Rechnen in Parallelen und Verteilten Umgebungen

Experimente und Messungen

**Michael Hauf, Thomas Lehmann, Martin Meyka,
Andreas Polze, Jan Richling, Thomas Röblitz,
Daniel Runge, Daniel Schulz,
Janek Schwarz und Claus Wagner**

Inhaltsverzeichnis

Einleitung, Motivation

Die Idee, vernetzte Workstations und PCs für parallele Berechnungen einzusetzen, ist verlockend. Die heutigen, weit verbreiteten Arbeitsplatzcomputer besitzen leistungsfähige Prozessoren, einen beachtlichen Hauptspeicher und eine Netzwerkschnittstelle - typischerweise wird Ethernet unterstützt.

Einziger Unterschied zwischen solch einem Arbeitsplatzcomputer und einem Knoten (*processing element*) in einem klassischen Parallelrechner ist die Geschwindigkeit und Bandbreite, mit der Kommunikation zwischen Prozessoren möglich ist. Neue, schnelle Netzwerktechnologien, wie ATM und Fast Ethernet, werden diesen Unterschied weiter verringern. Cluster von vernetzten Workstations und PCs stellen sich also als die Parallelrechner von morgen dar.

Parallele Programmierung ist ein schwieriges Unterfangen. Kommunikationsbibliotheken, die auf asynchronem Nachrichtenaustausch (*message passing*) beruhen oder Systeme mit gemeinsamen Speicher (*shared memory*) sind die Grundlage vieler, speziell für Parallelrechner geschriebenen Programme. Dabei unterscheiden sich die Versionen eines Programms von Maschine zu Maschine oft erheblich.

Versucht man nun, parallele Algorithmen in einer verteilten Umgebung zu implementieren, so steht man vor einer ähnlichen Situation. Es existieren wenige, portable Implementationen für *message passing*-Kommunikationsbibliotheken (PVM, MPI). Weiterhin gibt es einige Ansätze, um *distributed shared memory*-Systeme in Clustern zu implementieren. Schließlich existieren die Mechanismen zur Interprozeßkommunikation auf Betriebssystemebene. Diese sind oftmals für verteiltes, paralleles Programmieren benutzbar (sokkets, RPC).

In dieser Arbeit werden nun eine Reihe von Ansätzen zur Kommunikation und Synchronisation auf ihre Eignung zum parallelen Programmieren in Netzwerkumgebungen hin untersucht. Im Einzelnen sind dies:

- Objective-C und Distributed Objects
- SUN Remote Procedure Call (RPC)
- Occam auf Transputern
- System V Interprozeßkommunikation
- Teknekrons Rendezvous Software Bus
- PVM - Parallel Virtual Machine
- SONiC - Shared Objects Net-interconnected Computer

Die verschiedenen Umgebungen wurden anhand dreier Beispielprogramme untersucht: einem Primzahltestprogramm nach dem “Sieb des Eratosthenes”, einem Programm zur Berechnung der Mandelbrotmenge und einem Programm, das das Gauß’sche Eliminationsverfahren implementierte.

In Experimenten wurde das Laufzeit- und Kommunikationsverhalten der Programme in einer Reihe verschiedener Umgebungen untersucht. Neben dem gegenüber der sequentiellen Version eines Programmes erzielbaren *speedup* interessierte vor allem der Aufwand, mit dem der Programmierer (Studenten im Hauptstudium) bei der Entwicklung der parallelen Programme konfrontiert wurde.

Die genauen Resultate der Experimente werden in nachfolgenden Abschnitten dargestellt, generell lassen sich aber folgende Aussagen über die Schwierigkeit des parallelen Rechnens in verteilten Umgebungen machen:

- Unter den betrachteten Softwareumgebungen gibt es keinen klaren Sieger, wenn man Performance und *speedu*p zum Maß macht.
- Programmiersprachliche Ansätze (Objective-C und Distributed Objects, Occam) ließen sich am einfachsten beherrschen. Details der Kommunikation (Aufbau von Verbindungen, etc.) werden vor dem Programmierer gut verborgen.
- Ansätze auf der Ebene der Systemprogrammierung (SUN RPC, UNIX System V IPC) sind schwer zu beherrschen. Die Programmierung mit diesen Mechanismen erwies sich als sehr fehleranfällig.
- PVM, Teknekrons Software Bus und SONiC waren vergleichbar schwierig zu benutzen. Im Gegensatz zu den ersten beiden *message passing*-Systemen implementiert SONiC einen objektbasierten *distributed shared memory*. Der erhoffte einfache Umgang mit dem letztgenannten System scheiterte am Mangel großer Klassenbibliotheken für *shared objects* - hier ist weitere Arbeit erforderlich.
- Generell erwies sich *Input/Output* als zentraler Flaschenhals. Im Gegensatz zu dem Transputerrechner, der tatsächlich nur eine externe Festplatte aufwies, besitzen Workstations lokale Festplatten. Die Benutzung von parallelen I/O-Techniken wie *Software RAID* und *Caching* erscheint essentiell für den Erfolg des parallelen Rechnens in Clustern von Workstations und PCs.

Die Entwicklung der Programme und die Ausführung der Experimente erfolgte im Rahmen von studentischen Projektarbeiten zum Kurs “Rechnen in Parallelen und Verteilten Umgebungen”. Dieser Kurz wurde von Dr. Andreas Polze im Sommersemester 1996 am Institut für Informatik der Humboldt-Universität zu Berlin gehalten.

Objective-C und Distributed Objects

Janek Schwarz und Michael Hauf
{schwarz,hauf}@informatik.hu-berlin.de

1.0 überblick

Dieser Bericht beschäftigt sich mit der Programmierumgebung *Objective-C* und *Distributed Objects*. Wir geben eine Einührung in die Programmierung mit *Objective-C* und demonstrieren an einem einfachen Client/Server-Beispiel die Nutzung von *Distributed Objects*. In Abschnitt 4 erläutern wir Algorithmen, die wir für diese Umgebung geschrieben haben und präsentieren die dabei erzielten Resultate.

Wir haben mit Hilfe von *Objective-C* und *Distributed Objects* Algorithmen implementiert, die Primzahlen nach dem Prinzip des Siebs von Eratosthenes suchen, lineare Gleichungssystem lösen und die Mandelbrotmenge berechnen. Die verwendeten Rechner waren HP-715/33, auf denen das Betriebssystem NeXTSTEP 3.2 lief.

2.0 Objective-C

2.1 Allgemeines zu Objective-C

Objective-C wurde 1986 von Brad Cox erfunden und von ihm in dem Buch “Object Oriented Programming, An Evolutionary Approach” beschrieben. Die in diesem Buch beschriebene Variante von Objective-C ist heute allgemein als Version 2 bekannt. Aktuell sind Objective-C-Compiler von den Firmen *Stepstone*, *NeXT* und von der *Free Software Foundation* verfügbar. Leider existiert kein verbindlicher Standard für Objective-C, was ein babylonisches Sprachgewirr zur Folge hat. Während *NeXT* im Zuge der Entwicklung von *NeXTSTEP* die Objective-C Version 4 entwickelt hat, beschritt *Stepstone*, die Firma von Cox, einen anderen Weg. Der *Stepstone*-Compiler kombiniert die Syntax der Version 4 mit Klassen und Methoden der Version 2. Damit ist der *Stepstone*-Compiler eingeschränkt kompatibel zur Version 2, da einfach nur die Syntax alter Version-2-Programme geändert werden muß. Im Gegensatz dazu ist der Compiler von *NeXT* sowohl zu *Stepstone* als auch zur Version 2 inkompatibel. Der GNU-Objective-C-Compiler der *FSF* ist kompatibel zum *NeXT*-Compiler, ihm fehlen jedoch die Klassenbibliothek des *NeXT*-Compilers. Im Zusammenhang mit dem zu *OpenStep* kompatiblen *GnuStep*, das bei der *FSF* entwikkelt wird, kann jedoch auch bald mit einer Klassenbibliothek gerechnet werden, die kompatibel zu der des *NeXT*-Compilers ist.

2.2 Die Sprache

Objective-C ist eine Mischung der Sprachen C und SmallTalk. SmallTalk ist eine streng objektorientierte Sprache, in der nur Objekte und Nachrichten an Objekte existieren. Das hat zur Folge, daß sogar so elementare Datentypen wie Zahlen Objekte sind. Dieser Ansatz sichert die vollständige Plattformunabhängigkeit von SmallTalk-Programmen, hat

aber auch zur Folge, daß SmallTalk-Systeme sehr ressourcenhungrig sind und zum Teil auch die Effizienz der Programme darunter leidet. Objektive-C kombiniert nun die Effizienz von C mit der Eleganz und einfachen Benutzung von SmallTalk. Objective-C ist dabei eine Obermenge von C und beherrscht zusätzlich Objektorientierung, also Vererbung, Datenkapselung etc. Ein wesentliches Merkmal und sehr leistungsfähiger Mechanismus von Objective-C ist die dynamische Typbindung. Dadurch wird erst zur Laufzeit eines Programmes die Klasse eines Objektes, das einer Methode (Funktion) als Parameter übergeben wird oder das sie zurückgibt, bestimmt. Es ist also möglich, einer Methode beliebige Objekte zu übergeben, so daß diese abhängig von der Klasse des Objektes unterschiedliche Aktionen ausführt. Objekte haben den Typ `id`. Dieser sagt jedoch noch nichts über die Klasse des Objektes aus! Die Klasse eines Objektes wird erst zur Laufzeit vom Programmierer festgelegt.

2.3 Neue Konzepte

Jede Klasse eines Objective-C-Programms folgt einer bestimmten Struktur. Sie besteht aus dem Klasseninterface und der Klassenimplementation; es üblich, daß sich verschiedene Klassen in verschiedenen Files befinden, wobei das Klasseninterface in einem zur Klasse gehörenden Headerfile definiert wird. Um sicher zu stellen, daß eine Klasse eine gewisse Grundfunktionalität hat, kann man Protokolle definieren, die die Klasse erfüllen soll. Auf Protokolle werden wir im folgenden nicht weiter eingehen. Sie sind für eine effiziente Nutzung der Distributed Objects notwendig, man kommt aber auch ohne aus. Das Interface einer Klasse enthält die Beschreibung der Klasse. In ihm stehen die Superklasse (die Klasse, von der geerbt wird), die Instanzenvariablen und die Methodenköpfe. Die Methodenköpfe entsprechen den Funktionsprototypen in ANSI-C. Man unterscheidet Instanzen- und Klassenmethoden. Instanzenmethoden werden durch ein “`-`” gekennzeichnet und dienen der Manipulation der Daten des Objektes. Die durch ein “`+`” gekennzeichneten Klassenmethoden werden genutzt, um Eigenschaften der Klasse zu ändern. Parameter werden einer Methode übergegeben, indem ihrem Namen ein “`:`” angehängt wird und der Parameter und sein Typ angegeben wird. Typen werden mittels des C-Typecastkonstruktes angegeben. Ohne Angabe eines Typs wird implizit `id` angenommen.

Instanzvariablen sind die privaten Daten eines Objektes. Jedes Objekt hat seine eigenen Instanzvariablen und kann diese auf Grund der Datenkapselung auch nur mit seinen eigenen Methoden verändern. Um Methoden eines Objektes aufrufen zu können, wird dem Objekt mit Hilfe des Konstruktes

```
[ receiver message ];
```

eine Nachrichten gesandt, wobei `receiver` ein Bezeichner einer Klasse oder eines Objektes vom Typ `id` ist und `message` entweder ein Nachrichtenbezeichner zur Identifikation einer Methode oder einer Klasse. Erwähnt werden müssen auch die Bezeichner `self` und `super`. Wird eine Nachricht an `self` gesendet, dann bedeutet das, daß das Objekt sich selbst eine Nachricht schickt. Durch das Senden einer Nachricht an `super` wird diese Nachricht der Superklasse gesendet.

Zur Veranschaulichung der Programmierung mit Objective-C enthält der Anhang ein kleines Objective-C-Programm, in dem eine Liste implementiert wird.

3.0 Distributed Objects

3.1 Einführung in Distributed Objects

Die Distributed Objects sind eine auf dem Paradigma basierende leistungsfähige Klassenbibliothek zur Implementierung von C/S-Anwendungen. Objekte können dabei auf mehrere Prozesse und damit auch Rechner verteilt werden (verschiedene Architekturen stellen auch kein Problem dar). Das Senden von Nachrichten an verteilte Objekte wurde von NeXT so elegant gelöst, daß der Programmierer keinen Unterschied zu lokalen Objekten bemerkt. Entfernte Objekte werden angesprochen, indem man einfach ihre Methoden aufruft. Dabei können bis auf `union`, `void *` und Strukturen, die Pointer enthalten, alle Datentypen, insbesondere natürlich Objekte, als Parameter übergeben werden. Das schließt auch Strukturen ein, in denen keine Pointer außer `char *` und `id` auftreten. Das Problem des Verbindungsaufbaus zwischen Client und Server wird dem Programmierer durch die Distributed Objects abgenommen. Der Server muß nur als erster gestartet werden, womit er sich beim *Network Name Server* registriert und über diesen gefunden werden kann. Im Programm sieht das wie folgt aus:

```
id myServer=[Server new];
id myConnection=[NXConnection RegisterRoot: myServer
        withName: “ExampleServer”];
[myConnection run];
```

Wir nehmen an, daß eine Klasse *Server* existiert. Die Klasse *NXConnection* wird durch das System bereitgestellt. Die Methode *run* der Klasse *NXConnection* bewirkt, daß der Server auf Clienten wartet. Der Aufruf von *run* ist blockierend. Clienten können sich nun mit dem Server verbinden:

```
id connServer=[NXConnection connectToName:
        “ExampleServer”];
```

Nun besteht zwischen Client und Server eine unidirektionale Verbindung über die kommuniziert werden kann. Die Kommunikation zwischen Server und Client kann sowohl synchron als auch asynchron sein. Durch die Verwendung synchroner Kommunikation werden die Distributed Objects zu einer eleganten und sehr einfach zu nutztenden RPC-Umgebung. Für die von uns zu lösenden Probleme ist synchrone Kommunikation ungeeignet, da dadurch Client und Server nicht parallel laufen können. Wir haben deshalb asynchrone Kommunikation verwendet. Asynchrone Methoden unterscheiden sich von synchronen nur dadurch, daß sie keine Rückgabewerte haben und sie in einem Protokoll definiert werden müssen. Asynchrone Methoden werden durch das Schlüsselwort oneway vereinbart:

```
-(oneway)asynchMsg;
```

Es soll möglich sein, innerhalb eines Programmes sowohl synchrone als auch asynchrone Kommunikation zu verwenden. Wir hätten diese Möglichkeit gern genutzt, da es sich beim Gauß’schen Eliminationsverfahren angeboten hätte, einige Methoden synchron zu verwenden. Es ist uns jedoch nicht gelungen.

3.2 Ein Client/Server-Beispiel

Dieses kleine Beispiel soll zeigen, wie mit geringem Aufwand eine bidirektionale, asynchrone Client/Server-Verbindung aufgebaut werden kann. Da im Zusammenhang mit den Distributed Objects die Begriffe Client und Server nicht mit der Bedeutung verwendet werden, die man üblicherweise annimmt, haben wir hier die Begriffe Master und Worker genutzt. Die Implementationen sieht man in Bild 1 und Bild 2.

```
#import “List.h”
id list;
@implementation Master
 -(oneway)addWorker: (id)remoteWorker {
    [remoteWorker getList: list];
   }
 -(oneway)printList {
   [list print];
   }
@end

main() {
   id myMaster=[Master new];
   id connection=[NXConnection registerRoot: myMaster
                withName: “Master”];
   list=[List new];
   [connection run];
}
```

Bild 1: Text des Master-Programmes

In diesem Beispiel verwenden wir die Klasse List, wie sie im Anhang sect:appa implementiert ist. Die Liste ist ein verteiltes Objekt. Sowohl Master als auch Worker können auf diese Liste zugreifen, ohne daß darauf geachtet werden muß, wo sich dieses Objekt tatsächlich befindet. Der Master erzeugt in diesem Beispiel eine leere Liste, die er nach Aufforderung an den Worker sendet. Dieser fügt nun zwei Elemente in die Liste ein ein und sendet dem Server die Nachricht, daß er die Liste ausgeben soll.

Wenn man im Master statt der Methode *run* die Methode *runInNewThread* aufruft, läuft die Verbindung in einem eigenen Thread. Damit ist es möglich, daß der Master neben der Kommunikation mit einem Worker noch andere Aufgaben erledigen kann.

```
#import “List.h”
id master;
@implementation Worker
 -(oneway) getList: (id) aList {
   [aList addEntry: 5];
   [aList addEntry: 6];
   [master printList];
}
@end

main() {
   id myWorker, myConnection;
```

```
        myWorker=[Worker new];
        master=[NXConnection connectToName: “Master”
                    onHost: “*”];
        myConnection=[server connectionForProxy];
        [master addWorker: myWorker];
        [myConnection run];
    }
```

Bild 2: Text des Worker-Programmes

4.0 Die Algorithmen

Im Rahmen dieses Projektes haben wir drei Algorithmen implementiert. Zum einen den Gauß’schen Algorithmus zum Lösen linearer Gleichungssysteme, zum anderen die Berechnung von Primzahlen und die Berechnung der Mandelbrotmenge. Dieser Abschnitt erläutert die Algorithmen, die bei der Implementation aufgetretenen Probleme und zeigt die Ergebnisse, sowie Verbesserungsmöglichkeiten.

4.1 Das Gauß’sche Eliminationsverfahren

Das Gauß’sche Eliminationsverfahren ist ein Algorithmus zum Lösen linearer Gleichungssystem. Wir haben ihn mit folgendem Verfahren implementiert:

- Division der k-ten Zeile, so daß das k-te Zeilenelement gleich eins ist
- Elimination der k+1-ten bis n-ten Zeile mit der k-ten Zeile
- mit der k+1-ten Zeile wie bei 1. fortfahren

Dadurch erhält man eine obere Dreiecksmatrix, in die nur noch die Werte rückwärts eingesetzt werden müssen.

Wir haben diesen Algorithmus nach dem Master/Worker-Prinzip implementiert. Ein Prozeß, der Master, verwaltet die zu bearbeitende Matrix und verteilt die Zeilen an die Worker. Ein Worker erhält eine Zeile dieser Matrix, um sie zu bearbeiten. Der k-te Worker dividiert die k-te Zeile, woraufhin alle restlichen Worker beginnen die k+1-te bis n-te Zeile mit dieser Zeile zu eliminieren. Das Verfahren ist beendet, wenn die n-te Zeile dividiert ist. Da man im allgemeinen weniger Worker als Zeilen hat, bekommt ein Worker sobald er seine Berechnung beendet hat, vom Master eine neue Aufgabe zugeteilt (Division oder Elimination). Durch diese Herangehensweise wird eine dynamische Lastverteilung realisiert, so daß weniger belastete Prozessoren höher belastete Prozessoren (weniger leistungsfähig oder fremdbelastet) unterstützen können. Das Einsetzen der Werte zum Schluß wird vom Master durchgeführt, da es sich nicht vernünftig parallelisieren läßt.

4.1.1 Ergebnisse

Um die Leistungsfähigkeit unserer Implementation zu testen, haben wir eine Matrix der Größe 25x25 ausgewählt und die Anzahl der Prozessoren zwischen eins und fünf variiert. Auf jedem Prozessor lief ein Worker. Die Grafik in Bild 3 veranschaulicht die Ergebnisse.

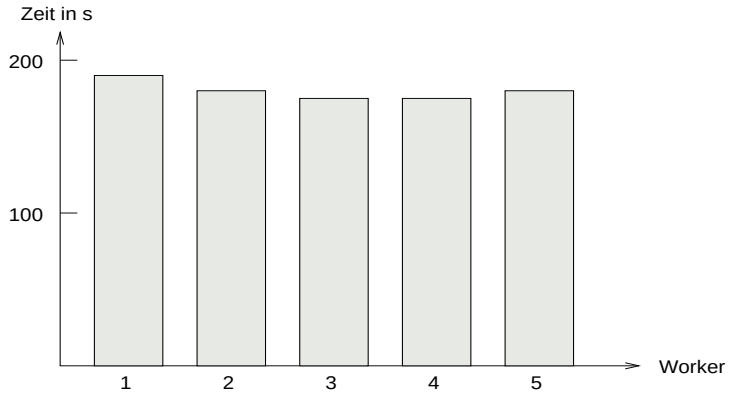

Bild 3: Performance des parallelen Gaußschen Eliminationsverfahrens

Offensichtlich wurde durch unsere Implementierung kein Performance-Gewinn erzielt. Das schlechte Ergebnis ist auf den sehr hohen Kommunikationsaufwand bei der Berechnung der Dreiecksmatrix zurückzuführen. Um die Implementierung allgemein zu halten, haben wir für die Matrix ein Objekt implementiert. Dieses Matrixobjekt ist eine Liste von Zeilenobjekten. Ein Zeilenobjekt hat zwei Instanzvariablen, einen Integer, der die Zeilennummer hält und ein Floatarray für die Elemente der Zeile. Und hier liegt das Problem. Da in C oder Objective-C Arrays das gleiche sind wie Pointer, wird bei einem Transfer eines Zeilenobjektes zu einem Worker nicht das gesamte Array übergeben, sondern nur der Pointer darauf. Das heißt, will der Worker auf die einzelnen Elemente der Zeile zugreifen, muß jedesmal der Pointer auf Masterseite dereferenziert und der Wert zum Worker kopiert werden. Das sind zwei Netzwerkzugriffe. Wird der neue Wert geschrieben, passiert das gleiche, nur diesmal vom Worker zum Master. Und das geschieht bei jedem Element der Matrix! Aufgrund unseres Ansatzes wird die Matrix überhaupt nicht verteilt, sondern bleibt vollständig beim Master, dessen Netzwerkinterface infolge dessen völlig überlastet wird.

