Hochgradiger Parallelismus

Walter F. Tichy und Michael Philippsen
Fakultät für Informatik, Universität Karlsruhe

Interner Bericht 14/91

Zusammenfassung

Hochgradig parallele Rechner mit tausenden von Recheneinheiten sind seit mehreren Jahren kommerziell erhältlich und werden bereits in vielfältigen Anwendungen eingesetzt. Gleichzeitig hat ein Strukturwandel in der Informatik begonnen, die sich nun stärker als bisher den Prinzipien der Parallelverarbeitung zuwenden wird.

Dieser Artikel gibt einen Überblick über das sich rasch entfaltende Gebiet der parallelen Informatik. Anhand einer Klassifikation von Parallelrechnern arbeiten wir wichtige Architekturmerkmale heraus. Der Stand der Technik wird mittels vier kommerziell erhältlicher Maschinen dargestellt, für die wir erreichbare Rechen- und Kommunikationsleistungen sowie Preis-Leistungsverhältnisse bestimmen. Dabei fällt auf, daß das Verhältnis zwischen Kommunikations- und Rechenleistung weiterhin ungünstig ist und für MIMD-Maschinen feingranulare, parallele Programme weitgehend verbietet.

Schließlich geben wir eine Übersicht über die Forschungsthemen, die sich in der parallelen Informatik eröffnen.

Summary

Highly parallel computers with thousands of processing elements have been commercially available for several years and are now being used in numerous applications. At the same time, Computer Science is shifting its attention towards studying the priniciples of parallel computation more intensively than before.

This article surveys the quickly developing field of parallelism. A classification of the main architectural principles of parallel computers is given. The state-of-the-art is analyzed by comparing arithmetic performance, communications performance, and price/performance ratios of four commercially available, parallel computers. We note that arithmetic and communications performance continue to be unbalanced, preventing MIMD machines from being used effectively in most fine-grained parallel applications.

A catalogue of research issues in the unfolding area of parallel information processing is given.

1 Einleitung

Seit dem ENIAC von 1946 war der sequentielle Rechner das dominierende Paradigma der Informatik. Trotz einiger Ansätze im Bereich der Parallelrechner in den späten 60'er und frühen 70'er Jahren (z.B. ILLIAC, C.mmp), und trotz vielfältiger Bemühungen, Netzwerke von Rechnern als parallele Rechenwerke einzusetzen, gelang der Durchbruch zum Hochparallelrechner mit zehntausenden von Recheneinheiten erst in der Mitte der letzten Dekade. Die Maschine, die als erste den Durchbruch schaffte, war Daniel Hillis' Connection Machine (CM)[20, 21]; sie zeigte, daß ein Rechner mit zehntausenden von Recheneinheiten nicht nur hardwaremässig,

sondern auch softwaremässig beherrschbar und erfolgreich einsetzbar ist. In rascher Folge kamen danach drei weitere Hochparallelrechner mit tausend und mehr Prozessoren auf den Markt; weitere Hersteller drängen auf den Markt.

Ähnlich wie sich die sequentielle Informatik mit der Verfügbarkeit von sequentiellen Rechnern entwicklte, so beginnt sich nun die parallele Informatik zu entfalten. Hochgradig parallele Systeme rücken in fast allen Bereichen der Informatik in den Mittelpunkt des Interesses. Noch ist der Hochparallelrechner mehr Forschungsgegenstand als Werkzeug. Wenn jedoch die Leistungssteigerungen dieser Rechner anhalten, die Preise weiter fallen, und die Programmierung dieser Maschinen stärker vereinfacht wird (und alles deutet darauf hin, daß diese drei Unterfangen glücken werden), dann wird der hochgradig parallele Rechner in breiter Front in die Nutzung übergehen. Damit wäre dann der bis dato bedeutendste Paradigmenwechsel in der Informatik vollzogen. Der heute noch als zentral angesehene sequentielle Rechner würde dann zu Recht als Spezialfall betrachtet. Im Vergleich dazu würde sich die inzwischen abgeschlossene sog. „Mikroprozessor-Revolution“ als zwar notwendiges, aber weniger spektakuläres Vorspiel entpuppen.

Die Triebkraft hinter dieser Entwicklung sind die enormen Leistungssteigerungen, die mit einem Parallelrechner möglich sind. Durch Vervielfachung und Kopplung von Recheneinheiten kann im Prinzip eine fast beliebige Leistungssteigerung erzielt werden. Gleichzeitig schreitet die Entwicklung in der Mikroelektronik, zumindest mittelfristig, fort, und die daraus entstehenden Leistungsteigerungen können mitvervielfacht werden. Beide dieser Effekte sind im Nachfolger der Connection Machine schon abzusehen: Durch bessere Technologie wird der Nachfolger 1991 etwa 25 mal mehr leisten (100 GFLOPS, Giga Floating Point Operations Per Second) als das Modell von 1987 (4 GFLOPS); in weiteren zwei Jahren hofft man, durch Vervielfachung die erreichbare Leistung um eine weitere Zehnerpotenz auf TFLOPS springen zu lassen. Dieser TFLOPS-Rechner wird ein wahrer Gigant von über $300m^2$ Stellfläche sein. Weitere Leistungssteigerungen dürften dann eher an den Kosten als am Grundprinzip scheitern. Doch auch die Kosten werden sinken.

Angesichts dieser gewaltigen Rechenleistung stellt man sich die Frage, ob derartiges überhaupt nötig ist. Die Antwort hierauf ist einfach: sicher nicht für Anwendungen, die heute auf existierenden Anlagen zufriedenstellend laufen; auch nicht für solche, für die die Fortentwicklung der Mikroelektronik ausreichend Verbesserungen bringt. Es gibt jedoch eine Reihe von Anwendungen, die mit diesen Superrechnern erst in Angriff genommen werden können. Allen voran sind dabei numerische Anwendungen in den natur- und ingenieurwissenschaftlichen Disziplinen. Zum Beispiel schätzen Luftfahrttechniker, daß eine vollständige numerische Simulation eines Flugzeuges in wenigen Stunden durchgeführt werden könnte, wenn ein Rechner mit einer Dauerleistung von 1 TFLOPS zur Verfügung stünde. Wissenschaftler auf den Gebieten Hochenergiephysik, Kosmologie, Erdbebenvorhersage, Materialforschung, Erdölexploration, Schaltungsentwurf, Maschinensehen, Atmosphärenforschung, Ozeanographie und anderen Disziplinen hoffen ebenfalls, mit TFLOPS-Rechnern Durchbrüche erzielen zu können [10].

Wie aber steht es um Anwendungen in nicht-wissenschaflichen Bereichen? Hier zielen die Voraussagen auf Maschinen mit einem wesentlich höherem Automatisierungsgrad und Bedienungskomfort. Möglich sind autonome Roboter, die gefährliche oder mühevolle Tätigkeiten selbständig ausführen, Maschinen, die sehen und hören, Telephone, die in andere Sprachen übersetzen, führerlose Fahrzeuge, die ihre Insassen mit hoher Geschwindigkeit sicher transportieren, vollautomatische Fabriken, die Gegenstände für Kunden nach indivduellen Wünschen produzieren, also sozusagen „nach Maß“ anfertigen, u.ä. In der Tat sind diese Anwendungen an manchen Stellen bereits in der Entwicklung, aber durch mangelnde Rechenleistung behindert. Die großen Überraschungen werden allerdings eher diejenigen Anwendungen sein, die wir heute nicht voraussehen. Wer hätte sich 1975 träumen lassen, an welch vielfältigen Stellen der Mikroprozessor Einzug halten würde?

Die schnellsten Arbeitsplatzrechner von heute leisten etwa 10 MFLOPS. Bell [5] schätzt, daß bis 1995 preisgünstige Ein-Chip-Prozessoren mit Geschwindigkeiten um die 20 MFLOPS weite Verbreitung finden werden. Es ist daher klar, daß TFLOPS-Maschinen Multirechner sein müssen, die aus einer großen Anzahl Prozessorelementen (Prozessor plus Speicher) bestehen und durch ein Hochgeschwindigkeitsnetz verbunden sind. Allerdings wird die Spitzenleistung, zu der diese Maschinen fähig sind, in der Praxis schwierig zu erreichen sein. Von den verhältnismässig einfach zu programmierenden Vektorrechnern weiß man, daß Programme sorgfältig auf die gegebene Maschinenarchitekur abgestimmt werden müssen, um wenigstens 70 bis 90% der Spitzenleistung zu erzielen; andernfalls erreicht man typischerweise nur 10 bis 20% oder noch weniger [11]. Für Parallelrechner heutiger Bauart kann der Ausnutzungsgrad bei mangelnder Sorgfalt in der Programmierung noch wesentlich schlechter sein. Es besteht daher großer Bedarf, Methoden und Werkzeuge für die systematische Programmierung von Parallelrechnern zu entwickeln, sowie Parallelrechnerarchitekturen so zu gestalten, daß die vorhandene

Leistung von Programmen auch genutzt werden kann.

Die zentrale Frage zu Beginn der 80'er Jahre war die Frage nach der Praktikabilität von Parallelrechnern. Heute ist diese Frage weitgehend positiv beantwortet: Eine Reihe von Rechnertypen sind auf dem Markt erhältlich, die tausende von gleichzeitig operierenden Prozessoren umfassen und die für erstaunlich vielfältige Zwecke eingesetzt werden. Wir müssen uns nun schwierigeren Fragen zuwenden. Welche Architekturen von Parallelrechnern sind am besten für gegebene Problemklassen geeignet? Wie kann ein vorliegendes Problem in tausende von Arbeitseinheiten zerlegt werden, die getrennt auf unterschiedlichen Prozessoren ablaufen können? Wie müssen parallele Algorithmen entworfen werden, damit sowohl die bei der Kommunikation auftretenden Verzögerungen klein bleiben als auch eine möglichst volle Auslastung aller Prozessoren gewährleistet wird? Welche Eigenschaften müssen skalierbare parallele Algorithmen haben, um sich automatisch, etwa nach Änderung weniger Parameter, an die vorhandene Anzahl von Prozessoren anzupassen? Wie kann die Korrektheit eines parallelen Programms nachgewiesen und wie können Fehler in parallelen Programmen gefunden werden, insbesondere da Fehler oft auf Indeterminismen beruhen und daher kaum reproduzierbar sind?

Die grundsätzlichen Fragen betreffen allesamt Kerngebiete der Informatik, nämlich die Bereiche der Rechnerarchitektur, der Algorithmen, der Programmiersprachen, der Übersetzer, der Betriebssysteme, der Leistungsbewertung und des Software Engineerings. Auch wollen Informatikgebiete wie Bildverarbeitung, Spracherkennung, neuronale Netze, Robotik, Datenbanken und Kryptographie von der enormen Leistung der Parallelrechner profitieren. Viele dieser Bereiche sind gegenwärtig von einer „Parallelisierungswelle“ ergriffen. Schließlich ist auch noch die überwältigende Vielfalt der Informatikanwendungen in Hinblick auf Parallelität aufzuarbeiten.

Der hochgradige Parallelismus erweist sich damit nicht als eine neue Teildisziplin der Informatik, sondern betrifft fast alle ihre Gebiete. Parallelrechner erfordern von uns ein Überdenken aller gewohnten Ansätze. Wie jede neue Technologie, die eine Änderung in der Organisation und Aufteilung der Arbeit mit sich bringt, sind weitreichende Wirkungen zu erwarten.

