Hochschule für Architektur und Bauwesen Weimar
– Universität –
Fakultät Informatik und Mathematik

Parallelverarbeitung

– Report –

Marko Meister[‡] **Jan Springer**[‡‡]

students of applied computer science
Fakultät Informatik und Mathematik
4/91/$\mathcal{A}$

Weimar, den 13. Januar 1997

[‡] meister1@nessi.informatik.hab-weimar.de.
[‡‡] springer@nessi.informatik.hab-weimar.de.

Abstract

Diese Arbeit beruht auf den Erfahrungen, die im Rahmen der Lehrveranstaltung "Parallelverabeitung" im Sommersemester 1994 an der Fakultät Informatik und Mathematik der Hochschule für Architektur und Bauwesen Weimar -Universität- gesammelt wurden.

Die Autoren hatten die Aufgabe, sich einerseits in eine parallele Programmierumgebung einzuarbeiten und Prototypen für den praktischen Teil der Lehrveranstaltung zu entwickeln, andererseits ihre eigenen Erfahrungen und Kenntnisse im Umgang mit der ausgewählten Programmierumgebung an ihre Kommilitonen zu vermitteln.

Dieser Report soll den Studierenden Hilfe und Anleitung im Umgang mit parallelen Programmierumgebungen bieten. Er soll die von den Autoren gesammelten Erfahrungen weitergeben, um zu effizienteren Lösungen und neuen Ansätzen im Bereich der Anwendung der Informatik zu gelangen.

Inhaltsverzeichnis

I Einführung **1**

1 Grundlagen und Ziele **2**

2 Voraussetzungen **2**
2.1 Hardware . . . 2
2.2 Software . . . 3

3 p4 – Erste Schritte **4**
3.1 Beschaffung . . . 4
3.2 Installation . . . 4
3.3 Benutzung . . . 5

II Softwareentwicklung **8**

4 Sequentielle Softwareentwicklung **9**
4.1 Probleme . . . 9
4.2 . . . und ihre Lösung . . . 9

5 Parallele Softwareentwicklung **9**
5.1 Vorbemerkung . . . 9
5.2 Frühe Entwurfsphase . . . 9
5.3 Prototyping vs. "Papierprogrammierung" . . . 10
5.4 Testen . . . 10

6 Beispiele **11**
6.1 Eine parallele Primzahlensuche . . . 11
6.2 Ein paralleles Mandelbrot–Programm . . . 11
6.3 Weitere parallele Probleme . . . 13

7 Verfahren der Parallelisierung **14**
7.1 Vorbemerkung . . . 15
7.2 Synchrone Parallelität . . . 15
7.3 Asynchrone Parallelität . . . 15
7.4 Versuch einer Bewertung . . . 15

III Seminar **17**

8 Erfahrungen im Seminar **18**
8.1 Vorbemerkung und Zielstellung . . . 18
8.2 Auswahl der Aufgaben . . . 18
8.3 Durchführung der praktischen Arbeit . . . 18

9 Auswertung des Seminars **19**
9.1 Resultate . . . 19
9.2 Didaktische Erkenntnisse . . . 19

IV Was vom Tage übrigblieb **20**

10 Artificial Life **21**

11 Weiterführung **21**

12 Erweiterte Systemumgebungen **21**

13 Über uns **22**

V Anhang **23**

A Exploiting the Paralellism Available in Loops **24**
A.1 Program dependences 24
A.2 Program transformations 25

B Quellen im InterNet **27**
B.1 News–Gruppen 27
B.2 `ftp`–Server 27
B.3 $\mathcal{WWW}$–Ressourcen 27

C `p4` Funktionsreferenz $\mathcal{C}$ **28**

D Ein simples `p4`–Programm **29**

E Das Primzahlen–Programm **30**

F Das Mandelbrot–Programm, statische Variante **35**

G Das Mandelbrot–Programm, dynamische Variante **39**
G.1 Das Includemodul `mandel.h` 39
G.2 Das Hauptmodul `main.c` 40
G.3 Das Mastermodul `master.c` 40
G.4 Das Slavemodul `slave.c` 45
G.5 Das Utilitymodul `misc.c` 47
G.6 Das Prozeßzuordnungsmodul `mandel.pg` 51

Literaturverzeichnis **52**

Abbildungsverzeichnis

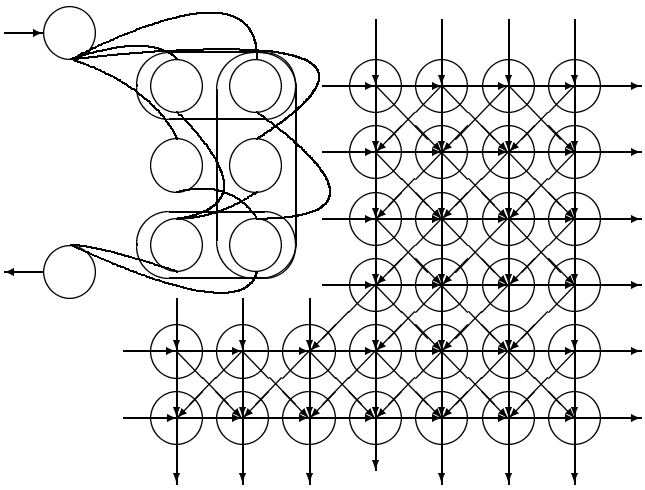
1 Kommandozeilenoptionen eines `p4`–Programms 7
2 graphische Ausgabe des Mandelbrot–Programms 12
3 Kommandozeilen–Parameter des Mandelbrot–Programms 13
4 statistische Ausgabe des Mandelbrot–Programms 14
5 Sequentieller Programmablauf 14
6 Paralleler Programmablauf, Statische Verteilung 16
7 Paralleler Programmablauf, load balancing 16
8 Paralleler Programmablauf, dynamic load balancing 16
9 `p4` Funktionsreferenz $\mathcal{C}$ 28

Tabellenverzeichnis

1 Maximum speedups for loops with cyclic dependence graphs 26

Teil I

Einführung



1 Grundlagen und Ziele

In der geschichtlichen Entwicklung der Rechentechnik war es ein Grundgedanke, Algorithmen parallel auszuführen. Erinnert sei hier nur an die von Babbage entwickelten Rechenmaschinen. Im Zuge der Formalisierung dieses Wissenschaftszweiges mit Hilfe der Algorithmen– und Komplexitätstheorie wurde schnell klar, daß bestimmte Problemklassen sich effizient nur mit Hilfe paralleler Programmierabstraktionen verwirklichen lassen konnten.

Seit der Entwicklung der Großrechentechnik Mitte der 40er Jahre dieses Jahrhunderts wurde dieser Ansatz der Informationsverarbeitung jedoch verdrängt. Der beherrschende Ansatz war der Sequentielle. Erst später gelangte die Parallelverarbeitung zu einer gewissen Renaissance, und zwar als klar wurde, daß die sequentielle Informationsverarbeitung tendenziell an ihre Grenzen gestoßen war.

Ein anderes Problem sind die auch heute noch immensen Kosten für Hardware, die Parallelverarbeitung a priori unterstützt. Der Gedanke, vorhandene, vernetzte Hardware für die Lösung komplexer Probleme einzusetzen, liegt deshalb sehr nahe. Im folgenden soll der an der Fakultät Informatik und Mathematik der Hochschule für Architektur und Bauwesen Weimar -Universität- beschrittene Weg beschrieben werden, solche vorhandene, vernetzte Hardware für Parallelverarbeitung zu nutzen. Dabei sollen vor allem die praktische Umsetzung und Anwendung im Vordergrund stehen.

Im Sommersemster 1994 wurde von Dr. Bernd Schalbe eine Vorlesung über Parallelverarbeitung gehalten. Beinahe zeitgleich beschäftigten sich die Autoren mit Mechanismen der Interprozeßkommunikation. Als Mentor stand dafür Dr. Günther Schatter zur Verfügung. Sehr bald stellte sich die enge Verwandschaft dieser beiden Gebiete heraus. Deshalb war die Aufgabe der Autoren von nun an, sich der praktische Seite der Parallelverarbeitung zu widmen. Die Ziele die dabei festgelegt wurden waren:

- den Kommilitonen Möglichkeiten zu bieten, praktische Parallelverarbeitung zu betreiben;
- wenn möglich eine Bibliothek zu erstellen, die weiterverwendbar ist.

Nach kurzer Zeit stellte sich jedoch heraus, daß es mit sehr großem Aufwand verbunden wäre, eine solche Bibliothek selbst zu erstellen. Deshalb wurde eine andere Richtung eingeschlagen. Die Autoren begannen zunächst nach Systemen zu suchen, die diese Aufgabe lösen konnten. Dabei wurden neben Hinweisen der Mentoren auch Fachliteratur und natürlich das Internet benutzt. Nachdem die Autoren so eine ganze Reihe von Systemen gefunden hatten, begann die eigentliche Arbeit. Im Kapitel 2.2 wird das weitere Vorgehen näher beschrieben. Zu diesem Zeitpunkt wurden die Ziele des Projektes noch einmal überarbeitet:

- Dokumentation der Informationsbeschaffung
- Beschaffung, Installation und Test der einzelnen Systeme
- ein System auswählen und Material für ein Seminar vorbereiten
- den Kommilitonen die Möglichkeit bieten, praktische Parallelverarbeitung zu betreiben
- den Kommilitonen die Informationsbeschaffung und Installation eines Systemes beispielhaft zu verklickern

2 Voraussetzungen

2.1 Hardware

Am Anfang aller Überlegungen, Parallelverarbeitung zu betreiben, steht das Problem der Hardware. Es wurde ein an der Fakultät Informatik und Mathematik existierender Workstationcluster benutzt. Diese besteht aus 8 *HP* Apollo 715/33 Workstations mit 32 MByte Hauptspeicher, 8 *HP*

Apollo 712/60 Workstations mit ebenfalls 32 MByte Hauptspeicher und einer *HP* Apollo 715/50 mit 80 MByte Hauptspeicher. Diese Geräte sind über ein ThinEthernet miteinander verbunden, deren Entrypoint in den Cluster die Apollo 715/50 ist. Die logische Netzstruktur ist linear.

Auf diesem Rechnern läuft das UNIX–Derivat *HP*–UX 9.03 von *HP*. Es waren sowohl der systemeigene $\mathcal{C}$–Compiler, K&R und ANSI Unterstützung, als auch der $\mathcal{GNU}$ $\mathcal{C}/\mathcal{C}$++–Compiler in der Version 2.6.1 verfügbar.

Die weiteren Aussagen beziehen auf diese Hardwareplattform, da die Autoren ihre praktischen Erfahrungen dort gesammelt haben. Prinzipiell ist es aber auch möglich, jeden anderen Workstationcluster, ja sogar heterogene, vernetzte Workstations für diese Zwecke zu nutzen. Es liefen ebenfalls erfolgreiche Tests auf einem SGI–Cluster; darauf soll aber hier nicht näher eingegangen werden.

2.2 Software

2.2.1 Informationsbeschaffung

Wie im Kapitel 1 beschrieben, wurde der Gedanke, eine eigene Bibliothek zu entwickeln, aus Zeitgründen fallengelassen. Die Autoren konzentrierten sich in erster Linie auf vorhandene Software. Als Informationsquelle dienten dafür die Informationsdienste des Internet. Beispielhaft seien hier

- die News–Gruppe `comp.parallel`,
- die Internetbibliothek für mathematische Software NetLib und
- das World Wide Web ($\mathcal{WWW}$)

genannt. Eine genaue Aufstellung der Quellen befindet sich im Anhang B auf Seite 27.

Über Fachzeitschriften wurde weitgehend der Einstieg zu diesen Bezugsquellen gefunden. Da jedoch viele Zeitschriften benutzt wurden, ist es den Autoren nicht mehr möglich nachzuvollziehen in welchen was stand.

2.2.2 Softwarebeschaffung

Für die Beschaffung von freier[1] Software über das Internet eignet sich immer noch am besten `ftp`[2]. Wem der Umgang mit dem `file transfer protocol`–Programm zu kryptisch ist, der kann auch das $\mathcal{WWW}$ für diese Art der Softwarebeschaffung benutzen. Die Autoren wollen hier nur am Rande auf diese Möglichkeiten verweisen. Ein weitaus größeres Problem stellt die Installation der Systeme dar.

2.2.3 Installation, Bewertung und Auswahl

Im Ergebnis der Informations– und Softwarebeschaffung konzentrierten wir uns auf die folgenden Systeme. Gleichzeitig wird eine Bewertung vorgenommen. Grundlage dieser Bewertung war eine Probeinstallation und die Sichtung der jeweiligen Dokumentation. Dieser Teil ist bewußt etwas allgemein gehalten:

- `p4` 1.4
  `p4` ist eine Funktionsbibliothek mit FORTRAN– und $\mathcal{C}$–Schnittstellen. Die Routinen, welche zur Compilezeit eingelinkt werden, ermöglichen einfache Handhabung des Message–Passing und ein einfaches Debugging. Das Programmiermodell ist bewußt einfach gehalten und ermöglicht ein einfaches Einarbeiten.

[1] "Frei" im Sinne von nichtkommerziell, meistens jedoch nicht weniger professionell als kommerzielle Software.
[2] Für weitere Informationen sei auf die `man`–page oder ein UNIX–Handbuch verwiesen

– PVM 3.2.6
PVM[3] schafft die Möglichkeit ein Workstationcluster als *virtuelle parallele Maschine* zu abstrahieren. Dieses geschieht über einen vorkonfigurierten Dæmon, welcher die Kommunikation der Prozessoren in dieser virtuellen Maschine organisiert. Es stehen Schnittstellen in FORTRAN und $\mathcal{C}$ zur Verfügung. Die notwendigen Biblitheksroutinen werden zur Compilezeit eingelinkt.
Es stehen einige Erweiterungen, z.B. HeNCE[4] als grafisches Programmiersystem, XPVM als Visualisierung der virtuellen Maschine, zur Verfügung, die die Einsetzbarkeit des Systems erheblich verbessern, seine Komplexität aber auch stark ansteigen lassen.

Die Autoren wählten das System p4, weil es eine gewisse Robustheit und auch leichte Verständlichkeit aufwies, um es in der Lehre einsetzen zu können. Hinzu kamen Startschwierigkeiten bei der Installation und Konfiguration von PVM. Ein weiterer Grund ist die leichte Einbindung von p4 in Programme, da die p4–Bibliothek nur statisch zum Programm hinzugelinkt werden muß. In PVM muß zusätzlich zum Linken der Bibliothek zur Laufzeit ein Dæmon existieren, der dann die Kommunikation innerhalb der "virtuellen Maschine" übernimmt. Im Weiteren soll anhand des Systemes p4 beschrieben werden, wie man bei der Beschaffung, Installation und der Arbeit mit einem solchen System vorgehen kann.