Die Distributed Objects bieten jedoch eine Möglichkeit, das Problem zu lösen. Sie erlauben es ein gesamtes Objekt zu kopieren, wenn es dem *NXTransport*-Protokoll entspricht. Das bedeutet, daß Methoden existieren müssen, die den Inhalt eines Objektes codieren bzw. decodieren. Damit wird das Objekt transparent für den Nutzer physisch über das Netz kopiert. Leider war es uns aus zeitlichen Gründen, sowie aufgrund mangelnder Dokumentation nicht möglich, das Matrixobjekt so zu programmieren.

4.2 Berechnung von Primzahlen

Die Aufgabe bestand in der Implementation eines verteilten Programmes, das mittels des ‘Sieb des Eratosthenes’ Primzahlen sucht. Wir haben folgenden Master/Worker-Algorithmus implementiert:

- Aufteilen des zu durchsuchenden Bereich in n Teile
- Jeder Worker durchsucht den ihm zugeteilten Bereich

- Nach Beendigung der Suche in diesem Bereich wird dem Worker ein neuer Bereich zugeteilt

Jeder Worker benutzt für die Berechnung der Primzahlen zwei Felder. Im ersten legt er alle bereits gefundenen Primzahlen ab, im zweiten alle zur Berechnung benötigten. Das zweite Feld wird verwendet, um jede zu untersuchende Primzahl nur durch schon gefundene Primzahlen zu dividieren. Damit enthält das zweite Feld die Primzahlen, die kleiner als die Wurzel des zu durchsuchenden Bereichs sind. Zahlen dieses Feldes werden für Durchsuchungen weiterer Bereiche wiederverwendet. Aus Effizienzgründen haben wir nur ungerade Zahlen auf Primeigenschaft untersucht. Dieser Algorithmus entspricht zwar nicht exakt dem Sieb des Eratosthenes, erscheint uns aber effizient. Da wir wieder weniger als n Worker haben, haben wir wieder eine dynamische Lastverteilung implementiert.

4.2.1 Ergebnisse

Wir haben die Zahlen zwischen 0 und 1000000 als zu untersuchenden Zahlenbereich gewählt, den wir in Abschnitte zu je 20000 unterteilt haben. Wie schon beim Gauß'schen Algorithmus variiert die Anzahl der Worker zwischen eins und fünf. Bild 4 zeigt die Ergebnisse.

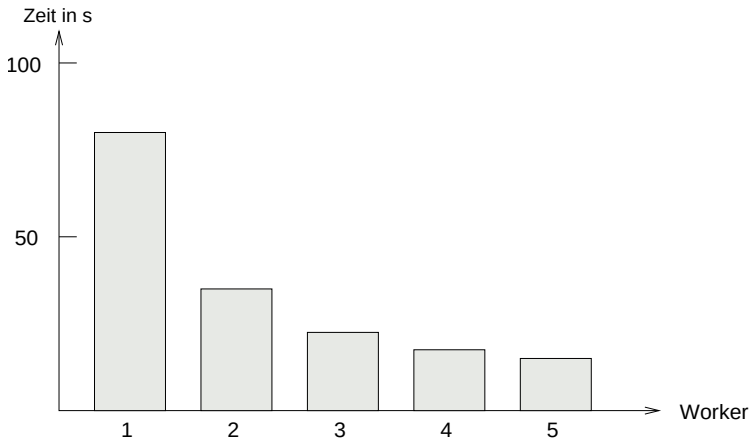

Bild 4: Performance des Sieb des Eratosthenes

Wir sehen in Bild 4 eine zufriedenstellende Geschwindigkeitssteigerung. Die Ursache dafür liegt im sehr geringen Anteil der Kommunikation gegenüber reinen Berechnungen. Für jede Berechnung müssen nur Beginn und Ende des zu durchsuchenden Bereiches an die Worker übertragen werden. Man kann sagen, daß dieses Problem für unsere Umgebung sehr gut geeignet ist. Bei der Speicherung der gefundenen Primzahlen haben wir einen etwas unrealistischen, aber einfachen Weg gewählt. Jeder Worker legt die gefundenen Primzahlen eines Bereiches in einer Datei ab, deren Name so gewählt ist, daß sich durch die Verwendung des unix-Kommandos '`cat *.prim > prim`' alle Primzahlen geordnet in der Datei `prim` befinden. Das sollte man in Zukunft ändern, da es mit relativ wenig Kommunikation möglich ist, dem Master die Primzahlen zu übergeben. Wenn man davon ausgeht, daß auf der von uns verwendeten Architektur ein Integer 32 Bit hat, reicht

es aus,anstelle von 20000 Integern nur 625 zu versenden, wenn man für jede Primzahl ein Bit setzt. Statt 80000 Bytes müssen dann nur 2500 übertragen werden.

4.3 Berechnen der Mandelbrotmenge

Die letzte Aufgabe war die verteilte Berechnung der Mandelbrotmenge. Das Problem ist dem der Primzahlsuche sehr ähnlich. Der Master teilt die Menge der zu berechnenden Bildpunkte in Zeilen auf und verschickt dann lediglich die Nummern der zu berechnenden Zeilen an die Worker. Anders als bei der Primzahlsuche werden hier alle Ergebnisse an den Master gesendet. Dieser schreibt diese Ergebnisse in eine Datei, die später von einem separaten Programm zur bildlichen Darstellung des Fraktals ausgelesen werden kann.

4.3.1 Ergebnisse

Zu berechnen war ein Fraktal der Größe von 640x480 Bildpunkten. Wir haben eine Iterationsanzahl von 100 gewählt und mit 16 Farben für die Darstellung gearbeitet. Die Berechnung erfolgte im Bereich von -2 bis 2. Die Anzahl der Prozessoren variierte von eins bis vier. Die Ergebnisse zeigt die Grafik in Bild 5.

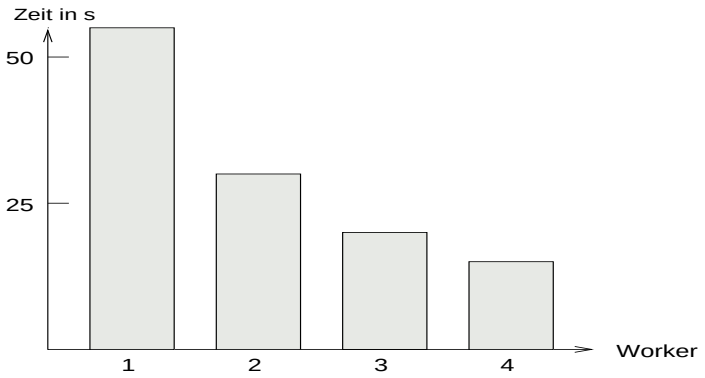

Bild 5: Performance des Mandelbrot-Programmes

Die Gründe für den Performance-Gewinn sind wiederum durch den geringen Kommunikationsaufwand zu erklären. Eventuell lassen sich der Kommunikationsaufwand noch weiter verringern und bessere Ergebnisse erreichen, wenn man die Bereiche etwas vergrößert und kleine Gruppen von Zeilen statt einzelner Zeilen versendet.

5.0 Zusammenfassung

Wie man den vorherigen Kapiteln und der Tabelle 1, die die benötigte Zeit in Abhängigkeit der Anzahl der Worker für jeden Algorithmus enthält und alle Ergebnisse noch ein-

mal zusammenfaßt,entnehmen kann, eignen sich die Distributed Objects gut für verteilte Algorithmen mit wenig Kommunikations- und hohem Rechenaufwand.

Tabelle 1:Alle Ergebnisse auf einen Blick

	1	**2**	**3**	**4**	**5**
Gauss	193	179	178	178	180
Primzahlen	79	34	23	17	14
Fraktal	61	29	21	15	-

Wenn man das Gauß'sche Eliminationsverfahren so implementiert, wie in Abschnitt 4.1 auf Seite 11 beschrieben, ist es durchaus möglich, mit den *Distributed Objects* auch bei kommunikationsintensiveren Algorithmen gute Ergebnisse zu erzielen. Für *Objective-C* und *Distributed Objects* spricht im Gegensatz zu anderen Systemen auf jeden Fall ihre einfache Benutzbarkeit. Sie sind abstrakt und gleichzeitig einfacher als vergleichbare Systeme und nehmen dem Programmierer sehr viel Arbeit ab, z.B. beim Versenden von Daten an verteilte Objekte. Ein weiterer nicht zu unterschätzender Vorteil ist die Objekt-orientierung des Systems, das es leichter macht, vorhandenen Code wieder zu nutzen.

System V Interprozeßkommunikation

Thomas Lehmann
tlehmann@informatik.hu-berlin.de

1.0 Programmierumgebung

Hier sollen die im UNIX System V vorhandenen Mechanismen zur Interprozeßkommunikation (IPC) zum parallelen Programmieren ausgenuzt werden. Die System VIPC-Mechanismen umfassen drei Kommunikationsmöglichkeiten: *Message Queues*, *Semaphore* und *Shared Memory*. Die beiden letzt genannten sind eigentlich nur in Verbindung sinnvoll. Zugang zu den IPCs erhält man über entsprechende get-Funktionen, die in den Bibliotheken <sys/msg.h>, <sys/sem.h> und <sys/shm.h> definiert sind. Zusätzlich muss noch <sys/ipc.h> mit in entsprechende Programme aufgenommen werden. Ansonsten sind die Beispielprogramme in ANSI-C in der normalen UNIX-Umgebung formuliert.

- Messagequeues: `int msgget(key_t key, int flag);`
- Semaphore:`int semget(key_t key, int nsem, int flag);`
- Shared Memory: `int shmget(key_t key, int size, int flag);`

Dabei gilt für die Parameter folgendes:

key ist ein long-Wert, der als Name der Ressource dient (ähnlich Filename).

flag sollte auf `PERMS` oder `IPC_CREAT|PERMS` gesetzt werden, wobei `PERMS` = `0666L` Lesen und Schreiben fuer alle ermöglicht. Ist *flag* nur auf `PERMS` gesetzt, kann nur auf eine bestehende Ressource zugegriffen werden. Bei der Kombination `IPC_CREAT|PERMS` wird entweder auf eine bestehende Ressource zugegriffen (vorausgestzt es existiert eine mit *key* als Bezeichner) oder eine neue angelegt.

nsem gibt die Anzahl der Semaphore an, die in der Menge mit Namen key angelegt werden sollen.

size bestimmt die Größe des Shared Memory Segments, das angefordert wird.

Alle Funktionen geben im Fehlerfall einen Wert kleiner als Null zurück. Bei Erfolg wird ein Wert größer oder gleich Null zurückgegeben, der als Bezeichner der entsprechenden Ressource gilt.

Das Entfernen der IPCs aus dem System geschieht durch folgende Funktionen:

- Messagequeues: `int msgctl(int msgid,IPC_RMID,(struct msqid_ds*)0);`
- Semaphore: `int semctl(int semid, 0,IPC_RMID,(union semun)0);`
- Shared Memory: `int shmctl(int shmid,IPC_RMID,(struct shmid_ds*)0);`

Die entsprechenden IDs sind die Rückgabewerte der *get*-Funktionen. Die *ctl*-Funktionen geben bei Mißerfolg einen Wert kleiner als Null zurück. Alle Ressourcen sollten aus dem System entfernt werden, wenn sie nicht mehr benötigt werden, da sie nur in begrenzter,

kleiner Zahl vorhanden sind. Werden sie nicht entfernt und sind schon maximal viele vorhanden, können Prozesse, die neue Recourcen anlegen wollen nicht laufen. Die im System verwendeten Ressourcen zur Interprozeßkommunikation kann mann sich mit dem Befehl `ipcs` ausgeben lassen. Mit dem Befehl `ipcrm` kann man Ressourcen entfernen, wenn man Superuser oder Eigentümer ist.

1.1 Senden und Empfangen von Nachrichten in Message Queues:

Folgende Systemaufrufe stehen zum Senden und Empfangen von Nachrichten über *Message Queues* zur Verfügung:

- `int msgsnd(int msgid, const void* ptr, size_t nbytes, int flag);`
- `int msgrcv(int msgid, void* ptr, size_t nbytes, long type, int flag);`

Wobei für die Parameter der Systemaufrufe folgendes gilt:

ptr ist ein Pointer auf eine Struktur mit folgendem Aussehen: struct mymessage { long type, Text der Messages }.

nbytes gibt die Anzahl der Bytes an, die der Nachrichtentext belegt.

type entspricht dem Typ-Feld in der Nachrichtenstruktur. Damit ist ein gezieltes Empfangen von Nachrichten eines Typs möglich, bzw es können mehrere Nachrichtentypen pro *Message Queue* gesendet und empfangen werden. *flag* ist entweder auf `IPC_NOWAIT` oder auf 0 gesetzt. 0 entspricht blockierendem Lesen bzw. Schreiben, `IPC_NOWAIT` nicht blockierendem.

Konnte eine Nachricht gesendet oder empfangen werden so wird ein Wert >= Null zurückgegeben.

1.2 Semaphoroperationen

Operationen mit Semaphoren lassen sich folgendermaßen beschreiben:

- `int semop(int semid,struct sembuf **semoparray,size_t nops);`

semoparray ist ein Feld aus Strukturen folgender Bauart:

```
struct sembuf {
   ushort      semnum;
   short       sem_op;
   short       sem_flg;
};
```

semnum ist die Nummer des Semaphores im Set auf den sich die Operation bezieht.

Verschiedene Werte für *sem_op* haben folgende Bedeutung:

- >0 => der Wert wird auf den Semaphor addiert.
- = 0 => der Prozeß wartet bis der Semaphor == 0 wird.

- < 0 => der Prozeß wartet bis der Semaphor den Wert des absoluten Betrages von semop angenommen hat. `sem_flag` entspricht flag bei *msgsnd* und *msgrcv*.

Konnte die Kette von Operationen ausgeführt werden (die Ausführung geschieht in einer Einheit im Kernel), so wird ein Wert >= 0 zurückgegeben.

1.3 Zugriff auf Shared Memory:

- `void* shmat (int shmid, 0, 0);`

Bei Erfolg wird die Adresse des Blocks zurückgegen.

2.0 Kommunikation:

Allen Beispielprogrammen liegt eine Master-Slave-Architektur zugrunde. Der Master startet x Slaves. Das Gesamtproblem wird in Teilprobleme zerlegt, die von den Slaves gelöst werden. Diese wiederum senden die Teillösungen an den Master zurück, der daraus die Gesamtlösung zusammensetzt.

Die Kommunikation sieht im einzelnen wie folgt aus:

- Master und alle x Slaves stehen über eine gemeinsame *Message Queue* in Verbindung. Zusätzlich besteht genau eine Verbindung zwischen jeweils einem Slave und dem Master. Diese Verbindung ist über ein *Shared Memory* Segment realisiert, das von einem *Semaphor* geschützt wird. Bei x Slaves werden also x *Semaphore* und x *Shared Memory*-Segmente benötigt.
- Der Aufbau der Kommunikation geschieht mit Hilfe der oben beschriebenen Funktionen.

Die Programme für Master und Slave sind in Bild 6 und Bild 7 beschrieben. Der Datenaustausch zwischen Master und Slave erfolgt via *Shared Memory* in Task-Strukturen.

```
loop
   i := i mod Number_Of_Slaves;
   if  Signal von Slave i in Messagequeue then
         P(i);
         /* lock auf Shared Segment i */
         Read (Task[i]);
         if  Task[i].Done = TRUE then
               Speichere Ergebnis;
         fi
         if  es existiert noch ein Auftrag then
               Auftrag in Task[i] packen;
               Write (Task[i]);
         fi
         V(i)
         /* unlock auf Shared Segment i */
   fi
   i++;
end
```

Bild 6: Ablauf des Master-Programmes

```
loop
   P();
   /* wartet bis Shared Segment durch Semaphor frei*/
   Read (Task);
   Auftrag ausführen;
   Ergebnis in Task packen;
   V () + Signal an Master;
end
```

Bild 7: Ablauf des Slave-Programmes

Das Signal an den Master ist wichtig, damit dieser nur dann auf das Shared Segment zugreift, wenn ein Ergebnis darin enthalten ist. Das wird mittels des Signals und des Semaphorwertes erreicht. Ein Slave kann nur auf den Shared Memory Block zugreifen, wenn der schützende Semaphor aufzwei gesetzt ist. Der Master kann in den kritischen Abschnitt eintreten, wenn der Semaphor größer oder gleich eins ist. Durch das zusätzliche Signal weiß der Master, daß auch wirklich er das entsprechende Shared Segment bearbeiten soll.

Die Funktionen für die Kommunikation sind in *"comm.h"* deklariert. In den Modulen *"slavecomm.c"* und *"mastercomm.c"* sind die entsprechenden Implementationen. Sie unterscheiden sich nur geringfügig voneinander (Master *msgrcv*, Slave *msgsnd*).

Mit *CommInit ()* werden die Ressourcen angelegt. P () und V () sichern den gegenseitigen Ausschluß. *Read ()* und *Write ()* sind zum Lesen bzw. Schreiben des Shared Blocks da. *CommDeInit ()* entfernt die Ressourcen wieder aus dem System.

3.0 Beispiele

Die Beispielprogramme befinden sich in den Verzeichnissen PRIME, GAUSS, FRACTAL. Außerdem befindet sich in jedem dieser Verzeichnisse ein Makefile.

3.1 Primzahlen:

Die Task-Struktur sieht hier so aus:

```
typedef struct task_t
{
         long      Prime[10000];
         Bool_t    Done;
} Task_t;
```

Bild 8: Task-Struktur

Bei x Slaves testet jeder Slave jeweils die x-ten ungeraden Zahlen auf Primzahligkeit. Dadurch kommt eine streifenweise Verteilung der Last zustande.

Beispiel: 5 Slaves

Tabelle 2:gefundene Primzahlen

S0	S1	S2	S3	S4
3	5	7	9	11
13	15	17	19	21
23	25	27	29	31
33	35	37	39	41
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
MAX	MAX	MAX	MAX	MAX

Ist die letzte gefundene Primzahl größer MAX, so meldet sich der Slave beim Master und liefert das Ergebnis ab. Der Master sammelt die Ergebnisse ein und speichert sie unsortiert in einer Liste.

Nachteil dieser Vorgehensweise ist, daß das Gesamtergebnis vom Master eigentlich noch sortiert werden müßte. Bei meinen Messungen habe ich das Sortieren weggelassen, um das Verhältnis Berechnung / Kommunikation besser ermitteln zu können.

