CHIP SPECIAL

82005/89001
28,– DM

TRANSPUTER

Der Transputer und seine Programmiersprachen

- Möglichkeiten mit parallelen Architekturen
- Transputer-Betriebssystem Helios
- Parallel-Programmierung in Occam, C und Fortran
- Compiler für C, Ada, Pascal, Prolog, Forth
- Entwicklungs-Tools
- Multiprozessoren optimal nutzen
- Die Vielfalt der Anwendungen

TRANSPUTER

CHIP SPECIAL

BETRIEBS-SYSTEME

Edlinger/Schwalm
MS-DOS, Ausgabe 1*
0949 DM/sfr. 49,-, öS 380,-

Edlinger/Schwalm
MS-DOS, Ausgabe 2*
0951 DM/sfr. 49,-, öS 380,-

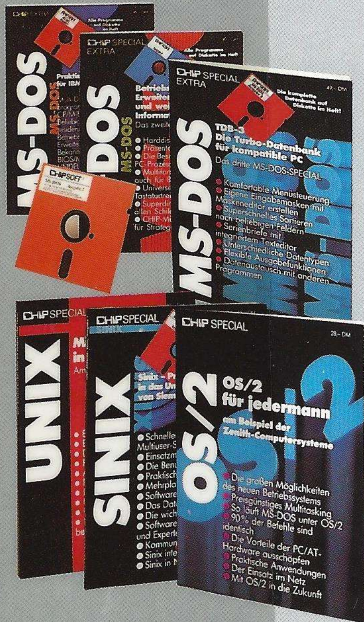
Kern
MS-DOS (Datenbank), Ausgabe 3*
0969 DM/sfr. 49,-, öS 380,-

Dereser
CP/M +
0460 DM/sfr. 28,-, öS 230,-

Ruprecht
Sinix* vergriffen
0967 DM/sfr. 28,-, öS 230,-

Ruprecht
UNIX
0954 DM/sfr. 28,-, öS 230,-

Meder
OS/2
0982 DM/sfr. 28,-, öS 230,-



PROGRAMMIER-SPRACHEN

Binder
C-Der schnelle Einstieg, Ausg. 1
0440 DM/sfr. 28,-, öS 230,-

Binder
Professionell arbeiten mit Turbo-C, Ausg. 2
0964 DM/sfr. 28,-, öS 230,-

Binder
C Praxis für Experten, Ausg. 3
0976 DM/sfr. 28,-, öS 230,-

Rahlff/Knappe
C auf dem Amiga, Ausg. 4
0640 DM/sfr. 28,-, öS 230,-

Freiberg/Obermaier
Microsoft-BASIC
0200 Sonderpreis DM 7,50

Simon
Atari ST Pascal plus 2.0
0956 DM/sfr. 28,-, öS 230,-

CHIP-SPECIAL-Team
CHIP Professional Programmieren, Ausg. 1
0965 DM/sfr. 19,-, öS 150,-

CHIP-SPECIAL-Team
CHIP Professional Programmieren, Ausg. 2
0971 DM/sfr. 19,-, öS 150,-

CHIP-SPECIAL-Team
CHIP Professional Programmieren, Ausg. 3
0974 DM/sfr. 19,-, öS 150,-

* Mit 5,25'' Diskette im Heft

DIE TURBO-SERIE

Geise
Praktisch arbeiten mit Turbo-Basic, Ausg. 1*
0959 DM/sfr. 28,-, öS 230,-

Kern u. a.
Turbo-Pascal, Ausgabe 1
0120 DM/sfr. 28,-, öS 230,-

Kern u. a.
Turbo-Pascal, Ausgabe 2
0310 DM/sfr. 28,-, öS 230,-

Kern u. a.
Turbo-Pascal, Ausgabe 3
0400 DM/sfr. 28,-, öS 230,-

Kern u. a.
Turbo-Pascal, Ausgabe 4
0450 DM/sfr. 28,-, öS 230,-

Kern u. a.
Turbo-Pascal, Ausgabe 5
0560 DM/sfr. 28,-, öS 230,-

Kern u. a.
Turbo-Pascal, Ausgabe 6
0580 DM/sfr. 28,-, öS 230,-

Kern u. a.
Turbo-Pascal, Ausgabe 7
0610 DM/sfr. 28,-, öS 230,-

CHIP-SPECIAL-Team
Turbo-Pascal, Ausgabe 8
0968 DM/sfr. 28,-, öS 230,-

CHIP-SPECIAL-Team
Turbo-Pascal, Ausgabe 9
0972 DM/sfr. 28,-, öS 230,-

CHIP-SPECIAL-Team
Turbo-Pascal, Ausgabe 10
0975 DM/sfr. 28,-, öS 230,-

CHIP-SPECIAL-Team **NEU**
Turbo-Pascal, Ausgabe 11
0978 DM/sfr. 28,-, öS 230,-

Geise
Turbo-Prolog*
0540 DM/sfr. 28,-, öS 230,-

Bitte benutzen Sie für Ihre Bestellung die Bestellkarten aus diesem Heft

449/89

Für jeden das Richtige.

Neue Erkenntnisse

"Wir sind in der Lage, die vielleicht leistungsfähigste Entwicklungsplattform vorzubereiten, die der Software-Welt jemals zur Verfügung gestanden hat." Martyn Jordan, Marketing Director beim amerikanischen Compiler-Entwickler Alsys, ist mit dieser Prognose, die den Einsatz von Transputern und darauf abgestimmten Übersetzerprogrammen voraussetzt, keineswegs alleine.

Mit Hilfe extrem schneller RISC-Maschinen, die mit einem 32-bit-Datenbus angesteuert werden, lassen sich nicht nur externe Geräte - wie Terminals oder Drucker - ansteuern, sondern auch beliebig komplexere Netze von Transputern realisieren.

Was ist ein Transputer? Welche Möglichkeiten bieten parallele Rechnerarchitekturen? Welche Rechner lassen sich mit Transputern aufrüsten? Welche Betriebssysteme und welche Programmiersprachen könnten eingesetzt werden? Das sind Fragen, die das vorliegende erste Transputer-SPECIAL beantwortet.

Noch hat das innovative Prozessor-Konzept nicht auf breiter Basis Einzug am Markt gehalten. Doch die Entwicklungen, die bis jetzt in die Praxis umgesetzt wurden, sind höchst interessant: Die aktuellen Einsatzbeispiele reichen vom PC-Bereich, in dem Firmen wie Atari und Commodore bereits Entwicklungen vorzuweisen haben, bis zur Großrechneranwendung, die unter dem Begriff Number-Cruncher subsummiert wird.

Die Autoren haben sich unter der Projektleitung von Ulrich Parthier bemüht, die Grundlagen der Hardware hier ebenso zu beschreiben wie das für Transputer maßgeschneiderte Betriebssystem Helios und die Programmiersprache Occam. Darauf aufbauend zeigt der Praxisteil das breite Spektrum an derzeit verfügbarer Software.

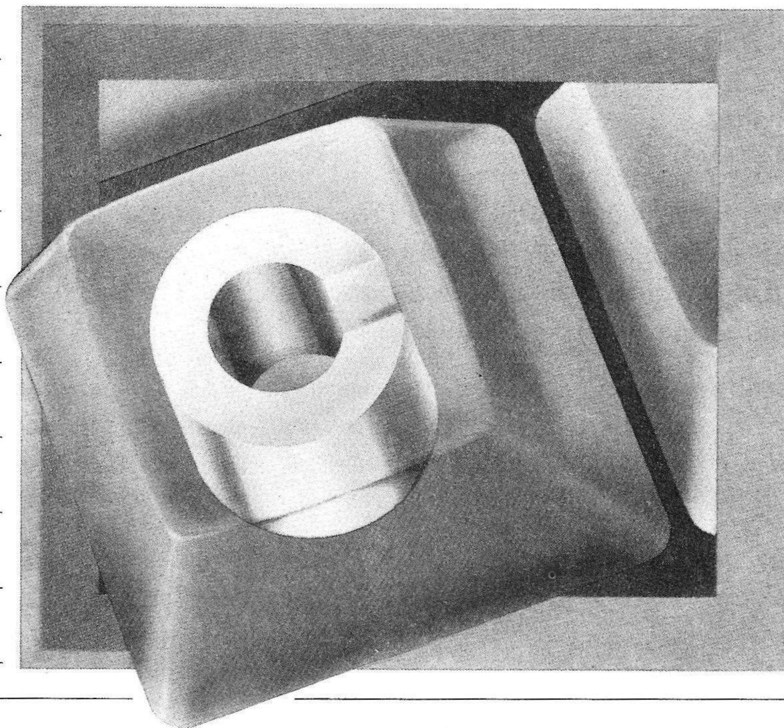
Wilhelm von Ockham, der im 14. Jahrhundert als Philosoph und Theologe in England lehrte, förderte besonders die Entwicklung der Logik. Nach ihm wurde die realtime-fähige, prozeßorientierte Programmiersprache Occam benannt, deren zugrundeliegendes Sprachmodell nebenläufige, parallele Prozesse vorsieht, die mit gleichzeitiger Kommunikation auf Kanäle zwischen gleichlaufenden Prozessen synchronisiert werden können. Daß Ihnen die neuen Programmiersprachen für die neuen Rechnerarchitekturen viele neue Erkenntnisse bringen, wünscht Ihnen

Ihre Redaktion CHIP-SPECIAL

Hans-Walter Beilstein

Wie man in C programmiert

Eine Einführung anhand von Turbo C und DeSmet-C



CHIP WISSEN

Diese Einführung in die Programmiersprache C behandelt den gesamten Wortschatz nach Kernighan und Ritchie sowie nach ANSI-Standard. Sämtliche Bibliotheksroutinen von Turbo C und DeSmet-C werden beschrieben und größtenteils in Programmen vorgeführt.

Aus dem Inhalt: C-Variable und Zahlensysteme, logische Verknüpfungen, Verzweigungen und Schleifen, Zeiger und Vektoren, Dateien und Dateiroutinen, der C-Präprozessor, Funktionen und Datentypen in C, Strukturen, Speicherverwaltung und C-Speichermodelle, Schnittstelle zum Betriebssystem, Aufgaben mit Lösungsvorschlägen. Das Buch geht außerdem ausführlich auf die Grundprinzipien der Computerprogrammierung und des Betriebssystems MS-DOS ein. Alle Programme wurden auf einem IBM-kompatiblen MS-DOS-Rechner der XT-Klasse entwickelt und getestet.

Reihe CHIP WISSEN, 504 Seiten, 53,– DM

Inhaltsverzeichnis

Grundlagen	7	Die zweite Generation
	14	Die Programmiersprache Occam
	19	Ergänzende Datenstrukturen in Occam
	27	Rekursion in Occam
Compiler	37	Der C-Compiler Par.C und seine Anwendung
	42	Parallele Programmierung in CS-Prolog
	47	Die Programmierung des Transputers
	52	Parallelverarbeitung und Ada
	58	Alternativen zu Occam
Programmierumfeld	69	Das Betriebssystem Helios
	75	Software-Entwicklung mit Spectral
Projektrealisierung	81	Konfigurierbare Transputerarchitekturen
	85	Der Amiga 2000 und seine Transputer-Implementierung
	88	Der Atari-Transputer unter Helios
	92	Occam-Alternativen im Netzwerk
Forschungsprojekte	100	Ein Multi-Transputernetz für die Simulation digitaler Systeme
	105	Multitop, ein Multiprozessor mit dynamisch variabler Topologie

Editorial 3

Impressum 112

CHIP SPECIAL

82005/88005
28,– DM

C-Praxis für Experten

Das dritte C-SPECIAL

Turbo C / Quick C
Lattice C / Microsoft C

C

- Tabellen sortieren
- Multilisten und invertierte Organisation
- Arbeiten mit hash-adressierten Listen
- Einführung in die [illegible]theorie
- [illegible]fahren in Listen [illegible]strukturen
- Bildschirmmasken erstellen
- Mit Unix in die Zukunft

[illegible] für alle [illegible]

SPECIAL

82005/87001
28,– DM

BESSER PROGRAMMIEREN

C

Der schnelle C-Einstieg

Mein erstes Programm
Umgang mit Konstanten
Alle C-Operatoren
Elementare Datentypen
Anweisungen in C
Funktionen aufrufen
Die vier Speicherklassen
Zeiger und Vektoren
Nützliche Strukturen
Der C-Präprozessor
Wichtige Routinen
Arbeiten mit Syntax-diagrammen

Mit Anwendungsbeispielen für
MS-DOS: Lattice-C,
Manx-Aztec-C, Microsoft-C
CP/M: Manx-Aztec-C, BDS-C
Atari ST: Megamax-C, Lattice-C
Amiga: Manx-Aztec-C
Macintosh: Megamax-C

CHIP SPECIAL

82005/88001
28,– DM

Professionell arbeiten mit C

Das zweite C-SPECIAL

Turbo C / [illegible]
Lattice C / [illegible]

C

- Moderne Software-Entwicklung mit Turbo C
- Elegante Programm-Strukturen durch [illegible]
- [illegible]funktionen
- [illegible]routinen
- [illegible] Strings
- [illegible]ische Daten-strukturen
- [illegible]verwaltung [illegible] FIFO und LIFO

[illegible] für alle [illegible]
• [illegible] und Kompatible
• XT und AT

CHIP SPECIAL

82005/87011
28,– DM

C auf dem Amiga

Viertes Amiga-SPECIAL

AMIGA 500
AMIGA 1000
AMIGA 2000

AMIGA

- Für Einsteiger, Umsteiger und Aufsteiger
- Das Betriebssystem richtig genutzt
- C-Compiler im Vergleich
- Compilieren – gewußt wie!
- Intuition: Screens, Fenster, Menüs, Gadgets im Eigenbau
- Grafik leichtgemacht
- Digitizer: Soundeffekte zum Selbermachen
- DOS – Disk im Griff
- Tips und Tricks

Alle Beispiel-Programme auf Diskette erhältlich

Die zweite Generation

Als vor fünf Jahren das erste Mal der Begriff "Transputer" auftauchte, stand man allgemein einem solchen neuartigen, zur Parallelverarbeitung befähigten Mikroprozessor skeptisch gegenüber. Seitdem hat diese Art von kommunizierendem Rechnerchip die zweite Generation erreicht. Grund genug also, einen tiefen Blick in die Besonderheiten von Occam und Transputer zu werfen und auch das Umfeld näher zu beleuchten.

Bei der Entwicklung des Transputers standen mehrere Prämissen im Vordergrund: hohe Verarbeitungsleistung und Geschwindigkeit, Befähigung zum Einsatz in Multiprozessoranwendungen sowie Flexibilität und problemloser Einsatz. Zur Leistungsoptimierung wurde der Prozessor des Transputers nach RISC- Gesichtspunkten entwikkelt und mit einem großen, schnellen RAM auf dem Chip ausgestattet. Unterstützung für Kommunikation in Multitransputersystemen bieten die Links, schnelle serielle Schnittstellen mit eigenen DMA-Controllern (Bild 1). Außerdem wurde zur Erleichterung des Einsatzes auf effiziente Hochsprachenimplementierung geachtet sowie eine flexible, umfassende Speicherschnittstelle mit integriertem DRAM-Controller eingebaut.

Schlüsselbegriff für Parallelverarbeitung mit Multiprozessoren ist die Kommunikation. Nach allgemein akzeptierter Auffassung ist das Bus-Konzept für die Kommunikation eines parallel arbeitenden Prozessorsystems absolut ungeeignet. Wer herkömmliche Prozessoren über einen Bus gekoppelt hat, mußte irgendwann erkennen: Ab einer gewissen Anzahl von Prozessoren läßt sich keine Leistungssteigerung mehr erreichen. Im Gegenteil, die Rechenleistung wird irgendwann wieder sinken, bis zu dem Punkt, wo die Prozessoren nur noch mit Verwaltungsaufgaben beschäftigt sind. Der Scheitelpunkt liegt in herkömmlichen Systemen bei vier bis sechs Prozessoren. Im Vergleich dazu werden heute Systeme mit mehreren Tausend Transputern aufgebaut, die dank ihrer Kopplung untereinander und durch ihre autonomen Kommunikationseinrichtungen auf dem Chip ungestört rechnen, während gleichzeitig Daten übertragen werden - der Schlüssel zur Leistungssteigerung in parallelen Systemen.

Eine klare, orthogonale Rechnerstruktur und einfache Programmierarbeit sind ebenfalls sehr wichtige Punkte. Hochkomplizierte Sprachen werden heute teilweise in Anwendungen eingesetzt, in denen kleinste Fehler katastrophale Folgen nach sich ziehen können. Wer sich jemals mit Assemblerprogrammierung beschäftigt hat, kennt die Fehleranfälligkeit und Mühseligkeit dieser Art von Entwicklung, die nur der Ausführungsgeschwindigkeit wegen gewählt wird. Deshalb war das Schaffen einer "angepaßten", effizienten Programmiersprache mit direkter Integration der Parallelverarbeitung und Kommunikation Teil der Transputerentwicklung. Erste lauffähige Versionen des Occam-Compilers wurden übrigens schon 1983 vorgestellt.

Ein Blick in das Innenleben des Transputers

Schnell arbeitende RISC-Prozessoren benötigen häufige Zugriffe auf das RAM, das daher besonders schnell sein sollte. Transputer besitzen 4 Kbyte SRAM auf dem Chip (IMS T800, 2 KByte für IMS T414, T212), die natürlich extern erweitert werden können. Laufzeitvariable werden intern abgelegt und dadurch mit Registergeschwindigkeit angesprochen. Wenn Platz bleibt, kann auch das Programm oder ein Teil davon noch intern residieren, im Normalfall gelangt das Programm zusammen mit den größeren Datenvektoren in das externe RAM.

In der CPU selbst ist alles auf Geschwindigkeit ausgelegt und nicht auf Programmierkomfort. Register gibt es dank des großen Internspeichers nicht, sondern nur den Work-

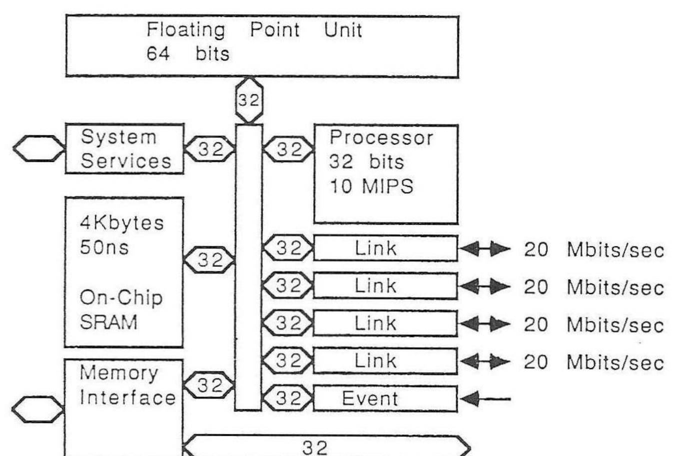


Bild 1: Blockdiagramm IMS T800

Der IMS T800 von INMOS. Neben einem 32-bit-Prozessor enthält der 1 cm^2 große CMOS-Chip eine 64-bit-Gleitkomma-Einheit, 4 KByte RAM und vier serielle Schnittstellen. Der IMS T800 ist 10 Mips bzw. 1,5 MFlops schnell.

space-Zeiger, den Befehlszeiger sowie einen dreistufigen Stack, der ALU-Funktionen wahrnimmt. Das Operandenregister wird nur benutzt, um längere Konstanten und Befehle aus den kurzen 8-bit-Befehlsfolgen zu bilden. Dennoch kommt der Transputer nur auf 104 Befehle, von denen die meisten sehr kurz und knapp sind. Lediglich einige wenige Instruktionen, die systembedingt einen hohen Stellenwert haben, lösen längere Mikrocode-Sequenzen aus. Dazu gehören die Multiplikation, das Multitasking von Prozessen auf dem gleichen Chip sowie die Kommunikation zwischen diesen Prozessen wie auch mit anderen Transputern.

Für den IMS T800 wurde dem T414 eine Fließkomma-Einheit hinzugefügt und der Befehlssatz entsprechend erweitert. Außerdem nutzte man die Gelegenheit, einige nützliche Befehle für 2-D-Blocktransfer und CRC-Check von Speicherbereichen hinzuzufügen.

Die Kommunikationsschnittstellen

Die externe Kommunikation findet über serielle Leitungen mit hoher Übertragungsrate statt, von denen jeder Transputer vier besitzt. Es handelt sich dabei um Zwei-Drahtleitungen, die pro Kanal eine Transferrate von bis zu 20 MBit/s ermöglichen. Die Links ersetzen eine Kommunikation über einen Bus und ermöglichen eine Punkt-zu-Punkt-Datenübertragung. Jede Link besitzt pro Richtung einen eigenen DMA-Controller. Dadurch ist auch das Problem des Datenaustausches in Multiprozessorsystemen gelöst, in denen keine globalen Speichermedien existieren, sondern nur jeweils ein lokaler Speicher mit feststehendem Umfang pro Prozessor zur Verfügung steht.

Durch die vier Links pro Tansputer kann die Struktur eines Multiprozessorsystems optimal an jedes Problem angepaßt werden. Aus- und Eingänge der Links sind TTL-kompatibel, ihre Reichweite läßt sich mit einfachen Treiberbausteinen erheblich vergrößern. Somit können die Rechner auch ohne teuren Hardware-Aufwand über größere Entfernungen miteinander kommunizieren.

Die Kommunikation basiert auf einem einfachen Protokoll, das ausschließlich von der Hardware ausgeführt wird. Jedes Datenpaket besteht aus elf Bits, einem Startbit, einem weiteren Einserbit, den acht Datenbits und einem Stoppbit. Für die Übertragung jedes Datenpaketes erwartet der Sender eine Bestätigung. Dieses Acknowledge erfüllt zwei Funktionen. Es besagt, daß der Empfänger bereit war, zu empfangen, und daß der Sender ein weiteres Byte senden darf. Beim IMS T800 kann diese Bestätigung überlappend mit dem Datenwort gesendet werden, so daß die Netto-Datenrate erheblich höher liegt. Und noch etwas: Die Link-Kommunikation ist prozessorunabhängig. Das heißt, der Prozessor kann weiter an seinem Problem arbeiten, während die Links ihre Datenblöcke übertragen. Speicherzugriffe der Links finden nur selten statt und werden transparent eingeschoben. Beim T800 beträgt die Netto-Datenrate 1,8 MByte/s, bei den anderen Typen 800 KByte/s.

Bei externen Speicherzugriffen tritt ein eingebauter Controller in Aktion, der vom Anwender an den Typ und die Geschwindigkeit der Speicher angepaßt werden kann. Es

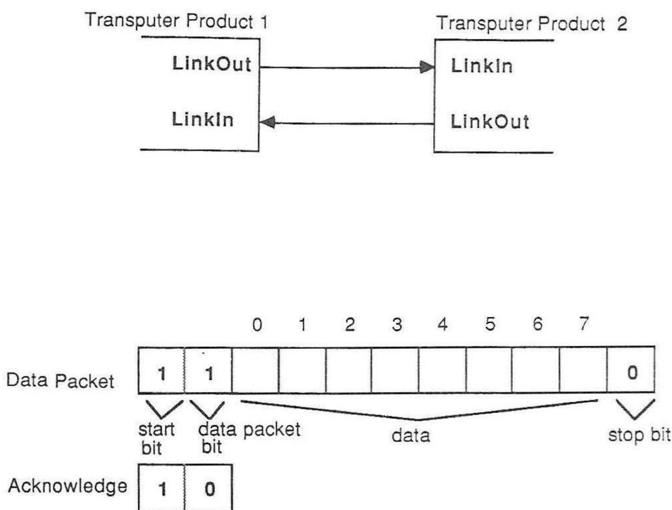


Bild 2: Blockdiagramm der Transputer-CPU

Die 32-bit-Register A, B und C formen einen dreistufigen Stack, so daß Adressierung dafür entfällt. Das Operandenregister dient zur Bildung längerer Operanden (Konstanten, Adressen etc.). Arbeitsregister legt ein Transputer stets im schnellen internen RAM an und adressiert sie relativ zum Workspace-Pointer.

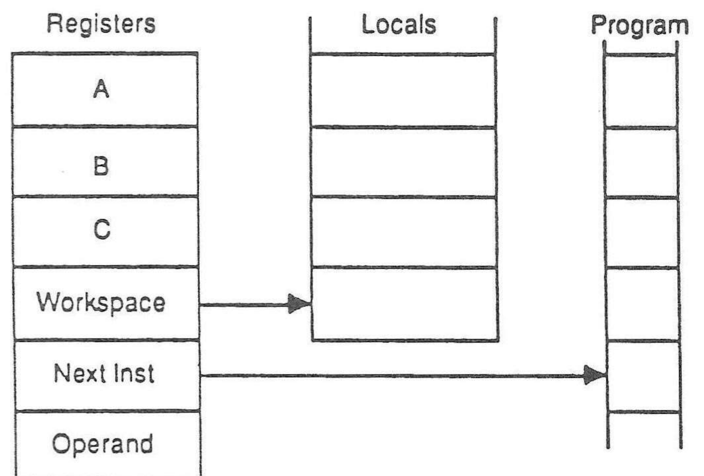


Bild 3: Protokoll der Link-Datenübertragung

Die Links des Transputers verwenden ein byteweise synchronisiertes Protokoll, das von der Hardware ausgeführt wird. Softwaremäßig behandelt das Programm nur Daten, die zu versenden sind

handelt sich dabei um eine Schnittstelle, die alle benötigten Timing- Signale zur Verfügung stellt: Zum Beispiel RAS, CAS und auch der Refresh für dynamische Speicher werden auf dem Chip erzeugt. Bei den 32-bit-Transputern T414 und T800 sind die Daten und Adressen gemultiplext, und die Zykluszeit beträgt 150 ns. Dagegen verfügt der 16-bit-Typ über getrennte Daten und Adressen. Seine kürzeste Zugriffszeit beträgt 100 ns.

Occam - die Sprache des Transputers

Die Sprache Occam mußte für den Transputer neu entwickelt werden, weil existierende Sprachen keine oder nur ungenügende Möglichkeiten für parallele Programmierung nebenläufiger Prozesse aufweisen. Occam enthält spezielle Sprachelemente, mit denen es möglich ist, sequentielle Abläufe in parallele Prozesse aufzuteilen: Dabei ist es unerheblich, ob das Programm auf einem oder auf mehreren Transputern abläuft, denn bei Occam werden im Vereinbarungsteil nicht nur die Konstanten, Variablen, Felder und Kanäle, sondern auch die Prozesse definiert. Das bedeutet, daß unabhängig von dem eigentlichen Programm die Zuordnung zwischen Software, der Logik also, und der Hardware, den ausführenden Mikroprozessoren, definiert und jederzeit ohne größere Programmänderungen modifiziert werden kann. Mehr Leistung durch mehr Transputer oder niedrigere Kosten durch Verringerung der Transputerzahl sind also ebenso leicht zu erzielen wie der Test eines Multitransputer-Programms auf nur einem Transputer.

Drei primitive Prozesse bilden die Basis von Occam:

Assignment	verändert den Wert einer Variablen
Input	übernimmt einen Wert von einem Eingabekanal
Output	sendet einen Wert durch einen Ausgabekanal

Um diese Anweisungen zu verknüpfen, stehen einige Sprachelemente zur Verfügung, die "Konstruktoren" genannt werden:

PAR	folgende Programmteile werden parallel zueinander abgearbeitet
SEQ	folgende Programmteile werden nacheinander abgearbeitet
WHILE	Schleifenoperation mit Abbruchbedingung
IF	Auswahl aus mehreren Bedingungen; Verzweigung
ALT	temporäres IF, zeitlich bedingte Verzweigung

Außer dem WHILE-Konstruktor lassen sich diese Elemente mittels des "FOR- Replikators" zur Mehrfachausführung anweisen, wie man es sonst nur von sequentiellen FOR-Schleifen gewöhnt ist. Folgendes Beispiel soll dies unterstreichen:

```
SEQ i = 0 FOR element.zahl
  vector[i] := 0
```

Hier werden nacheinander alle Elemente des Vektors zu Null gesetzt. Der FOR- Replikator, angewendet auf den SEQ-Konstruktor, erspart also nur das n-fache Schreiben der Zeile "vector[0] : = 0" und so weiter. Besondere Dimensionen bekommt der Replikator, wenn man ihn auf das PAR anwendet:

```
PAR n = 0 FOR 12
  prozess(param1, param2,...)
```

Dabei wird ein Software-Modul, hier "Prozess" genannt, zwölfmal zur gleichzeitigen Ausführung aufgerufen und bekommt einige Parameter zugewiesen, die Variablen, Konstanten oder Kanäle sein können.

Aus dieser kurzen Einführung ist schon zu ersehen, daß Occam einerseits den gleichen Komfort bietet wie andere populäre Hochsprachen und andererseits den vollen Zugriff erlaubt auf die wesentlichen Neuerungen, die der Transputer in die Hand gibt: Kommunikation und Parallelverarbeitung. Durch die Effizienz des Occam-Compilers bekommt die These wirkungsvolle Unterstützung, daß nämlich Occam "der Assembler des Transputers ist".

Transputer-Bausteine

Derzeit stehen vier verschiedene Transputer zur Verfügung. Der T414 ist ein 32-bit-Transputer mit einem 32-bit-Prozessor, 2 KByte SRAM, einer Speicherschnitt-stelle und vier Links. Beim T212 handelt es sich um einen Typ, der intern und extern vollkommen auf 16 bit ausgelegt ist, ebenfalls mit 2 KByte RAM und vier Links. Der M212 ist ein Disc-Controller, der über ein ST506/412-kompatibles Disc-Interface verfügt. Er kann auch als ganz normaler 16-bit-Transputer eingesetzt werden, verfügt aber nur über zwei Links. Alle Links können per Pin auf Übertragungsraten von 5, 10 oder 20 Mbit/s eingestellt werden, um unterschiedlichen Anforderungen zu genügen.

Seit Ende 1987 gibt es mit dem IMS T800 die zweite Generation der Transputer, der neben der normalen Integer-CPU mit einer Koprozessoreinheit für 64-bit- Fließkomma-Arithmetik ausgestattet ist. Außerdem verfügt der T800 über 4 KByte internes RAM, wobei der T800 Pin- und Software-kompatibel zu seinem "kleineren Bruder" T414 ist. Bei einer 32-bit-Kalkulation wird für den T800 mit 20 MHz Taktfrequenz eine Leistung von 1,5 MFLOPS angegeben. Eine 30-MHz-Version soll eine Leistung von 2,25 MFLOPS erbringen.

Weiterhin gibt es die "Link-Adaptoren" IMS C011 und C0L2, Peripheriebausteine für die Links, die dabei in statische oder gemultiplexte 8-bit-Ports umgesetzt werden.

Der ganz neu vorgestellte IMS C004 ist ein "Vermittlungsbaustein" für jeweils 32 Transputer-Links, der aber auch kaskadiert werden kann, um die Links von beliebig großen Transputer-Netzwerken (auch dynamisch) zu verbinden.

Module und Motherboards: Der neue Standard

Wenn man bei anderen Typen von Mikroprozessoren Einschubkarten und Backplanes verwendet, so kann man sich bei Transputern durch die weniger komplexe Link- Verschaltung auf sehr viel einfachere Module beschränken: In jedem System wird ein Rechenknoten lediglich aus einem Transputer mit einer gewissen Menge an externem Speicher gebildet. Für die Verbindung untereinander reichen einige Anschlüsse für Links, Versorgungsspannung und Takt sowie wenige Steuerleitungen (Reset etc.) aus.

Aus dieser Erkenntnis heraus wurde bei Inmos ein Standard entwickelt, der das Anschlußschema solcher Transputer-Module (TRAM) festlegt. Er unterscheidet zwischen der kleinsten Größe, die etwa die Ausmaße einer Scheckkarte besitzt und mit 16 Anschlüssen auskommt, und geradzahligen Vielfachen davon, die dann entsprechend mehrere Steckplätze belegen. Die nur mechanisch in Anspruch genommenen Steckplätze können durch Übereinanderstecken weiterer Module von anderen Transputern belegt werden.

Die "Backplane" früherer Bausysteme wird nunmehr von den "Motherboards" gebildet, die Stecksockel im übergroßen DIL-Format aufweisen, in die eine persönliche Auswahl von TRAM gesteckt werden kann. Solche Trägerkarten können bis zu zehn Module und Transputer (IMS B008 für PC) oder sogar 16 TRAM (IMS B012 im Doppeleuropa-Format) aufnehmen und dementsprechend eine große Zahl von freien Links aufweisen. Inmos hat aus diesem Grund jedes Motherboard mit "Link-Switch"-Bausteinen vom Typ IMS C004 ausgestattet, wodurch die Transputer-Verschaltung des oder der Boards von der Anwenderprogrammierung abhängt.

So wie für viele Prozessorfamilien eigene Bussysteme existieren, die jeweils am besten an die Charakteristika des Chips angepaßt sind, so setzt das Schema der Module und Motherboards Standards für die Support-Hardware für den Transputer. Obwohl erst gegen Ende 1987 eingeführt, haben die TRAM dennoch schon ein großes Marktpotential erreicht und sind bei Herstellern, wie etwa Apollo oder Niche, in Verwendung. Weitere Hersteller haben eigene Module oder Boards für diesen Standard angekündigt, und Inmos selbst hat das PC-Motherboard mit zwei T800-Modulen zur Basis seines Transputer-Entwicklungssystems gemacht.

Entwicklungsumgebungen für Transputer

Zur Unterstützung der Entwicklung von Einzel- oder Multitransputer-Systemen hat Inmos die Chance ergriffen, eine moderne und völlig neu konzipierte Entwicklungsumgebung zu schaffen. Das neue Konzept der Programmierung auch von vielen Prozessoren in einem einzigen Hochsprachenprogramm war Grund genug, alten Ballast über Bord zu werfen und neue Standards für Entwicklungskomfort einzuführen. Basis dieses Systems ist der schon beim Kunden vorhandene Gastrechner. Derzeit gibt es schon Versionen für kompatible PC und VAX- oder Micro-VAX-Rechner, in Vorbereitung ist die Unterstützung für Sun-Workstations unter Unix sowie ähnliche Rechner auf VME-Basis.

Es gibt mittlerweile zwei Grundausführungen des Entwicklungssystems: Einmal das TDS (Transputer Development System) und zum anderen das Toolset, das im Gegensatz zum TDS auf VAX- oder Sun-Implementierungen auch multiuserfähig ist und außerdem das Einbinden von Programmen erlaubt, die nicht in Occam, sondern in den Sprachen C, Fortran oder Pascal geschrieben worden sind.

Das Transputer Development System (TDS)

Das TDS ist nur für PC unter MS-DOS verfügbar und erlaubt auch nur eine Programmentwicklung in der Sprache Occam. Dafür ist es aber mit vielen Raffinessen ausgestattet, die in der Entstehungsphase eines Programms behilflich sind.

Der Entwickler befindet sich beim TDS IMS D700 praktisch immer im bildschirmorientierten Editor, in den der Compiler integriert ist. Ein Teil davon, Checker genannt, kann jederzeit auf beliebige Teile des Programms angewendet werden, um syntaktische Prüfungen vorzunehmen, die sich dann zu sofortiger Korrektur anbieten.

Ein Merkmal des Editors sind "Faltungen" (Folds), die einen Teil des Textes dem Betrachter verbergen und nur eine Überschrift zeigen. Folds dürfen beliebig verschachtelt werden und lassen sich auf Knopfdruck erzeugen oder entfernen. Damit lassen sich auch komplexe und unübersichtliche Programme (oder einfach Textstrukturen) klar und prägnant darstellen. Das im Kasten abgebildete Occam-Programm zeigt drei Stufen des "Aufblätterns": Zuerst hat man die Faltung ("...system") vor sich. Öffnet man diese Fold, hat man das ganze Programm auf dem Bildschirm - reduziert auf die wesentlichsten Elemente.

Einzelne Blöcke können dann separat angeschaut werden, wie dies im nächsten Schritt erfolgt ist. Im Teilprogramm "screen handler" sind wiederum die Details

weggefaltet, damit der Blick nicht für das Wesentliche verbaut wird.

Der Bedarf, Programme zum Nachvollziehen des Listings auszudrucken, wird dadurch sehr stark verringert. Gefördert wird aber das strukturierte Programmieren, was im Sinne der Lesbarkeit und Fehlerfindung nur von Vorteil sein kann.

Faltungen bilden die Basis der separat zu kompilierenden Programmblöcke (nur diejenigen Folds, die geändert wurden, müssen neu übersetzt werden), der Debugging-Unterstützung und auch der Dateien, die vom Filer angelegt werden.

```
...   system
```

```
{{{  system
CHAN Echo, App.in, App.out :
PAR
  ...  keyboard handler
  ...  application
  ...  screen handler
}}}
```

```
{{{  screen handler
...  declaration and initialisation
WHILE running
  SEQ
    ...  reset alarm clock
    ALT
      ...  deal with Echo channel
      ...  deal with App.out channel
      ...  and timeout
}}}
```

```
{{{  screen handler
...  declaration and initialisation
WHILE running
  SEQ
    ...  reset alarm clock
    ALT
      ...  deal with Echo channel
      {{{  deal with App.out channel
      App.out ? ch
        IF
          ch = terminate.character
            running := FALSE
          TRUE
            screen ! ch
      }}}
      ...  and timeout
}}}
```

Bild 4: Faltung als Editierfunktion

Eine Textfaltung enthält beliebig viele Textzeilen und bietet sich dem Betrachter durch die drei Punkte auf der Kommentarzeile an. Offene Faltungen werden durch drei geschweifte Klammern dargestellt

Hilfsfunktionen des TDS

Für viele der üblichen Funktionen gibt es "Libraries", die bei Bedarf vom Compiler in das Programm eingebunden werden. Sie umfassen im wesentlichen die üblichen Arithmetik-, Trigonometrie- und Ein-Ausgabe-Funktionen. Mehrere Sätze an "Utilities" helfen bei der Programmentwicklung und dem anschließenden Systemtest. Sie werden aus ihren Folds "geholt", was den Vorteil hat, daß die Funktionstasten zum Anwählen einzelner Funktionen nie überbelegt sind. Folgerichtig werden alle Utilities über Knopfdruck gestartet; es gibt kein mühevolles Eintippen der Funktion oder gar der Parameter dafür: Der Cursor zeigt den oder die Operanden. Folds übernehmen in diesem Fall die Aufgaben, die sonst von Windowing-Techniken her bekannt sind.

Durch diese Aufteilung besteht die Möglichkeit, unbegrenzt viele Utility-Sätze anzubieten. Derzeit schon enthaltene Werkzeuge umfassen unter anderem Compiler für die Gastmaschine und für 16- und 32-bit-Transputer, den Konfigurator für einen oder wirklich beliebig viele Transputer, den Loader zum seriellen Transfer des übersetzten Programms auf das oder die angeschlossenen Transputer-Boards, den Debugger für parallelverarbeitende Systeme (der Debuggen von Multiprozessorsystemen auf das Durchleuchten von einzelnen Prozessoren in jedem Schritt zurückführt), den Extractor zum Erzeugen von Programm-EPROM, Search-and-Replace-Funktionen und Speicher-Hilfsprogramme.

Besonders interessant ist die Unterstützung der programmierbaren Speicherschnittstelle des Transputers. Nachdem dieses Interface durch den ebenfalls auf dem Chip enthaltenen Memory-Controller derartig vielfältig konfiguriert werden kann (unter anderem stehen fünf Strobe-Signale zur Vergügung, die in ihrer zeitlichen Position frei gewählt werden können), läßt sich das resultierende Timing nicht mehr durch ein Datenblatt vollständig abdecken. Inmos ging deshalb den Weg der Programmunterstützung für diese Schnittstelle, die ebenfalls als Utility ausgebildet ist. Interaktiv verändert man am Bildschirm die Dauer jedes Teilzyklus und die Lage des Strobes (die zum Beispiel RAS, CAS und AMUX darstellen können), bis das resultierende, detailliert ausgerechnete Timing den Erfordernissen entspricht. Refresh- Frequenz, Early oder Late Write und dergleichen können dabei ebenfalls programmiert werden.

Der Transputer-Debugger

Auch wenn Occam noch so gut strukturierte Programme zu schreiben "aufdrängt", auch wenn die einzelnen Teilprozesse noch so geschlossen sind und nur über das Standardprotokoll der Kanäle miteinander kommunizieren - es wird immer wieder vorkommen, daß der bekannte "Bug" so gut in den "Falten" des Programms versteckt ist, daß zu dem schweren Geschütz des Debuggers gegriffen werden muß. Dabei soll allerdings angemerkt werden, daß das "Entwanzen" bei Transputern durch vielerlei Hilfsmittel und nicht nur durch eine einige Utility möglich ist.

Sehr leicht findet man Fehler, die von der externen Hardware verursacht sind: Transputer sind funktionell wie Mikrocomputer mit einem (seriell einzuladenden) Testprogramm im internen RAM betreibbar, solange wenigstens Versorgungsspannung und Takt anliegen. Damit ist ein Hardwaretest ganz ohne aufwendigen, komplizierten und teuren Emulator möglich - der Transputer emuliert sich selbst, unterstützt von der Software des TDS-Hosts.

Das Anwenderprogramm, das auch für eine beliebige Anzahl von Prozessoren geschrieben sein kann, läßt sich eine sehr einfache Konfiguration zur Ausführung auf nur einem Transputer oder sogar nur dem Entwicklungssystem bestimmen. Dabei bleibt der Occam-Code unverändert, was nur durch die Kanal- Verbindung der Prozesse möglich ist. Ein-/Ausgaben der Peripheriebausteine im externen Memorybus werden durch die Port-Anweisung funktionell der Link- Kommunikation gleichgestellt und können somit ebenso durch interne Kanäle simuliert werden. Der logische Fluß des Programms kann somit auf nur einer Maschine - auch dem Gastcomputer - verifiziert werden. Hier, aber auch bei der Ausführung auf dem Zielsystem, das aus vielen Transputern bestehen kann, die wiederum jeder mehrere parallele Prozesse ausführen, bietet der TDS-Debugger die Möglichkeit, jederzeit und mit voller Referenz auf das Quellprogramm die Fehler einer "verlausten" Anwenderprogrammierung aufzuspüren.

Besondere Aufmerksamkeit hat man hierbei dem vertracktesten Typ von Fehler bei der neuartigen Parallelprogrammierung gewidmet: Deadlock wird der Zustand genannt, wenn Kommunikation nicht mehr synchronisiert ablaufen kann, oder salopp ausgedrückt: wenn die Prozesse "aneinander vorbei reden". Da die weitere Bearbeitung eines Vorgangs davon abhängt, daß die Datenübertragung nicht nur gestartet, sondern auch quittiert abgeschlossen wird (erledigt alles die Hardware im Transputer), kann sich ein solches System selbst lahmlegen, wenn Fehler im logischen Aufbau gemacht wurden. Zur Suche dieser und ähnlicher Mechanismen (wie zum Beispiel Array-Zugriff außerhalb der festgelegten Grenzen, Division durch Null etc.) bedient sich das TDS der in jedem Transputer eingebauten Analyseeinrichtungen, die auch vom Anwender beim Aufbau eigenständiger Hardware-Systeme einbezogen werden können.

Das Toolset als Entwicklungssystem

Zielsetzung bei der Entwicklung des Toolset war der Gedanke, eine leicht portable Software für die Transputer-Entwicklung zu schaffen, die sich auf dem jeweiligen Gastsystem als ein Satz von Werkzeugen (Tools) darstellt, die unter Multi-User-Betriebssystemen wie Unix durchaus auch von vielen Benutzern verwendet werden können. Besonders wichtig war aber auch die Berücksichtigung der Transputer-Programmierung in anderen Sprachen als Occam.

Ein Programm für einen oder mehrere Transputer kann aus Abschnitten bestehen, die jeweils in verschiedenen Sprachen geschrieben wurden. Für diese Zwecke stehen neben Occam die bekannten Sprachen C, Pascal und Fortran zur Verfügung, die jeweils nach den modernsten Standards gewählt wurden und ohne jede Einschränkung verwendet werden können. Natürlich sind diese Sprachen rein sequentiell, so daß damit keine parallelen Programme geschrieben werden können. Dafür ist ein Rahmenprogramm aus wenigen Occam-Zeilen notwendig, das die einzelnen Module parallel aufruft und für ihre Kommunikation untereinander sorgt.

Man kann zusammenfassend sagen, daß C, Fortran und Pascal durch Verwendung mit dem Toolset "parallelisiert" werden, ohne daß an der Syntax der Sprachen Veränderungen vorgenommen wurden. Wie aufregend muß es für Programmierer von Fortran sein, die alten Programme praktisch ohne Änderung auf einzelne Prozessoren verteilen zu können und die vielfache Leistung des früher verwendeten, teueren Großrechners nunmehr mit nur einigen preiswerten, fingernagelgroßen Siliziumchips zu erzielen!

Im Unterschied zum TDS enthält das Toolset keinen Editor. Es kann aber der vertraute Editor des Gastsystems verwendet werden. Ansonsten sind praktisch alle Funktionen des TDS anzutreffen, jedoch in einer mehr isolierten Form, die aber ohnehin den Gewohnheiten der Entwicklung mit vergleichbaren Systemen entspricht.

Blick in die Zukunft

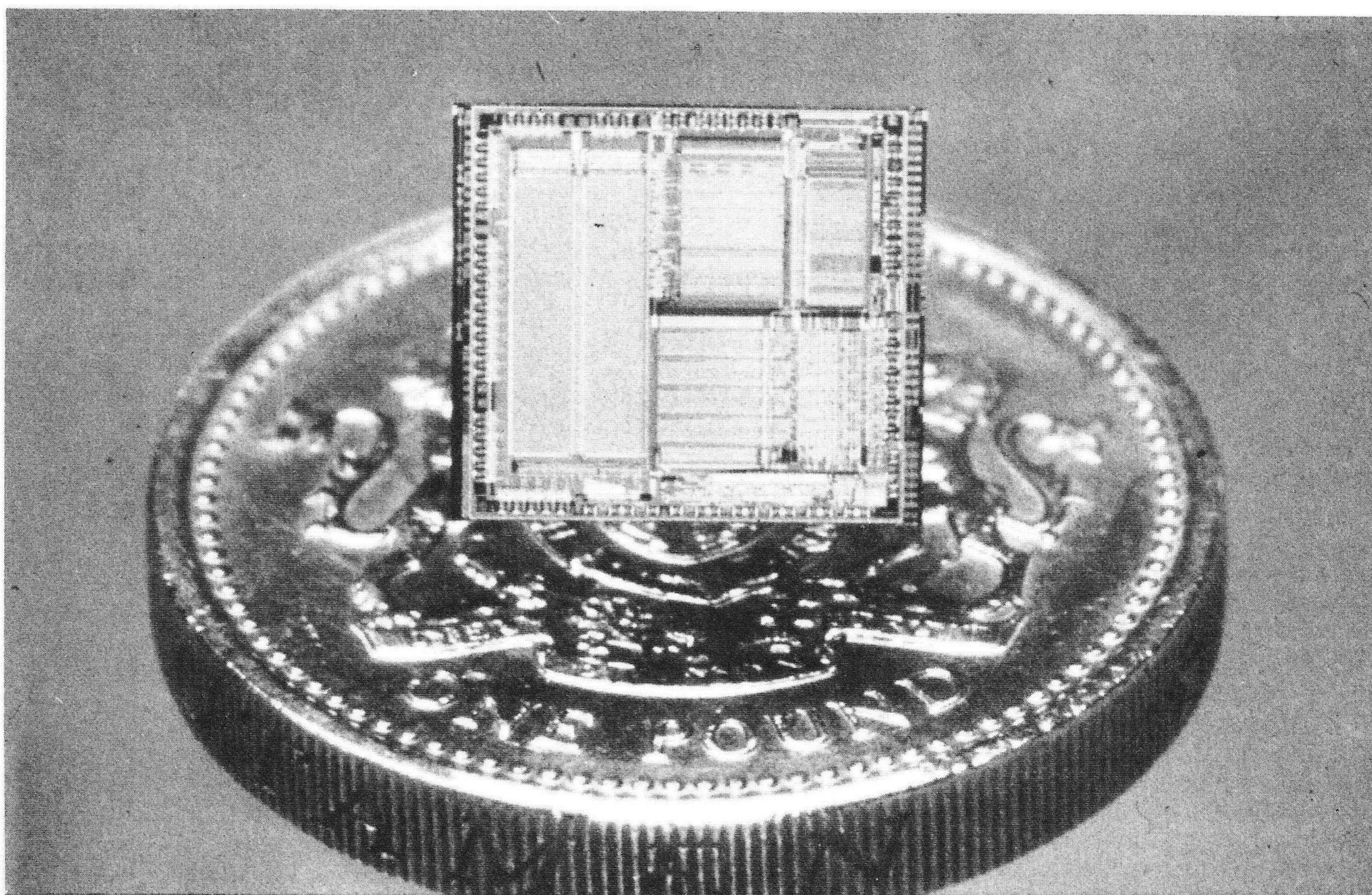
Obwohl der Transputer kein reiner RISC-Prozessor ist, muß man ihn doch am ehesten mit Vertretern dieser Architekturlinie vergleichen, da sie seiner Leistung am nächsten kommen. Nach einer Aufstellung des Marktforschungsunternehmens Dataquest hat der Transputer den Markt der RISC- Prozessoren 1987 eindeutig beherrscht und die meisten Stückzahlen abgesetzt. Die steigernde Popularität - besonders des neuen T800 - läßt vermuten, daß dieser Tendenz kein Abbruch widerfahren wird. Laufend drängen neue Hersteller mit Transputer-Ent-

wicklungen auf den Markt, von denen hier nur Atari, Commodore, Apollo und Philips genannt werden sollen.

Auch wenn in neuerer Zeit immer neue RISC-Designs mit immer phantastischeren MIPS-Zahlen gemeldet werden, so stellt das keineswegs die Existenzberechtigung des Transputers in Frage: Seine einzigartigen Einrichtungen zur beliebigen Leistungssteigerung in entsprechend großen Multiprozessorsystemen erlauben ihm, jederzeit genügend Leistung für jede geforderte Verarbeitungsgeschwindigkeit bereitzustellen. Darüber hinaus hat er aber auch Maßstäbe für Komfort und Effizienz von Hardware- und Software-Entwicklung gesetzt, die bis dato noch nicht wieder erreicht worden sind.

Für die Zukunft hat man sich weiterhin viel vorgenommen. Natürlich werden durch den steten Fortschritt der Halbleitertechnik alle Transputer schneller selektiert werden können, aber das bringt jeweils nur wenige Prozent Leistungszunahme. Eine Fortentwicklung der heutigen Transputer-Basis wird innerhalb der nächsten zwei Jahre sicherlich mehr Speicher auf dem Chip, mehr Links pro Transputer und mehr Leistung in CPU und FPU bringen.

Peter Eckelmann



Die Programmiersprache Occam

Programmierung in Occam lebt im besonderen davon, daß große Aufgaben in eine Anzahl kleinerer, unabhängiger Module zerteilt werden, die untereinander nur über "Kanäle" verbunden sind. Von jedem Modul ("Prozeß") wird angenommen, daß es gleichzeitig mit allen anderen abläuft. Die Kanäle wiederum, synchronisierende Nachrichtenleitungen, sind direkte Abbildungen der Kommunikationsmöglichkeiten des Transputers.

Eine Hardware-Konfiguration könnte zum Beispiel für jeden Prozeß einen Prozessor aufweisen, der über seine Links mit jeweils einem anderen kommunizieren kann. Oder es steht nur ein Mikroprozessor zur Verfügung, der dann die einzelnen Module durch ein Multitasking-Schema verwaltet, das in Hardware realisiert ist und nacheinander bearbeitet. In diesem Fall legt der Compiler die Kanäle als Speicherzellen im Hauptspeicher ab. Dennoch kommen in jedem Fall die gleichen Maschinenbefehle zum Tragen, egal ob ein interner Kanal verwendet wird oder eine Link dafür zugeordnet wird.

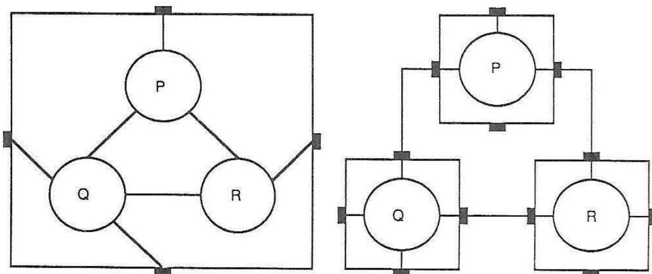


Bild 1: Kommunizierende parallele Prozeßmodule

Die Unabhängigkeit paralleler Prozesse in Occam erlaubt es, in einfacher Weise die Zahl der ausführenden Transputer zu verändern. Dabei ist es nicht notwendig, das Hauptprogramm zu modifizieren.

Multitasking und Parallelverarbeitung

In zunehmenden Maße bestimmt der Software-Aufwand die Kosten eines Mikrocomputer-Projektes. Die wesentliche Verbesserung der Leistungsfähigkeit von Mikroprozessoren und der Qualität von Hochsprachen-Compilern erlauben es allerdings in den meisten Fällen, von der mühsamen und zeitintensiven Assemblerprogrammierung abzugehen und eine höhere Abstraktionsebene zur Beschreibung der Funktionalität eines Systems zu wählen. Die Sprachen C und Pascal haben sich dort etabliert und können auf eine recht große Zahl von Anwendungen verweisen.

Unglücklicherweise sind beide Sprachen entstanden, um sehr allgemeine Probleme auf größeren Rechnersystemen zu beschreiben (C zum Beispiel diente der Entwicklung eines Betriebssystems), so daß Hilfestellungen für das Einbringen der Problematik der "echten Welt" und der "echten Zeit" weitgehend fehlen. In jedem System aus dem Bereich der Steuer-, Meß- und Regeltechnik sind dies aber unabdingbare Voraussetzungen, die dementsprechend durch Behelfsmaßnahmen einzuflechten sind.

Multitasking heißt das Schlagwort für Anwendungen, die aus mehreren relativ unabhängigen Aufgaben bestehen, die dennoch von einem einzigen Rechner quasi gleichzeitig auszuführen sind. Die Echtzeit-Multitasking-Betriebssysteme verwalten die Prozesse und weisen ihnen mehr oder weniger gerecht Rechnerzeit zu. Obwohl heute Mikroprozessoren so billig geworden sind, daß man keine so große Notwendigkeit für dieses Vielfachausnutzen der ehedem teuren Computereinheit hat, gibt es dennoch eine gute Rechtfertigung für das Beibehalten des Multitaskings: die Modularität und Portabilität der Software.

Die Software als Abbild der Realität

Doch nicht nur die Vereinfachung der Programmierung durch Unterteilung des Gesamtsystems in überschaubare kleinere Module spielt eine Rolle, sondern auch der Effekt der möglichst genauen Nachbildung der Realität.

Wenn in der Wirklichkeit zehn Sensoren und fünf Ventile zu kontrollieren sind, dann tun diese "Echtzeit"-Größen dem Anwender in den seltensten Fällen den Gefallen, wohlgeordnet und nacheinander die Aufmerksamkeit des Prozessors zu verlangen, sondern verhalten sich so, wie unsere Umwelt nun einmal ist: Unabhängig voneinander und gleichzeitig zueinander laufen reale Vorgänge ab, die am besten durch ebenso parallele und unabhängige Prozeß-Programmierungen kontrolliert werden können.

Auch erfordern viele Ereignisse völlig andere und getrennte Bearbeitung, so daß hier nicht nur keine Notwen-

digkeit besteht für die Beschreibung im Kontext nur eines einzigen Programms, sondern die fast vollkommene Isolation der Prozeßmodule Übersicht und Fehlersicherheit schafft.

Architektur der parallelen Sprache

Occam kann nahezu für jede Art von Rechnersystem Anwendung finden. Um besonders eine Implementierung auch auf verteilten Transputernetzen zu ermöglichen, die nicht über gemeinsame Speicherbereiche verfügen, ermöglicht Occam das Arbeiten mit völlig getrennten Arbeitsspeichern. Sofern also der Datenaustausch ausschließlich über die Occam-Kanäle erfolgt, kann ohne Änderung des Programms nachträglich die Verteilung der Prozeßmodule auf eine unterschiedliche Anzahl von Prozessoren vorgenommen werden.

Occam wurde auch so einfach wie möglich gehalten, um den Einsatz zu erleichtern und die Suche nach dem besten Weg des Beschreibens eines gegebenen Sachverhalts zu erleichtern. In diesem Bezug ist der Vergleich mit der Sprache C angebracht, obwohl auch eine äußere Ähnlichkeit mit Pascal vorliegt.

Konsequenterweise wurde aber auch die Möglichkeit, die ohnehin in der Realität vorliegende Gleichzeitigkeit in Occam-Programme einzubringen, bis auf die niedrigste Ebene beibehalten. Jedes kleinste Sprachelement kann zur aufeinanderfolgenden (sequentiellen) oder gleichzeitigen (parallelen) Ausführung mit den übrigen Elementen des Programms bestimmt werden. Sofern das verwendete Prozessorsystem tatsächlich über parallele Einrichtungen verfügt, macht diese Tatsache auch - abgesehen von der Schönheit und Geradlinigkeit der Programmierung - Sinn. Solche Einrichtungen besitzen heute fast alle Mikroprozessoren, man denke nur an serielle Schnittstellen oder autonome Grafik- oder Massenspeicher-Controller.

Grundelemente und Konstruktoren

Occam-Programme bestehen aus drei Grundelementen, in denen sich fast alle Primitiv-Funktionen anderer Sprachen wiederfinden:

v := e Zuweisung es Ausdrucks e zur Variablen v

c ! e Ausgabe des Ausdrucks e über den Kanal c

c ? v Eingabe vom Kanal c in die Variable v

Diese Operationen werden zusammengefaßt, um komplexere Konstruktionen zu ergeben. Dies geschieht mittels

SEQ zur sequentiellen Ausführung

IF zur bedingten Ausführung

WHILE für Schleifenbedingungen

PAR zur gleichzeitigen Bearbeitung

ALT für temporäre Alternativen

Jede Form von Konstruktion kann wiederum zu größeren Gebilden zusammengefaßt werden, wobei wiederum über die Art der Zusammenschließung neu entschieden werden kann.

Alle vier hier gezeigten "Konstruktoren" können durch Einführen eines "Replikators", der eine ähnliche Funktion wie eine "FOR"-Schleife hat, zur mehrfachen Ausführung bestimmt werden:

```
SEQ i = 1 FOR 10 -- Führt 10mal aus:
  a := a + i
```

Schleifenbildung mit unbestimmter Durchlaufzahl baut man besser mit der WHILE-Funktion auf:

```
WHILE istwert < max -- Solange nicht
Maxwert:
  sensor.a ? istwert
```

Der SEQ-Operator legt die nacheinanderfolgende Bearbeitung fest (die in praktisch allen anderen Sprachen außer KI-Sprachen implizit enthalten ist und nicht seperat angegeben werden muß).

```
SEQ
  sensor.b ? wert -- lese erst den Sensor
  anzeige ! wert -- und zeige ihn dann an
```

Alle Konstruktoren können jede Form von Element miteinander verknüpfen, angefangen vom Primitiv-Prozeß bis hin zu Programmteilen und Modulen. Daher ist es zweckmäßig, für die Elemente eine unspezifischere Schreibweise zu wählen, die wir im folgenden durch drei Punkte mit nachfolgendem Kommentar "...ein Occam-Element" einführen.

Besonders beim nun vorzustellenden PAR-Konstruktor ist diese Schreibweise vorzuziehen, verwendet man doch das PAR wesentlich häufiger, die Gleichzeitigkeit ganzer Programmblöcke zu definieren, als für einfache Primitive.

```
PAR
  ...Eingabefunktion
  ...Auswertung
  ...Ausgabefunktion
```

Erscheint es zwar auf dem ersten Blick eher logisch, das PAR durch ein SEQ zu ersetzen, weil doch die Auswertung die Eingabe voraussetzt und die Ausgabe die Auswertung, so ist gerade dies ein gutes Beispiel dafür, daß es praktisch keinen Sachverhalt gibt, den man nicht in die modularere, überschaubarere parallele Form überführen kann. Vorausgesetzt, es existieren adäquate Datenleitungen zwischen den Funktionen (Prozessen), können sie

durchaus gleichzeitig aktiv sein, sofern sie sich nur über die Kommunikation gleichzeitig auch synchronisieren.

Wie erwähnt, macht es aber durchaus auch Sinn, das PAR für die kleinsten der Elemente anzuwenden, sofern für deren Ausführung parallele Einheiten bereitstehen:

```
PAR
  x := wert1 / (a + y)
  sensor.a ? wert2
  stellglied.1 ! wert3
```

In vielen Fällen übernehmen autonome Einheiten die Ein- und Ausgaben, so daß es richtig ist, die CPU durch andere Aufgaben zwischenzeitlich zu beschäftigen.