3 p4 – Erste Schritte

Die Beschreibung der Installation bezieht sich auf eine Einzelinstallation. Für eine systemweite Nutzung empfiehlt sich eine einmalige systemglobale Installation. Im Workstation–Pool der Fakultät Informatik und Mathematik wurde folgende Installation vorgenommen:

– die Bibliothek befindet sich in `/usr/local/lib` und heißt `libp4.a`,

– die Header–Files befinden sich in `/usr/local/include/p4` und

– die p4–Distribution mit Dokumentation befindet sich unter `/usr/local/src/p4`.

3.1 Beschaffung

p4 ist z.B. via `ftp` von `info.mcs.anl.gov` unter `/pub/p4` zu besorgen. Fragen zu p4 können an die Entwickler gerichtet werden, E–mail: `p4@mcs.anl.gov`.

Als zusätzliche Möglichkeit sei hier noch die NETLIB Bibliothek genannt, welche über einen $\mathcal{WWW}$–Client via

```
http://www.netlib.org
```

zu erreichen ist. Siehe dazu auch Anhang B.3.

3.2 Installation

Die Installation ist in der Dokumentation sehr ausführlich beschrieben. Trotzdem sollen hier moch einige Hinweise erfolgen, wie bei einer Einzelinstallation[5] günstigerweise vorzugehen ist:

– das System wurde mit `ftp` beschafft und liegt nun als Archiv–Datei vor; es empfiehlt sich, eine Sicherungskopie dieser Datei anzulegen, um evtl. die Installation noch einmal durchführen zu können;

– entpacken des Archives mittels `uncompress` oder `gunzip`[6];
Bsp.: `gzip -d p4-1.4.tar.gz` ergibt `p4-1.4.tar`

[3] Parallel Virtual Machine
[4] Heterogenous Network Computing Environment, s. [2]
[5] Es hat sich jedoch bewährt, die Installation nur einmal global durchzuführen.
[6] bei Problemen mit `man uncompress` oder `man gunzip` Hilfe anfordern

- Aufspaltung der Archiv–Datei in die ursprüngliche Datei- und Verzeichnisstruktur mittels `tar -x`;
  Bsp.: `tar -xvf p4-1.4.tar` ergibt die ursprüngliche Verzeichnisstruktur
- die Dokumentation befindet sich im Verzeichnis . . . `/doc`;
- die weitere Installation sollte so durchgeführt werden, wie die Entwickler es beschreiben[7].

Eine genaue Beschreibung der Installationsschritte soll an dieser Stelle nicht erfolgen. Wer im Umgang mit UNIX `make` und `cc` einigermaßen geübt ist, sollte auch bei Schwierigkeiten keine Probleme haben, die `Makefiles` an die lokalen Gegebenheiten anzupassen. Eine Beschreibung wie dabei vorzugehen ist, kann man nur sehr schwer geben. Mit folgenden prinzipiellen Schwierigkeiten sollte gerechnet werden:

- die in dem/den `Makefile`(s) gesetzten Umgebungsvariablen, Pfade und Compileroptionen sind nicht oder nur teilweise für das benutzte System gültig
  1.) anpassen des/der `Makefile`s an die lokalen Gegebenheiten
  2.) wenn notwendig in Zusammenarbeit mit dem Sysadministrator
- es fehlen Bibliotheken im Betriebssystem oder das zu installierende System ist unvollständig
  1.) installieren der fehlenden Komponenten oder
  2.) Auswahl eines anderen Systems
- der Compiler weigert sich den Source–Code zu compilieren
  1.) feststellen der Source–Code Qualität (ANSI-$\mathcal{C}$ oder K&R)
  2.) setzen der entsprechenden Compiler–Schalter

Im speziellen Fall unserer Installation auf *HP*–UX bleibt anzumerken, daß es wichtig ist dem Compiler explizit mitzuteilen was für Source–Code er vor sich hat. Dies geschieht mit dem Compiler–Schalter `-Aa` für das Einschalten des ANSI–$\mathcal{C}$ Modus. Zusätzlich zu dieser Option muß das Symbol `_HPUX_SOURCE` definiert werden damit der Compiler beim Lesen seiner eigenen System–Header zurück in den K&R–Modus "switcht"; dies geschieht mittels `-D_HPUX_SOURCE` als Kommandozeilenoption des Compilers.

3.3 Benutzung

Die folgenden Ausführungen basieren auf [4].
`p4` ist eine Sammlung von Makros und Subroutinen, die vom Argonne National Laboratory entwickelt wurden, um damit eine Vielzahl paralleler Maschinen programmieren zu können. Die Entwickler sind Ralph Butler und Ewing Lusk. Die Bibliotheken existieren mit Interfaces zu den Programmiersprachen FORTRAN und $\mathcal{C}$. Im weiteren werden wir uns nur auf das $\mathcal{C}$–Interface beschränken.

`p4` wurde so konzipiert, daß es möglichst portabel ist. Es sollen also alle möglichen Rechnerarchitekturen — von echten Parallelrechnern über Workstationcluster bis hin zu Uniprozessormaschinen — unterstützt werden. Parallelverarbeitung auf Uniprozessorsystemen macht z.B. bei der Entwicklung von paralleler Software Sinn, da der Verbrauch von Ressourcen geringer ist als auf Parallelrechnern und der programmablauf der zu entwickelnden Applikation sich besser verfolgen läßt. Das System wurde von den Autoren in der Version 1.4 benutzt. `p4` unterstützt verschiedene Mechanismen der Interprozeßkommunikation. Im folgenden wird aber nur das Message–Passing behandelt werden. Message–Passing hat gerade bei Workstationclustern eine sehr große Bedeutung. Die Benutzung von Shared Memory wäre möglich, rentiert sich im allgemeinen nur auf enggekoppelten Systemen; in der Installation unter *HP*–UX mußte auf dieses Feature verzichtet

[7] [4, Kapitel Installation]

werden, obgleich p4 dieses für Uniprozessorsysteme anbietet, weil es nicht stabil funktionierte; der Fehler liegt wahrscheinlich innerhalb des Betriebssystems.

Wenn man das Programmiermodell beschreiben will, auf dem p4–Programme beruhen, so läßt sich das am einfachsten mit SPMD[8] tun. SPMD–Architekturen beruhen darauf, daß auf jedem Knoten (Prozessor) des (virtuellen) Parallelrechners eine Kopie eines einzigen Programmes läuft. Die Prozessoren können nun mit Hilfe einer eindeutigen Prozessor– bzw. Prozeßkennung entscheiden, welchen Teil des Programmes sie auszuführen haben.
Um diese Erkenntnisse praktisch anwenden zu können, muß man sich mit grundlegenden p4–Funktionen bekanntmachen:

- `p4_initenv (&argc,argv)`
  Diese Funktion initialisiert das p4–System; sie sollte die erste p4–Funktion sein, die in einem p4–Programm aufgerufen wird;

- `p4_wait_for_end ()`
  Diese Funktion schließt ein p4–Programm ab; diese Funktion wartet auf die Beendigung aller p4–Prozesse und räumt die Scherben weg, die vielleicht vorhanden sind;

- `p4_create_procgroup ()`
  generiert auf der Basis eines Processgroup–Files eine Menge von Slave–Prozessen; Erst nach Aufruf dieser Funktion, kann man davon ausgehen, daß Slaveprozeße gestartet wurden (d.h. das Programm entsprechend oft vervielfältigt wurde).

- `p4_get_my_id ()`
  ist eine Funktion, die die eindeutige Prozeßkennung des rufenden Prozeßes zurückliefert.

Wie man sieht, beginnen alle diese Funktionsaufrufe mit p4_, was für alle p4–Funktionen gilt. Siehe dazu im Anhang C die Funktionsreferenz der $\mathcal{C}$–Schnittstelle für p4.

Nun zu dem schon erwähnten Processgroup–File. In dieser Datei gibt der Nutzer explizit an, welche Rechner mit in den virtuellen Parallelrechner aufgenommen werden sollen. Dabei ist es prinzipiell auch möglich, auf einem Rechner (Prozessor) mehrere Slaves laufen zu lassen. Auf dieses Feature, der Benutzung von Shared Memory auf Uniprozessormaschinen, soll aus dem oben genannten Grund nicht näher eingegangen werden. Die Struktur dieser Datei ist wie folgt:

```
local n [full_path_name] [login_name]
remote_machine n [full_path_name] [login_name]
⋮
```

Dabei bedeutet das n hinter local, wieviele Slave–Prozesse auf dem lokalen Rechner zu starten sind. Im Normalfall ist diese Zahl 0, weil auf dem lokalen Prozessor nur der Master–Prozeß laufen soll. Die Zahlen hinter den Namen der angeschlossenen Hosts haben dieselbe Bedeutung. Sie sind bei den von uns betrachteten Beispielen immer mit 1 zu wählen. Der Pfadname spezifiziert die ausführbare Programmdatei auf diesem Rechner und Loginname ist der Loginname des Nutzers auf diesem System. Für den Nessi–Pool der Fakultät Informatik und Mathematik würde das dann etwa so aussehen (lokaler Rechner ist Nessi13):

```
local 0
#nessi13 1 $home/p4-1.4/examples/Mandel/mandel
nessi14 1 $home/p4-1.4/examples/Mandel/mandel
nessi15 1 $home/p4-1.4/examples/Mandel/mandel
```

Das # Zeichen am Anfang einer Zeile kommentiert den Rest der Zeile aus. Auf einer Multiprozessormaschine mit Shared Memory könnte das Prozessgroup–File so aussehen:

```
local 32 $home/p4-1.4/examples/Mandel/mandel
```

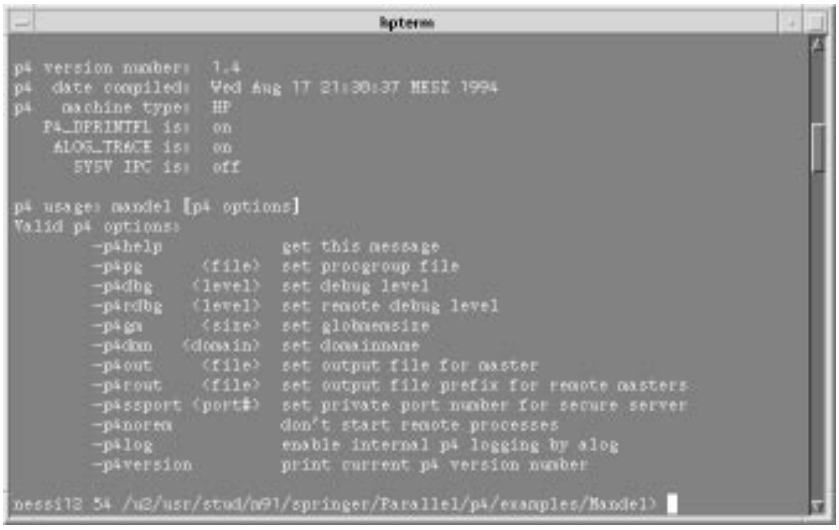


Abbildung 1: Kommandozeilenoptionen eines p4–Programms

Das würde bedeuten, das auf der lokalen Maschine der Master–Prozeß und 32 Slave–Prozesse gestartet werden.

Doch nun zur Praxis. Das Programm auf S. 29 ist das Einstiegsbeispiel für die Programmierung in p4. Dieses Programm läuft wie folgt ab:

1.) die Header–Datei `p4.h` muß eingefügt werden, um dem Compiler die Definition der p4–Bibliotheksfunktionen bekannt zu machen

2.) `p4_initenv()` initialisiert das Laufzeitsystem und parst die Kommandozeilenparameter nach p4–spezifischen Optionen ab[9]

3.) in der `main`–Routine werden als nächstes alle angeschlossenen Rechner, Slaves, initialisiert und syncronisiert; dies geschieht mittels der Funktion `p4_create_procgroup()`; dabei werden die Informationen benutzt, welche im Processgroup–File spezifiziert wurden

4.) nun liegen auf allen Hosts Kopien des Programms vor und jeder Slave startet "seine" `slave()`–Routine

5.) ist diese abgearbeitet, wird `p4_wait_for_end()` aufgerufen; der Prozeß wartet nun, bis alle anderen auch fertig sind und verabschiedet sich sodann aus dem Laufzeitsystem des Hostrechners

Das Programm wird folgendermaßen erzeugt[10]

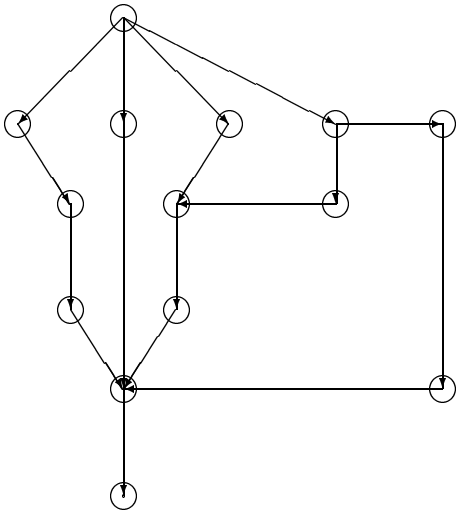
```
cc -o simple simple.c -I<Pfad zu p4-Header-Files> -L<Pfad zu p4-Library> -lp4 -lV3
```

Nun sollte eine ausführbare Datei namens `simple` existieren. Das zugehörige Processgroup–File heißt standardmäßig <Programmname>.pg; in diesem Falle also `simple.pg`. Beim Aufruf des Programms gibt nun — so Gott und das Betriebsystem will — jeder Slave seine Prozeß–ID aus.

Ein Beispiel zur Benutzung der Message–Passing Routinen befindet sich in [4, Kapitel 6.3].

[8] Single Program Multiple Data
[9] diese können bei jedem p4–Programm mit der Option `-p4help` abgerufen werden; siehe Abb. 1
[10] es empfiehlt sich ein Standard`Makefile` des p4–Systems zu adaptieren

Teil II
Softwareentwicklung



4 Sequentielle Softwareentwicklung

4.1 Probleme...

Die Entwicklung sequentieller Softwarelösungen ist grundlegender Bestandteil der Lehre und kann somit als bekannt vorausgesetzt werden[1].

4.2 ...und ihre Lösung

Als Lösungsmöglichkeiten bieten sich an:

- intuitives "debuggen" mit Ausgaben von aktuellen Wertbelegungen;
- Programmablaufpläne;
- Strukturpläne;
- Datenflußpläne;
- formale Korrektheitsbeweise von Algorithmen oder Teile von diesen;

Natürlich gibt es darüber hinaus weitere Verfahren, die in speziellen Fällen Anwendung finden können.