In diesem Artikel geben wir einen Überblick über das sich rasch entwickelnde Gebiet des hochgradigen Parallelismus. Dazu klassifizieren wir im nächsten Abschnitt die diversen Parallelrechnerarchitekturen. Sodann stellen wir eine Übersicht über die Leistungsfähigkeit und die Kosten einiger der am Markt verfügbaren Maschinen zusammen. Im letzten Abschnitt sammeln wir Forschungsthemen, die in der parallelen Informatik nun aktuell werden.

2 Multiprozessorarchitekturen

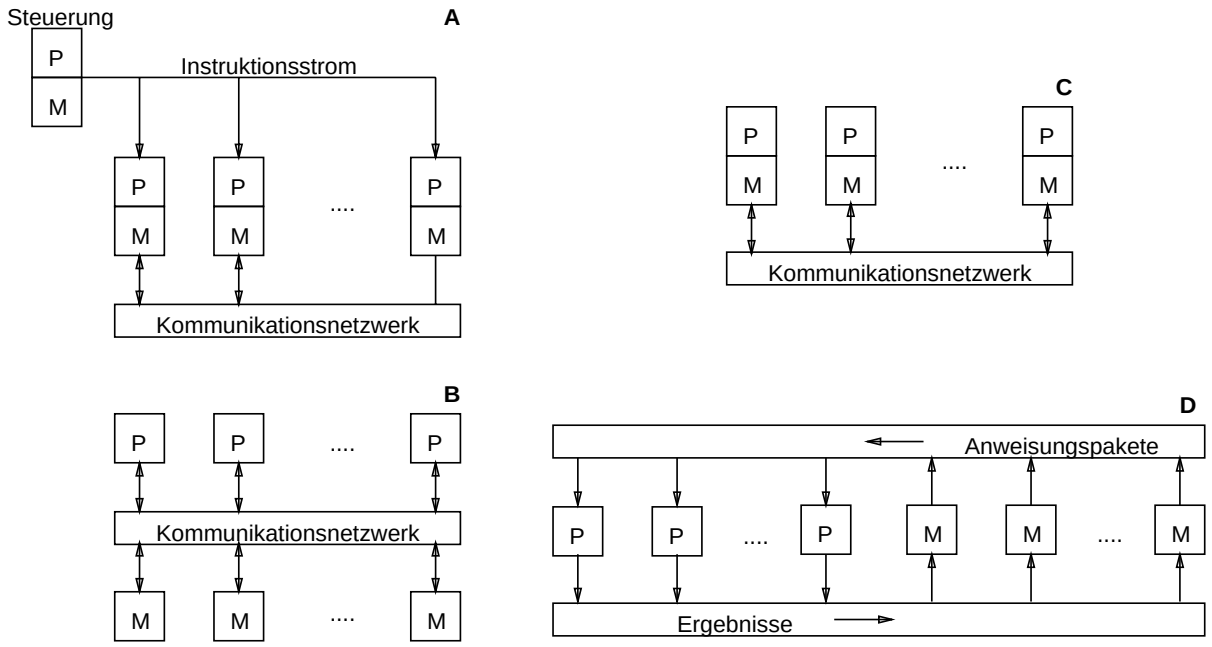
Die drei wichtigsten Dimensionen zur Klassifizierung von Parallelrechnern sind:

1. **Zentraler versus verteilter Arbeitsspeicher:** Ein zentraler Arbeitsspeicher besteht aus einer Einheit, die alle Prozessoren gleich schnell bedient; ein verteilter Speicher hingegen besteht aus mehreren Einheiten, die jeweils einem oder einer Gruppe von Prozessoren zugeordnet sind. Bei verteiltem Speicher kann ein Prozessor normalerweise deutlich schneller auf die ihm zugeordnete Einheit zugreifen als auf andere.

2. **Feinkörnige versus grobkörnige Granularität:** Die Granularität bestimmt den Grad der Aufgliederung einer Gesamtaufgabe in Teilaufgaben, die die einzelnen Prozessoren parallel lösen. Bei niedriger (also grobkörniger) Granularität enthält jede Teilaufgabe viele Datenelemente, bei hoher (oder feinkörniger) Granularität nur eines oder nur einige wenige. Um für feinkörnige Granularität geeignet zu sein, muß ein Rechner entsprechend viele Prozessoren aufweisen und schnelle, parallele Kommunikation zwischen Prozessoren bieten.

3. **SIMD versus MIMD:** In einem Rechner mit Betriebsweise SIMD („Single Instruction stream, Multiple Data stream“) sendet ein einzelner Steuerungsprozessor Befehle nacheinander an alle anderen Prozessoren, welche diese an eigenen Daten ausführen. In einem Rechner mit Steuerungsmodus MIMD („Multiple Instruction stream, Multiple Data stream“) führt jeder Prozessor sein eigenes Programm auf eigenen Daten aus. Beide Bezeichner sind Teil einer von Flynn bereits 1972 vorgelegten Klassifikation [12].

In den nun folgenden Abschnitten kommentieren wir diese Unterscheidungsmerkmale. Abbildung 1 stellt vier praktikable Parallelrechnerarchitekturen vor, die die Unterschiede verdeutlichen.



Die ersten drei Architekturen werden praktisch eingesetzt, während zur Form (**D**) nur Prototypen existieren. (**A**) Eine SIMD-Maschine mit verteiltem Speicher. Ein Steuerungsprozessor sendet eine Folge von Instruktionen an alle Datenprozessoren. Jede dieser Instruktionen löst entweder eine Operation auf allen Datenprozessoren aus oder stößt einen Kommunikationsvorgang im Netzwerk an. (**B**) Eine MIMD-Architektur mit zentralem Speicher. Jeder der Prozessoren führt eigenständig ein eigenes Programm aus und operiert dabei auf Daten, die in Speichereinheiten abgelegt sind, auf die alle Prozessoren gleichberechtigt Zugriff haben. Alle Speicherzugriffe laufen über ein Hochgeschwindigkeitsnetz, das als Teil des zentralen Speichers zu verstehen ist. (**C**) Eine MIMD-Maschine mit verteiltem Speicher. Die Prozessoren operieren auf lokal gespeicherten Daten und tauschen Botschaften über ein Netzwerk aus. (**D**) Ein MIMD-Datenflußrechner mit zentralem Speicher. Pakete von Anweisungen, die sich im Speicher befinden, fließen zu den Prozessoren, sobald ihre Operanden vorhanden sind; nach der Bearbeitung fließen die erzeugten Ergebnisse zurück in den Speicher und lösen dort die Aussendung neuer Instruktionspakete aus. (P bedeutet Prozessor, M Speicher).

Abbildung 1: Vier Parallelrechnerarchitekturen.

2.1 Kommunikationsnetzwerk

Eine wichtige Komponente eines Parallelrechners ist das Kommunikationsnetzwerk. Es ermöglicht den Prozessoren, Lese-, Schreib- und Sperranforderungen an Speicherzellen zu stellen oder Nachrichten mit anderen Prozessoren auszutauschen. Ein Kommunikationsnetzwerk für hochgradig parallele Maschinen mit tausenden von Prozessoren sollte folgende vier Eigenschaften erfüllen:

1. **Volle Konnektivität.** Jeder Prozessor muß in der Lage sein, jedem anderen Prozessor eine Nachricht zu schicken. (Möglich, wenn auch in der Praxis noch nicht ausreichend erprobt, ist eine Konnektivität durch virtuelle Adressierung von Speicherzellen in einem Rechnernetz.)

2. **Paralleler Nachrichtenaustausch.** Das Netzwerk muß in der Lage sein, gleichzeitig Anforderungen von allen Prozessoren zu bearbeiten und dabei sicherstellen, daß an Knotenpunkten nur minimale Verzögerungen auftreten. Ausgezeichnete Punkte im Netzwerk, wie etwa die Wurzel eines Baumes oder das Zentrum eines Sterns, sollten im Hinblick auf potentielle Engpässe vermieden oder mit entsprechender Kapazität ausgestattet werden.

3. **Kurze Kommunikationswege.** Sowohl die Anzahl der Vermittlungspunkte, über die eine Botschaft von Sender zu Empfänger geschickt werden muß, als auch die Leitungslängen zwischen den einzelnen

Vermittlungspunkten, sollten klein sein. Das Produkt dieser Maßzahlen sollte von kleinerer Ordnung als die Anzahl der Prozessoren sein.

4. **Skalierbarkeit.** Die Anzahl der Leitungen und Vermittlungspunkte im gesamten Netzwerk sollte von kleinerer Ordnung als das Quadrat der Prozessoranzahl sein.

Einige der heute verwendeten Kommunikationsnetzwerke erfüllen diese Bedingungen [38]. Eines der einfachsten ist der Hyperkubus [46], der jeden der $N = 2^k$ Prozessoren mit k anderen verbindet und dabei eine Maximalanzahl von Vermittlungspunkten pro Botschaft von k, eine maximale Leitungslänge zwischen benachbarten Prozessoren von k und einen Leitungsaufwand proportional zu $N \log N$ aufweist. Der Hyperkubus wird z.B. von der Connection Machine CM-2 [20] und dem nCUBE-2 [28] verwendet.

Manche Multiprozessoren benutzen noch Kommunikationsnetzwerke, die eine oder mehrere der obigen Eigenschaften nicht erfüllen, aber dennoch effektiv sind. Grund dafür ist stets eine geringe Anzahl an Prozessoren. Ein solches Netzwerk ist der Bus, der zu jedem Zeitpunkt nur von einem einzigen Prozessor belegt werden darf. Dieses Netzwerk erfüllt Eigenschaft (2) nicht. Steigt die Prozessoranzahl bis in die Größenordnung von 100, so wird das Kollisionsproblem so schwerwiegend oder das Busvergabeprotokoll so komplex, daß die Kommunikationsleistung des Busses die Gesamtleistung der Maschine begrenzt. Rechner wie der Sequent Symmetry und der Encore Multimax benutzen eine solche Busarchitektur.

Ein anderes, oft benutztes Kommunikationsnetzwerk ist der Kreuzschienenverteiler. Dieses Netz wird häufig bei Maschinen mit zentralem Speicher eingesetzt, um jeden Prozessor mit einem beliebigen Speicherbaustein verbinden zu können. Dieses Netz verstößt gegen Eigenschaft (4), da es mit seinen N^2 Vermittlungspunkten schlecht skaliert und bei einer Größenordnung von 100 Prozessoren ebenfalls seine Anwendbarkeitsgrenze findet. Die Cray X-MP und Cray Y-MP verwenden diese Architektur [24]. Bei diesen Rechnern ist die Kreuzschienen-Logik in die einzelnen Speichereinheiten integriert.

Kommunikationsnetze sind ein zentrales Thema für die Architektur von Parallelrechnern (s.a. Abschnitt 4.1). Einen guten Überblick über verschiedene Netztopologien, deren Eigenschaften und Einsetzbarkeit in Parallelrechnern geben [38, 34, 16].