Synchronisierende Kommunikation

Die Kanalkommunikation von Occam zwischen parallelen Prozessoren (oder dem Prozessor und der Außenwelt, die ja ein besonders "paralleler" Prozeß ist) müßte zu einem heillosen Durcheinander führen, wenn sie nicht synchronisierende Züge tragen würde. Eine Zeile mit "!" oder "?" wird erst dann verlassen, wenn die gewünschte Datenübertragung tatsächlich erfolgt ist und beide beteiligten Seiten den zugehörigen Punkt des Programms auch erreicht haben. Es kann also durchaus zu Wartezeiten kommen, wo ein Prozeß auf einen anderen warten muß. Hier zeigt sich besonders der Vorteil der PAR- Konstruktoren, die besonders einfach erlauben, die notwendige Wartezeit sinnvoll zu verbringen:

```
PAR
  Zeitglied ? Signal
  ...alternative Aufgaben
```

Im nächsten Abschnitt wird eine elegante Handhabung dieser alternativen Aufgaben mittels des ALT-Konstruktors vorgestellt, der aber noch weitaus weitreichendere Aufgaben besitzt.

Die Hochsprache kennt Echtzeit

Einen Bezug zur Zeit (zum Beispiel für Verzögerungen etc.) hat manche Hochsprache. Hier stellt Occam keine Ausnahme dar. Aufgrund der Möglichkeit, beliebig viele Prozesse (quasi-)gleichzeitig aktiviert zu haben, ist es aber nicht möglich, den Zeitgeber (Timer) anzuhalten oder zu setzen. Statt dessen wurde ein Zeitschema eingebracht, das auf dem Prinzip der Relativität beruht - allerdings die einfache ohne mathematische Formeln:

```
SEQ
  Zeitgeber ? Zeit1 - liest den
Momentanwert
  Zeitgeber ? AFTER (Zeit1 PLUS
Verzögerung)
  ...verzögerte Ausführung
```

Die erste Zeile hinterlegt den Momentanwert der Zeit in der Zelle Zeit1. Durch die zweite Zeile wird die Ausführung des folgenden Elements um die Zeit "Verzögerung" aufgeschoben. Dieser Wert kann entsprechend der Timerauflösung frei gewählt werden (und sogar auch variabel sein). Zeitabstände mißt man natürlich durch die zweifache Momentanwertaufnahme und anschließender Differenzbildung.

Das Besondere an der Handhabung der Zeit in Occam ist die elegante Möglichkeit, Kommunikation, Zeitverzögerung und sogar Interrupts harmonisch in das Gefüge der Hochsprache einzubetten. Dazu vorweg ein Ausflug in die bedingte Verzweigung (IF):

```
SEQ
  sensor.b ? wert
  IF
    wert > 0
      Resultat := Resultat + 1
    wert = 0
      SKIP -- NOP, tue nichts
    wert < 0
      Resultat := Resultat - 1
```

Das IF, selbsterläuternd wie es ist, verfügt nur über einen "Schnappschuß" der Bedingung, die zu Verzweigungen führt. Will man hingegen Verzweigungen von temporären Ereignissen abhängig machen, gelingt dies mit dem IF nur unter größtem Aufwand. Ganz besonders umständlich war bisher die Beschreibung des Systemverhaltens bei Auftreten von Interrupts und ähnlichen Ereignissen.

Das ALT von Occam faßt hingegen genau diesen Sachverhalt in einfach zu überschauende Strukturen:

```
ALT
  Interrupt ? Signal
    ...Reaktion auf Interrupt
  Sensor.a ? Wert
    ...Sensorauswertung
```

Dieses Programmfragment erlaubt die Reaktion auf einen Interrupt ebenso wie die Auswertung einer anderen Quelle von Signalen. Das ALT ist nicht deterministisch, es ist also nicht zwingend vorgeschrieben, daß die erste zeitliche Bedingung ("Guard") zuerst abgefragt wird. Für den Fall, daß dies notwendig oder gewünscht ist, kann man einzelne Elemente des ALT ebenso priorisieren wie einzelne Prozesse eines PAR-Konstrukts (durch Zusatz von PRI vor dem PAR oder ALT).

Datentypen

Zusammen mit dem Transputer wurde die Sprache Occam im vollständigen Ausbau als "Occam II" einge-

führt. Eine frühere Prototypenversion unterschied noch nicht zwischen verschiedenen Datentypen und kannte keine mehrdimensionalen Arrays. Nunmehr ist die einfache Variable ersetzt worden durch Bool- und Byte-Typen, durch Integer- und Fließkomma-Variablen einfacher und doppelter Genauigkeit sowie durch Arrays beliebiger Dimension.

Ein Anwendungsbeispiel

Zum Abschluß soll ein vereinfachtes Beispiel verdeutlichen, wie Occam die Verteilung von Aufgaben unterstützt und welche inhärenten Mechanismen benützt werden können.

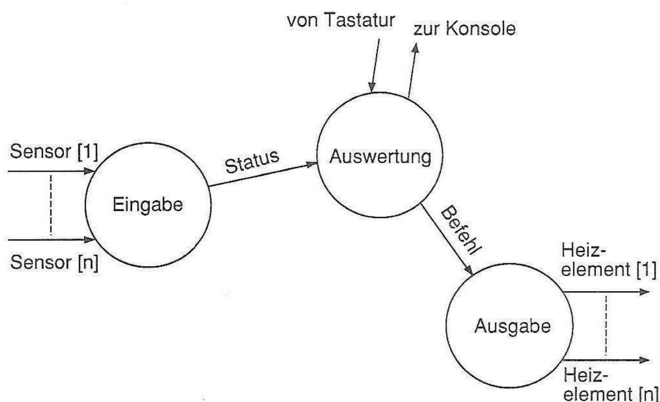


Bild 2: Temperaturregelung

Das Programm des Temperaturreglers läßt sich auf natürliche Weise in drei parallele Prozesse zerlegen, die mit jeweils einem Transputer ausgeführt werden können.

Es besteht die Aufgabe, die Temperatur eines Werkstükkes flächenbezogen konstant zu halten. Dazu werden n Sensoren abgefragt, die gleichmäßig über das Objekt verteilt angebracht sind. Als Stellgrößen stehen ebensoviele Heizelemente zur Verfügung, die einzeln verändert werden können.

Das Programm ist zur Steigerung der Übersichtlichkeit vereinfacht worden, indem die Initialisierung der Prozesse und auch die Erklärung von Variablen, Kanälen und Konstanten weggelassen wurden. Diese Dinge sind ähnlich wie in anderen Sprachen und zudem so einfach, daß dem Leser nichts wichtiges vorenthalten wird.

```
PROC Eingabe ()
  SEQ
    Zeit ? jetzt
    WHILE TRUE
      PRI ALT
        Zeit ? AFTER (jetzt PLUS interval)
          SEQ
            status ! temperaturen  -- sendet ganzen Block
            Zeit ? jetzt
        ALT i = 1 FOR n
          sensor [i] ? temperaturen [i]
            SKIP
:
```

Dieser "Eingabe"-Prozeß besteht aus einer Schleife, durch das bedingungslose "WHILE TRUE" gebildet, die fortlaufend n Sensoren abfragt, während priorisiert auf den Timer-Interrupt gewartet wird, der nach jeweils der Zeit "interval" auftritt. Dann wird der ganze Block der gemessenen Temperaturen an den Auswerteprozeß übermittelt.

```
PROC Auswertung ()
  WHILE TRUE
    ALT
      von.tastatur ? ANY
        zur.konsole ! mittel
      status ? ist.werte
        SEQ
          hilf := 0
          SEQ i = 1 FOR n
            hilf := hilf + ist.werte [i]
          mittel := hilf / n
                                          -- <>-Prüfung der Werte
          IF
            IF j = 1 FOR n
              ist.werte [j] > mittel
                befehl ! j; '-'
              ist.werte [j] < mittel
                befehl ! j; '+'
            TRUE
              SKIP
:
```

Auch hier finden wir wieder ein ALT, das diesmal aber nicht priorisiert zu werden braucht, da die Anforderung der Konsole nach Ausgabe des Temperatur- Mittelwertes hinreichend selten kommt. Wann immer der Eingabeprozeß einen neuen Werteblock überreicht, wird damit der Mittelwert gebildet und daraus die Abweichung der einzelnen Zonen errechnet. Bei Abweichung davon wird der Ausgabeprozeß veranlaßt, das entsprechende Heizelement entweder zu drosseln oder stärker zu aktivieren.

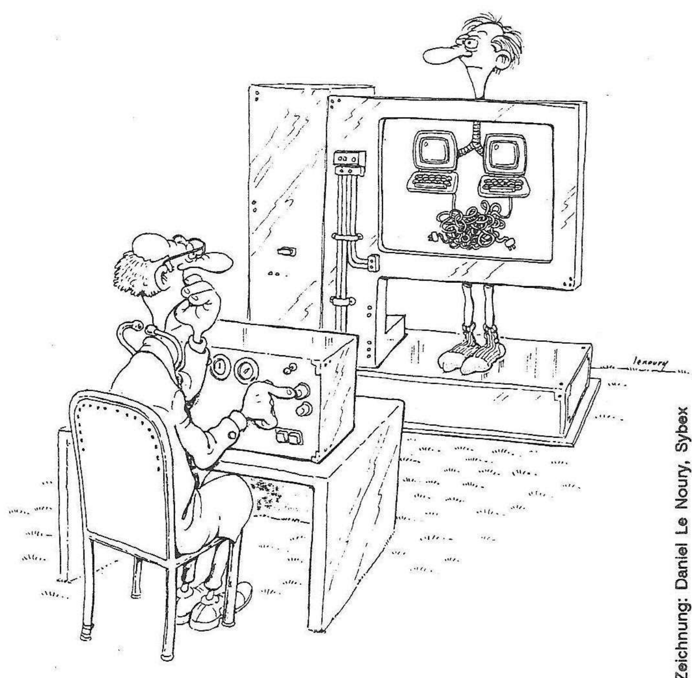
```
PROC Ausgabe ()
  SEQ
    SEQ k = 1 FOR n             -- Erstinitialisierung
      wert [k] := 0
    WHILE TRUE
      SEQ
        befehl ? zahl; auftrag
        IF
          auftrag = '+'
            heizelement [zahl] ! wert [zahl] + 1
          auftrag = '-'
            heizelement [zahl] ! wert [zahl] - 1
          TRUE
            SKIP
:
```

Der Ausgabeprozeß baut sich einen Vektor von Zahlenwerten auf, der pro Heizelement jeweils den gerade gültigen Wert enthält, so daß einfach inkrementiert oder dekrementiert werden kann. Man bemerke, daß der Index "zahl" sowohl für die Wertauswahl als auch die Kanalauswahl Anwendung findet.

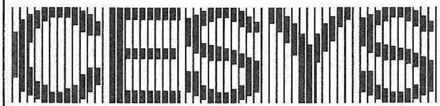
Ein derart kleines Beispiel kann natürlich nicht in allen Belangen den Programmen der Wirklichkeit nahekommen; so wäre es müßig, über die Ratio der Trennung von "Auswertung" und "Ausgabe" nachzudenken. Es sollte vielmehr anhand eines möglichst kurzen und einfachen Beispiels gezeigt werden, wie eine (an und für sich nicht parallele) Aufgabe ohne Schwierigkeiten in einzelne, zueinander parallel ablaufende Module zerlegt werden kann.

Wer immer über die zunehmende Komplexität der Software klagt, der wird die Methoden der Vereinfachung durch "divide and conquer", die Occam bietet, wohl zu schätzen wissen. Und wenn die Module, in die man sein Problem zerteilt hat, noch immer zu komplex sind - nun, dann zerlegt man weiter jeden Prozeß in eine Anzahl von mehreren kleinen Prozessen. Vielleicht bis so kleine Sequenzen herauskommen wie in unserem obigen Beispiel.

Peter Eckelmann



Zeichnung: Daniel Le Noury, Sybex



Gesellschaft f. angew. Mikroelektronik mbH

8520 Erlangen Henkestr. 8 Postfach 1844
Tel. 09131/21098 Telefax: 09131/206749

macht Transputer Entwicklungen

Kundenspezifische Applikationen

Anwenderunterstützung

Schulungen

Auszug der von uns bisher entwickelten Produkte:

mc-Transputerkarte

Ein T414 oder T800 mit 15 oder 20 MHz auf Euroformat.
2 serielle und eine parallele Schnittstelle.
128k statisches RAM und 128k EPROM oder 256k nur statisches RAM auf der Platine.
Alle Transputer Bussignale auf 96 polige VG-Leiste herausgeführt.

TD-1 PC Einsteckkarte

PC Einsteckkarte mit T414 15MHz oder T414 20MHz.
4 oder 8 MByte dynamisches RAM.

Transputer C Compiler

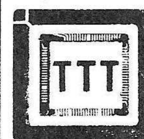
K&R C Compiler mit ANSI Erweiterungen.
Alle Möglichkeiten des Transputers können durch eine Parallelbibliothek genutzt werden.

Linkadapterkarte

PC Einsteckkarte erweitert den PC um einen Transputer kompatiblen Link Anschluss.

Verkauf und Lieferung:

Kostenlose Datenbl.,techn. Fragen bitte an:



Transputer Technologie Transfer

Postanschrift: TTT M. Dreyer
Postfach 8525 D-4800 Bielefeld 1
Tel (0521) 28 78 12 Fax: (0521) 89 45 74

Ergänzende Datenstrukturen in Occam

Die Programmiersprache Occam wurde von Inmos entwikkelt, um Algorithmen und deren Implementierung auf Transputernetzwerken programmieren zu können. Occams Datenstrukturen sind statisch, das heißt, nach einem erfolgreichen Compilerdurchlauf steht der Speicherplatzbedarf fest. Dies erlaubt, die Möglichkeit eines Run-Time-Errors durch Speicherplatzüberschreitung klein zu halten. Leider sind rekursive Algorithmen auf dynamischen Speicherplatz angewiesen und somit nicht direkt in Occam implementiert. Weiter nicht direkt unterstützt werden in Occam Datentypen wie Bäume, ein Stack oder verkettete Listen.

Diese technische Note wurde geschrieben, um Möglichkeiten aufzuzeigen, in Occam rekursive und nicht occamunterstützte Datentypen zu programmieren. Sie beschreibt einige der gebräuchlichen Datenstrukturen und hilft, sie in Occam zu implementieren. Es wird die Rekursion besprochen und Methoden, rekursive Algorithmen in iterative umzuwandeln. Außerdem werden zwei Methoden der Implementierung rekursiver Algorithmen in Occam gezeigt, die zum besseren Verständnis jeweils mit einem kleinen Beispiel versehen sind.

Beachten Sie bitte, daß sämtliche Occam-Beispiele mit dem Occam-2-Compiler compiliert werden müssen.

Datenstrukturen

Datenstrukturen werden verwendet, um Speicherung, Überprüfung und Manipulation von Information möglichst übersichtlich zu gestalten. Die folgenden Kapitel geben einen kurzen Einblick in Occam und beschreiben die Einbindung einiger gebräuchlicher Datenstrukturen. Mehr Information über Datenstrukturen entnehmen Sie bitte dem Literaturnachweis.

Die Programmiersprache Occam

Die Sprache Occam ermöglicht ein System, das nur gleichwertige Prozesse kennt, die mit anderen Transputern oder dem "Rest der Welt" über Kanäle kommunizieren können. Occam-Programme sind aus den drei folgenden Primitivprozessen aufgebaut:

```
variable := expression        -- Zuweisung
kanal1 ? variable             -- Eingabe
kanal1 ! expression           -- Ausgabe
```

Jeder Occam-Kanal versorgt einen Einweg-Kommunikationspfad zwischen gleichartigen Prozessen. Kommunikation ist synchron und ungepuffert. Die Primitivprozesse können dazu verwendet werden, Konstrukte zu formen, die ihrerseits Prozesse darstellen und wiederum Komponenten anderer Konstrukte sind. Herkömmliche sequentielle Programme können durch die Kombination von sequentiellen Konstrukten wie SEQ, IF, CASE und WHILE ausgedrückt werden.

Parallele Programme werden durch Verwendung des PAR, ALT und der Kanalkommunikation ausgedrückt. PAR wird dazu verwendet, eine beliebige Anzahl von Prozessen parallel ablaufen zu lassen, wobei diese über Kanäle miteinander kommunizieren können. Das ALT-Konstrukt erlaubt einem Prozeß, auf eine Eingabe von einer beliebigen Anzahl von Kanälen zu warten. Die Eingabe desjenigen Kanals, der zuerst senden kann, wird eingelesen und der zugehörige Prozeß ausgeführt.

Records - Ein verbreitetes Beispiel

Records sind Ansammlungen von Daten, die verschiedenen Typs sein können, die zur Erleichterung der Handhabung zusammengefaßt werden. Das traditionelle Beispiel eines Records ist das eines Arbeitnehmers, der durch mehrere Eigenschaften wie Name, Geburtstag, Gehalt usw. beschrieben wird. Die Komponenten eines Records werden Felder oder Komponenten des Records genannt. Occam erlaubt Transparenz und einfache Manipulation von Records und Feldern von Records durch die Verwendung von Abkürzungen und RETYPES.

Benutzung von INT-Arrays und RETYPE

Records können in Occam durch Verwendung eines Arrays, das ihnen als Speicherplatz dient, implementiert werden. Durch die Verwendung von Abkürzungen und der RETYPE-Anweisung können die Felder des Records angesprochen werden. Eine Implementation in Occam speichert Variablen als eine Ansammlung von Bytes in

diesem Array ab. Der RETYPE-Befehl erlaubt es, diese Byteansammlung als eine Variable beliebigen Typs anzusehen (Der Record ist geboren). So können wir einen String von Bytes als Integer-, Real- oder Boolean-Ausdruck betrachten. RETYPE ändert die Art des Compilers, die Daten anzusprechen.

RETYPE zu Real oder zu Integer muß der Wortlänge (4Byte T400 und T800, 2Byte T212) entsprechen. In der Praxis ist es gut, alle Retypes in Wortlänge zu haben. Das bedeutet, daß der Speicherplatz als Array von Integers, nicht von Bytes deklariert werden sollte, um sicherzustellen, daß die Arrays korrekt an die Wortlänge angepaßt sind und aus einer ganzen Wortanzahl bestehen. Der Compiler überprüft, ob eine Retype-Variable der Wortlänge entspricht. Der Gebrauch von Retype erlaubt mehr explizite Kontrolle über die Datenverwaltung als in manchen anderen Sprachen (siehe Pascal), wo automatische Wortanpassung Byteverschwendung gegen den Wunsch des Programmierers bedeutet.

Wir definieren zum Beispiel ein Array von 150 Mitarbeiter(employee)-Records mit je acht Wörtern Speicherplatz und übergeben die ersten wie folgt an die Prozedur UseRecord:

```
VAL INT NumberOfRecords IS 150:
VAL INT RecordWordSize  IS   8:
VAL INT BytesInWord     IS   4:

[NumberOfRecords] [RecordWordSize]INT32 Employee:
SEQ
  ...
  UseRecord(Employee[0])
  ...
```

Die Prozedur UseRecord könnte die Felder des Records mit den RETYPES und Abkürzungen, die unten dargestellt sind, ansprechen. Occam implementiert formale Parameter als Abkürzung für wirkliche Parameter. So kürzt der Parameter der Prozedur genau den momentan verwendeten Record ab, in diesem Fall Employee. Eine solche Abkürzung hilft, um die Funktion einer Prozedur zu verdeutlichen. Abkürzungen sollten in diesem Sinne auch für die Anzahl der Bytes, die jedes Feld benötigt, und für die Stelle im Array, an der das Feld liegt, benutzt werden. NameSize und NameBase wurden zwecks Klarheit explizit ausgedrückt.

```
PROC UseRecord([RecordWordSize]INT32 record)

  [RecordWordSize * BytesInWord]BYTE record.b RETYPES record:

  VAL INT NameSize IS 20:
  VAL INT NameBase IS  0:

  [NameSize]BYTE surname IS [record.b FROM NameBase FOR NameSize]:
  INT16          BirthYear  RETYPES [record.b FROM 20 FOR 2]:
  BYTE           BirthMonth IS      record.b[22]:
  BYTE           BirthDay   IS      record.b[23]:
  REAL32         Salary     RETYPES [record.b FROM 24 FOR 4]:
  BOOL           Married    RETYPES [record.b FROM 28 FOR 1]:
  --FILLER                          [record.b FROM 29 FOR 3]:

  SEQ
    ...
:
```

Danach können die Felder wie Variablen benutzt werden, wie in dem Beispiel unterhalb. Der UsageChecker stellt sicher, daß keine Zuweisung oder Eingabe direkt auf record gemacht wird, da record verwendet wurde, um Abkürzungen und RETYPES zu definieren.

```
Surname     := "Smith"
BirthYear   := 1962 (INT16)
BirthMonth  := BYTE 05
BirthDay    := BYTE 22
Salary      := 820.85 (REAL32)
Married     := TRUE
```

In dem Beispiel oberhalb wurden die RETYPES zu REALS und INTS der Wortlänge angepaßt. Die ersten fünf Worte wurden für den Namen des Mitarbeiters genutzt, die ersten beiden Bytes des sechsten Wortes sind das INT Jahr, und das letzte Wort ist der REAL32 Lohn.

Die Kosten eines RETYPES sind der Initialisierungs-Overhead für das Einführen einer Abkürzung und - für die nicht korrekt an die Wortlänge angepaßten - ein 'alignment check'. Außerdem ist der Zugriff auf eine 'retypte' oder abgekürzte Variable zeitaufwendiger als der auf eine lokale skalare Variable.

Protokolle

Benannte Protokolle ermöglichen einen einfachen Weg, Records zwischen parallelen Prozessen zu transportieren. Wenn wir das letzte Beispiel betrachten, können der ganze Record oder nur die Geburtsdatenfelder mit folgenden Protokollen übergeben werden:

```
PROTOCOL Record IS [RecordSize]BYTE:

PROTOCOL DateOfBirth IS BYTE; BYTE; INT16:
```

Arrays für jedes Feld

Eine andere Methode zur Implementierung einer Ansammlung von Records ist, für jedes Feld ein eigenes Array zu verwenden. So wird das gegebene Beispiel wie folgt implementiert:

```
[NumberOfRecords][NameSize]BYTE Employee.Surname:
[NumberOfRecords]BYTE           Employee.BirthDay:
[NumberOfRecords]BYTE           Employee.BirthMonth:
[NumberOfRecords]INT16          Employee.BirthYear:
[NumberOfRecords]BOOL           Employee.Married:
[NumberOfRecords]REAL32         Employee.Salary:
```

Oder als Mischform der beiden Methoden:

```
VAL INT DateSize IS 4:

[NumberOfRecords][NameSize]  BYTE  Employee.Surname:
[NumberOfRecords][DateSize]  BYTE  Employee.BirthDate:
[NumberOfRecords]BOOL              Employee.Married:
[NumberOfRecords]REAL32            Employee.Salary:
```

Unter Verwendung von RETYPES und Abkürzungen können nun diese Felder angesprochen werden. Dieses Beispiel verwendet den ersten Record.

```
BirthDate.b IS Employee.BirthDate[0]:

INT16 BirthYear RETYPES [BirthDate.b FROM 2 FOR 2]:
BirthMonth      IS BirthDate.b[1]:
BirthDay        IS BirthDate.b[0]:
```

Dies verhindert Alignment-Probleme, aber macht die Protokolle für die Kommunikation mit ganzen Records etwas komplexer.

Vermeiden des Alignment

Ein anderer Weg, die Probleme der Wortanpassung zu umgehen, ist die Übergabe an eine Prozedur oder Funktion. Dazu benötigen wir eine Reihe von Konstruktions-, Lösch-, Vergleichs- und Updateprozeduren. Diese Methode 'versteckt' die Details der Record-Manipulation so weitgehend wie möglich. In Anwendungen, in denen nur einige Felder durchsucht oder manipuliert werden müssen, ist diese Methode effizienter als lokales Abkürzen. So ist, wenn die Hauptdatenstruktur im Off-Chip-Memory gespeichert ist und nur die lokalen Variablen, in die kopiert werden soll, im OnChipRam liegen, die Adressierung um einiges schneller.

Die folgenden Routinen benötigen keine Wortanpassung der Daten im Record, dadurch werden mögliche Fußangeln vermieden und der Speicherbedarf auf ein Minimum reduziert. Einige Beispiele werden gegeben, der Rest sollte offensichtlich sein. Die Prozedur Name.of.Person wurde nicht als Funktion eingegliedert, da die Regeln von Occam-2-Funktionen nur die Rückübergabe einfacher Datentypen erlauben. Man kann auch BirthYear.of.Person und andere Leseroutinen als Prozedur implementieren, ähnlich wie Name.of.Person.

```
VAL INT NumberOfRecords IS 150:
VAL INT PersonSize       IS  29:

VAL INT NameBase        IS  0:
VAL INT NameSize        IS 20:
VAL INT BirthYearBase   IS 20:
VAL INT BirthYearSize   IS  2:
VAL INT BirthMonthBase  IS 22:
VAL INT BirthDayBase    IS 23:
VAL INT SalaryBase      IS 24:
VAL INT SalarySize      IS  4:
VAL INT MarriedBase     IS 28:

INT16 FUNCTION BirthYear.of.Person(VAL [PersonSize]BYTE Person)
  INT16 BirthYear:
  VALOF
    [BirthYearSize]BYTE BirthYear.b RETYPES BirthYear:
    BirthYear.b := [Person FROM BirthYearBase FOR BirthYearSize]
    RESULT BirthYear
:

PROC Name.of.Person(VAL [PersonSize]BYTE Person,
                        [NameSize]BYTE Surname)
  Surname := [Person FROM NameBase FOR NameSize]
:
```

```
PROC Update.BirthYear.of.Person([PersonSize]BYTE Person,
                                VAL INT16 BirthYear)
  VAL [BirthYearSize]BYTE BirthYear.b RETYPES BirthYear:
  [Person FROM BirthYearBase FOR BirthYearSize] := BirthYear.b
:

PROC Create.Person([PersonSize]BYTE Person,
                   VAL [NameSize]BYTE Surname,
                   VAL INT16 BirthYear,
                   VAL BYTE BirthMonth,
                   VAL BYTE BirthDay,
                   VAL REAL32 Salary,
                   VAL BOOL Married)
  SEQ
    [Person FROM NameBase FOR NameSize] := Surname

    VAL [BirthYearSize]BYTE BirthYear.b RETYPES BirthYear:
    [Person FROM BirthYearBase FOR BirthYearSize] := BirthYear.b

    Person [BirthMonthBase] := BirthMonth
    Person [BirthDayBase]   := BirthDay

    VAL [SalarySize]BYTE Salary.b RETYPES Salary:
    [Person FROM SalaryBase FOR SalarySize] := Salary.b

    VAL BYTE Married.b RETYPES Married:
    Person [MarriedBase] := Married.b
:
```

Die Stackstruktur und die mit ihr verbundenen Operationen

Einen Stack kann man sich als einen Stapel von Objekten vorstellen. Die Spitze des Stacks ist der einzige Platz, auf den Objekte gelegt oder von dem Objekte gelesen werden können. Mit einem Stack sind drei Prozeduren und zwei boolsche Funktionen verbunden:

InitStack(): initialisiert den Stack, indem der Stackpointer sp, eine freie Variable, auf NIL gesetzt wird.

Pop(d): nimmt das oberste Element, weist es d zu und erniedrigt den Stackpointer um eins.

Push(d): legt d auf den Stack und erhöht den Stackpointer um eins.

StackEmpty(): gibt TRUE zurück, wenn der Stack leer ist, sonst FALSE.

StackFull(): gibt TRUE zurück, wenn der Stack voll ist, sonst FALSE.

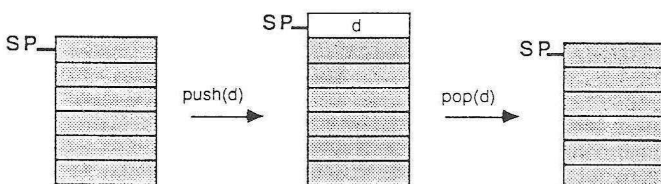


Bild 1: Stackoperationen

Einbindung in Occam

Ein Stack wird in Occam durch ein Array implementiert, in dem die Daten abgelegt werden. Der Stackpointer wird als Integerindex implementiert. Beispiel: Ein Stack, der 100 Integers speichern und ein Integer pro Zugriff von Stack nehmen oder hinauflegen kann, wird durch folgenden Code ausgedrückt:

```
VAL INT StackStep IS 1:
VAL INT NIL IS -StackStep:
VAL INT StackSize IS (100 * StackStep):

[StackSize]INT Stack:
INT       sp:

BOOL FUNCTION StackEmpty() IS sp = NIL:

BOOL FUNCTION StackFull() IS sp = (StackSize - StackStep):

PROC InitStack()
  sp := NIL
:

PROC pop(INT d)
  SEQ
    d := Stack[sp]
    sp := sp - StackStep
:

PROC push(VAL INT d)
  SEQ
    sp := sp + StackStep
    Stack[sp] := d
:
```

Ein Programm könnte verschiedene Stacks verwenden, einen für die Parametersicherung, einen anderen für die Resultate. Dieselben Subroutinen können auch für Operationen an verschiedenen Stacks verwendet werden, wenn man den Stackpointer als Parameter an die Subroutinen

übergibt. Ein Stack kann auch mehr als ein Element irgendeines Typs speichern. Werden nur Daten auf den Stack gelegt, so kann ein Stack auch durchaus Records (wie oben) speichern. Genauso kann auch mehr als ein Element zur gleichen Zeit auf den Stack gelegt werden. Zwei oder mehr "popping"-Prozeduren können verwendet werden, eine beispielsweise um zwei Elemente auf den Stack zu legen, eine andere für drei. Stacks können auch von oben nach unten anstatt von unten nach oben gefüllt werden.

Die Queue und die mit ihr verbundenen Operationen

Eine Queue ist eine Liste von Elementen, die von der Spitze der Queue genommen oder an das Ende der Queue angereiht werden können. Das Element, das als erstes in die Queue eingereiht wird, wird auch als erstes entnommen (FIFO- Prinzip). Auch hier werden drei Prozeduren und zwei boolsche Funktionen angewendet:

init.q(): löscht die Queue.

on.q(d, full): hängt das Datenelement d an das Ende der Queue an.

off.q(d, empty): nimmt ein Datenelement d von der Spitze der Queue.

s.full(): gibt TRUE zurück, wenn die Queue voll ist, FALSE im andern Fall.

q.is.empty(): gibt TRUE zurück, wenn die Queue leer ist, FALSE im anderen Fall.

Einbindung in Occam

Eine Queue kann wie der Stack als Array implementiert werden. Hier werden zwei Pointer benutzt, einer für die Spitze und einer für das Ende der Queue. Die Pointer sind als Integer-Index auf das Array implementiert. Das Array wird als zirkularer Speicherbereich verwendet, das heißt, wenn ein Element in queue[q.size] abgelegt werden soll, wird es in queue[0] abgelegt. So ist, wenn die Spitze gleich dem Ende ist, die Queue entweder voll oder leer. Um zwischen den beiden Zuständen zu unterscheiden, wird immer ein leerer Eintrag im Array belassen. Wenn die Spitze und das Ende gleich sind, ist die Queue leer, wenn die Spitze ein Element kleiner als das Ende ist, ist sie voll.

Eine Queue, die 99 integers speichert, kann folgenden Code benutzen:

```
VAL INT q.size IS (100 * 1):

[q.size]INT queue:
INT head, tail:

PROC init.q()
  SEQ
    head := 0
    tail := 0
:

PROC on.q(VAL INT d)
  SEQ
    IF
      tail < (q.size - 1)
        tail := tail + 1
      tail = (q.size - 1)
        tail := 0
    queue[tail] := d
:

PROC off.q(INT d)
  SEQ
    IF
      head < (q.size - 1)
        head := head + 1
      head = (q.size - 1)
        head := 0
    d := queue[head]
:

BOOL FUNCTION q.is.full() IS (head = (tail + 1)) OR
              ((head = 0) AND (tail = (q.size - 1))):

BOOL FUNCTION q.is.empty() IS head = tail:
```

Natürlich können auch Queues verwendet werden, um mehr als ein Element auf einmal abzuspeichern.

Verkettete Listen

Verkettete Listen ermöglichen es, Informationen zu verbinden und zu ordnen. Eine lineare oder sequentielle Liste ist die einfachste Form einer derartigen Struktur. Jedem Datenelement der Liste ist ein Pointer zugeordnet, der auf das nächste Element der Liste zeigt (Bild 2). Die Datenelemente der Liste heißen Knoten. Die Liste kann als Array von Records mit einem Extra-Feld pro Record, das den Pointer auf den nächsten Record enthält, implementiert werden. Die Pointer werden durch Integerindizes ausgedrückt.

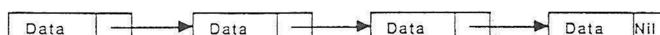


Bild 2: Eine verkettete Liste

Ein einfaches Beispiel ist in Bild 3 dargestellt. Jeder Record enthält einen Buchstaben und einen Pointer zum nächsten Knoten. Einige Records sind noch nicht belegt, diese können in einer 'free list', der Liste aller freien Records, verwaltet werden, wodurch bei einer Abspeicherung die entsprechende Prozedur einen einfacheren Zugriff auf alle freien Records hat. Die leeren Records sollten, unter Benützung ihrer Pointerfelder und einer Integervariablen, die einen Pointer auf den ersten Record enthält, verkettet werden. Abkürzungen können freizügig verwendet werden, um den Code verständlicher zu gestalten.

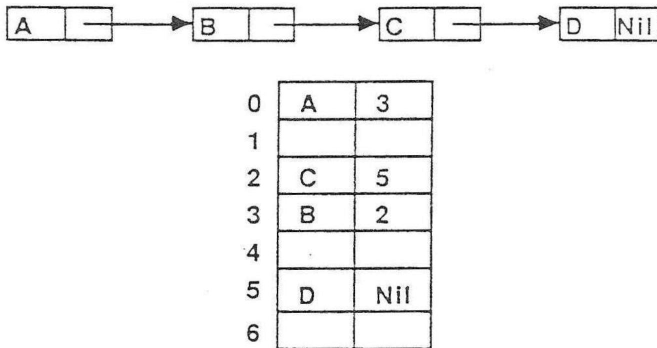


Bild 3: Ein Implementationsbeispiel

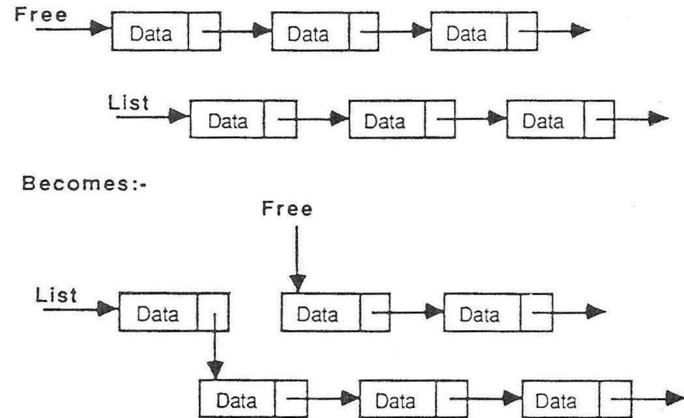


Bild 4: Einfügen eines Knoten am Beginn einer Liste

Einfügen eines Knotens am Beginn der Liste

Die folgende Prozedur illustriert das Hinzufügen eines Knotens an den Beginn einer linear verketteten Liste. Es werden die Strukturen und Abkürzungen, die oberhalb vereinbart wurden, verwendet.

```
VAL INT ListSize     IS 100:
VAL INT NodeSize     IS   2:
VAL INT NIL          IS  -1:
VAL INT ListValue    IS   0:
VAL INT ListPointer  IS   1:

[ListSize][NodeSize]INT List:
INT FreeStart: -- pointer to the start of the free list
INT ListStart: -- pointer to the start of the list

PROC insert (VAL BYTE data)

  INT NewFreeStart:
  SEQ
    node IS List[FreeStart]:
    SEQ
      node[ListValue] := INT data
      NewFreeStart := node[ListPointer]
      node[ListPointer] := ListStart
    ListStart := FreeStart
    FreeStart := NewFreeStart

:
```

Der Effekt dieser Prozedur geht aus Bild 4 hervor. Beachten Sie den Bereich der Knotenabkürzungen. Eine Variable, die zur Definition einer Abkürzung verwendet wurde, darf nicht im Bereich dieser Abkürzung einer anderen Variablen zugewiesen oder eingelesen werden. Dies erlaubt, die Abkürzung als Pointer und nicht als Kopie einer Variablen zu implementieren.

Verschiedene Strukturen können unter Verwendung von verketteten Listen mit einem oder mehreren Pointern pro Knoten aufgebaut werden. Eine beliebte Struktur ist die einer doppelt verketteten seriellen Liste. Diese hat zwei Pointer pro Knoten, einen auf den nächsten Knoten in der Liste und einen auf den Listenvorgänger. Das erlaubt ein effizientes Entfernen und Eingliedern von Knoten. Ebenso können nichtlineare Strukturen aufgebaut werden, zum Beispiel Bäume.

Die Baumstruktur und die mit ihr verbundenen Operationen

Ein Baum ist eine Form eines gerichteten Graphen. Es ist die natürliche Datenstruktur für Objekte, die in einer hierarchischen Beziehung zueinander stehen. Die Verwendung eines Baumes erhält die Beziehungen zwischen den einzelnen Objekten und erlaubt effizienten Datenzugriff. Bild 5 zeigt beispielsweise einen biblischen Stammbaum.

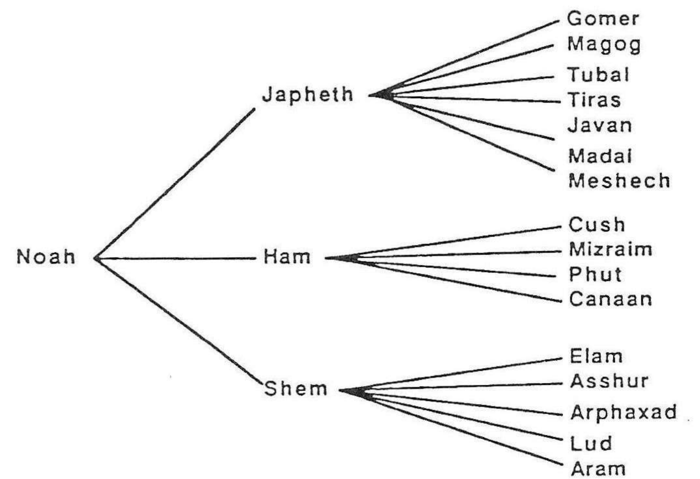


Bild 5: Ein Stammbaum

Ein binärer Baum hat folgende Eigenschaften:

- Er besteht aus einer Anzahl verbundener Knoten.
- Jeder Knoten hat höchstens zwei Pointer.
- Wenn zwei Knoten mit einem Pointer verbunden sind, so ist ein 'Zweig' zwischen ihnen.
- Ein Baum hat einen Knoten, der 'Wurzel' genannt wird, und der am Beginn des Baumes steht.

- Knoten, von denen keine 'Zweige' wegführen, werden 'Blätter' genannt.
- Jeder Knoten außer der 'Wurzel' muß von genau einem anderen Knoten abstammen. Dies stellt sicher, daß keine Kreisläufe gebildet werden und alle Knoten verbunden sind.
- Jeder Knoten außer der 'Wurzel' wird Subtree oder 'Unterbaum'genannt.

Ein binärer Baum ist die natürliche Datenstruktur für arithmetische Ausdrücke. Die Baumstruktur in Bild 6 repräsentiert den Ausdruck a+b*c.

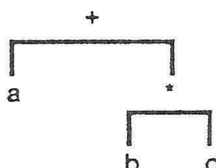


Bild 6: Ein binärer Baum

Ein binärer Suchbaum ist entweder leer oder besteht aus einer Wurzel mit zwei binären Suchbäumen, dem linken und rechten Unterbaum. Jeder Knoten enthält einen Wert, der Schlüssel genannt wird. Alle Schlüssel im linken Unterbaum müssen kleiner als der Schlüssel in der Wurzel sein. Alle Schlüssel im rechten Unterbaum müssen größer oder gleich dem Schlüssel in der Wurzel sein.

Um einen bestimmten Schlüssel zu finden, beginnt man bei der Wurzel und verfolgt den linken oder rechten Unterbaum abhängig vom Schlüssel(Wert) des Knotens. Bäume können dazu gezwungen werden, ausgeglichen zu 'wachsen', so daß n Elemente in einem Baum der Höhe log n gespeichert werden. Dadurch ist es möglich, für eine Suche innerhalb von n Objekten nur log n Vergleiche zu benötigen.

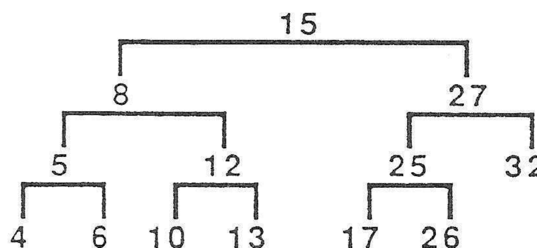


Bild 7: Ein binärer Suchbaum

Bild 7 ist ein Beispiel eines solchen Suchbaumes. Die Zahl, die in jedem Knoten vorhanden ist, ist der Integer-Schlüssel. Im Beispiel werden zwei Äste benötigt, um den Knoten mit dem Schlüsselwert 12 zu erreichen. In einer geordneten linearen verketteten Liste würde man dazu fünf Schritte benötigen (Bild 8). Man benötigt durchschnittlich log2 n Zweige (Schritte), um den gesuchten Knoten zu erreichen, im Gegensatz zu n/2 Schritten in einer sequentiellen verketteten Liste, wenn n die Anzahl der Knoten(Daten) ist.

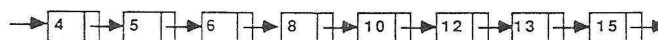


Bild 8: Ein geordnete verkettete Liste

Einbindung in Occam

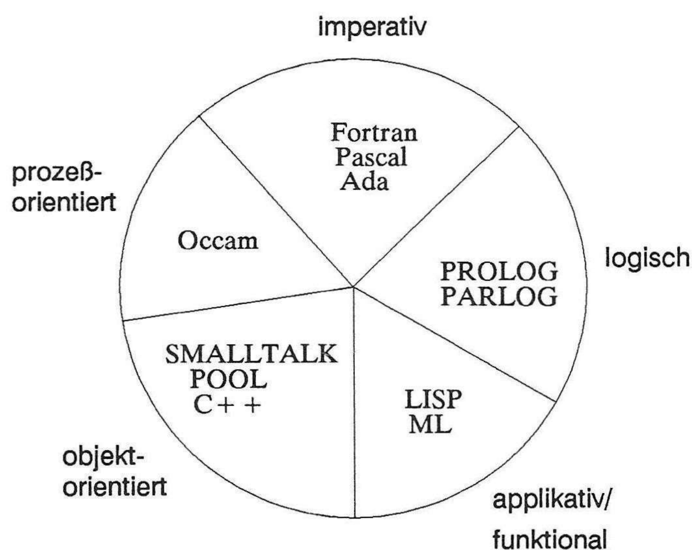
Als Beispiel eine Prozedur, die einen Baum durchsucht. Die Prozedur verwendet die aufgeführten Definitionen.

```
VAL INT tree.left  IS   0 :
VAL INT tree.right IS   1 :
VAL INT tree.value IS   2 :
VAL INT node.size  IS   3 :
VAL INT tree.size  IS  40 :
VAL INT NIL        IS  -1 :
VAL INT tree.root  IS   0 :

[tree.size][node.size]INT tree:

PROC find(VAL INT  req.value,
          BOOL found,
          VAL [tree.size][node.size]INT tree)
  BOOL done:
  [node.size]INT node:
  SEQ
    done := FALSE
    node := tree[tree.root]
    WHILE NOT done
      IF
```

```
node[tree.value] = req.value
  SEQ
    found := TRUE
    done  := TRUE
-- Search Left Subtree
(req.value < node[tree.value]) AND (node[tree.left] <> NIL)
  node := tree[node[tree.left]]
-- Search Right Subtree
(req.value >= node[tree.value]) AND (node[tree.right] <> NIL)
  node := tree[node[tree.right]]
-- req.value not in node
TRUE
  SEQ
    done  := TRUE
    found := FALSE :
```

Die besprochenen Datenstrukturen bestehen aus einem Speicherbereich und damit verbundenen Variablen wie Stackpointer, Pointer zur Spitze und zum Ende einer Queue sowie aus Pointern für den Start von Listen und Knoten von Bäumen. Diese Variablen können in dem Array gespeichert werden, in dem auch die eigentliche Datenstruktur, auf die sie sich beziehen, liegt. So kann zum Beispiel das erste Element in einem Array, das einen Stack darstellt, für den Stackpointer reserviert sein. Die damit verbundenen Unterroutinen würden dann statt einer freien Variablen dieses Element ansprechen. Dies vereinfacht die Übertragung derartiger Datenstrukturen zwischen parallelen Prozessen.

Sara Redfern, Gottfried Nestyak



Occam im Vergleich mit anderen Programmiersprachen

Rekursion in Occam

In der EDV ruft eine rekursive Funktion oder Prozedur sich selbst als Funktion oder Prozedur auf. Rekursion macht manche Programme leserlicher, und viele komplexe Algorithmen können elegant mit Rekursion gelöst werden. Um Rekursion direkt zu implementieren, benötigt man eine dynamische Speicherverwaltung. Daher erlauben viele Programmiersprachen (z.B. Fortran) keiner Unterroutine, sich selbst aufzurufen.

Das Problem, das die Rekursion aufwirft, ist die Speicherung und der Zugriff auf die Variablen im Unterprogramm. Wird das Unterprogramm erneut aufgerufen, würden die neuen Werte der Variablen die Werte der Variablen des letzten Aufrufs überschreiben. Deshalb müssen rekursive Unterprogramme die Werte abspeichern, die später benötigt werden.

Als Beispiel eine in C geschriebene faktorielle Funktion:

```
int factorial(n)
int n;
{
   if (n <= 1)
      return (1);
   else
      return (n * factorial(n-1));
}
```

Rekursive Unterprogramme können in jeder Sprache durch explizites Ausprogrammieren des Speicherns und Rückholen der Variablen geschrieben werden.

Eine rekursive mathematische Funktion ist für jeden Fall n so definiert, daß der Fall $n = 1$ gegeben ist und für jeden weiteren Fall gezeigt wird, wie er von dem vorhergehenden abgeleitet werden kann. Eine monadische rekursive Funktion beinhaltet nur einen rekursiven Aufruf auf sich selbst, eine dyadische rekursive Funktion beinhaltet zwei solcher Aufrufe.

Zum Beispiel:

```
n! ≡ f(n)      where    f(0) = 1
               and      f(n) = n * f(n-1) for n>0
```

Umwandeln von Rekursion in Iteration

Das Problem, das sich uns stellt, ist die Frage, wie eine rekursive Definition in ein iteratives Programm umgewandelt werden kann. Viele rekursive Algorithmen lassen sich elegant und effizient in einfache iterative Programme umwandeln, bei denen nicht unbedingt ein Stack benötigt wird. Hier sollen kurz monadische rekursive Definitionen des folgenden Typs[5] besprochen werden:

```
f(x) ≡ IF (x = x0) THEN a ELSE f(b(x)) * d(x)
```

a(x), b(x) und c(x) sind direkt berechenbare Funktionen von x. c(x) ist eine direkt berechenbare Funktion von x mit einem boolschen Ausdruck, und * repräsentiert eine binäre Operation.

Diese rekursiven Definitionen sind als Endrekursion bekannt, da der rekursive Aufruf beim letzten Statement auftritt. Diese Gruppe von Algorithmen läßt sich einfach in iterative übertragen. Das äquivalente iterative Programm für die obige Definition wird im unten stehenden Pseudo-Code beschrieben:

```
SEQ
  InitStack()
  nextx := x
  WHILE NOT c(nextx)
    SEQ
      Push(nextx)
      nextx := b(nextx)
  subresult := a(nextx)
  WHILE NOT StackEmpty()
    SEQ
      Pop(nextx)
      subresult := subresult * d(nextx)
  result := subresult
```

StackEmpty() ist eine Funktion, die TRUE zurückgibt, wenn der Stack leer ist, FALSE im anderen Fall. Der Code verwendet einen Stack und enthält zwei WHILE-Schleifen. Die erste Schleife errechnet die Nachfolger von x bis die Grundbedingung erreicht ist. Die zweite Schleife errechnet aus diesen Werten das Endresultat. Es wurden zwei Hilfsvariablen nextx und subresult eingeführt.

Ist b(x) eine Umkehrfunktion $h = b^{-1}(x)$, dann benötigen wir keinen Stack, um die nachfolgenden Werte von x ab-

zuspeichern, da diese direkt von der Grundbedingung berechnet werden können:

```
SEQ
  nextx := x
  WHILE NOT c(nextx)
    nextx := b(nextx)
  subresult := a(nextx)
  WHILE (nextx <> x)
    SEQ
      nextx := h(nextx)
      subresult := subresult * d(nextx)
  result := subresult
```

Das ergibt einen kompakteren Code, kann aber die Performance vermindern. Mehr über Performancesteigerung siehe [3].

Ist c(x) äquivalent zu x = x0 und x0 eine Konstante, dann nimmt die rekursive Definition folgende Form an:

```
f(x) ≡ IF (x = x0) THEN a ELSE f(b(x)) * d(x)
```

Hier ist a eine Konstante des Wertes a(x0) und $h = b^{-1}(x)$. Dies kann dann durch ein iteratives Programm ausgedrückt werden, das keinen Stack und nur eine WHILE-Schleife benötigt.

```
SEQ
  nextx := x0
  subresult := a
  WHILE (nextx <> x)
    SEQ
      nextx := h(nextx)
      subresult := subresult * d(nextx)
  result := subresult
```

Beispiel:

Als einfaches Beispiel die bereits erwähnte faktorielle Prozedur. Die äquivalente rekursive Definition ist:

```
f(x) ≡ IF (x = 0) THEN 1 ELSE f(x-1) * x
```

der äquivalente Pseudo-Code:

```
SEQ
  nextx := 0
  subresult := 1
  WHILE (nextx <> x)
    SEQ
      nextx := nextx + 1
      subresult := subresult * nextx
  result := subresult
```

der durch Ersetzung von subresult durch result weiter reduziert werden kann.

```
SEQ
  nextx := 0
  result := 1
  WHILE (nextx <> x)
    SEQ
      nextx := nextx + 1
      result := result * nextx
```

Dyadische und nicht endrekursive Definitionen

Dyadische und nicht endrekursive Definitionen sind üblicherweise nicht einfach in iterative Programme umzuwandeln. J. Arsac hat auf diesem Gebiet einige Vorarbeiten geleistet [5][7]; seine Methode sei hier zusammengefaßt.

Zuerst muß die Umwandlung sämtlicher lokaler Variablen und formaler Parameter in globale Variablen erfolgen. Als nächstes ist das rekursive Programm in einzelne Prozesse aufzuspalten. Jeder Prozeß muß benannt werden und kann auch Aufrufe von anderen Prozessen enthalten. Einer oder mehrere dieser Prozesse sind rekursiv.

Die Prozesse werden dann 'reguliert' durch Sicherstellen, daß jeder Prozeß mit einem Aufruf eines anderen Prozesses terminiert, daß jeder Prozeß nur einmal beschrieben wird - mit Ausnahme des Stopp-Prozesses, der nicht beschrieben werden muß - und daß jeder Prozeßaufruf in einer terminierenden Position auftritt.

Diese Prozesse werden dann zu einer iterativen Prozedur kombiniert und mit jedem Schritt vereinfacht. Die drei Gesetze, die das Zusammenfassen regeln, lauten:

Ersatz: Ein Aufruf eines Prozesses kann durch den Sourcecode dieses Prozesses ersetzt werden.

Identität: Wenn zwei Prozesse X und Y sich nur durch ihren Namen unterscheiden, dann kann Y immer durch X ersetzt und der Prozeß Y entfernt werden.

Entfernen der Rekursion: Das Auftreten von Rekursion wird nun endrekursiv sein, da wir nun mit regulären Prozessen arbeiten - jeder Aufruf eines Prozesses tritt nunmehr in einer terminierenden Position auf. Hat der Prozeß die Form X=f(X,Y), das heißt X hängt nur von X und Y ab, so kann er durch Einbinden der rekursiven Definitionen in eine WHILE-Schleife durch einen nicht rekursiven Prozeß ersetzt werden. Dabei muß X durch ein Null-Statement und Y in Austrittsstatement ersetzt und Y nach der Schleife plaziert werden.

Unglücklicherweise haben viele rekursive Definitionen nicht die oben beschriebene Form. Andere Algorithmen müssen deshalb entwickelt werden, deren Form von Art

des umzuwandelnden Problemes abhängt. Zwei Methoden werden nun anhand von Beispielen erklärt.

Die Pipeline-Methode

Diese Methode implementiert Rekursion durch eine Pipeline kommunizierender Prozesse. Die Pipeline besteht aus einer Feed-Prozedur, die die Startparameter in die Pipeline schiebt, den Knoten der Pipeline, die identische Prozesse sind und die rekursive Definition darstellen, während die Bleed-Prozedur die Werte von der Pipeline erhält (Bild 1).

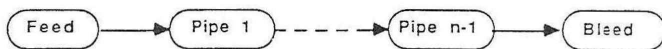


Bild 1: Eine Pipeline

Dies wird in Occam durch eine replizierte PAR-Struktur implementiert. Der Replikatorindex wird verwendet, um Arrays von Kanälen zu indizieren, so daß Kommunikation zwischen den Knoten der Pipeline möglich ist. Die Größe des Replikators muß zur Kompilierzeit feststehen. Das folgende Beispiel zeigt eine bidirektionale Pipeline.

```
[PipeSize] CHAN OF data DownPipe:
[PipeSize] CHAN OF results UpPipe:
PAR
  Feed(DownPipe[0], UpPipe[0])
  PAR i = 0 FOR PipeSize
    pipe(DownPipe[i], DownPipe[i+1], UpPipe[i], UpPipe[i+1])
  Bleed(DownPipe[PipeSize], UpPipe[PipeSize])
```

Bei jedem rekursiven Aufruf wird der nächste Knoten der Pipeline 'aktiviert', indem ihm Parameter zur Abarbeitung übergeben werden. Beim ersten Aufruf der rekursiven Prozedur werden die Parameter an den ersten Knoten in der Pipeline übergeben.

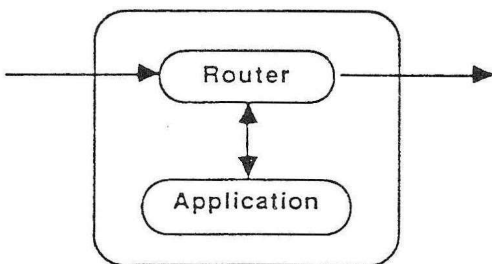


Bild 2: Pipelineknoten mit einem Router

Die Knoten der Pipeline können eine von drei Formen annehmen: die Anwendung allein (die rekursive Definition), die Anwendung mit einem Router (Bild 2) oder die Anwendung mit zwei Routern (Bild 3).

Auf Transputern sollten die Router auf hoher Priorität laufen, die Anwendung hingegen mit niedriger Priorität[3]. Dadurch muß, wenn Daten an einen wartenden Knoten weitergegeben werden sollen, der Routerprozeß nicht auf Terminierung des Applikationsprozesses warten. Die Router empfangen Daten vom verhergehenden Prozeß und senden diese Daten entweder zur Applikation (wenn diese Daten benötigt) oder zum nächsten Knoten in der Pipeline. Router transportieren auch Resultate die Pipeline hinauf zu Feed oder zu Bleed hinunter. Bei Verwendung von zwei Routern nehmen die Daten den kürzestmöglichen Weg.

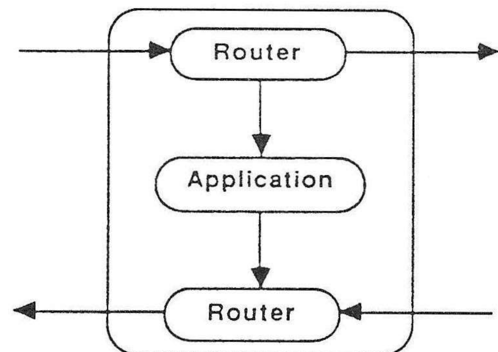


Bild 3: Pipelineknoten mit zwei Routern

Dyadische rekursive Definitionen schicken zu berechnende Parameter die Pipeline hinunter, bis ein unbeschäftigter Knoten gefunden ist. So enthält jeder Router eine boolsche Variable, die anzeigt, ob die zum Router gehörige Applikation beschäftigt ist oder nicht. Es kann eine beliebige Anzahl von rekursiven Aufrufen in der Definition befriedigt werden, indem diese Methode angewandt wird. Monadische rekursive Definitionen nützen immer den nächsten Knoten in der Pipeline, so daß sie ohne Router auskommen. Wenn die Anwendung nicht auf die Resultate des rekursiven Aufrufs angewiesen ist, kann die Kommunikation parallel zur Berechnung durchgeführt werden.

Betrachten Sie die sich gegenseitig aufrufenden Prozeduren unterhalb:

```
PROC X()                PROC Y()
  SEQ                     SEQ
    ...                     ...
    Y()                     X()
    ...                     ...
:                       :
```

Jede dieser Prozeduren ruft die andere auf, X ruft Y, die wiederum X aufruft, welche Y aufruft usw. Dies kann durch eine Pipeline implementiert werden, die aus beiden Prozeduren besteht, die sich gegenseitig abwechseln (Bild 4).

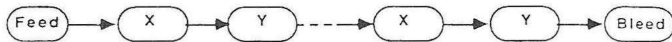


Bild 4: Pipelineknoten mit zwei Routern

Passende Router und mit Tags versehene Protokolle für die Parameterkontrolle sollten für Variationen dieses Problems, zum Beispiel für gegenseitige Rekursion kombiniert mit monadischer Rekursion, verwendet werden.

Die Anzahl der Knoten in der Pipeline muß für den schlechtesten Fall ausreichen. Das kann sehr große Redundanz bewirken. Eine Alternative dazu hat weniger Knoten als im schlechtesten Fall benötigt und wendet Farming-Methoden an, die sicherstellen, daß kein Überlauf auftritt. Ein Prozeß, der den Datenfluß in der Pipeline kontrolliert, würde das Farming, die Pufferung der zu berechnenden Daten, übernehmen und nur dann Daten senden, wenn ein Knoten fertig ist[1][8]. Alle Knoten müssen zum Terminieren gebracht werden. Die gebräuchliche Methode für diese Art von Termination ist das Aussenden eines Signals, welches über die gesamte Pipeline verbreitet wird. Dies ist durch mit Tags versehene Protokolle erreichbar.

Eine günstige Anwendung für die Pipeline-Methode ist eine sehr rechenintensive Applikation mit einem relativ kleinen Anteil an Kommunikation zwischen den Prozessen.

Die Stackmethode

Diese Methode implementiert die Rekursion durch explizites Speichern der Variablen auf einem Stack. Die zu speichernden Variablen sind die lokalen Variablen und alle Parameter für den rekursiven Aufruf. Diese Variablen sollten in globale Variablen für die iterative Prozedur verwandelt werden, um so wirksam Parameter aus der Prozedur zu entfernen. Die grundlegende Methode, einen rekursiven Algorithmus in einen iterativen umzuwandeln, ist die folgende:

Die relevanten Variablen werden vor jedem rekursiven Aufruf auf einen Stack gelegt (push) und bei der Rückkehr zur aufrufenden Prozedur wieder vom Stack geholt (pop). Der rekursive Aufruf von F in der Form F(g(x)) wird zu:

```
push(x)
x := g(x)
F
pop(x)
```

Wenn die Funktion g eine inverse h besitzt, müssen die Parameter nicht auf dem Stack gespeichert werden, da der alte Wert von x durch die Umkehrfunktion nach der Rückkehr vom rekursiven Aufruf zurückberechnet werden kann.

```
x := g(x)
F
x := h(x)
```

Wenn wir nun die rekursiven Aufrufe entfernen und die Prozedur mit einer WHILE-Schleife umgeben, die terminiert, wenn der Stack leer ist, müssen wir sicherstellen, daß wir am richtigen Punkt nach jedem Aufruf der Schleife fortfahren. Die Definition sollte nun die folgende Form haben:

F ≡ IF bool a; ELSE b; F; c; ... F; d;

Dabei sind a, b, c und d Folgen von Anweisungen und bool ein boolscher Ausdruck.