5 Parallele Softwareentwicklung

5.1 Vorbemerkung

Es gibt bis heute kaum Unterstützungswerkzeuge, wie man sie von der sequentiellen Programmierung her kennt. Gründe dafür sind ganz einfach die Komplexität und damit die Kompliziertheit der Umsetzung von parallelen Algorithmen. Es gibt einige gute Ansätze, die parallele Programmierung zu unterstützen. Diese Ansätze gehen jedoch andere Wege als in der sequentiellen Programmierung üblich. Es reicht hier nicht aus, programmiersprachenorientierte Editoren, Projektverwaltungen und simple Debugger bereitzustellen.
Diese Ansätze tendieren dahin mittels einer graphischen Repräsentation den Algorithmus zu formulieren. Nach dieser Repräsentation wird an den dafür vorgesehenen Stellen der entsprechende Programmcode eingefügt, wobei hier auf neuere Entwicklungen innerhalb der Entwicklung von Programmiermodellen zurückgegriffen wird (OOP, nichtanweisungsorientierte Konzepte etc.)[2]. Ein anderer Entwicklungsschwerpunkt werden sicherlich "parallelisierende" Compiler sein, die automatisch parallele Strukturen in herkömmlichen, sequentiell gedachten Programmen erkennen und umsetzen, sofern die Zielplattform dies zuläßt.

Wir wollen hier nun einige Fragen diskutieren, wie man denn nun "parallel" programmiert ohne diese ausgereiften Hilfsmittel aus der sequentiellen Programmierung zur Verfügung zu haben[3].

5.2 Frühe Entwurfsphase

Die frühe Entwurfsphase sollte klarstellen

- wie das zu bearbeitende Problem parallelisiert wird,
- welche Interfaces benutzt werden,
- wie die Datenabstraktion aussieht,

[1] vgl. Vorlesungen im Fachgebiet Softwaretechnologie.
[2] Als Vorbild könnte man sich die Programmierung unter NextStep denken.
[3] Dabei wird vieles wie ein alter Hut aus den Tagen von SWT Vorlesungen klingen. Richtig, aber auch hier zum großen Teil gültig.

– welche formalen Methoden der parallelen Datenverarbeitung benutzt werden sollen,

so daß hier schon ein klares Bild entsteht, wie der Algorithmus umzusetzen ist **und** welche Abhängigkeiten zwischen den zu bearbeitenden Daten existieren.

Als ein Problem hat sich das Auffinden von zu parallelisierenden Strukturen beziehungsweise von Abhängigkeiten herausgestellt. Dabei ist es hilfreich, folgende Abstraktion einzuführen. Man nimmt an, daß ein sequentielles Programm als lineare Abfolge von Schritten beschrieben werden kann, so daß das Bild eines Fadens mit Knoten entsteht und jeder Knoten, außer dem ersten, einen Vorgänger hat und jeder Knoten hat einen Nachfolger, außer dem Letzten. Es ist nun einsichtig, daß die Bearbeitung von Knoten X die Bearbeitung von Knoten $X-1$ ($\forall \quad X \geq 2$) voraussetzt. Im Falle paralleler Algorithmen ist es hilfreich, sich eine Fläche vorzustellen, in welcher sich die Knoten in einer mehr oder weniger regelmäßigen Struktur angeordnet befinden. Definiert werden ein Start- und ein Endknoten. Falls nun eine Verbindung zwischen zwei Knoten existiert, kann die Verarbeitung vom Vorgänger zum Nachfolger stattfinden. Haben nun ein oder mehrere Knoten ein oder mehrere Nachfolger, so ist der zeitliche Ablauf von Knoten zu Knoten nicht mehr vorhersehbar in dem Sinne, daß man weiß, welche Knoten zum Zeitpunkt t was bearbeiten. Es ist aber möglich mit Hilfe von DAG's[4] Konflikte in diesem "Netzwerk" zu finden, und diese gegebenfalls zu lösen.

5.3 Prototyping vs. "Papierprogrammierung"

Eine erschöpfende Antwort auf diese "Streitfrage" läßt sich eindeutig nicht geben. Das hängt damit zusammen, daß man immer aus dem Kontext des zu bearbeitenden Problems entscheiden muß, ob man sofort implementiert und im Nachhinein verbessert oder ob man erst mal auf dem Papier ausprobiert, wie es gehen könnte. Folgende Punkte sollten dennoch Beachtung finden:

1.) Läßt sich das Problem wiederum in Teilprobleme zerlegen, so ist es angeraten, erst einmal auf dem Papier auszuprobieren. Ist es hingegen *atomar* sollte mit schnellen Prototypen begonnen werden.

2.) Probleme der Lastverteilung sollten als Teil der zu lösenden Aufgabe betrachtet werden, d.h., daß mit dem Entstehen der Implementation auch die Art der Verteilung und deren Verwaltung in den Algorithmus einzubauen ist.

5.4 Testen

Bei synchroner Parallelität ist das Debuggen von parallelen Programmen eine relativ einfache Aufgabe, die mit den gleichen Methoden wie bei konventionellen, sequentiellen Lösungen angegangen werden kann. Der wesentliche Grund dafür ist das synchrone Laufzeitverhalten der Prozessoren. Neben den normalen, herkömmlichen Schwierigkeiten beim Testen von Software allgemein, hat man es bei parallelen Programmen mit einer neuen Qualität von Problemen zu tun. Insbesondere trifft das für die asynchron–parallelisierten Lösungen zu. Nicht nur, daß man theoretisch so viel Code testen und gegebenenfalls debuggen muß, wie man Prozessoren nutzen will, man muß vor allen Dingen die komplexen Auswirkungen der Nebenläufigkeiten bei voneinander abhängigen Daten beachten, voraussehen, und testen. Diese Aufgabe ist vollständig noch weniger lösbar als der Beweis der Fehlerfreiheit bei herkömmlichen Programmen. Niemand sollte aber an dieser Stelle die Flinte ins Korn werfen, nur weil hier so große Probleme aufgezeigt werden.

Der Test von paralleler Software — und das ist auch ein Ergebnis der Studien der Autoren — sollte schon damit beginnen, daß die frühen Phasen des Softwareentwurfes sehr sorgfältig durchgeführt werden. Natürlich spart man sich dadurch nicht die Testphase nach Fertigstellung des Produktes, aber man kann sich die Arbeit sehr vereinfachen.

Der praktische Test von paralleler Software wird meistens so aussehen, daß ein Prototyp auf seine Funktion getestet wird – wie im sequentiellen Falle auch. Um den speziellen Problemen bei parallelen Programmen zu begegnen, sollte man spezielle Tools verwenden. Für einfache Aufgaben

[4] Directed Acyclic Graphs; s. [7]

eignet sich aber auch das p4–System mit seinen umfangreichen, bereits implementierten Debuginformationen. Diese werden über explizite Funktionsaufrufe genutzt. Das Konzept beruht auf einer gestaffelten Ausgabe von Debuginformationen. Über den Kommandozeilenparameter -p4dbg <level> kann eingestellt werden, wie detailiert die Informationen sein sollen, die zur Laufzeit ausgegeben werden; siehe dazu auch Abblildung 1 Seite 7.

6 Beispiele

6.1 Eine parallele Primzahlensuche

Ein sehr schönes Problem aus dem Bereich der Zahlentheorie ist das Suchen bzw. das Bestimmen von Primzahlen.
Eine natürliche Zahl, die echt größer als eins ist, heißt Primzahl, wenn sie nur durch eins und durch sich selbst ohne Rest teilbar ist. Andernfalls heißt sie zusammengesetzt. Diese Definition impliziert nun eine Möglichkeit Primzahlen zu suchen, in dem man die Bedingungen überprüft, d.h. man dividiert die zu überprüfende Zahl durch alle ihre Vorgänger[5].

Der damit beschriebene Algorithmus läßt sich sequentiell leicht implementieren. Doch wo liegt der parallele Ansatz? Und hat dieser parallele Ansatz, so es ihn gibt, Vorteile?

Der parallele Ansatz besteht in der Erkenntnis, daß zum Überprüfen der Zahl X keine weiteren Ergebnisse benötigt werden. Man kann also im einfachsten Falle den Zahlbereich der zu überprüfen ist so aufteilen, daß jede natürliche Zahl in diesem Bereich für sich untersucht werden kann.

Do In Parallel Over X
  $test\_if\_prim(X)$
Enddo

Der Vorteil gegenüber der sequentiellen Abarbeitung liegt nun im Wesentlichen darin, daß man in der Lage ist mehrere Zahlen *gleichzeitig* zu untersuchen.
Zum besseren Verständnis sei auf die Beispielimplementation mittels p4 auf S. 30 verwiesen.

6.2 Ein paralleles Mandelbrot–Programm

6.2.1 Das Problem

Nunja, die Mandelbrotmenge zu berechnen ist ein sehr beliebtes und oft strapaziertes Problem. Hier soll nur eine kurze Beschreibung erfolgen. Die Grundidee ist, auf einem zweidimensionalen Gebiet (komplexe Zahlenebene) eine Funktion zu berechnen. Diese Funktion wird iterativ für jeden Punkt der komplexen Zahlenebene nach der Vorschrift

$n = 0$
$z_0 = 0$
Do
  $z_{n+1} = {z_n}^2 - c$
  $n = n + 1$
Until $|z| > lim$ **Or** $n > max$

berechnet, wobei gilt:

z,c ... komplexe Zahlen

max ... maximale Zyklenanzahl, Abbruchkriterium der Iteration

lim ... reelle Zahl, Abbruchkriterium der Iteration

Die Schleife wird solange durchlaufen, bis der Betrag der Zahl z über einen bestimmten Betrag (lim) anwächst. In diesem Fall gehört die Zahl c nicht zur Mandelbrot–Menge. Bleibt der Betrag

[5] Alle nun auch wieder nicht. Es reicht wenn alle Zahlen $\leq \frac{n}{2} + 1$ getestet werden. Siehe dazu auch [6].

der Zahl z immer unter dem Wert lim, ist die Zahl Element der Mandelbrot–Menge. In der Praxis bricht man die Berechnung nach einer endlichen Anzahl von Zyklen ab. Will man nun die Ergebnisse visualisieren, so kann man die Anzahl der benötigten Zyklen als Farbwerte in einem Rasterbild benutzen. Dabei kann man sich das Raster des Bildes auf die komplexe Zahlenebene gelegt vorstellen. Für jeden Rasterpunkt kann man nun die entsprechende komplexe Zahl als Ausgangspunkt für die Berechnung benutzen. Die Anzahl der benötigten Zyklen wird dann als Farbwert in das Rasterbild eingetragen. Logischerweise sind also die Punkte mit der höchsten Farbnummer (die Abbruchnummer max) diejenigen Punkte, die zur Mandelbrotmenge gehören.

Der interessierte Leser sei an dieser Stelle auf [14] und [21] verwiesen. [21] wird mit einer umfangreichen Dokumentation angeboten, die sich gut für das praktische Studium der fraktalen Welten eignet.

Hat man den Berechnungsalgorithmus begriffen, so sieht man leicht, daß jeder Punkt der komplexen Zahlenebene (oder anders ausgedrückt jedes Pixel des Rasterbildes) einzeln und unabhängig von allen anderen Punkten der Ebene berechnet werden kann. An dieser Stelle sollte der aufmerksame Parallelisierer aufhorchen.

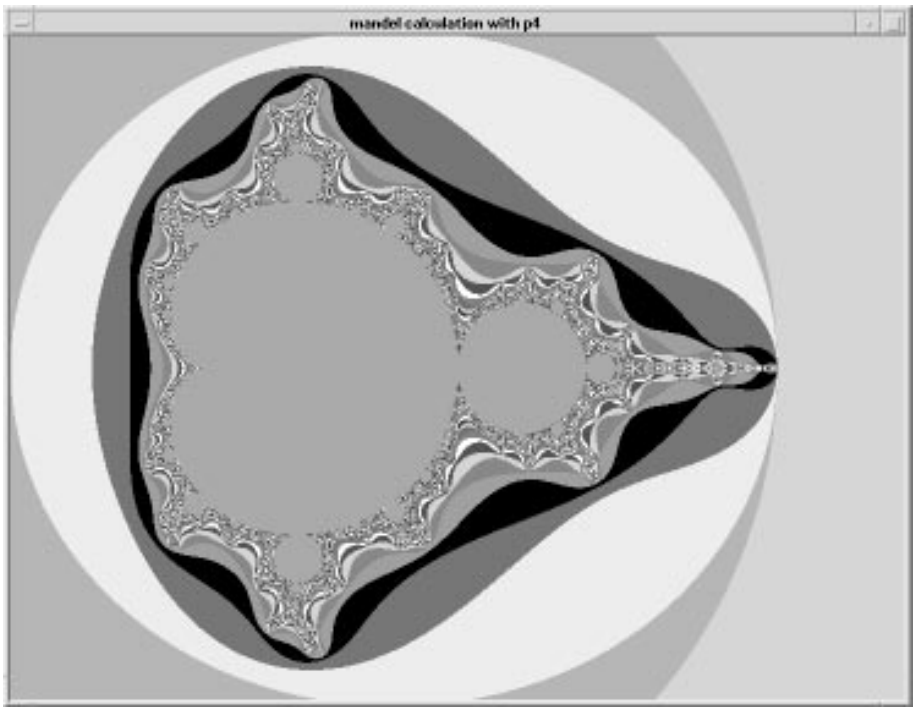


Abbildung 2: graphische Ausgabe des Mandelbrot–Programms

6.2.2 Die statische Variante

Nun zur Parallelisierung des oben beschriebenen Algorithmus. Im einfachsten Fall ist eine statische Verteilung der Aufgabe zu wählen. In der Praxis sieht das dann so aus, daß die Aufgabe in soviele Teile zerlegt wird, wie Slaves zur Verfügung stehen. Jeder Slave–Prozeß erhält dann genau eine Aufgabe. Der Master–Prozeß wird beendet, wenn der letzte Slave sein Ergebnis zurückgeliefert hat. Diese Methode hat den Vorteil, daß der Aufwand an Kommunikation zwischen Slaves und Master auf ein Minimum beschränkt wird. Natürlich macht eine solche Lastverteilung nur Sinn, wenn die zur Verfügung stehenden Maschinen alle in etwa die gleiche Leistungsfähigkeit haben, und die verteilten Aufgaben etwa den gleichen Berechnungsaufwand erfordern. Auch die

Netzwerkausführung und die Netzwerklaufzeiten spielen hier eine Rolle. Der große Nachteil dieser Methode ist, daß durch Laufzeitunterschiede zwischen den Slaves Leerlaufzeiten entstehen, die aus der unterschiedlichen Iterationstiefe der einzelnen Slaves resultieren.

Siehe Anhang **F**, S. 35ff. für den Quelltext.

6.2.3 Die dynamische Variante

Bei Problemen, bei denen der Rechenaufwand erst nach der Berechnung abgeschätzt werden kann, ist die statische Methode nur sehr beschränkt einsetzbar. Hier sollte eine dynamische Variante zum Einsatz kommen. Praktisch wird dabei so vorgegangen, daß die Aufgabe in unabhängige Teilaufgaben zerlegt wird. Dabei ist die Anzahl der Teilaufgaben bedeutend größer als die der Slaves. Nun werden Aufgaben verteilt, soviele wie Slaves vorhanden sind, und immer dann, wenn ein Ergebnis beim Master–Prozeß ankommt, wird an den nun arbeitslosen Slave eine neue Teilaufgabe abgeschickt; so lange noch Aufgaben vorhanden sind. Dieses Prinzip führt dazu, daß die Aufgaben so verteilt werden, daß praktisch nur geringer Leerlauf auftritt.

