraph and all is done. Among configurable systems, on the other hand, are the various Transputer networks. In these systems the actual hardware connectivity can be changed by either programming electrical switches or by moving ribbon cables. In these cases *Express* has to be configured for the underlying hardware topology.

Using cnftool to configure the hardware...

Fortunately this is not as daunting a prospect as it sounds; *Express* contains a graphical configuration tool `cnftool` which guides you through the configuration procedure in a fairly simple manner. This tool is actually rather sophisticated allowing many different combinations of host computers and transputer nodes. For further information consult the companion document "Configuring *Express*".

... and exinit to boot Express

Once your system is correctly configured for *Express* it is necessary to start it running. This procedure is actually simplicity itself - the single command

```
        exinit
```

serves to download the kernel and start it operating. Hopefully the self contained diagnostics will suffice to enable corrective action to be taken if problems are found. The most common sources of error with Transputer systems are incorrectly placed hardware links - i.e., those which do not correspond to the information supplied to `cnftool`. If you consistently have trouble booting the system give us a call. Note that you may have to use the `exreset` call as well as `exinit` if you have multiple transputer boards or are connecting several machines together. The `cnftool` manual contains the details.

Finding out what other programs are running

One other program that will probably get heavy use is `exstat` which simply provides a summary of current system usage. The simple command

```
        exstat
```

prints out a simple statement such as

```
        4 nodes in use out of a total of 16
```

The two numbers indicated here show how many nodes are currently in use and the total number that the kernel believes to exist. The former is useful - before resetting the system with `exinit` it is probably prudent to check that nobody else is running jobs. The latter number should reflect the total number of nodes available in your hardware configuration. Note that since the `exstat` command sends messages to the *Express* kernel running on the nodes it can only function if the system has been booted and is still running. If, therefore, you obtain no response from `exstat` this indicates that the nodes should be reset with the `exinit` command.

In certain circumstances *Express* may claim to be unable to allocate, say, 8 processors even when there seem to be eight available. In this case one might use the command "`exstat -l`" to obtain more information about the allocated processors - it will usually be the case that the allocated nodes fall in positions which block the allocation of the remaining processors in a single block.

2.2 Programming models

Express actually contains two completely disjoint programming models tailored to the needs of application developers. While these are discussed at some length in the companion document: "*Express*: An Overview" it seems wise to repeat some of that discussion here since it is extremely important that one understands the relative merits of the two systems and understands how to switch between them.

The Cubix programming model

The *Cubix* model is conceptually the simplest. One takes a piece of code and executes it on the parallel computer nodes by invoking the `cubix` command. In this model of computation the parallel program may call on most operating system services as though it were running on the host computer itself - an obvious example is I/O. In C one is able to call the various functions to be found in the runtime library - `printf`, `scanf`, `fopen`, etc.... while Fortran support is provided for `READ`, `WRITE`, `OPEN`, etc. Graphics is also available through the *Plotix* subroutine library which offers a simple but fully functional device independent graphics capability to parallel programs.

The "Host-Node" programming model

The alternative model, called "Host-Node" in these manuals, entails writing a program to run on the native host computer which communicates with the parallel computer nodes using basic *Express* system calls. In this model the host program can use any of the host services that were previously available to it plus the additional ones provided by *Express* to communicate and control the parallel computer. On the other hand the programs running in the nodes may only make use of the facilities naturally available to the nodes processors - this usually means that any I/O must be handled by the host program and then sent in messages to the nodes.

Deciding which programming model to use

The two models have their own advantages and disadvantages and it is important to decide which is more appropriate for your application. While in many cases it is possible to switch back and forth between the two styles as development progresses it is usually a good idea to have in mind one style for any finished project.

Among the features to be considered are the following:

- How large is my application? In the *Cubix* model EVERYTHING must go into the parallel computer, I/O, graphics, user interface, etc... This requirement may necessitate more memory than is available on the nodes of the parallel computer. In the "Host-Node" model some parts of the code may be kept on the host.

- How much machine specific code do I have? If many man-years have been spent developing a complex graphical user interface, for example, it may be wasteful to attempt porting it to the parallel computer environment under *Cubix* and *Plotix* when it may run intact on the host machine.
- What I/O bandwidth do I need? This is a much more complicated issue. Some vendors supply hardware which directly connects the parallel computer to I/O services such as disks. In this case the *Cubix* model is able to take advantage of such hardware and provide fast, *parallel*, I/O directly to the nodes. If such hardware is not available one must consider the relative merits of having the host do I/O and send data to the nodes in messages (Host-Node) or having the nodes do it themselves (*Cubix*). In this case the appropriate issues include the availability of overlapped or non-blocking I/O facilities on the host. If available the "Host-Node" approach is probably faster. Otherwise both are about the same.
- Debugging. In the "Host-Node" model the nodes are unable to perform their own I/O without coordination with the host. If, therefore, one wishes to debug in the old style with print statements one has to change both host and node codes, recompile both and hope that the communication calls were inserted in the correct places. This is actually harder to do than it sounds and can be an annoying source of minor bugs - it can take as long to get the output to work as it does to find the bug itself! In the *Cubix* model one simply inserts the print statements into the node code and recompiles. (Of course one could also use *ParaSoft*'s source level debugger, `ndb`.)
- Prototyping. Trying out new pieces of code in the "Host-Node" model can be quite time consuming for essentially the same reasons as mentioned in the last point - compiling, coordinating and debugging two pieces of code for the host and nodes can take time. Under *Cubix*, on the other hand, one can take sequential codes and run them intact on a single node of the parallel machine. In many cases this is sufficient to evaluate development strategies.
- Portability and maintenance. Under *Cubix* one only has a single piece of code that runs in the parallel computer nodes. It can often be maintained in the same source files as the sequential version of the code - especially since dummy *Express* libraries can be supplied for most machines. If the "Host-Node" approach is adopted the source code, at the very least, has to be divided between that which runs on the host and that which runs on the nodes. In most cases a certain amount of common interface between the two has also to be maintained. A further problem arises when the host and node machines are different CPU types - e.g. Motorola host and INTEL nodes. In this case the byte orderings of the two machines are different as can be their word lengths. This makes the interface between nodes fairly complex - byte swapping and casting all over the place!

- Redevelopment cost. Many large applications have their origins in the dim and distant past - their may be few people who actually understand the whole codes. In this case it might make sense to adopt the "Host-Node" model and attempt to parallelize only a small portion of the code while leaving the rest untouched.

As can be seen there are many issues involved in making the decision as to which programming model is bet suited to your application. One of the virtues of *Express* is that both are available to the user within a single package.

In this and other manuals many references will be made to either the "Host-Node" or *Cubix* programming models since the *Express* interfaces to the two systems are often subtly different. It is important to bear in mind which system you will be using when reading this documentation.

2.3 Software Initialization; Languages

What languages are supported by Express?

Before the programs you write can use *Express* functions certain important data structures have to be initialized. This brings up the question of the different high level software languages that support, or are supported by, *Express*.

The nature of Express - a subroutine library

Express is a subroutine library. This means that it can be added to ANY existing high level language for which your parallel machine has a compiler. In particular, this means that *Express* is available to both C and FORTRAN programmers. As other languages gain support and their compilers become available *Express* will be supported in these too.

Each different language, however, has its own characteristics which affect the implementation of *Express*.

Writing C programs

`express.h`

C, for example, already requires a fair amount of system support and as a result the user is presented with a fairly simple interface. Most constants/variables needed by the user are defined in the header file `express.h` which should be included in all *Express* programs. On most systems you can use the standard angle bracket notation for this file and the compiler will know how to find it. Similarly *Express* is initialized before the user `main` routine is called so that no extra system calls are necessary to setup *Express*. A typical *Express* program written in C, therefore, has the skeletal form

Skeleton C program

```
#include "express.h"
           /* Define system constants/macros */

main(argc, argv)
int argc;
char *argv[];
{
    ............./* User Program .... */
```

KXINIT and the XPRESS common block

FORTRAN, on the other hand, is less oriented to system support and as a result more is required of the user. The "`include`" mechanism is very non-standard in FORTRAN which precludes making system variables available in a header file. Instead *Express* has a named common block which contains the FORTRAN equivalents of the C include file.

This common block has the following structure

```
     COMMON/XPRESS/NOCARE,NORDER,NONODE,IHOST,IALNOD,IALPRC
     INTEGER      NOCARE,NORDER,NONODE,IHOST,IALNOD,IALPRC
```

where all parameters are of type `INTEGER`. Each of these system parameters has a corresponding value in the C implementation which is explained more fully in the manual page for the `KXINIT` function. (See the reference manuals for more details.) This latter function is the one which serves to setup *Express* and initialize the above common block. It must be called in every program that uses *Express* on both host and nodes. The prototype FORTRAN *Express* program has the following form

Skeleton FORTRAN program

```
         PROGRAM MYPROG
      C
         COMMON/XPRESS/NOCARE,NORDER,NONODE,
        $                             IHOST,IALNOD,IALPRC
      C
      C Start up EXPRESS
      C
         CALL KXINIT
      C
      C Proceed with user program
      C
```

Note that the user is not restricted to one or the other language. It is quite possible, for example, to write a FORTRAN "host" program which interfaces with a C "node" program, or vice versa. Similarly one can mix languages within the same program although *Express* has no explicit mechanism for this purpose - standard compiler implemented techniques must be used.

3 Processor Allocation and Program Loading

Allocating nodes and loading programs can only be done from the host

The most fundamental operation required before running a parallel application is to allocate a bunch of processors and somehow load a program into them. There are several levels at which this can be achieved and/or is necessary depending on the programming model you have adopted. In the *Cubix* style all of these processes are managed by the `cubix` program and so you should have no interest in this section. If you have adopted the "Host-Node" model there are several important variations on a theme which might be considered and which are discussed in this section.

1. Complete ignorance; The user application has no interest in the details of this procedure and has no interest running on the host processor.
2. Host control; The application, for whatever reason, needs to have a process or processes running on the host computer which controls the allocation of processing elements and the flow of data to/from them.
3. Full control; The application wishes to control all aspects of the allocation

and loading of the parallel program including which physical processors to use and what to use them for.

The first level will not be described in this section - since the user requested no information none will be given. The mechanisms required to run parallel applications in this fashion are described in section 5. The other two categories will be described in this section. Not all the routines available to perform loading will be described here. For more details the reader is urged to peruse the reference manual Also, no information is supplied as to the method of actually generating a program to run on the parallel machine. These details are provided in the introductory guide to your particular version of *Express*.

3.1 Processor Allocation

Different loading configurations

Express provides a large set of tools for loading applications into groups of processors. As well as sending a single application to all processors one can also load individual nodes with their own applications, pass arguments to all nodes, or individual argument lists to separate nodes. These utilities are provided independently for each user of the parallel machine - it is quite possible to have two users simultaneously access two independent sets of nodes although this is not the only model supported. In particular we allow for the possibility of single host programs accessing multiple groups of nodes in the parallel computer, as well as the possibility of multiple host processes sharing access to a single group of nodes.

The central concept in discussing low-level processor allocation etc. is that of the ***processor group*** and the associated ***processor group index***. A processor group is a collection of nodes allocated with a single call to the routine `exopen`. The processor group index is the value returned by a successful call to `exopen` and which is used to subsequently indicate the particular set of nodes to which an operation should be applied.

Basic allocation routines

The routine which allocates processor groups is invoked as

```
        pgindex = exopen(device, Nnodes, where);
```

where the first argument indicates the fundamental device from which nodes are to be allocated, the second is the number of processors required and the last optionally indicates exactly which processors are required. The purpose of the last argument is so that applications built around custom networks can physically place certain applications on certain nodes. The macro value `DONTCARE` may also be given in the last position to indicate no interest in the physical placement of the program.

`exopen` returns a value which, if negative, indicates an error. Obvious sources of error include specifying an illegal device or asking for more nodes than are currently available through use by other users. Otherwise the value returned is the ***processor group index*** by which this particular set of nodes should be identified in future calls to *Express* control utilities.

Loading programs

Having allocated a set of processors one must load an application. The simplest way to do this is with the `exload` and `expload` system calls. The former loads the same program into all nodes of the processor group while the latter loads a single node with an application and is used in cases where different programs must be loaded into different nodes within the same processor group. As an example consider the following code segment

```
#include <stdio.h>
#include "express.h"/* Defines DONTCARE */

main()
{
    int pgindex;

    if((pgindex=exopen("/dev/transputer", 4,
                                  DONTCARE)) < 0) {
        fprintf(stderr,
               "Failed to allocate processors\n");
        exit(1);
    }
    if(exload(pgindex, "noddy") < 0) {
        fprintf(stderr,
               "Failed to load application\n");
        exit(1);
    }
```

The single application "noddy" will be loaded into all 4 processors and will immediately begin execution. Sometimes this isn't exactly what we wanted. An important case occurs when we use the debugger, ndb. In this case we want to load the application "stopped" so that we have time to leisurely fire up the debugger and take control of things. We do not want the node program to go screaming off into the wild blue yonder before we get a change to debug it! This behavior can be achieved with the expause function call. If we call this function before loading program then they will be loaded with a breakpoint set at the normal entry point. It is often convenient to make this behavior conditional upon some variable so that the program can be run in either mode without recompiling; while things are going well one just loads the node program and blasts away but when problems arise one can immediately switch back to debugging mode without time consuming recompilation. Simple code which makes use of this feature is shown below. *Debugging*

```
#include <stdio.h>
#include "express.h"   /* Defines DONTCARE */

main(argc, argv)
int argc;
char *argv[];
{
    int pgindex;

    if((pgindex=exopen("/dev/transputer", 4,
                          DONTCARE)) < 0) {
        fprintf(stderr,
               "Failed to allocate processors\n");
```

```
            exit(1);
        }

        if(argc > 1) expause();

        if(exload(pgindex, "noddy") < 0) {
            fprintf(stderr,
                    "Failed to load application\n");
            exit(1);
        }
```

Note that the call to `expause` is made conditional upon the number of runtime arguments passed to the user program.

Loading different programs into different nodes

If we needed to load different applications into different nodes this could be accomplished by changing `exload` to `expload`. The following code segment loads "`noddy`" into nodes 0, 1 and 2 but "`big_ears`" into node 3.

```
#include <stdio.h>
#include "express.h"/* Defines DONTCARE */

main()
{
    int pgindex, s0, s1, s2, s3;

    if((pgindex=exopen("/dev/transputer", 4,
                            DONTCARE)) < 0) {
        fprintf(stderr,
                "Failed to allocate processors\n");
        exit(1);
    }
    s0 = expload(pgindex, "noddy", 0);
    s1 = expload(pgindex, "noddy", 1);
    s2 = expload(pgindex, "noddy", 2);
    s3 = expload(pgindex, "big_ears", 3);
    if(s0 < 0 || s1 < 0 || s2 < 0 || s3 < 0) {
        fprintf(stderr,
                "Failed to load applications\n");
        exit(1);
    }
    exstart(pgindex, ALLNODES);
    exmain(pgindex, ALLNODES);
```

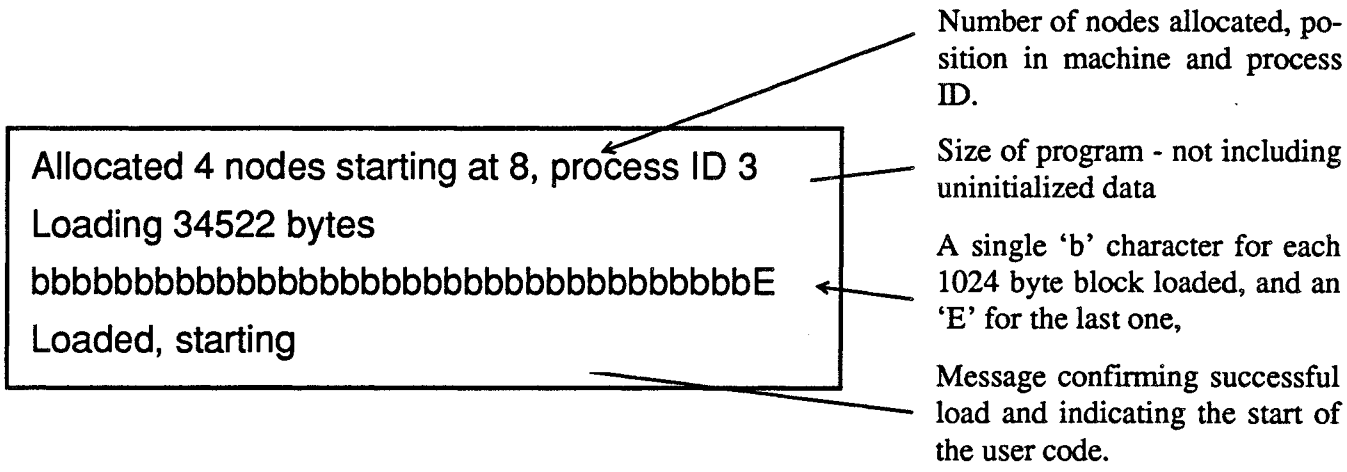
This code contains several new features. Each call to `expload` loads an application into the indicated node. After checking for failures there is a call to `exstart`. The first argument here is the ***processor group index*** as before and the second is a processor number.

This routine serves to tell the nodes that program loading is complete and we should get ready to load arguments. The special value `ALLNODES` means "everybody". Since, in this example, we don't want to go off and load any more arguments or environment we can immediately start up the user program. This is done with the call to `exmain` whose arguments are similar to `exstart`. At this point the loading process is complete and the node programs will begin to execute unless, of course, a call to `expause` has been made.

Generalized loading primitives

The extra complication involved in this last example (i.e., the `exstart` and `exmain` calls) is necessitated by the fact that the loading primitives provided by *Express* are very general. You can load different programs into each node, different arguments into each node and also download different environment variables into each node. Discussion of the means by which this is achieved is postponed to the appropriate manual pages.

In order to confirm that everything is going well the system keeps you informed of the loading status of the machine. In response to the `exload` command, for example, a display such as that of Figure 1 will appear. (In windowing systems such as MicroSoft Windows or the Macintosh an oscillating cursor is used during the loading process in place of the text display.)



```
Allocated 4 nodes starting at 8, process ID 3
Loading 34522 bytes
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbE
Loaded, starting
```

Number of nodes allocated, position in machine and process ID.

Size of program - not including uninitialized data

A single 'b' character for each 1024 byte block loaded, and an 'E' for the last one,

Message confirming successful load and indicating the start of the user code.

Figure 1. Downloading a user application

If this information appears incorrect, no '`b`' characters appear, or no '`E`' then the kernel is probably out of action and one might consider resetting everything with `exinit`.

Passing arguments to node programs

Having gone through the loading and initialization calls on the host processor several copies of the application have sprung from the vacuum on the nodes and are running. One way to proceed would be to dive right into the message passing systems and "go for it" with the various tools to pass information to the node programs. Another way, however, takes advantage of the fact that C programs are usually created, not in a vacuum, but with a ready made environment, usually known as `argc`, `argv` and `environ`. Just as in a regular sequential program these variables can also be made available to the parallel programs through alternate loading procedures.

For simplicity, consider the case where you want to pass three arguments to your node programs; the three strings "`go`", "`very`" and "`fast`". Furthermore you want each processor to receive the same arguments. (Many other possibilities are available including NULL terminated vectors of arguments and different arguments in different nodes, see the reference manuals.) Then the simple modification to make is to change the call to `exload` to

```
  exloadl(pgindex, "noddy", "go", "very", "fast", NULLPTR);
```

where the first two arguments are the same as before and the following ***list*** of strings will be passed to the node programs. Note that the list is completed with a NULL pointer. This is important - if you omit this last item then your node programs will probably get very long and strange argument lists!. The effect of this function call is best demonstrated by considering the following hypothetical node program

```
/*
 * Hypothetical node program showing argument passing.
 */
main(argc, argv)
int argc;
char *argv[];
{
    int i;

    printf("We were given %d arguments\n", argc);
    for(i=0; i<argc; i++) printf(" %s\n", argv[i]);
    exit(0);
```

If we load with the `exload` call then the output from the above code would be

```
We were given 0 arguments
```

whereas the second call, to `exloadl`, would give

```
We were given 3 arguments
 go
 very
 fast
```

In this way we are able to create node programs that have some knowledge of what we want them to do - even without invoking the communication system.

As we alluded previously there are many other high level allocation and loading procedures. For example we can supply arguments as a NULL terminated vector, or optionally load different arguments into different nodes. We can also load different **programs** into different nodes and alter the setting of their environment variables, or any combination of these techniques. The details are left for the reference manual and several useful examples are described in section 7.

Sharing groups of nodes between multiple host processes

Finally, among the loading and allocation options, we must discuss another possibility; processor group sharing. Occasionally it will be convenient for multiple host processes to share access to the same group of nodes. A simple example might be a host file server and a graphical interface, or a regular application and the debugger. This facility is available through the `exshare` function. To share a processor group with another process it is sufficient to know its process-ID number and make the call

```
        pgindex = exshare("/dev/transputer", pid, &Nnodes);
```

The first argument is the name of the device containing the nodes and is used in the same manner as in the `exopen` call. The second argument is the process-ID and the last is a pointer under which will be returned the number of processors in the shared group. The returned value is treated identically to that returned by `exopen`. If negative it indicates that the attempt to share nodes failed for some reason. A positive value returned from `exshare` is the processor group index to be used in subsequent references to the shared nodes.

Express process IDs

A tricky point in this regard is the specification of the process ID with which you wish to share nodes. The only totally reliable way to find this is to remember what `exopen` told you when the processors were originally allocated. This number is always valid. A second, less reliable mechanism on UNIX-like machines is the `expid` system call. The single argument to this function is the UNIX process ID of the process which originally allocated the nodes. The returned value if that which `exshare` will understand. On non-UNIX systems the same information may be available by using the `exstat -l` command which also indicates the necessary process-ID.

Race conditions between multiple host programs

An important point to make in connection with sharing processors is that the user is responsible for avoiding potential clashes between requests. When reading messages, for example, the user must be aware of the fact that the two, or more, host processes are "racing" against one another and that the order in which their requests are satisfied cannot be guaranteed unless some precautions are taken at the user level. The next section discusses this problem in some detail and offers routines to manage these difficulties.

Many routines have been introduced in this section. Fortunately most applications need only two of them; `exopen` and `exload`. It is even easy to write a simple subroutine which combines both! The extra sophistication available in *Express* is provided for those applications which do not fit simply into a "homogeneous" mold and where it is crucial to be able to do "different things" in "different nodes". After all, this is merely another way of expressing parallelism.

4 Node Addressed Interprocessor Communication

The underlying model of communication

The lowest level of the *Express* system is based on an asynchronous, point-to-point message passing system. This means that messages may be sent from any node in the system to any other node (including any attached host processor) at any time and the kernel is responsible for intermediate buffering and routing of the messages. This level of operation is available to user applications as well as the kernel and may be freely interchanged with the system described in the section 5.

Terminology: "asynchronous" or "blocking"

A system such as this is typically termed "asynchronous". This is rather a misnomer - in most implementations the "read" function call blocks until a message has arrived and been read; not very asynchronous behavior. A better term would probably be "long range" describing the fact that messages may be sent to arbitrary destinations with the intermediate routing being handled by the *Express* kernel.

In fact *Express* does offer both synchronous and asynchronous functionality and several intermediate flavors too. For simplicity we will divide this section and describe the two modes separately since they typically occur in different types of applications. Before diving into specific details of the message passing primitives a few basic principles which hold throughout these discussions will be discussed.

What is a message?

The communication system is built around the concept of a "message". Of the several interpretations of this term the one adopted by *Express* is that a message is the result of a single call to one of the "write" functions. It contains a number of bytes but is otherwise without structure. Each message will be consumed by a single call to one of the "read" functions - if the receiving node wanted fewer bytes than were sent the extra are discarded and if less were sent than were required then the read function takes what it sees and returns a status value to its caller indicating that not enough data were sent.

Contrast with UNIX pipes

This picture is rather different from a "pipe" style where any extra bytes in a message would remain to be picked up by a subsequent read operation and if too few bytes were sent a "read" would wait for more to arrive before returning. This approach is quite tricky to program - the message approach makes bookkeeping easy since a single "read" should go with each "write".

Message "types"

As well as the obvious attributes of a message (length, data, source and destination) *Express* also associates a "type" with each. This value is used to allow various types of decision to be made on the receiving node about exactly which message is to be read. In a multi-tasking environment, for example, different message types can be used to specify that certain messages are intended for one task rather than another. Alternatively in a real-time environment one might set aside certain message types for immediate processing while assigning the others somewhat lower priority.

Differentiating between incoming messages

In order to select between messages when executing a "read" function one has an "acceptance criterion". Each send operation associates a positive type value with the message. When a message is to be read the receiver specifies which node to read from and which type of message to read. If several messages satisfy the acceptance criterion then the one which arrived first is read. Varying degrees of `DONTCARE` or "wildcard" behavior can be specified to allow great flexibility in selecting between possible messages. It is also possible to specify that certain message types should be processed by a second process rather than being delivered to the user application.

Ignoring message types

While the "type" concept is central to the *Express* communication system it is always an "option". In fact many applications don't ever need to deal with these features - all messages can be of a single type and every read operation has a well defined source to read from. Again, however, the extra functionality is required by some applications.

4.1 Messages, Nodes and Types

Before describing the functions used for interprocessor communication a few basics concerning the *Express* implementation should be explained.

The structure of a message

A message is a stream of bytes with no structure. The user is free to send any data to any node at any time. *Express* does not attempt to interpret the message in any manner. As a result you are free to send integers, floating point values, strings, structures, whatever you like as long as your code supplies all the necessary knowledge about the data. If the receiver needs to know what sort of data is coming then it can either be encoded in some user-level protocol or the message type.

Message destinations

Every *Express* message is addressed to a "node" - a specific processor either within the parallel processor network itself or attached as a "host". This latter category is somewhat vague; a "host" is a processor, attached to the parallel processor network which is conceptually capable of providing system services such as graphics, disks or just downloading user applications. One particular host is distinguished for every application that is loaded into the machine. It has the special node number `HOST` and is the machine which actually started up your application. Since this machine is the obvious target for your I/O and operating system service requests it has a special identifier. (In all respect it is just like another node in the network - you send and receive messages to it in the same way.)

Sending messages to hosts

Express also supports the idea of multiple hosts. This means that your parallel processing system may have several different types of host providing the sort of system services described in the previous paragraph. To send a message to one of these hosts (other than THE `HOST`, of course) you need to know its *Express* processor number. To find this it is probably simplest to consult the information provided by the system configuration tool, `cnftool`. It can provide you with a picture of the interconnections between nodes and hosts and also with a small integer for each host. To translate this number into a processor number for *Express* you merely "turn on bit 15"! While this sounds at first like the worst kind of black magic it is actually quite simple in practice; the value 0 becomes `0x8000` (hex) or 32768 decimal, the value 1 becomes `0x8001` or 32769 ***etc.***

16 vs. 32 bits - a potential problem

The use of 16 bit values is of some concern here. One must always be aware of the differences between host and node processors. It is most common in the current parallel computer systems for the nodes to be 32 bit machines while the hosts are only 16 bit. This has several consequences for *Express*.

Firstly the host cannot read as long messages as the nodes. For simplicity the argument to the *Express* functions which describes the message length is an `int` (Except for the `exreadfd` and `exwritefd` functions which use long integers.) which means that biggest message the host can read is of length 32767 bytes - much shorter than the nodes. Secondly "host" processor numbers are distinguished by setting bit 15 in both host and nodes even though this restricts the number of processors in the network to 32768!

Restrictions on message types

The final restriction is in the message type argument. Certain types of messages are treated specially by *Express* and cannot be used in applications. As a result user message types must lie in the range `0 - 16383` - i.e., they cannot have either of the two highest bits set.

While it is generally true that one can ignore the questions discussed in this section it is probably wise to keep these points somewhere to hand since they are an irritating source of minor problems. The minor differences between host and nodes is often unnecessarily unpleasant - in the worst case one might find that a perfectly acceptable value in one place is total garbage in the other because the byte ordering of the two machines is different. This is one of the major strengths of the *Cubix* programming model - since you only write a program for the nodes it never has to deal directly with any hosts. This system is discussed briefly in section 6 and at greater length in the document "Programming Parallel Computers Without Programming Hosts".

4.2 Blocking Communication Functions

In this section we discuss the simple "blocking" primitives. Once again this adjective is rather a misnomer - the write function, for example, never blocks. However the concept implies something at the level of the application programming model and so it remains.

The basic communication system

The basis of the system are the three routines `exread`, `exwrite` and `extest` which are available both on the host computer and within the nodes themselves. Messages are sent with the system call

```
exwrite(buffer, length, &destination, &type);
```

in which `buffer` is a pointer to the data to be sent and `length` is the number of bytes of data. No restrictions are placed on the message length - any length from zero to the maximum integer are allowed. It is often the case, for example, that zero length messages can be used to some effect - interprocessor synchronization, for example. The `destination` argument specifies the processor to which the message is to be sent and the `type` argument specifies a "message type" for the data. These "types" provide the mechanism used by readers to differentiate between various messages and also to allow multiple readers on a processor to access different message streams. Notice that both `destination` and `type` are specified by pointer values which may seem rather unnatural. This is done for uniformity - the "read" functions specify the source node and type with pointers so *Express* adopts this policy uniformly. This prevents later confusion - especially in the functions which combine the send and receive functionality!

Argument types and ordering

A slightly less obvious point concerns the use of the wildcard system for processor numbers and message types. As we will see shortly a program may request a message to be read from *any* processor or with *any* type by using the special values `DONTCARE` in its calls to `exread`. Analogously one might expect to be able to send a message whose type is irrelevant. Sadly this is not so. If you attempt to send a message whose type or destination is `DONTCARE` *Express* will return an error.

Messages are received with the system call `exread` which has the calling sequence

```
status = exread(buffer, length, &src, &type);
```

The important fact about this function is that it blocks until an acceptable message has been read - i.e. the process making the `exread` call stops executing until it has received a message.

The "acceptance criterion" allows readers to pick between incoming messages

Furthermore the `src` and `type` fields potentially make the `exread` call very particular about which messages can be read. The method of operation is as follows: all messages that have arrived at this processor are examined in turn and the first one that meets the requirements specified in the `src` and `type` fields is selected for reading. Then a maximum of `length` bytes are transferred to the user supplied `buffer` and any extra are discarded. The number of bytes read is returned to the user as the `status` - if the message read is actually shorter than `length` then this smaller value will be returned. If any type of hard error occurs `status` will have the special value -1. If more than one message on this node meets the selection criterion then the one that arrived first is accepted.

The method for choosing acceptable messages is to examine the `src` and `type` variables. Both are considered, in turn, and both must match for a message to be accepted. The matching procedure is the same for both `src` and `type` fields (The `type` field is actually subject to one extra level of processing - see the calls `exinctype` and `exexctype`.) and depends upon the values initially supplied in the call to `exread`

- If the NULLPTR pointer is supplied as an argument then ANY value is considered a match.
- If the supplied value is `DONTCARE` (Defined in the header file `express.h` then any value is considered a match **AND** the accepted value is written over the supplied parameter. It is for this reason that the `src` and `type` parameters are supplied as pointers rather than explicit values.
- If the supplied value is any positive (or zero) value then only an exact match is allowed.

A couple of example program fragments should make this process clearer. We assume that two integer variables `msg_src` and `msg_type` have been declared elsewhere and that `buffer` is a region of memory sufficient to hold up to 512 bytes.

```
                        /* Read ANY type from ANY node */
        #include "express.h"       /* Defines NULLPTR */

        exread(buffer, 512, NULLPTR, NULLPTR);
```

Using the NULLPTR value

Since both source and type fields are `NULLPTR` pointers then any message on this node will be accepted for reading. Up to 512 bytes of the first message to arrive at this processor will be read into the user buffer.

```
        #include "express.h"   /* Defines DONTCARE macro */

        msg_src = DONTCARE;
        msg_type = 3;
        exread(buffer, 512, &msg_src, &msg_type);
```

Using the DONTCARE value

In this example the source field has the wildcard value `DONTCARE` so a message will be accepted from any node. The type field is explicitly set to 3 so that only messages of this type will be read. Upon the completion of this call the `msg_src` variable will contain the processor identification of the node which sent the message that was read.

Using combinations of these parameters it is possible to control exactly the messages that will be read by any process executing the `exread` call. Note that the special value `HOST` is available in the header file `express.h` to indicate the host computer as either a message source or destination.

Testing for an incoming message

As stated earlier the call exread blocks until a suitable message has been read. This behavior is not suited to all applications so an extra call extest is available to facilitate non-blocking read processes. The syntax is

```
        status = extest(&src, &type);
```

where the src and type are interpreted exactly as in the exread system call - i.e., they are used to distinguish between various messages on a node. The difference between this function and exread is that extest returns immediately. The returned value, status, indicates whether or not a message was found that matches the supplied src and type parameters. A negative value indicates that no suitable message is currently available (and hence that a corresponding call to exread would block) while any other value is the length of the message that would be read with an immediate call to exread with the same parameters. Note that the wildcard value DONTCARE is interpreted in just the same way as in exread and that it will be replaced with the actual source or type in the call to extest.

Possible "bugs" when using wildcards

While the wildcard values offer useful possibilities to programmers a problem exists when one considers multiple processes on a node both attempting to read messages. This is exemplified by he following piece of code

```
     Process 1.
          msg_src = DONTCARE;
          msg_type = DONTCARE;
          status = exread(buffer, 512, &msg_src, &msg_type);

     Process 2.
          msg_src = 3;
          msg_type = 145;
          status = exread(buffer, 512, &msg_src, &msg_type);
```

Note that the programmer has been quite careful - only one of the processes uses the DONTCARE value while the other specifies exactly what source/type combination is acceptable - and yet the code still fails intermittently. The problem is that the way this code works is time dependent. When the message from node 3 with type 145 arrives it can still go to either process because it is acceptable to both. Obviously if a message of type 144 arrives first then all is well because process 1 will read it allowing process 2 the second message. However, if the two messages arrive in the wrong order then process 1 will happily read the type 145 message leaving process 2 high and dry waiting for its message and ignoring the message of type 144.

Restricting the range of acceptable message types

The problem here is that the wildcard mechanism is TOO wild! In order to correct this situation *Express* offers two functions to alter the functionality offered by the wildcard values in the "type" field; exinctype and exexctype. Both functions have the same format

```
        exinctype(lotype, hitype);

        exexctype(lotype, hitype);
```

The arguments to this function specify an inclusive range of message types to be either considered or rejected in matching wildcard values.

The way these functions work is slightly different. After a call to `exinctype` only the given type range will be considered in matching `DONTCARE` arguments in `exread` and `extest` calls. All other types will be ignored. On the other hand `exexctype` specifies a range of types that will be ignored when matching wildcards - all other types will remain acceptable.

Using these functions it is possible to set up multiple processes on a single node both reading with wildcards but without interfering with each other. In particular the previous code segment can be fixed in several ways, for example

```
     Process 1.
          exexctype(145, 145);
          msg_src = DONTCARE;
          msg_type = DONTCARE;
          status = exread(buffer, 512, &msg_src, &msg_type);

     Process 2.
          msg_src = 3;
          msg_type = 145;
          status = exread(buffer, 512, &msg_src, &msg_type);
```

One might note that if process 1 were to later on want to read a message of type 145 then the call

```
        exexctype(DONTCARE, DONTCARE);
```

would indicate to the kernel that it should henceforth consider all types in matching wildcard arguments.

Obviously this mechanism requires some sort of coordination between the processes executing on a processor. Typically, however, this is not difficult to arrange and the benefit to be accrued from the continued availability of the wildcard values is a very useful feature.

The function calls described so far in this section provide the basis for all interprocessor communication facilities. However, whereas most systems stop at this point *Express* continues to add extra functionality to the system designed to meet the requirements of typical user applications. It is obviously true that the routines described can already form the basis for almost arbitrary communication strategies but these often require some care in their implementation so *Express* offers several additional functions.

"Global" communication functions

`exbroadcast`	Perform a broadcast operation to some or all the nodes.
`exchange`	Combine the send and receive operations into a single function.
`excombine`	Gather messages from some or all the nodes and apply some "combining" function to the data - useful for calculating global averages, maximum and minimum etc.
`exconcat`	Gather messages from some or all nodes into a single buffer.

`exreadfd`	Similar to `exread` but write the message to a file.
`exsync`	Synchronize all processors.
`exvchange` `exvread` `exvwrite`	Similar to `exchange`, `exread` and `exwrite` but non-continuous memory blocks can be sent in a single message - useful for dealing with rows and columns of matrices.
`exwritefd`	Similar to `exwrite` but take the message to be sent from a file.

Together these functions provide a user interface which should be sufficient for the vast majority of user applications. The `exchange` routines are particularly powerful - in section 5 we develop an example program in its entirety using this function and the automatic decomposition tools.

Due to our choice of introducing the `exread` and `exwrite` functions first the reader may well be assuming that these should form the basis for all interprocessor communication in *Express*. In our view, however, the higher level functions provide a much simpler interface between the user and the parallel computer and their use should be encouraged wherever possible. In particular these functions are designed for use with the automatic domain decomposition system to be described in a later section which allows the user to program in a manner totally independent of the underlying hardware architecture. It is hard to overemphasize the power of this programming style and we strongly recommend users to study its application to their own problems since it provides a level of portability and abstraction which can actually make parallel processing easy.

"Deadlock"

These functions provide the basis for the communication system under *Express*. As mentioned earlier the *Express* kernel is responsible for routing messages between any pair of processors and also for optimizing broadcast operations. A crucial issue in this regard is "deadlock". This occurs whenever the internal kernel buffers overflow or whenever some particularly strange combination of read and write requests leads to a situation where no single processor can proceed. In this case there is usually no recourse but to reset the machine with `exinit` and try to figure out the problem area. Fortunately there are known, deadlock-free, routing strategies for certain processor interconnects: the hypercube and two-dimensional mesh. Both of these options are supported within the *Express* kernel. Correct, problem free, routing cannot be guaranteed on other topologies.

The advantage of "blocking" communication

The routines described in this section form the basis of all inter processor communication facilities. The strategy they implement, however, is "blocking" in the sense that a "read" function will hang if there has been no "write" function to send it data. This programming style is actually very powerful - bugs appear repeatably and without time-dependencies leading to simple diagnosis, especially with a debugger such as *ParaSoft*'s `ndb`.

4.3 Non-blocking Communication Functions

Real-time systems and double buffering

Occasionally an application will arise that has particular requirements not easily met by the functions described in the previous section. Important examples are the fields of real-time control, in which it is important to react quickly and flexibly to input data and "pipelined" operations in which one may wish to process one set of data while waiting for another to arrive. Both of these applications are characterized by the requirement of a non-blocking read function.

In real-time systems one would like to set up a service, or services, which will accept messages and process them without program intervention while another part of the application continues to process data from other sources.

In the pipeline case one wants to set up a read request that does not block the calling program which is free to continue processing currently available data. Eventually new data will have arrived and can be processed while possibly waiting for still more messages.

Express provides functions for both these applications.

exhandle and interrupt processing

`exhandle` provides a mechanism for "handling" messages as soon as they arrive at a processor. The calling sequence

```
    type = 123;
    src = DONTCARE;
    exhandle(proc_msg, &src, &type);
```

indicates that every message of type 123 is to be processed by the user specified function `proc_msg`. From this point on *Express* will respond to any message of type 123 with a call such as the following

```
    kernel_src = 12;
    kernel_type = 123;
    proc_msg(kernel_ptr,length,&kernel_src,&kernel_type);
```

Speed of processing messages

Notice that this has the same calling sequence as the `exread` function described in the previous section with some important differences. The pointer supplied as the first argument is the address in *Express* memory where the message is to be found - no time has been spent copying the data to a user memory area. The `kernel_src` and `kernel_type` fields denote the source and type of the message respectively and `length` is its size.

Note that the user "sees" the message with the absolute minimum of delay - it is essentially passed to a user level routine for processing as soon as it has arrived, interrupting normal program flow. Notice that this means that the user routine must take special precautions if it wishes to retain the message - otherwise the kernel will discard it after the application-level call completes.

Communication "interrupts"

This style of processing is often referred to as "interrupt time" since many implementations generate a "hardware interrupt" when a message arrives that causes some action to be taken. Usually the action taken is for the kernel to arrange to buffer or forward the message but `exhandle` allows the user the first "crack" at the data.

This function actually forms the basis of an extremely elegant multi-tasking programming style under *Express*. In an abstract sense it can be said to provide a mapping from the space of message types into the set of functions/subroutines within a program. This mapping essentially allows us the freedom to execute, *from another processor*, any function in the program and to simultaneously pass it the data it needs.

The problems with asynchronous programs

In connection with this style of programming one should mention that asynchronous programming is rather tricky. One has to indulge in protection of critical code sections with semaphores and the like and must be careful with global variables, etc. Bugs tend to occur in a haphazard and unreproducible fashion which makes debugging much harder than it might otherwise be.

Load balancing

A simple example of this sort of processing is the construction of a "load balancing" supervisor. Consider a parallel system in which work is being generated by some phenomenon - for example, turbulence is developing over some body necessitating extra processing. This additional workload is to be distributed among the parallel processors by some "task creation" scheme. In order to load-balance such a strategy one might wish to provide some means of gathering statistics. The following code segment treats all messages of type 99 as requests for load information and returns a message to the inquiring node about the total workload in this node.

```
#include "express.h"
#define LOAD_REQ(99)
#define LOAD_ACK(100)

extern float loadvector[18];
           /* Data about processor loading */

main()
{
    int type;
    int src;
    int load_avg();

/* Set up message handler to service requests for
 * load information
 */
    type = LOAD_REQ;
    src = DONTCARE;
    exhandle(load_avg, &src, &type);

/* A good idea is to "sync" the processors after
 * installing a handler so that no-one sends off
 * messages to a node that isn't ready yet.
 */
    exsync();

/* Proceed with simulation .........

}
```

```
load_avg(ptr, length, psrc, ptype)
char *ptr;
int length, *psrc, *ptype;
{
    int rtype = LOAD_ACK;

    exwrite(loadvector, sizeof(loadvector),
                              psrc, &rtype);
    return 0;
}
```

Notice that the handler is very simple in this case - it merely returns the `loadvector` parameter to the requesting node in a message of a different type as the request. Obviously much more complicated structures could be constructed. It is very important that the returned message be of a different type to that received. If we returned the results in a message of type `LOAD_REQ` it would get picked up by the message handler on the requesting node which would, in turn, bounce it back to the other node and so on, ***ad infinitum***. This is typical of the strange bugs one can generate with asynchronous message handlers.

An tricky "bug" in writing exhandle routines

Invoking the message handler is similarly simple

```
get_load(node, data)
int node;
float *data;
{
    int stype = LOAD_REQ;
    int rtype = LOAD_ACK;

    exwrite(NULLPTR, 0, &node, &stype);
    exread(data, 18*sizeof(float), &node, &rtype);
}
```

Notice that we send a zero length message to the node whose load data we wish to get - after all the data in the message is going to be ignored on receipt anyway. We then read the results back from the same node into a local buffer. Notice that there are no constraints on the use of this function - we can even use it to enquire about our own loading. (Some precautions may have to be taken to ensure that a message is not sent to a node before it has started up its handling routine - hence the call to `exsync` in the previous example).

Zero length messages are OK

A final note in connection with the `exhandle` routine is that the connection between messages and the handler need not be permanent - in fact it is terminated whenever the handling routine returns a negative value to the kernel. This facility can be used to install once only handlers or those that function only until some specific condition applies.

Using exhandle to

exreceive - a "non-blocking" read function

The second style of asynchronous processing is that of a non-blocking read. This is implemented with the call

```
exreceive(buffer, length, &src, &type, &status);
```

The first four arguments to this call are treated exactly as in the corresponding call to `exread`. The difference, however, is that this function returns immediately to its caller irrespective of whether or not a message has been read allowing processing to continue. If no message is available at the time of the call then the value -1 is written under the `status` variable and the `src` and `type` arguments are left unchanged. When a message finally arrives the `status` value is updated to reflect the length of the message read and `src` and `type` fields are also modified to denote the actual message parameters.

"Double-buffering"

This function is of use in many types of application. Since I/O bandwidth is often low on parallel processors, especially when compared to brute CPU power, "double-buffering" is a good strategy - one buffer is written to disk while another is being read through the communication system. Similarly graphics applications benefit from such treatment.

Signal processing

In the following example we assume that the parallel machine is being used in a signal processing "pipeline" - each node performs a particular processing phase and passes its result on to the next "black box" in the chain. In this case it is important to keep data flowing smoothly through the pipe. For simplicity we assume that incoming data buffers are of length 1024 bytes and must be processed by the `sig_proc` function before being passed on to the next node. We use two message types: `PROCESS` for most buffers and `FINISHED` for the last buffer. This is again rather artificial but serves to illustrate one of the trickier points of the "double-buffering" technique - stopping it when it's done!.

```
#include "express.h"

#define PROCESS  (0x7001)  /* Type for grinding on */
#define FINISHED (0x7002)  /* Type for "done" */

char buffer[2][1024];/* For buffering data */

process(innode, tonode)
int innode; /* Source of data for processing */
int tonode; /* Destination for next processing phase.*/
{
    int done, type[2], this, next;
    int stat[2];

    done = 0;/* Not done yet */
    this = 0;/* Start using "slot" 0 */
    next = 1;

/* Get first buffer, blocking read this time */

    type[this] = DONTCARE;
    stat[this] = exread(buffer[this], 1024,
                                 &innode, type+this);

          do {
              if(type[this] != FINISHED) {
                              /* Read next block */
                  type[next] = DONTCARE;
                  exreceive(buffer[next], 1024, &innode,
                       type+next, stat+next);
              }
              else done = TRUE;/* Finish after processing
                             this block */

      /* Process the oldest buffer and send the result to
       * the next node with the same type as we received. */

              sig_proc(buffer[this], stat[this]);
              exwrite(buffer[this], stat[this],
                              &tonode, type+this);

      /* If we're not done wait for next buffer */

              if(!done) {
                  while(stat[next] < 0);
                  next = (next + 1) % 2;
                  this = (this + 1) % 2;
              }
          }
          while(done != TRUE);
      }
```

Notice that we have saved the incoming message length for passing to the processing function. This is superfluous in cases such as signal processing where all buffers are (presumably) of the same length but again serves to illustrate a more general case. Also important is the duplication of all status and type information. This has to be preserved since the value of the `stat` and `type` variables get overwritten whenever a new message arrives - an asynchronous event.

Another potential bug with asynchronous processing

In common with the `exhandle` call all the standard interpretations for the `src` and `type` arguments are valid as is processing by the `exinctype` and `exexctype` functions.

The previous paragraphs describe and motivate two non-blocking read functions that have particular applications in application areas. These functions are part of the *Express* library because they supply necessary programming paradigms. One additional non-blocking function is supplied solely for reasons of speed, `exsend`. This function is analogous to the `exwrite` function in that it sends a message to another processor. The difference is that whereas `exwrite` waits until the message has been sent before returning to its caller `exsend` returns immediately. The calling sequence for this function

```
        exsend(buffer, length, &dest, &type, &status);
```

can be seen to directly match that of the `exreceive` routine and, in fact, the use of the additional `status` argument is identical - its value is initialized to -1 by the system and is changed to the length of the transmitted message as soon as the data has been sent.

This function is actually very useful and while its motivation appears to be solely on grounds of speed it can be used in most cases where `exwrite` would normally be used. The only real difference between the two routines is that since the data has not necessarily been sent when the call to `exsend` returns one should be careful not to modify the data contained in the message `buffer` until the `status` variable indicates that it is safe to do so. It may, of course, be possible to imagine applications in which it is safe to modify the data even before it has been sent, but we have been unable to do so!

As a final note we might point out that a call to the standard `exwrite` function is functionally equivalent to the sequence

```
        exsend(buffer, length, &dest, &type, &status);
        while(status < 0) exsleep(1);
```

5 Topology Independent Communication

The relation between hardware and software topologies

The previous sections described a communication system that directly addresses the processors in the parallel computer by sending and receiving messages addressed according to their processor numbers. This strategy typically involves a certain degree of user intervention in the placement and distribution of data to make best use of the parallel machine - for instance it makes sense to have data decomposed in such a way that processors that need to communicate frequently are neighbors in the hardware topology.

Topology independent communication

While this layer is useful (and in fact necessary) for some applications another level can be provided in which no knowledge of the underlying topology is required. We have found this level to be extremely important in the regular problems common in scientific applications since some degree of automatic decomposition is possible which hides most details of the parallel machine from the user. Hiding machine dependencies in this way also enhances the portability of the resulting code - since it is independent of the underlying processor topology the code can be implemented on a wide variety of architectures including both shared and distributed memory machines and also sequential computers. (This latter point is often unreasonably neglected. Since software development is typically extremely expensive it is very unfortunate if a program that has been successfully parallelized cannot be run (and maintained) on a regular sequential computer.)

5.1 Automatic Decomposition and Run-Time Configuration

How many processors are we using?

One of the most important features of parallel processors is reconfigurability. When running on a sequential machine one has limited options - the program runs and that's it. A parallel processor has many more dynamic features; particularly the availability of different numbers of nodes. It is particularly important for a parallel program to know the details of its run-time environment; how many processors are available, how to communicate with the host computer, etc. Within *Express* this information is made available with the call exparam. The header file express.h defines the following C structure.

```
    struct nodenv {
        int procnum;    /* Processor number */
        int nprocs;     /* Number of processors in group */
        int groupid;    /* Identifier for processor group */
        int taskid;     /* Identifier of calling task */
    };
```

The elements in this structure are used to specify the runtime environment within which a process finds itself. The procnum and nprocs fields specify how many processors are currently active and uniquely identify each processor within a ***processor group***.

This information is obtained at run time by executing the exparam system call whose argument is a pointer to a nodenv structure that will be filled in by *Express*. The following is a sketch of the appropriate code

```
        #include "express.h"
                   /* Defines nodenv structure */

        {
            struct nodenv nodedata;

            exparam(&nodedata);
                       /* Get runtime parameters */
        }
```

Automatic decomposition routines - exgrid

The automatic decomposition mentioned in the heading of this section is implemented in a set of function calls collectively known as exgrid. Their purpose is to take a user specification of a problem domain and perform a mapping to the underlying processor topology. The system then makes available any "node numbers" that may be required for use in communication calls. In this way the user never has to understand the exact location of the processes in the application or which nodes they have to communicate with - all this is handled transparently.

Protecting the user from the hardware

"Data parallelism"

The "incantation" that makes this magic happen is for the user to specify the way that data is to be distributed over the processors. Extracting parallelism this way is often known as "data parallelism" and is very common in a wide range of application areas. Basically the user informs the system of the way that the application level data is distributed and *Express* contrives to hand back the parameters that cause the correct communication to be performed. Note that this is just the opposite of the conventional approach in which the user is presented with a given topology by the system and has to make the best possible use of it.

Automatic decomposition - an example

To expand these ideas a little consider the following example; a model of road traffic in a major city. For simplicity we shall assume that the net of roads is evenly spaced in both directions and that we will ignore load balancing concerns or other abstruse properties of parallel machines.

Our problem is shown diagrammatically in the upper part of Figure 2.

A traffic-flow problem

The road network is shown by the solid lines and a set of eight processors are delineated by the dashed lines. The basic idea is that each processor in the system will be responsible for a subregion of the total road network, moving cars around and generally controlling things. An example subregion is shown in the center of the same figure. As any car reaches the edge of the area controlled by a particular processor we assume that it has to get sent to the processor who controls the neighboring roads.

Essentially the question becomes one of assigning the subregions to the processors and working out how to communicate with neighboring areas; this is the purpose of "`exgrid`".

The "dimensionality" of the physical system

To set up the problem we have to inform the system of the dimensionality of the user problem and how many processors should be assigned to each of these dimensions. Note that these quantities are specified in the space in which the users problem lies rather than the abstract space defined by the topology of the parallel processor network. In the case of road traffic the problem has dimension two - at least if we don't have overpasses, tunnels etc.! In order to assign the number of processors in each direction we can either hard code some values or else use the `exparam` and `exgridsplit` system calls to distribute the number of available processors at runtime. For the sake of simplicity we adopt the convention that dimension 0 will denote North-South and dimension 1 East-West. Then we initialize the system by making a call to `exgridinit` as follows

```
#define North_South    0   /* For convenience */
#define East_West      1
{
    int nprocs[2], dimension;

    dimension = 2;
                /* N-S and E-W ==> 2 dimensions */
    nprocs[North_South] = 2;
                /* Two processors to N-S direction */
    nprocs[East_West] = 4;
                /* Four processors to E-W direction */

    if(exgridinit(dimension, nprocs) < 0) abort(-1);
```

Assigning processors to the physical domain

Notice that we took the easy way out here and just hardwired the fact that there will be eight processors working on the problem decomposed as a two by four mesh. We could do better by making a call to `exparam` and dividing up the `nprocs` field to make the mesh as nearly square as possible. The code to do this has the form

```
#include "express.h"/* Define nodenv structure */
#define North_South    0
#define East_West      1
{
    struct nodenv nodedata;
```

Street Map

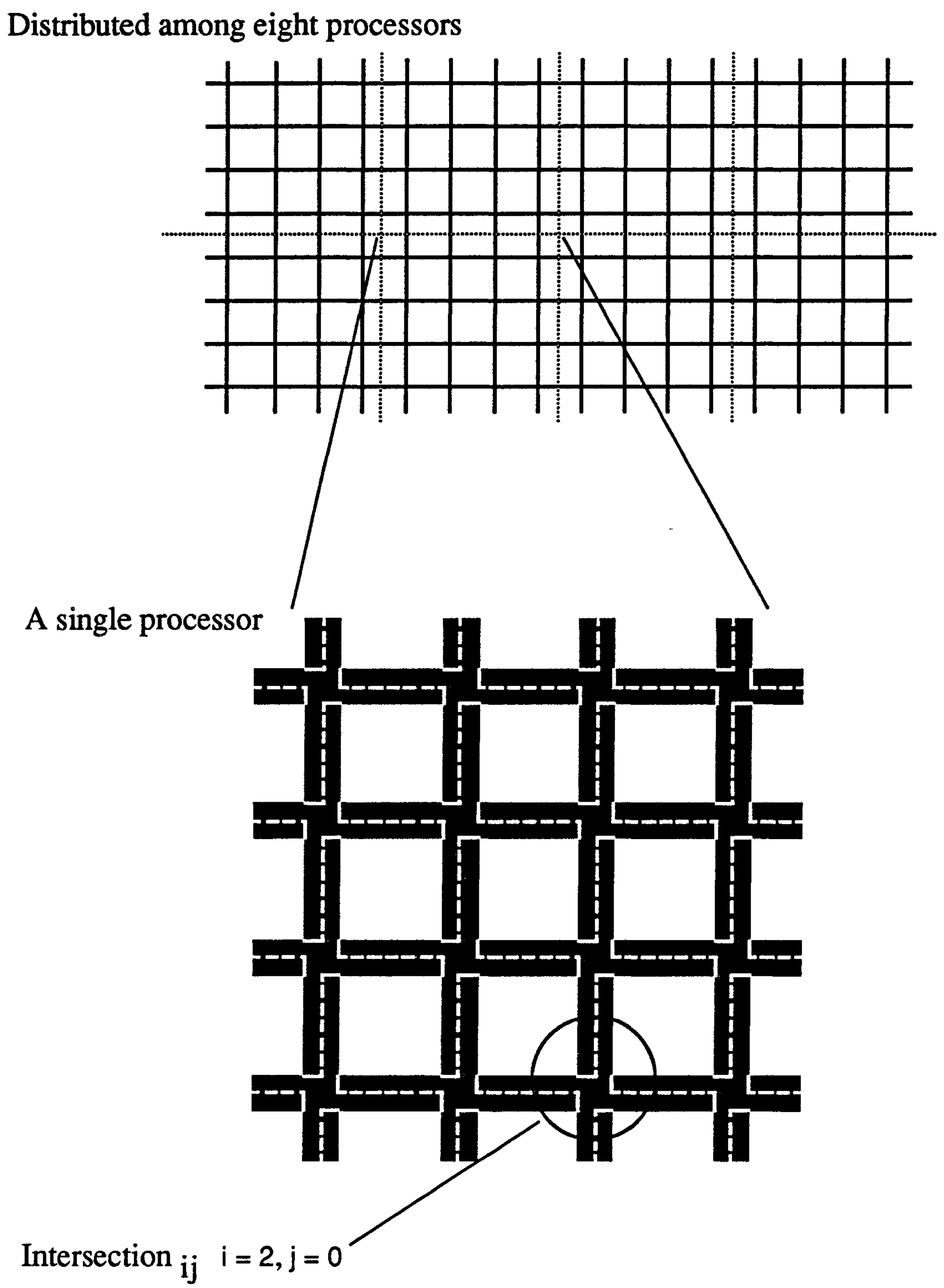

Distributed among eight processors
A single processor
Intersection $_{ij}$ i = 2, j = 0

Figure 2. Decomposition of road network problem for eight nodes

```
        int dimension, nprocs[2];

        exparam(&nodedata);/* Get runtime parameters */

        dimension = 2;
        exgridsplit(nodedata.nprocs, dimension, nprocs);
```

The `exgridsplit` function takes as arguments the total number of processors and the number of dimensions to decompose over and returns, in the array pointed to by the last argument, a "square" decomposition of this many processors. For example a two dimensional decomposition of 8 processors would yield a 4 x 2 decomposition while 9 nodes would give 3 x 3. No account is made for "silly" input values - eleven processors would yield an 11 x 1 decomposition which is probably less efficient than only using ten processors!

Express and the real physical problem

Express now understands that we are modeling a two dimensional real-world situation and that we have some number of processors in each direction. Next we can go ahead and find other parameters of our decomposition. One we might need to know is the coordinates of a particular processor in the physical grid. For instance we might know that several roads are closed in the most South-Westerly region and the processor controlling that area has to be able to make decisions relating to this fact. To do this we use the `exgridcoord` function which takes as arguments a processor number and returns the coordinates of that node in the grid. So, for example, the following code might be used to pick out the processor with all the closed roads. (Assume the same macros and headers are defined as before)

Finding out which processor is where

```
    {
        int recpnum[2];/* For the processor coordinates */

    /* Code to setup and call exgridinit and exparam as
     * before.
     */

            ...

    /* Find global coordinates and treat closed streets */

        exgridcoord(nodedata.procnum, recpnum);
        if(recpnum[North_South] == 0 &&
           recpnum[East_West] == 0) {

            ... /* Code to deal with traffic congestion */

        }
        else {

            ... /* Code to deal with easy case */
        }
```

Note that the user determined coordinate system has intersection (0,0) at the South-Western corner of the mesh and hence the code in the above `if` statement.

Communication without processor numbers

As well as allowing access to this sort of information ***defined totally within the user problem domain*** a fundamental purpose of the `exgrid` tools is to facilitate communication between processors. The utility which allows this is `exgridnode` which calculates the "destination" parameters associated with communication to any of the neighboring processors in the physical domain. The use of this routine is easily explained by example. Suppose that a processor needs to communicate with its neighbor to the East. Then the following call calculates the appropriate destination.

```
    #define North_South    0
    #define East_West      1
    {
        struct nodenv nodedata;
        int dest;

        exparam(&nodedata);
        dest = exgridnode(nodedata.procnum, East_West, 1);
```

The first argument specifies which processor to start from and the next two give the direction in which we wish to go; the second argument names the basic direction - North/South or East/West and the last says how many "hops" we want to make - positive values indicate motion along the positive axis and negative values along the negative axis. In our case we wanted the next processor along to the "East" which is along the positive direction. To find the node for the "West" direction we simply change the final argument to -1.

Boundary conditions

Notice that we have skipped over the question of "boundary conditions" in this discussion - i.e., what processor is directly to the East of the most Easterly and so on. We will return to such matters in the next section where we will also discuss what is actually done with values calculated by `exgrid`. Before doing so it must be emphasized that the `exgrid` utility is actually very general. You can adopt ring type structures in which the physical decomposition is basically one dimensional (An example might be freeway traffic) or multi-dimensional decompositions such as might be suited to modeling other real-world phenomena. In each case the procedure is basically the same. Also one is not restricted to inquiring about conditions and neighbors of your own processor - one can discover the environment of any processor in the decomposition.

Notice also that we have yet to make any statements about the nature of the parallel processing system on which we intend to execute this program. We will not, in fact, make *any* such statement throughout the entire development sequence. One of the beauties of the `exgrid` decomposition system is that it lets us express the parallelism of the program in "physical" terms - i.e., in terms related to the inherent parallelism of the problem we are trying to solve. In some sense we can make the claim that we are really still writing sequential programs but planning to execute them in parallel.

5.2 Using the Automated Decomposition Tools

In the previous section we discussed how the exgrid utility can be used to generate "topology independent decompositions" - i.e., ones in which the decomposition is carried out in the domain of the application rather than the underlying connectivity of the parallel computer. In this section we will show how these tools are used in conjunction with the communication primitives discussed in section 4.

The traffic flow model

In order to do so we will add a little more detail to the previous example. We will make a drastic simplification and assume that traffic is only allowed to proceed from West to East and from South to North - i.e. in the positive direction along each axis. Furthermore vehicles are not allowed to make turns. We can now represent the state of the traffic flow by having two dimensional arrays whose elements are the number of cars at a particular intersection in each processors area of responsibility. For example StoN[0][0] will contain the number of cars traveling North from the South-West intersection of each processors region, StoN[1][0] denotes the number of cars in the next block to the East and so on. A second array (called WtoE) represents the flow in the West-East direction. The naming conventions for these arrays is shown in the lower part of Figure 2.

Updating the traffic flow in a sequential program

Now our basic problem is to update the traffic flows as time progresses. We will make further sweeping assumptions that all vehicles travel at the same speed and all blocks are the same length. Thus the update cycle merely consists of moving each element in the WtoE array one element to the "East" and each element in the StoN array one element to the "North". Schematically this code looks like the following, for the cars moving West-East, on a sequential computer.

```
/* Do cars moving W->E: SEQUENTIAL */

for(j=0; j<BLOCKS[North_South]; j++) {
    for(i=BLOCKS[East_West]-1; i>0; i--)
        WtoE[i][j] = WtoE[i-1][j];
    WtoE[0][j] = random();
}
```

where BLOCKS is an array containing the number of street blocks in the two directions.

Fictitious boundary conditions

Note that we have introduced another fictitious object - cars appear randomly "out of the West" at each step, and disappear forever off the "East" edge of our city. These conditions are the so-called "boundary conditions" which should be familiar to most scientists and engineers. The proper specification of these effects is crucial to the model being constructed.

Running the code on a parallel computer

Now let us attempt to run this code on a parallel computer. For a first attempt consider running the above piece of code in each processor. We make the substitution blocks for BLOCKS. As in the sequential case BLOCKS is an array saying how many street "blocks" there are in each direction of the city, ***in total***. The new array blocks will denote the number in each processor after partitioning the data across the nodes.

The parallel computer "boundary conditions"

The code which is correct for a sequential computer is wrong for a parallel machine because of the "boundary conditions" at the junctions between the processors. If we run the above code in multiple parallel processing nodes new cars are generated randomly along the West edge of **each** processor's sub-region while cars going off the Eastern edge disappear forever rather than appearing in the next area to the East. Not very realistic.

Interprocessor communication

The problem can be solved in parallel by adding some simple communication calls to the above example. Essentially what we have to do is to have each processor send to its Easterly neighbor the number of cars in each of its Eastmost blocks. This data is then read by the adjacent processor and used to fill in its data for the Westmost blocks. A suitably modified version is the following (Assume all arrays/variables are suitably declared elsewhere)

A common bug

```
/* Do cars moving W->E with boundaries:
 * FIRST EFFORT -- WRONG
 */

Enode = exgridnode(nodedata.procnum, East_West, 1);
Wnode = exgridnode(nodedata.procnum, East_West, -1);
exgridsize(nodedata.procnum, BLOCKS, blocks, starting);

type = TRAFFIC_FLOW;
for(j=0; j<blocks[East_West]; j++) {
    exread(&temp, sizeof(int), &Wnode, &type);
    exwrite(&WtoE[blocks[East_West]-1][j],
                    sizeof(int), &Enode, &type);
    for(i=blocks[East_West]-1; i>0; i--)
        WtoE[i][j] = WtoE[i-1][j];
    WtoE[0][j] = temp;
}
```

In this piece of code we assume that the `nodedata` structure is defined and setup elsewhere with a call to `exparam`. Notice that we use `exgridnode` to give us the magic "nodes" for the processors to our East and West, and `exgridsize` to actually tell us how many blocks lie in our processor. Further we use a temporary variable, `temp`, to store the information coming from our neighbors.

So where is the bug in the above code?

The problem is with the order of the `exread` and `exwrite` functions. Since we are using the blocking "read" function each processor will come to its `exread` and stop waiting for a suitable message to arrive. Since every processor is now waiting and none of them are writing data the machine is now "hung".

A trivial fix is to reverse the order of the `exread` and `exwrite` calls. Now each processor sends its boundary value and then looks for an incoming message with new data. This method works - we have parallelized our program!

Optimizing the traffic flow program

However, the best solution is not yet found. Three questions can be asked about the current solution

- Can I avoid having to remember to write before reading - especially when there may be real cases in which the other order is appropriate?
- Is this solution the most efficient?
- Is this method guaranteed - even when the messages being sent are very large?

The first question may seem trivial but is actually quite important. With the proper tools errors such as this are easy to find but they still require a fair amount of recoding and rethinking which is wasted effort. The second question is the central topic of parallel processing and obviously important. The last point is rather subtle and concerns the internal buffering which is happening inside the *Express* kernel.

Fortunately there is a solution which satisfies all three questions - `exchange`. The problem at hand required one processor to both send and receive data. Instead of having separate "read" and "write" operations the `exchange` function combines them both into a single function call. Conceptually the read and write operations are made simultaneously (which is really what we need in our example) and the implementation allows us to take advantage of hardware capabilities for maximum speed. Further we can take precautions internally to avoid possible buffering problems completely.