Daher sollte die Berechnung nach dem ersten rekursiven Aufruf mit c und nach dem letzten Aufruf mit d fortfahren und nach der ersten Eingabe mit b starten. Die korrekte Steuerung kann durch Speichern eines Action-Feldes auf dem Stack und Auswahl der korrekten Sequenz von Anweisungen auf folgende Art erfolgen:

```
VAL INT done.action IS 0:
VAL INT a.action    IS 1:
VAL INT b.action    IS 2:
...
SEQ
  initstack
  push(done.action)
  action := b.action
  WHILE action <> done.action
    IF
      bool
        SEQ
          a
          pop(action)
      action = b.action
        SEQ
          b
          push(c.action)
      action = c.action
        SEQ
          c
      .
      .
      .
      action = d.action
        SEQ
          d
          push(b.action)
```

Üblicherweise werden die push/pop-Operationen für die Action-Befehle und die zu speichernden Variablen auf den gleichen Stack gelegt. Für die Action- Befehle sollten Namen gewählt werden, die ihre Tätigkeit reflektieren.

Zwei Rekursionsbeispiele

Um die Implementation der Rekursion in Occam zu zeigen, werden hier nun zwei Beispiele vorgestellt, die den Stack und die Pipeline benutzen. Die Beispiele sind zwei 'klassische' Rekursionsanwendungen, das rekursive Durchschreiten eines Baumes und ein Sortieralgorithmus, der unter dem Namen Quicksort bekannt ist [9][10].

Keines dieser Beispiele eignet sich besonders für die Pipeline-Methode, sie wurden aus Gründen der Einfachheit gewählt.

Rekursives Durchschreiten eines Baumes

Die Prozedur durchschreitet den Baum, indem sie die Rekursion durch einen Stack ausdrückt. Der äquivalente rekursive Algorithmus ist wie folgt:

```
TreeTraverse(pointer)
{
 if (pointer <> NIL)
 {
     TreeTraverse (pointer->left);
     PutChar (pointer->value);
     TreeTraverse (pointer->right);
 }
}
```

Die Einbindung des Stacks

Die rekursive Definition hat keine lokalen Variablen, und der rekursive Aufruf hat nur einen Parameter, einen Pointer zum momentanen Knoten des Baumes. Daher besteht jeder Record im Stack aus zwei Feldern, nämlich einem Actionfeld und einem Pointerfeld. Der Pointer beinhaltet einen Index auf das Array, das die Baumstruktur enthält und zeigt genau auf einen Knoten im Baum. Es gibt drei Formen von Aktion: DownLeft, DownRight und Done. Diese bewirken, daß der linke oder rechte Unterbaum des Knotens, auf den gezeigt wird, durchschritten wird, oder daß der Baum vollständig durchschritten worden ist und der Prozeß terminiert.

Der folgende Pseudo-Code beschreibt den Algorithmus:

Jeder durchschnittene Knoten bewirkt daher, daß ein Record auf den Stack gelegt wird, der den rekursiven Aufruf zum Durchschreiten des rechten Unterbaumes darstellt. Der endrekursive Aufruf, der das Durchschreiten des linken Unterbaumes bewirkt, ist durch eine WHILE-Schleife implementiert.

Der Baum wird durch folgendes Array in Occam eingebunden:

```
VAL INT tree.left     IS     0:
VAL INT tree.right    IS     1:
VAL INT node.size     IS     3:
VAL INT tree.size     IS   500:
VAL INT NIL           IS    -1:
VAL INT tree.root     IS     0:

[tree.size][node.size]INT tree
```

```
PROC TreeTraverse()
  SEQ
    ...initialise
    WHILE action <> Done
      IF
        pointer = NIL
          ...  At end of subtree so back up a level
        action = DownLeft
          SEQ
            -- Remember to descend to the right from this level
            push(pointer, DownRight)
            ...  Descend a level to the left
        action = DownRight
          SEQ
            ...  Print value in node then descend to the right
            -- Start descending to the left again at the new level
            action := DownLeft
:
```

Der Stack und die damit verbundenen Operationen funktionieren auf folgende Weise:

```
VAL INT Done        IS 0 :
VAL INT DownLeft    IS 1 :
VAL INT DownRight IS 2 :
VAL INT record.size     IS 2:
VAL INT record.pointer IS 0:
VAL INT record.action  IS 1:

[tree.size][record.size]INT stack :
INT stack.pointer :

PROC push (VAL INT pointer, action)
  SEQ
    stack[stack.pointer][record.pointer] := pointer
    stack[stack.pointer][record.action]  := action
    stack.pointer := stack.pointer + 1
:

PROC pop (INT pointer, action)
  SEQ
    stack.pointer := stack.pointer - 1
    pointer := stack[stack.pointer][record.pointer]
    action  := stack[stack.pointer][record.action]
:
```

Und hier die eigentliche Procedure :

```
PROC TreeTraverse()
  INT action, pointer :
  SEQ
    stack.pointer := 0
    pointer := tree.root
    action := DownLeft
    push (NIL,Done)
    WHILE action <> Done
      IF
        pointer = NIL
          pop (pointer, action)
        action = DownLeft
          SEQ
            push (pointer, DownRight)
            pointer := tree[pointer][tree.left]
        action = DownRight
          SEQ
            print(screen, tree[pointer][tree.value])
            pointer := tree[pointer][tree.right]
            action := DownLeft :
:
```

Die Pipeline-Einbindung

Diese Prozedur durchschreitet einen sortierten Baum unter Verwendung einer Pipeline miteinander kommunizierender paralleler Prozesse. Den gleichwertigen rekursiven Algorithmus sehen Sie in "Der rekursive Algorithmus".

Jedem Knoten in der Pipeline wird ein Pointer zu einem Unterbaum des globalen Baumes übergeben. Jede Ebene im Baum entspricht einer Rekursionsebene im obigen Algorithmus. Jeder Knoten der Pipeline führt dieselbe Aktion auf einer unterschiedlichen Ebene des Baumes aus. Die maximale Anzahl der Ebenen muß vorher bekannt sein, da die Pipeline aus der entsprechenden Anzahl von Knoten besteht.

Der erste Pipelineknoten in der Pipeline erhält einen Pointer auf die Wurzel und schickt den linken Unterbaum der Wurzel an den nächsten Knoten weiter. Wenn der Unterbaum durchsucht ist, wird der Wert an der Wurzel ausgegeben und der rechte Unterbaum an den nächsten Pipelineknoten übergeben (Bild 5).

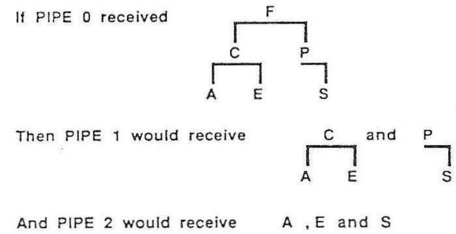


Bild 5: Ein Baumdurchschreitungsbeispiel

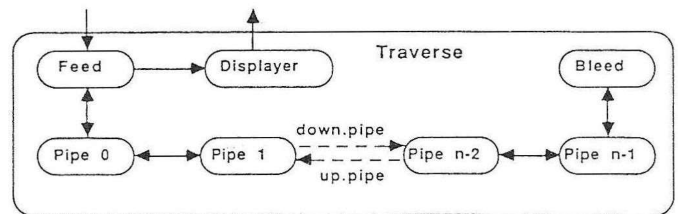


Bild 6: Baumdurchschreitungsprozeß

Bild 6 zeigt die Prozesse, die das Durchschreiten des Baumes und die Kommunikationskanäle zwischen ihnen darstellen. Die Prozesse laufen parallel zueinander, wobei die Reihenfolge der Zeichen durch Kommunikationssynchronisation zwischen den Prozessen erhalten wird.

Feed: sendet die Wurzel des Baumes zum Anfang der Pipeline und reicht die auszugebenden Daten zur Ausgabe weiter.

Bleed: arbeitet mit der letzten Ebene des Baumes, die aus NIL besteht.

down pipe: reicht die Unterbäume innerhalb der Pipeline weiter.

up pipe: überträgt die auszugebenden Daten zur Wurzel und verständigt die höheren Ebenen von der Terminierung eines Unterbaumes.

Der folgende Code zeigt die Occam-Struktur, die bleed, feed, display und die Pipeline initialisiert. Die Pipeline wird durch ein repliziertes PAR eingeführt, und die Kanäle werden durch den Replikatorindex indiziert.

```
PAR
  Feed(down.pipe[0], up.pipe[0])
  PAR i = 0 FOR SIZE(tree)
    pipe(down.pipe[i], down.pipe[i+1], up.pipe[i], up.pipe[i+1])
  Bleed(down.pipe[SIZE(tree)], up.pipe[SIZE(tree)])
  displayer()
```

Bild 7 zeigt die Kanäle, auf die jeder Knoten in der Pipeline Zugriff hat, und der Pseudo-Code unterhalb beschreibt den Algorithmus, der die Knoten implementiert.

```
SEQ
  going := TRUE
  WHILE going
    down.pipe[i] ? CASE
      subtree; pointer
      ...  received pointer to a root of a subtree
        IF
          pointer <> NIL
            SEQ
              ...  pass left subtree down pipe
              ...  pass up data values until a NIL is received
              ...  pass up data value at root of current subtree
              ...  pass right subtree down pipe
              ...  pass up data values until a NIL is received
              ...  pass up a NIL
          pointer = NIL
            ...  pass up a NIL
      ...  received termination signal
        SEQ
          going := FALSE
          ... pass down termination signal
```

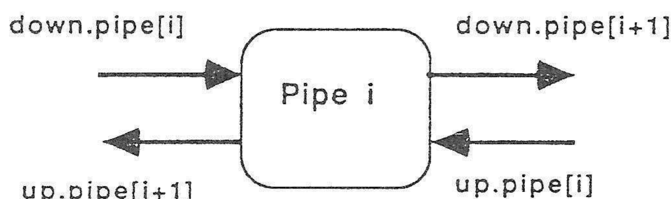


Bild 7: Ein Knoten in der Pipeline

Wenn Feed vom ersten Knoten in der Pipeline ein NIL erhält, ist der ganze Baum durchschritten. Feed sendet dann das Terminierungssignal zur Ausgabe und zu PIPE[0], die das Signal an alle anderen Pipes weitersendet. Der letzte Knoten in der Pipeline schickt dann das Terminierungssignal zu Bleed.

Rekursiver Quicksort

Der Quicksortalgorithmus sortiert einen String. Er findet die korrekte Position des ersten Elementes im String. Dieses Element wird Trennungspunkt genannt. Der Trennungspunkt wird hinter allen Elementen plaziert, die kleiner oder gleich ihm sind, und vor den Elementen, die größer sind. Dies teilt den String in den positionierten Trennungspunkt und die beiden Unterstrings vor und hinter dem Trennungspunkt, welche dann rekursiv sortiert werden. Das folgende Beispiel demonstriert diese Methode.

```
[17   16    2    10    21    12    5   19]
[12   16    2    10     5]   17  [21   19]
[10    5    2]   12   [16]  [19]   21
[ 2    5]   10
  2    5
```

Die eckigen Klammern repräsentieren dabei noch zu sortierende Substrings. Auf diese Weise ersetzt der Algorithmus das Sortieren von n Elementen durch das Sortieren von zwei Listen von weniger als n Elementen. Bild 8 zeigt ein Nassi-Schneiderman-Diagramm[12] (Struktogramm) des Algorithmus, der die Komponenten des Arrays A[L..R] sortiert, wobei gilt: A[R+1] = jedes A[L..R].

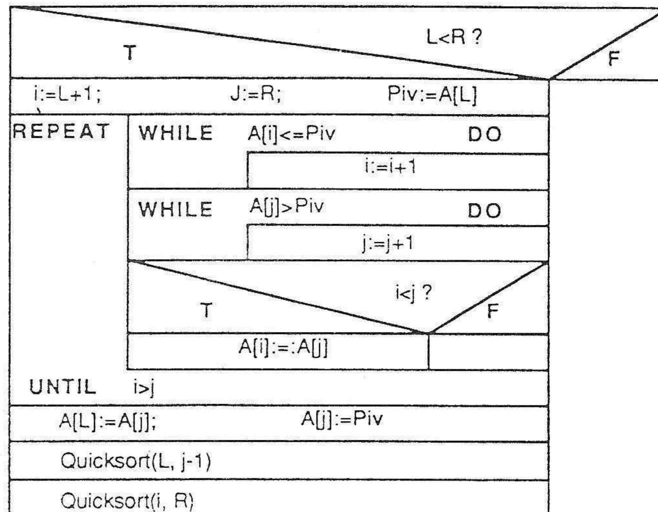


Bild 8: Der Quicksortalgorithmus als Struktogramm

Die Stack-Implementation

Die rekursive Definition hat zwei Endrekursionen mit Substrings als Parameter. Diese können auf dem Stack als zwei Integer, linker und rechter Pointer, abgespeichert werden, um den Substring innerhalb des zu sortierenden

Strings abzugrenzen. Da beide Aufrufe Endrekursionen sind, haben sie denselben Eintrittspunkt, und es werden daher nur zwei Aktionen benötigt: Done und Sort. Die Records des Stack bestehen daher aus zwei Integerpointern auf das Array, das den String und ein Integeraktionsfeld enthält. Der untenstehende Pseudo-Code beschreibt den verwendeten Algorithmus.

```
SEQ
  ...initialise
  Push(0,                  0, Done)
  Push(0, (SIZE(list) - 1), Sort)
  action = Sort
  WHILE action <> Done
    SEQ
      Pop(L, R, action)
      IF
        action = Sort
          SEQ
            ... Sort current segment
            Push(L, j-1, Sort)
            Push(i,   R, Sort)
        action = Done
          SKIP
```

Die Implementation als Pipeline

Der Quicksortalgorithmus wurde bereits im vorletzten Abschnitt beschrieben. Jeder Knoten der Pipeline besteht aus einer Sortierungsroutine und zwei Routern (Bild 9). Die rekursive Definition hat zwei Endrekursionen, deren eine leicht in iterative Form umgewandelt werden kann, indem man die linken und rechten Grenzen des momentanen Substrings ändert und die Prozedur in eine WHILE-Schleife einschließt. So werden alle links stehenden Substrings vom selben Knoten sortiert, während alle rechten Substrings die Pipeline hinunter zum nächsten nicht beschäftigten Knoten weitergereicht werden.

Wenn die Position des Trennungspunktes gefunden ist, wird der Trennungspunkt und ein Integerpointer auf seine Position im ursprünglichen String zur Feed-Prozedur geschickt. Wenn alle Zeichen (Elemente) positioniert (und damit zu Feed geschickt) sind, sendet Feed ein Terminationssignal zur Pipeline. Dieses Signal wird bis Bleed gebracht und wieder zu Feed zurückgeschickt.

Die Pipeline ist wie unter "Die Pipeline-Methode" beschrieben implementiert. Die Knoten der Pipeline werden im Pseudo-Code unterhalb beschrieben. In Bild 9 sehen Sie ein Diagramm der Knoten.

Sara Redfern und Gottfried Nestyak

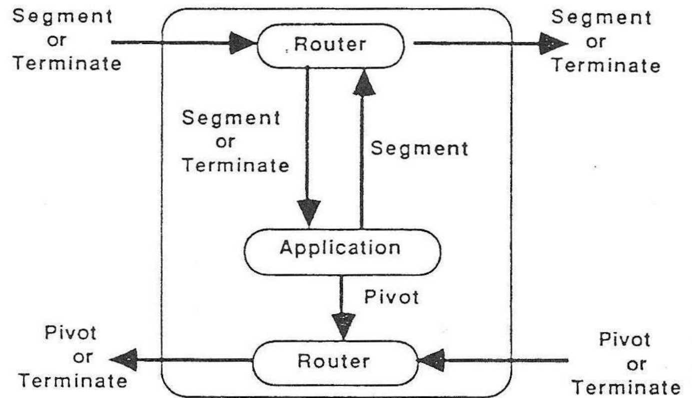


Bild 9: Datentransport in der Quicksort-Pipeline

Buchtips

[1] OCCAM 2 Reference Manual, INMOS Ltd, Prentice Hall 1987.

[2] Transputer Reference Manual, INMOS Ltd, Bristol.

[3] "Performance Maximisation", INMOS Technical Note 17, INMOS Ltd, Bristol.

[4] "Communicating Processes and OCCAM", INMOS Technical Note 20, INMOS Ltd, Bristol.

[5] "Foundations of Programming", Jacques Arsac, Academic Press, 1985

[6] OCCAM 2 language definition, David May, INMOS Ltd, Bristol.

[7] "The Transform of Recursive Definitions to Iterative Ones", Jacques Arsac, in "Tools and Notions for Program Construction", 1982, ISBN OS21248019.

[8] "Exploiting Concurrency; A Ray Tracing Example", INMOS Technical Note 7, INMOS Ltd, Bristol.

[9] "Proof of a Recursive Program: Quicksort", C.A.R. Hoare, Comp. J., No.4(1971), 391-395.

[10] "Quicksort", C.A.R. Hoare, Comp. J., 5, No.1(1962), 10-15.

[11] "Algorithms + Data Structures = Programs", Niklaus Wirth, Prentice Hall, 1976, ISBN 0-13-022418-9.

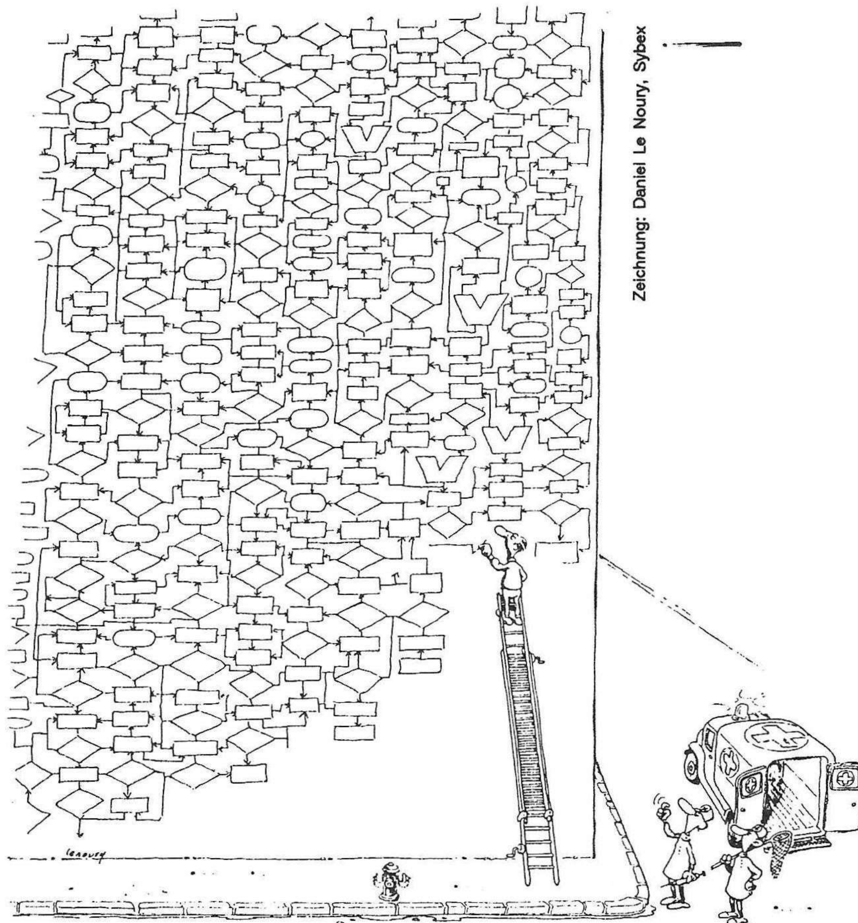
[12] "Flowchart Techniques for Structured Programming", Ben Schneiderman and Isaac Nassi, SIGPLAN Notices 8, No.8: pp12-26, August 1973.

[13] "The Art of Computer Programming 1", 2nd edition, D.E. Knuth, addison- Wesley, 1973.

```
PROC node()

  PROC up.router()
    SEQ
      ...  initialise
      WHILE going = TRUE
        ALT
          from.sort ? pivot; position
            to.last.node ! pivot; position
          from.next.node ? CASE
            pivot; position
              to.last.node ! pivot; position
            terminate
              SEQ
                going := FALSE
                to.last.node ! terminate
  :

  PROC down.router()
    SEQ
      ...  initialise
      WHILE going
        from.last.node ? CASE
          segment
            IF
              node.busy
                to.next.node ! segment
              NOT node.busy
                SEQ
                  to.sort ! segment
                  node.busy := TRUE
          terminate
            SEQ
              IF
                node.busy
                  to.next.node ! terminate
                NOT node.busy
                  SEQ
                    to.next.node ! terminate
                    to.sort ! terminate
              going := FALSE
  :

  PROC sort()
    from.down.router ? CASE
      segment
        WHILE SIZE(segment) > 0
          SEQ
            ...  sort segment
            to.up.router ! pivot; position
            to.down.router ! right.hand.segment
            segment := left.hand.segment
      terminate
        SKIP
  :

  PRI PAR
    PAR
      up.router()
      down.router()
    sort()
:
```

Die Feed- und Bleed-werden durch folgenden Pseudocode beschrieben :

```
PROC Feed()
  SEQ
    segment := list
    to.pipe ! segment
    WHILE going
      from.pipe ? CASE
        pivot; position
          SEQ
            ... place pivot in correct position
            IF
              SIZE(list) = number positioned
                to.pipe ! terminat
        terminate
          going := FALSE
:

PROC Bleed()
  from.pipe ? CASE
    terminate
      to.pipe ! terminate
:
```



Zeichnung: Daniel Le Noury, Sybex

Der C-Compiler Par.C und seine Anwendung

In erstaunlich kurzer Zeit hat sich der Transputer schon als eine Art de facto Standard für Parallelrechner in einem breiten Anwendungsspektrum durchgesetzt. Die Kürze dieser Zeit wird besonders deutlich, wenn man zum Vergleich betrachtet, wie viel länger für einen vergleichbaren Durchsetzungsgrad die vorherige Generation konventioneller Prozessoren wie etwa die 68000-Familie gebraucht hat. Im wesentlichen hat er dies wohl seiner leistungsfähigen Architektur mit ihrem hohen Grad an Einfachheit und Anschaulichkeit zu verdanken. Einen guten Anteil hieran hatte aber sicher aber auch die "Haussprache" des Transputers, das gleichzeitig mit ihm entwickelte Occam. Das Verdienst von Occam ist, daß es als Hochsprache die Entwicklung von Transputersystemen einfach macht, über konventionelle Hochsprachen hinaus aber in geradezu bestechend einfacher und anschaulicher Weise dem Programmierer alle die Rescourcen des Transputers unmittelbar an die Hand gibt, die ihn von konventionellen Prozessoren unterscheidet, wie Parallelität bis in beliebig feine Level, Kommunikation in allen Varianten, Real-Time-Programmierung, usw.

Im Zuge der Verbreitung paralleler Rechner auf Transputerbasis kam dann aber natürlich auch die Forderung nach Alternativen zu Occam auf. Diese Forderung gründet sich einerseits auf den Wunsch, vorhandene Programme in anderen Sprachen durch Neucompilation weiter - wenn auch sequentiell - nutzen zu können. Außerdem möchte nicht jeder mit einer neuen Rechnerarchitektur auch eine neue Programmiersprache lernen. Als Kompromiß wurden im Laufe der Zeit dann sogenannte "Parallel-C"-, "Parallel-Fortran"- und "Parallel-Pascal"- Compiler verfügbar. Diese Sprachen unterscheiden sich nicht, wie man es zunächst vermuten möchte, von sequentiellen Standard-Compilern, das heißt der Programmierer kann wie schon in früheren Zeiten seine Programme nur sequentiell kodieren. Durch mitgelieferte Bibliotheks-Routinen wurde es lediglich ermöglicht, solche sequentiellen Prozesse parallel zu starten und über weitere entsprechende Routinen diese miteinander Daten austauschen, das heißt kommunizieren zu lassen.

Für einige Anwendungen war dies ein brauchbarer Kompromiß. Für andere führte dies jedoch sehr schnell zu Beschränkungen, denn beim Übergang von der sequentiellen zur parallelen Anwendungsentwicklung möchte der Programmierer auch neuartige Denkweisen und Algorithmen verwenden können, bis hin zu massiv paralleler Programmierung und sogenannten "Fine Grain Parallelism". Nicht zuletzt bestimmt die Programmiersprache auch die Weise, wie man ein Problem durchdenkt, was man in ihr lösen möchte. Zum Erlernen dieser Denkweise war Occam sicher bisher konkurrenzlos. Mit einem neuen C-Compiler sind jetzt aber die besten Eigenschaften sowohl der Parallelverarbeitungswelt als auch der Standardsoftware-Welt verbunden worden.

Die erste echte Alternative zu Occam

Das von dem niederländischen Software-Haus parsec mit Unterstützung von Parsystec entwickelte und von der Parsytec-Schwester Paracom in Aachen vertriebene Par.C. ist der erste C-Compiler für Transputer, in dem die aus Occam bekannten parallelen Konstrukte als Spracherweiterung integriert sind. Damit ist erstmals den Anhängern von C ein Werkzeug an die Hand gegeben, mit einer vertrauten Sprache in der gleichen Weise wie Occam parallel programmieren zu können, ja es geht sogar darüber insofern hinaus, als in Occam nur statisch definierbare Parallelität mit der vollen in C möglichen dynamischen Programmierung möglich ist. Desweiteren gestattet dieser Ansatz die Erzeugung von wesentlich effizienterem Code als bei Verwendung von Bibliotheksfunktionen. Nicht zu unterschätzen ist auch eine deutlich übersichtlichere Struktur des Programms, wenn die Parallelität unmittelbar in der Programmiersprache ausdrückbar ist.

Systemvoraussetzungen

Par.C wurde entwickelt für Transputerrechner und ist derzeit bereits lauffähig auf Parsytecs gesamter Megaframe-Systemfamilie sowie auf allen Inmos-BOO4- kompatiblen Karten. Der Par.C-Compiler selbst läuft (auch aus Geschwindigkeitsgründen) auf einem der Transputer des Anwenders unter dem Entwicklungssystem Megatool. Dies hat für den Par.C-Benutzer eine angenehme Konsequenz: Da über entsprechende Schnittstellenkarten und Software-Treiber Megatool auf einer breiten Pallette unterschiedlichster Rechner und Betriebssysteme läuft, kann man mit Par.C von Anfang an parallele Programme entwickeln auf IBM-PC/XT/AT (und kompatiblen Rechnern), IBM-PS/2, DEC (micro-) VAX-Maschinen, Apple

Macintosh II, Sun-3/Sun-4 Workstations und anderen VME- Systemen (z.B. mit OS-9/68k oder Unix V). Eine Version zum Einsatz unter Helios ist geplant.

Für PC-basierte Transputerkarten steht darüber hinaus noch ein sogenannter Cross-Compiler zur Verfügung, mit dessen Hilfe man auf dem PC-Prozessor den entsprechenden Transputercode erzeugen kann.

Standard

Par.C ist eine Implementierung der Kernighan & Ritchie [1] Sprachdefinition mit den von Harbison & Steele [2] empfohlenen Spracherweiterungen. Mit einer vollständigen Anpassung an den ANSI-Standard ist im kommenden Jahr zu rechnen.

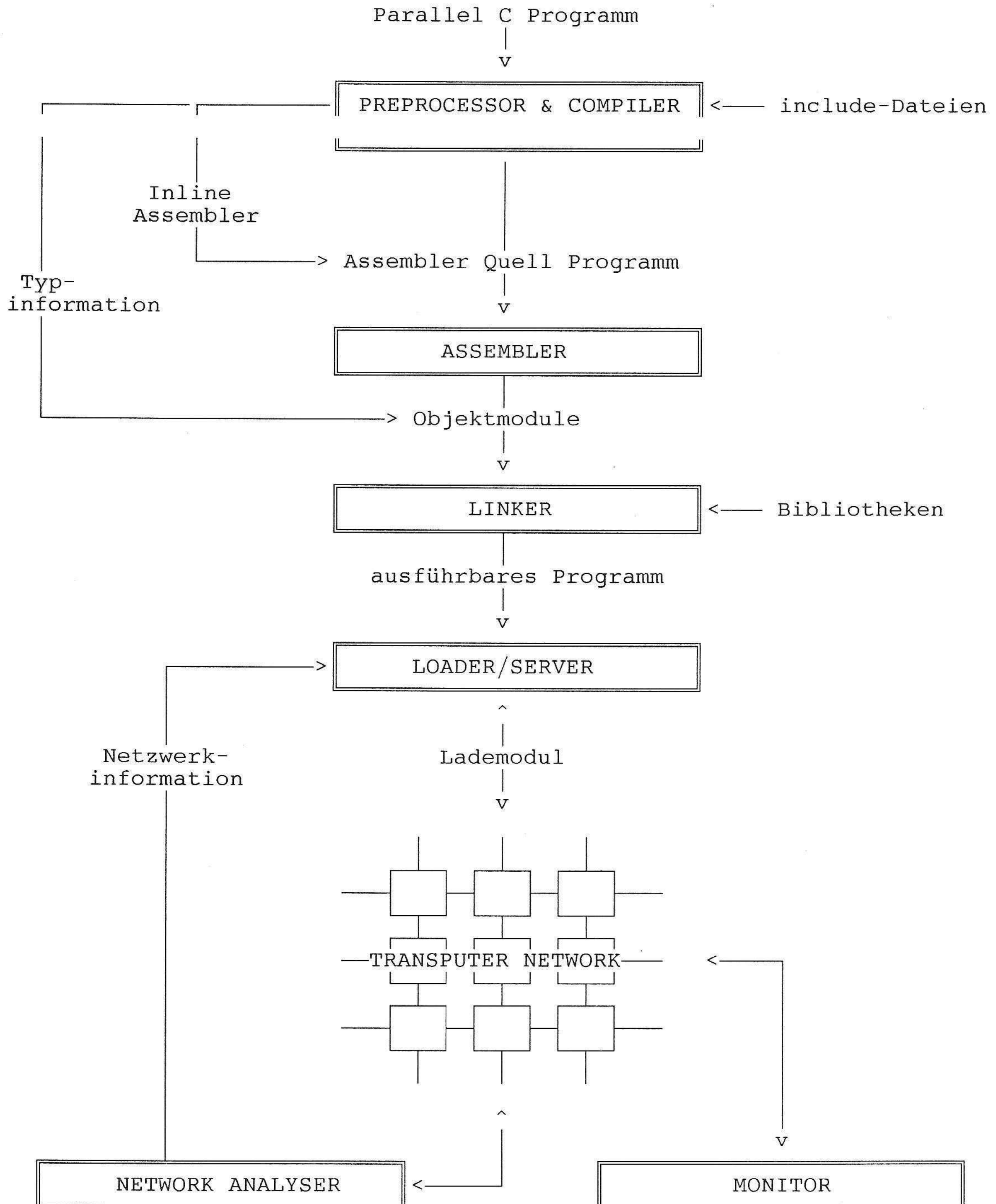


Bild 1: Struktureller Aufbau des Par.C Systems

Laufzeitsystem und Loader

Par.C enthält ein spezielles Laufzeitsystem, das aus einem minimalem Kern besteht, der auf jedem Transputer im Netzwerk läuft. Dieser Kern kümmert sich um die Speicherverwaltung und die Datenein- und Ausgabe. Ein ausgefeilter Loader gestattet es, die Größe und die Konfiguration - also die Struktur, in der die Prozessoren verschaltet sind - zu verändern, ohne bestehende Programme neu zu compilieren oder linken zu müssen. Dies gestattet eine Skalierung der Programmlaufzeit einfach durch Hinzufügen weiterer Prozessoren.

Spracherweiterungen

Neben dem üblichen Vokabular von C übersetzt Par.C eine Reihe weiterer Schlüsselworte, die die Formulierung paralleler Algorithmen unmittelbar in C unterstützen. Um diese einmal grob vorzustellen, werfen wir einen Blick auf folgendes Beispielprogramm:

```
/* Programmbeispiel in Par.C */

typedef struct {
      int Address;
      int Contents[100];
   } message;

MessagePasser (ChanIn, ChanOut)
channel ChanIn[], ChanOut[];
{
   channel ToBuffer[SIZE];
   int     i,j,k,n,Delay;
   message InMessage;

   Delay = MilliSecondsToTicks(10000);
   n = SIZE;

   par {
      par (i=0; i<<n; i++) {
         while (TRUE)
            ToBuffer[i] = (message) ChanOut[i];
      }
      while (TRUE) {
         select within Delay {
            alt (j=0; j<<n; j++) guard &ChanIn[j] : k = j; break;
            alt timeout : printf(".");
         }
         Message = ChanIn[k];
         k = Message.Address;
         ChanOut[k] = Message;
      }
   }
}
```

Dieses Programm demonstriert die Kommunikation zwischen SIZE parallelen Senderprozessen und einem Empfängerprozeß. Außerdem wird ersichtlich, wie einfach das Starten paralleler Prozesse sowie das asynchrone Empfangen von Nachrichten ist, auch wenn Zeitbedingungen einzuhalten sind oder nicht bekannt ist, von welchen Kanal die nächste Nachricht empfangen werden soll.

Schlüsselworte, um die Standard C erweitert wurde:

channel: neuer Datentyp, zur Kommunikation paralleler Prozesse

timer: neuer Datentyp, Hardware-Timer des Transputers

par: neues Statement, zum Starten paralleler Prozesse

select: neues Statement, zum Warten auf ein Ereignis

within: zur Definition des Timeout beim Warten (optional)

alt: zur Definition von möglichen Ereignissen bei "select"

cond: Bedingung zur selektiven Aktivierung von "alt"

guard: zur Definition des "channel" auf dem ein Ereignis erwartet wird.

timeout: zur Definition des "timeout"-Ereignisses

event: reserviert für geplante Erweiterungen

process: reserviert für geplante Erweiterungen

channel:

Datentyp, unterstützt die Kommunikation zweier Prozesse. Es können Nachrichten beliebiger Länge verschickt werden. Die aktuelle Länge wird vom Compiler ähnlich wie bei sizeof() zur Übersetzungszeit ermittelt.

timer:

Datentyp, erlaubt den Zugriff auf den Hardware-Timer des Transputers. Damit ermöglicht man die Vereinbarung mehrerer Timer zur Prozeßsynchronisation oder für statistische Aussagen über die Performance von Prozessen und Prozessoren.

par:

Kontrollstruktur, legt den Beginn eines parallel auszuführenden Programmteils fest. Alle Befehle auf gleicher Stufe werden nun zu parallelen Prozessen. Diese Prozesse können verschiedenen Prozessoren zugewiesen und auch mit Prioritäten versehen werden. Die Synchronisation der Prozesse erfolgt bei der schließenden Klammer des par-Befehls: alle Prozesse müssen ordnungsgemäß beendet sein, damit der Hauptprozeß weiterlaufen kann. Der par-Befehl kann auch geschachtelt oder analog zu einer for-Schleife verwendet werden.

Syntax:

```
"par" [ <replicator> ] "{"
   { <statement> ";" }
"}"
```

Beispiel:

```
par ( p = q; p != NULL; p = p->next ) {
   DoSomethingWith(p);
}
```

select:

Kontrollstruktur, veranlaßt einen Prozeß so lange zu warten, bis ein bestimmtes Ereignis eintritt, wie zum Beispiel das Anliegen von Daten auf einem channel, oder

das Erreichen eines bestimmten Timer-Wertes. Bei Hinzufügen des default- Falles wird nicht gewartet bis irgendein Ereignis eintritt, sondern das Programmstück für den default-Fall aktiviert.

Syntax:

```
"select" [ "within" <expression> ] "{"
  { "alt" [ <replicator> ]
          [ "cond" <expression> ]
          [ "guard" <channelpointer> ]
            ":" <statement> ";"  }
  [ "alt" "timeout" ":" <statement> ";" ]
"}"
```

Beispiel:

```
select within 1000 {
   alt (i=0;i<<N;i++) guard &ChannelArray[i]:
       msg = ChannelArray[i]; break;
   alt guard &AuxChannel:
       msg = AuxChannel; break;
   alt timeout:
       printf("\nfailed to receive msg within 1000 ticks");
}
```

process:

Schlüsselwort, identifiziert eine Funktion, die keine globalen Variablen enthält und Datenaustausch ausschließlich über Kanäle praktiziert. Ein "process" kann deshalb von jedem Transputer des Netzwerkes unter Verwendung von channel aufgerufen werden.

```
xcall (process ProzessName, int TNummer)
```

Diese Funktion gibt einen "channel" zurück, über den der Informationsaustausch stattfinden kann. Ein "process" kann mehr als einmal aufgerufen werden.

Rolf Geisen, Friedrich Lücking

Buchtips

[1]: Brian W. Kernighan, Dennis M. Ritchie: The C Programming Language, Prentice-Hall 1978.

[2]: Samuel P. Harbison, Guy L. Steele, jr.: C: a reference manual, Prentice-Hall 1984.

[3]: Draft Proposed American National Standard for Information Systems - Programming Language C , Accredited Standards Commitee X3, (ANSI) 1986.

```
/**** EXSEL.C: Beispielprogramm zur Demonstration von Par.C ****/

 /*Das Beispiel zeigt die Verwendung  des SELECT und PAR Befehls.
   Weiterhin wird der Gebrauch von clock() und onexit()
   demonstriert. Globale Variable sind ProgName und StartTime.
   Im Fall, daß das Programm unterbrochen wird (BREAK), wird die
   onexit Funktion dennoch aufgerufen. Dies ist eine Eigenschaft
   des Laufzeitsystems, was dafür sorgt, daß beim ersten ^C das
   Programm ordnungsgemäß heruntergefahren wird. Beim zweiten ^C
   wird lediglich das Serverprogramm beendet und das Beenden des
   Transputerprogramms nicht mehr abgewartet.*/

#include <stddef.h>                /* General definitions  */
#include <stdlib.h>                /* For onexit           */
#include <time.h>                  /* For clock()          */

#define MX       10
#define Delay    1000
#define TimeOut  10
#define Extra    5

char    *ProgName;
clock_t StartTime;

SayExit()
{
   printf( "Exit program %s, time spent: %8.3f\n", ProgName,
           (double)((clock()-StartTime)/CLK_TCK) );
}

main(argc,argv)
```

```
char *argv[] ;
{
   int i,j=3,N=MX,Again=1,nC=0;
   int NotYet[MX];               /* Array of booleans */
   channel C[MX];                /* Array of channels */

   for (i=0 ;i<MX ; i++)         /* Set all to TRUE */
      NotYet[i] = TRUE ;

   ProgName = argv[0] ;
   StartTime = clock() ;
   onexit(SayExit) ;

   par
   {{ /* 1st process outer par */

      int i;
      printf("Outputing process started\n") ;
      par (i=0 ; i<MX ;i++)
      {{
         /* double brackets, else all statements parallel */
         int r = i * i ;
         printf("Replicated par %d started\n",i) ;
         wait ( (Delay/MX) * (MX+Extra-i) );
         C[i] = r ;
         printf("Sent %d over %p\n", r, &C[i]) ;
      }}
      printf("End replicated par\n");

   }{ /* 2nd process outer par */

      printf("Inputing process started\n") ;
      while (Again)
         select within TimeOut * (j+1)
         {  /* Delay depends on the last channel selected */
            alt (j=0;j<MX;j++) cond NotYet[j] guard &C[j] :
               printf("Got %d from C[%d] at %x\n",C[j],j,&C[j]);
               NotYet[j] = 0;
               nC++;   /* Number of channels ready */
               break;
            alt timeout :
               printf("Time Out in Select\n");
               break ;
            alt cond (nC >= MX) : /* Just put here for example */
               /* more efficient is to put the inverse */
               /* condition in the while loop header : */
               /* while (nC < MX) */
               printf("Got them all\n");
               Again=0;
         }
      printf("End SELECT loop\n");

   }}

   printf("Exiting %s\n",argv[0]);
}
```

Parallele Programmierung in CS-Prolog

Das Herz eines jeden Computers, aber auch von vielen komplexen Steuer-, Regel- und Anzeigesystemen ist ein Prozessor. Wenn es sich dabei mehr oder weniger nur um eine einzige integrierte Schaltung (IC = Integrated Circuit) handelt, spricht man von einem Mikroprozessor. Dieser führt das vorgegebene Programm aus.

Mit der Zunahme der Aufgaben für solche Prozessoren verbunden war immer schon der Wunsch nach stetig wachsender Leistung und Geschwindingkeit. Lagen frühe Mikroprozessoren noch bei 200 bis 1000 Befehlen pro Millisekunde, findet man heute leicht Vertreter dieser Gattung, die in gleicher Zeit etwa 10 000 bis 20 000 Befehle ausführen können.

Dennoch sind dieser Entwicklung enge Grenzen gesetzt. Die natürliche Wortbreite vieler Rechnungen ist mit 16 oder 32 Bit erreicht, viele Textverarbeitungen begnügen sich mit lediglich 8 Bit (mit denen man leicht den gesamten Buchstaben-, Zahlen- und Zeichenvorrat unserer Schrift darstellen kann). Eine weitere Verbreitung auf mehr als etwa 64 Bit macht aus heutiger Sicht keinen Sinn mehr. Anderseits kann aus physikalischen Gründen die Taktrate der Chips nicht beliebig beschleunigt werden, da die Ausbreitungsgeschwindigkeit der Signale selbst auf den mikroskopisch kleinen Chips eine Rolle zu spielen beginnt.

Eine beliebige Steigerung der Gesamtleitung eines Computersystems ist also mit diesen Methoden nicht zu erreichen. Es verbleibt aber noch die Möglickeit, die auch den Menschen hilft, ihre begrenzte Leistung zu steigern: Teilaufgaben können auf mehrere Mitarbeiter (Prozessoren) verteilt werden, so daß die Gesamtaufgabe in der Zeit gelöst ist, die die langwierigste Teilaufgabe benötigt. Macht man nur die Aufgaben-"Brocken" klein genug und verwendet man ausreichend viele Mikroprozessoren, kann man mit einem solchen "Multiprozessorensystem" auf einfachste Weise praktisch jede beliebig hohe Rechenleistung erzielen.

In der Praxis bestand bisher die Hauptschwierigkeit solcher "parallelverarbeitender" Computer aus der notwendigen Abstimmung der Teilkomponenten untereinander (Kommunikation). Hier verbrauchten die Teil-Prozessoren soviel Zeit, daß sie dafür praktisch die gewonne Zeit durch Parallelverarbeitung wieder opfern mußten. Ein Fünf-Prozessor-System war gerade 30 Prozent schneller als ein einziger Prozessor, der die ganze Aufgabe allein löste.

Hier setzte die Firma Inmos ein, und versah einen ohnehin schon schnellen Mikroprozessor von 32 Bit mit einer Anzahl von privaten Verbindungsleitungen, die nur genutzt werden, um mit jeweils genau einem anderen Prozessorchip zu kommunizieren. Damit entfällt jegliche Wartezeit und beide Chips können unverzüglich ihrer Aufgabe weiter nachgehen. Diese geradezu banale Idee, verbunden mit einem extrem schnellen Speicherbereich von 4 000 Zeichen auf dem gleichen Chip, machte diesen Transputer auf Anhieb zum schnellsten Mikroprozessor seiner Zeit. Die inzwischen realisierten Computersysteme mit vielen tausend dieser Prozessorchips haben bewiesen, daß der Anspruch des Herstellers zu Recht besteht, mit der Anzahl der Transputer direkt die Gesamtleistung des Computersystems steigern zu können.

Es gibt keine Grenze für die Anzahl von Transputern, die man parallel zusammenschalten kann. Die niedrigen Kosten dieses vergleichsweise einfachen, nur 9 mm x 11 mm großen Chips machen schon heute Systeme mit gigantischer Rechenleistung um 10- bis 100mal billiger als hergebrachte Supercomputer. In der Zukunft wird sich das noch weiter ins Positive verschieben. Es zeigt sich aber, daß die Programmierer, die durch ihre Programme den Computern "Leben einhauchen", mit dieser Entwicklung nicht Schritt halten konnten. Für die heute schon herstellbaren Super-Supercomputer mit 10 000 bis 100 000 Transputern fehlt es noch an "Software", die diesem Genie geeignete Lösungen abringen könnte. Es ist aber abzusehen, daß die Informatiker diese Lücke in den nächsten zwei Jahren geschlossen haben werden. Entwicklungen auf dem Gebiet der Künstlichen Intelligenz wie auch der Neuronalen Netzwerke zeigen diese Tendenzen deutlich auf.

Der Entwicklung von Expertensystemen auf Transputer-Hardware wird überhaupt in der Zukunft eine deutlich steigende Bedeutung zukommen.

Experten in der "realen Welt": Expertenteams

Bei größeren Expertensystemen sind die Zusammenhänge gelegentlich so komplex, daß die Antwortzeiten zu wünschen übrig lassen. Dies ist häufig dann der Fall,

wenn das Arbeitsgebiet normalerweise von Expertenteams mit Experten unterschiedlicher Ausrichtung bearbeitet wird. Diese Aufteilung der Aufgaben auf einzelne Experten kann im Prinzip auch in Expertensystemen simuliert werden, oder den Rechnern, die auf einer Parallelarchitektur basieren, also parallel implementiert werden. Analysiert man die Vorgehensweise eines solchen Teams, so fallen folgende Kennzeichen auf: Die Aufgabe wird in einem ersten Schritt aufgeteilt. Das prinzipielle Verfahren dieser Aufteilung erfolgt in der Regel durch Vorschriften, zumindest aber über nachvollziehbare Regeln. Die Experten bearbeiten ihre Teilaufgaben. Dabei ziehen sie Daten, Fakten und andere Bearbeitungsressourcen heran. Immer wiederkehrend sind Gespräche der Experten, in denen sie sich gegenseitig über den Stand ihrer Arbeit informieren und Teilergebnisse austauschen, die zum Teil auch von anderen Experten für ihre weitere Arbeit benötigt werden. Die durchgeführten Arbeitsschritte sind also im einzelnen sequentieller Natur, die aber parallel zueinander abgearbeitet werden.

Die auf Transputer zugeschnittenen Sprachen, zum Beispiel Occam, Parallel C, Parallel Fortran oder CS-Prolog, stellen nun ihrerseits die Sprachkonzepte zur Verfügung, um die wichtigsten Eigenschaften der oben beschriebenen Arbeit von Expertenteams in Expertensysteme zu übertragen. Dies ist zum einen die Möglichkeit, Teilarbeiten zu delegieren (ein Masterprozeß erzeugt Unterprozesse), diese abarbeiten zu lassen (die Unterprozesse werden parallel ausgeführt) und diese Prozesse zu synchronisieren. Aufgrund dieser starken Homologie der Anforderungen auf der Expertenebene und den Möglickeiten der Transputer bezeichnen wir die entsprechende Expertensystemarchitektur (in Anlehnung an Tony Hoares "Communicating Sequential Processes" (CSP) als "Communicating Sequential Expert Systems" (CSES).

Entwicklungsschritte in der Erstellung

Wie sehen nun die Schritte in der Entwicklung solcher "kommunizierender sequentieller Expertensysteme" aus? Im Prinzip muß für jeden (menschlichen) Experten ein (senquentielles) Expertensystem in konventioneller Art erstellt werden. Hierzu stehen mittlerweile eine Reihe von Werkzeugen zur Verfügung, vor allem sind Zeno - als ein (auch parallel abarbeitbares) C-Code generierendes System - und - CS-Prolog zu nennen. Zusätzlich benötigt man ein weiteres sequentielles Expertensystem, das die Verteilung der Aufgaben vornimmt und entweder ebenfalls über das Werkzeug Zeno oder - im Falle der Verwendung von CS-Prolog - in dem System ALL-EX, einem in CS-Prolog implementierten Expert-System-Shell, geschrieben werden kann. Die entsprechenden Kommunikationsmethoden werden von den Entwicklungswerkzeugen Parallel C beziehungsweise CS-Prolog zur Verfügung gestellt.

Parallel C

Parallel C ist ein auf dem Kernighan-Richie-Standard basierender C-Computer für Transputer, der alle Software-Tools integriert, die für paralleles Programmieren erfolgreich sind.

Der C-Computer implementiert die komplette K&R-C-Sprache mit vielen Erweiterungen:

- Multi-Threaded Prozesse, entweder durch Transputer-Kanäle oder durch Semaphoren synchronisiert

- Zugriff auf den Transputer-Kanal I/O und auf Zeitgeberfunktionen

- In-Linie-Transputer-Assemblersprache mit Zugriff auf C-Variablen

- Microsoft-C-kompatible DOS-Schnittstellen-Fuktion (int86, bdos etc.)

Der Linker ist kompatibel mit 3L-Fortran- und Pascal-Compilern für Transputer und Inmos-Compilern, einschließlich der Stand-Alone-Vision "Beta-2" von Occam-2.

Der FILE SERVER lädt ausführbare Datein in Single- oder Multi-Transputer-Netzwerke. Der Server läuft auf dem Host-PC und wickelt den File-I/O für Transputer ab.

Der statische Konfigurator ermöglicht es, unterschiedliche Prozesse parallel auszuführen. Eine von dem Benutzer geschriebene Textdatei, die sogenannte Konfigura-

```
/* upc.c standalone processing task; communicates with driver.c */

#include <chan.h>
#include <ctype.h>

main(argc, argv, envp, in_ports, ins, out_ports, outs)
int argc, ins, outs;
char *argv[], *envp[];
CHAN *in_ports[], *out_ports[];
{
   int c;

   for (;;) {
      chan_in_word(&c, in_ports[0]);
      if (c == -1) break;   /* terminate task */
      chan_out_word( _toupper(c), out_ports[0] );
   }
}
```

Bild 1

Abbildung 1 illustriert ein aus einer einzelnen main-Funktion bestehendes C-Programm, das einen Stream wort-großer Messages abarbeitet. Die Message-Ports werden als Argumente an die main-Funktion übergeben. Dieser Task liest Messages von Port 0 ein und behandelt sie als ASCII-Characterwerte, die in Großbuchstaben umzusetzen sind

```
/* driver.c file I/O for uppercasing example */

#include <chan.h>
#include <stdio.h>

main(argc, argv, envp, in_ports, ins, out_ports, outs)
int argc, ins, outs;
char *argv[], *envp[];
CHAN *in_ports[], *out_ports[];
{
   int c;
   for (;;) {
      c = getchar();
      chan_out_word( c, out_ports[2] );
      if (c == EOF) break;
      chan_in_word( &c, in_ports[2] );
      putchar(c);
   }
}
```

Bild 2

Abbildung 2 zeigt einen korrespondierenden "Treiber"-Task, der zu dem ersten Task parallel läuft und ihn mit Messages versorgt, die durch das Einlesen von Zeichen aus einem File erzeugt werden.

tionsdatei, beschreibt das Zieltransputer-Netzwerk mit Prozessoren, Links und Verbindungen. Diese Datei sagt dem Konfigurator, welche Software- Aufgaben auf welche Transputer übertragen werden. Die Plazierung von C-Aufgaben auf dem physikalischen Transputer kann einfach durch Wechsel der Konfigurationsdatei geändert werden. Rekompilation oder Relinking sind nicht notwendig.

Insbesondere kann eine Multi-Tasking-C-Applikation auf einem Single- Transputer-System entwickelt werden und dann einfach durch Wechsel der Konfigurationsdatei auf einem Multi-Transputer-Netzwerk ablaufen.

Der dynamische Konfigurator ermöglicht darüber hinaus noch mehr Flexibilität. Es können parallele Programme erzeugt werden, die auf jedem Netzwerk von Transputern mit ausreichendem Speicherplatz laufen.

Die Konfiguration des Zielnetzwerkes braucht nicht bestimmt zu werden, wenn das Programm geschrieben wird. Parallel C kann auf ein beliebiges Netzwerk mit Kopien eines vom Benutzer geschriebenen Arbeitsprogrammes übertragen werden. Parallel C beinhaltet die Software, die automatisch Arbeitspakete, die vom Master-Prozeß generiert worden sind, auf freie Prozessoren im Netzwerk ausgelagert und die Resultate zurückgibt.

Multi-threaded Prozesse können geschrieben werden, in denen neue "Execution-Threads" analog zu Modula-2-Prozessen (oder zu Co-Routinen in einigen anderen Sprachen) dynamisch erzeugt werden können. Threads stehen miteinander in Verbindung und werden entweder durch den Transputer-Kanal I/O oder durch ein mittels Sema-

```
processor host
processor root
! wire connects PC to transputer
wire ? host[0] root[0]

!
! Task ports and memory requirements
!
task afserver ins=1 outs=1
task driver ins=3 outs=3
task upc ins=1 outs=1 data=1k

!
! Assign software tasks to physical
! processors
!
place afserver host
place driver root
place upc root

!
! Connections between the tasks.
!
connect ? afserver[0] driver[1]
connect ? driver[1] afserver[0]

connect ? driver[2] upc[0]
connect ? upc[0] driver[2]
```

Bild 3

Das Konfigurationsfile ist einfach und basiert auf Schlüsselworten, ähnelt damit eher der Kommandosprache eines Betriebssystems als einer Programmiersprache. Abbildung 3 sollte die Art der Notation hinreichend verdeutlichen.

phoren geschlossenes Shared Memory synchronisiert. Die Decoder Utility ermöglicht Source-Level-Disassemblierung von Objektdaten, die vom Compiler erzeugt wurden.

CS-Prolog

CS-Prolog ist neues Mitglied in der Familie der Prolog-Sprachen. CS-Prolog steht hierbei für "Communicating Sequential Prolog", ein Prolog-System auf Prozeßbasis also, das darüber hinaus Zeitmodellierung gestattet. Prozesse werden mittels Messages aktiviert und deaktiviert. Damit wird diese Sprache zu einem geeigneten Instrument zur Entwicklung von Applikationen im Bereich der Simulation.

```
#include <chan.h>                     /* required header files */
#include <thread.h>
#include <sema.h>

char buf[1024];
SEMA buf_free;                        /* controls access to buf */
CHAN **in_p, **out_p;                 /* global pointers to port vectors */

main(argc, argv, envp, in_ports, ins, out_ports, outs)
int argc, ins, outs;
char *argv[], *envp[];
CHAN *in_ports[], *out_ports[];
{
   extern void receive();
   int i;
   sema_init( &buf_free, 1 );         /* buffer is initially free */
   in_p = in_ports;                   /* make in_ports[] & out_ports[] */
   out_p = out_ports;                 /* globally available */
   for (i=0; i < ins; i++)            /* one thread per input port */
      thread_create( receive,         /* function */
                     50*sizeof(int),  /* workspace size in bytes */
                     1,               /* 1 arguments */
                     i );             /* tell thread which port */
}

void receive(i)                       /* handle messages from 1 input */
int i;                                /* which input port to service */
{
   int msglen;                        /* each thread has its own msglen */
   for (;;) {                         /* forever... */
      chan_in_word(&msglen, in_p[i]); /* await next message */
      sema_wait(&buf_free);           /* wait till no one else using buf */
         chan_in_message( msglen,     /* read body of message into */
                          &buf[0],    /* the shared global buffer */
                          in_p[i] );  /* from our port */
         chan_out_word(msglen, out_p[0]); /* copy message to out_ports[0] */
         chan_out_message(msglen, &buf[0], out_p[0]);
      sema_signal(&buf_free);         /* let someone else in again */
   }
}
```

Bild 4

Threads kommunizieren über Transputerkanäle unter Verwendung der normalen Message-Passing-Funktionen. Darüber hinaus können sie natürlich auch über die gemeinsamen Datenbereiche kommunizieren. Abbildung 4 verdeutlicht diese Zusammenhänge.

Zeit spielt bei Simulationen von Vorgängen eine wichtige Rolle. Mit CS-Prolog ist es zum ersten Mal möglich, Zeit an sich explizit in einem Prolog-Programm zu verwenden. So kann man in CS-Prolog zum Beispiel ausdrücken, daß ein Vorgang eine bestimmte Zeit andauert, eine Aufgabe bis zu einem bestimmten Zeitpunkt erledigt sein muß oder die Auswertung eines Prädikates zu einem bestimmten Zeitpunkt erfolgen soll. Das Zeitkonzept von CS-Prolog ist abstrakt, das heißt Zeit in CS-Prolog läuft nicht

```
root :-
   run(problem).

problem:-
   new(dick_gets_the_money(prolog_savings),dick,0,25),
   new(jim_gets_the_money(prolog_savings),jim,0,25).

jim_gets_the_money(BANK):-
   jim_climbs_into(BANK),  chooses(SAFE),  send(safe(SAFE),[dick]),
   wait_for(tools(TOOLS)),  opens(SAFE,TOOLS),  outputs(SAFE,BANK).

dick_gets_the_money(BANK):-
   wait_for(safe(SAFE)),  has(TOOLS,SAFE),  send(tools(TOOLS),[jim]).

chooses(wertheim).
chooses(milner).
chooses(chatwood).

has(tool_set_a,milner).
has(tool_set_b,chatwood).

  m_climbs_into(BANK):-
   during(5).

opens(milner,TOOLS):-
   during(40).
opens(chatwood,TOOLS):-
   during(10).

outputs(SAFE,BANK):-
   systemtime(T),  write("the bandits got the money from the safe "),
   write(SAFE), write(" in the "),  write(BANK),
   write(" bank at time "),  write(T),  write(.),  nl.

during(T):-
   hold(T).
```

Bild 5

Das Konzept der "virtuellen Zeit" in der Praxis: das Bankräuberspiel in CS-Prolog. Die Bankräuber Jim und Dick repräsentieren jeweils eigenständige, miteinander kommunizierende Prozesse.

```
(rule_1:if life_possible(PLANET) and within_reach(PLANET)
      then interesting_planet(PLANET))
(rule_2:if carbon_can_be_found(PLANET)
       then life_possible(PLANET))
(rule_3:if silicon_can_be_found(PLANET)
       then life_possible(PLANET))
(rule_4:if distance(PLANET,DISTANCE)
       then within_reach(PLANET))
(carbon_can_be_found(alpha_127))
(silicon_can_be_found(beta_256))
(distance(alpha_127,10000))
(distance(beta_256,5000))
(goal=interesting_planet(PLANET))
```

Bild 6

Die Wissensbasis eines kommunizierenden, sequentiellen Expertensystems.

gleich unserer Uhrzeit, noch hängt es direkt mit der Rechenzeit des Computers zusammen.

Darüber hinaus ist CS-Prolog in der Lage, mit Wissensbasen, die auf der Basis des zugehörigen Shells ALL-EX erstellt wurden, zu kommunizieren. In Kombination mit CS-Prolog stellt ALL-EX die geeignete Umgebung zur Entwicklung kommunizierender sequentieller Expertensysteme bereit.

Diese Systeme bestehen aus zwei Teilen: der Wissensbasis und der Inferenzmaschine. Die Wissensbasis wird auf der Grundlage des Wissens eines Experten vom Wissensingenieur erstellt. Dieses Wissen wird in Fakten und Regeln abgebildet (z.B. Initialdata, Goal, Question, askable, Nocache etc.). Wir nennen Fakten, Regeln und Metafakten Wissensbasiseinträge.

Die Wissensbasis wird entsprechend einer gegebenen Syntax, die Wissensrepräsentationssprache genannt wird, konstruiert. Die Wissensbasis kann entweder in der ALL-EX-Umgebung oder mit einem Standardeditor erzeugt werden.

Die Inferenzmaschine

Die Inferenzmaschine benutzt die Wissensbasis, um über ein bestimmtes Problem zu folgern und Schlüsse zu ziehen oder dem Benutzer Hinweise zu geben. Der Folgerungsprozeß bedeutet die Ausführung eines Prolog-Programmes, das die Wissensbasis benutzt, aber mit dem Benutzer in interaktiver Verbindung bleibt.

Das vollständige und konsistente Expertensystem kann dem Endbenutzer gegeben werden, das heißt einem Benutzer ohne Programmiererfahrung und ohne Fachkenntnis der bearbeiteten Domäne; der Benutzer muß nur in der Lage sein, dem System richtige Antworten zu geben.

Joachim Stender

Die Programmierung des Transputers

Sprachen wie C, Fortran und Pascal sind sehr populär und es existiert im allgemeinen ein großer Bestand an Programmen, die in diesen Sprachen verfaßt sind. Auch kann es wünschenswert sein, den rein sequentiellen Teil einer Transputer-Programmierung in einer Sprache zu formulieren, die dem Problem vielleicht besonders entgegenkommt. Schließlich spielen auch die Erfahrung und die Vorlieben des Anwenders eine besondere Rolle, so daß es genügend Gründe für das Unterstützen anderer Sprachen als nur Occam gibt. Das Toolset von Inmos wurde speziell für diesen Zweck als alternatives Entwicklungssystem zusammengestellt, so daß damit Programm-Module verknüpft oder sogar parallelisiert werden können, die in C, Fortran, Pascal oder Occam geschrieben wurden.

Das Toolset im PC

In diesem Artikel wird besonderer Bezug genommen auf die Version des Toolset, die für die Ausführung in Standard-PCs konfiguriert worden ist (IMS D705). Sinngemäß gilt gleiches aber auch für die anderen zwei Toolset- Implementationen für VAX-Rechner unter dem VMS-Betriebssystem sowie Sun- Workstations unter Unix.