- Aufruf mit: `master <anzahl_der_slaves> <maximum>`

3.2 Gauß'sches Eliminationsverfahren

```
typedef struct task_t {
   double      In1[MATRIXSIZE];
   double      In2[MATRIXSIZE];
   double      Out[MATRIXSIZE];
   long        N;
   Bool_t      Done;
} Task_t;
```

Bild 9: Datenstruktur zur Beschreibung der Sub-Taks

Bild 9 zeigt Datenstruktur zur Beschreibung der für die Lösung eines linearen Gleichungssystems nach dem Gauß'schen Eliminationsverfahren benötigten Slave-Tasks. Dabei enthält `In1` die Werte der n-ten Zeile der Koeffizientenmatrix. Für diese Zeile wurde ein Divisionsschritt ausgeführt. In `In2` werden die Werte der i-ten Zeile gespeichert. `Out` enthält nach Abschluß eines Eliminationsschritts die neuen Werte der i-ten Zeile. N wird vom Master auf n gesetzt. Der Index i läuft von n + 1 bis zur Dimension der Matrix. Bild 10 zeigt das Master-Programm in einer Pidgin-Notation, während der Text des Slave-Programms in Bild 11 dargestellt ist.

```
typedef struct line_t {
         double    elem[MATRIXSIZE];
         long       first;
/*  Nummer des ersten von 0.0 verschiedenen Elements in der Zeile  */
} Line_t;
```

```
Line_t    line;

while (n > MATRIXSIZE-1) do
   i = n+1;
   while (i >MATRIXSIZE) do
         slave := slave mod Number_Of_Slaves
         if  Signal von slave then
               P (slave);
               Read (Task[slave]);
               if Task[slave].Done == TRUE then
                     line.elem := Task[slave].Out;
                     line.first  := Task[slave].N;
                     hänge line an Ergebnisliste;
               fi
               Task[slave].In1 := Zeile n;
               Task[slave].In2 := Zeile i;
               Write (Task[slave]);
               V(slave);
         fi
         i++;
         slave++;
   od;
   Sammle restliche Ergebnisse ein und hänge sie an die Ergebnisliste;
   Ordne die Ergebnisliste nach line.first (kleinstes first zuerst);
   Schreibe alle elems ab der n+1-ten Zeile zurück in die Matrix.
   Lösche die Ergebnisliste.
   n++;
od
```

Bild 10: Gauß'sches Eliminationsverfahren: Master-Programm

```
while (1) do
   P ();
   Read (Task);
   q := Task.In2/Task.In1;
   for (i = 0; i < MATRIXSIZE; i++)
         Task.Out[i] = Task.In2[i] - q*Task.In1[i];
   i = 0;
   while (i < MATRIXSIZE && Task.Out[i] == 0.0)
         do i++;
   od
   Task.N := i;
   Task.Done :=TRUE;
   Write (Task);
   V ();
od
```

Bild 11: Gauß'sches Eliminationsverfahren: Slave-Programm

- Aufruf mit: `master <anzahl_der_slaves>`

3.3 Fraktal:

```
typedef struct task_t {
```

```
   unsigned char     Color[MAX_X];
   int               Line;
   Bool_t            Done;
} Task_t;
```

Bild 12: Beschreibung von Sub-Tasks für die Mandelbrot-Berechnung

Die Berechnung der Mandelbrotmenge erfolgt zeilenweise parallel. Auch hier wurde ien Master/Slave-Ansatz gewählt. Jeder Slave bekommt immer eine Zeile zugeteilt. Er berechnet die Farbwerte für die Bildpunkte in der Zeile und gibt das Ergebnis zurück. Der Master gibt die berechnete Zeile aus und stellt sie auf dem Bildschirm dar. Wir geben wiederum den Text für Master- und Slave-Prozesse in Bild 13 und Bild 14 an.

```
while (line > MAX_Y) do
   slave := slave mod Number_Of_Slave
   if  Signal von slave then
         P (slave);
         Read (Task[slave]);
         if (Task[slave].Done = TRUE) then
               Output (Task[slave].Line,
                           Task[slave].Color);
         fi
         Task[slave].Done := FALSE;
         Task[slave].Line   := line;
         Write (Task[slave]);
         V (slave);
   fi;
od
Restliche Ergebnisse einsammeln und ausgeben.
```

Bild 13: Text des Master-Prozesses für das Fraktal-Programm

```
while (1) do
   P ()
   Read (Task);
   for (i = 0; i < MAX_X; i++)
         Task.Color[i] = Orbital (-2.0 + i/160.0,
                           -2.0 + Task.Line/120.0);
   Task.Done := TRUE;
   Write (Task);V ();
od
```

Bild 14: Text eines Slave-Prozesses für das Fraktal-Programm

Das Fraktal wurde für ein 640x480 Bild berechnet. Die Ausgabe erfolgt leider noch in ein Textfile, da bei der Darstellung im X Window System Probleme auftraten.

- Aufruf mit: `master <anzahl_der_slaves>`

Hypothese

Man sollte vermuten, daß eine Aufteilung des Gesamtproblems auf mehrere Slaves ein umgekehrt propotionales Verhältnis zwischen Anzahl der Slaves und Rechenzeit schafft.

Dies sollte zumindest solange der Fall sein, solange die Anzahl der Slaves die Anzahl der vorhandenen Prozessoren nicht übersteigt. Es sollte also gelten:

```
t = n*(Problem/#Slaves)
```

wobei n ein Faktor ist, der den, mit Zahl der Slaves steigenden, Kommunikationsaufwand ausdrückt. Interessant ist, ob n konstant ist oder nicht.

4.0 Messungen

Die Messungen erfolgten auf einem *Linux* Rechner mit zwei 100MHz getakteten Penthium Prozessoren.

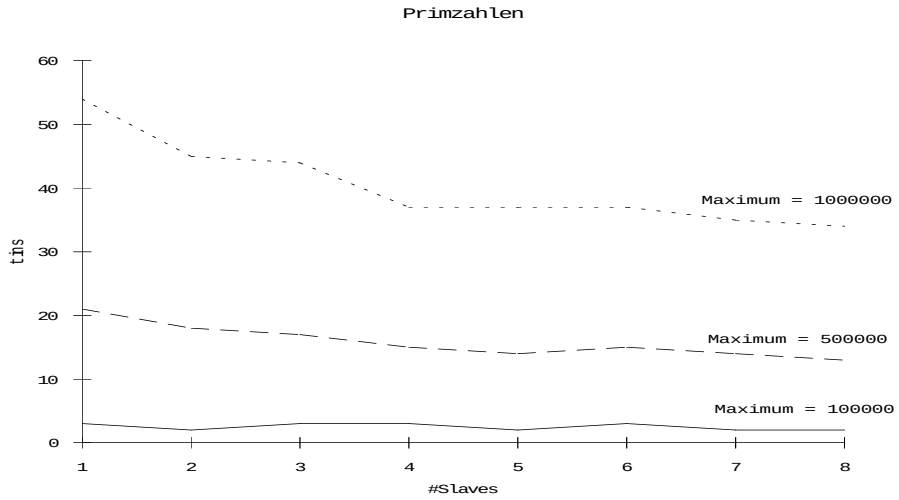

Bild 15: Verhalten des parallelen Primzahlprogramms

Tabelle 3:Programmlaufzeit in Sekunden

Slaves	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Maxim.								
100000	3	2	3	3	3	2	2	3
500000	21	18	3	15	14	15	14	13
1000000	54	45	44	37	37	37	35	34

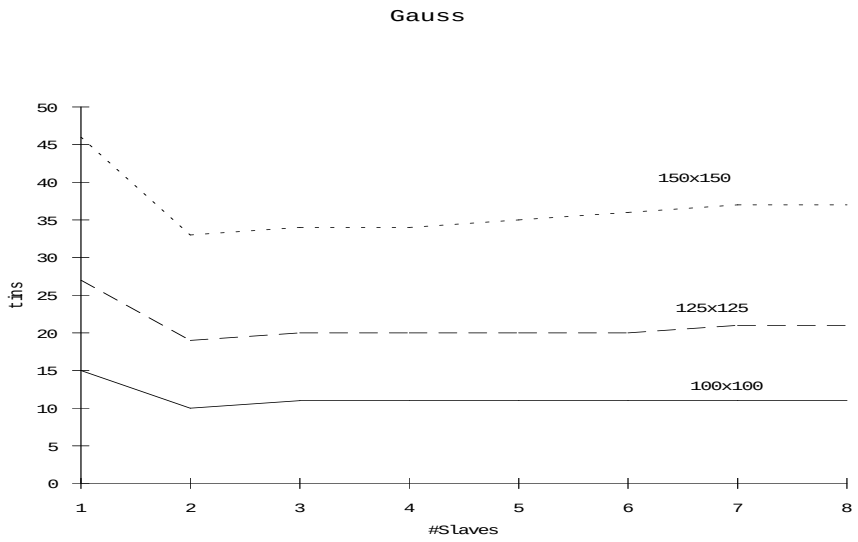

Bild 16: Verhalten des parallelen Gauß'schen Eliminationsverfahrens

Tabelle 4:Programmlaufzeit in Sekunden

Slaves	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
MATRIX-SIZE								
100x100	15	10	11	11	11	11	11	11
125x125	27	19	20	20	20	20	20	21
150x150	46	33	34	34	34	35	36	36

5.0 Auswertung:

Leider habe ich nur eine Maschine mit zwei Prozessoren zur Verfügung gehabt, von denen man meines Wissens auch keinen stillegen kann. Dadurch konnte ich keine großen Erkenntnisse gewinnen, welche Konstellation der Verteilung am günstigsten für meine Lösungsansätze ist. Den deutlichsten Geschwindigkeitsgewinn habe ich erhalten wenn ich einen Prozeß mehr als Prozessoren im Rechner sind laufen ließ. Bei Problemen, bei denen die Teilaufgaben eine konstante Größe haben, zB. der Gaußalgorithmus, frißt der zusätzliche Kommunikationsaufwand, der bei weiterer Verteilung auftritt, den Zeitgewinn langsam wieder auf. (siehe Bild 16). Wachsen die Teilaufgaben aber stärker als linear, scheint eine weitere Verteilung auch weiteren Zeitgewinn zu bringen, wenn auch nur einen geringfügigen (siehe Bild 15).

Meine Vermutung scheint sich damit bestätigt zu haben. Leider konnte ich mit der vorhandenen Rechnerarchitektur den Proportionalitätsfaktor n nicht weiter untersuchen. Dafür hätte ich mindestens drei Prozessoren benötigt.

S O N i C
The Shared Net-interconnected Computer

Martin Meyka und Claus Wagner
{meyka,cwagner}@informatik.hu-berlin.de

1.0 Einführung

Der “Shared Objects Net-interconnected Computer (SONiC)” erlaubt die Abarbeitung paralleler Programme in verteilten, Mach-basierenden Umgebungen. SONiC implementiert ein Distributed Shared Memory-System auf der Basis von Objekten. Im Gegensatz zu üblichen, seitenbasierten Ansätzen [Li/Hudak 89][Lo/Nitzberg91] stehen Objekte als gemeinsam benutzte Einheiten unter voller Kontrolle durch den Programmierer.Das Problem des *false sharing* von unabhängigen Variablen, die nur zufällig auf derselben Speicherseite liegen, kann völlig vermieden werden. Im SONiC-System wird die Aktualisierung der Replikate eines Objekts durch schwache Konsistenzprotokolle gesichert. Auf diese Weise kann die zur Konsistenzerhaltung nötige Kommunikation minimiert werden.

SONiC besteht einer Reihe von Komponenten: dem *Object Repository*, dem *Remote Execution Service*, dem *Scheduling Server* und einer C++-Klassenbibliothek, mit der parallele Programmthreads Zugriffzu *shared objects* und Synchronisationskonstrukten erhalten. Somit bietet SONiC dem Nutzer eine Programmierumgebung, mit der er Programme verteilt auf einem Cluster von Workstations ausführen kann. Die Programmiersprache ist ein um die Funktionalitäten von *shared Objects* erweitertes C++.

Die einzelnen Programmteile werden als Threads auf den entfernten Rechnern des Clusters gestartet, wodurch die Rechenleistung des gesamten Clusters genutzt werden kann. Um Threads verteilt abzuarbeiten müssen folgende Funktionalitäten bereitgestellt werden:

1. Initialisierung und Termination der Threads
2. Kommunikation der Threads
3. Synchronisation der Threads

Die 1. Bedingung wird durch den Remote Execution Service erfuellt. Den Bedingungen 2 und 3 genügen die Shared Objekte.

2.0 Shared Objekte

Klassische *Distributed Shared Memory*-Systeme bieten sich an, wenn Threads oder andere verteilte Programme auf gemeinsame Daten zugreifen sollten. Seitenbasierter *Shared Memory* hat den Nachteil, daß die Daten immer in großen Blöcken gespeichert sind. Daraus resultiert ein großer Kommunikationsaufwand bei Zugriff und Veränderung auf diese Seiten. Die Alternative, Message Passing-Systeme sind dagegen in ihrer Benutzung Fehleranfällig, sie erfordern eine große Disziplin von Programmierer.

Shared Objekte sind Instanzen von C++ Klassen, die die Kommunikation und Synchronisation in den Methoden verstecken und somit vor dem Benutzer verbergen. In SONiC wird ein Shared Objekt durch in verschiedenen Adreßräumen replizierte C++ Objekte implementiert. Jede Task eines parallelen Programmes erhält eine Kopie dieses Objektes. Objekte sind meist kleiner als sine Speicherseite - Shared Objekte verbrauchen daher weniger Speicher als ein klassisches *Distributed Shared Memory*-System. Für den Programmierer sind Shared Objekte ebenso einfach zu benutzen, wie normale C++ Objekte.

Für die Implementation der Shared Objekte stellte sich die Frage, welche Konsistenzprotokolle man benutzt. Das Problem ergibt sich aus der Tatsache, daß jede Task eine Kopie des Objektes hält und sichergestellt werden muß, daß bei Veränderung nur die aktuellen Werte benutzt werden. Bei SONiC entschied man sich sowohl *release consistency* als auch *entry consistency* zu unterstützen.

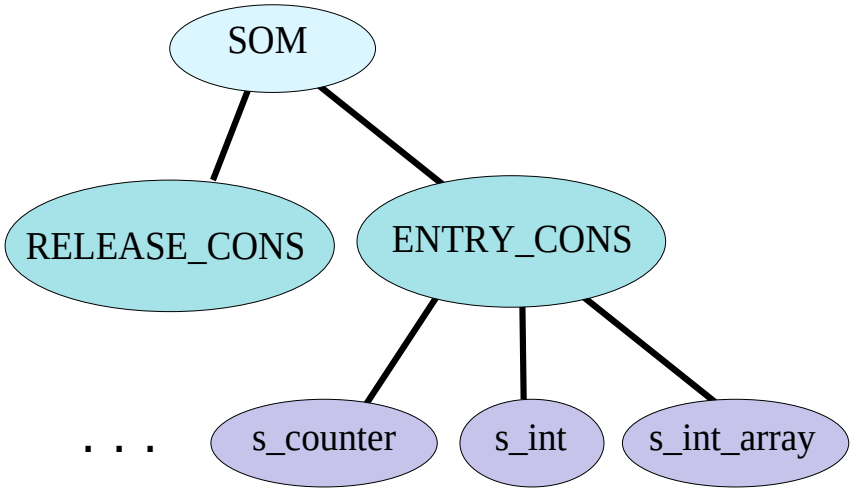

Bild 17: Klassenhierarchie für SONiC's Shared Objekte

Bild 17 zeigt die in der SONiC-Programmierbibliothek implementierte Klassenhierarchie. Als Basisklasse wurde SOM (Shared Object Memory) generiert. In dieser Klasse wird zu jedem Shared Object eine Synchronisationsvariable angelegt. Abgeleitete Klassen sind ENTRY_CONS und RELEASE_CONS. Diese Klassen implementieren Konsistzneprotokolle. Die von uns verwendeten Klassen sind aus ENTRY_CONS abgeleitet, wo also die Funktionen '*acquire_read_lock*', '*acquire_write_lock*' und '*release_lock*' implementiert sind.

Als Beispiel wie solche Klassen für den Benutzer aussehen, möchten wir die aus ENTRY_CONS hervorgegangene Klasse `s_int` (shared Integer) vorstellen :

```
# include "entry_cons.h"
class s_int : public ENTRY_CONS {
   int _val;
   virtual int size_of() { return sizeof( *this ); }
   virtual void* assign( void* p )
         { *this = *((s_int*) p); }
public:
   s_int( int val = 0 );
```

```
    virtual ~s_int();

    operator int();
    int operator+=( int i );
    int operator-=( int i );
    s_int & operator=( int s );
    s_int & operator=( s_int & s );
};
```

Bild 18: Shared Integer Klasse

In unseren Programmen meldet jeder Thread ein Objekt `s_int` unter demselben Namen an (Namen sind Integer Zahlen). Dann kann jeder Thread auf diese Variable zugreifen ohne sich um Kommunikationmechanismen oder die Synchronisation kümmern zu müssen. Der Zugriff erfolgt über die gegebenen Methoden (z.B. Shared_Var += 3). Hier der Auszug aus der zugehörigen Implementation:

```
s_int::operator+=(int i) {
    acquire_write_lock();
    _val += i;
    release_lock();
    return _val;
}
```

Bild 19: Implementation des Operators '+=' in der Klasse s_int

3.0 Systemkomponenten

3.1 Object Repository

Das *Object Repository* läuft im Hintergrund und verwaltet die Kopien unserer Objekte, das heißt hier werden die einzelenen Threads darüber informiert, wo die letzte Veränderung vorgenommen wurde und ob die gehaltene Kopie aktuell ist. Hier könnte allerdings bei sehr komunikationsintensiven Programmen ein Flaschenhals auftreten und ein weiters Ziel bei der Weiterentwicklung von SONiC ist es auch, das Object Repository, das bisher auf einem Rechner gehalten wird, zu verteilen.

3.2 Pmach.app

Pmach.app ist SONiC's graphische Oberfläche zur Konfiguration einer virtuellen parallelen Maschine auf einem Cluster von Workstations. Pmach.app ruft den rexec_server auf, der die Maschinen zu einem Cluster verbindet. Die Maschinen werden durch ihre Namen identifiziert und es wird jeder mit jedem verbunden (vollständiger Graph). Das entstandene Cluster nennt sich Virtuelle Parallele Maschine.

Beide Systemkomponenten können sowohl aus der Shell heraus gestartet werden als auch aus der graphischen Nutzeroberfläche. Letztere bietet alle vorhandenen Rechner mit Namen an und der Cluster kann per Knopfdruck erstellt werden. Danach kann das Programm normal gestartet werden.

4.0 Hardware

Die nachfolgend vorgestellten Testprogramme und Ergebnisse wurden auf HP Apollo 715/33 implementiert unter dem Betriebssystem NeXTSTEP 3.2 mit einem Mach Kern. Es waren Rechner Cluster bis zu 6 Rechnern verfügbar.

5.0 Algorithmen

Im folgenden betrachten wir verteilte Algorithmen für Primzahltest (Sieb des Eratosthenes), zur Berechnung der Mandelbrotmenge und das Gauß'sches Eliminierungsverfahren. Alle Algorithmen funktionieren nach dem Client/Server Prinzip, d.h wir starten einen Master-Prozeß zur Initialisierung, zum Aufruf der Worker und zum Einsammeln der Ergebnisse und korrekten Terminierung.

Beim Primzahltest generiert der Master ein Array (Shared Objekt) von der Größe des zu testenden Bereichs, in dem die Worker an die Position der Zahlen eine 1 schreiben wenn es sich um eine Primzahl handelt, sonst eine 0. Dem Master wird beim Aufruf die Bereichsgröße, die Anzahl der zu startenden Prozesse und die Schrittlänge auf der die Worker arbeiten, übergeben. Die Worker prüfen unter Zuhilfenahme der bereits errechneten Primzahlen, die ihnen zugeteilten Teilbereiche.

Die Mandelbrotmenge wird innerhalb einer Matrix zeilenweise errechnet, wobei die Zeilen einen Status mitführen, welcher anzeigt, ob die Zeile unbearbeitet,in Bearbeitung, bearbeitet oder schon angezeigt ist. Die Worker bekommen also die nächste zu berechnende Zeile zugeteilt, verändern anschliessend den Status und können sich die nächste Zeile holen. Der Master kann zur Laufzeit alle schon berechneten Zeilen anzeigen.

Der Gauß'sche Algorithmus verteilt die Berechnungen auch zeilenweise, wobei ebenso wie bei der Mandelbrotmenge ein Statusfeld zur Koeffizientenmatrix mitgeführt wird. Jede Zeile kann die Stati *untouched*, *working* und *computed* annehmen. Die Worker streiten sich um die Zeilen, welche sie berechnen sollen. Um eine Lastverteilung zu garantieren erhält jeder Worker zur i-ten Zeile die er berechnen soll noch die (*Dimension*-*i*)-te Zeile. Wenn mehr als ein Worker arbeiten, so kann sich kein Worker zwei aufeinanderfolgende Zeilen sichern. Der Worker der i-ten Zeile kann mit allen schon berechneten Zeilen seinen Divisions- und Eliminationsschritt durchführen.

6.0 Ergebnisse

6.1 Primzahltest

Im folgenden betrachten wir Testergebnisse des Primzahlprogramms. Getestet wurde zunächst der Bereich <10.000 mit einer Schrittweite von 100. Jeder Worker testet also 100 Zahlen auf Primeigenschaft, bevor er sich dem nächsten Job widmet. Ein Cluster von bis zu 4 Rechnern stand zur Verfügung. Zur Schrittweite ist anzumerken, daß sie eine signifikante Auswirkung auf die Performance hat, d.h. wenn sie zu klein ist erhöht sich der Kommunikationsaufwand, wenn sie zu groß ist, wird zu viel berechnet, da die Worker auf

keine vorhandenen Ergebnisse zurückgreifen können. Die Schrittweite ist somit ein Maß für die Granularität der Parallelverarbeitung.