Siehe Anhang **G**, S. 39ff. für den Quelltext.

Abbildung 3: Kommandozeilen–Parameter des Mandelbrot–Programms

6.2.4 statistische Auswertung

Das Mandelbrot–Programm wurde von den Autoren dazu benutzt, um mittels statistischer Methoden Aussagen über optimale Problemgrößen und optimale Lastverteilung in lose gekoppelten Umgebungen zu erhalten.
Nach Abgabe der Belege konnten folgende Ergebnisse ermittelt werden.
[⇒ Ich weiß nicht was ich da schreiben soll.]

6.3 Weitere parallele Probleme

Prinzipiell gibt es in fast allen Algorithmen Abläufe, die unabhängig voneinander sind und damit nach Parallelisierung rufen. Oftmals lohnt es sich sogar eine Parallelisierung durchzuführen. Es gibt sehr viele Probleme, die ohne große Schwierigkeiten parallelisierbar sind. Neben der Berechnung von Funktionen (Mandelbrot–Menge) würden sich auch andere Probleme anbieten. Als

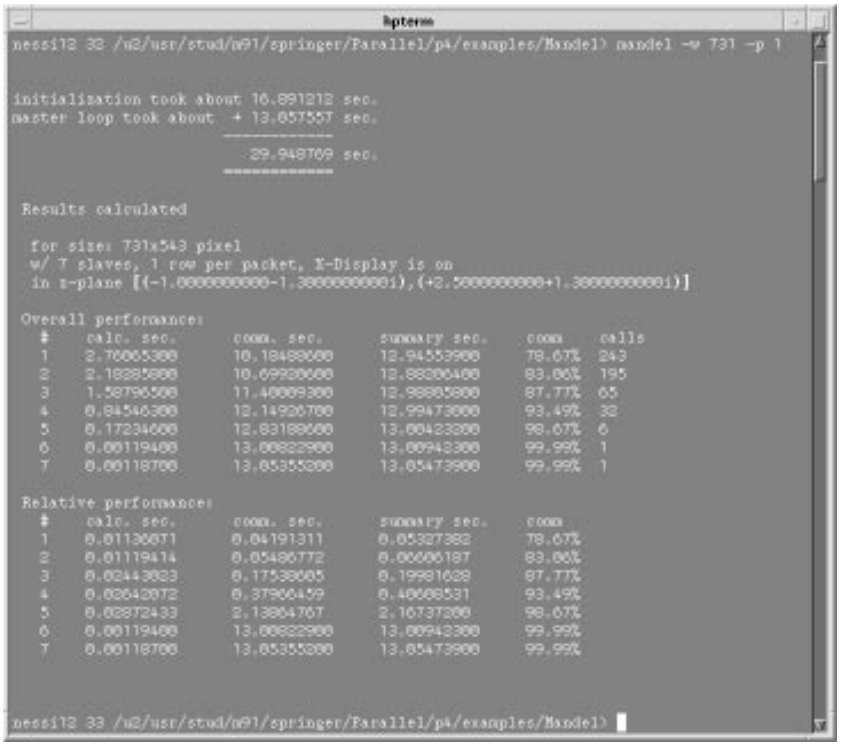


Abbildung 4: statistische Ausgabe des Mandelbrot–Programms

Beispiel sei hier *Ray–Tracing*, bestimmte Arten von Bildbearbeitung (*Edge–Detection*) aber auch die Berechnung von Animationen genannt. Hier sind die Grenzen nur durch die Phantasie und die Motivation der "Programmierer" gesetzt.

Besonders interessant erschien den Autoren die Parallelisierung von Problemen aus dem Bereich des AL^6. Weitere Ausführungen findet der interessierte Leser im Kapitel 10 auf S. 21.

7 Verfahren der Parallelisierung

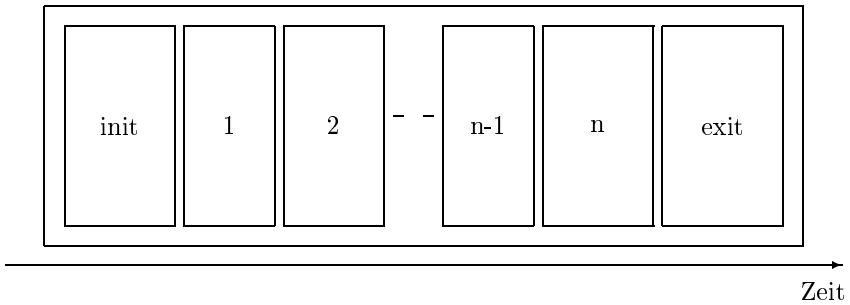


Abbildung 5: Sequentieller Programmablauf

[6] Artificial Life, siehe [16] und [17].

7.1 Vorbemerkung

Nachdem im letzten Kapitel eine eher intuitive Herangehensweise anhand praktischer Beispiele beschrieben wurde, soll nun ein allgemeingültigerer Ansatz nachgereicht werden. In diesem Zusammenhang sei auch auf [11] verwiesen. Im Anhang A befinden sich in den Teilen A.1 (S. 24) und A.2 (S. 25) Auszüge aus dieser Arbeit. Eine "formale" Betrachtung findet sich in [9] und in [20].

7.2 Synchrone Parallelität

Man geht von der Annahme aus, daß alle Prozessoren zum gleichen Zeitpunkt die gleiche Instruktion ausführen, allerdings auf unterschiedlichen Daten. Dieses Verfahren wird auch als SIMD[7] bezeichnet.

7.3 Asynchrone Parallelität

Das Verfahren der synchronen Parallelität hat, wie oben erwähnt, den Nachteil, daß man nicht unterschiedliche Instruktionen auf unterschiedlichen Daten ausführen kann. Dies führt nun zum Verfahren der asynchronen Parallelität, auch als MIMD[8] bezeichnet. Dabei lassen sich folgende Unterscheidungen treffen:

- die **statische Verteilung**, bei der jeder Prozessor genau eine Aufgabe abarbeitet, wobei alle Aufgaben gleich groß sind;
  Statische Verteilung ist eine Form der Paralellisierung, welche stark mit dem SIMD–Modell zusammenhängt. Es wird davon ausgegangen, daß jeder Prozessor in der Lage ist, über eine einfache Berechnungsvorschrift ausrechnen zu können, welchen Teil der Gesamtaufgabe er zu erledigen hat. Aufgaben dieser Art wären zum Beispiel

  ▷ Berechnung der Inversen einer Matrix,

  ▷ Summation/Multiplikation über einem Array,

  ▷ Iterationen ohne Datenabhängigkeit

  und anderes mehr.

  Programmbeispiel: siehe S. 35ff.

- das **load balancing**, wobei jedem Prozessor eine Aufgabe explizit vom Hauptprozeß zugewiesen wird, die Aufgabengröße richtet sich dabei nach der Leistung der Prozessoren;

  Programmbeispiel: siehe S. 30ff.

- das **dynamic load balancing**, bei welchen die Aufgaben dynamisch an freie Prozessoren verteilt werden, wobei mehr Aufgaben als Prozessoren vorhanden sein können.

  Programmbeispiel: siehe S. 40ff. und 45ff.

7.4 Versuch einer Bewertung

Im Prinzip ist es immer von der Aufgabenstellung und von benutzten System abhängig, welche Art der Parallelisierung benutzt wird. Deshalb ist eine pauschale Bewertung schlecht möglich. Alle Arten haben ihre Einsatzgebiete.

[7] Single Instruction Multiple Data
[8] Multiple Instruction Multiple Data

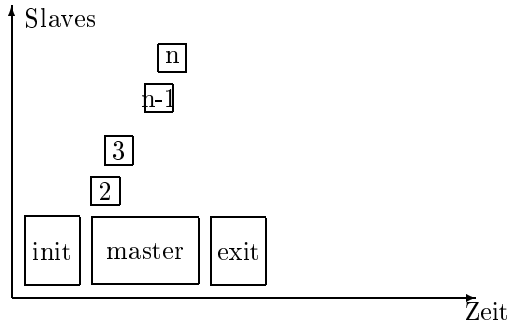


Abbildung 6: Paralleler Programmablauf, Statische Verteilung

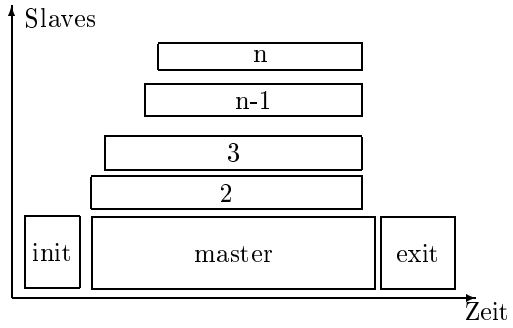


Abbildung 7: Paralleler Programmablauf, load balancing

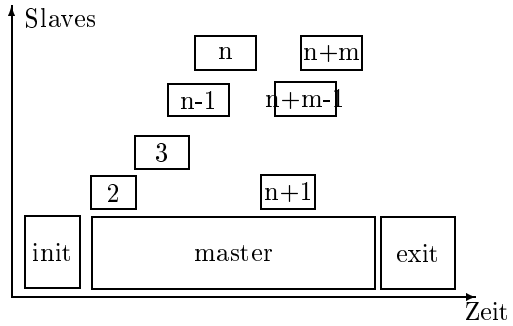
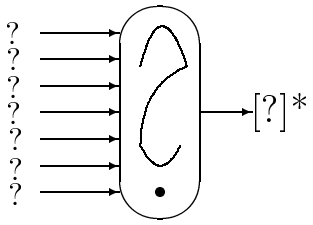


Abbildung 8: Paralleler Programmablauf, dynamic load balancing

Teil III

Seminar



8 Erfahrungen im Seminar

8.1 Vorbemerkung und Zielstellung

[hier muß eine Zielstellung eingefügt werden.]
⇒ Dr. Schalbe und Dr. Schatter

8.2 Auswahl der Aufgaben

[hier muß eine Aufgabenstellung eingefügt werden.]
⇒ Dr. Schalbe und Dr. Schatter

8.3 Durchführung der praktischen Arbeit

Eine wesentliche Aufgabe der Lehrveranstaltung Parallelverarbeitung war auch die Vermittlung praktischer Programmiererfahrungen. Dabei wurde folgendendermaßen vorgegangen.

1.Seminar (90 min)

Die Verfahrensweise bei der Beschaffung und Installation von freier Software über das Inter-Net wurde erläutert. Dabei kamen sowohl die Recherche mit `archie`, die Beschaffung mit `ftp` als auch die Installation via `make` zur Sprache. Die Arbeitsweise bei der Beschaffung und Installation von Software wird in keiner anderen Lehrveranstaltung vermittelt. Deshalb erschien es den Lehrenden und den Autoren wichtig, den Studierenden auch solche Vorgehensweisen näher zu bringen. Da die Beschaffung und Installation der Software Aufgabe der Autoren war, konnten die Erfahrungen die dabei gesammelt hatten, somit doppelt genutzt werden — einmal bei der Installation der parallelen Entwicklungstools und ein anderes Mal bei der Weitergabe der Erfahrungen an die Kommilitonen.

2.Seminar (90 min)

Die grundlegenden Gedanken von `p4` wurden erläutert. Als Arbeitsmaterial setzten die Autoren [4] ein, da dieses Material erstens original von den Entwicklern des `p4`–Systems kommt und zweitens gut für Lehrzwecke verwendet werden kann. Es wurden die Funktionen zum Message–Passing vorgestellt, die Mechanismen beim Start eines `p4`–Programme erläutert und andere Besonderheiten vorgestellt (z.B. Processgroupfiles, siehe S. 51ff.). Besonderer Wert wurde in dieser Veranstaltung auf die schrittweise Vorführung von einfachsten Programmen gelegt, um den Studierenden ein Gefühl für die Probleme, die hier auftreten können, zu vermitteln. Auch als Einstieg in die Problematik des Message–Passing hat sich eine solche "Trockenübung" bewährt. Ein Beispielprogramm aus diesem Seminar ist im Anhang auf S. 29 abgedruckt.

3.Seminar (90 min)

Einige einfache Probleme wurden diskutiert und auf Parallelisierbarkeit untersucht. Dabei wurde Wert auf die Mitarbeit der Studierenden gelegt, nicht auf die Anzahl der besprochenen Beispiele. Voraussetzung für diese Veranstaltung war die Implementation von Beispielen (Primzahlenermittlung, siehe S. 30ff.).

4.Seminar (90 min) und
5.Seminar (90 min)

Die Studierenden arbeiteten nun im Workstation-Pool. Jeder Student installierte das `p4`–System in seinem `home`–Verzeichnis. Später wurde das System global installiert. Die besprochenen Beispiele wurden implementiert und ausprobiert. Dabei wurde in Gruppen zu etwa 3 Studenten gearbeitet. Die Lehrenden und die Autoren standen den Studenten dabei zur Seite.

6.Seminar (90 min)

In dieser Veranstaltung wurde ein umfangreicheres Beispiel vorgestellt. Das parallele Mandelbrot–Programm wurde von den Autoren im Vorfeld des Seminares erstellt. Es wurde kurz die Implementation erklärt. Etwas umfangreicher war dagegen die Erläuterung der Handhabung. Dieses Programm sollte Grundlage für einen Teil des Abschlußbeleges sein. Deshalb wurden Mechanismen eingebaut, die es dem Nutzer erlauben, statistische Auswertungen zum Laufzeitverhalten vorzunehmen. Im Zuge einer solchen Auswertung sollte den Studierenden klargemacht werden, daß Parallelisierung nicht immer Vorteile bringt. Das Mandelbrot–Programm ist in diesem Papier im Kapitel 6.2 ausführlicher beschrieben.
Im zweiten Teil dieses Seminares wurde die graphische Entwicklungsumgebung HeNCE vorgestellt. Den Studierenden wurde damit eine weitere Möglichkeit der parallelen Programmierung nähergebracht werden. Diesem zweiten Teil konnte leider nicht genügend Zeit eingeräumt werden. Die Vorstellung dieses Systems benötigt mindestens ein ganzes Seminar (90 min).

9 Auswertung des Seminars

9.1 Resultate

Aus der praktischen Tätigkeit konnte folgendes Fazit gezogen werden:

1.) Es ist möglich ohne, tiefere Kenntnis der theoretischen Grundlagen praktische Parallelverabeitung zu betreiben. Hierfür eignen sich Probleme gut, die offensichtlich parallelisierbar sind.

2.) Für komplexere Probleme ist es jedoch unabdingbar, theoretische Kenntnisse der Parallelverabeitung zu erwerben.

3.) Es zeigte sich, daß Kenntnisse der Programmiersprache $\mathcal{C}$ und handwerkliche Fähigkeiten im Umgang mit UNIX notwendig für das Verständnis solcher Programmierumgebungen, wie p4 sie darstellt, sind.

