Parallelrechner aus wissenschaftlicher und kommerzieller Sicht.

W. J. Paul
Fachbereich Informatik
Universität
6600 Saarbrücken

1. Kommerzielles. Parallelrechner sind Rechner mit 2 oder mehr CPU's. Der Einsatz mehrerer CPU's kann der Steigerung der Leistung oder der Zuverlässigkeit dienen. Setzt man $p \geq 2$ identische CPU's ein, so hofft man auf einen Gewinn an Geschwindigkeit αp gegenüber einer einzelnen CPU gleichen Typs. Gelingt es, $0.8 \leq \alpha$ zu erreichen ist man glücklich, bei $\alpha < 1$ ist man verblüfft. Die Größe α = Laufzeit auf einer CPU / (p*Laufzeit auf p CPU's) heißt *Effektivität.*

Am einfachsten erreicht man $\alpha = 1$, wenn jede CPU unabhängig von den anderen CPU's auf ihrem Speicher (bzw. Cache) arbeitet. Diese Situation tritt in natürlicher Weise in Mehrbenutzersystemen auf. Aus wissenschaftlicher Sicht ist das nicht furchtbar aufregend. Kommerziell wird dies von namhaften Herstellern schon lange benutzt. Die Anzahl der CPU's ist in der Regel klein.

Aufregender für die Wissenschaftler und auch für die Hersteller wird es, wenn viele CPU's bei der Lösung einer gemeinsamen Aufgabe kooperieren müssen. Auch diese Idee ist nicht sehr neu, bekam aber am Anfang dieses Jahrzehnts großen Auftrieb aus 2 Quellen: VLSI-Technologie und sehr hohe Forschungsmittel in Amerika. Parallelismus wurde Forschungsthema Nummer 1 in der Informatik und es entstand der Eindruck, daß ein großer Markt für solche Rechner schnell entstehen würde.

Dies führte zu zahlreichen kommerziellen Entwicklungen, die fachlich in direkter Konkurrenz zu Projekten an Universitäten standen oder stehen. Den Universitäten wurde dadurch die Show gestolen, weil die kommerziellen Entwicklungen mit größeren Ressourcen betrieben wurden. Viele kommerzielle Hersteller ihrerseits verloren Geld; Forschung und Entwicklung im Stil von Universitäten ist eben riskant.

Die überlebenden Hersteller bieten derzeit eine enorme Vielfalt von verschiedenen Architekturen an. Diese Vielfalt wäre weniger nachteilig, wenn die Sterblichkeit der Hersteller niedriger wäre oder wenn Programme zwischen den Rechnern verschiedener Hersteller portabel wären. Dies ist jedoch kaum einmal der Fall. Eine bemerkenswerte Ausnahme bilden OCCAM-Programme für Transputer-Systeme.

Die potentiellen Käufer fallen in 2 Klassen: Informatiker, die den Parallelismus an sich erforschen wollen, und Benutzer, die wirklich etwas ausrechnen wollen. Erstere sind weniger an hoher Rechenleistung als an hohen Prozessorzahlen interessiert. Sie werden also lieber für viele billige, einfache als für wenige teure, leistungsfähige

Prozessoren Geld ausgeben. Zu jedem Zeitpunkt dürfte die Anzahl der installierten mittelgoßen Systeme kleiner oder gleich der Anzahl der Informatik-Fachbereiche sein.

Für die anderen Benutzer ist Parallelismus bestenfalls ein notwendiges Übel, das man wegen überlegener Leistung oder wegen eines günstigen Preis-Leistungsverhältnisses in Kauf nimmt. Überlegene Leistung bieten - von Special Purpose Maschinen abgesehen - immer noch die klassischen Supercomputer. Workstations auf der anderen Seite bieten mehr und mehr Rechenleistung zu sehr günstigen Preisen. Parallelrechner mit vielen Prozessoren hoher Rechenleistung sind also für solche Benutzer interessant, denen eine Workstation nicht genügt und ein Supercomputer zu teuer ist. Diese Benutzer führen meistens große numerische Rechnungen durch. Sie legen erheblichen Wert auf das Preis-Leistungsverhältnis der Prozessoren. Ein günstiges Preis/Leistungsverhältnis erzielt man durch einfache Prozessoren.

Zusammenfassend kann man sagen: der Markt für Parallelrechner mit vielen Prozessoren ist derzeit nicht groß. Er besteht aus den Informatik-Fachbereichen und Gruppen von Benutzern, denen eine eigene Workstation nicht genügt. Aus verschiedenen Gründen werden Rechner mit einfachen Prozessoren bevorzugt.