A better version of the above algorithm is, therefore

```
    /* Do cars moving W->E with boundaries:
     * CORRECT and BETTER
     */

        Enode = exgridnode(nodedata.procnum, East_West,1);
        Wnode = exgridnode(nodedata.procnum, East_West,-
1);
        exgridsize(nodedata.procnum,
                        BLOCKS, blocks, starting);

        for(j=0; j<blocks[East_West]; j++) {

            exchange(&temp, sizeof(int), &Wnode, &type,
                    &WtoE[blocks[East_West]-1][j],
                        sizeof(int), &Enode, &type);

            for(i=blocks[East_West]-1; i>0; i--)
                WtoE[i][j] = WtoE[i-1][j];
            WtoE[0][j] = temp;
        }
```

One point to note is that we have apparently lost touch with the "boundary conditions" that made up part of the specification of the original (sequential) problem. Cars are supposedly generated randomly on the West edge of the system and disappear off the Eastern edge. The first condition can be easily fixed with a call to `exgridcoord` which will tell us if we are on the Western edge of the city and hence need to generate random cars. The other part of the problem is a little trickier.

Periodic and non-periodic boundary conditions

By default `exgridnode` assumes that the user domain is "periodic". This means that the left and right hand edges are connected as are the top and bottom. The simple consequence of this fact is that when the Eastmost processors ask for a node to the East they are given the node number of the Westmost processor. This is actually very useful in typical scientific applications where the "periodic" assumption is often encountered but is obviously incorrect in our example. What we would like to happen is for no data to get sent off the Eastmost edge of the city and, likewise, no data to get read on the Westmost edge.

Running Express programs on sequential computers

Fortunately this is very easily achieved with the `exgridbc` call which overrides the defaults and makes `exgridnode` behave the way we want it to. When we ask for a processor number which is "off the edge of the city" `exgridnode` will return the magic value `NONODE` which, when passed to `exchange`, will denote that no communication should actually be attempted. This mechanism is very general - all the *Express* functions understand the `NONODE` argument and use it to indicate that no attempt should be made to communicate. We use this feature to run codes on sequential computers. In this case `exgridnode` will only ever return `NONODE` values since there is only one processor - there are no other nodes to send messages to.

With the addition of a suitable call to `exgridbc` the code becomes

```
/* Do cars moving W->E with CORRECT boundaries:
 * CORRECT and SMART
 */

int perbc[2];

perbc[North_South] = perbc[East_West] = 0;
exgridbc(perbc);/* Override periodic boundaries */

Enode = exgridnode(nodedata.procnum, East_West, 1);
Wnode = exgridnode(nodedata.procnum, East_West, -1);
exgridsize(nodedata.procnum, BLOCKS, blocks, starting);

exgridcoord(nodedata.procnum, recpnum);

for(j=0; j<blocks[East_West]; j++) {

    exchange(&temp, sizeof(int), &Wnode, &type,
     &WtoE[blocks[East_West]-1][j], sizeof(int),
     &Enode, &type);
```

```
          for(i=blocks[East_West]-1; i>0; i--)
           WtoE[i][j] = WtoE[i-1][j];

          if(recpnum[East_West] != 0) WtoE[0][j] = temp;
          else WtoE[0][j] = random();
     }
```

This piece of code now deals correctly with all the cases and is fully parallel. We can duplicate it trivially for the case of cars traveling N-S and everything is done. Note that the structure is still quite like the original program and the user had to have no knowledge of the underlying topology of the parallel machine - `exgrid` and *Express* did all the work.

Parallel programming with exgrid is just like sequential programming

Note that there are no strange looking parallel processing "incantations" of any kind in this code - it consists of a set of standard C statements and calls to a runtime library. In this sense it is still a sequential program and can be thought of, developed and debugged in that way. Everything that happens in the program is fully deterministic and totally under the control of the programmer. As a result it is easy to understand every factor while both designing and analyzing the algorithm. This is the reason that we advocate this programming style so strongly - one can use ones normal intuition about programming sequential computers to understand how this parallel program works!

Reducing Express overheads

One might at this point worry about the efficiency of the above approach. Inside the loop over `j` we are making a call to the communication system and hence the kernel with all the overhead that this entails. A much more efficient method would be to do all the communication in one swoop. We can do this by making the `temp` variable into an array and trivially modifying the program

```
     /* Do cars moving W->E with CORRECT boundaries:
      * CORRECT and SMARTEST
      */

     int perbc[2];

     perbc[North_South] = perbc[East_West] = 0;
     exgridbc(perbc);/* Override periodic boundaries*/

     Enode = exgridnode(nodedata.procnum, East_West, 1);
     Wnode = exgridnode(nodedata.procnum, East_West, -1);
     exgridsize(nodedata.procnum, BLOCKS, blocks, starting);

     exgridcoord(nodedata.procnum, recpnum);
     exchange(temp, 4*blocks[East_West], &Wnode, &type,
          &WtoE[blocks[East_West]-1][0],
          4*blocks[East_West], &Enode, &type);

     for(j=0; j<blocks[East_West]; j++) {
        for(i=blocks[East_West]-1; i>0; i--)
            WtoE[i][j] = WtoE[i-1][j];
        if(recpnum[East_West] != 0) WtoE[0][j] = temp[j];
        else WtoE[0][j] = random();
}
```

Communicating arrays: exvread, exvwrite, exvchange

Note that we've used an important property of the C language here which is that arrays are stored in the order which makes the last index increase fastest. So, when we use `exchange` with the data pointer set to `&WtoE[blocks[East_West]-1][0]` and with length `4*blocks[East_West]` we will actually transmit the data along the correct column of the array.

Having made this important optimization one might wonder what happens to the traffic in the South-North direction. If we make the same optimization there then the data that we want to send off the Northern edge doesn't lie in adjacent memory locations, at least in C. Even in languages where the South-North array works right the West-East one wouldn't - you can't have it both ways! This is the reason for the existence of the "v" routines: `exvread`, `exvwrite`, and `exvchange`. These routines, as well as the conventional pointer, length, node and type arguments have two extra: item size and skip distance. So, for instance, the actual syntax of the `exvwrite` call is

```
exvwrite(data, size, skip, nitems, &node, &type);
```

The `data`, `node` and `type` arguments are exactly as before. However the actual data sent consists of `nitems` of `size` bytes, each separated by `skip` bytes. Note that we do not specify the total length of the data to be sent in bytes as with `exwrite` but rather give the number of items and the size of each.

As an example of this call suppose that we wish to send every third element of a simple array of 32 bit integers. In total there will be 23 items to be sent and the array they come from is called `mybuffer`. Then the appropriate call to `exvwrite` is

```
exvwrite(mybuffer, 4, 12, 23, &node, &type);
```

The variations `exvread` and `exvchange` are implemented in a similar way. To see how one might use these calls in our traffic flow problem it is easiest to just present the code that deals with the South-North flow. Note that we only have to call `exgridbc` once to set up the boundary conditions properly for both West-East and South-North flow.

```
/* Do cars moving S->N with CORRECT boundaries:
 * CORRECT and SMARTEST
 */

Nnode = exgridnode(nodedata.procnum, North_South, 1);
Snode = exgridnode(nodedata.procnum, North_South, -1);
exgridsize(nodedata.procnum, BLOCKS, blocks, starting);

exgridcoord(nodedata.procnum, recpnum);
blks = blocks[North_South];
```

```
exvchange(temp, 4, 4*blks, blks, &Snode, &type,
    &StoN[0][blks-1], 4, 4*blks, blks, &Nnode, &type);

for(j=0; j<blks; j++) {
    for(i=blks-1; i>0; i--)
        StoN[j][i] = StoN[j][i-1];
    if(recpnum[North_South] != 0)
        StoN[j][0] = temp[j];
    else StoN[j][0] = random();
}
```

This code now has exactly the same form as that for the West-East flow but with a call to `exvchange` replacing the call to `exchange`. Obviously the call to `exchange` could actually be replaced with a suitably "hacked" call to `exvchange` making the codes look even more alike.

When you don't need to use message types

A final point to note is that the variable `type` shows up a lot but doesn't get much attention. Very early on we set its value to the macro `TRAFFIC_FLOW` and have since ignored it. This is typical of these "synchronous" simulations - the type parameter is superfluous since every node knows who to send data to and when - no extra level of classification is necessary.

Don't send messages with type DONTCARE

> WARNING: It is tempting to try to the use macro value `DONTCARE` for the message type in this type of problem since you really don't care! Unfortunately there is no way for *Express* to send a message with this type and attempting it will cause weird and mysterious problems

While trivial in principal this example has hopefully served to show how the `exgrid` system and its associated function calls can be used to generate codes that look extremely similar to their sequential counterparts and which require no knowledge on the part of the programmer of the underlying topology of the parallel computer. Obviously this system will not be appropriate in certain circumstances where algorithms have complicated constraints - for example it is not always possible to make do with the synchronized communication system used here. Even in cases less synchronous than that considered here the use of the `exgrid` mechanism is not precluded and is still a very powerful tool. Another point to note is that the problem and solution presented here form most of the code needed to model fluid mechanics via the "cellular automata" approach - it is not such a trivial model after all.

A real case of traffic flow - cellular automata

At present the `exgrid` system is designed for dealing with regular meshes and their many dimensional derivatives. We are interested in extending this model to other common types of data structure such as trees and would encourage users to make their requirements known to us.

5.3 Utility Functions and global communication

As well as providing the basic node to node communication facilities described in the previous sections *Express* offers other "utility" functions that are commonly used: `excombine`, `exbroadcast` and `exconcat`.

The word utility in the previous sentence is in quotes because it probably represents something of a misnomer in this case. While the functions described here do indeed perform functions that might be considered less obvious in parallel processing terms they themselves represent an extremely powerful method of writing complete parallel programs. Just as `exchange` and `exvchange` were the only two functions required to parallelize the traffic example in the previous section (with `exgrid` too, of course!), we have developed complete commercial parallel applications using only functions from this section.

The basic difference between these routines and `exchange/exvchange` is that whereas the latter were hooks into interprocessor communication facilitated by the `exgrid` system these routines perform "global" or "collective" actions on several processors. Again since no hardware specific details are required these routine are guaranteed to be portable across all types of parallel (and sequential) computers and yet they provide interfaces to a rich set of parallel processing primitives which still let us apply conventional sequential intuition to our programs.

Global communication functions carry out common operations automatically

The excombine function is used to apply a user specified function to data distributed among the processors of the machine. The basic calling sequence is

```
        excombine(data, function, size,
                nitems, Nnodes, Nlist, ptype);
```

where the combining function is applied to `nitems` data items, each of `size` bytes. The function provided must satisfy certain constraints in order to be effective; namely associativity and Commutativity. (Basically this means that the result of applying the function to items A and B is the same irrespective of the order of application. Addition and multiplication are good examples and subtraction is not.) The last arguments specify which set of nodes to apply the function to; `Nnodes` is the number of nodes listed in the array `Nlist`. If `Nnodes` takes the special value `ALLNODES` then the `excombine` will be done on all processors. Finally the `ptype` argument serves to assign a "type" to the combine function in the same manner as `exread` and `exwrite`.

Counting the cars - a global sum over all processors

As an example of the use of this function consider another extension to the traffic problem discussed previously. Assume that after each iteration we want to find out how many vehicles are left in the system. A simple way to achieve this in a sequential program is the following code

Counting the cars sequentially..

```
     /* Count total number of cars: SEQUENTIAL */

        total = 0;
        for(i=0; i<BLOCKS[East_West]; i++) {
        for(j=0; j<BLOCKS[North_South]; j++)
                total += WtoE[i][j] + StoN[i][j];
        }
```

To modify this code for a parallel processor we just add a call to excombine as follows

.. and in parallel

```
    /* Count total number of cars: PARALLEL */

     #include "express.h"      /* Defines ALLNODES */

         int add_function(), type=123;
         total = 0;
         exgridsize(nodedata.procnum, BLOCKS, blocks);

         for(i=0; i<blocks[East_West]; i++) {
             for(j=0; j<blocks[North_South]; j++)
                 total += WtoE[i][j] + StoN[i][j];
         }
         excombine(&total, add_function, sizeof(total), 1,
             ALLNODES, NULLPTR, &type);
```

where the add_function is defined elsewhere as

```
         int add_function(p1, p2, size)
         int *p1, *p2, size;
         {
             *p1 += *p2;
             return 0;
         }
```

This code is actually quite straightforward. For each of the nitems mentioned in the call to excombine the combining function is called with, as arguments, pointers to two items of the length given by the user. The combining function should then overwrite the first argument with the result of combining the two elements and return a zero value to its caller. (Returning other values cause excombine to fail and/or perform other tasks - see the *Express* reference text for details). The final result of the call to excombine is just what was wanted - the value total is now the total number of cars summed over all the processors in the system. Furthermore, every processor that participated in the call to excombine has this result.

An important extension of the abilities of excombine not shown here regards the nitems argument. In our case we wished to combine only a single value, the total number of cars in each node. In general, however, we may have a vector of values to combine in some manner. Rather than repeatedly calling excombine for each element we can, instead, increase the nitems argument and combine the entire vector in one call. This obviously leads to significantly reduced overheads.

Converting distributed data to global data

The other two "utility" functions, exconcat and exbroadcast serve similarly useful purposes.

exconcat takes data items from each processor and makes a single long buffer in each node by concatenating the individual contributions from each node. A feature of the way this is done guarantees that each node ends up with the same result. The `exbroadcast` function, as its name implies, performs a broadcast operation to a set of nodes in the system. Any node may be the originator of the broadcast and the message may be restricted to a subset of the processors. (See the *Express* reference for details).

Broadcasting a single processor's data to others

Notice that these functions also have "type" parameters. This is so that one can distinguish between several overlapping function calls. Without such a parameter, for example, the following course of actions would be illegal and probably cause the machine to "hang" since the `excombine` in node 1 will pick up the message sent with the `exwrite` in node 0.

```
Processor 0.
    exwrite to processor 1.
    excombine.

Processor 1.
    excombine.
    exread from processor 0.
```

Even if the machine doesn't "deadlock" the results will be gibberish. With the "type" field, however, it can be arranged that the "`excombine`" and "`exread/exwrite`" operations have different types. In this case *Express* will sort out which message goes with what and all will be well. Of course, if you give the same `type` argument to both then chaos will still result.

Avoiding "deadlock" by using the message types

Fortunately errors such as this are extremely easily detected using an interactive debugger such as *ParaSoft*'s `ndb`.

6 I/O and *Cubix*

Up to this point all the high level communication primitives described have been intended primarily for node-to-node communication within the parallel machine. No reference has been made to the host computer.

One of the major decisions which must be taken by an application developer is whether or not any of the code must run on the host processor or whether the entire application can run in the distributed machine. Several factors influence this decision;

Deciding whether to use the host processor

- Is there enough memory in the parallel machine for the application?
- Can I satisfy my I/O requirements entirely within the parallel machine?
- How difficult will it be to maintain a code which is divided into host and parallel parts, in particular since this configuration will almost certainly not run on a purely sequential computer?
- Does my application have real-time constraints or similar which require a tightly coupled interface to the host processor - perhaps at the device driver level?

The answers to these and other questions can only be decided on a per-application basis. Experience shows, however, that programs that run entirely within the parallel computer are **significantly** easier to write and maintain than those which are divided into inhomogeneous pieces. In particular we have found applications which, when written to operate in this mode, can be supported **WITHOUT CHANGES** upon a wide variety of **BOTH** sequential and parallel machines. Porting code to new machines is often merely a question of recompiling - no other changes need to be made.

Having advertised some reasons for using this model of computation its basis is in an *Express* subsystem known as *Cubix*. *Cubix* is a full-function I/O and operating system server that enables distributed applications full access to the operating system resources available on the host computer. Multiple host processors can be supported and also distributed "disk farms" for file access. The interface to the host operating system is sufficiently sophisticated that one of the functions you can perform is to start up and run a host program to which you can communicate in a natural way. This allows you to actually run a user written host program from within the *Cubix* programming model. Essentially you can have the best of both worlds - full file system access from within the nodes and all the advantages of a sequential host program.

Parallel file I/O under Cubix

At the lowest level file access is available through the standard UNIX functions `read`, `write`, `open`, `lseek` etc. At a higher level it involves a complete model of distributed I/O involving three totally distinct modes.

Synchronous mode:
: All processors make requests together and each receives the same response. This situation occurs a lot in interactions with the user - for example, issuing prompts and reading values for global variables. It allows us, for example, to input single data items and have *Express* broadcast the values to all nodes automatically.

Multiple mode:
: All processors make requests together and each receives a different response. This mode is used most often for reading and writing the bulk of data generated or required by a parallel code. It's feature is that it is possible to construct a solid model for the various I/O functions allowing deterministic and repeatable behavior.

Asynchronous mode:
: Any processor may make a request at any time and each is serviced independently. This mode is rather hard to control since asynchronous and unrepeatable behavior results but is central to certain applications and situations.

I/O modes

The coexistence of these three distinct I/O modes and the ability to switch between them makes *Cubix* an extremely versatile system. In principle any function that the host could perform is available to the node processors - including such things as spawning new processes on the host and controlling external devices. The details of programming in this style are to be found in the accompanying document "Programming Parallel Computers Without Programming Hosts" which is the major reference for this system. Meanwhile a couple of examples might serve to illustrate some of the functionality

One's first piece of code in C is supposed to be

The "Hello world" program

```
#include <stdio.h> /* Day 1: SEQUENTIAL */

main()
{
    printf("Hello world\n");
}
```

which generates the immortal line

```
Hello world
```

Actually writing the program that does this in the nodes is quite hard if a user written host program has to be used since messages must be coordinated between host and nodes. If, however, one uses the *Cubix* model then the above code, when executed with the command

```
cubix -n 1 noddy
```

would also generate the same output. (Note that we assumed that the program resulting from compiling the previous code fragment has been called noddy. Details of the compilation procedures are given in the introductory guide for your version of *Express*.) Even if run on more than a single processor the output would appear the same because by default all files appear in "single" mode in which only a single node actually generates any output. The trivial modification of the program to

"Hello world" in parallel. Multi mode I/O

```
#include <stdio.h> /* Day 1: PARALLEL. Multi mode */
#include "express.h"
struct nodenv nodedata

main()
{
    exparam(&nodedata);
    fmulti(stdout);
    printf("Hello world from processor %d\n",
                                nodedata.procnum);
    exit(0);
}
```

produces the output

```
Hello world from processor 0
Hello world from processor 1
Hello world from processor 2
Hello world from processor 3
```

when executed on four processors. The trivial addition of the `fmulti` call switches the I/O mode for `stdout` so that output appears in order of increasing processor number. Note that no other files are affected - the I/O mode is specified for each stream independently. Additional function calls are available to fully specify the order in which input or output are performed while a file is in multi mode. This facility is extremely useful in conjunction with the `exgrid` system allowing users to perform I/O specification on the basis of the application data domain rather than the underlying processor topology.

Asynchronous I/O - the advantages and disadvantages

Taking the previous program and switching the `fmulti` call to `fasync` enables the asynchronous mode. In this case the output from the various processors will appear in arbitrary order and may even change from one trial to the next.

This is one of the penalties of using the asynchronous I/O mode - especially with the buffered functions like `printf`, `scanf`, `fread` etc. Use of the lower level functions such as `open`, `close`, `read`, `write`, `lseek` is recommended when using the asynchronous mode - further details are presented in the companion document describing *Cubix.*

The *Cubix* model of parallel computation is extremely powerful and yet very straightforward. Many applications can either run intact or require very minor modifications to use this system and its use is strongly recommended in all cases that can take advantage of it.

7 Hardware Dependent Communication

Much of this chapter has been devoted to a discussion of the issues which make parallel programs portable and/or easy to develop. One significant point which has seemingly gone overlooked, however, is performance.

Optimizing performance instead of portability

The issues of parallel program performance is extremely complex. Not least of the problems is the fact that parallel processing hardware is developing quite quickly and so what are "good" techniques this year may be hopelessly old fashioned a couple of years hence. It is for this reason that we have concentrated so heavily on portability and standardization since this automatically leads to a situation where an application can take advantage of developing technology.

Hardware dependent communication is faster but un-standardized

There are many applications, however, which can make good use of current technology if only its performance could be improved a little. For this reason *Express* supports a layer of communication primitives that directly address the hardware present in the parallel computer in use. These routines are optimized for one thing only - speed. The interface that they present to the user is extremely simple and cannot be guaranteed to exist, or even function in the same manner from one machine to the next. The decision on whether or not to use these routines must lie with the developer and should be based on a reasonably detailed study of the issues at hand. The use of the profiling system to be described in a later chapter is strongly advised before embarking on a revision of the code to use these routines.

Communication between hardware connected "neighbors"

The basic idea embodied in this system is that of "nearest neighbor communication" - i.e., communication only between processors which are directly connected by the underlying hardware. To describe such a connection *Express* uses the terminology of a "channel". This is an integer quantity that describes the connections between one node and some set of other nodes. In a hypercube topology, for example, the concept of a communication channel is well defined by the bits that make up the processor numbers. Similarly a transputer machine has nodes that each have four "links" which again can be mapped onto the integers 0, 1, 2 and 3.

The Express interface

Given this mapping from the hardware description to a set of small positive integers *Express* provides four routines to implement communication: `exchanon`, `exchanoff`, `exchanrd` and `exchanwt`.

`exchanon` and `exchanoff` are the functions that control the use to which a particular channel is put. `exchanoff` disables the normal processing of a hardware channel by *Express* and sets it in a mode where the low level communication functions can operate. `exchanon` performs the complementary task, re-enabling *Express* on a previously disabled channel.

"Wild and dangerous" routines - no checking

Neither of these routines performs any checking on the validity of a particular operation. As such it is the responsibility of the user to ensure that no messages are still *in* the system which will need to be forwarded on a particular channel which is to be disabled. Typically this means that some sort of synchronization is required before disabling channels. Similarly it is the responsibility of the user to make sure that no regular *Express* communication is attempted which uses a channel which is still disabled.

Once a channel has been successfully disabled the `exchanrd` and `exchanwt` functions can be used to pass data along a channel. The calling sequences are extremely simple:

```
exchanrd(chan, buffer, nbytes)

exchanwt(chan, buffer, nbytes)
```

Mapping Express channels to the hardware concepts

As can be seen no `node` or `type` arguments are present in these lists - the functionality is merely to transmit `nbytes` bytes of data from the indicated `buffer` into or out of the named channel.The actual node with which these routines communicate depends solely on the hardware interpretation of the `chan` parameter. For the two simplest types of hardware this association is as follows

Hypercube	Channel `chan` connects the processor whose node number is `nd` with that whose processor number is given by `nd ^ (1 << chan)` i.e., by switching bit "`chan`" of the processor number.
Transputers	Channel 0 corresponds to hardware link 0 - i.e., `Link0In` in a call to `exchanrd` or `Link0Out` in a call to `exchanwt`.

Blocking communication - in the strongest sense

These routines are completely "blocking" in the strong sense that the call to `exchanwt` in one node will not return until a corresponding call to `exchanrd` has been made in the receiving node. Furthermore the message lengths, indicated by the respective `nbytes` arguments must match exactly.

Note that this means that the following sequence of calls, which would be valid if made with `exread` and `exwrite` will lead to "deadlock" when made with the lower level functions.

A "deadlock" situation

(We assume that nodes A and B are connected on channel 0.)

```
/*                              /*
 * Node A                        * Node B
 */                              */
exchanwt(0, buf, 12);           exchanwt(0, buf, 12);
exchanrd(0, buf, 12);           exchanrd(0, buf, 12);
```

Avoiding "deadlock" by assigning parities

The problem in this case is that both nodes call exchanwt together. As a result both wait for a call to exchanrd to consume their data. As a result neither can proceed and the outcome is "deadlock". To alleviate this problem one commonly introduces the concept of "parity" in which nodes on opposite ends of a communication channel are assigned opposite values. The code above could then be re-written as follows:

```
/*
 * Nodes A and B
 */
extern int parity;      /* Initialized elsewhere */

    if(parity == 0) {
        exchanwt(0, buf, 12);
        exchanrd(0, buf, 12);
    }
    else {
        exchanrd(0, buf, 12);
        exchanwt(0, buf, 12);
    }
```

Note, however, that the assignment of parity to processors is not necessarily trivial. For a hypercube connected machine one can always assign parity to the processors based on the number of "bits" set in their processor numbers. For a more general interconnection strategy such as is possible with a transputer system, however, it may be impossible to perform this assignment. Consider, for example, the simple net shown in Figure 3.

If we assign parity 0 to node 0 then node 1 will need to have parity 1. But since node 2 is connected to both nodes 0 and 1 it cannot have either parity assignment. Programming with the low level channel communication primitives is still possible in such a case but extreme care must be taken to ensure that the calls to exchanrd and exchanwt match correctly.

Successful applications of a "nearest neighbor" programming model

Having gone, at great length, into the difficulties present in using these routines it should be stated that they can significantly improve the performance of a great many parallel processing algorithms. A good reference for the types of problem which can be successfully tackled in this way is the book "Solving Problems on Concurrent Processors" by G.C.Fox *et al.* published by Prentice-Hall (1988). This book deals with a broad range of scientific problems solved by the research group at Caltech using a communication system based exclusively on nearest neighbor interactions.

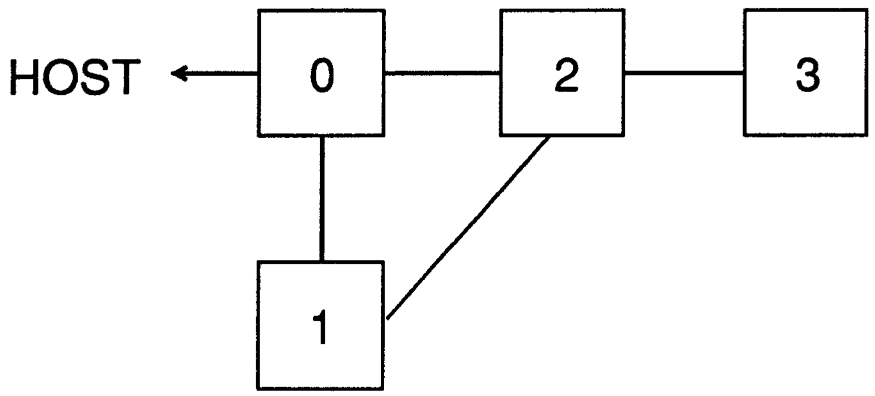


Figure 3. Processor interconnect with no possible parity assignment

An important feature of *Express* in connection with these routines is that can be gradually merged into a working code when performance analysis indicates that some gain can be made. We strongly advocate the use of the profiling systems to be described in a later chapter before embarking on a major code revision which takes advantage of these routines - it is important to understand just how much one stands to gain from such labor. Furthermore the step to using these calls should not be taken too early in the development of a parallel project since their use inhibits the functioning of such powerful development tools as the debugger and performance monitor - while a channel is disabled it cannot be used to forward system messages such as those used by the debugging tools.

Before proceeding, make sure that it will be worth while

Several strategies are available to lessen this barrier

Minimizing the pain associated with the fast routines

- Construct a working program, fully benchmarked and debugged and then turn on the fast communication. Hope that nothing bad happens.
- Since most applications go through cycles in which the faster performance of the `exchan` routines is sometimes unnecessary one can restrict ones debugging attention to the phases where *Express* has been re-enabled.
- Use a communication strategy in which *Express* can be left enabled on a set of channels that "spans" the hardware topology. In this case the debugging tools can be used at all times.

The first method is basically sound but suffers from the defect that certain program bugs may only manifest when the faster communication is used since this affects the relative timing of different activities on different nodes. One cannot be sure, therefore, that enabling the faster communication will not engender new bugs of its own.

The middle option is reasonably workable. Most applications, particularly those in scientific and technical domains have periodic structures in which *Express* can be alternately off and on. With care one can then use the development tools during the times that *Express* is enabled.

Using cnftool to build "hybrid" networks

The last strategy is probably the easiest to use but requires some preparation in setting up and also more connectivity from the underlying hardware than may be available. The basic idea is to reserve a set of channels for use by the fast communication routines and then use the full *Express* system on the others. The simplest way to achieve this is to simply delete the corresponding channels using the system configuration tool, `cnftool`. (This is only available if the system in use supports such reconfiguration.) Links which do not appear in the network description generated by `cnftool` will., by default, have *Express* disabled just as though a call to `exchanoff` had been made at the beginning of the program. These links can then be used for fast communication.

Note that it is not necessarily sufficient to merely disable certain channels at the beginning of the user application.since *Express* may have already decided to use these channels for its own message routing. Deleting channels with `cnftool` avoids this problem since the internal routing is built upon the information supplied by `cnftool`.

8 Complete Example Programs

This section contains complete examples of the use of *Express* in both *Cubix* and Host-Node applications.

8.1 The "RING" Program

A one-dimensional

The program is shown schematically in Figure 4.

exgrid makes the decomposition straightforward

The `exgrid` tools are used to set up a one-dimensional processor decomposition - i.e. a ring and then we use `exgridnode` to obtain the processor numbers to be used in communicating with the next and previous node around the ring. Each processor then sends data to it successor using the `fwdnode` and reads from its predecessor using `bcknode` as shown in the figure. Repeating this operation as many times as there are nodes in the ring has the result of sending each processor's message to every other node.

At the end of each cycle we use the `excombine` function to gather up some data either to be printed (in the *Cubix* version) or sent to the host. This latter operation has nothing to do with sending messages around a ring but is added to give a little more variety to the program.

A "model" program

While this program looks just as silly as some of the other examples used it is actually quite a common programming model - many parallel applications take the form where an inner loop performs calculations and internode communication which ends with some data being gathered together for later analysis or display.

The two programming models

Two distinct versions of this program are presented: a *Cubix* version and another split into host and node programs. We present the latter last since it is more complex and will serve to advertise the *Cubix* model. It may be surprising to readers that the *Cubix* implementation is actually a parallel program - it looks just like a sequential program.

8.1.1 *Cubix* Program

This program would be compiled with the *Cubix* libraries by specifying some machine dependent compiler options - see the introductory guide to your version of *Express* for more details. Once compiled we can execute it on four processors with a command similar to

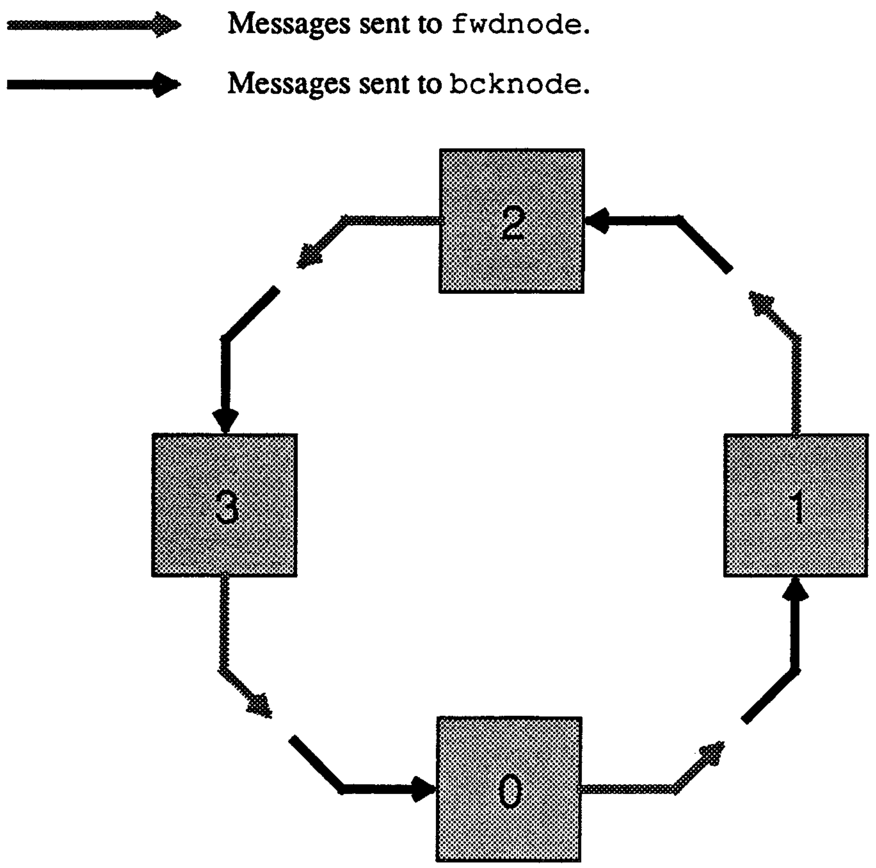


Figure 4. Communication around a ring of processors

```
    cubix -n 4 cubix_demo
```

where we have assumed that the program has been called “cubix_demo”.

```
/*******************************************************
 * EXPRESS Demonstration program.                      *
 * -------                                             *
 * Parasoft Corporation, 1988. CUBIX program           *
 *                                                     *
 *******************************************************/

#include <stdio.h>
#include "express.h"

short indata[256], outdata[256];

main(argc, argv)
int argc;
```

```
char *argv[];
{
    int chksum[2], check, numtimes, k, type = 123;
    int nprocs[1], nshift, fwdnode, bcknode, f_add();
    struct nodenv env;

/* Get system parameters and construct a checksum. */

    exparam(&env);          /* Get system parameters */
    check = 0;
    for(k=0 ; k<env.nproc ; k++) check += k;

/* Now set up the channels to use in the ring. Map a
 * one dimensional chain of processors onto the nodes.
 */
    nprocs[0] = env.nprocs;
    exgridinit(1, nprocs);

    fwdnode = exgridnode(env.procnum, 0, 1);
                                /* Forward route */
    bcknode = exgridnode(env.procnum, 0, -1);
                                /* Reverse path */

/* Now prompt for the number of times to pass the
 * message around a ring
 */

    printf("How many times around the ring ? : ");
    scanf("%d",&numtimes);

/* Now let's go !!!!! Send a 512 byte message around
 * the processor ring. For each successful round trip
 * put out a '#' character and issue diagnostics if
 * something seems wrong.
 */
    for(k=0 ; k<numtimes ; k++){

/* Shift data around the ring we just set up --
 * note that we have to do "nprocs" shifts to get it
 * round the ring once.
 */
        for(nshift=0; nshift < env.nproc; nshift++)
            exchange(indata, 512, &bcknode, &type,
                         outdata, 512, &fwdnode, &type);

/* Now do the checksum business -- with excombine */
```

```
            chksum[0] = 1;
            chksum[1] = env.procnum;
            excombine(chksum, f_add, sizeof(int), 2,
                                 ALLNODES, (int *)0, &type);

            if(chksum[0] != env.nproc ||
               chksum[1] != check) {
                fprintf(stderr,
                    "\nError in node communication\n");
                fprintf(stderr,
                    "\tExpecting %d, %d\n",
                    env.nproc, check);
                fprintf(stderr,
                    "\tReceived %d, %d\n",
                    chksum[0],chksum[1]);
                exit(0);
            }
            else {
                putchar('#');
                if(((k+1) % 64) == 0) putchar('\n');
                fflush(stdout);
            }
        }

        printf("\nDone\n");
        exit(0);
    }

    f_add(i,j, size)
    short *i, *j;
    int size;
    {
        *i += *j;
        return 1;
    }
```

8.1.2 Host-Node Program, "Host" code.

Dividing the "RING" program into two pieces for "host-node" execution

This version of the "RING" program is in two pieces, one which would execute on the host processor of your machine and another for the nodes of the attached parallel computer. The former is presented first and raises several important issues which are noted at the end of the program text.

To run this code one would compile it with some C compiler for the host processor in use but with the addition of the *Express* library. More information about this process can be obtained from the introductory guide to *Express* on your system.

To run the program on four nodes we would execute a command similar to

```
host 4
```

where we have assumed that the program resulting from the compilation of this code has been called "`host`". If you have named it something else, or your machine requires a different syntax to execute programs then the above command line will have to be modified accordingly.

Debugging "host-node" programs

Note that make provision in this code for debugging by the specification of *any* second argument. This forces the host program to execute a call to the `expause` routine which loads the node program at a breakpoint so that the debugger can be invoked. To take advantage of this feature one might use the command line

```
host 4 dummy_for_debugging
```

with suitable modification for your own operating system.

```
/***************************************************
 * EXPRESS Demonstration program.                  *
 * -------                                         *
 * ParaSoft Corporation, 1988. HOST program.       *
 ***************************************************/

 #include <stdio.h>
 #include "express.h"

 main(argc, argv)
 int argc;
 char *argv[];
 {
     long fromnodes[2], numtimes;
                /* For communicating with nodes */
     int check, k;
     int pgind, nodes;
     int src, type = 123;
     struct nodenv env;

 /* Read number of nodes from command line, or set
  * default. If we give a second argument then stop the
  * node program at its entry point so that we can use
  * the debugger
  */
     nodes = (argc > 1) ? atoi(argv[1]) : 4;
     if(pgind=exopen("/dev/ncube", nodes,
            DONTCARE) < 0) exit(1);
     if(argc > 2) expause();
     exload(pgind, "node");
```

```
/* Get system parameters and construct a checksum to
 * compare with the values returned from the cube.
 */
    exparam(&env);/* Get system parameters */
    check = 0;
    for(k=0 ; k<env.nprocs ; k++) check += k;

/* Now prompt for the number of times to pass the
 * message around a ring
 */
    printf("How many times around the ring ? : ");
    fflush(stdout);
    scanf("%ld",&numtimes);

/* Send the count to the nodes and then read back a
 * message for each cycle. Note that this is rather
 * tricky on machines with reversed byte orders. We
 * have to swap the bytes, send them to the nodes, and
 * then swap them back again to use on the host.
 */
#ifdef SWAP
    _ex_swaw(&numtimes, &numtimes, 4);
#endif
    exbroadcast(&numtimes, HOST, 4, ALLNODES,
                                    (int *)0, &type);
#ifdef SWAP
    _ex_swaw(&numtimes, &numtimes, 4);
#endif

    for(k=0;k<numtimes;k++){
        src = 0;
        exread(fromnodes, 8, &src, &type);
#ifdef SWAP
        _ex_swaw(fromnodes, fromnodes, 8);
#endif
        if(fromnodes[0] != env.nprocs ||
           fromnodes[1] != check) {
            fprintf(stderr,
                "\nError in node communication\n");
            fprintf(stderr,"\tExpecting %d, %d\n",
                env.nprocs, check);
            fprintf(stderr,"\tReceived %d, %d\n",
                fromnodes[0],fromnodes[1]);
            exit(0);
        }
        else {
```

```
                    putchar('#');
                    if(((k+1) % 64) == 0) putchar('\n');
                    fflush(stdout);
               }
          }

          printf("\nDone\n");
          exit(0);
     }
```

Problem 1: Data types not compatible between host and node processors

NOTE 1. Note that we use the "long" data type for all parameters which must be communicated between host and node programs and that the `excombine` function has the size parameter explicitly set to 4 for the size of a `long`. While ugly this is the only way to make this program portable between machines with potentially different word sizes - the more natural `int` type is often of length 16 bits on the host and 32 bits on the nodes making it unsuitable for our purpose. Compare the *Cubix* version which has no equivalent problems.

Problem 2: Data format not compatible between host and node processors

NOTE 2. We use a preprocessor directive `SWAP` to control byte swapping of data which must be communicated between host and nodes. This is a particularly nasty feature of the "host-node" programming model. Occasionally systems are built in which the data format used by the CPU on the "host" processor is different from that of the nodes. In these cases one or the other processor must take care to transform the data to a format that the other can use whenever communication between them must occur. In this example we elect the "host" for this task and control the behavior with a preprocessor macro. This latter allows us to use the program on either type of machine and select at compile-time the appropriate option.

8.1.3 Host-Node Program, "Node" code.

Building "node" programs from sequential code

The following code makes up the "node" half of the "Host-Node" version of the "RING" program. The most interesting things to note is its similarity to the entire *Cubix* version of the code. This is often the case - in practice one obtains node programs by copying the entire sequential version of the code and deleting the initial and final I/O relating to parameter input and result output. In many cases where structured programming practices have been adopted the node program can often be made up quite simply by calling the main processing routines of the original sequential program.

The compilation of this code is quite straightforward but one must be careful NOT to use the switches that invoke the *Cubix* libraries. While this would apparently succeed, at the compile/link stage, the resulting program would not run properly because it would contain *Cubix*-specific code which requires that the host be executing the `cubix` program rather than the one we showed in the previous section.

The name of the node program

A final important issue involves the naming of this program. While any name can, in principle, be chosen it must match that used in the call to `exload` in the host program. In this case we should name the resulting program: node.

```
/*****************************************************
 * EXPRESS Demonstration program.                    *
 * -------                                           *
 * ParaSoft Corporation, 1988. NODE program          *
 *****************************************************/

#include "express.h"

struct nodenv env;

short indata[256], outdata[256];
long tohost[2], numtimes;

main()
{
    int k, nshift;
    int fwdnode, bcknode;
    int type = 123, dest;
    int nprocs[1], f_add();

/* Read system parameters, number of nodes etc...... */

    exparam(&env);

/* Now set up the channels to use in the ring. Map a
 * one dimensional chain of processors onto the nodes
 * that we have.
 */
    nprocs[0] = env.nprocs;
    exgridinit(1, nprocs);

    fwdnode = exgridnode(env.procnum, 0, +1);
                                  /* Forward route */
    bcknode = exgridnode(env.procnum, 0, -1);
                                  /* Reverse path */

/* Now read the number of iterations from the host -
 * note that the number of forwarding operations is
 * this parameters times the length of the ring.
 */
    exbroadcast(&numtimes, HOST, 4, ALLNODES,
                                  (int *)0, &type);

    for(k=0;k<numtimes;k++){
```

```
/* Shift data around the ring we just set up */

        for(nshift=0; nshift < env.nprocs; nshift++)
            exchange(indata, 512, &bcknode, &type,
                        outdata, 512, &fwdnode, &type);

/* Now send a silly message to the host. Add up a bunch
 * of ones and also our processor numbers.
 */
        tohost[0] = 1;
        tohost[1] = env.procnum;
        excombine(tohost, f_add, 4, 2,
                    ALLNODES, (int *)0, &type);
        if(env.procnum == 0) {
            dest = HOST;
            exwrite(tohost, 8, &dest, &type);
        }
    }
}

f_add(i,j, size)
long *i, *j;
int size;
{
    *i += *j;
    return 1;
}
```

8.2 Other Examples

All versions of *Express* are shipped with an extensive set of example programs. The exact location of these files depends on the particular system but most often they can be found in a subdirectory called "`examples`" of the main *Express* installation. This directory will itself contain several other subdirectories, each exhibiting a particular feature of the system. Of particular interest are the `express` and `cubix` directories which contain further examples of the two programming models and the elementary use of the basic *Express* functions.

Cubix

Programming parallel computers without programming hosts

1 Introduction

Historically, application programs for parallel computers consist of two parts, a master process running on the host and a server running in the parallel machine

Which processor is in charge?

Cubix adopts a different viewpoint. Once a program is loaded into the nodes, that program assumes control of the machine. The host process only serves requests for operating system services. Since it is no more than a file server, the host program is universal; it is unchanged from one application to the next.

This programming model has some important advantages.

- Program development is easier because it is not necessary to write a separate program for the host.
- Parallel programs are easier to develop and debug because they can use standard I/O routines, rather than machine dependent system calls.
- Parallel programs can often be run on sequential machines with minimal modification.
- The programming model extends in a natural way to distributed I/O, systems such as disk farms, attached directly to the parallel machine.

I/O modes, disk farms

The currently implemented versions of *Cubix* supports both synchronous and asynchronous I/O modes for maximum flexibility and portability of the resulting parallel codes. In addition certain versions of the system incorporate support for distributed disk systems and multiple host computers.

Cubix was created to make programming parallel computers easier. Its goal is to eliminate significant duplication of effort on the part of programmers, and to make the environment in the parallel machine appear much more familiar to application programmers. It is also intended to make programs more easily portable to sequential machines as well as between different brands of parallel computers.

The "hello world" problem

The motivation for *Cubix* can probably best be understood by sitting down with one's favorite distributed machine and trying to get each of the nodes to perform a trivial task involving input and output to the terminal. For example, have each processor identify itself, and multiply its processor number by a number entered on the console, printing an informative message like:

```
I am processor 17 and 3 times 17 is 51
```

in response to the number 3 being entered. This is an extraordinarily difficult exercise because the nodes of the parallel machine do not have direct access to the operating system facilities available on the host. One can not, for instance, execute a `scanf` in the nodes to obtain data from the console. Instead, the host must allocate a buffer, read data from the console into it, pass the contents of the buffer to the nodes, read a message for each node containing the results of that node's calculation, format those messages and print the results. Programming this exercise requires two programs, one for the host and one for the nodes of the machine; often compiled with different compilers and different compiler options. One must also worry about the sizes of the various data types on the host and in the nodes and, in extreme cases, the byte ordering within the types.

Important issues - debugging and maintenance

This example is obviously frivolous, but it illustrates an important shortcoming in parallel programming environments. Maintaining and debugging "real" programs is unnecessarily difficult for exactly the same reason as in the exercise: it is too hard to use the host's operating system. Debugging is extremely difficult because programs cannot be easily modified to produce output tracing the flow of control. Additionally, when a program is modified, it often requires separate but coordinated changes to both the node program and the host program. The necessary coordination is a rich source of minor bugs.

A further deficiency in parallel computing environments is the duplication of effort involved in this programming style. Each programmer is forced to re-invent a host-node protocol which resembles, functionally at least, the protocols that have been written hundreds, if not thousands, of times already. After writing a few protocols, each programmer tends to develop a characteristic signature. Programmers quickly learn to reuse their '`main`' routines, but by then, their time has already been wasted.

Keeping a sequential version

Finally, after expending the effort to develop a parallel application, the programmer finds that the program will not run on a sequential machine. The I/O protocol designed for the host-node link is completely foreign to the sequential machine. Even though the bulk of the application would operate correctly by linking with a very simple library of dummy communication routines, the host program and node program must be "glued" back together. Maintaining an evolving code intended to run on both sequential and parallel machines is quite difficult for this reason. (Note that the program, once glued, no longer runs in parallel!).

All these deficiencies can be traced to a single source. Parallel computers are often viewed as high-speed peripherals attached to a host computer which controls their operation. As peripherals go, they are extremely flexible and programmable, but control, nevertheless, resides in the host. The host loads programs and data into the nodes which then compute and eventually return results which are expected, in number and length, by the host. In more sophisticated applications, the nodes analyze various tokens passed by the host and may perform different computations based on their values.

2 A Different Perspective

The Cubix programming model

The basic idea behind *Cubix* is that the program running in parallel should control the operation of the associated program running on the host. This is exactly opposite to the common style of programming discussed above. In *Cubix*, tokens are passed from the nodes to the host requesting activities like opening and closing files, reading the time-of-day clock, reading and writing the file system, etc. The host program does nothing more than read requests, act on them and return appropriate responses. All such requests are generated by subroutine calls in the parallel processor. The host program which serves the requests is universal; it is unchanged from one application to the next, and the programmer need not be concerned with its internal operation.

It is convenient to give the node subroutines the same names and calling conventions as the system calls they generate on the host. This relieves the programmer of the task of learning a new lexicon of system calls. Any operation he would have performed in a host program can be encoded in a syntactically identical way in the cube. It is of no consequence that the subroutine called in the cube might actually translate the request into Swahili before sending it to the host. All the programmer sees is a call to, e.g. `write(fd, ptr, cnt)`.

High level utilities are often written in terms of a set of standard system calls. Since the *Cubix* system calls have the usual names and calling sequences, system utilities designed for the sequential host computer can be readily ported to the parallel machine. For example, the C Standard I/O Library can be compiled and linked with *Cubix* allowing various forms of formatted and unformatted buffered I/O. (See the introductory section which describes the *Express* subroutine library, to be found in the accompanying reference manual) Under *Cubix*, the exercise of Section 1 would be programmed as:

```
#include <stdio.h>
#include <express.h>
                /* Defines nodenv structure */

main()
{
     int entry, pnum;
     struct nodenv nodedata;

     exparam(&nodedata);
     pnum = nodedata.procnum;
     scanf("%d", &entry);
     fmulti(stdout);       /* See section 3. */
     printf("I am proc %d, and %d times %d is %d\n",
            pnum, entry, pnum, pnum*entry);
     exit(0);
}
```

3 The Catch - I/O modes

It is highly optimistic to think that a set of system calls designed for a sequential computer can be sufficient for use in a parallel environment without modifications or additions. In fact, the requirements of the parallel environment do force one to restrict the use of some routines and also to add a few additional ones. The issue to be addressed is:

> How does one resolve the problem that different processors may need to do different things - maybe at different times?

To address this question we will classify applications into two types.

Two types of program: synchronous and asynchronous

Synchronous programs are characterized by uniformity from processor to processor and structured communication and calculation stages. In particular each node computes for a while and then all processors communicate data among themselves before engaging in another round of computation. The significant point in this model is that the interprocessor communication channels are essentially free while computation is being done.

The second category might be termed asynchronous. They are characterized by having completely individual behavior in each processor and no regular communication-calculation cycles. In these applications there is no way of knowing when all communication channels will be idle other than by explicitly synchronizing the processors.

These two styles are supported in different ways by both the *Express* communication facilities and the *Cubix* I/O model. The differences are explained in the next sections. Note that the distinction may not be as clear cut as stated above. In particular asynchronous programs often have internal points of synchronization and may well proceed in this manner for lengthy periods of their operation. Similarly, synchronous programs may occasionally benefit from the ability to use asynchronous function calls - a good example is the processing of a run time error. Often these appear in data-dependent ways that mean that an error condition in one processor might not be duplicated in all the others. Then it is of benefit to the ailing processor to be able to take its own corrective or diagnostic action independent of the other processors.

3.1 Synchronous I/O Modes

single and multi modes

Since a large majority of applications in science and engineering fall into this category we will discuss the synchronous I/O mode first. The sample code of the previous section is a good example of this style. If we had run it on four processors the output would look like

```
I am processor 0, and 3 times 0 is 0
I am processor 1, and 3 times 1 is 3
I am processor 2, and 3 times 2 is 6
I am processor 3, and 3 times 3 is 9
```

in response to the number "3" being input. There are several points to notice in regard to this example, simple as it is. Only a single value was entered at the console yet all processors received the value 3 as input. On the other hand only a single `printf` call was made but four lines of output resulted. This is an example of the difference between the "single" and "multi" modes of *Cubix.*

In single mode a single function call has the same effect in every processor whereas in multi mode a single function call has a unique effect in every processor. The call that makes all the difference in the example is `fmulti(stdout);` which switches the standard output stream over from single to multi mode. Thereafter the call to `printf` produces a unique output string from each processor. To make this even more obvious consider the following simple example

```
#include <stdio.h>
#include <express.h>/* Defines nodenv structure */

main()
{
     struct nodenv nodedata;

     exparam(&nodedata);
     printf("Hello world\n");
     fmulti(stdout);
```

```
        printf("I am processor %d\n",
                        nodedata.procnum);

        fsingl(stdout);

        printf(" .... and that's that !!\n");
        exit(0);
    }
```

If this were run on four nodes then the output would be

```
Hello world
I am processor 0
I am processor 1
I am processor 2
I am processor 3
.... and that's that !!
```

In this example we start off in single mode (The default for all I/O streams) and utter the immortal phrase "Hello world" which appears once. We then switch over to multi mode and print out a unique string from each processor. Finally we switch back to single mode and print out another string that only appears once.

singl and multi mode input

The singular and multiple modes are not restricted to output operations. Consider, for example, the next code fragment, where we assume that a variable nproc has been set to the number of processors we have allocated.

```
        printf("Please enter a number: ");
        fflush(stdout);
        scanf("%d", &n);

        printf("Please enter %d numbers: ", nproc);
        fflush(stdout);
        fmulti(stdin);
        scanf("%d", &i);

        fmulti(stdout);
        printf("You gave %d and %d to proc %d\n",
                                n, i, procnum);
```

When run on eight processors with input

```
123
8 7 6 5 4 3 2 1
```

this will produce the output

```
You gave 123 and 8 to proc 0
You gave 123 and 7 to proc 1
You gave 123 and 6 to proc 2
You gave 123 and 5 to proc 3
You gave 123 and 4 to proc 4
You gave 123 and 3 to proc 5
You gave 123 and 2 to proc 6
You gave 123 and 1 to proc 7
```

Again the important point to notice is that while the stream `stdin` was in single mode a single value typed at the console is sufficient to satisfy the call to scanf in all eight processors while eight values must be input to satisfy a similar request when stdin has been switched to multi-mode.

Changing the order of multi mode I/O

Also note that we can freely mix single and multi modes whenever convenient. The former is obviously useful whenever entering values for global variables that are constant in each processor while the latter allows for independent data in each node. Additionally we can alter the order of the output from, or input to, a multi mode file through the `forder` system call. By default all I/O is ordered by increasing processor number (as should be readily apparent). The following code segment reverses this order for output;

```
#include "express.h"/* Defines nodenv structure */

{
      struct nodenv nodedata;

      exparam(&nodedata);/* Get runtime parameters */
      forder(stdout,nodedata.nproc-nodedata.procnum-1);

      fmulti(stdout);
      printf("Hello, this is processor %d\n",
                                  nodedata.procnum);
      fflush(stdout);
      exit(0);
}
```

Run on four processors this produces the output

```
Hello, this is processor 3
Hello, this is processor 2
Hello, this is processor 1
Hello, this is processor 0
```

The interaction with exgrid

This option is particularly useful in conjunction with the `exgrid` utilities. It is a simple matter to reorder I/O so that processors read data blocks in an order determined by the decomposition of the physical data rather than some arbitrary ordering according to the underlying topology of the machine. As an example consider the following code fragment;

```
#include <stdio.h>
#include "express.h"

{
     struct nodenv nodedata;
     int my_val, nprocs[1], recpnum[1];

     exparam(&nodedata);/* Get runtime parameters */

/* Assign processors to a ring topology */

     nprocs[0] = nodedata.nproc;
     exgridinit(1, nprocs);

/* Now reorder the input stream to correspond to
 * the processor location around the ring
 */
     exgridcoord(nodedata.procnum, recpnum);
     forder(stdin, recpnum[0]);

/* Now read in input parameters */

     fmulti(stdin);
     scanf("%d", &my_value);
```

The processors are assigned to a ring topology by the `exgridinit` call - i.e., they are logically assigned to a one dimensional chain. Then the `exgridcoord` routine is used to discover which slot in this decomposition is occupied by a processor and this value is used to re-order the input stream `stdin`. This has the end result that run on four processors and presented with the input

```
100 101 102 103
```

the first value, 100, would be read by the processor first in the logical chain, 101 by the second processor, 102 by the third and 103 by the fourth completely independent of the underlying topology of the parallel computer. Such independence of the hardware configuration is the key element in making programs portable between parallel computers.

Image processing - two dimensional problems

Another example of the use of this technology is image processing - using `exgrid` routines and the `forder` function it is possible to arrange to read the image data according to rows and columns of the image. In more complex cases the `mread2d` and `mwrite2d` functions are available to read an arbitrary portion of a two-dimensional data set into individual nodes.

Finally note that the mode and ordering of files are totally independent - it is quite possible to have `stdin` configured to read data in multi mode ordered according to a three dimensional model of some structure while `stdout` remains in single mode to issue prompts to the user.

Applying common sense principles to parallel I/O

At this point the power and simplicity of the *Cubix* I/O picture should be apparent. In the single and multi modes we have a system that actually makes sense - when the same value is required in each node you only have to enter it once while different output can easily be obtained from individual nodes. With the exception of the `fmulti` and `fsingl` function calls everything looks just as it would in a sequential program. *Cubix* is more than just a file serving protocol, however. As well as allowing file I/O functions such as `printf` and `scanf` all other facilities usually available on the host are available to the programmer. Files may be opened and closed (`fopen`, `fclose`, `open`, `close`), processes may be executed on the host (`system`, `popen`), times may be accessed (`ctime`, `ftime`) and so on. The node program can, to all intents and purposes, behave as though it were running on the host computer.

Buffering modes and flushing files

Having extolled the virtues and simplicity of the *Cubix* model one should understand some of the detailed ways that `fmulti` extends the sequential computer I/O model. Consider the following code fragment.

```
fmulti(stdout);
printf("hello\n");
fflush(stdout);
printf("good-bye\n");
fflush(stdout);
printf("CUBIX ");
printf("is flexible \n");
fflush(stdout);
```

If executed on a parallel machine this would produce the following output

```
hello
hello
 ...
hello
good-bye
good-bye
 ...
good-bye
CUBIX is flexible
CUBIX is flexible
 ...
CUBIX is flexible
```

The important point to notice is contained in the last block of output. Notice that the line `CUBIX is flexible` appeared intact from each processor despite having been written in two `printf` statements. This is, in fact, an often overlooked feature of any sequential I/O library - output is "buffered". Instead of each character appearing on your terminal individually the system saves up some number and then spits them out at once - this improves efficiency. The actual flushing of the data to the terminal can also be controlled by the user via the `fflush` system call.

In the light of this discussion we can examine the previous example more carefully. Note that the calls to `fflush` are each responsible for certain lines in the output. Furthermore there is no such call between the two `printf` calls that make up the last line. What happened in this last case is that the two calls merely stored up characters in an internal buffer. After two calls had been made the buffer on each node contained the string `CUBIX is flexible` which then appeared all at once when the `fflush` call was made.

fflush

At this point the user may well be somewhat confused by the ***buffering*** that seems to be going on all over the place and what they can and cannot expect to happen. Fortunately this is rarely a problem given the fundamental rule:

> Multi mode files never flush automatically. The only way to get at the data in such a file is to call `fflush` explicitly.

If this rule is followed then everything will be as expected. In single mode output appears on the terminal under well-defined conditions; whenever a carriage return in seen, whenever the internal system buffer gets full, the user calls `fflush` or when input is requested on any stream. In multi mode nothing ever appears until `fflush` is called.

Line-buffering, the default for terminal devices

One of the more common errors using *Cubix* is the failure to flush buffers when files are in `multiple-mode`. We list, in Section 7, some of the other common errors.

The previous discussion is actually a piece of a rather larger picture. The concept of buffering is actually rather sophisticated and has many options and variants. Those that are relevant to *Cubix* users are discussed in section 3.3. For regular use, however, the previous simplified discussion is more than adequate.

At this point one has to address the "synchronous" nature of these calls. So far the examples have been characterized by one thing - whenever one node made a system call the others did so too. Admittedly, given the simple nature of our examples, it is actually quite tricky to do otherwise. However, this is a fundamental requirement of the single and multi I/O modes.

The central concept in this discussion is that of "loosely synchronous" behavior. The adjective "loosely" is applied here because no real program is ever completely synchronous since this would have to imply that EVERY processor was executing the same instruction as all the others ALL the time. This situation arises rarely - even in SIMD machines!

"Loose synchronization"

"Loose synchronization" is the concept behind the alternation of compute and communicate cycles discussed earlier in this section. Essentially an action is loosely synchronous if it occurs when all communication channels are known to be free. This need not actually be restricted to the so-called compute phases - in between two communication calls will also satisfy the constraint as long as all processors make the call together. To (hopefully) clarify this picture a little consider the following example for two processors

```
Processor 0.
     Send message to processor 1.
     Call printf    /* NOT loosely synchronous */
```

```
          Receive message from processor 1.

     Processor 1.

          Call printf.   /* NOT loosely synchronous */
          Receive message from processor 0.
          Send message to processor 0.
```

In this example the call to `printf` is not loosely synchronous because the communication channel between processors 0 and 1 is blocked by the message that node 0 has sent but node 1 has not read. If we modify the actions to the following

```
     Processor 0.

          Send message to processor 1.
          Call printf.   /* Loosely synchronous */
          Receive message from processor 1.

     Processor 1.

          Receive message from processor 0.
          Call printf.   /* Loosely synchronous */
          Send message to processor 0.
```

then the call to `printf` is loosely synchronous. Note that we have assumed that the system only contains these two processors. If there are actually eight in the system then the behavior of the others is also important - the concept here is a global one in that all processors must satisfy the conditions before an action can be said to be loosely synchronous.

Having defined and (hopefully) explained what the term means we now make the statement that "synchronous mode" *Cubix* requires that all system calls be made loosely synchronously. This requirement is, in fact, an overstatement of the true facts which is what makes the multi mode so useful. The requirement of "loose synchronicity" is actually only required for system calls that interact with the host computer. Obviously commands that merely buffer up data on a node do not interact with the host and so do not have the requirement. A good example is `printf` in multi-mode. Since no flushing is ever done until explicitly requested by the user, calls to `printf` may be made completely asynchronously to multi-mode files. Only the eventual call to `fflush` must be loosely synchronous.

The details and restrictions on the various system calls interact in a fairly complex manner with the "mode" of the associated stream. A list of which functions may be in what modes may be found in the introduction to the *Express* subroutine library to be found in the accompanying reference manual.

Unbuffered I/O

All the discussion so far has revolved around the buffered I/O functions; `printf`, `scanf`, fopen, fflush, etc. An interface also exists to the low level read, write, open, lseek style of system call. In these cases, however, the structure is slightly different. Instead of having different modes settable with a call similar to fmulti and fsingl there are distinct functions for implementing the two I/O types. Whereas read and write deal with single mode files mread and mwrite deal with multi- mode files. Also of interest are the mread2d and mwrite2d functions which deal with multi-mode files in a manner especially tailored to those users with problems in two-dimensions such as image analysis, partial differential equations, navigation, etc. A detailed discussion of these functions, with examples, can be found in the accompanying reference manual.

3.2 Asynchronous Mode

Catching run-time errors - an easy error to make

Occasionally circumstances arise in otherwise synchronous programs that require asynchronous behavior. A particularly good example which was mentioned earlier is error detection and recovery. How many C programmers, for example, have had malloc fail on them and wished that at the very least some diagnostic message had been available. Unfortunately the regular *Cubix* code segment that one might naively use is wrong

```
/* Asynchronous problem - incorrect version */

if((ptr=malloc(big_buffer_size)) == (char *)0) {
     fprintf(stderr,
         "We have big problems ... no more memory!!\n");
     exit(666);
}
```

because there is no guarantee that the error will occur in all nodes at the same time. A fix along the lines of

```
/* Asynchronous problem - partial fix */

fmulti(stderr);
if((ptr=malloc(big_buffer_size)) == (char *)0) {
     fprintf(stderr,
         "We have big problems ... no more memory!!\n");
     fflush(stderr);
     exit(666);
}
fflush(stderr);
```

is partially correct. Now you see an error message from any node that got the error. However multi mode still has the loose synchronicity requirement for the fflush operation so that the above piece of code won't work unless all processors are actually going to be doing this together. While this may often be the case one can easily foresee

cases where only some of the nodes are even in this piece of code. Then there is no chance of the fflush being successful and even worse the machine will hang. An error that was caused by insufficient memory has now caused the program to "hang" and may be misdiagnosed as a communication problem.

Asynchronous I/O

The solution to this problem is an asynchronous I/O mode. Code that can be guaranteed to work is as follows

```
/* Asynchronous problem - working code */

fasync(stderr);
if((ptr=malloc(big_buffer_size)) == (char *)0) {
     fprintf(stderr,
         "We have big problems ... no more memory!!\n");
     fflush(stderr);
     abort(666);
}
```

The call to `fasync` is the key. This switches on the asynchronous mode for the stream `stderr` and allows any processor to individually make system calls and requests.

Why not always use async mode?

Having introduced this concept one might wonder why not make it the default in all cases - indeed why bother having the synchronous modes at all? Several important reasons can be distinguished

- Asynchronous output introduces a randomness to the behavior of a program. Different runs will produce different output making it hard to reproduce bugs.
- Asynchronous input is very hard to maintain. Which data goes to which processor is very hard to control since the requests to "read" data arrive in some random order.

One approach commonly taken for the second point is to introduce a windowing environment and allocate one window for each processor. You can now do more sensible input by typing in each window. At least this ensures that you really can direct data to the processor you wanted to get it. The downside of this scenario is that you have to continually move from one window to the next - this is particularly bad if you really wanted to give the same value to all the processors, or if there are 512 processors - the windows will be awfully small. A scheme like this also has problems with concepts like redirecting standard I/O and pipes.

In the light of these problems it makes sense to use the synchronous I/O modes described in the previous section to perform most I/O functions. Some applications, however, are just asynchronous by nature and for these *Cubix* does provide an asynchronous mode.

singl, multi and async modes operate independently

Note that this mode is orthogonal to the `single` and `multi` modes described in the previous section - you can't have asynchronous multi mode, for example.

It is important to note that the "`async`" mode is useful for actually doing ascynchronous I/O *but* we still have the restriction that the call to `fopen` that opened the file must be made synchronously - i.e., all nodes must open the same file with the same access mode at the same time. In really asynchronous situations this can be a problem. Consider the following code, for example,

Opening different files in different nodes

```
/*
 * Attempt to open files, asynchronously
 * WRONG!
 */
#include <stdio.h>

open_wild(procnum)
     int procnum;
{
     char name[32];
     sprintf(name, "/tmp/junk%d", procnum);
     fp = fopen(name, "w");     /* ABORTS HERE!! */
     fasync(fp);
         ...
```

This code aborts at the call to `fopen` because the "loosely synchronous" constraint has been violated unless all values of "`procnum`" are the same. To get around this restriction merely add the letter "A" to the access mode string passed to "`fopen`" - in this case the code can be made to function by using the call

```
fp = fopen(name, "wA");
```

Note that this call opens the file but does *not* automatically enable "`async`" mode. A subsequent call to `fasync` must still be made as shown in the previous example.

In addition to the "standard I/O" interface, *Cubix* also provides an asynchronous version of `open` which allows a node to open a file independent of all other nodes. Files opened in asynchronous mode can be addressed at will with no synchronization constraints. All operations such as `lseek`, `read`, `write` and `close` will be applied asynchronously. Further it is possible to switch a file that was opened in synchronous mode over to asynchronous mode.

Low-level asynchronous I/O

Opening a new file in this mode is achieved by adding the `O_CBXASYNC` flag to those usually supplied. For example

```
/*
 * Open in asynchronous mode
 */
#include <fcntl.h>/* Defines O_ flags */

foo()
{
     int mode;

     mode = O_WRONLY | O_CREAT | O_CBXASYNC;
     if((fd = open("freddy.dat", mode, 0666)) < 0) {
```

```
                abort(12);
            }
    }
```

opens (and optionally creates) a file in asynchronous mode. Further references to the file descriptor fd will occur totally asynchronously on a first-come first-served basis. Nevertheless, the open call must be made synchronously in all nodes. If you intend to use the file in only one processor, then the aopen system call may be used instead of open. Unlike open, aopen may itself be called asynchronously, and returns a file descriptor that is ready for asynchronous I/O. Beware of calling aopen from more than a handful of nodes. Each processor obtains a completely distinct file descriptor and it is very easy to reach the host operating system's limitations on the maximum number of open files.