2.2 Zentraler versus verteilter Speicher

Die Architekur des zentralen Speichers wurde von Burroughs bei der B5000 Maschine in den späten 50'er Jahren eingeführt und wird noch heute in Maschinen wie dem Sequent Symmetry Computer und dem Encore Multimax verwendet. Diese Technologie gibt jedem Prozessor gleichberechtigten Zugriff zu allen Speicherzellen über ein Kommunikationsnetz. Botschaften beliebiger Länge können in konstanter Zeit lediglich durch die Übermittlung von Adresse und Länge der Botschaft ausgetauscht werden. Ein verteilter Speicher hingegen gibt jedem Prozessor direkten Zugriff nur auf eine Speichereinheit; Zugriff auf andere erfordert das Senden von Nachrichten durch ein Netz und ist im allgemeinen langsamer als der direkte Zugriff. Ein verteilter Speicher ist angebracht, falls lokale Zugriffe dominieren.

Algorithmen, die eine gemeinsame und gleichmäßige Nutzung des gesamten Speichers durch alle Prozessoren bewirken, haben eine fundamentale Beschränkung. Da ein einzelnes Speichermodul zu jedem Zeitpunkt nur durch einen einzigen Prozessor benutzt werden kann, müssen alle anderen Prozessoren, die ebenfalls auf dieses Speichermodul zugreifen wollen, für die Dauer eines Schreib-Lese-Zyklus' warten. Für N Prozessoren und M Speichermodule geben Baskett and Smith [4] den Anteil der während jedes Speicherzyklus' leerlaufenden Speichereinheiten mit guter Näherung an. Daraus gewinnt man den Anteil B der im Zugriff auf den Speicher blockierten Prozessoren:

$$B(M,N) = \sqrt{1 + \left(\frac{M}{N}\right)^2} - \frac{M}{N}$$

Demnach sind bei $N = M$ etwa 40% der Prozessoren blockiert. Abhilfe bringt hier nur eine Erhöhung der Anzahl an Speichereinheiten oder, was auf das gleiche hinausläuft, das Erhöhen der parallel operierenden Eingänge pro Speichereinheit, wie für Monarch [35], einem Rechner mit 65.536 Prozessoren, vorgeschlagen.

Kritisch sind die Speicherzugriffsmuster der Programme. Da die meisten parallelen Algorithmen Zugriffe mit starker Lokalität aufweisen und diese von einem schnellen, lokalen Speicher abgewickelt werden können, werden Entwickler hochgradig paralleler Rechner weiterhin Architekturen mit verteiltem Speicher favorisieren.

2.3 Granularität der Parallelität

Die *Beschleunigung* (Speed-Up) ist die übliche Größe zur Bewertung der Leistungssteigerung durch Parallelrechner. Sie ist definiert als das Verhältnis zwischen der Zeit, die erforderlich ist, um ein Problem auf einem einzelnen Prozessor durchzurechnen, zu der Zeit, die die Lösung desselben Problems auf N Prozessoren erfordert [25]. Eine optimale Beschleunigung, ein Faktor von N, kann nur auf eine der beiden folgenden Arten erreicht werden: Maschinen, bei denen Arbeitseinheiten statisch einzelnen Prozessoren zugeordnet sind, erfordern zur Erreichung einer optimalen Beschleunigung ungefähr gleichgroße Arbeitseinheiten sowie verzögerungsfreien Nachrichtenaustausch zwischen den Prozessoren. Bei Maschinen mit dynamischer Zuteilung von Arbeitseinheiten reicht es in der Regel, dafür zu sorgen, daß zu jeder Zeit zumindest N solche Arbeitseinheiten ausführbereit sind.[1]

Mit Granularität bezeichnet man die Anzahl der Teilaufgaben in einem parallelen Programm, und damit die Feinheit der Arbeitsaufteilung. Je feiner die Granularität, desto mehr Prozessoren können zur Lösung eingesetzt werden, und um so größer ist die potentielle Beschleunigung. Demgegenüber treten aber Verluste in Form von Synchronisationskosten und, bei Maschinen mit verteilem Speicher, in Form von Kommunikationskosten auf. Auf einer sequentiellen Maschine gibt es naturgemäß keine Synchronisationskosten und auch keine Kommunikationskosten, da alle Daten im lokalen Speicher liegen. Das Zeitverhältnis zwischen nichtlokalem und lokalem Speicherzugriff kann bis zu drei Größenordnungen umfassen, weshalb eine sorgfältige Abstimmung der Granularität gegenüber den Kommunikationskosten erforderlich ist.

Eine gegebene Parallelrechnerarchitektur legt das Verhältnis q zwischen den Zeiten einer Nachrichtenübermittlung und einer Instruktionsausführung fest. Dieses Verhältnis definiert die untere Grenze für die Granularität: Zur Berechnung eines Zwischenergebnisses auf einem anderen Prozessor sollten mindestens q Instruktionen durchgeführt werden; andernfalls dominieren die Kommunikationskosten um das Zwischenergebnis zu transferieren und es ist günstiger, diese Ergebnis lokal zu berechnen. Falls q hoch ist, wird der Algorithmenentwickler daher ein Problem in große Teilaufgaben zerlegen. Nur bei kleinem q kann sich der Entwickler kleine Teilaufgaben leisten und ihre Anzahl proportional zur Problemgröße wählen. Die Größe von q ist auch der Grund warum Programmierer eine größere Maschine besser für ein größeres Problem bei gleicher Granularität einsetzen, als das gleiche Problem mit feinerer Granularität zu berechnen [17]. Der Beitrag [7] zeigt exemplarisch, wie bei parallelen Sortieralgorithmen die optimale Granularität zum Teil analytisch und zum Teil experimentell bestimmt werden kann.

Feine Granularität wird unterstützt durch Maschinen wie die Connection Machine [20], der MasPar Maschine [6, 29] und dem Massively Parallel Processor (MPP) [24]. Diese Maschinen sind in der Lage, einzelne Datenelemente zwischen benachbarten Prozessoren mit einem Zeitaufwand zu verschicken, der in der Größenordnung des Zeitbedarfs einer „gewöhnlichen“ Operation liegt. Durch möglichst ausschließliche Anwendung von Nachbarschaftskommunikation kann eine Prozessorauslastung von zumindest 0,5 erreicht werden, was durch den hochgradigen Parallelismus in der Summe zu einer beachtlichen Beschleunigung führt.

Schließlich sei darauf hingewiesen, daß sich Algorithmen durch Erhöhung der Granularität keineswegs beliebig beschleunigen lassen, selbst wenn $q = 0$ ist. Für jeden Algorithmus gibt es eine untere Schranke für die Laufzeit, die auch durch beliebig viele Prozessoren nicht mehr verbessert werden kann. Zum Beispiel zeigt ein einfaches „Fan-in-“Argument, daß die Summierung von n Zahlen eine Laufzeit proportional zu $\log n$ erfordert, und auch mit beliebig vielen Prozessoren nicht weiter beschleunigt werden kann. Ein Schatz paralleler Algorithmen und ihrer Analyse findet sich in [1, 15]. Zur Problematik der Beschleunigung sei auch noch auf die Artikel von Amdahl [2] und Gustafson [17] verwiesen.

[1] Die sogenannte „superlineare“ Beschleunigung, bei der die Berechnung eines Problems mit N Prozessoren weniger als $1/N$-tel der Zeit für eine Lösung mit nur einem Prozessor benötigt, ist ein Mythos. Wenn es superlineare Beschleunigung gäbe, dann könnte man das entsprechende parallele Programm mit nur einem Prozessor simulieren und hätte damit ein schnelles, sequentielles Programm, gegenüber dem das parallele eine nur lineare Beschleunigung aufwiese. Superlineare Beschleunigung ist nur denkbar, wenn man von einem suboptimalen sequentiellen Algorithmus ausgeht.

2.4 Steuerungsmodus SIMD versus MIMD

Ein fundamentales Problem beim Entwurf paralleler Algorithmen ist es, sicherzustellen, daß zu dem Zeitpunkt, in dem ein Prozessor eine Operation ausführen soll, alle dafür erforderlichen Operanden vorliegen. Ohne diese Garantie bleiben die Resultate indeterministisch, das heißt abhängig von der relativen Geschwindigkeit der verschiedenen Prozessoren. Um Laufzeitabhängigkeiten (race conditions) und das damit verbundene indeterministische Verhalten zu vermeiden, müssen Synchronisierungspunkte in die Programme eingeflochten werden.

In klassischen Einprozessorsystemen ist die Synchronisierung kein Problem, da die einzelnen Instruktionen sequentiell hintereinander ausgeführt werden. Die Ergebnisse der Instruktionen werden in Registern oder im Speicher für die spätere Benutzung abgelegt. Optimierende Übersetzer können die Auswertungsreihenfolge von Instruktionen vertauschen, falls keine bestehenden Datenabhängigkeiten dies verbieten.

Eine direkte Erweiterung dieser Arbeitsweise für Parallelrechner ist der SIMD-Rechner. Die Instruktionen werden der Reihe nach von allen oder einem Teil der Prozessoren simultan ausgeführt. Zum Beispiel sei ein planares Gitter betrachtet. Der einem Punkt zugeordnete Wert ergebe sich als Mittelwert der Werte der vier umliegenden Punkte. In der Schreibweise einer Programmiersprache muß bei Punkt (i, j) folgendes ausgeführt werden:

$$v(i,j) := [v(i-1,j) + v(i+1,j) + v(i,j-1) + v(i,j+1)]/4 \tag{1}$$

Auf einer SIMD-Maschine kann jedem Prozessor ein Punkt dieses Gitters zugeordnet werden; jeder Prozessor verfügt also in seinem lokalen Speicher über den Wert $v(i,j)$. Der Steuerungsprozessor versendet die Anweisungsfolge, die den Ausdruck (1) implementiert, an alle Datenprozessoren. Diese führen die Anweisungen unter Benutzung ihres lokal vorhandenen Werts $v(i,j)$ aus. Im Beispiel greifen also alle Prozessoren lesend auf die Werte der Nachbarpunkte zu. Nach der Kommunikationsphase führt jeder Prozessor auf den lokal gespeicherten Zwischenwerten die Additionen und die Division durch. Anschließend ersetzen alle Prozessoren, wiederum vollständig simultan, ihre Werte $v(i,j)$ durch die berechneten Mittelwerte. Programme dieser Art sind leicht zu verstehen, da sie ihren sequentiellen Analoga stark ähneln. Laut Hillis und Steel [22] ist der beste Weg, sich die SIMD-Betriebsweise klarzumachen, die Vorstellung eines sequentiellen Programms, das gleichzeitig auf Mengen von Daten statt auf einem einzelnen Datenelement ausgeführt wird. Auf SIMD-Maschinen ist es prinzipiell unmöglich, ein laufzeitabhängiges Fehlverhalten zu programmieren.