2. Superlinear Speedup. Wir können der Versuchung nicht widerstehen, kurz ein Beispiel zu schildern, wo p Prozessoren P_j mehr als p mal schnell so sind wie einer. Dies scheint nicht plausibel, da doch ein einzelner Prozessor in Runden arbeiten kann und in jeder Runde r eine feste Zahl von Schritten s von Prozessor $P_{r \bmod p}$ simulieren kann. Das Zählen bis s und das Wechseln zwischen den Runden kostet jedoch Zeit, so daß man zur Simulation von s Schritten aller Prozessoren P_j etwas mehr als $sp + c$ Schritte braucht für ein $c > 0$. Hier kann c von s abhängen, falls man keinen Timer hat. Durchsuchen die Prozessoren P_j disjunkte Suchräume, so kann $\alpha = 1 + c/sp$ auftreten.

3. Klassifikation von Parallelrechnern kann auf viele Weisen betrieben werden.

3.1. Nach der Art der Speicherorganisation unterscheidet man "private memory"- und "shared memory"-Maschinen. Bei "private memory"-Maschinen hat jeder Prozessor seinen privaten Speicher, und dieses Bild wird auch dem Programmierer vermittelt. Kommunikation zwischen Prozessoren geschieht durch explizite Botschaften. Bei "shared memory"-Maschinen wird dem Programmierer das Bild von p Prozessoren vermittelt, die auf einen gemeinsamen Speicher zugreifen. Kommuniziert wird über den gemeinsamen Speicher.

Realisiert man eine "shared memory"-Maschine physikalisch so, wie der Benutzer sie sieht, wird der Speicher zum Flaschenhals. Die Heilmittel dagegen sind große private caches und die Aufteilung des Speichers in mehrere Speicherbänke, die ebenfalls heimlich still und leise den Prozessoren zugeordnet werden.

Beide Klassen von Maschinen sind also physikalisch ähnlich organisiert. Bei beiden ist der Zugriff auf Daten im Speicher eines fremden Prozessors wesentlich langsamer als auf Daten im eigenen Speicher. Im einen Fall kann und muß der Programmierer

dies direkt kontrollieren, im anderen Fall nicht.

3.2 Vernetzung. Am einfachsten ist es, alle Rechner auf einen gemeinsamen Bus zu setzen. Die Kosten sind O(p). Für große p wird der Bus zum Flaschenhals. Am komfortabelsten ist es, durch einen Crossbar jeden Rechner direkt mit jedem anderen verbinden zu können. Die Kosten sind $O(p^2)$. Zwischen diesen Extremen gibt es viele Kompromisse. In vielen Rechnern werden die Prozessoren so vernetzt wie die Ecken eines $\log_2 p$-dimensionalen Würfels. Die Kosten für die Komponenten zur Unterstützung der Kommunikation sind $O(p \log_2 p)$.

Hochdimensionale Würfel haben viele schöne graphentheoretische Eigenschaften. Da man für numerische Anwendungen typischerweise seine Prozessoren gerne in Form eines d-dimensionalen Gitters vernetzt hätte, ist folgendes entscheidend: sei Γ ein regelmäßiges d-dimensionales Gitter mit $p = g_1 * \ldots * g_d$ vielen Gitterpunkten und p eine Zweierpotenz. Dann ist Γ Subgraph eines $\log_2 p$-dimensionalen Würfels.

3.3. Kommunikation. Bei "shared memory"-Maschinen läuft die Kommunikation über den globalen Speicher. Bei "private memory"-maschinen geschieht die Kommunikation durch explizite Botschaften. Die primitivere Methode ist *synchrone* Kommunikation: will ein Prozessor einem anderen Prozessor eine Nachricht senden, so kann der Sender nicht weiterarbeiten, bis der Empfänger empfangsbereit ist. Die komfortablere Methode ist *asynchrone* Kommunikation. Der Sender deponiert die Nachricht im Briefkasten des Empfängers. Synchrone Kommunikation ist einfach zu implementieren und scheint für die überwältigende Mehrzahl numerischer Anwendungen völlig adäquat zu sein. Asynchrone Kommunikation erfordert mehr Implementierungsaufwand. Wir vermuten, daß sie allenfalls bei solchen numerischen Verfahren nützt, wo die Rechenlast *in extrem kurzen Abständen* zwischen den Prozessoren neu verteilt werden muß. Solche Anwendungen sind uns bisher nicht bekannt,