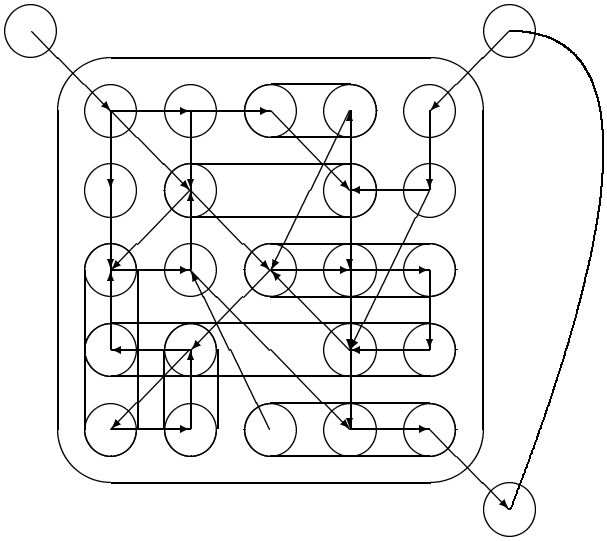
4.) Es ist möglich mit vorhandener Hardware (Workstation–Cluster) und der eingesetzten Software Parallelverarbeitung zu betreiben, deren Sinn darin liegt, existierende Rechenleistung besser auszunutzen[1].

9.2 Didaktische Erkenntnisse

Ebenfalls positiv hat sich der Fakt ausgewirkt, daß die Tutoren, der selben Seminargruppe angehörten, wie das Auditorium. Dadurch war das Seminar durch eine lockere und offene Atmosphäre geprägt. Im gleichen Moment war es für die Autoren eine gute Möglichkeit einige Techniken der Wissensvermittlung an Andere zu erlernen und zu erproben.

[1] siehe zugrunde liegende Philosophie des Systems PVM in [5]

Teil IV
Was vom Tage übrigblieb



10 Artificial Life

Am Anfang aller Gedanken, die sich die Autoren lange vor der Lehrveranstaltung über Parallelisierung gemacht hatten, stand eigentlich eine Software von Thomas S. Ray. Tierra, so der Name dieses Paketes, ist ein Programm, mit dessen Hilfe Programme erzeugt werden, die sich selbst optimieren. Man könnte diese Programme als künstliche Organismen bezeichnen. Der Gedanke an die Implementation eines ähnlichen Systems auf der Basis eines Workstation–Clusters versetzte die Autoren in freudige Erregung. Man spielte in Gedanken schon mit der Berechnung des Lebens[1] und der Lösung jeglicher Optimierungsaufgabe.

Aber wie so oft im Leben kam alles ganz, ganz anders. Zwar haben die Autoren bis heute noch nicht die Zeit gefunden diese Vision umzusetzen, trotzdem erscheint die Umsetzung von genetischen Algorithmen und evolutionärer Optimierung auf der Basis von Parallelrechnern ein erfolgversprechendes Ziel zu sein. Erste Ansätze wurden in [15] untersucht.

11 Weiterführung

Einen tieferen Einstieg in diese Problematik bietet die Lehrveranstaltung "Parallelverarbeitung II", gehalten im Wintersemester '94/95 durch Dr. Schalbe an der Fakultät Informatik und Mathematik der Hochschule für Architektur und Bauwesen Weimar -Universität-. Des weiteren wäre die Lehrveranstaltung "Algorithmen und Komplexitätstheorie" von Dr. Kalex zu nennen, in welcher diese Problematik eine große Beachtung findet.

12 Erweiterte Systemumgebungen

Die bisher vorgestellten parallelen Programmierumgebungen verfolgten in ihrer Art parallele Strukturen auszudrücken, bisher den Ansatz der Umsetzung dieser Strukturen in konventionelle Programmiersprachen. Dies führt aber leicht zu Problemen der Art, daß das, was in einem parallelen Algorithmus ausgedrückt werden soll, vorallem die Nebenläufigkeit, sich nur mit viel Mühe in einer konventionellen Programmiersprache ohne Konstrukte für Prozeß–Zeit–Abhängigkeiten ausdrücken läßt. Der nächstliegende Schritt ist, (vielleicht) logischerweise konventionelle Programmiersprachen so zu erweitern, daß in ihnen Nebenläufigkeiten und Prozeß–Zeit–Abhängigkeiten dargestellt werden können. Als Beispiel sei hier Concurrent–$\mathcal{C}$ gennant. Eine andere Möglichkeit ist eine Sprache gänzlich neu zu "designen" und deren Mächtigkeit von vornherein auf nebenläufige Strukturen auszurichten, so geschehen im Falle der Programmiersprache $\mathcal{OCCAM}$.
Was beiden Ansätzen gemein ist, ist die Bindung an schriftsymbolische Strukturen, das heißt auf einem Blatt Papier stehen immer Statements in einer speziellen Sprache und natürlich in einer sequentiellen Abfolge, denen man nur schwer ihre Nebenläufugkeit ansieht.

Wenn man jedoch aus dieser "Mikrostruktur" herausgeht und einige formale Methoden der Mathematik zu Hilfe nimmt, läßt sich das Problem auf eine sehr interessante Weise transformieren. Man nehme *DAG*'s; *directed acyclic graphs*.
Der Vorteil dieser Verfahrensweise liegt darin, daß in einer Art Top–Down–Entwurf zuerst der grobe Ablauf in einem DAG spezifiziert wird, in späteren Phasen den Knoten in diesem DAG ihr sequentieller Programmteil zugeordnet werden kann. Desweiteren kann in einem solch spezifizierten DAG, mittels formaler Methoden entschieden werden, ob eventuell Zustände oder Wege existieren, die zu Programminkonsistenzen führen können.

Ein einen solchen Ansatz verfolgendes Projekt ist HeNCE. In HeNCE wird die oben spezifizierte Softwareentwicklungsstrategie angewandt. HeNCE setzt dabei auf die Dienste von PVM auf und ist im Endeffekt nur ein X–Programm für den Anwender. HeNCE bietet dem Nutzer dadurch die Möglichkeit der Softwareentwicklung in einer "virtuellen parallelen Umgebung" unter Nutzung der realen Ressourcen dieser Umgebung. Somit ist es möglich paralleler Software auf existierenden,

[1] Obwohl es hierzu schon eine Lösung gibt, nämlich 42, s. [1].

lose gekoppelten Systemen zu entwickeln. Dies mindert wiederum die Kosten zur Anschaffung extrem teuerer Spezialhardware für Parallelverarbeitung.

Ein anderes System dieser ist XPVM. XPVM visualisiert eine virtuelle parallele Maschine und gestattet es dem Nutzer in dieser Maschine Programme auszuführen, zu debuggen und Laufzeiten zu messen.

Diese Systeme sind leider an sich so komplex, daß sie den Rahmen dieses Reportes gänzlich sprengen würden. Die Autoren geben jedoch gerne Hinweise darüber.

13 Über uns

Die Autoren studieren Informatik[2] an der Hochschule für Architektur und Bauwesen Weimar -Universität-.

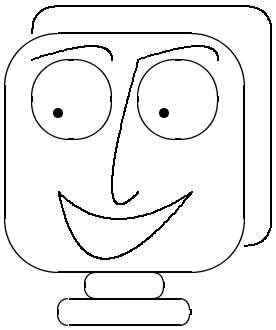
Wir möchten an dieser Stelle Dr. B. Schalbe und Dr. G. Schatter für ihre hilfreiche und konstruktive Kritik während der Entstehung dieses Papiers danken.

Dieses Dokument wurde mit *LaTeX2e* unter Linux Slakware 2.02 und *HP*–UX 9.03 erstellt.

[2] *applied computer science* trifft es nach Meinung der Autoren besser

[3] siehe auch "GNU General Public License", Version 2, 1991

Teil V
Anhang



A Exploiting the Paralellism Available in Loops

A.1 Program dependences[1]

The actual parallelism available in a program is limited by its *dependences*. A dependence between two program statements is a conflict that prevents the statements from executing concurrently[2,3,4]. Dependences can be categorized in three types:

- resource dependence,
- data dependence,
- control dependence.

A **resource dependence** between two statements is a consequence of the limited hardware available in any physical computer system. This type of dependence occurs when two different statements simultaneously attempt to use the same physical resource, such as two multiply operations competing for a single multiplier or two memory referencing operations trying to access a single memory port.

A **data dependence** exists between two statements when both reference the same memory location or access the same physical register. For example, a *flow dependence* (also called a read–after–write hazard[5]) exits from statement S_1 to S_2 in the following program fragment, since S_2 needs the value of A produced by S_1 before it can begin executing.

$$S_1 : A = B + C$$
$$S_2 : D = A - E$$

Two statements writing to the same storage location create an *output* dependence (write–after–write hazard). In the following program fragment

$$S_1 : X = Y + Z$$
$$S_2 : C = X * 22$$
$$S_3 : X = A - B$$

statement S_1 must execute before S_3, since the intervening statement S_2 uses the result produced by S_1 (that is, there is a flow dependence from S_1 to S_2). In this example, an *antidependence* (write–after–read hazard) occurs between S_2 and S_3, since S_2 reads the value of X that is overwritten by S_3. Consequently, S_2 must execute before S_3.

Flow dependences are the only true dependences in that a result produced by the first statement is used as an input value to the second statement. Both antidependences and output dependences occur when the programmer or compiler reuses storage locations to reduce the program's memory requirements. Variable renaming[6,7] can be used to eliminate these two types of depenmdences. For instance, instead of reusing a single array for two independent computations in different parts of the program, the programmer could allocate two separate arrays. The additional array would eliminate the output and antidependences for the first array. This might increase the available parallelism, but that would come at the expense of extra memory for the additional array[8].

Intuitively, a **control dependence** from statement S_i to statement S_j exists when statement S_j should be executed only if statement S_i produces a certain value. This type of dependences

[1] aus [11, S. 15]

[2] U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Press, Norwll, Mass., 1988.

[3] D. J. Kuck, *The Structure of Computers and Computations*, John Wiley and Sons, New York, 1978.

[4] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, Cambridge, Mass., 1989.

[5] P. M. Kogge, *The Architecture of Pipelined Computers*, Hemisphere, New York, 1981.

[6] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *Proc. Sigplan 88 Conf. Programming Language Design and Implementation*, ACM, New York, 1988, pp. 318–328.

[7] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Research and Development*, Vol. 11, No. 1, Jan. 1967, pp. 25–33

[8] M. Kumar, "Effect of Storage Allocation/Reclamation Methods on Paralellism and Storage Requirements," *Prc. Int'l Conf. Computer Architecture*, 1987, pp. 197–205.

occurs, for example, when S_i is a conditional statement and S_j is to be executed only if the condition evaluates to true. Control dependences can limit parallelism by causing an early exit from the loop. Ferrante et al.[9] formally define control dependences and present a technique that merges control and data dependences into a single *program dependence graph*.

Symbolic data dependence tests can determine whether or not dependences exist between program statements that have array references with complex subscripts[10]. For example, consider the following loop:

Do $I = L, H$
$M(a * I + b) = X(I) + Y(I)$
$Z(I) = 2 * M(c * I + d)$
Enddo.

The iterations from this loop can be executed in parallel using a doall scheduling strategy if there are no cross–iteration dependences due the references to array $M()$. To determine if such dependence exists, the compiler's symbolic data dependence analyzer must solve the Diophantine equation $aI_1 + b = cI_2 + d$. This equation is constrained to have $L \leq I_1 \leq I_2 \leq H$, where I_1 and I_2 are two specific values of the loop index, L and H are the lower and upper bounds on the loop index, and a, b, c, and d are integers. If no solution exists, there is no dependence between the two statements within this loop. Techniques for solving these equations have been extended to analyze other, more complex subscripts.

A.2 Program transformations[11]

Significant research has been devoted to developing compilation techniques that automatically transform programs to execute on parallel computers. Program transformations can reduce task scheduling overhead and increase the program's available parallelism. By improving memory referencing locality, they can also reduce the effects of false sharing and thereby improve the cache performance.

Loop interchanging is one type of program transformation[12,13]. It relies on the fact that the result produced by many nested loops will remein unchanged if the nesting order is swapped. For example, in the following nested loops, the inner I loop can be parallelized, but the cross–iteration dependence from the J subscript prevents parallelizing the outer loop.

Do $J = 1, N$
Do $I = 1, N$
$X(I, J) = X(I, J - 1) + Z$
Enddo
Enddo

Since the parallel iterations for the inner loop comprise only a single statement, the scheduling overhead for this parallel partitioning is high relative to the execution time of each iteration. Interchanging the execution order of the loops produces

Do $I = 1, N$
Do $J = 1, N$
$X(I, J) = X(I, J - 1) + Z$
Enddo
Enddo.

[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Trans. Programming Languages and Systems*, Vol. 9, No. 3, July 1987, pp. 319–349.

[10] D. J. Duck, Y. Muraoka, and S. Chen, "On the Number of Operations Simultaneously Executable in Fortran–Like Programs and Their Resulting Speedup, " *IEEE Trans. Computers*, Vol. C–21, No. 12, Dec. 1972, pp. 1293–1310.

[11] aus [11, S. 24]

[12] D. A. Padua and M. J. Wolfe, "Adavanced Compiler Optimizations for Supercomputers", *Comm. ACM*, Vol. 29, No. 12, Dec. 1986, pp. 1184–1201

[13] M. J. Wolfe, *Optimizing SuperCompilers for Supercomputers*, MIT Press, Cambridge, Mass., 1989.

Now when the outer I loop is parallelized, the parallel iterations consist of the entire sequential J loop. These larger granularity parallel tasks will reduce the overall execution time by reducing the total number of scheduling steps from the original nesting order.

Loop interchanging can also improve the memory locality of nested loops[14]. In the following loop nest, the inner J loop copies the one–dimensional $Y()$ vector into row I of array $X()$.

Do $I = 1, N$
  Do $J = 1, N$
    $X(I, J) = Y(J)$
  Enddo
Enddo

The outer loop repeats this copying process for each row in $X()$. Since the inner loop reads all elements of vector $Y()$ for each outer–loop iteration, there is a little locality of memory references. If the $Y()$ vector is too large to fit into the data cache, N/b references to this vector will produce a cache miss, where b is the number of elements of $Y()$ in each cache block. However, interchanging the loops reduces the number of $Y()$ elements each outer–loop iteration reads to one. This new memory referencing order will significantly improve overall performance.

Several other program transformation techniques that increase available parallelism and improve memory locality are given in Kennedy and McKinley[15] and Wolf and Lam[16].