Im Fall des PC als Gastrechner muß eine Transputer-Karte mit einem 32-bit-Chip (entweder IMS T414 oder T800) eingesteckt sein, da der PC lediglich Terminal- und Fileserver-Aufgaben wahrnimmt und die eigentliche Entwicklungssoftware auf dem Transputer selbst abläuft. Die Kommunikation zwischen dem ausführenden Transputer und dem in C geschriebenen Server-Programm auf dem PC geschieht mittels einer Transputer-Link. Alle Nachrichten nehmen die Form einer "Message" ein, die stets vom Transputer ausgelöst wird und nie vom Server.

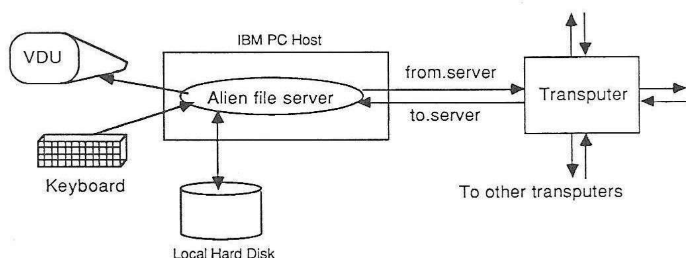


Bild 1: Zusammenspiel von Transputer und Gastrechner.
Im besonderen Fall des Inmos Toolset IMS D705 für den PC als Gastrechner übernimmt der PC die Rolle des Terminals und File Servers, während ein Transputer die Entwicklungssoftware (Compiler, Linker, Lader, Debugger) ausführt.

Es ist interessant zu bemerken, daß dieses Zusammenspiel schon eine echte Form von Parallelverarbeitung darstellt, da ja Transputer und PC zur gleichen Zeit unterschiedliche Dinge zu tun haben und sich durch ein wohldefiniertes Kommunikationsprotokoll verständigen, wie man es oft in parallelverarbeitenden Transputer-Netzwerken vorfindet. Die Sprache Occam hat die Frage der Kommunikations-Protokolle übrigens zu einem Kernpunkt der Sprachdefinition gemacht.

Wie das Bild zeigt, hat der Server Zugriff zum Bildschirm, der Tastatur und dem Filesystem des Gastrechners. Er ist aber auch das "Tor" für den Zugang zu angeschlossenen Transputer-Netzwerken, die untereinander mittels Links verbunden sind. Manchmal wird der Server auch als "AFServer" apostrophiert, was auch seine Funktion bei Verwendung von anderen Sprachen als Occam hinweisen soll.

Ein etikettiertes Protokoll wurde für den Nachrichtenaustausch zwischen dem Server und dem Transputer-System eingerichtet. Da alle Kommandos über dieselbe Link übertragen werden, muß dem Server über ein Etikett mitgeteilt werden, welches der rund dreißig Kommandos seines Repertoirs gerade vom Transputersystem verlangt wird und welche Parameter danach noch zu erwarten sind.

Compiler für Transputer - eine Übersicht

Es gibt mittlerweile eine große Anzahl von Compiler-Paketen, die für die Programmierung von Transputern herangezogen werden können. Man findet darunter Sprache wie Modula-2, Ada, Forth, Lisp, Prolog und Basic, aber die mit Abstand populärsten sind mit Sicherheit C, Fortran und Pascal, die aus diesem Grunde in einer speziellen Form angeboten werden, damit sie mit Hilfe des Toolset-Entwicklungspaketes in Zusammenarbeit mit Occam parallelisiert werden können. Diese Com-

piler können von den 32-bit-Transputern T414 oder T800 ausgeführt werden und jeweils auch ausführbaren Code für beide Prozessoren erzeugen.

Das Inmos-Produkt IMS D711C ist ein C-Compiler, der den vollen Kernighan- Ritchie-Befehlssatz umfaßt und zudem einige Erweiterungen aufweist. Mit dem IMS D712C steht ein Pascal-Compiler mit Erweiterungen zur Verfügung, der dem ISO-Standard 7185 entspricht. Das IMS D713C ist ein voller Fortran-77-Compiler, der wiederum über Erweiterungen verfügt. Diese drei Compiler besitzen einige gemeinsame Eigenschaften:

- Sie unterstützen alle T4- und T8-Transputer.

- Derselbe Linker und derselbe Loader kann für alle drei Sprachen verwendet werden.

- Sie können über dasselbe Software-Interface von Occam aus aufgerufen werden.

- Das Onchip-RAM des Transputers kann als Stack verwendet werden.

- Separat compilierte Module sind erlaubt.

- Es gibt zwei Versionen der Runtine-Library: eine mit vollem Host-Zugriff und eine für Standalone-Betrieb.

Entwicklung für nur einen Transputer

Die Entwicklungshilfsmittel des Toolset sind fast immer erforderlich, wenn Programme für mehrere Transputer entwickelt werden sollen oder wenn mehrere Programm-Module in verschiedenen Sprachen geschrieben wurden. Es gibt dennoch einen Sonderfall, bei dem nur der Compiler und der Linker erforderlich sind und sonst keine weiteren Tools.

Besteht das Anwenderprogramm aus lediglich einem sequentiellen Teil, der in C, Fortran oder Pascal geschrieben wurde, so kann es durch Eingabe von nur drei Kommandos übersetzt, gelinkt und gleich zum Ablauf auf den Transputer geladen werden. Im Fall der Sprache C würden die Kommandos lauten:

t4c prog.c -- ruft den Compiler auf für das C-Programm "prog.c"

t4clink prog -- linkt das Binärfile mit der Laufzeit-Bibliothek

run prob.b4 -- lädt das ausführbare Programm auf die Transputer-Karte

Entsprechende Kommandos existieren für Pascal und Fortran auch. Es ist wichtig, sich zu vergegenwärtigen, daß derart einfache Programme nur einfache sequentielle Abläufe auf dem Transputer auslösen können und weder Gebrauch machen von der Parallelfähigkeit des Transputers noch von seinen Kommunikations-Links.

Hilfsmittel im Toolset

Das Toolset wurde in erster Linie entwickelt, um das Verbinden von sequentiellen Sprachen (C, Fortran, Pascal) mit Occam zu ermöglichen, wodurch diese von den parallelen Sprachelementen von Occam profitieren können. Das File-Format entspricht dem des Gastrechners, so daß der Anwender in der ihm vertrauten Umgebung seines Rechners verbleibt und nicht ein neues Systemumfeld erlernen muß. Das Toolset besteht im wesentlichen aus einem Occam-2-Compiler, einer Utility zur Erkennung von File- Abhängigkeiten, einem Linker, dem Configurer, einem Loader und File-Server sowie verschiedenen unterstützenden Bibliotheken.

Die Occam-Teile einer Entwicklung und die Module, die in anderen Sprachen zu schreiben sind, werden grundsätzlich getrennt voneinander entwickelt. Erst der Linker verbindet die Binärfiles miteinander, wonach sie durch den Configurer für das Laden eines Transputer-Netzwerkes vorbereitet werden. Dabei wird dem eigentlichen Programm-Code eine Präambel mit organisatorischem Code vorangestellt, der das Laden beliebig vieler Transputer mittels ihrer Link- Verbindungen bewerkstelligt.

Das Serverprogramm, das das Bindeglied zum Gastrechner darstellt, liegt im vollen Source-Format vor und kann vom Anwender den jeweiligen Anforderungen angepaßt werden. Daher liegt auch kein Copyright auf diesem Teil des Toolset.

Der allgemeine Fall: C, Fortran und Pascal im parallelen Programm

Sobald Programmmodule in einem parallelen Occam-Programm auftauchen, die in den sogenannten "Alien"-Sprachen C, Fortran oder Pascal geschrieben wurden, erscheinen sie als Occam-Prozesse mit einem definierten Interface. Für die Kommunikation mit anderen Prozessen oder Prozessoren bedienen sie sich der Occam-Kanäle, die vom Occam-Rahmenprogramm ("Harness") deklariert und an die jeweiligen Sprach-Module übergeben werden.

Ein Code-Modul, das eine volle Runtime-Library mitbekommt, benötigt zwei Kanalpaare für die Benutzung durch die Bibliothek. Die notwendige Konzentration auf nur ein Kanalpaar, das den Zugriff auf den Hostrechner übernimmt, bewirkt ein mitgelieferter Multiplexer-Prozeß ("screen.handler"), der parallel mit dem Anwenderprogramm mitläuft. Er ist nur einmal notwendig und nur dort, wo der Zugriff auf den Server des Gastrechners stattfindet.

Wenn Programme nicht auf den Filer und die I/O-Fähigkeiten des Servers zurückgreifen (das ist insbesondere für diejenigen Transputer eines Netzwerkes zutreffend, die nicht direkt mit dem Hostrechner verbunden sind),

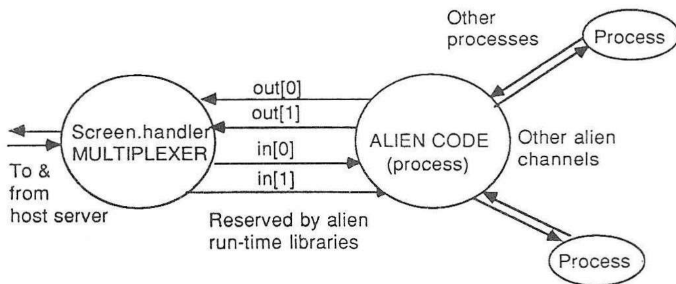


Bild 2: Das Umfeld eines parallelen "Alien"-Moduls. Ein Programm, das in C, Fortran oder Pascal geschrieben wurde und über die volle Laufzeit-Bibliothek der jeweiligen Sprache verfügt, benötigt einen Multiplexer-Prozeß zur Synchronisierung der Gastrechner-Zugriffe. Andere Prozesse, die alleinstehend sind, können unbelastet arbeiten und miteinander über die Occam-Kanäle (Channels) kommunizieren.

können sie mit einer kleineren Version der Laufzeit-Bibliothek gelinkt werden, die den Vorteil hat, daß sie weniger Speicherplatz benötigt. Dann können solche Funktionen wie printf(), fopen() oder fprintf() natürlich nicht aufgerufen werden.

Kommunikation in C, Fortran und Pascal

Damit in den anderen Sprachen das Kanalkonzept wie in Occam zugänglich wird, finden sich jeweils vier Erweiterungen in den Laufzeitbibliotheken vor, die den sequentiellen Sprachen den Aufruf der Kommunikation über Kanäle oder Links gestatten.

1) C-Kommunikation

In der Sprache C lauten die Kanalaufrufe wie folgt:

```
_outword (w, chanp)
int w, *chanp;

_outbyte (n, chanp)
char b;
int *chanp;

_inmess (chanp, buf, nbytes)
int *chanp, nbytes;
char buf[];

_outmess (chanp, buf, nbytes)
int *chanp, nbytes;
char buf[];
```

Im C-Hauptprogramm finden sich die folgenden Argumente:

```
typedef int CHAN;
```

```
main (argc, argv, envp, in, inlen, out,
outlen)
int argc, inlen, outlen;
char *argv[], *envp[];
CHAN *in[], *out[];
```

Die Primitive der Kanalkommunikation werden durch Einschließen der Header-File zugänglich gemacht:

```
#include hanio.h
```

Die folgenden Beispiele setzen voraus, daß die Kommunikations-Routinen innerhalb des "main()" aufgerufen werden, weil andernfalls die Vektoren "in" und "out" nicht gültig sind.

- Empfange einen BYTE-Wert auf Kanal 3 und speichere ihn als eine Integer- Zahl

```
int tag = 0; _inmess (in[3], &tag, 1);
```

- Empfange einen Integerwert aus 4 BYTE und zeige ihn an

```
int value; _inmess (in[2], &value, 4);
printf ("%d\n", value);
```

- Empfange ein Doppelwort auf Kanal 4 und reiche es auf Kanal 3 weiter

```
double item;
_inmess (in[4],    &item, 8);
_outmess (out[3], &item, 8);
```

- Sende ein BYTE #02 auf Kanal 4, dann die Integer 3

```
_outbyte (2, out[4]);
_outbyte (3, out[4]);
```

2) Fortran-Kommunikation

In Fortran lauten die vier Kommunikationsaufrufe wie folgt:

SUBROUTINE CHANOUTMESSAGE (ICHANNEL, BUFFER, NBYTES)

SUBROUTINE CHANINMESSAGE (ICHANNEL, BUFFER, NBYTES)

SUBROUTINE CHANOUTBYTE (VALUE, ICHANNEL)

SUBROUTINE CHANOUTWORD (VALUE, ICHANNEL)

Weitere Information muß nicht spezifiziert werden, da der Linker sie der Laufzeit-Bibliothek entnimmt. Hier nun einige Beispiele der Kommunikation, wie sie in Fortran-Programmen aufgerufen werden kann:

- Sende eine REAL-Zahl über Kanal 2

```
      REAL*4 A
C     A ist 4 BYTE lang
      CALL CHANOUTMESSAGE (2, A, 4)
```

- Empfange eine Integer-Zahl von 4 BYTE Länge

```
      INTEGER*4 B
C     B ist 4 BYTE lang
      CALL CHANINMESSAGE (2, B, 4)
```

- Empfange ein BYTE von Kanal 2 und bringe es nach "TAG"

```
       INTEGER TAG
       CALL CHANINMESSAGE (2, TAG, 1)
```

- Sende ein BYTE #01 und dann die Wortvariable "VALUE"

```
       INTEGER VALUE
       VALUE = 1
       CALL CHANOUTBYTE (1, 2)
       CALL CHANOUTWORD (VALUE, 2)
```

3) Pascal-Kommunikation

In Pascal lauten die Kommunikationsroutinen:

```
PROCEDURE outword (w, channel:INTEGER);
PROCEDURE outbyte (b: CHAR;
channel:INTEGER);
PROCEDURE inmess (channel:INTEGER;
                  VAR bufp: UNIV CHAR;
                  nbytes:INTEGER);
PROCEDURE outmess (channel:INTEGER;
                   VAR bufp: UNIV CHAR;
                   nbytes:INTEGER);
```

Folgender Aufruf schließt die Bibliothek in das Programm ein:

```
$include '\tp1v2\channels.inc'
```

Hier nun noch einige Pascal-Beispiele für die Verwendung der Kommunikations- Befehle:

- Empfange ein BYTE in die Zelle "tag" über Kanal 2

```
inmess (2, tag, 1)
```

- Empfange eine Integer in die Zelle "data" über Kanal 3

```
inmess (3, data, 4)
```

- Gebe eine Integer von "count" auf Kanal 2 aus

```
outword (count, 2)
```

- Gebe den BYTE-Wert #05 auf Kanal 3 aus

```
outbyte (chr(5), 3)
```

Das Occam-Rahmenprogramm

Nicht in Occam geschriebene Programme, die mit anderen Prozessen oder Prozessoren kommunizieren oder einfach parallelisiert werden sollen, benötigen ein minimales Rahmenwerk in Occam, das eine Anzahl von wichtigen Aufgaben übernimmt. Dieser "harness" wird beispielhaft dem Toolset mitgegeben und bewirkt folgende Funktionen:

- Ruft die benutzten Fremdsprachenprogramme auf

- Weist den Nicht-Occam-Modulen Speicherplatz zu

- Initialisiert die Vektoren der Occam-Kanälen

- Ordnet den Fremdsprachenprogrammen evtl. Hilfsmodule zu

- Beendet das PC-Serverprogramm nach Beendigung des Anwenderprogramms

Ein typischer Rahmenprozeß besteht aus 20-30 Zeilen Occam und kann dem Umfang des Toolset unverändert entnommen werden. Es besteht hier aber die Möglichkeit, durch Modifikationen die Funktionalität des "harness" über das Maß des Notwendigen hinaus zu erweitern und besondere Features einzubauen, die zum Beispiel im Bereich der Ausfallsicherheit, Fehlerunterdrükkung und Redundanz verstärkte Sicherheitsaspekte unterstützen. Auch ist es denkbar, daß hier die Reihenfolge der Bearbeitung von Programm-Modulen festgelegt wird, wenn sie nicht alle zur gleichzeitigen (parallelen) Verarbeitung bestimmt werden.

Freie Wahl der Programmiersprache

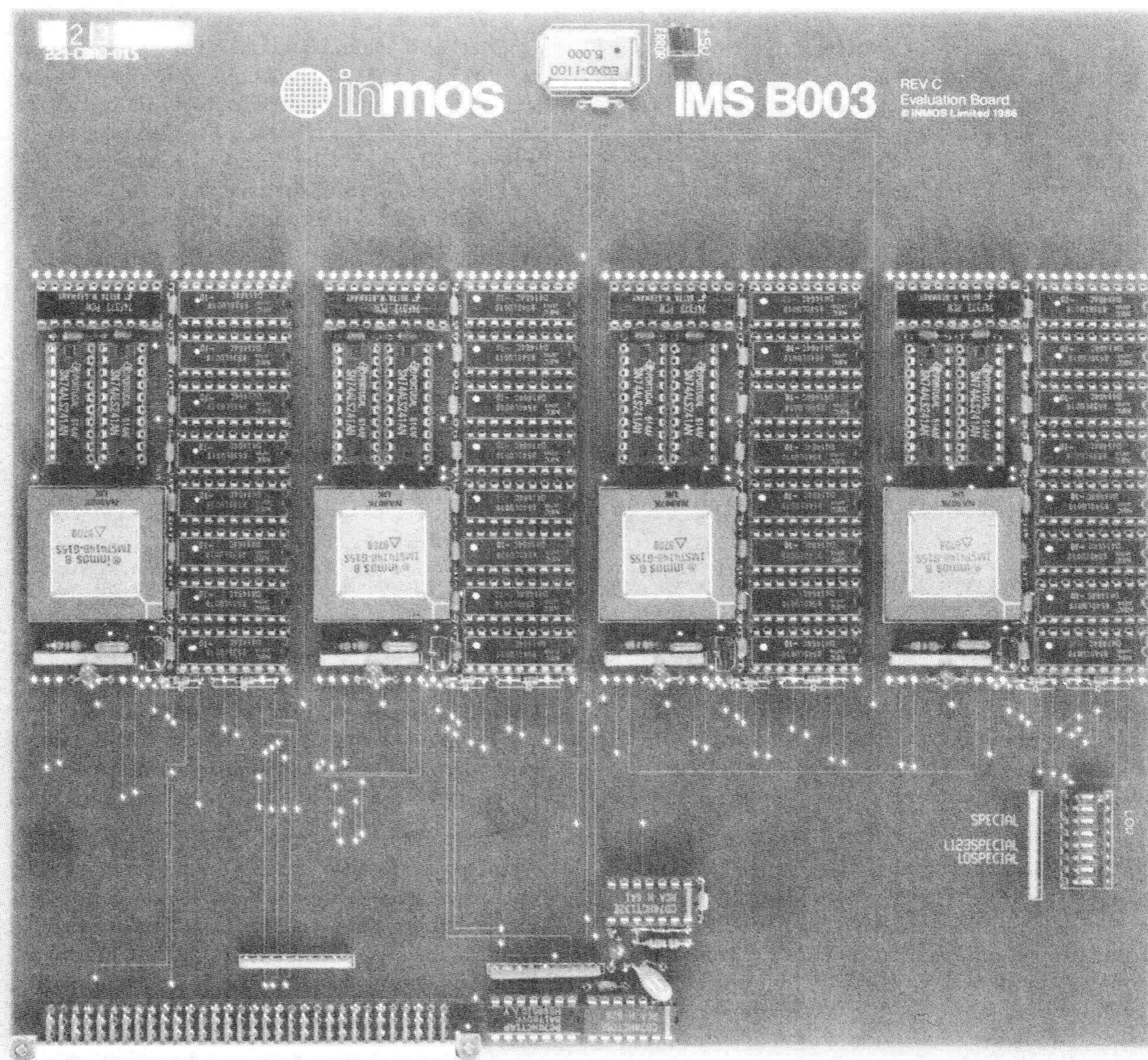
Mit sehr wenigen Hilfsmitteln ist es durch das Inmos-Toolset möglich, die eigentlich rein sequentiellen Sprachen C, Fortran und Pascal zur Programmierung parallelverarbeitender Aufgaben heranzuziehen. Dabei lassen sich natürlich nur die rein sequentiellen Teile eines Programms in diesen Sprachen formulieren, so daß für die parallele Anordnung von Programm-Modulen ein minimaler Rahmen von Occam-Aufrufen notwendig ist. Dieses Verfahren besitzt jedoch den großen Vorteil, daß innerhalb der Module die ursprüngliche Sprache ohne Veränderungen angewendet werden kann. Die Übernahme existierender Programmteile wird dadurch besonders vereinfacht.

Wo Ein- oder Ausgabe benötigt wird, ist dies durch "organisch" eingefügte Kommunikationsbefehle (nämlich durch

die Erweiterung der Laufzeitbibliothek) möglich. Dies geschieht an Stellen des Programms, wo wegen der peripheren Abhängigkeiten der Software von der ausführenden Hardware ohnehin eine Anpassung vorzunehmen wäre.

Aufgrund dieser transparenten Maßnahmen zur Parallelisierung und Kommunikation, die zudem für die Sprachen C, Fortran und Pascal gleichartig gewählt wurden, ist es nun in einfacher Weise möglich, Programmelemente in unterschiedlichen Sprachen zu schreiben und völlig willkürlich (d.h. der Aufgabenstellung angepaßt) zur gleichzeitigen, parallelen oder zur aufeinanderfolgenden, sequentiellen Abarbeitung zu bestimmen. Zusammenfassend kann man sagen, daß das Toolset dem Anwender die Entscheidung freistellt, auf der Basis jedes neuen Programm-Moduls (Unterprogramm, Prozedur oder Prozeß) die geeignete Sprache jeweils aufs Neue auszuwählen.

Peter Eckelmann



Parallelverarbeitung und Ada

Die Programmiersprache Ada wurde Mitte der siebziger Jahre im Auftrage des amerikanischen Verteidgungsministerium (Department of Defense) als Hochsprache für anspruchsvolle Echtzeit-Anwendungen entwickelt. Diese Applikationen sind zwar alle inhärent parallel, aber die meisten Implementierungen von Ada nutzen nur jeweils einzelne sequentielle Systeme, die Parallelverarbeitung mit Hilfe aufwendiger Multitasking-Software nur simulieren. Im Gegensatz dazu erlaubt eine Transputer-Implementierung wirklich, mit vielen Prozessoren gleichzeitig zu arbeiten und so die Parallelität, die Performance und die Echtzeit- Fähigkeiten zu liefern, die Ada verspricht. Die Implementierung ist integriert in eine professionelle Software-Entwicklungsumgebung, die es ermöglicht, hochwertige Ada-Systeme sowohl zuverlässig als auch schnell zu erstellen.

Die Entwicklung des Transputers, eines hochintegrierten Chips, der Verarbeitung, Speicher und Kommunikation in sich vereinigt, hat es ermöglicht, Anwendungen in einem so hohen Grad durch Parallelverarbeitung auszuführen, wie es vorher nicht möglich war.

Denkbare Anwendungen schließen die Supercomputer mit Gigaflop-Leistung, Ingenieur-Workstations und eine breite Palette von Embedded Systems von der Avionik über die Robotik bis hin zur Spracherkennung ein.

Das Ada-Entwicklungssystem

Das Ada-Entwicklungssystem für den Transputer, derzeit noch in der Entwicklung, verbindet die Vorteile einer Hochsprachen-Standard- Programmiersprache mit der Fähigkeit, High-Performance-Systeme zu konstruieren. Nach einer Einführung in die Programmiersprache Ada und den Transputer wird der Artikel die wichtigsten Ziele aufzeigen, die hinter dem Design eines Transputersystems unter Ada stehen.

Nachdem Ada Mitte der siebziger Jahre als Folge der sogenannten "Softwarekrise" entstand, weil keine der anderen 23 gängigen Programmierspachen die Randbedingen des Department of Defense, wie etwa Echtzeitfähigkeit und Grundkonzepte von Pascal, PL1 oder Algo 68 erfüllen konnte, war es lange ruhig geworden um das einstige "Wunderkind", das in seiner Konzeption allerdings wenig Revolutionäres gegenüber den anderen Programmiersprachen bietet. Auch die ANSI(American National Standardisation Institute)-Normierung im Jahre 1983 brachte keinen Durchbruch. Mittlerweile hat sich aber Entscheidendes getan. Zum einen ist Ada seit März 1987 auch ISO- Standard, zum anderen führt das amerikanische Verteidgungsministerium in Zukunft 60 bis 80 Prozent aller neuen Projekte nur noch mit Ada durch. Im Gegensatz zu anderen Programmiersprachen erfolgt nämlich bei Ada parallel zur Sprachentwicklung die Validierung der Compiler. Das hat nicht nur zur Folge, daß es aufgrund fehlender Dialekte kein "Sprachwirrwar" gibt, sondern daß die Sprache um Klassen stabiler ist als alle ihre Vorgänger. Im übrigen liegt der Prüfstandard für Validierungen unglaublich hoch. Bevor die Compiler zertifiziert werden, müssen sie knapp 4000 Testprogramme bestehen. Nach der ersten Freigabe im Jahre 1983 gibt es nun mittlerweile 127 weitere Ada- Compiler sowie weitere 67 Derived Compiler (Stand August 1988) und bis zum Ende der in Kürze ablaufenden Valierungsfrist kann noch einmal mit rund 30 weiteren Compilern gerechnet werden.

Adas Stärken

Ada ist denn für die Programmierung geeigneter, je größer die Projekte sind. Der wesentliche Vorteil der Hochsprache gegenüber den klassischen, etablierten Sprachen liegt nämlich in ihrer Designphase. Sie dauert zwar erheblich länger als bei den "konventionellen" Sprachen, wird jedoch durch die erheblich kürzere Codierungs- und Implementierungsphase mehr als wett gemacht. Zudem verkürzen sich die Wartungsaufwendungen bei Ada- Implementierungen erheblich. Ada hat nämlich bereits eine ganze Reihe von Software-Engineering-Prinzipien in der Sprache implementiert. Deshalb werden Fehlerquellen in der Regel schon während der Übersetzungszeit aufgedeckt. Das wiederum führt zu Produktivitätssteigerungen der Programmierer. Eine Reihe von CASE-Tools die angekündigt sind beziehungsweise kurz bevorstehen, werden für einen weiteren Schub für diese Programmiersprache sorgen.

Nun zum Transputer. Im Zusammenhang mit Ada wird auf diesen VLSI-Chip mit Rücksicht auf seine ausführliche Darstellung am Anfang des Heftes nur kurz eingegangen. Ein Transputer kann grundsätzlich sowohl als Einzel-Prozessorsystem als auch in Netzwerken als Multi-Prozessorsystem zur Bildung von High-Performance-Systemen eingesetzt werden. Ein Netzwerk von Transputern und Peripherie-Controllern kann leicht mit Punkt-zu-Punkt Kommunikations-Links realisiert werden. Der sichtbare Nutzen dieser einfachen Punkt-zu-Punkt- Verbindungen liegt im Entstehen einer offenen, erweiterbaren Systemar-

chitektur, die selbst Systeme von 1000 und mehr Transputern erlauben.

Ein Transputer kann generell als General-Purpose-Prozessor eingesetzt werden oder aber programmiert werden, um eine spezielle Aufgabe zu erfüllen. So wird eine Embedded-Transputer-Anwendung typischerweise aus einem Netzwerk von Transputern bestehen, die durch Punkt-zu-Punkt-Links verbunden sind, Sie sind jedoch individuell programmiert und übernehmen jeweils eine spezifische Aufgabe innerhalb des Applikation.

Jeder Transputer innerhalb des Systems besitzt seinen eigenen lokalen Speicher. Die Gesamtspeicherbandbreite ist proportional der Anzahl der Transputer im System. Daher vermeidet der Transputer die Probleme, die in Verbindung mit großen speichergekoppelten Systemen auftreten, wo zusätzliche Prozessoren sich die Speicherbandbreite teilen müssen, was wiederum zu teuren und komplizierten Memory-Switching-Systemen führt. Die Trennung des Kommunikationsteils vom Speicher-System vereinfacht die Entwicklung von beiden und verhindert somit Leistung-Engpässe.

Der erste Transputer auf dem Markt, der T414, verfügte über einen 10 MIPS 32- bit-Integer-Prozessor, 2 KByte On-Chip RAM, 4 GByte Adressierungsmöglichkeit und vier bidirektionale serielle Links, die eine Gesamt-DMA-Kommunikationsbandbreite von rund 10 MByte pro Sekunde ergaben und das alles realisiert in einem 84-pin-Pin-Grid-Array-Gehäuse. Ein externes Speicher- Interface erweitert die Adressierungsmöglichkeit des Chips und arbeitet mit einer Bandbreite von über 26 MBytes pro Sekunde. Der auf dem Chip befindliche Memory-Controller sorgt für das Timing, die Kontrolle und die DRAM-Refresh- Signale für eine große Auswahl von Mixed-Memory-Systemen.

Der kurze Zeit später vorgestellte T800 enthält eine 64-bit-Floating-Point- Unit (Gleitkomma-Einheit), die einfache und doppelte Gleitkomma-Operationen entsprechend den ANSI-IEEE-754-1985-Bestimmungen für Gleitkomma-Arithmetik durchführt. Sie ist in der Lage, Gleitkomma-Operationen zeitgleich mit dem 32- bit-Integer-Prozessor auszuführen und eine Leistung von 1,5 MFlops aufrechtzuerhalten. Der Chip enthält außerdem 4 KByte RAM, die extern erweitert werden können und ist pinkompatibel zum T414.

Mit dem 16-bit-Transputer T212 und dem 16-bit-Peripherie-Prozessor M212 stehen außerdem innerhalb der Transputer-Familie zwei weitere Chips zur Verfügung. Die Instruktionssätze beider sind kompatibel zu den beiden anderen Prozessoren T414 und T800. Der M212 verfügt über zwei 8-bit-Datenports und kann für die Verbindung zu Winchester- oder Floppy-Disk-Laufwerken konfiguriert werden.

Zusätzliche Flexibilität für Transputersysteme gewährleisten Link-Adaptoren, die zwischen die seriellen Link-Protokolle von Inmos und Byte-Wide-Interfaces geschaltet werden, sowie der C004, einen programmierbaren Link-Switch, also einen Kreuzschienenverteiler für 32 Transputer-Links, der über einen zusätzlichen Konfigurations-Link programmiert wird.

Der DMA-Support für die Hochgeschwindigkeits-Punkt-zu-Punkt-Verbindungen ist fest in der Hardware verankert. Die daraus resultierende niedrige Reaktionszeit hat einen wichtigen Effekt für den Nutzen von Transputern in massiv parallelen Systemen, wo Hunderte von Transputern im Einsatz sind. Solche Rechner können die Systemkosten auf 2000 bis 4000 Dollar pro Megaflop drücken. Gegenüber den traditionellen Vektorprozessoren ergeben sich dadurch neue Preis-/Leistungsdimensionen. Solche auf Transputerbasis arbeitenden Supercomputer sind prädestiniert für Einsatzgebiete wie Wettervorhersagen, Bildanalysen, Bildgenerierung einschließlich Animation, Finite Elemente Analyse (FEM), elektronische und mechanische Designanalyse und Simaulation, Molekular-Modellierungen und Biochemie, Datenbank-Management und Informationsabfragen sowie die Einsatzgebiete innerhalb der Künstlichen Intelligenz.

Einsatzgebiete

Die einfache Technik (Luftkühlung, niedriger Energieverbrauch) dieser Systeme und ihre geringe Größe lassen spezielle Räume zum Aufstellen der Systeme entfallen. Transputer eignen sich nicht zuletzt wegen ihrer Fähigkeit, ein logisches Design direkt auf eine einfache physikalische Konfiguration abzubilden, für Embedded Applications. Die Avionik, die Kommunikation, die Digitale Signalverarbeitung, Flugsimulatoren, die Bildverarbeitung, Prozeßkontrolle, die Robotik, Raumfahrtsysteme, Spracherkennungs- und Waffensysteme sind nur einige dieser als "embedded" bezeichneten industriellen Einsatzgebiete. Ganz besonders nützlich sind sie jedoch in kritischen Applikationen wie beispielsweise der Raumfahrt, wo ein Höchstmaß von Fehlertoleranz durch die gegenseitige Überwachung verschiedener Transputer im System erreicht werden kann. Nach dieser Hintergrundinformationen über Ada und den Transputer nun zur Praxis.

Zuerst zu den Design-Zielen. Ada ist bereits für eine Vielzahl von auf Transputern basierenden Anwendungen als Design-Sprache genutzt worden. In der Vergangenheit haben jedoch fehlende Compilierungssysteme, ein jetzt behobenes Manko, die endgültige Implementierung verhindert. In der Entwicklung einer Implementierung von Ada für den Transputer wurde von dem britischen Unternehmen Alsys Ltd. großer Wert darauf gelegt, sicherzustellen, daß das Entwicklungssystem ein wirkungsvolles Werkzeug sowohl in Bezug auf die Leistungsfähigkeit der Applikation als auch hinsichtlich der Unterstützung für das Projektmanagement ist. Performance wird hierbei definiert als der Zeitraum für Prozessorzyklen. So erreicht zum Beispiel der T800-20 zwanzig Prozessorzyklen pro

Mikrosekunde, während der T800-30 nochmals um 50 Prozent schneller ist. Zu beachten ist, daß die Zykluszeiten sich auf den Gebrauch des On-Chip-Speichers beziehen, zusätzliche Zyklen können erforderlich sein, wenn ein Off-Chip-Speicher gebraucht wird.

Ziele

Das erste Ziel ist die Ausnutzung der individuellen Transputer-Prozessoren für die sequentielle Programmierung. Hier übersetzt der Ada-Compiler direkt in das Instruktionset des Transputers und so ist zum Beispiel eine Dauerleistung von 1,5 MFlops einfach erreichbar. Der Transputer führt die Arbeiten direkt im lokalen Arbeitsbereich aus, wo die Variablen für das gegenwärtige Unterprogramm gespeichert sind, und stellt einen dreistufigen Stack für die Evaluierung der Integer-Ausdrücke (mit den als A, B und C bezeichneten Registern) sowie ein Gleitkomma-Stack (FA, FB und FC) zur Verfügung. Eine Zuweisung in einem einfachen Ada-Fragment

```
X, Y, Z : INTEGER;
. . .
Z := X + Y;
```

wird dann in den folgenden optimalen Code übersetzt, der die Zuweisung in einem mit 20 MHz getakteten Transputer in 600ns (Nanosekunden) wie folgt ausgeführt

```
                (cycles)
ldl X              2
ldl Y              2
add                1
stl Z              1
```

Im Fall der Fließkomma-Verarbeitung werden die Operanden geladen und die Gleitkomma-Ergebnisse über Adressen gespeichert, die auf dem Integer Stack berechnet werden. Boolsche Resultate von FPU-Vergleichen werden zur Integer-CPU übertragen. Das folgende Beispiel veranschaulicht einen Gleitkomma-Vergleich.

```
converged : BOOLEAN;
absolute_error, epsilon : FLOAT;
. . . .
converged := absolute_error  epsilon;
. . . .
```

dies wird übersetzt zu

```
                             (cycles)
ldlp   epsilon                  1
fpldnlsn                        2
ldlp   absolute_error           1
fpldnlsn                        2
fpgt                            4
stl                             1
```

Die Rechnung in der Gleitkomma-Einheit kann mit Operationen der Integer-CPU überlappen. Dies sorgt für signifikante Leistungsverbesserungen, wenn die Gleitkomma-Arrays eingesetzt werden, weil der Compiler den Zeitpunkt zur Durchführung von Adreßkalkulationen für maximale Überlappung wählt. Dies erfordert keine zusätzlichen Instruktionen, weil der Transputer alle notwendigen Synchronisationsschritte automatisch ausführt. Als Beispiel sei dies am folgenden anspruchsvollen Auszug des Livermore Loop-Benchmark gezeigt, der so compiliert werden würde, als ob die Gleitkomma-Einheit nie für eine Adreßkalulation zu warten gehabt hätte, obwohl sie die volle Gleitkomma-Leistung aufrecht erhalten muß.

```
for k in 0..n loop
  x(k) := u(k)  +r*(z(k)  +r*y(k))
      +t*(u(k+3)+r*(u(k+2)+r*u(k+1)))
      +t*(u(k+6)+r*(u(k+5)+r*u(k+4)));
end loop;
```

Die Implementierung nutzt die natürliche Wortlänge der Maschine für den vorgeschriebenen INTEGER-Typ (16 bit auf dem T2, 32 bit auf dem T4 und T8 Transputer). FLOAT ist einheitlich auf dem T4 und T8 als 32 bit implementiert. Für eine größere Flexibilität stehen zudem die Typen SHORT_INTEGER (8 bit), LONG_INTEGER (32 bit auf dem T2 und 64 bit auf dem T4 und T8) sowie LONG_FLOAT (64 bit) zur Verfügung.

Ada unterstützt verschiedene Klassen von Objekten und zusammengesetzten Strukturen, von denen einige einen weiteren Stack und einen dynamischen Heap für ihre Implementierung benötigen. Der Zugriff zu diesen Objekten geschieht über Zeiger, die im lokalen Arbeitsbereich gespeichert sind. Besondere Sorgfalt ist auf einen effizienten Subroutinen-Aufrufmechanismus gelegt worden. Dieser nutzt die eingebauten Aufruf- und Rückkehrinstruktionen des Transputers. Parameter und Ergebnisse werden wo möglich durch den Evaluation Stack übergeben. So dauert beispielsweise die Ausführungszeit für eine typische Funktion mit zwei Integer-Parametern und das rückkehrende Ergebnis 24 Zyklen, was bei einem mit 30 MHz getakteten Transputer weniger als eine Mikrosekunde bedeutet.

Die Behandlung der sogenannten Exceptions (Ausnahmen) hat den Anspruch, daß die am schnellsten mögliche Ausführung unter normalen Umständen erforderlich ist und das Exceptions nur in wirklichen Notfällen stattfinden. Die Overheads, die mit dem Ada-Ausnahmemechanismus gekoppelt sind, kommen nur zum Tragen, wenn die Ausnahmen gefordert werden.

Der Standard-Ada-File-I/O wird vollständig mittels eines Nachrichtenprotokolls über eine Transputer-Link implementiert. Dies ermöglicht dem Anwender ein Höchstmaß an Flexibilität hinsichtlich der Tatsache, wie der Dateispeicher unterstützt werden soll. Zum Beispiel kann die

Link dazu benutzt werden, Zugriff zu einem einfachen File-Server unter MS-DOS oder einer DEC VAX unter VMS zu gestatten. Ebenso kann ein Geräte-I/O durch den Gebrauch eines Ada- Darstellungssatzes integriert werden. Ein solches Fragment könnte wie folgt lauten

```
type STATUS is (OFF, READY, ON);
for STATUS use (OFF=1, READY=2, ON=4);
My_device: STATUS;
for My_device use at 16#7FFFFFD0#;
```

und würde den Zugriff zu einem speichergekoppelten Peripheriebaustein mittels der Variablen My_device gestatten. Ein direkter Zugriff zu den Transputerkanälen kann im übrigen durch das Ada-Package CHANNEL_IO erfolgen. Schließlich gibt es auch noch die Möglichkeit des direkten Zugriffs auf Occam, der Programmiersprache des Transputer.

Der nächste Punkt betrifft die Ausnutzung der Parallelität bei Ada auf einem einzigen Transputer. Das "Gleichzeitigkeitsmodell" des Transputers ist in seinem Mikrocode implementiert und basiert auf einem synchronen Nachrichten- Austausch. Dies hat seine Wurzeln ähnlich wie das Ada-Modell in den Forschungen bei Professor Hoare, auf denen die Gleichzeitigkeit in Occam basiert. Mit etwas über einer halben Mikrosekunde verfügt der Transputer über einen sehr schnellen Kontext-Switch bei einfachen Occam-Prozeßmodellen. Das Ada-Modell, das das Rendezvous und Exception-Handling integriert, wird uneingeschränkt von der Hardware unterstützt und durch eine

```
package monitor is
  task T1 is
    entry EQUIPMENT_INFO ( . . .) ;
  end;
  . . .
end monitor;

package operator is
  task T2 is
    entry CONTROL_INFO ( . . .) ;
  end;
  . . .
end operator;

package display is
  task T3;
  . . .
  task body T3 is
    . . .
    T1.EQUIPMENT_INFO ( . . .)
    . . .
    T2.CONTROL_INFO ( . . .)
    . . .
  end T3;
end display;
```

Laufzeit-Software noch verstärkt. Die daraus folgende Switch-Zeit für die Task ist dadurch erheblich niedriger als bei vergleichbaren Systemen, die mit anderen, konventionellen Prozessoren ausgestattet sind.

Als Beispiel betrachten wir einen einfachen Prozeß-Controller, der in drei Ada-Pakete unterteilt ist, von denen jedes einzelne eine einzelne Task enthält. Eine Task überwacht die Signale der unter seiner Obhut stehenden Geräte, eine zweite erhält die Kontrollinformationen von einem Operator. Die dritte Task sammelt die verarbeiteten Ergebnisse der beiden anderen Tasks und zeigt dies an. Die Kommunikation zwischen den drei Tasks erfolgt über das Ada- Rendezvous.

Die drei Aufgaben T1, T2 und T3 sind auf einem einzelnen Transputer durch das automatische Timesharing implementiert.

Weitere Vorteile

Ein weiterer Vorteil eines Transputer-/Ada-Systems liegt darin, die Leistung mehrerer Transputer für eine Anwendung auszunutzen. Der Transputer ermöglicht mit dem Einsatz einfacher Standardkomponeneten eine hohe, kosteneffektive Performance. Diese Leistung ist nur möglich, weil der Transputer über private Speichersysteme und lokale Punkt-zu-Punkt- Kommunikation verfügt.

Es gilt jedoch zu beachten, daß es zwei klare Performance-Erfordernisse gibt. Die erste hängt mit dem Durchsatz, die zweite mit dem Reaktionsvermögen zusammen. Die Fähigkeit, eine große Anzahl von Transputern zu verbinden, ohne die gefürchteten "Flaschenhälse", also Engpässe in der Hardware zu haben, ermöglicht den Durchsatz. Die Fähigkeit, einem Transputer eine bestimmte Aufgabe zuzuordnen, zum Beispiel die Überwachung eines Sensors, und ihn trotzdem in das Netzwerk zu integrieren, erbringt das erforderliche Reaktionsvermögen.

Im ersten Fall wird Ada genutzt, um einen numerischen Algorithmus in Ada zu programmieren, im zweiten Fall, um die Echtzeit-Geräte-Routine (Device- Handler) zu codieren. Im hier gezeigten Beispiel können drei Pakete auf drei verschiedenen Transputern konfiguriert und plaziert werden. Die Kommunikation zwischen den Tasks wird nun durch den Einsatz des CHANNEL_IO-Pakets verrichtet, die Konfiguration ist durch Occam als eine Network Definition Language definiert, das heißt als Sprache zur Deklaration der Abbildung der Soft- auf die Hardware, aber ansonsten ist der Ada-Text unverändert. Das System kann wie folgt visualisiert werden:

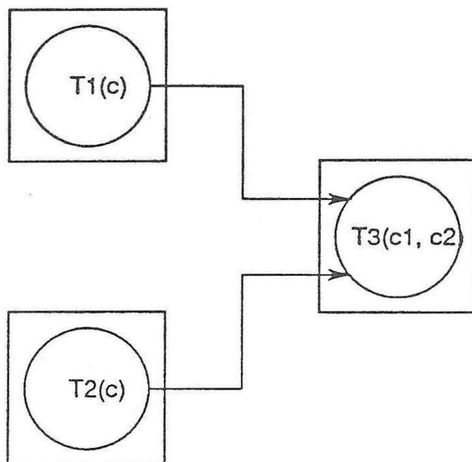


Bild 1: Ada auf drei Transputern

Die Definition der Konfiguration in Occam lautet

```
CHAN OF ANY ca, cb:
PLACED PAR
  PROCESSOR 1
    T1(ca)
  PROCESSOR 2
    T2(cb)
  PROCESSOR 3
    T3(ca, cb)
```

Das Compilierungs-System steht bereits auf einer Reihe von Architekturen zur Verfügung. Dazu zählen unter anderem die Intel 80x86-Familie, Motorolas 680x0-Familie sowie die DEC VAX, die HP1000 und die IBM 370.

Ada-Programme bestehen aus getrennt übersetzbaren Einheiten. Diese Einheiten werden in den Transputer-Maschinencode übersetzt und in einer Ada-Library (Bibliotek) gespeichert, wartend auf ein Ersuchen, um sie mit dem Runtime- System in ein ausführbares Modul zu linken.

Um Integrationsfehler zu verhindern, verlangt Ada die rigorose Prüfung der Software-Schnittstellen zwischen Programm-Modulen. Konsequenterweise verlangt der Compiler Zugriff zu den Einheiten, bevor sie in die Ada-Bibliothek übersetzt werden. Die Ada-Libraries und die enthaltenen Einheiten, können durch den Library Manager und den Unit Manager geprüft werden.

Das Compilierungs-System erlaubt die gleichzeitige Existenz mehrerer Ada- Libraries. Verbindungen zwischen den Einheiten in verschiedenen Bibliotheken können errichtet werden, wobei sie den effizienten Wiedergebrauch des existierenden Codes und die gleichzeitige Entwicklung gestatten, ferner die Kontrolle von Anwendungsvarianten ohne verschwenderische und fehleranfällige Duplizierung von Code. Die Einrichtung erlaubt ebenso die Benutzung von Bibliothekeneinheiten von Programmentwicklern quer durch ein Netzwerk.

Der Compiler besteht aus drei wesentlichen Bestandteilen: dem Analyser, dem Expander und dem Code-Generator. Alle diese Komponenten sind in Ada geschrieben. Analyser und Expander sind dabei Teile der "common root"- Technologie der britischen Firma Alsys Ltd. und deshalb bestehende Komponenten mit einer gut definierten und bewährten Funktionalität. Der Code-Generator hingegen ist eine neue, eigens für den Transputer entwikkelte Komponenete. Er generiert den relokatiblen Objektcode, den er in seiner Programm-Bibliothek speichert. Optional können Optimierungsphasen zwischen diese drei Hauptkomponenten eingefügt werden.

Ein weiteres separates Tool von Alsys ist der Binder. Er baut aus einem Satz compilierter Einheiten ein komplettes Ada-Programm zusammen. Dabei nutzt der Binder die Informationen über Abhängigkeiten ("dependency information"), die sich in der Programm-Bibliothek befinden, um die Einheiten, die zur Erstellung des Anwendungsprogramms benötigt werden zu lokalisieren und zu verbinden. Der Binder bringt den Objektcode in eine für den Inmos-Linker und das Konfigurationssystem geeignete Form.

Der Binder schließt in sein Objekt-File nur diejenigen Unterprogramme ein, die tatsächlich gebraucht werden. Vor allem ermöglicht dies tatsächlich das automatische Selektieren der benötigten Teile durch eine Benutzer- oder Standard-Bibliothek und verhindert beispielsweise solche Teile der Input/Output-Library aufzunehmen, die in einer besonderen Applikation gar nicht benötigt werden. Dadurch kann sich der Umfang des Laufzeitpaketes erheblich reduzieren.

Das Alsys Programmbibliotheks-Managementsystem stellt einen ganzen Satz von flexiblen Einrichtungen zur Verfügung, um bei der Entwicklung von großen, durch Projektteams realisierten Ada-Programmen zu helfen. Gemeinsame Komponenten einer Applikation können so von verschiedenen Entwicklern geteilt werden, ohne daß jeder Entwickler seine eigene Kopie haben muß. Teile von Paketen oder Subprogrammen können vergeben werden, um verschiedene Versionen von Varianten einer Applikation zu erstellen.

Dies erlaubt dem Entwickler einer Komponente eine neue Version zu testen, ohne sofort die gegenwärtig von anderen Anwendern benutzte Version zu beeinflussen. Durch den gleichen Mechanismus ist es möglich, die Entwicklung von Programmen zu kontrollieren, die in weiten Teilen gleich sind, aber die unterschiedliche Zielkonfigurationen unterstützen oder über verschiedene Leistungscharakteristika verfügen.

Es gibt drei unterschiedliche Hierarchiestufen im Library-System. Individuelle Programmeinheiten werden in die

Programmbibliothek übersetzt und die Biblioteken werden in Familien zusammengefaßt. Eine bestimmte Bibliothek gehört zu einer einzigen Familie. Einheiten können zwischen allen Libraries innerhalb der gleichen Familie ausgetauscht werden. Einheiten können nicht zwischen Familien getauscht werden, außer wenn während der Installation des Compilierungs-Systems eine eindeutige Installationsfamilie geschaffen worden ist, die die vordefinierte Ada-Library enthält.

Die Übersetzung einer bestimmten Programmeinheit ist wird im Kontext einer einzigen Programmbibliothek durchgeführt. Einheiten, die nicht explizit in die Programmbibliothek übersetzt werden, können importiert werden, so daß sie während der Compilierung und der Binde-Operationen in der Library im Zugriffsbereich liegen. Das Entwicklungssystem enthält drei interaktive Programm-Bibliotheks-Management-Dienstprogramme, die ein detailliertes Management der Bibliotheken auf jeder der drei Ebenen möglich machen.

Zusammenfassend sei gesagt, daß die Performance von einzelnen Prozessorsystemen sich nun fundamentalen Grenzen nähert, die nicht überschritten werden können. So wird ein Leistungswachstum durch das innovative Konzept des Transputers und die Parallelverarbeitung einerseits sowie die Höchstintegration der Chips andererseits kommerziell attraktiv.

Die Ada-Sprache mit ihrer eingebauten Tasking-Fähigkeit eignet sich ausgezeichnet für den Entwurf und die Programmierung von Multiprozessor- Systemen. Die Kopplung von Ada und dem Transputer stellt einen Meilenstein für eine professionelle Software-Entwicklungsumgebung dar, die die frühe Einführung einer neuen Generation kosteneffektiver und leistungsfähiger Computeranwendungen unterstützen wird.

John Barnes, Colin Whitby-Strevens, Ulrich Parthier

Alternativen zu Occam

Die auf Transputer umgesetzten und mit Libraries zur Implementierung von Parallelität und Link-Kommunikation ausgestatteten Compiler der 'klassischen' Programmiersprachen C, Fortran und Pascal sind eine hervorragende Alternative zu Occam.

Der rapide wachsende Bedarf an hoher und höchster Rechenleistung in einem weiten Spektrum von Anwendungen, das von CAD/CAM über Finite-Elemente-Analyse, Bildverarbeitung und Robotik bis hin zur Künstlichen Intelligenz reicht, kann mit konventionellen Rechnerarchitekturen nicht mehr befriedigt werden.

Der Grund hierfür liegt darin, daß die physikalischen Gesetzmäßigkeiten - wie Elektronenbeweglichkeit, Temperaturentwicklung und Auswirkungen der Heisenbergschen Unschärferelation - weitere Leistungssteigerungen von Mikroprozessoren unverhältnismäßig aufwendig und kostenintensiv werden lassen.

Der vielversprechendste Ansatz, diese strukturelle Schwäche zu überwinden, ist das Konzept der parallelen Datenverarbeitung; hierbei ragen insbesondere die Transputer T414 und T800 (in Kürze zusätzlich T424, T801 und T810) der englischen Inmos Ltd. als leistungsfähigste Vertreter dieses Konzeptes hervor. Wie die extreme Leistungsfähgkeit der Transputer in einem auf PC basierenden Transputersystem eingesetzt werden kann, soll der dieser Artikel zeigen.

Als Hardware-Plattform wurde ein PC/AT-kompatibler Computer (Compaq Deskpro 386/20e) mit dem Multi-Transputer-System Mega-Link01 der Firma Sang- Computersysteme eingesetzt, das kompatibel zum B004-Entwicklungsboard von Inmos ist (Bild 1). Interfaces für Atari ST und Commodore Amiga 2000 - einschließlich Software-Anpassung für die meisten Entwicklungsumgebungen - sind ebenfalls verfügbar.

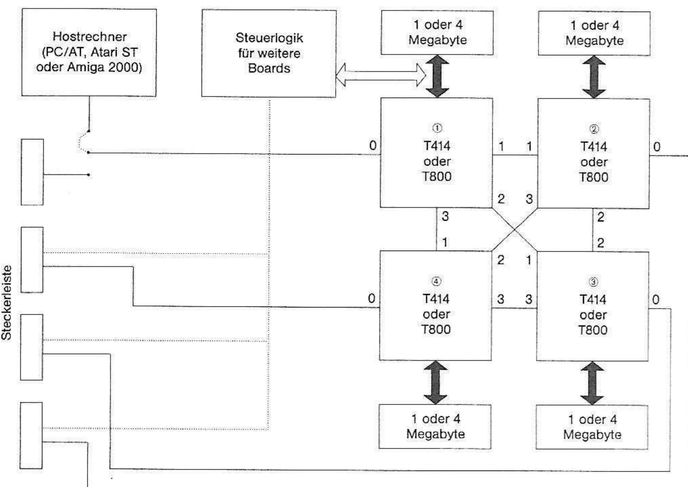


Bild 1: MEGA-Link01 im Vollausbau

Bei diesem System handelt es sich um eine PC-Slotkarte, die - neben einer Interface-Sektion zum PC - über wahlweise 1, 2, 3 oder 4 Transputer-Sektionen verfügt. Die exakten Leistungsdaten der einzelnen Transputer-Sektionen, die beliebig miteinander kombiniert werden können, sind aus Bild 2 ersichtlich. Im Vollausbau bringt es die Mega-Link01 also auf 60 MIps (=Millionen Instruktionen pro Sekunde) und 9 MFlops (Millionen Fließkomma-Operationen pro Sekunde) und insgesamt 16 MByte RAM.

Transputer	RAM	CPU: MIPS	FPU: MFLOPS
T414-20	1 MByte (80 ns)	10	-----
T414-20	4 MByte (120 ns)	10	-----
T800-20	1 MByte (80 ns)	10	1,5
T800-20	4 MByte (120 ns)	10	1,5
T800-30	1 MByte (60 ns)	15	2,25
T800-30	4 MByte (80 ns)	15	2,25

Supertransputer

Die Transputer einer Mega-Link01 sind in der Topologie eines virtuellen 'Supertransputers' miteinander verbunden (Bild 3).

Bei diesem Konzept besitzt jeder Transputer eine direkte Verbindung zu seinen drei Nachbarn. Diese Struktur optimiert die lokale Kommunikationsgeschwindigkeit, die in der überwiegenden Mehrzahl der Applikationen den höchsten Stellenwert besitzt. Außerdem ermöglicht sie es, die verbreitetsten Topologien wie Pipeline, Ring, Ternärbaum oder Hypercube direkt - und ohne Umstecken von Kabeln oder Umprogrammieren von Link-Switches - per Software zu realisieren.

Von jedem Transputer ist ein Link - also insgesamt vier - an einen externen Standardkonnektor geführt, von wo aus die Verbindung mit weiteren Mega-Link01-Systemen erfolgen kann, die bis zu 20 m vom ursprünglichen System entfernt sein können.

Der Vorteil dieser Topologie liegt - neben der schon erwähnten hohen lokalen Kommunikationsgeschwindigkeit,

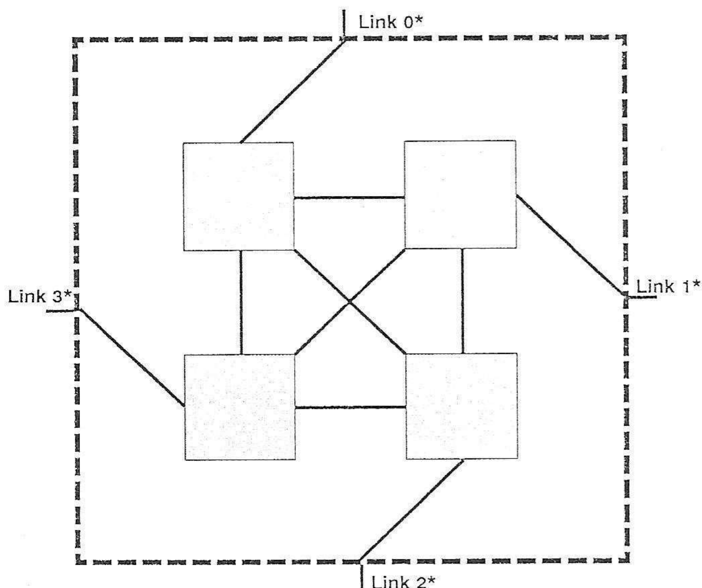


Bild 3: Der virtuelle Supertransputer

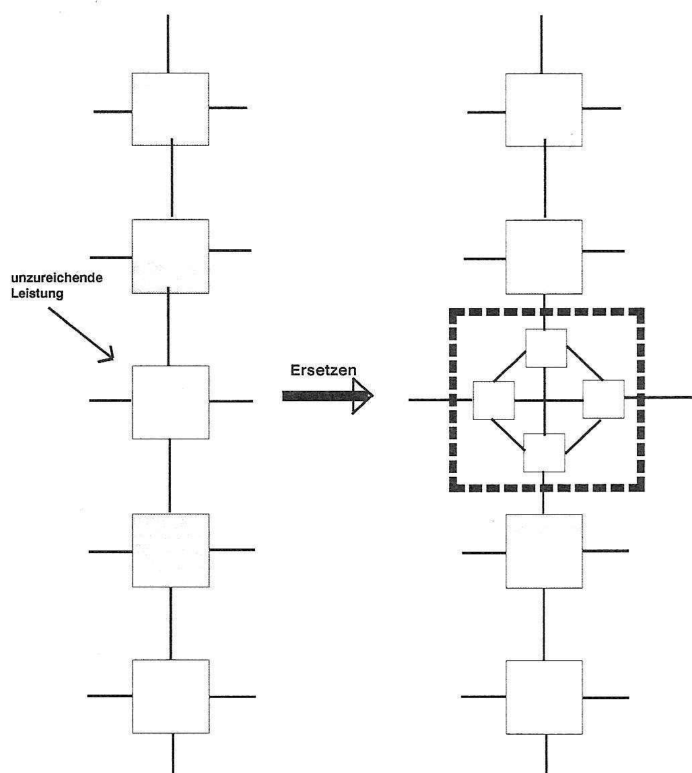


Bild 4: Ersetzen eines Transputers

die 2-3 mal höher ist als beim massiven Einsatz von C004-Link-Switches - darin, daß sie eine 'geballte' Betrachtung der Rechenknoten erlaubt. Dies bedeutet, daß ein System mit vier Transputern sich nach außen hin - bis auf die höhere Leistung - genauso verhält, wie ein System mit einem Transputer. Ein Beispiel soll das veranschaulichen (Bild 4):

In einer Pipeline aus Transputern gibt es ein Rechenelement, das eine zu geringe Rechenleistung besitzt, und somit den Gesamtdurchsatz des Systems merklich verringert. Um diesen Engpaß zu beseitigen, wird es durch einen aus vier Transputern bestehenden 'Supertransputer' ersetzt. Die notwendige Software-Anpassung besteht aus einem Multiplexer-, einem Demultiplexer- und einem 'Flood-Fill'-Prozeß. Mit vergleichsweise geringem Aufwand kann so das schwächste Glied gegen eines mit viermal so hoher Leistung ausgetauscht werden.

Und schließlich ermöglicht das Konzept des 'Supertransputers' durch seine rekursive - also durch einfachere Teile seiner selbst definierte - Struktur den Aufbau von hochgradig konsistenten Transputer-Netzwerken mit theoretisch unbegrenzter Performance.

Der virtuelle Supertransputer in Stichworten:

- geballte Betrachtung: Knoten mit 10 MIPS und 640 MIPS sind im Verhalten nach außen identisch
- hohe Konsistenz und einfache mathematische Beschreibbarkeit
- optimierte lokale Kommunikationsgeschwindigkeit / typischerweise 2-3 mal höher als beim massiven Einsatz von C004-Link-Switches
- homogenes Konzept vereinfacht und beschleunigt 'message-routing'-Algorithmen für Netzwerke
- günstiger Preis, da keine Link-Switches erforderlich

Spezialisierung

Es muß nicht immer die reine Erhöhung der Rechenleistung sein, die über die standardisierten Link-Verbindungen vorgenommen wird. In identischer Weise können beispielsweise Grafik-Systeme, Harddisk- und I/O-Controller oder Systeme mit Signalprozessoren in ein Transputer-Netzwerk integriert werden. Entsprechende Systeme befinden sich in Entwicklung und werden im Laufe des Jahres 1989 verfügbar sein.

Hier ein kleiner Vorgeschmack auf die Leistungsdaten der Systeme, die zur 89er CeBit verfügbar sein werden:

- Mega-Link02-CAD: High-Performance Grafik-System mit T800, eigenem Grafikprozessor, max. Auflösung 1280*960 Bildpunkte in 256 aus 16 Millionen Farben.