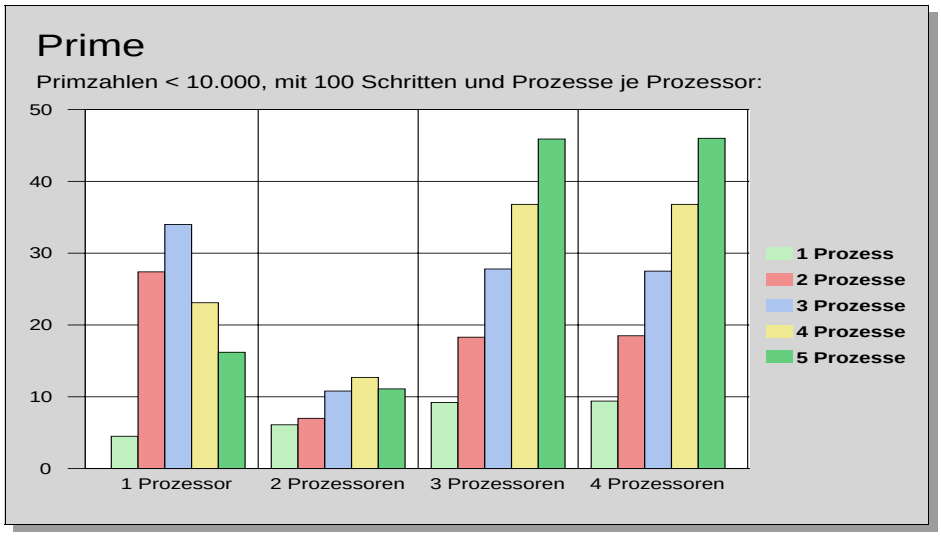

Bild 20: Resultate des Prinzahltests bei kleiner Schrittweite (feingranular)

Die Testergebnisse bieten ein sehr unregelmäßiges Bild. Bei einer sehr kleinen Schrittweite - fein-granulare Parallelität - bringt eine Parallelverarbeitung keinen Geschwindigkeitsgewinn. Vielmehr führt der Kommunikationsoverhead zu einer größeren Programmlaufzeit bei wachsender Zahl von Prozessoren (siehe Bild 20).

Wählt man dagegen eine große Schrittweite (1000 in unserem Beispiel), so sinkt die Programmlaufzeit bei wachsenderProzessorzahl. Bei hinreichend großen Problemen führt nun die Parallelverarbeitung zu einem Geschwindigkeitsgewinn (siehe Bild 21).

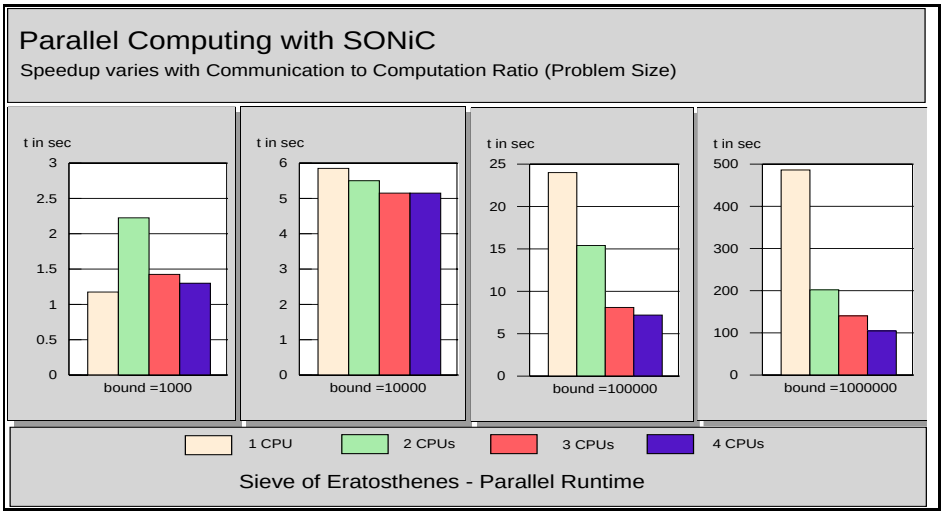

Bild 21: Resultate des Primzahltest bei großer Schrittweite (grobgranular)

Generell kann man erkennen, daß die Geschwindigkeit der Ausführung eines parallelen Programmes von Prozessanzahl, Rechneranzahl und dem Verhältnis zwischen Kommunikation zu Berechnung (Granularität) abhängig ist. Wenn das Programm auf einem Rechner mit mehreren Prozessen läuft (multitasking), so ist festzustellen, daß die Verwaltung

der Zugriffsrechte für ein Shared Objekt entscheidend davon abhängt, wieviele Prozesse darauf zugreifen wollen.

6.2 Mandelbrotmenge

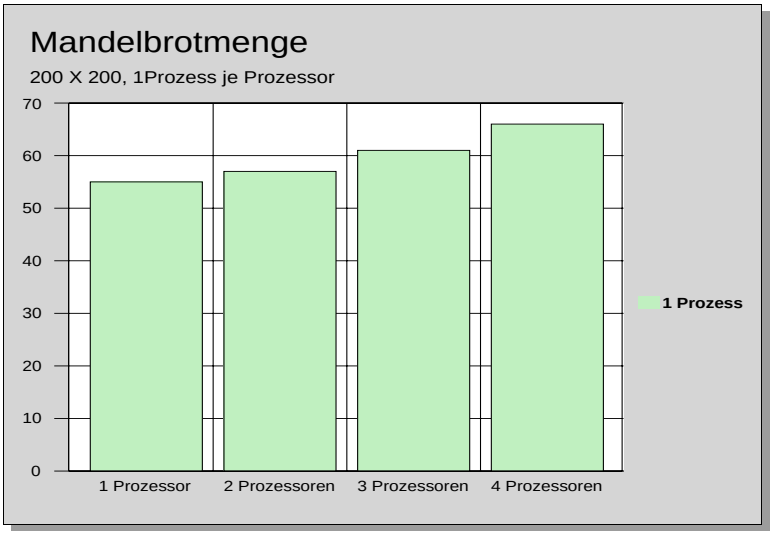

Bild 22: Berechnung eines Fraktals

Unser Mandelbrotprogramm benutzt als Datenstruktur eine Matrix, die reihenweise aus shared-Submatrizen aufgebaut ist. Es erfolgt eine dynamische Lastverteilung, dazu wird der Zustand der einzelnen Zeilen (*initial*, *computing*, *ready*, *displayed*) in einem weiteren Shared Object gespeichert. Zugriffe auf dieses Objekt erfolgen häufig und konkurrierend - sie werden zu einem Flaschenhals (siehe Bild 22).

Die sequentielle Ausführung ist bei diesem Algorithmus die Schnellste, wobei der Unterschied zu der verteilten Ausfuehrung geringfügig ist. Allerdings scheint hierbei die Kommunikation jeglichen Zeitgewinn durch verteilte Ausfuehrung zu egalisieren. Für die Ausführung mit Shared Objekten ist die derzeitige Implementation des Algorithmus schlecht geeignet, da die Prozesse ständig um Schreibrechte auf das ‘‘Zustands’’-Objekt streiten, obwohl sie untereinander sonst keine Kommunikation benötigen.

Verteiltes Rechnen mit RPC

Jan Richling
richling@informatik.hu-berlin.de

1.0 Grundsätzliches zu RPC

1.1 Ablauf der Kommunikation

RPC bietet dem Nutzer die Möglichkeit, aus einem C-Programm heraus eine entfernte Prozedur aufzurufen, wobei an Inforamtionen über diese Prozedur deren RPC-Nummer (im Code einkompiliert), die Versionsnummer (dito) und die IP-Adresse des Hosts, auf dem der Server läuft, bekannt sein. Dazu erhalten Client und Server zu den Nutzerprozeduren zusätzlich noch einen sogenannten Stub, der sich um die Kommunikation kümmert. Bis auf den Aufbau der Verbindung ist die Verwendung der Remote Procedures damit weitgehend transparent. Konkret läuft ein RPC-Ruf durch einen Clienten wie folgt ab:

1. lokaler Prozedurruf
2. Stub kodiert Aufruf und übergibt Daten an Kern
3. Datentransport via Netz an Server-Host
4. Ausliefern der Daten an Server-Stub, dekodieren
5. lokaler Ruf der Server-Prozedur
6. Resultat an Server-Stub übergeben
7. Resultat verpacken und an Kern
8. Daten via Netz an Client-Host
9. Auslieferung an Client Stub
10. Dekodieren und Ausliefern des Resultats

1.2 XDR als Datenformat

Der *External Data Representation Standard* (Juni 1987) ist eine Sammlung von architekturunabhängigen Datentypen und Codierungsreglungen. Die Typen sind c-ähnlich, wobei auch Felder variabler Länge zugelassen sind. Die Datenübertragung erfolgt maschinenunabhängig mit big endian order (höchstwertiges Bit/Byte zuerst)

1.3 Der Compiler rpcgen

Das Interface-Modul wird als XDR-Spezifikation der Datentypen und Funktionsvereinbarungen aufgeschrieben. Aus diesen Informationen werden mit Hilfe des Compilers *rpcgen* das Headerfile, Clientstub, Serverstub und ein File für die Konvertierung eigener XDR-Datentypen erzeugt. All diese C-Files müssen zusammen mit den eigenen Files für Server und Client kompiliert werden.

2.0 Verteiltes Rechnen auf Basis von RPC

Grundsätzlich gibt es zwei Ansätze, um auf Basis von RPC verteilte Programme zu schreiben. In beiden Fällen liegt das Problem darin, Asynchronizität zu schaffen. Beide Wege unterscheiden sich in der Aufgabenverteilung für Client und Server.

2.1 Server als Worker

Ein Client verteilt die Aufgaben an mehrere Server, die auf verschiedenen Hosts laufen. Synchrones RPC kann dafür nicht benutzt werden, da RPC-Rufe dann erst zurückkehren, wenn die Abarbeitung der gerufenen Funktion abgeschlossen ist, was insgesamt gegenüber der sequentiellen Bearbeitung auf einem Rechner keine Vorteile bringt. Also müßte asynchrones RPC benutzt werden, womit das Problem der Ergebnisrückgabe entsteht, da ein sofort zurückkehrender RPC-Ruf (timeout 0) kein Ergebnis übermittelt. Dafür bieten sich mehrere Lösungsideen an:

- Client forkt für jeden RPC-Ruf, dann kann der Hauptprozess weiterarbeiten und beispielsweise weitere Server auf die gleiche Weise rufen, während der Child-Prozess auf die Rückkehr des RPC Rufs wartet. Damit stellt sich wiederum die Frage, wie das Resultat vom Child zum Hauptprozess zurückkommt. IPC wäre eine Lösung, die Benutzung von Threads anstelle von Child-Prozessen eine weitere.
- Der Server gibt den Ruf sofort zurück, und zwar mit dem Ergebnis der letzten Berechnung, die er erhalten hat. Damit kann asynchron gearbeitet werden, nur ist unklar, woher der Client weiß, wann er diesen Server das nächste Mal rufen kann.
- Server und Client haben sowohl Server als auch Client-Funktionalität, so daß sie gegenseitig ihre Funktionen aufrufen können. Das Ergebnis sind viele RPC-Rufe und umständliche Programme, so daß diese Lösung unpraktisch erscheint.

2.2 Client als Worker

Das ist die Umkehrung des Client-Server-Konzeptes, doch auf diese Weise kann das Problem der Asynchronizität sehr einfach gelöst werden. Ich habe mich für diese Lösung entschieden, wobei ich einen Teil der oben als Zweites beschriebenen Idee benutzt habe. Im folgenden nun die grobe Beschreibung:

Der Server bietet als Dienst Aufgaben an und erwartet deren Lösungen zurück. Mit anderen Worten: Ein Client, der rechnen "möchte", muß ein früheres Resultat abliefern, und bekommt dann eine neue Aufgabe. Auf diese Weise kann völlig asynchron gearbeitet werden, da jeder Client genau dann die nächste Aufgabe abholt, wenn er die vorige erledigt hat. Die Aufgabe des Servers beschränkt sich auf das Einsammeln der Ergebnisse, was kaum rechenaufwendig ist, so daß an dieser Stelle kein Engpaß entstehen dürfte (später dazu mehr bei den Beispielen). Durch diese Architektur ist die Anzahl der Clients beliebig und muß dem Server nicht einmal bekannt sein, genauso können Clients noch nachträglich in die Rechnung einsteigen, da der Server jeden Ruf in der Reihenfolge des Ankommens beantwortet und grundsätzlich die Clients nicht kennt. Das wirft zwei weitere Probleme auf:

- Der Start muß synchron erfolgen, um Zeitmessungen zu ermöglichen
- Der Server muß erfahren, welche Antwort zu welcher Frage gehört

Beide Probleme sind einfach lösbar. Für den Start wird eine Funktion eingeführt, die von einem der Clients explizit gerufen wird, was den Start der Rechnung zur Folge hat. Das bedeutet gleichzeitig, daß alle anderen Clients in einer Warteschleife verharren müssen, indem sie in regelmäßigen Abständen Aufgaben anfordern und als Antwort die "leere Aufgabe" erhalten. Das zweite Problem ist ebenso leicht lösbar, indem nämlich die Struktur der Antwort eine eindeutig identifizierbare Kurzform der Aufgabe ist (z.B. die Nummer einer Matrixzeile).

3.0 Implementation einer verteilten Berechnung mit RPC

3.1 Das Make-File (am Beispiel der Primzahlenberechnung)

```
CC = gcc
LIB = -lc -lm
all : server client
client : parac.o para_clnt.o para_xdr.o
                     $(CC) -o parac  parac.o
para_clnt.o para_xdr.o $(LIB)

server : paras.o para_svc.o para_xdr.o
    $(CC) -o paras paras.o para_svc.o para_xdr.o
$(LIB)
para.h : para.x
   rpcgen para.x
.c.o   :
   $(CC) -c $<
```

3.2 Der Server in Grobdarstellung (mit Auszügen aus der Primzahlberechnung):

```
#include    <rpc/rpc.h>
#include    "para.h"
                            /* generated by rpcgen */
int nc = 0;

primcomm * primcalc_1_svc(prims *range){
static primcomm rueck;
if (nc==0) {
      /* noch nicht gestartet */
   rueck.low = 1;
   rueck.high = 1;
   return(&rueck); /* Dummy-Rueckgabe */
}
rueck.low = nc; /* Bereich fuer Rechnung durch Client
*/
rueck.high = nc+NUMBEROFPRIME*32;
if (rueck.high>4000001) rueck.high=4000001;
```

```
   /* Ende */
nc+=NUMBEROFPRIME*32;
/* printf("%d\n",nc); */
if (nc-NUMBEROFPRIME*32>4000001) {
   rueck.low = 0; /* Ende fuer Client */
   rueck.high = 0;
   printf("Fertig!\n");
}
return(&rueck);
}

int * primstart_1_svc(int *flag)
{
/* Start der Rechnung durch Setzen von nc */
static int i=0;
printf("%d\n",*flag);
i=1;
nc = 1;
return(&i);
}
```

3.3 Der Client in Grobdarstellung (mit Auszügen aus der Primzahlberechnung):

```
#include   <stdio.h>
#include   <string.h>
#include   <rpc/rpc.h>  /* standard RPC include file */
#include   "para.h"
               /* this file is generated by rpcgen */
#ifndef PROTOCOL
#define PROTOCOL        "udp"
#endif /* PROTOCOL */
...

main (int argc, char *argv[] )
{
  CLIENT   *cl;              /* RPC handle */
  char     *server;
  prims    hin;
  primcomm *rueck;
  int      *i;
...
  server=argv[1];

  if (( cl=clnt_create(server, PARA, PARA_VERS, PROTO-
COL)) == NULL )
  {
         /* couldn't get connection to server */
    clnt_pcreateerror(server);
    exit(2);
  }
  ...
if (argc == 3) /* Start der Rechnung */
  {
```

```
    i=primstart_1(&j, cl);
    printf("%d\n",*i);
  }

for(;;)
  {
    rueck=primcalc_1(&hin,cl);
    for (m=0; m<NUMBEROFPRIME; m++)
/*Initialisierung der naechsten Runde */
         hin.primlist[m]=0;
    printf("Von %d bis %d\n",
         rueck->low, rueck->high-1);
if ((rueck->low == 0) && (rueck->high == 0)){
/* Ende der Rechnung */
   printf("\nAnzahl Runden: %d\n",k);
   printf("Anzahl Primzahlen : %d\n",l2);
         clnt_destroy(cl);
         exit (0);
  }
else
  if ((rueck->low == 1)
    && (rueck->high == 1))
/*warten auf Beginn der Rechnung */
      sleep(1);
else /* Rechnung */
    {
    for (j=rueck->low;j<rueck->high;j++)
      {  ... }
    hin.low=rueck->low;
    hin.high=rueck->high;
/* Rueckgabe der letzten Runde */
    }
  }
         clnt_destroy(cl);
         exit (0);
}
```

4.0 Implementation der einzelnen Probleme:

4.1 Allgemeine Bemerkungen

Ein RPC-Programm besteht aus dem Interface-Modul (*.x) und den C-Programmen für Client und Server. Aus dem Interface-Modul werden durch den RPC-Compiler rpcgen ein Headerfile und drei C-Files (Client-Stub, Server-Stub und XDR-Konvertierung) erzeugt.

Der Konvention gemäß wird in den *.x - Files das Interface-Modul implementiert. Konkret bedeutet das die Deklaration der benutzten Datentypen und die Deklaration der Remote-Funktionen. Weiterhin werden Programm-Nummer und Versions-Nummer des RPC-Programms vereinbart. An dieser Stelle zeigt sich auch ein großer Nachteil des RPC-Konzeptes, denn jede dieser Nummern darf es in einem Netz nur einmal geben, und sie wird statisch einkompiliert, kann also im Nachhinein nicht mehr geändert werden. Damit kann auf einem Rechner ein und dasselbe RPC-Programm (z.B. das der Primzahlberech-

nung) nur einmal ausgeführt werden, ein Umstand, der in Multiuser-Umgebungen nicht unbedingt gegeben ist.

Im folgenden zeige ich die Interface-Module der drei Beispielprogramme, um dann jeweils auf die Details des Algorithmus einzugehen.

4.2 Primzahlen

```
const NUMBEROFPRIME = 2000;

struct prims {
unsigned int low;
unsigned int high;
unsigned int primlist[NUMBEROFPRIME];
};

struct primcomm {
unsigned int low;
unsigned int high;
};

program PARA {
   version PARA_VERS {
         primcomm PRIMCALC(prims) = 1;
         int PRIMSTART(int) = 2;
   } = 1;                /* version number   = 1 */
} = 0x31111444;
/* program number = 0x31111444 */
```

Bild 23: Primzahl-Interface Modul

Der Algorithmus

Der verteilte Primzahlalgorithmus benutzt einen Trick, der vor allem am Anfang die Kommunikation erheblich verringert und das ganze Programm beschleunigt. Um x^2 Zahlen zu testen, muß man die Primzahlen bis x testen. Übliche Algorithmen benutzen diese Idee rekursiv, indem mit Hilfe der zuerst ermittelten Primzahlen die nächsten berechnet werden usw. In parallelen Umgebungen wirft das das Problem der Datenverteilung auf, denn Client A muß die von Client B berechneten Primzahlen kennen und umgekehrt. Bei realistischer Problemgröße, zum Beispiel wie in meinem Programm die Primzahlen bis 4 Millionen, müssen also zumindest die Primzahlen bis 2000 (das sind 305 Stück) bekannt sein, also pro Client etwa 300 RPC-Rufe durchgeführt werden. Kommunikation ist bei der Rechengeschwindigkeit heutiger Workstations teurer als Berechnung, so daß meine Clients zu Beginn diese 305 Primzahlen sequentiell berechnen (das dauert aufgrund der kleinen Zahlen nicht viel länger, als man braucht, um den Finger wieder von der Enter-Taste zu nehmen). Auf dieser Basis können die Clients dann völlig unabhängig weiterrechnen.

Da, wie oben bereits beschrieben, dem Server die Anzahl seiner Clients unbekannt ist, würde eine statische Lastverteilung, die jedem Prozessor einen Bruchteil der Aufgabe fest zuordnet, größere Schwierigkeiten beim Programmieren machen als eine dynamische Lastverteilung. Ich habe sie dadurch erreicht, daß der Server jeweils verhältnismäßig

kleine Aufgaben bereitstellt (64000 Zahlen zum Testen), so daß die Gesamtzahl der Aufgaben um ein Vielfaches höher ist als die Zahl der Prozessoren. Natürlich bringt das einen Mehraufwand an Kommunikation, doch der ist gering, wenn man ihn mit dem Zeitverlust vergleicht, der bei der statischen Methode entsteht (alle müssen warten, bis der letzte fertig ist). So ein Zustand entsteht auch bei meiner Methode am Ende, doch durch die kleinen Aufgaben ist auch die Wartezeit gering.