Ideal:	$\frac{T_L}{\mathcal{P}_{crit}}$	$\mathcal{P}_{crit} = \max\left[\frac{\delta_c}{\lambda_c}\right]$
SW Pipe:	$\frac{T_L}{t_{ii}}$	$1 \leq \max\left[t_{ii}\left(resource\right), \mathcal{P}_{crit}\right] \leq t_{ii} \leq T_L$
Doacross:	$\max\left[\frac{T_L}{d}, p\right]$	$\mathcal{P}_{crit} \leq d \leq T_L$

$t_{ii}\left(resource\right) = \max\left\lceil n_r/q_r \right\rceil$

p	number of processors in a shared memory multiprocessor
T_L	sequential execution time of a single iteration of the loop
δ_c	$= \sum \delta_k =$ total execution time of the statements in a dependence cycle
λ_c	$= \sum \lambda_k =$ number of iterations crossed by a dependence cycle
t_{ii}	time interval between starting consecutive iterations for software pipelining
d	estimated time delay between starting iterations for doacross scheduling
n_r	number of time units a resource of class r is busy in one loop iteration
q_r	number of resource units of class r available to one VLIW[17] instruction

Tabelle 1: Maximum speedups for loops with cyclic dependence graphs, [11], S. 25

[14] D. Cannon, W. Jalby, and K. Gallivan, "Strategies for Cache and Local Memory Management by Global Program Transformation," *J. Parallel and Distributed Computing*, Vol. 5, No. 5, Oct. 1988, pp. 587–616.

[15] K. Kennedy and K. S. McKinley, "Optimizing for Parallelism and Data Locality," *Proc. ACM Int'l Conf. Supercomputing*, ACM, New York, 1992, pp. 323–334.

[16] M. E. Wolf, "A Data Locality Optimizing Algorithm," *Proc. ACM Sigplan Conf. Programming Language Design and Implementation*, ACM, New York, 1991, pp. 30–44.

[17] Very Long Instruction Word, s. [19]

B Quellen im InterNet

B.1 News–Gruppen

- `comp.parallel`
- `comp.parallel.mpi`
- `comp.parallel.pvm`
- `comp.sys.super`
- `comp.sys.transputer`

B.2 `ftp`–Server

- `ftp.netlib.org`

  der `ftp`–Server der NetLib Biblithek

- `info.mcs.anl.gov`

  der Heimatserver des Systems `p4`

B.3 *WWW*–Ressourcen

- `http://www.mcs.anl.gov/home/lusk/p4/p4-manual/p4.html`

  [4] als Online–Dokument lesbar

- `http://www.netlib.org/pvm3/book/pvm-book.html`

  [5] als Online Dokument lesbar

- `http://www.netlib.org/hence/hence-2.0-doc-html/hence-2.0-doc-html`

  [2] als Online Dokument lesbar

- `http://alphamor.zdv.uni-tuebingen/PARALLEL`

  Zusammenstellung aller gängigen Systeme und Dokumente über Parallelverarbeitunng, verteilte Datenverarbeitung und dazu in Beziehung stehende Forschung/Anwendung

Abbildung 9: `p4` Funktionsreferenz $\mathcal{C}$

D Ein simples p4–Programm

```
#include ``p4.h''

main (int argc, char **argv)
{

    p4_initenv(&argc,argv);
    p4_create_procgroup();
    slave();
    p4_wait_for_end();
}

slave()
{
    p4_dprintf(``Hello from %d\n'', p4_get_my_id ());
}
```

E Das Primzahlen–Programm

```
/*
 * — dynamic loadbalancing example
 *   be aware that putting just one number to one slave at a time is too
 *   fine grained 'coz the managemant overhead is even higher than the
 *   simple algorithm itself
 *   it should have better performance when let's say every slave has to
 *   check up to 10 or better 20 numbers at a time
 *   this will be done in the near future (perhaps)
 *
 * — this software runs under the gpl, version 2, june 1991
 *
 *   (c) CopyLeft by jsd
 *   (c) CopyLeft for p4 by r.butler (rbutler@sinkhole.unf.edu) and e.lusk
 *
 * — for verification
 *
 *         x │ p(x)
 *     ——+———
 *       100 │  25           p(x) is the amount of primes which are <= x
 *      1000 │ 168           (from ``primes and programming; an introduction
 *      2000 │ 303            to number theory with computing by peter giblin;
 *      3000 │ 430            cambridge university press 1993; Mat 11.2.—21;
 *      4000 │ 550            p. 40)
 *      5000 │ 669
 *      6000 │ 783
 *      7000 │ 900
 *      8000 │1007
 *      9000 │1117
 *
 */

/*  includes */

#include <unistd.h>
#include ``p4.h''

/*  defines */

#define MASTER    0
#define CALC    100
#define ISPRIME 101
#define NOPRIME 102
#define DONE    103
#define True      1
#define False     0

/*
 * the master process; distributes the work to do and collects the results
 */

void prime_master (long int lLimit, long int uLimit, int verbose)
{
```

```
int         slaves = p4_num_total_slaves ();
int         i, from, size, type, done = False;
long int   *buf = NULL;
long int    current = lLimit-1;
long int    scount[100], pcount = 0;
usc_time_t  start, end;

fprintf (stdout, ``\nsearching primes between %d and %d with %d slaves ...\n\n'',
         lLimit, uLimit, slaves);
if (!verbose)
  fprintf (stdout, ``up and away ...\n'');
fflush (stdout);
start = p4_ustimer ();
/*  initialize all available slaves */
for (i = slaves; i > 0; i--) {
  ++current;
  p4_sendx (CALC, i, (char *) &current, sizeof (long int),
            P4LNG);
  scount[i-1] = 1;
}
/*  receive from slave i and send new question to i */
while (!done) {
  from = -1;
  type = -1;
  while (!p4_messages_available (&type, &from))
    ;
  buf = NULL;
  p4_recv (&type, &from, &buf, &size);
  switch (type) {
  case ISPRIME:
    ++pcount;
    if (verbose) {
      fprintf (stdout, `` %d,'', *buf);
      if (pcount % 8 == 0)
        fprintf (stdout, ``\n'');
      fflush (stdout);
    }
  case NOPRIME:
    p4_msg_free (buf);
    ++current;
    if (current <= uLimit) {
      p4_sendx (CALC, from, (char *) &current, sizeof (long int),
                P4LNG);
      ++scount[from-1];
    } else
      done = True;
    if ((current % 1000 == 0) && !verbose) {
      fprintf (stdout, ``#'');
      fflush (stdout);
    }
    break;
  }
}
/*  along there are answers available receive them */
```

```
  from = -1;
  type = -1;
  buf = NULL;
  while (p4_messages_available (&type, &from)) {
    p4_recv (&type, &from, &buf, &size);
    switch (type) {
    case ISPRIME:
      ++pcount;
      if (verbose) {
        fprintf (stdout, `` %s,'', buf);
        if (pcount % 8 == 0)
          fprintf (stdout, ``\n'');
        fflush (stdout);
      }
      break;
    }
    p4_msg_free (buf);
    from = -1;
    type = -1;
    buf = NULL;
  }
  /*  at this point there should be no questions open, so close all slaves */
  p4_broadcast (DONE, NULL, 0);
  end = p4_ustimer ();
  fprintf (stdout, ``\n\ntime needed for computing: %.6f sec.\n'',
           ((end - start)/1000000.0));
  fprintf (stdout, ``\n\n'');
  for (i = 1; i <= slaves; i++)
    fprintf (stdout, ``slave #%3d has been called %6d time(s)\n'',
             i, scount[i-1]);
  fprintf (stdout, ``\n\n'');
  fprintf (stdout, ``there are %d primes between %d and %d\n\n'',
           pcount, lLimit, uLimit);
}

/*
 * this little thing checks if test is prime or not (trivial)
 */

int isPrime (long int test)
{
  long int i;

  if (test == 2)
    return True;
  else if (test == 3)
    return True;
  for (i = 2; i < (test / 2) + 1; i++)
    if (test % i == 0)
      return False;
  return True;
}

/*
```

```
 * this is the slave; waiting, checking, waiting, .... exiting
 */

void prime_slave ()
{
  int       type, from, size, done = False;
  long int *test;

  while (!done) {
    type = -1;
    from = MASTER;
    while (!p4_messages_available (&type, &from))
      ;
    test = NULL;
    size = -1;
    p4_recv (&type, &from, &test, &size);
    switch (type) {
    case CALC:
      if (isPrime (*test))
        type = ISPRIME;
      else
        type = NOPRIME;
      p4_sendx (type, MASTER, (char *) test, sizeof (long int),
                P4LNG);
      p4_msg_free (test);
      break;
    case DONE:
      done = True;
      p4_msg_free (test);
      break;
    }
  }
}

/*
 * how to use this little stuff
 */

void usage (char *p, long int l, long int u, int v)
{
  fprintf (stderr, ``\n\nusage:  %s [[-l arg1] [-u arg2] [-v]] | [-h]\n\n'', p);
  fprintf (stderr, ``    -l arg1: searching primes from arg1; <default is %d>\n'', l);
  fprintf (stderr, ``    -u arg2: searching primes 'til arg2; <default is %d>\n'', u);
  fprintf (stderr, ``             arg1 < arg2 && arg1 >= 2;\n'');
  fprintf (stderr, ``    -v     : print primes found; <default is %s>\n\n'',
           (v == True) ? ``ON'' : ``OFF'');
  fprintf (stderr, ``    -h     : prints this helpscreen\n'');
  fprintf (stderr, ``  (c) 1994 by mm && jsd for %s\n\n'', p);
  fprintf (stderr, ``    this program is freeware and runs under the terms of\n'');
  fprintf (stderr, ``    the GNU General Public License, Version 2, June 1991\n\n'');
  fprintf (stderr, ``  (c) 1994 by r.butler && e.lusk for p4 v. %s\n\n\n\n'',
           p4_version ());
  exit (1);
}
```

```
/*
 * the main-loop
 */

int main (int argc, char **argv)
{
  long int lLimit = 2;
  long int uLimit = 9000;
  int      verbose = False;
  char     c = '\0';

  p4_initenv (&argc, argv);
  while ((c = getopt (argc, argv, ``hl:u:v'')) != -1) {
    switch (c) {
    case 'h':
      usage (argv[0], lLimit, uLimit, verbose);
      break;
    case 'l':
      if (sscanf (optarg, ``%d'', &lLimit) != 1)
        usage (argv[0], lLimit, uLimit, verbose);
      break;
    case 'u':
      if (sscanf (optarg, ``%d'', &uLimit) != 1)
        usage (argv[0], lLimit, uLimit, verbose);
      break;
    case 'v':
      verbose = True;
      break;
    default:
      usage (argv[0], lLimit, uLimit, verbose);
      break;
    }
  }
  if ((lLimit < 2) || (uLimit <= lLimit))
    usage (argv[0], lLimit, uLimit, verbose);
  p4_create_procgroup ();
  if (p4_get_my_id () == MASTER)
    prime_master (lLimit, uLimit, verbose);
  else
    prime_slave ();
  p4_wait_for_end ();
  return (0);
}
```

F Das Mandelbrot–Programm, statische Variante

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include ``p4.h''

char name[20] = ``parallel mandel'';

#define CALC     100
#define DISP     101
#define CALCSIZE 100

#define MAXX     500
#define MAXY     500
#define DISPSIZE MAXX*MAXY

typedef struct  {
  double z_anf_r;
  double z_anf_i;
  double z_end_r;
  double z_end_i;
  int    p_anf_x;
  int    p_anf_y;
  int    p_end_x;
  int    p_end_y;
  int    maxzyklen;
  double grenze;
}  mandel_para;

typedef struct {
   Display       *d;
   Window         w;
   GC             gc;
   XEvent         event;
   XSizeHints     hint;
   int            screen;
   unsigned long  fg,bg;
   int            done;
   Pixmap         pix;
}  window_para;

void window_init (window_para *p, int argc, char **argv)
{
  p->d = XOpenDisplay (``'');
  p->screen = DefaultScreen (p->d);
  p->bg = WhitePixel (p->d,p->screen);
  p->fg = BlackPixel (p->d,p->screen);
  p->hint.x = 200;
  p->hint.y = 200;
  p->hint.width = MAXX;
  p->hint.height = MAXY;
  p->hint.flags = PPosition | PSize;
  p->w = XCreateSimpleWindow (p->d, DefaultRootWindow (p->d),
                              p->hint.x, p->hint.y, p->hint.width, p->hint.height,
```

```
                            5, p->fg, p->bg);
  XSetStandardProperties (p->d, p->w, name, name, None, argv, argc, &(p->hint));
  p->gc = XCreateGC (p->d, p->w, 0, 0);
  XSetBackground (p->d, p->gc, p->bg);
  XSetForeground (p->d, p->gc, p->fg);
  p->pix = XCreatePixmap (p->d, p->w, MAXX, MAXY, 8);
  XFillRectangle (p->d, p->pix, p->gc, 0, 0, MAXX, MAXY);
  XSelectInput (p->d, p->w, ButtonPressMask | ExposureMask);
  XMapRaised (p->d, p->w);
}

void mandel (mandel_para *par, char *erg)
{
  int  a, b;
  int  z = 0;
  double zr, zi, cr, quadzr, quadzi;
  int   zyklus;
  double schritt_r = (par->z_end_r-par->z_anf_r) /
                  (double) (par->p_end_x - par->p_anf_x);
  double schritt_i = (par->z_end_i-par->z_anf_i) /
                  (double) (par->p_end_y - par->p_anf_y);
  double ci = par->z_anf_i - schritt_i;

  for (a = par->p_anf_y; a < par->p_end_y; a++) {
    ci += schritt_i;
    cr = par->z_anf_r-schritt_r;
    for (b = par->p_anf_x; b < par->p_end_x; b++) {
      cr += schritt_r;
      zr = zi = quadzr = quadzi = 0.0;
      for (zyklus = 1;
          (zyklus < par->maxzyklen) &&
          ((quadzr + quadzi) < par->grenze);
          zyklus++) {
        quadzr = zr * zr;
        quadzi = zi * zi;
        zi = 2 * zr * zi - ci;
        zr = quadzr - quadzi - cr;
      }
      erg[z++] = zyklus;
    }
  }
}

void mandel_disp (int argc,char **argv)
{
  int          my_id, procs;
  char         msg[CALCSIZE];
  int          type, from, size;
  char        *buf;
  window_para  my_win;
  int          i, a, x, y, z, slaves, received = 0;
  int          done = 0;

  my_id = p4_get_my_id ();
```

```
  slaves = p4_num_total_ids () - 1;
  window_init (&my_win, argc, argv);

  for (i = 0; i < slaves + 1; i++)
      if (i != my_id ) {
    p4_send (CALC, i, NULL, 0);
    dprintfl (10, ``DISP: msg sent to %d !\n'', i);
  }

  while (done == 0) {
    if (0 != XCheckMaskEvent (my_win.d, ButtonPressMask | ExposureMask,
                            &(my_win.event)))
      switch  (my_win.event.type ) {
      case Expose:
        if (my_win.event.xexpose.count == 0)
          XCopyArea (my_win.d, my_win.pix, my_win.w, my_win.gc,
                   0, 0, MAXX, MAXY, 0, 0 );
        break;
      case ButtonPress:
        done = 1;
        break;
    }
    else if (received < slaves) {
      type = DISP;
      from = -1;
      buf = NULL;
      p4_recv (&type, &from, &buf, &size);
      received++;
      z = 0;
      for (y = ((MAXY * (from - 1)) / slaves); y < (MAXY * from) / slaves; y++)
        for (x  =0; x < MAXX; x++) {
        XSetForeground (my_win.d, my_win.gc, buf[z++]);
        XDrawPoint(my_win.d, my_win.pix, my_win.gc, x, y);
        p4_dprintfl (10, ``x: %d , y: %d , buf[z]: %i\n'', x, y, buf[z]);
      }
      p4_msg_free (buf);
      XCopyArea (my_win.d, my_win.pix, my_win.w, my_win.gc,
                0, 0, MAXX, MAXY, 0, 0 );
    }
  }
  XCloseDisplay (my_win.d);
}