- Mega-Link02-Video: Video-Grafik-System mit Videodigitalisierer, max. Auflösung 512*512 Bildpunkte in 16 Millionen Farben, Ansteuerungslogik für Videorecorder.

Aber die beste Hardware nützt wenig ohne Software, die diese Qualitäten auch zugänglich macht.

Für die genannten Transputer-Boards besteht ein weites Spektrum von Entwicklungsumgebungen, das vom Transputer-Development-System TDS, dem Occam-Compiler OCS und dem Toolset über parallele C-, Fortran- und Pascal-Compiler verschiedener Hersteller zum Betriebssystem Helios von Perihelion reicht.

Wie sieht nun die typische Entwicklung eines Programms aus? Im folgenden soll dies an einem Programm veranschaulicht werden, das ursprünglich für einen PC/AT-kompatiblen Rechner unter Turbo-C vorlag, und nun schrittweise auf eine Mega-Link01 portiert werden soll, die mit vier T800 (20 MHz Taktfrequenz) mit jeweils einem Megabyte 80 ns-DRAM bestückt ist.

Strandspiele

Der 'Beach-Ball' (Listing 1) ist ein Programm, das ursprünglich für die Demonstration der Leistungsfähigkeit der mathematischen Coprozessoren Intel 80387 und Weitek 3167 geschrieben wurde. Es eignet sich aber ebenso hervorragend, um die Leistungsfähigkeit von Transputersystemen zu demonstrieren.

Das Programm nutzt die Phong-Schattiertechnik, um die Intensität und die Farben der Oberfläche des Balls zu berechnen. Die Idee hinter dieser Technik ist, die Normalenvektoren zu jedem Punkt auf der Oberfläche mittels eines Interpolations-Schemas zu berechnen, dann das Schattierungsmodell auf jeden angezeigten Bildpunkt anzuwenden und schließlich auf einer EGA-Karte darzustellen.

Das Programm beinhaltet eine große Anzahl von Fließkomma-Berechnungen auf einfach genaue (= REAL32) Zahlen, sowohl die vier Grundrechenarten (Addition, Subtraktion, Multiplikation und Division) als auch spezialisierte Funktionen (Sinus, Cosinus, Wurzel).

Auf dem Compaq Deskpro 386/20e ohne Coprozessor benötigt das Programm 8 Minuten Laufzeit, mit 80387 sinkt dieser Wert auf 28 Sekunden, mit Weitek 3167 auf 10 Sekunden.

Dieses Programm soll nun mit Hilfe des Parallel C-Compilers von 3L auf die Mega-Link01 portiert werden. Der erste Schritt ist die Portierung des Programms auf einen einzelnen Transputer. Da C eine äußerst portable Sprache ist, stellt dies erwartungsgemäß kein Problem dar. Es müssen lediglich einige weitere 'include'-Files eingebunden werden; desweiteren müssen die Funktionen zum Ansteuern der EGA-Karte (set_video_mode(), set_palette(), set_pixel()) durch ihr Äquivalent auf dem Transputer (video_mode(), palette(), ega_plot()) ersetzt werden (Listing 2).

Der Lohn dieser relativ kleinen Mühen ist ein enorm beschleunigter Programmablauf. Statt der 8 Minuten, die der 20-MHz-Compaq benötigt, sind nur noch 9 Sekunden Programmausführungszeit zu veranschlagen, was schon über der Leistung des Compaq mit Weitek-3167-memory-mapped-Coprozessor liegt.

Master- und Worker-Prozesse

Aber nun zum wirklich interessanten Teil der Portierung: Der Umsetzung auf alle vier Transputer der Mega-Link01. Dazu zunächst etwas Theorie. Für viele parallele Berechnungen ist es sinnvoll, Anwendungen zu schreiben, die sich automatisch auf ein beliebiges Netzwerk von Transputern verteilen. Solche Anwendungen laufen automatisch schneller, wenn sich die Anzahl der Transputer eines Netzwerkes erhöht, und das ohne Recompilation oder Rekonfiguration. In dieser Methode, die als 'flood-fill'- oder 'processor-farm'-Technik bezeichnet wird, wird ein 'Master'-Prozeß kreiert, der die Aufgabe in kleine, voneinander unabhängige Teile - sogenannte 'work-packets' - aufsplittet, die dann von einer beliebigen Anzahl von gleichartigen 'Worker'-Prozessen bearbeitet werden. 'Work-packets' werden durch Routing-Software, die in Parallel C enthalten ist, auf das Transputer-Netzwerk verteilt. Die einzelnen 'Worker'- Prozesse kommunizieren über 'messages' mit dem 'Master'-Prozeß. Diese Message- Datenstruktur, die jeweils zur Anwendung passend gewählt werden muß, wird im Deklarationsteil des 'Master'- und des 'Worker'-Programms definiert (Listing 3 + 4). In unserem Beispiel besteht eine 'message' aus einem Integer-Wert für die Zeilennummer und 640 Byte für die Farbwerte jedes Pixels der Zeile.

Im 'Master'-Prozeß werden zu Beginn desweiteren zwei Funktionen definiert (send() und receive()), die parallel zum eigentlichen Programm als sogenannte 'threads' laufen und das Senden und Empfangen der 'work-packets' erledigen. Bis zum Programmteil, in dem die Pixel schattiert werden, sind dann keine Änderungen mehr notwendig.

Die Schleife selbst entfällt im 'Master'-Prozeß, da dieser Teil jetzt von den 'Worker'-Prozessen, die auf allen Transputern arbeiten, erledigt wird. Vielmehr werden hier die beiden 'threads' send() und receive() als hochpriorisierte Prozesse aufgerufen, die die einzelnen 'work-packets' - jeweils aus der Nummer der zu berechnenden Zeile bestehend - in das Netzwerk senden, und sich die Ergebnisse - aus der Zeilennummer und 640 Farbwerten bestehend - zurückholen und grafisch auf der EGA-Karte darstellen. Die hohe Priorisierung von send() und receive() folgt einem für Transputersysteme (fast) immer gültigen Merksatz, daß Kommunikation immer vorrangig abzuwickeln ist.

Somit laufen auf dem ersten Transputer jetzt insgesamt vier Prozesse ab:

send()- hoch priorisiert

receive()- hoch priorisiert

main()- niedrig priorisiert

worker-Prozeß- niedrig priorisiert

Die 'Worker'-Prozesse, die sowohl auf dem ersten als auch den weiteren drei Transputern der Mega-Link01 ablaufen, enthalten - neben der Berechnung einiger benötigter Variablen zu Beginn - jetzt nur noch den Teil, der ursprünglich in der Schleife codiert war, und einige zusätzliche Kommunikationsbefehle, die das reibungslose Zusammenspiel mit dem 'Master'- Prozeß ermöglichen. Im einzelnen sind diese net_receive(), der die Nummer der zu berechnenden Zeile einliest, und net_send(), der den gesamten Datenvektor - bestehend aus Zeilennummer und 640 Farbwerten - in das Netzwerk und somit zum 'Master'-Prozeß zurücksendet. Nach dem Zurücksenden dieses Vektors wird die Schleife mit einer anderen Zeilennummer erneut durchlaufen, bis alle Zeilen berechnet sind. Insgesamt dauert dies knapp unter 3 Sekunden. Der Grund dafür, daß hier 'nur' ein Geschwindigkeitszuwachs um den Faktor 3 - und nicht, wie zu erwarten gewesen wäre, Faktor 4 - erreicht wurde, liegt zum einen daran, daß die verwendete 'processor-farm'-Technik die hohe lokale Kommunikationsgeschwindigkeit des Supertransputer-Konzeptes der Mega-Link01 nicht voll ausnutzt, zum anderen, daß die Art der Datenversendung aus Vereinfachungsgründen nicht so optimiert wurde, wie das normalerweise geschehen würde. Eine - hier nicht abgedruckte - Version, die diese Optimierungen beinhaltet, erreichte einen Wert von 2,3 Sekunden, was einer Geschwindigkeitssteigerung um den Faktor 3,9 entspricht.

Da es meist nicht einfach ist, ein Programm, das auf einem Netzwerk von Transputern abläuft, zu debuggen, und herauszufinden, welchen Anteil an der Gesamtausführungszeit die einzelnen Programmteile haben (und somit, wo weitere Optimierungen sinnvoll und möglich sind) werden speziell für Parallel C, Parallel Fortran und Parallel Pascal einige leistungsfähige Tools angeboten, die bei der Entwicklung und Optimierung von Programmen in diesen Sprachen außerordentlich hilfreich sind. Ein komfortabler Debugger ermmöglicht schrittweises Abarbeiten der Programme, Auslesen und Änderung der benötigten Variablen auf allen Transputern. Ein Performance-Monitor zeigt genau an, welcher Prozeß wieviel Rechen- und Kommunikationszeit benötigt. Desweiteren sind Tools enthalten, die die Konfiguration eines Programms auf ein beliebiges Transputer-Netzwerk noch effizienter machen.

Robert Sang

Listing 1:

```
#include <math.h>
#include <stdio.h>
#include <dos.h>

floatpi;
intcolors[] = {3,6,10,13,6,3,10,13,6,3,13,10},
d[]       = {640,350,1},i,k,
x,y,x_min,x_max,y_min,y_max;
charpalette[]= {0,8,1,9,16,2,18,63,
              32,4,36,48,6,54,7,63,0};
unsigned shortrandom;

#include "ega.c";

main()
{
  float a,b,c,l0,l1,l2,ln,ln1,n0,n1,n2,p,q,r=128,s,t,v[12][3];
  int n;

  set_video_mode(0x10);
  set_palette();

  printf("\n\t\t80386 Phong-Schattierungs-Demo");

  /* Pixel aspect ratio */
  a=1.1;
```

```
/* Koordinaten der Bildschirmmitte */
b=.5*(d[0]-1);
c=.5*(d[1]-1);

/* Einheitsvektor der Lichtquelle */
l0=-1/sqrt(3.);
l1=l0;
l2=-l0;

pi=4*atan(1.);

v[0][0]=0;
v[0][1]=0;
v[0][2]=1;
s=sqrt(5.);
for (i=1;i<11;i++)
{
  p=pi*i/5;
  v[i][0]=2*cos(p)/s;
  v[i][1]=2*sin(p)/s;
  v[i][2]=(1.-i%2*2)/s;
}
v[11][0]=0;
v[11][1]=0;
v[11][2]=-1;

/* Schleife zur Schattierung jedes Bildpunktes */
y_max=c+r;
y_min=2*c-y_max;
for (y=y_min;y<=y_max;y++)
{
  s=y-c;
  n1=s/r;
  ln1=l1*n1;
  s=r*r-s*s;
  x_max=b+a*sqrt(s);
  x_min=2*b-x_max;
  for (x=x_min;x<=x_max;x++)
  {
    t=(x-b)/a;
    n0=t/r;
    t=sqrt(s-t*t);
    n2=t/r;
    ln=l0*n0+ln1+l2*n2;
    if (ln<0) ln=0;

    t=ln*n2;
    t+=t-l2;
    t*=t*t;
    t*=t*t;
    t*=t*t;
    for (i=0,p=0;i<11;i++)
      if(p<(q=n0*v[i][0]+n1*v[i][1]+n2*v[i][2]))
      {
        p=q;
        k=colors[i];
      };
```

```
      i=k-2.5+2.5*ln+t+(random=37*random+1)/65536.;
      if(i<k-2) i=0; else if (i>k) i=15;
      set_pixel(x,y,i);
    }
  }
}
```

Listing 2:

```
#include <math.h>
#include <par.h>
#include <stdio.h>
#include <dos.h>
#include <chan.h>
#include "ega.c"


floatpi;
intcolors[] = {3,6,10,13,6,3,10,13,6,3,13,10},
d[]       = {640,350,1},i,k,
x,y,x_min,x_max,y_min,y_max;
charpalette[]= {0,8,1,9,16,2,18,63,
             32,4,36,48,6,54,7,63,0};
unsigned shortrandom;


main()
{
  float a,b,c,l0,l1,l2,ln,ln1,n0,n1,n2,p,q,r=128,s,t,v[12][3];
  int n;

  video_mode(0x10);

  printf("\n\t\t  T800 Phong-Schattierungs-Demo");

  /* Pixel aspect ratio */
  a=1.1;

  /* Koordinaten der Bildschirmmitte */
  b=.5*(d[0]-1);
  c=.5*(d[1]-1);

  /* Einheitsvektor der Lichtquelle */
  l0=-1/sqrt(3.);
  l1=l0;
  l2=-l0;

  pi=4*atan(1.);

  v[0][0]=0;
  v[0][1]=0;
  v[0][2]=1;
  s=sqrt(5.);
  for (i=1;i<11;i++)
  {
    p=pi*i/5;
```

```
    v[i][0]=2*cos(p)/s;
    v[i][1]=2*sin(p)/s;
    v[i][2]=(1.-i%2*2)/s;
  }
  v[11][0]=0;
  v[11][1]=0;
  v[11][2]=-1;

  /* Schleife zur Schattierung jedes Bildpunktes */
  y_max=c+r;
  y_min=2*c-y_max;
  for (y=y_min;y<=y_max;y++)
  {
    s=y-c;
    n1=s/r;
    ln1=l1*n1;
    s=r*r-s*s;
    x_max=b+a*sqrt(s);
    x_min=2*b-x_max;
    for (x=x_min;x<=x_max;x++)
    {
      t=(x-b)/a;
      n0=t/r;
      t=sqrt(s-t*t);
      n2=t/r;
      ln=l0*n0+ln1+l2*n2;
      if (ln<0) ln=0;

      t=ln*n2;
      t+=t-l2;
      t*=t*t;
      t*=t*t;
      t*=t*t;
      for (i=0,p=0;i<11;i++)
        if(p<(q=n0*v[i][0]+n1*v[i][1]+n2*v[i][2]))
        {
          p=q;
          k=colors[i];
        };
      i=k-2.5+2.5*ln+t+(random=37*random+1)/65536.;
      if(i<k-2) i=0; else if (i>k) i=15;
      ega_plot(x,y,i);
    }
  }
}
```

Listing 3:

```
#include <math.h>
#include <stdio.h>
#include <dos.h>
#include <sema.h>
#include <par.h>
#include <chan.h>
#include "ega.c"
#include <net.h>
#include <thread.h>

floatpi;
intcolors[] = {3,6,10,13,6,3,10,13,6,3,13,10},
d[]       = {640,350,1},i,k,
ok,x,y,x_min,x_max,y_min,y_max;
charpalette[]= {0,8,1,9,16,2,18,63,
              32,4,36,48,6,54,7,63,0};
*ws;

unsigned shortrandom;

/* Datenvektor deklarieren */
struct msg {
int ident;
chardata[640];
}msg;

/* 'thread' zum Senden der Zeilennummer */
send()
{
int y;

for(y=y_min;y<y_max;y++)
net_send(sizeof(int),&y,1);
}

/* 'thread' zum Empfangen der Datenvektoren */
receive()
{
int x,y,d;

for(y=y_min;y<y_max;y++) {
net_receive(&msg,&x);
y=msg.ident;
                for(x=0;x<640;x++) {
                d=msg.data[x];
                if (d) ega_plot(x,y,d);
                }
        }
ok=1;
}


main()
{
  float a,b,c,l0,l1,l2,ln,ln1,n0,n1,n2,p,q,r=128,s,t,v[12][3];
```

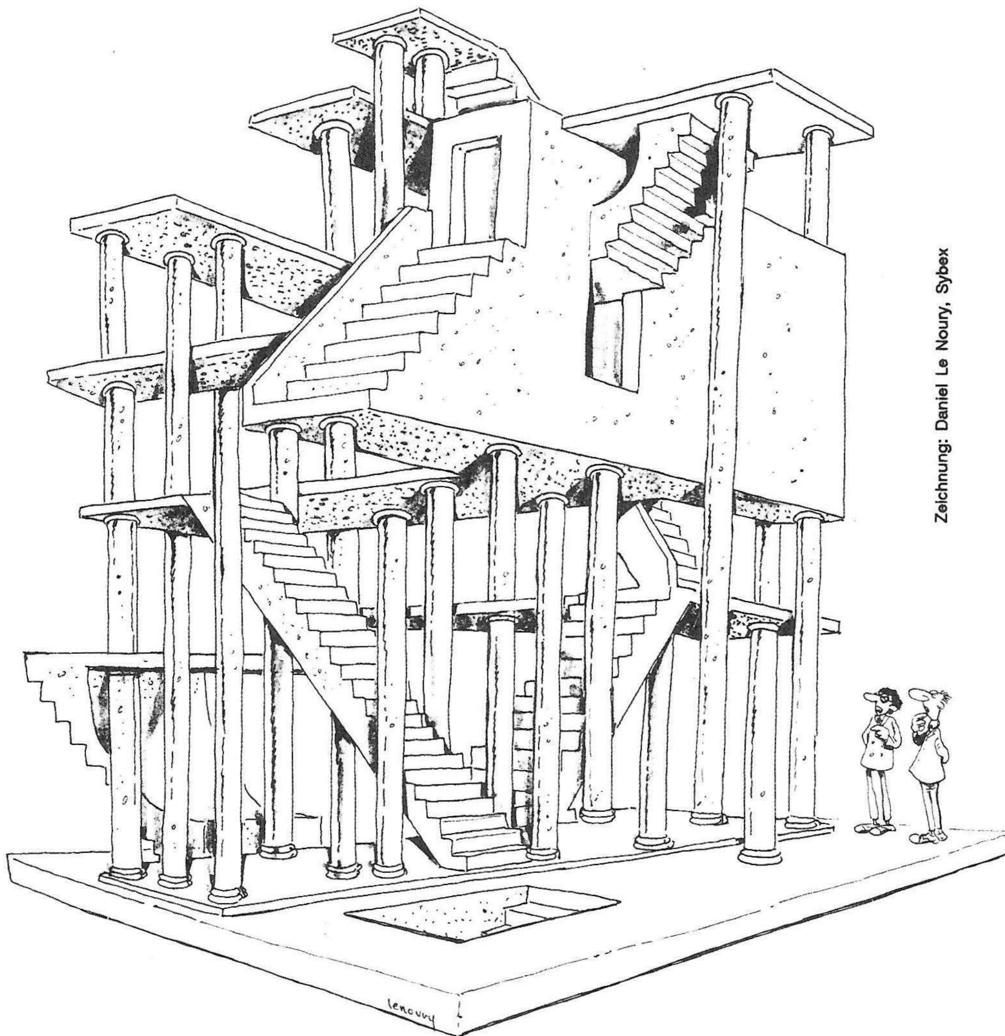
```
  int n;

  video_mode(0x10);
  palette();

  printf("\n\t\t4 * T800 Phong-Schattierungs-Demo");

  /* Pixel aspect ratio */
  a=1.1;

  /* Koordinaten der Bildschirmmitte */
  b=.5*(d[0]-1);
  c=.5*(d[1]-1);

  /* Schattierung jedes Bildpunktes */
  y_max=c+r;
  y_min=2*c-y_max;
  ok=0;

  /* Workspace fuer send() reservieren und send() starten */
  ws=calloc(1,10000);
  thread_start(send,ws,10000,THREAD_URGENT,0);

  /* Workspace fuer receive() reservieren und receive() starten */
  ws=calloc(1,10000);
  thread_start(receive,ws,10000,THREAD_URGENT,0);

  /* main() deschedulen = mehr Prozessorzeit fuer 'worker' */
  while(!ok) thread_deschedule();
}
```

Listing 4:

```
#include <math.h>
#include <sema.h>
#include <par.h>
#include <chan.h>


floatpi;
intcolors[] = {3,6,10,13,6,3,10,13,6,3,13,10},
d[]       = {640,350,1},i,k,
x,y,x_min,x_max;
charpalette[]= {0,8,1,9,16,2,18,63,
              32,4,36,48,6,54,7,63,0};

struct msg {
int ident;
chardata[640];
}msg;

unsigned shortrandom;


main()
{
```

```
float a,b,c,l0,l1,l2,ln,ln1,n0,n1,n2,p,q,r=128,s,t,v[12][3];
int n;

/* Pixel aspect ratio */
a=1.1;

/* Koordinaten der Bildschirmmitte */
b=.5*(d[0]-1);
c=.5*(d[1]-1);

/* Einheitsvektor der Lichtquelle */
l0=-1/sqrt(3.);
l1=l0;
l2=-l0;

pi=4*atan(1.);

v[0][0]=0;
v[0][1]=0;
v[0][2]=1;
s=sqrt(5.);
for (i=1;i<11;i++)
{
  p=pi*i/5;
  v[i][0]=2*cos(p)/s;
  v[i][1]=2*sin(p)/s;
  v[i][2]=(1.-i%2*2)/s;
}
v[11][0]=0;
v[11][1]=0;
v[11][2]=-1;

/* Schleife zur Schattierung jedes Bildpunktes */
for (;;)
{
  /* Datenvektor loeschen */
  for(x=0;x<640;x++) msg.data[x]=(char)0;
  /* Zeilennummer empfangen */
  net_receive(&y,&x);
  /* Zeilennummer in Datenvektor schreiben */
  msg.ident=y;
  s=y-c;
  n1=s/r;
  ln1=l1*n1;
  s=r*r-s*s;
  x_max=b+a*sqrt(s);
  x_min=2*b-x_max;
  for (x=x_min;x<=x_max;x++)
  {
    t=(x-b)/a;
    n0=t/r;
    t=sqrt(s-t*t);
    n2=t/r;
    ln=l0*n0+ln1+l2*n2;
    if (ln<0) ln=0;

    t=ln*n2;
```

```
        t+=t-12;
        t*=t*t;
        t*=t*t;
        t*=t*t;
        for (i=0,p=0;i<11;i++)
          if(p<(q=n0*v[i][0]+n1*v[i][1]+n2*v[i][2]))
          {
            p=q;
            k=colors[i];
          };
        i=k-2.5+2.5*ln+t+(random=37*random+1)/65536.;
        if(i<k-2) i=0; else if (i>k) i=15;
        /* berechnetes Element in Datenvektor schreiben */
        msg.data[x]=i;
      }
      /* Datenvektor zuruecksenden */
      net_send(sizeof(msg),&msg,1);
    }
}
```



Zeichnung: Daniel Le Noury, Sybex

Das Betriebssystem Helios

In den letzten zehn Jahren hat wohl keine neue Entwicklung die Fachwelt so aufgerührt wie das Transputer-Konzept. Seine wohldosierte Kombination aus Revolution und evolutionärer Weiterentwicklung, aus Avantgarde und Praktikabilität hat unter Ingenieuren und Informatikern überwiegend tiefe, ja geradezu erleichterte Zustimmung gefunden, zum Teil aber auch Bedenken hervorgerufen.

Neben einer Reihe von Bedenken, die eher auf die Neuheit und Ungewohntheit zurückzuführen waren, wurde auch immer wieder das Fehlen von Software-Standards vorgebracht. Zweifellos kann eine neue Architektur, die durch ihre Eigenschaften ja gerade aus bestehenden (Leistungs-)Standards ausbrechen will, sich nicht gleichzeitig zu ihnen konform verhalten. Diese Problematik wird ganz deutlich in der Frage der Programmiersprache ebenso wie in der Frage von Betriebssystemen.

Der Bereich der Programmiersprachen ist schon ziemlich früh abgedeckt worden. Beispiele sind Standard-Compiler für C, Fortran und Pascal, aber auch besonders gut zum Konzept passende Parallelverarbeitungssprachen wie Occam oder etwa Compiler für Parallel-C und Parallel-Prolog. Demgegenüber blieb die Betriebssystemfrage lange offen. Zum Teil mag dies daran gelegen haben, daß von den beiden Grundaufgaben des Betriebssystems, nämlich erstens Verwaltung der Entwicklungszeit-Ressourcen des Programmierers und zweitens Verwaltung der Laufzeit-Ressourcen des Prozessors wie Scheduling und Kommunikation nur noch die erste Aufgabe geblieben ist, weil die zweite beim Transputer bereits in Hardware und Microcode erledigt ist. So reichte es aus, daß zunächst leistungsfähige Entwicklungssysteme entstanden, während das Laufzeitgeschehen von der "nackten" Transputer-Hardware sehr leistungsfähig gesteuert wurde, insbesondere in allen Gebieten der Embedded Systeme, Automatisierungstechnik oder Spezialanwendungen wie Hochleistungsgrafik und Mustererkennung.

Helios versus Unix

Das Vordringen der Transputer auch in Anwendungsbereiche von Number-Crunchern und Workstations ließ schon früh den Ruf nach einem Standard-Betriebssystem ertönen, um Software-Austausch, Portabilität und standardisierte Anwendungsschnittstellen zu erreichen. Auf den ersten Blick lag es nahe, das hier weitverbreitete Unix zu nehmen und auf Transputer-Maschinen zu portieren. Folgerichtig wurde dies im ersten Eifer auch von vielen angekündigt, ohne daß sichtbare Erfolge zustande kamen. Der Grund war offensichtlich: Während Unix auf der einen Seite wertvolle Impulse für die Rechnerentwicklung gegeben hat, darf man auf der anderen Seite bei einem Konzept aus dem Ende der sechziger Jahre nicht erwarten, daß es auch noch die Prozessorarchitektur der neunziger Jahre abdeckt. Im Gegenteil: Beispielsweise würde die Verwendung der Unix- Methoden für Concurrency und Kommunikation ja gerade verlangen, die besten Eigenschaften des Transputers lahmzulegen.

Wie so oft, zeichnet sich die sehnlich erwartete Lösung auch hier durch ein verblüffendes Maß an Natürlichkeit und Einfachheit aus. Sie heißt Helios und verbindet das Beste beider Welten. Das von der Firma Perihelion aus der Taufe gehobene und in Kooperation mit Parsytec weiterentwickelte Betriebssystem kombiniert vertraute Benutzer- und Programm-Schnittstellen von Unix mit den überlegenden Parallelisierungs- und Kommunikationseigenschaften des Transputers. Darüber hinaus erlaubt es die Entwicklung von Standard- Software, die automatisch die in einem System vorhandenen Verarbeitungsressourcen ausnutzt und dabei auch auf gerade nicht verwendete Prozessoren einer Nachbarmaschine zurückgreift. Im Gegensatz zu Unix, das meistens über eine Sourcecode-Portabilität nicht hinaus geht, die nur bei Public-Domain-Software nützt, erlaubt Helios durch ein geschicktes Konzept in Verbindung mit den einheitlichen Transputer-Eigenschaften eine echte Objekt-Code-Portabilität für Maschinen unterschiedlicher Hersteller, so daß wegen der Marktbreite ein Anreiz für Software-Häuser besteht.

Das beste aber ist: Helios ist verfügbar und läuft bereits auf einer Reihe von Rechnersystemen von der 1-Transputer-Einsteckkarte bis zur großen rekonfigurierbaren "Supercluster"-Maschine mit 64 und mehr Prozessoren. Daß im nächsten Jahr die Transputer-Systeme von Atari und Commodore ebenfalls auf Helios basieren werden, läßt eine geradezu sprungartige Vergrößerung der Software-Basis voraussehen und festigt weiter die Position des Transputers als De-facto-Industriestandard für Parallelsysteme.

Die Helios-Konzeption

Helios ist ein echtes verteiltes Betriebssystem und eine Novität für kommerziell verfügbare Software, die erst mit dem Transputer praktisch möglich wurde. In seiner erstaunlich einfachen und anschaulichen Konzeption läßt Helios unwillkürlich an die arbeitsteilige Aufgaben- und

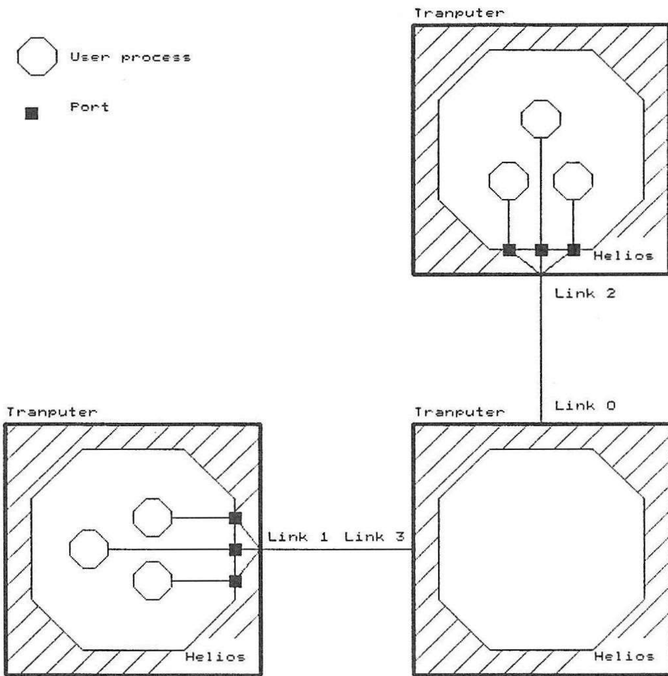


Bild 1: Etablierung eines Kommunikationsweges

Kommunikations- Organisation eines Büros denken. Hier wie dort gibt es ein globales Informationsverarbeitungsziel, wobei dieses in viele unterschiedliche Einzelaufgaben zerlegt ist, die von verschiedenen, miteinander Daten austauschenden "Prozessoren" durchgeführt werden. Im einen Fall sind dies Menschen als Sachbearbeiter,

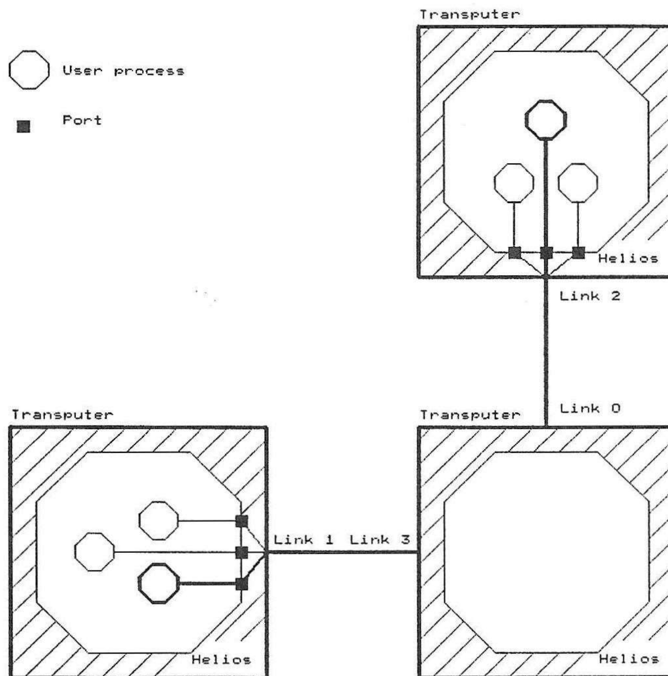


Bild 2: Message Passing über Ports

Manager und Archivverwalter, im anderen Fall Transputer in analogen Tätigkeiten. Die streng kommunikationsorientierte Architektur des Transputers ohne Direktzugriffe auf globale Speicher hat dieses Konzept sowohl angeregt als auch erst wirtschaftlich möglich gemacht.

Die Ressourcen des Systems sind durch sogenannte "Capabilities" für alle Arten von Objekten geschützt. Beim Anlegen eines Objektes wird eine verschlüsselte Zugriffsmaske zurückgegeben, die später beim Zugriff auf dieses Objekt mitgegeben werden muß und die auch in reduzierter Form an andere weitergegeben werden kann. Es gibt keinen "Super-User", allein schon auf Grund der verteilten Natur von Helios. Privilegien können nur von Capabilities mit höheren Zugriffsrechten als denen anderer herrühren. Für niemanden ist es möglich, automatisch vollen Zugriff zu allem zu haben.

Die Benutzerschnittstellen

Um einen möglichst einfachen Übergang zu Helios sicherzustellen, steht die allen Workstation-Benutzern vertraute Unix-C-Shell zur Verfügung. Diese enthält die üblichen Kommandos wie ls, grep, more usw. Auch nützliche und wichtige Hilfsmittel wie die make-Utility stehen in vollem Umfang zur Verfügung genauso wie zum Beispiel Foreground/Background-Processing. Einiger historischer

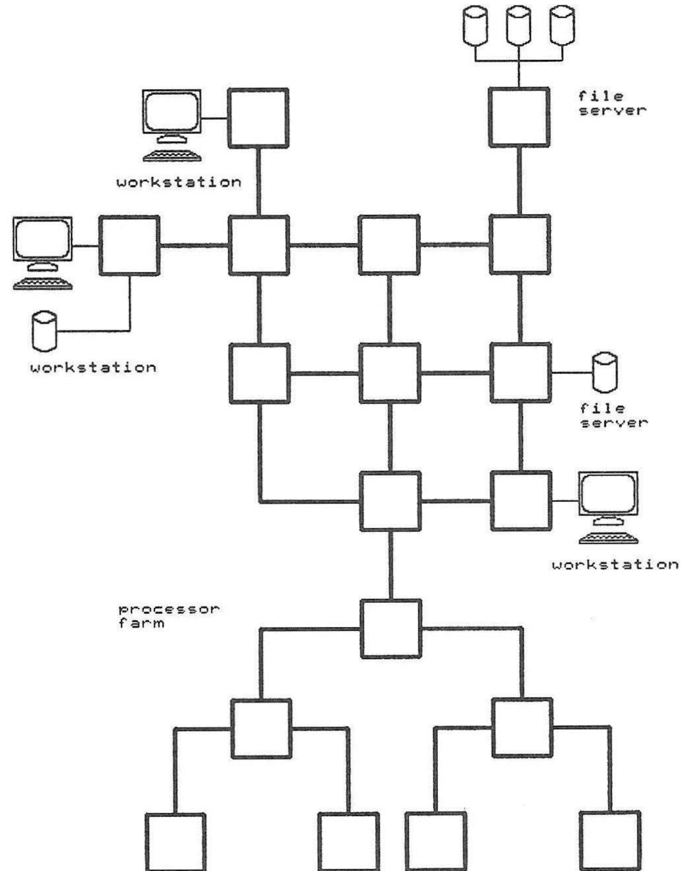


Bild 3: Helios Systemarchitektur

Ballast konnte ohne Nachteile für den Bediener weggelassen werden, so daß sich insgesamt ein recht benutzerfreundlicher Eindruck ergibt, unter anderem auch durch solche Eigenschaften wie Command History und Command Line Editing.

Wegen der einfachen Konvertierbarkeit existierender Unix-Programme steht insbesondere auch ein großer Fundus von Public-Domain-Software zur Benutzung unter der Helios-Shell zur Verfügung. So kann der Programmierer zum Beispiel für das Erstellen seiner Sources den bekannten icroemacs-Editor verwenden.

Die Standard-C-Shell wird für die meisten Helios-Benutzer das am besten gewohnte Interface sein für den Einsatz auf Standardsystemen wie zum Beispiel Personal-Computern mit Transputereinschüben oder den industriellen Megaframe- Transputersystemen. Da Transputersysteme durch ihre hohe Leistung und Erweiterbarkeit gerade auch im Grafikbereich besondere Stärken haben, wird zur weiteren Software-Unterstützung unter Helios zur Zeit der X-Windows-Standard in der Version 11 implementiert. Dieser moderne Grafikstandard basiert ebenfalls auf einem Client/Server-Modell und bietet dadurch besonders gute Performance-Eigenschaften unter Helios. Das X-Windows-System selbst läuft auf einem grafikfähigen Transputermodul und stellt als Server die entsprechenden Dienstleistungen im Helios-Netz zur Verfügung, unter anderem auch für eine Benutzer-Shell. Diese Shell kann dann natürlich auch Gebrauch von allen X- Windows-Eigenschaften machen und grafikorientierte Benutzerschnittstellen mit Maus und Pull-Down-Menüs bieten.

Die für derzeitige Unix-Benutzer sofort vertraute C-Shell erscheint auf den ersten Blick so, wie man es von einer konventionellen Maschine erwartet. Es gibt jedoch einige Unterschiede, die sich aus den zusätzlichen Leistungen eines Transputernetzes gegenüber Standard-Hardware erklären. Zum Beispiel kann Helios eine Pipe von Kommandos auf mehr als einen Prozessor verteilen, so daß die Ausführungsgeschwindigkeit steigt. Die können sowohl Prozessoren des eigenen Systems sein, aber genauso auch Prozessoren aus dem Pool, der durch die Zusammenschaltung mehrerer Systeme entstanden ist. Die Einfachheit und Natürlichkeit des Arbeitens unter diesem Betriebssystem mit Prozessornetzen wird aber auch durch andere Erweiterungen deutlich. So startet ein Benutzer zunächst auf einer Shell, die beim Booten auf den Prozessor gelegt wurde, der sich in direkter (Link-)Verbindung zu einem I/O-Prozessor befindet. Dieser I/O-Prozessor kann ebenso ein Standard-Host wie etwa ein IBM PC oder eine Sun- Workstation sein oder auch ein transputerbasiertes I/O-System. Von dieser Shell aus steht dem Benutzer alles zur Verfügung, was sich im System befindet, ohne daß er sich um Prozessorgrenzen kümmern muß. Er kann aber auch explizit, wenn etwa die derzeitige Shell auf Prozessor 00 läuft, mit dem Shell-Kommando eine Shell auf dem Prozessor 08 eröffnen und in dieser mit dem cd-Kommando als Working Directory ein Filesystem auf dem Prozessor 12 festlegen.

Eine weitere interessante Eigenschaft der Benutzeroberfläche liegt darin, daß Helios in einer Baumstruktur nicht nur wie ein klassisches Betriebssystem Dateien verwaltet, sondern in einem allgemeineren Sinne Objekte. Das können, wie aus Unix bekannt, Dateien und Directories sein, aber auch Prozessoren, Server-Prozesse und Kommunikationseinrichtungen.

Ein Beispiel verdeutlicht dies. Wenn der Benutzer mit dem Kommando:

```
ls/
```

ein Verzeichnis der Root-Directory anfordert, bekommt er mit

```
/00   /01   /IO
```

als Antwort zwei normale Prozessoren und einen I/O-Prozessor angezeigt (ls -1 liefert weitere Details). Das Kommando

```
ls /IO
```

liefert mit

```
Clock/   Console/    helios/
```

zwei Server und eine Directory, die man mit

```
ls /IO/helios
```

in der bekannten Weise aufgelöst bekommt:

```
bin/  include/   lib/   login    server.exe
```

Das Kommando

```
ls /00
```

zeigt, daß unter den verschiedenen auf Prozessor 00 verwalteten Objekten

```
Loader/ Tasks/  Window/ Link.0/
Link.1/ Link.2/ Link.3/
```

auch der Task-Verwalter ist, der nach

```
ls /00/Tasks
```

anzeigt, daß in diesem Augenblick die Prozesse

```
Loader   ProcMan Shell.1 ls.7    Window
```

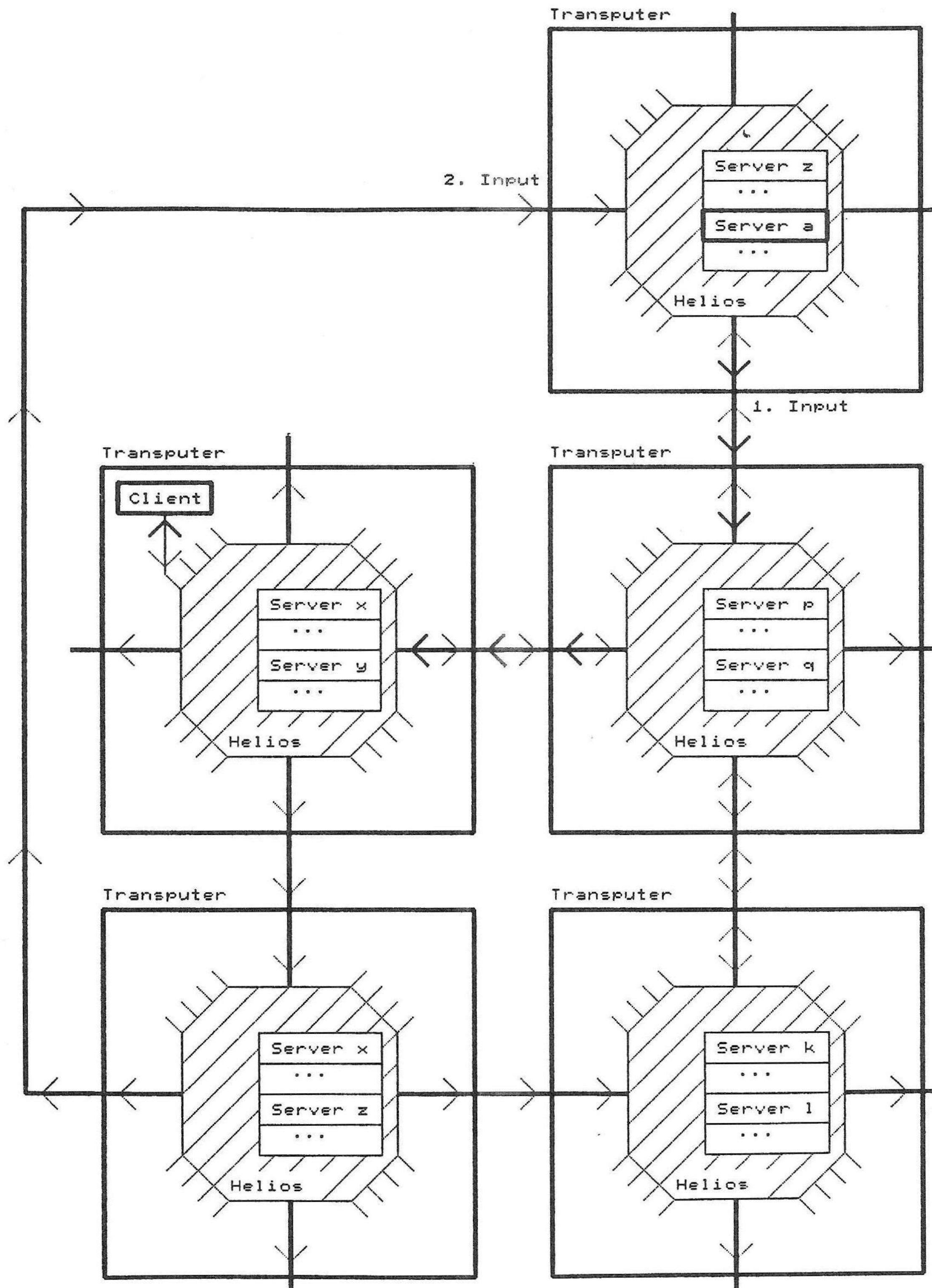


Bild 4: Identifikation von Systemresourcen

laufen. Auch die physikalischen Kommunikationseinrichtungen können explizit angesprochen werden. Wenn zum Beispiel der Link 2 des Prozessors 00 mit dem Prozessor 01 verbunden ist, liefert das Kommando

```
ls /00/Link.2
```

genau alle Objekte des Prozessors 01 (der natürlich auch direkt angesprochen werden könnte).

Programme unter Helios

Anwendungsprogramme finden unter Helios ähnliche Schnittstellen vor, wie sie unter Unix bekannt sind, um leichte Portabilität für existierende Anwendungssysteme zu bieten. Diese Schnittstellen und die Laufzeitverwaltung werden von dem Helios Nucleus geboten. Der Nucleus belegt einschließlich Daten etwa 100 KByte auf jedem Prozessor und besteht aus dem Kernel, der System-Library, dem Loader und dem Processor Manager.

Der Kernel verwaltet die Hardware-Ressourcen des jeweiligen Prozessors und ist für die Untersützung der Message-Passing-Mechanismen zuständig. Darüber hinaus ist in ihm der bereits erwähnte Name Server enthalten, der nachhält, wo sich im Netzwerk in Anspruch genommene Objekte befinden, oder bei der ersten Inanspruchnahme eine globale Suche im Netz initiiert.

Die System-Library stellt die Dienste bereit, die man von einer Laufzeit- Bibliothek erwartet. Da diese Dienste in Form von Servern eventuell auf anderen Prozessoren lokalisiert sein können, stellt die System-Library gleichzeitig eine Art von prozeduralem Interface dar, das die systemcall- ähnlichen Aufrufe der Anwendungsprogramme in das message-passing-orientierte Server-Protokoll des Helios umsetzt. Die funktional an Unix orientierte System-Library stellt zur Unterstützung sehr schneller Portierungen auch einen zweiten Satz von system-calls zur Verfügung, die exakt kompatibel zu Unix sind, dafür aber natürlich auf einige mit dem Transputer mögliche Optimierungen verzichten.

Der Loader verwaltet jede Art von Codes, die auf dem jeweiligen Prozessor geladen werden. Neben dem Code von Anwendungsprogrammen können dies auch residente Module sein, die von mehreren Programmen aus verwendet werden. Darüber hinaus ist er auch für das Laden von Objekten wie Zeichensätzen und Bit-Maps zuständig sowie für die Wiederfreigabe nicht mehr benötigten Platzes.

Der Processor Manager startet Tasks und unterstützt sie während ihres Laufes über einen IOC I/O-Controller als Interface zum Rest des Systems. Die Task ist die kleinste Einheit, die von Helios verwaltet wird. Auf dem Transputer ist der einzelne Prozeß eine so einfach und effizient verwaltete Einheit, daß das Betriebssystem sich um Process Creation und Scheduling genauso wenig kümmern muß wie etwa um Procedure Calls auf einem konventionellen Prozessor. Die Task besteht aus mindestens einem, im Normalfall aber einer Reihe von parallelen Prozessen. Umgebungsdaten wie geöffnete Files, Speicher und andere Ressourcen sind der ganzen Task zugeordnet.

Innerhalb einer Task kann ein Programm durch den Aufruf entsprechender System- Calls weitere Subtasks entweder auf dem gleichen Prozessor oder auch einem anderen im Netz laden und durch einen weiteren System-Call starten. Die Kommunikation über Message-Passing ist transparent bezüglich der physikalischen Lage.

Tasks können in verschiedenen Programmiersprachen implementiert sein, da die Kommunikation zwischen ihnen jeweils über die Message-Passing Primitive der System-Library erfolgt. Die ursprüngliche Standardsprache unter Helios ist C. Zur Zeit wird an einer Portierung des Megatool Transputer-Entwicklungssystems unter Helios gearbeitet, so daß bald das ganze Spektrum bisher entwickelter Compiler einschließlich Occam, Fortran und Parallel-Prolog sowie sonstiger Tools für Helios zur Verfügung steht.

Verteilte Server

Helios stellt sämtliche Dienste im System nach dem Client/Server-Modell zur Verfügung. Server werden über Message-Passing angesprochen und können dadurch, für die Client-Programme transparent, in beliebigen physikalischen Teilen des Netzes sitzen. Sie sind durch ihre Namen identifiziert und können mittels entsprechender Systemdienste im Helios-Nucleus vor der ersten Verwendung automatisch gesucht werden.

Bild 2 veranschaulicht einen solchen Vorgang. Hinter dem Client im mittleren linken Prozessor kann entweder ein Benutzer an einer Shell stehen oder ein Anwendungsprogramm. Wenn nach einem bestimmten Objekt, zum Beispiel nach einem Filesystem mit der Kennung A gesucht wird (etwa dem Server für ein Laufwerk mit diesem Namen), wird über den Nucleus eine globale Suche initiiert. Hierbei wird zunächst auf dem eingenen Prozessor gesucht und anschließend der Request über alle verbundenen Transputerlinks an den Nucleus der jeweiligen Nachbarprozessoren weitergegeben. Hier wiederholt sich das gleiche, so daß sich die Suchanfrage rasch in Form einer Welle ausbreitet. Über die interne Vergabe von Sequenz-Nummern wird verhindert, daß sich Schleifen bilden, in denen eine Anfrage kreist. Sobald ein Prozessor angesprochen wird, auf dem der entsprechende Server existiert (in Bild 2 auf dem rechten oberen), sendet dieser ein entsprechendes Acknowledge einschließlich einer Identifizierung des gesamten Pfades als Message-Port zurück. Der Rest der Welle "verläuft im Sande". Sobald das erste Acknowledge zum anfragenden Prozessor zurückgekommen ist, sperrt sich dieser für eventuelle spätere Acknowledges, da es im System weitere Server unter dem gleichen Namen geben kann.

In Form eines General-Server-Protocol ist die Form festgelegt, in der zwischen Client und Server Informationen ausgetauscht werden. Insbesondere arbeiten die Server "statusfrei", das heißt das Ergebnis eines Requests ist nicht abhängig von früheren Requests. Dies sichert auf einfache Weise Toleranz gegen physikalische Ausfälle im System. Wenn zum Beispiel ein Client feststellt, daß ein Server plötzlich nicht mehr antwortet, wird er einfach den

Suchvorgang noch einmal initiieren und dann die Anfrage wiederholen. Falls die Unterbrechung entstanden ist, weil eine Linkverbindung unterbrochen wurde (zum Beispiel durch Ausschalten eines zwischengeschalteten Rechners), wird der neue Verbindungsaufbau automatisch einen anderen Weg zu dem gleichen bisherigen Server finden. Falls der Server selbst ausgefallen ist (zum Beispiel ebenfalls durch Ausschalten des Rechners, auf dem er sitzt), wird sich automatisch ein weiter entfernt im Netz sitzender Server gleichen Namens und mit gleicher Funktion melden. Erst wenn beides nicht gegeben ist, entsteht für das Anwendungsprogramm eine nicht korrigierbare Fehlerbedingung.

Die im Netz verteilten Server können auf unterschiedlichster Hardware implementiert sein. Da ein Client nicht weiß und auch nicht wissen will, wo "sein" Server sitzt, kann er auch auf einem Nicht-Transputersystem laufen. Einzige Bedingung ist, daß dieses System über einen Link-Adapter mit dem Netz verbunden ist und sich an das General-Server-Protocol hält. Hierdurch ist es möglich, in einem Helios-System kostengünstige Hostsysteme wie zum Beispiel einen IBM PC für Filing-System und Bedienerschnittstellen zu verwenden und Spezialrechner für Sonderaufgaben wie etwa Grafiksysteme. Auch können so die Systemdienste verschiedener Hosts in einheitlicher Weise in einem einzigen Helios-System verknüpft werden.

Message-Passing und Kommunikations-Support

Wie bereits mehrfach dargestellt, ist Message-Passing die zentrale Kommunikationsmethodik unter Helios. Veranlaßt durch einen Benutzerrequest werden Messages über Helios zwischen zwei Prozessen ausgetauscht, wobei es keine Rolle spielt, ob die Prozesse auf dem gleichen oder auf verschiedenen Prozessoren sitzen. Durch diese Transparenz kann ein Programm so geschrieben sein, daß es entweder auf einem oder auch auf mehreren Prozessoren läuft, sofern diese verfügbar sind. Das Anwendungsprogramm sendet seine Messages an einen sogenannten Message-Port, der von Helios verwaltet wird. Ein Programm kann natürlich viele Ports verwenden. Jeder Port steht für eine logische Punkt-zu-Punkt-Verbindung in ähnlicher Weise wie ein Kommunikationskanal in der Programmiersprache Occam. Helios selbst verwendet exklusiv die physikalischen Links, um die über Ports realisierten Kanäle abzubilden. Multiplexing und Routing geschieht dabei völlig transparent.

Ein Port ist durch einen 32-bit-Descriptor beschrieben, der durch einen Locate-System-Call beim Verbindungsaufbau an das Anwendungsprogramm zurückgegeben wird. Der Empfänger kann entweder durch seinen Namen oder durch den Weg dorthin über Links beschrieben sein. Falls der Empfänger-Prozeß wie im Beispiel auf einem anderen Prozessor residiert, gibt Helios eine (formal gleichen) Port-Descriptor für einen sogenannten Surrogate-Port zurück. Dieser repräsentiert einen Port auf dem nächsten Prozessor auf dem physikalischen Weg zum Empfänger. Dieser Port kann natürlich wiederum ein Surrogate-Port sein bis hin zum letztendlichen Empfänger. Auf diese Weise etabliert Helios eine Art Standleitung über mehrere Zwischenstationen wie in Bild 3 angedeutet. Für das Anwendungsprogramm ist die Anzahl der Zwischenstationen belanglos, da es zum Senden und Empfangen von Messages an die zuständigen System-Calls PutMsg und GetMsg nur die lokalen Port-Descriptoren übergibt.

Die Kommunikation über Ports entspricht in vollem Umfang dem Kommunikationskonzept des Transputers. Sie ist synchron, wodurch sowohl Anwendungslogik als auch Systemimplementierung einfach werden, und darüber hinaus ist sie auch schnell. So wurden zum Beispiel auf einem Megaframe MTN-2 Doppeltransputermodul als Netto-Kommunikationsleistung zwischen zwei Prozessoren 4,255 Messages pro Sekunde der Länge 1024 Byte gemessen oder 7,011 Messages der Länge 100 Byte. Hierbei können Anwendungsprozesse noch parallel auf dem gleichen Prozessor weiterarbeiten. Bemerkenswert ist insbesondere auch der Erhalt der dem Transputer eigenen sehr kurzen Kommunikations-Setupzeiten.

Portabilität und Verfügbarkeit

Helios ist durch sein Client/Server-Konzept in Verbindung mit dem linkgestützten Message-Passing sehr portabel. Da es keine Bedingungen an die Topologie eines Transputersystems stellt und der Maschinencode der Transputer per definitionem in jedem System gleich ist, laufen Anwendungsprogramme ohne Problem auf Transputermaschinen verschiedener Hersteller. I/O-Schnittstellen können sowohl auf Transputern implementiert sein als auch auf Standard- Hostrechnern. Die Protokolle erlauben, auf einem Standard-Host auch mehrere Server zu implementieren. So können zum Beispiel Basisfunktionen in kompatibler Weise für eine Reihe sehr unterschiedlicher Hersteller implementiert sein, leistungsfähige Spezial-Hardware erlaubt darüber hinaus für bestimmte Hosts auch die Übernahme von Server-Aufgaben, die in anderen Systemen von Transputer-Prozessoren erbracht werden. Eine X-Windows- Schnittstelle wird etwa in einem System mit einem IBM PC als Bediener-Frontend aus Geschwindigkeitsgründen sicher in ein Transputer-Grafikmodul gelegt, während sie bei einem Sun-Frontend-Rechner im Host liegen kann. Die Transparenz des Helios-Client/Server-Modells läßt diese Freiheit auch nicht zu Lasten der Anwendungsprogramme gehen.

Helios ist keine Zukunftsvision, sondern Realität. Versionen für die Verwendung mit PC-Frontends sind neben anderen bereits ausgeliefert. Voll kompatible Versionen für die Verwendung mit Sun-, Vax-, Macintosh II- und anderen Hosts stehen unmittelbar vor der Fertigstellung.

Falk-D. Kübler

Software-Entwicklung mit Spectral

Transputersysteme besitzen von Haus aus eine Reihe von Eigenschaften, die die Bearbeitung eines Problems durch eine Anzahl von Prozessoren wirkungsvoll unterstützen. Bei der Erstellung von Software für diese Systeme, insbesondere bei einer größeren Anzahl von Prozessoren oder Prozessen, treten immer wieder die gleichen Schwierigkeiten auf. Sie rühren daher, daß in Occam Software- und Hardware-Definitionen vermischt worden sind, und daß keine Betriebssystem- Funktionen vorhanden sind.

Im folgenden wird dargestellt, bei welchen Entwicklungsschritten Probleme entstehen. Es wird geschildert, wie mit Hilfe des grafischen Spezifikationssystems Spectral (Spezifikationstool für transputer-basierte Sprachen) diese Schwierigkeiten umgangen werden. Als Ergebnis wird dadurch der erforderliche Codierungsaufwand um einen bedeutsamen Teil reduziert.

Was leistet Spectral?

Die Entwicklung von Software für größere Systeme erfolgt in vielen Fällen schrittweise, indem zunächst die einzelnen Programmteile unabhängig voneinander auf ihre Funktion getestet werden. Dazu reicht ein Einprozessor- System aus, da die Parallelität hierbei noch keine Rolle spielt. Im nächsten Schritt werden die parallelen Eigenschaften des Programms überprüft. Da hierbei noch keine hohe Rechenleistung erforderlich ist, genügt es, ein kleines Transputer-System mit zwei bis vier Prozessoren zur Verfügung zu haben, um das parallele Verhalten auszutesten. Nach Abschluß dieser Phase wird dann das Software-System auf ein größeres Transputer-System portiert, um dort die eingentliche Arbeit auszuführen.

Beim Übergang von kleineren Systemen auf größere Systeme treten folgende Schwierigkeiten auf:

Der Benutzer muß in Occam immer die Plazierung der Prozesse auf die Prozessoren und die Abbildung der im Programm definierten Kanäle auf die Links angeben. Sobald er auf ein anderes System mit einer anderen Transputer-Anzahl übergeht, müssen diese Teile im Programm verändert werden. Bei Programmen für kleine Systeme ist das ein Schritt, der wenig Aufwand erfordert. Bei der Erstellung von Programmen mit hundert oder mehr Prozessen ist dafür jedoch ein erheblicher Aufwand an Überlegung und Codierungsarbeiten zu erbringen.

Die PLACED PAR-Anweisung sieht zum Beispiel auf einem Vier-Transputer-System vollständig anders als auf einem 16-Transputer-System aus. Wenn man eine reguläre Anordnung der Prozesse auf die Prozessoren im Programm vorgesehen hat, kann man mit Hilfe von Index-Berechnungen diesen Schritt vereinfachen. Bei Prozeßgraphen, die nicht wie etwa Gitter, Ring, Kette regulär sind, ist dieser Schritt jedoch nicht möglich und es muß eine neue Zuordnung programmiert werden. Dieser Vorgang ist darüberhinaus sehr fehleranfällig.

Um dieses Problem grundsätzlich zu umgehen, stellt Spectral für den Programmierer ein hardware-unabhängiges Bild zur Verfügung. Das bedeutet, daß der Benutzer sich während der Programmierung keine Gedanken darüber zu machen braucht, wie viele Prozessoren für sein Problem zur Verfügung stehen. Bei der Erstellung seiner Software hat er eine unbeschränkte Anzahl von Prozessen zur Verfügung. Diese Menge von Prozessen wird dann automatisch auf die real zur Verfügung stehende Menge von Prozessoren abgebildet.

Eine weitere Einschränkung ist die Anzahl der Links, die pro Transputer zur Verfügung stehen. Um auch hier von der Hardware unabhängiger zu werden, ist es möglich, für jeden Prozeß eine beliebige Anzahl von Kanälen zu spezifizieren. Die Kanäle werden ebenfalls automatisch den Links der Prozessoren zugewiesen. Der Benutzer ist also nicht darauf angewiesen, mit vier Kanälen pro Prozeß oder Prozessor auszukommen.

Dieser Punkt kann so zusammengefaßt werden, daß bei der Benutzung von Spectral Hardware-Einschränkungen nicht berücksichtigt werden müssen. Die erstellten Programme können ohne zusätzlichen Programmierungsaufwand auf unterschiedlichen Hardware-Konfigurationen ablaufen.

Wie erreicht Spectral dieses Ziel?

Bei der Konzeption wurde davon ausgegangen, daß möglichst wenig Verluste durch ein Betriebs- oder Laufzeitsystem entstehen sollten, um die volle Leistung des Transputer-Systems zu erhalten. Daher schied eine für alle Algorithmen gleiche Lösung, wie sie zum Beispiel durch ein Betriebssystem zur Verfügung gestellt wird, aus.

Spectral arbeitet so, daß zuerst jedes spezifizierte Programm daraufhin untersucht wird, wie viele Prozesse es enthält. Wenn die Anzahl der Prozesse größer ist als die Anzahl der zur Verfügung stehenden Prozessoren, muß entschieden werden, welche Prozesse gemeinsam auf einen Prozessor kommen, um das Programm auf die Hardware abzubilden (Mapping).

Als nächstes wird geprüft, ob die auf einem Prozessor plazierten Prozesse mehr als vier Links benötigen. Dann muß mit Hilfe von zusätzlichen Prozessen sichergestellt werden, daß alle Kanäle auf die vorhandenen vier Links abgebildet werden können. Ein Multiplexing-Verfahren vereinigt alle zu einem Link gehörenden Kanäle; ein zugehöriges Demultiplexing-Verfahren auf der Gegenseite einer Verbindung untersucht die eintreffenden Nachrichten und verteilt sie entsprechend weiter.

Da dieses Verfahren generell angewendet wird, können beliebige Hardware- Konfigurationen als Ziel für das generierte Programm angegeben werden.

Wie hilft das grafische Spezifikationssystem dem Benutzer weiter?

Bei der Software-Entwicklung wird ein hohes Maß an grafischer Unterstützung angeboten, da Parallelität einfach und beherrschbar sein soll. Grafische Darstellungen sind dabei eine große Hilfe für den Benutzer.

Das Design eines parallelen Programmes erfolgt häufig so, daß zunächst eine Skizze mit den Prozessen und ihren Verbindungen erstellt wird. Dazu gehört eine Angabe, welchen Beitrag die einzelnen Prozesse für die Bearbeitung des Problems leisten sollen. Dieser Vorgang ist mit Spectral nachgebildet worden. Der Benutzer erstellt an einem Grafik-Bildschirm das Bild der Prozesse mit ihren Verbindungen und definiert dann die Programme der einzelnen Prozesse. Aus diesen Angaben wird das ablauffähige parallele Programm für eine zugrunde liegende Hardware generiert.

