CDL - A Distributed Language for Helios

Perihelion Software Technical Report No. 3

C. Grimsdale

August 1988

Perihelion Software Limited
The Maltings
Charlton Road
Shepton Mallet
Somerset
BA4 5QE
England
Telephone +44 749 4203
Fax. +44 749 4977

Contents

1 Introduction

Helios is an Operating System designed to exploit highly distributed systems. The central goal of such a distributed system, is not only to provide increased fault tolerance, but also increased performance. This necessitates adequate support for parallelism at the application level. It is proposed that instead of extending an existing language, or producing a completely new language, parallelism can be defined at the distribution phase. A high level Component Distribution Language has been designed, which allows simple sequential components to be combined by simple parallel constructors to form more complex parallel structures. The implementation language is irrelevant thus providing consistent support for standard sequential languages in a distributed environment.

Although occam provides extensive support for the elegant expression of fairly fine grain parallelism, there is still considerable commercial pressure to provide extensions to more conventional sequential languages, to facilitate their use in the development of parallel programs. Such non standard extensions however severely limit the portability of the resulting application. It is therefore desirable to provide support for parallelism at a higher (implementation language independent) level.

Many applications also demand a more abstract model of parallelism, than that afforded by occam. The necessity for a higher level parallel language has been discussed by May and Hoare (D. May, CAR. Hoare - Unpublished notes) and these discussions set the foundation for this work.

2 The Helios Task

The Helios task represents that element of a program which most logically resides on a single processor. It is not necessarily single threaded but the resources allocated to multiple threads are managed collectively, and the task is thus restricted to processor boundaries. Many tasks may collaborate in the solution of a problem, with tasks communicating via the message passing facilities of Helios. Distributed programs are thus composed of one or more tasks which may be distributed across processor boundaries. This collection of tasks is referred to as a Task Force after a construct of the same name in the StarOS Operating System [1, 2]. The Task Force control structure concept was also adopted by the designers of Medusa [3]. The multiple tasks, or components of a Task Force represent relatively coarse grain parallelism.

3 CDL

CDL is a simple language which defines the interconnection of Task Force components and thus the overall topology of the Task Force.

A Task Force is written as a number of primarily sequential tasks which communicate with their environment. These tasks are separately compiled and linked, and form autonomous objects. A Task Force can then be defined either by a simple textual script (submitted to a compiler) or as a standard command line to the Helios shell. The Helios C Shell supports a subset of the CDL syntax and as such the two are closely linked.

This enables the specification of many different Task Forces from different connections of the same simple components (tasks). CDL is translated into a binary representation of the problem topology. The binary format defines each component task and its connections with the environment and fellow tasks. This Task Force description is submitted to a distributed system server called the Task Force Manager, which automatically maps the components of the Task Force onto the existing processor network.

4 Parallel Constructors

The basic parallel constructors provided by CDL are derived from CSP [4]. These constructors not only denote parallelism, but also structure. Unfortunately the contorted nature of the Unix C Shell syntax has necessitated unavoidable divergence from CSP notation.

A brief introduction to the constructors follows.

Pipe

```
A | B
```

The pipe constructor denotes a relationship between the parallel components A and B, whereby the output of A is connected to the input of B. Pipes are useful structures that are familiar to most Unix programmers; however, unlike conventional Unix command line piping, this pipe results in true parallel execution of the piped components.

Subordinate

```
A <> B
```

B is a subordinate of A and so this constructor is analogous to a conventional subroutine.

General Parallel Constructor

```
A ^^ B
```

The general parallel constructor, ^^, causes the components A and B to execute in parallel although they are otherwise unconnected. The result is a system composed of the components acting in lockstep synchronisation.

Farm

```
A ||| B
```

The farm constructor is a CSP interleave construct. It defines that the components A and B, which may have the same alphabet, are executed in parallel. Where both processes contain the same action, the action of the system (that is, the Task Force) is an action of exactly one of the components.

5 Multi Component Structures

These constructors can be used to define complex structures based on collections of much more basic components.

The pipe although simple is also very powerful. Several examples are provided by May and Shepherd [5]. There are many other examples where simple systolic array type structures can be used and in many cases these decompose to a pipe. The pipe is a structure with which UNIX programmers are already familiar, and it often forms the backbone of other more complex structures. The example given here is a simple data recorder, RECORDER. It reads data serially from a simplex link; the data is buffered, transposed, and then displayed.

The definition of the RECORDER structure is parsed from left to right. Modifications to this structure can be easily implemented by adding further components. To strip out all line idle tags, include a component squash.

```
CLEVER.RECORDER =
linkserver | squash | buffer | transpose | display
```

The farm is a structure well known to occam programmers. It is a particularly powerful structure because of its inherent load balancing properties, which can obviously be useful in many applications. The solid modelling example given by May and Shepherd [5], for instance, can be easily described using CDL.

Replicated Components

Structures can be replicated in CDL; this is used, for instance, in defining the four components of the Farm. Replicated pipes and bi-directional pipes can be defined, for example:

```
[4]| transform
[5]<> transform
```

Compound Components

New components may be defined as combinations of existing actual components, referred to as compound components; for example, the statistical mechanics example of May and Shepherd can be defined as follows:

```
Updata = [4]| update
Host <> Controller <> Updata
```

with Updata defined as a compound of the four elements of the update pipe.

Nested Components

CDL supports nesting of structures, which means that complex nested structures (for example, a tertiary tree) can be produced. The example given here is a bi-directional pipe in which each element of the pipe supports two subordinates.

6 Component Declaration

CDL also supports the declaration of individual components, where the resource requirements of components can be defined. These resources include the input and output streams by which the component communicates with its environment. It is similar to the occam placement system but has valuable extensions: it enables the user to specify the exact requirements of a

component task, and so results in a more economic management of system resources.

7 Future Extensions to CDL

Work is currently under way to include support for multidimensional structures, such as arrays. There is obvious potential for static checking to ensure absence of deadlock. The standalone C compiler currently under development at Inmos is of considerable interest, and the possibility of producing a configurer to generate CDL Object from occam placement information could lead to the automatic placement of occam programs.

References

[1] Jones, A.K, Chansler, R.J. Jr., Durham, I., Schwans, K., and Vegdahl, S.R. StarOS, a Multiprocessor Operating System for the Support of Task Forces. Proceedings of the 7th Symposium on Operating System Principles. pp 117-127 1979

[2] Jones, A.K., and Schwans, K. TASK Forces: Distributed Software for Solving Problems of Substantial Size. 4th International Conference on Software Engineering pp 315-330 1979

[3] Ousterhout, J.K. Medusa A Distributed Operating System UMI Research Press Computer Science: Distributed Database Systems

[4] Hoare, C.A.R. Communicating Sequential Processes Prentice Hall International Series on Computer Science

[5] May, D., and Shepherd, R. Communicating Process Computers Inmos Technical Note 22