Beware the limitation on the number of open files

Often it is useful to switch a file from synchronous to asynchronous mode, and vice versa. The function fcntl, may be used for this purpose. In general, fcntl serves to modify the status of a file descriptor which is just what we want to do. It's use is typically a two stage process - we first get the flags currently in use for that file and then add the flag for asynchronous mode. Similarly we can restore a file to its normal behavior by taking away the asynchronous flag. A schematic of how this is done is given in the following code fragment

```
    /*
     * Switch stdout to asynchronous mode: fcntl
     */
    #include <fcntl.h>  /* Needed to define O_ flags */

    foo()
    {    int fd = 1;            /* refers to stdout */
         flags = fcntl(fd, F_GETFL);
                     /* Read current file settings */
         flags |= O_CBXASYNC;
                     /* Add in the asynchronous "bit" */
         fcntl(fd, F_SETFL, flags);/* Set new flags */
    }
```

To disable the asynchronous feature one would change the middle line of the above code to

```
        flags &= ~O_CBXASYNC;
```

which merely zeros the "bit" corresponding to asynchronous behavior.

Having acquired asynchronous access to files in this way the user is pretty much free. The functions read, write and lseek work together to maintain, on each processor, a record of that processor's location in the file. Each request that is sent to the host contains with it information that repositions the file correctly before the appropriate operation, i.e., reading or writing.

Beware that each processor begins pointing at the beginning of the file. If different processors simply begin writing into the file, they will write over one another's data. Use the system call `lseek` to avoid this situation.

The other system calls (`dup`, `fstat`, `ioctl`, etc.) are also performed asynchronously. A file must be closed the same way it was opened. Thus, if `aopen` was used to open the file, then `close` should be called asynchronously, and if `open` was used, then `close` should be called synchronously.

Asynchronous mode I/O is at best a rather hazardous exercise. Apart from any other considerations it may introduce non-repeatability into your code. A program may be running "correctly" in asynchronous mode and produce different looking output given identical input. Despite these difficulties the asynchronous mode does provide useful functionality to parallel programmers if used carefully.

**Asynchronous system calls*

This section has dealt exclusively with asynchronous I/O. *Cubix*, however is more than just a file server - it is a complete interface to the operating system of the host computer. In order to complete the set of synchronous/asynchronous modes a further function `syncmode` is provided. Be default all system calls to the host are made loosely synchronously: if you execute the `system` command then all nodes must do so together and the command is only executed once. Giving a zero-argument to the `syncmode` command, however, enables system calls to be made totally asynchronously. The following code, for example, would print out the date five times if executed on four processors.

Making everything asynchronous with syncmode

```
    syncmode(1);     /* Ensure we start synchronously */
    system("date"); /* Single date command on host */

    syncmode(0);     /* Enter asynchronous mode */
    system("date"); /* One date command for each node */
```

Note that ordinary I/O operations are also affected by this switch so that files which were originally opened in synchronous mode can be addressed asynchronously after this call. There is, however, a significant difference between the two modes.

The difference between asynchronous mode for a file and asynchronous mode for Cubix

A file opened for asynchronous operation is read repeatedly on each node - i.e., each node's first `read` or `fscanf` from the file results in reading the first data.

A file opened for synchronous operations, but read with `syncmode` set to zero allows "first come, first served" access - the first node to make its request will get the first data from the file, the second will get the second etc.

While this may occasionally be just what you want it tends to introduce a time dependency into your code which makes it hard to reproduce behavior and hence find bugs. In most cases we have observed it to be better to use the "real" asynchronous mode and judicious calls to `fseek` which introduce no timing dependencies.

3.3 Buffering Modes

When dealing with functions such as `printf` and `fopen` in sequential environments the user is being carefully shielded from the internal workings of the operating system. Most of the time this will not matter but occasionally most programmers find obscure cases where output doesn't quite look right and which they eventually cure with a well placed call to `fflush`. The *Cubix* implementation of the standard I/O library has similar features.

Typically users will not be aware of the problems and all should work as normal. In cases where this is not the case, however, the following discussion should act as a guide for when and where to put the `fflush` calls and even when to use the mysterious `setvbuf` system call.

ANSI standard file buffering modes

In standard I/O implementations such as the proposed ANSI standard there are three basic buffering modes for `FILE *` objects such as `stdout` and `stdin`. These are briefly summarized as follows (where the strange looking object is the macro from `stdio.h` that defines that particular mode)

`_IOFBF` Fully buffered I/O. Characters are stored up in the system buffers until full when an automatic call to `fflush` is generated to send the characters to the output device.

`_IOLBF` Line buffered I/O. Characters are again stored in the system buffers, but the buffers are now flushed automatically when a newline character is written and when input is requested from any other stream. This is the default for streams such as `stdout` which are connected to an interactive device, and is the reason that prompts which are immediately followed by an input request do not have to be flushed explicitly.

`_IONBF` No buffering. Each character is sent to the output device as it is generated.

Cubix additions to the proposed standard

To these three modes *Cubix* adds two others:

`_IOCBF` Circular buffering. When the buffer becomes full no data are written to the output device but extra data overwrites that at the front of the buffer. `fflush` calls are completely ignored. To flush data from a circular buffer you must switch its mode to, for example, `_IOFBF` and then call `fflush`.

`_IOEBF` Extensible buffering. When the buffer becomes full a new, larger, one is obtained by the system and the current data placed in it together with any additional bytes. This process can, in principle, continue until the entire memory is full. `fflush` must be called explicitly by the user.

Default buffering attributes

By default singular mode streams are either line buffered or fully buffered, according to whether they are connected to an interactive device, as in the ANSI standard, with the exception that `stderr` is always line buffered. When switched over to `multi` mode, streams default to the extensible mode and the user MUST call `fflush` in order to obtain data. This is a common source of error in *Cubix* programs. Asynchronous streams are buffered the same way as singular streams.

Buffering modes "stick" to the I/O mode of a file

Note that the buffering mode is a property associated with the "multiplicity" of a file and which therefore changes whenever that file's mode changes. If you want to reset the characteristics of `stderr` for use in `multi` mode then you have to call the `setvbuf` function WHILE the stream is in `multi` mode. The detailed use of the `setvbuf` command is documented in the reference manual.

Having described the way the system deals with the internal buffering mechanism explains certain features of the *Cubix* modes. In particular `multi` mode files have, as default, the `_IOEBF` mode which has "infinite" buffers and never generates automatic calls to `fflush`. This is the reason that `fprintf` is allowed asynchronously for a file in `multi` mode - the data just keeps accumulating in the buffer on the node making the call. Only the eventual call to `fflush` has to be synchronous.

3.4 Multiple hosts, Distributed filesystems, etc.

Distributed filesystems

A particularly common feature of advanced workstations is the support for distributed filesystems. In order to take advantage of such a system *Cubix* may be configured for multiple hosts with their own attached file systems. By default all system calls, including requests to open files are sent to the *Cubix* console - normally the terminal from which you submitted the `cubix` job. You can, however, specify alternate destinations in two ways.

Redirecting Cubix system calls

The system call `console_node` is provided to override the default choice for system calls. This call may be made at any time and does not have to be the same in all nodes. In particular, one might use the `syncmode` function described earlier in conjunction with `console_node` to partition a system into sets of nodes which each, independently, interacts with its own "host" finding files and making system requests to that particular target. For more details consult the manuals "Using *Express* on systems with multiple hosts".

An alternate, and somewhat simpler, system is implemented just using the naming convention for files. A request for the file `8001:fred.dat` will be sent to the host node with the name "`H1`" in `cnftool`'s naming convention. All further references to this file will also be sent to this node. Note that the exact mechanism required to indicate a special "host" when looking for files is operating system dependent.

4 Debugging: A Last Resort

Debugging Cubix programs

Debugging is a problem dear to every programmer's heart. One of the major successes of *Cubix* is that it makes debugging on a parallel processor almost as straightforward as on a conventional machine which lacks a source level debugger. The standard method of "print it and see" is quite applicable now that we have made the I/O system transparent. Of course you could also use the debugger `ndb` designed explicitly for debugging parallel applications and described elsewhere for this sort of job. An advantage of the debugger is that it still works in some cases where the "print" method fails - namely when there is a communication problem that blocks some channel and hence the intended output. Of course some people just like to use debuggers just as some people hate them. The choice is yours.

Overwriting critical memory regions

There is one category of bug, however, that neither of the above methods can catch and that is what would normally be classified as a "memory fault" on a conventional machine - the code attempts to scribble on some piece of memory that doesn't really belong to it. This is especially easy to do in languages like C where pointers are everywhere - especially the

`NULL` pointer! Unfortunately the consequence of doing this sort of thing on current parallel computers is that you wipe out crucial kernel data leaving the machine completely dead in the water. At this point no I/O is going to occur at all so one is wasting one's time with `printf` or its companions.

RAM files

For this reason *Cubix* has an extra file type: RAM files. These are accessed in the same way as all other standard streams (`stdin`, `stdout` and `stderr`) and files but nothing is ever flushed when you `fprintf`. In fact, RAM files are, by default, in circular mode, so that data is being accumulated in an internal buffer at a specific memory location. Since buffer is circular, in the sense of section 3.3, it has a finite length and just keeps wrapping round - if you write more data to it than its length then new data just starts overwriting old. In this way the actual amount of memory dedicated to this stream is constant. This type of construct is often called a RAM file -essentially it has the same characteristics as a file - you can read it, write it, seek on it, even close it if you wish, but the data, rather than living on a disk just sits in memory somewhere.

To open a RAM file one uses the `ramfopen` system call

```
FILE *ramfp;

ramfp = ramfopen(address, length);
```

where the `address` argument is the memory location at which the file should start and `length` is the number of bytes to use. In normal circumstances one places the RAM files in absolute memory locations by using some knowledge of the particular hardware on which *Express* is running although an array would do just as well.

To use the new file one just makes calls like

```
fprintf(ramfp,"Oh boy, we blew it!, crucial=%d\n",
        crucial);
```

which will deposit the value of the variable `crucial` in the RAM file, along with a helpful message. This file is recognized by the various standard I/O routines and never requires flushing. The data just stream in.

RAM files require no communication

The trick to the RAM file, and the reason that it's major use is in debugging when the kernel crashes is that the data in the file can be retrieved, even after the machine has been reset by a call to `exinit`, with the `exdump` utility. Also, since no communication is involved in writing data to this stream it can be used totally asynchronously and will continue to function even after all communication channels have ceased to function.

Debugging asynchronous programs

This system also has a significant advantage when debugging totally asynchronous codes. These latter are characterized by an annoying habit of showing unrepeatable behavior - what shows up as a bug in one place might appear somewhere else entirely on the next run. In particular one is often unable to use conventional debuggers since they impose synchronicity on the program by their very nature - even a good typist can't type in debugger commands at the speed of the underlying program. Often one can't use the usual "print" style of debugging alluded to earlier because the usual printing style on parallel machines involves communication which, in turn, involves other processors due to message routing. It is often the case that the insertion of a single `printf` call can drastically change the appearance of a bug by interfering with the timing relationship between two processors. The RAM file alleviates this problem somewhat - writing to it involves no communication so no other processors are affected. The only effect is to slow down somewhat the processor doing the writing. Even this effect is under the control of the programmer - for instance `putc` runs a whole lot faster than `fprintf` and thus might be a better bet.

exdump

The use of the `exdump` utility is straightforward. If the program you want to dump data from is still running then you can say

```
        exdump -b 0x80069000 -p pid
```

where the process ID of the process has been specified in the '`-p`' switch and the starting address used in the `ramfopen` call has been given in the '`-b`' switch. (The number shown here is NOT necessarily the one you should use on your machine - consult machine specific manuals for information about the memory layout of executing programs.) The second case is where the machine is completely hung and nothing seems to work anymore. In this case one must use the sequence

```
        exinit -m XXXXXX
        exdump -n nodes -b 0x80069000
```

Preserving RAM files even after calling exinit

where we have specified the number of nodes from which we wish to display the RAM data and the string "XXXXXX" should be replaced by a physical address in node memory which will not clash with the data in the RAM file being retrieved. (See the Reference manual page describing `exinit` for details.) Note that the data in the RAM file can be preserved even across invocations of the reboot procedure, `exinit`, if we use the '-m' switch. This

can be invaluable. In passing one might note that many other options are available for use with the `exdump` command to specify which nodes to read data from, whether to read ASCII or binary data, where in memory to look etc. These are discussed in the attached reference manual.

The problems with RAM files

Having given out the good news, however, there are some idiosyncrasies to be aware of when using RAM files. Basically, the file consumes memory and hence can engender new bugs of its own. A common situation is to position the RAM file in high memory leaving a predetermined space above for the program stack. However, since the program stack grows downward, and the RAM file grows upward, this is not completely robust and it is possible for the two to collide with dire and unpredictable results. This problem can be avoided with some care. It is possible to arrange for the RAM file to live in low memory. This has the attractive feature that if the stack grows down too far it will now be detected by the hardware and you'll get a more sensible diagnostic. On the other hand you have to tell the linker, dumper and run-time environment exactly where the RAM file should be located. Telling the compiler where to go is often as simple as creating an array for the RAM file to work into and using its name and size in the call to `ramfopen`. Unfortunately we now have to find the address of this object so that we can pass it to `exdump` when we wish to recover the data. This can usually be achieved either from "map" files produced by the various compiler/linker combinations or through the judicious use of the debugger, `ndb`.

5 Executing *Cubix*

Having expounded at some length about the plentiful virtues of *Cubix* it remains only to explain how one goes about using it. This section contains only the most elementary usage of the system and more details are given in the accompanying reference manual.

The first task is to compile the parallel program that will be executed. The procedure for doing this is shown in the introductory guide to using *Express* on your system. For the present we will assume that a program called `noddy` has been successfully compiled and linked and is ready to be run.

Executing the cubix command - passing arguments to the nodes

The next step is to execute the program. This is done by running the `cubix` utility, which typically has a form like

```
        cubix -n 4 noddy test.dat 12 3.45
```

This executes the previously compiled program (noddy) on four processors and passes it the arguments test.dat, 12 and 3.45 as character strings in the usual argc, argv format. That is, `argv[0]` is the program name, "noddy", `argv[1]` is "test.dat" and so on. In addition, the shell environment is also passed on to the program so you can say `getenv("TERM");` to find out about your terminal.

The `cubix` utility has several other options to control its behavior including options to run different programs in different processors, pass different arguments to different processors and load programs in a stopped state suitable for use with the debugger. These options are described more fully in the reference manuals.

6 Examples

Cubix is supplied with four example programs in the `examples/cubix` subdirectory of the main installation.

The first, `hello`, is the code discussed in Sections 2 and 3. It is a simple demonstration of the primary I/O modes and is comparable in complexity to the conventional "hello world" program.

The second, `ring`, is a *Cubix* version of the *Express* example program. If you aren't already convinced that *Cubix* is simpler than the conventional programming model compare these examples for clarity, simplicity and ease of maintenance.

`tstwild`, is a rather contrived program. It goes overboard in exercising the features of the system but might prove instructive.

Finally, `async` introduces the asynchronous I/O mode and its interaction with the other two I/O modes. A random number of random numbers is printed out in random order!

Included in this directory is a "`README`" file which contains instructions for executing these programs and a `makefile` suitable for compiling and linking the programs.

Several less contrived examples can be found in the `example` directory for *Plotix*. As well as showing off the graphical capabilities of *Plotix*, they also a demonstrate how simple

Cubix programs really are - after looking at the source code for these examples, the reader might need reminding that they are really parallel programs - they look so much like conventional C-code.

7 Common Errors

In this short section we list a few of the more common errors encountered in using *Cubix*. No doubt this list is not exhaustive and users are encouraged to complain about their individual mishaps.

abort(-1)

The most irritating occurrence when running *Cubix* programs is the "`abort, status -1`" which occurs with annoying regularity. This basically means that you have either placed a call to

```
        abort(-1);
```

in your program, or you have violated the "loosely synchronous" constraint. Since the latter is by far the most common this section describes some of the most common problems.

Abusing the "loosely synchronous" constraint

Most simple errors are connected with the abuse of the concept of loose synchronicity. Probably the most common error is to attempt to print out different strings while in singular mode. For example

```
        printf("Hello there, this is processor %d\n",
                     procnum);
```

is an error in singular mode because the strings to be output are not identical.

A similar error is to try to detect catastrophes in programs with

```
    if(error_condition) fprintf(stderr,"Death !!");
```

which fails unless it can be guaranteed that the error occurs in all nodes if it occurs in any. A somewhat longer but better way to do the above example is

```
    fmulti(stderr);
    if(error_condition)
        fprintf(stderr,"Death in node %d\n", procnum);
    fflush(stderr);
```

which uses multiple mode. (Obviously asynchronous mode could also be used). Note the explicit call to fflush which is necessary in multiple mode. This is actually another common error - forgetting to flush multi- files. A somewhat obscure flavor of this is the failure to call exit whenever the program is to stop. Without this files may not be closed properly or flushed and data might appear to be getting lost.

Another pitfall associated with the code we've just seen is to try to write it as follows

```
    fmulti(stderr);}
    if(error_condition) {
          fprintf(stderr,"Death in node %d\n", procnum);
          fflush(stderr);
     }
```

which may look more natural but has the above mentioned error: the call to `fflush` is not made "loosely synchronously".

Printing the addresses of data items in singl mode

A final *very* obscure flavor of this type of error occurs when printing the addresses of local variables in a C program. (Not many people do this, but those that do may appreciate the warning!). Code similar to the following can sometimes cause problems.

```
     mysub(arg1, arg2)
     int arg1, arg2;
     {
          printf("arg1 is at 0x%08lx\n", (long)&arg1);
     /*
      * Normal processing ......
      */
```

The problem here is that this code will run on the vast majority of parallel computers whose nodes are identical. On some machines, however, a few of the nodes may have special properties such as more memory, for instance. In this case the addresses of local variables may be different in different nodes because the user program has been loaded into a different piece of memory and so the above code violates the "loosely synchronous" rule too.

8 Conclusions

A version of *Cubix* has been running at Caltech since early 1986. Since its introduction, *Cubix* has become quite popular, and the system has been implemented on a range of parallel processors. The prevailing attitude among users is that use of *Cubix* is vastly

Cubix is very simple to use

simpler than the old host-node protocols (even among persons not in the author's immediate family). Many programs have been written for which the same code can be compiled and run on a sequential machine, as well as a parallel machine running *Cubix/ Express.*

The bad news about Cubix programs

Cubix's most significant drawback seems to be the increased code size in node programs. All computation that would have been done on the host is now done in the nodes. Although it is not any slower to perform inherently sequential tasks simultaneously in many processors, a copy of the code must reside in each processor. It is important to realize that both Standard I/O routines like `printf`, which usually does not appear in non-*Cubix* programs, and application dependent sequential code which would have appeared in the host program must now be included in the code that runs in every node. The size of this code can be significant, and reduces the amount of space available for data. The code and data linked by a call to `printf`, for example, requires about 6 -10Kbytes on each node in our implementation.

While *Cubix* offers the developer access to a wide variety of operating system functions it cannot be all things to all men. In particular it has a fairly strong bent towards UNIX supporting most of the elementary operating system calls. On the other hand as software techniques evolve and diversify *Cubix* is unlikely to be able to support them all. A good example is provided by the sophisticated windowing systems in use on modern systems. Not only is each individual system huge, there are as many different "standards" as there are implementations - far too many for *Cubix* to support. As a result we believe that there comes a point in the development of a major software project when *Cubix* will be unable to fulfill all of the software needs of the application.

Deficiencies in the Cubix model

One possibility is to use *Cubix* as an I/O server and fire up alternate host processes with calls such as `popen`. These host processes can then communicate with the node process through a standard pipe mechanism. We have not found much need for this type of interface although it is available. (This facility is only available on systems that support multiple host processes. It is not supported under DOS, for example.)

Merging Cubix programs with host programs

An alternative strategy which uses the facilities of *Express* directly rather than UNIX pipes is to have a second host process "share" access to the nodes allocated by the *Cubix* program with the `exshare` system call. This method has the advantage that it imposes no structure on the communication mechanism between the user interface in the second host program and the nodes of the parallel computer - they are free to communicate at will through the standard *Express* runtime library.

A third possibility is to "link" user functions directly into the *Cubix* server process. This can be done with some care but represents a rather inflexible solution which has little room for real growth. It can, however, serve the explicit demands of some custom applications.

Adopting the viewpoint that the program running in the nodes of the parallel machine should control the behavior of the host has some extremely desirable consequences.

Putting the nodes in control of the parallel computer

- It is possible to write a universal host program which accepts commands generated by subroutine calls in the nodes.
- Given a universal host program, programmers only write one program (the one for the nodes) for any application, eliminating considerable labor and an annoying source of bugs.
- All details of the host-node interface are hidden from the application programmer. Operating system services are obtained by system calls identical to those used on the host.
- Since applications require only one program to operate in parallel, it is usually a simple matter to run them on a sequential machine as a special case or to port them to new parallel machines which support the *Cubix* I/O model.
- Since operating system interaction is, for the most part, the same as in sequential programs, there is considerably less to learn before one can begin writing significant hypercube programs.

Multitasking

Executing multiple processes on individual processors

1 Introduction

Conventional computer programs, like the conventional computers on which they execute, are sequential. This means that they execute a single line of code at a time in a predefined sequence. This programming style is very straightforward since everything is predictable beforehand. Even the bugs are predictable since the same thing happens every time - with sufficient willpower we could trace the execution of every line of code on a piece of paper in order to find our problems. The successes of this programming method are quite obvious - nearly all scientific calculations are done this way and most "canned" applications are like this.

The need for multitasking systems

The most obvious example of a task which cannot be performed in this way is the operating system of a large multi-user machine. The operating system has to be able to cope with (more or less) random requests for services by its users and be able to satisfy one of these while simultaneously processing many other programs. This is a classic example of a multi-tasking environment. The users and the operating system together make up a "pool" of tasks each of which makes as much progress as it can on its own and then requests and optionally waits for services provided by other tasks. Systems like this are significantly harder to debug since their behavior is extremely time-dependent. What may crash the machine one time may be completely benign the next due to some user in a completely disconnected part of the system doing something slightly different.

The structure of parallel programs

When we consider parallel processing similar distinctions can be drawn. Most scientific and indeed many other algorithms have a regular structure based on the data being manipulated. This often maps in a very straightforward manner to the nodes of the parallel machine. Furthermore the structure of the algorithm is also regular with processors regularly synchronizing by exchanging results necessary for further computation. This programming model has been called "loosely synchronous" the processors proceed in an ordered fashion. Each processor is free to execute its own code on its own data but the overall picture is one of alternate periods of calculation and communication.

"Loosely synchronous" programs

This style of programming is quite straightforward under *Express*. Tools are provided to automatically compute the optimal distribution of data between nodes and also to facilitate the communication necessary at each stage. Furthermore bugs are reasonably easy to find since they occur repeatably - every time a program is run in this mode the same problem should arise. Armed with a sophisticated debugger like *ParaSoft*'s ndb programming in this mode can be almost as simple as programming a sequential computer.

Totally asynchronous programs

There exist, however, applications which do not fit into this "loosely synchronous" mode. Several reasons may be advanced for this. Some have poor load balance when decomposed in this way - it may be difficult to arrange for all the processors to work equally hard. If this happens in the synchronous model the whole machine slows down to the speed of the slowest node due to the synchronization points. A classic case of this is when the workload cannot be evaluated ahead of time. One application we have seen in which this is true is computer chess. The basic problem to arise in this application is tree-searching. Each move must be evaluated and then the possible consequences of that move and then their consequences and so on. Unfortunately, to evaluate all possible moves would be prohibitive so clever methods are used to "prune" the search tree. As a result the load resulting from the evaluation of a particular branch is difficult to predict leading to the load

An example: computer chess

imbalance problems described above.

"Master-Slave" solutions

A viable solution to this sort of problem is the "master-slave" approach. Various nodes in the tree (and corresponding nodes in the parallel processor) are designated "masters" in that they will be responsible for allocating work to the "slave" nodes. These slaves then process whatever data they are supplied by their master until a given processing task is complete. Typically the number of slaves is made smaller than the number of tasks to be performed so that load-balance can be achieved in a statistical sense - if one node receives a large task it will process it slowly but in the meantime the other slaves can be processing several smaller tasks each. As a result the entire problem is sped up.

Note that we can implement this style of programming without multi-tasking on each node of the parallel machine by allocating one piece of work to each slave node. On the other hand it may be more natural to just distribute all the subtasks at once to the slave nodes which in turn execute the various pieces in parallel on each node. This increases the amount of parallelism at the expense of some complexity.

Data-base and transaction processing

Other fields in which the "synchronous" style is not appropriate include data-base searches and transaction processing. These cases are characterized by distributed data which needs to be searched in an inhomogeneous manner. If we consider a large transaction processing machine we can imagine many requests appearing at once. Each will be dispatched to the part of the machine responsible for this sort of task which will, in turn, generate more requests in other nodes. The particular pattern of requests to be made in each processor is totally unpredictable in advance so a synchronous programming style is very poor - it would be disastrous to have some key node waiting for a complex result to be calculated holding up several other quite simple requests.

In these cases we wish to have some ability to asynchronously process events on a particular node and create arbitrary tasks on different nodes. *Express* provides these facilities.

Multitasking under Express

The multitasking extensions of *Express* form a particularly elegant solution to the above problems. In many other systems (The UNIX operating system is the classic example) the multitasking and interprocess communication features seem to be part of disjoint systems "tacked together" at the last minute to present a usable interface for programmers. Since *Express* is basically a system designed around the inter-processor message passing system we use these utilities in the implementation of the multitasking system. As a result the overall picture for the application designer is extremely clean and simple.

The basic principle involved is that messages become tasks when they reach their destination processor. As a result we can transparently create tasks locally, by sending messages to our own node, or remotely by sending messages to other nodes. In either case we can also send data to the created tasks in the message which created it.

Problems with asynchronous multitasking

Having said that the *Express* system provides an elegant solution to the multi-tasking problem we must, however, make the caveat that this style of programming is significantly more complex than the "loosely synchronous" model described earlier. The behavior of the system is extremely difficult to follow analytically if programmed in this way which makes the detection of errors much more complex. In particular powerful source level debuggers become a much reduced resource in many cases since they operate at interactive, human, speeds rather than the computer clock rate. As a result, the time it takes to type in a debugger command upsets the timing properties of the program being debugged so much that often the bug does not appear. Real time debugging tools are a complex and little developed issue. *Express* provides one utility which has proved useful but the problem remains. (The RAM file, see the *Cubix* documentation for more details.)

Hardware dependencies

We must also note that the availability of the *Express* multitasking features is extremely hardware dependent. At present we are only able to offer the full system on transputer based architectures. There is, however, a level at which the *Express* functions described in this document can be used portably across all machines. The techniques involved are discussed in Section 5.

This manual is structured as follows. In Section 2 we describe the function which implements the majority of the multitasking interface under *Express*. Some simple examples are shown to exhibit the features of the system. In Section 3 we discuss the central issue of "semaphores" which are needed to prevent multiple tasks from corrupting one another. In the absence of memory protection hardware this is a very important issue. We then present a fairly sophisticated example of the use of the multi-tasking functions. In Section 4 is discussed a simple mechanism similar to the UNIX `exec` in which a node program can be swapped out of memory and replaced by another on demand. Finally we describe the portable use of the *Express* multitasking functions.

2 Asynchronous Processing - `exhandle`

exhandle - a communication function

The function which implements the heart of *Express*' multitasking facility is `exhandle`. If you consult the *Express* documentation you will find this function described among the communication functions rather than in its own section for multiprocessing. This is because multi-tasking under *Express* is a function of the communication system rather than a separate set of function calls. This is what makes the interface so clean and simple - since a user of *Express* is already accustomed to the basic message passing principles we merely extend them a little to provide multitasking support.

Associating a message with a function

The syntax of the `exhandle` function call is

```
     int exhandle(function, source, type)
     int (*function)();
     int *source;
     int *type;
```

What is achieved by this call is to associate the named `function` with messages from the given `source` which have the specified `type`. Subsequent to this call every time a message arrives at this node which matches the `source/type` combination results in `function` being executed with the message parameters as arguments.

A contrived example

To clarify this procedure let us consider a trivial example. We assume that node 0 will repeatedly make some request from node 1, at unpredictable intervals. Further, node 1 is expected to respond to each request with a counter value which, for the sake of simplicity, we will assume is being calculated by the normal, sequential program, executed on the processor. The *Express* code to achieve this is

```
/*
 * Node 0.
 */
#include "express.h"
#include <stdio.h
/*
 * Types for counter 'request' and 'responses'.
 */
#define  READ_VAL    (123)
#define  RESP_VAL    (124)

main()
{
     int counter, dest, type;

     fasync(stdout);
            /* Allow independent access to stdout */
     exsync();
            /* Synchronize processors */

     counter = 0;    /* Prevents early termination */
     dest = 1;       /* Read data from node 1 */

     while(counter != 0) {

         type = READ_VAL;
         exwrite(&dummy, 0, &dest, &type);

         type = RESP_VAL;
         exread(&counter, sizeof(counter),
                                &dest, &type);

         printf("This time we got %d\n", counter);
     }
     printf("Node 1 told us to quit\n");
     exit(0);
}
```

```
/*
 * Node 1.
 */
#include <stdio.h>
#include <express.h>
#define  READ_VAL    (123)
```

```
#define  RESP_VAL    (124)

int counter = 534231;   /* The value being requested */

main()
{
     int type, src;
     int send_counter();

/* Set up message handler */

     type = READ_VAL;
     src = DONTCARE;
     if(exhandle(send_counter, &src, &type) < 0)
         abort(1);
     fasync(stdout);
     exsync();  /* Synchronize with other nodes */

     while(1) recalc(counter);
}

/* This function will be called whenever the message
 * of type READ_VAL is sent to this node.
 */
int send_counter(buf, len, src, type)
char *buf;
int len, *src, *type;
{
     int rtype = RESP_VAL;

     exwrite(&counter, sizeof(counter), src, &rtype);
     return 1;
}
```

Let us examine this code step by step.

The first action performed by node 0 is to execute `fasync` for the standard output stream `stdout`. This enables node 0 to print things independently of the other nodes in the machine.

Use asynchronous I/O is asynchronous programs

At the same time node 1 is setting up its message handler with the call to `exhandle`. Note that we specify a particular message type but the wildcard `DONTCARE` value for the message source. This allows the processor to automatically respond to requests for the `counter` value from *any* node including the hosts or the node itself. Note that we check the return code from `exhandle` and abort cleanly if something bad has happened. This is good practice since a limited number of message handlers may be registered.

Avoiding race conditions and tricky bugs

The second action performed in both nodes is a call to `exsync`. This is very important. Without this call we generate a potential bug because node 0 could send off its request for data before node 1 is ready to receive it. In this case *Express* would be forced to treat the message as normal interprocessor communication. In this case, node 1 makes no calls to `exread` and so would never see the message. Finally node 0 would never get a response to its request so it would hang forever in `exread`. By making the nodes synchronize after the setting up of the message handler we guarantee that no node sends off a request that is unexpected.

After setting up its message handler node 1 proceeds to calculate successive values of `counter` via some procedure not shown here.

Activating the handler function

Node 0, however, sits in a loop sending requests to node 1 for the current value. Note that these are merely dummy messages of zero length but with the correct type. When each of these messages arrives in node 1 the `send_counter` function is called with the arguments shown. In our case the message buffer and length are superfluous since no information was sent by node 0. The `src` field tells us which node sent the request for data - in our case it is known to be node 0 but if more than two nodes were involved this field would enable us to identify a node to which data should be sent. Finally the `type` field is again superfluous in this example but we could have specified `DONTCARE` in the original call to `exhandle` in which case the `send_counter` function would be invoked for every message arriving on node 1. In this case the last field serves to discriminate between types.

Important properties of the handling function

Two other important features of the `send_counter` function are its return code and the message type it uses for its response.

The return code is used to decide whether or not to terminate the association between message handler and message types. If the `send_counter` function had returned -1 then *Express* would have treated all subsequent messages of type `READ_VAL` as normal interprocessor communication rather than calling the handler function. Returning 1 maintains the association.

Avoiding infinite chains of request/ ack messages

The message type used in the response of the message handler must also be considered carefully. The simplest solution is to merely return the results in a message of the same type that we received. After all, it is supplied to us as one of the arguments. In the case shown here this will work properly. In a more general case, however, each node would be calculating values and would have its own message handler installed to trap messages of type `READ_VAL`. If we now return the results in a message of this type the handler on the originating node would treat it as a Vrequest for a value and would send something else back. This would lead to an infinite sequence of requests and acknowledgments which would grind the machine to a standstill.

Stopping asynchronous programs

Note that the code shown above might run forever - node 0 will repeatedly request the `counter` value from node 1 stopping only when the value 0 is received. It is obviously up to node 1, in this scenario, to decide when things have gone far enough and stop. This is actually one of the trickier aspects of this style of processing - knowing when and how to stop cleanly.

This example has shown the most basic usage of the message handling system. What may have slipped by unnoticed, however, is that we have actually been multitasking here. While the main loop in node 1 is repeatedly calculating new values of `counter` a second task has been answering enquiries from node 0 about the current status.

Extension to signal processing

We can extend this model more if we realize that each message which arrives on node 1 is actually creating a new task there which runs in parallel with the other tasks on that node. (This is only literally true on machines which actually support multitasking such as transputers. An alternative (portable) interpretation is given in Section 5. To examine this feature more consider a signal processing application.

Signal processing applications involve the interconnection of numerous "black boxes" representing the individual system components. Each of these black boxes receives, as input, a piece of the signal which it processes and then passes on to the next black box in the chain. The feature of this system which most recommends a multi-tasking solution is that the number of components is very variable as are their computational requirements. Simple components may require small amounts of CPU power while FFT's can themselves benefit from additional parallelism. We might thus implement them as multiple processors.

An additional problem is that the prototyping process typically involves repeated minor modifications of the circuit. This is most easily implemented by merely generating a new task for each extra system component which we can then position on some underloaded processor.

The basic scenario is shown below. Each node sets up handlers for the various modules it may be called upon to execute. Several shown include simple FFT's, linear filters and several varieties of other "black box". Each of these handlers is set up to receive requests from any node and is triggered by the receipt of a particular message type.

```
#include <express.h>
#include "msgtypes.h"
                /* Types for various functions */

extern int FFT(), linear_filt(),
           black_box1(), black_box2();
extern int stop();
int done = 0;    /* For termination */

main()
{
     int src = DONTCARE, type;

     type = FFT_REQ;
     exhandle(FFT, &src, &type);

     type = LINFILT_REQ;
     exhandle(linear_filt, &src, &type);

     type = BBX1;
```

```
        exhandle(black_box1, &src, &type);

        type = BBX2;
        exhandle(block_box1, &src, &type);

        type = STOP;
        exhandle(stop, &src, &type);

        exsync();

        while(!done);/* BUG ..... see next section!! */
        exit(0);
}

int stop(buf, len, src, type)
        char *buf;
        int len, *src, *type;
{
        done = 1;
        return 0;
}
```

A solution to the "stopping" problem

There are several interesting features of this code. First each node sets up handlers for each of the functions and then calls `exsync` to prevent race conditions as described in the previous example. We also set up a fairly primitive but effective mechanism for stopping the program - one of the message handlers is for the `STOP` message which sets a flag to let us drop out of the main loop. The most striking thing about this code is the main loop itself which is empty!

We assume that a master processor somewhere knows the details of the signal net to be tested and will create processing units on processors by sending the appropriate message types.

This scenario has a number of important advantages.

- Overall control is exercised from some central location which minimizes the total amount of data which must be transmitted to the individual nodes.
- Any node in the machine (including the hosts) can create a task by sending an appropriately typed message to any node. This means that we can adopt some sophisticated dynamic load balancing strategy - as nodes become too busy we can create processes elsewhere.
- Individual modules may themselves operate in parallel. The FFT routine, for example, might use four or more nodes to perform the FFT in parallel. Within this subset of the machine it can take advantage of a "loosely synchronous" programming style with all its advantages.
- Any necessary multitasking is completely transparent. If the number of modules

we need to create is requires fewer nodes than are available then each node can be given one process. If more are required then we can create more simply by sending messages.

- Each fundamental process can be debugged in isolation and then "plugged" into the larger system.
- Prototyping is very straightforward since the master process merely has to order the creation of extra modules by sending the necessary messages. Networks can be created with a graphical interface and then implemented trivially through this mechanism.

Having looked at the overall structure of the `main` routine each of the modules would probably have the overall structure shown below.

```
black_box1(data, length, src, type)
char *data;
int length;
int *src, *type;
{
     int next_node, next_type;

/* We got some data sent to us in the instantiating
 * message. If this is not enough then we can read
 * more from the node which started us up.
 */
     exread(buffer, length, src, msg_type);
     exread(buffer, ...........);

         .......

/* Process the data. */

             .......

/* Find out who to pass the results on to and then
 * send the message. We also have to decide exactly
 * what message types to use since some will create
 * tasks while others will merely interface to
 * existing ones.
 */
     next_node = .....
     next_type = .....
     exwrite(results, length, &next_node, &next_type);

     return 0;
}
```

Some of the data to be transformed is sent within the triggering message itself. This is an important optimization in simple cases but may not be sufficient. `exhandle` is only able to deal with messages of lengths up to the *Express* buffer size (as set in the customization procedure - see `excustom` for more details.). If the data to be transformed were larger extra messages could be sent and read directly by the message handler as shown above. These would be normal *Express* messages, sent with `exread/exwrite`.

Independent tasks make for the easiest processing

So far all the examples have been quite straightforward. Tasks have been totally independent. In this case few precautions have to be taken. In general, however, this will not be the case - several tasks on a particular node must coordinate their actions so that neither voids the operation of the other. We discuss this issue in the next section.

Avoid "busy" waiting

Also discussed in the next section are techniques for "waiting" in a multitasking environment. When a normal sequential process cannot proceed any further it typically "blocks" - i.e., waits in some sort of loop for an event to occur which will restart it. In a multitasking system this is not good since a task which waits in this manner never yields the CPU to another task which could potentially do useful work. In the worst case one task might "busy wait" for another task on the same node which will result in deadlock since the task that could potentially free up the situation will never get access to the CPU. For this reason a task under *Express* should never wait in a loop. If it becomes necessary to wait for some event then a "sleep" function is provided which will yield the CPU to any other pending process. This is the bug mentioned in the source code of the above examples and techniques for avoiding it are described in the next section.

3 Mutual Exclusion - Semaphores

A "critical section" is a piece of code that can only be executed by one process at once. A simple example is provided by the oft-quoted bank-teller model. Consider service at a bank. We assume that two transactions need to be performed which both have to be recorded in your account. If only one teller is available all is well since the two transactions will be processed separately and recorded correctly. If two tellers happen to decide to help you together, however, problems can arise. Let us assume that the first rushes ahead and gets half way through recording your transaction when some interruption occurs which allows the second teller to catch up. When it comes to recording the results chaos will result since the second teller is modifying data that is only partially correct.

A model of a "banking" operation

This is the classic case of a critical section. Some mechanism must be provided which prevents one teller modifying data unless it is in a sensible state. Many mechanisms are available for this of which *Express* adopts the semaphore technique.

Semaphores

A semaphore is a variable which controls access to certain pieces of code. A simplified case is shown below

```
/*
 * Simple semaphore code.
 * This code is illustrative but not sufficient for real
 * use.
 */
```

```
int semaphore = 0;

modify_accounts(account, credit)
int account;
int credit;
{
     while(semaphore != 0);
     semaphore = 1;

     account += credit;

     semaphore = 0;
     return;
}
```

This example is a simple model of the teller's problem. The routine `modify_accounts` has two arguments: the current account standing and an amount to credit to this account. In order to prevent several tellers modifying things incorrectly we invent the extra variable, `semaphore`. While the value of this flag is zero any teller is allowed to carry out the transaction but whenever one of them does so the value is immediately changed to 1. This prevents any other tellers inadvertently messing things up. Whenever a modification is complete the `semaphore` is set back to zero allowing another teller to modify the balance.

This example is rather trivial and the code shown above is actually insufficient for correct operation in the *Express* multitasking environment but it serves to illustrate the purpose of the semaphores. This type of operation occurs whenever some data structure on a node will be shared *and modified* by several tasks. If only one task is allowed to alter the balance and the others only read it all will be well. We could, for example, make use of the two tellers in the example by having one do the calculations and the other modify the balance. There will then be no conflicts and no semaphore is necessary.

Where are the problems in the above solution?

Problem 1: "Busy" waiting

The first problem is that tasks which get to the `semaphore` test while it is inaccessible will "busy wait". This phrase was introduced at the end of the previous section and refers to the situation where one task continually uses the CPU. In this case the `while` loop could execute forever preventing the task which is actually performing the transaction from finishing - it may actually be just on the verge of resetting `semaphore` to zero and thus freeing the waiting task. It may be unable to do so since it cannot get to the CPU.

Problem 2: atomic semaphore operations

The second problem is with the semaphore itself. The assignment

```
semaphore = 1
```

is not attached to the end of the `while` loop in any concrete fashion. We could thus be pessimistic and consider the possibility that one task, having decided to modify the balance wants to set the semaphore preventing others from doing the same. Unfortunately, just as this decision is made another task comes along and makes the same decision since the first hasn't had chance to lock the semaphore. The result is that both tasks proceed equally convinced that everything is OK. Both will eventually set the semaphore variable and unset it correctly while the account that we set out to protect is vulnerable.

In order to prevent this from happening the semaphore modification process must be made "atomic" - once the decision has been made to lock the gate it must be able to do so without interference from other tasks which may potentially make the same decision.

The Express semaphore functions

To implement this strategy *Express* provides several semaphore functions: exsemalloc, exsemwait, exsemsig, exsemfree and exsleep.

The basic variable type for these routines is EXSEM as defined in the standard header file express.h. (In FORTRAN an array of integers will suffice - see the reference manual for details.) exsemalloc returns a pointer to a new semaphore structure which can be freed, if necessary, with exsemfree. The other two functions perform the two assignment operations of the previous example. A correct *Express* version of the above code is

```
/*
 * Simple semaphore code.
 * Uses EXPRESS functions to lock records.
 */

#include <express.h>

EXSEM *account_sem;
                    /* Semaphore protecting accounts */
#define CREDIT_ACCT     (400)
                    /* Message type for task */

main()
{
     int type, src, credit_account();

/* Create necessary structure for modify_accounts
 * tasks
 */
     account_sem = exsemalloc();

     if(account_sem == (EXSEM *)0) abort(1);
     type = CREDIT_ACCT;
     src = DONTCARE;
     exhandle(credit_account, &src, &type);

             ............
}

struct acct_data {
     int account;   /* Account number for transaction */
```

```
        int credit;      /* Credit/debit amount */
        int ssno;        /* Social security number */
         ......
    };

    credit_account(acct_data, length, src, type)
    struct transaction acct_data;
    int length;
    int *src, *type;
    {
         modify_accounts(acct_data.account,
                         acct_data.credit);
         return 1;
    }

    modify_accounts(account, credit)
    int account;
    int credit;
    {
        exsemwait(account_sem);

        account += credit;

        exsemsig(account_sem);
        return;
    }
```

There is quite a lot of code here since we actually built a working program.

The only changes to the `modify_accounts` function, however, are trivial. We call `exsemwait` before the critical section and `exsemsig` after it. The former call encapsulates the waiting loop and semaphore assignment of the original example. It arranges that any task which is prevented from proceeding by the lock semaphore waits in such a way as to allow other tasks to use the CPU. It further arranges that the semaphore locking procedure is atomic preventing more than one task from getting into the critical section.

The `exsemsig` call releases the locked semaphore thus allowing any waiting processes to proceed into the critical section.

A simple transaction processing system

The rest of this code is shown to illustrate how simply we can build a prototype transaction processing engine from the *Express* functions. In the `main` routine we create a semaphore for the `accounts` procedures by calling `exsemalloc`. We then set up a handler so that we can easily create tasks to modify accounts by simply sending the appropriate messages. Together with the message that creates a task will be sent the data necessary for the modification. For our convenience we define a structure for this information. The task then calls the account_modify procedure which is now correctly protected by semaphores.

To complete the picture of this system let us assume that every node has executed the above code to create the appropriate tasks. We can now have a transaction processing task which decides which account to modify and then does so by sending a simple *Express* message as follows

```
#include <express.h>

#define CREDIT_ACCT     (400)
                 /* Message type for task */
struct transaction acct_data;
                     /* For passing to the remote node */

fix_account(global_account, credit)
int account;
int credit;
{
     int node, local_account;

/* Calculate which node the actual account data is
 * stored on and which local account number
 * corresponds to the global one we've been given.
 */
     node = ......
     local_account = .....

/* Now send the transaction request to the appropriate
 * node. The invocation message contains all the data
 * necessary for the operation to be completed.
 */
     acct_data.account = local_account;
     acct_data.credit = credit;

     type = CREDIT_ACCT;
     exwrite(&acct_data, sizeof(acct_data),
                        &node, &type);
     return;
}
```

In this code we assume that we have been passed an account number which must be located within the parallel machine. The code to deal with this could involve some "name-server" or other technology easily constructed with the *Express* functions. Having decided where the account is kept we can just send that node a simple message to have it update its records. Note that this system is very robust - the semaphore protection prevents multiple transactions from simultaneously modifying the accounts. Note also that the above system works even if the account to be modified is on the same node as the task which sends the message! This is nice since it provides neat modularity to the code. One might otherwise be tempted to "cheat" by having the local node modify its accounts directly. This would be an easy way to make a mistake since the semaphore would have to protect both the `modify_accounts` function for remote access and whatever code were used to deal with local cases.

Solving the "busy" wait problem with exsleep

So far we have discussed all the semaphore function except `exsleep`. This function has a rather messy use - it is required whenever a task needs to wait but will not be waiting on a semaphore. One possible use would be at the end of our signal processing example from the previous section. This code set up several potential tasks with the `exhandle` call and then waited for a signal to terminate with the code

```
/*
 * Wait for messages to start up multitasking system.
 * INCORRECT
 */
main()
{
          ....

     exsync();

     while(!done);        /* BUG - see next section!! */
     exit(0);
}
```

The problem here is that mentioned several times already - the node is "busy waiting". This can potentially hang the system since no other process can gain access to the CPU. The fix to this bug is quite simple

```
/*
 * Wait for messages to start up multitasking system.
 * CORRECT
 */
main()
{
          ....

     exsync();

     while(!done) exsleep(100); /* Wait for messages */
     exit(0);
}
```

Note that we now "sleep" in the loop waiting for messages. This allows other tasks to use the CPU and prevents deadlock. The argument to this function is a time in milliseconds for which the process should sleep. In this particular application the actual value is rather unimportant - only the effect is crucial.

This section has described in detail the methods which can be used to "secure" a multitasking environment. The example, though easily coded shows the simple manner in which *Express* is able to cope with a typical multitasking problem. It has also pointed out several common sources of error. We only wish that detecting errors in this type of code were as simple as describing them!

4 Executing alternate node programs under *Cubix*

An "exec" model of multiprocessing

The previous sections have described a traditional multitasking system in which processes are created, perform their actions and disappear. A simpler form of multitasking is to simply replace one node program with another, *in toto*. While this is not a particularly subtle or elegant solution it can occasionally have its uses. One might, for example, design a system which is too big to fit into the memory of a single processing node. In this case it might be useful to partition the application into large "phases" which are loaded one after another to accomplish some processing task.

The basic function which achieves this is `execve`. In its most basic form one node program can call

```
execve("phase2", (char **)0, (char **)0);
```

which replaces the current node program with one called `phase2`. Notice that two other arguments are passed to the function. These are analogous to the arguments used by the `exload` system calls - arguments and an environment for the new program. In the above example neither will be given to the new program.

A more general example might be the following code segment

```
extern char **environ;

do_exec()
{
     char *args[5];

     args[0] = "This";
     args[1] = "is";
     args[2] = "a";
     args[3] = "test";
     args[4] = (char *)0;

     execve("next_phase", args, environ);
}
```

Passing arguments to another node program

In this case we build an array of character string pointers (including the `NULL` at the end) and then use `execve` to start up a new program. In this case we pass the vector of arguments and also our current environment to the new program which will receive them through the normal `argc`, `argv` mechanism.

The system call just described operates "loosely synchronously" in that it must be made in all nodes and results in the replacement of all node programs. Single nodes may start up new program with the `aexecve` system call which has identical arguments to that shown above but operates independently in each node.

An important possibility with this function is that program can share data if done carefully. When *Express* loads a new program into the machine it only zeros the memory explicitly used by the new code. It is possible, therefore, by judicial use of physical memory addresses to have successive phases of an application share data through the `exec` mechanism. This practice must be used carefully since program sizes and memory allocation strategies are rather machine dependent but might prove invaluable on occasion.

5 Portable use of `exhandle`

The multitasking system described in Sections 2 and 3 provides significant flexibility in the generation and manipulation of tasks. `exhandle` can, however, be used in two modes, one portable between different *Express* implementations, the other not.

Interpreting exhandle as an interrupt handler

At its simplest `exhandle` can be considered to setup an "event" triggered by the arrival of a message. This event executes the user supplied function and eventually terminates returning control to the main user program. As we have seen, however, we can implement systems using `exhandle` which have essentially no "main" program. They merely create tasks at the request of the message system which then execute forever in parallel.

On machines such as the transputer which support multi-tasking all features of *Express* will be implemented and either style of programming will be available. In the absence of such facilities, however, `exhandle` events will be dealt with as normal hardware interrupts would be - normal program execution is suspended while the interrupt handling routine is serviced. As a result portable programs should not assume that two tasks can necessarily interact simultaneously.

If used only in this mode *Express* programs will be portable between all machines that use `exhandle`. Certain restrictions apply, however, due to the fact that the user supplied function will essentially be invoked at interrupt time.

The most fundamental of these constraints is that the user routine must never "wait" for anything. In particular it should never call `exread`, `exsleep` or `exsemwait`. It can, however, call `exwrite`.

While this might seem a large restriction it turns out to be less so in practice. If we consider the banking example of Section 3 we could modify the function which performs transactions so that it merely failed whenever a conflict might arise between the interrupt handler and the normal code. In this case a message would be sent back to the caller indicating this and a retry could be issued. While not as efficient as the other solution it offers enhanced portability.

Parallel Graphics

A simple, portable, parallel graphics system: *Plotix*

1 Introduction

Why graphical presentation is so important

Graphical presentation is rapidly becoming one of the major concerns of all types of programmers. At one end of the scale are sophisticated menu-driven applications offering an enormous degree of interaction and freedom while at the other end might be the simple graph displaying the final result of many hours of CPU power. Both examples are really questions of data-reduction; the menu interface reduces what might otherwise be an extremely complex input syntax to a simple "point and click" model while the graph takes thousands of data points and presents them in a form that can be readily understood.

Parallel computers pose the same problems to a larger extent. If the sequential computer could produce a thousand data points then the parallel machine might generate a hundred thousand. Similarly, if one wishes to get any "feel" for what is happening inside a parallel machine, one rapidly has to resort to graphical displays since the human mind cannot comprehend what would otherwise be pages of randomly sorted data.

The philosophy of Plotix

Plotix is a graphical system designed for parallel machines. In keeping with the rest of the *Express* philosophy, an important goal is the portability of the resulting parallel program and *Plotix* is thus implemented in both sequential and parallel forms. We have found that quite complicated systems can easily be developed that really enhance parallel applications - finite element systems that display, "on the fly", images of bending plates as well as documenting (and incidentally aiding the debugging of) the internal data structures of the code. Menu-driven applications have also been developed - the *ParaSoft* performance monitor is a good example.

Achieving device independence at runtime

Plotix programs produce device independent output in the sense that the same parallel program can produce output on a variety of different devices merely by altering a run-time switch in the `cubix` command that executes the program. This allows one, for example, to generate hardcopy output without having to recompile a known working program.

Due to its inherent simplicity we have been able to port the system to a wide variety of devices. Currently supported are the 4010 and 4105 series Tektronix terminals, the HP475 series of pen plotters from Hewlett-Packard, the IBM EGA under both XENIX and MS-DOS, PostScript, CGI (on systems such as the SUN), Xwindows, QuickDraw and many others.

Output models

While the simplest *Plotix* model involves a single output device, usually attached to the system console alternative schemes are supported. It is possible, for example, to redirect graphical output to special purpose devices attached to the parallel processing network in fairly arbitrary ways. It is even possible for different nodes to send their output to distinct displays. All of this support is provided at runtime through simple function calls.

The rest of this document is arranged as follows. In section 2 are notes concerning the coordinate system used by *Plotix*. Section 3 discusses the calls required to initialize and shutdown the plotting system and the output modes available in *Plotix* which actually put images up on your display surface. Section 4 describes the input routines available under *Plotix*, Section 5 the contouring package and Section 6 the interpretation of "color" supported by *Plotix*. Section 7 describes the issue of device-dependency and introduces the *Plotix* functions designed to deal with this issue. Section 8 describes the example programs that are supplied with *Plotix* and also contains the complete C source code for a simple program. Section 9 describes the inner workings of the contouring system which are occasionally valuable when working on non-rectangular domains and Section 10 describes the devices on which *Plotix* is available and lists the particular idiosyncrasies of each.

2 Coordinate systems

In most systems the user is bombarded from the outset with a bewildering collection of coordinate systems which must be understood before plotting can commence. *Plotix* can in the simplest case have a single coordinate system covering the view-surface, or by appropriate function calls build up to sets of 'windows' or 'viewports' each with its own coordinates.

All problems have their own natural scale, for instance meters, kilograms, minutes or slug-furlongs. *Plotix* provides the `space` function call that allows you to set things up so that you can `move` around and `draw` lines in this coordinate system rather than having to rescale to an internal coordinate system.

An extremely simple coordinate system

By default *Plotix* assumes that the values you want to plot lie in the range from zero to one in both x and y directions and it sets up its coordinate transformations so that this range covers the entire screen. To see the effect of this consider the following simple code extract

```
            move(0.0, 0.0);
            cont(1.0, 1.0);
```

which draws a diagonal line from one corner of the display to the other as shown in Figure 1.

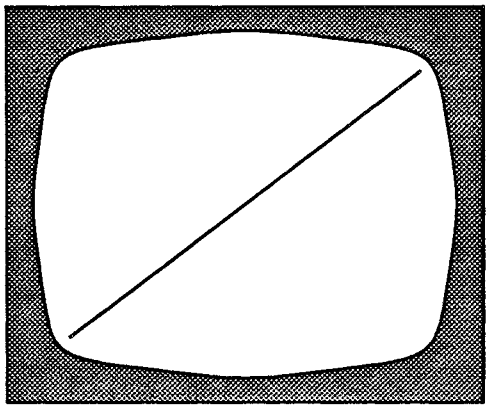


Figure 1. Default Coordinate Range

If your data happen to lie in the range from zero to one all is obviously well but experience shows that this happens rarely. In this case you call `space` to set up your own coordinate range. The arguments to this function are the lower left and upper right corner of the rectangle that encloses your data, in your units. So, for example, if you have to plot salary against month of the year you might want to have the x-scale range from 0 to 12 and the y-scale from 10,000 to 100,000 ("Dream on?"); this would be accomplished with the call

```
        space(0.0, 10000., 12., 100000.);
```

As a concrete example of the effect of this call modify the above example code segment by the addition of a call to space.

Modifying the default coordinate system

```
        space(0.0, 0.0, 2.0, 2.0);
        move(0.0, 0.0);
        cont(1.0, 1.0);
```

The effect of this is shown in Figure 2. As can be seen the space call has doubled the scales of the axes so that the unit diagonal now stretches only half way across the screen.

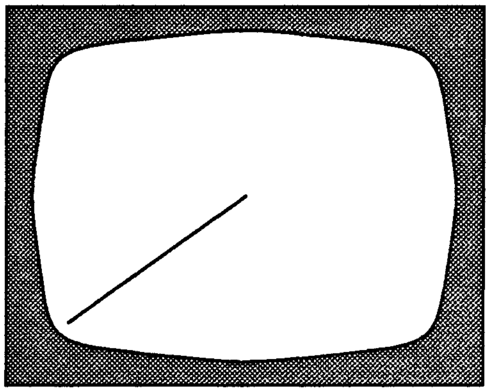


Figure 2. User Supplied Coordinate Range - space

There is another flavor of the space function called ortho_space which ensures that when a square is drawn it appears on the display as a square rather than a rectangle.

Making "squares" square

```
        justify = 0;
        ortho_space(0.0, 0.0, 2.0, 2.0, justify);
        move(0.0, 0.0);
        cont(1.0, 1.0);
```

The effect of this is shown in Figure 3. Now the diagonal line is actually at 45 degrees to the horizontal. ortho_space finds the largest rectangular region of the display which has the aspect ratio specified by its arguments and maps the first pair of user-coordinates to the lower left corner of the rectangle and the second pair to the upper right corner. Notice the last argument to ortho_space, which specifies where this largest rectangle should be placed on the display surface. If justify is -1, the rectangle is placed as far left (or down) as possible, if 1 it is as far right (or up) as possible, if 0 then it is centered in the left-right (or up-down) direction.

Positioning the output on the display - viewports

Now that one is able to rescale the image to a particular set of units the next question is how to place it on the display surface. Again, by default, *Plotix* fills up the entire view surface with its output and makes available the vport call to override this choice. The arguments

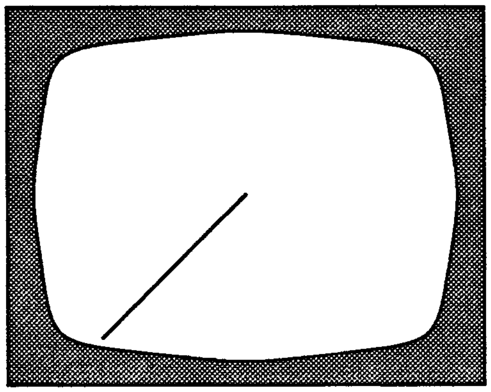


Figure 3. User Supplied Coordinate Range - `ortho_space`

to this function are again the lower left and upper right corners of the display surface upon which you wish to draw your picture, expressed as fractions of the whole. This means that the default situation corresponds to the call

```
vport(0.0, 0.0, 1.0, 1.0);
```

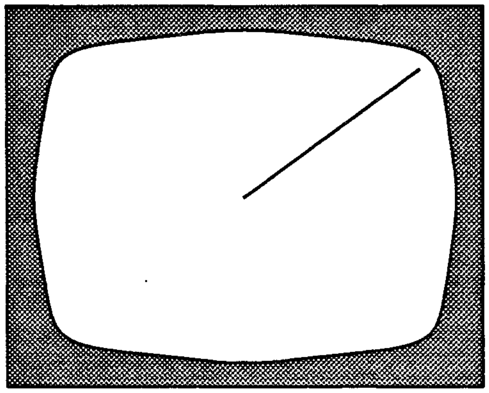
while the call

```
vport(0.5, 0.5, 1.0, 1.0);
```

places your image in the upper right quadrant of the screen. As a specific example consider the code fragment

```
    vport(0.5, 0.5, 1.0, 1.0);
    move(0.0, 0.0);
    cont(1.0, 1.0);
```

whose effect is shown in Figure 4. The call to `vport` places the entire image so that the diagonal line is now in the upper right hand corner of the screen.



Independent calls to space and vport allow the nodes to create special effects

Figure 4. Modifying the Display Area

Calls to `space` and `vport` may be combined. So, for instance, we can take the screen and divide it up into regions controlled by individual processors of a multi-processor system and then assign within each processor its own coordinate mapping system. An application where this is of use, for example, is image processing where each processor has a sub-image of the whole picture. It make sense to divide up the display surface so that each processor has an area of the screen corresponding to its own sub-image. Then one can map the individual subsections of the screen so that actual plotting commands can be made on the basis of array indices within a processor.

Multiple viewports are a substitute for "windows"

There may be multiple viewports in *Plotix*, so that for example we can show a menu in one, and plan and elevation views of an object in two other viewports. First we divide up the screen into pieces:

```
int menuport, planport, elevport;
menuport = vport(0.0, 0.0, 0.2, 1.0);
elevport = vport(0.2, 0.0, 1.0, 0.5);
planport = vport(0.2, 0.5, 1.0, 1.0);
```

which puts the menu in a strip at the left, and splits the rest of the screen horizontally for the plan and elevation. Now we can set up coordinates for each of the viewports:

```
setvport(menuport);
space(0.0, 0.0, 10.0, 1.0);
setvport(elevport);
ortho_space(0.0, 0.0, 1.0, 1.0, 0);
setvport(planport);
ortho_space(0.0, 0.0, 1.0, 1.0, 0);
```

which makes the vertical coordinates in the menu viewport range from 0 to 10, and provides a unit square for the coordinates in the plan and elevation viewports. Notice that these last two will not be distorted. Finally we can draw:

```
setvport(elevport);
    ...draw elevation...
setvport(planport);
    ...draw plan...
```

Notice the order in which these functions were called. Several calls to `vport` preceded the calls to `space` which were, in turn, each preceded by a call to `setvport`. The reason for this sequence is that each call to `space` affects only the currently selected `vport`. Furthermore a viewport is selected either by its creation with the `vport` function or its explicit selection with `setvport`. Output which might be generated from this code is shown in Figure 5.

In parallel, each processor may make a different (set of) viewport(s). As a final example of this sort of procedure let's consider a case of four processors arranged in a 2 x 2 square. Furthermore, assume that each processor has already figured out it's coordinates within this

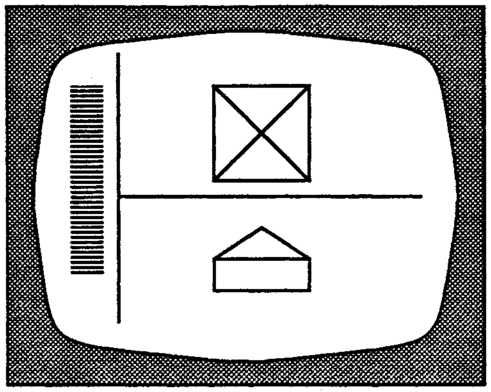


Figure 5. Multiple viewports

square pattern and stored them in the variables `xcoord` and `ycoord`. (This is trivially done through the `exgrid` system supplied as part of *Express* - a full example is given in Section 8.) Then the following code segment

```
     {
          double xcoord, ycoord;

     /* Code to calculate x and ycoord using exgrid */

          vport(xcoord*.5, ycoord*.5,
                      (xcoord+1.)*.5, (ycoord+1.)*.5);
          space(0.0, 0.0, 2.0, 2.0);

          move(0.0, 0.0);
          cont(1.0, 1.0);
```

produces the output shown in Figure 6.

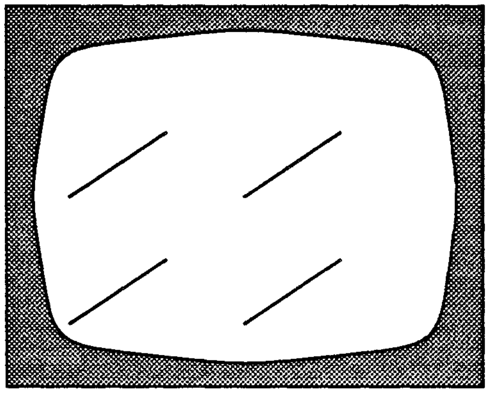


Figure 6. Combined vport and space transformations

The complete code showing the use of the `exgrid` software to perform the decomposition calculations is shown in Section 8.

3 Starting, Stopping and Flushing

One of the advantages of the simple graphical model provided by *Plotix* is that there is only a single function call required to setup the system (`openpl`) and a single call (`closepl`) to shut it down. There are, however, some complications due to the parallel nature of the system.

The arguments to openpl

The routine that starts plotting, `openpl`, has two arguments, a buffer size and a file pointer. The first argument is used to allocate a buffer into which graphical commands are placed until flushed by the user application. This makes parallel graphics more efficient by bundling up several calls and issuing them at once rather than sending many small messages for each graphical command as it comes. On the other hand it introduces "flushing" commands that are rather unfamiliar and unnatural in the sequential computing world. These commands are described in some detail later in this section - for now it is enough to know that a default buffer size is obtained by giving the `DONTCARE` value for this argument. (`DONTCARE` is defined in the header file `express.h`) This results in a buffer of a size determined by the operating system. If problems arise a more reasonable value can be determined by the user - this is also covered later in this section.

The second argument is a file pointer (i.e., of type "`FILE *`" in C notation - e.g., `stdout`). This argument is required for compatibility with future releases of *Plotix* which may feature a limited "metafile" capability. At the present this argument should be the `NULL` pointer value which directs the graphical output to go directly to the graphics device. Specifying any non-NULL pointer directs the system to create a metafile on the specified file - this will most likely blow away your terminal!

After processing the arguments given to it `openpl` returns a status value which indicates how well the system has coped. Positive values mean that plotting can now continue while negative values indicate some catastrophic error such as being unable to access a particular device, or the inability to acquire a buffer of the specified size. It is a good idea to always check the value returned and `abort` or `exit` if something is amiss.

After plotting is completed the single call `closepl` suffices to turn off any plotting systems. Note that this command only turns off the graphics device - it makes no output appear. The user is responsible for making sure that all graphical objects have been flushed to the display surface before calling `closepl`.

A skeleton Plotix program

The following two routines are shown in the following example. Note that the return status from `openpl` is checked and processing stops if something is wrong. This code is a good prototype for anything dealing with the *Plotix* devices.

```
/* Simple PLOTIX proto-code. */

#include "express.h"     /* Defines DONTCARE macro */
#include <stdio.h>
```

```
main()
{

/* Initialize graphics using default buffer size */

     if(openpl(DONTCARE, (FILE *)NULL) < 0)
             abort(101);

/* Application code using any graphics primitives
 * and flushing data to display at appropriate
 * times
 */

             ...

/* Application completed; call closepl to turn off
 * device
 */
     closepl();
     exit(0);
}
```

Buffering modes and graphical output

Why is output buffered?

Having opened up the graphical system one may now use any of the *Plotix* functions. There is, however, a significant departure from conventional graphics on a sequential computer in that output is "buffered". This means that as you draw objects nothing actually appears on the display surface until one of the three "flush" commands is executed. This is done for reasons of efficiency. Parallel computers typically have fairly low I/O rates - especially when compared with their large computing power and so it is a waste to send out graphics commands a few at a time - instead they are stored up in an internal buffer and then emitted when the user decides.