Wie wird Spectral eingesetzt?

Der Benutzer beginnt die Spezifikation eines parallelen Programmes mit der grafischen Definition des Prozeßnetzes. Dazu stehen ihm vier Basisobjekte zur Verfügung: Prozesse, Ports, Protokollspezifikationen, Verbindungen.

Prozesse werden als große Rechtecke dargestellt. Bei parallelen Programmen mit einer großen Anzahl von Prozessen kann man davon ausgehen, daß viele Prozesse gleiche oder ähnliche Aufgaben wahrnehmen. Dann bietet sich die Benutzung des Prozeßtypen-Konzepts an. Das bedeutet, daß die Eigenschaften in einem Typ festgelegt werden und dieser Typ dann mehrfach in das Prozeßnetz eingefügt (inkarniert) wird. Mit einer kleinen Anzahl von Prozeßtypen (nach unseren Erfahrungen drei bis zehn) kann man wirkungsvoll ein großes Prozeßnetz spezifizieren.

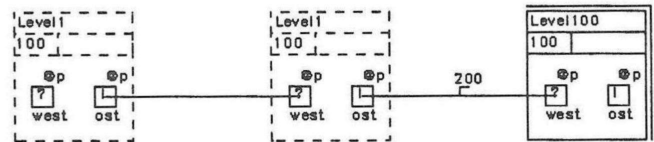


Bild 1

Dazu steht der Replikationsmechanismus zur Verfügung. Mit ihm kann man beliebig viele Inkarnationen eines Prozeßtyps erzeugen und dabei gleichzeitig eine Verbindungsregel angeben.

Der Name des Prozeßtyps wird in die erste Zeile des Prozeßkastens eingetragen. Im Beispiel sieht man drei Prozesse und zwar zwei Prozesse des Typs Level1 und einen Prozeß des Typs Level100.

Wenn die Eigenschaften einer einzelnen Inkarnation von mehreren Inkarnationen eines Typs nachträglich verändert werden, wird automatisch durch das Anhängen von Ziffern an den Typnamen ein neuer Name erzeugt. Der Prozeß Level100 war ursprünglich auch eine Inkarnation von Level1 und ist durch die Verfeinerung zu einem Netz geändert worden. Da er dadurch andere Eigenschaften besitzt, mußte ein neuer Typ generiert werden.

Ein Port dient zur Kennzeichnung einer Kommunikationsbeziehung zu einem anderen Prozeß. Er wird als ein kleines Quadrat dargestellt, in das entweder ein ?, ein ! oder ein # zur Kennzeichnung der Richtung eingetragen wird. Ein ? bedeutet dabei Eingabeport, ein ! Ausgabeport und ein # bezeichnet einen Port, der in beiden Richtungen (bidirektional) benutzt wird.

Im Beispiel gibt es zwei Port-Typen, einen Eingabeport-Typ West und einen Ausgabeport-Typ Ost.

Die Protokolle, die bei der Kommunikation zwischen Prozessen verwendet werden, müssen in einer besonderen Datei angegeben werden. In der Grafik werden Bezüge auf diese Datei mit einem @ und einem Buchstaben angegeben. Im Beispiel wird nur ein Eintrag benutzt, der unter dem Buchstaben p zu finden ist.

Verbindungen zwischen Prozessen werden als Kanten zwischen Ports dieser Prozesse dargestellt. So sieht man in dem Beispiel, daß der Port Ost des linken Prozesses vom Typ Level1 mit dem Port West des rechten Prozesses des gleichen Typs verbunden ist. Der Port Ost dieses Prozesses ist mit dem Port West des Prozesses Level100 verbunden.

Die Zahlen an den Verbindungen und in den Prozeßkästen dienen dazu, dem Mappingprogramm, das die Zuordnung der Prozesse zu den Prozessoren leistet, Informationen zu liefern. Um die Prozessoren gleichmäßig zu bela-

sten, muß der benötigte Rechenaufwand der Prozesse geschätzt werden. Der Benutzer kann nun Prozesse mit einer vom Durchschnitt stark abweichenden Rechenleistung kennzeichnen. Eine große Zahl bedeutet dabei hohen und eine niedrige geringen Rechenbedarf. Im Beispiel haben alle Prozesse den Standardbedarf 100.

Da der Nachrichtenaustausch zwischen Prozessen auf einem Prozessor wesentlich schneller abläuft als zwischen Prozessen auf verschiedenen Prozessoren, wird dieses Konzept auch für die Kommunikation verwendet. Verbindungen, die besonders häufig benutzt werden, können vom Benutzer besonders gekennzeichnet werden, um dem Mappingprogramm mitzuteilen, daß es dies bei der Zuordnung von Prozessen auf Prozessoren beachten soll. So ist im Beispiel die rechte Verbindung mit dem Gewicht 200 gekennzeichnet worden, um anzuzeigen, daß hier überdurchschnittlich viel kommuniziert wird.

Wie werden große Prozeßmengen dargestellt?

Da auf einem Grafik-Bildschirm nur eine beschränkte Menge von grafischen Symbolen angeordnet werden kann (bei Spectral sind es etwa 40 Prozeßkästen), wurden zwei Mechanismen realisiert, um große Prozeßmengen zu spezifizieren.

Zum einem ist es möglich, den Grafik-Bildschirm als Fenster zu betrachten, das jeweils einen Ausschnitt des Prozeßnetzes zeigt. Durch Bewegung dieses Fensters können die einzelnen Teile des Prozeßnetzes bearbeitet werden. Mit Hilfe der Zoomfunktion kann der Aufbau des gesamten Prozeßnetzes angezeigt werden und der Ausschnitt, der als nächstes bearbeitet werden soll, ausgewählt werden.

Diese Methode hat den Nachteil, daß sie den strukturellen Aufbau eines Algorithmus nicht wiedergibt. Dazu dient das zweite Verfahren: die Hierarchisierung.

Hierbei wird ein Prozeßkasten wieder als ein Netz betrachtet; anstelle eines Kastens wird ein Prozeßnetz eingefügt. So ist in dem obigen Beispiel der Prozeß mit der Typbezeichnung Level100 zu einem Netz verfeinert worden. Die Ports West und Ost dieses Prozesses müssen in diesem Prozeßnetz verwendet (gebunden) werden.

Der Eingabeport West wird dazu benutzt, Informationen in dieses (unterliegende) Prozeßnetz einzugeben und der Port Ost dazu, Informationen aus diesem Prozeßnetz nach außen zu senden. Die Berechnung, die in diesem Prozeßnetz durchgeführt wird, ist somit von der grafischen Darstellung vollkommen getrennt. Mit Hilfe der Open-Funktion kann angegeben werden, daß ein Prozeß zu einem Netz verfeinert werden soll und mit Hilfe der Close- Funktion kann zu der darüberliegenden Ebene zurückgekehrt werden. In dem Beispiel würde der Open-Befehl den Prozeßkasten Level100 öffnen und das

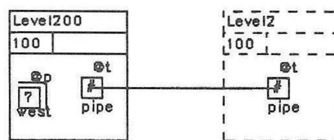


Bild 2

gesamte Bild durch das unterliegende Prozeßnetz ersetzen.

Hier sieht man eine begonnene Spezifikation, die auch schon ausreichend ist, ein teilweise definiertes paralleles Programm zu erzeugen. Es sind zwei Prozesse definiert (Level200 und Level2). Der Prozeß Level200 besitzt zwei Ports West und Pipe. Der Port West besitzt einen teilweise doppelten Rand, um anzuzeigen, daß es sich um einen verfeinerten Port handelt. Ebenso wie der Port Ost sind diese beiden Ports dadurch entstanden, daß der Prozeß auf der höheren Ebene zu einem Netz verfeinert worden ist. Diese Ports müssen in entsprechende Prozesse auf der tieferen Ebene eingebunden werden. Der Prozeß mit Namen Ost ist noch ungebunden und muß im Verlauf der weiteren Definition an einen Prozeß gebunden werden.

Die Programme der einzelnen Prozesse werden auf eine ähnliche Weise in das Prozeßnetz eingefügt. Es ist möglich, einen Prozeßkasten zu einem sequentiellen Prozeß zu verfeinern; hierbei wird der Prozeßkasten mit einer durchgezogenen Linie gekennzeichnet. Im Beispiel ist der Prozeß Level200 zu einem sequentiellen Prozeß verfeinert worden. Beim Öffnen dieses Prozesses erhält man ein herkömmliches Textfenster, in das der Programmtext eingetragen werden muß.

Es lassen sich also drei verschiedene Prozeßobjekte unterscheiden: Anfangs ist ein Prozeßobjekt undefiniert (durch gestrichelten Kasten dargestellt). Es ist möglich, ein solches Prozeßobjekt entweder zu einem Prozeßnetz (teilweise doppelte Linie am Rand) oder zu einem sequentiellen Prozeß (durchgezogener Rand) zu verfeinern.

Die Programmierung der einzelnen Prozesse

Der Programmtext eines sequentiellen Prozesses, wie er vom Benutzer zu definieren ist, beginnt mit der Zeile nach der PROC-Deklaration. Die Parameter der PROC-Deklaration werden von Spectral dazu benutzt, Kanalparameter zu übergeben und gegebenenfalls das Multiplexing- und Demultiplexing-System zu steuern, das erforderlich ist, wenn mehr als vier Verbindungen pro Transputer benötigt werden. Es ist nicht erforderlich, die CHAN-An-

weisung zu verwenden, da alle Kanäle automatisch erzeugt werden.

Der Benutzer braucht nur die herkömmlichen sequentiellen Teile von Occam zu verwenden sowie die Send- und Receive-Anweisung und das PAR-Statement. Der parallele Aufruf der einzelnen Prozesse in den Transputern wird ebenfalls von Spectral generiert. Wie schon oben erwähnt werden auch die PLACED PAR-, PROCESSOR- und PLACE-Anweisungen automatisch generiert, so daß der Benutzer diesen Teil des Occam-Sprachumfanges nicht benötigt.

Es reicht also aus, wenn der Benutzer das Prozeßnetz mit den Kommunikationsbeziehungen zwischen den Prozessen definiert und zusätzlich die sequentiellen Prozeßteile programmiert. Spectral generiert daraus automatisch ein paralleles Programm mit den entsprechenden Anweisungen und fügt - falls erforderlich - das entsprechende Laufzeitsystem hinzu, um die Hardware-Beschränkungen (mehr als ein Prozeß pro Prozessor, mehr als vier Verbindungen pro Transputer) zu umgehen.

Ein Beispiel - Game of Life

Der Algorithmus des Game of Life (Spiel des Lebens) stellt die Welt mit einer Gitterstruktur dar. Jeder Gitterpunkt besitzt einen Zustand, der in Abhängigkeit von den Nachbarzuständen ausgerechnet wird. Die Punkte kann man mit Lebewesen vergleichen, die bei guten Umgebungsbedingungen wachsen und bei schlechten nicht mehr existieren können.

Die Ausgabe des Programms soll ein Animationssystem steuern, das die Gitterpunkte und ihre Veränderungen farbig darstellt.

Die Spezifikation des Programms erfolgt auf zwei Ebenen. Auf der ersten Ebene werden zwei Prozeßobjekte definiert: der sequentielle Prozeß HOST und der Prozeß SUBNET, der zu einem Unternetz verfeinert worden ist.

Die beiden Prozesse besitzen jeweils einen bidirektionalen Port Pipe und den Eintrag in der Protokollspezifikationsdatei c. Der sequentielle Teil von HOST (auszugsweise in Bild 4 wiedergegeben) besteht im wesentlichen aus Ein-, Ausgabekommandos, aus der Initialisierung des Netzes und dem Sammeln und Darstellen von Ergebnissen. Wie schon der Prozeßtyp-Name angibt, steuert dieser Prozeß die eigentliche Berechnung und dient als Schnittstelle zwischen dem parallelen Teil und dem Benutzer.

Unter dem Prozeß SUBNET liegt das eigentliche Gitter, auf dem die Berechnung durchgeführt wird. Es wurde eine 6 x 6 Gitterstruktur (ein Ausschnitt in Bild 5) gewählt, um darzustellen, wie man einen Bildschirm von Spectral ausnutzen kann. Mit den oben erwähnten Möglichkeiten - Zoom und weitere Unternetze - lassen sich natürlich beliebig höhere Prozeßanzahlen spezifizieren.

Es werden zwei verschiedene Kommunikationsbeziehungen angelegt. Einmal eine Pipeline über alle Prozesse mit Hilfe der Ports Up und Down. Diese Pipeline dient dazu, die Anfangsbelegung des Netzes allen Prozessen bekannt zu machen und die Ergebnisse an HOST zurückzusenden. Der Prozeß links oben mit Namen CELL00 besitzt keinen Port Up, sondern einen verfeinerten Port Pipe (kenntlich am teilweise doppelten Rand). Dieser Port Pipe ist durch die Verfeinerung des Prozesses SUBNET auf der höheren Ebene entstanden und dient dazu, die Ergebnisse aus dieser Pipeline an HOST weiterzugeben.

Jeder Prozeß besitzt darüberhinaus vier weitere Ports mit den Anfangsbuchstaben der vier Himmelsrichtungen, die dazu dienen, die eigentliche Berechnung mit den vier Nachbarn eines jeden Prozesses durchzuführen.

Man kann auf dem Ausschnitt zwei Prozeßtypen unterscheiden. Der Prozeß CELL00 ist aus dem Prozeß CELL hervorgegangen, da er statt des Ports Up einen Port Pipe

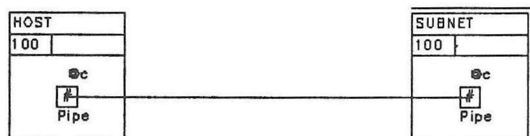


Bild 3

```
...  DECLARATIONS
SEQ
  ...  INITIALIZATION
  ...  READ FROM INPUT FILE & SEND TO CELL ARRAY
  {{{  RECEIVE RESULTS FROM CELL ARRAY & WRITE TO FILE AND SCREEN
  PAR
    scrstream.to.file (Sink, from.user.filer[0], to.user.filer[0],
                       "File", SinkFoldNumber, Result)
    SEQ
      CreateLattice (Width, Color.0)
      SEQ i = 0 FOR Width
        SEQ j = 0 FOR Width
          IF
            CellStateArray[i][j] = 1
              ColorNode (i, j, Color.1)
            TRUE
              SKIP
      SEQ LoopCounter = 0 FOR NumberOfLoops
        SEQ
          SEQ i = 0 FOR Width
            SEQ j = 0 FOR Width
              SEQ
              --ReceiveDataOutOfPipe:
                Pipe.in ? Size ; Data.array
                IF
                  Data.array[0] <> CellStateArray[i][j]
                    SEQ
                    --UpdateCellStateArray:
                      CellStateArray[i][j] := Data.array[0]
                    --UpdateNodeColor:
                      IF
                        CellStateArray[i][j] = 0
                          ColorNode (i, j, Color.0)
                        CellStateArray[i][j] = 1
                          ColorNode (i, j, Color.1)
                        TRUE
                          SKIP
                  TRUE
                    SKIP
          write.int(screen,LoopCounter+1,8)
          write.full.string(screen,". Iteration done ...")
      write.endstream (Sink)
  }}}
```

Bild 4

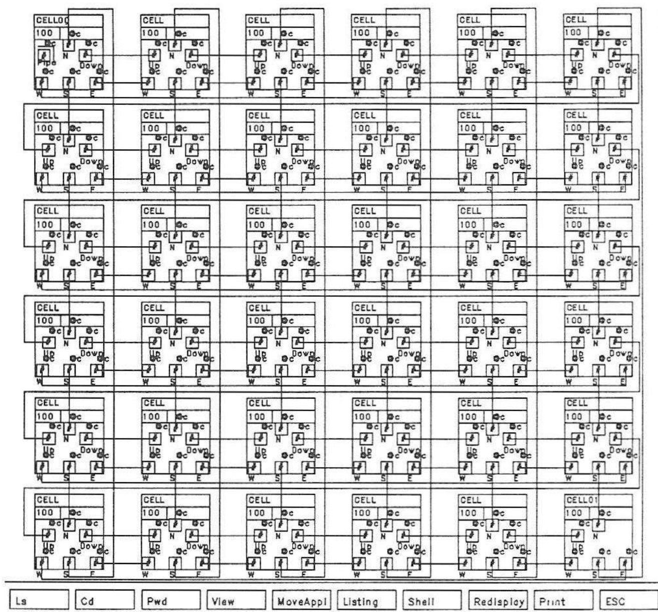


Bild 5

Das gesamte Programm besteht aus einer Kanaldeklaration, den SC für die einzelnen Transputer einschließlich Laufzeitsystem und den Placement- Anweisungen. In dem gewählten Beispiel wird das Gitter auf ein Drei- Transputer-System abgebildet. Das gesamte Programm besitzt 1 129 Zeilen, davon sind rund 300 Zeilen sequentieller Anteil, die der Benutzer mit einem Text- Editor definiert hat.

Der Rest entsteht durch Hinzufügen der erforderlichen Prozesse und Anweisungen, um 36 Prozesse mit jeweils sechs Kanälen auf drei Prozessoren ablaufen zu lassen. Da hierbei auf jedem Prozessor mehr als ein Prozeß liegt und jeder Prozeß sechs Kanäle benötigt, müssen die von der Hardware zur Verfügung gestellten Links von diesen Prozessen geteilt werden. Jede Nachricht muß mit einem besitzt. Dementsprechend muß der sequentielle Anteil an dieser Stelle leicht modifiziert werden.

Ein Ausschnitt des sequentiellen Teils von CELL ist in Bild 6 wiedergegeben. In der ersten Phase sendet jeder Prozeß seinen eigenen Zustand an seine vier Nachbarn und empfängt daraufhin den Zustand der vier Nachbarn. In der zweiten Phase wird der eigene Zustand aus den Zuständen der vier Nachbarn berechnet und die Pipeline zur Sammlung der Ergebnisse eingesetzt. In Abhängigkeit von dem Ergebnis wird entweder der eigene Zustand auf 1 oder auf 0 gesetzt. Die Anzahl der Schleifendurchläufe wird in der Initialisierung so gesetzt, wie es vom Benutzer angegeben worden ist.

Spectral erzeugt daraus ein Programm, das auf dem Host ausgeführt wird (EXE) und ein Programm, das auf dem parallelen Teil des Systems ausgeführt wird (PRG). Die Verminderung des Programmieraufwands wird am deutlichsten bei dem Programm für den parallelen Teil.

```
...  DECLARATIONS
SEQ
  ...  INITIALIZATION
  Up.in ? Size ; Data.array
  WHILE Data.array[0] <> EndOfData[0]
    SEQ
      Down.out ! Size ; Data.array
      Up.in    ? Size ; Data.array
  Down.out ! Size ; EndOfData
  SEQ LoopCounter = 0 FOR NumberOfLoops
    SEQ
      PAR
        N.out ! Size ; OwnState
        S.out ! Size ; OwnState
        W.out ! Size ; OwnState
        E.out ! Size ; OwnState
        N.in  ? Size ; NorthState
        S.in  ? Size ; SouthState
        W.in  ? Size ; WestState
        E.in  ? Size ; EastState
      NumberOf1s := NorthState[0] +
                   (SouthState[0] +
                    (WestState[0] +
                     EastState[0]))
      IF
        OwnState[0] = 0
          IF
            NumberOf1s = 1
              OwnState[0] := 1
            TRUE
              SKIP
        OwnState[0] = 1
          IF
            NumberOf1s < 1
              OwnState[0] := 0
            NumberOf1s > 1
              OwnState[0] := 0
            TRUE
              SKIP
        TRUE
          SKIP
      Up.out ! Size ; OwnState
      Down.in ? Size ; Data.array
      WHILE  Data.array[0] <> EndOfData[0]
        SEQ
          Up.out  ! Size ; Data.array
          Down.in ? Size ; Data.array
      Up.out ! Size ; EndOfData
```

Bild 6

Kopf versehen werden, um anzuzeigen, für welchen Prozeß und welchen Kanal sie bestimmt ist. Es wird automatisch ein Laufzeitsystem hinzugebunden, das diese Aufgabe erfüllt. Durch die Aufrufe der einzelnen Prozesse wird definiert, wie die Beziehung zwischen Kanälen und Links aussieht. Der Benutzer ist von diesen Aufgaben befreit und ist nur dafür verantwortlich, daß die Send- und Receive-Anweisungen in seinem Programm die Funktionen erfüllen, die notwendig sind, um das Problem zu lösen.

Eine weitere Leistung von Spectral ist, daß eine sehr gute Zuordnung von Prozessen zu Prozessoren gefunden wird. Experimente haben gezeigt, daß gerade bei irregulären Prozeßgraphen, auch wenn sie nur rund 50 Prozesse umfassen, die Maschine eine bessere Abbildung findet als der Mensch - und das in wesentlich kürzerer Zeit. Diese Information fließt in die Generierung der PLACED PAR-Anweisungen ein, wo diese Abbildung entsprechend definiert wird.

Falls der Benutzer die Generierung für ein anderes Transputer-System wünscht, braucht er nur eine andere Anzahl von Transputern anzugeben, auf denen das Problem gerechnet werden soll. Dadurch ist die leichte Portabilität der Software von kleinen bis großen Systemen gegeben. Der Portierungsaufwand ist nahezu null.

Bei der Generierung wird noch eine Reihe weiterer nützlicher Informationen erzeugt. So wird für elektronisch konfigurierbare Computer-Systeme die Konfiguration geschaltet, es werden Protokolldateien erzeugt und es erfolgt die automatische Einbindung in das TDS-File-System. Gegebenenfalls werden Ein- und Ausgabedateien zusätzlich mit eingebunden, so daß dafür keine Interaktionen des Benutzers mit dem TDS erforderlich sind.

Zusammenfassung

Spectral dient dazu, parallele Software für Transputersysteme zu entwickeln. Der Benutzer kann unabhängig von der gerade zur Verfügung stehenden Hardware sein paralleles Programm entwickeln; Beschränkungen durch die Hardware (Anzahl der Transputer oder Links) spielen dabei keine Rolle. Die grafische Benutzeroberfläche ermöglicht es, das Prozeßnetz so zu spezifizieren, wie es sich der Benutzer - in Gedanken - vorstellt. Die Anweisungen CHAN, PROCESSOR, PLACE und PLACED PAR des Occam2-Sprachumfanges braucht der Benutzer nicht anzuwenden. Er muß lediglich die sequentiellen Teile der einzelnen Prozesse einschließlich der Send- und Receive-Anweisungen definieren. Daraus wird automatisch - für eine beliebige Anzahl von Transputern - ein ausführbares paralleles Programm erzeugt. Unter anderem werden auch die hardware- spezifischen Anweisungen vom System generiert.

Insgesamt wird der Codierungsaufwand um mehr als die Hälfte reduziert. Das Laufzeitsystem, das die mehrfache Benutzung der Links und das Multiprocessing auf einem Transputer unterstützt, wird für jede Anwendung spezifisch generiert. Dazu werden nur auf den Prozessoren zusätzliche Prozesse gelegt, wo es die Hardware unumgänglich macht. Wenn einem Prozessor nur ein Prozeß zugeordnet wird, der nur vier Links benötigt, werden keinerlei zusätzliche Prozesse auf diesen Prozessor gelegt. Das Laufzeitsystem ist insoweit minimal, da es nur dort eingreift, wo Hardware-Beschränkungen durch die Software aufgehoben werden sollen.

Gegenwärtig wird Spectral in der Gesellschaft für Mathematik und Datenverarbeitung (GMD) und bei verschiedenen Kooperationspartnern eingesetzt, um prototyphafte Anwendungen für Transputersysteme zu erstellen.

Dr. Siegfried Streitz

Konfigurierbare Transputerarchitekturen

Mit dem hier vorgestellten Konzept wird ein neuer Weg zur Leistungssteigerung beschritten. So wurde bisher weltweit versucht, die Geschwindigkeit des einzelnen Prozessors zu verbessern oder durch Vektorisierung einen gewissen Grad an Parallelität zu erreichen. Als Prozessor wird der momentan leistungsstärkste am Markt befindliche Transputer eingesetzt.

"Leistung wird einfach" - mit dieser Aussage wirbt ein junger Computerhersteller aus Aachen für seine Produkte. Im Frühjahr 1988 konnte er bereits seine zweite Rechnergeneration präsentieren. Es handelt sich hierbei um die Supercluster-Serie, ein auf Transputern basierender Parallelrechner der MIMD-Klasse (Multiple Instruction - Multiple Data).

Nach nur einem halben Jahr Entwicklungszeit entstand die Supercluster-Serie, deren Basissystem - das Supercluster Modell 64 - aus 64 Prozessoren besteht. Als Prozessor wird der Transputer T800 eingesetzt. Vier Basissysteme bilden mit entsprechender Peripherie das nächstgrößere "Modell 256". Theoretisch können unbegrenzt viele dieser Bausteine zu beliebig großen Systemen zusammengeschaltet werden.

Mit diesem Ansatz wird ein neuer Weg zur Leistungssteigerung verfolgt. Die Firma Parsytec sowie einige Firmen aus den USA (FPS, Intel), Großbritannien (Meiko) und der Bundesrepublik (iP-Systems, P1, Suprenum) gehen durch den Einsatz vieler parallel arbeitender Prozessoren einen Schritt in Richtung "massive Parallelität". Jeder Prozessor verfügt über einen lokalen Speicher. Über ein Kommunikationssystem werden Nachrichten ausgetauscht, so daß sich Prozesse synchronisieren und gemeinsam eine Gesamtaufgabe bearbeiten können.

Durch dieses Prinzip werden Leistungssteigerungen möglich, die von heutigen (sequentiellen) Großrechnern nicht mehr erbracht werden können. Der Wunsch des Superrechners am Arbeitsplatz wird Wirklichkeit.

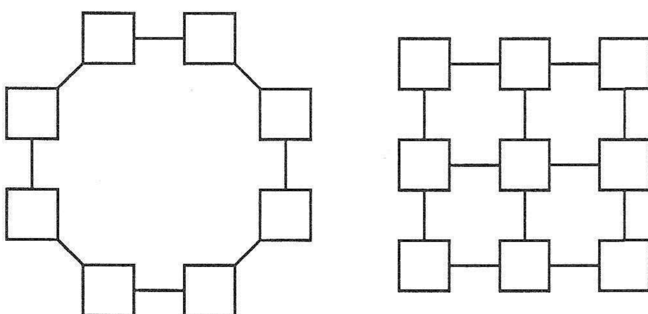
Parsytec kamen Erfahrungen aus der letzten Rechnergeneration zugute. Die 1987 vorgestellte Megaframe-Serie besteht aus Transputern des Typs T414, die über steckbare Kabel miteinander verbunden werden. Bis zu 40 Transputer und ein I/O-System (Input/Output-System) bilden eine Systemeinheit, die über Erweiterungseinheiten um weitere Prozessoren aufgerüstet werden kann.

Die Hardware

Die Supercluster-Serie basiert auf zwei neuen Bausteinen der Firma Inmos: den Transputern T800/T801 mit eingebauter Gleitkomma-Einheit und dem Kommunikationsprozessor C004. Der C004 realisiert eine elektronisch schaltbare Verbindung zwischen Paaren von Transputern und wurde für diesen Zweck speziell entwickelt.

Als Prozessor werden zwei verschiedene Transputertypen verwendet. Der T800 mit 20-MHz-Takt und der T801 mit 30-MHz-Takt. Sie sind jeweils mit einem 4-MByte- Hauptspeicher ausgestattet und leisten 1,5 MFlops (2,25 MFlops = Millionen Gleitkomma-Operationen pro Sekunde).

Jeder Prozessor trägt mit vier seriellen Hochgeschwindigkeitskanälen (20 MBit/s) zur Gesamtkommunikation bei.



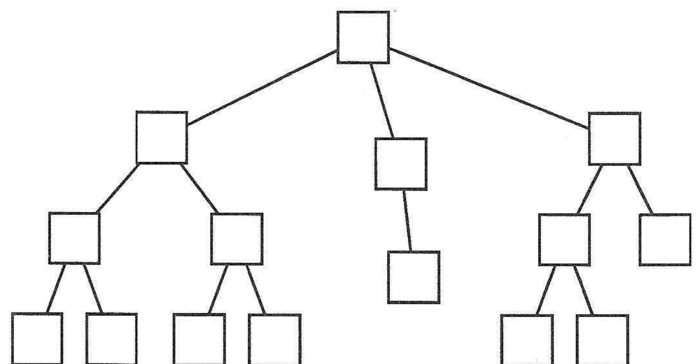


Bild 1: Transputer-Topologien

Somit wächst bei steigender Anzahl der Prozessoren auch die Kommunikationsleistung des Systems. Die Engpaßgefahr, die busgekoppelte Systeme besitzen, entfällt. So ist eine praktisch unbegrenzte Ausbaubarkeit der Verarbeitungsleistung möglich.

Sowohl mit den steckbaren Verbindungen der ersten Generation als auch mit der elektronisch schaltbaren Kommunikationseinheit sind beliebige Prozessor- Topologien (mit maximal vier Links pro Transputer) realisierbar. Je nach Anwendungsproblem werden die Prozessoren zu einem Ring, einem Gitter, einem Baum oder einer beliebigen anderen Struktur miteinander verbunden. Diese Topologie ist vom Programmierer frei wählbar und wird automatisch unter Betriebssystemkontrolle hergestellt. Dies kann auch unabhängig voneinander für mehrere Benutzer gleichzeitig geschehen.

Technisch gewährleistet wird die anwendungsspezifische Verschaltung von Transputern zur gewünschten Topologie durch sogenannte "Network Configuration Units" (NCU). Eine NCU besitzt 96 Eingänge, an die die Links von bis zu 24 Transputern angeschlossen werden können. Diese werden paarweise miteinander verbunden, so daß beliebige Konfigurationen erzeugt werden können.

Ein "Computing Cluster" besteht aus 16 Transputern und enthält eine NCU. Diese gestattet eine beliebige Verschaltbarkeit aller 16 Transputer untereinander. Die verbleibenden 32 freien Anschlüsse der NCU werden dazu verwendet, aus mehreren Computing Cluster größere Systeme aufzubauen. Die Verschaltung dieser Systeme mit den aus PC herausführenden Kanälen wiederum ist Aufgabe von zwei übergeordneten NCU.

Das "Modell 64" ist das Basissystem der Supercluster-Serie. In dieses Basissystem kann Massenspeicher integriert werden. Zum Anschluß peripherer Geräte, wie Bildschirmgeräten, PC oder grafische Workstations stehen 32 Anschlüsse zur Verfügung. Zusätzlich werden Netzanschlüsse (Ethernet) angeboten.

Das nächstgrößere "Modell 256" besteht aus vier Einheiten zu je 64 Prozessoren. Seine Rechenleistung reicht mit 384 MFlops für den T800 beziehungsweise 576 MFlops für ein T801-Modell bereits in den Bereich der Supercomputer hinein.

Mehrere dieser Modelle können zu größeren Systemen zusammengeschaltet werden. Dem schrittweisen Systemaufbau sind keine Grenzen gesetzt.

Die Software-Umgebung

Programmierbar sind die Systeme in der für Transputer entwickelten parallelen Programmiersprache Occam2. Zu-

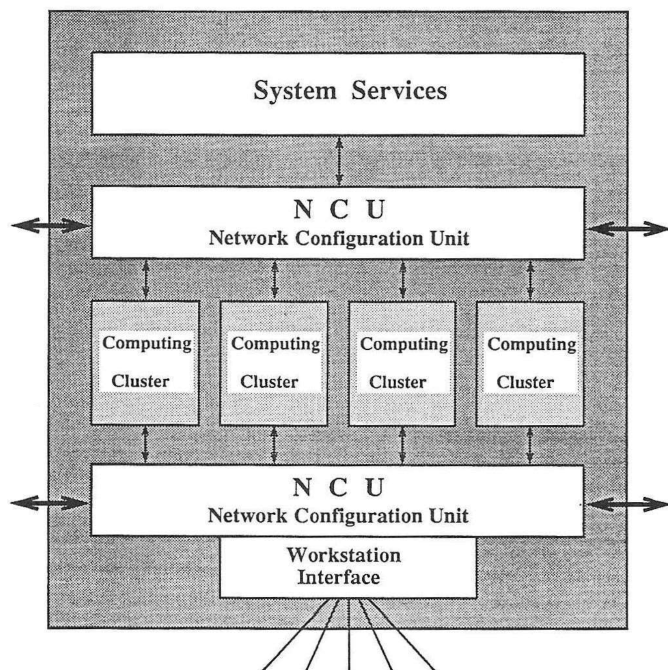


Bild 2: Computing Cluster (links) und Modell 64 (rechts)

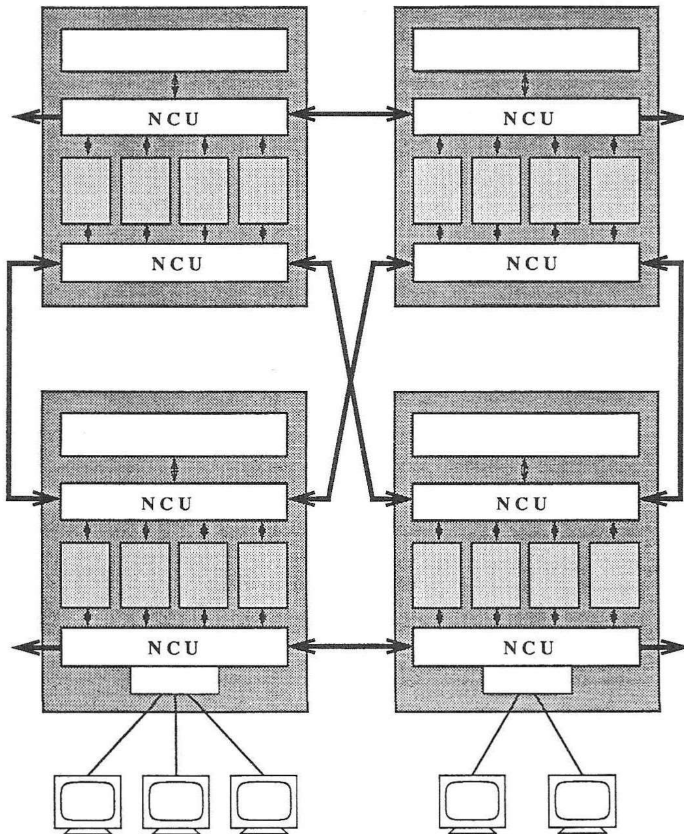


Bild 3: Supercluster-Modell 256

sätzlich existieren Compiler für eine Reihe von Sprachen wie Pascal, C, Fortran oder Prolog.

Als Betriebssystem wird das Multiprozessor-Betriebssystem Helios zur Verfügung stehen. Kern der Betriebssoftware der Supercluster-Serie ist der Network Configuration Manager. Er teilt die Prozessoren des Systems auf mehrere Benutzer auf und realisiert so einen Multiuser-Betrieb. Für jeden Benutzer wird seine individuelle Verschaltung der ihm zugewiesenen Transputer realisiert.

Mehrere Arbeitsplätze werden über das Workstation-Interface angeschlossen. Damit an jeder Station ein ungestörter Betrieb stattfinden kann, muß jeder Benutzer die Anzahl der benötigten Prozessoren und deren Verbindungsstruktur festlegen. Der Network Configuration Manager sucht daraufhin die angegebene Anzahl freier Transputer und verschaltet diese für die Dauer des Anwendungsprogrammes in der gewünschten Art und Weise.

Bei der Gesellschaft für Mathematik und Datenverarbeitung (GMD) wird die Programmierumgebung Muppet entwickelt, die eine komfortable Programmierung paralleler Algorithmen ermöglicht. Sie besteht aus einem grafischen Spezifikationswerkzeug Spectral, einem Transformationssystem Trollo zur Code-Optimierung und einer automatischen Konfigurierung, die auf einem allgemeinen Mapping-Werkzeug basiert.

Die Entwicklung von sowohl seriellen als auch parallelen Programmen wird üblicherweise in der Form vorgenommen, daß die logische Programmstruktur in einer grafischen Darstellung (Flußdiagramm, Nassi-Schneidermann, ...) definiert wird. Erst anschließend wird diese vom Programmierer in Programmcode umgesetzt.

Mit Hilfe des Spectral kann der Programmierer sein Programm direkt in einer grafischen Form erstellen. Auf einem Grafikbildschirm zeichnet er für jeden Prozeß eine eigene Box. Diese Boxen werden untereinander durch Linien verbunden, so daß ein Prozeßgraph entsteht, der der Struktur des parallelen Programmes entspricht. Ein Replikationsmechanismus erlaubt das Erzeugen von beliebig vielen Kopien.

Durch Anklicken können die einzelnen Prozesse geöffnet werden. Es erscheint ein Textfenster, das den lokalen Programmcode des ausgewählten Prozesses enthält. Dieser Code ist in Occam2 geschrieben und beschreibt die Aktionen des ausgewählten Prozeßtyps. In diesen lokalen Programmen kann der Programmierer die Kommunikationskanäle zu Nachbarprozessen benutzen.

Das Transformationssystem Trollo nimmt eine Optimierung dieser grafischen Programmspezifikation vor. Durch Teilung oder Verschmelzung von Prozessen oder durch Reorganisation der Kommunikationsbeziehung wird versucht, die Laufzeit des Programmes zu verbessern. Dies geschieht unsichtbar für den Programmierer.

Nach Fertigstellung der grafischen Spezifikation aller Prozesse wird das parallele Programm dann generiert. Aus den grafischen Komponenten wird ein Programmrahmen erzeugt, in den die Programmcodes der einzelnen Prozesse eingefügt werden.

Um dieses Programm auf der Hardware ablaufen zu lassen, ist eine Konfigurierung notwendig. Sie geschieht

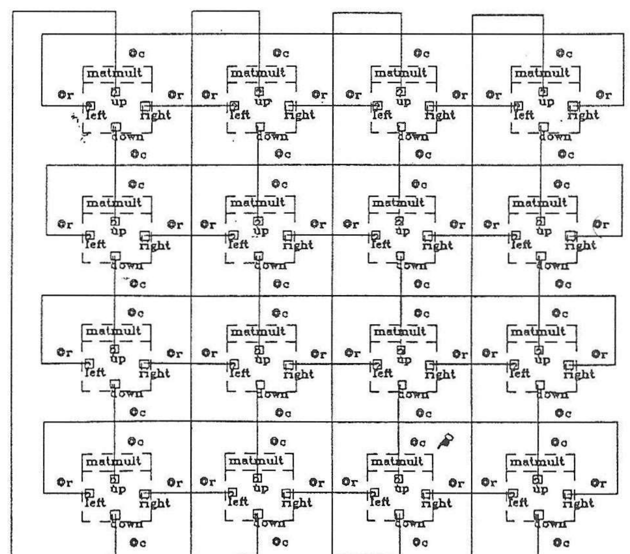


Bild 4: Grafische Programmspezifikation mit Spectral

ebenfalls automatisch, so daß das grafisch spezifizierte Programm ohne weitere Benutzereingriffe voll ablauffähig ist.

Das Mapping - die Entscheidung, welcher Prozeß auf welchem Prozessor abläuft - erfolgt automatisch. Es wird folgendermaßen bestimmt: der Prozeßgraph wird in vier Teile unterteilt, die jeweils einem Computing Cluster zugeordnet werden. Innerhalb jedes Clusters werden diese Prozeßmengen weiter in 16 Teile aufgeteilt und jeweils einem Prozessor zugeordnet. Ziel ist es, daß jeder Transputer einen möglichst gleich großen Anteil am Gesamtprogramm erhält und dabei die Kommunikation zwischen ihnen gering gehalten wird. Auf der Grundlage dieser Prozeß-Prozessor-Zuordnung wird der Schaltplan für die NCU berechnet.

Weil jeder Transputer nur vier Links besitzt, gelten für die Prozeßstrukturen Einschränkungen. Prozesse, die auf einem Prozessor ablaufen, dürfen zusammen maximal vier externe Kanäle besitzen. Kommunizierende Prozesse müssen entweder auf demselben Transputer ablaufen oder auf Transputern, die durch ein Link miteinander verbunden sind.

Um diese Einschränkungen zu überwinden, werden spezielle Multiplexer- und Router-Prozesse erzeugt. Diese werden an den Stellen eingefügt, an denen die Beschränkungen verletzt werden. Multiplexer (M) ermöglichen, daß mehr als vier (logische) Kanäle einen Transputer verlassen können. Zwei oder mehrere dieser Kanäle werden über einen gemeinsamen Link realisiert.

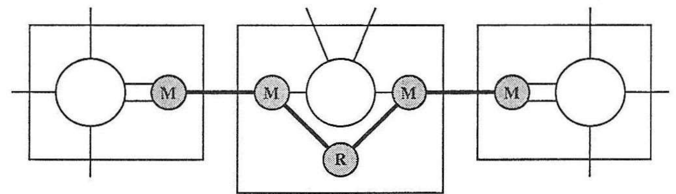


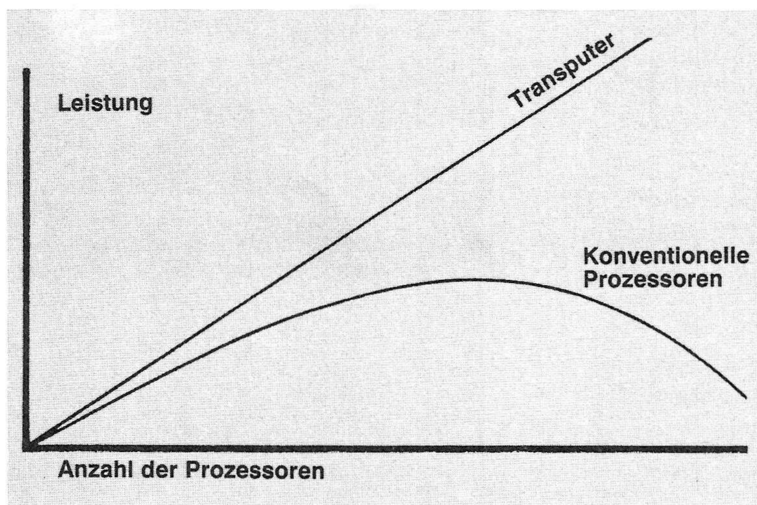
Bild 5: Multiplexer- und Router-Prozesse

Router-Prozesse (R) erlauben die Kommunikation zwischen Prozessen, die auf nicht benachbarten Prozessoren ablaufen. Hierzu wird eine Nachricht über den kürzesten Weg über eine Kette von Transputern transportiert.

Mit diesen Werkzeugen, die zusammen die Programmierumgebung Muppet bilden, ist eine komfortable Programmierung der Supercluster-Serie möglich. Programme können in einer nur vom Problem abhängigen Struktur entwickelt werden, ohne daß Rücksichten auf die zugrundeliegende Hardware genommen werden müssen. Sowohl die Generierung und Optimierung des Codes als auch die Konfiguration des Systems erfolgt automatisch.

Durch das Zusammenspiel der leistungsfähigen Hardware der Supercluster-Serie mit der komfortablen Programmierumgebung Muppet wird relativ einfach eine hohe Rechenleistung erzielt.

Ottmar Krämer



In konventionell aufgebauten, busgekoppelten Systemen mit mehreren Prozessoren kann die Gesamtleistung nicht über einen bestimmten Punkt hinaus gesteigert werden. Transputer besitzen eine Architektur, die theoretisch unbegrenzte Leistungssteigerung zuläßt.

Der Amiga 2000 und seine Transputer-Implementierung

Die Transputer von Inmos stellen eine Chipfamilie dar, die neben dem Prozessor einen internen Speicher sowie Kommunikationseinrichtungen, sogenannte Links, zur Verfügung stellen und somit als Mikrocomputerchips bezeichnet werden können. Um diese Bausteine in einem anderen Rechnersystem benutzen zu können, sind nur ein externer Speicher und ein Interface für den entsprechenden Hostrechner hinzuzufügen. über die Links ist eine einfache Erweiterung zu einem Multiprozessorsystem möglich. Die Links werden im Voll-Duplex-Betrieb über eigene DMA-Kanäle gesteuert und setzen die Verarbeitungsgeschwindigkeit auch bei maximaler Kommunikationstätigkeit nicht herab.

Im folgenden soll eine Implementierung eines Transputer in die Amiga 2000- Umgebung beschrieben werden. Aufgrund seiner offenen Systemarchitektur ist es im Prinzip nicht schwierig, den Amiga mit zusätzlichen Prozessoren auszustatten (z.B. MS-DOS-Bridgeboards), die nicht der Motorolawelt angehören.

Überblick

Der Amiga 2000 basiert auf einer 68000er CPU und kann mit Hilfe von Einsteckkarten aufgerüstet werden. Die Steckkartengröße beträgt 337x114mm (identisch zum PC). Die Besonderheit am Amiga Expansionsbus ist der Autokonfigurationsmechanismus.

Auf einer Transputerkarte für den Amiga 2000 müssen daher die folgenden Komponenten vorhanden sein.

Die Autokonfigurationslogik wird für das Autokonfigurationsprotokoll benötigt und puffert die Daten sowie Steuersignale für das Linkinterface. Die parallelen Daten des Amiga werden vom Linkinterface in die seriellen Daten für den Transputer umgesetzt. Zusätzlich werden hier die Steuersignale für den Roottransputer erzeugt. Der Transputer bedient mit dem External Memory Interface (EMI) seinen externen Speicher. Über die Aalener Linkverbindungen können weitere Transputer mit dem Roottransputer gekoppelt werden.

Autokonfiguration und Linkinterface

Autokonfiguration ist ein Standard-Hardware-Protokoll, durch das sich Einsteckkarten verschiedener Hersteller automatisch den IO- und Adreßraum teilen. Mit Autokonfiguration ist es nicht notwendig, DIP-Schalter oder Adreß-Jumper zu setzen, wenn die Konfiguration des Amiga 2000 verändert wird.

Um das Autokonfigurationsprotokoll zu unterstützen, müssen Amiga- Einsteckkarten einen kleinen ROM-Speicherbereich besitzen. Anhand dieses Speicherbereiches kann festgestellt werden, welcher Kartentyp eingebaut ist, wieviel Speicher benötigt wird und welche Treibersoftwa-

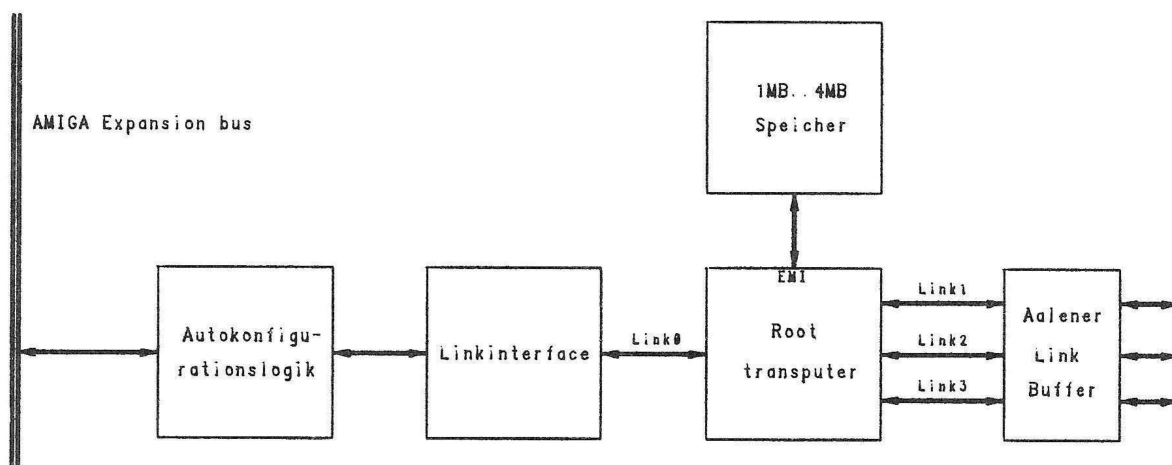


Bild 1: Blockschaltbild

re, falls notwendig, benutzt werden muß. Mit Amiga-DOS-Version 1.3 können auch Autoboot-ROM auf der Karte vorhanden sein.

Beim Einschalten des Systems wird jede Karte abgefragt und eine Speicheradresse, je nach Anfrage der Karte, zugeordnet. Alle Karten werden unter Kontrolle des Betriebssystems in eine Systemtabelle für Erweiterungskarten eingetragen. Wird zum Beispiel eine Speicherkarte erkannt, erhöht sich automatisch der frei verfügbare Speicher im Amiga. Wenn das System hochgefahren ist, lädt das Programm Binddrivers den Software-Treiber jeder eingebauten Karte. Ist eine Karte nicht installiert, wird einfach der Treiber nicht geladen.

Für den Endanwender ist dies natürlich eine feine Sache, da er sich in diesem Fall nicht mit Mäuseklavieren oder Jumpern den Adreßbereich seines Rechners absuchen muß, um einen freien Speicherbereich zu finden.

Die Kontrollsignale für den Linkchip werden vom Linkinterface erzeugt, ebenso die Steuersignale RESET, ERROR und ANALYSE für den Roottransputer. Das Bindeglied zwischen Transputer und Amiga Expansionsbus ist ein C012 Linkadapter von Inmos. Dieser spezielle Baustein der Transputerfamilie setzt die parallelen Daten eines herkömmlichen Mikroprozessors in die spezielle bitserielle Linkübertragung der Transputer um. Der Linkadapter ist fest mit LINK0 verdrahtet, so daß der Transputer direkt vom Amiga gebootet werden kann.

Roottransputer

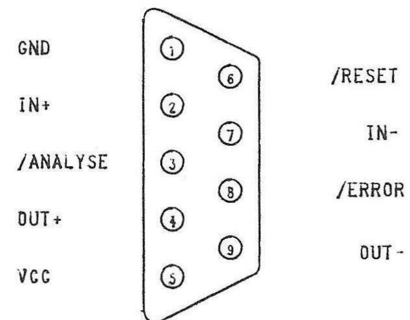
Die Amiga-Transputerkarte kann mit bis zu 4 MByte Speicher ausgerüstet werden. Die Erzeugung der RAM-Steuersignale übernimmt das External Memory Interface (EMI) des Transputers. Das EMI stellt fünf programmierbare Statussignale zur Verfügung, die nach jedem Reset selbsttätig programmiert werden. Aus diesen Steuersignalen lassen sich alle notwendigen Signale für einen DRAM-Bereich erzeugen. Die Zugriffszeit ist abhängig von den verwendeten Speicherbausteinen und beträgt im Regelfall 200ns. Die Grundausstattung wird 1 MByte betragen, so daß durch Bestücken der freien Fassungen nach Eigenbedarf die Speichergröße bis auf insgesamt 4 MByte erweitert werden kann.

Linkverbindungen

Eine Interessensgruppe der DOIT (Deutsche OCCAM-Interessengemeinschaft der Transputeranwender) hat den Aalener Link Connector als Standardverbindung von Transputersystemen verschiedener Hersteller vorgeschlagen. Dabei wurde auf eine kostengünstige und gleichzeitig sichere Datenverbindung Wert gelegt.

Als mechanische Konstruktion wird ein 9poliger Sub-Min-D Steckverbinder verwendet. Die Signale Link_In und Link_Out werden nach RS422-Norm als komplementäre

Sub-Min-D Steckverbinder



Masterbuchse fuer DOWN oder SUB-System

Bild 2: Aalener Link

Signale übertragen. Die Systemsignale werden als TTL-Pegel gepuffert, weil sie quasi statisch sind und keine gravierenden Übertragungsfehler auftreten können. Die RS422-Norm gilt strenggenommen nicht für Übertragungsraten von 20 Mbit/s, das heißt wer sein Transputersystem auf der sicheren Seite betreiben will, sollte sich auf Datenraten von 10 Mbit/s beschränken. Bei Linkverbindungen über 10 m sollte die Übertragungsrate grundsätzlich 10 Mbit/s betragen.

Die übertragungsrichtung wird durch die Ausführung als Buchse (DOWN- oder SUB- System) oder Stecker (UP-System) automatisch bestimmt.

Die drei übrigen Links werden auf das Anschlußblech an der Rückseite des Amiga 2000 nach außen geführt. Sie sind als 9polige Sub-Min-D-Buchsen ausgeführt, und sind entsprechend der Aalener Link "Norm" von einem DOWN- oder SUB-System benutzbar.

Bestückungsplan

Bild 3 zeigt den derzeitigen Bestückungsplan. Am unteren Rand ist der Amiga- Busanschluß (J2) zu sehen. Darüber befindet sich das Linkinterface und die Autokonfigurationslogik. Den Transputer stellt U3 dar, der standardmäßig ein T414-15 sein wird. Es kann aber auch ein T800-25 eingesetzt werden. In der linken oberen Hälfte befindet sich der Speicherbereich. Links davon sitzen die Treiber für die Speicher. An der rechten Seite werden die Sub-Min-D-Buchsen befestigt. Dieser Teil kann später vom Endanwender auch abgetrennt werden, um eine Linkpufferung nach eigenen Anforderungen vorzunehmen.

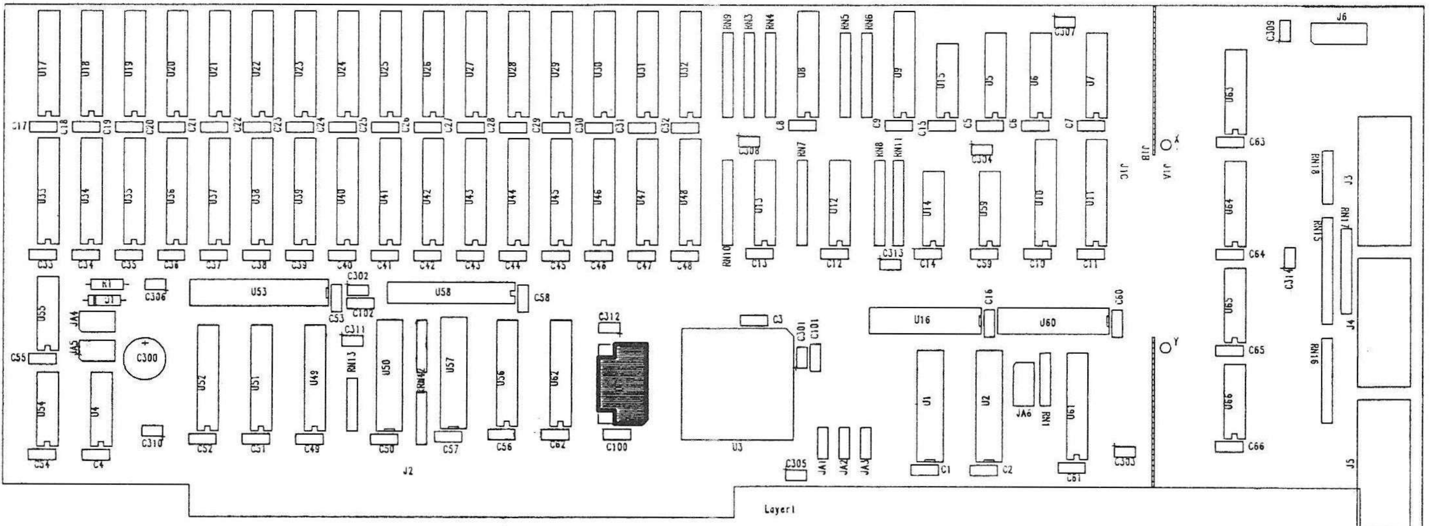


Bild 3: Bestückungsplan

Insgesamt bietet der Amiga eine sehr interessante Alternative für Transputeranwendungen zum herkömmlichen Industriestandard.

Heiko Czerwinski, Wolf Schmidt

Betriebssystem

Als Betriebssystem auf dem Transputern wird Helios angeboten. Das Helios wird von Perihelion entwickelt, die auch die ersten Versionen des Amiga-DOS geschrieben haben. Als windoworientiertes Multitasking-Betriebssystem bietet Amiga-DOS eine optimale Ausgangslage als Frontend für ein Transputernetzwerk. Helios ist ein verteiltes Betriebssystem, das heißt auf jedem Transputer wird "ein" Helios laufen. Über eine Windowshell kann jedem Transputer ein eigenes Fenster zugeordnet werden, wobei mit der Amiga-Maus die Fensterparameter manipuliert werden können.

Alle Ein- und Ausgaben von der Tastatur, Bildschirm oder Festplatten werden von einem Server übernommen, der als eigene Task auf dem Amiga im Hintergrund läuft.

Als Zugabe ohne Aufpreis fällt ganz nebenbei auch ein Netzwerk für mehrere Amigas ab. Sie werden einfach über die Linkverbindungen der Transputerkarten miteinander verschaltet. Eine getrennte Netzwerksoftware ist nicht notwendig, weil das Linkprotokoll bereits Bestandteil des Betriebssystems Helios ist.

Ausblick

In der Amiga-Expansionsarchitektur sind auch Steckplätze nach dem Industriestandard vorhanden. Bild 4 zeigt diesen Teil des Amiga 2000- Mainboards. Diese Steckplätze können natürlich auch mit Transputerkarten belegt werden. Die Steckplätze dienen dabei nur der Spannungsversorgung der Karten. Die Datenübertragung findet dann ausschließlich über die Links statt. Bei Verwendung von vierfach Transputerkarten können maximal 4 Steckplätze benutzt werden, daraus ergibt sich eine Anzahl von 17 Transputern im Amiga 2000.

Buchtips

- Technical Reference Manual A500/A2000
- TN 09 INMOS Limited
- Datasheet T414, T800
- DOIT Newsletter 1/88

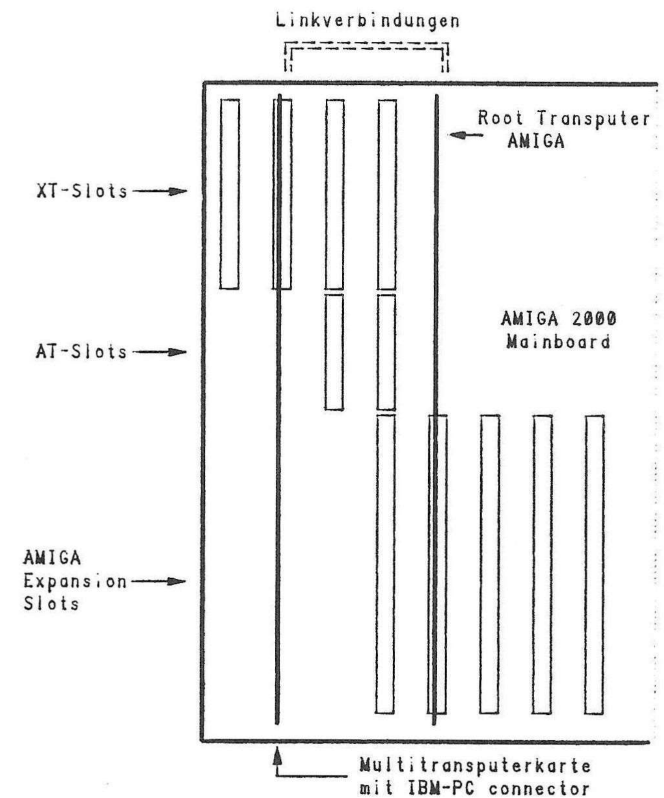


Bild 4: Mehrtransputersystem Amiga 2000

Atari-Transputer-Workstation unter Helios

Drei Jahre nach der Markteinführung des Transputers hat Atari, auf diesem Chip aufbauend, eine leistungsfähige Grafik-Workstation entwickelt. Die Atari Transputer-Workstation (ATW) zeichnet sich durch ihre grafische Leistungsfähigkeit und ein modulares Konzept aus. Durch Zustecken sogenannter Farmcards läßt sich die Rechenleistung nahezu beliebig erhöhen. Das Gerät (multitasking/multiuserfähig) ist offen konzipiert: Praktisch alle Schnittstellen (Hardware und Software) sind standardisiert. Die Software ist zukunftsorientiert ausgelegt: Helios ist weitgehend kompatibel zu Unix; die grafische Ausgabe erfolgt über X-Windows V11.

Die Komponenten der Workstation sind im Blockschaltbild dargestellt und werden im folgenden erläutert.

Die Grundkonfiguration

Die Grundkonfiguration enthält einen Transputer T800-20. Im Baustein sind 4 KByte RAM, eine 32-bit-CPU, eine 2,25 MFlop/s FPU, eine Echtzeitlogik und vier serielle Kommunikationskanäle mit hohem Datendurchsatz (Links) integriert. Der Prozessor entspricht der RISC-Architektur und unterstützt das Abarbeiten paralleler Prozesse. Die Rechenleistung eines Chips beträgt 10 Mips. Transputer lassen sich durch ihre Links zu Feldern beliebiger Konfiguration vernetzen.