Das *.x File weiter oben zeigt bereits die verwendeten Datentypen: Mit der Struktur prims wird ein berechneter Bereich zurückgegeben (dafür die Grenzen, das Feld enthält die Primzahlen in geeigneter Kodierung, s.u.) und ein neuer angefordert, der mit der Struktur primcomm als Rückgabe übergeben wird. Die Funktion primcalc dient dieser Datenverteilung, primstart ist für die oben erwähnte Start-Synchronisierung zuständig. RPC ist auf 8 KByte Daten beschränkt, wenn man UDP benutzt, was dazu führt, daß man die zurückgegebenen Datenmengen klein halten muß. Darum sind die Primzahlen in ein Int-Array der Länge 2000 kodiert, das sind 8000 Byte zuzüglich der Bytes für die Begrenzungen. Bei 32 Bit Int-Länge sind das insgesamt 64000 Bits, die für jeweils eine getestete Zahl stehen. Das ist übrigens auch der Grund für die Wahl von Stücken von jeweils 64000 Zahlen.

4.3 Fraktal

```
const LINEX = 1000;
const LINEY = 1000;
const MAXIT = 255;

struct fracline {
unsigned int linenumber;
char line[LINEX];
};

program FRAC {
   version FRAC_VERS {
         int FRACCALC(fracline) = 1;
         int FRACSTART(int) = 2;
   } = 1;                        /* version number   = 1
*/
} = 0x31111448;
/* program number = 0x31111448 */
```

Bild 24: Fraktal-Interface Modul

Der Algorithmus

Der Fraktal-Algorithmus ist der einfachste in diesem Beitrag, da es keinerlei Notwendigkeit für eine Kommunikation zwischen den Clients gibt. Grundsätzlich entspricht er dem Primzahl-Algorithmus, wenn man von den geänderten Bezeichnungen für die Datentypen absieht. Die naheliegende Einteilung sind diesmal Zeilen des zu berechnenden Fraktals, bei 1000 Länge und Kodierung als Char gibt es keine Probleme mit der 8 KB Grenze. Insgesamt gibt es 1000 Zeilen, die durch dynamische Lastverteilung auf die Clients verteilt werden. Da die Antworten jeweils die Zeilennummer enthalten, ist der "Zusammenbau" des Ergebnisses für den Server kein Problem mehr.

4.4 Gauß’sches Eliminationsverfahren

```
const LINEX = 200; /* Anzahl Zeilen */
const LINEY = 200; /* Anzahl Spalten */
struct matline {
unsigned int linenumber1; /*wer*/
unsigned int linenumber2; /*womit*/
double line[LINEX];
};

struct reduceline {
unsigned int linenumber1; /*wer*/
unsigned int linenumber2; /*womit*/
double line1[LINEX]; /*wer*/
double line2[LINEX]; /*womit*/
};

program MAT {
   version MAT_VERS {
         reduceline MATCALC(matline) = 1;
         int MATSTART(int) = 2;
   } = 1;       /* version number   = 1 */
} = 0x31111488; /* program number = 0x31111488 */
```

Bild 25: Gauß’sche Elimination-Interface Modul

Der Algorithmus

Das Gauß’sche Eliminationsverfahren ist in diesem Beitrag der am schwersten zu parallelisierende Algorithmus. Die Probleme entstehen vor allem dadurch, daß massiv Kommunikation zwischen Clients nötig ist. Damit entsteht zum ersten Mal die Notwendigkeit, gleichzeitige Zugriffe auf gewisse Datenelemente auszuschließen.

Ich bin vom sequentiellen Ansatz ausgegangen und habe daraus Aufgaben abgeleitet, die parallel abgearbeitet werden können. Zuerst nimmt man Zeile 0 und eliminiert damit in allen übrigen (darunter liegenden) Zeilen das vorderste Element, dann nimmt man Zeile 1 und eliminiert damit, usw. Ehe man mit einer Zeile eliminieren kann, muß sie normalisiert werden, also die 1 auf der Hauptdiagonalen erzeugt werden. Damit ist eine feste Reihenfolge der Aufgaben gefunden, die vom Server in dieser Reihenfolge angeboten werden können. Probleme ergeben sich bei den Schreibzugriffen, denn eine Zeile, die herausgereicht wurde, um sie zu ändern, darf keinem anderen Client gegeben werden (weder zum eliminieren, noch, um damit zu eliminieren). Weiterhin stellt sich die Frage, wann man normalisieren darf. Letztere Frage ist leicht zu beantworten, die Zeile x ist nämlich "fertig" und kann normalisiert werden, nachdem sie mit x-1 eliminiert wurde. Der Server prüft also zurückkommende Ergebnisse und normalisiert selber (das geht schneller als die zweimalige Kommunikation, wenn dies ein anderer tun müßte, außerdem wird wesentlich seltener normalisiert als eliminiert). Das Problem des Schreibzugriffs wird dadurch gelöst, daß der Server aufgrund der Tatsache, daß er ganz genau weiß, wann Schreibzugriffe beabsichtigt sind, diese über eine einfache Einrichtung sequentialisieren kann: Es gibt ein Feld int lock[Anzahl Matrixzeilen], in das eine 1 eingetragen wird, wenn die entsprechende Zeile für einen Schreibzugriff (also zum eliminieren) herausgereicht wurde. Vor

jeder Benutzung einer Zeile wird lock getestet, falls gesetzt, so wird jedem Client, der eine Aufgabe haben will, die leere Aufgabe übergeben. Um zu keiner Überlast durch rasant aufeinander folgende Rufe aller Clients zu kommen, führen diese ein busy wait durch, indem sie die Elimination mit der gesperrten Zeile rechnen, um Zeit zu verbrauchen.

5.0 Messungen an den Algorithmen

5.1 Die Umgebung

Alle Messungen habe ich im Sun-Pool Raum 429 (Rechner-Pool des Lehrstuhls von Pr. Starke) durchgeführt. Dort stehen 5 Axil 311 und 5 Axil 235 Workstations zur Verfügung, beides sind Nachbauten der SparcStation 10, wobei die Axil 311 über 1 MB Onboard-Cache verfügen und auch von der Architektur her etwa schneller sind. Ich habe eine Axil 311 als Server benutzt; und je 4 Axil 311 und Axil 235 als Clients, bei den Testläufen mit weniger als acht Rechnern dabei die Axil 311 bevorzugt.

5.2 Primzahlen

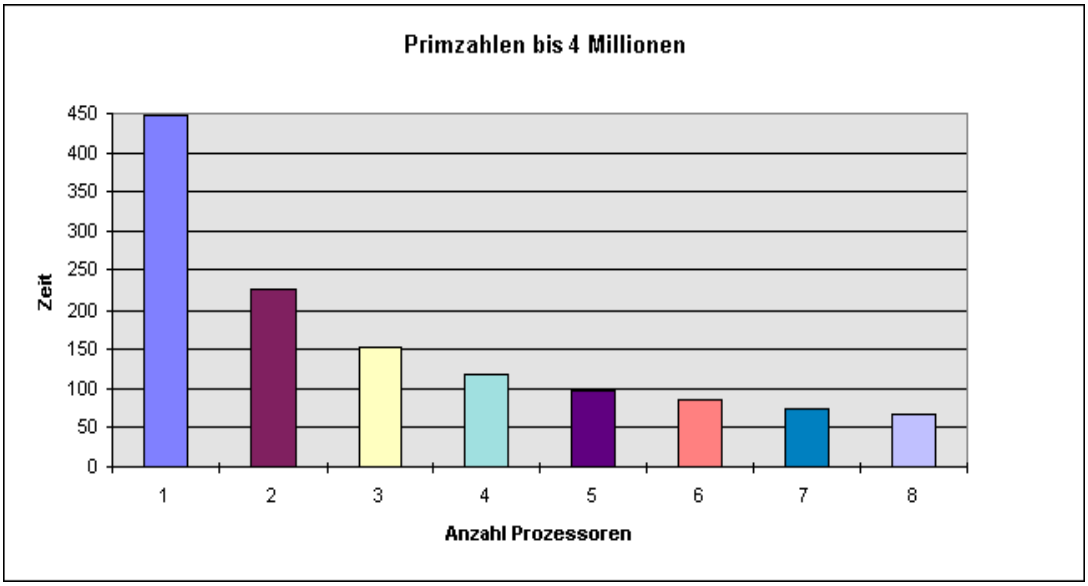

Bild 26: Performance des parallelen Primzahltest-Programmes

Das Ergebnis in Bild 26 entspricht den Erwartungen, die leichte Un-Linearität ab 5 Rechnern entsteht durch die langsameren Axil 235. Der Algorithmus gibt die Anzahl der Zahlen aus, die jeder Prozessor getestet hat. Bei dem Testaluf mit 8 Rechnern haben die Axil 311 etwa je 512000 Zahlen getestet, und die Axil 235 je 448000. An dieser Stelle zeigt sich die Wirkung der dynamischen Lastverteilung.

5.3 Fraktal

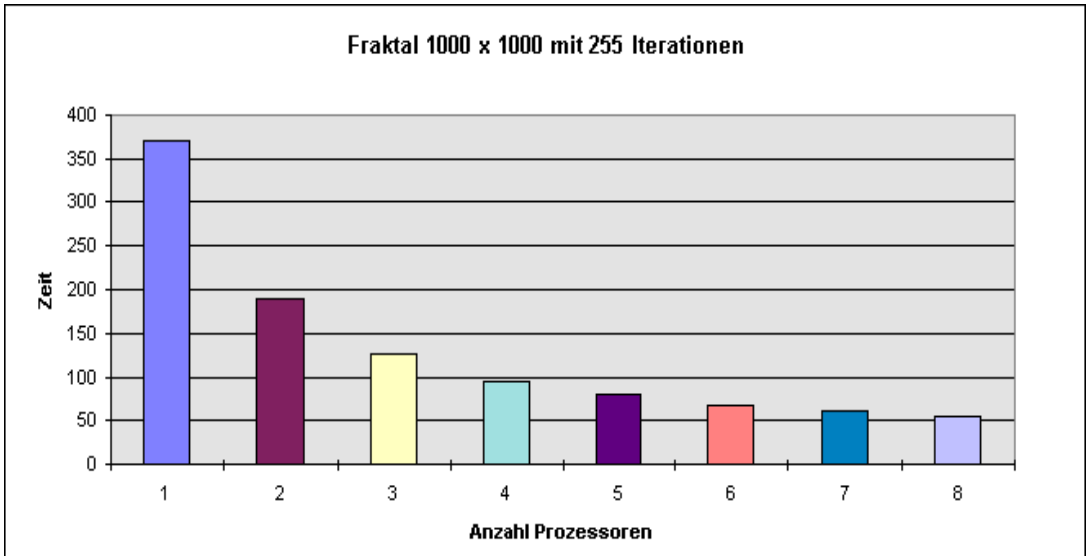

Bild 27: Resultate des Fraktal-Programmes

Auch bei der Mandelbrot-Berechnung entspricht der Zeitverlauf den Erwartungen, denn auch hier ist der Anteil der Kommunikation zu klein, um sehr zu bremsen (siehe Bild 27).

5.4 Gauß'sches Eliminationsverfahren

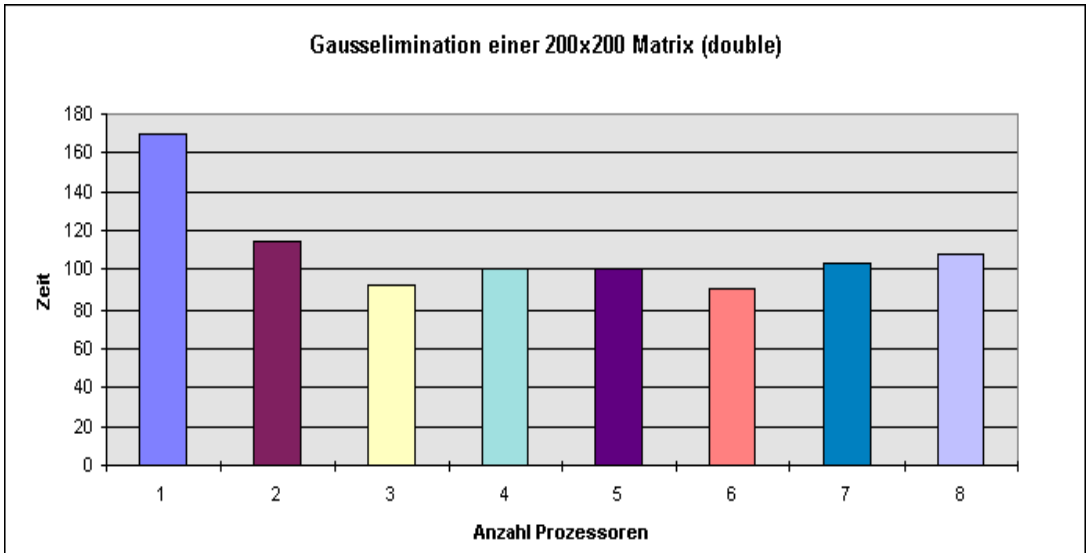

Bild 28: Resultate beim Gauß'schen Eliminationsverfahren

Ganz anders verhält es sich jedoch bei dem sehr kommunikationsintensiven Matrix-Algorithmus. Die oben erwähnte Blockade tritt in der Realität sehr oft auf, was zu einer zusätzlichen Kommunikation führt. Insgesamt, also ohne Blockaden, finden 19900 RPC-Rufe statt, die jeweils 3 Matrixzeilen (200 double, also 1,6 KB) übertragen. Insgesamt bedeutet das eine Datenmenge von 95 MB, die innerhalb kurzer Zeit über das Ethernet übertragen werden müssen - dieses stößt dabei, wie in Bild 28 ersichtlich, an seine Grenzen.

6.0 Nachteile von verteilten Berechnungen mit RPC

- RPC überläßt den Aufbau einer "virtuellen Maschine" dem Nutzer, so daß einige Überlegungen nötig sind, die bei anderen Systemen nicht gebraucht werden.
- wie oben schon erwähnt ist RPC durch die feste RPC-Nummer unflexibel, vor allem bei vielen Programmen in großen Netzen kann es Probleme geben.

7.0 Zusammenfassung

Nach diesen, zum Teil sehr befriedigenden Ergebnissen stellt sich die Frage, welche Probleme mit dem hier vorgestellten Verfahren effizient parallel gelöst werden können. Dazu kann man eine Reihe von Bedingungen angeben:

- Das Problem muß in eine Folge von Teilproblemen zerlegbar sein (nicht zu groß, um dynamischen Lastausgleich zu optimieren und den Einsatz vieler Worker zu ermöglichen, aber auch nicht zu klein, um Netzlast zu beschränken), die parallel oder mit Einschränkungen parallel abgearbeitet werden können.
- Die einzelnen Teilprobleme sollen möglichst wenige Abhängigkeiten haben.
- Die zu übermittelnden Datenmengen sollen im Verhältnis zur Rechengeschwindigkeit der Prozessorknoten (= Workstations) so klein bleiben, daß das Ethernet nicht zum Engpaß wird.
- Die Folgen von Teilproblemen, die keine Abhängigkeiten untereinander haben, sollen möglichst lang sein.

8.0 Ausblicke und weitere Möglichkeiten - Fehlertoleranz

Durch den Umstand, daß dem Server die genaue Anzahl der Clients egal ist, kann auf eine recht einfache Weise eine Fehlertoleranz gegenüber Ausfall von Clients realisiert werden. Der umgekehrte Fall - Ausfall des Servers, ist schwieriger handzuhaben, bietet aber in Verbindung mit der rasch implementierten Sicherheit gegen Client-Ausfall eine völlige Fehlertoleranz

8.1 Toleranz gegen Client-Ausfall

Das System rechnet ohne Schwierigkeiten weiter, wenn ein Client mitten in der Rechnung ausfällt. Das einzige Problem sind die Resultate, die dabei verlorengehen. Eine Lösungsidee wäre es, dem Server eine Datenstruktur (Feld) zu geben, die für jede ausgegebene Aufgabe eine Art Zeittabelle führt, die den Zeitpunkt der Ausgabe registriert. Mit Hilfe eines Signals (SIGALRM) kann dann in festgelegten Abständen ein Signalhandler im Server (ist nötig, da Serverroutinen sonst nur nach Ruf durch Client laufen) prüfen, welche Aufgabe "überfällig" ist. "Überfällig" müßte man in Abhängigkeit von der Aufgabe definieren. Wenn eine solche Aufgabe gefunden wird, so wird sie in eine ToDo-Liste eingefügt, die vor den normalen Aufgaben Priorität hat, was bewirkt, daß der nächste Client sie erneut erhält. Bei den Beispielprogrammen wäre das leicht zu realisieren, lediglich im Fall der Gauss-Elimination würde es nach Client-Ausfall Wartezeiten für alle Clients geben, in jedem Fall würde das System aber weiterarbeiten. Man könnte auch einen Zähler für die

Clients mitlaufen lassen und gegebenenfalls vom Server neue Clients starten lassen, falls dies nötig ist.

8.2 Toleranz gegen Server-Ausfall

Dieser Fall kann durch den Umstand abgehandelt werden, daß ein Client auch mehrer Server rufen kann. Im Gegensatz zum Server merkt ein Client auch, wenn ein Server ausgefallen ist (RPC-Ruf kehrt mit Fehlermeldung zurück). Bei mehreren Servern gibt es also einen, der Aufgaben verteilt. Die Resultate werden von den Clients an alle Server weitergegeben, was besonders bei kommunikationsintensiven Problemen eine erhebliche Netzlast bringt. Die nichtarbeitenden Server protokollieren alles mit und können im Fehlerfall übernehmen, da sie den aktuellen Stand und alle Resultate kennen. Problem wäre, wie der nächste Server vom Ausfall erfährt - Lösung könnte eine Numerierung der Server sein, verbunden mit einer Nachricht der Clients, die den Ausfall festgestellt haben (jeweils den mit der nächsthöheren Nummer).

PVM
The Parallel Virtual Machine

Thomas Röblitz
roeblitz@informatik.hu-berlin.de

1.0 Einleitung

Dieser Artikel gibt eine Zusammenfassung meiner Arbeiten mit PVM im Rahmen eines Vorlesungsprojektes. Ziel dieses Projektes war es, drei Algorithmen mit Hilfe von PVM zu implementieren und dabei Erfahrungen im Umgang mit PVM zu sammeln.

Zunächst werde ich die verwendete Umgebung beschreiben und dabei auch kurz auf PVM eingehen. Im Anschluß lege ich meine Ideen für die Implementierungen dar. Schließlich stelle ich meine Ergebnisse vor und diskutiere sie am Ende dieses Artikels.

2.0 Die Umgebung

PVM ist ein Softwaresystem, welches das Rechnen in Netzwerkumgebungen mit Rechnern unterschiedlicher Architektur erlaubt. Durch PVM wird ein Parallelrechner mit verteiltem Speicher simuliert. Alle Rechner, auf die man Zugriff hat, können zu einem solchen Rechner konfiguriert werden.

2.1 Hardware

Für meine Testläufe verwendete ich Rechner verschiedener Pools, die alle über ein Ethernet miteinander verbunden sind. Ich habe dabei Rechner folgenden Typs benutzt:

- Sun SPARCstation 5
- Sun SPARCstation 10
- Axil HWS 210

Auf allen Rechner ist SunOS installiert. Für die Grafikausgaben verwendete ich eine Maschine mit X11R6.

2.2 Software

Das PVM System besteht aus zwei Komponenten, einer Bibliothek mit Programmierschnittstelle und einem Programm, welches auf jedem Knoten der virtuellen Maschine läuft. Diese Dämonen sorgen für den Datentransfer zwischen den Knoten(messages). Es unterstützt ein einfaches, aber funktionell vollständiges Message Passing Modell. Eine ausführliche Einführung in PVM findet man in [Sunderam et al.].

3.0 Die Algorithmen

3.1 Primzahlen

Die Aufgabe besteht darin, alle Primzahlen in einem bestimmten Intervall zu berechnen. Sei *min* die untere und *max* die obere Grenze des Intervalls.

Angenommen, es stehen *n* Knoten für die Berechnungen zur Verfügung. Ein Knoten übernimmt die Aufgabe des Masters. Die verbleibenden *n-1* Knoten sind die Worker und führen die Berechnungen durch. Der Master wertet die Eingangsparameter aus und berechnet alle Primzahlen von Null bis zur Wurzel der oberen Grenze (*max*). Danach sendet er die Konfigurationsdaten(Anzahl der Worker, *min*, *max*, berechnete Primzahlen, *id* des Worker) an die Worker. Schließlich erwartet er die Ergebnisse der Worker.