The relation between the buffering modes of Cubix and Plotix

The three "flush" commands follow closely the I/O modes of *Cubix* about which details are given in a previous chapter - "Programming Parallel Computers Without Programming Hosts". This document should be consulted as the major reference for the following discussion.

The three modes correspond roughly to the following situations

1. Each node has been drawing the same image - either a menu or an outline which is most easily done by all the nodes together. Only one copy of the resulting picture should actually be drawn on the display.

2. Each processor has been working on its own piece of the image and some natural synchronization point occurs at which it is convenient to update the display surface.

3. Each processor is working completely independently and needs to update

the display at unpredictable times.

These three modes are called "single-mode", "multi-mode" and "asynchronous mode" respectively after the corresponding concepts in *Cubix* I/O and each has its own "flush" command with its own constraints.

1. `sendplot` - Called in all processors at once this takes a single copy of the output image and flushes it to the display surface.
2. `usendplot` - Called in all processors at once this takes the graphical buffer from each node in turn and flushes it to the display surface. The individual nodes' images appear in order of increasing processor number.
3. `asendplot` - Called at any time in any processor this command flushes the graphics buffer from a particular node to the output device.

Synchronization constraints

Notice the correspondence between the three function calls and their usage as defined in the previous paragraph. The first two calls must be made "loosely synchronously" (This concept is discussed extensively in the *Cubix* documentation previously mentioned.) while the last can occur whenever required.

The three routines just described empty the internal graphics buffer on the nodes and make the display surface "current" in computer graphics jargon - i.e., it reflects exactly what you told it to.

Totally asynchronous operation

In addition to the asynchronous flushing command, `asendplot`, other commands are available which have no synchronization constraints; `aopenpl`, `aclosepl`, `aerase` and `agin`. The first three of these perform the same functions as their similarly named counterparts - i.e., opening, closing and erasing the display system respectively. The last performs graphical input and will be described in the next section.

The size of the graphics buffer

A question that remains is the size of the graphics buffer. As mentioned above we have found a system default of 8 Kbytes to be sufficient for most purposes. Occasionally this will be insufficient in cases where it is either inconvenient or impossible to call one of the flushing commands frequently enough. Alternatively the user may be able to call these commands sufficiently often that an 8 Kbyte buffer is wasteful and the size should be reduced. In order to diagnose these cases the function `plothwm` is provided. Called with no arguments this function merely returns the "high water mark" from the graphics buffer - i.e., the maximum usage to have occurred between any pair of flushing commands. Using this function the user is able to "tune" the size of the graphics buffer to an appropriate size.

The need for sendplot in sequential programs

For compatibility with the parallel implementation the calls `sendplot`, `usendplot`, `asendplot` and `plothwm` are available in the sequential *Plotix* libraries although they are usually stubs; the three flushing functions are identical and `plothwm` returns 0. (**NOTE:** It is still recommended that users make periodic calls to the flushing functions since this both simplifies the transition between sequential and parallel codes and also ensures that flushing is carried out on devices which need it even at the sequential level - X-Windows is a good example!)

Using alternate display devices

While discussing the commands which actually draw things on the display one should discuss the support provided for different output devices within the *Plotix* model. By default all graphical data is sent to the "console" device for processing and display. This device is normally the one at which you executed the original cubix command that loaded and executed your program. In certain cases, however, is may be beneficial to have an alternate scheme; special hardware systems or multiple output devices are good reasons.

In *Plotix* this is supported through the display_node function call. At any time one may alter the destination of further graphical output by naming either a new processor number in a call to display_node. Further output will be sent to this device which is assumed to be running a suitable server process. This exact procedure to be followed is described in the manual "Using *Express* on system with multiple hosts".

Note that the means by which nodes are "named" depends upon the manner in which the system was originally configured and is similar to the mechanism used by cubix to locate files.

It is important to note that different nodes are quite at liberty to flush their output to different places although they must do so with the asendplot function to prevent deadlock. This enables a further degree of parallel processing useful when one considers the typically small bandwidth of most machines in talking to the "real world". It is quite possible to build a system with multiple graphical devices to enhance throughput of displayed data.

4 Graphical Input

The area of graphical input is one of great complexity in most systems. *Plotix* circumvents this difficulty by providing only two input functions gin an agin. Both correspond to what most systems call *locator* input - i.e., you get a cross hair cursor to move around and eventually you click on some button/key and the call completes, returning appropriate information to you.

The Plotix input model - gin

To see that the practice is as simple as the above description consider the following sample code.

```
/* Sample code demonstrating the use of the "gin"
 * function
 */
    if((status = gin(&key, &x, &y)) < 0) abort(-1);
    else {
                /* Process gin values */
    }
```

Using the gin function to identify processors

The three parameters to gin are all pointers to values which are returned by the call. The first is an indication of which key or mouse button was pressed to terminate the call. It's exact interpretation depends upon the particular device in use and details can be found in Section 10. The second and third parameters will contain the x and y coordinates of the selected point in the user coordinate system - i.e., the one set up by the most recent call to space. The return value from gin is rather important. If the graphics device in use cannot perform input functions (e.g., PostScript or other hardcopy devices) then -1 is returned. Otherwise the return value is 0 or 1 depending on whether the selected point is within the window of the processor making the call. Consider the situation depicted in Figure 5 where each processor was responsible for a quarter of the screen and assume the user selects the point shown in Figure 7.

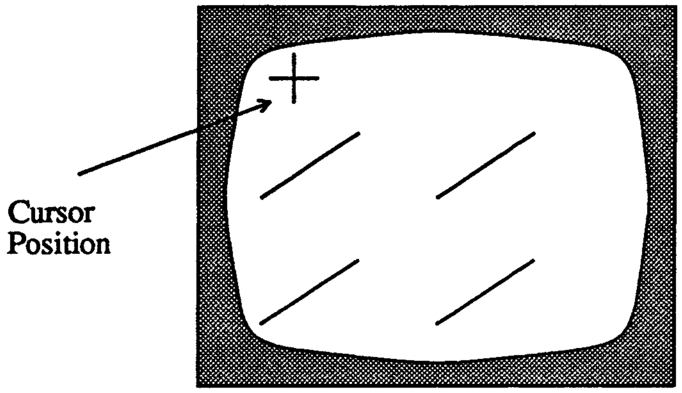


Figure 7. Effect of Processor Decomposition on Input

Then the return value from the call to `gin` would be zero in all but the processor which has the top left region which would find the value 1. This feature is extremely useful in highly interactive situations where individual processors need to be indicated by the user.

A final point in connection with input is that the `gin` function must be called loosely synchronously in all nodes. This is really a benefit rather than a restriction since asynchronous calls to an input function are difficult to handle - it is tricky to keep track of which processor wants you to do what, and when! (If the need for an asynchronous input request really exists the `agin` function is available to satisfy it -called just the same way as `gin` and returns the same results, but without any synchronization constraints.)

5 High Level Functionality - Contouring, Clipping, etc.

One of the higher level packages that have currently been added to the *Plotix* system is a contouring package. This system operates at two levels; a simple user interface that allows contouring of arbitrary functions on rectangular regions and a low level interface that allows contouring on arbitrary shaped regions. The latter interface is useful, for example, in contouring functions on spheres is described in detail in Section 9.

Contouring a function on a rectangular domain

The simple contouring package is accessed through the system call `contour` which has the calling sequence

```
        contour(func, gx, gy, levmin, levmax,
                               nlevels, panels)
        double (*func)(), levmin, levmax;
        int gx, gy, nlevels, panels;
```

The first argument is a pointer to a function that actually returns the values to be contoured. It will be called for each pair of integers i, j in the region $0<i\leq gx$ and $0<j\leq gy$ - i.e. on a

rectangular region of size `gx by gy`. This interface differs from others in common use in that the user supplies a function to the contouring program rather than merely an array of values. This option seems more flexible though at the cost of extra CPU time.

Overall `nlevels` contours will be drawn at equally spaced intervals. The arguments `levmin` and `levmax` denote the minimum and maximum contouring levels to be used. (If the user supplies both `levmin` and `levmax` as zero then sensible values will be selected automatically.) Finally the flag `panels` indicates how to draw the actual contours. If a non-zero value is chosen then the contour map will be in the form of filled regions of color. Otherwise only the lines separating the contours will be drawn.

This package is extremely simple. In order to maintain flexibility in all cases it performs NO internode communication. At the boundaries of the processor regions a simple linear fit is performed to approximate the contours as they cross into another processor. Also, for convenience and generality the contouring function assumes that the user coordinate range is set to the system default region from zero to one. If this is not the case then the routines described in Section 9 can be used instead.

Finally note that since this routine uses the multi-mode flushing commands (See section 3.) it must be called loosely synchronously.

Clipping

Also available is a two dimensional clipping system. The function `setclip` defines a rectangular window relative to the user coordinate system against which all line, point, marker and polygon primitives will be clipped. As an example of this system consider the following code segment

```
space(0.,0.,4., 4.);/* Define coordinate system */
setclip(1., 1., 3., 3.);

box(0., 2., 4., 4., 1, 1);

sendplot();
```

Without the call to `setclip` this code would draw a diamond-shaped polygon across the screen. After clipping is enabled the result is as shown in Figure 8. The corners of the

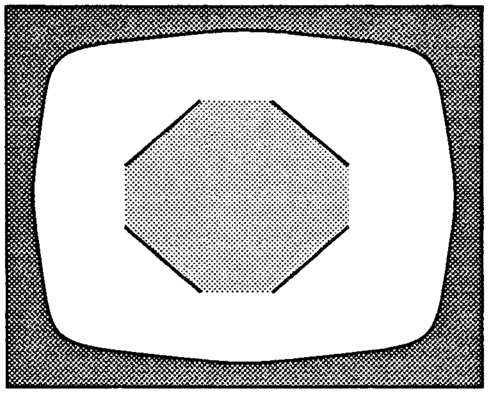


Figure 8. Two dimensional clipping

diamond which lie outside the clipping window have been removed leaving an 'octagonal' shaped region.

Note that all clipping operations are performed in parallel resulting in a system which has rather less overhead than traditional clipping systems.

Currently under development is a three-dimensional modeling package for parallel applications. Hidden line removal is done by means of the Painter's algorithm. We expect to make this system available with a future release of *Plotix.*

6 Colors

Plotix and colors

Plotix runs on many different devices with different capabilities, so difficult questions arise when, for example, a monochrome device is asked to draw a red line. We have tried to address this problem as follows. Each device is either color or monochrome. When *Plotix* is started a color map is in place which is different for lines and for filled polygons. For a monochrome device there are two line colors, black and white, and 8 shades of grey, which are implemented by half-toning. For a color device there are 8 line colors which are the same as the 8 fill colors, these being white, black, red, green, blue, cyan, magenta and yellow. Color numbers outside this range are interpreted modulo 8.

Extending the basic color map

Two functions are supplied to modify this simple color map; `greyscale` and `rainbow`. An example of their use is:

```
        greyscale(9, 40);
        rainbow(41, 104);
```

which has the following effect

Color Device	Color indices from 0 to 8 are as before; the eight standard colors. Indices from 9 to 40 now create a smoothly varying greyscale from white (9) to black (40). Colors in the range from 41 to 104 produce smoothly varying hue with full saturation and full value. The `rainbow` effect is periodic, starting with red, through magenta, blue, yellow, green, cyan, and back to red.
Mono Device	Color indices from 0 to 8 are as before and the `greyscale` call functions exactly as in the color case. The call to `rainbow` is treated as another call to `greyscale`.

These functions change both the line colors and fill colors in the same way.

Cross-hatching

In addition to these 'normal' colors, *Plotix* provides several hatch patterns, which can be accessed by using negative color indices. These appear the same on both color and monochrome devices.

The following code is part of that required to draw shaded three dimensional objects.

```
    #define GREYSTART       9
    #define GREYEND         40
```

```
{
    float shade;
    int polygon, vertex;

    greyscale(GREYSTART, GREYEND);

    for(polygon=0; polygon<n_polygons; polygon++){

/* Use shading model to calculate shade
 * between 0 and 1
 */
        shade = ...

/* Draw the polygon in the correct shade of grey. */

        initpanel((int)(shade*(GREYEND - GREYSTART)
                        + GREYSTART), 0);

        for(vertex=0; vertex<n_vertices[polygon];
                vertex++)
            panelpoint(
                x[vertex][polygon],
                 y[vertex][polygon]);

        endpanel();
    }
```

7 Hardware Dependencies

No graphics package can offer truly "device independent" output because the capabilities of output devices vary so much. The simple graphical model provided by *Plotix* is an advantage in this regard since its capabilities can usually be implemented in full on most devices. Despite this, however, there will be times when the programmer wishes to know what sort of device is in use or wishes to take advantage of a known hardware feature in a non-portable way.

Simple systems are easy to implement

To facillitate this kind of behavior *Plotix* provides two functions: `getplxopt` and `setplxopt` whose behavior depends entirely on the hardware curently in use.

For each potential output device *Plotix* identifies a set of properties which can be manipulated with these two routines. These properties are identified by name through a character string passed to the routine. If the name is recongized for the particular device in use a parameter is either set or returned to the user and the routine returns 0. If the name is not one associated with the current output device nothing is changed and the the routine returns -1.

Hardware dependent "properties" can be manipulated vt Plotix commands

To make this a little more definite let us consider the elementary problem of drawing "distinct" lines. On a color display we might try to do this by using the `color` routine to switch between one of several colors. On a monochrome display, or a color display with too few colors for our purpose, we might instead use the `linemod` routine to display the lines in a different linestyle. One solution to this problem would be to "hardwire" into the application which devices are to be used and set some appropriate switches. An alternative is shown below:

Drawing distinct lines

```
long ncols;       /* Number of line colors supported */

start_graph()
{
     if(openpl(8192, (FILE *)0) < 0) {
         fprintf(stderr,"Failed to start graphics\n");
         return -1;
     }
     if(getplxopt("nlcolors", &ncols) < 0) {
         fprintf(stderr,"No data, assume monochrome\n");
         ncols = 2;
     }
     return 0;
}
```

This code might be used to start up *Plotix*. As well as calling the normal `openpl` function it attempts to find the number of supported line colors using the `nlcolors` property. Note that we take care to deal correctly with the case when this property is not defined for our current device by making the defensive assumption that we are dealing with a monochrome device. If the property is defined the variable `ncols` will be set to the number of supported line colors which can later be used to make a decision about calling `color` or `linemod`.

It is important to note that the call to `getplxopt` comes after the call to `openpl`. This is because many output devices must be initialized before determining the number of supported colors.

Normally `getplxopt` *follows* `openpl`

In a similar way `setplxopt` may be used to set certain operational parameters. One particularly important issue in modern "windowing" systems concerns the repainting of the window when resized or uncovered by the user. These types of system typically require a re-paint procedure. If your application can supply such a thing you can tell certain *Plotix* implementations to use it by calling `setplxopt` with the `redraw` property as shown in the next example.

Re-painting in windowing systems.

```
extern int my_repaint();

start_graph()
{
     setplxopt("redraw", my_repaint);
     if(openpl(8192, (FILE *)0) < 0) {
         fprintf(stderr,"Failed to start graphics\n");
         return -1;
```

```
        }
        return 0;
}
```

Normally `setplxopt` *precedes* `openpl`

Note that this case differs from the previous one in that we call `setplxopt` before `openpl`. This is again typical - the parameters we are setting may have some effect on the way in which the *Plotix* device is initialized. This represents the only exception to the rule which says that the first call to any *Plotix* function should be a call to `openpl`.

Variable types passed to `getplxopt` *and* `setplxopt`

It is important to note that the type of the variable passed to the `getplxopt` and `setplxopt` routines varies according to its use. In the first example we passed a pointer to a `long` to `getplxopt` while the second passed a pointer to a function to `setplxopt`. Other properties may require other combinations. The list of recognzied properties and the types of the associated parameters are shown in Section 10 where device dependencies are discussed for each output device in turn.

8 Examples

8.1 The Interaction between *Plotix* and the `exgrid` tools

Using exgrid with Plotix

As an example of code typically found in *Plotix* programs we present the source code which generates the output shown in Figure 6. To make the code flexible we make use of the `exgrid` tools from *Express* to automatically decompose the actual number of processors to our two-dimensional display surface. This allows us to run the program on *any* number of processors without the need to recompile.

```
/*
 * ParaSoft Corporation: PLOTIX demonstration code.
 *
 * Demonstrate simple usage of PLOTIX primitives and
 * the interaction with the data decomposition tools
 * available in EXPRESS.
 */
#include "stdio.h"
#include "express.h"

main(argc, argv)
int argc;
char *argv[];
{
        struct nodenv cparm;
                /* Run-time information about processors */
        int nprocs[2];
                /* Processors in X and Y directions */
        int recpnum[2];
                /* Coordinates of this node in 2-D mesh */
        double x0, y0;
```

```
                /* Start of processor vport on screen */

    /* Attempt to initialize the graphics system. */

            if(openpl(8192, (FILE *)NULL) < 0) abort(101);

    /* Get the run-time information about the number of
     * processors that we're using.
     */
            exparam(&cparm);
    /* Divide up the processors more or less evenly between
     * the X and Y directions.
     */
            exgridsplit(cparm.nprocs, 2, nprocs);

    /* Now use the exgrid routines to decompose a two
     * dimensional mesh of nprocs[0] by nprocs[1] nodes
     * onto the underlying processor topology
     */
            exgridinit(2, nprocs);
            exgridcoord(cparm.procnum, recpnum);

    /* Now we know which processor we are in the two-D
     * decomposition get the corresponding region of the
     * screen for ourselves.
     */
            x0 = recpnum[0] / (double)nprocs[0];
            y0 = recpnum[1] / (double)nprocs[1];
            vport(x0, y0, x0+1.0/(double)nprocs[0],
                           y0+1.0/(double)nprocs[1]);
            space(0.0, 0.0, 2.0, 2.0);

    /* Finally draw some lines on the page ..... this
     * picture is supposed to end up looking like figure 4.
     */
            move(0.0, 0.0);
            cont(1.0, 1.0);
            usendplot();
    /*
     * NOTE: each node sends a different bit of picture
     */
            closepl();
            exit(0);
    }
```

To run this code we select an output device and use the cubix command as follows

```
        cubix -n4 -Tbgi noddy
```

This command executes the program `noddy` on 4 nodes sending graphics to a monitor supported by the Borland Graphics Interface. Changing the value given to the '`-n`' switch allows us to run on an arbitrary number of processors - one of the virtues of using the `exgrid` decomposition system.

8.2 Other example programs.

Plotix is supplied with a set of example programs for which the C source is available. These are distributed in the `plotix` subdirectory of the main *Express* examples directory. In this directory you should be able to find `makefiles` or similar which will enable you to compile and run these demonstrations. Each is designed to illustrate a particular feature of the *Plotix* system and you are welcome to use them as the basis for your own software systems. Instructions for running the programs can be found on-line in the README files in the same directory.

The codes are

`map`	Trivial program showing the color map supported by the output device
`coord`	Demonstrates the use of the `vport, space` and `aspect` commands as well as the text and symbol drawing routines.
`input`	Demonstrates the use of the cursor in a parallel program allowing selection of particular nodes.
`phases`	A large example which solves a partial differential equation describing phase separation and then displays the result as a contour plot. This example contains most of the plotting routines including the `clip` and `contour` utilities. As well as demonstrating the power of *Plotix* this program is a good example of *Cubix*. Phases was written by Roy Williams of Caltech.
`async`	An example of the "wild and dangerous" asynchronous mode. Each node draws a random number of rectangles of random sizes and randomly erases the screen!

9 The Low Level Contouring System

Contouring in alternative coordinate systems

The contouring package described in section 5 is actually a simple instance of a much more sophisticated contouring subsystem. In general the high level function `contour` should be sufficient for most applications. Occasionally, however, one wishes to contour functions defined on other than rectangular regions - a good example is the contouring of a function defined in polar coordinates; one might define an array, `values[r][theta]`, in which the `r` and `theta` indices correspond to radius and angle. Then the standard function, `contour`, would contour and display this function in a rectangular coordinate system whereas we actually want to display it in the real polar coordinate system. This can be done with the functions `initlevel` and `getpoint`.

The `getpoint` function is the heart of the contouring system. Called repeatedly it returns the next point along the current contour. Note that this is not such a trivial operation - various messy cases arise since, for example, the contour lines of a given height might not be in one piece. There might be ten contour lines for height 150.0 which have to be drawn independently.

Also important is the question of what should be done on the boundary of the region to be contoured. In some cases it is sufficient to merely generate a line from the top of the display to the bottom (for example) but this will not do if we are really interested in plotting in polar coordinates since we should draw a circle instead.

In general, the method of use of these low level functions is to call `initlevel` to start off a new contour and then call `getpoint` repeatedly, plotting the points returned, until an indication is given that there are no more points to draw at this contour level.

The calls to `getpoint` all take the form

```
        stat = getpoint(&x, &y);
```

in which x and y are returned as the `(x,y)` coordinates of the next point along the contour. The return value is extremely important and indicates the following conditions

`stat = 0`	There are no more points to plot along this contour level. Ignore the values of x and y.
`stat = 1`	The point `(x,y)` is a new point on the current contour line.
`stat = 2`	The particular piece of the contour map for this level is finished. Ignore the values `(x,y)` just found and call `getpoint` again. If the next call returns 1 then it is the first point of another disjoint piece of the contour for this level. Otherwise it will return 0 and there are no more points in this contour level.

The `initlevel` function is responsible for initializing each level to be contoured

```
        initlevel(func, nx, ny, level, panels);
```

The arguments are similar to those of the `contour` function. The first is a pointer to a function that returns a double. It is the function that will be contoured. The next two arguments give the range of values for which the function will be called. The `level` argument specifies the height at which this contour is to be calculated. The `panels` argument is somewhat different from that used in `contour` because of the various types of output which might be used. The three allowed values have the following meanings.

`panels = 0`	Designed for simple line plotting. All interior points are returned and the surrounding box is treated as a real rectangle and only its corner points are returned.
`panels = 1`	Designed for polygonal filling. The plot is cut into strips which are contoured separately. The strips are calculated so that the resulting polygonal regions are simply connected allowing hardware polygon fill algorithms to be applied. The bounding box is still assumed to be rectangular and only its vertices are returned.
`panels = 2`	Designed for cases where the underlying surface is not rectangular.

While basically the same as in case 1 the bounding box is also discretized and points are returned all along its edge. This allows the box surrounding the array (which is logically a rectangle) to be mapped into a circle if we are dealing with polar coordinates.

Contouring in polar coordinates

In order to demonstrate the use of these routines we present a sketch of a code that would actually contour an array of polar coordinates. We assume that the array `polars[10][90]` contains values of a function defined in the first quadrant of a polar coordinate system. The first index refers to the radial coordinate and the second to the angular variable - we define the function at every degree in this quadrant. For simplicity we only show code to draw the contour corresponding to the function value 3. Other contours can be added to the image with a simple loop. Finally we assume that we wish to contour the function with filled polygons.

```
/*
 * Example of use of low level contouring functions to
 * display data defined in polar coordinates....
 * sketch code only
 */
#include <math.h>        /* Needed for sin and cos */

radial_contour(color)
int color;
{
     double zfunc();
            /* Function returning contour values */
     double level = 3.0;
            /* Contour height to plot */
     int panels = 2;
            /* Return ALL points - even on boundary */
     int Rrange = 10;
         /* Number of radial coords available */
     int Arange = 90;
         /* Number of angular coords available */
     int status;
         /* Used for return value from getpoint() */
     int start_afresh = 1;
         /* Flag: move or draw to next point ? */
     double r, theta;
         /* Coords returned by getpoint */
     double x,y;
         /* Coords passed to panelpoint */

     initlevel(zfunc, Rrange, Arange, level, panels);

     while((status=getpoint(&r, &theta)) != 0) {
        if(status == 1) {
            /* Another point on the current contour */
            if(start_afresh) {
                initpanel(color, 1);
                start_afresh = 0;
            }
            x = r*sin(3.14*theta/180.);
            y = r*cos(3.14*theta/180.);
            panelpoint(x,y);
        }
        else if(status == 2) {
                /* Maybe start another contour line */
            endpanel();
            start_afresh = 1;
        }
    }
}
```

Note that the only part of this code relevant to polar coordinates is the place where the "warping" takes place and we convert the value returned by `getpoint`, which actually lies in the `(r,θ)` coordinate system into an (x,y) point suitable for use in `panelpoint`.

10 Output Device Characteristics

Specifying output devices at runtime

This section describes the particular idiosyncracies of the various output devices supported under *Plotix* as well as the appropriate switch to give in the `cubix` command line to send graphics to the requested device. These switches are all of the form "`-Tname`" where the "name" is some mnemonic for the required device. Thus, for instance, if we have a program that would normally be executed with the command line

```
cubix -n4 program
```

then one can have it perform graphics on an IBM EGA with the command

```
cubix -n4 -Tega program
```

Each section that follows includes an indication of the appropriate `cubix` switch to invoke the necessary graphics server.

Device dependent property lists and their format.

The following sections also indicate the various supported "properties" as described in Section 7 which are used to manipulate device dependent operations. The following sections also indicate which properties are supported on a particular device and what variable types should be used to access them. In general this information is provided in tabular form with entries of the following type

`nlcolors`	g	pointer to `long`	number of supported line colors

The meaning of the various columns is as follows

- The text string used to identify the proprty.
- Whether the property can be specified to `getplxopt` (g) or `setplxopt` (s).
- The C variable type expected in the second argument to the call.
- A description of the property.

10.1 IBM PC and compatibles, Borland Graphics Interface - "`-Tbgi`"

Problems with the mouse

This version of the system is built around the standard device drivers supplied in the Borland graphics package supplied with the various Turbo languages. It supports a wide variety of PC class systems and graphics cards in a reasonably device-independent manner. All of the features of *Plotix* are supported although we have occasionally observed failures to auto-detect non MicroSoft mouse systems. If you seem to be having problems in this area unload the mouse driver and use the keyboard interface as explained in the previous section on the EGA monitor.

Properties **-Tbgi**

Name	get/set	type	Description
`nlcolors`	g	pointer to `long`	number of supported line colors
`nlstyles`	g	pointer to `long`	number of supported linestyles

10.2 IBM Enhanced Graphics Adapter - "`-Tega`"

This device is supported under XENIX and is obtained by specifying '`-Tega`' on the `cubix` command line. Thus to execute the program `toyland` on 4 nodes with graphical output going to the EGA one might use

```
cubix -n4 -Tega toyland
```

After loading the node program the screen should be blanked and graphical output should appear on the display. Terminal I/O to `stdin`, `stdout` and `stderr` (unless redirected to files on the command line) will appear in a special four line window at the bottom of the screen.

Using the gin command without a mouse

When input is requested with the `gin` command the cursor should appear in the upper left corner of the screen. If you have a mouse then it can be used to move the cursor around. Clicking on any of the mouse buttons terminates the `gin` operation. In the absence of a mouse the cursor can be moved with the arrow keys on the right of the keyboard. Two speeds are available and one toggles between fast and slow cursor motion by hitting the "`Home`" key. Striking any key other than "`Home`" and the arrow keys terminates the input request - the key used is returned as the first argument to the `gin` call.

Resetting the screen after plotting

When plotting is completed or the application finishes the screen *should* revert to its normal appearance - if you called `closepl`. In certain circumstances, however, this may not be the case. Under XENIX one types

```
        norm
```

Notice that no characters will appear on the screen while you are typing these commands - you have to keep on trying until you finally get it right and the screen goes back to its normal state. The XENIX command has the unfortunate side effect of unsetting several terminal characteristics - in particular the delete and CTRL-C keys. As a last resort logging off and on again will clear all problems.

Properties			**-Tega**
Name	get/set	type	Description
`nlcolors`	g	pointer to `long`	number of supported line colors
`nlstyles`	g	pointer to `long`	number of supported linestyles

10.3 SUN system, suntools environment - "`-Tsun`"

This system is built upon the SUN windowing system, "sunview". Each call to `openpl` creates a new window under `suntools` in which graphical operations are performed.

On monochrome displays only two colors are supported by default although eight grey-scales are available through the `greyscale` function. Eight hatch patterns are available with indices -1 thru -8.

On color displays the basic eight colors are supported by default and the full range of colors can be accessed with the `greyscale` and `rainbow` functions. Note that remapping the higher color indices has a strange effects on the basic `suntools` background.

Properties			**-Tsun**
Name	get/set	type	Description
`nlcolors`	g	pointer to `long`	number of supported line colors
`nlstyles`	g	pointer to `long`	number of supported linestyles
`redraw`	s	pointer to function	Function called to re-paint the window when re-sized by user.
`width`	s	`int`	Window width
`height`	s	`int`	Window height

10.4 SUN system, Xwindows - "`-TX`"

This version of *Plotix* is built upon the SUN implementation of MIT's XWindows. Each call to `openpl` creates a new window in which graphical operations are performed.

On monochrome displays only two colors are supported by default although eight greyscales are available through the `greyscale` function. Eight hatch patterns are available with indices -1 thru -8.

On color displays the basic eight colors are supported by default and the full range of colors can be accessed with the `greyscale` and `rainbow` functions.

Properties **-TX**

Name	get/set	type	Description
`nlcolors`	g	pointer to `long`	number of supported line colors
`nlstyles`	g	pointer to `long`	number of supported linestyles
`redraw`	s	pointer to function	Function called to re-paint the window when re-sized by user.
`width`	s	`int`	Window width
`height`	s	`int`	Window height

10.5 PostScript - "`-Tps`"

This option generates standard PostScript suitable for printing on laser printers and similar. Each invocation of the program creates a single ASCII file called `plot.ps` which contains the image. It is important to note that this means that you must be careful to rename files between runs or successive programs will overwrite earlier files.

The color model used is similar to that described in the SUN sections above - 256 colors are supported with the standard calls. On simple monochrome laser printers these will appear in various shades of grey.

The input function "`gin`" returns -1 whenever invoked.

Properties			**-Tps**
Name	get/set	type	Description
`nlcolors`	g	pointer to `long`	number of supported line colors
`nlstyles`	g	pointer to `long`	number of supported linestyles
`landscape`	s	none	Request "landscape" paper orientation.

10.6 AXIS NCUBE systems - "`-Trt`", and "`-Tnat`"

These systems are provided for NCUBE machines. `-Trt` selects the "Real Time Graphics" device - the parallel graphics interface to the NCUBE/10 systems and `-Tnat` the PC-NCUBE graphics system.

In each case the implementation of *Plotix* is complete and supports the full color model. Only one text size is available, however, and it has the disadvantage of overwriting the underlying graphics.

Properties			**-Trt,-Tnat**
Name	get/set	type	Description
`nlcolors`	g	pointer to `long`	number of supported line colors
`nlstyles`	g	pointer to `long`	number of supported linestyles

10.7 Macintosh systems - `no switch`

Since all Macintosh machines are equipped with graphical displays and "mice" no special switches are required to use them under *Plotix*. All features are supported both monochrome or black and white monitors except that the `gin` function can only be triggered from the mouse - no keyboard triggers are enabled. As a result the value returned in the "`button`" parameter is meaningless.

Properties			**"Macintosh"**
Name	get/set	type	Description
`nlcolors`	g	pointer to `long`	number of supported line colors
`nlstyles`	g	pointer to `long`	number of supported linestyles

Debugging Parallel Programs

Using `ndb`, a source level debugger for parallel programs

1 Introduction

Debugging is probably ***the*** most time consuming part of any computing project. Although overall system design and integration may seem more intellectually demanding it is often accomplished much more quickly than the actual execution of the resulting specification - even on conventional and well understood computing systems. When one turns to parallel computers the problems are much worse. While the design phase may seem daunting at first sight, experience shows that the major problems encountered are in debugging. Indeed it has often been said that correcting even the simplest parallel program is as hard as debugging an entire operating system.

Debugging parallel computer codes - a potential nightmare

This difficulty can, in the most part, be traced to an absence of real programming tools. For many years computer experts have been dreaming up, and implementing, sophisticated software packages to help software implementors on conventional machines. Most of these facilities are not yet available for parallel architectures.

A significant step forward is `ndb` - an interactive, symbolic, source level debugger for a distributed array of processors.

Why ndb?

So what?
How will this help?

Source level debuggers provide an extremely simple interface between the code you originally wrote and the nasty machine dependent things that were done to it by the compilers, assembler, linkers and so on. Whereas one used to have to debug by staring hard at a huge dump of hexadecimal numbers one can now perform such elegant operations as "stop at line 23" and "print out the variable `foobar`". You can step through your program slowly and watch what happens to its variables. You can have the program stop whenever a certain variable gets modified. You can examine the sequence of subroutine calls that led to the current state and look at the arguments that were passed to each function.

`ndb` can do all of these things. Furthermore it can do them on arbitrary collections of processors.

This means that you no longer have to poke around in the dark when your parallel program just sits there and stares at you whenever you run it. Now you can get in there and find out why without going through the tedious "edit, compile, run" sequence. Also, `ndb` contains important extensions that are designed specifically for debugging parallel programs - facilities that let you look, for example, at the state of the communication between processors as well as providing completely independent control over each node.

Having described what `ndb` can do there seems no alternative but to describe how you go about it. What follows is an introduction to the various possibilities available. It is not an exhaustive list - for that you should consult the reference manual. However we try to at least demonstrate one flavor of all the commands in the hope that those similar which are left out can be understood. The syntax of `ndb` is, hopefully, quite straightforward. Further, two levels of operation are available. This manual describes what might be called the high level, or source code, interface to the system intended for general use. The syntax in this area is strongly patterned after that of the `dbx` system in use on Sun workstations. Also available, however, is a complete version of the standard UNIX debugger `adb`. Those who find this

dbx + adb in one package

adequate for their needs should be able to dive right in without reading much of this document.

In addition there is an extensive on-line "help" facility. This contains information at all levels from generalities about the way ndb interprets the commands you type to the details of the syntax of a particular command.

The system is implemented under several different operating systems, in conjunction with various compilers and hardware configurations. However, there is only one manual, so in some cases certain commands may be unavailable or may operate slightly differently from the description in this manual. In general differences between various implementations are indicated in the user guides accompanying each *Express* system.

2 ndb in action

This section is designed to give you a rather detailed picture of ndb in action as we debug a "real" program. Since none of the commands have actually been introduced yet do not be too surprised if this makes little sense - it is designed more to give you a picture of what ndb can accomplish. Later, when you have learned the syntax of ndb's commands this section will serve as a useful reference on how you might go about debugging your own programs.

A "master-slave" program with a bug

The basic idea our of program is that of a "master-slave" code in which node 0 distributes work to the other nodes and receives their responses when ready. The basic code to achieve this is shown in Figure 1.

```
/* This code contains a "bug" which will be
 * used to demonstrate the elementary use of
 * the ParaSoft debugger, ndb.
 *
 * Copyright ParaSoft Corporation, May 1989.
 */

#include <express.h>
#include <stdio.h>

float work[1024];
struct nodenv cparm;

main()
{
    int procnum, nprocs;

    exparam(&cparm);
    procnum = cparm.procnum;
    nprocs = cparm.nprocs;

    printf("Here we go!!!\n");
23       fflush(stdout);
24       async(stdout);
25
26       if(procnum == 0) {
27           master(nprocs);
28       }
29       else {
30           slave(procnum);
31       }
32       exit(123);
33   }
```

Figure 1. Sample code - `main` **routine.**

Each node calls `exparam` to find out its processor number and the number of nodes in use. We then modify the status of the `stdout` file stream so that nodes can do I/O independently. This technique is most profitable in this style of program since it allows the "master" task complete freedom to print values irrespective of the behavior of the "slave" nodes.

The central piece of the code is contained in lines 25-30 where the routines that actually do the (fictitious) work are called. Note that node 0 calls the `master` routine which the others invoke `slave`.

```
34
35   master(nprocs)
36   int nprocs;     /* How many nodes are going? */
37   {
38       int i, node, type;
39
40       for(node=1; node<nprocs; node++) {
41           type = 123;
42           printf("Sending work to node %d\n", node);
43           fflush(stdout);
44           exwrite(work, 100*node*sizeof(work[0]),
                          &node, &type);
45       }
46
47   /* Now we poll for results from the slave nodes. */
48
49       node = DONTCARE;
50       type = 124;
51       for(i=1; i<nprocs; i++) {
52           exread(work, sizeof(work), &node, &type);
53           printf("Got result from node %d: %f\n",
                                       node, work[0]);
54           fflush(stdout);
55          }
56    }
```

Figure 2. Sample code - `master` routine.

Using DONTCARE arguments to read messages in any order

The master task merely loops over processors sending a message to each in turn. It then loops again asking for replies. Notice the trick here of using `node = DONTCARE` in `exread`. This allows the "master" to read the answers in any order - the first node to finish its work has its reply read and, in principle, the master could then issue a second piece of work. This is the principle of the "processor farm" in which dynamic load balancing is achieved by sending the faster nodes more work than the slower ones.

The code for the slave routine is

```
58    slave(myproc)
59    int myproc;             /* Which slave am I? */
60    {
61        int node, type, nread;
62
63        node = 0;
64        type = 123;
65        nread = exread(work, sizeof(work), &node,
                                 &type);
66
67    /* Instead of processing the "work"
68     * we just send back a reply to the master.
69     */
70        type = 124;
71        work[0] = 3.14159;
72        exwrite(work, sizeof(work[0]), &node, &type);
73    }
```

Figure 3. Sample code - `slave` routine.

This code runs opposite the `master` routine shown above. Each node reads its message and then makes a simple reply. If this were a real code we could insert some processing after line 65.

Special compilation requirements for using ndb

So, did you spot the error yet? This code has a deliberate and fairly common *Express* bug in it which we'll now find using the debugger. To convince yourself that the program is indeed incorrect we should compile and run it with commands such as:

```
tcc -o code -g code.c -lcubix
cubix -n4 code
```

(Note that the compilation command may be different on your system - consult the introductory guide to *Express* for more details.) We have assumed that the program is (not very imaginatively) stored in a file called `code.c` and we compiled it into the program `code`. Note the specification of the '`-g`' switch at compile time which included information about the source lines and local variables in the output file and associated symbol table.

Executing a program with a bug

If you actually compiled the code and ran it as above the output would be something like that shown below

```
Here we go!!!
Sending work to proc 1
Sending work to proc 2
Sending work to proc 3
Got result from node 1: 3.141590
```

As can be seen we only get a response from the first node. What can be wrong? Given the simple nature of the program you may have enough data now to find the bug but we'll pretend ignorance and use the debugger to find it anyway.

The following is an example of the procedure that one might typically use to catch such a problem. We proceed slowly at first to give a flavor of the commands that `ndb` understands and how you might want to use them and then plunge on to the heart of the matter and the bug itself.

Loading programs in the "stopped" state

We will assume for this discussion that you know how to load the `code` program in the stopped state (i.e., `cubix -P`) and also how to start up the debugger. If not you should consult the introductory guide to *Express* which discusses this issue.

ndb's start-up messages

Assuming that you have figured out how to start up the debugger you should see a message similar to that below.

```
Reading symbol table: 100%
Symbol table: 248 public, 10 local
Sharing 4 nodes, origin at 0 process pid 2
Process has 4 nodes. (Origin at 0)
Node 0>
```

This display contains rather a lot of information. The first line is merely a mechanism for holding your interest while the debugger reads the symbol table from your file - this can be a fairly lengthy process. The second line tells you how many global and local variable names were found. This tells you, for instance, if you forgot to include the '`-g`' switch when compiling some of the files since in that case the number of local variables will be small or zero. The next two lines contain information about the node group allocated to the process you are about to debug. In particular you are told how many and exactly which nodes are in use. This can be useful in diagnosing hardware related problems. Finally the last line is `ndb`'s prompt. It is especially important - it tells you which node or nodes are the target of the commands you type. By default only node 0 is affected as indicated by this prompt.

What is happening on the nodes?

What now? The most elementary, and probably most important, piece of information you can get from the debugger is the answer to the question "What is happening?". To get this type the command **show state** (During this discussion things you type will be highlighted in the text with underlines and will then be shown again as the first line in the debuggers reply.) You will see the following display

```
Node 0> show state
Node 0: Proc state: Breakpoint PC = __PS_strt [?,?]
```

After the program is loaded "stopped" we see the system start-up code

Notice how the first line shows you what you have to type and subsequent lines are the debugger's response. From this display we can learn many things. Node 0 is currently stopped at a breakpoint. This much we might have guessed since we specifically asked *Express* to load the program stopped. Furthermore it is stopped at location `__PS_strt`. This doesn't look quite so friendly but is actually quite harmless. All C programs have some funny code that executes before your `main` routine which is responsible for passing `argc` and `argv` to your code, setting up the environment variables, profiling etc. *Express* is no exception and the funny name for the funny start-up code is `__PS_strt`. The last piece of information in this display is the `[?,?]`. Not, you might say, very informative. In this case you'd be correct but this entry is potentially very important too - it contains the source file and line number of the location of your code. In the case of the system start-up code ndb knows neither piece of information and hence the '`?`' signs. In later examples we'll see more information in this field.

Getting information from more than one node

Now we know all about the activity on node 0. Big deal you might say - what about the others? ndb has a fairly simple mechanism for extending the range of the commands you type - try **on all show st**

```
Node 0> on all show st
Node 0: Proc state: Breakpoint PC = __PS_strt [?,?]
Node 1: Proc state: Breakpoint PC = __PS_strt [?,?]
Node 2: Proc state: Breakpoint PC = __PS_strt [?,?]
Node 3: Proc state: Breakpoint PC = __PS_strt [?,?]
```

Notice two things. We abbreviated the `show state` command to `show st` and prefixed the whole thing with `on all`. The former is just a convenience - you'll end up typing this command so often that it's convenient to have some shorthand for it. The latter is the simplest example of ndb's ability to specify commands on arbitrary sets of nodes. In this case we wanted to see the program state on all nodes so we just asked for it. In the case in question all nodes are in the same place so we haven't learned anything new yet.

Adding breakpoints

To get into more familiar territory we should try to skip over the start-up code and into our own `main` routine. To do this we will set a breakpoint. The command

on all stop in main

will give you the following message

```
Node 0> on all stop in main
Inserting breakpoint at main+0x7 .... ..... done
```

This command inserts a breakpoint in your program at the first source code line of the named function. ndb's response is rather more formal - it tells you the actual machine address of the breakpoint. Notice that we used the "`on all`" construction to again override the default target node - if we hadn't used this prefix the breakpoint would only have been inserted in node 0.

Restarting from a breakpoint

To proceed from the breakpoint issue the **on all cont** command.

```
Node 0> on all cont
Node 0>
```

An important difference between ndb and sequential debuggers

Hm. No response from this one. What has actually happened is that ndb has started up the nodes indicated which are now happily travelling towards the breakpoint in main. In contrast to debuggers for sequential computers ndb doesn't wait for the breakpoint to be hit but returns immediately to you for more input. This may seem inconvenient at first but we will see the reason a little later - in the meantime you can perhaps take comfort in the fact that ndb will mimic the conventional behavior if you ask it to - see the manual section about the ndbenv command for details.

Changing the default processor set

In order to see that something really did happen when you typed the cont command we should look at the program state again, yes? Correct. However you're probably already bored with typing on all in front of everything so its time to try something new. Type **pick all** to see the following

```
Node 0> pick all
All>
```

As you've probably guess this command merely changes ndb's internal idea of the current set of target nodes. From now on all commands which are not prefixed by on will be directed to all processors. Notice how the prompt changes to indicate this fact. Now we can type merely **show st** to get the information we need.

```
All> show st
Node 0: Proc state: Breakpoint PC = main+0x7 [code.c,18]
Node 1: Proc state: Breakpoint PC = main+0x7 [code.c,18]
Node 2: Proc state: Breakpoint PC = main+0x7 [code.c,18]
Node 3: Proc state: Breakpoint PC = main+0x7 [code.c,18]
```

Listing the source code

This display should be fairly familiar by now. From it we can see that all the nodes are now at the breakpoint we inserted at the top of main. Furthermore the last two fields of each line now contains useful information about the source file and line number of the current position. To extract even more information try typing **list** to get the following

```
All> list
  18: exparam(&cparm);
  19: procnum = cparm.procnum;
  20: nprocs = cparm.nprocs;
  21:
  22: printf("Here we go!!!\n");
  23: fflush(stdout);
  24: fasync(stdout);
  25:
  26: if(procnum == 0) {
  27:      master(nprocs);
```

This probably looks a little more familiar. Basically the "list" command, when issued with no arguments, prints out source code from the position that ndb thinks you've reached. As you might guess this is actually a little less well defined on a parallel computer than a sequential one since each of the nodes could be in a different place. ndb tries its best to second guess you as you debug - you can easily fool it if you try but in the case we're looking at all the nodes are in the same place so it's a piece of cake!

Single stepping

Before we really get going on the bug we could try a few more simple commands. Go back to node 0 by typing **pick 0**

```
All> pick 0
Node 0>
```

and then single step the program by typing **next**.

```
Node 0> next
  19: procnum = cparm.procnum;
Node 0>
```

As you might guess the program has gone on through a single line of source code and has printed out the line which would be executed next. An important point to notice is that the line of source code executed actually contains a function call exparam. (Look in the output from the previous list command to check.) This is the powerful feature of the "next" command - it actually steps over function calls. An alternative is provided to let you enter each routine as it comes - step.

Let's single step a few more times until the important variables procnum and nprocs have been assigned and then check their values. Type **next** again

```
Node 0> next
  20: nprocs = cparm.nprocs;
Node 0>
```

and **next** again

```
Node 0> next
  22: printf("Here we go!!!\n");
Node 0>
```

Displaying variables

Now we can check that everything is going according to plan by typing out the values of our variables with **print procnum, nprocs**

```
Node 0> print procnum, nprocs
Node 0:   procnum = 0
          nprocs = 4
Node 0>
```

A common confusion - parallel debugging is different from sequential debugging

So now we know that there are four processors and node 0 really thinks it's node 0. It's good that this worked since it would otherwise mean that *Express* was confused.

To continue further try typing **next** again.

```
Node 0> next
  At least one node has stepped into a routine which
  contains no line number information. To step through
```

```
         this code use "stepi" or "nexti" instructions
      Node 0>
```

This is finally the explanation of our earlier remark about why the "`cont`" command returns immediately rather than waiting for a breakpoint to appear. What happened is that we tried to single step node 0 through a call to `printf`. However `stdout` is still in "single-mode" (the default) which implies a synchronization between all processors. (If this comment is gibberish you might want to investigate the *Cubix* documentation a little - otherwise just take my word for it that this call to `printf` requires synchronization.) Since the other three processors are currently still at the first breakpoint in `main` they can hardly synchronize with node 0 at this point and `ndb` eventually gives up waiting - hence the message. `ndb` has found that node 0 is now executing code somewhere in the middle of *Express*' internals where there are no source line numbers. Furthermore node 0 cannot make any progress until the other three nodes catch up with it to synchronize. As a result the machine is temporarily "hung". If `ndb` were to wait for the situation to resolve itself you would now be staring at a dead computer. This is the same reason that `ndb` doesn't wait for breakpoints when you type "`cont`" - if there are many processors, each executing code in different places, the machine could lock up in some internal wait state (looking for a message, for example) before getting to another breakpoint.

Resynchronizing the processors

To get everybody back in sync again let's set a breakpoint in the other nodes at a place where they could potentially get back together. Type **pick 1-3** to select the other three nodes.

```
      Node 0> pick 1-3
      Array>
```

The prompt is now a little less informative but represents the fact that the particular collection of nodes 1, 2 and 3 is not one that `ndb` recognizes by name.

Listing code by name

Having selected the correct group of nodes let's look for a candidate for a breakpoint. Try typing **list main** to see the appropriate source code.

```
      Array> list main
        10:
        11: float work[1024];
        12: struct nodenv cparm;
        13:
        14: main()
        15: {
        16:     int procnum, nprocs;
        17:
        18:     exparam(&cparm);
        19:     procnum = cparm.procnum;
```

This lists out ten lines of code centered on the named function. Since we didn't get far enough to see anything interesting we can continue listing the source by merely typing **list**

```
      Array> list
        20:     nprocs = cparm.nprocs;
        21:
```

```
22:     printf("Here we go!!!\n");
23:     fflush(stdout);
24:     fasync(stdout);
25:
26:     if(procnum == 0) {
27:         master(nprocs);
28:     }
29:     else {
30:         slave(procnum);
```

which now contains more likely code. Given the discussion above we believe that node 0 is now in limbo somewhere within the guts of the `printf` call. We could try to stop the other nodes just after it at line 23 with the command **stop at 23**

```
Array> stop at 23
 Inserting breakpoint at main+0x22.... ..... done
Array>
```

As before this code tells us where, in the machine code, the breakpoint has been set and we can continue to it by typing **cont**.

```
Array> cont
Array>
```

Note that ndb again offers no response but immediately returns for more input. If we look in the other window, however, where the program we're debugging is running we would now see the line

```
Here we go!!!
```

generated by the call to `printf`. (By "window" we mean the piece of the display where output from the running application is directed. This may be another window or merely another piece of the one the debugger is using.) This might lead us to suspect that all the nodes have now reached line 23 - verify this by typing **on all show st**.

```
Array> on all show st
Node 0: Proc state: Single step PC = main+0x22
          [code.c,23]
Node 1: Proc state: Breakpoint PC = main+0x22 [code.c,23]
Node 2: Proc state: Breakpoint PC = main+0x22 [code.c,23]
Node 3: Proc state: Breakpoint PC = main+0x22 [code.c,23]
Array>
```

This display confirms our thoughts except that node 0 is not in the "`breakpoint`" state but rather "`Single step`". This is a result of our earlier attempt to single step over the `printf` call. At the time when ndb finally gave up the attempt to finish our single step request node 0 was in the "`Single step`" state and since we've done nothing new to it since it still is.

Serious debugging Now we've played around with some of ndb's commands and have seen some of the ways in which it differs from familiar debuggers on sequential computers it is time to get down and find the bug we've been looking for all along. Since we know that the "master" task gets a message from node 1 but none of the others we should get the program as near as possible to that state and then look at some variables. To find a suitable place let's look at the code in `master` with the command **list master**

```
Array> list master
  32:          exit(123);
  33:      }
  34:
  35: master(nprocs)
  36: int nprocs; /* How many nodes are participating */
  37: {
  38:      int i, proc, type;
  39:
  40:      for(proc=1; proc<nprocs; proc++) {
  41:          type = 123;
Array>
```

and then **list** again for a few more lines

```
Array> list
  42:          printf("Sending work to proc %d\n", proc);
  43:          fflush(stdout);
  44:          exwrite(work, 100*proc*sizeof(work[0]),
               &proc, &type);
  45:      }
  46:
  47: /* Now we poll for results from the slaves. */
  48:
  49:      type = 124;
  50:      proc = DONTCARE;
  51:      for(i=1; i<nprocs; i++) {
  52:          exread(work, sizeof(work), &proc, &type);
```

The second half of this listing contains the code where node 0 picks up the results from the other nodes. If we put a breakpoint at line 52 we'll be able to "watch" the messages arrive and check that they are O.K. To do this type **on 0 stop at 52** (Remember we have the prompt "`Array>`" indicating that the default target would have been nodes 1 through 3.

```
Array> on 0 stop at 52
 Inserting breakpoint at master+0x4b .... ..... done
Array>
```

To find a good place to stop the "slave" tasks let's look at their code with **list slave**

```
Array> list slave
  55:      }
  56: }
  57:
  58: slave(myproc)
  59:      int myproc; /* Which slave am I? */
```

```
    60: {
    61:     int proc, type, nread;
    62:
    63:     proc = 0;
    64:     type = 123;
  Array>
```

and then **list** for more information.

```
  Array> list
    65:     nread = exread(work, sizeof(work), &proc,
                                &type);
    66:
    67: /* Instead of processing the "work" buffer.
    68:  * we just send back a reply to the master.
    69:  */
    70:     type = 124;
    71:     work[0] = 3.14159;
    72:     exwrite(work, sizeof(work[0]), &proc, &type);
    73: }
   Array>
```

To match the breakpoint in node 0 that we just added try stopping the slave tasks at line 72 with the command **stop at 72**. Notice that this time the default target set of nodes is correct so we need no prefix.

```
  Array> on 1-3 stop at 72
   Inserting breakpoint at slave+0x32 .... ..... done
  Array>
```

Now that we've finally got the whole thing set up let's let the nodes go to work by typing **on all cont**.

```
  Array> on all cont
  Array>
```

Not a staggering response from ndb considering the work we put in to set it up but if we look in the "window" where the program is actually running (Same comments apply as before!) we'd now see the output

```
  Sending work to proc 1
  Sending work to proc 2
  Sending work to proc 3
```

which is generated by node 0 through the `printf` call at line 42.

To find out where we are now try **on all show st** again

```
  Array> on all show st
  Node 0: Proc state: Breakpoint PC = master+0x4b
  [code.c,52]
```

```
Node 1: Proc state: Breakpoint PC = slave+0x32
[code.c,72]
Node 2: Proc state: Breakpoint PC = slave+0x32
[code.c,72]
Node 3: Proc state: Breakpoint PC = slave+0x32
[code.c,72]
Array>
```

Looking at local variables

Amazing - it's all just as we thought it would be; node 0 is stopped at line 52 waiting for the slaves to send it data and the other nodes are poised at line 72 on the verge of responding. To check that things are about to go the way we want type **dump** to see

```
Array> dump
 Node 1: slave+0x32 [code.c,72] (myproc= 1) at 0x800d8f28
              myproc: 1
              nread: 400
              type: 124
              proc: 0
 Node 2: slave+0x32 [code.c,72] (myproc= 2) at 0x800d8f28
               myproc: 2
              nread: 800
              type: 124
               proc: 0
 Node 3: slave+0x32 [code.c,72] (myproc= 3) at 0x800d8f28
               myproc: 3
              nread: 1200
              type: 124
               proc: 0
Array>
```

This display contains much information. The first line in each block shows the current position (which we already knew) together with a list of the arguments passed to the current function and the physical memory address at which the arguments are located. The next few lines list the local variables of the named function, displayed according to their variable types. For us the most important information is that `proc` and `type` really do have the values we wanted them to have.

To actually send the messages back to node 0 the simplest method is to single step the slave nodes over their calls to `exwrite`. Type **next**.

```
Array> next
Array>
```

[illegible]

Displaying the message queues

Note that ndb hasn't displayed the "next" line of source code as it did in the earlier cases. This is because only one node was affected in the previous case making it easy for ndb to figure out the appropriate line to print. In this case we are stepping three nodes and although all actually go to the same place ndb doesn't waste the time it takes to figure that out and print anything.

So, did anything happen? What we actually wanted was to send three messages to node zero, one from each slave. To see if they actually went anywhere type **on all show q**.

```
Array> on all show q
 Node 0:
  src 1 type 124 pack 1 length 4 (d0 0f 49 40) @0x800e453c
  src 2 type 124 pack 1 length 4 (d0 0f 49 40) @0x800e4968
  src 3 type 124 pack 1 length 4 (d0 0f 49 40) @0x800e4d94
 Node 1: No messages
 Node 2: No messages
 Node 3: No messages
Array>
```

This all looks too good to be true. The display indicates that node 0 has received three messages, one from each slave node, of type 124. Furthermore each message is 4 bytes long just as we would expect. (If we get really picky we could use the address shown at the end of each line to actually investigate the contents of each message. We'll leave this as a "exercise for the dedicated reader!)

So we now know that the bug isn't on the slave nodes - they did everything we asked them to do. Let's switch back to node 0 with **pick 0**

```
Array> pick 0
Node 0>
```

and check out the variables in the `master` subroutine by typing **dump master**

```
Node 0> dump master
     master+0x4b [code.c,52] (nprocs= 4) at 0x800d8f28
               nprocs: 4
               type: 124
               proc: -1
               i: 1
```

The magic value DONTCARE

Does this look reasonable? We're about to call `exread` with `type` set to 124 and `proc` to -1. The `type` variable obviously matches properly and a little thought would remind us that we asked to read from processor `DONTCARE` which probably has the magic value -1. It's all O.K.

Type **cont** to pick up the first message.

```
Node 0> cont
Node 0>
```

ndb is silent as usual when we continue or single step but a glance at the program's output shows the text

```
Got result from node 1: 3.141590
```

which shows us that we really did pick up the message from node 1. To confirm this we can look at the message queues again with **show q**

```
Node 0> show q
  src 2 type 124 pack 1 length 4 (d0 0f 49 40)@0x800e4968
```

```
    src 3 type 124 pack 1 length 4 (d0 0f 49 40)@0x800e4d94
  Node 0>
```

Again this all looks just as it should. The message from node 1 has gone and that from node two is ready for delivery. Looking at the local variables again with **dump** shows us

```
Node 0> dump
   master+0x4b [code.c,52] (nprocs= 4) at 0x800d8f28
            nprocs: 4
            type: 124
            proc: 1
            i: 2
```

which looks O.K -`type` is still 124 and we're about to read from `proc = 1`.

Pardon?

Why is `proc` equal to 1? We just read the message from node 1 and should now be reading from another processor - in fact we should be reading from processor DONTCARE because we really don't care what order we get the replies in.

This is the bug.

Be careful with DONTCARE arguments

What we've forgotten is that the call to `exread` in line 52 overwrites the pointer to `proc` with the source of the message that was actually read, in this case node 1. Then on the second iteration of the loop we omitted to reassign `proc` to DONTCARE and so it again tried to read from node 1. Since node 1 isn't talking to us anymore the program hangs waiting!

To fix the problem take line 49 where we assign `node = DONTCARE` and insert it before the call to `exread` in line 52 and all will be well - sometimes there really is a happy ending!

The correct version of the lower loop in the `master` routine is shown in Figure 4.

```
46
47 /* Now we poll for results from the slave nodes. */
48
49  type = 124;
50  for(i=1; i<nprocs; i++) {
51       node = DONTCARE;
52       exread(work, sizeof(work), &node, &type);
53       printf("Got result from node %d: %f\n",
              node, work[0]);
54       fflush(stdout);
55    }
56 }
```

Figure 4. Sample code - corrected bug.

3 Getting Started

By default compilers do not produce information about the local variables or line numbers in your source code. This means that ndb doesn't know about them either. In some cases this doesn't matter too much - for example you can still tell where your program crashed and what sequence of subroutine calls led to the disaster. You can also inspect the values of external variables. However, if you want to delve deeper into a mysterious bug then you must recompile your code with special switches which force the compilers to generate the relevant information; the introductory guide to using *Express* on your system has details of the appropriate commands.

Compiler switches and managing without them

The next problem to be addressed is how to actually run ndb. Detailed discussion of this is left to the individual introductory guides supplied with versions of *Express* but we may observe some general points. Basically there are two scenarios. In the first the code has crashed in mid-flight and you want to go and look around inside the nodes to see what happened. In this case you need to know the process ID of the dead program. This is most easily found out on UNIX systems with the ps command. Armed with this information you type

Two methods of running ndb

```
ndb -p process_id -n nodes program_name
```

where process_id is the number obtained from ps, program_name is the name of the node program to be debugged and nodes is the number of nodes on which the program is running. ndb responds by reading in the symbol table from the node program and telling you how many local and public names were found. The former are variables local to subroutines while the latter include external variables, named common blocks and subroutine/function names. If everything you typed on the command line was correct you should now see the ndb prompt:

```
Node 0>
```

The prompt you get back from ndb is rather important since it tries to tell you which nodes are currently being debugged. We will return to this topic later - all that you need to know at this point is that the above prompt means that ndb found some nodes to debug.

The second scenario under which ndb is used is where the user wants to start the program under ndb's control and then run it with breakpoints set and so on. In this case you just type

```
ndb -n nodes program_name44
```

where the arguments mean the same as before. In this case ndb will read the symbol table and then issue the prompt

```
ndb>
```

Running programs from within ndb

which means that no nodes are currently in use. Now in order to run the program you issue a run command such as

```
run cubix -d 3 toyland <input
```

which is executed just as if you had typed it to the shell - as can be seen you can still redirect I/O with the regular constructions. You will probably see some start-up messages and then the ndb prompt will change to

```
Node 0>
```

showing that ndb now recognizes that your program has allocated some nodes.

4 Sets and Prompts

In order to debug a parallel program it is necessary to execute commands on different nodes - for example you might want to see the value of the variable `ivalue` on nodes 3 and 4. In order to do this sort of thing `ndb` includes a "set" concept; a "set" is a collection of processors to which `ndb` commands are directed. `ndb` keeps track of a "current set" and also supplies commands that let you change the current set. Every time `ndb` prompts for more input it tries to show you its idea of the default. For sets containing single nodes this is just the processor number as seen in the previous section. Another simple set is that containing all the processors which has the prompt

Parallel debugging means being able to debug more than one processor at a time

```
All>
```

Three commands are available for manipulating sets. The simplest is `pick` which changes the current set. For example, you might type

```
pick 0-3, 6, 9-13
```

which changes the default set to that containing nodes 0, 1, 2, 3, 6, 9, 10, 11, 12 and 13. All further commands will be executed in turn on each of these nodes. Having changed the current set in this way the prompt changes to

```
Array>
```

which is `ndb`'s way of saying that it can't figure out any logical reason for choosing such a strange collection of nodes.

Having changed the current set in this way it is occasionally useful to execute commands on different sets without changing the default. To do this one just uses the `on` command as a prefix to the instruction you want executed. For example

```
on all show state
```

shows the current state of all processors without changing the default. Any legal set specification can be used in any of these commands.

The final possibility is that of user defined sets. In the first example we picked a rather bizarre collection of nodes to debug. If it turns out that such sets arise with any frequency it is rather inconvenient to have to type in all the node numbers every time. For this reason you can define arbitrary sets of nodes by using the `setdef` command.

User defined node sets

```
setdef 0, 1, nof 4, even
```

This command defines a complicated set containing nodes 0 and 1, the hypercube connected neighbors of node 4 (indicated by the mysterious `nof`) and all processors whose `procnum`'s have even parity (i.e., an even number of 1 bits in their processor numbers). `ndb` remembers this set and responds with, for example

```
Defined set 23
```

which can be used in further set specifications; one could now type

```
pick set 23
```

to change the current set or even use `set 23` in further `setdef` commands.

The currently available user defined sets can be listed by executing the `show sets` command.

5 Programs that need input

Directing terminal I/O to the parallel computer

Most sequential debuggers adopt the view that once the user program is running all further terminal input should be sent directly to it. Thus, for instance, a program that reads in input values will execute more or less normally. In a distributed environment the situation is not quite so clear - it may be necessary to have some nodes progress while others are still at breakpoints so that message traffic can be analyzed more carefully. For this reason ndb does not pass on terminal input to the user process until you issue the command

```
io
```

to which ndb responds with the message

```
Redirecting terminal input to user process
```

From this point on everything you type on the keyboard is sent as input to your program rather than the debugger. At any time it is possible to return control to ndb by using the keyboard interrupt sequence - usually `CTRL-C`. Note that you can ALWAYS use this sequence to stop whatever ndb is doing and return control to the prompt. In particular you might want to curtail ndb's rather prolific output when certain commands are given. If your program later requires more input you can type `io` again.

If your system supports multiple windows the best way to use ndb is probably to start-up the application "stopped" in one window and then attach ndb to it with the `-p` switch from another window. In this way you will be able to satisfy I/O requests to the application by merely typing in its window.

6 Examining the process state

The most elementary and important ndb command

Often the first thing that users want to know about their program is where it is and what it is doing. Several commands are available for doing this. The simplest is `show state` (or equivalently `show st`). So, for example, the command

```
on all show st
```

might produce the output

```
Node 0: Proc state: Breakpoint PC = _start [start.c, ?]
Node 1: Proc state: Idle PC = main+0x12 [foo.c,10]
Node 2: Proc state: Idle PC = main+0x12 [foo.c,10]
Node 3: Proc state: Running PC = error+0x32
                  [catastrophe.c,128]
```

There is quite a lot of information in this display. The `Proc state` field shows what condition the various nodes are in. This is where serious bugs and/or hardware errors can be found. States such as `Breakpoint`, `Single step` and `Running` are self-explanatory. An `Idle` processor is one which is waiting for a communication call to be completed. (On some systems a node waiting for communication will appear to be

RUNNING.)

The next field describes, symbolically, the instruction currently being executed. If this instruction lies within some user subroutine then that name should appear (modulo a preceding underscore) together with some hex offset. In certain cases the operating system will be active which should be obvious from the unusual subroutine name or from its complete absence. In this latter case the PC will be indicated by some hexadecimal constant.

Information related to the source code

The last field displays source code information. The format is

```
[Source file, line number]
```

where the first value is the name (as known to the compiler) of the source file which contained the code. In most cases this will be a real file name but occasionally one might see names without any suffix or translated to upper case. In any event it should be obvious which source file is indicated. The second value is the line number which contains the current instruction. If the object file were compiled with the appropriate compiler debugging switches (see Appendix C) this value will be a line number in the source code. Otherwise the value '?' will appear meaning that ndb doesn't know what line number is involved. So, for example, the output

```
[contour, 123]
```

Name conversions, '_' and capitalization

indicates line 123 in file contour.c. Typically Fortran names end up converted to upper case so that one might see

```
[CONTOUR, 123]
```

to indicate the file contour.f as the culprit.

DEAD nodes

Before discussing the other features of ndb which allow access to the user source code it is interesting to note that the show state command is useful in detecting dead processors. If no response is received from a node for 5 seconds the message DEAD NODE is displayed and ndb moves on to the next. This is a useful feature since it allows one to figure out which nodes have been trashed without interfering with, or having to restart, ndb. One can then construct a user defined set (discussed in section 4) which doesn't include the non-functioning nodes and continue.

Tracking the source code

While you are issuing commands ndb attempts to "second guess" the source file and line number that you might want to see next. In particular the show state command already discussed and the stack backtracing features described in the next section determine ndb's idea of the current position. The command

```
list
```

then prints out this and the following 9 lines of source code. So if show state indicated [contour, 19] then list might, for example, come back with

```
19: for(i=0; i<MAX_SIZE; i++) value[i] = 0;
20:
21: /* The following code section communicates results
22:    between processors. The excombine function gets
```

```
   a sum over all nodes.
*/

excombine(&total, gbl_add, sizeof(total), 1,
         ALLNODES, NULLPTR, &type);
printf("Global sum = %f\n", total);

```

Listing source code Further `list` commands print out consecutive lines of code.

As well as listing lines starting at the current point several other possibilities are available. For example,

```
list 10, 30
```

lists lines 10 through 30 inclusive from the current file.

```
list -5
```

writes out 10 lines starting 5 before the current position, and

```
list contour
```

prints out the first ten lines of the function `contour`.