Die Übertragung von n Daten zwischen zwei Prozessoren über einen ansonsten freien Übertragungskanal kostet Zeit $S + n/B$. Die Größe S heißt *Startup-Zeit*. B ist die *Bandbreite* des Kanals. Wählt man als Einheit die Zeit, die ein Prozessor für eine arithmetische Operation braucht (Peak-Leistung ohne chaining, siehe 3.4), so liegt B heute typischerweise zwischen 1/2 und 1/10, S bis zu einigen 1000. Die hohen Startup-Zeiten entstehen durch Software-Protokolle und durch Kontext-Switching, wenn die Kommunikation zwischen Prozessoren durch Betriebssystemroutinen erledigt wird (und dadurch, daß arithmetische Operationen extrem schnell sind). Trotz teilweise nicht sehr hohen Bandbreiten und oft hohen Startup-Zeiten hört man kaum Klagen darüber, daß deswegen numerische parallele Programme ineffizient laufen. Im 4. und 5. Abschnitt werden wir versuchen, dies zu erklären.

3.4. Prozessoren. Hohe Rechenleistung ist offensichtlich wichtig. Schnelle Schaltkreise für arithmetische Operationen auf n-Bit Zahlen haben eine Tiefe O(log n). Führt man L arithmetische Operationen nacheinander aus, so kostet dies Zeit O(L*log n). Teilt man den Schaltkreis jedoch durch s Register in Stufen der Tiefe $c = O((\log n)/s)$ und clockt die Stufen gleichzeitig, so entsteht eine *Pipeline*, in der man L Operationen im günstig-

sten Fall in Zeit $O(c*(s + L)) = O(\log n + c*L)$ ausführen kann. Die Pipeline erreicht dann ihre *peak-performance* π, üblicherweise gemessen in MFLOPS (Millionen Floating Point Operationen pro Sekunde). s heißt die *Tiefe* der Pipeline.

Ungünstig ist in diesem Zusammenhang stets, wenn für irgendein j die j-te Operation einen Operanden verarbeitet, der erst in einer Operation k mit $k > j - s$ berechnet wird. Im Extremfall benutzt für alle j Operation j einen Operanden von Operation j-1. Dann hat man durch die Pipeline nichts gewonnen und die Leistung der Pipeline sinkt auf ungefähr π/s. Günstig ist natürlich die elementweise Verknüpfung von Vektoren, weshalb man Maschinen mit gepipelineter arithmetischer Einheit auch als *Vektorrechner* bezeichnet.

In Prozessoren mit mehreren arithmetischen Einheiten, z.B. ALU und Multiplizierer, kann man in gewissen Fällen beide Einheiten gleichzeitig betreiben und die Ergebnisse der einen direkt als Operanden in die andere leiten. Dies nennt man *chaining*. Als Peak-Leistung Π des Prozessors wird die Summe der Peak-Leistungen der pipelines genannt, die man chainen kann. Umgekehrt kann man bei einem Prozessor mit q pipelines der Tiefe s, die man chainen kann, in ungünstigen Fällen einen Abfall der Leistung auf $\Pi/(q*s)$ erwarten.

Abhängigkeiten zwischen Operanden sind nicht das einzige, was dem Erreichen der Peak-Performance entgegensteht. Die Versorgung mit Operanden und das Abspeichern der Ergebnisse von arithmetischen Einheiten mit hoher Leistung (gepipelined oder nicht) stellt enorme Anforderungen an die Bandbreite des Speichers. Diese Bandbreite beim Hauptspeicher bereitzustellen ist mit mehreren Nachteilen verbunden, auf die wir hier nicht eingehen. Man behilft sich mit der Einführung eines kleinen schnellen Speichers zwischen Hauptspeicher und arithmetischer Einheit. Je nach Organisation dieses Speichers spricht man von *Vektorregistern*, Cache etc.

Somit stellt die Bandbreite zwischen Haupspeicher und schnellem Speicher einen weiteren Flaschenhals für die Leistung des Prozessors dar. Je weniger Operationen man auf Daten ausführen kann, nachdem man sie einmal in den schnellen Speicher geladen hat, desto deutlicher macht sich dieser Flaschenhals bemerkbar. Für Prozessoren, deren CPU auf einem Chip realisiert ist und eine Gleitkommaeinheit besitzt, liefert dies eine wichtige Methode, die Leistung nach oben abzuschätzen: man kann nicht schneller rechnen, als man die Daten über die Pins der Datenpfade in die CPU rein und aus der CPU raus kriegt.