Ein Worker empfängt zunächst die Konfigurationsdaten vom Master. Anhand seiner *id* berechnet er alle Primzahlen in einem Teil der Ausgangsmenge [*min*, *max*]. Die Ausgangsmenge wird in *m*(n-1)* gleich große Bereiche zerlegt(*m* Element {1,...}, *m* kann durch einen Parameter spezifiziert werden). Jeder Knoten führt die Berechnungen für *(max-min)/(n-1)* Zahlen durch. Die Datenverteilung ist statisch und wird durch die *id* des Workers, die Anzahl der Worker und die Grenzen des Intervalles *min*, *max* festgelegt(algorithmisch während der Laufzeit bestimmt). Nach der Berechnung eines Teilstükkes werden die Primzahlen an den Master geschickt. Jeder Knoten schickt *m*-mal seine Teilergebnisse an den Master.

3.2 Fraktale

Hier ist die Aufgabe, die Mandelbrotmenge für ein Teilgebiet des R^2 zu bestimmen. Dieses sei durch die Koordinaten der linken unteren und rechten oberen Ecke eindeutig festgelegt. Weiterhin benötigt man die Größe des Darstellungsbereiches in Punkten(Bildpunkte).

Die Anzahl der beteiligten Knoten sei wieder *n*. Die Grundidee für den Algorithmus ist die gleiche, wie bei der Primzahlberechnung. Ein Knoten übernimmt wieder die Aufgabe des Masters. Dieser führt die Initialisierungen durch, empfängt von den Workern zeilenweise die Ergebnisse und stellt sie graphisch dar. Ein Worker bearbeitet immer eine Zeile und schickt sie dann zum Master. Der gesamte Bildbereich ist in *y* Zeilen zerlegt. Welche Zeile ein Worker bearbeitet, wird durch die *id* des Workers, die Anzahl der Worker und die Größe des Bildbereichs statisch festgelegt(vom Worker selbst "berechnet"). Bearbeitet ein Worker gerade die *x*-te Zeile, so wird er als nächstes die *x+(n-1)* -te Zeile(falls diese noch im Bildbereich liegt) bearbeiten.

3.3 Gauß'sches Eliminierungsverfahren

Gegeben sei eine reguläre *n x n* Matrix(*A*). Die Matrix soll in eine obere Dreiecksmatrix umgeformt werden. Unter *m* Knoten wird ein Master ausgewählt. Dieser führt Initialisierungen durch und wartet nach dem Starten der Worker auf Zeilen, die schon bearbeitet wurden. Die anderen *m-1* Knoten sind die Worker.

Jeder Worker bearbeitet einen festgelegten Teil der Matrix. Dieser Teil setzt sich aus einzelnen Zeilen zusammen, die wie bei der Fraktalberechnung verteilt sind. Der Knoten der die erste Zeile bearbeitet, beginnt mit dem Divisionsschritt. Während dieser Zeit ruhen die anderen Knoten. Sie warten auf das Ergebnis der Division. Nachdem der erste Knoten den Divisionsschritt durchgeführt hat, schickt er das Ergebnis an alle anderen Knoten(auch an den Master). Nun führen alle Knoten den Eliminierungsschritt für ihre Zeilen durch. Danach beginnt der Knoten, dem die nächste Zeile zugeordnet ist, mit dem Divisionsschritt, u.s.w. Ein Knoten beendet seine Arbeit, wenn alle Zeilen, die ihm zugeordnet wurden, bearbeitet wurden. Der Master beendet seine Arbeit nach Erhalt der letzten Zeile, d.h., wenn die Matrix in die gewünschte Form überführt wurde.

4.0 Messungen

Bei allen Algorithmen habe ich jeweils die Ausführungszeit in Sekunden gemessen. Die verwendeten Rechner befanden sich während meiner Messungen im normalen Betrieb, d.h., daß auch andere Nutzer an diesen arbeiteten. Die sich daraus ergebenden Einflüsse auf die Laufzeiten habe ich nicht näher untersucht. Bei den Primzahlen und den Fraktalen waren diese aufgrund der statischen Datenverteilung(jeder Knoten hatte praktisch die gleiche Anzahl von “Operationen” auszuführen) vernachlässigbar. Die Einflüsse bei dem Gauß’schen Eliminierungsverfahren, bei dem sich die Rechner bei Austausch einer Zeile nach dem Divisionsschritt immer wieder synchronisierten, waren natürlich nicht vernachlässigbar. Allerdings ist hier auch nicht mit einem ähnlichen Verhalten, wie bei den anderen Algorithmen zu rechnen gewesen. Der zu erwartende Effekt konnte trotzdem beobachtet werden.

4.1 Primzahlen

Hier habe ich bis zu 15 Worker verwendet. Als Bereich habe ich 1 - 3.603.600 gewählt. Diese “krumme” Zahl hat den Vorteil, daß sie durch alle Zahlen von 1 bis 15 teilbar ist. Dadurch wird erreicht, daß alle Knoten ungefähr die gleiche Anzahl an Operationen durchzuführen haben. In Bild 29 sieht man die gemessenen Zeiten (in Sekunden) als Balken dargestellt. Die angedeutete Kurve zeigt einen linearen Speedup.

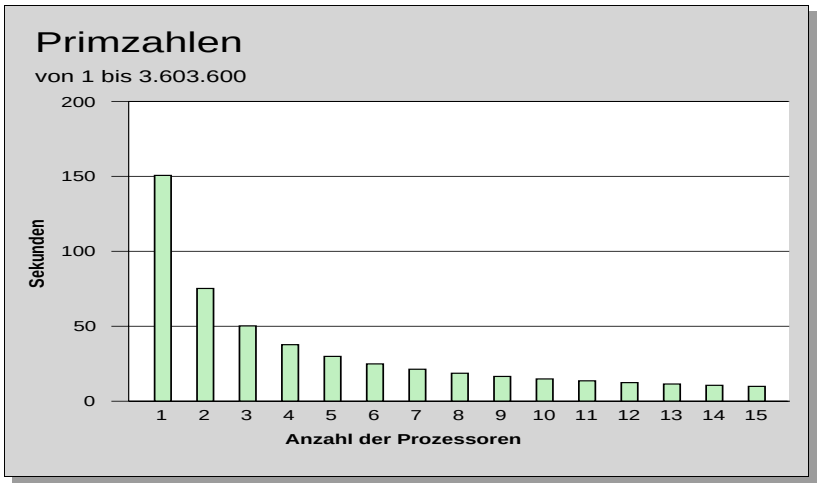

Bild 29: Meßergebnisse für den Primzahlalgorithmus

4.2 Fraktale

Bei meinen Messungen habe ich wieder bis zu 15 Worker eingesetzt. Für die Koordinaten habe ich die Punkte (-2,-2) (linke untere Ecke) und (2,2) (rechte obere Ecke) verwendet. Der Bildbereich betrug 800 x 800 Punkte, wobei bis zu 500 Iterationen durchgeführt werden sollten. Bild 30 zeigt die gemessenen Werte in Sekunden. Auch bei diesem Algorithmus ergibt sich ein linearer Speedup.

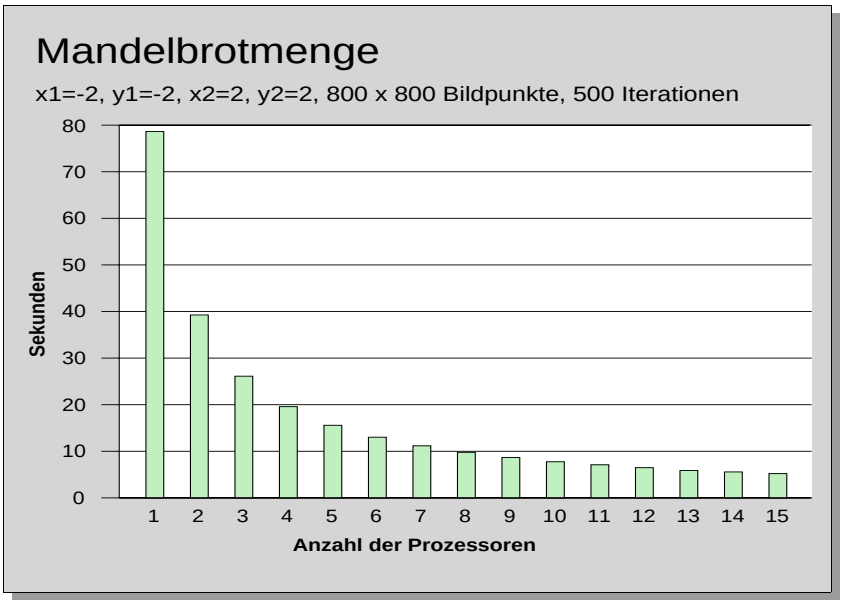

Bild 30: Meßergebnisse für den Fraktalalgorithmus

4.3 Gauß'sches Eliminierungsverfahren

Hierbei habe ich reguläre quadratische Matrizen in den Größen 100, 200 und 500 verwendet, wobei ich bis zu 10 Worker eingesetzt habe.

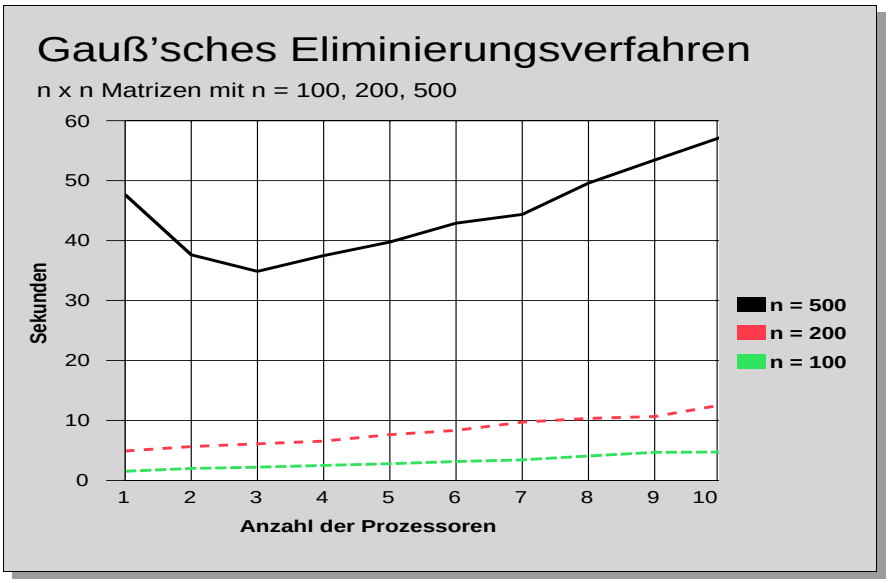

Bild 31: Meßergebnisse für den Gaußalgorithmus

Das Bild 31 zeigt alle gemessenen Werte(die Zahlen in den Klammern geben die Überhöhung an). Es zeigt sich, daß sich erst ab einer bestimmten Matrixgröße ein gewisser Speedup einstellt. Außerdem sieht man, daß man nicht beliebig viele Prozessoren(Knoten) hinzunehmen kann, da sich die Laufzeiten wieder erhöhen. Warum dies so ist, versuche ich, unter Punkt 5 zu erläutern.

5.0 Diskussion und Erfahrungsbericht

5.1 Diskussion der Meßergebnisse

Beginnen möchte ich mit den ersten beiden Algorithmen. Bei beiden war ein linearer Speedup zu beobachten. Das ist darauf zurückzuführen, daß die Worker nie untereinander kommunizieren mußten. Ihre Laufzeit hing also nur von der jeweiligen aktuellen Belastung durch andere Nutzer ab. Natürlich mußten auch die Worker Nachrichten mit dem Master austauschen, wenn sie ihre Ergebnisse übermittelten. Dieser Datenaustausch fällt aber kaum ins Gewicht, sonst hätte man keinen solchen Speedup beobachten können. Entscheidend für die Laufzeit ist vielmehr die Wahl der Datenverteilung, insbesondere bei statischer. Zerlegt man die Bildmatrix bei der Fraktalberechnung nicht in dünne Streifen(1 Pixel dick), sondern in *n* gleich große Streifen(bei *n* Workern), hängt die Laufzeit davon ab, welches Stück ein Knoten bekommt. Bei dem typischen Apfelmännchenbild muß der Knoten, der einen Strei-fen aus der Mitte zugeordnet bekommt, wesentlich mehr Berechnungen durchführen, als einer, der ein Randstück zu bearbeiten hat.

Bei den Primzahlenberechnungen sind solche Effekte nicht zu beobachten. Unterschiedliche Laufzeiten gleicher Rechner sind hierbei auf die unterschiedliche Anzahl der Tests zurückzuführen, die benötigt werden, um zu entscheiden, ob eine Zahl eine Primzahl ist oder nicht.

Allgemein läßt sich sagen, daß eine statische Datenverteilung günstig ist, wenn alle Knoten vom gleichen Rechnertyp sind und man davon ausgehen kann, daß alle die gleiche Rechenleistung erbringen(die lokale Last also vernachlässigbar ist). Eine dynamische Datenverteilung ist nicht nur in dem anderen Fall, d.h. unterschiedliche Rechenleistung der Knoten(durch Typ oder Last verursacht), empfehlenswert, sondern auch dann, wenn nur schwierig abschätzbar(bzw. berechenbar) ist, wie aufwendig einzelne Teilaufgaben sind. Bei den Fraktalberechnungen müßte man bei statischer Datenverteilung gewährleisten können, daß alle Streifen die gleiche Rechenarbeit erfordern. Sonst sind einige Knoten sehr schnell fertig, während andere noch viel zu tun haben. Dieser Fall tritt natürlich besonders häufig in Rechnerpools, in denen gleichzeitig von anderen Nutzern gearbeitet wird, auf. In diesem Fall sollte man den höheren Kommunikationsaufwand in Kauf nehmen.

Nun werde ich versuchen, die Ergebnisse der Messungen für das Gauß’sche Eliminierungsverfahren zu erklären. Man sieht zunächst, daß sich bei kleinen Matrizen überhaupt kein Speedup ergibt. Bei den größeren Matrizen ist zunächst eine Beschleunigung zu beobachten, nimmt man aber weitere Knoten hinzu, steigt auch hier die Laufzeit wieder an. Da die Anzahl der Berechnungen, die notwendig waren, um eine Matrix(es wurde immer die gleiche Ausgangsmatrix verwendet) umzuformen, konstant war, kann der

Anstieg der Laufzeit nur an einer Zunahme der Kommunikationszeit oder einer höheren Auslastung eines Knotens, welcher dann alle anderen “mitbremst”, liegen. Den zweiten Fall kann ich allerdings ausschliessen, da ich bei meinen Messungen für diesen Algorithmus darauf geachtet habe, daß alle benutzten Rechner nur mit meinen Berechnungen beschäftigt waren. In der Tat nimmt der Kommunikationsaufwand schnell zu. Dazu habe ich Messungen durchgeführt, bei denen keine Berechnungen vollzogen wurden. Dabei habe ich festge-stellt, daß die Laufzeit nahezu linear zunimmt, wenn man mehr Knoten einsetzt. Während bei den beiden anderen Problemen immer nur direkte Kommunikation zwischen dem Master und einem Worker stattgefunden hat, findet hier die Kommunikation immer zwischen allen Knoten(Worker & Master) mittels Broadcast statt. Darin sehe ich die Ursache für die Zunahme der Kommunikationszeit.

5.2 Erfahrungen mit PVM

PVM unterstützt das Message Passing Modell. Die Funktionen zum Datenaustausch sind komfortabel benutzbar. Der Sender einer Nachricht kann Daten verschiedenen Typs nacheinander in einen Puffer schreiben. Dazu existieren für jeden Typ Bibiliotheksfunktionen. Der Empfänger muß diese Daten dann nur in der richtigen Reihenfolge wieder aus einem Puffer entnehmen. Außer einfachen Sende- und Empfangsfunktionen habe ich noch Funktionen zur Verwaltung und Gruppenkommunikation benutzt. Insgesamt hat die Bibliothek einen sehr guten Eindruck bei mir hinterlassen.

Probleme bereitet vor allem das Debugging, da Ausschriften der Worker nur in eine Datei geschrieben werden können. Diese Datei ist jedoch erst nach Beendigung der Prozesse lesbar, wahrscheinlich erst, wenn der Masterprozeß die Funktion *pvm_exit()* aufruft. Gerät ein Knoten in einen Deadlock, sind alle Informationen verloren. So ist das Finden eines Fehlers mitunter sehr mühsam. Ein weiteres kleines Problem besteht in den Fehlermeldungen der Konsole, falls auf einem anderen Knoten der Dämon nicht gestartet werden konnte. Selbst nach Beachtung der Hinweise in [Sunderam et al.] konnte das Problem, daß auf einigen Rechnern der Dämon nicht gestartet werden konnte, nicht behoben werden, obwohl durch den Befehl *rsh host pvmd* der Dämon offensichtlich per Hand gestartet werden konnte.

Rendezvous Software Bus for Developers of Distributed Applications (Teknekron Software Systems, Inc.)

Daniel Runge
runge@informatik.hu-berlin.de

1.0 Der Rendezvous Software Bus

1.1 Was ist der Rendezvous Software Bus

Der Rendezvous Software Bus ist ein Kommunikations Software Paket, das den Datenaustausch über ein Netzwerk ermöglicht. Es stellt alle Softwareunterstützung bereit, die benötigt wird für Netzwerkdatentransport und -repräsentation.

Der Rendezvous Software Bus besteht im Wesentlichen aus zwei Komponenten, dem Rendezvous Application Programming Interface (API) und dem Rendezvous Daemon. Die Rendezvous API Funktionen werden gebraucht, um die Rendezvous Applikationen zu schreiben. Der Rendezvous Daemon ist der Kommunikationsprozeß, der für die eigentliche Kommunikation im Netz verantwortlich ist. Näheres hierzu ist im Kapitel 2 zu finden.

1.2 Vorteile beim Programmieren mit dem Rendezvous Software Bus

Entwickelt wurde der Rendezvous Software Bus, um die Kommunikation zwischen Rechnern eines Netzwerkes zu erleichtern. Es wurde versucht, die Kommunikation möglichst transparent zu gestalten, so daß der Programmierer wenig bis gar nichts über die Struktur des Netzwerkes wissen muß. So entfällt z.B. die Angabe der Netzwerk-Adresse vollständig.

Weitere Vorteile des Software Paketes, die ich an dieser Stelle nur nennen möchte, sind die Unterstützung von Multi-Thread-Anwendungen und die Möglichkeit Broadcast-Messages zu verschicken. Der Rendezvous Software Bus steht für die meisten Hardware- und Softwareplatformen zur Verfügung und weitere sollen in der nahen Zukunft hinzugefügt werden, wie z.B. Microsoft Windows ‘95 und IBM OS/2. Die dadurch erreichte Systemunabhängigkeit ermöglicht eine leichtere Wartung und Portabilität der Applikationen.

2.0 Entwicklung verteilter Applikationen mit dem Rendzvous Software Bus

2.1 Subjekt-basierte Adressierung

Wie in Kapitel 1 schon erwähnt, braucht man für die Kommunikation zwischen Rechner eines Netzwerkes keine Netzwerkadressen anzugeben. Wie aber kann man trotzdem ein-

deutig beschreiben, für wen eine Nachricht bestimmt ist. Dies geschieht mit Hilfe der Subjekt-basierten Adressierung. Verteilte Applikationen, d.h. die Prozesse, die in einem Netz zusammenarbeiten (z.B. Client/Server) verwenden dieselben Namen zum Versenden und Empfangen von Nachrichten, sogenannte Subjekt-Namen, die sich also auf den Inhalt der Nachricht beziehen. Die Syntax von Subjekt-Namen wird im Abschnitt 2.5 behandelt.

2.2 Rendezvous Kommunikation

Rendezvous Applikationen können zwei Arten von Nachrichten senden: Point-to-point Messages und Broadcast Messages. Jedoch ist die Kommunikation immer anonym, egal welche Art

von Kommunikation verwendet wird. Auf Broadcast Messages können alle interessierten Applikationen zugreifen, indem sie auf eine Message mit dem jeweiligen Subjekt-Namen warten. Der Sender einer Broadcast Message muß weder wissen, welche Applikationen auf seine Message warten noch wieviele. Es genügt, den Subjekt-Namen als Adressierung anzugeben. Der Rest wird vom Rendezvous Daemon erledigt. (Siehe Abschnitt 2.4)

Point-to-Point-Messages tragen einen Namen, der den Bestimmungsort eindeutig beschreibt. Das aber heißt, daß nur eine einzige Applikation auf Messages mit diesem Subjekt-Namen hören darf. Wie ausgeschlossen wird, daß noch eine weitere Applikation auf diesen Namen wartet, wird in Abschnitt 2.5 unter Callback-Funktionen behandelt.

Eine weitere Erleichterung, die der Rendezvous Software Bus bietet, ist, daß man sich nicht um Details bei der Paketübertragung kümmern muß. So wird die Segmentierung und das Zusammenfügen großer Nachrichten, die Bestätigung und das erneute Senden von Nachrichten im Falle eines Verlustes und die Garantie, das die Reihenfolge der Nachrichten korrekt ist vom Rendezvous Software Bus übernommen.