This last option shows one way of overriding `ndb`'s idea of the current position. You may print out specific functions by name or merely change source file by saying

```
file catastrophe.c
```

in which case further `list` commands start at the top of the newly named file. Similarly one can switch to a new function by typing

```
func error
```

which makes further `list` commands start at the function `error`.

String searching A final option available on some systems is the ability to search for expressions within files. Thus, if you can't remember where in a file the string `combine` occurs you can just search for it with the command

```
/combine
```

which starts at the current point and searches forward through the file for the given regular expression. One can similarly search backward by replacing the / with ?

7 Tracing back through subroutine calls

Having discovered where the program is currently executing it is usually wise to find out how it got there. In order to do this `ndb` has a command that lets you trace the sequence of subroutine calls which led to the current point. Thus one might type

```
on 1,2 where
```

to yield the output

```
Node 1:
 contour+0x12e [contour.c,12] (0x400,0x44230434)
```

```
@0x77434
 main+0x94 [noddy.c, 8] (0x1, 0xAC10) @0x77660
 start+0x38 [START.c,?] ()

Node 2:
 contour+0x12e [contour.c,12] (0x400,0x00000000)@
0x77434
 main+0x94 [noddy.c, 8] (0x1, 0xAC10) @ 0x77660
 start+0x38 [START.c,?] ()
```

The stack trace

The information here consists of the names of all subroutine calls together with the instruction which will be executed whenever the called routine `return`'s and the first five arguments to the function displayed in hex. (If a function has more than five arguments the first five are shown followed by periods, . . .) The final item is the memory address of the arguments. This is useful if either more than five arguments were passed to a function or some of them are floating point values. Note also that the second column of the above display contains the same source code/line number information that was available in the `show state` command.

Programming languages

Note that in certain cases it may be necessary to tell `ndb` whether the program you are debugging was written in Fortran or C. This is due to incompatibilities between compilers which may not be apparent to `ndb`. In this case use the commands

```
ndbenv C
```

and

```
ndbenv F
```

to switch between languages.

> **WARNING:** The stack tracing commands work best when the node under investigation is stopped - either at a breakpoint or with some error. This is because these operations require several messages to be sent to the node being traced. If the node is actually executing code the stack will look different upon receipt of each message leading to inconsistent and possibly misleading results from `ndb`. In particular references may be made to out of memory locations.

By default the stack tracing commands "unpeel" the stack through eight levels of subroutine calls or to the top level function, whichever comes soonest. This can sometimes be a nuisance, especially when large numbers of nodes are in use when it might be better to merely display the last level of calls in order to avoid being submerged beneath a vast amount of data. To do this one uses the extended form of the "`where`" command which has the form

```
where 3
```

The isin command simplifies some requests

which displays the last three levels of a program. One can use the same method to display more than the default number of stack levels.

A particularly useful command in this context is `isin` which only displays the calling sequence for a particular function. You can, for example, say

```
on all isin contour
```

to display the arguments given to the `contour` function in each node. This is a useful command since it reduces the amount of data produced by the full `where` command and also indicates which nodes have not actually called the named function.

8 Examining Data

Having found out where the program is dying or hung one typically wishes to print out data values. In ndb this is accomplished with the `print` command. To examine the variable `ultimate`, for example, you type

```
on all print ultimate
```

to which the response might be

```
Node 0:
                _ultimate = 42
Node 1:
                _ultimate = 42
Node 2:
                _ultimate = 42
Node 3:
                _ultimate = 42
```

Overriding variable type information

showing the value to be 42. Each variable will be displayed in a format appropriate to its type as determined from the program - you can print structures, arrays and so on. If you wish to override the default type in which a variable is printed the lower level formatting command are available and are described in the accompanying reference manual.

Scope rules for variable names

In discussing the use of variable names it is obviously necessary to be able to distinguish between various variables of the same name. In order to resolve these conflicts ndb follows the following procedure when translating variable names.

1. Look for a local variable in the current function; the one to which the program counter register is pointing or that named in the most recent `func` command.
2. Look for an external variable.

As can be seen no attempt is made to find a local variable in other than the present function. This can be explicitly requested by specifying a full name in the form `function'name`. Thus to display the variable `i` in the routine `contour` one might need to say

```
print contour'i
```

Note that this construction will fail if the function `contour` cannot be found in the stack backtrace - i.e., this function must have been called and not have returned.

Local variables

Another useful command for displaying data is `dump` which displays all local variables of the named function, or the current function if no argument is given. So, for example, the command

```
dump contour
```

will display all the local variables of the function `contour` together with its arguments and a description of the place from which it was called. The variables are all displayed in hex, but can easily be redisplayed in other formats using the tricks discussed in section 10.

Comparing node memories

A final command useful in manipulating data is `compare` which allows you to discover where two or more nodes have different values. Typing

```
on 0-2 compare array, 128
```

in turn compares 128 bytes of memory starting at `array` on the first three nodes. The length argument can be omitted for a default of 64 bytes.

9 Running programs - Breakpoints

As well as diagnosing program failures in a post-mortem mode it is possible to use `ndb` to trace the execution of a program and control it by setting breakpoints. As execution reaches a breakpoint the node is stopped and its state is then available to be studied using the commands already described. Execution can then be continued from the breakpoint either a single instruction at a time or until another breakpoint is found (or else the program terminates in some way).

Three types of breakpoint

Breakpoints are inserted into the code with the `stop` command of which there are three varieties. The first simply names a function and the breakpoint is inserted before the first instruction of that function. For example

```
on all stop in catastrophe
```

puts a breakpoint in every node at the first line of the function `catastrophe`. Secondly one can say

```
stop at 23
```

in which case the breakpoint is placed at line 23 of the current source file.

The final type of breakpoint is rather different and is obtained by giving the name of a program variable, such as

```
stop error_flag
```

in which case the program continues until the named variable is altered in which case execution immediately stops. This form is especially useful when it is known that memory is being corrupted and the affected memory location is known but not the instruction that causes the problem. (This form of breakpoint may not be available in some implementations of `ndb`. Further, certain versions have to perform the memory checks in software so that programs may run extremely slowly with this feature enabled.)

Note that breakpoints are inserted on nodes in the current set just as for any other command - it is quite legal to have a breakpoint in some nodes and not in others, or indeed to have different breakpoints in different nodes.

The current list of breakpoints is obtained with the

```
show breakpoints
```

or

```
show br
```

commands. The breakpoint locations are listed together with a string of ones and zeros in which a 1 in position j indicates that the listed breakpoint is active in node j. An index number is also associated with each breakpoint which can be used in deletions.

Removing breakpoints

Breakpoints are deleted with the `delete` command in the obvious way, e.g.,

```
delete 2
```

deletes the breakpoint whose index number is 2.

Having stopped at a breakpoint all of the commands of `ndb` are available to inspect the state of the node. Execution can be continued either one instruction at a time (with the `step` and `next` commands) or continuously with the `cont` command. The difference between the single stepping commands concerns their action when the next line contains a function call. `step` enters the function, stopping at its first source line, while `next` steps over it and stops at the next line of the current function.

Different behavior of ndb and sequential debuggers

Note that the `cont` command functions differently from that encountered in sequential debuggers in that it, by default, leaves `ndb` in control of the terminal. Thus one can type

```
on even cont
```

to allow the even parity processors to continue and then still use the debugger to issue commands to, for example, monitor the message queues on the odd parity processors. In a sequential environment the debugger would halt until the program being debugged stopped or hit another breakpoint when one could again enter commands. The default situation seems more useful in a distributed environment but the sequential behavior can be made the default with the `ndbenv` command by typing

```
ndbenv wait
```

or

```
ndbenv w
```

While in the "wait" mode `ndb` continuously polls the nodes in the set which have been `continued` until all have stopped at which point the user is informed and can again enter commands to `ndb`. While the polling is going on `ndb` displays a message indicating which node it is currently examining which can often be used to detect "hung" nodes or those with large workloads. In any case control can always be given back to `ndb` by typing the interrupt command (usually CTRL-C).

10 Using `ndb` as a calculator - format modification

It is occasionally annoying that `ndb` chooses to display things in hex rather than, shall we say, single precision floating point. It is, however, possible to make `ndb` perform the necessary transformations by using the "calculator" mode. Any command line of the form

```
expression = format_specifier
```

causes the translation of the expression on the left to the format specified on the right. (The format specification characters are the same as those used in displaying data). The `expression` can be almost any expression containing integer values and variable addresses with operators given by their C symbols. (i.e., left shift is << and logical `and` is `&`)

Forcing type conversions

As an example of this mechanism consider the previously mentioned gripe. Having displayed a function call traceback with the `where` command one finds that the value specified as an argument was `0x44230434` where one was really expecting a 4 byte floating point value. To translate this one types

```
0x44230434 = f
```

which yields

```
Constant: 652.06567
```

as the true value.

If you can't put your hand on your calculator `ndb` is also capable of general arithmetic so that, for example,

```
0x123 + 4*(1 << 3 -5) ^ 0200 = D
```

is a perfectly legal expression and results in the value 419.

Looking at machine registers

As well as the various arithmetic functions used in expressions `ndb` also recognizes several other symbols. The period symbol "`.`" denotes the value of the current address on each node. The values contained in machine registers are available using the < syntax so that `<PC` or `<IP` is the value of the program counter register.

Finally, symbol names evaluate to their addresses NOT their values. To access the latter value use the `*` operator; the value of variable `i` is `*i`.

Note that expressions may be used in most places where `ndb` accepts an integer value. The only exception is in set specification. You can't say

```
pick 1+3
```

to select node 4.

11 Assembly Level Debugging

`ndb` is designed to allow debugging at the level of the original C or FORTRAN source code. In certain circumstances, however, is may be necessary to "get one's hands dirty" and descend to the machine instruction level. For those people who relish such prospects `ndb` actually contains a second complete debugging system - the conventional UNIX tool `adb`. This has a complete syntax for debugging at the assembler level and is described in the accompanying reference manual.

Machine level debugging without learning adb

Many users, however, have no wish to get involved with the machine at this level so `ndb` provides a handful of commands to "do" assembly level debugging without getting too involved.

`listi`	Display the machine code from the "current location". By default ten lines of assembler are listed including symbolic references to variables and subroutines which should give some indication of their correspondence to the source code. Merely repeating this command prints out successive blocks of lines.
`listi address, count`	Display `count` lines of machine code starting at the named `address`. This is obviously just another variety of the previous instruction. You may omit the `count` argument for a default of ten so that `listi contour` prints out the first ten lines of the function `contour`.
`stepi`	Single step one machine instruction. If the next instruction is a function call then go to its first instruction and stop. Steps over "traps" to the operating system. `stepi 10` single steps 10 times and so on.
`nexti`	Single step one machine instruction but go over function calls rather than stopping inside them.
`show regs`	Display the machine registers. This can be useful in cases such as "illegal operand" or "numeric overflow" since it may be possible to figure out which operand is illegal from the disassembled instruction.

Many other "assembly level" commands exist but they should only be necessary in dire situations. For more details consult the reference manual.

12 Miscellaneous Commands

The simplest of these commands is the blank line - just repeatedly hitting the return key repeats the last command entered. This is useful for scrolling through source code for instance. (If you find this "feature" annoying you can disable it with the `ndbenv` command.)

The most important of the miscellaneous commands is "help" which invokes the on-line manual for `ndb`. The initial display describes the options which are available for further selection and which are obtained by typing `help` followed by a keyword. For example,

```
help source
```

produces a voluminous display of the various commands intended for source level debugging together with their arguments, defaults etc. As well as having information about various topics of general interest one can also type

```
help keyword
```

where "keyword" is any of the special identifying words known to `ndb`. Thus, for example, if you can't remember the exact syntax of the `list` command you can type `help list`.

Any command line that begins with `!` or `sh` is passed to a shell for execution. This is useful for looking at source files with editors or taking a break and firing up `hack` for a while.

Command aliasing

ndb has an alias command similar to that of the UNIX C-shell. This allows one to redefine any sequence of keystrokes to replace a regular ndb command. For instance it can get quite tiresome continuously typing `list` so one can say

```
    alias l list
```

which allows the single letter "`l`" to replace the word "`list`". For convenience one can place any ndb commands, including alias commands, in a file called `.ndbinit` in either the current directory or the user's home directory and this will be read in and processed whenever ndb is started.

Searching for source files

If ndb is unable to find source files for various subroutines it is possible that it is not searching the correct directories. This can be corrected in two ways. When starting up ndb directories can be given with the -I switch so that

```
 ndb -p 4256 -I/usr/jwf/code -I/usr/sys/buggy -d 4 toyland
```

starts the debugger in the usual way but adds two extra directories to the path which is searched whenever a source file is needed. To see the current path list enter

```
    use
```

which is also the second means of modifying the list. If the `use` command contains any arguments then the given list replaces the current one. So, for example, the command

```
    use /usr/jwf/code /usr/sys/buggy
```

has the same effect as the invocation of ndb shown previously.

Analyzing the message queues

Finally, commands are available to examine the state of the machine with regard to inter-processor communication. As well as being able to display whatever hardware information is available concerning communication (with the command `show pregs`) it is possible to obtain information about messages both *in transit* and upon arrival at their destinations. The command

```
    show q
```

displays the messages that have arrived on a node together with their source, type and length and a brief summary of the message contents. A sample output might be

```
    src = 1 type = 2748 len = 4 (12 00 00 00)@ 0x00079634
    src = 2 type = 2748 len = 16 (23 e4 ff ff ...)@ 0x00079804
```

which indicates the information described above for each message intended for this processor. The final item in each line is an indication of the memory location at which the message can be examined. This is useful in conjunction with the data formatting commands described in sections 8 and 10.

Another useful command in the context of communication is `show buffers` which analyses the usage of operating system resources on a node. This command is useful in detecting situations where too many messages have been sent overflowing some internal buffering and causing node failure.

The last, and possibly most useful, command is `quit` which exits from `ndb` and returns to the command prompt. If you originally started your program running from within `ndb` you will be prompted as to whether it should be killed or not. If the program started outside of `ndb` it will be left alone. This latter option allows the debugger to be used at will to examine the internal state of the nodes without adversely affecting the running program.

Performance Analysis

PM: A profiling system for parallel programs

1 Introduction

The most obvious goal of parallel computing is the acceleration of algorithms that execute too slowly on conventional machines. While other goals, such as fault tolerance, are also important most applications are ported to parallel machines with one aim in mind - running rings around expensive supercomputers.

The importance of performance analysis in parallel processing

Since this goal holds such a central position in the realm of parallel processing it is important that users be able to rapidly and effectively analyze their algorithms' performance. Even in cases where absolute speed is not the most important factor it is crucial to a thorough understanding of an algorithm to see the strengths and weaknesses of particular parallelization scheme. In this way it may be possible to see where bottlenecks occur and to devise alternative algorithms to avoid such problems.

Sequential vs. parallel profiling

Profiling parallel programs is, however, not so straightforward as on sequential computers. In the latter case the only really important piece of information is "How long am I spending in routine XXX?" and "Which routines should I speed up in order to accelerate the code most?". The style of profiling most often used in this context is a simple printout of elapsed times in each routine and, possibly, the number of times each was called and by whom. Armed with this information one can attempt to speed-up certain areas of the algorithm which are known to be heavily used. Alternatively, of course, one might be able to see that there are no real bottlenecks and that, therefore, the code is running as fast as it possibly can on the given hardware.

Factors affecting parallel program performance

Parallel programs are trickier because more factors arise which affect their performance. The most obvious, for a message-passing architecture, is the amount of time spent sending and receiving messages. One of the most quoted parameters of such machines is the "Efficiency" or "Overhead" which basically expresses how many times faster N processors are than 1. Once the parameter is known one might want to break it down further into times during which I/O is occurring, times when intermediate results are being accumulated globally and times when processors are communicating boundary values, for example.

A final factor which may be extremely important in parallel algorithm development is "load balance". This sort of problem can take many forms but is most clearly characterized by differences in execution speed of the different nodes in the parallel machine. Sometimes this can be caused because the workload is not evenly distributed between processors resulting in one node working exceptionally hard and correspondingly slowly. Many times this will slow down the other processors who are waiting to communicate with the slow node degrading the performance of the machine as a whole. Other problems may be more algorithmic in nature - a particular scheme for parallelizing a program may have some inherent defects which make some processors run more slowly than others. Detecting and correcting this sort of problem requires an ability to observe activities in several processors simultaneously at many levels of detail.

The profiling tools

The *ParaSoft* profiling utilities are designed with just these goals in mind. The three tools each serve one of the categories described above.

- The "execution profiler" monitors time spent in individual routines.
- The "communication profiler" assesses time spent in communication and I/O.

- The "event profiler" shows the interactions between processors and allows user specified "events" to be monitored.

Each is kept separate so that the user is free to concentrate on particular problems as they arise and can be selective in the amount of information available - it is one thing to provide detailed analysis tools but quite another to present the user with 200 Megabytes of data to analyze in order to understand the problems. As a result the majority of the tools have graphical interfaces. Menu driven utilities allow the presentation of accumulated data in simple graphical form under the complete control of the user. Optionally data can be presented in both hardcopy graphical and tabular form for more detailed analysis.

As mentioned in the previous paragraph one has to be rather selective in the data acquired for analysis. One of the more pressing needs for this ability is the fact that performance tools which significantly alter the execution of the target program are of little use. Essentially one ends up analyzing the profiling system rather than the user application! For this reason the tools described in this manual are of the "*post mortem*" type - that is, data is accumulated during the execution of the user program and then analyzed off-line, after execution has completed. This is done for two reasons

Problems with profiling in real-time

I/O in parallel computing systems in notoriously slow - especially when compared to the high computing power of typical machines. Even worse, I/O in one processor causes other processors to be affected in routing messages to the outside world. As a result even limited amounts of real-time I/O can cause significant modifications in program execution which completely invalidate the profiling procedure.

Displaying profiling data "real-time" looks quite attractive but rapidly overwhelms the human mind - particularly when more than a handful of processors are involved. Due to the constraints mentioned above it is difficult to present enough context to render a wildly varying display meaningful. Furthermore, saving the data on some physical medium for later use introduces a sequential bottleneck affecting all processors.

The profiler is built around *ParaSoft*'s interactive graphical system *Plotix* and hence supports all the same devices as that system. Among those included are; Tektronix 4010 and 4105, IBM EGA/VGA, SunCGI and the various graphical systems offered by NCUBE. Hardcopy is supported in either Postscript or Hewlett-Packard form.

The effect of programming models on profiling systems

Some mention is made of the *Cubix* I/O system. This is a feature of *ParaSoft*'s *Express* operating system which makes porting sequential codes to parallel computers particularly easy. For the present purposes, however, the only important question is whether or not your application is using *Cubix*. This should be straightforward but if you have any queries please call us. Having resolved this question different sections of the text and manuals will apply to your application - the profiler interface is subtly different in the two cases. In this manual applications which do not use *Cubix* facilities will be denoted as "Host-Node" programs.

The rest of this document is arranged as follows; Section 2 discusses the "Execution Profiler" which performs the function of common profiling tools on sequential computer systems - monitoring of subroutine usage. Section 3 describes the "Communication Profiler", a utility designed to analyze and quantify the time spent communicating, calculating and performing I/O functions. Section 4 is concerned with the "Event Driven

Profiler". This is a tool which allows the analysis of user specified "events" with particular emphasis on the interactions between multiple processors and which provides the most dynamic view of program execution. Each section is divided into several parts which discuss the instrumentation of user programs, the control of the profiling systems and the usage of the analysis tools. Section 5 contains complete example codes in C showing the use of the profiling system.

2 Execution Profiling

This section describes what might alternatively be called "sequential profiling" since the utilities described are those most familiar in the context of sequential programming. The goal is to analyze the time spent in the various subroutines and functions that make up an application. This allows the user to immediately focus attention on the most time consuming areas which would benefit most by improvement.

The data accumulated enable the user, on a node by node basis, to evaluate the time spent in each function and also that spent "idle" waiting on some external condition such as communication with another processor.

2.1 General Profiling Commands

Statistical profiling

This system works on a statistical principle. Every few milliseconds a system routine runs which looks at the current instruction being executed in the user application and increments a counter noting that this particular memory address was in use. In this way one builds up a histogram of the frequencies of hits in various areas of the program and hence the amount of time spent in particular routines.

Possible pitfalls in statistical profiling

Obviously the technique is not foolproof but if the application executes for a sufficiently long time to collect a reasonable number of samples then one can be fairly confident that the results are representative of the true behavior of the algorithm. (If the code only takes a few milliseconds to run, who cares anyway?). Among the obvious defects in this approach are possibly sick behavior if the program cycles at a similar rate to the routine which logs events. In this case one might always catch the program in a routine that actually doesn't take very long leading to incorrect conclusions - not very likely but possible. A more irritating problem is that it requires a lot of memory to make up the program histogram.

Despite these deficiencies this style of profiling is standard in most sequential computing environments and is part of the *ParaSoft* profiling system. It has the advantage that it's operation is mostly automatic - little or no change need be made to an existing program in order to "profile" it. An alternative system, the event driven profiler, is discussed in section 4 which requires more of the user but is probably more reliable in delicate situations.

Setting up the execution profiler

The two most elementary profiling functions are `xprof_on` and `xprof_off` which are used to enable and disable the profiling system, respectively. This allows the user to maintain fine control over the regions of the application that are actually analyzed - for example it may be sensible to turn off the profiler if one section of code is known to be much more heavily used than any other since one would otherwise be swamped by a vast amount of information about something already understood.

The heart of the profiling system is provided by the `profil` function. This tool is based on the standard UNIX utility of the same name and shares the same arguments

```
profil(buffer, buflen, start_addr, scale);
```

The first argument is a pointer to a buffer into which the profiling data will be dumped; the length of this buffer is the second argument. The `start_addr` argument specifies the lowest address to be considered in profiling the program. This is most easily discovered by searching through the "program map" files that are often produced by compilers.

The final argument, `scale`, has a rather obvious meaning but a rather bizarre interpretation. It is used to specify how many memory addresses to "bin" together. In order to conserve memory when profiling the system actually builds a histogram of memory locations and this argument specifies how wide the histogram bins should be. The method in which this is done, however, is quite obscure. The simplest way to explain this value is just to demonstrate how various values work, and then let your imagination take over;

`scale = 0x10000`	Maps individual addresses into separate bins.
`scale = 0x8000`	Maps each pair of instructions together into the data buffer.
`scale = 0x4000`	Maps four instructions together into a single profiling bin.

and so on.

This function does not actually turn the profiling system on; to do this one must use the `xprof_on` function. A sample code to use this profiling subsystem, therefore, is

```
/*
 * Sample program demonstrating the setup of the
 * execution profiling system
 */
#define SCALE   (0x2000)
int profbuf[2048];
extern int myfunc();

main()
{
/* Enable profiling system, and turn it on. */

    profil(profbuf, sizeof(profbuf), myfunc, SCALE);
    xprof_on();

/* Algorithm phase 1., profiler running ..... */

        ..........

/* Algorithm phase 2., turn profiler off .... */

    xprof_off();
        ..........
```

In this example we choose to profile at a resolution of eight bytes, selected by the value of the SCALE macro, with an 8 Kbyte buffer. Since the individual bins are 4 bytes this means that we have a range of 16 Kbytes in total (2048 bins, each with a resolution of 8 memory addresses). The starting point is selected as the beginning of the function myfunc. Since 16 Kbytes is not an awful lot of space for a large application we might expect some misses - at times the profiler will detect that the program is executing at addresses outside the range we have covered. These cases do not, however, crash the system but rather get logged into a special histogram bin with the label "misses". When analyzing the profile later the number of "misses" will be reported allowing a good guide to how badly the profiling range was selected. (For an example see Figure 1)

"Missing" the profiling buffer

This interface is not especially clean since it requires either good guesswork in picking the profiling range or else some time spent looking through a memory map generated by the compiler or linker. At some future date the system will interact directly with the user application and compilers to figure out a sensible range and call the profil routine automatically.

2.2 Details for *Cubix* Programs

In addition to the general routines discussed in the previous section users of the *Cubix* system have access to additional routines to facilitate the profiling process.

Enabling the profiler at runtime

The xprof_inq function may be called to inquire about the runtime status of the -m switch on the *Cubix* command line. Consider an application which is normally executed with the cubix command

```
        cubix -n 4 noddy <app.in
```

In this case the function xprof_inq will return 0 indicating that no appropriate runtime switch had been set. Modifying this command to

```
        cubix -n 4 -mx noddy <app.in
```

causes xprof_inq to return 1. This allows one to have the node application turn the profiling system on or off according to the setting of a runtime switch - no recompilation or special input is required. Note that the -m switch comes in several flavors. The 'x' character in the above invocation is used to specify the "eXection profiler" - the values 'c' and 'e' are also allowed and perform similar functions for the other two profiling subsystems. Note that the various options can be combined so that the switch -mxec will also generate a positive response from xprof_inq since it contains the 'x' character.

The second *Cubix* function is xprof_end. Called with no arguments this function is responsible for turning off the profiler, resetting its state and dumping out the profile data that has been accumulated to a disk file on the host computer. It must be called synchronously in all nodes and is typically used at the end of an application in code that might take the following form

```
    /*
     * Interface to execution profiling system:
     * CUBIX program.
     */
```

```
#define SCALE (0x2000)/* Map memory in 8 byte bins */
int profbuf[2048];
extern int myfunc();

main()
{
/* Enable profiling system, and turn it on IF -mx
 * switch given
 */
     profil(profbuf, sizeof(profbuf), myfunc, SCALE);
     if(xprof_inq()) xprof_on();

/* Application code */
         ...
/* Application code done. Dump out profile if
 * requested
 */
     if(xprof_inq()) xprof_end();
     exit(0);
}
```

Dumping the profiling data

The xprof_end function creates a file called "xprof.out" in the current directory. This name is chosen since it is the default for the profiling analysis tool discussed in section 2.4. Sometimes, however, a profile may be required more than once in the run of an application. The code might, for example, consist of multiple disjoint "phases" which it might be convenient to profile separately. In this case we can use the xprof_end function multiple times but each use must be followed by a call to the rename function. Otherwise the second call would simply overwrite the "xprof.out" file with another of the same name. Sample code to achieve this effect is

```
/*
 * Demonstrating the use of the "dump" functions to
 * make multiple output files containing profiles of
 * different phases of an application. CUBIX program.
 */
main()
{
 /* Application phase 1. */

     profil( ... );/* Initialize profiler */
     xprof_on();/* ...and turn it on */
      ...

     xprof_end();/* Write out prof data */
     rename("xprof.out", "phase1.out");
```

```
/* Application phase 2. (Don't profile since xprof_end
 * turned system off.)
 */
      ...

/* Application phase 3., reset profiler, and turn on */

     profil( ... );
     xprof_on();
      ...

     xprof_end();/* Write out prof data again */
     rename("xprof.out", "phase3.out");

        ...
```

Since we wish to profile phases 1 and 3 of the application separately we use the `xprof_end` function to write out the profile data. The `rename` function is used to keep the data in files with names of our choosing rather than `xprof.out` in which case the second call to `xprof_end` would overwrite the data file created by the first. Note the calls to `xprof_on` and `profil` which restart the profiler for phase 3 of the application.

2.3 Details for Host-Node Programs

Dumping the profiling data for analysis

Applications which have user written programs running on the host computer have a slightly different interface to the mechanics of dumping out data files to the host disk. In this case two functions are provided; `xprofcp` for the host computer and `xprofelt` for the nodes. The host routine has no arguments and merely serves to read in data from the nodes and output them in a suitable format to a disk file for later analysis. The node routine, `xprofelt` has a single argument which is the name of the file in which data should be placed.

Typical code to use these routines has the following form.

1. Host Program

```
/*
 * Dumping execution profile data to disk file.
 * Host program.
 */
main()
{
/* Allocate nodes, load programs */

      ...
```

```
    /* Execute application */

        ...

    /* Application finished, dump profile data */

        xprofcp();
        exit(0);
    }
```

2. Node Program

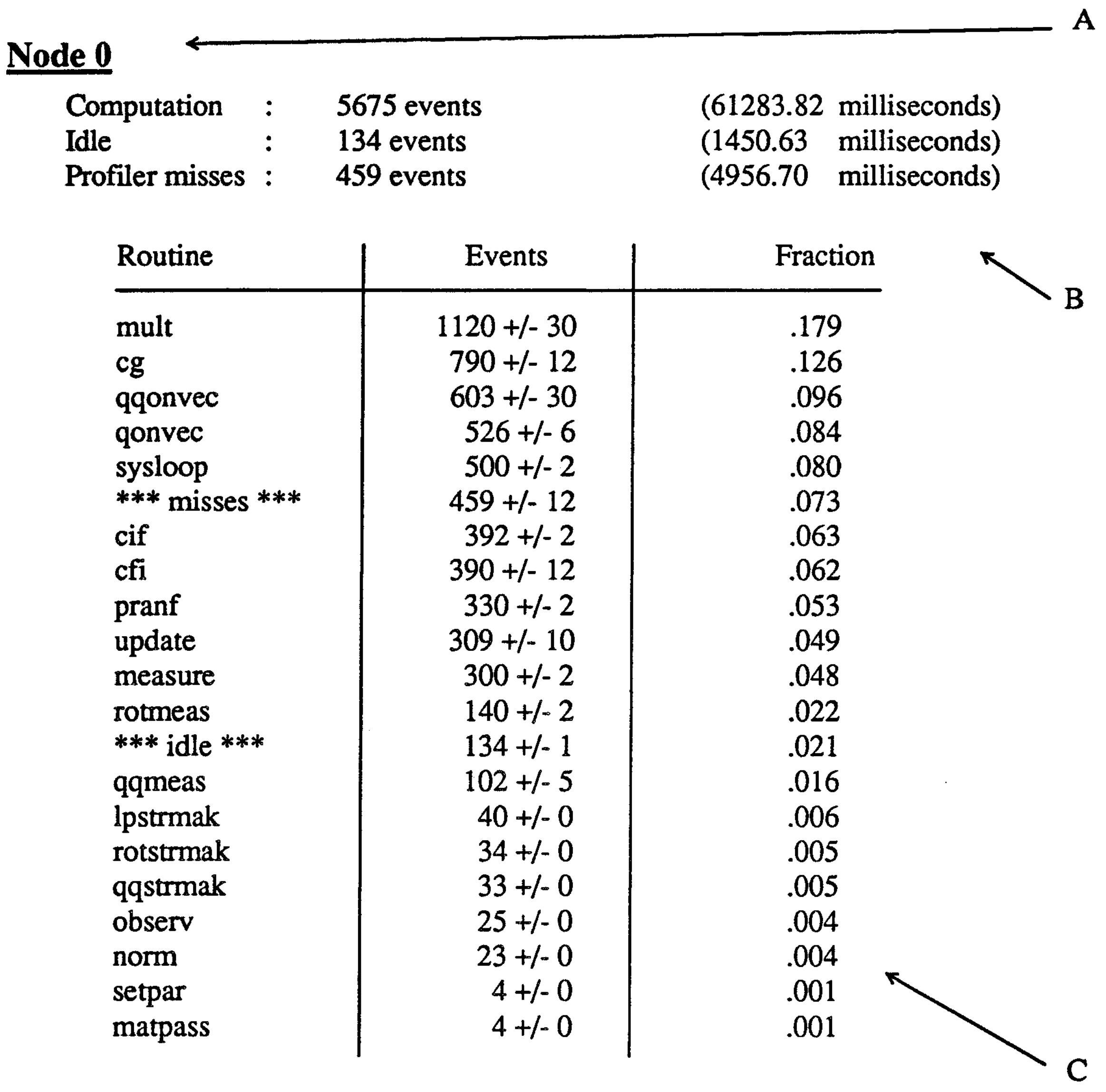
```
    /*
     * Dumping execution profile data to disk.
     * Node program.
     */
    main()
    {
        profil( ... );          /* Initialize profiler */
        xprof_on();             /* Enable profiler */

    /* Node application code */

        ...

    /* Node application over, dump profile data */

        xprofelt("xprof.out");
        exit(0);
    }
```

Notice that the name chosen for the profile data is the same as in the *Cubix* case. This is merely a convenience measure since it allows one to miss out an argument when using the analysis tool - if the file containing the profile data is "`xprof.out`" then you don't need to give it's name!

A final important point is that the `xprofcp` and `xprofelt` routines may be called repeatedly in a single application. The only constraints are that for each call to `xprofelt` in the nodes there must be a call to `xprofcp` in the host computer, and each call to `xprofelt` must be made "loosely synchronously". Also important is the fact that each call to `xprofelt` resets the internal state of the execution profiler and turns it off.

2.4 Analyzing the Execution Profile - `xtool`

After a file containing profiling data has been collected it is analyzed with the `xtool` command. Two arguments must be supplied; the name of the program which is to be



Node 0

```
Computation     :   5675 events        (61283.82  milliseconds)
Idle            :   134 events         (1450.63   milliseconds)
Profiler misses :   459 events         (4956.70   milliseconds)
```

Routine	Events	Fraction
mult	1120 +/- 30	.179
cg	790 +/- 12	.126
qqonvec	603 +/- 30	.096
qonvec	526 +/- 6	.084
sysloop	500 +/- 2	.080
*** misses ***	459 +/- 12	.073
cif	392 +/- 2	.063
cfi	390 +/- 12	.062
pranf	330 +/- 2	.053
update	309 +/- 10	.049
measure	300 +/- 2	.048
rotmeas	140 +/- 2	.022
*** idle ***	134 +/- 1	.021
qqmeas	102 +/- 5	.016
lpstrmak	40 +/- 0	.006
rotstrmak	34 +/- 0	.005
qqstrmak	33 +/- 0	.005
observ	25 +/- 0	.004
norm	23 +/- 0	.004
setpar	4 +/- 0	.001
matpass	4 +/- 0	.001

Figure 1. Sample Output from the Execution Profiler

profiled and the name of the file containing the written data. If this latter is "`xprof.out`" then, as advertized in the previous section, you can omit this last argument. A typical invocation, therefore, is

```
xtool noddy
```

The output from this process will consist of several tables similar to that shown in Figure 1.

The information provided by this utility is described below. The index letters refer to the figure.

A   Node identifier. A separate table is presented for the data from each node and is identified by its processor number.

B Node Utilization. The information here indicates active and idle time in the CPU and also the proportion of "misses" in the profiling histogram. "Idle" time consists of periods when the CPU is actually idle - such events only occur when a node is blocked waiting for communication to complete. The `misses` quantity provides some means of assessing the success or failure of the profiling range selected. If a large percentage of the profiling events missed the selected range then the tabulated data may be a poor representation of the application performance.

C Subroutine analysis. For each node a table is presented of the twenty "busiest" routines based on the number of profiling "hits" in that function. Together with the `*** misses ***` entry this information serves to indicate routines which might benefit from fine-tuning operations. Occasionally this information might also show up explicit defects in the parallelization scheme selected.

Depending on compiler support there may also be a field indicating the number of calls to each function and the mean time of execution for each. If present this data will also be displayed in the area denoted C in the figure.

The statistical nature of the profiler can again be seen in the '`+/-`' entries in the middle column of the display. Because of the finite width of the histogram bins it is not always possible to associate a number of "hits" with a single routine - the bin may overlap two (or more) functions. In this case `xtool` assigns a portion of the hit count to each of the affected routines and indicates by the '`+/-`' notation the possible error.

3 Communication Profiling

The most obvious difference between a sequential program and its parallel counterpart is in the interaction between the multiple processors. The most basic of these interactions is the communication traffic between nodes. The "communication profiler" is a tool designed to monitor internode message traffic and estimate overheads in various types of communication.

The communication profiler collects data about interprocessor communication

On each node data is accumulated to measure

- Time spent calculating, communicating between processors and performing I/O functions. Leads to an estimate of program overheads and "Efficiency".
- Number of errors encountered in communication systems. Various types of common error conditions can cause programs to behave differently than expected. This provides a quick and dirty check to see if such a fact might explain mysterious bugs.
- Total number of calls to the communication system. Provides a simple estimate of load imbalances.

As well as keeping track of the above statistics for each node the following data are maintained in each node on a function-by-function basis for every entry point into the communication system

- Number of calls to each individual function.
- Number of errors incurred in each function.
- Distribution of return values from each function. Each function in the communication system returns a value indicative of the nature of the communication performed; message length written, message length read, number of objects broadcast etc. This allows the user to evaluate the communication policy of an algorithm - in particular it may be more effective to bundle up short messages into longer communication packets rather than sending data piecemeal in short messages.

3.1 General Profiling Commands

The communication profiler is almost completely automated requiring little interaction from the user. Two routines `cprof_on` and `cprof_off` are provided to control the profiler. They turn the system on and off respectively. This allows complete selectivity as to exactly what portions of the code are profiled. This can be very important - when using the event-driven profiler of section 4 it may be important to suppress the recording of communication events to conserve memory. More will be said about this point in the appropriate section.

3.2 Details for *Cubix* Programs

Enabling the profiler at runtime

Applications running under *Cubix* have a particularly clean interface to the communication profiler. At run-time one can turn the system on or off using the `-m` switch on the `cubix` command line. This switch is a general purpose facility for controlling the profiling systems and the particular value of concern in this section is the 'c' subswitch. Consider, for example, a typical *Cubix* application which is executed with the command

```
        cubix -d 4 noddy 1.2 6 20 20
```

If this line is modified to

```
        cubix -mc -d 4 noddy 1.2 6 20 20
```

then the communication profiler is enabled and collection of statistics will be performed. Note that this switch is a simple variant of the `-mx` switch introduced in the previous section for "execution profiling". In fact the two may be combined as `-mcx` to enable both systems. If the application were executed in the above manner then a file `cprof.out` would be written in the current directory at completion which contains the profile data.

The above construct is actually implemented in terms of two functions which are also available to the user; `cprof_inq` and `cprof_end`. The former merely queries the host file server to see if the `-mc` switch were specified on the command line while the second is used to write out the final communication profile to disk. Either function may be used at any time from within an application with the caveat that each invocation of `cprof_end` writes a file with the name "`cprof.out`". If it is to be executed multiple times then each version of the output file should probably be saved under a different name with the `rename` function. (See similar discussion in the context of `xprof_end`.) Note that `cprof_end` also turns off the profiler and resets its internal state. This allows further data collection to begin with a "clean slate".

In the simplest case no modifications need to be made to an existing *Cubix* in order to use the communication profiler. The only change is the addition of the `-mc` switch at runtime which fires up the system and writes out the data; the other functions are merely provided for extra control should it be needed.

3.3 Details for Host-Node programs

The interface to the communication profiler for applications that have programs to run on the host computer is very simple. The only difference from the *Cubix* is that the user has to coordinate the writing out of the profile to disk using the `cprofcp` and `cprofelt` functions. At any point one can write out the current profile by calling `cprofcp` in the host computer and `cprofelt` in the nodes. `cprofcp` has no arguments while `cprofelt` has one: the name of the file to which data is to be dumped. Note that this is slightly different from the *Cubix* which always used the same file name for the profile output. Also the profiling system must be explicitly turned on with a call to `cprof_on`. Note that the `cprofelt` function turns off the profiler and also resets its data so that further profiling starts from zero. The prototype code for an application running with a user host-program is as follows;

1. Host Program

```
/*
 * Demonstration of host-node interface to routines
 * which dump out communication profile data.
 * Host program.
 */
 #include "express.h"

 main()
 {
 /* Allocate nodes, load programs */

     if(exopen("/dev/transputer", 4, DONTCARE) < 0) {
         fprintf(stderr, "Failed to allocate nodes\n");
         exit(1);
     }

 /* Load and execute application */
         .......

 /* Application finished, dump profile data. */

     cprofcp();
     exit(0);
 }
```

2. Node Program

```
/*
 * Demonstration of host-node interface to routines for
 * dumping out communication profile data.
 * Node program
 */
main()
{
/* Turn on communication profiler */

    cprof_on();

/* Node application code */
     .......

/* Node application over, dump profile data. */

    cprofelt("cprof.out")
    exit(0);
}
```

The communication profiler's data file

Notice that the node code explicitly enables the profiler and has to dump out the final data. We chose the name "`cprof.out`" for later convenience since this is the default name for the analysis tools - any other name would be allowed. Also notice that one is completely at liberty to call `cprofelt` many times within a node application to dump out profiles from different parts of the code. The only constraint is that there must be a corresponding call to `cprofcp` in the host program for each `cprofelt` in the nodes.

3.4 Analyzing the Communication Profile - `ctool`

Tabular output with no graphics

After program execution has completed one or more files should be left containing the communication profiles for the application. These are analyzed with the `ctool` utility. For the moment we will neglect the graphical interface and simply present a tabular version of the profile. This is achieved with the command

```
ctool -p
```

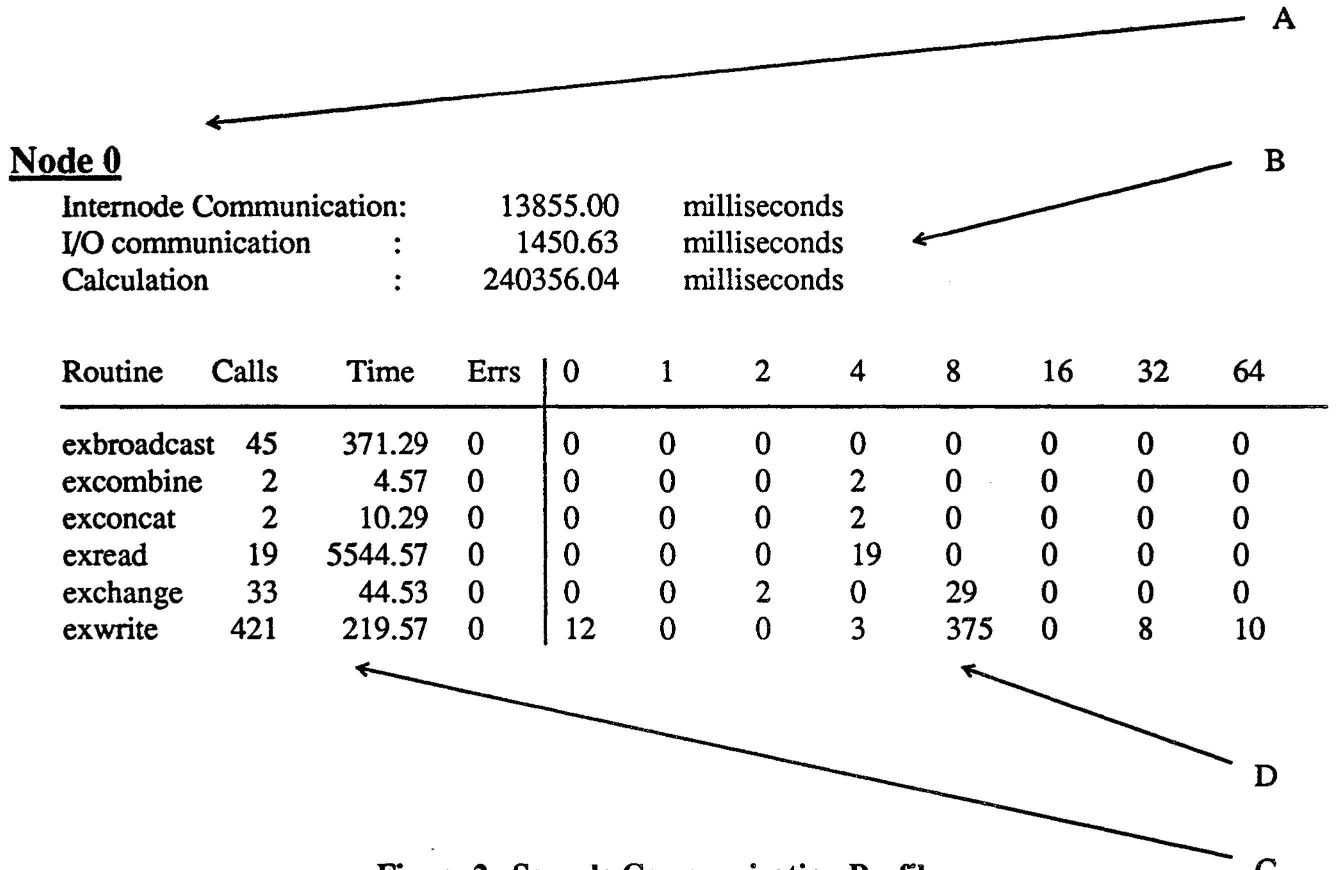
which expects to find the communication profile in a file called "`cprof.out`". If the file was renamed for some reason then one might instead use

```
ctool -p phase1.dat
```

to read profiling information from a file called "`phase1.dat`". The result of this command is a table of the form of Figure 2.

The various fields in this display are summarized as follows

A. A separate section of the table is provided for each node, and is identified



```
Node 0
    Internode Communication:      13855.00    milliseconds
    I/O communication        :     1450.63    milliseconds
    Calculation              :   240356.04    milliseconds

    Routine      Calls     Time    Errs | 0    1    2    4    8    16   32   64
    ----------------------------------------------------------------------------
    exbroadcast  45      371.29    0    | 0    0    0    0    0    0    0    0
    excombine     2        4.57    0    | 0    0    0    2    0    0    0    0
    exconcat      2       10.29    0    | 0    0    0    2    0    0    0    0
    exread       19     5544.57    0    | 0    0    0    19   0    0    0    0
    exchange     33       44.53    0    | 0    0    2    0    29   0    0    0
    exwrite     421      219.57    0    | 12   0    0    3    375  0    8    10
```

Figure 2. Sample Communication Profile

by its processor number.

B. A brief summary of the total times spent calculating, performing I/O and communicating between processors is provided with times in milliseconds. Note that basic communication with the host using primitives such as `exread`, `exwrite`, `exreadfd` etc. is counted as interprocessor communication; only standard I/O functions such as `read`, `write`, `printf`, `fopen` etc. are classed in the I/O category.

C. For each node there is a breakdown of the calls to each of the basic communication functions. For each function that was called at least once the number of calls, the total time in that function, and the number of errors returned by that function are shown. Times are again in milliseconds.

D The final panel of the display shows a brief analysis of the way in which the various functions were called. The tabulated values show the frequency of return values from a function, binned in logarithmic steps. Thus, the first column indicates the number of times that the given function returned 0 to its caller, the first column the number of times 1 was returned etc. In figure 2, for example, we see that Node 0 called `exwrite` 421 times and the return value was between 8 and 15 375 times. The exact interpretation of this information obviously depends somewhat on the function being called but, in almost all cases, is related to the message length being dealt with. The number of bins displayed can be modified with the '`-b`' switch.

The information tabulated by this command allows a rather detailed study of the algorithmic communication patterns to be made. In particular we have found it invaluable for finding program errors in which too much data is sent in some system call. This type of error tends to show up very easily in the tabular output. Alternatively a graphical interface

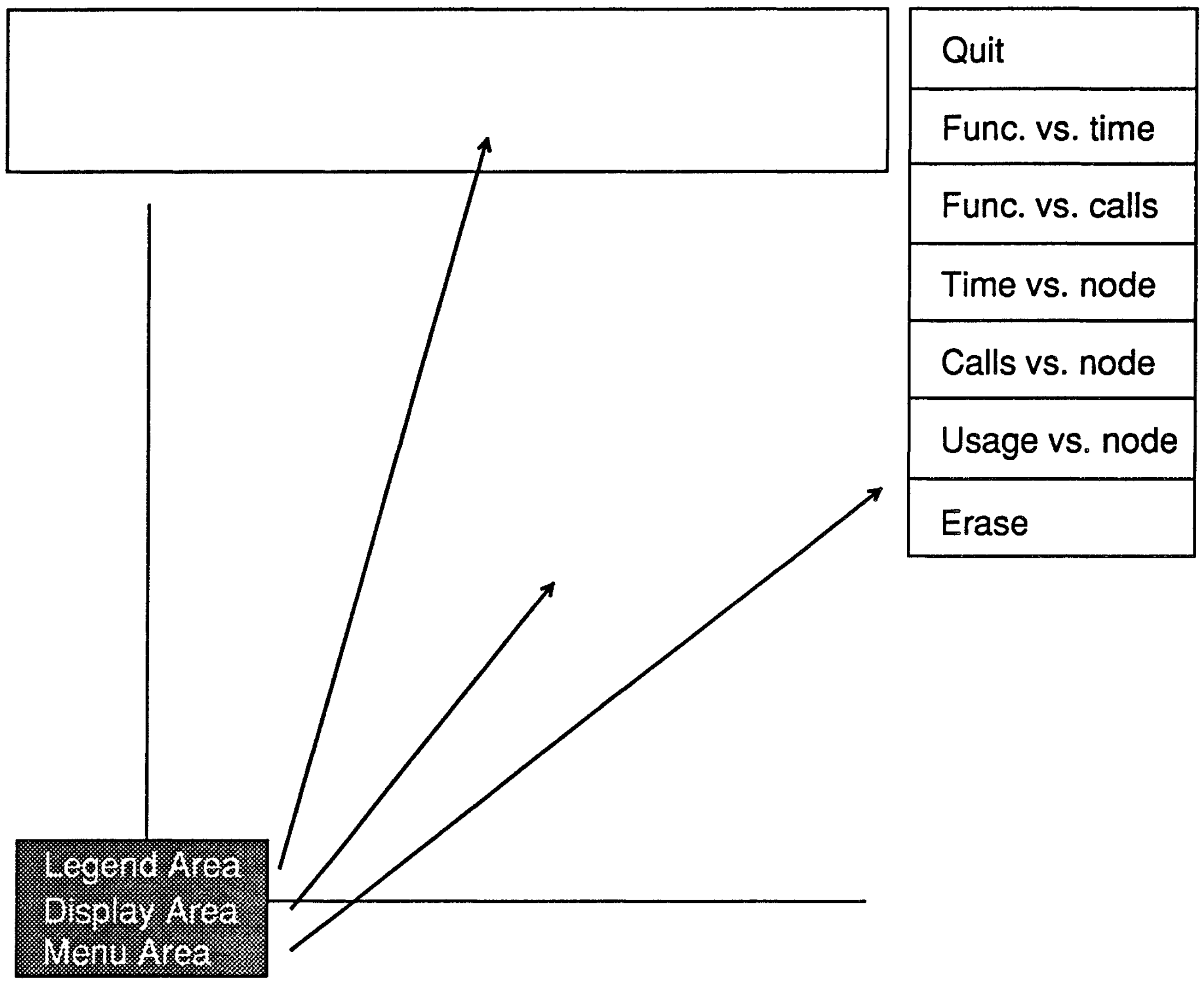


Figure 3. Initial `ctool` Menu and Display

Analyzing the communication profile graphically

is available for displaying the data. This is accessed even more simply by omitting the `-p` switch from the `ctool` command - in most cases one simply executes

```
ctool
```

although occasionally one may have to give the name of the file containing the profile data. After a couple of seconds of initialization a diagram similar to Figure 3 should appear.

Various pieces of the display are used for special functions

Menu Area	One selects from the various options available by positioning the cursor over one of the boxes in this area and "clicking". The text in each box should help you make the appropriate selection. (If your system has a mouse you should be able to make selections in the usual "point-and-click" manner. If not use the arrow keys on the numeric keypad to move the cursor and strike any alphanumeric key to make a selection. The "HOME" key toggles between fast and slow cursor motion)
Display Area	This area of the display is used to present graphical data. Various types of graphs are available to show different aspects of the communication profile.
Legend Area	While data is being displayed graphically this box should contain a "legend" indicating the meaning of the various items shown.

Analyzing two degrees of freedom at a time

The selections in the main menu which should now be visible represent various ways of showing profile data. There are essentially three variables involved in each case; the particular function, a node and an "interesting" quantity which in this case is either "time" or the number of "calls" to a particular function. Since graphs are really designed to show only two variables - one on each axis, there are several ways of showing the data.

The first display to select should be **Function vs. Calls**. This presents a horizontal bar chart showing the number of times each function has been called in each processor. An example is shown in Figure 4

Dealing with large numbers of processors

Notice that the various processors have been displayed by individual (color-coded, if you have a color device) bars and that a key appears in the "Legend Area" showing which processor is which. In the example shown we assumed that only four processors were being profiled which fit quite nicely on the display. If you have 512 nodes then the bars are going to be awfully thin! In this case `ctool` leaves out all but the first 16 encountered. However, you can control which bars you see by using the menu which should also have appeared (See Figure 4). Two of the options should be **Add Nodes** and **Remove Nodes** which allow you to either add or delete nodes from the display.

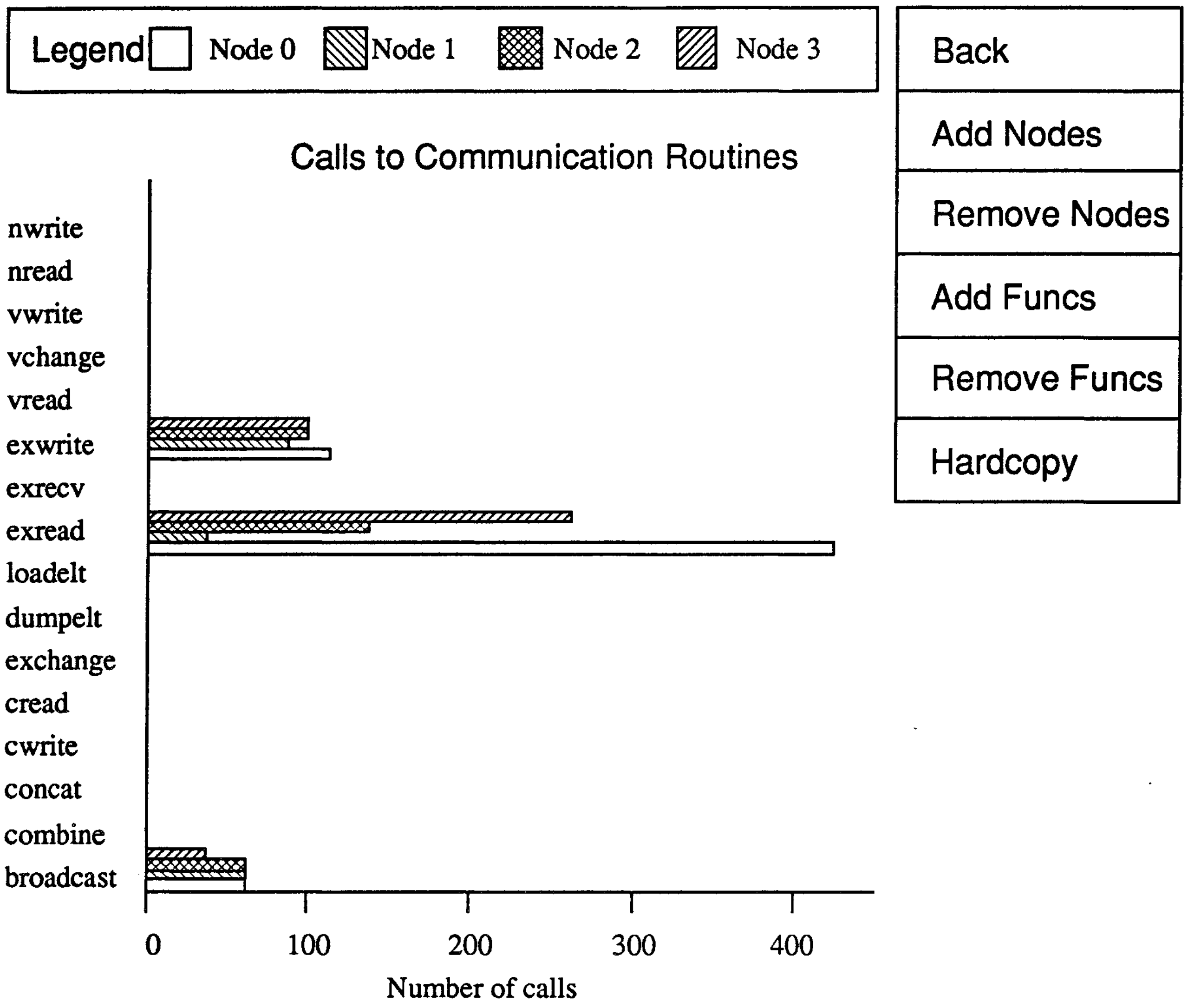


Figure 4. "Function vs. Calls" display

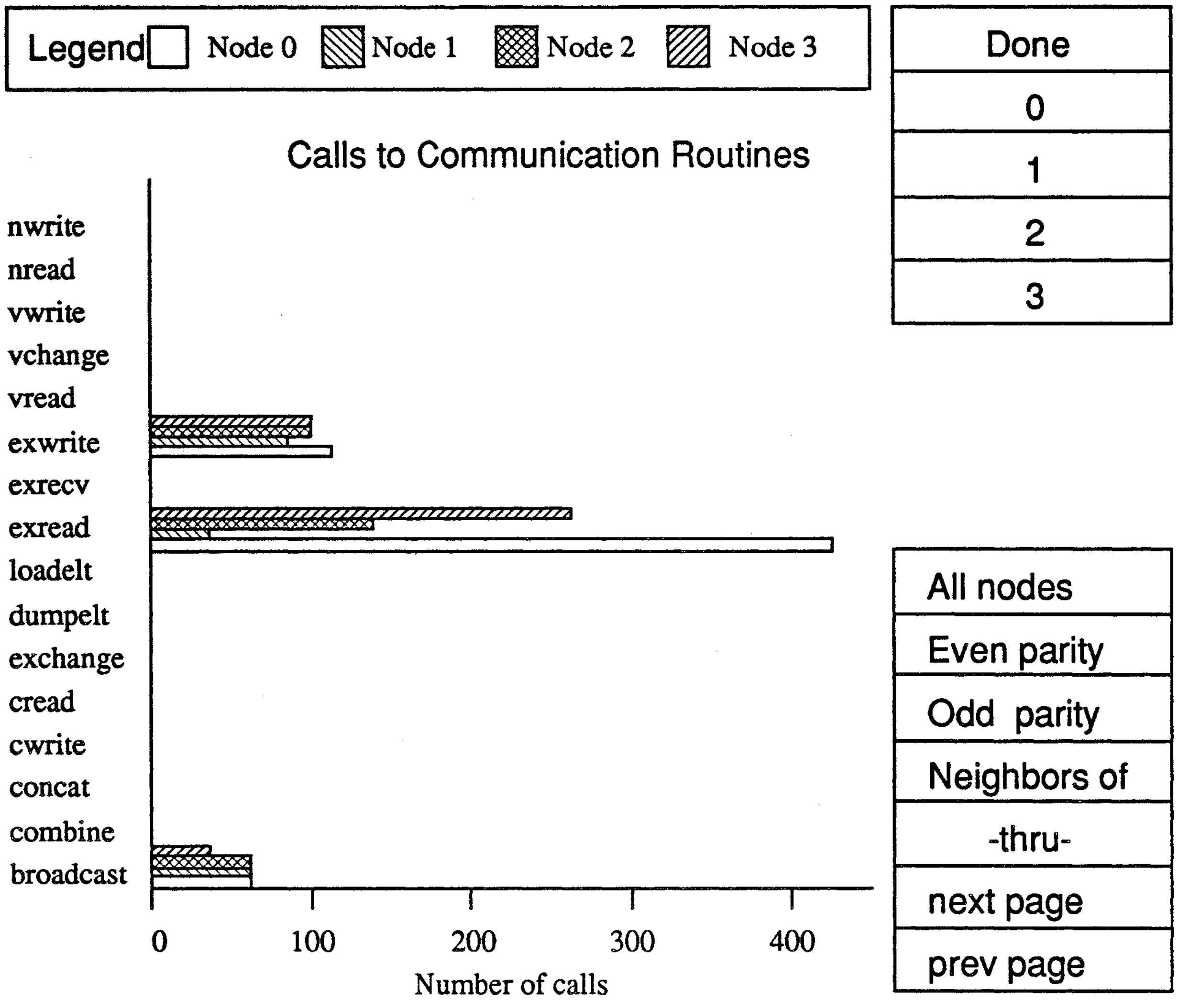


Figure 5. Node Selection Menu

Selecting either of these options yields yet another menu which looks like Figure 5.

Selecting nodes for display

This menu is typical of the lower level menus. It has a **Done** entry at the top which takes you back to the previous selection and then a whole bunch of other options. In order to select some nodes for either addition or removal you can

- Select individual nodes by clicking on their numbers. As you do this they should change pattern to let you know that they have been selected.
- Select all nodes by clicking on the **All Nodes** box.
- Page through the nodes, if there are too many to fit on the menu all at once. **Page Forward** and **Page Back** move through the set of nodes.
- Select a range of nodes by clicking on the first number, then the **Thru** box and

then the second number. You can switch pages in the middle of this operation if the range spans multiple pages.

- Select the neighbors of a particular node. One interesting property of parallel machines is the way in which one node's behavior can affect those connected to it. To select this option click in the **Neighbors of** box and then the node whose neighbors you want to pick.
- Select nodes according to their "parity". This concept corresponds to the familiar "red-black" coloring often used in parallel processing. Node 0 is defined to have even parity and its neighbors to have odd parity. This then extends naturally so that no even parity node is adjacent to another even parity node, and similarly for the odd parity processors.

After selecting one (or more) of the options from this menu, clicking **Done** takes you back to the previous menu level and you will see the consequence of your selection on the "Display Area". If, for instance, you decided to add all 512 nodes to the display you are probably waiting quite a while for the very tiny graphs to get drawn!

Droppingfunctions from the display

This technique is one way of getting additional node displays onto the screen. Another method is to "lose interest" in particular functions. By default every function has a trace even though it may never have been used. To remove some functions from the display select the **Remove Funcs** option. The menu that appears looks a little like the node selection menu - it has **Done** at the top and then a list of communication functions. You click away at the names (which should change background when selected) until the uninteresting ones have been deleted and then select **Done**. This returns to the previous level and updates the "Display Area" with less functions, and correspondingly more space for node displays. Another thing to notice is that the horizontal axis rescales whenever functions are added/deleted so it is occasionally useful to remove functions whose usage dwarfs the others to force a magnification of the horizontal axis showing more detail. The deleted functions can always be restored later if required.

The final option on this, and the other display menus, is **Hardcopy**. Clicking on this box saves a copy of the current display (Without the attached menus) in a form suitable for printing. Either PostScript or Hewlett-Packard devices are supported.

Once all interesting information has been extracted from this display one can click on the **Back** box to return to the main selection menu. The box **Function vs. Time** presents the same style of graph but with the horizontal axis displaying the time spent in each function rather than the number of times it was called as in the previous case. The two boxes **Time vs. Node** and **Calls vs. Node** present alternative views of the profile data.

These options present typical graphs of either the time spent in a routine or the number of calls to a routine against the processor number on the horizontal axis. Individual curves are drawn for each function selected by the user. Clicking on the **Time vs. Node** box, for example, should produce the display shown in Figure 6.

By default no curves are drawn on these graphs which makes them rather dull to look at. However, the menu selection **Add Funcs** should display a list of functions for which curves can be added to the "Display Area". The selection technique is just as before - click away until enough curves have been selected and then **Done** takes you back to the previous

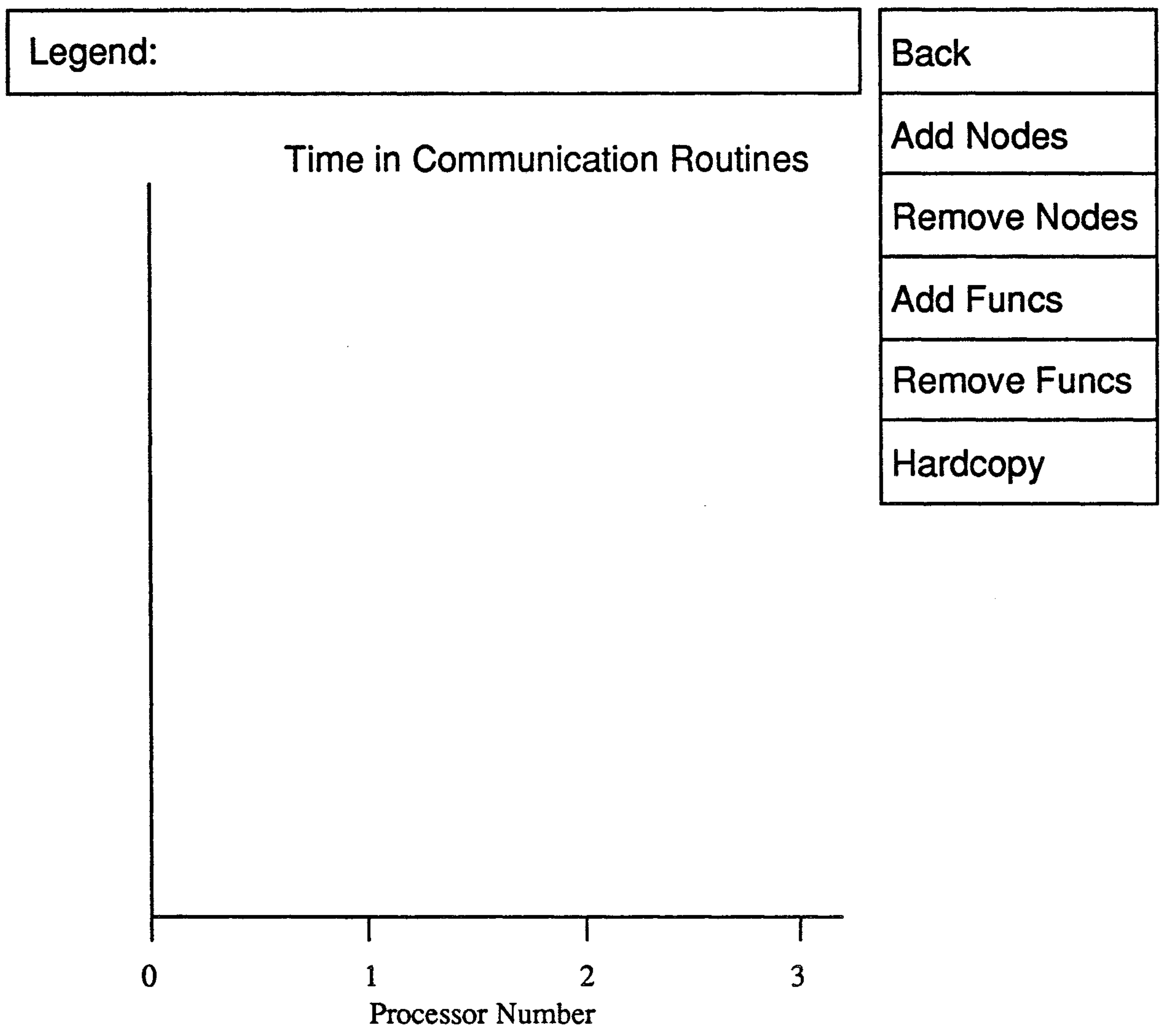


Figure 6. Basic display for "... vs. Node" Graphs

level and displays the results. Also notice that the "Legend Area" is updated to reflect the new style of graph. Selecting `exread`, `exwrite` and `exchange`, for example, might lead to a display like that of Figure 7.