Ein Farbblitter (genannt 'Blossom') unterstützt den schnellen Bildaufbau (z.B das Ausführen zweidimensionaler Blockverschiebeoperationen). Seine Funktionen sind den Anforderungen der X-Window-Routinen angepaßt. Ferner erzeugt er alle, für die RAM der Grundkonfiguration erforderlichen, Refreshsignale. Die Register des Blitters liegen im Adreßbereich des Transputers. Beide Schaltkreise, der Transputer und der Blitter, greifen über einen Hauptbus auf den Speicher zu.

Der Hauptbus (32 Bit Daten- und 32 Bit Adreßleitungen) koppelt die Erweiterungskarten, den Arbeitsspeicher und das Video-RAM. Der Arbeitsspeicher umfaßt 4 MByte (70 ns Zugriffszeit) und bildet die Erweiterung zum integrierten RAM des Transputers. Der Speicherbereich ist für spätere Aufrüstungen bezüglich Zugriffszeiten und Speichergröße vorbereitet.

Das Video-RAM (100 ns Zugriffszeit) umfaßt 1 MByte und ist dual port. In das eine Port wird in der Regel geschrieben und die gespeicherten Informationen aus dem anderen Port herausgeschoben. Durch diese Technik wird der Bus von den Videoinformationen entlastet.

Die Speicherformate sind über vier Video-Modi softwaremäßig einstellbar. Es kann zwischen 4, 8 und 32 Bit Tieferinformation pro Pixel unterschieden werden. Die Adreßdekodierung verwaltet die Zuordnung einer Pixelkoordinate zur physikalischen Speicheradresse. Die Adresse eines Pixels kann unabhängig von Video-Modus stets durch dieselbe Formel berechnet werden.

Eine spätere Erweiterung sieht pro Grundfarbe (Rot, Grün, Blau) einen eigenen Videospeicher vor, um die Zahl der Bitplanes für besonders professionelle Anwendungen weiter zu erhöhen.

Die digitalen, vom Video-RAM herausgeschobenen Pixelinformationen werden auf einer Videokarte in analoge RGB Signale umgesetzt. Die Ausgabe erfolgt auf einem Grafikschirm (z.B. NEC Multisync).

Ein für seine I/O Operationen optimierter Mega ST dient als Ein- /Ausgabeeinheit. Der Speicher umfaßt 512 KByte und ist auf 1 MByte aufrüstbar. Der Bus des Mega ST ist herausgeführt. Es steht ein Steckplatz für eine Erweiterungskarte zur Verfügung. Der Zugriff auf die integrierte 40-MByte- Platte und weiterer Massenspeicher erfolgt über eine SCSI-Schnittstelle. Ein neuer Baustein ('Morpheus') koppelt den 68000-Prozessor mit Transputerlink.

Die Ein-/Ausgabekarte verwaltet Peripheriegeräte, wie zum Beispiel ein weiteres Terminal, Laserdrucker oder auch selbstentwickelte Laborschaltungen über seine ST-kompatiblen Schnittstellen (RS-232, Centronics, Romport, Midi etc.).

Die Erweiterungen

Alle Erweiterungen setzen am Hauptbus der Grundkonfiguration an. Die Register der Erweiterungskarten sind in den Adreßbereich des Transputers 'gemapped'.

Eine Speicherkarte erweitert den RAM-Bereich auf bis zu 16 MByte. Bei einer späteren Umrüstung mit 4-Mbit-Chips kann der Arbeitsspeicher auf bis zu 64 MByte aufgerüstet werden.

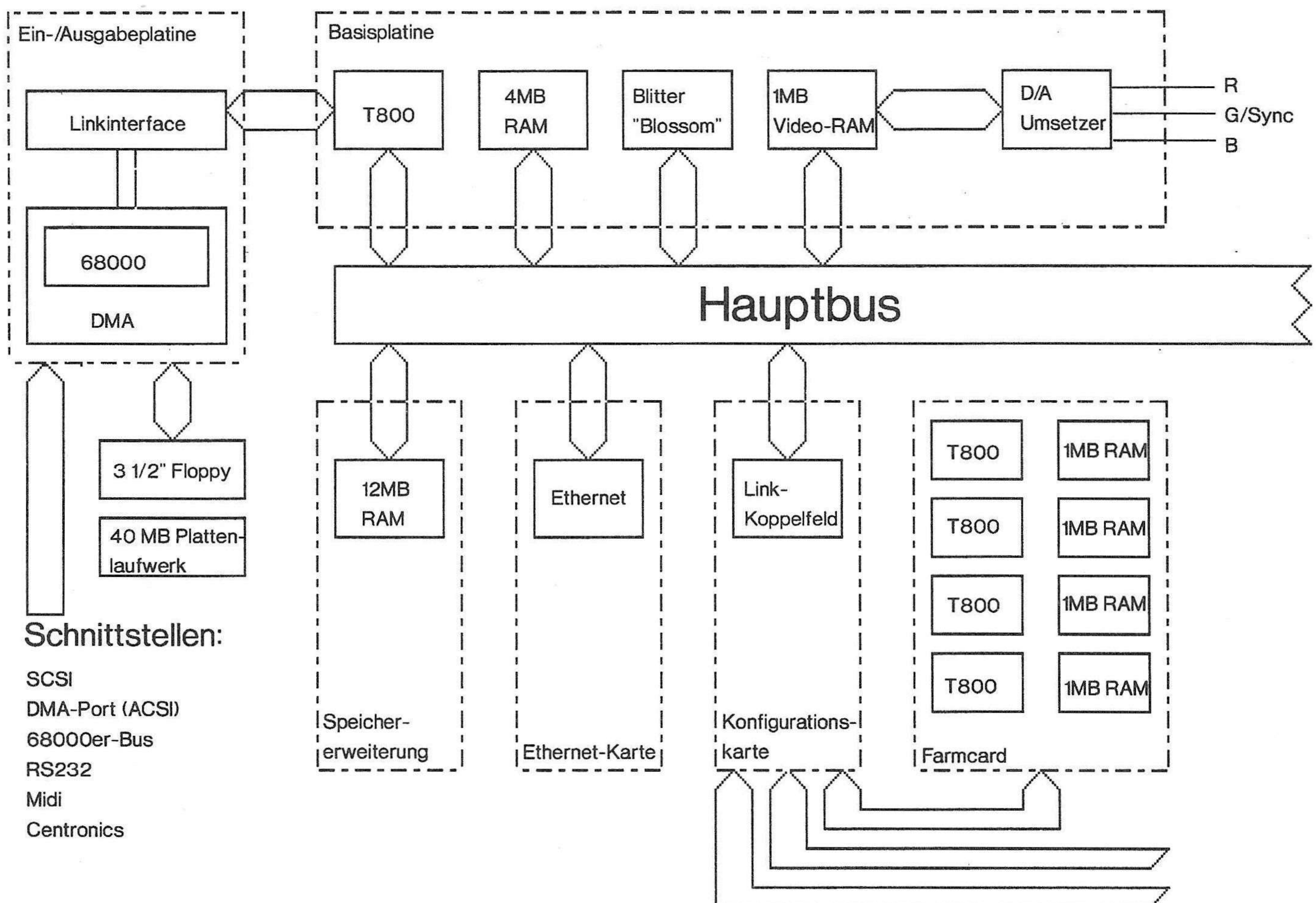


Bild 1: Blockschaltbild der Atari-Transputer-Workstation

Eine Ethernetkarte koppelt die Atari Transputer-Workstation mit anderen Rechnern über ein standardisiertes Netz.

Die Rechenleistung der ATW läßt sich durch Farmcards weiter erhöhen. Eine Farmcard enthält 1-4 Transputer mit wahlweise je 1-, 4- oder später auch 16- MByte-Speicher. Die Workstation nimmt bis zu 4 Farmcards auf. Im Blockschaltbild ist lediglich eine Farmcard dargestellt.

Die softwaremäßige Vernetzung der Transputer erfolgt durch eine Konfigurationskarte. Diese Erweiterung wird ebenfalls über den Hauptbus angesprochen. Ein Kreuzschienenverteiler (C004) bildet ein Koppelfeld für alle Transputer-Links.

Die Erweiterungsbox Polyhedron stellt weitere Rechenleistung zur Verfügung. Das Gerät nimmt zusätzliche 12 Farmcards auf. Eine Konfigurationskarte bestimmt das Transputernetzwerk von je 4 Farmcards. Ein Transputer (T212) und ein Kreuzschienenverteiler (C004) auf einer Konfigurationskarte unterstützen ein softwaremäßiges Konfigurieren des Transputernetzes. Die Gesamtrechenleistung beträgt bei vollem Ausbau 500 Mips.

Mehrere ATW lassen sich, in einer Ringstruktur vernetzt, zu einem Mehrbenutzersystem zusammenfassen. Jede Workstation bildet dann eine eigene Benutzerschnittstelle. Das Konzept von Helios akzeptiert auch eine Vernetzung von Transputersystemen verschiedener Hersteller (Atari, Inmos, Parsytec, Kuma, Commodore, Hema, Transfertech etc.).

Zur Zeit wird eine Video-Ein-/Ausgabekarte entwickelt. Diese Erweiterung ist auf den Einsatz im Videobereich zugeschnitten. Sie enthält einen Frame-Grabber (Bewegtbilddigitalisierer) zum Laden von Bildern aus einer Videoquelle in einen Speicher. Von der ATW generierte Grafiken werden als Videosignal ausgegeben.

Grundlagensoftware

Software-Entwicklern stehen alle 'etablierten' Hochsprachen zur Verfügung (C, Fortran, Pascal). Sie entsprechen dem ANSI-Standard. Bereits bestehende Software kann in die Heliosumgebung portiert werden. Auch Assembler-Programmierung wird durch entsprechende Werkzeuge unterstützt.

Zur Zeit wird der TDS-Server für die Helios-Umgebung umgeschrieben. Das TDS (Transputer Developement System) unterstützt das Programmieren in der Hochsprache Occam. Diese Transputerentwicklungsumgebung läuft auf Transputern ab und umfaßt einen Editor, Disassembler, Debugger, Compiler etc. Der Server stellt die Hardware-Ressourcen und die Benutzerschnittstelle zur Verfügung.

Occam ist die derzeit einzige Sprache, die ein paralleles Programmieren in einem Transputernetz vollständig, aus einem Programm heraus, unterstützt. Da Occam mit dem Betriebssystem Helios nicht sinnvoll zusammenarbeitet, wird das TDS in einem autonomen, betriebssystemfreien Teilnetz betrieben. Der Server bildet die Schnittstelle zwischen beiden Software-Systemen.

Voneinander unabhängige, rechenintensive Programmteile sind vorzugsweise in Occam zu programmieren.

Anwendersoftware

Zunächst kann die ATW aus dem großen Pool der Unix-Software schöpfen. Das Betriebssystem Helios unterstützt das Portieren von Unix-Programmen auf die ATW. Auf lange Sicht werden zunehmend mehr Anwenderprogramme verfügbar sein, die gezielt die Vorteile der ATW nutzen.

Derzeit wird Software für viele Anwendungsbereiche entwickelt. Die Entwicklertätigkeit reicht von dreidimensionalen Grafikeditoren über Echtzeitsimulatoren bis hin zu CIM-Anwendungen.

Zwei kurze Beispiele:

Für das Verständnis der Beispiele werden elementare Kenntnisse des Betriebssystems Helios und der Sprache C vorausgesetzt. Die Verwaltung der Hardware-Ressourcen erfolgt von Helios aus. Die Peripheriegeräte werden als virtuelle Dateien dargestellt.

```
% cp /rs232/default /centronics/default &
```

Die obige Kommandoanweisung unter der Helios-Shell überträgt alle über die RS-232-Schnittstelle eingehenden Zeichen direkt zum Drucker. Das & gibt an, daß der vom Benutzer aktivierte Prozeß im Hintergrund ausgeführt wird.

Das zweite Beispiel beschreibt kurz die Vorgehensweise beim Erstellen von Programmen unter Helios:

```
% emacs fib.c                     Aufruf des Micro-Emacs Editors

#include <stdio.h>
#include <time.h>

/* Programm zur Berechnung von Fibinacci-Zahlen          */
/* Eingabebereich: 1..45, Option -r: rekursive Berechnung */

#define FIB_MAX 45

int main (int argc, char **argv)

{
        long i, nr_fib(long), r_fib(long);
        long atol(char *);
        short r = 0;

        if (argc > 2 && argv[1][0] == '-' && argv[1][1] == 'r'){
                r = 1;
                argc--;
                argv++;
        }

        while (--argc) {
                clock_t clock (void);
                long ticks = (long) clock ();
                if ((i = atol (*(++argv))) > FIB_MAX)
                        printf ("%ld > %d zu gross
                                 fuer Fibonacci-Berechnung\n",
                                 i, FIB_MAX);
                else
```

```
                        printf ("fib(%ld) = %ld\n", i,
                                 r?r_fib (i):nr_fib(i));
                ticks -= (long) clock (); ticks *= -5;
                printf ("Rechenzeit = %ld ms\n", ticks);
        }
        return 0;
}

long r_fib (long i)

{
        if (i <= 1)
                return 1;
        else
                return (r_fib(i-1) + r_fib(i-2));
}

long nr_fib (long i)

{
        long j, fibo[FIB_MAX + 1];

        if (i <= 1)
                return (i);
        else {
                fibo[0] = fibo[1] = 1;
                for (j = 2; j <= i; ) {
                        fibo [j] = fibo[j-1] + fibo[j-2];
                        ++j;
                }
                return fibo[i];
        }
}

% cc fib.c -s fib.s                Compilieren der Quelldatei

% asm -f /helios/lib/cstart.o fib.s -o fib
                          Einbinden der erzeugten Assemblerdatei

% fib -r 30   cr

  fib (30) = 1346269
  Rechenzeit = 9865 ms
```

Das erstellte Programm ermittelt die Zeitdauer zur Berechnung einer Fibonacci-Zahl. Dies ist ein Test für Vergleiche zwischen Rechnern und C- Compilern.

Im obigen Beispiel zeigte sich das Programm unter Helios auf der ATW um den Faktor 4 schneller als auf dem Atari Mega ST.

Die Option -r in der Parameterzeile des ausführbaren Programms bewirkt eine rekursive Berechnung der Fibonacci-Zahl.

Hermann Göken

Occam-Alternativen im Netzwerk

Ziel dieses Beitrages ist es zu zeigen, daß es natürlich möglich ist, Transputer zu vernetzen und bei der Anwendung auch die Parallelität eines Transputernetzwerkes auszunutzen und für den Anwender zugänglich zu machen. Zum anderen soll darauf hingewiesen werden, daß Transputer und vor allem parallele Programmierung nicht unzertrennlich mit der Programmiersprache Occam verbunden sein muß. Occam ist sicher eine exzellente Sprache, und auf die Möglichkeiten des Transputers zugeschnitten, jedoch ist eine Sprache wie beispielsweise C zweifellos eine akzeptable Alternative.

Die vorliegende Anwendung ist in C implementiert, wobei der C-Compiler CCT mit der dazu angebotenen Parallel-Bibliothek 'parlib' für die Programmierung benutzt wurde. Als Hardware wurden 13 Transputerkarten verwendet und unter dem Namen TA-1 vermarktet.

Das Bild zeigt die Vernetzung der 13 Transputerkarten, es wurde dabei eine Baumstruktur verwendet. Bei der TA-1 Transputerkarte handelt es sich um eine Europlatine, die entweder mit 128-KByte-EPROM und 128-KByte-RAM oder mit 256- KByte-RAM bestückt werden kann. Da die Karte für den Industrie-Einsatz konzipiert wurde, wurden statische RAM verwendet. Daneben befindet sich auf der Karte noch ein programmierbarer I/O-Baustein, der neben zwei seriellen Schnittstellen noch über parallele Ein- und Ausgänge verfügt. Die vier Link- Schnittstellen des Transputers sind über vier zehnpolige Buchsen an die Frontseite der Europlatine herausgeführt. Dabei sind hier die Transputersignale Reset und Analyse auf diese Stecker herausgeführt. Dies hat den Vorteil, daß Subsysteme vom übergeordneten System zurückgesetzt und analysiert werden können.

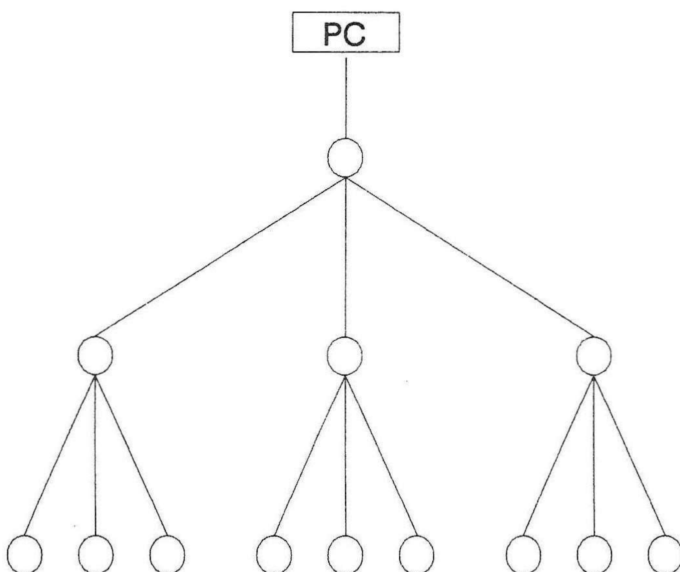


Bild 1: Vernetzung der 13 Transputerkarten

Die 13 Transputerkarten waren über Flachbandkabel über diese Buchsen verbunden - mehr Aufwand war nicht nötig. Man sieht hier, wie einfach der Aufbau eines parallelen Systems mit Transputerkarten vonstatten geht, vor allem, wenn man dies mit anderen bus- oder speichergekoppelten Parallelsystemen vergleicht. Die Verbindung zum Hostrechner, in diesem Fall ein AT-286 mit EGA-Karte, wurde über eine Linkadapterkarte realisiert. Diese Karte ist eine kurze PC- Einsteckkarte, die parallele Daten vom I/O-Port des PC in die seriellen Linkdaten umwandelt. Diese haben eine Datenrate von beachtlichen 10 MBaud.

Die Software

Programmiert wurde das System, wie schon erwähnt in C. Das vollständige Listing des Programms befindet sich am Ende des Artikels. Das System wurde derart konzipiert, daß in jeder Transputerkarte identisch dieselbe Software läuft. Es handelt sich, wie wohl leicht ersichtlich ist, um ein Programm zur Berechnung der Mandelbrotmenge. Inzwischen dürfte jedem bekannt sein, daß die Berechnung eines Bildes oder Ausschnitts dieser Menge auf einem AT bei einer Auflösung von 640 x 380 Punkten durchschnittlich 20 Minuten dauert. Um es gleich vorwegzunehmen, die 13 T-414 brauchen für diese Aufgabe etwa acht Sekunden. Dabei wurde, wie aus dem Listing ersichtlich ist, ohne Tricks gearbeitet.

In jeder Transputerkarte laufen effektiv drei Prozesse ab. Einer (der Vaterprozeß) berechnet die Mandelbrotmenge, der zweite Prozeß übernimmt die Kommunikation mit einem Subsystem, der dritte Prozeß übernimmt die Kommunikation mit dem übergeordneten System. Das System ist nun so konzipiert, daß die Baumtiefe theoretisch beliebig groß werden kann.

Das Grundprinzip der Berechnung eines Bildes läuft nun folgendermaßen ab. Der Hostrechner sendet Datenpakete, die mit einer Zieladresse versehen werden, an den obersten Transputer. Jedem Transputer ist dabei eine Adresse zugeordnet, die sich aus der Art und Weise der Vernetzung implizit ergibt. Ein Transputer weiß im Prinzip seine Adresse gar nicht. Die Adresse eines Transputers beschreibt vielmehr den Weg, den diese Nachricht durch den Transputerbaum nehmen muß, um am Ziel anzukommen.

Nun zum Adressierungsprinzip. Eine Nachricht, die vom PC zum Transputer mit der Nummer 14 geschickt werden soll, muß praktisch vom PC in Link0 des Transputers mit der Nummer 0 gesandt, dann von dort über Link2 an Transputer 2, und von dort von Link3 an Transputer 14 gesandt werden.

Wenn man sich nun diesen Weg in Binärdarstellung aufschreibt, erkennt man sehr schnell, warum Transputer 14 die Nummer 14 hat! Mit diesem Kommunikationsmodell ist sichergestellt, daß ein einzelner Empfänger seine eigene Adresse nicht notwendigerweise kennen muß. Eine Nachricht ist, wenn davon ausgegangen wird, daß die Zieladresse bei jedem Durchgang um 2 Bits nach rechts geshiftet wird, dann am Ziel, wenn sie den Wert Null hat. Und da es in diesem Fall für die Antworten nur einen Empfänger, nämlich den Host gibt, ist die Kommunikation in die andere Richtung ebenfalls genial einfach. Jeder Transputer schickt Antworten, die von Link1-3 kommen, einfach an Link0 weiter. Die Antwort (Rechenergebnis) seiner eigenen Berechnung wird ebenfalls an Link0 weitergegeben.

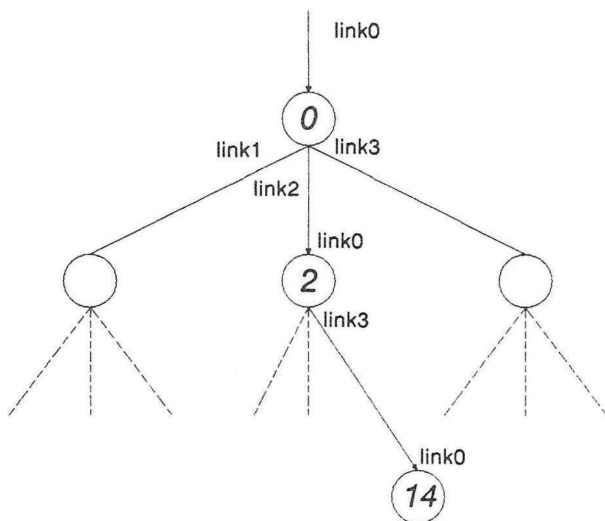


Bild 2:

Die Adressierung von Transputer mit der Nummer 14 schematisch dargestellt. Der Weg führt von Link0 des Transputers 0 über Link2 nach Transputer 2 und von dort über Link3 zum Transputer mit der Adresse 14.

Der PC schickt also einfach die Aufträge an die 13 Transputer (mit den sich durch dieses Kommunikationsmodell ergebenden Adresse), und wartet auf die Antworten. Dies wird im Detail so realisiert, daß jeweils Zeilenkoordinaten an die einzelnen Transputer verschickt werden. Diese schicken dann die Zeilen schon in die einzelnen Farbanteile, die zur Programmierung der EGA-Karte benötigt werden, aufgeteilt an den PC. Die Prozedur drawline() rechnet dabei eine Zeile in diese Farbanteile um. Dies hat zwei Gründe. Zum einen wäre der PC ganz einfach zu langsam, um diese Umrechnung selbst durchzuführen, und zweitens ist diese Arbeit im Transputer sowieso besser aufgehoben, denn es gibt 13 Transputer und nur einen PC.

Die Prozedur host_slhp() übernimmt diese Kommunikation vom einem übergeordneten zu einem untergeordneten System. Zuerst werden über die Library-Funktion getchannel() vier Headerworte der Nachricht empfangen. Dieser Header besteht aus der Zieladresse der Nachricht, einem Flag, ob es sich um ein Programm oder um einen Datensatz handelt, der Nummer des Ziel-Transputers, sowie der Anzahl Bytes des nachfolgenden Datenpakets.

Diese Prozedur übernimmt also auch das Booten der nachfolgenden untergeordneten Systeme. Dies geschieht im Detail folgendermaßen. Zuerst muß über das normale Loaderprogramm der erste Transputer (Root) im Baum über die Linkadapterkarte vom PC aus gebootet werden. Danach wird vom PC aus für jede im Baum vorhandene Adresse das gleiche Programm über die Linkadapterkarte ins System eingespielt. Der am Anfang als einziger Knoten arbeitende Transputer an der Baumwurzel sendet nun anhand der angegebenen Adresse das Datenpaket (Programm) an den Link 1, 2 oder 3 weiter.

Ist nun die Zieladresse nach dem schon angesprochenen Shiften um zwei Bit gleich Null, so bedeutet dies, daß dieses Datenpaket für den unmittelbar folgenden Transputer bestimmt ist. Dieser wird deswegen zuerst zurückgesetzt, damit die an seinem Link0 ankommenden Daten als Boot-Daten erkannt werden.

Die Erzeugung einer solchen bootfähigen Datei ist denkbar einfach. Der zum C- Compiler und zur Linkadap-

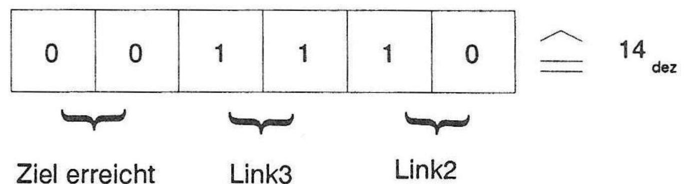


Bild 3: Aufbau des Adreßfeldes

Jeweils zwei Bit geben den Weg der Nachricht durch den Transputerbaum an. Adresse 14 bedeutet also, daß die Nachricht zuerst an Link2 und dann vom Empfänger an Link3 weitergesendet wird.

terkarte mitgelieferte Loader kann über eine Option die Daten, die normalerweise über den Link ins Zielsystem gehen, auch in eine Datei spielen. Diese Datei enthält dann genau die Bytefolgen, die notwendig sind, um einen rückgesetzten Transputer über einen beliebigen Link zu booten.

Sind alle Transputer im Baum nach diesem Schema gebootet, warten nach erfolgter Initialisierung die Transputer mit ihrem Hauptprozeß in der while(1)-Schleife in der Funktion main() auf ein Signal des Prozesses host_slhp(). Dieses Signal ist einfach ein Byte, das durch den Kanal can2 übertragen wird. Dieses Byte hat keinerlei Dateninformation, es dient lediglich zur Synchronisation. Die Funktionen getchannel() und putchannel() warten so lange, bis die Kommunikation über den angegebenen Kanal erfolgen kann, das heißt bis Sender und Empfänger bereit sind. Dieses Warten ist allerdings kein sogenanntes busy-wait, denn hier wird beim Warten keine Prozessorzeit verschwendet.

Eine Kanalkommunikation ist somit eine Möglichkeit zur Prozeßkommunikation und zur Prozeßsynchronisation. Wenn dann das Synchronisationbyte im Hauptprozeß empfangen wurde, wird die Prozedur fractals3() aufgerufen, die jeweils anhand der empfangenen Koordinationsdaten die Berechnung einiger Zeilen der Mandelbrotmenge übernimmt. Sobald eine Zeile vollständig berechnet wurde, wird sie über den internen Kanal can1 an den Prozeß sl_host2() gesendet. Interner Kanal bedeutet, daß dieser Kanal innerhalb eines Transputers zur Kommunikation zwischen Prozessen verwendet wird. Im Gegensatz dazu werden externe Kanäle durch die Transputerlinks gebildet. Interne und externe Kanäle werden im Prozeßkommunikationsmodell der Transputer jedoch nicht unterschieden. Aus diesem Grund sind in beiden Fällen auch die Funktionen getchannel() und putchannel() zu verwenden. Die Kanaladresse ist die einzige Unterscheidung zwischen internen und externen Kanälen. Interne Kanäle sind an Speicheradressen im normalen RAM-Speicher gebunden, externe Kanäle sind spezielle Adressen im unteren Adreßbereich des Transputers.

Der Prozeß sl_host2() muß allerdings nicht nur die Daten seines Transputers empfangen und weiterleiten, sondern auch die Daten der eventuell ihm untergeordneten Transputer. Dieser Prozeß muß also von vier Kanälen Daten empfangen, und diese sozusagen im Multiplexbetrieb über einen Kanal nach oben weitersenden. Um solche Fälle in den Griff zu bekommen, wurde die Funktion alt4in() implementiert. Sie wartet praktisch auf Daten von den vier angegebenen Kanäle. Genau der Kanal, der als erster seine Daten schickt, wird ausgewählt und empfangen. Falls zwei oder mehrere Kanäle gleichzeitig senden, wird der Kanal ausgewählt, der anhand der Reihenfolge der Kanaladressen die höhere Priorität besitzt. Die Priorität nimmt dabei von rechts nach links ab, das heißt der am ersten aufgeführte Kanal hat die höchste Priorität.

Die Prozedur blink() zeigt noch eine weitere Funktion der parlib des C- Compilers CCT. Dieser Prozeß wird ebenfalls als paralleler Prozeß gestartet. Er schaltet abhängig von den Variablen bldauer1 und bldauer2 über die Funktionen seterr() und testerr() die Error-LED der TA-1 Karte an und aus. Die Wartedauer zwischen Ein- und Ausschaltvorgang wird durch die Funtkion waittime() erzielt. Diese Funktion wartet die als Parameter übergebene Anzahl von Millisekunden. Dies erfolgt natürlich auch ohne busy-wait. Der Prozeß wird so lange aus der Scheduling-Liste herausgenommen, bis die gewünschte Anzahl von Millisekunden abgelaufen ist.

Die Darstellung 4 zeigt alle verwendeten Funktionen zur Realisierung des Programms.Dieses Beispiel zeigt, daß es mit Transputerkarten und einem C- Compiler, wie beispielsweise dem CCT, sehr einfach möglich ist, komplexe parallele Anwendungen, das heißt sowohl mehrere parallele Prozesse innerhalb eines Transputers verbunden mit einer Vernetzung von mehreren Transputern, durchzuführen. Die Funktionen start_p(), getchannel() und putchannel() bilden praktisch die Basis der parallelen Programmierung. Mit CCT wurde erfolgreich versucht, die Möglichkeiten des Transputers auch mit der Sprache C auszunutzen. Dies wurde dabei nicht durch eine Spracherweiterung von C erreicht (dies hätte auch wenig Sinn, da sonst wieder eine neue Sprache enstehen würde), sondern durch geeignete Library-Funktionen. Diese Beispiel zeigt auch, daß ein Parallelrechner auf Transputerbasis sehr einfach zu realisieren ist, denn sehr viele haben davon zwar schon gehört, aber einen Parallelrechner in Funktion gesehen, haben wahrscheinlich relativ wenige.

Gerd Häußler

Listing mn.c:

```
/*
  Mandelbrot : Auf T414
  */

#include "parlib.h"

typedef long    norm;
typedef long    supernorm;
```

```
#define TRUE            1
#define FALSE           0

#define MESS_SIZE       40
#define MESS_SIZE1      800

#define MANDFARB 0
#define FARBEN  16
#define RESET   0x20000000L             /* Adresse Event/Reset-Logik */

#define BLACK           0       /* dark colors */
#define BLUE            1
#define GREEN           2
#define CYAN            3
#define RED             4
#define MAGENTA         5
#define BROWN           6
#define LIGHTGRAY       7
#define DARKGRAY        8       /* light colors */
#define LIGHTBLUE       9
#define LIGHTGREEN      10
#define LIGHTCYAN       11
#define LIGHTRED        12
#define LIGHTMAGENTA    13
#define YELLOW          14
#define WHITE           15

#ifdef NOEGA
#define MESSUP  (HRES+3)
#else
#define MESSUP  ((HRES+1)/2 + 2)
#endif

#define HIGH

#ifdef HIGH
#define F 16384         /* Normalisierungsfaktor */
#define FLOG 14         /* 2er-Logarithmus von F */
#else
#define F 4096          /* Normalisierungsfaktor */
#define FLOG 12         /* 2er-Logarithmus von F */
#endif

int REALMIN,REALMAX,IMAGMIN,IMAGMAX;
int VRES,HRES;
int hstart,hend,vstart,vend,vstep;
int ITMAX;

int dummy;              /* Fuer Synchronisation */
int bldauer1,bldauer2;
int num;

CHANNEL can1,can2,can3;         /* Internere Kanuele */
int start;                      /* Synchronisation */
int     ws1[200],
        ws2[200],
        ws3[200],
```

```
        ws4[200];                /* Workspacebereiche fuer host_sl() und
                                    sl_host() */
char buffer1[MESS_SIZE1];
char buffer2[MESS_SIZE1];

char line[800];                  /* Gebufferte Zeile */
char line1[400];                 /* Umkodierte Daten */

#define F1      0
#define F2      6
#define F3      12
#define F4      20

int tab1[]={ BLUE,LIGHTBLUE };
int tab2[]={ CYAN,LIGHTCYAN };
int tab3[]={ MAGENTA, LIGHTRED };
int tab4[]={ YELLOW, RED };

ftab(farbe)
{
 if ( farbe >= ITMAX) return BLACK;
 if ( farbe > F4) return tab4[farbe%2];
 if ( farbe > F3) return tab3[farbe%2];
 if ( farbe > F2) return tab2[farbe%2];
 if (farbe==3) return GREEN;
 if ( farbe > F1) return tab1[farbe%2];
}

drawline()
{
 unsigned lb1,
        lb2,
        lb3,
        lb4;                     /* 80 Bytes/Zeile */

 int f,bit,i;
 int plind;

 plind=0; i=0;
 while (i<HRES) {
        for (bit=0; bit<8; bit++) {
                f=line[i++];

                lb1 <<= 1;
                lb1 |= (f & 1);

                lb2 <<= 1;
                lb2 |= (f & 2) >> 1;

                lb3 <<= 1;
                lb3 |= (f & 4) >> 2;

                lb4 <<= 1;
                lb4 |= (f & 8) >> 3;
        }
        line1[plind+2]=lb1;
        line1[plind+82]=lb2;
```

```
        line1[plind+162]=lb3;
        line1[plind+242]=lb4;
        plind++;
 }
}

fractals3()
{
   norm real0, imag0;             /* Startwerte                          */
   norm real, imag;               /* in der Iteration errechnete Werte   */
   supernorm realq, imagq;        /* real^2, imag^2                      */

   int h_kord, v_kord;            /* Horizontal- bzw. Vertikalkoordinate
                                     des betrachteten Bildschirmpunktes  */
   int farbe, itschritt;
   int DELTAREAL,DELTAIMAG;

   DELTAREAL = ((REALMAX-REALMIN) / HRES);
   DELTAIMAG = ((IMAGMAX-IMAGMIN) / VRES);

   imag0 = IMAGMAX;
   for ( v_kord=0; v_kord<vstart; v_kord++) imag0 -= DELTAIMAG;
   for( v_kord=vstart; v_kord<vend; v_kord+= vstep ) {
        real0 = REALMIN;
        blockmove(&v_kord,line,2);
        for( h_kord=hstart; h_kord< hend; h_kord++ ) {
           real=real0;
           imag=imag0;
           for( itschritt=0; itschritt<=ITMAX; itschritt++ )
           {
                if (real >= 2*F || real < -2*F ||
                        imag >= 2*F || imag < -2*F) break;

                realq = (real*real) >> FLOG;
                imagq = (imag*imag) >> FLOG;
                if( realq+imagq >= 4*F )
                        break;
                imag = ((real*imag) >> (FLOG-1)) + imag0;
                real = realq - imagq + real0;
           }
           line[h_kord]=ftab(itschritt);
           real0 += DELTAREAL;
        }
       drawline();                       /* Line umkodieren */
       blockmove(&v_kord,line1,2);       /* Header (y-Wert) */
       putchannel(line1,322,&can1);
       imag0 -= DELTAIMAG*vstep;
   }
}

/*
 Kommunikationsprozess Host -> Slave .
 Es werden Initial.- Daten erkannt, und an richtige
 Adresse weitergeleitet
*/

char buffer[30000];                         /* Zwischenbuffer */
```

```
host_slhp()
{
 int adr,bytes,prog,nummer;

 getchannel(&adr,4,LINK0IN);                          /* Zieladresse */
 getchannel(&prog,4,LINK0IN);
 getchannel(&nummer,4,LINK0IN);                       /* Transputernummer */
 getchannel(&bytes,4,LINK0IN);                        /* Anzahl Bytes */
 getchannel(buffer,bytes,LINK0IN);                    /* Init-Daten laden */
 switch (adr & 0x3) {
        case 0 :
                blockmove(buffer,&REALMIN,bytes);
                num=nummer;
                putchannel(&dummy,1,&can2);
                break;
        case 1 :
                adr >>= 2;
                if (prog && !adr) {                   /* Resete Subsystem */
                        *((char *) RESET) = 0xcf;
                        *((char *) RESET) = 0xff;
                        waittime(1);
                } else {
                        putchannel(&adr,4,LINK1OUT);
                        putchannel(&prog,4,LINK1OUT);
                        putchannel(&nummer,4,LINK1OUT);
                        putchannel(&bytes,4,LINK1OUT);
                }
                putchannel(buffer,bytes,LINK1OUT);
                break;                                /* An Link 1 weiter */
        case 2 :
                adr >>= 2;
                if (prog && !adr) {                   /* Resete Subsystem */
                        *((char *) RESET) = 0xbf;
                        *((char *) RESET) = 0xff;
                        waittime(1);
                } else {
                        putchannel(&adr,4,LINK2OUT);
                        putchannel(&prog,4,LINK2OUT);
                        putchannel(&nummer,4,LINK2OUT);
                        putchannel(&bytes,4,LINK2OUT);
                }
                putchannel(buffer,bytes,LINK2OUT);
                break;                                /* An Link 1 weiter */
        case 3 :
                adr >>= 2;
                if (prog && !adr) {                   /* Resete Subsystem */
                        *((char *) RESET) = 0x7f;
                        *((char *) RESET) = 0xff;
                        waittime(1);
                } else {
                        putchannel(&adr,4,LINK3OUT);
                        putchannel(&prog,4,LINK3OUT);
                        putchannel(&nummer,4,LINK3OUT);
                        putchannel(&bytes,4,LINK3OUT);
                }
                putchannel(buffer,bytes,LINK3OUT);
```

```
                break;                              /* An Link 1 weiter */
 }
}

host_sl()
{
 int i,j,k,l,m;

 while (1) host_slhp();
}


blink()
{
 while (1) {
        waittime(bldauer1);
        seterr();
        waittime(bldauer2);
        testerr();
 }
}

sl_host2()
{
 while (1) {
        alt4in(&can1,LINK1IN,LINK2IN,LINK3IN,322,buffer2);
        putchannel(buffer2,322,LINK0OUT);
        alt4in(LINK1IN,LINK2IN,LINK3IN,&can1,322,buffer2);
        putchannel(buffer2,322,LINK0OUT);
        alt4in(LINK2IN,LINK3IN,&can1,LINK1IN,322,buffer2);
        putchannel(buffer2,322,LINK0OUT);
        alt4in(LINK3IN,&can1,LINK1IN,LINK2IN,322,buffer2);
        putchannel(buffer2,322,LINK0OUT);
 }
}

main()
{
 inittimer();
 reschannel(&can1); reschannel(&can2);
 reschannel(LINK0IN); reschannel(LINK0OUT);
 reschannel(LINK1IN); reschannel(LINK1OUT);
 reschannel(LINK2IN); reschannel(LINK2OUT);      /* Kanaele ruecksetzen */
 reschannel(LINK3IN); reschannel(LINK3OUT);
 bldauer1=50; bldauer2=50;
 startp(sl_host2,ws1,LOPRI);
 startp(blink,ws4,LOPRI);
 startp(host_sl,ws2,LOPRI);             /* Prozesse starten */
 while (1) {
        getchannel(&dummy,1,&can2);     /* Starte, wenn Daten da */
        bldauer1=100;
        bldauer2=10;
        fractals3();                    /* Berechne */
        bldauer2=20;
        bldauer1=1600;
 }
}
```

Ein Multi-Transputernetz für die Simulation digitaler Systeme

Der "Munich Simulation Computer (MuSiC)" ist ein hochparalleler Spezialrechner für die schnelle Simulation digitaler Systeme. Seine Stärke liegt darin, daß nur das Berechnen von Signaländerungen Rechenzeit erfordert. Der Nachweis der Korrektheit des MuSiC-Entwurfs wie auch der zu MuSiC erarbeiteten Leistungsvorhersagen wird anhand eines Modells in Analogie zur Arbeitsweise von MuSiC selbst durch änderungsgetriebene Programmauswertung auf einem Parallelrechener, einem Netz von 55 kooperierenden Transputern, geführt. Möglich wird damit die Simulation eines MuSiC-Prototypen mit acht Prozessoren, 465 000 Gattern und 60 000 Flipflops, modelliert durch 3 800 Occam-Prozesse, der etwa 400 MuSiC-Maschinenbefehle je Sekunde ausführen kann.

Ein Logiksimulator ist traditionell ein Werkzeug für das Testen digitaler Entwürfe. Über das reine Testen hinaus erwarten Entwerfer heute jedoch, daß zum Beispiel auch die zu erwartende Rechenleistung eines neuen Systems ermittelt werden kann, ohne daß dafür ein Prototyp gebaut werden muß. Konnte bisher Modul für Modul auf der Basis strikter Schnittstellenregeln simuliert werden, so erfordert diese Aufgabenerweiterung die Fähigkeit, vollständige Entwürfe mit unter Umständen einigen Millionen Gattern handhaben zu können.

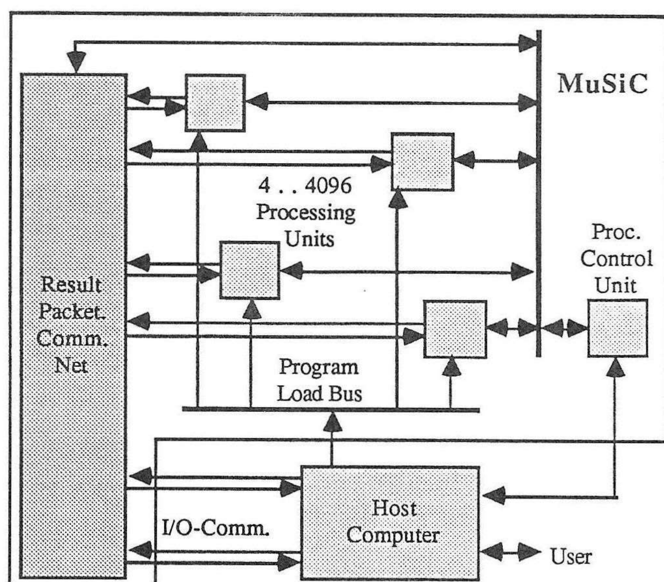


Bild 1: MuSiC-Struktur

Ein Logiksimulator operiert auf der Basis diskreter Zeitabschnitte gleicher Länge und ermittelt jeweils für das Ende eines Abschnittes den Signalzustand aller Gatterausgänge. Ein präzises Maß für die Leistungsfähigkeit eines Logiksimulators ist daher das Verhältnis von realer Länge dieses Zeitabschnittes zu der Rechenzeit, die der Simulator benötigt, um den Signalzustand des Entwurfs jeweils einmal zu berechnen. Diese Rechenzeit wird in der Regel jedoch in Relation zur Anzahl der zu berechnenden Gatterausgänge gesetzt, was zu dem abstrakteren Leistungsmaß Gatterauswertungen je Sekunde führt. Programmierte Simulatoren, die immer noch weit verbreitet sind, leisten bis etwa 20 000 Gatterauswertungen je Sekunde.

Entwürfe werden ständig umfangreicher, entsprechend steigt die zur Simulation benötigte Rechenzeit. Die Erfahrung lehrt dabei, daß die Rechenzeit zur Vorbereitung einer Simulation, zum Beispiel das Übersetzen der Entwurfsbeschreibung in ausführbaren Code, proportional zu Anzahl n der Gatter des Entwurfs, die Rechenzeit für die Simulation selbst proportional zu n^2 und erst die Rechenzeit für die Aufbereitung der Simulationsergebnisse wieder proportional zu n ansteigt. Die Antwort auf diese Herausforderung sind Spezialrechner wie etwa die Yorktown Simulation Engine der Firma IBM, die mittels 256 Prozessoren eine maximale Simulationsleistung von $3 \cdot 10^9$ Gatterauswertungen je Sekunde bietet.

Das Ergebnis von Arbeiten mit dem Ziel, die Leistungsgrenze in den Bereich von 10^9 bis 10^{12} Gatterauswertungen je Sekunde zu bringen, ist der Munich Simulation Computer (MuSiC), ein ebenfalls hochparalleler Spezialrechner, der Datenflußrechnerkonzepte benutzt, um Rechenleistung statt für die periodische Berechnung aller Gatterausgänge gezielt nur für das Berechnen der Gatterausgänge einzusetzen, für die sich aufgrund einer Ände-

rung eines oder mehrerer Gattereingänge eine Änderung des Ausgangssignals ergeben kann.

Das Arbeiten am Detailentwurf von MuSiC bis hin zur Prüfung von Aussagen über die zu erwartende Rechenleistung führt jedoch unmittelbar zu dem Problem der quadratisch wachsenden Simulationszeit, ohne daß man dem durch Zugriff auf eine Simulation Engine begegnen kann. Einen Ausweg aus diesem Dilemma, wirtschaftlich und dennoch, wie sich zeigen wird, hinreichend effizient, zeigt der nachfolgende Text auf.

Munich Simulation Computer

MuSiC setzt sich aus folgenden Komponenten zusammen:

- Processing Units (PU), den Prozessoren des Systems,

- einer Processing Control Unit (PCU) zur Überwachung der Prozessoraktionen,

- einem Result Packet Communication Net (RPCN), einem mehrstufigen Paketvermittlungsnetz, über das Prozessoren Ergebnisse austauschen und Ein/Ausgabe abwickeln können, sowie einem

- Host-Computer, der die Schnittstelle zum Benutzer darstellt und als Betriebssystem von MuSiC gesehen werden kann.

Eine Processing Unit ist wiederum modular aus Pipeline-Stufen zusammengesetzt. Ihre Hauptkomponenten sind: Cell Block, Operation Packet Distribution Unit (OPDU), Function Unit und Load Control Unit (LCU). Vereinfacht dargestellt ist ein Cell Block jeweils der lokale Speicher einer Processing Unit, sind in einer Function Unit jeweils die Rechenwerke einer Processing Unit zusammengefaßt, nimmt eine Operation Packet Distribution Unit (OPDU) jeweils aus dem lokalen Speicher gelesene Maschinenbefehle als Operationspakete entgegen und leitet sie einem freien Rechenwerk zu. Die Load Control Unit steuert das Einlesen eines MuSiC-Maschinenprogramme in die lokalen Speicher.

Einheit	Anzahl der Basisprozesse	Codegröße in Byte
PU	367	325.116
PCU	30	13.916
2x2-Router	42	20.796
RPCN	840	415.920
8-PU-Prototyp	3,800	3.000.000

Bild 2: Code-Komplexität des MuSiC-Prototyp-Modells

Modellierung digitaler Systeme durch Occam-Prozesse

Grundkonzept der Programmiersprache Occam ist das Prozeßkonzept, so daß ein Occam-Programm aus einer Menge von Prozessen besteht. Ein Prozeß kann selbst wieder aus Prozessen aufgebaut sein oder, als Basis der Prozeßhierarchie, eine Sequenz von Operationen beinhalten.

Es gibt verschiedene Möglichkeiten, ein Occam-Programm ablaufen zu lassen. Im Extremfall könnten so viele Prozessoren bereitgestellt werden, wie Prozesse im Programm enthalten sind, so daß alle Prozesse uneingeschränkt parallel agieren können. Kostengünstiger ist es natürlich, zwar mehr als nur einen Prozessor zu Verfügung zu stellen, nicht jedoch jedem Prozeß einen Prozessor zuzuordnen. Ein Programm enthält dann zum einen parallel ablaufende Prozesse, die jeweils über einen Prozessor verfügen und zum Beispiel Pipeline-Stufen des MuSiC- Entwurfs repräsentieren, und zum anderen innerhalb von Pipeline-Stufen- Prozessen angesiedelte Prozesse, zum Beispiel zur Modellierung von Registertransfer-Stufen, die in traditioneller Time-Sharing-Technik pseudoparallel ablaufen.

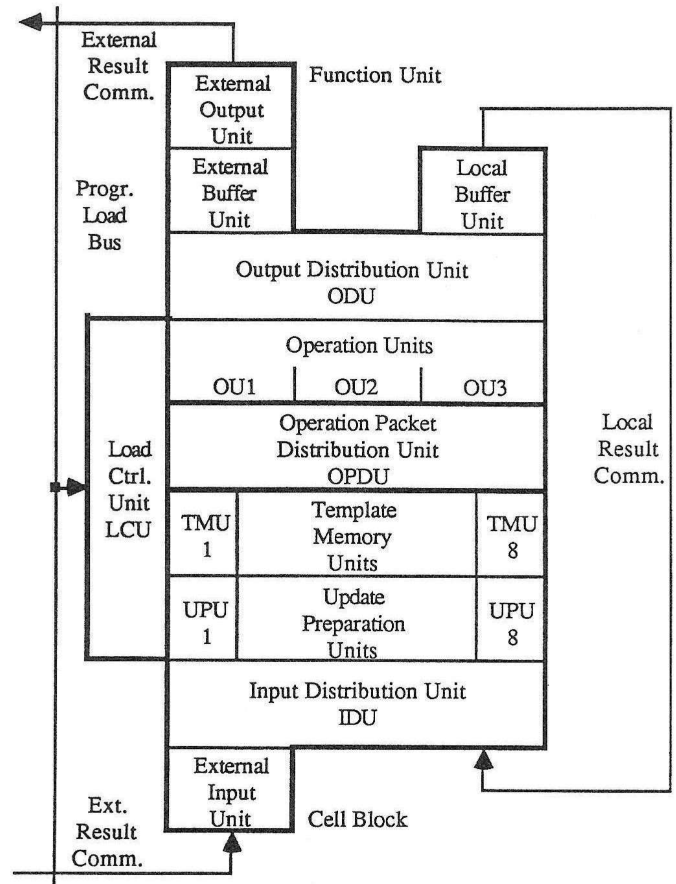


Bild 3: Innere Struktur einer Processing Unit

Occam-Prozesse kommunizieren mittels sogenannter channels, die jeweils eine unidirektionale Verbindung zwischen einem Prozeßpaar darstellen (Punkt-zu- Punkt-Verbindung). Dabei übergibt ein Prozeß dem Kanal eine Nachricht, die dann der Partnerprozeß dem Kanal entnehmen kann. Beide Operationen erfolgen sychron, denn der Prozeß, der zuerst auf den Kanal zugreift, kann erst wieder agieren, wenn auch sein Partner die komplementäre Kanaloperation ausgeführt hat. Erst nach Abschluß der Nachrichtenübergabe können beide Prozesse beliebig weiterarbeiten.

Diese Art der Kommunikation unterscheidet sich vollständig von der zwischen Registertransfer-Stufen (RT-Stufen) eines Hardware-Systems. In Occam sind Prozeßkommunikation und Prozeßsynchronisation an den gleichen Mechanismus, Nachrichtenzustellung, gebunden. In Hardware-Systemen hingegen basiert

- Prozeß(RT-Stufen)-Kommunikation darauf, daß Leitungen als Träger von Information gemeinsam genutztes Betriebsmittel zweier oder mehr Prozesse sind,

- Prozeß(RT-Stufen)-Synchronisation auf der Existenz eines speziellen Kontrollsignals, in der Regel eines zentralen Taktsignals. Eine ansteigende Flanke des Taktsignals veranlaßt dann zum Beispiel eine RT-Stufe, gemäß der auf den Eingangsleitungen angebotenen Information einen neuen Registerzustand zu bestimmen und diesen in "Master"- Register einzutragen. Die anschließende, abfallende Flanke des Taktsignals steuert das Kopieren des neuen Zustandes vom Master- Register in ein "Slave"-Register, womit eine Zustandsänderung auf den Ausgangsleitungen der RT-Stufe sichtbar wird.

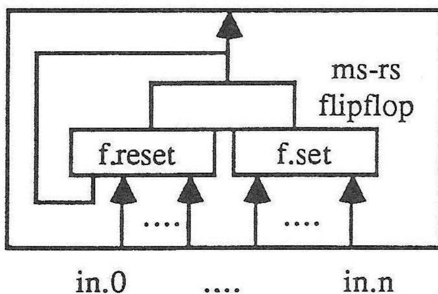


Bild 4a: Steuerwerk-Prozeß

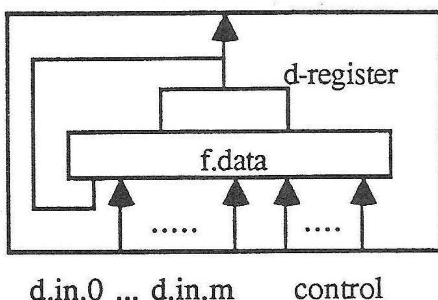


Bild 4b: Operationswerk-Prozeß

Hardware-Kommunikation bedeutet also erstens, daß das Ändern eines Leitungszustandes und das Verarbeiten eines Leitungszustandes zu jeweils verschiedenen Zeitpunkten erfolgt, und zweitens, daß ein Ausgangssignal durch Leitungsverzweigung Eingangssignal mehrerer RT-Stufen sein kann.

Mittels zusätzlicher Pufferprozesse, gegebenenfalls zusätzlicher Nachrichtenkopierprozesse (auch fan.out processes genannt), kann dieses Verhalten natürlich in der Occam-Welt nachgebildet werden. Solange dabei die Intention besteht, Hardware-Systeme auf der Ebene ganzer Systembausteine wie Prozessoren, Speicher, Geräte etc. zu modellieren, mag der damit verbundene Effizienzverlust akzeptabel sein. Bei einer Detailmodellierung eines vollständigen Rechners auf RT-Stufen-Niveau ist es ausgeschlossen, für eine einfache Kommunikation zwischen einer sendenden und einer empfangenden RT-Stufe drei Kanaloperationen, zwischen einer sendenden und zwei empfangenden RT-Stufen sieben Kanaloperationen und so fort auszuführen.

Der im MuSiC-Projekt bei der Modellierung gewählte Weg übernimmt unmittelbar die Hardware-Kommunikationsprinzipien. Globale Variable, jeweils für zwei oder mehr Prozesse sichtbare Träger von Information, dienen nur der Prozeßkommunikation. Der Zugriff auf diese Variablen wird durch eine Synchronisationsnachricht geregelt, welche die Prozesse jeweils mittels Kanalkommunikation erreicht. Erhält ein Prozeß die Nachricht "operate", so aktiviert dies den Prozeß mit dem Recht des Zugriffs auf seine Eingangssignale zur Berechnung eines neuen Zustandes, der in eine lokale Variable "Master" eingetragen wird. Folgt danach die Nachricht "output", so wird der Inhalt von Master in eine lokale "Slave" sowie in die das Ausgangssignal repräsentierende, globale Variable kopiert.

Synchronisationsnachrichten müssen die Aktionen von Prozessen in der gleichen Weise kontrollieren wie ein globales Taktsignal die Aktionen von RT-Stufen. Wie oben erwähnt, hat ein Taktsignal periodisch zwei Zeitpunkte zu markieren: Zeitpunkt eins für die Auswertung der Eingangssituation und Zeitpunkt zwei für das Umsetzen von Ausgängen im Falle einer Zustandsänderung.

Es gibt einen Taktprozeß "clock", von dem aus die Verteilung der Synchronisationsnachrichten baumartig erfolgt, unter Abbildung der Baumstruktur auf das Verbindungsschema des Prozessornetzes. Für die Prozesse, die auf einem gemeinsamen Prozessor ablaufen, ist es jedoch besser, die Nachricht sequentiell von Prozeß zu Prozeß weiterzureichen, da dann immer nur ein Prozeß aktiv ist und ihm der Prozessor nicht zugunsten eines anderen, ebenfalls aktiven Prozesses, entzogen werden kann. Von den Blättern des Verteilbaumes müssen Fertig-Meldungen zum Taktprozeß als Quittung dafür zurückfließen,

daß ein Taktzyklus abgeschlossen ist. Im Gegensatz zu Signallaufzeiten in Hardware-Systemen können nämlich Rechenzeiten und Nachrichtenlaufzeiten nicht hinreichend genau abgeschätzt oder gemessen werden.

Der Vorteil des gewählten Ansatzes liegt eindeutig bei der Beschleunigung der Prozeßkommunikation. Bei Prozessen, die auf demselben Transputer ablaufen, ist beispielsweise eine Eins-zu-Eins-Kommunikation 75mal, eine Eins-zu-Zwei- Kommunikation 175mal, eine Eins-zu-Drei-Kommunikation 250mal schneller. Aber auch bei Kommunikation über Prozessorgrenzen hinaus ist dieser Ansatz effizienter. Paare von Kopierprozessen, von denen jeweils der sendende Prozeß die Synchronisationsnachricht nach den RT-Stufen modellierenden Prozessen sieht, übernehmen jeweils den konsistenten Abgleich von Signalvektoren auf zwei Prozessoren. Verglichen mit einem vollständig nachrichtensynchronisierten Prozeß-System, bei dem Multiplexer/Demultiplexer-Prozesse jeweils eine Vielzahl von Kanälen zwischen Prozessen auf einen Kanal ("Link") zwischen den beiden Prozessoren abbilden, ist der Nachrichtenumfang etwa der gleiche, die Transfer-Startzeit fällt jedoch nur einmal an.

Durch Synchronisationsnachrichten kontollierter Zugriff ist im übrigen der Schlüssel zu änderungsgetriebener Prozeßaktivierung. Dazu wird die Gruppe der Eingangsvariablen eines Prozesses jeweils um eine Variable "Event" ergänzt. Ein Prozeß, der die Nachricht "output" erhält und feststellt, daß seine Ausgangsvariable mit einem veränderten Wert zu besetzen ist, setzt bei jedem Prozeß, der von dieser Änderung betroffen ist, die zugehörige Variable Event. Nach Erhalt der Nachricht "operate" hingegen berechnet ein Prozeß nur bei gesetztem Event unter Rücksetzen von Event einen neuen Zustand. Auf diese Weise wird nur von den Prozessoren Rechenleistung aufgenommen, die eine Änderung des Systemzustandes bewirken können.

Modellierung von MuSiC

Die Modellierung wird durch folgende Entwurfsregel erleichtert:

- Steuerwerke werden in Schieberegister-Technik entworfen,

- Dateneingänge bleiben stabil, solange ein Kontrolleingang aktiv ist.

Der Modellierung eines Steuerwerkes kann dadurch ein Prozeßmuster gemäß Steuerwerk-Prozeß und der eines Operationswerkes ein Prozeßmuster gemäß Operationswerk-Prozeß zugrundegelegt werden, wobei die zweite Regel das Setzen von Event nur dann notwendig macht, wenn ein Kontrolleingang geändert worden ist.

Die Programmierung des MuSiC-Modells reduziert sich auf das Auslesen des benötigten Prozeßmusters aus einer Bibliothek, seiner Individualisierung,

- im Falle einer Steuerwerkskomponente durch Definition der Setz- beziehungsweise Rücksetz-Funktion,

- im Falle eines Operationswerkes durch Definition der auf die Eingangsdaten anzuwendenden Funktion,

und seiner "Verdrahtung" mit der umgebenden Prozeßwelt über globale Variablen. Diese Prozesse sind dann die Basis-Prozesse, mittels derer Pipeline-Stufen modelliert werden. Anzahl und resultierender Codeumfang sind in der Tabelle für die Ebenen PU und RPCN sowie für einen vollständigen 8-PU-Prototypen aufgelistet.

Damit die Modellauswertung schnell erfolgt, muß dieser Code auf einem Netz von Transputern ablaufen, wobei die Zuteilung von Codesegmenten zu Transputern ein diffiziles Problem ist. Die gegenwärtige Aufteilung basiert auf folgenden Grundsätzen:

1. Nicht mehr als 60 Transputer sollen notwendig sein.

Eine intuitiv attraktive Lösung des Zuteilungsproblems wäre die Zuordnung jeweils einer Pipeline-Stufe zu einem Transputer, was bei einem 8-PU- Prototypen 230 Transputer erfordern würde. Eine Beschränkung auf zwei PU hingegen ließe keine Untersuchung der Auswirkungen des RPCN auf die MuSiC- Leistung mehr zu. Also muß der Code von m Pipeline-Stufen einem Cluster von n ($< m$) Transputern zugeordnet werden.