2.3 Die Rendezvous Bibliothek

Die Rendezvous Bibliothek besteht aus vier Gruppen von Funktionen. Die erste Gruppe, das Communications API (Application Programming Interface), stellt Funktionen zum Senden und Empfangen von Nachrichten bereit. Dazu gehören Funktionen wie *rv_Init()*, die eine Rendezvous Session erzeugt und initialisiert, oder *rv_Send()*, die dazu dient, Nachrichten zu Applikationen zu schicken, die auf Rendezvous warten. Eine dritte Funktion ist *rv_ListenSubject*, mit der man die gewünschten Informationen empfangen kann.

Die zweite Gruppe von Funktionen ist das Message API, dessen Funktionen dem Ein- und Auspacken von Nachrichten dienen, d.h. diese Funktionen benötigt man, um die in einer Nachricht enthaltenen Informationen lesen zu können und damit nutzbar zu machen.

Die dritte Gruppe heißt Event Manager API und dient der Behandlung asynchroner Nachrichten.

Das Advisory API schließlich enthält Funktionen zur Behandlung von Informations-, Warnungs- und Fehlernachrichten.

2.4 Der Rendezvous Daemon

Der Rendezvous Daemon ist ein Hintergrundprozeß für verläßliche und effiziente Netzwerk-Kommunikation. Auf jedem Rechner, auf dem eine Rendezvous Applikation läuft, läuft auch eine Kopie des Rendezvous Daemon, allerdings nur eine pro Rechner, egal wieviele Applikationen auf diesem Rechner aktiv sind.(Siehe hierzu auch Abb. 2.1) Der Daemon wird automatisch gestartet, wenn die erste Applikation auf einem Rechner gestartet wird.Die Applikationen sind also abhängig vom Daemon, der für sie aber unsichtbar ist. Die Applikationen senden und erhalten Nachrichten, indem sie die Funktionen des Communications API benutzen. Der Daemon sorgt dann dafür, daß die Nachrichten an den richtigen Ort gelangen. Alle Nachrichten, die zwischen Applikationen ausgetauscht werden, passieren den Rendezvous Daemon, auch wenn diese zwischen Applikationen auf einem Rechner ausgetauscht werden.

Leider habe ich nirgends eine genaue Erklärung gefunden, wie der Daemon funktioniert. Meine Idee ist die folgende. Wenn eine Applikation gestartet wird auf einem bestimmten Rechner, so wird zunächst getestet, ob schon ein Daemon läuft. Ist dies nicht der Fall wird ein Daemon gestartet und anschließend wird sich bei ihm angemeldet. Ist schon ein Daemon aktiv,

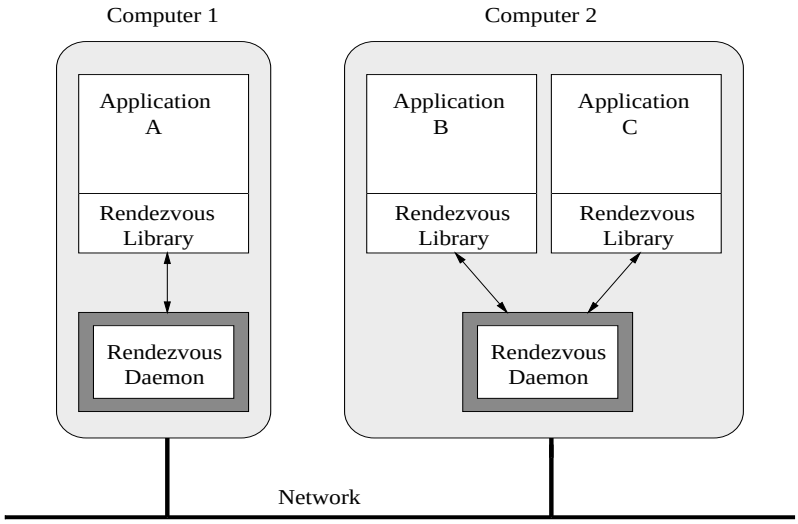

Bild 32: Rendezvous Operations Umgebung

so meldet sich die neue Applikation lediglich beim Daemon an, indem sie ihm mitteilt, auf welche Nachrichten sie wartet. Der Daemon kommuniziert daraufhin mit allen im Netz aktiven anderen Daemons, und teilt ihnen mit, welche Nachrichten zu ihm geschickt werden müssen. Damit wird überflüssige Kommunikation während der Laufzeit vermieden. Würde keine Kommunikation der Daemons stattfinden, müßten immer alle Nachrichten zu jedem Daemon geschickt werden (bzw. sogar zu allen Rechnern im Netz). Die Daemons würden dann erst bei Erhalt einer Nachricht entscheiden, ob sie die Nachricht überhaupt benötigen, d.h. weiterleiten müssen. So ist es auch verständlich, warum man mit einem einzigen Befehl Broadcast Messages verschicken kann. Auch das erledigt der

Daemon, indem er nämlich an alle Daemons, die auf diese Broadcast Message warten, eine einzelne Nachricht schickt.

2.5 Programmierkonzepte

In diesem Abschnitt möchte ich drei wichtige Programmierkonzepte des Rendezvous Software Busses nocheinmal zusammenfassend vorstellen. Intuitiv dürfte wohl jedem klar sein, was eine Message ist und auch das Konzept der Subjekt-Namen sollte schon klar sein. An dieser Stelle möchte ich noch auf einige Details eingehen. Callback Funktionen wurden bisher nur erwähnt. Deshalb hier nun die Erläuterung, was dieses Konzept beinhaltet.

2.5.1 Messages

Informationen, die zwischen Applikationen ausgetauscht werden, sind in eine Message eingeschlossen. Dabei kann diese Message ein opaque binary Puffer sein, dessen Inhalt nur von den eigenen Applikationen verstanden wird. Meistens allerdings beinhaltet eine Message self-desribing data, d.h. um den Inhalt der Message extrahieren zu können ist nicht erforderlich zu wissen, welchen Datentyp die jeweilige Message hat. Jede Message trägt einen Subjekt-Namen (*rv_Name*), der den Inhalt der Nachricht bzw. den Bestimmungsort beschreibt.

2.5.2 Callback Funktionen

Die Funktionen *rv_ListenSubjct* und *rv_ListenInbox* aus dem Communications API assoziieren einen Subjekt-Namen mit einer Callback Funktion, d.h. wenn eine Funktion eine Nachricht erhält, wird dem Subjekt-Namen entsprechend eine Funktion (eine sogenannte Callback Funktion) aufgerufen, die beliebige Aktionen ausführen kann, z.B. die erhaltene Nachricht auswerten etc.

Die Funktion *rv_ListenInbox* ist speziell für point-to-point Messages. Das Öffnen einer Inbox erzeugt einen Endpunkt mit einem global eindeutigen Inbox Namen. Der Inbox Name wird per Broadcast mit Hilfe eines vorher vereinbarten Subjekt-Namens an potentielle Sender verschickt.

2.5.3 Subjekt-Namen

Subjekt-Namen bestehen aus einem oder mehreren Elementen, die jeweils durch einen Punkt (‘.‘) getrennt werden. Die Elemente repräsentieren eine Subjekt-Namen-Hierarchie.

Beim Empfangen von Nachrichten sind Wildcards erlaubt, beim Senden von Nachrichten allerdings nicht. Es gibt zwei verschiedene Wildcards: ‘*‘ und ‘>‘. ‘*‘ steht für genau ein Element an einer beliebigen Stelle. ‘>‘ hingegen steht für ein oder mehrere Elemente, muß aber am weitesten rechts stehen, d.h. rechts davon darf kein weiteres Element stehen.

„*.your.*“ matches z.B. „Amaze.your.friends“ oder „Dam.your.socks“, aber nicht „your“ oder „Pick.up.your.foot“. „RUN.>“ matches „RUN.RUN.RUN“, „RUN.SWIM.BIKE“ oder „RUN.away“, aber nicht „HOME.RUN“ oder „Run.away“.

3.0 Die Algorithmen

Da ich die Implementierung der Algorithmen mit dem Rendezvous Software Bus durchgeführt habe, d.h. also in einem verteilten System, stellte sich mir die Frage gar nicht, ob ich die Probleme mit shared data oder message passing lösen soll. Natürlich ist der einzige Weg hier, der über message passing. Dieser Aspekt wird daher in der Diskussion der einzelnen Algorithmen keine Erwähnung finden.
Alle Algorithmen wurden nach dem Worker/Master-Prinzip bzw. Client/Server-Prinzip implementiert. Hierbei verwaltet der Master die Daten, die auf die Worker zur Berechnung verteilt werden, und die Worker erledigen die eigentliche Berechnung. Nach Beendigung liefern sie die Ergebnisse beim Master ab.
Die Datenverteilung erfolgte bei allen Algorithmen statisch. Eine Diskussion über den Einfluß dieser Einschränkung auf die Ergebnisse erfolgt in den jeweiligen Abschnitten über die Ergebnisse der Tests.

3.1 Verteilte Berechnung von Primzahlen

3.1.1 Implementierung

Der Master erhält über die Eingabezeile die Informationen, in welchen Bereich Primzahlen berechnet werden sollen und wieviel Worker an der Berechnung teilnehmen. Diese Informationen werden den Workern per Broadcast übermittelt. Dabei besteht die Annahme, daß jedem Worker beim Start korrekt mirgeteilt wurde, welche Nummer er hat und das keine Nummer doppelt vorkommt. Jeder Worker berechnet sich jetzt seinen Abschnitt, baut sich eine verkettete Liste der Zahlen auf und beginnt, alle Zahlen, die durch die erste Primzahl (d.h.2) ohne Rest teilbar sind, aus dieser Liste zu entfernen. Der erste Worker hat bald die 3 als Primzahl gefunden und verschickt diese an alle anderen Worker und den Master (per Broadcast). Jede Primzahl die auf diese Weise gefunden wird und deren Quadrat kleiner ist als die Zahl, bis zu der Primzahlen berechnet werden sollen, wird an alle anderen Worker verschickt und in die eigene Primzahlliste eingetragen. Außerdem wird auch jede Primzahl, die ein Worker erhält, in seine private Primzahlliste eingetragen. Ist das Quadrat einer Primzahl größer als die Primzahlgrenze, so wird sie nur an den Master geschickt. Haben alle Worker alle Primzahlen in ihrem Bereich berechnet, teilen sie dies dem Master mit und dieser terminiert.

3.1.2 Ergebnisse der Primzahlberechnung

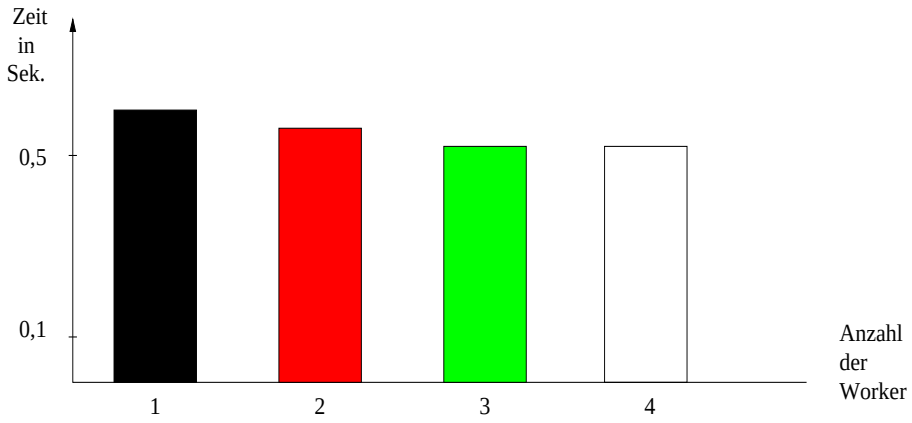

Bild 33: Ergebnisse der Primzahlberechnung bis 1000

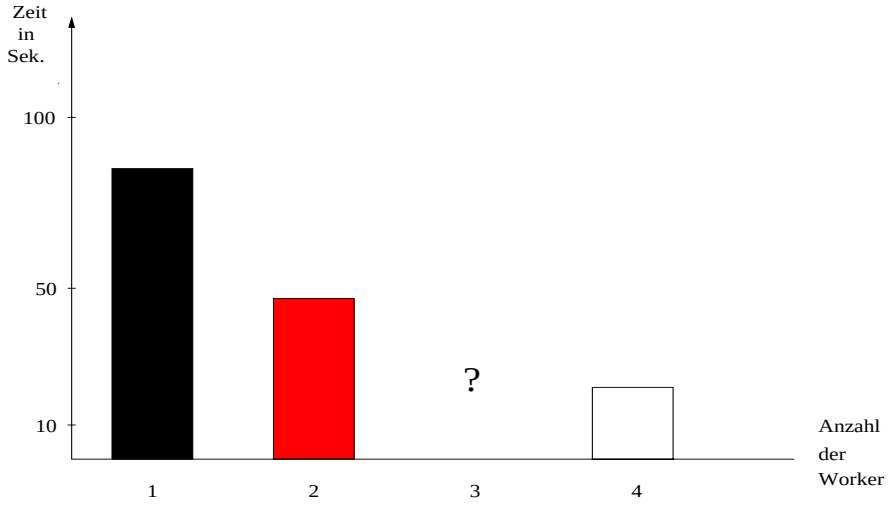

Bild 34: Ergebnisse der Primzahlberechnung bis 100000

Bei der Primzahlberechnung bis 1000 (Bild 33) sind kaum Unterschiede zu erkennen.Das ist damit zu erklären, daß die Berechnung nicht sehr lange dauert im Vergleich zur Kommunikation. Die Kommunikation ist also hier der bestimmende Faktor.

Bei der Primzahlberechnung bis 100000 (Bild 34) sind die Ergebnisse schon zufriedenstellender. Allerdings trat bei 3 Workern ein interessantes Phänomen auf. Die Berechnung terminierte nicht, bzw. nach ein paar Minuten war die Berechnung immer noch nicht abgeschlossen, so daß ich die Berechnung abbrach.

3.1.3 Diskussion

Bei der Primzahlberechnung bis 100000 ist ein guter Speedup zu erkennen bei Erhöhung der Anzahl der Worker. Der Rechenaufwand liegt hier bedeutend höher als der Kommuni-

kationsaufwand. Es lohnt sich also, dieses Problem verteilt zu lösen. Bei der Programmausführung fiel auf, daß die Worker mit kleineren Zahlen früher fertig waren als die mit größeren. Daher würde sich eine dynamische Lastverteilung bei diesem Problem anbieten, d.h. jeder Worker bekommt kleinere Bereiche. Wenn ein Worker fertig ist, wendet er sich an den Master, um einen neuen Auftrag zu bekommen. Der Master ist für die Vergabe der Aufträge verantwortlich. Damit erreicht man eine bessere Auslastung aller Worker. Da zu bestimmten Tageszeiten einige Rechner überlastet sind und andere nicht bzw. einige Rechner einfach schneller sind, erreicht man mit dynamischer Lastverteilung die besten Ergebnisse. Um auch noch die anfängliche Kommunikation einzusparen, könnte man jeden Worker die Primzahlen bis zur Wurzel der Obergrenze selbst berechnen lassen, was im Vergleich zur restlichen Berechnung relativ schnell geht.

Aus der Idee, die gesamte (also auch die anfängliche) Primzahlberechnung verteilt auszuführen, entstand der Zwang, jede Primzahl einzeln zu verschicken. Der Einfachheit halber habe ich die gesamte Kommunikation - auch mit dem Master - auf diese Weise bewerkstelligt. Natürlich wächst dadurch der Kommunikationsaufwand beträchtlich, und es geht viel an Effizienz verloren. Außerdem Tritt bei mehr als 100000 Zahlen der Fakt auf, daß der Bus überlastet ist, und nicht mehr alle Daten korrekt übertragen werden können.

3.2 Verteilte Berechnung von Fraktalen

3.2.1 Implementierung

Wie auch schon bei der Primzahlberechnung, ist auch hier der Master derjenige, der die Berechnung anstößt. Wieder berechnet sich jeder Worker selbst, welche Zeilen ihm zustehen. zwar haben wir wieder eine statische Verteilung, allerdings bekommt diesmal eine Worker keinen zusammenhängenden Bereich, sondern die Zeilen werden zyklisch verteilt.

3.2.2 Ergebnisse der verteilten Fraktalberechnung

Ich habe die Mandelbrotberechnung einmal mit Ausgabe der berechneten Zeilen durch den Master und einmal ohne durchgeführt. Mit Ausgabe der Zeilen war überhaupt kein Speedup zu erkennen. Die Ergebnisse, die man in Bild 35 sieht, sind ohne Ausgabe der Zeilen durch den Master. Er empfnägt zwar alle berechneten Zeilen, die Ausgabe ist aber unterdrückt. Daß kein Speedup zu erkennen ist, liegt daran, daß der Master einen Flaschenhals darstellt. Die Ausgabe der Zeilen ist der aufwändigste Teil. Die Berechnung der Zeilen erfolgt so schnell, daß der Master die Zeile nicht schafft vollständig anzuzeigen, bevor ihn die nächste Zeile erreicht. Abgesehen davon ist auch hier ein guter Speedup zu erkennen. Allerdings bleibt die Frage, was einem die schnelle Berechnung der Zeilen bringt, wenn man sie nicht ebenso schnell anzeigen kann.

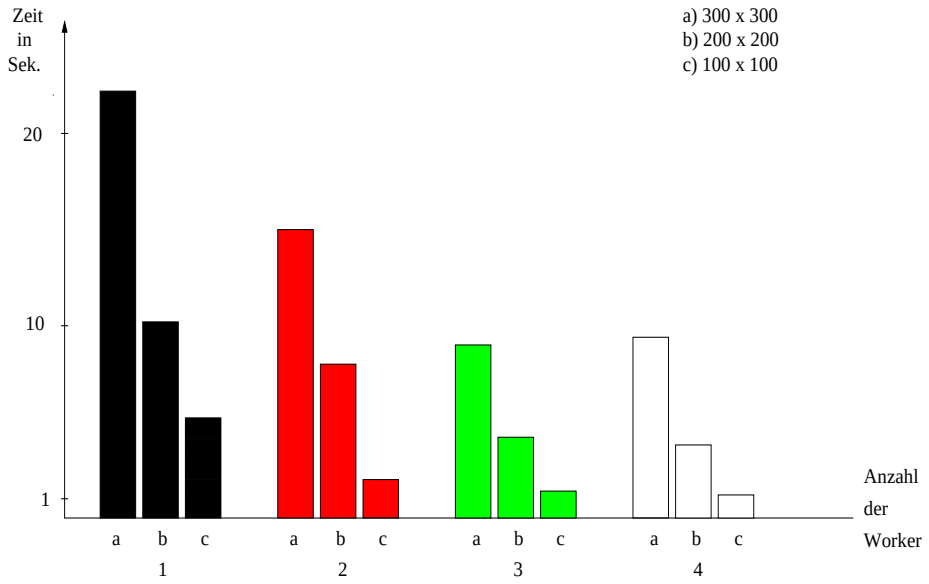

Bild 35: Ergebnisse der Berechnung des Mandelbrot-Fraktals

Bei einem Fenster von 300 x 300 Punkten kann man sehen, daß die Berechnung mit 3 Workern schneller ist als mit 4 Workern. Einer der Worker brauchte deutlich länger als die anderen drei. Wir haben es hier mit einer ungünstigen Verteilung der Arbeit zu tun.

3.2.3 Diskussion

Die zyklische Verteilung der Zeilen bringt schon recht gute Ergebnisse, aber auch hier wäre eine dynamische Lastverteilung die günstigere. Sicherlich wäre es nicht sinnvoll, daß ein Worker jede Zeile extra anfordern muß, da dadurch der Kommunikationsaufwand enorm wächst, aber für ein Paket von mehreren Zeilen wäre das schon denkbar.Die Kommunikation könnte man dadurch wieder verringern, daß dann auch immer nur ein fertiges Paket von Zeilen zum Master geschickt wird.

3.3 Verteiltes Gauß‘sches Eliminierungsverfahren

3.3.1 Implementierung

Bei diesem Algorithmus ist der Kommunikationsaufwand am Beginn der Berechnung zum ersten Mal etwas größer. Und zwar wird an alle Worker die vollständige Matrix geschickt. Jeder Worker nimmt sucht sich die entsprechenden Zeilen raus, führt die Eliminierung durch und wenn er mit allen Zeilen darüber eliminiert hat, führt er die Division durch, bevor er die Zeile verschickt. Die erste Zeile wird vom ersten Worker dividiert und gleich an alle anderen Worker verschickt. Erhält ein Worker eine Zeile, führt er die Elimination aller seiner Zeilen mit dieser Zeile durch. Falls eine Zeile schon so weit eliminiert wurde, daß sie dividiert werden kann, wird dies getan und die Zeile anschließend an alle verschickt. Sind alle Zeilen berechnet, wird dies dem Master mitgeteilt, und er terminiert.