Bei MIMD-Betriebsweise hat jeder Prozessor ein eigenes Programm gespeichert. Diese Programme brauchen nicht identisch zu sein. Nun muß aber die Maschine explizite Synchronisierungsmechanismen zur Verfügung stellen. Diese erfordern Puffer zum Zwischenspeichern von Nachrichten, Anzeigen für eingelaufene Signale und Botschaften, sowie Instruktionen, die auf die Anzeigen warten und gegebenenfalls das Programm verzweigen. Der Programmierer muß die Synchronisierungsinstruktionen immer dann verwenden, wenn eine definierte Reihenfolge von Ereignissen erforderlich ist. Die Anweisung (1) muß zum Beispiel realisiert werden als

$$\begin{array}{l} PUT(v, i-1, j) \\ PUT(v, i+1, j) \\ PUT(v, i, j-1) \\ PUT(v, i, j+1) \\ v = [GET(i-1,j) + GET(i+1,j) + GET(i,j-1) + GET(i,j+1)]/4 \end{array} \tag{2}$$

Dabei sendet PUT eine Nachricht, die aus v besteht, an den angegebenen Prozessor, während GET auf das Eintreffen einer Nachricht vom spezifizierten Prozessor wartet. Der Programmierer muß sicherstellen, daß jede GET-Anweisung beim empfangenden Prozessor zu einer zugehörigen PUT-Anweisung im Programm des Senders paßt, was zu einer enormen Zunahme der Programmkomplexität und Fehleranfälligkeit führt. Diese Fehleranfälligkeit ist insbesondere deshalb von gravierender Bedeutung, da geschwindigkeits-abhängige Fehler nur schwierig zu finden sind. Man stelle sich vor, welche Möglichkeiten bestehen, wenn nicht alle GET-Anweisungen vor den PUT-Anweisungen ausgeführt werden.

Die wichtigste Einschränkung des SIMD-Konzeptes ist die Tatsache, daß alle Prozessoren die gleiche Instruktion ausführen müssen; lediglich ein geplantes Ignorieren von Instruktionen ist erlaubt. Selbst bei hochgradig regelmäßigen Problemen gibt es immer Sonderfälle, die eine — zumindest kurzfristig — andersartige Behandlung erfordern. Man betrachte zum Beispiel die Ränder eines Gitters. In solchen durch `IF`-Anweisungen gehandhabten Fällen muß die SIMD-Maschine zunächst die Prozessoren, die für die Ränder zuständig sind, abschalten, dann die Instruktionen für die Mehrheit versenden, anschließend den Aktivitätszustand aller Prozessoren invertieren

(eingeschaltete werden abgeschaltet und umgekehrt), ehe die Instruktionen für die Ränder ausgesandt werden. Ein MIMD-Rechner, der die Programme für das Innere und die Ränder des Gitters gleichzeitig ausführen kann, leidet nicht unter dieser Einschränkung. Allerdings erfordert ein MIMD-Rechner viel mehr Platz zur Speicherung des Programmes. Angenommen, ein Programm erfordert nur 100 Kbyte Speicherplatz pro Prozessor. Für eine 10.000-Prozessor MIMD-Anlage summiert sich dies zu 1 GByte, und damit zu einem beachtlichen Kostenfaktor der Anlage. Die dazu erforderlichen Transistoren (plus die Transistoren, die im komplexeren Steuerungsteil eines MIMD-Prozessors gebraucht werden), könnten gut zu einer Vervielfachung der Leistung eines vergleichbaren SIMD-Rechners genutzt werden. Bell [5] schätzt daher, daß die schnellsten und kosteneffektivsten, hochparallelen Rechner zur SIMD-Klasse gehören werden.

Einem oftmals gehörten Einwand gegen parallelen Maschinen (sowohl SIMD als auch MIMD), daß eine volle Auslastung der Prozessoren praktisch nicht erreichbar ist, stellen wir hier einen Vergleich mit der heutigen Verwendung des Hauptspeichers bei sequentiellen Maschinen entgegen: Hauptspeicher ist so preisgünstig geworden, daß es sich der Programmierer leisten kann, enorme Gebiete im Speicher nur selten zu nutzen. Die überwiegende Mehrheit des vorhandenen Speicherplatzes liegt brach und wartet auf den nächsten Zugriff. In analoger Weise ist bei der Entwicklung paralleler Algorithmen ein Umdenken erforderlich. Ein paralleler Algorithmus muß nicht mehr einzig auf die Vollauslastung der Prozessorkapazität hin ausgelegt sein: Eine gewisse Unterauslastung der Prozessoren ist genauso tragbar wie eine nur sporadische Nutzung des Hauptspeichers. Statt die Auslastung allein durch den Grad der Aktivität der Prozessoren zu definieren, sollte die aktive Nutzung aller Transistoren, also von Hauptspeicher und Prozessoren, berücksichtigt werden. Hillis [20] formuliert eine ähnliche Überlegung: er fordert, daß jede Speicherzelle rechnen kann. Damit verwischt sich der Unterschied zwischen Speicher und Prozessor.

3 Stand der Technik

Wir untersuchen in diesem Abschnitt marktgängige Parallelrechner und stellen für einige wichtige Modelle realistische Leistungsdaten und das Preis/Leistungsverhältnis gegenüber.

3.1 Übersicht

Aus der oben dargestellten Klassifikation wird deutlich, daß es zumindest acht klar unterscheidbare Architekturformen für Parallelrechner gibt. In der Praxis haben sich jedoch erst drei dieser Typen durchgesetzt.

- **MIMD - grobe Granularität - zentraler Speicher.**
  Sequent Symmetry, Encore Multimax, Alliant, Convex, Cray.
- **MIMD - grobe Granularität - verteilter Speicher.**
  Intel iPSC/2, nCUBE-2, Suprenum.
- **SIMD - feine Granularität - verteilter Speicher.**
  Connection Machine CM-2, MasPar MP-1, MPP, ICL DAP.

Für die Beschränkung auf drei Typen lassen sich folgende Gründe angeben. Der zentrale Speicher hat sich bei hochparallelen Anlagen bis jetzt nicht durchsetzen können. Architekturen, die den zentralen Speicher über einen Bus realisieren, sind nur eingeschränkt einsetzbar (s.o.). Ein zentraler Bus läßt sich nicht auf hochgradige Parallelität mit einigen tausend Prozessoren ausbauen. Wenn hingegen der zentrale Speicher durch ein vorgeschaltetes Netzwerk verwirklicht wird, das die Anforderungen auf die (in Wirklichkeit verteilten) Speichermodule weiterleitet, dann macht es wenig Sinn, jedem Prozessor gleich hohe Kommunikationskosten für alle Zugriffe aufzubürden. In diesem Fall ist es technisch einfach, jedem Prozessor ein Speichermodul direkt zuzuordnen und für dieses Modul die maximale Zugriffsgeschwindigkeit zuzulassen. Schließlich sind keine veröffentlichten Testergebnisse bekannt, die wenigstens für eine geringe Prozessoranzahl nachweisen würden, daß mit zentralem Speicher ein echter Vorteil zu erringen wäre. Somit blieb der zentrale Speicher bis jetzt bei hochparallelen Maschinen weitgehend unbeachtet.

Des weiteren beobachtet man, daß bis jetzt MIMD mit grober Granularität und SIMD mit feiner Granularität gepaart wurden. Der Grund sind die hohen Kosten für die Synchronisation bei den bisherigen MIMD-Architekturen. Bei feinkörnigen Algorithmen ist eine häufige Synchronisation erforderlich, was heute mittels Synchronisationsbotschaften im Netz verwirklicht wird und daher teuer ist. Bei SIMD-Maschinen ist die Synchronität direkt gegeben; nur deshalb haben SIMD-Maschinen mit feiner Granularität Erfolg.

Die Vorherrschaft obiger Architekturen bedeutet keinesfalls, daß die anderen Ansätze auch in der Zukunft keine Chance hätten. Bei MIMD-Maschinen könnte man mit Hilfe eines einfachen, globalen Signalnetzes eine schnelle Synchronisation verwirklichen. Die Datenflußarchitektur [23] und der bei Monarch [35] gewählte Ansatz basieren ebenfalls auf dem MIMD-Prinzip, während es gleichzeitig sehr wohl gelingt, mit feiner Granularität umzugehen. Eine weiteres Architekturprinzip, das für hochparallele Rechnung angewendet werden kann, ist das der neuronalen Netze. Diese werden für Probleme der kombinatorischen Optimierung und der Mustererkennung angewandt, hier aber aus Platzgründen nicht weiter betrachtet.

3.2 Preis-/Leistungsverhältnis

Wir haben jeweils zwei typische SIMD- und MIMD-Maschinen untersucht und stellen unsere Ergebnisse in diesem Abschnitt vor. Wir betrachten hierzu lediglich hochparallele Systeme, die mindestens 1.000 Prozessoren enthalten. Nur solche Maschinen geben einen deutlichen Vorsprung gegenüber herkömmlichen sequentiellen Maschinen und Vektorrechnern, der auch langfristig gehalten werden kann. Zahlreiche andere, niedrig-gradig parallele Multiprozessoren, wie etwa Suprenum, ip-Systems, iPSC oder andere, kommen durch diese Beschränkung nicht in Betracht.

Untersuchte Maschinen

Wir untersuchten folgende Modellreihen:

- **MasPar MP-1.** Die MP-1 ist eine SIMD-Maschine mit verteiltem Speicher. Sie besteht aus einem Gitter von bis zu 16.384 4-Bit Prozessoren, die jeweils über 40 32-Bit-Register und 16 kByte lokalem Speicher verfügen. Neben einem schnellen Kommuniaktionsnetz für Datenaustausch mit acht Nachbarprozessoren verfügt die Maschine über einen hierarchischen Kreuzschienenverteiler, welcher Nachrichten an beliebige Prozessoren vermittelt.

  Verwendet der Programmierer FORTRAN- oder C, so sind Prozessoren nicht explizit sichtbar, da die Übersetzer die Abbildung auf die Maschine realisieren. Bei der Erstellung maschinennahen Codes hingegen muß sich der Programmierer stets der zweidimensionalen Gitterstruktur und der tatsächlich vorhandenen Prozessorzahl bewußt sein.

- **Connection Machine CM-2.** Auch die CM-2 ist ein SIMD-Rechner mit verteiltem Speicher. Sie besteht aus bis zu 65.536 1-Bit-Prozessoren, die in einem 16-dimensionalem Hyperkubus angeordnet sind. Jeweils 32 davon können mit einem Gleitkommabeschleuniger (entweder 32 oder 64 Bit breit) ausgestattet werden, also mit insgesamt 2.048 solcher Einheiten.

  Die CM-2 realisiert auf dem Hyperkubus zweierlei Kommunikationsprotokolle: ein schnelles für Nachbarschaftskommunikation in n-dimensionalen Tori, und ein langsameres für allgemeine Kommunikationsmuster. Damit weist die CM-2 wie die MP-1 eine Optimierung der Nachbarschaftskommunikation auf.

  Die CM-2 bietet außerdem zweierlei Programmiermodelle: Das eine ist auf die 1-Bit-Prozessoren hin orientiert und wird von der Assemblersprache PARIS verwirklicht. Das zweite Modell ist auf die Nutzung der Gleitkommaeinheiten abgestellt und von der Assemblersprache PEAC realisiert. Das PARIS-Modell ist das abstraktere, denn es nimmt eine sog. Prozessorvirtualisierung vor: Die einzelnen PARIS-Instruktionen simulieren, wenn erforderlich, automatisch eine höhere Anzahl von Prozessoren als tatsächlich vorhanden. Die Gleitkommaeinheiten sind in diesem Modell unsichtbar und werden automatisch angesprochen. Auch gibt es in PARIS keine Register.