This type of display is obviously better suited to showing data from a lot of processors. Even so it occasionally becomes too crowded and the **Add Nodes** and **Remove Nodes** menu selections are available as before. Note, however, that it doesn't make too much sense to pick out weird node combinations for this type of display and, in fact, `ctool` will ignore you if you try. While displaying data in this fashion only the highest and lowest processor numbers are considered and everything in between is also plotted.

Looking at program "efficiencies"

The final graphical tool available is obtained by selecting **Back** from this menu level and **Usage** from the main menu. The result will look something like Figure 8.

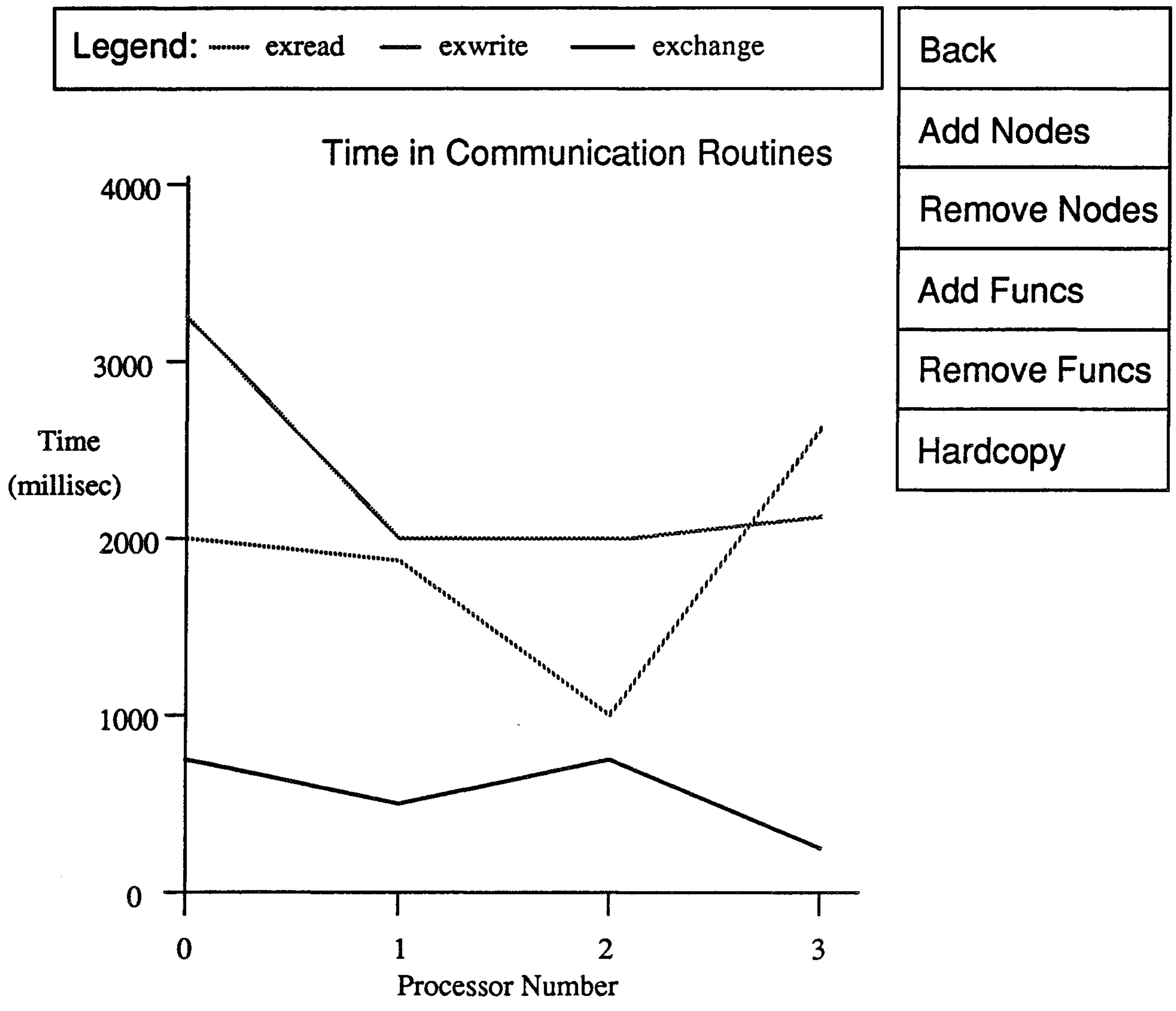


Figure 7. Time vs. Node" display for several functions

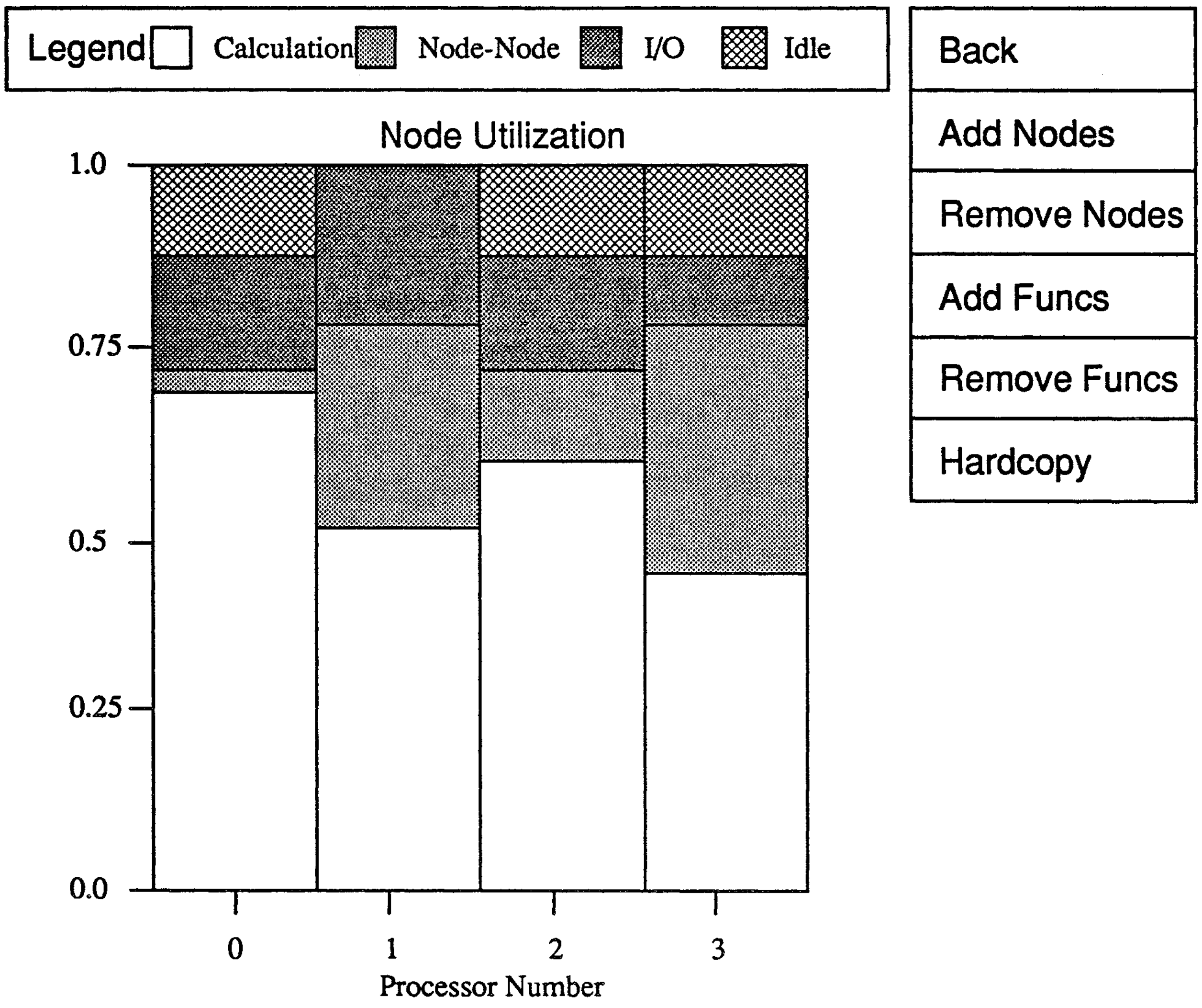


Figure 8. Node Usage Display

Along the horizontal axis are the processors just as in the previous displays. For each processor a stacked bar chart is presented showing the division of time between the three fundamental tasks; calculation, interprocessor communication and I/O.

Termination of the profiler is achieved by selecting **Done** from the main display menu.

4 Event Driven Profiling

Contents of "events"

The two profiling techniques discussed so far have been tailored to examining the behavior of nodes in isolation. The event driven profiler is provided to allow more detailed examination of the interaction between various nodes as time progresses through the application. An "event" is a user-specified point in the execution of an application which will be recorded in an internal log for later analysis. Together with the fact of the event's occurrence one can also record

- The time at which the event occurred.

- An "index" value indicating the nature of the event.
- A program variable whose value at the time of the event will be recorded. This will help in later identifying events during analysis.

Techniques for identifying events during analysis

As well as the above data items which are recorded every time an event occurs the following can optionally be supplied

- A "title" which identifies all events with a given "index" value.
- A `printf`-style format string which will be used when printing the value of the program variable stored at the time of the event.

These last two items are intended to facilitate identification of program events in the analysis phase. They may be omitted if desired.

It is important that a user program containing event specifications does not have to incur the overhead of the profiling system. As with the communication profiler discussed in the previous section one is free to turn the event profiler on or off at will, completely independently of the other profiling systems.

Also in common with the communication profiler some of the details of the user interface differ slightly depending on whether the application is running under the *Cubix* system or with a user written host program. These details are discussed later in this section after a discussion of the features which are common to all applications.

4.1 General Profiling Commands

The most obvious of the event profiling commands are `eprof_on` and `eprof_off`. Called with no arguments these functions serve to turn on and off (respectively) the event profiling system. This allows fine control over the areas which will be profiled and also lets completed applications run intact without removing the profiling commands.

Registering user events at runtime

The most important function in this section is `eprof_add` which causes an "event" entry to be added to the log-file. Its usage is

```
eprof_add(index, datum);
```

The first argument is an identifier for the "type" of event being recorded which allows one level of identification when analyzing the trace. One example of its use might be to flag all calls to a particular routine with `index = 1` while calls to another routine might have `index = 2`. The second value is another means of identifying events. It is a 32-bit integer value. A good example might be a loop counter or the value returned by some function.

The following code illustrates one use of these functions

```
/*
 * Sample program demonstrating the use of the event
 * profiler.
 */
#include "express.h"

main()
```

```
{
    int iter;
    float value, crunch();

/* Initialize profiling system to defaults, and turn
 * it on.
 */
    eprof_init(DONTCARE, DONTCARE);
    eprof_on();

/* Set up algorithm, loop several times - each time
 * record an event of type 1 and also record the loop
 * index.
 */
    for(iter=0; iter<100; iter++) {
        eprof_add(1, iter);

/* Now record a type 2 event for the completion of the
 * CRUNCH function and also save the value it returned.
 */
     value = crunch(iter);
     eprof_add(2, (int)value);

     enditer(value);
    }
    exit(0);
}
```

Initializing the event system

Notice the call to `eprof_init` in the previous example. This is important. Each call to `eprof_add` stores additional information in an internal log-file using up memory on each node. The amount of memory set aside for event logging is determined by the call to `eprof_init` which must occur before attempting to turn on the profiler, or use any of the event profiler functions. This routine takes two arguments

```
        eprof_init(numlogs, numlabs);
```

in which the first argument specifies how many event log entries should be allocated. If an attempt is made to write more log-file entries than specified in this call the extra are discarded and a warning is issued whenever the data is analyzed. The second argument specifies how many "title" or "label" entries will be made to aid in identifying the program state when an event occurs. Either argument may take the special value DONTCARE, defined in `express.h` in which case a default is chosen. One important fact about this routine is that it is used to synchronize the clocks on the various processors and must, therefore, be called "loosely synchronously" in each processor. A strict definition of this term can be found in the *Cubix* documentation but briefly it means that the call must be made at a point where each node is free to communicate with all the others - i.e., there must be no pending communication calls, unread messages etc.

Analyzing the event data

Having described how the system is initialized and how events are entered into the log one must consider the steps taken to aid in analyzing the profile data. The `etool` utility presents "time-lines" for the individual processors upon which are superimposed the user-specified events. Each event is identified in this display with its "index" argument from the `eprof_add` call responsible for its existence. One problem with this style of display, however, is that it is often quite tricky to figure out the correspondence between the time lines and what the actual application is trying to do. This problem is somewhat alleviated by intelligent choices of the "`index`" parameters. Since each event is labelled with this value one gets a rough guide.

Adding labels to events makes them easier to find

The connection can be strengthened by specifying a title field for each "index" value with the `eprof_label` call. This is called with three arguments

```
        eprof_label(index, title, format);
```

We will return to the last argument later but the "`title`" is merely a character string that will appear in a "legend" on the display of the time-lines. Each "`index`" can have a unique "title" assigned to it in this manner allowing reasonable identification of the various event types. In the previous example one might add the calls

```
       eprof_label(1, "Top of major iteration loop",
                   "Iteration %d");
      eprof_label(2, "After crunching",
                   "crunch returns %d");
```

Note how the title strings identify the purpose of the two types of events. In addition to displaying the user events on the time-lines `etool` also allows access to a second layer of information - that supplied in the second argument to `eprof_add`. This information is available upon request and interacts with the last argument to `eprof_label`. Whenever the next layer of detail is requested the user datum corresponding to the selected event is passed, together with the last argument to `eprof_label` to be printed out. Thus, for example, one might inquire about the details for a particular event and be told

```
        1. T = 246.23 ms, "Iteration 39"
```

The information contained here is the "index" number, the time at which the event occurred and the user data item formatted in conjunction with the format string given to `eprof_label`. Notice how this information can be used to exactly locate a position on the time line according to which iteration of the major loop it signifies. Even program bugs might be detected this way since clicking on a type 2 event might yield

```
        2. T = 253.60 ms, "crunch returns -2461"
```

which, in conjunction with the previous output, might be enough to detect that the program is going crazy at iteration 39 since the value returned by the `crunch` function is negative.

Careful use of the labelling facility is the key to using the event profiler. Without it one often has to resort to guesswork in order to relate the events shown on the time lines to the program's behavior. If the labelling is performed carefully, especially the specification of the second piece of information, the "data item" argument to `eprof_add`, the event profiler will be a rich source of information about the performance (and maybe even bugs) of an application.

Logging interprocessor communication as "system" events

The event profiling tools described here can also be used to record important "system" events. A particularly important class of interesting events are communication calls between processors. If the event-driven and communication profilers are both enabled (i.e., `eprof_on` and `cprof_on`) then each communication call also makes entries in the log file. As well as recording which function was called and the value it returned to its caller one can also determine exactly how long each communication call takes. This is invaluable, for instance, in determining the affects of poor load-balance - typically one processor will wait for an excessive amount of time in communicating with an overworked node.

This latter is actually another method of performing communication profiling. Even if no user events are specified the system can still be used to log "communication events" allowing a detailed analysis of the internode communication patterns to be performed.

4.2 Measuring time intervals with "Toggles"

Measuring statistics at the level of groups of source lines

The event profiler also provides a mechanism for measuring important statistics in relation to section of program code. While the "execution profiler" described in section 2 is useful for collecting information at the level of individual subroutines it is often important to be able to analyze code at a finer level, or to gather statistics about the frequency with which a given program segment is being used.

To facilitate the gathering of such statistics the event profiler uses the "toggle" concept. A "toggle" is a structure which gathers information about the time spent within a particular program segment and the number of times this code is executed. A simple example of its use is shown in the following code

```
#include "express.h"

ETOGGLE looptog, grindtog;

main()
{
     float Energy, grind_away();
     int iter, i;

/* Initialize toggle data structures. */

     eprof_toginit(&looptog, "Main iteration loop");
     eprof_toginit(&grindtog, "Calls to grind_away");

/* Start application code, then go into main loop */

          ...

     for(iter=0; iter<100; iter++) {
         eprof_toggle(&looptog);

              /* Other processing going on here.... */
```

```
                                  ........

                   for(i=0; i<4; i++) {
                       eprof_toggle(&grindtog);
                       grind_away(Energy, i);
                       eprof_toggle(&grindtog);
                   }
                   eprof_toggle(&looptog);
               }
       /*
        * Dump data to host for later analysis.....
        */
                         .........

               exit(0);
       }
```

We set up two "toggle" variables using the ETOGGLE type defined in the express.h header file. We then surround interesting pieces of code with identical calls to the eprof_toggle function which alternately starts and then stops recording information about the code section. (This is why the tool is known as a "toggle" - successive calls alternate between turning it on and off.)

The ETOGGLE data type

The statistics gathered include the time taken to execute the enclosed call and the number of calls to this code fragment.

Each toggle structure *must* be initialized with a call to eprof_toginit as shown at the top of the previous example. This notifies the system of the use of the particular variable and also allows the user to associate a title string with the indicated "toggle". This makes it easier to analyze the resulting data since the string will be displayed along with the associated data.

Initializing "toggles"

4.3 Details for *Cubix* Programs

The *Cubix* interface to the profiling system is extremely simple. If one is happy to use all defaults then the profiler can be enabled at runtime by using the -m switch on the cubix command line. This is similar to the mechanism discussed in the communication profiling section except that the magic character is 'e' (event-driven) rather than 'c' (communication). To turn on the event-profiler in a case where the usual cubix command is

Enabling the profiler at runtime

```
        cubix -n 4 noddy 1.2 1024 1024
```

one just uses

```
        cubix -n 4 -me noddy 1.2 1024 102
```

which automatically takes care of calling both eprof_init and eprof_on. It further arranges that upon the completion of the user application a file called `eprof.out` will be created containing the event-log information. Note that the various options of the `-m` switch can be combined so that, for example, to enable both communication and event driven profilers one uses the `-mec` switch.

Notice that, if the event profiling system is to be used to monitor communication only -i.e., no user events, then no modifications need be made to an existing node program. Merely giving the `-mec` switch upon execution creates the required profiling information!

If the system defaults are inadequate for a particular application then the `eprof_init` call may still be used. The values specified merely override those chosen by the system. An alternative scheme which is occasionally useful is to dump out the profile data manually rather than having the system do it at program termination. This is achieved with the

```
eprof_end
```

function which dumps the current profile log into a file called "`eprof.out`", turns off the profiler and resets its state. Note that the same comments apply as in the discussion of `cprof_end` in section 3.2 - the same filename is always used by this function so one has to be careful to `rename` the output each time to avoid it being overwritten.

The `eprof_on` and `eprof_off` functions may be used freely to control the periods during which profile data is being accumulated.

4.4 Details for Host-Node programs

If a user application is being executed on the host processor then the times at which profiling data are dumped must be specified by the user. In order to do this two functions are provided; `eprofcp` for the host and `eprofelt` for the nodes. The former has no arguments and merely serves to receive the data transmitted by the `eprofelt` call whose single argument is the name of the file to which data is to be dumped. Note that these functions must be called opposite one another - i.e. the host and nodes must execute their respective functions together. The usage of these calls is identical to that of `cprofcp/ cprofelt` and the pseudo-code of section 3.3 can be used after suitable changing of function and file names.

The user is responsible for ensuring that `eprof_init` is properly called before using the event-profiler and that `eprof_on` is used to enable it.

4.5 Analyzing the Event Profile - `etool`

Graphical analysis of "events"

As has already been alluded the event profile is analyzed with the `etool` utility. To execute this command one types

```
etool log_file_name
```

where the last argument is the name of the file containing the event log. If this has the default name "eprof.out" then it can be omitted leaving the extremely simple command line

```
etool
```

This command should result in a display that looks rather like Figure 9.

The different areas of the display are used for various purposes in manipulating and

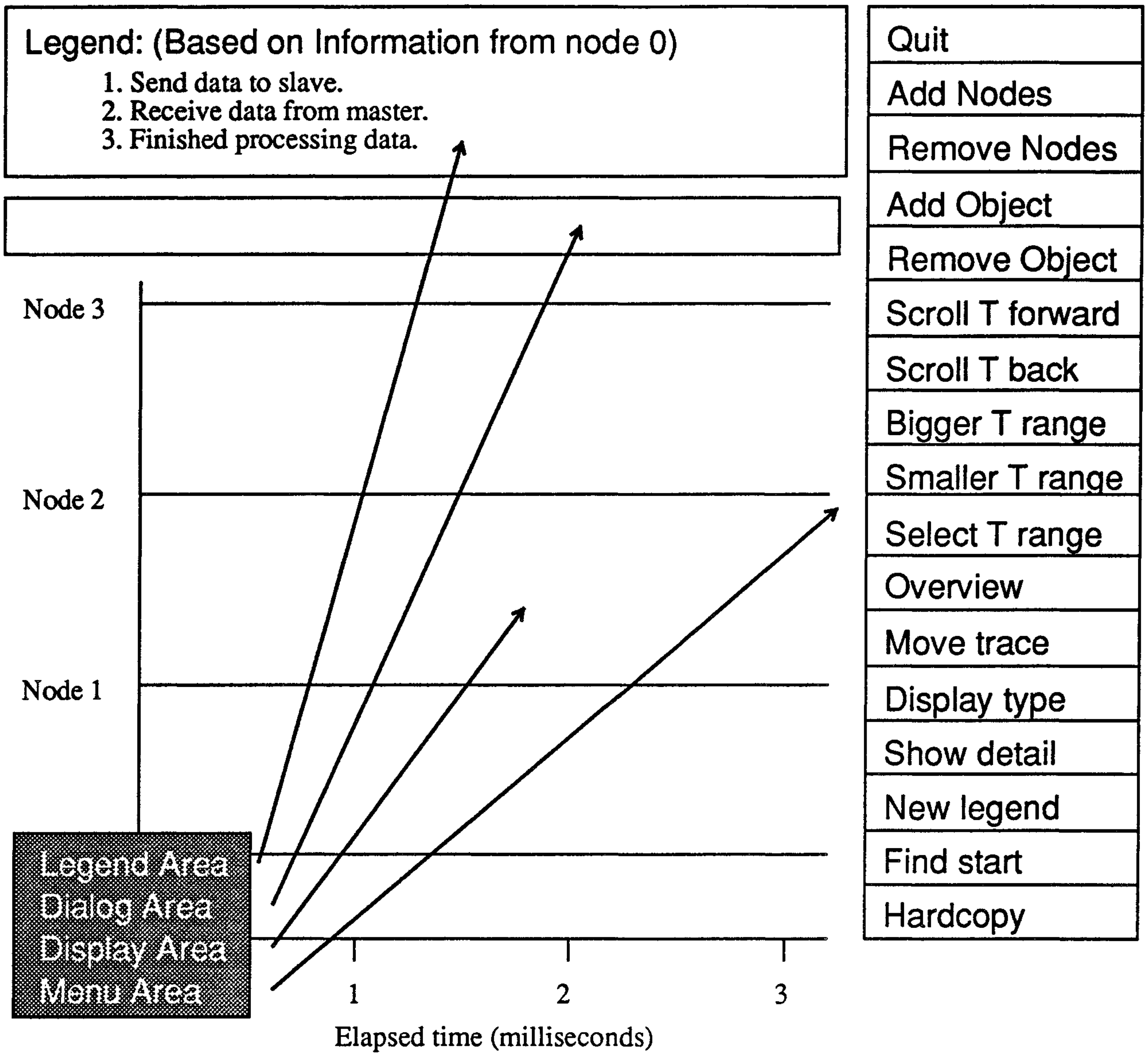


Figure 9. Basic Event Profile display

interpreting the event log:

Menu Area	This is usually the area in which user selections are made which cause various operations to be carried out in the event log display.
Display Area	This region contains the "time-lines" for the various processors. A simple horizontal line indicates "computing" activity while various types of colored and shaded boxes indicate user and system events.
Legend Area	This area is used to indicate the meanings of the various objects shown in the "Display Area". It usually contains an index to the

various event types defined in a particular node, although it can also be used to display a "key" to the encoding of system events.

Dialog Area	This region is used to interact with the user. Prompts for user actions are displayed here as well as information concerning events displayed on the screen.

Manipulating the time lines

To manipulate and analyze the display one selects from the options in the "Menu Area" on the right using the cursor. (The various types of mouse and their use is discussed in a previous section.) Some of the more obvious selections concern the amount of "time" displayed on the horizontal axis. By default `etool` begins by showing approximately 3 milliseconds of elapsed time. Often there will be no interesting events in this range as typified by the dull display of Figure 9. The most naive things to do to correct this situation are the various "T-range" commands

Scroll T forward Scrolls the time-lines forward by half the current width. Thus, if the current display goes from 3 to 4 milliseconds then scrolling would alter the range to 3.5

Scroll T forward Scrolls the time-lines forward by half the current width. Thus, if the current display goes from 3 to 4 milliseconds then scrolling would alter the range to 3.5<T<4.5.

Scroll T back Scrolls the time-lines backward by half the current width.

Bigger T range Doubles the range along the horizontal axis while keeping the start point fixed. If the originally display were from 3 to 4 milliseconds then this command would yield the range 3<T<5.

Smaller T range Zooms in on the time axis by halving the current range while keeping the start point fixed.

Select T range This option allows you to pick out an interesting range from the current display with the cursor. Immediately after selecting this option the message

```
Select lower time limit
```

will appear in the Dialog Area. You should then move the cursor into the Display Area and click when it is at the lower limit of some interesting range. At this point the prompt in the dialog area changes, asking you to select an `upper limit` and the process is repeated for an upper limit. When this has been selected the time-lines will be redrawn with the horizontal axis displaying the selected range of time values. Note that the range selection takes place *inside* the Display Area and not on the time axis itself. This allows you to pick out interesting *objects* from the display as the guide to an interesting time range rather than having to trace down to the horizontal axis to make the selection.

These options may be used at any time and merely manipulate the data on the display. They are useful when either too much or too little detail is being displayed and one needs to either "zoom in" or "zoom out" a little in order to make sense of the events being shown.

Getting the whole picture

Of course these options might not help much if your events are widely spread or start after a lengthy period of program execution. To cover these possibilities two further options are provided

Find Start As its name implies this option is used to "get going". After selecting it you will be prompted in the Dialog Area to

```
Select a node
```

To do this move the cursor into the Display Area and click over one of the traces on the screen. This "selects" that node and, in the current context, looks for the first enabled event in that processor and resets the horizontal range so that this event is displayed.

Overview This option draws all the time-lines in the selected processors from start to finish. User specified events are indicated by vertical bars rather than their full symbols to conserve space. (See Figure 10 for an example of the output produced by this command). This is often a useful option to select first followed by **Select T Range** to pick out interesting areas for finer scrutiny.

Dealing with large numbers of processors

These commands are provided to display various regions of interest. Another necessary ability is that of adding more processor time-lines to the display and possibly removing ones already present. By default `etool` displays the traces of the first eight processors encountered in the log-file. This situation can be altered with the **Add Nodes** and **Remove Nodes** commands. Both present a node selection menu which has the same form as that discussed in connection with the `ctool` system. One can select nodes either individually, in ranges, according to their neighbors, according to their "parity", or all at once. Having selected some nodes the **Done** option takes you back to the main display menu and performs the requested action with the selected nodes. If this was **Add Nodes** then additional time-lines will be added to the display for each selected node, while the opposite **Remove Nodes** option will remove traces for selected nodes.

The option **Move Trace** is available to re-order the time-lines along the vertical axis. By default this ordering is in order of increasing processor number or selection. Occasionally, however, it is convenient to put together certain traces to better understand the relationships between processors. To do this select the **Move Trace** option when you will be prompted in the Dialog Area to

```
Select a trace to move
```

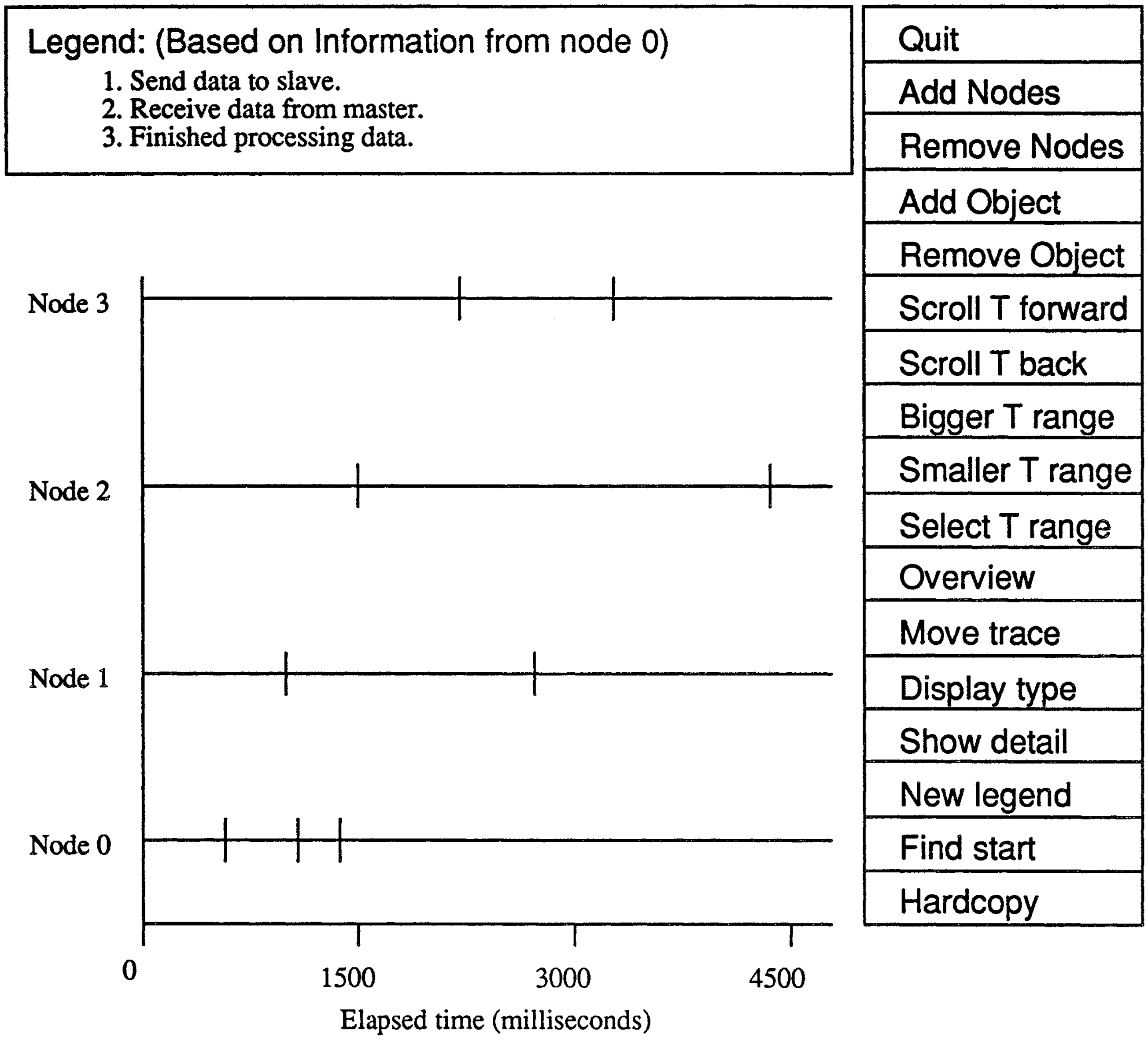
To do this move the cursor into the Display Area and click over a processor time-line. The prompt then changes to

```
Select a position to move it to
```

Selecting objects to watch

at which point you should click in the gap between two processor traces. The display area will be updated with the selected node trace positioned between the two indicated processors.

As well as altering the various quantities displayed along the axes of the Display Area one


Legend: (Based on Information from node 0)

1. Send data to slave.
2. Receive data from master.
3. Finished processing data.

Figure 10. Sample output from the "OverView" command

can also modify the appearance of the time-lines themselves with the following options

Remove Object This option allows one to selectively disable various types of "events". To disable an object click on **Remove Object** and you will be asked to click over the object you wish to delete. Move the cursor into the Display Area and click over any displayed object. Henceforth objects of this type will no longer be displayed or eligible for "detailing". When you have finished removing objects click on **Done** to return to the main menu.

Add Object This option reverses the effect of the previous choice. Upon selection a menu will be displayed containing the objects which have been disabled. Click over the items you wish to enable and then on **Done** to return to the main menu and update the display.

Adding system events to the display

These options allow piecemeal addition and deletion of specific events from the display. The **Display Type** option, however, allows sweeping alterations to be made to the time-line display. By default only user defined events are shown - i.e., those which were explicitly logged with the `eprof_add` system call in the application code. These events are represented by numbered boxes on the display lines, the numbers indicating the event "index" as given in the `eprof_add` call. However, the system is also potentially logging events, particularly communication calls. This option allows the "display type' to be modified to one of three styles

User Only	This is the default and shows only explicitly defined user events.
Comm. Only	This option disables all display of user events and instead shows communication calls in each processor.
Everything	This choice enables all events.

The effect of choosing the **Everything** option is shown in Figure 11. As can be seen the communication events are indicated by long narrow bars which are coded according to the particular function called.

Having added extra data to the Display Area one might be interested in figuring out what they represent. To do this one invokes the **New Legend** option. By default the Legend Area shows the "titles" that were assigned to user events in node 0 with the `eprof_label` system call. This information is often enough to understand all user events - their index numbers appear in the boxes on the time-lines and the associated titles appear in the Legend Area. Sometimes, however, the same event number might mean different things in different processors. While this might be classed as bad coding practice it may be unavoidable in real applications and so the **New Legend** option allows you to switch to a different node's set of titles. To do this click on **New Legend** and at the prompt

```
        Select a node or click outside the Display Area
```

move the cursor into the Display Area and click over a processor's time-line. This will immediately switch the Legend Area over to that taken from the indicated node. An alternative possibility is to click outside the Display Area completely. In this case a legend is drawn indicating the coding of the system defined communication events. A sample of the "system legend" is shown in Figure 12.

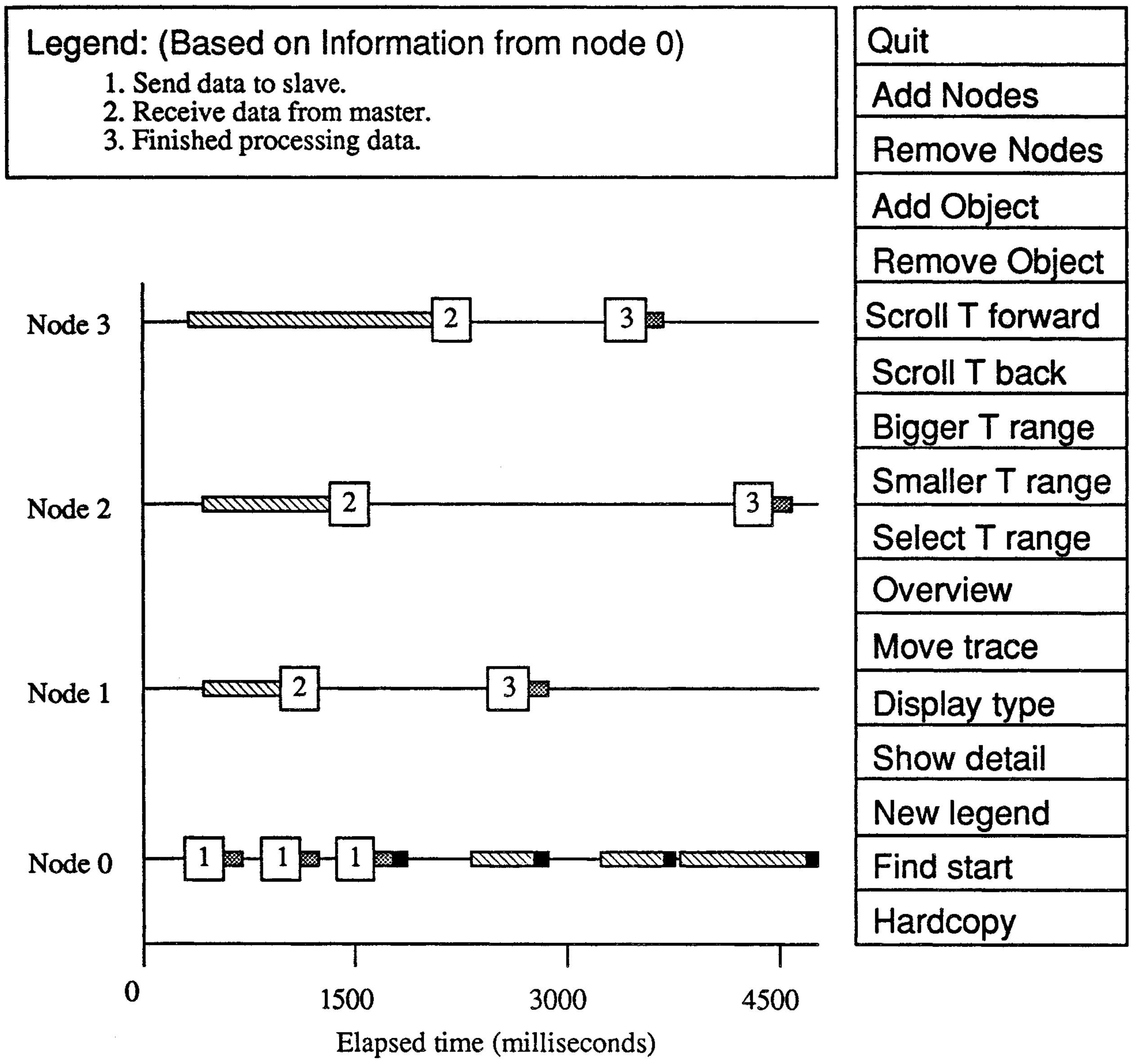


Figure 11. Sample display showing both system and user events

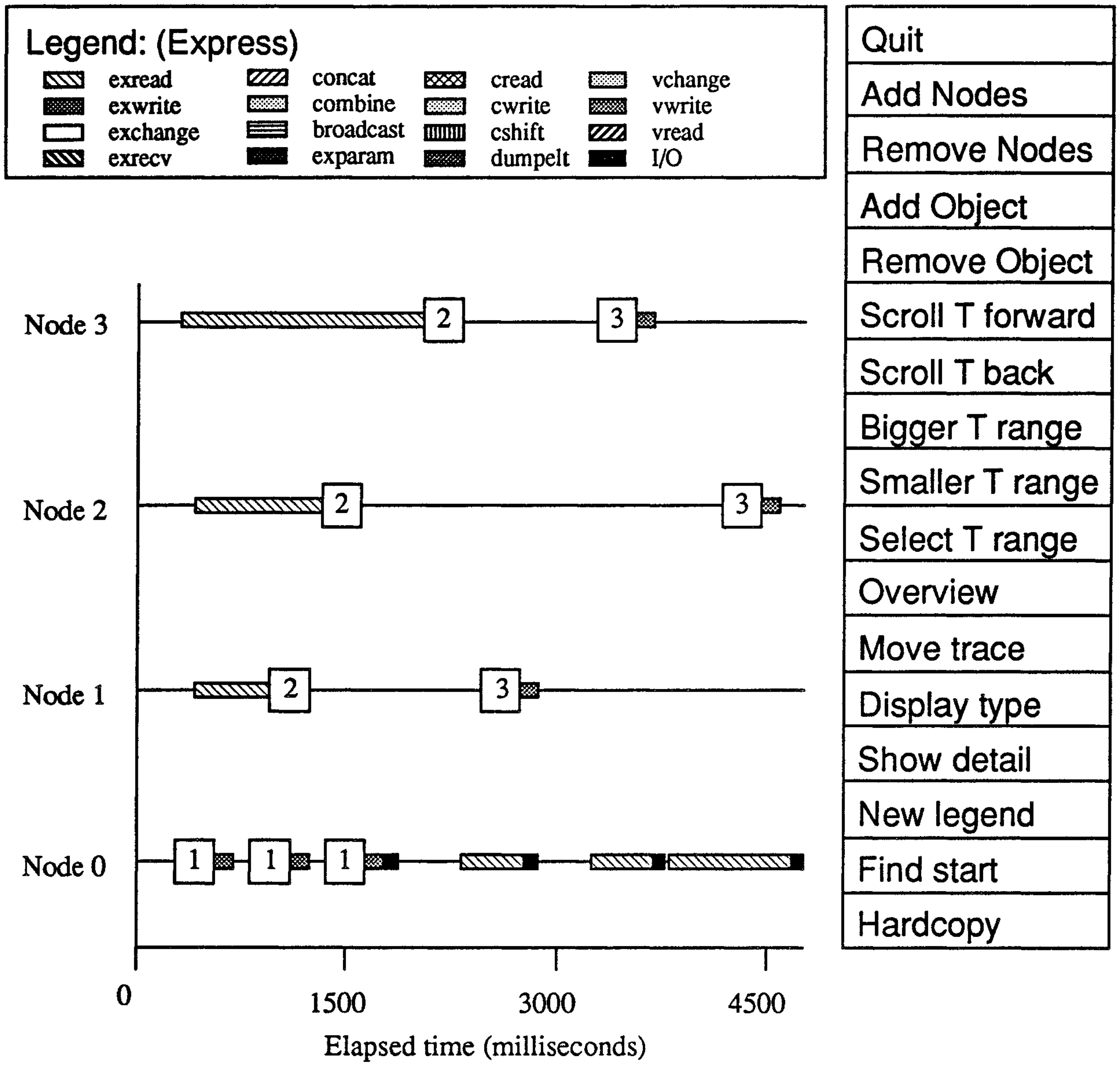


Figure 12. Legend identifying system communication events

Displaying the second level of information

This last command allowed more information to be displayed about the events shown in the Display Area. Usually this will be enough to get a reasonable feel for the part of the application being shown. In order to access the next level of information one uses the **Show Detail** option. Having selected this the menu changes to a single **Done** entry and one is free to poke around in the display area with the cursor. Every time that one clicks on an event in the Display Area extra detail concerning that event appears in the Dialog Area.

For user defined events the information shown includes the index number, the exact time at which the event occurred and the user data value which was supplied to the `eprof_add` call. The index number is provided to give confirmation that the correct event was actually selected with the cursor - this can get quite tricky in a crowded display although the commands to modify the horizontal axis can, in principle, be used to alleviate a dense time-line. The time of the event is shown to allow monitoring of execution times - for example if you have an event both at the beginning and end of a function then you can use this option to find out how long it took to execute. The final value is presented to either correlate the displayed data with a point in the application or to understand the way the program is behaving. The supplied data item is processed with the format string optionally supplied in a call to `eprof_label` and the result appears in the Dialog Area. In the first example of this section, for example, we defined labels containing `"Iteration %d"` and `"crunch returns %f"`. If we click on an event of the first type for which the supplied data value is 39 then the following might appear in the Dialog Area

```
        1. T = 246.23 ms, "Iteration 39"
```

Note that it is not essential to supply a label for an event type in order for this option to succeed. If no format string has been associated with an event then the result of the **Show Detail** click will just be

```
        2. T = 57.23 ms, value = -357832
```

in which the first two fields indicate the same information as previously and the last is the value of the user-supplied data item interpreted as an integer. Obviously this is not quite as informative as would be the case if a label were supplied, especially if the user data value is not an integer but a floating point value, but occasionally the space saving might be relevant.

Using the **Show Detail** function it becomes a simple matter to relate the events displayed on the time-lines to the underlying application algorithm which is the key to successful use of the event profiler. This type of information might also be used to understand the times indicated in the event profile. For example, one might supply a data item which represented the work-load in a processor. This might then be related to the event display by passing it to `eprof_add` and later picking it out with the **Show Detail** command.

While in this mode one can also click on system events. This should produce a message in the Dialog Area which looks like

```
    exread, T = 357.68ms, elapsed 127.54 ms, returned 580
```

The information supplied here is; the name of the communication function invoked (which should correspond to the system event legend if that is displayed), the time at which the communication began, the time taken for the communication to complete and the value returned to the caller. The interpretation of this last piece of information depends upon the particular communication routine invoked but is typically related to the length of the message being transmitted.

The last two options available on the main menu are, hopefully, self-explanatory. **Quit** terminates the `etool` program and returns you to the command line prompt. **Hardcopy** makes a file which, when suitably processed, will show the current state of the graphics screen, less the menus.

Using the event profiler effectively

Having now discussed all the options available to users of `etool` the question remains: "What can I do with it?". Among the various possibilities are

- Analysis of time taken in particular routines or pieces of code. Logging "events" around important code sections and subroutine calls allows one to evaluate the time spent in various portions of code.
- Relation of time spent to data conditions. Careful specification of crucial data items as the auxiliary value in `eprof_add` calls allows the connection to be made between program performance and data dependencies that arise on the nodes.
- Analysis of complex communication patterns and their effect on performance. Enabling the communication profiler while the event profiler is running logs extra information about internode communication.
- Analysis of interprocessor effects such as load imbalance and communication "skewing". It becomes immediately apparent if one node is working much harder than the others, or if a particularly crucial communication cycle is being delayed by another processor.
- Analysis of algorithms. In non-deterministic algorithms it may be useful to understand exactly, what functions are called and in what order. This can be achieved with suitable event placement. This type of information may be important in understanding the advantages or deficiencies of a particular parallel algorithm.
- Analysis of algorithmic behavior. In certain algorithms it may be important to understand the sequence of events leading to some strange behavior. A good example might be an ill-conditioned matrix problem in which the time taken for an algorithm to operate might depend on some parameter which can be logged and later related to the algorithm performance.

4.6 Analyzing the "Toggle" data - `etool -t`

Looking at the toggle data without graphics

The event profiling analysis tool, `eprof`, is also used to examine the data collected by the "toggle" system. To do this we execute the command

```
etool -p -t
```

in which the switches indicate that no graphics should be used ('`-p`') and that toggle data should be analyzed ('`-t`'). The resulting display will appear similar to that shown in Figure 13

For each "toggle" is presented the total time spent within that section of code, the number of times executed and mean and variance data. Also shown is the title given to the "toggle" in the call to `eprof_toginit`.

Using this system it is possible to build up extremely accurate pictures of program execution.

5 Example programs

The profiling system is supplied with an extensive collection of complete examples whose source code can be found on the distribution disk(s) in the `PM` subdirectory of the main *Express* `examples` directory. All the examples are based around the same piece of code

Node 0

Description	Total	#Calls	Avge.	Var.
Main iteration loop	478.32	100	4.78	.28
Calls to grind_away	363.96	400	0.91	.03

Node 1

Description	Total	#Calls	Avge.	Var.
Main iteration loop	478.32	100	4.78	.28
Calls to grind_away	363.96	400	0.91	.03

Figure 13. Sample output from the "toggle" utilities

which implements a basic "master-slave" approach to parallel processing, shown schematically in Figure 14.

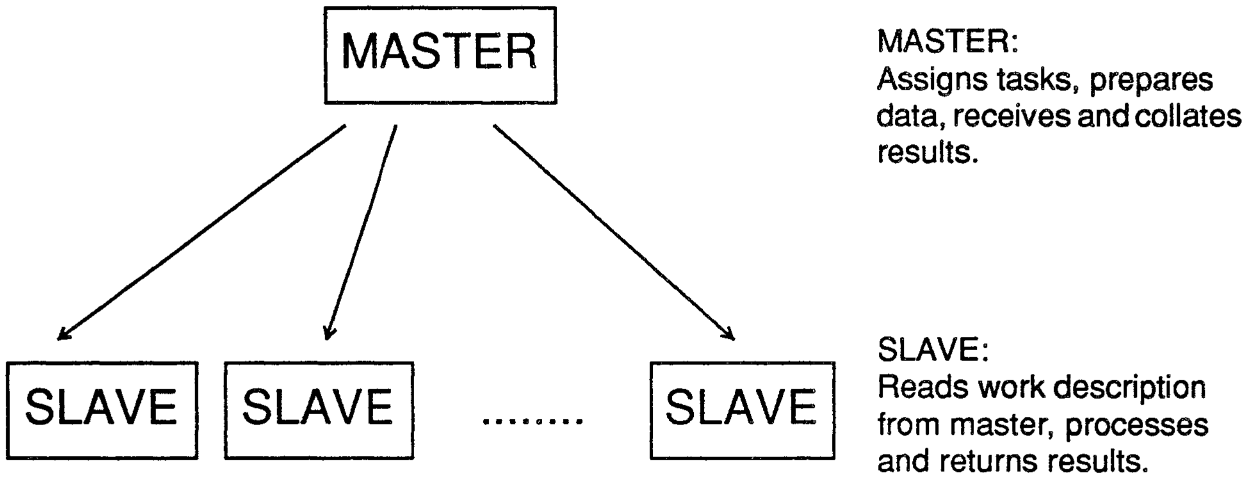

MASTER
MASTER:
Assigns tasks, prepares data, receives and collates results.
SLAVE
SLAVE
....
SLAVE
SLAVE:
Reads work description from master, processes and returns results.

Figure 14. Master-slave computation in parallel

"Master-Slave" programming on a parallel computer

In this programming style we nominate one processor as the "master" who is responsible for distributing work to a group of "slave" nodes. Each in turn receives a message describing a particular operation or operations to be performed from the master. It then performs all necessary calculations and returns the results to the master task. In the example program to be demonstrated here the specific details are quite trivial; the master creates a buffer containing a random number of random numbers and sends it to a slave node. The slave then computes the sum of the exponentials of the values in the buffer and returns this total to the master.

While this program is rather trivial it is actually quite a common programming model - many parallel applications take this form. Obviously an efficient implementation of this strategy would necessitate a larger workload per message; for example the current program could be sped up by having the slave nodes generate the random numbers themselves. However, it serves to illustrate several important areas where the profiling tools can be of help in detecting load imbalance and communication overheads.

Adding profiling constructs to an existing program

To this simple program we add the following profiling constructs:

- Enable the execution profiler.
- Enable the communication profiler for the entire course of the program.
- Initialize the event profiler with the default settings and turn it on.
- Add an event in the "master" task just before sending data to each slave. The additional data value indicates the node number of the slave to which data is being transmitted.
- Add an event in each "slave" to indicate the receipt of data from the master. The subsidiary data value indicates the amount of work received.
- Add an event in each "slave" to indicate that its work has been completed. The subsidiary item indicates the value returned to the master.
- Assign labels for the three events just described so that we will be able to find out just what was happening when the profile data is later analyzed.

Two distinct versions of this program are presented: a *Cubix* version and another split into host and node programs. We present the *Cubix* version first since it is simpler. It may be surprising to readers that the *Cubix* code is actually a fully parallel program since it looks so much like sequential code.

5.1 *Cubix* program

```
/*********
* master *       ParaSoft Corporation. 1988
**********

Master/slave program showing off the profiler. Should
exhibit some good "skewing" behavior.
*/
 #include <stdio.h>
 #include <express.h>
 #include <math.h>

 /* The next buffer is for the execution profiler. */
 char pbuffer[8192];

 extern float pranf();
```

```
#define NMAX (2048)
float msg_buf[NMAX];

main(argc, argv)
int argc;
char *argv[];
{
    long seed;              /* Random number seed */
    struct nodenv nodedata;/* Runtime parameters */
    int node, N, i;         /* Various counters */
    float total;            /* Cumulative exp total */
    int dattype = 2, acktype = 3;
                            /* Message types */

/* Set up the event profiler */

    eprof_init(DONTCARE, DONTCARE);
    eprof_label(1,"Sending data to node", "Node %d");
    eprof_label(2,"Received data from master",
                                "%d items");
    eprof_label(3,"Finished processing", "Result %d");

/* Set up the execution profiler */

    if(xprof_inq()) {
        profil(pbuffer, sizeof(pbuffer), main, 0x1000);
        xprof_on();
    }

/* Get runtime parameters, processor number etc.... */

    exparam(&nodedata);

/* Start up the random number generator */

    seed = (argc > 1) ? atoi(argv[1]) : 12345;
    pranset((long)seed);

/* Node zero will send up to NMAX floating point
 * numbers to each other node and then process them by
 * calculating their exponentials.
 */
    fasync(stdout);
    if(nodedata.procnum == 0) {
        for(node=1; node<nodedata.nprocs; node++) {
            eprof_add(1, node);
```

```
                N = (int)(pranf()*NMAX);
                printf("Sending %d values to node %d\n",
                                  N, node);
                fflush(stdout);
                for(i=0; i<N; i++) msg_buf[i] = 2.*pranf();
                exwrite(msg_buf, N*sizeof(float),
                                  &node, &dattype);
            }

    /* Wait for the acknowledgment to come back from the
     * nodes that have finished.
     */
            for(node=1; node<nodedata.nprocs; node++) {
                exread(&total, sizeof(float),
                                  &node, &acktype);
                printf("Node %d done with result: %f\n",
                              node, total);
            }
        }
        else {  /* Slave nodes. Get buffer and calc.
                   exponentials */
            node = 0;
            N = exread(msg_buf, NMAX*sizeof(float),
                           &node, &dattype);
            N /= sizeof(float);
            eprof_add(2, N);
            total = 0.;
            for(i=0; i<N; i++) {
                total += exp(msg_buf[i]);
            }
            eprof_add(3, (int)total);
            exwrite(&total, sizeof(float),
                                &node, &acktype);
        }
        exit(123);
    }
```

NOTE 1. The node program does not explicitly initialize or enable the profilers. This is taken care of by the *Express* kernel interpreting the runtime flags on the `cubix` command line.

NOTE 2. No calls are required to dump out profiling data. This action is performed automatically by the system. Note that no dumps will be produced if the profiling systems are not enabled - this is a much more flexible situation than in the host-node case since the user is free to choose at runtime between having and not having a profile produced.

NOTE 3. We assign node 0 the master role and relegate the others to slave positions. Note that we expect to find events of type 1 only in node 0.

NOTE 4. The code for the random number generator `pranset` and `pranf` is not shown.

5.2 Host-Node Program, "Host" code

```
/*******
* host *        ParaSoft Corporation. 1988
********

Master/slave program showing off the profiler. Should
exhibit some good "skewing" behavior.
HOST program.
*/
 #include <stdio.h>
 #include <express.h>
 #include <math.h>

 extern float pranf();
 #define NMAX (2048)
 float msg_buf[NMAX];

 main(argc, argv)
 int argc;
 char *argv[];
 {
     long seed;                  /* Random number seed */
     struct nodenv nodedata;     /* Runtime parameters */
     int Nnodes, node, N, i;     /* Various counters */
     int node_fd;            /* Process group index */
     float total;            /* Cumulative exp total */
     int dattype = 2, acktype = 3; /* Message types */

     Nnodes = (argc > 1) ? atoi(argv[1]) : 4;
     if(node_fd=
         exopen("/dev/ncube", Nnodes, DONTCARE) < 0) {
         fprintf(stderr,
             "Failed to allocated %d nodes\n", Nnodes);
         exit(-1);
     }
     exload(node_fd, "node");
/* Get runtime parameters, processor number etc.... */

    exparam(&nodedata);

/* Start up the random number generator */

    seed = (argc > 1) ? atoi(argv[1]) : 12345;
    pranset((long)seed);

/* The host will send up to NMAX floating point
 * numbers to each node and then process them by
 * calculating their exponentials
 */
    for(node=0; node<nodedata.nprocs; node++) {
        N = (int)(pranf()*NMAX);
        printf("Sending %d values to node %d\n",
                                N, node);
        fflush(stdout);

        for(i=0; i<N; i++) msg_buf[i] = 2.*pranf();
#ifdef SWAP
        _ex_swaw(msg_buf, msg_buf, sizeof(msg_buf));
#endif
        exwrite(msg_buf, N*sizeof(float),
                                &node, &dattype);
    }

/* Wait for the acknowledgment to come back from the
 * nodes that have finished.
 */
    for(node=0; node<nodedata.nprocs; node++) {
        exread(msg_buf, sizeof(float),
                                &node, &acktype);
#ifdef SWAP
        _ex_swaw(msg_buf, msg_buf, sizeof(float));
#endif
        printf("Node %d finished with result: %f\n",
                                node, msg_buf[0]);
    }
    cprofcp();
    eprofcp();
    exit(123);
}
```

NOTE 1. The host program is only slightly modified for profiling. At the end are added calls to the eprofcp and cprofcp functions which will read the profile data dumped by the node program. We have not instrumented this program for the execution profiler although that would be quite straightforward.

NOTE 2. The node program explicitly initializes the event profiler (eprof_init) and enables the event and communication profilers (eprof_on and cprof_on).

NOTE 3. The node program ends with calls to eprofelt and cprofelt which dump out data to the host. Notice that they must be in the same order in both host and node programs to avoid deadlock.

NOTE 4. The profiler is always enabled when this code runs. An extra feature which could probably be usefully added is to prompt the user for a value indicating whether or not the profiling systems should be enabled. These flags could then be passed down to the node program and used in "if" statements.

NOTE 5. The host processor plays the "master" role and the nodes are "slaves". This is a natural division of labor and avoids the long if loop in the *Cubix* implementation.

NOTE 6. The code for the random number generator pranset and pranf is not shown.

NOTE 7. Notice that the host potentially has to swap byte ordering when sending data to the nodes; a problem not encountered in the *Cubix* version of the code. Another related problem which might arise is the difference in length between data types on the host and in the nodes. In this case the latter problem does not arise and we control the byte ordering code with a preprocessor macro SWAP.

5.3 Host-Node Program, "Node" code

```
/*******
* node *        ParaSoft Corporation. 1988
********

Master/slave program showing off the profiler. Should
exhibit some good "skewing" behavior.
Node code.
*/
 #include <stdio.h>
 #include <express.h>
 #include <math.h>

 extern float pranf();
 #define NMAX (2048)
```

```
float msg_buf[NMAX];

main(argc, argv)
int argc;
char *argv[];
{
    struct nodenv nodedata;    /* Runtime parameters */
    int node, N, i;            /* Various counters */
    float total;           /* Cumulative exp total */
    int dattype = 2, acktype = 3; /* Message types */

/* Set up the event profiler */

    eprof_init(DONTCARE, DONTCARE);
    eprof_label(1,"Sending data to node", "Node %d");
    eprof_label(2,"Received data from master",
                          "%d items");
    eprof_label(3,"Finished processing", "Result %d");

/* Get runtime parameters, processor number etc.... */

    exparam(&nodedata);

/* The host will send up to NMAX floating point
 * numbers to each node and then process them by
 * calculating their exponentials
 */
    node = HOST;
    N = exread(msg_buf, NMAX*sizeof(float),
                           &node, &dattype);
    N /= sizeof(float);
    eprof_add(2, N);
    total = 0.;
    for(i=0; i<N; i++) total += exp(msg_buf[i]);
    eprof_add(3, (int)total);
    exwrite(&total, sizeof(float), &node, &acktype);

    cprofelt("cprof.out");
    eprofelt("eprof.out");
    exit(123);
}
```

Network Configuration

Using `Cnftool` to build multi- transputer networks for *Express*

1 Introduction

Topology changes

A significant feature of transputer based systems is the reconfigurability of the transputer links. Hardware systems may be "hooked up" in a variety of different ways and optimized for a particular problem. *Express*, by providing automatic message forwarding hides most of the details of the hardware interconnect from the user. On the other hand *Express* needs to know about the underlying hardware topology in order to perform correctly.

When to use cnftool

There are typically two times when the hardware configuration issues need to be addressed: setting up an initial system and optimizing a configuration for a particular application. `cnftool` is a *ParaSoft* utility designed to help in these areas.

Two types of transputer hardware

We can divide transputer hardware into two disjoint classes. In some systems (Definicon, Inmos, Microway, etc.) the links are implemented with mechanical cables which must be attached by the user. In this case `cnftool` provides tools to automatically explore a network and display the interconnections. It then allows you to modify parameters of the network at the software level. Any alterations made must be accompanied by similar reconnection of the link cables. The second type of hardware has electrically configured link switches which can be programmed to link the network in various ways. In this case `cnftool` provides a "graphical editor" for the network. You can indicate configuration changes which are then carried out by subsequent system initialization commands.

Performance enhancement

A second important use of `cnftool` is in conjunction with the *ParaSoft* performance analysis tools. The basic procedure would be to develop and debug an algorithm on some general topology and then to analyze its performance. On the basis of information gained one can reconfigure the underlying hardware network to minimize communication overheads. This procedure is completely transparent to the user since *Express* automatically takes care of message forwarding whatever the underlying machine connectivity.

This manual is structured as follows: In section 2 we describe the issue of "topologies" and their effect on the overall routing behavior. Section 3 is an extended example of the use of `cnftool` in the construction of a typical network with a single host computer. Section 4 builds this example into a multi-host system suitable for use as a multi-user resource. Section 5 describes some cosmetic touches which may be useful if extensive use of `cnftool` is planned. Section 6 describes some of the additional utilities which are included within `cnftool` and Section 7 presents a line-oriented interface to `cnftool` for those users without graphics. Finally, Section 8 describes the construction of standard networks from transputers and the underlying restrictions due to the limited number of links on each.

2 Topologies and Routing Strategies

"Deadlock"

While *Express* will run applications of any topology there are certain issues which should be considered when building transputer networks. The most important of these is the issue of "deadlock-free" routing. A simple example of "deadlock" is created when every processor sends out messages and no-one reads them. This will continue until all the "buffer" memory on the transputer is used up and then the network will "hang" - no more messages will be able to get in or out.

This is obviously a rather extreme case. "User errors" of this type are unavoidable -no system can hope to continue indefinitely under such circumstances. The problem to be avoided, however, is that in which the network "hangs" in regular use due to inadequacies of the underlying forwarding strategy. In this regard there are "good" and "bad" network topologies. Certain networks have well-known deadlock-free routing strategies which guarantee that only user errors can cause problems. *Express* understands three such topologies: hypercubes, two-dimensional meshes (torus), and trees.

Choosing topologies

The choice of topology is still a matter of taste. Certain algorithms may be able to take advantage of particular configurations. We have found that, even in the most obscure configurations, deadlock occurs infrequently. On the other hand it is quite difficult to detect and one can often waste considerable development time while searching for a non-existent bug.

For small size machines which will be used by single users the choice of topology is not very important. Most applications will run under *Express* without any problems. For bigger machines the choice of topology becomes critical since excessive forwarding can degrade algorithm performance significantly. It is even more important for machines which will be shared between several simultaneous users. In this case safe routing is necessary so that problems in one users' code do not affect other users.

A simple goal which enhances performance is to maximize network interconnectivity.

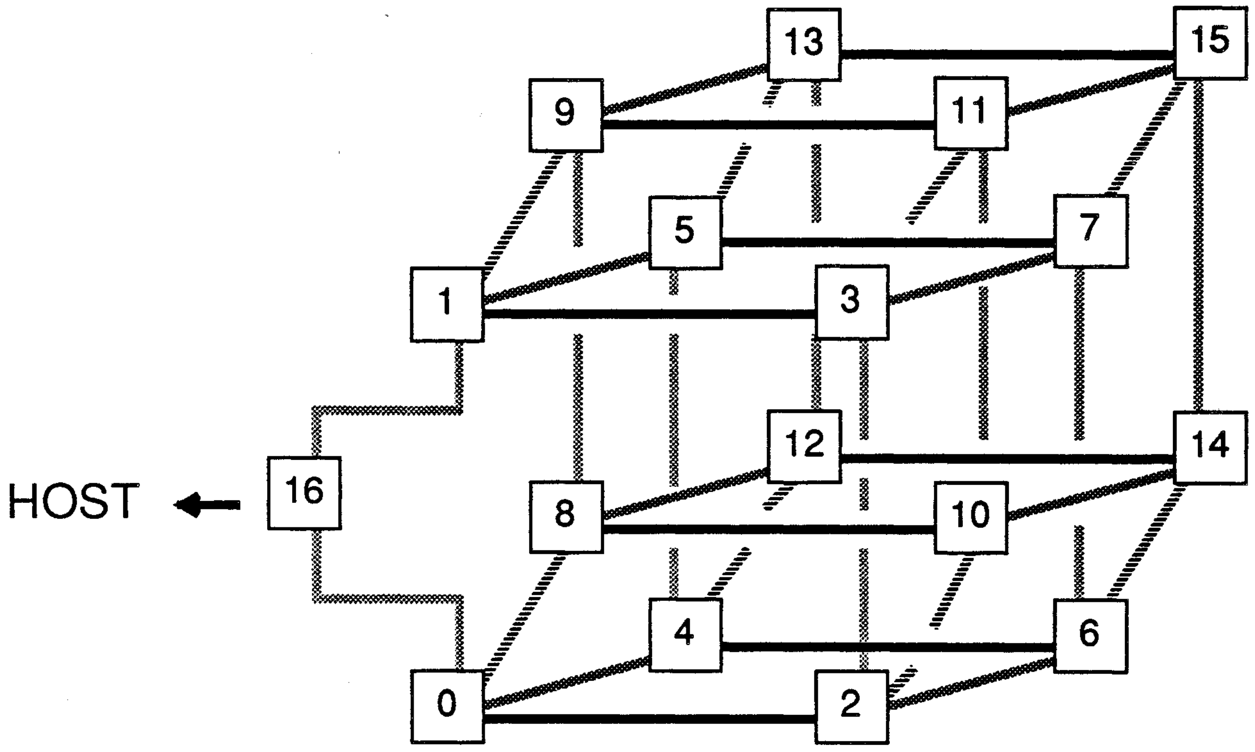


Figure 1. Dimension 4 hypercube with 17 transputers

From this point of view the most satisfactory network for small numbers of transputers is the hypercube. Since current transputers have only four links it is impossible to build large hypercubes - the maximum is the 17-node "hypercube" of dimension four shown in Figure 1. Note that we have to have an extra node so that the hypercube proper can communicate with the outside world. This topology provides extremely good connectivity and significant algorithm development has been performed on such machines.