3.3.2 Ergebnisse des Gauß‘schen Eliminierungsverfahrens

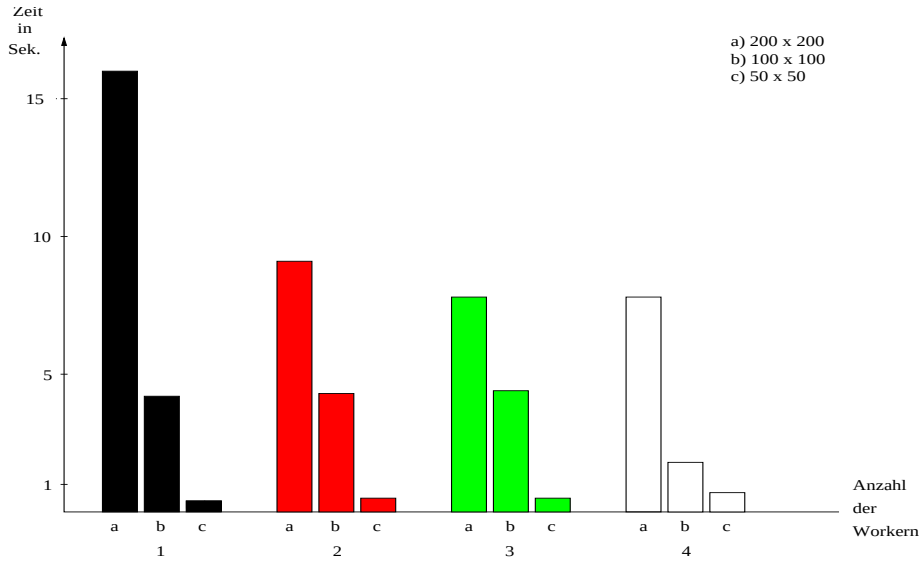

Bild 36: Ergebnisse des Gauß‘schen Eliminierungsverfahrens

In Bild 36 ist ein deutlicher Speedup zu erkennen. Am größten ist der Speedup, wenn die Matrix genügend groß, d.h. wenn die Berechnung im Vergleich zur Kommunikation hinreichend groß ist.

3.3.3 Diskussion

Bei diesem Algorithmus würde eine dynamische Lastverteilung relativ wenig bzw. gar keine Veränderung bewirken, da die Berechnung der Zeilen von einander abhängt. So kann z.B. die n+1 -te Zeile nicht vor der n-ten Zeile fertig sein. Die einzelnen Worker müssen sich synchronisieren, d.h. aufeinander warten. Der Kommunikationsaufwand ist bei diesem Algorithmus relativ hoch. Trotzdem sind bei großen Matrizen gute Ergebnisse zu erzielen.

4.0 Abschließende Bemerkungen

Der Rendezvous Software Bus bietet eine relativ einfach zu handhabende Umgebung für verteilte Applikationen, mit der auch gute Ergebnisse zu erzielen sind. Wie diese Ergebnisse im Einzelnen ausfallen, hängt von dem jeweiligen Algorithmus ab. Manche Algorithmen eignen sich besser für eine verteilte Ausführung als andere. Es gilt daher immer vorher abzuwägen, wie sich der Algorithmus verhält, d.h. wie groß der Kommunikationsaufwand im Vergleich zum Berechnungsaufwand ist. Auch kann man in jeder Umgebung einen Algorithmus so implementieren, daß der Gewinn durch die Parallelität bei der Berechnung durch unnötige Kommunikation wieder aufgehoben wird.

Die einfache Kommunikationsmöglichkeit mit Broadcast Messages verleitet dazu, diese allzu häufig einzusetzen. Hier gilt es besonders vorsichtig zu sein, da man mit Broadcast Messages sehr schnell das Netz überlasten kann.

Occam und Transputer

Daniel Schulz
dschulz@informatik.hu-berlin.de

1.0 Entwicklungsumgebung

Transputer-Cluster als MIMD-Rechner und die Hochsprache Occam bilden eine Einheit von Hard- und Software. Sie erleichtern die Implementation von verteilten Algorithmen, die sich über Nachrichten synchronisieren (message passing).

1.1 Transputer

- Transputer sind leistungsfähige Mikroprozessoren, die als Besonderheit vier bidirektionale Linkschnittstellen aufweisen, über die sie mit anderen Transputern kommunizieren.
- Ein Transputer-Cluster ist ein lose gekoppeltes Netzwerk, das über eine sogenannte Network Control Unit (NCU) konfiguriert werden kann. Es lassen sich Parallelrechner mit nahezu beliebiger Größe, Komplexität und Topologie aufbauen. Damit läßt sich in beispielhafter Weise eine Anpassung der Hardware an den gegebenen Algorithmus und die Problemstellung erreichen. Zum Beispiel könnte man für einen Suchalgorithmus eine Baumstruktur und für einen pipelineartigen Filteralgorithmus eine Ring-Topologie wählen.
- Kennwerte:
  - CPU: 10-30 MIPS (anno 1986)
  - Linkschnittstellen:
    - 20 Mbit/s Transferrate
    - arbeiten parallel zur CPU
    - DMA Zugriff auf Speicher
  - interner Bus:
    - verbindet CPU, Links und Speicher
    - 80-120 MByte/s Transferrate
  - mikrokodiertes Betriebssystem (Multiprozeß-Unterstützung)

1.2 Occam

Occam bietet dem Programmierer von verteilten Systemen folgende Möglichkeiten:

- Erzeugung von Parallelismus über das PAR-Konstrukt
- komfortable Untersützung von Synchronous Message Passing
- explizite Verteilung von Prozessen auf physische Prozessoren

Die kleinsten unteilbaren Vorgänge sind die drei atomaren Aktionen Zuweisung, Eingabe (Receive) und Ausgabe (Send). Die Datenübertragung erfolgt in Occam über unidirektionale, ungepufferte Verbindungen, sogenannte Kanäle. Bei der Benutzung von Kanälen in Occam gibt es für den Programmierer keinen Unterschied zwischen Intra- und Interprozessorkommunikation. Das Message-Passing Paradigma wird in Occam konsequent verfolgt, indem parallelen Prozessen (auch auf einem Prozessor) der Zugriff auf gemeinsamen Speicher verboten ist und dafür die Send/Receive-Operatoren `!` und `?` bereitgestellt werden. Der folgende Occam-Code zeigt ein kleines Beispiel für Prozeßkommunikation.

```
CHAN OF INTEGER K1,K2:      -- 2 Kanäle
INTEGER x1,x2:              -- Daten
PAR                         -- parallele Abarbeitung
   SEQ                  -- Prozess arbeitet sequentiell
         K1 ! 5         -- 5 wird gesendet
         K2 ? x1        -- K2 lesen und x1 neu schreiben mit 15
   SEQ                  -- Prozess 2
         K1 ? x2        -- zuerst empfangen (sonst Deadlock-Situation,
                                wenn beide Prozesse auf ! blockieren)
                                und x2 bekommt 5
         K2 ! 15        -- 15 senden
```

Bild 37: 2 parallele Prozesse kommunizieren über Kanäle K1, K2

Die Kommunikation über Kanäle kann sehr bequem programmiert werden. Man hat die Möglichkeit, Protokolle zu definieren, die mehrere Paketformate unterstützen. Die Formate sind mit Etiketten behaftet, anhand derer man die Daten identifizieren kann.

```
PROTOCOL Senden
   CASE
   -- Etikett;Adresse;Zeilennummer;
         Anzahl::vektor[0..Anzahl-1] Data; INT; INT;
         INT::[]REAL32
   -- Etikett;Zeilennummer;Anzahl::vektor[0..Anzahl-1]
         Elim.Data; INT; INT::[]REAL32
:

CHAN OF Senden Kanal1:
```

Bild 38: Definition eines varianten Protokolls für Matrixzeilen

Wird im gezeigten Beispiel etwas auf Kanal1 geschrieben, so kann der Empfänger anhand des Etiketts Data bzw. Elim.Data entscheiden, was er mit der Zeile macht.

2.0 Algorithmen

Um die Leistungsfähigkeit des Systems zu untersuchen, wurden drei unterschiedliche Probleme betrachtet:

- Gauß'scher Algorithmus
- Primzahlsuche
- Fraktalberechnung

Die drei Probleme wurden unterschiedlich gelöst, doch die Implementationen basieren auf folgenden Gemeinsamkeiten:

- Topologie des Netzes ist ein Ring
- gleiche Definition der Kanäle für Interprozeßkommunikation

Die Wahl einer Netztopologie für die Lösung verschiedener Probleme ist zwar eine Einschränkung, jedoch zum Vergleichen gut geeignet.

2.1 Gauß'sches Eliminationsverfahren

2.1.1 Datenverteilung

Da es sich bei unserem System um ein Netz gleichleistungsfähiger Prozessoren handelt, die zudem keine weiteren Aufgaben während der Abarbeitung unseres Programms haben, verteilen wir die Daten statisch nach folgendem Schema. Die Zeilen der Matrix werden zyklisch auf die Prozessoren veteilt, d.h. der i-te Prozessor bekommt alle Zeilen k mit (k MOD nr.of.processors+1)=i. Somit kann eine gleichbleibende Last aller Prozessoren gesichert werden.

2.1.2 Kommunikation

Während der Berechnung der oberen Dreiecksmatrix ist es erforderlich, im k-ten Iterationsschritt die Zeile durch das k-te Element zu dividieren und an die Besitzer der Zeilen k+1 bis n zu verschicken. Das heißt, ein Broadcast der k-ten Zeile ist erforderlich. Diesen Broadcast führen wir pipelined aus, d.h. der Prozessor mit Zeile k+1 empfängt die k-te Zeile, schickt sie sofort weiter und benutzt sie erst dann zur Subtraktion der Zeilen größer k. Das ist gerade in unserem System sehr effizient, da wir nur acht Prozessoren. Bei einer 80x80-Matrix z.B. erhält jeder Prozessor 10 Zeilen. Bekommt der dritte Prozessor die Zeile 1 vom zweiten Prozessor, so schickt er diese weiter und subtrahiert sie von den Zeilen 2,10,..,74.

2.1.3 Implementation

Der 0-te Prozessor im Ring ist der Initiator der Berechnung,kommuniziert mit dem Host, an dem der Benutzer die Eingaben macht und während der Berechnung verhält er sich wie die anseren Prozessoren.Die Berechnung geschieht in folgenden Phasen:

1. Verschicken der Daten: Der 0-te Prozessor schickt die Zeilen, die nicht ihm zugeteilt werden, an seinen Nachfolger. Jeder andere Prozessor i schickt die Zeilen weiter, die nicht ihm zugeteilt sind.
2. Berechnen der oberen Dreiecksmatrix: Der 0-te Prozessor initiiert die Berechnung, denn er hat die erste Zeile (Zeile 0). Danach verhält er sich wie jeder andere Knoten. Alle Knoten warten auf eine Zeile vom Vorgänger. Erhalten sie eine Zeile k, so gibt es zwei Fälle: 1. Zeile k+1 ist auf dem Knoten und 2.Fall Zeile k+1 nicht vorhanden. Im 1.Fall wird Zeile k von Zeile k+1 subtrahiert und danach werden beide Zeilen weiterge-

schickt, im 2.Fall wird nur Zeile k weitergeschickt. Danach werden die restlichen Zeilen auf dem Knoten subtrahiert. Der erste Knoten, der die letzte Zeile erhält, leitet die nächste Berechnungsphase ein.

3. Eliminationsschritt: Dieser Schritt läuft im wesentlichen wie der letzte Schritt ab, nur die Senderichtung der Zeilen ist entgegengesetzt. Das Ende des Eliminationsschrittes ist erreicht, wenn der 0-te Prozessor die erste Zeile zum eliminieren bekommt.
4. Einsammeln der Daten: Prozessor 0 sendet ein Signal an die Prozessoren zum Einsammeln der Daten. Die Daten werden nun zurückgeschickt. Ende.

2.1.4 Fazit

Die zeitraubendsten Phasen der Berechnung sind das Versenden sowie das Einsammeln der Daten im Ring. Es wäre daher angebracht, auf die Strukur des Ringes noch eine Baumstrukur zum Datentransport zu legen. Während der Berechnung ist die Ringstruktur vorteilhaft, da bei der zyklischen Datenverteilung der (i+1)-te Prozessor alle Nachfolgerzeilen des i-ten Prozessors hat.

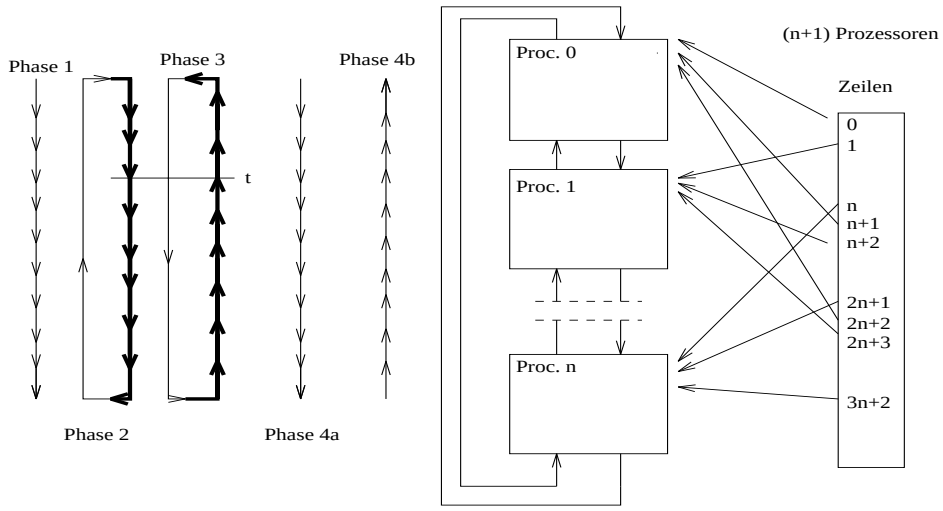

Bild 39: Kommunikationsstruktur der Occam-Programme

2.1.5 Messungen

Prozessoren	60x80 Matrix	100x150 Matrix
1	0,81sec	4,07sec
2	0,99 sec	4,50sec
3	0,93sec	4,44sec
4	0,56sec	2,41sec
5	0,47sec	1,96sec
6	0,40sec	1,54sec
7	0,38sec	1,37sec
8	0,36sec	1,24sec

2.2 Primzahlsuche

2.2.1 Datenverteilung

Da bei der Primzahlsuche der zu durchsuchende Bereich [a..b] nur von den Primzahlen bis $\sqrt{b}$ abhängt, ist bei großen Bereichen der Aufwand für die Suche der Siebzahlen gering. Wir haben den Gesamtbereich in nr.of.processors viele Teile geteilt, die alle von den ersten $\sqrt{b}$ Primzahlen abhängig sind.

2.2.2 Kommunikation

Kommunikation tritt nur bei der Berechnung der Siebzahlen und beim Aufteilen der Jobs und Einsammeln der Ergebnisse auf. Bei großen Bereichen ist der Grad der Parallelität sehr hoch, da während des Suchens nur lokale Berechnungen ausgeführt werden und keine Kommmunikation stattfindet.

2.2.3 Implementation

Wir haben einen Prozessor dafür eingesetzt, die Jobs zu verteilen und die Ergebnisse einzusammeln. Alle anderen Prozessoren laufen als Worker. In der Anfangsphase gibt der Master-Prozeß soviele Jobs raus, bis die Siebzahlen berechnet sind. Danach müssen die Worker sieben. Zum Schluß werden die Primzahlen an den Master zurückgesendet.

2.2.4 Fazit

Es gibt zwar viel Parallelität während des Siebens, dafür hat man aber am Schluß viel Kommunikation und der Master wird zum Flaschenhals.

2.2.5 Messung

Alle Zeiten in sec; Primzahlen suchen bis:

Prozessoren	1000	10000	100000	1000000	1500000
1 (1W)	0,06	0,27	3,79	68,22	-
2 (1M/1W)	0,13	0,31	3,54	64,58	-
3 (1M/2W)	0,20	0,32	1,96	32,90	55,65
4 (1M/3W)	0,28	0,39	1,50	22,43	37,81
5 (1M/4W)	0,36	0,47	1,31	17,23	28,91
6 (1M/5W)	0,44	0,56	1,25	14,13	23,57
7 (1M/6W)	0,52	0,66	1,24	12,09	20,06
8 (1M/7W)	0,60	0,76	1,26	10,65	17,55

2.3 Fraktalberechnung

2.3.1 Datenverteilung

Da wir einen Bereich des R^2 auswerten, in dem die Elemente der Mandelbrotmenge im Zentrum liegen, müssen die zu berechnenden Zeilen zyklisch auf die Prozessoren verteilt werden um eine überlastung der mittleren Prozessoren zu vermeiden.

2.3.2 Kommunikation

Da man die fortschreitende Berechnung des Fraktals am Bildschirm mitverfolgen möchte, muß ein Senden der Zeile nach jeder Berechnung möglich sein.

2.3.3 Implementation

Es gibt einen Master-Prozessor, der mit dem Host (Workstation oder PC) kommuniziert und die Aufträge verteilt. Um den Kommunikationsanforderungen zu genügen, laufen auf jedem Prozessor zwei paralele Prozesse. Ein Prozeß empfängt den Job vom Master un berechnet die Punkte. Der andere Prozeß empfängt vom Nachfolgerknoten oder vom lokalen Rechnerprozeß bereits berechnete Daten und schickt diese zum Vorgänger weiter. Der Master-Knoten am Anfang erhält alle Daten und gibt sie aus.

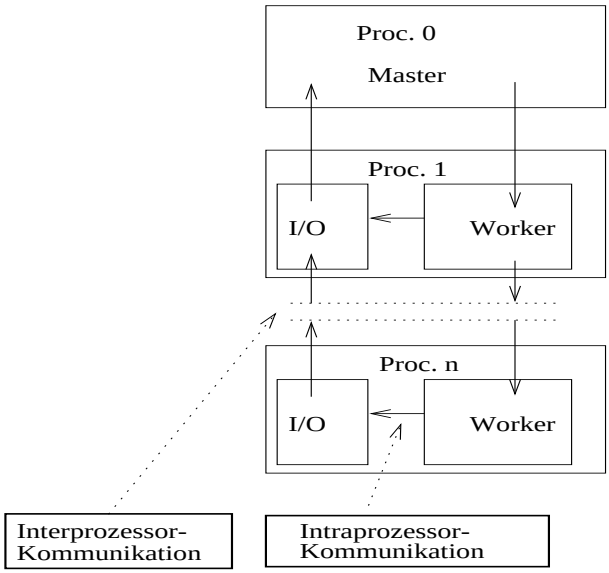

Bild 40: Kommunikation im Fraktal-Programm

2.3.4 Fazit

Der Master macht die gesamten I/0-Operationen mit den externen Geräten wie Festplatte, Grafikinterface usw., er ist wieder der Flaschenhals im System. Bei Festplattenzugriffen war zu beobachten, daß diese das System synchronisierten, weil sie zu lange dauerten.

2.3.5 Messungen

Prozessoren	**320x240**
1(Single)	10,80
2(1M/1W)	16,78
3(1M/2W)	8,72
4(1M/3W)	6,31
5(1M/4W)	5,02
6(1M/5W)	5,14
7(1M/6W)	4,96
8(1M/7W)	4,8

Alle Zeiten in sec.

Literatur

F. Bause, W. Toelle; *Einführung in die Programmiersprache C++*; Vieweg '89, Braunschweig.

H. Ebert.*; Transputer und Occam*; Heise-Verlag, Hannover, 1993.

A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam*; PVM 3 USER'S GUIDE AND REFERENCE MANUAL*; Oak Ridge National Laboratory, September 1994.

NeXT Computer , Inc.; *NeXTSTEP Operating System Software*; Addison-Wesley, September 1992.

A. Polze; *Verteiltes Programmieren unter Mach - Implementation eines rexec-Server*; unix/mail 3/1995, Carl Hanser Verlag München

A. Polze, M. Malek; *The Shared Objects Net-interconnected Computer (SONiC)*; HUB Informatik Berichte No. 52/95, 18 pages, ISSN 0863-95 52, Berlin, Dezember 95

A. Polze; *Shared Objects Memory - Kommunikation unter Mach*; unix/mail 2/1995, Carl Hanser Verlag München

V. Sunderam, A. Geist, J. Dongarra, R. Manchek; *Parallel Computing: The PVM concurrent computing system: Evolution, experiences, and trends*; Oak Ridge National Laboratory, Special issue 1994, S. 531-545.

Teknekron Software Systems, Inc.; *Rendezvous Software Bus Programmer's Guide*; Teknekron, Palo Alto, 1995.

C. Zimmermann, A.W.Kraas; *Mach-Konzepte und Programmierung*; Springer '93, Berlin