  Die Automatik der Prozessorvirtualisierung und das implizite Ansteuern der Gleitkommaeinheiten zieht Leistungseinbußen nach sich, die in PEAC vermieden werden. Das PEAC-Modell bietet im wesentlichen

2.048 Gleitkommaprozessoren, die nicht automatisch virtualisert werden, und über jeweils 32 vorgeschaltete 1-Bit-Prozessoren in einen 16-dimensionalen Hyperkubus eingebettet sind. Die Virtualisierung muß der Programmierer oder der Übersetzer vornehmen, aber die erreichbaren Leistungen sind deutlich höher. Das PEAC-Modell ähnelt damit dem Programmiermodell der MP-1 insofern, als auch dort die Virtualisierung entweder durch den Programmierer oder den Übersetzer erfolgt.

- **Transputernetz fester Topologie.** Der INMOS Transputer ist ein 64-Bit Mikroprozessorchip mit vier Kommunikationskanälen. Über diese Kanäle können Transputer entweder statisch oder dynamisch in Netzen zusammengeschaltet werden. Wir betrachten aus Preis- und Leistungsgründen ein System mit statischer Gittertopologie mit 1.024 Transputern T800/30. Es handelt sich hier also um einen MIMD-Rechner mit verteiltem Speicher.

  Auch hier gibt es wieder zwei Kommunikationsprotokolle. Kommunikation zwischen unmittelbaren Nachbarn wird von der Transputer-Hardware selbstätig durchgeführt. Bei Kommunikation zwischen entfernten Transputern sind spezielle Software-Vorkehrungen erforderlich: In jedem Transputer muß ein hochpriorer Prozeß lauffähig sein, der Empfang, Pfadberechnung und Weiterreichen der Botschaften vornimmt. Eine Nachricht durchläuft entlang der geschalteten Transputertopologie dabei im allgemeinen eine Reihe dieser Vermittlungsprozesse.

  Das Programmiermodell ist durch die Sprache Occam gegeben. Das Kanalkonzept in Occam ist direkt durch die Transputerkanäle realisiert. Automatische Prozessorvirtualisierung oder virtualisierende Übersetzer gibt es nicht. Der Programmierer kann natürlich mehrere Prozesse pro Transputer starten, muß aber deren Verwaltung selbst übernehmen. Software zur Durchführung der Kommunikation über mehrere Vermittlungspunkte hinweg gibt es bis jetzt nur als Prototypen, und nicht vom Hersteller.

- **nCUBE-2.** Der nCUBE-Rechner ist, wie das Transputernetz, ein MIMD-Rechner mit verteiltem Speicher. Ein nCUBE-2 besteht aus einem 13-dimensionalen Hyperkubus mit bis zu 8.192 64-Bit-Rechnern. Der Hyperkubus vermittelt vollautomatisch Botschaften zwischen Prozessen, auch über mehrere Vermittlungspunkte. Spezielle Vorkehrungen zur beschleunigten Behandlung von Nachbarschaftskommunikation, wie bei den anderen drei Parallelrechnern, gibt es nicht. Auch gibt es keine automatische Prozessorvirtualisierung oder virtualisierende Übersetzer. Das Programmiermodell ist daher ähnlich wie beim Transputernetz, mit der Ausnahme daß die Nachrichtenvermittlung beim nCUBE von Hersteller implementiert ist.

Preisangaben der Hersteller

Die Preise in Tabelle 1 entsprechen dem Stand vom Januar 1991. Angegeben sind die Preise für Mustermaschinen mit verschiedener Prozessorausstattung und minimaler Hauptspeichergröße. Bei der CM-2 sind die Preise für eine entsprechende Anzahl Gleitkommaeinheiten (64 Bit) eingeschlossen. Bei Parsytec lag nur ein Angebot für einen Speicher von 4 MByte pro Prozessor vor, obwohl die Minimalausstattung nur 1 MByte erfordert. Wartungskosten und die Preise aller Extras, wie etwa zusätzlicher Software, Massenspeicher oder Graphikterminale, sind nicht berücksichtigt. Außerdem ist in den Preisen nur die Vorbereitung für den Anschluß eines Vorrechners inbegriffen; nicht enthalten ist jedoch der Vorrechner selbst.

Durchführung der Messungen

Da für Parallelrechner noch keine allgemein anerkannten Benchmarks wie etwa die Whetstone-, Dhrystone- oder SPEC-Mixe [44, 45] vorliegen, mußten wir notgedrungen auf die weniger aussagekräftigen Zählungen von Instruktionen in synthetischen Programmen zurückgreifen. Wir bestimmten daher die tatsächlich erreichbaren Ausführungsraten von Ganzzahl- und Gleitkommaoperationen, gemessen in **MIOPS** (**M**illion **I**nteger **O**perations **P**er **S**econd) und **MFLOPS** (**M**illion **FL**oatingpoint **O**perations **P**er **S**econd). Beide Größen ergeben sich, indem man die Ausführungsraten für Addition und Multipliaktion mittelt; die Division bleibt dabei unberücksichtigt. Für MFLOPS unterscheiden wir auch noch die Rechnung in einfacher (32 Bit) und doppelter (64 Bit) Genauigkeit.

Um die Kommunikationsleistung zu messen, führen wir die Größe **MCPS** (**M**illion **C**onnections **P**er **S**econd) ein. Diese Größe gibt die Anzahl der pro Sekunde austauschbaren Pakete an. In der Messung senden alle Prozessoren gleichzeitig, aber keine zwei Prozessoren senden an die gleiche Zieladresse. Es entsteht also keine

Anzahl Prozessoren	Hauptspeicher pro Prozessor	Hauptspeicher insgesamt	Listenpreis (ohne MWST)
MasPar MP-1			
1.024	16 kB	16 MB	570 TDM
4.096	16 kB	64 MB	900 TDM
8.192	16 kB	128 MB	1.330 TDM
16.384	16 kB	256 MB	2.200 TDM
Connection Machine CM-2			
4.096	8 kB	32 MB	380 TDM
8.192	8 kB	64 MB	740 TDM
16.384	8 kB	128 MB	2.270 TDM
32.768	8 kB	256 MB	4.630 TDM
65.536	8 kB	512 MB	8.580 TDM
Transputer SC-1024 FT			
1.024	4.096 kB	4096 MB	4.000 TDM
nCUBE-2			
1.024	1.024 kB	1.024 MB	7.110 TDM
2.048	1.024 kB	2.048 MB	14.030 TDM
4.096	1.024 kB	4.096 MB	25.000 TDM
8.192	1.024 kB	8.192 MB	50.000 TDM

Tabelle 1: Preise

Kollision bei den Empfängern, und der Datenaustausch bewirkt lediglich eine Permutation der gesamten Pakete. Der Dateninhalt jeden Paketes umfaßt nur vier Bytes. MCPS ist für hochparallele Rechner aussagekräftiger als die sonst übliche Netzwerkbandbreite. Die Größe MCPS deckt vor allem die Latenzzeit auf, die in hochparallelen Anwendungen mit ihren häufig gesandten, aber kleinen Paketen das wesentliche Bewertungskriterium ist. Auch die Qualität der Kollisionsauflösung beim Fluten des Kommunikationsnetzes wird von MCPS gut erfaßt.

Bei der Kommunikationsleistung hat die gewählte Permutation einen wesentlichen Einfluß. Wir unterscheiden daher die Nachbarschaftskommunikation von eine allgemeiner Permutation; letztere erfordert für jede Nachricht eine Vermittlung über Zwischenknoten. Die Nachbarschaftskommunikation ist bei dreien der untersuchten Maschinen deutlich optimiert. Für die allgemeine Permutation bestimmten wir den Mittelwert aus folgenden Permutationen: Mischungspermutation (perfect shuffle), Butterfly, Adreßkomplement (die Zieladresse ist das bitweise Komplement der Startadresse), und Adressenspiegelung (die Zieladresse besteht aus den Bits der Startadresse in umgekehrter Reihenfolge).

Tabellen 2 und 3 geben Rechen- und Kommunikationsleistung wieder. In realistischen Programmen, die sowohl rechnen als auch kommunizieren, sind diese Leistungen gleichzeitig natürlich nicht zu erzielen. Aus einem (evtl. geschätzten) Verhältnis der Häufigkeiten von Rechen- und Kommunikationsoperationen läßt sich die tatsächlich erzielbare Leistung aber errechnen.

Die angegebenen Rechenleistungen wurden durch Programme erzielt, die lediglich die entsprechenden Operationen enthielten. Von den Laufzeiten subtrahierten wir außerdem noch die Kosten für Schleifenkontrolle und Initialisierungen.

Für die MP-1-Messungen stand eine Anlage mit nur 4.096 Prozessoren zur Verfügung; die Angaben für andere Prozessorausstattungen sind entsprechend herauf- oder heruntergerechnet. Wegen der modularen Architektur dieser Maschine sind aber die errechneten Leistungsdaten als zuverlässig zu betrachten. Lediglich die allgemeine Kommunikation könnte bei größeren Modellen etwas langsamer sein. Der Steuerungsrechner war ein VAX 3250. Programmiert wurde in MPL, einer C-ähnlichen Sprache die alle Architekturdetails sichtbar macht.

Für die CM-2-Messungen benutzten wir 8.192 sowie 16.384 Prozessoren mit 64-Bit Gleitkommaeinheiten, sowie dem schnellsten verfügbaren Steuerungsrechner, einer SUN-4. Auch hier ist das Herauf- oder Herunterrechnen der Leistung für andere Prozessorkonfigurationen zuverlässig.

Anzahl Prozessoren	MIOPS (32 Bit)	MFLOPS (32 Bit)	MFLOPS (64 Bit)	Preis pro MFLOPS(64)
MP-1, MPL				
1.024	250	125	75	7,6 TDM
4.096	500	250	150	6,0 TDM
8.192	1.000	500	300	4,4 TDM
16.384	2.000	1.000	600	3,7 TDM
CM-2, PARIS-Model, C†				
4.096	70	125	70	5,4 TDM
8.192	150	250	150	4,9 TDM
16.384	300	500	300	7,6 TDM
32.768	600	1.000	600	7,7 TDM
65.536	1.200	2.000	1.200	7,1 TDM
CM-2, PEAC-Model, FORTRAN				
4.096	225	375	270	1,4 TDM
8.192	450	750	540	1,4 TDM
16.384	900	1.500	1.075	2,1 TDM
32.768	1.800	3.000	2.150	2.1 TDM
65.536	3.600	6.000	4.300	2,0 TDM
SC-1042 FT, Occam				
1.024	2.000	2.000	1.000	4,0 TDM
nCUBE-2, FORTRAN				
1.024	4.222	2.993	2.289	3,1 TDM
2.048	8.444	5.986	4.578	3.1 TDM
4.096	16.889	11.973	9.156	2,7 TDM
8.192	33.778	23.946	18.312	2,7 TDM

†Die Leistungsdaten für dieses Modell wurden ab einem Virtualisierungsgrad von 32 (d.h. 32 virtuelle Prozessoren pro realem Prozessor) erreicht. Bei Virtualisierungsgrad 1 sinkt die Leistung auf 75%.