2. Transputer-Transputer-Kommunikation soll minimal sein.

Über das Abgleichen der Rechenlast des einzelnen Transputers hinaus und unter Berücksichtigung, daß je Transputer nur vier Links zur Verfügung stehen, muß die Zuteilung die Kooperationsbeziehungen zwischen Pipeline-

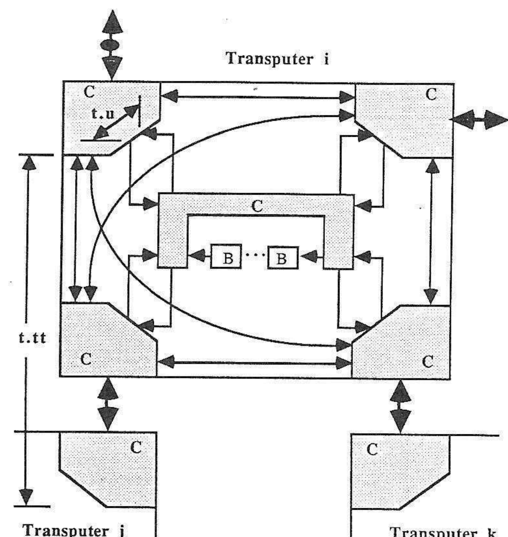


Bild 5: Verteilung von Synchronisationsnachrichten

C: Prozesse für das Kopieren und Zuleiten von Nachrichten sowie das anschließende Einsammeln der Fertig-Meldungen, B: Basis-Prozesse; t.tt: Transputer-Transputer-Transferzeit, t.u: Umlaufzeit

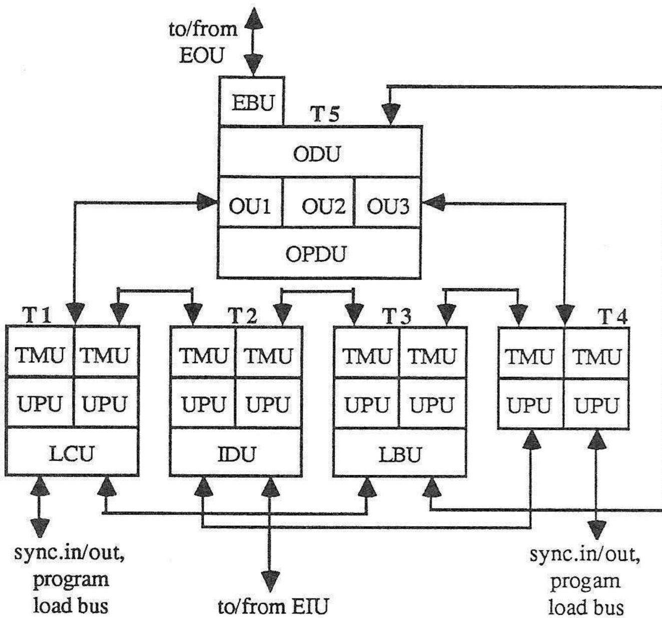


Bild 6: Cluster-Beispiel

Stufen beachten. Im Cluster-Beispiel sind 25 Pipeline-Stufen für die Zuordnung zu einem Cluster von fünf Transputern ausgewählt worden. Läßt man die nach dem Laden inaktive LCU außer acht, so ist die Summe der Kooperationsbeziehungen 33. Davon sind 17 transputer-lokal, zwölf erfordern Transputer-Transputer-Kommunikation und vier benötigen einen Transputer als Zwischenstation, da zwischen den Transputern zwei und drei einerseits und dem Transputer fünf andererseits keine direkte Link-Verbindung besteht.

3. Ausgleich der Pfadlängen von Synchronisationsnachrichten.

Transputer sind Knoten des Verteilbaumes für Synchronisationsnachrichten, während die Nachricht auf einem Transputer die Prozeßkette sequentiell durchläuft. Für die Nachrichtenlaufzeit gilt demnach

$$t = \underset{i=1}{\overset{n}{\mathrm{MAX}}} \left[2 \times (e_i \times t.tt) + t.u_i + t.r\right]$$

mit

e der Verzweigungsebene, auf der sich der jeweilige Transputer i befindet,

t.tt der Transputer-Transputer-Transferzeit,

t.u der Umlaufzeit auf dem jeweiligen Transputer,

t.r der Rechenzeit aller dem jeweiligen Transputer zugeordneten Prozesse und

n der Gesamtzahl der Transputer im Netz.

Für das gegenwärtige Modell läßt sich für t.r = 0, also die reine Verteilung einer Nachrichtensequenz "operate" - "output", eine Laufzeitsumme von 2.425 ms erreichen (Transputer-Typ T414 - 15 MHz, Linkgeschwindigkeit 10 Mbit/s). Dies stellt das Zeitminimum für die Simulation einer MuSiC-Taktperiode dar.

Sobald jedoch Pipeline-Stufen aktiv sind (t.r = 0), erhöht die Rechenzeit, die notwendig ist, um diese Aktivitäten durchzurechnen, diese Zeit auf 4,6 ms für die Simulation einer MuSiC-Taktperiode von real 30 ns. Das in der Einleitung angesprochene Verhältnis von Realzeit zu Simulationszeit liegt somit in der Größenordnung von 1 : 100 000. Zugleich entspricht die Fähigkeit, den Signalzustand von 465 000 Gattern in 4,6 ms durchzurechnen, einer Simulationsleistung von 10^8 Gatterauswertungen je Sekunde. Dies bedeutet, daß ein MuSiC-Prototyp verfügbar ist, der, ohne real gebaut worden zu sein, 400 Maschinenbefehle je Sekunde ausführen kann.

Dr. Winfried Hahn, Herbert Anger, Andreas Hagerer, Bernd Schuster

Multitop, ein Multiprozessor mit dynamisch variabler Topologie

Multitop ist ein Multiprozessorsystem, bei dem die Interprozessor- Kommunikation nicht über einen gemeinsamen Speicher, sondern über schaltbare Punkt-zu-Punkt-Verbindungen abläuft. Die Verbindungen können während des Programmablaufs umgeschaltet werden, so daß ihre Topologie der Topologie des Programms optimal angepaßt werden kann. Dies wird durch ein für alle Permutationen blockierungsfreies Koppelnetz erreicht, dessen modularer Aufbau eine beliebige Erweiterung des Systems gestattet. Am Beispiel der schnellen Fourier-Transformation und einer Spline-Approximation wird gezeigt, daß der Wirkungsgrad der Programmausführung sich gegenüber festen Verbindungen deutlich erhöht. Als Prozessoren werden Transputer verwendet. Die Programmiersprache ist Occam.

Das Konzept von Multitop

Die folgenden drei Probleme erschweren die Anwendung paralleler Architekturen im Rechnerbau:

1. Das Problem der parallelen Programmierung. Das heißt, daß es bislang keine einfache Methode gibt, einen parallelen Algorithmus auf einen Parallelrechner abzubilden. Damit verbunden sind die Fragen nach der optimalen Architektur eines Parallelrechners, nach seinem Kommunikationsmodell sowie der verwendeten Programmiersprache.

2. Das Problem der Effizienz bei paralleler Verarbeitung, die durch die stets notwendige Interprozessor-Kommunikation reduziert wird. Methoden zur Minimierung der Interprozessor-Kommunikation sind deshalb sowohl auf algorithmischer als auch auf architektonischer Seite notwendig.

3. Das Problem der Skalierbarkeit. Darunter versteht man die Möglichkeit, die Rechenleistung eines Systems in weiten Grenzen variieren zu können, um das Angebot an Leistung dem jeweiligen Bedarf anpassen zu können. Die Architekturmerkmale sollen dabei unverändert bleiben.

Das Konzept von Multitop ist ein Vorschlag zur Lösung dieser Probleme.

Das Kommunikationsmodell

Das Multitop zugrundeliegende Kommunikationsmodell basiert aus der Sicht des Benutzers auf sequentiellen Prozessen, die durch Botschaftenaustausch kommunizieren. Die Implementierung dieses Modells auf der physikalischen Ebene erfolgt durch Prozessoren (Transputer), die über sogenannte Kanäle miteinander verbunden sind. Als Programmiersprache wird Occam verwendet. Die Vorteile dieses Konzepts gegenüber der Kommunikation über einen gemeinsamen Speicher und gemeinsame Variable sind (Bild 1):

1. Der Fall, daß mehrere Prozessoren auf dieselbe Variable zugreifen, tritt bei Botschaftenaustausch nicht auf. Dadurch entfällt das Konsistenzproblem dieser Variablen bei schreibendem Mehrfachzugriff.

2. Die Zahl der über Kanäle gekoppelten Prozessoren ist nicht durch die endliche Bandbreite eines gemeinsamen Speichers limitiert.

3. Die Zuverlässigkeit der Interprozessor-Kommunikation ist wegen der Vielzahl verwendeter Kanäle größer als bei einem zentralen Bus.

4. Die Testbarkeit paralleler Programme wird dadurch erhöht, daß die Zahl der Testpunkte größer als bei gemeinsamen Variablen ist, weil jeder Kanal eine Schnittstelle mit definiertem Protokoll darstellt.

Die Nachteile dieses Kommunikationsmodells liegen in:

1. Gemeinsame Daten und Programme müssen mehrfach gehalten werden.

Kommunikationskonzept: aus logischer Sicht: Botschaften aus physikalischer: Kanäle	
Vorteile: o keine Synchronisation bei Mehrfachzugriff o Zahl der Prozessoren nicht begrenzt o erhöhte Zuverlässigkeit o bessere Testbarkeit	Nachteile: o gemeinsame Daten mehrfach o Kanäle langsamer als Speicher

Bild 1: Kommunikationskonzept

Aufgrund des Preisverfalls bei Speichern ist dieses Gegenargument jedoch relativiert.

2. Der Datenzugriff ist bei gemeinsamem Speicher prinzipiell schneller, da bei einem Kanal die Übertragungszeit zwischen den lokalen Speichern hinzukommt. Für den Fall eines Zugriffskonflikts auf den gemeinsamen Speicher kehren sich allerdings die Zeitverhältnisse um, da dann eine zeitaufwendige Arbitrierung der Zugriffe erforderlich ist.

Schließlich ermöglicht dieses Kommunikationsmodell die neue Eigenschaft eines Multiprozessors - die dynamisch variable Topologie - zu realisieren.

Die dynamisch variable Topologie

Warum ist eine dynamisch variable Topologie sinnvoll? Zur Klärung dieser Frage soll stellvertretend für viele Probleme aus der Numerik die Parallelisierung der schnellen Fourier-Transformation (FFT) und einer Spline-Glättung dargestellt werden.

Anhand des Signalflußgraphen der FFT wird ersichtlich (Bild 2), daß dieser Algorithmus in Stufen abläuft. Zwischen den Stufen werden Daten ausgetauscht, wobei die Art des Austauschs sich von Stufe zu Stufe ändert (Bild 3). Eine bestimmte feste Topologie von Prozessorverbindungen kann deshalb für die verschiedenen Phasen der FFT nicht optimal bezüglich der Interprozessor- Kommunikation sein. Weiterhin ist die Vereinigungsmenge aller Teil-Topologien zu komplex für eine reale Implementierung. Es ist also besser, die Topologien der Verbindungen zwischen den Stufen so zu verändern, daß sie dem Datenfluß der FFT entspricht.

Derselbe Sachverhalt wird beim Algorithmus einer parallelen Spline-Glättung deutlich: Bei dem von mir entwickelten Verfahren der Spline-Approximation wird mit Hilfe von sogenannten Basis-Splines eine Ausgleichskurve durch eine Folge von Punkten beziehungsweise Meßwerten gelegt, wobei gleichzeitig eine Datenreduktion durch Glättung stattfindet. Dieses Verfahren wurde auch im Hinblick auf gute Parallelisierbarkeit entwickelt. Es muß

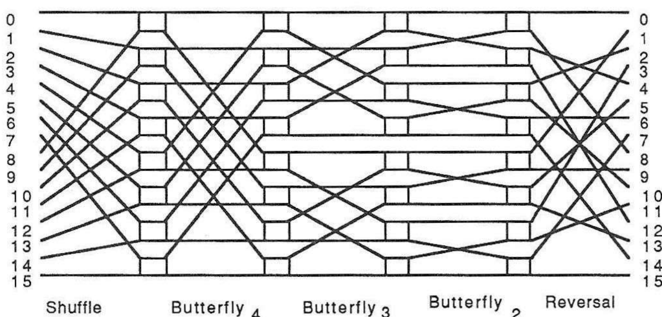


Bild 2: Signalfluß der schnellen Fouriertransformation

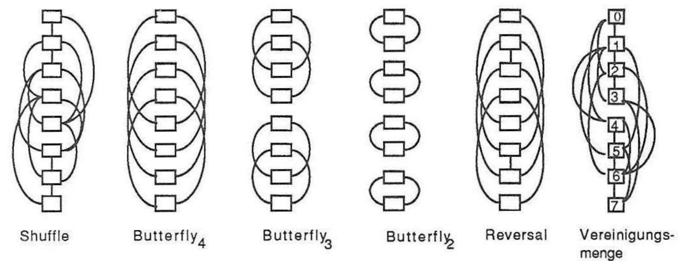


Bild 3: Sequenz von Topologien

dabei das lineare Gleichungssystem $A^T Ap = A^T y$ gelöst werden (Bild 4).

Dazu sind mehrere Schritte notwendig, die - jeder für sich - eine eigene optimale Topologie von Prozessorverbindungen aufweisen:

1. Die Dateneingabe erfolgt am zweckmäßigsten von einem zentralen Punkt aus mit Hilfe einer sternförmigen Topologie zur Verteilung der Daten.

2. Das Aufstellen der Matrix A kann dadurch erfolgen, daß jeder Prozessor eine Submatrix innerhalb einer gitterförmigen Topologie erzeugt.

3. Die Transposition von A kann in log N Schritten mit Hilfe eines Shuffle- Netzwerks realisiert werden.

4. Die Multiplikation von A^T mit y beziehungsweise mit A erfolgt in einer Kette beziehungsweise einem Gitter von Prozessoren am einfachsten.

5. Das Lösen des linearen Gleichungssystems kann mit Hilfe des Gauss-Jordan- Verfahrens auf einem Gitter optimal ablaufen.

6. Die Ausgabe der Unbekannten p erfolgt wiederum über den Stern.

In beiden Fällen - FFT und Spline-Glättung - ist eine dynamisch variable Topologie also sinnvoll. Für viele andere parallele Algorithmen gilt ebenso, daß sie phasenweise ablaufen, wobei für jede Phase ein optimale Topologie existiert. Ein Umkonfigurieren der Verbindungen zwischen den Prozessoren zur Laufzeit eines Programms ermöglicht deshalb, die Topologie des Rechners ständig an die Topologie des Problems anzupassen. Voraussetzung dafür ist, daß die Umschaltung schnell genug erfolgt. Bei Multitop wird eine zur Übersetzungszeit bekannte Topologie in etwa 10 Mikro-Sekunden eingestellt, so daß ein schneller Topologie-Wechsel möglich ist. Durch diese

$$A^T Ap = A^T y$$

1. Daten eingeben: Stern
2. Matrix a aufstellen: Gitter
3. A transponieren: Shuffle
4. Matrix-Vektor-Mult.: Kette
5. lineare Gleich. lösen: Gitter
6. p ausgeben: Stern

Bild 4: Parallele Spline-Glättung

neue Eigenschaft können folgende Verbesserungen erzielt werden:

1. Paralleles Programmieren wird in der Hinsicht vereinfacht, daß sich der Rechner an das Programm anpaßt und nicht umgekehrt.

2. Die stets notwendige Interprozessor-Kommunikation wird dadurch minimiert, daß die Erzeuger von Daten direkt, das heißt ohne Einschalten von Zwischenstufen, mit den Verbrauchern verbunden sind. Dies wird durch die schaltbaren Punkt-zu-Punkt-Verbindungen der dynamisch variablen Topologie erreicht. Somit kann die Effizienz bei der Ausführung paralleler Programme gesteigert werden.

Aufgrund dieser Überlegungen ergibt sich eine bestimmte Architektur für das Multitop-System:

Die Architektur

Ausgangspunkt der Darstellung der Architektur von Multitop ist die Idee, Prozessoren mit lokalen Speichern über Kanäle zu koppeln. Die Forderung der variablen Topologie erzwingt als Verbindungsnetzwerk entweder einen vollständig vermaschten Graphen (Bild 5) oder einen Kreuzschienenverteiler (Bild 6). Denn der vollständig vermaschte Graph enthält alle möglichen Topologien als Teilgraphen, beziehungsweise diese lassen sich bei Bedarf mit Hilfe des Kreuzschienenverteilers einstellen.

Beide Lösungen sind aber für die Kopplung vieler Prozessoren unwirtschaftlich, da sie $O(N^2)$ Kanäle beziehungsweise Schalter erfordern. Die Lösung dieses Problems liegt in der Einführung von schaltbaren Verbindungen, die mit Hilfe eines Koppelnetzes, das aus O(N*log N) Schaltern besteht, realisiert werden (Bild 7).

Allerdings ist der mit der Durchschaltung der Verbindungen verbundene Zeitverlust, der durch das Setzen der

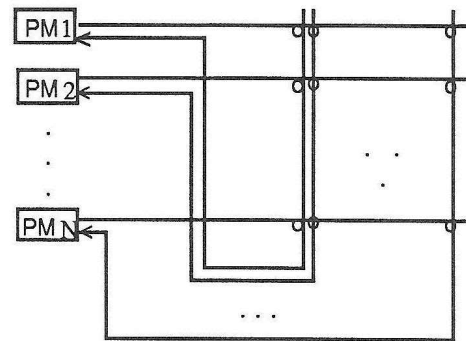


Bild 6: Kreuzschienenverteiler

Schalter des Netzes entsteht, so klein zu halten, daß er bei der Programmausführung nicht wesentlich ins Gewicht fällt. Dies führt zu der Idee, die Schalter des Netzes parallel, das heißt von mehreren Prozessoren gleichzeitig, setzen zu lassen. Im Prinzip könnten dazu dieselben Prozessoren PM1 - PMN verwendet werden, die auch dem Benutzerprogramm zur Verfügung stehen. Dieses würde dann allerdings in der Zeit, in der die Berechnung der Schalterstellungen vorgenommen wird, nicht ausgeführt werden können. Um diesen Zeitverlust bei jeder neuen Verbindung zu vermeiden, können zusätzliche Prozessor-Speicher-Module (PM1 - PMM) eingesetzt werden, deren Zahl M wesentlich kleiner als N sein kann (Bild 8). Die M Module müssen gemäß dem Algorithmus des Schalter-Berechnens miteinander verbunden sein. Diese Art der Verbindung kann fest sein, da stets dasselbe Programm ausgeführt wird. Geeignet ist eine Baum- oder Gitter-Topologie.

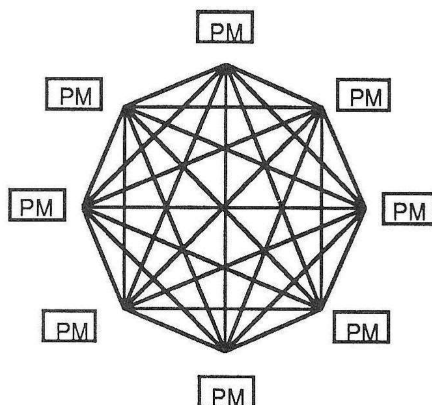


Bild 5: Vollständig vermaschter Graph

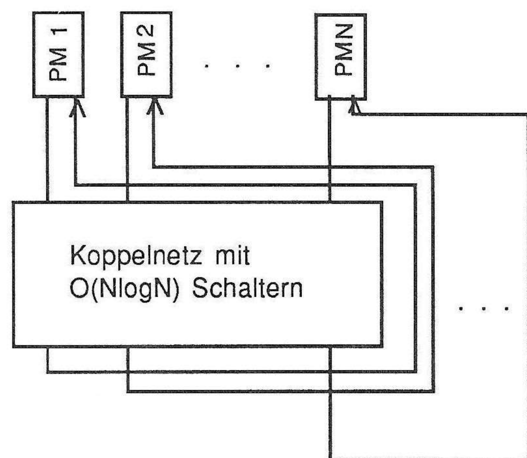


Bild 7: Koppelnetz

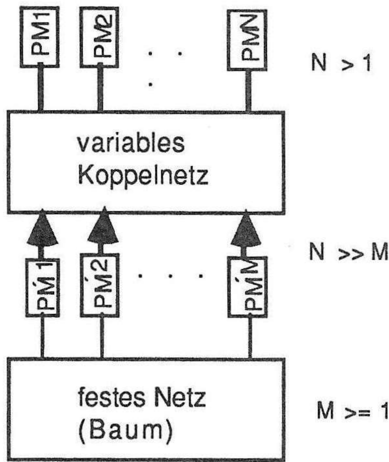


Bild 8: Paralleles Setzen

Bei den bisherigen Betrachtungen wurde die Datenein/-ausgabe nicht berücksichtigt. Eine schnelle Verarbeitung ist allerdings ohne eine ebensolche Datenein/-ausgabe wirkungslos. Bei Rechnern erfolgt sie zumeist von einer zentralen Stelle aus - der Schnittstelle zur Außenwelt. An diesem Punkt werden Eingabedaten eingespeist, die danach auf alle Prozessoren verteilt werden müssen. Nach der Verarbeitungsphase werden Ergebniswerte an dieser Stelle gesammelt und von dort ausgegeben. Aus Gründen der Auslastung ist es zweckmäßig, die zentrale Datenein/-ausgabe mit Hilfe derselben Prozessor- Speicher-Moduln vorzunehmen, die auch für das Koppelnetz zuständig sind, da diese nur während der Verarbeitungsphase mit

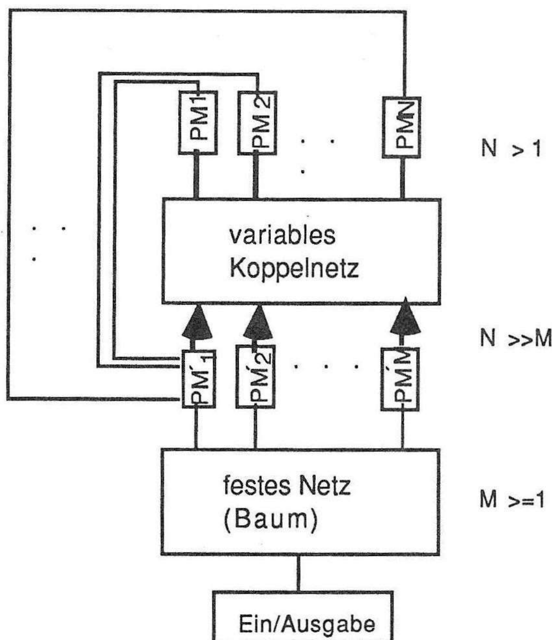


Bild 9: Stern zur Datenein/-ausgabe

dem Herstellen neuer Topologien beschäftigt sind. Dazu müssen sie in einer bestimmten, festen Topologie mit den Benutzer-Moduln verbunden sein. Für das Verteilen und Sammeln von Daten eignet sich eine sternförmige Topologie besonders, im Zentrum des Sterns steht die Schnittstelle zur Außenwelt. Somit ist dem Aufbau von Multitop noch ein Stern von festen Verbindungen überlagert (Bild 9).

Das Multitop-Koppelnetz

Das modulare Koppelnetz des Multitop-Rechners verbindet die N Eingänge des Netzes mit den N Ausgängen auf N! verschiedene Arten (Bild 10). Aus diesem Grunde ist das Netz für alle Permutationen von Punkt-zu-Punkt-Verbindungen blockierungsfrei. Dabei besteht es, wie das bereits bekannte Benes-Netz [1], aus nur (N/2)(2*log N - 1) Schaltern, die aufgrund ihrer möglichen Stellungen - parallel oder gekreuzt- als Kreuzschalter bezeichnet werden. Aufgrund der nicht quadratisch anwachsenden Zahl von Kreuzschaltern ergibt sich für große N eine erhebliche Einsparung an Schaltern verglichen mit einem Kreuzschienenverteiler. So müssen beispielsweise für N = 1000 Eingänge beim Multitop-Netz etwa 10 000 Schalter aufgewendet werden, während der Kreuzschienenverteiler bereits eine Million Schalter erfordert.

Das Multitop-Netz weist eine Selbstähnlichkeit in seiner Topologie auf, die bewirkt, daß die beiden Moduln eines bestehenden Netzes rekursiv in einem Netz mit der doppelten Zahl von Eingängen weiterverwendet werden können (Bild 11).

Bei einem Netz mit vier Eingängen beispielsweise können der linke und der rechte Teil dieses Netzes unverändert in einem Netz mit 8 Eingängen weiterverwendet werden. Dieser Sachverhalt gilt genauso für ein Netz mit 16 Eingängen u.s.w.

Die Selbstähnlichkeit der Topologie erlaubt weiterhin, ein bestehendes Netz modular zu erweitern, wobei mit jeder Verdopplung der Zahl der Eingänge sich auch die Kommunikations-Bandbreite verdoppelt. Diese modulare Erweiterbarkeit kann beispielsweise dazu genutzt werden, die Netto-Rechenleistung eines Multiprozessor-Systems nach dem Baukastenprinzip zu erhöhen.

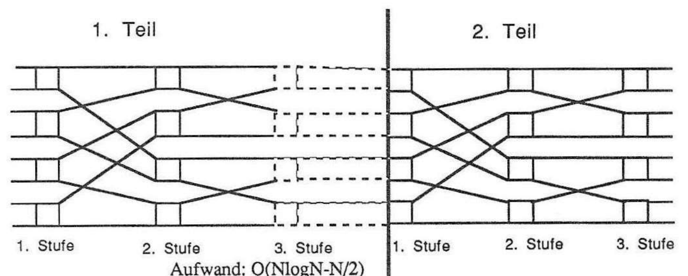


Bild 10: Multitop-Netz

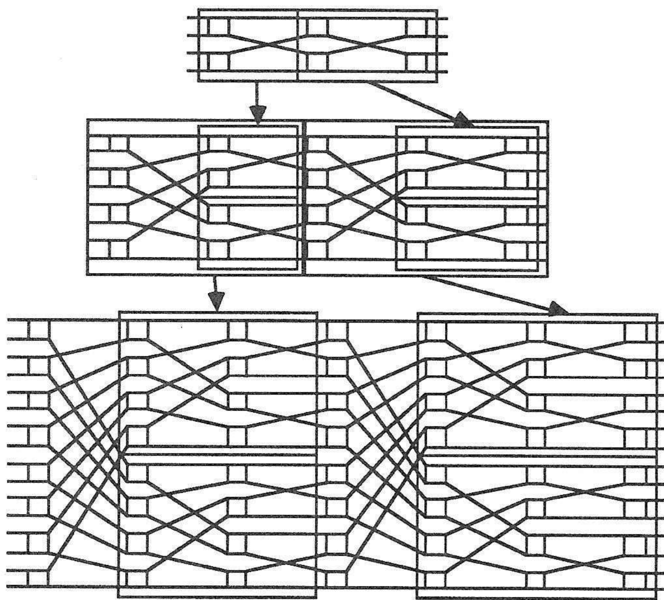


Bild 11: Modulare Erweiterung

Ein weiterer Vorteil ist beim Vergleich mit dem ebenfalls bekannten Lee-Netz [2] zu erkennen (Bild 12).

Da die Vermaschung im Netz von Stufe zu Stufe abnimmt, ist die Verdrahtung insgesamt wesentlich lokaler.

Schließlich kann man zeigen, daß bei Verwendung von Kreuzschaltern mit zusätzlicher Broadcast-Funktion das ganze Netz auch als Broadcast-Netz verwendet werden kann.

Wie funktioniert das Multitop-Netz?

Eine äquivalente Betrachtungsweise für die Frage der Wegewahl eines Koppelnetzes ist, wenn man das Netz als eine Maschine zum Sortieren von Zahlen ansieht (Bild 13). Am Eingang der Sortiermaschine wird eine Permutation von Zahlen eingespeist, am Ausgang erhält man die Zahlen in ihrer natürlichen Reihenfolge. Das Problem, Eingänge mit Ausgängen zu verbinden, ist also eng verknüpft mit der Aufgabe Zahlen zu ordnen. Dies führt zu dem analogen Problem, in einem Speicher Zahlen zu sortieren (Bild 14). Die adressierbaren Zellen des Speichers

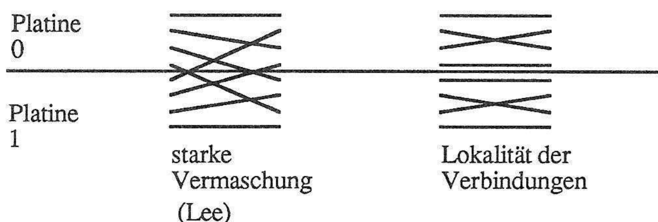


Bild 12: Geringe Vermaschung

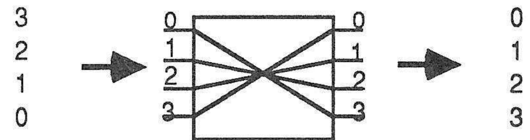


Bild 13: Sortiermaschine

entsprechen dann den verschiedenen Eingängen des Netzes.

Das analoge Problem ist deshalb einfacher zu lösen, weil Aussagen wie "die obere und untere Hälfte der Eingänge" oder "die zwei Eingänge eines Kreuzschalters" leicht durch die höchst- beziehungsweise niederstwertigen Adreßbits dargestellt werden können. Damit kann das folgende neue Verfahren zum Sortieren von Zahlen angegeben werden (Bild 15).

Im ersten Schritt werden aus den N Zahlen N/2 Paare gebildet. In jedem Paar sollen sich die zwei Zahlen nur in ihrem niederwertigsten Bit unterscheiden, das heißt ihre log N - 1 höherwertigen Bits sind gleich. Da es sich um eine Permutation von N Zahlen handelt, und N eine Zweierpotenz ist, gibt es zu jeder Zahl genau einen Partner, der diese Bedingung erfüllt. Die Paarbildung kann deshalb eindeutig und vollständig ablaufen.

Danach werden die beiden Zahlen jedes Paares dadurch getrennt, daß sie in die obere und untere Hälfte des Adreßraums verteilt werden. Dabei ist es irrelevant, welche Zahl in welche Hälfte kommt, wichtig ist nur, daß sie in verschiedene Hälften transportiert werden.

Der erste Schritt wird nun rekursiv auf die obere und untere Hälfte des Adreßraums angewandt, wobei jetzt 2x(N/4) Paare mit log N - 2 gleichen höherwertigen Bits gebildet werden. Der erste Teil des Sortierverfahrens terminiert nach log N - 1 Schritten, da dann keine Zahlen mit gleichen Bits im jeweiligen Adreßraum mehr existieren.

Der zweite Teil des Sortierverfahrens ist bereits bekannt. Er wird auch als sukzessive Approximation bezeichnet, da in jedem Schritt die Genauigkeit des Ziels verdoppelt wird. Nach log N Schritten ist das Ziel auf log N Bit genau.

Speicher Adresse	Inhalt		Adresse	Inhalt
0	3		0	0
1	2	→	1	1
2	1		2	2
3	0		3	3
vorher			nachher	

Bild 14: Zahlen im Speicher

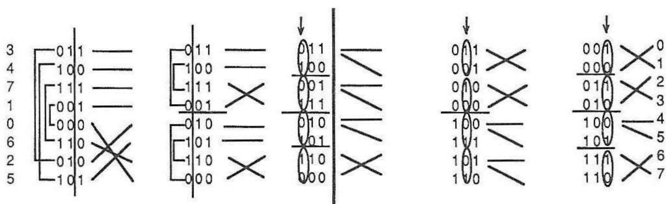


Bild 15: Sortierbeispiel

Neu ist, daß in jedem Schritt jeweils ein Paar benachbarter Zahlen in komplementäre Hälften verteilt werden kann, weil deren relevante Bits komplementär sind. Diese neue Eigenschaft wurde durch den ersten Teil des Sortierverfahrens erreicht. Damit verbunden ist aber auch die Möglichkeit, das Sortierverfahren mit Hilfe eines Netzes, das aus Kreuzschaltern und einer sogenannten Unshuffle-Verdrahtung besteht, zu implementieren (Bild 16). In diesem Netz ist die Unshuffle-Verdrahtung dafür zuständig, in verschiedene Hälften zu transportieren, während je nach Stellung des Kreuzschalters entschieden wird, in welche Hälfte transportiert wird.

Für ein blockierungsfreies Arbeiten des rechten Teils des Netzes ist es notwendig, daß an jedem Kreuzschalter zwei Zahlen mit komplementärem Steuerbit anliegen, da diese aufgrund der Unshuffle-Verdrahtung in komplementäre Hälften transportiert werden. Um dies sicherzustellen, ist der erste Teil des Sortierverfahrens beziehungsweise der linke Teil des Netzes notwendig. Die Idee, die dabei zugrunde liegt, entspringt folgender Beobachtung (Bild 17).

Wenn sich im rechten Teil des Netzes zwei Zahlen an einem Kreuzschalter der Stufe s treffen, müssen diese Zahlen s gleiche höherwertige Bits aufweisen. Denn beide steuern ihr Ziel nach dem Prinzip der sukzessiven Approximation an, wobei ihre s höherwertigen Bits als Steuerbits verwendet wurden.

Damit der Kreuzschalter, an dem die beiden Zahlen anliegen, konfliktfrei gesetzt werden kann, muß ihr s + 1. Steuerbit komplementär sein. Das Prinzip des linken Teils ist es also, Paare mit s gleichen höherwertigen Bits zu bilden, und die beiden Zahlen dann so weit zu trennen, daß sie sich im rechten Teil des Netzes erst nach s Stufen treffen können. Dazu müssen sie am Eingang des rechten Teils die Distanz 2^s im Adreßraum aufweisen. Denn wenn ihre Distanz kleiner als 2^s wäre, würden sie sich bereits vor

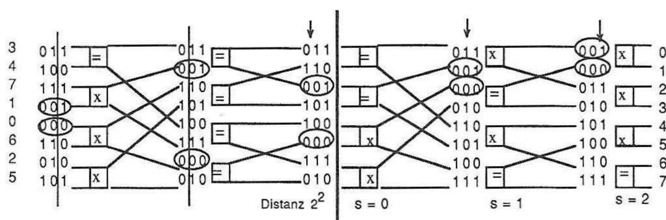


Bild 16: Implementierung

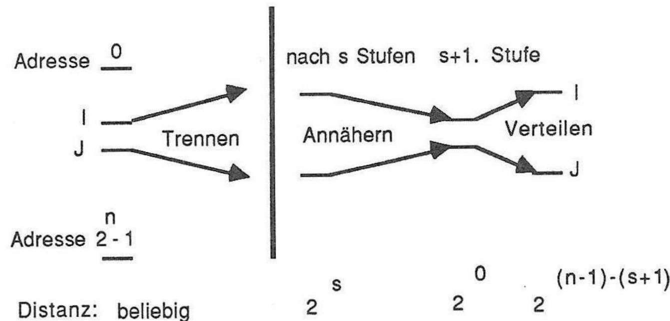


Bild 17: Blockierungfreier Weg

der Stufe s treffen. Die dann relevanten Steuerbits sind nach Voraussetzung aber nicht komplementär, so daß ihr Kreuzschalter nicht widerspruchsfrei gesetzt werden könnte. Der Algorithmus des ersten Teils des Sortierverfahrens beschreibt also einen Vorgang, der auch als "Konfliktpaartrennung" bezeichnet werden kann.

Entscheidend ist nun, daß ich zeigen konnte [3], daß dieser Algorithmus auf einem Netz derselben Topologie wie das für die sukzessive Approximation zuständige Netz ausgeführt werden kann. Deshalb besteht das Multitop-Netz aus zwei gleichen Teilen mit allen daraus resultierenden Vorteilen.

Ergebnisse

Es wurde ein Prototyp von Multitop mit vier Rechen-Transputern und dynamisch variabler Topologie realisiert. Die Implementierung der FFT auf diesem Prototyp sowie die Implementierung der Spline-Glättung lieferten bezüglich der drei betrachteten Probleme die folgenden Resultate:

1. Problem der parallelen Programmierung: Die verwendete Programmiersprache Occam basiert auf kommunizierenden sequentiellen Prozessen. Es wurde in der Praxis bestätigt, daß die parallele Programmierung dadurch vereinfacht wird, daß die Topologie der Prozesse direkt auf die Topologie der Prozessoren abbildbar ist. Auf maschinengegebene Besonderheiten mußte so keine Rücksicht genommen werden. Das Konzept der dynamisch variablen Topologie erwies sich deshalb als große Hilfe bei der Implementierung der parallelen Programme.

$E_{CT}(n,p) \approx 2^{n-1}(p+4)$ $\qquad E_{CT} = E_{GS}$ $\qquad p = \log P,\ n = \log N$

$E_{ST}(n,p) \approx \begin{cases} 2^{n-1}(p+6) \\ 2^{n-1}(p+5) \end{cases}$ $\qquad \frac{E_{PS}}{E_{GS}} = 2{,}75$ für $N = 1024,\ P = 16$

$E_{CT} = E_{GS} < E_{ST} < E_{PS}$ für $p + 6 < 2(n+1)$ $\qquad E_{PS}(n,p) \approx 2^{n-1}\, 2(n+1)$

Bild 18: Datenflußanalyse von sequentiellen Formen der FFT

	Zahl der Prozessoren 1	2	4
Gesamtzeit [ms]	478	276	152
Speedup	-	1,73	3,13
Wirkungsgrad	-	0,87	0,78
Transport/Rechn.	0,44	0,52	0,58

Bild 19: Messungen an Multitop

2. Problem der effizienten Ausführung: Die zu parallelisierenden Probleme wurden einer Datenflußanalyse zur Minimierung der Interprozessor- Kommunikation unterzogen. Es wurden dabei die vier sequentiell gleichwertigen Formen der FFT, die als Cooley-Tukey-(CT), Gentleman-Sande-(GS), Stockham-(ST) und Pease-FFT (PS) bezeichnet werden, als nicht gleichwertig bezüglich ihrer parallelen Ausführung erkannt, da deren Interprozessor-Kommunikation sich bis zum Faktor drei unterscheidet (Bild 18).

Es wurde daraufhin die Gentleman-Sande-FFT, als die Form mit der kleinsten Interprozessor-Kommunikation implementiert. Dabei zeigte sich ein hoher Wirkungsgrad von etwa 90 Prozent bei zwei Prozessoren beziehungsweise 80 Prozent bei vieren gegenüber einem Prozessor (Bild 19).

Um auf eine große Zahl von Prozessoren extrapolieren zu können, wurde ein mathematisches Modell entwickelt, in das die reine Berechnungszeit der FFT, die Interprozessor-Kommunikation sowie die Topologie der Prozessorverbindungen eingeht (Bild 20).

Die Topologie drückt sich darin durch die "mittlere Entfernung" zweier Prozessoren aus. Anhand des Modells zeigte es sich, daß das Multitop-Konzept mit d = 1 gegenüber dem Hypercube (d - LogP) und dem Ring (d - P/4) eine wesentlich größere Effizienz aufweist (Bild 21).

Für die zu parallelisierende Spline-Glättung mit Datenreduktion wurde zur Minimierung der Interprozessor-Kommunikation ein neues Verfahren entwickelt, das sich besonders gut parallelisieren läßt, da es ausschließlich auf Operationen der linearen Algebra, wie Matrix-Matrix-Multiplikationen basiert.

Gesamtzeit: $T_{FFT}(n,p,d) = 2^{n-p}(an + bd(1 + (p/4) - 2^{-p}))$

Daten-Transportzeit: $T_{MOV}(n,p,d) = bd2^{n-p}(1 + (p/4) - 2^{-p})$

Wirkungsgrad: $e(n,p,d) = \frac{n}{n + cd(1 + (p/4) - 2^{-p})}$

a,b : Konstanten
d: mittlere Entfernung
c = b/a

Bild 20: Modell zur Extrapolation

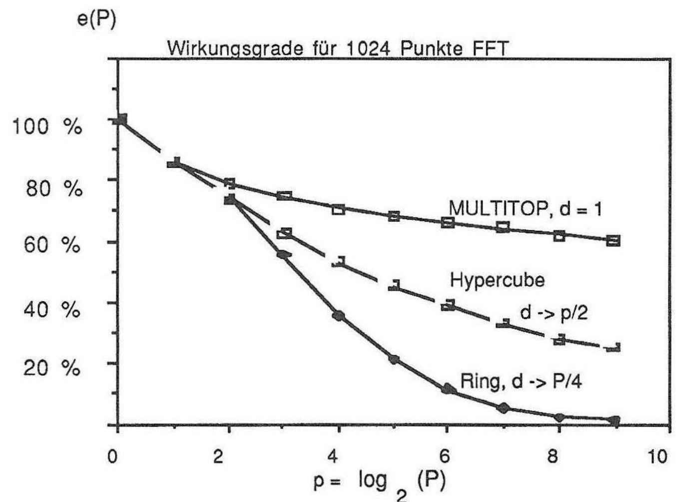


Bild 21: Hoher Wirkungsgrad bei Multitop

Insgesamt kann gesagt werden, daß eine hohe Effizienz bei paralleler Verarbeitung dadurch erzielt werden kann, daß die Interprozessor-Kommunikation sowohl auf algorithmischer Seite durch eine Datenflußanalyse als auch auf architektonischer Seite durch eine dynamisch variable Topologie verringert wird.

3. Problem der skalierbaren Leistung: Der Prototyp von Multitop wurde zwischenzeitlich auf acht Rechen-Transputer erweitert. Die dazu notwendige Verdopplung der Zahl der Eingänge des Koppelnetzes erfolgte unter Verwendung des vorhandenen Netzes. Das neu entwickelte Multitop-Koppelnetz erfüllt also die Forderung nach modularer Erweiterbarkeit.

Dr. Harald Richter

Buchtips

[1] V.E. Benes, Math. theory of connecting networks and telphone traffic, Academic Press, 1965.

[2] K.Y. Lee, IEEE Transactions on computers, Vol. c-34, No.5, May 1985.

[3] H. Richter, Multiprozessor mit dynamisch variabler Topologie, Diss., München 1988.

CHIP SPECIAL

1. Auflage 1989

Best.-Nr. 0999

Redaktionsdirektor: Richard Kerler

Chefredakteur: Armin Schwarz
(verantwortlich für den Inhalt)

Stellvertretender Chefredakteur: Ulrich Kern

Schlußredaktion: Ingeborg Kurz

Redaktionssekretariat: Petra Eibicht

Leserservice: Petra Eibicht

Autoren dieser Ausgabe: Herbert Anger, John Barnes, Heiko Czerwinski, Peter Eckelmann, Rolf Geisen, Hermann Göken, Andreas Hagerer, Dr. Winfried Hahn, Gerd Häußler, Ottmar Krämer, Falk-D. Kübler, Friedrich Lücking, Gottfried Nestyak, Ulrich Parthier, Sara Redfern, Dr. Harald Richter, Robert Sang, Wolf Schmidt, Bernd Schuster, Joachim Stender, Dr. Siegfried Streitz, Colin Whitby-Strevens

Titelgestaltung: Hans Kuh

Titelfoto: Michael Spakowski

Layout: Karlheinz Dereser

Redaktion: Vogel-Verlag und Druck KG
Redaktion CHIP-Special, Schillerstr. 23 a,
D-8000 München 2, Telefon (089) 51 4930,
Telekopierer 535000, Teletex (17) 897190

Verlag: Vogel-Verlag, Postfach 6740,
D-8700 Würzburg 1, Tel. (0931) 418-0,
Telex 68883, Telefax (0931) 418-2100.
Telegramme: CHIP-Würzburg

Verlagsdirektor: Dr. Andreas Kaiser

Anzeigenleiter: Friedrich Mangold, Würzburg
(verantwortlich für Anzeigen)

Anzeigenservice: CHIP, Postfach 6740,
8700 Würzburg 1, Tel. (0931) 418-0,
Telex 68883, Michael Belgrad,
Durchwahl 418-2433.

Vertriebsleitung: Axel Herbschleb, Würzburg

Vertrieb Handelsauflage: Vereinigte Motor-Verlage GmbH & Co. KG, Leuschnerstr. 1,
D-7000 Stuttgart 1, Tel. (0711) 2043-1

Bezugsmöglichkeiten: Bestellungen nehmen der Verlag und alle Buchhandlungen im In- und Ausland entgegen. Sollte die Zeitschrift aus Gründen, die nicht vom Verlag zu vertreten sind, nicht geliefert werden können, besteht kein Anspruch auf Nachlieferung oder Erstattung vorausbezahlter Bezugsgelder.

Bankverbindungen Vogel-Verlag:
Dresdner Bank AG, Würzburg
(BLZ 79080052) 3148890000,
Bay. Vereinsbank AG, Würzburg
(BLZ 79020076) 2506173,
Kreissparkasse Würzburg
(BLZ 79050130) 17400,
Postscheckkonto Nürnberg
(BLZ 76010085) 9991-853
Ausland: Postscheckkonto Zürich
8047064,
Niederlande 2662395
Banque Veuve Morin-Pons, Paris
155410314

Gesamtherstellung und Versand: VOGEL-DRUCK WÜRZBURG, Max-Planck-Str. 7/9,
D-8700 Würzburg

Unverlangte Manuskripte werden nur zurückgesandt, wenn Rückporto beigefügt ist. Für die mit Namen oder Signatur des Verfassers gekennzeichneten Beiträge übernimmt die Redaktion lediglich die presserechtliche Verantwortung.
Die in dieser Zeitschrift veröffentlichten Beiträge sind urheberrechtlich geschützt. Übersetzung, Nachdruck, Vervielfältigung sowie Speicherung in Datenverarbeitungsanlagen nur mit ausdrücklicher Genehmigung des Verlages.
Jede im Bereich eines gewerblichen Unternehmens hergestellte oder benutzte Kopie dient gewerblichen Zwecken gem. § 54 (2) UrhG und verpflichtet zur Gebührenzahlung an die VG Wort, Abteilung Wissenschaft, Goethestraße 49, 8000 München 2, von der die Zahlungsmodalitäten zu erfragen sind.
Die Redaktion hat die Manuskripte und Programme sorgfältig geprüft. Für Fehler im Text, in Schaltbildern, Aufbauskizzen, Listings usw. sowie deren Folgen kann keine Haftung übernommen werden. Sämtliche Veröffentlichungen erfolgen ohne Berücksichtigung eines eventuellen Patentschutzes, auch werden Warennamen ohne Gewährleistung einer freien Verwendung benutzt.

CIP-Titelaufnahme der Deutschen Bibliothek

Parthier, Ulrich:
Der Transputer und seine Programmiersprachen / [Autor dieser Ausg.: Ulrich Parthier]. – 1. Aufl. – Würzburg: Vogel, 1989
(Chip special) ISBN 3-8023-0999-5
NE: HST

TRANSPUTER – TECHNOLOGIE – TRANSPUTER

mc Transputer – Karte

32 Bit Transputer T4/T8 auf Eurokarte,256k SRAM oder 128k SRAM + 128k EPROM, 1 paral., 2 seriel. Schnittstellen.Herausgeführte Links ermöglichen beliebige Systemtopologien, 5V Spann.versorgung,Bus und Steuersignale auf 96 pol.VG.Messerleiste abgreifbar, Monitorprogram im Eprom wird mitgeliefert (RS232 Booting), PC – Entwicklungssystem wird mitgeliefert (Assembler, Linker, Server,..). Ideal für profess. industrielle Steuerung. Mit PC – Link – Adapter in jedem XT/AT als B004, Inmos – Karten – kompatibel.

Ab 1633,– o.MwSt

Transputer – C – Compiler

C – CrossCompilersystem für T4/T8, läuft auf jedem IBM – PC oder kompat.. Der Compiler erzeugt Assembler Source Code, der mit mitgeliefertem CrossAssembler weiterverarbeitet werden kann. Code für den T4/T8 opt.. Vorhandene C – Programme können ohne Aufwand auf die neue Hardware portiert werden, müssen nur neu compiliert, assembliert u. gelinkt werden.

Ab 660,– o.MwSt

Transp. – Modulmotherboard PO1

Das Modulmotherboard PO1 ist für den Aufbau eines flexiblen Netzwerkes mit bis zu 8 Inmos kompatiblen Modulen (TRAMS) bestückbar. Alle Linkverbindungen sind über Linkumschalter C004 geführt, die ein T212 mit 64kByte Speicher steuert. Zusätzlich sind auf der Karte 4 Linkadaptoren C011 einsetzbar, mit denen parallele Schnittstellen ins Netz integriert werden. Anwendungen in der Bildverarbeitung u. Simulation neuronaler Netze.

Platine mit D2, Sockel 2180,– o.MwSt

VME – Transputer – Board TP3/TP5

Alle Transputer auf Doppeleurok. VMEbus – Anschluß, TP3 als erste industrielle E/A – Karte, eine Einheit mit 3x T222 und Modulen für Achsenregelung, hochdynam. Antriebe (als Basis – Baugr. ist TP3,TP5). VME – TP5 – Karte als Subprozessor 5x T8, 6MB RAM, EPROM, V24 seriel. Schnittstelle, konfigurierbares Koppelfeld, VMEbus – Kopplung über 2 Link – Adapter, interruptfähig zum VMEbus. Standardmäßig mit T4/T425 bestückt. PC – Anschluß über die PC – TP.

16998,– o.MwSt

HTS/E1 steuert ECB – Bus

32 BIT T4/T8 bis zu 30 MHz, 256 KB bis 4 MB(!) RAM, 128 KB EPROM. Die ECB – Bus – Steuerung ermöglicht den direkten Austausch Ihrer 8 – Bit CPU Karte gegen unsere 100 – fach leistungsfähigere Transputer – Karte. Alle ECB – EO – Karten können weiter genutzt werden, sogar das IM2, Daisy chain mit RETI, NMI & WAIT werden unterstützt.
Mit PC – Link – Adapter in jedem XT/AT als B004 – Inmos – kompatibler Co – Prozessor zur Programmentwickl. in OCCAM,PASCAL,C,FORTRAN u. Assembler einsetzbar; ideal für Steuerungsapplikationen, Bildverarbeit., Problemlösung in Meß – /Regeltechnik.

2350,– o.MwSt

Multi – Transputer – System MTS – 6

Doppel – Eurok., 6 Transputer T4/T8 – 20MHz, 6x 1MByte DRAM, C004 per Link konfigurierbar, 14 gepufferte Linkanschlüsse 5,10,20MBit/s, 60 MIPS, 9 MFLOPS. MTS – 6 einsetzbar wo hohe Rechenleistung bei flexibler Topologie gefordert.

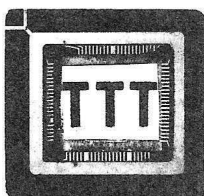
19940,– o.MwSt

Transputer – Karte für SMP – Bus

Uni – SMP – Bus als Zentralbaugruppe für SMP – Bus verwendbar. T2/T222 16 Bit, 32 KB SRAM, 16KB ON – BOARD – RAM über SMP erreichbar, I/O ICs ansprechbar, I/O – Zugriff mit 8/16 Wortbreite, interruptfähig, Hold – Logik koordinert den Zugang zum SMP – Bus bei DMA – Betrieb, Karte ersetzt Ihre Zentraleinheit (Intel 8080/85/88).

1680,– o.MwSt

*** Unsere Applikationsabt. steht Ihnen zur Verfügung.**
*** Kostenlose Datenblätter, technische Fragen bitte an:**



Transputer Technologie Transfer

Postanschrift: TTT M. Dreyer Postf.8525 D – 4800 Bielefeld 1
Tel (05 21) 28 78 12 Fax: (05 21) 89 45 74 .

TRANSPUTER – TECHNOLOGIE – TRANSPUTER

TTT – TRANSFER – TECHNOLOGIE – TRANSPUTER – TTT – TRANSFER – TECHNOLOGIE – TRANSPUTER – TTT – TRANSFER – TECHNOLOGIE

TTT – TRANSPUTER – TECHNOLOGIE – TRANSFER – TTT – TRANSPUTER – TECHNOLOGIE – TRANSFER – TTT – TRANSPUTER – TECHNOLOGIE

IBM UND KOMPATIBLE

Pohl/Bader
Der schnelle GEM-Einstieg*
0955 DM/sfr. 28,-, öS 230,-

Hofer/Straub
IBM PC, Kompatible, Ausg. 1
0917 DM/sfr. 28,-, öS 230,-

Kern
IBM PC, Kompatible, Ausg. 2
0070 DM/sfr. 28,-, öS 230,-

Mutschler/Holighaus
Netze/Lokale PC-Netzwerke und Kommunikation
0957 DM/sfr. 49,-, öS 380,-

Görgens
PC-Datenaustausch
0530 DM/sfr. 28,-, öS 230,-

Körber
Atari 260 …
0230

CHIP-SPECIAL-…
Atari ST, Ausg. …
0390 D…

Müller/Cambeis
Atari ST User Guide
0470 DM/sfr. …

CHIP-SPECIAL-Team
The Best of Atari ST (GFA BASIC), Ausg. 4
0490 DM/sfr. 28,-, öS 230,-

Schupp **NEU**
Desktop-Publishing mit Atari ST
0979 DM/sfr. 28,-, öS 230,-

Simon
Atari ST Pascal plus 2.0
0956 DM/sfr. 28,-, öS 230,-

CHIP-SPECIAL-Team
The Best of Atari XL/XE
0480 Sonderpreis DM 7,50

Schreiber
Amiga, Ausgabe 1
0290 DM/sfr. 28,-, öS 230,-

Kunz
Amiga, Ausgabe 2
0410 DM/sfr. 28,-, öS 230,-

Körber
Amiga Handbuch Grafik Anwendungen, Ausg. 3
0550 DM/sfr. 28,-, öS 230,-

Rahlff/Knappe
C auf dem Amiga, Ausg. 4
0640 DM/sfr. 28,-, öS 230,-

Rahlff/Knappe
Amiga 2.000, Ausgabe 5
0980 DM/sfr. 28,-, öS 230,-

Seibold
User-Guide zum Prozessor 68000
0590 DM/sfr. 28,-, öS 230,-

CHIP-SPECIAL-Team
Die schönsten Sportspiele für C64/C128
0380 Sonderpreis DM 7,50

Rügheimer/Spanik
PEEK POKE C64
0280 DM/sfr. 18,-, öS 150,-

…TANDARD SOFTWARE

…olff **NEU**
…sktop-Publishing mit …ntura Publisher
…85 DM/sfr. 28,-, öS 230,-

…llmuth
…PEN ACCESS II
…520 DM/sfr. 28,-, öS 230,-

…rbert
…ase, Ausgabe 1
…981 DM/sfr. 49,-, öS 380,-

CHIP-SPECIAL-Team
PC Public-Domain*
0970 DM/sfr. 34,-, öS 280,-

SPEZIELLE COMPUTER

Kunz
Apple IIGS, Ausgabe 1
0570 DM/sfr. 28,-, öS 230,-

CHIP-SPECIAL-Team
Philips: YES, Ausgabe 1
0330 Sonderpreis DM 7,50

Kunz
Philips: YES, Ausgabe 2
0510 Sonderpreis DM 7,50

CHIP-SPECIAL-Team
SHARP MZ 700/800, Ausgabe 1
0030 Sonderpreis DM 7,50

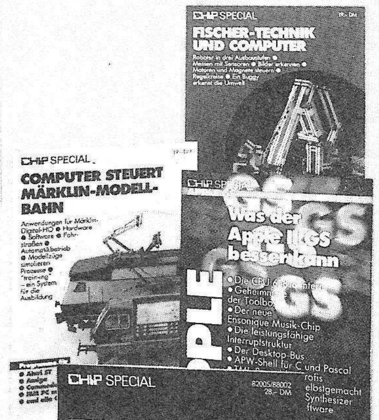
CHIP-SPECIAL-Team
SHARP PC 2500, 1500, 1401/2, 1350
0160 Sonderpreis DM 7,50

Bader
Computer steuert Märklin-Modellbahn
0953 DM/sfr. 19,-, öS 150,-

Bader
Fischer-Technik und Computer
0952 DM/sfr. 19,-, öS 150,-

Vollmuth/Müller
MIDI/Computer und Musik
0966 DM/sfr. 28,-, öS 230,-

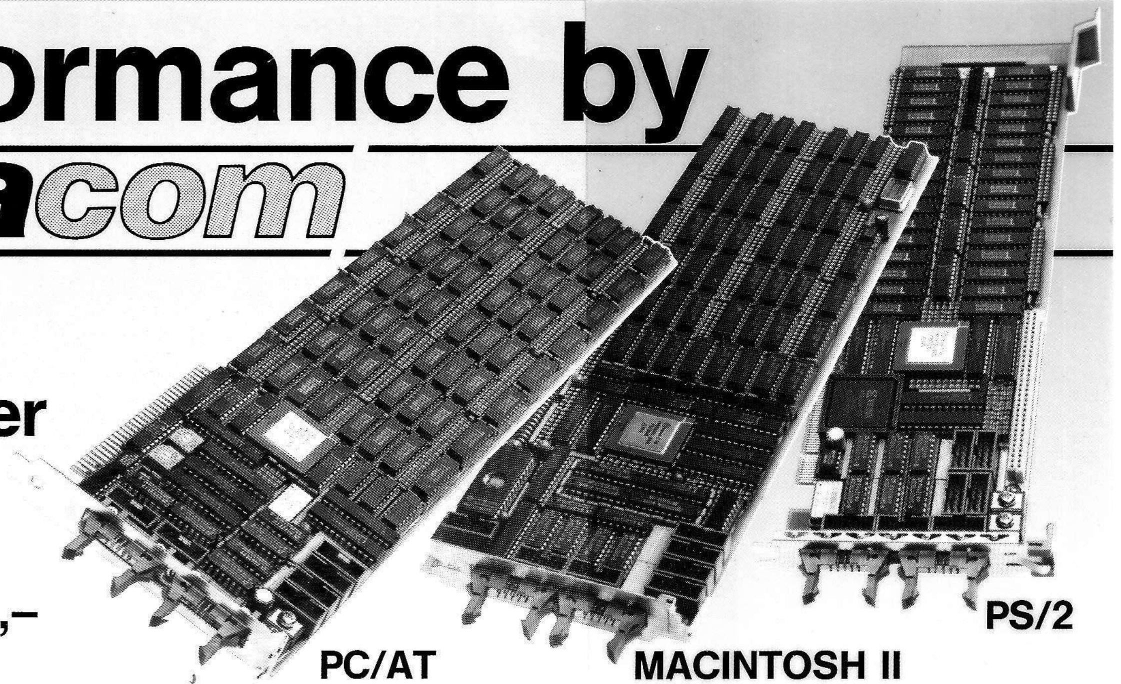
CHIP-SPECIAL-Team
Schneider-Programm CPC 464, 664, 6128: Brot und Spiele
0370 Sonderpreis DM 7,50



CHIP SPECIAL

Für jeden das Richtige.

Performance by paracom



Transputer Starter Kits
ab DM 2964,–
inklusive C-Compiler

PC/AT MACINTOSH II PS/2

- T800/414 20 MHz, 256K bis 8 MB Speicher
- flexible Aufsteckmodule
  - bis 3 weitere Transputer
  - elektronische Konfiguration
  - I/O-Schnittstellen für RS 232, IEC, Centronics...
- über 20 weitere Transputerkarten für VME, Q-bus, SUN, SCSI, Graphics, Video I/O, Parallel I/O u. a.

paracom GmbH
Gesellschaft für Parallele Computer Systeme
Jülicher Straße 338 · D-5100 Aachen
Tel. 0241/1660010 · Telefax 0241/1660050

PARACOM – Vertrieb und Anwendungsunterstützung der PARSYTEC GmbH

Inhalt

Hardware für Programmierer
Software-Aspekte bei Transputern

Informatik-Grundkurs
Schaltalgebra – Minimierung logischer Funktionen

Programmierwerkzeuge
TopSpeed, ein Modula-2-System für den PC

Programmierwerkzeug
Nachdokumentation mit Petrinetzen

Aufbaukurs Informatik
Formale Spezifikation von Software

Software-Know-how
Die Hash-Methode

Philosophische Grundlagen
Die fraktale Struktur der Evolutionen

Berufsbild
Gesucht: Systemprogrammierer/in

Künstliche Intelligenz
Denken ist kontrolliertes Tolerieren

In dieser Ausgabe:

Autoren: Herbert Anger, John Barnes, Heiko Czerwinski, Peter Eckelmann, Rolf Geisen, Hermann Göken, Andreas Hagerer, Dr. Winfried Hahn, Gerd Häußler, Ottmar Krämer, Falk-D. Kübler, Friedrich Lücking, Gottfried Nestyak, Ulrich Parthier, Sara Redfern, Dr. Harald Richter, Robert Sang, Wolf Schmidt, Bernd Schuster, Joachim Stender, Dr. Siegfried Streitz, Colin Whitby-Strevens

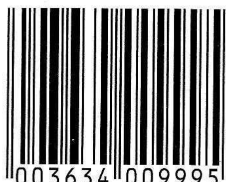
Grundlagen
Die zweite Generation
Die Programmiersprache Occam
Ergänzende Datenstrukturen in Occam
Rekursion in Occam

Compiler
Der C-Compiler Par.C und seine Anwendung
Parallele Programmierung in CS-Prolog
Die Programmierung des Transputers
Parallelverarbeitung und Ada
Alternativen zu Occam

Programmierumfeld
Das Betriebssystem Helios
Software-Entwicklung mit Spectral

Projektrealisierung
Konfigurierbare Transputerarchitekturen
Der Amiga 2000 und seine Transputer-Implementierung
Der Atari-Transputer unter Helios
Occam-Alternativen im Netzwerk

Forschungsprojekte
Ein Multi-Transputernetz für die Simulation digitaler Systeme
Multitop, ein Multiprozessor mit dynamisch variabler Topologie



ISBN 3-8023-0999-5