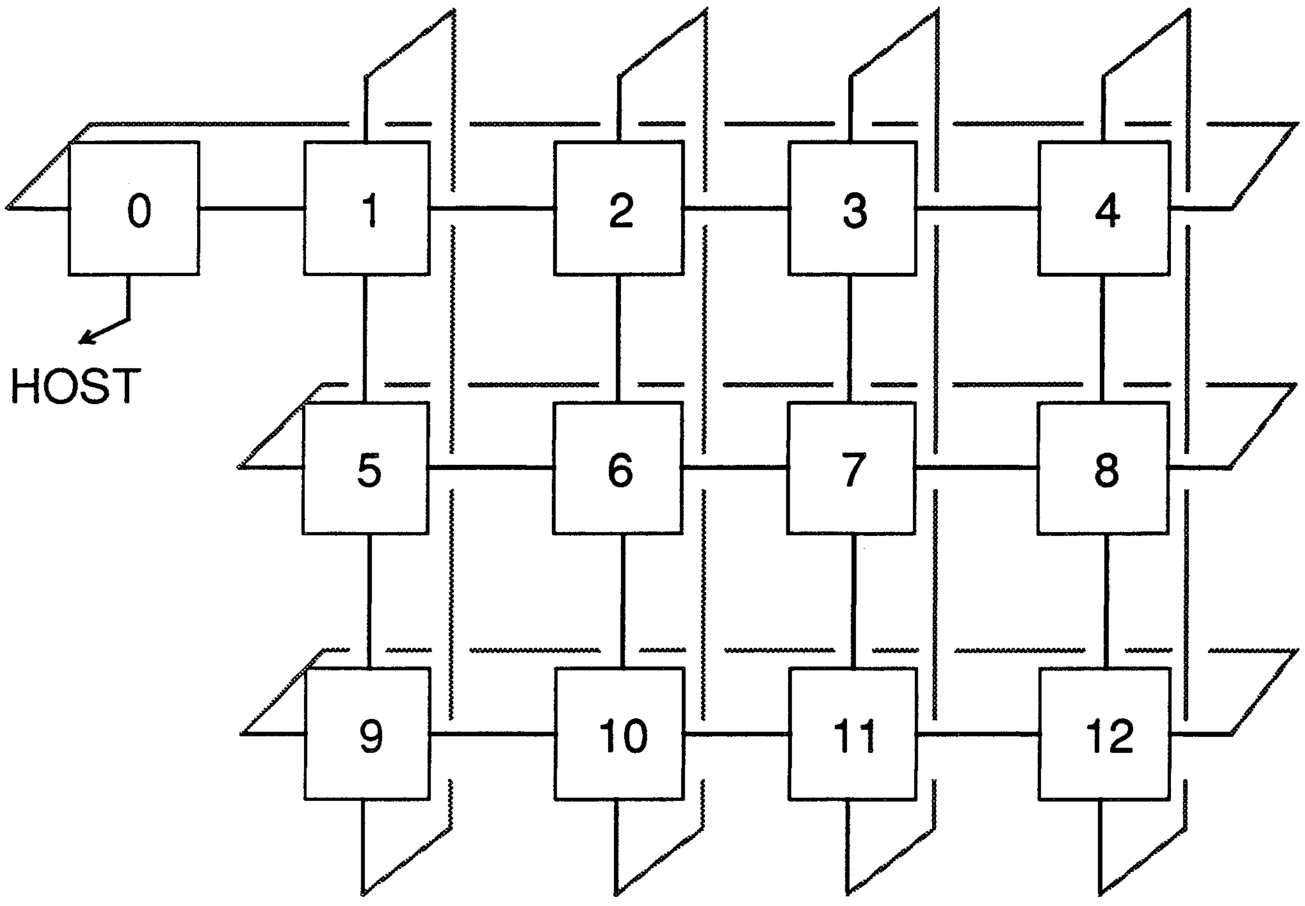


Figure 2. Torus Configuration

The second guaranteed safe topology is the two-dimensional mesh. A simple example is shown in Figure 2. Note that again we have a spare transputer to connect the mesh to the host. This network has the advantage of being completely extensible. Whereas the largest hypercube we can construct has 17 nodes a mesh can be constructed with an unlimited number of transputers.

The tree topology, shown in Figure 3, does not provide particularly high connectivity. It is, however, suited to certain applications and shares with the mesh the property of extensibility.

These are the networks on which *Express* is able to guarantee safe routing. If one of these can be tailored to your problem then all is well. `cnftool` will construct the necessary

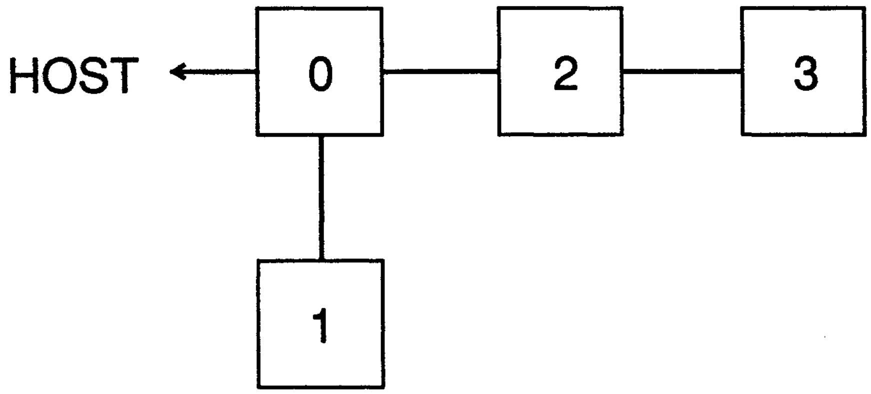


Figure 3. Tree Topology

configuration information quite simply. If, on the other hand, your network does not fit into any of these categories then cnftool will still be able to deal with it and *Express* will run applications but with the proviso that the underlying routing is not secure: programs which are logically correct may still exhibit "deadlock".

3 Configuring Simple Networks

Mechanical vs. electrical cofiguration

As described previously cnftool is a graphical utility that allows the user to build or make modifications to a transputer network. For systems with mechanical link connectors cnftool will be able to figure out the hardware connections in place and present them to you in an editable form. You can assign processor numbers, add extra host machines etc. If your system is electrically configured you can describe any network with cnftool which will perform the appropriate actions to initialize the hardware.

In this section we will discuss, in turn, the two types of hardware. To simplify the discussion we will consider the network shown in Figure 4. In this figure we have shown the processor numbers and also the links which should be attached. Note that no link number is given for the host - we assume that only one connection is available.

3.1 Machines with mechanical links.

Certain hardware manufacturers (e.g., Inmos, Microway, Definicon etc.) implement the transputer links with mechanical cables. In this case the first step in using cnftool is to attach the cables in some configuration. It doesn't matter at this stage if the configuration is optimal for your needs since *Express* will be able to use it anyway but you might want to connect up one of the topologies that guarantee safe routing.

Resetting any "additional" hardware

After this is done we must use cnftool to build the information necessary to *Express*. Before starting cnftool, however, it is necessary that all transputers, other than those on the first processor board be in the reset state.

To make this more solid consider again the network shown in Figure 4. If the hardware is a single Definicon or Microway board then nothing needs to be done prior to the execution of cnftool. The "worm" program will reset the entire network and detect all the nodes. If, however, the network is composed of, say, an Inmos B004 board in one machine and a Definicon or some other board in another then you will have to execute the single command

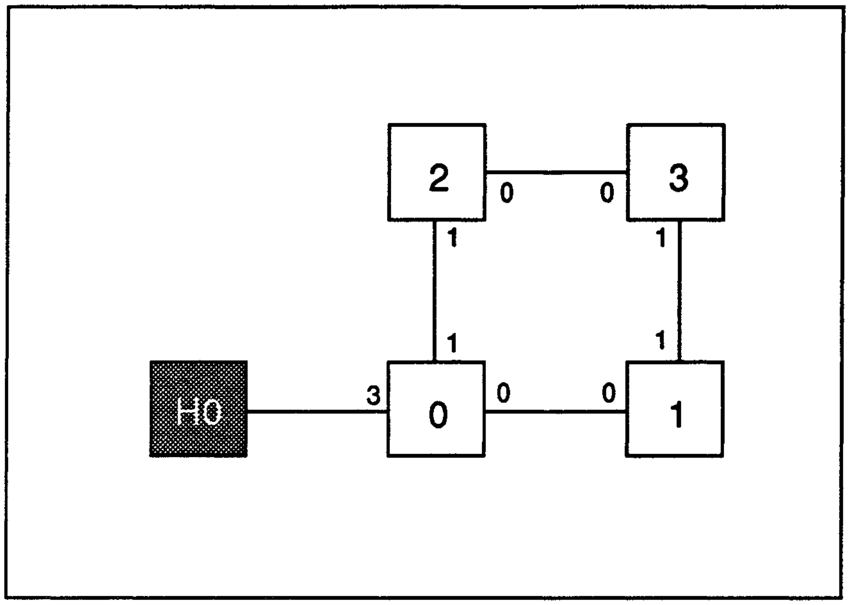


Figure 4. Sample Transputer Network

```
exreset
```

on each host other than the first. This procedure sounds somewhat complex but is quite simple in practice. Failing to reset some part of the system prior to executing `cnftool` is not fatal but will result in only part of the network being detected.

In connection with this point we note that with some hardware (Definicon, Microway) it is possible to chain the reset connections between boards so that each resets the next. If this has been done then no special commands have to be executed before `cnftool`

Running cnftool

When started, the first thing that `cnftool` does is to check for the existence of certain system files. The names and locations of these files depend on both your hardware/ operating system and also the way in which *Express* was installed on your system. The full pathnames of the relevant files can be discovered by looking up the values of the following customization variables:

`NIFFILE`	This file contains a description of the interconnection between nodes. Information is also given about the way in which the various "reset" lines are connected.
`CONFILE`	File containing the forwarding information that *Express* will use to send messages between nodes.
`PLOTFIL`	Describes the most recent image of the network as displayed by `cnftool`.

Note that *ParaSoft* reserves the right to alter the contents or names of these files in any future release of *Express.*

If these are detected the user will be asked whether to continue with the existing configuration or start again from scratch. If you are merely adding features to an existing network then you should elect to continue with the existing configuration.

The "worm" program

If you elect to start again from scratch `cnftool` starts up its "worm" program which will explore the transputer links and report, on the screen, a picture of all the nodes and links found. The first thing to do is to check that all nodes and links have been discovered. If this is not the case it probably indicates some problems with the associated hardware. If an entire board is missing it is probably because it wasn't reset - you should either reread the earlier paragraph about chaining reset signals together or else check cabling. Note one very important point in this respect; if you elect to chain together several reset lines you should ensure that all boards share a common earth. Otherwise the signal may not be recognized as a proper "reset" by some nodes. At this point the screen should contain an image such as that shown in Figure 5. (Note that we omit the transputer network from this figure for clarity.)

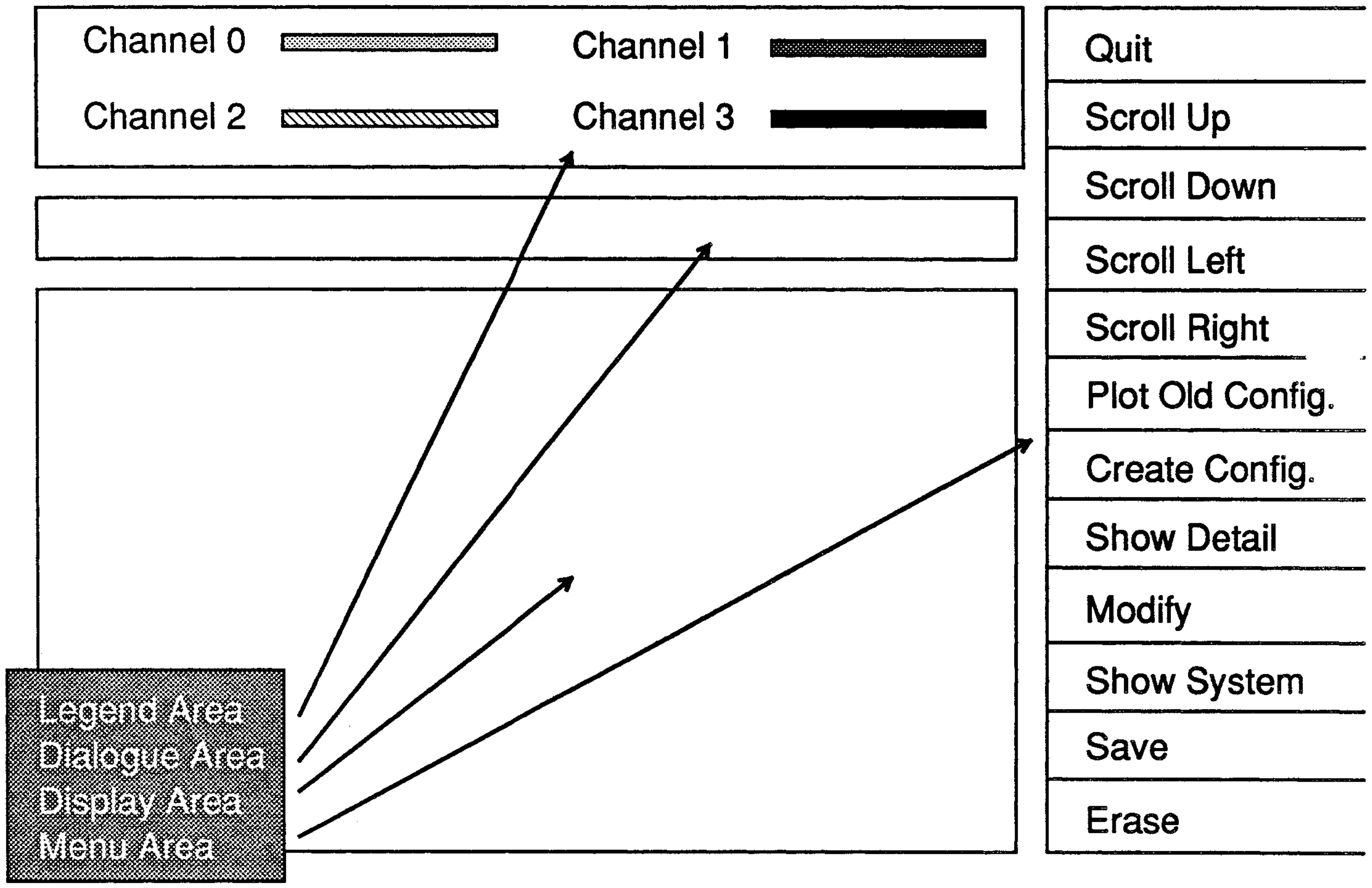


Figure 5. Basic `cnftool` display

Using cnftool without a mouse

The top part of the display contains a legend which describes the color coding of the different links. We adopt the convention that the links are numbered from 0 to 3. The right hand side of the display has a menu containing commands which are executed by "clicking". If you do not have a mouse then the cursor can be moved with the keypad arrow keys; two speeds of motion can be toggled with the "home" key. In the main display area the transputers are represented by squares with color coded edges. (On monochrome monitors it may be impossible to distinguish the various links - in this case one can observe that the "bottom" line corresponds to link 0 and the numbers increase in a clockwise direction.) Each edge represents one of the transputer links. "Hosts" are represented as red square with processor numbers which begin with the letter 'H'. Connections between processors are shown as white lines. The "Dialogue Area" shown in the Figure is used to indicate instructions and other informative details to the user. While manipulating networks with `cnftool` this area will contains details of how to perform certain operations.

After checking that all nodes and links are correctly shown it is necessary to create a "forwarding table" for *Express*. To do this select **Show System** in the main menu. (On a machine with a mouse move the cursor to the box marked `Show System` and press any button on the mouse. If you have no mouse move the cursor to the same box and press any key on the keyboard.) The screen should change to that shown in Figure 6. Selecting **Forwarding Table** in the system menu will produce the menu of Figure 7.

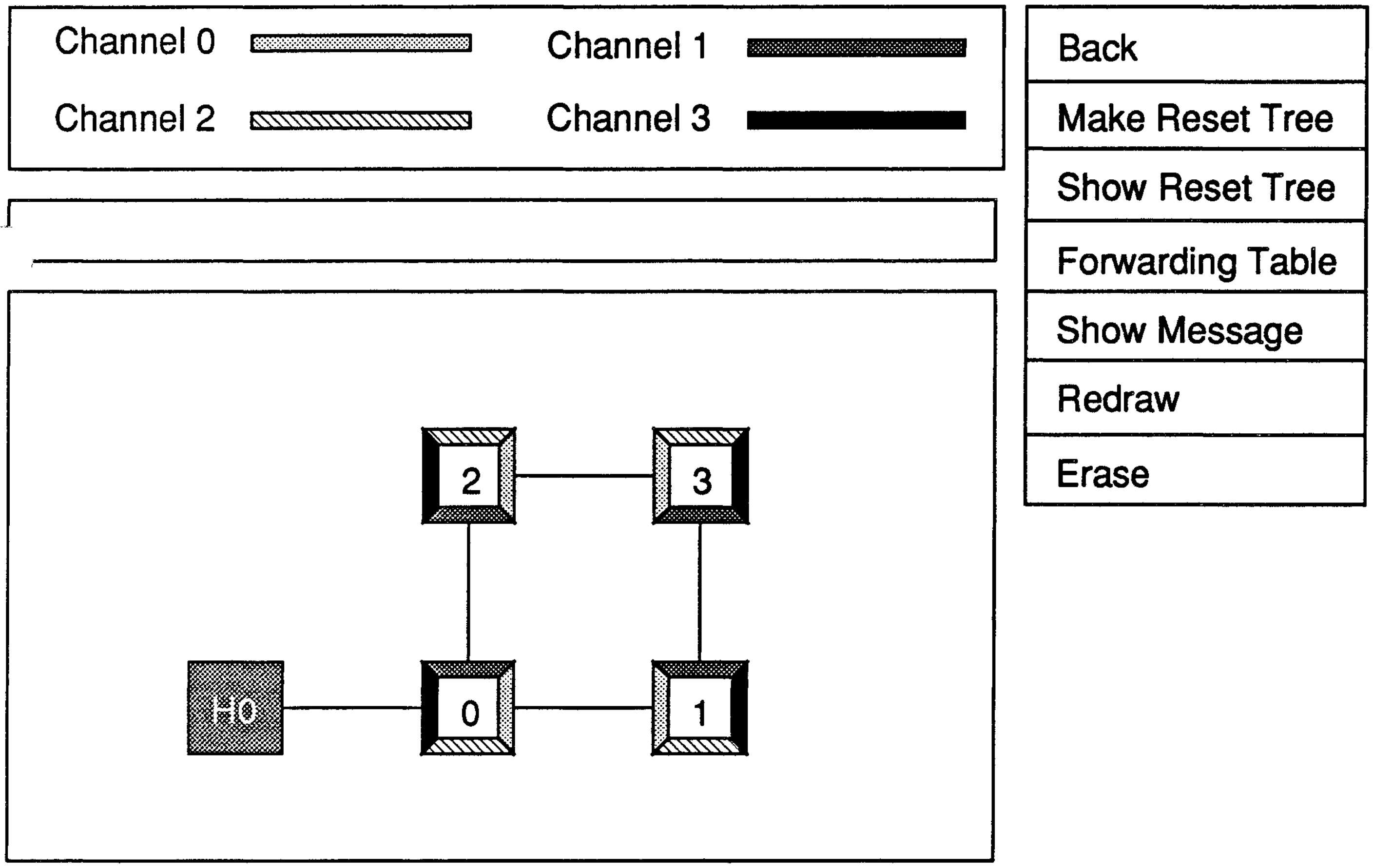


Figure 6. System Function Menu

Selecting a topology

At this point you have to decide whether or not your machine has one of the topologies that *Express* will be able to route safely. If so you indicate this fact by selecting the appropriate

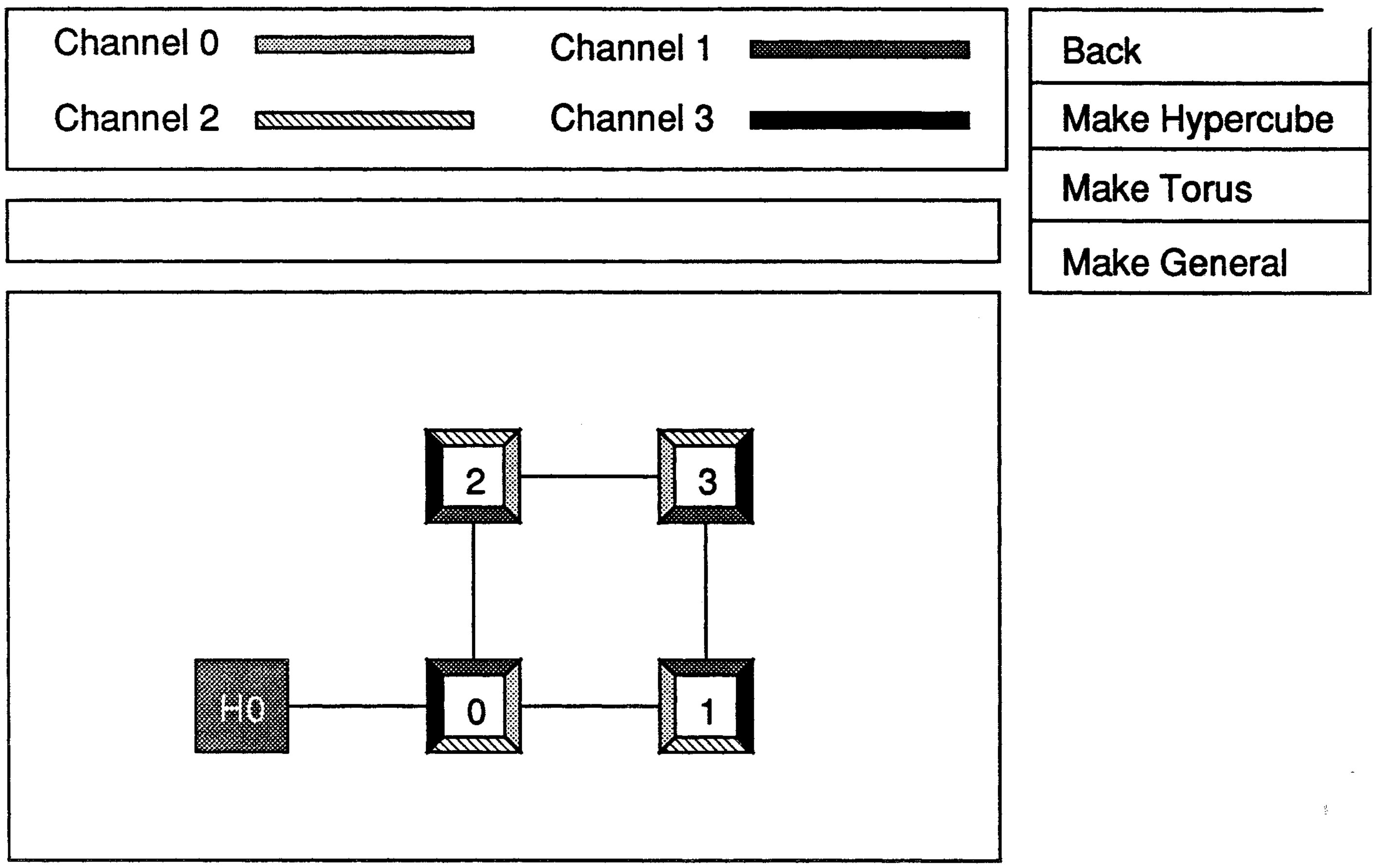


Figure 7. Hardware topology menu

item from the menu. (The tree topology is included in the **General** case.) As you do this the word `WORKING` should appear in the "Dialogue area" just below the legend. When the system has completed its configuration `WORKING` will change to `DONE`. At this point selecting **Back** from the menu will return to the previous level. Notice that you can always select the **Make General** topology. This will cause *Express* to use a minimum distance path when communicating between processors. Careful examination of Figure 7 shows that our demonstration network is actually connected as an 4-node hypercube so we could select **Make Hypercube** in order to guarantee safe routing.

The next thing which needs to be created is the "reset tree" - a path through the nodes over which the reset signal will propagate. `cnftool` can create this automatically by selecting **Make Reset Tree** from the menu. Again you should see `DOING` and `DONE` messages in the dialogue area.

The reset tree which has been generated will always be correct for both Definicon and Microway hardware but may not be correct for Inmos systems. The best solution in this case is to defer the connection of the hardware resets until after `cnftool` has created its path. At this point you can complete the connections according to the image generated by selecting the **Show Reset Tree** item from the menu. In this display the red lines indicate the path followed by the system reset while the blue lines show the subsystem reset. (Note that you can safely connect the reset lines while the machine is operating: `cnftool` will not attempt to interact with the transputers.)

Saving the configuration data

After generating the reset tree and connecting the reset connections, the configuration information should be saved. To do this click on the **Back** icon and go to main menu. In this menu select on **Save**. `cnftool` will save relevant information in system files such that further invocations of `cnftool` will be able to begin with everything set up correctly.

Loading Express

If your machine has only one host you may quit at this point by clicking on **Quit** and initialize *Express* with the command

```
        exinit
```

Even if you eventually intend to connect multiple hosts to your system it is a good idea at this point to check that everything is working correctly and that the machine can be initialized. When you are happy with the current configuration proceed to Section 4.

3.2 Electrically Configured Machines

Designing topologies on machines with electrical connectors - C004s

If your hardware has electrically configured links (Levco, Meiko, etc.) then a worm program should not be used to detect the current hardware configuration. Instead we can use `cnftool` as a graphical editor to construct custom hardware configurations. Even if your hardware has mechanical links it might be wise to read the description of this section since it covers several of the manipulations that occur frequently while using `cnftool`.

Having started up `cnftool` you may be asked whether to continue from some previously saved configuration or start afresh. In this section we will assume that we are starting from scratch.

The network description we will create consists of three elements: transputers, hosts and channels or links. The transputers represent the nodes of your machine and each has a unique processor number. Hosts represent computers like MS-DOS or UNIX PCs, SUN workstations, VMS machines, Macintoshes, disk servers and graphics servers, A/D converters etc. Links or channels represent the physical connections between transputers and hosts. Our convention is that a transputer is presented as a green square with colored edges and an identifying processor number. Hosts are represented by red squares containing the letter 'H' followed by the number identifying the host. Links are represented by white lines. The colored edges of the transputers represent the different links as shown in the legend. A host can have a link connected to any edge, but it can have only one link connected to it.

Creating a new configuration

To begin creating your new configuration select the **Modify** icon which will display the menu shown in Figure 8. There are three primary entries: **Modify Hosts** allows you to add, delete and move host processors in the configuration while **Modify Transputers** allows similar operations to be performed on transputer nodes. **Modify Channels** allows you to create and edit links between transputers and hosts.

To create the topology shown in Figure 4 we will begin by creating the host. Click on **Modify Hosts** and then **Add Host** on the subsequent menu. In the dialogue box will

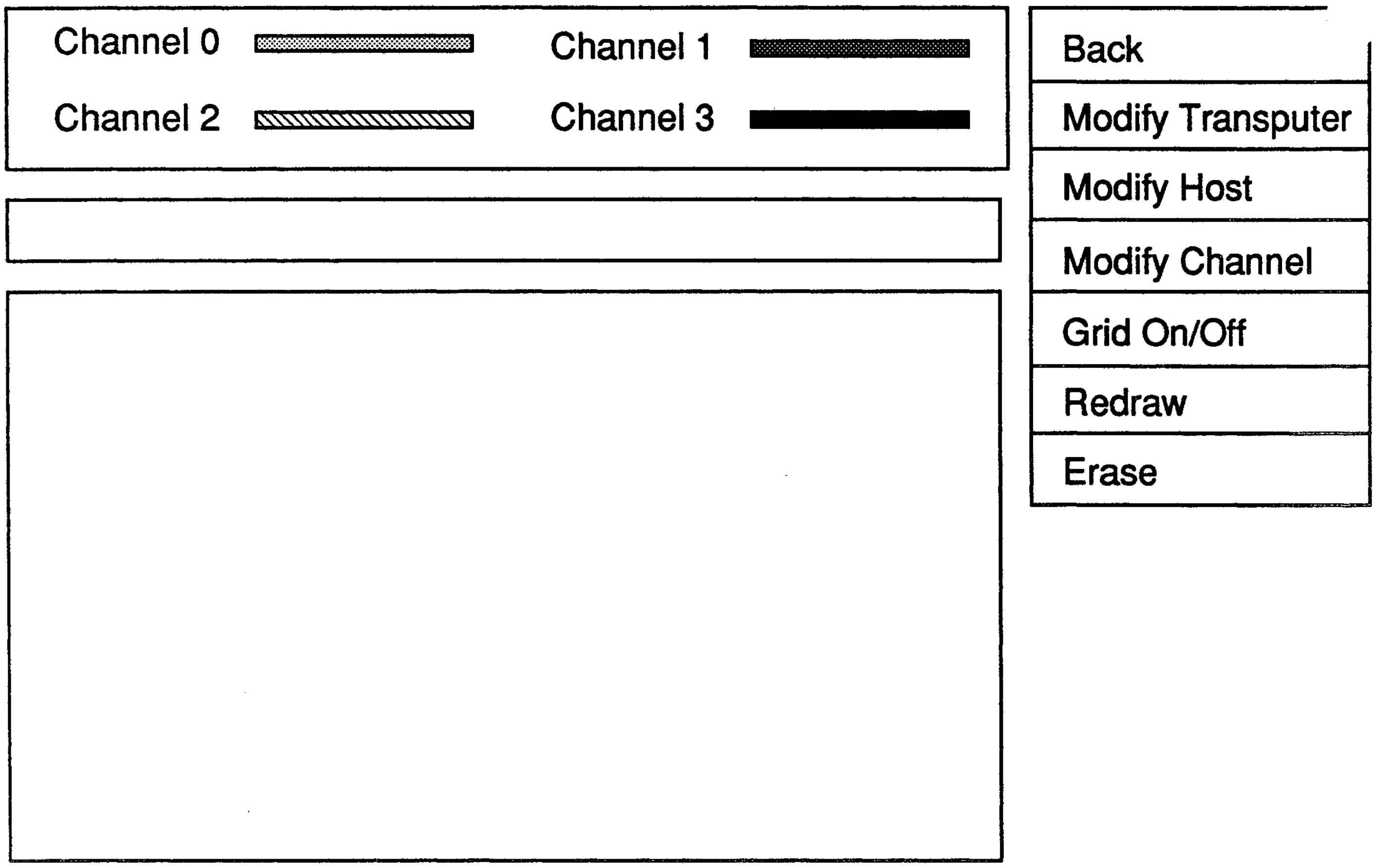


Figure 8. "Modify" menu for network creation and editing

appear the instruction to

```
        Indicate where the host should be placed
```

If we select, by clicking, a position such as that used in Figure 4 then the display should look like that shown in Figure 9.

Now we select **Back** to return us to the main menu.

The next operation should be the creation of the individual transputer nodes. Select **Modify Transputers** and then **Add Transputer**. The dialogue box will instruct you to position the first transputer which will be assigned the processor number 0. By default this node will be reset by "Host 0". To add successive nodes repeatedly select **Add Transputer**, placing them as shown in Figure 10.

While adding the transputers a new feature will become evident. After positioning each node on the display you will be asked to

```
     Please click on the transputer which resets this one
```

In most cases this can safely be ignored and any processor can be indicated. In certain cases, however, where the reset tree is explicitly connected at the hardware level you must be certain to indicate the correct node in the reset tree.

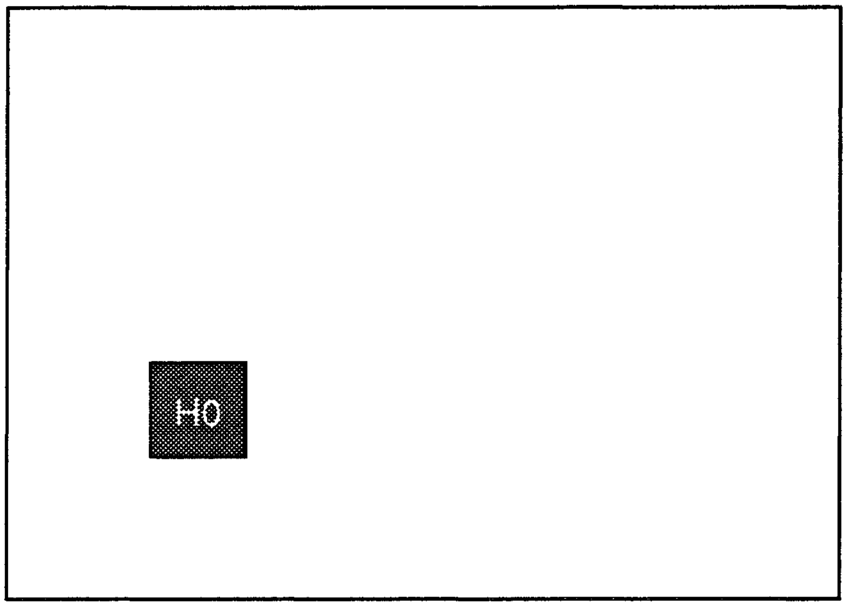


Figure 9. Adding a host to a transputer network

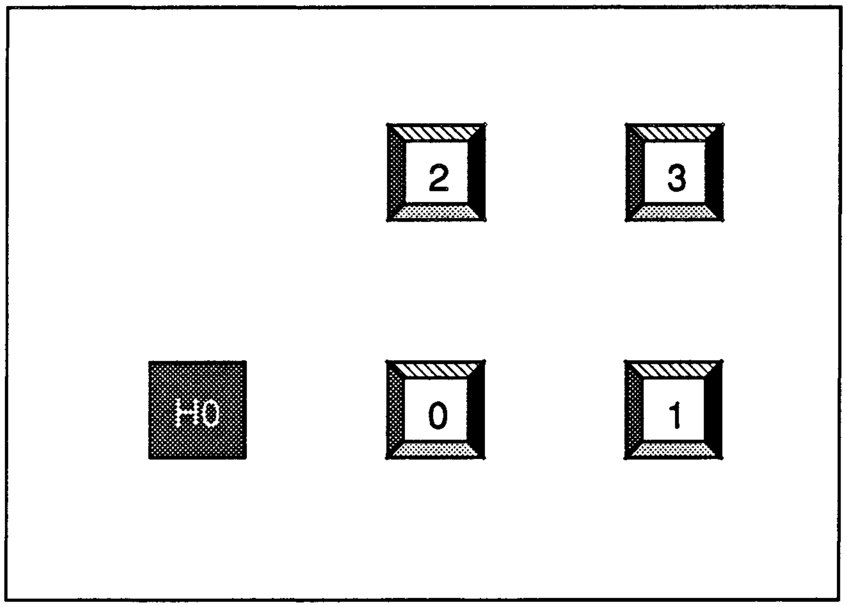


Figure 10. Adding transputers to a network

After creating all four transputers you should select **Back** to return to the main `modify` menu, Figure 8.

To complete our description of the system we need to connect the transputer links. To add links select **Modify Links** and then **Add Channel**. The first channel we will add is that between the host and transputer 0. After selecting **Add channel** the dialogue area will instruct you to

```
Click on transputer or host to start link
```

You may choose to click on either the host or transputer 0 to create this link. If you select the host the instructions will change to

```
Please click on channel to start link
```

. Click on the right edge of the host square. Next you will be asked to

```
Click on transputer to end link
```

Click on transputer 0. Finally you will be asked to

```
Click on channel where to end link
```

Click on channel zero of transputer 0; the red edge. If you did everything correctly the picture will appear as in Figure 11.

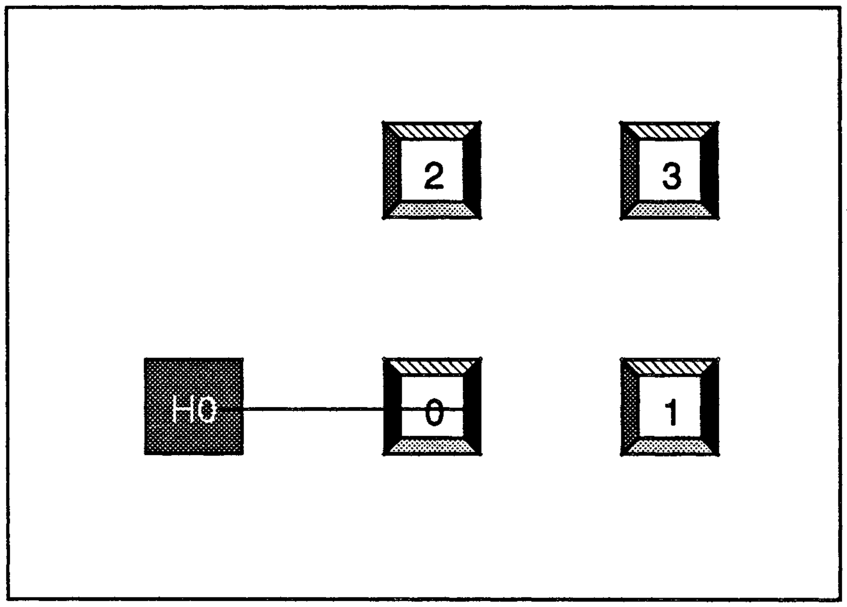


Figure 11. Adding a Host-Node link to a network

Accuracy with the mouse

If for any reason the phrase

```
Please press right button to exit
```

should appear in the dialogue area it indicates that you did not click near enough to something; try again.

cnftool's numbering algorithm

Having made a link between the host and the first node we can proceed similarly to make the other connections. If we continue to add the link between nodes 0 and 1, however, a slightly strange things happens. Notice that we have to be very careful in selecting the correct links according to the color coded legend at the top of the screen. After connecting link 0 of node 0 to link 0 of node 1 we see the display of Figure 12.

We notice that this doesn't look quite like the corresponding link in Figure 4 because the two nodes appear to be oriented incorrectly - link 0 is at the bottom of both transputers

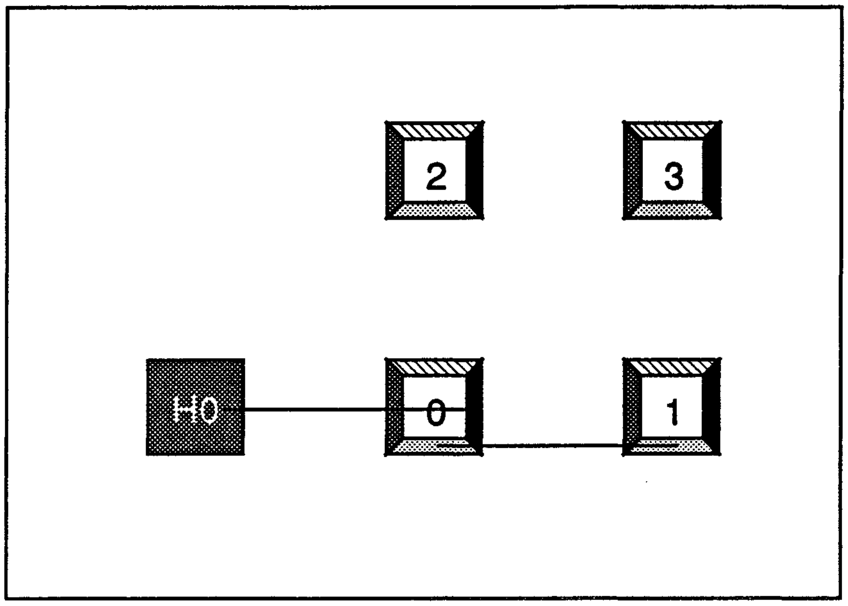


Figure 12. Adding a Node-Node link to a network

rather than on the "inside" where we want it to be. This is a fairly common problem - nothing is actually wrong with Figure 12 as it stands, but it looks messy. To tidy things up select **Switch Channels** from the menu. You will be prompted to indicate the transputer on which you wish to operate and then on the pair of channels that you wish to exchange. Swapping channels 0 and 3 on node 0 will make the picture shown in Figure 13.

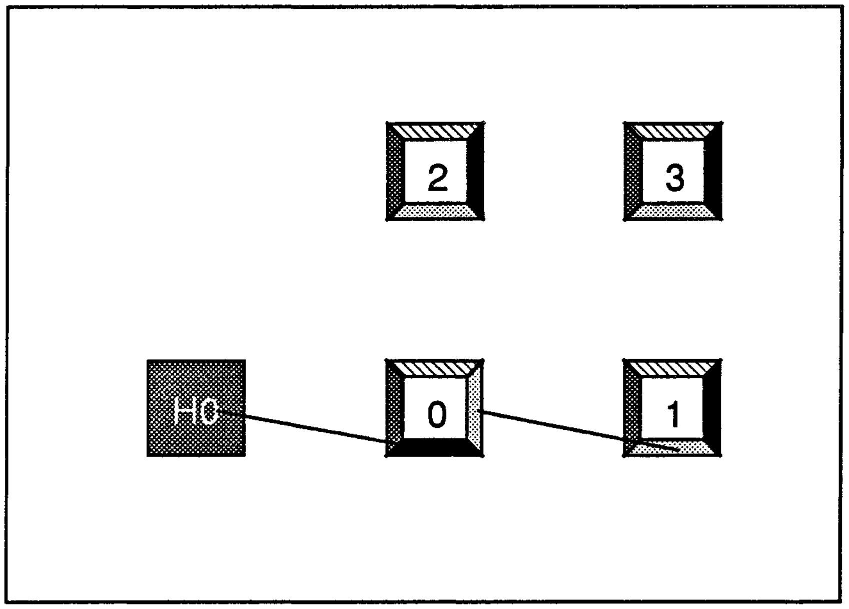


Figure 13. Swapping channels on a single transputer

Notice that we are now partway to Figure 4. Swapping links 3 and 1 on node 0 and links 0 and 1 on node 1 will produce the "nicer" picture.

Repeating the previous steps for the other links in the network should finally lead to a picture similar to that of Figure 4. We have now finished creating network components and can now construct various system structures such as the "forwarding table" and "reset tree". To do this keep selecting **Back** until you reach the main menu and then select **Show System**. The instructions of Section 3.1 should guide you through the relevant operations up to, and including, the `exinit` command which should start up *Express* in your newly configured hardware.

3.3 Minimal use of `cnftool`

The minimal cnftool procedure

Since using `cnftool` is rather important and the effects of mistakes are sometimes quite difficult to understand this section lists the minimal set of operations which are necessary for the successful operation of the system.

The easiest use of the system is obtained by giving the command

```
cnftool -p
```

which avoids the mouse interface entirely. You are still prompted as to whether you should start afresh or continue with the existing data but the "worm" program should find all the nodes and the only remaining question will concern the overall network topology.

If you wish to make use of the graphical interface the problems you may find will be connected with omitting some part of the `cnftool` procedure. The following list of commands should, therefore, be executed whenever you wish to terminate a session with the tool to ensure that the correct disk files are created and that all system information actually reflects the system you have been modifying on the screen.

Show System

Create Reset Tree

Create Forwarding Table

Make General

Back

Back

Save

If you follow this procedure whenever leaving `cnftool` you should never leave inconsistent system files. Occasionally, however, problems may arise which do leave the system in an inconsistent state. The most common problem concerns the "worm" which may not work properly on all types of hardware. In this case the image on the display may not be correct because the plotting file has not been created properly. (This file has the name given by the `PLOTIL` customization variable - see the documentation for `excustom` for more details.) A simple solution in this case is to merely delete the plotting file and restart `cnftool`. This forces the recreation of an image which corresponds directly to the information currently stored in the other system configuration files.

4 Configuring a Multi Host System

Express has a very general definition of a "host". Essentially any machine or board that is capable of talking to the main transputer array can be a host. In most cases this will be a workstation or personal computer running one of a variety of operating systems: MS-DOS, UNIX, XENIX, VMS, Macintosh, etc. This type of host provides fundamental operating system resources to the transputers such as terminal access, editing, printing, file systems and others. A second type of host is a "server" connected to the transputer network. These machines typically offer either disk or graphical services or external data streams such as from signal processing or A/D convertors and may, again, be almost any type of computer, including transputer based systems themselves.

The client-server model of processing

In the case of a multi-host machine one must be selected as the "master console". This will be the place from which the whole network will be initialized. The sole restriction on this choice is that it must be running a native operating system rather than a "server" process - it must be capable of executing the command which resets and reloads *Express*. By default this host will have the number 0 as indicated in the previous figures.

As an example of as multi-host system consider that shown in Figure 14. We assume that we have two MS-DOS PCs each with a four node transputer system. While the two systems could be used independently *Express* offers the capability to connect the systems together and share the parallel computing resource. To do this we have to add two additional connections as shown: a common ground and an additional transputer link. If there is no common ground the reset signal cannot be guaranteed to propagate from one machine to the other. The extra transputer link is the medium which will be used to communicate between the two halves of the system. Note that we could connect more than one additional link between the transputer boards thus increasing the communication bandwidth.

Hardware issues in connecting multiple transputer boards

Before connecting links on transputer products from different vendors it is important to be sure that the same link standards are being used. Most transputer manufacturers use standard TTL links. A few, such as LEVCO, for example, use RS434 links which cannot be connected directly to TTL types. Before connecting different vendor's hardware, therefore, it is a good idea to check that similar standards are in use and whether or not suitable converters are available.

For the sake of the next discussion we will assume that PC-1 has been nominated "master console".

To set up this system we can use `cnftool` to detect the hardware configuration with its "worm" program. Before doing this however we must consider the "reset problem" alluded to several times. In the simplest scenario we make no attempt to connect the reset lines on the two boards. We have to execute the

```
exreset
```

command on PC-2 to reset its nodes. We then start up `cnftool` from PC-1. If, on the other hand, the reset signals have been linked together we need not do anything on PC-2 and can go ahead with `cnftool` on PC-1.

As usual `cnftool` will look for an old configuration and, if found, ask you whether to proceed with it or start afresh. To make use of the "worm" you should choose to start again.

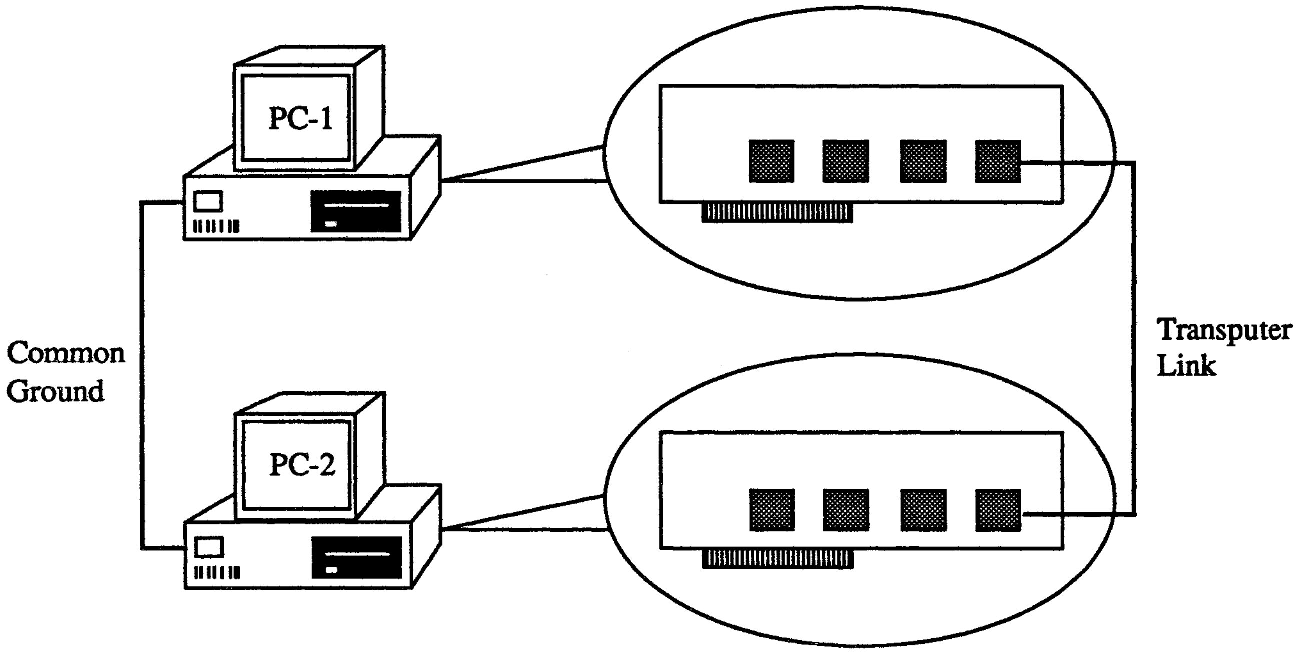


Figure 14. System with multiple hosts

After a couple of seconds your screen should display something similar to Figure 15.

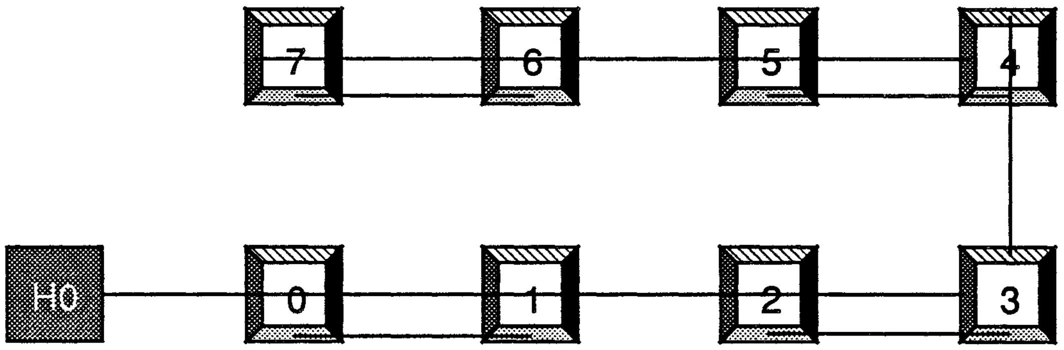


Figure 15. Multi-Host system as described by the "worm"

Troubleshooting systems with multiple transputer boards

If you do not see all the transputers connected to your network this may mean one of several things:

- Boards from different vendors may have incompatible link standards.
- The link connecting the two (or more) boards together is not connected correctly. In this case several transputers may be missing.
- The reset signal did not make it to some board. In this case an entire part of the

network will be missing.

- Some of the transputers are defective. Sometimes individual nodes are missing which probably indicates defective hardware. Occasionally mismatched link speeds also cause this problem. (We nearly always generate configurations with all links set to 10 Mbit/sec. When the network is working correctly the speeds are carefully upgraded.)

You should make sure that all links are correctly connected and that the above mentioned precautions have been taken with the "reset" lines.

Note that Figure 15 does not contain the second host, PC-2. We have to add this manually by the procedure described in the previous section: select **Modify**, **Modify Hosts** and **Add Host** and then follow the instructions from the Dialogue Area. Be careful that you connect the host to the correct transputer and link! It is quite easy to make mistakes in this area since the "worm" program may not have numbered the processors in the same manner as you expect - particularly if you are connecting two systems that used to be used separately.

After adding all necessary host/links/nodes by hand you must construct forwarding tables and a reset tree in the manner described in Section 3.1 It is probable that your network topology selection is now **Make General** - even if you began with two four node hypercubes it is unlikely that the combined system has the correct configuration for an 8-node hypercube. (You can, of course, construct this network by hand if you need safe routing.) One thing which may be useful is the **Show Reset Tree** option. If you began working by manually resetting the various boards with the `exreset` command then you can have `cnftool` build and show you a "reset tree" that you can then hook up with the appropriate cables. Then you will be able to reset the system from the "master console" without having to `exreset` the other hosts.

After completing all the necessary book-keeping tasks you should return to the main menu and **Save** the current configuration. Now you can exit from `cnftool` and try to initialize your machine from the console by typing

```
exinit
```

If your network refuses to load *Express* properly the most common source of error, we have found, is with the reset signals. This is, unfortunately, a rather tricky area and varies quite a lot with the particular hardware you are using. If your problems persist give us a call.

Adding and debugging the connection of a second host

The last step of the configuration procedure is to check that the second host understands the network. To check this try typing

```
exstat
```

from the console of PC-2. It will report the number of nodes and hosts in the system. In our case we should see

```
Total nodes: 8, Allocated 0,
Number of hosts in the system: 2.
```

If all is well your system is now running *Express* on all 8 nodes with two hosts. The benefits for user programs include:

- Twice as many nodes to pick from: we can run on up to 8 nodes at once or merely share space with another user on the other console. If 6 nodes are already allocated, for example, we can use the other 2.
- Transparent access to the file systems on either host through the `cubix` commands and servers. We can redirect operating system requests to either host - even if it is executing an entirely different operating system!
- Use of either machine as a graphics server. We can run from one PC and redirect our graphical output to the other.
- Debugging from either console. If we have an application that, for example, makes use of the *Plotix* system for graphics it will be difficult to debug with `ndb` on one PC since it will overwrite the display with pictures. We could, however, run the debugger from the other host leaving the graphics alone.

5 Cosmetic Improvements

Making cnftool clean up its image

While logically correct the network shown in Figure 15 is quite hard to understand - several links track horizontally across the picture and it is not clear where one finishes and the other begins. `cnftool` contains facilities for improving the legibility of the image. Each of the individual "Modify" menus has already been used to create new network components. You may have noticed, however, that each also contains commands to "move" or "delete" objects. The former command ask you to select an object to move and then lets you select new positions for it in the Display Area. No configuration changes are made. The "delete" options, however, do cause configuration changes in that links to removed nodes/hosts are discarded.

As well as moving hosts and nodes around there is another facility for altering the appearance of the links. The most common problem has already been addressed - you can "swap links" as shown in Figures 12 and 13 to improve the display. Another common problem which is not as easily solved is that shown in Figure 15 - some of the links lie across the nodes making it unclear who is connected to what. To alleviate this problem you can make "nodes" in the links - i.e., bends in the lines. You can then make the links move away from their straight line paths allowing easier understanding of the network. Simple combinations of these two procedures yielded Figure 16 from which the underlying network is much clearer.

Note that you should **Save** the enhanced picture in the main menu before exiting `cnftool` so that you can add more features to the network by starting from the cleaner image.

6 Displaying Routing Information

Examining routing data

Since `cnftool` knows all the details of the routing strategy on a particular network it can be used to show the paths taken by messages through the machine. This can be both a useful debugging tool and also a guide to enhancing performance by alleviating obvious message bottlenecks. The normal procedure in such a case would be to develop an algorithm on some simple network and then use the *ParaSoft* profiling tools to examine possible sources of inefficiency. If the network topology is considered inadequate then `cnftool` can be

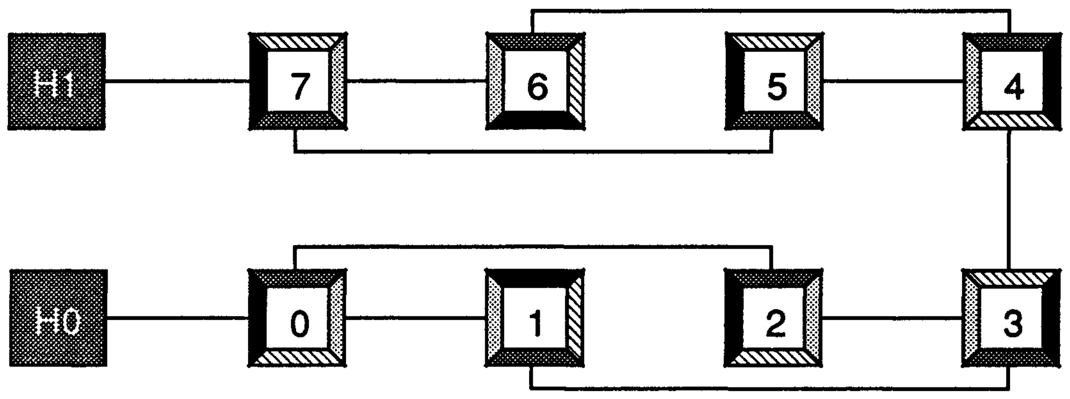


Figure 16. Multi-Host network after cosmetic improvement

used to display the message forwarding strategy as follows:

1. Start up `cnftool`. Choose to work with the current configuration files rather than beginning anew.
2. Select **Show System** from the main menu and then **Show Message** from the subsequent menu.
3. In response to the prompt in the Dialogue Area indicate two nodes between which you wish to communicate. `cnftool` will indicate the forwarding path from the sender to the receiver.

Notice that the path is shown assuming that the first node selected originates the message which is to be received by the second. This is potentially important on networks where different paths are used between two nodes depending on which of them sends the message.

Another potentially important structure is the "tree" through which messages are broadcast by the `exbroadcast` system call. The path taken by a broadcast originating at any node can also be examined from the **Show System** menu by selecting the **Show Broadcast** option. Again the user is asked to indicate the node which originates the broadcast and the corresponding path is indicated.

7 `cnftool` without graphics

The simplest use of `cnftool` is obtained through the graphical interface described in the previous sections. If, however, you do not have a supported graphics device or are unable to use it for some reason a simple line interface is available - essentially one types in manually the required information about processors and links. This procedure is, unfortunately, somewhat tedious but may be necessary in some cases.

In order to present a concrete example of the configuration procedure let us consider the "tree" configuration shown in Figure 17.

Designing a network without pictures

In this diagram the transputers themselves are indicated by the numbered boxes with the interprocessor links shown by the solid lines. At each end of a "link" is indicated the channel number to which it will be connected on the appropriate transputer. Notice that

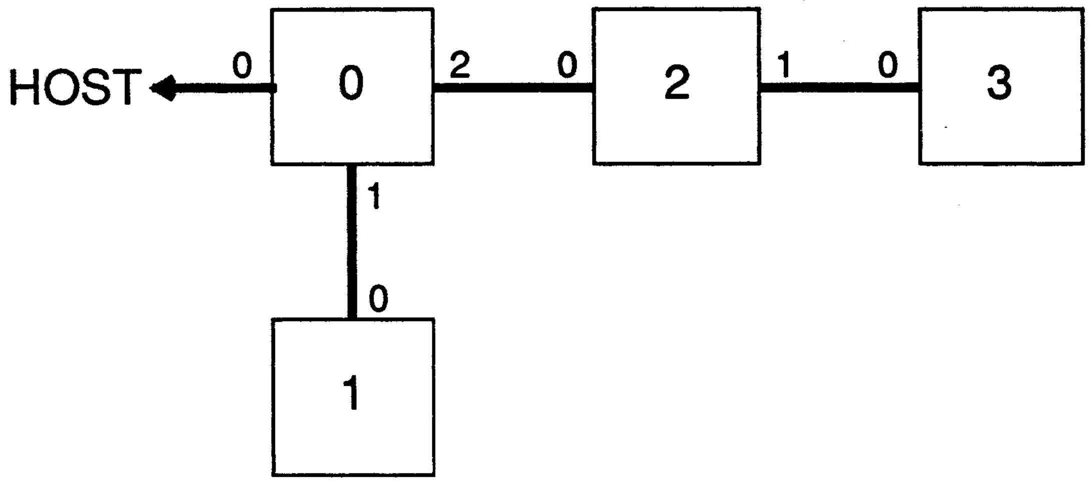


Figure 17. Sample configuration for manual entry

there is no requirement for this number to be identical at both ends of a link. Also note that we have indicated a link which is connected to the "HOST". This is the machine which will be responsible for loading the *Express* system and your applications and which will be the default system for resolving file names and producing graphical output. You will find it much easier to configure the system if you have such a picture of the network you wish to design in front of you. We will only connect this single host in this example - more can easily be added. Also notice that the transputers are numbered consecutively, from 0. This is one of the constraints implicit in the system - more details are contained in Section 8.

Constraints on what you can build

To configure *Express* for this topology and execute the command

```
cnftool -p
```

As in the graphical case you may be asked whether you want to proceed from an existing configuration or begin again. It is normally simplest, in the absence of graphics, to start afresh.

Initially you will be asked to indicate the number of transputers in your network. Respond with a number and a `Return`, e.g.,

```
4 Return
```

for the example shown in Figure 17. Note that we do not count the host.

Next you will be prompted to enter specific information for each transputer in the network.

The first question for each node is its "logical number". This is the number indicated in the boxes of Figure 17. Again, for the sake of argument we will begin entering the data for node 0 so we would enter

```
0 Return
```

The next question requests both the number of the parent node (that which resets and "boots" the current transputer), and an indication about whether the "system" or "sub-system" output of the parent controls the reset for this transputer. This is the trickiest part of the configuration process because of the "system" and "sub-system" concept and the rules governing the actual loading process. Basically the rules can be summarized by saying that we have to be able to draw a binary tree through the transputer network - i.e., there can be at most two legs coming out of any node. One leg represents the "system" reset and the other the "subsystem". If only one leg comes out of any node it must be the "system" reset. A sample tree which we will use in our example is shown in Figure 18.

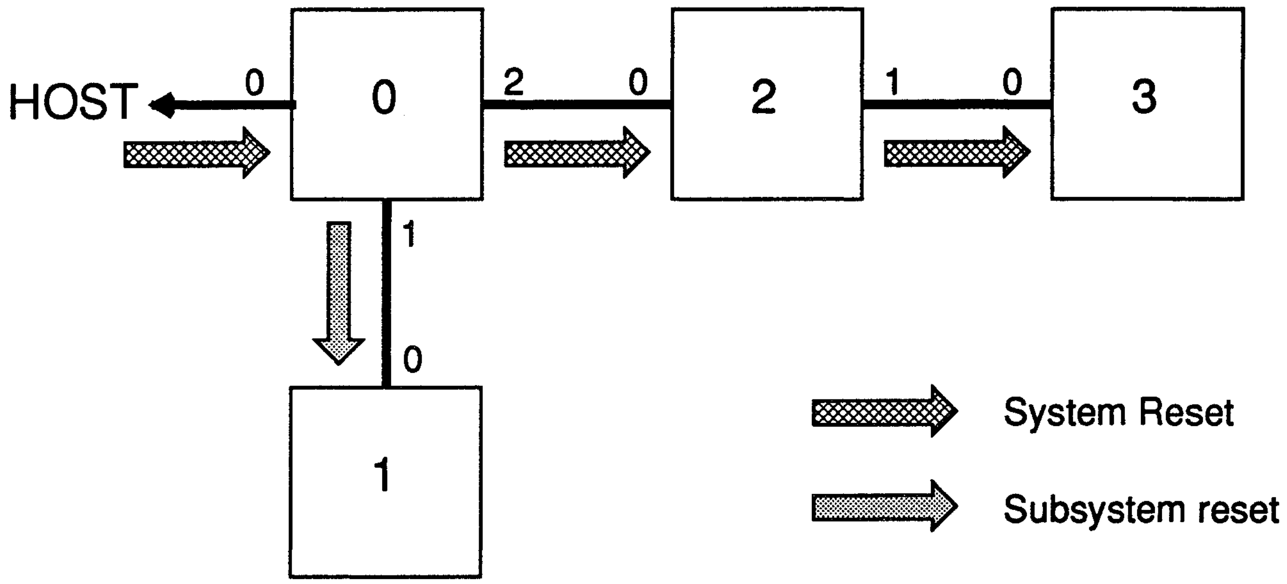


Figure 18. Network with superimposed reset tree

On some types of hardware the "reset" controls being described here are actually part of the hardware itself while in others all the reset signals are connected together with cables. In the former case the reset network being described must match that of the hardware while in the second the information is only required to load the network with the operating system.

`cnftool` asks two questions about the reset system at each node. The first requests the identity of the parent node and the second whether we are connected to its "system" or "subsystem" reset. In our example we would enter

```
-2 Return
1  Return
```

in which the first answer indicates that node 0 is to be reset by the host processor (with the "magical" node number -2) through its "system" reset. (In this case we have little choice since the host has only one link to the transputer net.)

The "host" as processor number - 2, in cnftool ONLY

The last information requested about the transputer concerns (surprise, surprise) the nodes to which it is to be connected. Since each node must have a direct link with the node which boots it one link entry must have the same node number as the parent field given earlier.

`cnftool` asks, in turn, about each of the transputer links. If that link is connected to another node we enter the node number of the connected processor. If no transputer is connected enter "-1". The answers we would give for node 1 of our network, therefore, are

```
        -2 Return                    (Link 0)
         1 Return                    (Link 1)
         2 Return                    (Link 2)
        -1 Return                    (Link 3)
```

i.e., link 0 goes to the host, link 1 to node 1, link 2 to node 2 and link 3 is unconnected.

We have now described everything necessary about node 1. Other parameters, such as the particular transputer type or the size of its local memory will be determined automatically by the system.

`cnftool` will now prompt you to enter similar data about the other transputers in the network. Rather than go through the laborious details of explaining the various responses we merely present the correct answers. Hopefully their meaning will be clear.

```
Node 2: 1 0 1 0 -1 -1 -1

Node 3: 2 0 1 0 3 -1 -1

Node 4: 3 2 1 2 -1 -1 -1
```

After finishing the final transputer entry `cnftool` asks for a suggestion as to what type of topology you have described. The allowed possibilities are: `torus`, `hypercube` and `general`. Obviously the last alternative is the "fall-through" - any network which doesn't fall into the first two categories fits here. In order to answer this question correctly one has to be slightly flexible about ones definition of terms since the transputer has a limited number of links. The details are given in Section 8.

"Deadlock" free routing

If your network conforms to one of the two special configurations then you should indicate so in your response to `cnftool`. If not then indicate a general topology. The important difference between the classes is that *Express* **GUARANTEES** deadlock free routing on both torus and hypercube meshes - your program can send messages to whatever nodes it wishes (up to the limit on kernel buffer space) with no possibility of "hanging" the machine. The general topologies, however, cannot be guaranteed in this manner. (One common exception to this are the "tree" structures like that of Figure 17 which also have safe routing.)

This completes the configuration procedure. From the information just entered *Express* generates two files called `run.nif` and `confile` which describe the network and forwarding strategy to be used on it. At this point you can go ahead and re-load the system with the `exinit` command and proceed as before.

8 Transputer Variants of Standard Topologies

The transputer has only four links. As a consequence certain common computer networks have to be modified slightly before they can be implemented on transputers. In this section we will discuss the construction and restrictions on the "hypercube" and "torus" topologies on which *Express* guarantees safe routing.

The transputer "torus" topology

The transputer variant of the 'torus' topology has already been shown in Figure 2. It is a generalization of the usual torus; edges are connected periodically with the exception of the "spare" transputer between the host and the 'torus' proper. This extra node is required to overcome the problems inherent in having only four links. This topology is very good for simulations in two physical dimensions or for general use.