void mandel_calc ()
{
  int my_id,procs,type,from,size;
  char        *buf;
  char        *erg;
  int          time1,time2;
  mandel_para  par;

  my_id = p4_get_my_id ();
  procs = p4_num_total_ids();
```

```
  erg = p4_msg_alloc (DISPSIZE);

  buf = NULL;
  type = CALC;
  from = 0;
  p4_recv (&type, &from, &buf, &size);
  p4_msg_free (buf);

  dprintfl (10, ``%d  CALC : starting calculation!\n'', my_id);
  time1 = p4_clock();

  par.z_anf_r = -1.5;
  par.z_end_r =  2.8;
  par.z_anf_i = -1.5 + (3.0 * (my_id-1)/ (double) procs);
  par.z_end_i = -1.5 + (3.0 * (my_id) / (double) procs);

  par.p_anf_x = 0;
  par.p_end_x = MAXX;
  par.p_anf_y = (MAXY * my_id-1) / procs;
  par.p_end_y = (MAXY * my_id) / procs;

  par.maxzyklen = 255;
  par.grenze = 4.0;

  mandel (&par, erg);

  time2 = p4_clock ();
  dprintfl (10, ``%d  CALC : calculation ready: %d ms!\n'', my_id, time2 - time1);

  p4_send (DISP, 0, erg, strlen (erg) + 1);
  dprintfl (10, ``%d  CALC :msg sent to master!\n'', my_id);
}

int main (int argc, char **argv)
{
  int my_id;

  p4_initenv (&argc, argv);
  p4_create_procgroup ();

  if (0 == p4_get_my_id ())
    mandel_disp (argc, argv);
  else
    mandel_calc ();

  p4_wait_for_end ();

  return (0);
}
```

G Das Mandelbrot–Programm, dynamische Variante

G.1 Das Includemodul `mandel.h`

```
/*  includes */

#include "p4.h"

/*  defines */

/*  we use understandable constant names (not cryptic numbers) to */
/*  1. identify the targets of the messages                       */

#define MASTER      0
#define ANYSLAVE   -1
#define ANYMSG     -1
#define ANYSIZE    -1

/*  2. specify the action that should be involved                 */

#define CALC      100
#define TIME      101
#define DISP      102
#define DONE      103

/*  typedefs */

/*  this one holds calulation parameters for the mandelbrot-set   */

typedef struct  {
  double     zReStart;
  double     zReEnd;
  double     zImStart;
  double     zImEnd;
  long int   xStart;
  long int   xEnd;
  long int   yStart;
  long int   yEnd;
  double     limit;
  long int   cycles;
}  mparam;

/*  used for timing and statistical evaluation */
typedef struct  {
  int           count;
  long int      size;
  p4_usc_time_t calc;
  p4_usc_time_t all;
}  tparam;

/*  variables */

p4_usc_time_t overalltime;
int           deltaY;
```

```
double        ReStart;
double        ReEnd;
double        ImStart;
double        ImEnd;
int           MAXX;
int           MAXY;
int           useX; /*  toggles the graphical output on or off */
```

G.2 Das Hauptmodul `main.c`

```
/*  includes */

#include "mandel.h"

/*
**   Function name : int main
**
**   Description : main program loop
**   Input : argc, argv
**   Output : 0 if all's ok
*/

int main (int argc, char **argv)
{
  p4_initenv (&argc, argv);

  parse_and_set (argc, argv);

  if (MASTER == p4_get_my_id ())
    overalltime = p4_ustimer ();

  p4_create_procgroup ();

  if (MASTER == p4_get_my_id ()) {
    overalltime = p4_ustimer () - overalltime;
    display (argc, argv);
  } else
    calc ();

  p4_wait_for_end ();

  return (0);
}
```

G.3 Das Mastermodul `master.c`

```
/*  includes */

#include <math.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include "mandel.h"
```

```
/*  const */

const char *AppName = "mandel calculation with p4";

/*  typdefs */

/*  this abstracts the X-Output-Window (is it really all needed ?)*/

typedef struct {
  Display       *display;
  Window         window;
  GC             gc;
  XEvent         event;
  XSizeHints     hint;
  int            screen;
  unsigned long  fGround;
  unsigned long  bGround;
  int            ready;
  Pixmap         pixmap;
}  wparam;

/*
**   Function name : window_init
**
**   Description : set up the window used to show the data
**   Input : argc, argv
**   Output : p
*/

void window_init (wparam *p, int argc, char **argv)
{
  p->display = XOpenDisplay ("");
  p->screen = DefaultScreen (p->display);
  p->bGround = WhitePixel (p->display, p->screen);
  p->fGround = BlackPixel (p->display, p->screen);
  p->hint.x = 200;
  p->hint.y = 200;
  p->hint.width = MAXX;
  p->hint.height = MAXY;
  p->hint.flags = PPosition | PSize;
  p->window = XCreateSimpleWindow (p->display,
                                   DefaultRootWindow (p->display),
                                   p->hint.x, p->hint.y,
                                   p->hint.width, p->hint.height, 5,
                                   p->fGround, p->bGround);
  XSetStandardProperties (p->display, p->window,
                          AppName, AppName,
                          None,
                          argv, argc,
                          &(p->hint));
  p->gc = XCreateGC (p->display, p->window, 0, 0);
  XSetBackground (p->display, p->gc, p->bGround);
  XSetForeground (p->display, p->gc, p->fGround);
  p->pixmap = XCreatePixmap (p->display, p->window, MAXX, MAXY, 8);
```

```
  XFillRectangle (p->display, p->pixmap,p->gc, 0, 0, MAXX, MAXY);
  XSelectInput (p->display, p->window, ButtonPressMask | ExposureMask );
  XMapRaised (p->display, p->window );
}

/*
**   Function name : display
**
**   Description : master routine to distribute the work, collect the
**                    results, and timestamps from the slaves
**   Input : argc, argc
**   Output : void
*/

void display (int argc, char **argv)
{
  wparam  output;
  mparam  mandel;
  int     done = FALSE;
  int     slaves = p4_num_total_slaves ();
  int     i, type, from, size, x, y, z;
  char   *buf = NULL;
  int     row = 0;
  int    *list = (int *) calloc (slaves+1, sizeof (int));
  tparam *tcount = (tparam *) calloc (slaves+1, sizeof (tparam));

/*  init the statistics */
  tcount[0].all = p4_ustimer ();
  tcount[0].count = slaves;
  list[0] = 0;

/*  first distribution of the work to be done */
  for (i = slaves; i > MASTER; i--) {
    if (row < MAXY) {
      mandel.zReStart = ReStart;
      mandel.zReEnd = ReEnd;
      mandel.zImStart = ImStart + (((ImEnd-ImStart) / MAXY) * (double) row);
      mandel.zImEnd = ImStart + (((ImEnd-ImStart) / MAXY) * (double) (row+deltaY));
      mandel.xStart = 0;
      mandel.xEnd = MAXX;
      mandel.yStart = row;
      mandel.yEnd = row+deltaY;
      mandel.cycles = 255;
      mandel.limit = 4.0;
      p4_send (CALC, i, (char *) &mandel, sizeof (mparam));
      tcount[i].all = p4_ustimer ();
      p4_dprintfl (10, "sending mparam-struct at 0x%x\n", &mandel);

/*  list[i] contains the row number which slave i starts to calc from */
      list[i] = row;
      row += deltaY;
    } else
      ++list[0];
    tcount[i].count = 0;
```

```
    tcount[i].calc = 0.0;
  }

  if (useX)
    window_init (&output, argc, argv);

  while (!done) {

/*  handle the X Window System specificae
 * note that an X—program requieres eventhandling often.
 * you should take care that the next section is passed frequently
 */
    if (useX)
   if (0 != XCheckMaskEvent (output.display,
                             ButtonPressMask | ExposureMask,
                             &(output.event)))
     switch (output.event.type) {
     case Expose:
       if (output.event.xexpose.count == 0)
         XCopyArea (output.display,
                    output.pixmap,
                    output.window,
                    output.gc,
                    0, 0,
                    MAXX, MAXY,
                    0, 0);
       break;
     case ButtonPress:
       done = True;
       break;
          }

/*  receive data && send new data */

    type = ANYMSG;
    from = ANYSLAVE;
    if (p4_messages_available (&type, &from)) {
      switch (type) {
      case DISP:
   z = 0;
   p4_recv (&type, &from, &buf, &size);
   if (useX) {
     for (y = list[from]; y < list[from]+deltaY; y++)
       for (x  = 0; x < MAXX; x++) {
         XSetForeground (output.display, output.gc, buf[z++]);
         XDrawPoint (output.display, output.pixmap, output.gc,
                     x, y);
       }
     XCopyArea (output.display, output.pixmap, output.window,
                output.gc, 0, 0, MAXX, MAXY, 0, 0);
   }
   p4_msg_free (buf);
   buf = NULL;
/*  if we have some work left, send it to the idle slave */
```

```
   if (row < MAXY) {
     mandel.zReStart = ReStart;
     mandel.zReEnd = ReEnd;
     mandel.zImStart = ImStart + (((ImEnd-ImStart) / MAXY) * (double) row);
     mandel.zImEnd = ImStart + (((ImEnd -ImStart) / MAXY) * (double) (row+deltaY));
     mandel.xStart = 0;
     mandel.xEnd = MAXX;
     mandel.yStart = row;
     mandel.yEnd = row+deltaY;
     mandel.cycles = 255;
     mandel.limit = 4.0;
     p4_send (CALC, from, (char *) &mandel, sizeof (mparam));
     p4_dprintfl (10, "sending mparam-struct at 0x%x\n", &mandel);
     list[from] = row;
     row += deltaY;
   } else  {
     tcount[from].all = p4_ustimer () - tcount[from].all;
     ++list[0];
   }
   break;
      case TIME:
   {
     p4_usc_time_t *tmp = NULL;

     p4_recv (&type, &from, &tmp, &size);
     p4_dprintfl (10, "received time %ld from %d\n", *tmp, from);
     ++tcount[from].count;
     tcount[from].calc += *tmp;
     p4_msg_free (tmp);
     if (list[0] == slaves)
       ++list[0];
   }
   break;
      }
    }

    if (list[0] == slaves+1) {
      p4_broadcast (DONE, NULL, 0);
      tcount[0].all = p4_ustimer () - tcount[0].all;
      if (!useX) {
        write_tcount (tcount);

/*  to keep track of the while-loop and exit the program while in silent mode */
        done = True;
      }
      print_tcount (tcount);
      ++list[0];
    }

/*  cleaning up everything && exiting slaves */
    if (done) {
      if (list[0] < slaves)
        p4_broadcast (DONE, NULL, 0);
```

```
      type = ANYMSG;
      from = ANYSLAVE;
      buf = NULL;
      while (p4_messages_available (&type, &from)) {
        p4_recv (&type, &from, &buf, &size);
        p4_msg_free (buf);
        type = ANYMSG;
        from = ANYSLAVE;
        buf = NULL;
      }
    }
  }

  free (list);
  free (tcount);
  if (useX)
    XCloseDisplay (output.display);
  p4_dprintfl (10, "master exiting normally ...\n");
}
```

G.4 Das Slavemodul `slave.c`

```
/*  includes */

#include "mandel.h"

/*
**   Function name : mandel (const mparam *p, char *r)
**
**   Description : calculates the mandelbrot set inside a
**                    given range w/ certain parms
**                    and put it into r
**   Input : *p
**   Output : *r
*/

void mandel (const mparam *p, char *r)
{
  register a, b, act, z = 0;
  double   zr, zi, cr, sqrZr, sqrZi;
  double   stepR = (p->zReEnd - p->zReStart) / ((double) (p->xEnd - p->xStart));
  double   stepI = (p->zImEnd - p->zImStart) / ((double) (p->yEnd - p->yStart));
  double   ci = p->zImStart - stepI;

  for (a = p->yStart; a < p->yEnd; a++) {
    ci += stepI;
    cr = p->zReStart - stepR;
    for (b = p->xStart; b < p->xEnd; b++) {
      cr += stepR;
      zr = zi = sqrZr = sqrZi = 0.0;
      for (act = 1; (act < p->cycles) && ((sqrZr + sqrZi) < p->limit); act++) {
        sqrZr = zr * zr;
        sqrZi = zi * zi;
```

```
        zi = 2 * zr * zi - ci;
        zr = sqrZr - sqrZi - cr;
      }
      r[z++] = act;
    }
  }
}

/*
**   Function name : void calc ()
**
**   Description : slave function for computing a part of the mandelbrot set
**   Input : void
**   Output : void
*/

void calc ()
{
  p4_usc_time_t  time;
  mparam        *params = NULL;
  int            done = FALSE;
  int            type = ANYMSG;
  int            from = MASTER;
  char          *buf = NULL;
  int            size = ANYSIZE;

  while (!done) {
    type = ANYMSG;
    from = MASTER;
    buf = NULL;
    size = ANYSIZE;

    while (!p4_messages_available (&type, &from))
      ;

    switch (type) {
    case CALC:

/*  receive the params from the MASTER */
      p4_recv (&type, &from, &params, &size);
      p4_dprintfl (20, "calc() => received mparam-struct at 0x%x\n", &params);
      if (NULL == params)
        return;
      p4_dprintfl (20, "calc() => xStart: %ld, xEnd: %ld, yStart: %ld, yEnd: %ld\n",
                   params->xStart,
                   params->xEnd,
                   params->yStart,
                   params->yEnd);
      p4_dprintfl (20, "calc() => zReStart: %.4f, zReEnd: %.4f, zImStart: %.4f, zImEnd: %.4f\n",
                   params->zReStart,
                   params->zReEnd,
                   params->zImStart,
                   params->zImEnd);
```

```
/*  compute the size of return buffer && alloc (!) it */
      size = ((params->xEnd - params->xStart) * (params->yEnd - params->yStart)) + 1;
      buf = p4_msg_alloc (size);

/*  go on && remember the time waisted for it */
      time = p4_ustimer ();
      mandel (params, buf);
      time = p4_ustimer () - time;
      p4_dprintfl (10, "slave function calc() took about %.6f sec. for computing\n",
                   time/1000000.0);

/*  free the params struct currently used */
      p4_msg_free (params);
      params = NULL;

/*  send the return buffer to the master */
      p4_send (DISP, MASTER, buf, size);

/*  free the return buffer, 'coz it's copied to the MASTER */
      p4_msg_free (buf);
      buf = NULL;