Tabelle 2: Rechenleistung

Bei der CM-2 untersuchten wir die Rechenleistung für beide Programmiermodelle. Das PARIS-Modell mit automatischer Virtualisierung wurde in C mit eingeschobenen PARIS-Instruktionen (Assemblerinstruktionen) programmiert. Das PEAC-Modell wird vom Fortran-Übersetzer angesprochen. Wie man sieht, bietet das PEAC-Modell eine fast vier Mal höhere Rechenleistung, obwohl von einer höheren Programmiersprache ausgegangen wird. Allerdings erfordert diese höhere Leistung ein sorgfältiges Feinoptimieren des Fortran-Codes, um die Register der Gleitkommaeinheiten voll zu nutzen. Ein naives Fortran Programm ist leicht um einen Faktor fünf langsamer, und damit sogar schlechter als ein sorgfältig geschriebenes Programm in C/PARIS.

Der Transputer wurde in Occam programmiert. Die Rechenleistung ist aus Ein-Prozessor-Programmen hochgerechnet. Die Nachbarschaftskommunikation wurde zwischen Paaren von Transputern gemessen. Bei der Nachbarschaftskommunikation nehmen wir an, daß nur ein Ausgang und ein Eingang pro Prozessor aktiv sind, denn andernfalls wäre das Ergebnis unvergleichbar mit den Nachbarpermutationen der anderen Parallelrechner. Die Kommunikationsleistung für allgemeine Permutationen basieren auf Ergebnissen, die mit Vermittlungssoftware bei der Gesellschaft für Mathematik und Datenvarbeitung in Bonn erzielt wurden [47].

Eine nCUBE-Maschine stand uns zu Testzwecken nicht zur Verfügung. Hier berufen wir uns auf Messungen von Michael Carter bei Ames Laboratory, sowie auf Ergebnisse in [18, 19]. Die angegebene Spitzenleistung von nCUBE kann nur bei einer hohen Dichte von Gleitkommaoperationen und sorgfältigst optimierten FORTRAN-Programmen erreicht werden, da sich nur dann eine volle Auslastung des Prozessors einstellt.

Anzahl Prozessoren	MCPS Nachbar	MCPS allg.	Preis pro MCPS(allg.)
MP-1, MPL			
1.024	464	13	45,7 TDM
4.096	925	25	35,9 TDM
8.192	1.850	50	26,7 TDM
16.384	3.700	100	22,0 TDM
CM-2, PARIS-Modell, C CM-2, PEAC-Modell, FORTRAN			
4.096	125†	4‡	95,0 TDM
8.192	250	8	92,5 TDM
16.384	500	16	141,9 TDM
32.768	1.000	32	144,7 TDM
65.536	2.000	64	134,1 TDM
SC-1024 FT, Occam			
1.024	200	< 1	4.000,0 TDM
nCUBE-2, FORTRAN			
1.024	7	7	1.027,8 TDM
2.048	14	14	1.008,5 TDM
4.096	28	28	892,9 TDM
8.192	56	56	892,9 TDM

†Im PARIS-Modell wird die angeg. Leistung in dieser Spalte erst bei Virtualisierungsgrad 32 erreicht; bei einem Virtualisierungsgrad von 1 sinkt die erreichbare Leistung auf 25%. Das PEAC-Modell ist nicht sensitiv gegenüber Virtualisierung.

‡Im PARIS-Modell wird die angeg. Leistung in dieser Spalte nur bei Virtualisierungsgrad 1 erreicht. Bei Virtualisierungsgrad 32 sinkt sie auf 25% der angeg. Leistung. (Man beachte die umgekehrte Richtung der Leistungsminderung.) Das PEAC-Modell ist nicht sensitiv gegenüber Virtualisierung.

Tabelle 3: Kommunikationsleistung

Bewertung

Die höchste absolute Rechenleistung erreicht nCUBE mit 18 GFLOPS (64Bit), gefolgt von CM-2 mit 4 GFLOPS. Das Preis/Leistungsverhältnis bei der reinen Rechenleistung liegt bei allen Typen nicht sehr weit auseinander: etwa ein Fakor 2 bei vergleichbar schnellen Modellen.

Die höchste absolute Kommunikationsleistung schafft MP-1 mit 100 MCPS (allgemein), gefolgt von CM-2 mit 64 MCPS. Zu beachten ist, daß die CM-2 die Spitze der angegebenen, allgemeinen Kommunikationsleistung im PARIS-Modell nur bei einem niedrigen Virtualisierungsgrad von 1 erreicht, während sich die Spitze der Rechenleistung erst bei einem Virtualisierungsgrad von 32 einstellt. Die zwei Spitzenwerte sind also auf der CM unter PARIS nicht gleichzeitig zu erreichen. Dieser Unterschied ergibt sich weder im PEAC-Modell noch auf der MP-1, da diese nicht automatisch virtualisieren und explizit programmierte Virtualisierung die Leistungswerte nicht beeinfußt. Die Effekte der Virtualisierung auf den MIMD-Rechnern sind weitgehend ungeklärt.

Die MIMD-Maschinen liegen in der Kommunikationsleistung gleich um ein bis zwei Größenordnungen hinter den SIMD-Rechnern zurück. Das Preis/Leistungsverhältnis ist entsprechend drastisch unterschiedlich. Die schlechteste, allgemeine Kommunikationsleistung tritt beim Transputernetz auf. Für weniger als 1 MCPS allgemeiner Kommunikation müssen 4 Mio DM aufgewendet werden, vier Mal mehr als bei nCUBE, 40 Mal mehr als bei CM-2, und 100-200 Mal mehr als bei MP-1. Aufgrund der schwachen Kommunikationsleistung sind die MIMD-Maschinen für Anwendungen mit hohem Anteil an Kommunikationsoperationen, d.h. für Anwedungen mit feiner Granularität, schlecht geeignet.

Die Kosten für die Programmierung gehen aus den obigen Tabellen nicht hervor. Das Programmiermodell des Datenparallelismus (die gleiche Operation auf vielen Datenelementen) auf den SIMD-Maschinen scheint

jedenfalls wesentlich einfacher beherrschbar als der Kontrollparallelismus (parallel ablaufende Instruktionsströme) auf den MIMD-Maschinen. Die Übertragung des Datenparallelismus auf MIMD-Maschinen ist im Prinzip möglich. Jedoch sind hier hohe Laufzeitkosten zu erwarten, die von den häufig notwendigen Synchronisationspunkten herrühren. Die untersuchten MIMD-Rechner bieten hierzu keinerlei schnelle Hardware-Unterstützung an, wodurch die Synchronisation aufwendig in Software implementiert werden muß und durch die langsamen Netzwerke stark behindert wird. Mittels optimierender Übersetzer möglichst viele Synchronisationspunkte zu entfernen ist Gegenstand aktueller Forschung [32].

4 Forschungsthemen der Informatik

In der Einleitung stellten wir bereits den sich vollziehenden Paradigmenwechsel zum hochgradigen Parallelismus und dessen Bedeutung für die Informatik als Ganzes heraus. Die Messungen des letzten Abschnitts vermittelten einen Begriff für die Leistungssteigerungen, die mit hochparallelen Anlagen möglich sind. Dieser Abschnitt stellt den Versuch dar, die zum gegenwärtigen Zeitpunkt absehbaren Forschungsthemen der parallelen Informatik zu katalogisieren.

4.1 Rechnerarchitektur

Während sich bei den sequentiellen Rechnern eine deutliche Stabilisierung und Standardisierung der Architekturen abzeichnet, stehen wir bei Parallelrechnerarchitekturen noch ganz am Anfang. Auch bietet der Parallelrechnerentwurf ganz neue Freiheitsgrade. Der Raum möglicher Architekturen ist noch weitgehend unerforscht und das Entwicklungspotential paralleler Hardware noch unabsehbar.

Netzwerktopologie

Noch ist völlig offen, welche Netzwerkarchitektur oder welche Kombination unterschiedlicher Topologien langfristig am geeignetsten ist. Eine Lösung dieser Frage ist insbesondere deshalb wichtig, da die Leistung des Kommunikationsnetzes der wichtigste Engpaß heutiger, und wohl auch zukünfiger, Parallelrechner ist.

Wegen der Vielfalt der vorgschlagenen Netzwerktopologien ist hier keine vollständige Übersicht möglich. In existierenden, hochparallelen Anlagen werden heute folgende Netze benutzt: Hyperkubus, Gitter, hierarchische Kreuzschiene, Banyan, und Fat Tree (ein Baum mit zunehmender Leitungszahl pro Ast in Richtung zur Wurzel). Neben den im Abschnitt 2.1 genannten Kriterien sind von zusätzlicher Bedeutung die Packungsdichte, die Regularität der Verdrahtung, die Verklemmungsfreiheit der Kommunikation und die Unterstützung der Synchronisation.

Die aktuelle Forschung wendet sich gegenwärtig mehr den einfachen, zwei- oder dreidimensionalen Gittern und Tori zu. In der Theorie haben diese Strukturen pro Datenaustausch mehr Vermittlungspunkte als etwa der Hyperkubus und die Mischungspermutation, dafür aber haben sie durch ihre Einfachheit und leichte Verdrahtbarkeit wichtige Vorteile: Man kann mit ihnen hohe Packungsdichte und damit hohe Geschwindigkeit erreichen. Bei großen Netzen sind aber weiterhin Topologien mit asymptotisch günstigem (logarithmischem) Durchmesser zu bevorzugen.

Prozessorelement

Das Spektrum der heute in Parallelrechnern eingesetzten Prozessorelemente ist extrem breit: es reicht von äußerst einfachen, bit-seriellen Einheiten bis zu eigenständigen, leistungsstarken Rechnern (64 Bit breit), die mit eigenem Bus und Peripherie ausgestattet sind. Die zur Kommunikation und Synchronisation am besten geeigneten Instruktionstypen sind noch nicht bekannt. Man kennt noch nicht einmal das ungefähre Verhältnis zwischen Instruktionen für Arithmetik, lokalem Speicherzugriff, Kommunikation, und Synchronisation. Unterstützung zur Prozessorvirtualisierung, schnellen Prozeßerzeugung und Synchronisation steckt in den Anfängen. Auch ist zum Beispiel unklar, wieviele Register ein Prozessor zur Verfügung stellen soll. Da die Prozeßumschaltung sehr häufig stattfindet (in synchronen Anweisungen innerhalb weniger Instruktionen) ist ein Umladen der Register zu aufwendig. Im Vergleich dazu sind sequentielle RISC-Prozessoren heute sehr sorgfältig für die zu erwartenden

Instruktionsmixe ausgelegt. Die Abstimmung der Befehlssätze von Parallelrechnern auf die Anforderungen von realistischen Programmen ist eine wichtige Aufgabe.

Insgesamt ist zu erwarten, daß bei steigender Integrationsdichte Prozessoren, Kommunikationseinheiten und der gesamte, zugehörige Speicher auf ein Chip passen werden. Mittelfristig wird man aber weiterhin separate Speicherchips und Caches auf den Prozessorchips verwenden müssen.

Hochgradig Parallele Ein-/Ausgabe

Ein weiteres Problem heutiger Parallelrechner ist die oftmals festzustellende Unausgewogenheit zwischen Rechen- und E/A-Leistung. Denkbar ist eine E/A-Schnittstelle pro Prozessorelement, aber auch E/A-Knoten, die direkt ans Netzwerk angeschlossen sind und ihre Daten von den Prozessorelementen geschickt bekommen; sog. Plattenfelder. Mit ausreichend großem Adreßraum könnten Sekundärspeicher auch in den Adreßraum abgebildet werden. Damit würde jeder Sekundärspeicherknoten am Netz einen zwar relativ langsamen, dafür aber großen Speicher zur Verfügung stellen, und alle Rechenprozessoren könnten E/A lediglich mittels Transportbefehlen anstoßen.