The transputer "hypercube" topology

The second special network is the hypercube such as that shown in Figure 1. Again notice that we have inserted a "spare" node between nodes 0 and 1 so that the outside world can communicate with the main array. Note that we can actually construct lower dimension hypercubes without cheating in this way: dimension 0, 1, 2 and 3 hypercubes will have their usual elements and numbering schemes - only the dimension four case causes problems. Dimension 5 and higher hypercubes cannot be built from current transputers. (Unless, of course, you consider a "node" to have more than one transputer.)

Numbering problems with the "standard" topologies

For reasons too hideous to discuss in this document there are certain restrictions on the numbering scheme you may use in describing your transputer network. One which may already be apparent is that the numbers you specify in describing the network are not simply related to the node numbers you use in conjunction with the *Express* system calls `exread`, `exwrite` etc. The most obvious reason for this discrepancy is that *Express* allows several users to share the nodes in the array. Each user gets an independent set of nodes which are numbered logically from zero. In this case there is no relationship between the numbers given `cnftool` and those used by the user, nor should there be. In the more elementary case where one user has all the nodes the mapping is simple; the number given `cnftool` is the same as that used in `exread`. (The exception to this rule is the host which has the magic value `HOST`).

As well as these oddities the individual topologies themselves also have certain numbering restrictions associated with the "bootstrapping" procedure. These are as follows:

Torus	The node connected to the host is numbered 0 and then processor numbers increase row-wise as indicated in Figure 2. There must be an equal number of nodes in each row and in each column (although, of course, the overall network need not be square). The exception to this rule is the first row which contains the "spare" node. Link 3 of the spare node should be connected to the host.
Hypercube	The node numbers of the actual hypercube nodes must satisfy the usual numbering pattern. The "spare" node on the four dimensional hypercube will be numbered 16 and must be connected between nodes 0 and 1. On low dimensional systems link 3 of node 0 should be connected to the "master console". On the four dimensional system link 3 of node 16 should be connected to the host.
General	Node 0 must be connected to the host.

Multiple Hosts

Building networks with multiple host processors and using *Express* with it

1 Introduction

One of the most powerful features of the *Express* system is its ability to support multiple simultaneous users on a parallel processing network. Typically such systems still have only a single host to which any number of users may connect, shown in Fig. 1. These users may then allocate their own groups of nodes in the attached parallel processor and can, in effect, work independently.

Multiple simultaneous users

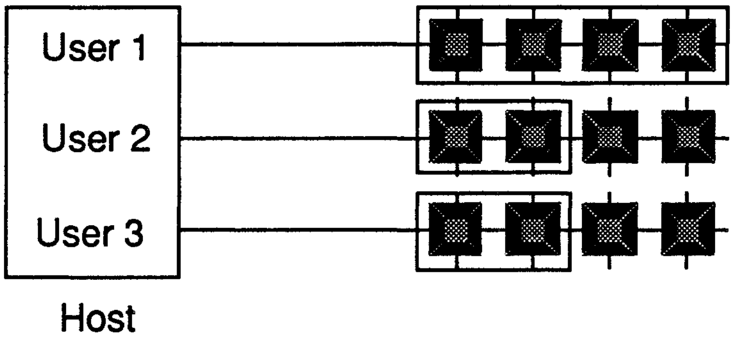


Figure 1. Multiple users sharing nodes from a single host.

In addition to this scenario *Express* also supports systems in which multiple hosts are connected to a single parallel computing resource. In this case it is possible for several users working on separate hosts to share access to the parallel computer as shown in Fig. 2. Again the users may operate independently, each using whatever resources are available on the host from which they are running.

Multiple hosts

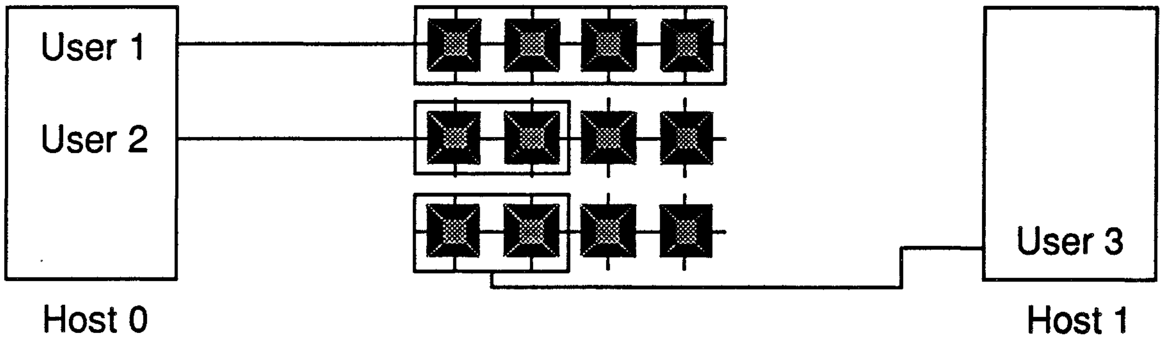


Figure 2. Multiple users sharing nodes from separate hosts.

Note that the hosts in the previous paragraph need not be the same. It is quite possible to use an MS-DOS PC as one host and a Sun Workstation as the other, or a Macintosh and a MicroVax, etc. The only requirement is that the host be connected to the parallel processing system through a supported interface - either a transputer board in the host or a link adapter. One is also not restricted to two hosts - as many hosts may be connected as the transputers have links to connect them. The hosts need not be real computers either - *Express* would treat an attached disk server for a data base as another host.

Heterogeneous networks

The scenarios described so far entail each user accessing only the utilities of the host from

Express servers

which they allocated the nodes being used. In addition to this picture *Express* allows the user access to the facilities of other hosts through various server mechanisms. Using the *Cubix* programming model, for example, one may download a program from one host and use its console for formatted I/O but simultaneously direct other I/O requests to another machine and graphical output to a third device, Fig. 3. Using such techniques it is obviously possible to build complex, custom tailored systems, for such applications as CAD/CAM, Data Base, etc.

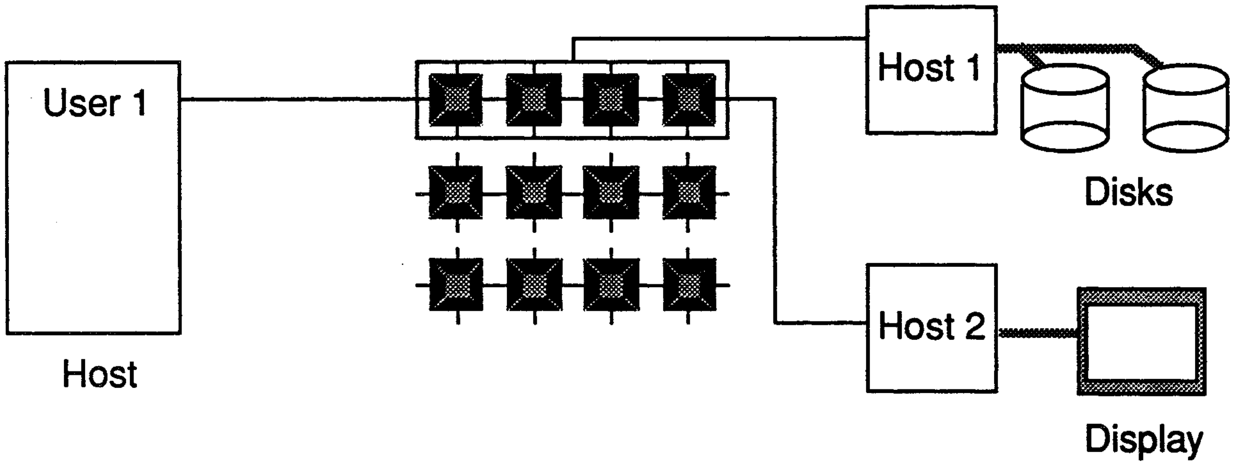


Figure 3. A Single user using multiple host facilities.

Using Cubix in non-Cubix programs

If you are not using the *Cubix* programming model you may still take advantage of the multiple host features. A program loaded from one host may communicate with any other node in the parallel processing network, or any attached host using the basic *Express* message functions. Similarly a host program may ask to *share* an existing group of nodes allocated by another host program, possibly running on a completely different host. In this way the two host programs may simultaneously communicate with a single node program. In addition the two host programs may even talk to each other through the transputer network.

Note that the facility of multiple hosts naturally brings together the *Cubix* and host-node programming models. You may load a program with *Cubix* onto a group of nodes and then have a host program share access to the processors. This gives you the best of both worlds.

This document is designed to lead you through the procedures necessary to setup and use a system with multiple hosts. It assumes that you are already familiar with the *Express* system on a single host and you should consult the standard *Express* manual for a thorough description of the system calls described in this document. Sections 2 and 3 address the problems associated with connecting multiple transputer boards in different machines which is often necessary to connect systems with multiple hosts. Much of this material is repeated from the similar discussion in the cnftool documentation but is repeated here to make this chapter self-contained. Section 4 describes the procedures necessary to configure and boot *Express* onto a system with multiple hosts. Section 5 describes the *Cubix* server process which is used to provide I/O and operating system services on a second host. Section 6 describes the mechanism which allows several host programs to share access to the group of nodes and section 7 explains how *Cubix* programs may be built with their own host programs attached, combining the information from sections 5 and 6.

2 The "RESET" problem - mechanical connectors

In this section we consider only those types of hardware in which the transputer links are connected with physical wires. Systems which use Inmos' electronic switch, the C004, are dealt with in section 3.

In building a system with multiple hosts there are basically two ways to proceed depending on how the second host will be attached to the existing nodes.

Connecting a second host via a link adaptor

In the simplest case the second host will be supplied with a link adapter (only) which can be connected to one of the links on the existing network. If you are content to always initialize the system and load *Express* (with `exinit`) from the original host there is no "RESET" problem and you should skip to Section 3 for a description of the configuration steps needed to put *Express* in contact with the second host. If, on the other hand you wish to be able to reset the system from either host, read on.

Connecting a second host through a second transputer board

If your system will contain multiple transputer boards you must address the question of how to make the "RESET" signals work. Before *Express* can be loaded onto the system every transputer node must be in the reset state. On a single board this is usually simple since manufacturers take care to provide a path through each of the nodes. When connecting multiple boards, however, you must do some work yourself. The simplest way to check on your progress as you attempt this task is to use the "worm" program supplied with `cnftool`. This program resets as many nodes as it can reach and then draws an image of everything found, see Fig. 4.

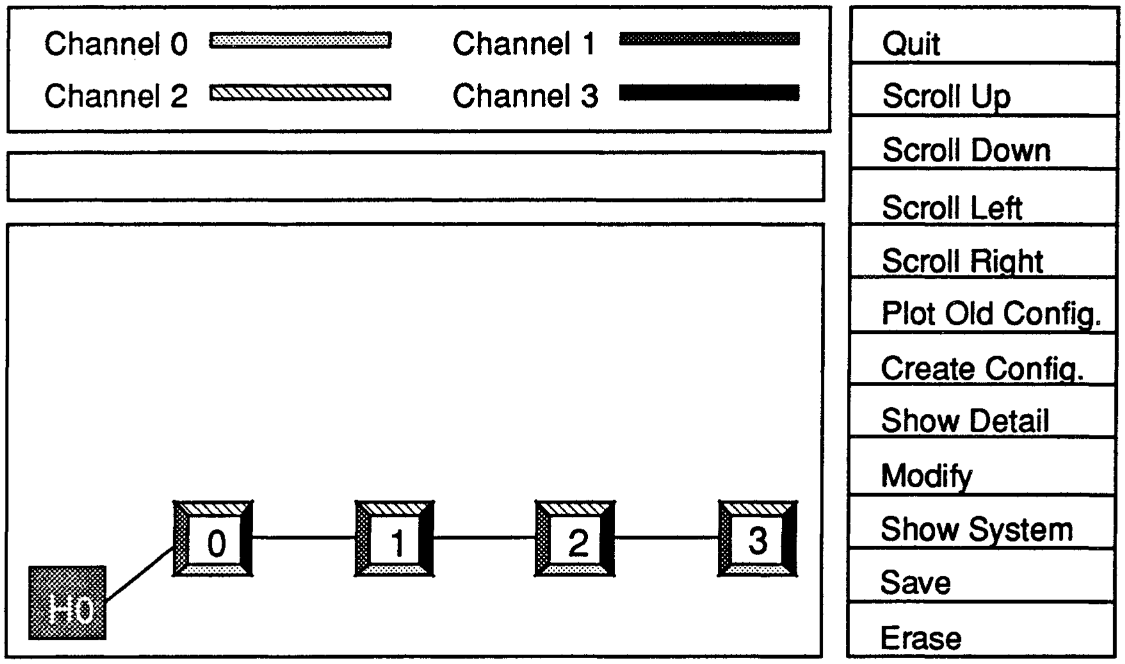


Figure 4. `cnftool`'s display of a single host system

The simplest procedure for connecting up multiple boards, therefore, is to hook up whatever wires seem appropriate, fire up `cnftool`'s worm and count the nodes on the display. When this number reaches the expected value you are home. Unfortunately life is not quite this straightforward on most systems.

Debugging multi-host systems; an example

To make this discussion more concrete let us assume that we have two transputer boards in separate hosts, each with four nodes. If you find that only the four nodes on your first board are being detected by `cnftool` it means that you are not managing to reset any nodes on the second board. You should probably consider the following steps:

i) Are the reset signals really connected up?

ii) Have I understood the "Up/Down" and system/subsystem business correctly?

iii) Do the machines share a common ground? (This is not usually necessary but we have seen it help in a least a couple of cases.)

iv) Were all the boards supplied by the same manufacturer? If not am I sure that the links all conform to the same, TTL or RS434, standard?

If everything seems to be well you should either consult with your hardware supplier or consider using `exreset`.

`exreset` is a program whose sole aim is to reset any nodes which refuse to be "done" by your system. In the present situation, even if all else has failed, you should be able to go to the second host's console and type

```
exreset
```

Resetting stubborn hardware

This will reset any of the nodes on the second board which are still connected to the second host - hopefully this includes all the ones that were missed before. Now you can go to the first host and execute `cnftool`'s worm. Since the second set were reset by `exreset` and the first set of nodes are reset by `cnftool` you should now be able to detect all eight nodes. If not check out your link connections.

Hopefully this process has resulted in `cnftool` describing all the nodes in your system. It will, however, only show one host - the one from which it was executed, see Fig. 5. One should also notice that the numbering of the nodes may not correspond to your idea of which node is in which host. `cnftool` numbers the nodes in increasing order of detection which may not correspond to your own numbering at all - it is not even guaranteed that the lowest numbered nodes will all lie on one board!

Having built a description of the transputer nodes in your system it remains to add the second host.

3 The "RESET" problem - Electrical connections

The problems with off-board links

Machines which have the electronic link switch pose tricky problems during configuration, mostly connected with the fact that while the on-board links are hooked up electronically the off-board and inter-board connections must be manipulated by hand. A further difficulty is that no "worm" program can help you diagnose problems with your connections since by their very nature there is no network to explore until it has been

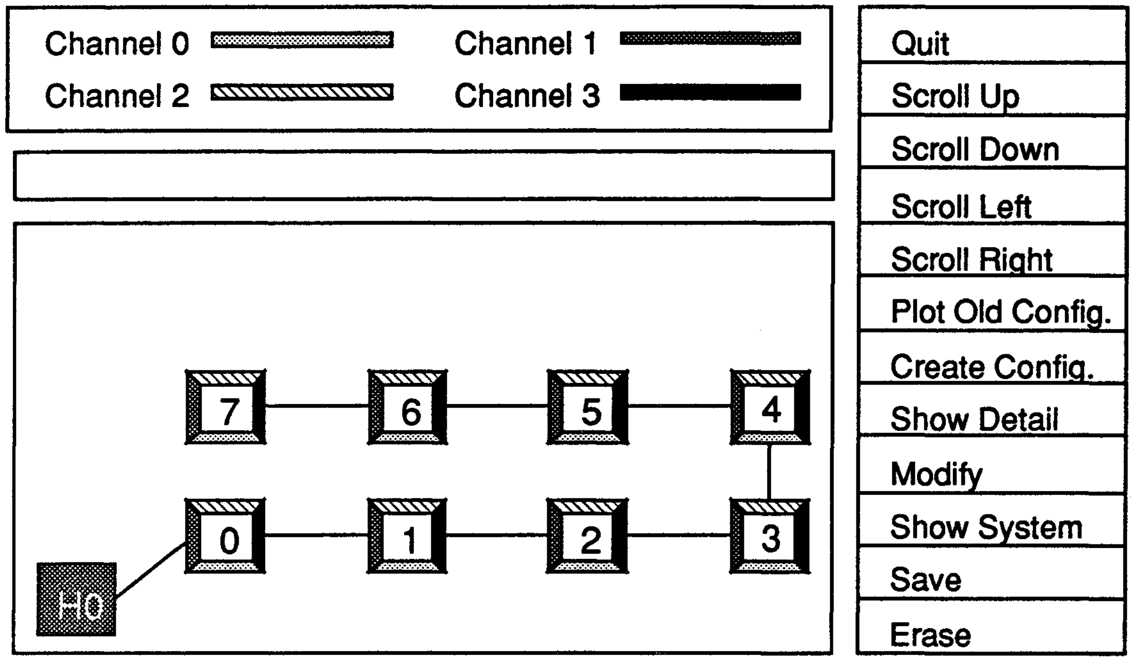


Figure 5. `cnftool`**'s display of multiple node boards.**

configured! (Note, however, that systems with mixed electrical and mechanical links can be explored with the "worm" once the electrical part of the network has been configured.)

To perform this task we again use the `cnftool` program.

Basically we start off by using `cnftool` to describe the entire network as we wish to set it up. In this sense we can ignore the fact that some wires may be missing and should proceed as though `cnftool` were somehow able to transcend this problem. Include the extra hosts that you will be adding to the system. Once the network has been described and its forwarding table created and safely stored away with the **Save** command we must decide which links need to be added to the system.

Configuring a second transputer board electronically; an example

To make this discussion more straightforward let us consider a system which will ultimately contain 12 transputers, four on each of three boards. For simplicity we will assume that each board has a C004 which can look after the on-board links but which requires mechanical cabling of the off-board links. Also, for simplicity let us assume that all boards are currently located in a single host - the transition to multiple hosts is quite straightforward.

We will assume that the nodes labeled 0-3 are to be found on the first board while 4-7 are on the second and 8-11 on the third. Also let us decide to make the following inter-board connections:

- Node 2, link 2 -> Node 4, link 3
- Node 6, link 3 -> Node 9, link 1

To implement this strategy we have to know how *Express* assigns off-board links to the connectors which are used for cabling the inter-board links. This is best seen by looking at the following piece of "pseudo-code" which contains the algorithm.

The algorithm for determining off-board connections

```
for each board in system {
     off board slot := 0
     for each node on board {
         for each link on node {
             if link goes off board {
                 assign link to next free off board slot
                 increment next free off board slot
             }
         }
     }
}
```

This algorithm is actually just what you might assume - for every link that goes off the board assign the next available slot in the edge connector.

Using this algorithm on the links shown above yields the following assignments:

- Node 2, link 2 assigned to slot 0 on first board.
- Node 4, link 3 assigned to slot 0 on second board.
- Node 6, link 3 assigned to slot 1 on second board.
- Node 9, link 1 assigned to slot 0 on third board.

As a result of this assignment we need to add the following physical connections

i) First board, slot 0 to second board, slot 0.

ii) Second board, slot 1 to third board, slot 0.

This completes the process required to configure multiple boards in a single host. To use the facilities of several hosts we merely move the `run.nif` file created by `cnftool` and the hardware to the second machine. The connections remain in place.

The "exreset" command on multiple boards

To boot this system for *Express* we must now reset the network and configure its links. On all but the first host this is achieved with the `exreset` command. When executed two questions will be asked:

- What node number, as given to `cnftool`, is has the lowest number of this board? In our example we might answer 4 is the second and third boards had been moved to a second host.
- Which node, if any, in this machine should be connected to the host and which link should be used for the connection?

After asking these questions `exreset` configures the C004's on the boards in this host and leaves the transputers in the reset state.

On the host containing the first board you can now execute the `exreset` command. Since all the links in the other hosts' boards have been configured by the `exreset` command `exinit` can now load the *Express* kernel into the entire network. While doing this `exinit` will realize that insufficient nodes are present in the first machine to build the entire network and it will say so before proceeding to the other machines and completing the loading.

4 Configuring Hosts and Booting *Express*.

Adding the second host

At this point you are probably sitting in `cnftool` with an image similar to that of Fig. 5 on the display - it shows your original host, with the legend "`H0`", and whatever new transputer nodes you have added to the machine but no second host. (If you skipped to this section from Section 2 because your second host has only a link adapter and no nodes then you should fire up `cnftool` to get to an image similar to that of Fig. 5.)

If you have not yet added the second host to the display you should do it now.

To add a second host select the **Modify** option to bring up the second level menu and then **Modify Hosts** to bring up a set of options connected with hosts. In this menu select **Add Host** to create a second host. `cnftool` will prompt you to position the new host somewhere on the display. Use the arrow keys or mouse to locate the position at which you wish to display the new host's icon - the position on the screen is irrelevant. When you click to select a new position a second host icon will appear with the legend "`H1`". This number is important - you need it whenever you wish to specify that some action is to be performed on a host other than the one which loaded your program.

To connect the second host to the network you must now add a "link". This is, unfortunately, rather tricky since the "worm" program numbered the nodes in a random pattern rather than that engraved on the side of your new hardware! You need to figure out which node in `cnftool`'s image corresponds to the one to which your second host is attached. This is usually possible by tracing the existing links on the display - `cnftool` may renumber the nodes but the link numbers stay correct.

Once you have decided which node in the network needs to be connected to the new host the procedure **Modify**, **Modify Links**, **Add Link** should be enough to take you from the top level menu through adding a new channel.

Having completed the description of the enhanced network you need to create a reset tree and forwarding table in the **Show System** menu and then **Save** the configuration before exiting, in exactly the same way as for a system with a single host.

Booting Express onto multiple boards

Once back at command level you should try booting *Express* onto the new network. Before rushing off and typing `exinit` you need to once again consider the "RESET" issue. If you brought the system up by typing `exreset` on one or more hosts you'll need to do this again before trying `exinit` (and, in fact, any time you type `exinit`).

Assuming all nodes are either reset or in a position to be reset by `exinit` you should go ahead and try to load *Express*. If all is going well you should see a message saying that *Express* has been loaded and a subsequent message allocating *all* the nodes for topology initialization. Any errors connected with incorrect links should be diagnosed by the loading program.

Diagnosing problems - exstat

Once *Express* has been loaded into the network the simplest way to check that everything is now connected correctly is the `exstat` command which lists the allocated nodes and the running processes. While the output from this command will not be very interesting until user programs are executed it is useful to detect misplaced connections - if, for example, this program fails to respond when executed on the second host it probably means that the connection you indicated to `cnftool` as being between the second host and the network is either misplaced or not working.

Once both your hosts have passed the "`exstat`" test you can confidently proceed to use the newly enhanced system.

System configuration files

The information so far presented should allow you to build a system that can, at worst, be rebooted by typing `exreset` on some of the hosts and then `exinit` on one machine. If you wish to have the option of downloading *Express* from more than one of the hosts several system configuration files must be copied from the host normally performing the `exinit` function. At present the list of files and their approximate contents are:

`NIFFILE`	Describes overall system configuration, which links are attached to which nodes, etc.
`CONFILE`	Routing table built by `cnftool` and loaded during `topinit`, the second phase of `exinit`.
`PLOTFIL`	Graphical display last printed by `cnftool`.

Note that the names shown here are the customization variables which are created by the `excustom` command. The exact filenames can be found from the system customization file - see the `excustom` description for more details.

In passing we might note that the above listed files currently contain all system configuration information. If you wish to keep track of a particular hardware configuration and later return to it without going through the `cnftool` process you can save these files and later restore them. *ParaSoft* cannot guarantee that later versions of *Express* will maintain all its information in the same files.

5 Using *Cubix* in a multi-host environment

Graphics and I/O servers

One of the most common uses of a network with multiple host machines is to take advantage of the file systems or graphics capabilities of the extra machines. In order to do this we introduce the concept of the *Cubix* server process.

If you have programmed in the standard *Cubix* environment you are probably familiar with the `cubix` program as the means to download an application to a group of transputers and then serve its I/O requests on the system console. Commands such as

```
cubix -n 4 myprog dog cat horse <input.dat
```

are used to load a program called `myprog` into 4 nodes, passing it arguments `dog`, `cat` and `horse` and connecting its input stream to the file `input.dat` on the host computer. Using this programming model we are able to take advantage of a number of standard operating system services which are typically unavailable to programs running on transputer networks.

Starting the Cubix server process

To take advantage of a second host we must execute the single command

```
        cubix -S
```

on the second host. This command starts up *Cubix* in its server mode. It begins by arranging to access all of the nodes in the transputer network to which it has access and then sits and waits for your node programs to ask it for services.

Multi-host programming example

The simplest mechanism by which a *Cubix* program can take advantage of multiple hosts is shown in the following program extract

```
          system("date");

          console_node(0x8001);
          system("date");

          console_node(0x8000);
```

The first "`system`" call should result in the date being displayed on the normal system console. The following function call, `console_node`, is then used to redirect *Cubix*' attention to an alternative host. Note that the "magic" number given to the function is merely the host number displayed by `cnftool` OR'ed with the hexadecimal value `0x8000` which sets the highest bit in the number. This is one of the reasons you need to remember the magic numbers generated by `cnftool`.

After making this call the second "`system`" call will display the date on the monitor of the second host.

Redirecting operating system requests

The `console_node` call is extremely powerful. Once this call has been made ALL operating system requests, with the exception of file I/O, will be directed to the indicated host. File I/O, on the other hand, remains bound to the host which has the appropriate data - *Cubix* keeps track of which file resides on which host and uses this information irrespective of any calls to `console_node`.

Locating files on alternative file systems

In discussing the question of which file lives where we have to address the `open` question. Basically a file is located by *Cubix* according to whatever host was controlling the system when the `open` or `fopen` system call was made.

For the three standard files streams, `stdin`, `stdout` and `stderr` this will always be the host from which you loaded your program. (Note that you can play clever games with `close` and `dup` to override this constraint.)

To access files on other machines we can modify the name of the file as given to the `open` or `fopen` commands by prefixing a magic number such as that given to `console_node`. The following example shows the general mechanism.

```
          fd1 = open("file1", O_RDONLY);

          console_node(0x8001);
          fd2 = open("file2", O_RDONLY);
```

```
        fd3 = open("8000:file3", O_RDONLY);

        console_node(0x8000);
        fd4 = open("file4", O_RDONLY);
```

The first file, "`file1`", will be opened on the normal host - the one which loaded the original program. The call to `console_node` then redirects attention to host 1 and opens "`file2`" there. The third call to `open` specifies a prefix to the file name overriding the current default and resulting in "`file3`" being opened on host 0. Finally we switch consoles again to host 0 and "`file4`" is opened there.

Note that while the above description concentrated exclusively on the `open` system call the same remarks apply to `fopen` calls. Furthermore any *digits* may appear before the '`:`' character in the override portion of the file name string. They are interpreted as a hexadecimal constant and interpreted in the same manner as in `console_node`.

The mechanisms described so far allow us to take advantage of the file systems and operating system utilities on any host connected to our system. A particularly common use of these techniques, however, is the use of additional graphical output. By analogy with the `console_node` routine which redirects *Cubix*' operating system requests, the *Plotix* system has a call `display_node` which redirects graphical output. Its use is identical to the *Cubix* routine and it applies to all subsequent calls to the `sendplot` utilities.

6 Writing host programs in a multihost environment

The previous section explained how to use the facilities of *Cubix* on a network with multiple hosts and how various system calls were available to gain access to the facilities of the various machines. The complementary information, about host programs, is given here.

The role of the host

Express' philosophy is that host computers play a role entirely analogous to the nodes of the parallel computer. Most of the system calls which are used to communicate data between node processors can also be used to talk to the host machine - the magic value `HOST` is used as the "node" parameter.

In a system with multiple hosts the situation is necessarily more complex - one `HOST` value is clearly insufficient to differentiate between several possible hosts.

Addressing multiple host processors from "host-node" programs

On the node side the solution is quite simple. To communicate with a host that `cnftool` denotes `H3`, for example, we use the magic value `0x8003` obtained by setting the highest bit of that used in configuration. You may, in fact, already have seen this in action if you ever printed out the value of `HOST` in your code. *Express* actually does this calculation for you when your program begins so that the value of `HOST` is always set to the machine from which you loaded your code.

Sharing groups of nodes

The host side is trickier because of the process of node allocation. A simple host program calls `exopen` to gain access to a set of nodes in the parallel computer. If the second host program were to make the same call *Express* would take this as a request for another, different, set of nodes and would attempt to allocate a different set of processors. For this reason the second host program should call either `exshare` or `exaccess`.

exshare is the most common system call used when two programs must share access to the same set of nodes. ndb, the source level debugger uses this system call internally to communicate with the nodes allocated to your node program. The three arguments to this function are:

exshare

device	The same argument as supplied to the exopen call in the first host program, with the same meaning. If the two programs attempting to share the nodes are executing on hosts with different operating systems these arguments might differ slightly according to local naming conventions but they should indicate the same parallel computing device.
process_ID	This argument is used to differentiate between the programs currently executing on the nodes. Use of the exstat command is the recommended method for determining this value.
pnodes	This value is returned to the caller containing the number of nodes in the group being shared.

The value returned by exshare is a node group identifier to be used in the same way as the return value from exopen. Note that calling exclose in one host program does not deallocate the nodes - both programs must call exclose before the nodes are free for someone else.

A simple piece of code which attaches a second host program to a set of nodes and sends messages is shown below

exshare example

```
/*
 * A host program to share access to the group of nodes
 * specified by the process ID given on the command
 * line. Sends a simple message to each and then quits.
 */
#include <express.h>
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
     int dummy, type, i, Nnodes;
     int pid, node_fd;

     if(argc < 2) {
         fprintf(stderr,
             "Usage: %s process_ID\n", argv[0]);
         exit(0);
     }

/* Get process ID to share from command line. */
```

```
        pid = atoi(argv[1]);

    /* Attempt to share nodes with this process. */

        if((node_fd = exshare("/dev/transputer",
                                pid, &Nnodes)) < 0) {
            fprintf(stderr,
                "Failed to share nodes with process %d\n",
                 pid);
            exit(0);
        }
        else printf("Sharing %d nodes\n", Nnodes);

    /* Send each node a simple message and quit */

        type = 123;
        dummy = 0;
        for(i=0; i<Nnodes; i++) {
            exwrite(&dummy, sizeof(dummy), &i, &type);
        }

        exclose(node_fd);
        exit(0);
    }
```

exaccess

The second method for sharing access to a set of nodes is exaccess, a rather "brute-force" approach. This routine has no process-ID argument but rather obtains access to *every* node in the machine irrespective of its current operation. Note that this is a rather desperate action to take but has the advantage that even the first host program can use it - no node program needs to be executing for this call to work. The arguments are, therefore:

device	The same argument with the same meaning as in the exshare system call.
pnodes	This value is returned to the user and contains the number of nodes in the entire system.

A subtle difference arises in the numbering of the nodes when using this system call. Note that when we used exshare in the previous example we could still address the nodes in the machine with the numbers 0, 1,. . ., Nnodes-1 irrespective of their physical position in the array. If, for example, two users were simultaneously using four nodes in the system then the first might be allocated the "first" four nodes in the machine and the second four higher numbered nodes. Both users, however, can communicate with their respective nodes using the numbers 0, 1, 2 and 3. *Express* performs the necessary mapping between the hardware node numbers (such as those known to cnftool) and the logical node numbers used in programs.

The use of exaccess, however, short-circuits this problem. As a result the arguments used in calls to exread, exwrite, etc. must be *physical* node numbers. This means that in the example just quoted the user of exaccess would have to know the physical node number at which the group of allocated nodes starts and add to this a logical node number to arrive at a value to pass to *Express*. Note that *Express* always allocates node in blocks with contiguous physical node numbers so the mapping is determined by a single number. Further note that in the case where only one program is to run on the system the logical and physical node numbers will be the same (since the origin of any allocated group will be physical node 0) and the problem evaporates.

Processor numbering after a call to exaccess

While exaccess is usually unnecessary it does have one useful feature in that it returns the number of nodes in the entire machine. One can, therefore, create the following routine to return the number of nodes in any system controlled by *Express*.

**Finding the number of nodes in the machine*

```
/*
 * Count_nodes.
 *
 * Returns number of nodes in array.
 */
int count_nodes()
{
     int Nnodes, node_fd;

     if((node_fd =
         exaccess("/dev/transputer", &Nnodes)) < 0) {
         return -1;
     }
     exclose(node_fd);
     return Nnodes;
}
```

Notice that we call exclose to terminate the connection to the processor group before returning the number of allocated nodes.

Having now addressed the issue of how the nodes communicate with multiple hosts and how the hosts attach themselves to a set of nodes without calling exopen the final question is the obvious one: "Can one host talk to another without the nodes intervening?"

Inter-host communication

The answer to this question is a qualified "yes". If the two host programs have used the standard mechanisms to share access to a set of nodes: exopen in one host and exshare or exaccess in the other then host "H0" can send a message to host "H1" by using the magic node value 0x8001, just as a node would communicate with another host. At present there is no way for two host programs to communicate without first allocating at least one node.

Hybrid Cubix and "host-node" programs

7 Combining *Cubix* with a host program

Having now discussed the possibilities for a node-based *Cubix* program to use the facilities of many hosts and also the mechanism by which multiple hosts might communicate with a single group of nodes a final possibility exists: a *Cubix* program using the I/O facilities of one or more hosts and simultaneously interfacing with a host program.

This option is actually quite straightforward using the constructs of the previous two sections. We create a standard *Cubix* program on one host and load it with the `cubix` program. We already understand from section 5 how this program can utilize its own and possibly other hosts through the *Cubix*-server. While this program is executing, however, we can also have a host program use the `exshare` and `exaccess` to gain access to the nodes.

The result of this is a means for programs to have the best of both worlds - I/O facilities and a native host interface. The sole strange feature of this process is that the loading process is inverted from that which one might naively expect - it is probably more natural for the host program to allocate the nodes and load the application which then communicates with a background *Cubix* server. This option may become available in a future version of *Express* if the present solution proves unworkable.

Customizing *Express*

Modifying the installation, size and performance of *Express* with `Excustom`

1 Customizing *Express*

When *Express* systems are shipped they contain default information which has been found to be appropriate for a wide variety of applications. This information relates to the setup of the *Express* system on the target host, the operation of the "tools" used while creating and evaluating *Express* programs, and the runtime behavior of *Express* applications.

Obtaining maximum performance

The customization system allows for explicit modification of most important *Express* system parameters with the explicit goal in mind of gaining maximum performance from a given parallel processing system. A secondary goal of this system is to allow users the ability to install *Express* in any place on their machines and for third-party software developers to build *Express* into their applications without the need for a complete *Express* installation.

This chapter is divided as follows. Section 2 describes the customization file which is the central component of the customization system. Section 3 discusses the `excustom` tool which is used to modify the information contained in the customization file. Section 4 describes the *Express* buffer management policy and the impact of changing the buffer sizes/allocations. Section 5 discusses the important issue of *Express*' usage of the transputer's memory. This information is crucial to anyone wishing to gain a better understanding of their program's performance or trying to use the advanced debugging tools present in *Express*. Section 6 lists the various customization parameters and their exact meaning. This is the major reference for those users whose needs are not adequately served by the simple `excustom` tool provided to modify high level system parameters. Finally, Section 7 describes the mechanisms used to locate the default system configuration under various operating systems.

2 The *Express* "Customization" file

The operation of Express

An *Express* system is characterized by a set of variables which describe how the various tools, subroutines and even the *Express* kernel operate. These values are maintained in a database known as the *Express* customization file. Every installation of *Express* must have one of these files whose contents vary widely with the type of the underlying system.

The customization file

The customization file is a line oriented ASCII file which contains definitions of important system variables, one to a line. Lines beginning with either '`;`' or '`#`' characters are treated as comments. Other lines take either of the symbolic forms

```
NAME=text
MACRO:=text
```

Macros make installation changes easy

As is suggested by the above notation the former type are merely assignments to *Express* system parameters while the second define macros that may be further used in the customization file to simplify definitions of multiple related objects. A good example might be the default start-up information required by the debugger, `ndb`. As part of its configuration information it needs to know the location of the on-line help facility and also the system start-up file which contains the definitions of system commands. Since these are often in the same or related directories one might imagine two entries in the customization file of the type

```
NDB_HELPDIR=c:\parasoft\help
NDB_STARTUP=c:\parasoft\lib
```

These entries could, however, be replaced by the lines

```
PARASOFT:=c:\parasoft
NDB_HELPDIR={PARASOFT}\help
NDB_STARTUP={PARASOFT}\lib
```

While three lines may seem more complex than the original two the use of the `PARASOFT` macro means that the *Express* system can be moved from one directory to another by simply changing the macro rather than each line of the customization file.

Modifying Express parameters

As has been implied by the above discussion the operating parameters of *Express* can be simply modified with a text-editor or word processor by locating the system default configuration file, modifying the appropriate parameters and reloading *Express*. This process is indeed all that is required although it mandates an explicit knowledge of the meanings of the variable names. These are discussed in Section 6.

Finding the customization file

The most tricky aspect of this entire discussion is the location of the customization file itself. While *Express* provides a simple function call that programs can use to direct attention to a particular file every system contains a default configuration whose name must either be known in advance to *Express* or which can be indicated at runtime. The exact mechanism is somewhat complex and is discussed in detail in Section 7.

3 Modifying System Parameters with `excustom`

As was previously indicated a simple, though inelegant, method for modifying the configuration information of *Express* is to use Section 7 to locate the default configuration file and then the definitions of Section 6 to find out which values are used for which purposes. One can then edit the configuration file with some text-editor or word processor, reload *Express* and continue.

Since this process is somewhat tedious, especially for those users who do not need to configure individual entries, an *Express* tool is available which automates this process: `excustom`.

The exact operation of this tool is dependent upon the exact hardware in your system and the operating system running on the host so it is possible that some of the details discussed here will be irrelevant on your system. The accompanying discussion should, however, be enough to indicate the general principles.

excustom

The basic idea of `excustom` is to offer you the chance to modify the most important of the *Express* variables along with defaults based either on the current system values or some "sensible" defaults. If you are operating *Express* from a simple terminal with a conventional "line oriented" interface you can invoke the system with the command

```
excustom
```

You will be prompted to modify the values of the various system parameters individually. If you don't wish to change any particular value just use the "Return" or "Enter" keys at the prompt without entering any text.

An example dialog, taken from MS-DOS, might be as follows:

An example: MS_DOS

```
What kind of machine are you using?        [STD_LINK] :
Where is your compiler located?              [c:\logc] :
Where is ParaSoft's home directory?    [c:\parasoft] :
How many buffers per transputer node?           [100] :
How many buffers on the host?                    [20] :
What size buffers?                             [1024] :
Where is the Express kernel be loaded?     [fffffff] :
Do you want to modify link adaptor parameters? [y,n] :
Do you wish to use DMA?                         [y,n] :
Do you wish to use a "block move" interface?   [y,n] :
Do you wish to enable accounting?               [y,n] :
```

In each case we can see that the user input, which would normally follow the ' : ' character is empty indicating that the default action should be taken. In most cases this is indicated by the value in brackets although it defaults to "no" for the simple questions.

Notice that the number of questions asked is somewhat less than the number of entries in the customization file. This is because of the macro facility described in Section 2. Instead of asking questions related to every one of the file related *Express* options we merely ask about the root of the *Express* and compiler installations and derive the other information with macros. If you wish to change individual file entries the customization file must be edited by hand.

Default values

The default values offered in the above customization example would be obtained by reading the current customization file. This allows incremental modification of a developing system. If you wish to restore your system to its "factory settings" then you can invoke the tool with the command

```
excustom -r
```

which tells the system to "reset" its default values when prompting you. Finally you can modify files other than the default system configuration by naming them on the command line

```
excustom mycustom
```

would modify the contents of the customization file called `mycustom`. In this way applications or individual users can maintain their own customization files independent of all others.

In windowing environments such as MicroSoft Windows, SunView or the Macintosh `excustom` presents a "dialog box" interface in which the various customization options are offered simultaneously for editing. Various buttons are provided to maintain the functionality of the line-oriented interface - a "reset" button is usually available which makes all parameters assume their default values.

4 *Express* buffers

Three of the most important questions asked in the customization process are related to the manner in which *Express* uses buffers.

Express is "packet switched"

Express is a packet oriented communication system. What this means is that when you send a message between processors *Express* breaks it up into fixed size pieces and transmits each individually. While the exact reasons for this are somewhat complex it can be shown that such "packetizing" systems are more reliable under heavy message traffic than those which send all messages as single blocks.

Overheads and how to avoid them

Obviously the process of breaking up a message into, for example, 1024-byte chunks, sending each full block and then sending any remaining bytes takes some time. (Note that a 1025 byte message, in this example, would be sent as a 1024-byte packet followed by a single byte, not as two 1024-byte packets.) In order to optimize the behavior of your applications the buffer size to be used can be changed through the customization system. If, for example, you know that the maximum sized message that your code will ever send is 3000 bytes then it makes some sense to tell *Express* to use buffers at least this big. Notice, however, that you can go too far. If you tell *Express* to use 64 Kbyte buffers all the time you are getting close to the position of telling it to ALWAYS send messages in one chunk, however large. This technique can fail in heavy traffic if insufficient memory is available to send/receive messages.

Controlling buffer allocation

In connection with this buffer size parameter one can also indicate the number of buffers to be allocated in both the transputer nodes and the host processor. The former is to allow you the freedom to use the node's memory as effectively as possible. By default, for example, *Express* allocates 100 buffers of size 1024 bytes. 100 Kbytes of memory is therefore used on each transputer node for the *Express* communication system. If your program is short of node memory you might want to reduce the number of buffers in each node. In particular, if you wish to increase the size of the individual buffers you might want to make fewer of them in order to not use up too much node memory.

> IMPORTANT: The size and number of *Express* buffers does not affect the ability of the system to send large messages since every message larger than a single packet will be automatically broken down into smaller ones.

Running out of memory

An important issue connected with buffer management is the question of what happens when *Express* runs out of buffers.

In the most senseless case one can imagine a configuration in which there are, for example 20 node buffers each of size 1024 bytes for a total of 20 Kbytes. Now let us assume that every node attempts to send a 10 Kbytes message to node 0, and no attempt is made by node 0 to read any of the messages. If we have four nodes then this will result in 30 Kbytes of data arriving at node 0 which has capacity to handle only 20 Kbytes. At this point the machine will almost certainly "hang" - no further communication is possible. What has happened in this case is that each node starts sending out 1024 byte packets to node 0. Eventually node 0 will run out of space to store these packets and so the nodes attempting to transmit and/or forward messages to node 0 will have to stop. At this point no further message traffic is possible which has to be routed through node 0. This "deadlock" situation will then backtrack out towards the other nodes in the system and each in turn will have to stop and wait for the blockage at node 0 to disappear.

"Deadlock"

If we modify this scenario slightly by having node 0 try to read the incoming messages with, for example, the wildcard DONTCARE value for the message source, then things will (probably) work out much better. Now the nodes dispose of their data by sending it out towards node 0 which is actually consuming packets as they arrive. Now there will be enough space to buffer the incoming messages and the machine will not "deadlock". Notice that everything works out correctly even though the capacity of node 0 to handle its incoming data is still smaller than the amount of data being transmitted. If the buffers temporarily become full in node 0 some other node may have to suspend its operation while space is made available on node 0 but it will then continue automatically.

Notice that it is still possible to create impossible situations. Let us suppose that node 0 decides to read the incoming messages in some specific order: node 3, node 2 and then node 1. Furthermore let us assume that due to some timing situation inherent in the program all ten buffers from node 1 arrive before all ten from node 2 which, in turn, arrive before any from node 3. Again we have a "deadlock" situation since the capacity of node 0 is exhausted by the messages from nodes 1 and 2 but the message requested for reading is from node 3. Since node 3 cannot find space on node 0 to store any of its packets it will stop and wait, forever. In this case we can trivially remove the deadlock by allocating 21 buffers on node 0. In this case we are guaranteed that node 0 will have at least one space available for the incoming message from node 3 and so the system can proceed.

If this discussion has only persuaded you that buffer allocation is too difficult a problem for the human mind you may be correct. The question of exactly how many buffers to allocate is an extremely complex one depending on the algorithmic demands of the application the operating characteristics of the hardware in use and the topology in which the network is connected. Simply because of this difficulty *Express* provides the user with the ability to choose the various operating parameters because they do play a role in optimizing program performance but cannot be predicted beforehand.

The "host" and its special problems

One area of central importance in this field concerns the host interface. On most current transputer systems the host-node link is much slower than the node-node links. As a result the host reads messages much more slowly than the nodes. While this would probably suggest that a large number of buffers should be allocated on the host one is often limited by the amount of memory available. To attempt to allocate 100 Kbytes on a DOS machine, for example, would meet with certain failure. To cope with this situation, on the host processor only, *Express* writes extraneous messages to disk if they cannot be processed immediately. This is an *extremely* slow process which can slow down the program to a crawl. (One way of seeing this happen is if your program slows to a virtual halt and the disk activity light starts flashing continuously.)

Optimizing the host interface

A simple way to optimize in this case is to make sure that the messages arriving at the host can be processed immediately. While this requires some discipline on the part of the user program it is often quite easy to achieve. Note that it is not enough to simply read all messages on the host with the wildcard DONTCARE value for the message source and type. To see this consider the case where the *Express* packet size has been set to 512 bytes and the incoming messages are of length 1024 bytes - i.e., two packets each. Since it is not possible to guarantee the order that packets arrive from different nodes we can now make the worst case assumption that the first packet of the message from node 0 is immediately followed by all the other packets being sent from the other nodes, and then finally by the second packet from node 0. On receipt of the first packet from node 0 the host becomes committed to reading the rest of that message before accepting any others so the next few packets will be buffered in host memory and then will begin spilling over to the disk. Finally the host will see the last packet from node 0 which will complete its message but it will now be forced to examine the disk overflow area for subsequent messages with the resulting loss in speed.

Note that the scenario in which the host always reads with wildcard values AND the packet size is set to be greater or equal to the length of the largest message to be sent DOES guarantee that no disk activity will occur.

5 *Express* and the Transputer Memory Map

Optimal use of the hardware; fast on-chip memory

Because of the nature of the transputer's hardware a particularly important optimization is the use of the node's fast on-chip memory. In order to understand the mechanisms which are useful in this respect we must first examine the way that *Express* is positioned in memory and how it loads user programs.

On the left of Figure 1 is shown the default location of the important system components when *Express* has been loaded and a user program is running. At the top of memory lie the *Express* buffers, occupying the space required by the parameters set in the customization file. Just below this lies the *Express* kernel itself. At the bottom of memory lie the crucial parts of the transputer hardware, the link registers, the scheduling queues, etc. Immediately above this region (at a location which depends upon the exact type of transputer in use) is the fast on-chip memory. This extends either 2 or 4 Kbytes up from the bottom of the memory space, again depending upon the type of transputer in use. The upper limit is indicated by the solid grey line in the figure.

The default memory map

When the user requests that a program be loaded into the machine it is, by default, positioned directly above the break between fast on-chip memory and the slower memory which makes up the bulk of the system. Furthermore the program's stack, which contains all "local" variables and is used to pass arguments to functions and subroutines, is positioned immediately below the *Express* kernel. Finally the space between the top of the user program and the bottom of the stack is used for dynamic memory allocation.

No on-chip memory is used at all.

Why use faster memory?

To make use of the faster memory we first note that it is really only effective for stack based variables. Due to the nature of the transputer it requires a single instruction to access a stack based variable but at least two to access a global variable. As a result the memory speed has most impact on local variables where the faster memory makes the instruction two or three times quicker than the same instruction accessing the slow memory. In the global variable case the best we can expect is to speed up one of the two (or more) instructions needed to access the data resulting in a smaller overall improvement.

Taking advantage of faster memory

What we can deduce from this is that the best use of the hardware is made by having lots of local variables and then placing them in the fast on-chip memory. If we must use global variables such as large arrays they are best accessed through a locally declared pointer variable.

With this picture in mind we can then take advantage of the faster memory by using the

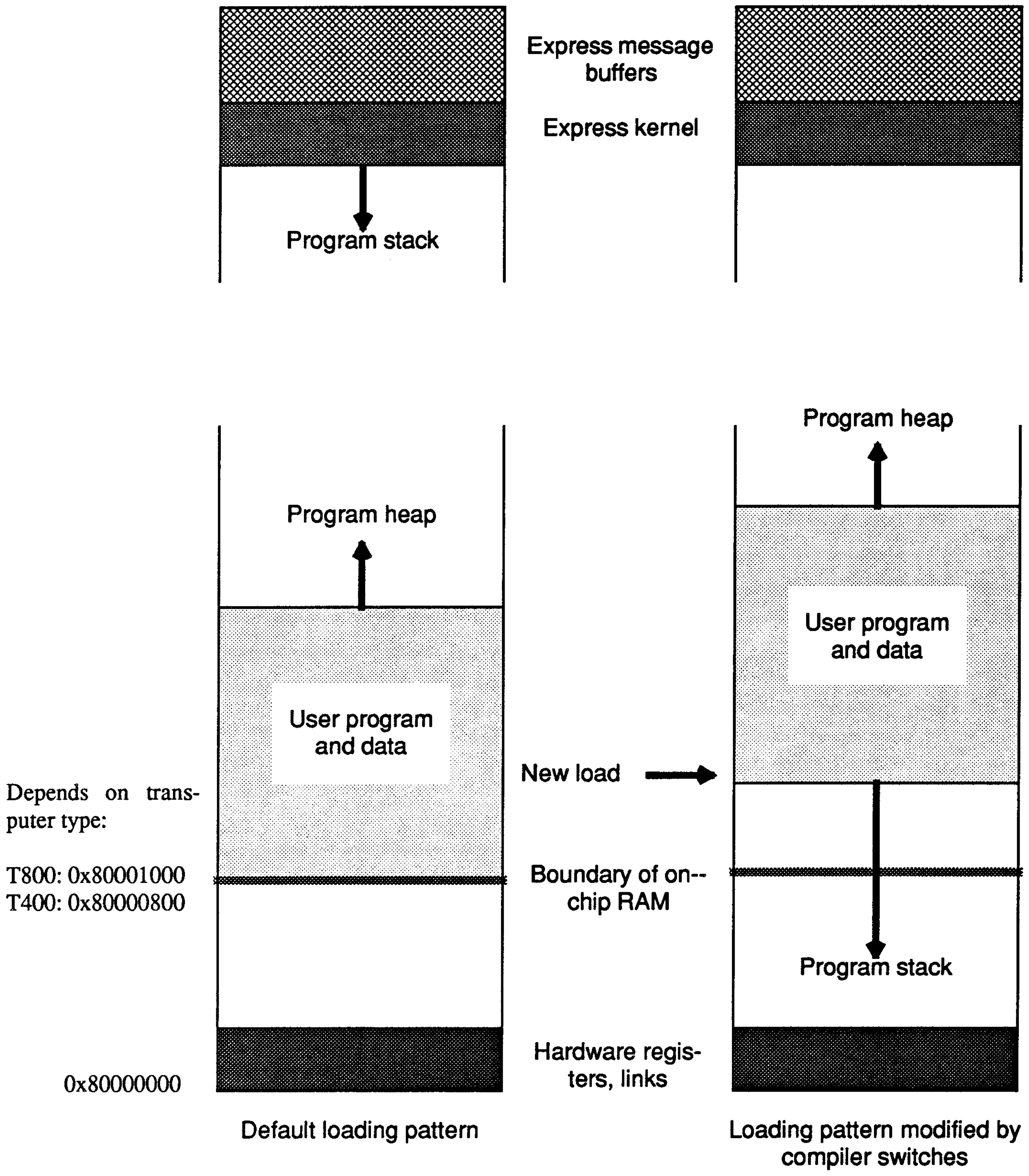


Figure 1. Transputer memory map with *Express* loaded

'-B' option of the various compiler commands; tcc and tfc. This switch requires a single argument which specifies the *base* of the user program AND the *top* of the stack. To see the effect let us consider the case where a value has been given which is slightly above the break between fast and slow memory. The resulting program layout is shown on the right side of Figure 1.

As indicated, the user program has been moved slightly upward and, the important feature, the stack now grows down from the bottom of the user program, toward the faster memory.

The effects of this are two-fold. Firstly the stack will eventually enter the fast memory making local variable accesses faster and secondly the program is more prone to crash catastrophically. While the former is just what we wanted the latter is rather nasty. The basic problem concerns the use of the lowest memory locations for crucial transputer registers. If your program uses enough stack space to overwrite the low memory addresses the system will die... dramatically. This is the reason that the exact positioning of the stack is allowed with the '-B' option. The game to play here is to "guess" how much stack space you might need and then position the base of your program far enough above the transputer critical region so that disasters never occur. This is typically quite straightforward. If you are using the debugger, ndb, you should be able to read the value of the workspace register at various places in your code - this tells you the current base of the stack. In other cases you might try printing out the address of a local variable.

In the worst case you can find out the correct place by trial and error - the symptoms of the disease are so easy to recognize that one can quite quickly tell where the correct location might be, although this method requires multiple compilations.

In many cases programs do not have great stack requirements and they can be positioned immediately below the break between fast and slow memory. In this case the entire program stack will be placed in the fast memory. (Note that you can't assign the *exact* address of the break between slow and fast memory since that is used to indicate the alternative configuration in which the stack is placed in high memory. Also, the address given should be word-aligned for best performance - i.e., a multiple of four.)

Code caching

A question often asked is whether or not it is beneficial to place code in the fast memory. (With the '-B' switch you could obtain this effect by giving an address below the break between fast and slow memory and then linking the program's object files in some special order.) The transputer's CPU includes a small instruction cache which supposedly means that normal sequential instruction operation is unaffected by the speed of the memory in use. If your program branches a lot, however, the CPU will miss the instruction cache a lot. In this latter case some improvement could be expected by running with the code "on-chip". In practice, however, we have seen little benefit from such a strategy.

Placement of the Express kernel

A last point to note in connection with the memory layout is concerned with another of the system configuration variables: the kernel load address. Normally this takes the default value -1 (0xffffffff hex) which indicates that *Express* should attempt to figure out the amount of memory in each node and then position the kernel as high as possible. This is the scenario shown on the left of Figure 1. The exact mechanism by which *Express* achieves this is to write and then read the entire contents of the node's memory. (That this is sometimes quite time consuming is indicated by the delay during exinit.) During this process each memory location is initialized to its address.

While this is perfectly good for most problems it necessarily destroys the contents of the node's memory. Upon occasion, especially when using the RAM files and the `exdump` debugging tool one would wish that the contents of the node memory were preserved across calls to `exinit`. To achieve this merely change the default kernel load address in the customization file to some physical address. In this case *Express* will place the kernel exactly where indicated without checking the node memory or destroying any of its contents, *other than those overwritten by the kernel itself.* Note that a small part of the node memory is destroyed in this manner but since you can control explicitly which portions are lost it is easy to avoid locations containing RAM files.(You can also achieve this effect by using the '`-m`' option to `exinit`, see the Reference manual page for more details.)

Interaction with RAM files

Preventing memory initialization

6 *Express* on UNIX machines

One of the significant benefits of complex operating systems such as UNIX is the protection afforded individual users by the separation of operating system or "kernel" activities from those of applications. The price one pays for this, however, is that every system call has to pass through the protection layer into the kernel - a time consuming operation. For parallel computer systems the most obvious effect of this is to slow down communication between the host and the nodes. Unfortunately this link is usually the slowest even before adding the price of UNIX system calls and, as such, may be too heavy a burden for real applications to bear.

The overheads of using UNIX

Because of this *Express* allows the hardware to be used without any protection from the UNIX kernel. This option is selected when executing the `excustom` program by answering '`y`' to the question

Using Express without the UNIX kernel

```
        Do you wish to run without the kernel?     [y/n] :
```

While running without the UNIX kernel *Express* programs will communicate more quickly with the host. Certain restrictions do, however, apply.

The most significant of these restrictions is that the machine can no longer be used in multi-user mode. Since the UNIX kernel is no longer available to make decisions regarding the destinations of the various messages that come from the nodes we must restrict access to a single user at any one time. While this may be too stringent a restriction for development purposes it is eminently reasonable for "canned" applications which use the parallel computer only for its speed.

Restrictions on multi-user access when running without the kernel

The second problem when running without the kernel is that one has to take care when using the debugger, `ndb`. The reasons behind this are essentially the same as in the previous paragraph - the debugger is a second process which must run and share access with another program. As a result there is potentially conflict between messages coming from the nodes for the debugger and the user program. While one would normally, therefore, advise that debugging be done with the UNIX kernel enabled one can, with some care, debug without it by ensuring that no conflicts arise. In practice this means that one should only query the node program with the debugger when it is "stopped" at a breakpoint. While in this state the user program will not try to read messages destined for the debugger and all should be well.

Difficulties debugging without the kernel's protection

7 Listing of *Express* customization variables

The following is an exhaustive list of the various customization variables which are normally to be found in a default customization file. Not all variables are present on all systems.

In each case we show the name of the variable together with any default which would normally be present. In most cases this will be a derivation of a file name from some macro.

Note that the following list shows pathnames derived from their respective ROOT's with the syntax used under MS-DOS. Other operating system use different syntax for their directory hierarchies which should be easy to derive from those shown here.

Supported transputer hardware

`MACHINE`

The type of hardware in use. The currently recognized transputer systems are

`B004`	Inmos B004 and compatibles.
`DEFINICON`	Definicon systems Inc.
`MICROWAY`	Microway.
`QUN`	Quintek.
`SUN_B011`	Inmos B011 with Sun Microsystems host.
`SUN_B014`	Inmos B014 with Sun Microsystems host.
`SUN_KOBE`	Kobe Steel board with Sun host.
`SUN_TOPO`	Topologix Inc.
`LEVCO`	Levco Inc.
`SUN_MEIKO`	Meiko Ltd.
`SUN_MK200`	Meiko "In-Sun" computing surface.

It should be noted that this list is growing all the time. The most complete source of information regarding the currently supported list of machines can be obtained by running the `excustom` tool.

`PARASOFT:`

This macro is used as the basis for finding the various subdirectories of the *Express* installation. It is usually assigned through `excustom`.

`KERNEL` `{PARASOFT}\bin\express.tld`

The name of the *Express* kernel.

`NIFFILE` `{PARASOFT}\bin\run.nif`

The name of the network information file which describes the interconnection between transputers. Usually built by `cnftool`.

`CONFILE` `{PARASOFT}\bin\confile`

The name of the file which describes the routing for messages. Usually set up by `cnftool`.

`PLOTFILE` `{PARASOFT}\bin\plotfil`

The name of the file containing the most recent pictorial representation of the transputer network. Used and created by `cnftool`.

WORMLD	{PARASOFT}\bin\worm.exe

The name of the program responsible for executing the "worm" which figures out the interconnection of the hardware. Used by `cnftool`.

The "worm" program

WORM	{PARASOFT}\bin\worm.run

The name of the program loaded into the nodes to detect their interconnection. Used in conjunction with the `WORMLD` value by `cnftool`.

TMPNIF	{PARASOFT}\bin_run.nif

A temporary configuration file built by the "worm" utilities and used by `cnftool`.

PARABIN	{PARASOFT}\bin

The location of the *Express* executables. Used by many of the tools.

TMP	{PARASOFT}\tmp

A temporary directory used to hold various intermediate files and the overflow area for messages coming to the host which cannot be buffered in the host's memory.

PARAINC	{PARASOFT}\include

The location of the "include" files used by C programs compiled to run on the transputers. Used by the compilers.

PARALIB	{PARASOFT}\lib

A general repository for program libraries and other binary files which are essential to the system but which are not executable. Used by most of the tools.

`NODE_DEV`

The default device for all calls to `exopen`, `exshare` and `exaccess` made by the system.

`RESETFILE`

The name of the file containing information used by the `exreset` program when initializing multiple transputer boards in multi-host systems.

T212

This is the name of a special program loaded into T212 transputers controlling link switches.

Controlling buffer sizes and their allocation

NNBUF	100

The number of *Express* message buffers to be allocated in each transputer node.

NHBUF	20

The number of *Express* message buffers to be allocated in the host. After this number is exceeded all further messages will be placed in an overflow area on the disk. No messages will be lost by choosing a small number although performance will be significantly degraded.

NBSIZE	1024

The size of each *Express* message buffer, in bytes. Messages longer that this will be cut into smaller pieces (by *Express*) and sent in packets.

Kernel start address

LOADSTART 0xffffffff

The address in transputer memory at which the *Express* kernel should be loaded. The default value indicates that the transputer memory should be scanned to find the amount on each node and then the *Express* kernel is placed as high in memory as possible. Since the scanning process destroys the contents of memory it should not be used in cases where, for example, RAM files need to be preserved. In these cases a non-default value of LOADSTART should be used.

PROCNUM 10

The maximum number of simultaneous processes on any given node. If you do not use the exhandle system calls this value is irrelevant.

Link adaptors

RESET	0x160
ANAL	0x161
LINK_RD	0x150
LINK_RD_STAT	0x152
LINK_WT	0x151
LINK_WT_STAT	0x152

These six I/O addresses are used to specify the standard "B004" style link adapter interface to the board. Any number may be individually modified for special hardware.

Host interfaces

DMA 0

Use to indicate the type of interface available between the host and the transputer board. The following values are recognized

0	Standard "B004" style link-adaptor.
1	DMA style interface as on MicroWay hardware or INMOS B008.
2	"Block move" interface on advanced Definicon boards.
3	Memory mapped interface on Sun, runs without UNIX kernel.

System accounting

ACDIR

The name of the directory into which accounting information is to be logged. A NULL value (i.e., ACDIR=) disables the accounting system.

Logical Systems C compiler

LOGC:

This macro is used as the "root" of the tree containing the Logical Systems C compiler.

LOGCINC {LOGC}\include

The directory containing the Logical C include files. Used by tcc.

LOGCLIB {LOGC}\library

The directory containing the Logical C standard libraries. Used by tcc.

LOGCPP {LOGC}\bin\pp

The name of the Logical Systems C preprocessor. Used by tcc.

`LOGCTCX` `{LOGC}\bin\tcx`

The name of the Logical Systems C compiler. Used by `tcc`.

`LOGCTASM` `{LOGC}\bin\tasm`

The name of the Logical Systems transputer assembler. Used by `tcc`.

`LOGCTLNK` `{LOGC}\bin\tlnk`

The name of the Logical Systems linker. Used by `tcc`.

`PS_HEADERS` `{PARASOFT}\lib`

The location of the header files used by the *Express* PostScript libraries and the `-Tps` option to `cubix`.

`BGI_DRIVERS` `{PARASOFT}\lib`

The location of the Borland Graphics Interface device drivers used to perform graphical output on a variety of DOS-based systems. Used by all tools and the `-Tbgi` option to `cubix`.

`3LC` `c:\tc2v0` *3L C compiler*

The installation directory for the 3L C compiler. Used by `tcc`.

`3LFORTRAN` `c:\tf2v0` *3L Fortran compilers*

The installation directory for the 3L FORTRAN compiler. Used by `tfc`.

`3LINCLUDE` `c:\tc2v0`

The directory containing the 3L C include files. Used by `tcc`.

`AFSERVER`

The path name of the directory containing the "afserver" program used to download and communicate with such things as the 3L C and Fortran compilers. Used by `tcc` and `tfc` on non MS-DOS systems.

`CUBIX_PLOTS`

Supported graphics options under `cubix`. These are the values allowed in the '`-T`' switch.

`MONITOR` `0`

Specifies whether the default graphics device is color (`MONITOR=0`) or monochrome (`MONITOR=1`).

`DVDIR` `c:\dv`

The directory containing the DesqView system used for with the debugger under DOS. Used by `ndb`.

`NDB_LINEFILE` `ndbXXXXXX`

The name of the file used to store source code line numbers by `ndb`. This template is used in conjunction with the `TMP` variable in a call to `mktemp`.

`NDB_HELPDIR` `{PARASOFT}\help`

The directory containing the debugger's on-line help.

ndb's start-up file

`NDB_STARTFILE`

The name of the start-up file read by `ndb`. Any file with this name in the current directory (i.e., the one in which `ndb` starts) will be processed before `ndb` begins. This name is also used in conjunction with the `NDB_STARTDIR` variable to locate the system startup file which is almost crucial to the useful use of the debugger.

`NDB_STARTDIR` `{PARASOFT}\lib`

The directory containing the `ndb` system startup file. The actual filename to be used is contained in the `NDB_STARTFILE` variable.

`WINBIN`

The directory containing the "window" versions of the *Express* tools. Used to setup the "server" program for easy access to the *Express* toolset.

8 Default locations for *Express* customization files

One of the trickier aspects of the *Express* configuration system is actually finding the default system customization file. While this is not usually necessary (`excustom` should know how to do it automatically) it is occasionally necessary for detailed modifications.

Default customization files

The simplest method for determining the name of the default file is to execute the command

```
excustom -?
```

If you wish to override this name or otherwise specify a default the following sections indicate the appropriate mechanisms under the various host operating system supported by *Express*.

8.1 MS-DOS

Upon starting the environment variable `EXPRESS` is consulted for the name of a customization file. If none is found the default

```
c:\parasoft\bin\express.cst
```

is used. Note that this is sufficient if you have installed *Express* in the default location on your hard disk. If you have installed *Express* in some other directory then the default system configuration file will be found in the `parasoft\bin` subdirectory with the name `express.cst`.

8.2 Unix and look-alikes

Upon starting the environment variable `EXPRESS` is consulted for the name of a customization file. If none is found the default

```
/express.cst
```

is used. A standard configuration file is typically to be found in the `bin` subdirectory of the *Express* installation with the name `express.cst`.

8.3 Macintosh

The default location of the *Express* customization file is in the `System folder` on the default boot disk. It should have the name `Express.cst`. While the *Express* system can itself be loaded anywhere on your system the configuration file *must* be placed in the system folder.

8.4 VMS

Upon starting the logical name translation tables are consulted for a variable with the name `EXPRESS` which should be set to the name of a customization file. There is no default and the system will abort if no such variable is found.

Index

General index to *Express* and the examples from the text

General Index

This index is the general reference for all the topics discussed in this manual. It lists not only the various functions/routines but also the examples and other points of note.