/*  send time to the master */
      p4_send (TIME, MASTER, (char *) &time, sizeof (p4_usc_time_t));

      break;
    case DONE:
/*  this is the end, my friend ... */
      p4_recv (&type, &from, &buf, &size);
      done = TRUE;
      break;
    default:
/*  just in case ... */
      p4_dprintfl (10, "slave function calc() received a msg from type %d from process %d.\n",
                   type, from);
      break;
    }
  }
  p4_dprintfl (10, "slave function calc() exiting normally ...\n");
}
```

G.5 Das Utilitymodul `misc.c`

```
/*  includes */

#include <unistd.h>
#include <math.h>
#include "mandel.h"

/*  defines */

#ifndef True
#define True  1
```

```
#endif
#ifndef False
#define False 0
#endif

/*  variables */

char *outFile = "RESULTS.OUT";
const int cMAXX = 2000;
const int cMINX = 100;

/*
**   Function name : void usage(char *p)
**
**   Description: prints out the usage of the program
**   Input : *p
**   Output : void
*/

void usage (char *p)
{
   fprintf (stderr, "\n\nusage:  %s [[-p size] [-r start] [-s end] [-i start] [-j end] , p);
   fprintf (stderr, "                [[-w width] |\n" [-q] [-o fname]]] | [-h]\n\n");
   fprintf (stderr, "     -p size  : the packetsize <default is %d>; size has to be\n", deltaY);
   fprintf (stderr, "                greater or equal one!\n");
   fprintf (stderr, "     -r start : z-plane real start    <default is [%lf]>\n", ReStart);
   fprintf (stderr, "     -s end   : z-plane real end      <default is [%lf]>\n", ReEnd);
   fprintf (stderr, "     -i start : z-plane imagine start <default is [%lf]>\n", ImStart);
   fprintf (stderr, "     -j end   : z-plane imagine end   <default is [%lf]>\n", ImEnd);
   fprintf (stderr, "     -w width : width of the output window in pixel <default is %d>;\n", MAXX);
   fprintf (stderr, "                the height will be computed automatically to suite the output;\n");
   fprintf (stderr, "                the width must be in range [%d, %d]!\n", cMINX, cMAXX);
   fprintf (stderr, "     -q       : means the program runs w/o any X-output\n");
   fprintf (stderr, "     -o fname : write the results in file <fname> when in silent\n");
   fprintf (stderr, "                mode; create if not exist; <default is %s>\n", outFile);
   fprintf (stderr, "     -h       : prints this help screen\n\n");
   fprintf (stderr, "     this program is freeware and runs under the terms of\n");
   fprintf (stderr, "     the GNU General Public License, Version 2, June 1991\n");
   fprintf (stderr, "   (c) 1994 by mm && jsd for %s\n", p);
   fprintf (stderr, "   (c) 1994 by r.butler && e.lusk for p4 v. %s\n\n", p4_version ());

   exit (1);
}

/*
**   Function name : void parse_and_set (int argc, char **argv)
**
**   Description : for setting some values
**   Input : argc, **argv
**   Output : void
*/

void parse_and_set (int argc, char **argv)
{
```

```
    char c;

/*  default settings*/
    deltaY = 17;
    ReStart = -1.0;
    ReEnd = 2.5;
    ImStart = -1.3;
    ImEnd = 1.3;
    MAXX = 641;
    MAXY =  (int) floor (MAXX * ((ImEnd-ImStart)/(ReEnd-ReStart)));
    useX = True;

    while ((c = getopt (argc, argv, "ho:p:r:s:i:j:w:q")) != -1)
      switch (c) {
      case 'h':
        usage (argv[0]);
        break;
      case 'o':
        if (sscanf (optarg, "%s", outFile) != 1)
          usage (argv [0]);
        break;
      case 'r':
        if (sscanf (optarg, "%lf", &ReStart) != 1)
          usage (argv [0]);
        break;
      case 's':
        if (sscanf (optarg, "%lf", &ReEnd) != 1)
          usage (argv [0]);
        break;
      case 'i':
        if (sscanf (optarg, "%lf", &ImStart) != 1)
          usage (argv [0]);
        break;
      case 'j':
        if (sscanf (optarg, "%lf", &ImEnd) != 1)
          usage (argv [0]);
        break;
      case 'w':
        if (sscanf (optarg, "%d", &MAXX) != 1)
          usage (argv [0]);
        if ((MAXX < cMINX) || (MAXX > cMAXX))
          MAXX = 641;
        break;
      case 'p':
        if (sscanf (optarg, "%d", &deltaY) != 1)
          usage (argv [0]);
        if (deltaY < 1)
          usage (argv[0]);
        break;
      case 'q':
        useX = False;
        break;
      default:
        usage (argv[0]);
```

```
      }

    if ((ReStart >= ReEnd) || (ImStart >= ImEnd))
      usage (argv [0]);
    MAXY =  (int) floor (MAXX * ((ImEnd - ImStart) / (ReEnd - ReStart)));
}

/*
**   Function name : void print_tcount
**
**   Description : for showing some results.
**   Input :
**   Output :
*/

void print_tcount (tparam *t)
{
  register i;
  fprintf (stdout, "\n\ninitialization took about %3.6f sec.\n",
           overalltime/1000000.0);
  fprintf (stdout, "master loop took about  + %3.6f sec.\n",
           t[0].all/1000000.0);
  fprintf (stdout, "                        ------------\n");
  fprintf (stdout, "                           %.6f sec.\n",
           (overalltime+t[0].all)/1000000.0);
  fprintf (stdout, "                        ============\n");
  fprintf (stdout, "\n Results calculated\n\n");
  fprintf (stdout, "  for size: %dx%d pixel\n", MAXX, MAXY);
  fprintf (stdout, ``  w/ %d slaves, %d %s per packet, X-Display is %s\n",
           t[0].count, deltaY, ((deltaY < 2) ? "row": "rows"), ((useX == False) ? "off" : "on"));
  fprintf (stdout, "  in z-plane [(%+.10f%+.10fi),(%+.10f%+.10fi)]\n",
           ReStart, ImStart, ReEnd, ImEnd);
  fprintf (stdout, "\n Overall performance:\n");
  fprintf (stdout, "   #\tcalc. sec.\tcomm. sec.\tsummary sec.\tcomm\tcalls\n");
  for (i = 1; i <= t[0].count; i++)
    if (t[i].count > 0)
      fprintf (stdout, "  %2d\t%6.8f\t%6.8f\t%6.8f\t%3.2f%%\t%i\n",
               i, t[i].calc/1000000.0, (t[i].all-t[i].calc)/1000000.0,
               t[i].all/1000000.0, (1-((float) t[i].calc/t[i].all))*100, t[i].count);
    else
      fprintf (stdout, "  %2d was unused\n", i);
  fprintf (stdout, "\n Relative performance:\n");
  fprintf (stdout, "   #\tcalc. sec.\tcomm. sec.\tsummary sec.\tcomm\n");
  for (i = 1; i <= t[0].count; i++)
    if (t[i].count > 0)
      fprintf (stdout, "  %2d\t%6.8f\t%6.8f\t%6.8f\t%3.2f%%\n",
               i, (t[i].calc/1000000.0)/t[i].count,
               ((t[i].all-t[i].calc)/1000000.0)/t[i].count,
               (t[i].all/1000000.0)/t[i].count,
               ((1-((float) t[i].calc/t[i].all))*100));
  fprintf (stdout, "\n\n\n");
}

/*
```

```
**   Function name : void write_tcount
**
**   Description :
**   Input :
**   Output :
*/

void write_tcount (tparam *t)
{
  FILE *fname = fopen (outFile, "a");
  int   i;

  fprintf (fname, "## %d %d %d %f %f %f %f\n",
           MAXX, MAXY, deltaY, ReStart, ImStart, ReEnd, ImEnd);
  fprintf (fname, "%.8f %.8f ", overalltime/1000000.0, t[0].all/1000000.0);
  for (i = 1; i <= t[0].count; i++)
    if (t[i].count > 0)
      fprintf (fname, "%d %.8f %.8f ", i, t[i].calc/1000000.0, t[i].all/1000000.0);
  fprintf (fname, "\n");
  fclose (fname);
}
```

G.6 Das Prozeßzuordnungsmodul `mandel.pg`

```
# process-group-file for mandel calc /w p4
# beside from nessi1 && nessi2 please comment your local machine out

local   0
nessi17 1 /usr/local/src/p4/mandelp4/mandel
nessi16 1 /usr/local/src/p4/mandelp4/mandel
nessi15 1 /usr/local/src/p4/mandelp4/mandel
nessi14 1 /usr/local/src/p4/mandelp4/mandel
nessi13 1 /usr/local/src/p4/mandelp4/mandel
#nessi12 1 /usr/local/src/p4/mandelp4/mandel
nessi11 1 /usr/local/src/p4/mandelp4/mandel
nessi10 1 /usr/local/src/p4/mandelp4/mandel
nessi9  1 /usr/local/src/p4/mandelp4/mandel
nessi8  1 /usr/local/src/p4/mandelp4/mandel
nessi7  1 /usr/local/src/p4/mandelp4/mandel
nessi6  1 /usr/local/src/p4/mandelp4/mandel
nessi5  1 /usr/local/src/p4/mandelp4/mandel
nessi4  1 /usr/local/src/p4/mandelp4/mandel
nessi3  1 /usr/local/src/p4/mandelp4/mandel
#nessi2  1 /usr/local/src/p4/mandelp4/mandel
#nessi1  1 /usr/local/src/p4/mandelp4/mandel
```

Literaturverzeichnis

[1] **Adams, Douglas**
Per Anhalter durch die Galaxis
Ullstein, 1984,
nicht im Bestand der HAB Bibliothek

[2] **Bequelin, Adam**, **Dongorra, Jack**, **Geist, Al**, **Manchek, Robert**, **Moore, Keith**, **Newton, Peter** and **Sunderam, Valdy**
HeNCE: A User's Guide, Version 2.0
-
`http://www.netlib.org/hence/hence-2.0-doc-html/hence-2.0-doc-html`

[3] **Berg, Dave**
Der große MAD Report – Moderne Technik
MAD Nr. 41, S. 35ff,
bei den Autoren einsehbar

[4] **Butler, Ralf** und **Lusk, Ewing**
User's Guide to the p4 Paralell Programming System
ANL–92/17, 9700 South Cass Avenue,
Argonne, IL 60439–4801,
`http://www.ncs.anl.gov/home/lusk/p4/p4-manual/p4-manual/p4.html`

[5] **Geist, Al**, **Bequelin, Adam**, **Dongorra, Jack**, **Jiang, Weicheng**, **Manchek, Robert** and **Sunderam, Valdy**
PVM: Parallel Virtual Machine; A User's Guide and Tutorial for Networked Parallel Computing
The MIT Press, Cambridge, Mass., London, England,
`http://www.netlib.org/pvm3/book/pvm-book.html`

[6] **Giblin, Peter**
Primes and Programming; An Introduction to Number Theory with Computing
Cambridge University Press, 1993,
Mat 11.2.-21

[7] **Hopcroft, John E.** und **Ullman, Jeffrey D.**
Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie
Addison–Wesley, 1990,
nicht im Bestand der HAB Bibliothek

[8] **Kale, L.V.**
A Tutorial Introduction to CHARM
University of Illinois at Urbana Champaign, 1304 W. Springfield Ave., Urbana, IL 61801,
nicht im Bestand der HAB Bibliothek

[9] **Kalex, U.**
Algorithmen und Komplexitätstheorie
–
–

[10] **Krishnamurthy, E.V.**
Parallel Processing; Principles and Practice
Addison–Wesly, 1989,
Mat 24.30.-22

[11] **Lilja, David J.**
Exploiting the Paralellism Available in Loops
in Computer, Feb. 1994, Vol. 27, No. 2, pp. 13–26,
Mat-Zeitschriften

[12] **Ludwig, Thomas**
Automatische Lastverteilung für Parallelrechner
BI–Wissenschaftsverlag, Mannheim, Leipzig, Wien, Zürich; 1993; Reihe Informatik; Bd. 94,
Mat 24.33.-3

[13] **Mandelbrot, Benoît B.**
Fractals: Forms, Chance and Dimension
San Fancisco, 1977,
nicht im Bestand der HAB Bibliothek

[14] **Mandelbrot, Benoît B.**
Die fraktale Geometrie der Natur
Akademie–Verlag Berlin, 1987,
Mat 11.419.-6

[15] **Meister, Marko** und **Renner, Wolfgang**
Mastermind; verschiedene Lösungsstrategien
–
E-mail: `meister1@nessi.informatik.hab-weimar.de`

[16] **Ray, Thomas S.**
Thoughts on the Synthesis of Artificial Life
School of Life & Health Science, University of Delaware,
Newark, DE 19716
nicht im Bestand der HAB Biblithek

[17] **Ray, Thomas S.**
Evolution and Optimization of Digital Organisms
School of Life & Health Science, University of Delaware,
Newark, DE 19716
nicht im Bestand der HAB Biblithek

[18] **Schalbe, Bernd**
Parallelverarbeitung; Script zur Vorlesung im Sommersemester '94
–
nessi1:`/u4/combau/pvss94.dvi`

[19] **Wayner, Peter**
VLIW Questions
Byte, No. 11/94,
Mat-Zeitschriften

[20] **Wegener, Ingo**
Effiziente Algorithmen für grundlegende Funktionen
B. G. Teubner Stuttgart, 1989,
LBS 97728/8

[21] **Wegner, Timothy** und **Peterson, Mark**
The Waite Group fractal creations: explore the magic of fractals on your PC
Waite Group, San Francisco, 1991,
Mat 24.737

Stichwortverzeichnis

HP–UX Compiler Flags, 5
p4, 3
    Beispiel, 7, 29
    Funktionsreferenz $\mathcal{C}$, 28
    Installation
        allgemein, 4
        local, 4
    Konzept, 5
    Processgroupfile, 6
        Beispiel, 51
p4–Beispiele, 11
    Mandelbrot–Programm, 11
        dynamische Variante, 13, 39
        Kommandozeilenparameter, 13
        statische Variante, 12, 35
    Primzahlensuche, 11, 30
PVM, 3
    HeNCE, 21
    XPVM, 22
    Erweiterte Systemumgebungen, 21
    Probleme, 4

Artificial Life, 21

control dependence, 24

data dependence, 24
Die Autoren, 22
diophantine equation, 25
Dr. Schalbe, 22
Dr. Schatter, 22

flow dependence, 24

Hardware, 2

Informationsbeschaffung, 3

loop interchanging, 25

Parallelitat
    asynchron, 15
        dynamisch, 15, 16
        statisch, 15, 16
    synchron, 15

Quellen
    $\mathcal{WWW}$, 27
    ftp, 27
    News, 27

resource dependence, 24

Seminar, 18
    Ablauf, 18
    Aufgabenstellung, 18
    Resultate, 19
    Zielstellung, 18
Softwarebeschaffung, 3
    p4, 4
Softwareentwicklung
    parallel, 9

Ziele, 2