4.2 Programmiersprachen und -methodik

Bei der Programmierung hochgradig paralleler Rechner lassen sich zwei grundsätzlich unterschiedliche Ansätze erkennen. Ein wichtiger Ansatz besteht darin, sequentielle, meist in FORTRAN formulierte Programme durch einen Übersetzer automatisch zu parallelisieren. Der andere Ansatz besteht darin, ein gegebenes Problem neu zu überdenken und dann einen parallelen Algorithmus dafür zu finden.

Der Ansatz der automatischen Parallelisierung durch einen Übersetzer wird bei der Berücksichtigung der enormen Investitionen verständlich, die in existierender Software und dem Ausbildungsstand der Programmierer gebunden sind. Parallelisierende Übersetzer haben deshalb die Informatik intensiv beschäftigt (siehe z.B. [33, 26, 30]) und werden es auch noch in Zukunft tun.

Es wird jedoch durch Algorithmenstudien, wie z.B. [42], immer deutlicher, daß automatische Parallelisierung mittelfristig nur begrenzte Erfolge erzielen kann. Das Ziel einer automatischen Erzeugung wirklich guter paralleler Programme kann — wenn überhaupt — nur durch eine Transformation erreicht werden, die bei der Problemspezifikation statt bei einer sequentiellen Implementierung beginnt. In der sequentiellen Lösung sind oftmals viele Parallelisierungsmöglichkeiten zu sehr versteckt oder bereits völlig eliminiert.

Problemorientierte, parallele Programmiersprachen

Die Entwicklung von explizit parallelen Programmen ist im Vergleich zu sequentiellen Programmen eine komplizierte und fehleranfällige Aufgabe. Bei der Programmierung müssen eine Vielzahl maschinenabhängiger Details, wie der Organisation des Speichers, der Struktur des Netzwerks, den Eigenschaften der Kommunikationsprotokolle, der Prozessoranzahl und dem Steuerungsmodus (SIMD, MIMD) gemeistert werden. Der Stand der Technik ist dadurch geprägt, daß parallele Programme nur mit großem Aufwand von einem zum anderen Parallelrechner übertragen werden können. Das gilt selbst für Sprachen wie C* oder *Lisp [41, 40], die auf einem relativ hohem Niveau zu stehen scheinen. Jedoch wird ein C*-Programm kaum effizient auf einer MIMD-Maschine laufen können, da die Sprache zu sehr auf eine SIMD-Maschine abgestimmt ist. Umgekehrtes trifft für MIMD-orientierte Sprachen wie Occam zu.

Damit parallele Programme leichter zu schreiben, zu pflegen und zu portieren sind, müssen parallele Programmiersprachen von maschinenabhängigen Details abstrahieren und Formulierungen in einer problemorientierten Weise zulassen. Maschinennahes Programmieren von Parallelrechnern wird mit zunehmender Verbreitung dieser Maschinen zu einem Luxus werden, den sich nur relativ wenige leisten können — eine Entwicklung, die dem sich ständig verringernden Einsatz von Assembler bei der Programmierung sequentieller Rechner vergleichbar ist.

Ein Ansatz, der in diese Richtung geht, ist Modula-2* [31, 43]. Modula-2* ist eine Erweiterung von Modula-2, die es gestattet, hochparallele und portable Programme zu schreiben, die durch Übersetzung auf einer Vielzahl paralleler Architekturen effizient ausführbar gemacht werden können. Erfreulicherweise sind die notwendigen Erweiterungen klein, leicht zu erlernen, und können ohne Schwierigkeiten von den meisten imperativen Sprachen

übernommen werden. Die Erweiterungen, um funktionale und logische Sprachen (insbes. Prolog) hochgradigem, möglicherweise expliziten Parallelismus zugänglich zu machen, sind ebenfalls Gegenstand aktueller Forschung [36, 9].

Programmiermethodik und -werkzeuge

Die Theorie der parallelen Algorithmen ist erfreulicherweise gut entwickelt und durch mehrere Lehrbücher zugänglich; siehe z.B. [1, 15]. Allerdings wurden die Algorithmen meist für Varianten des PRAM-Modells entwickelt. In diesem Modell sind die Kosten für die Kommunikation in einem verteiltem Speicher nicht berücksichtigt. Hierzu gibt es zweierlei Abhilfen: Die eine ist, das Netzwerk so schnell zu machen, daß eine Kommunikation nur unwesentlich langsamer abläuft als eine Rechenoperation. Dies ist zum Teil bei der MP-1 geglückt, wo die Nachbarschaftskommunikation schneller operiert als die Arithmetik. Die andere Abhilfe ist, Optimierungen zu finden, die den Anteil von langen Kommunikationswegen herabsetzen. Hierzu ist noch wenig systematische Arbeit bekannt.

Während parallele Algorithmen für den Bereich des „Programmierens im Kleinen“ gut entwickelt sind, befinden sich die Strukturen für das „ Programmieren im Großen“ noch weitgehend im Dunklen. Hier stellen sich Fragen der Systemarchitektur. Diese Fragen betreffen u.a. die Aufteilung der Aufgaben in einem parallel ablaufenden System und die Beherrschung der Parallelität auf zahlreichen Ebenen und mit unterschiedlichen Feinheitsgraden. Hierzu müssen Methodik und Bewertungsmaßstäbe erst gefunden werden.

Auch das Gebiet der Programmierwerkzeuge ist für parallele Rechner stark unterentwickelt. Selbst herkömmliche Debugger lassen sich nicht ohne weiteres auf Parallelrechner übertragen, da Synchronisierungsfehler als neue Fehlerklasse hinzukommen. Diese sind bekanntlich äußerst schwierig aufzudecken, da sie stark von Laufzeitbedingungen abhängen. Das Zuschalten eines Debuggers allein kann Fehlersymptome verschwinden lassen und ganz andere hervorrufen. Zusätzliche Werkzeuge sind nötig, um die Visualisierung verteilter Datenstrukturen, Datenabhängigkeiten und Kommunikationsmuster zu bewerkstelligen.

4.3 Übersetzerbau

Während der Übersetzerbau für sequentielle Rechner heute über ein umfangreiches Know-How verfügt, das sich in einer Vielzahl von Generierungswerkzeugen widerspiegelt, stellen sich für Parallelrechner neue Aufgaben und Chancen.

Automatische Parallelisierung

Im letzten Abschnitt wurde bereits auf diesen Problemkreis eingegangen. Heute sind jedoch noch längst nicht alle Möglichkeiten ausgeschöpft, aus einem sequentiellen Programm parallelisierbare Elemente herauszufinden. Eine Verfeinerung der Analysetechnik ist erforderlich.

Übersetzung problemorientierter Sprachen

Bei der Übersetzung von Hochsprachen, die die Formulierung paralleler Algorithmen in einer maschinenunabhängigen Weise erlauben, stellen sich ganz neue Probleme. Es mangelt an Techniken, mit denen einerseits die angegebenen Datenstrukturen günstig auf den insgesamt verfügbaren Speicher verteilt werden und andererseits Operationen so zusammengefaßt und auf die Prozessoren verteilt werden, daß es der Architektur der Maschine gerecht wird. Aufgrund der Vielzahl von Architekturen ist es besonders wichtig, herauszufinden, welche Transformationen auf Zwischensprachenebene maschinenunabhängig durchgeführt werden können. Die drastischen Unterschiede zwischen Instruktionsausführungs- und Kommunikationszeiten machen hierzu Überlegungen erforderlich, die weit über das bei herkömmlichen Übersetzern erforderliche Maß hinausgehen.

Parallelisierung des Übersetzungsprozesses

Wie im sequentiellen Fall ist auch der parallel arbeitende Übersetzer ein wichtiges Studienobjekt für Systemstruktur. Trotz einer Reihe von Ansätzen beschränkt sich die Parallelisierung des Übersetzungsprozesses bis

jetzt auf eine recht geringe Parallelität von höchstens zehn Prozessoren [37]. Hochgradige Parallelität kommt nur bruchstückhaft zum Einsatz. So gibt es Methoden für die parallele lexikalische und syntaktische Analyse [39]. Den Autoren sind jedoch bei der sematischen Analyse und Code-Erzeugung keine Ansätze bekannt, die aus tausenden von Prozessoren Nutzen ziehen. Auch die Integration mehrerer hochparalleler Phasen ist ungelöst.

4.4 Betriebssysteme

Existierende Parallelrechner bieten nur primitive Betriebssysteme, die meist keinen oder nur unbefriedigenden Mehrbenutzerbetrieb gestatten. Die Entwicklung eines Teilhabersystems für hochparallele Rechner ist dringend erforderlich. Dabei müssen insbesondere Probleme der parallelen Unterbrechungsbehandlung, der parallelen Betriebsmittelvergabe, des virtuellen Speichers, und der Synchronisierung von kommunizierenden, parallelen Prozessen gelöst werden. Interessante Aufgaben stellen sich auch bei der hochparallelen Ein-/Ausgabe und der Kopplung mehrerer Parallelrechner.

4.5 Spezielle Anwendungen innerhalb der Informatik

Für die folgenden Anwendungen aus dem Bereich der Informatik dürfte die Ausnutzung der Rechenleistung eines hochgradig parallelen Rechners wichtige, qualitative Fortschritte bringen. Die hier vorgestellte Liste kann natürlich nicht vollständig sein; sie diene eher als Anregung.

- Bild- und Signalverarbeitung (2D und 3D);
- Neuronale Netze;
- Robotik (Bahnplanung, Lernen reaktiven Verhaltens);
- Kryptographie (Zahlenfaktorisierung, Chiffrierverfahren);
- Datenbanken;
- Modellierung (Analyse und Leistungsbewertung digitaler Systeme).

4.6 Klassen von Anwendungsproblemen

Für eine große Menge von Anwendungen aus dem naturwissenschaftlichen Bereich kann zumindest eine der oben beschriebenen Parallelrechnerarchitekturen sinnvoll eingesetzt werden [13]. Fox hat eine Klassifikation von Problemen in drei große Klassen vorgeschlagen, die er mit folgenden Attributen kennzeichnet: *synchron*, *lose synchron* und *asynchron*. Synchrone Problem sind dabei solche, bei denen Gleichungen das Verhalten an jedem Punkt im Datenraum zu jedem Zeitpunkt festlegen. Lose synchrone Probleme zeichnen sich dadurch aus, daß sie über Zeitintervalle verfügen, an deren Enden Gleichungen die Werte der Datenelemente spezifizieren, aber innerhalb dieser Intervalle keine globale Festlegung aller Datenelemente möglich oder erforderlich ist. Alle noch verbleibenden Probleme zählt Fox zu der Klasse der asynchronen Probleme, für die in der Literatur, laut Fox, bisher nur wenig veröffentlich ist. Diese Tatsache zeigt, daß noch wenig Erfahrung vorhanden ist, auf welche Weise ein umfangreiches Problem in eine große Menge von unähnlichen Teilen zerlegt werden kann, die eine MIMD-Maschine auslasten.