4. Algorithmen. Wie gut die Architektur eines Rechnersystems ist, beurteilt man unter anderem danach, wie schnell Anwendungsprogramme laufen. Anwendungsprogramme sind sehr vielfältig, aber den vielen Anwendungsprogrammen liegen vergleichsweise wenige Algorithmen zugrunde. Bei numerischen Programmen sind dies Algorithmen aus der linearen Algebra, der numerischen und höheren numerischen Mathematik, sowie einige Simulationsmethoden. Die Inspektion einer großen Menge solcher Algorithmen offenbart, was die Verarbeitung auf Parallelrechnern angeht, *typischerweise* eine

gemeinsame Struktur:

Die Algorithmen operieren auf großen, meistens zweidimensionalen Feldern (Matrizen, Gitter). Sie verlaufen in Runden (Iterationen, Eliminationsschritte). Ist N die Problemgröße, so ist die Rechenzeit einer Runde auf einem einzelnen Rechner aN für ein $a \geq 1$. Man kann die Algorithmen se auf p Prozessoren wie folgt implementieren:

Jeder Prozessor arbeitet in einer Runde zunächst den ursprünglichen seriellen Algorithmus auf einem Teilproblem der Größe $G = N/p$ in Zeit aN/p. Wir nennen die lokale Problemgöße G die *Granularität* der Implementierung. Die Prozessoren arbeiten dabei auf Teilfeldern des ursprünglichen Felds. Typischerweise wird das ursprüngliche Feld in N/p Quadrate mit Kantenlänge $\sqrt{(N/p)}$ unterteilt.

Kommunikation zwischen Prozessoren geschieht meistens in 3 Formen:

i) Austausch der Ränder mit den Nachbarn. Ist ein zweidimensionales Feld in Quadrate unterteilt, kostet dies bei Kommunikationsnetzen, in die man Gitter gut einbetten kann, Zeit $8 (S + \sqrt{(N/p)}/B)$ (1 Paar von Runden für jede Himmelsrichtung. In jeder Runde sendet die Hälfte der Prozessoren, die andere Hälfte empfängt). Für andere Kommunikationsnetze muß man mehr rechnen; bei hoher Bandbreite, geringer Startup-Zeit und $p \leq 64$ kommt man selbst für baumartig vernetzte Rechner noch auf erträgliche Zeiten.

ii) Berechnung weniger globaler Größen aus wenigen lokalen Größen durch einfache assoziative Operationen (globale Summe, Maximum). In einem Parallelrechner berechnet man dies entlang den Kanten eines binären Baums mit p Knoten. Der Baum hat Tiefe $t = \log_2 p$. Man braucht 2t Kommunikationsrunden (für jedes Niveau senden erst die linken, dann die rechten Söhne an ihre Väter). Wir vernachlässigen die Übertragungszeit und die Rechenzeit gegenüber der Startup-Zeit und veranschlagen Zeit $2S(\log_2 p)$.

iii) Broadcast weniger globaler Größen. Dies geht bei passender Hardwareunterstützung schnell und wird hier vernachlässigt.

Die *Effektivität* dieser Implementierung ist ungefähr

$$\alpha = aN/(p*(aN/p + 8 (S + \sqrt{(N/p)}/B) + 2S(\log_2 p))$$

$$= a/(a + 8Sp/N + 8 /(B\sqrt{(N/p)}) + 2S(\log_2 p)p/N)$$

$$= a/(a + 8S/G + 8 /(B\sqrt{G}) + 2S(\log_2 p)/G)$$

$$= a/(a + 8/(B\sqrt{G}) + (8 + 2 \log_2 p)S/G).$$

Hinreichend für eine Effizienz um 80 % bei sehr vielen Algorithmen ist also

(4.1) $8/(B\sqrt{G}) \leq a/10$ und $(8 + 2\log_2 p)S/G \leq a/10$.

Für hohe Effektivität sind also günstig: hohe Bandbreite, niedrige Startup-Zeit, lange Rechnungen zwischen den Kommunikationsphasen (großes a) und hohe Granularität. Effektivität ist ein Verhältnis von Rechenzeiten, in das die Rechenleistung der Prozessoren nur indirekt als Einheit zum Messen von S und B eingeht. Die Formel erklärt auch extrem hohe Effektivitäten, die bei manchen Parallelrechnern gemessen werden: bei sehr langsamen Knotenrechnern werden S und 1/B, gemessen als Vielfache der Zeit für eine arithmetische Operation, sehr klein.