Synchrone und lose synchrone Probleme

In vielen Anwendungsproblemen muß eine einzige, einfache Funktion auf alle Elemente einer Datenstruktur angewendet werden. Eine solche Funktion kann mit Hilfe eines sequentiellen Algorithmus spezifiziert werden. Jede Instruktion wird dann simultan auf allen zutreffenden Datenelementen ausgeführt.

Es lassen sich eine Vielzahl von Problemen finden, die auf diese Weise „datenparallel“ realisiert werden können. Die folgende Liste soll einen kleinen motivierenden Überblick vermitteln, ist aber naturgemäß nicht vollständig. Viele Teilprobleme, deren Lösungen heute aus Standardbibliotheken verfügbar sind, sind in der Tat einem datenparallelen Algorithmus zugänglich und lassen sich mit beachtlicher Beschleunigung auf Parallelrechnern realisieren [1, 15].

- Schaltkreissimulationen;
- Moleküldynamik und Sequenzanalysen;
- Freitextsuche und Dokumentennachweis;
- Anwendungen der Geophysik (z.B. Wellenfronten im Erdinneren);
- Strömungssimulation;
- Methode der finiten Elemente;
- diverse Anwendungen aus der linearen Algebra und Numerik.

Datenparallele Algorithmen müssen, damit sie sinnvoll verwendet werden können, so entworfen werden, daß sie sich der Anzahl der tatsächlich vorhandenen Prozessoren anpassen. Beispielsweise können eine Million Datenelemente mit nur tausend Prozessoren in einer Zeit durchsucht werden, die proportional zu log 1000 ist, in dem jeder Prozessor auf den ihm zugeteilten 1000 Elementen eine binäre Suche durchführt. Es ist ein noch offenes Problem, in welcher Weise parallele und sequentielle Teilalgorithmen kombiniert werden müssen, um skalierbare und effiziente Algorithmen für Parallelrechner zu erhalten.

Asynchrone Probleme

Die Lösung vieler rechenintensiver Anwendungsprobleme basieren auf der Zusammenstellung einer ganzen Reihe von mehr oder weniger unabhängigen Funktionen. Die folgende Liste stellt wieder in eher anregender als vollständiger Weise einige Beispiele aus einer Vielzahl von Anwendungsbereichen zusammen.

- Lösung der Laplace-Gleichung mit finiten Differenzen;
- Statik inhomogener Strukturen;
- Strömungsprobleme mit mehreren Zonen;
- Strömungen nichthomogener, unterirdischer Formationen.

Die Lösung solcher Probleme ist — zumindest heute — nicht in naheliegender Weise mit einem datenparallelen Ansatz zu erbringen. Offensichtliche Algorithmen basieren auf einem Netzwerk von Prozessoren, die unterschiedliche Funktionen ausführen und dabei Daten austauschen. Einen guten Überblick über asynchrone Probleme und eine Fundgrube von Algorithmen dafür bieten [14, 3].

5 Zusammenfassung und Ausblick

Der erste digitale und elektronische Rechner, der Atanasoff-Berry-Computer von 1942, war bereits ein Parallelrechner mit 30 gleichzeitig operierenden Recheneinheiten[27, 8]. Diese Pionierleistung geriet für über 45 Jahre in Vergessenheit. Statt dessen dominierte der sequentielle Rechner, ausgehend vom ENIAC von 1946. Nun beginnt sich die Informatik in breiter Front von dieser Spezialiserung zu lösen.

Nach zwei Jahrzehnten intensiver Experimente auf dem Gebiet der Multiprozessoranlagen werden nun Maschinen erfolgreich eingesetzt, die tausende von gleichzeitig operierenden Prozessoren umfassen. Viele noch ungelöste Probleme und Herausforderungen stehen zur Lösung an. Neben der riesigen Zahl zu bearbeitender Anwendungsproblemen stammen viele Aufgaben aus Kerngebieten der Informatik, so z.B. der Rechnerarchitektur, der parallelen Algorithmen, der Programmiersprachen, der Übersetzer, der Betriebssysteme, der Leistungsbewertung und des Software Engineerings. Parallelismus erweist sich damit nicht als eine neue Teildisziplin, sondern als eine Grundlage der Informatik.

Die Autoren danken der Firma MasPar Distributor AG, Zürich für die leihweise Überlassung einer MP-1204, und Thinking Machines München und Cambridge für Beratung und Rechenzeit auf einer Connection Machine. Herr Thomas Umland nahm dankenswerter Weise die Messungen auf dem Transputercluster der Informatik Fakultät Karlsruhe vor. Auch danken wir INRIA in Sophia-Antipolis für die Unterstützung dieser Arbeit.

Literatur

[1] Selim G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. of the AFIPS Spring Joint Computer Conference*, volume 30, pages 483–485, April 18-20, 1967.

[3] Augus, Geoffrey C. Fox, Kim, and David W. Walker. *Solving Problems on Concurrent Processors*, volume II – Software for Concurrent Processors. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[4] Forest Baskett and Alan Jay Smith. Interference in multiprocessor computer systems with interleaved memory. *Communications of the ACM*, 19(6):327–334, June 1976.

[5] C. Gordon Bell. The future of high performance computers in science and engineering. *Communications of the ACM*, 32(9):1091–1101, September 1989.

[6] Tom Blank. The MasPar MP-1 architecture. In *Proc. of the COMPCON Spring 1990 – The 35th IEEE Computer Society International Conference*, pages 20–24, San Francisco, CA, February 26 – March 2, 1990.

[7] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Gregory Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, Hilton Head, South Carolina, July 21-24, 1991.

[8] Alice R. Burks and Arthur W. Burks. *The First Electronic Computer: The Atanasoff Story.* University of Michigan Press, 1988.

[9] K. Clark and S. Gregory. PARLOG: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[10] Peter J. Denning and Walter F. Tichy. Highly parallel computation. *Science*, 250:1217–1222, November 1990.

[11] Jack J. Dongarra. Performance of various computers using standard linear equations software. *Computer Architecture News*, 18(1):17–31, March 1990.

[12] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C–21(9):948–960, September 1972.

[13] Geoffrey C. Fox. What have we learnt from using real parallel machines to solve real problems? In *Proc. of the Third Conference on Hypercube Concurrent Computers and Applications*, volume 2, pages 897–955, Pasadena, CA, 1988. ACM Press, New York.

[14] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors*, volume I – General Techniques and Regular Problems. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[15] Alan Gibbons and Wojciech Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[16] Ran Ginosar and David Egozi. Topological comparison of perfect shuffle and hypercube. *International Journal of Parallel Computing*, 18(1):37–68, 1989.

[17] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[18] John L. Gustafson, Robert E. Benner, Mark P. Sears, and Thomas D. Sullivan. A radar simulation program for a 1024-processor hypercube. In *Proc. of Supercomputing 1989*, pages 96–105, Reno, NV, November 13–17, 1989.

[19] John L. Gustafson, Diane Rover, Stephen Elbert, and Michael Carter. SLALOM the first scalable supercomputer benchmark. *Supercomputing Review*, pages 56–61, November 1990.

[20] W. Daniel Hillis. *The Connection Machine.* ACM distiguished dissertations. MIT Press Cambridge, Massachusetts, London, England, 1985.

[21] W. Daniel Hillis. Ultraschnelle Prozessor-Netzwerke. *Spektrum der Wissenschaft*, pages 52–61, August 1987.

[22] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[23] F. Hutner and R. Holzner. Architektur, Programmierung und Leistungsbewertung des MIT-Datenflußrechners. *Informatik Spektrum*, 12(3):147–157, June 1989.

[24] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing.* Series in Computer Organization and Architecture. McGraw-Hill Book Company, 1987.

[25] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, May 1990.

[26] D. J. Kuck, A. H. Sameh, R. Cytron, A. V. Veidenbaum, C. D. Polychronopoulos, G. Lee, T. McDaniel, B. R. Leasure, C. Beckman, J. R. B. Davies, and C. P. Kruskal. The effects of program restructuring, algorithm change and architecture choice on performance. In R. M. Keller, editor, *Proc. of the 1984 International Conference on Parallel Processing*, pages 129–138. Pennsylvania State University Press, August 1984.

[27] Allan R. Mackintosh. Dr. Atanasoff's computer. *Scientific American*, pages 72–78, August 1988.

[28] nCUBE, 1825 NW 167th Place, Beaverton, OR 97006. *The nCUBE 6400 Processor Manual*, 1989.

[29] John Nickolls. The design of the MasPar MP-1, a cost-effective massively parallel computer. In *Proc. of the COMPCON Spring 1990 – The 35th IEEE Computer Society International Conference*, pages 25–28, San Francisco, CA, February 26 – March 2, 1990.

[30] D. A. Padua and M.J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[31] Michael Philippsen, Christian G. Herter, and Walter F. Tichy. The Triton project. Technical Report No. 33/90, University of Karlsruhe, Department of Informatics, December 1990.

[32] Michael Philippsen and Walter F. Tichy. Modula-2* and its compilation. In *First International Conference of the Austrian Center for Parallel Computation, Salzburg, Austria, 1991*, pages 169–183. Springer Verlag, Lecture Notes in Computer Science 591, 1992.

[33] Constantine D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, Boston, Dordrecht, London, 1989.

[34] U. Raabe, M. Lobjinski, and M. Horn. Verbindungsstrukturen für multiprozessoren. *Informatik Spektrum*, 11(4):195–206, August 1988.

[35] Rondall D. Rettberg, William R. Crowther, Philip P. Carvey, and Raymont S. Tomlinson. The monarch parallel processor hardware design. *IEEE Computer*, 23(4):18–30, April 1990.

[36] E. Y. Shapiro. A subset of concurrent prolog and its interpreter (revised version). Technical Report 003, ICOT, February 1983.

[37] R. M. Shell. *Methods for Constructing Parallel Compilers.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 1979.

[38] Howard J. Siegel. *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies.* Series in Computer Science. Lexington Books, Lexington, MA, 1985.

[39] Guy L. Steele and W. Daniel Hillis. Connection Machine Lisp: Fine-grained parallel programming. In *Proc. of the 1986 ACM Symposium on Lisp and Functional Programming*, pages 279–297, Cambridge, Ma., August 1986.

[40] Thinking Machines Corporation, Cambridge, Massachusetts. **Lisp Reference Manual, Version 5.0*, 1988.

[41] Thinking Machines Corporation, Cambridge, Massachusetts. *C* Programming Guide, Version 6.0*, November 1990.

[42] Walter F. Tichy. Parallel matrix multiplication on the Connection Machine. *International Journal of High Speed Computing*, 1(2):247–262, 1989.

[43] Walter F. Tichy and Christian G. Herter. Modula-2*: An extension of Modula-2 for highly parallel, portable programs. Technical Report No. 4/90, University of Karlsruhe, Department of Informatics, January 1990.

[44] Reinhold P. Weicker. An overview of common benchmarks. *IEEE Computer*, 23(12):65–75, December 1990.

[45] Reinhold P. Weicker. SPEC Benchmarks. *Informatik Spektrum*, 13(6):334–336, December 1990.

[46] Larry D. Wittíe. Communication structures for large networks of microcomputers. *IEEE Transactions on Computers*, C–30(4):264–273, April 1981.

[47] Klaus Wolf. RSK – Ein Routingsystem für groše Transputernetzwerke. Technical report, Gesellschaft für Mathematik und Datenverarbeitung mbH, D-5205 Sankt Augustin, 30. Mai 1990.