5. Granularität und Effektivität.

Sei M die Größe des Speichers der Prozessoren gemessen in Megabyte. Double Precision Floating Point-Zahlen sind 8 Byte lang. Ignorieren wir den Platz des Programms, so ist im Speicher jedes Prozessors Platz für $m = M/8$ Daten. In einer Rechenrunde mit aG Rechenschritten greift man üblicherweise nicht auf Daten zu, die mehr als a Runden zurückliegen und braucht für Zwischenergebnisse nicht mehr Platz als aG. *Nutzt man den Speicher voll aus*, so kann man also $G \geq m/(2a) = M/(16a)$ erreichen. Dann liefert (4.1) die Bedingungen

$$8/(B\sqrt{(M/16a)}) = a/10 \quad \text{bzw.}$$
$$320/(B\sqrt{M}) \leq \sqrt{a} \quad \text{bzw.}$$
$$51200/B^2 \leq Ma.$$

Für $B = 1/10$ und $a = 1$ liefert dies

(5.1) $$512 * 10^4 = 5.12 * 10^6 \leq M.$$

Bedingung (4.2) liefert

$$(8 + 2\log_2 p)S/(M/16a) \leq a/10 \quad \text{bzw.}$$

$$160(8 + \log_2 p)S \leq M.$$

Für $p = 1024$ und $S = 1000$ liefert dies

(5.2) $$16 * 28 * 10^4 = 4.48 * 10^6 \leq M.$$

Für $S = 2000$ genügt $8.96 * 10^6 \leq M$.

Die obigen Abschätzungen lassen sich wie folgt zusammenfassen: Hat jeder Prozessor einen genügend großen lokalen Speicher (4-8 Mbyte), und man nutzt diesen Speicher auch aus, dann kann man selbst bei hohen Startup-Zeiten (1000 - 2000) und geringer Bandbreite (1/10) bei parallelen numerischen Algorithmen typischerweise gute Effizienz (≥ 0.8) erzielen. Anders ausgedrückt: parallele numerische Algorithmen sind bezüglich ihrer Anforderungen an die Leistungsfähigkeit des Kommunikationsmechanismus sehr gutmütig. Solange die lokalen Speicher groß genug sind und man diese auch ausnutzt,

erreicht man fast mit jedem Mechanismus hohe Effizienz.

6. Zusammenfassung. Die Leistung von Parallelrechnern für numerische Algorithmen wird nur bei extrem ineffizienten Komunikationsmechanismen vom Kommunikationsnetz beeinflußt. Wesentlich ist das Preis/Leistungsverhältnis der Prozessoren. Die Prozessoren müssen großen lokalen Speicher haben. Man soll nicht so viele Prozessoren kaufen, daß man bei seinen Anwendungen die Speicher der Prozessoren nur zu einem geringen Teil ausnutzt.

Appendix. Obwohl keine konkreten Rechner genannt wurden, sind Beziehungen zu existierenden oder angekündigten Produkten nicht zufällig.

Preise, Peak-Leistung mit und ohne chaining und Speichergröße der Prozessoren entnimmt man den Angeboten der Hersteller, S und B in der Regel nicht. Das ist aber kein Problem.

Sei P die Peak-Performance eines Knotenrechners in MFLOPS. Dann ist $\tau = (1/P)10^{-6}$ sec die Einheit, in der S und B gemessen werden. S und B bestimmt man durch zwei einfache Benchmark-Programme:

Wähle zwei Prozessoren P_j und P_k in verschiedenen Schränken des Parallelrechners.

timer an;
for i = 1 to K do
sende 1 Datum von P_j and P_k;
timer aus;
S = timer/(Kτ)
(Overhead für Schleifenverwaltung wird wegen der Größe von S vernachlässigt).

Nun konfiguriert man die Prozessoren des Parallelrechners als Gitter, wählt eine Himmelsrichtung R und teilt die Prozessoren so in zwei gleichgroße Gruppen A und B, daß jeder Prozessor in Gruppe A in Richtung R genau einen Nachbarn in Gruppe B hat (z.B. R = Norden, A = die Prozessoren auf Zeilen des Gitters mit gerader Nummer), und läßt das folgende Programm laufen.

timer an;
für alle Prozessoren in A:
schicke 10 000 Daten an den Nachbarn in Richtung R.

Für alle Prozessoren in B:
empfange 10 000 Daten aus Richtung R^{-1} ($Norden^{-1}$ = Süden).
timer off;
B = 10 000/(timer - S)

Benchmarken eines einzelnen Prozessors mit den Livermoore Loops gibt einen guten Eindruck von der Leistung des Prozessors in vielen Situationen - und von der Stabilität des Compilers.