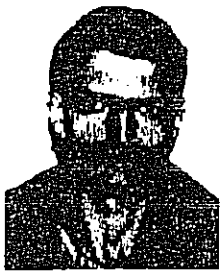
D. Hinz und W.J. Paul, Fachbereich Informatik, Universität des Saarlandes

Über Parallelrechner für numerische Anwendungen und ihre Programmierung

Hinz, Didier, Studium der Physik und Mathematik an der Universität des Saarlandes. Seit 1986 wissenschaftlicher Mitarbeiter am Lehrstuhl für Rechnerarchitektur und parallele Rechner, Saarbrükken. Arbeitsgebiet: Hardware und Programmierung paralleler Rechner, Entwicklung von Compilern für parallele Rechner.



Paul, Wolfgang J., Dr. rer. nat. an der Universität des Saarlandes 1973, Mathematikprofessor in Bielefeld von 1976–1982, Research Staff Member bei IBM Research in San Jose von 1982–1986, seit 1986 Professor für Informatik an der Universität des Saarlandes, spezielle Interessen: Rechnerarchitektur.

Es werden einige Beispiele numerischer Anwendungen für massiv parallele Rechner gegeben. Eine Klasse paralleler Algorithmen wird identifiziert, mit der man Anwendungen dieser Art einfach und effizient bearbeiten kann. Eine Rechnerarchitektur, welche diese Algorithmen unterstützt, und einige Implementierungen dieser Architektur werden beschrieben. Schließlich wird eine einfache Erweiterung von FORTRAN skizziert, mit der die Algorithmen formuliert werden können.

1 Numerische Anwendungen

Eine typische numerische Anwendung von Hochleistungsrechnern ist die Simulation der Natur, wobei die Natur durch Systeme partieller Differentialgleichungen beschrieben wird. Diese Systeme werden dann numerisch gelöst.

In Bild 1 mögen die Massenpunkte Moleküle sein, die sich in einer Ebene bewegen oder Sterne im Weltraum. Zwei Moleküle oder Sterne M_i und M_j ziehen sich mit einer Kraft F_{ij} an, die eine Funktion ihres Abstands r_{ij} ist. Fällt diese Funktion schnell in Abhängigkeit von r_{ij}, etwa wie $1/r^7$ bei Molekülen in einem Lenard-Jones Potential, so spricht man von Nahwirkungskräften. Fällt die Funktion langsam, etwa nur wie $1/r^2$ bei Sternen im Gravitationsfeld, so spricht man von Fernwirkungskräften. Die gesamte auf M_i ausgeübte Kraft F_i ist die vektorielle Summe der F_{ij}, $j \neq i$. Die Newtonsche Bewegungsgleichung (Kraft = Masse * Beschleunigung) liefert hierzu die Beschleunigung von M_i. Kennt man zu einem Zeitpunkt die Positionen x_i und die Geschwindigkeiten v_i, der M_i's, so kann man durch Lösen der Newtongleichungen für die M_i's mit diesen Anfangsbedingungen den weiteren Bewegungsablauf der M_i's gewinnen (und etwa Sonden zum Jupiter schießen).

Die exakte Lösung großer Systeme von Differentialgleichungen zu suchen ist im allgemeinen ein hoffnungsloses Unterfangen. Man behilft sich daher mit numerischen Methoden. Oft hilft die folgende bestechend einfache Idee (siehe Bild 2). Die Ableitung $f'(x)$ von f an der Stelle x ist der Grenzwert der Steigung der Sekante durch die Punkte $P_0 = (x, f(x))$ und $P_1 = (x + \Delta x, f(x+\Delta x))$ für $\Delta x \to 0$, und das ist gerade die Steigung der Tangente an f im Punkt P_0. Die Grenzwertbildung ist zu schwierig. Deshalb macht man Δx sehr klein und tut so, als ob Kurve, Sekante und Tangente zwischen x und $x + \Delta x$ übereinstimmen würden. Dabei macht man natürlich einen Fehler. Je kleiner man Δx macht, desto kleiner wird der Fehler, und desto mehr muß man rechnen. Dies erklärt den Hunger der Benutzer solcher Methoden nach Rechenleistung.

Mit dieser Idee vor Augen kann man sofort das Euler-Verfahren zur numerischen Lösung von Differentialgleichungen verstehen. In Bild 3 ist die numerische Lösung der Differentialgleichung $f'(x) = f(x)$ mit Anfangswert $f(0) = 1$ mit $\Delta x = 1$ dargestellt. Der Anfangswert $f(0) = 1$ liefert die Ableitung $f'(0) = 1$ der gesuchten Funktion f an der Stelle 0. Dies liefert die Steigung der Tangente im Punkt (0,1). Man tut so, als ob zwischen 0 und 1 Sekante und Tangente übereinstimmen und erhält als Approximation $f(1) = 2$. Hieraus liefert die Differentialgleichung die Steigung der Tangente im Punkt (1,2), nämlich $f'(1) = f(1) = 2$. Man tut wieder so, als ob Sekante und Tangente zwischen 1 und 2 übereinstimmen und erhält $f(2)=4$, usw. Macht man das gleiche Spielchen mit der Schrittweite 1/10, so kommt die numerische Lösung der in diesem einfachen Fall bekannten exakten Lösung (nämlich $f(x)=e^x$) schon erheblich näher.

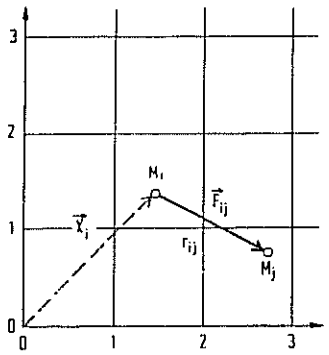


Bild 1. Massenpunkte M_i, M_j im Gravitationsfeld (2-dim.)

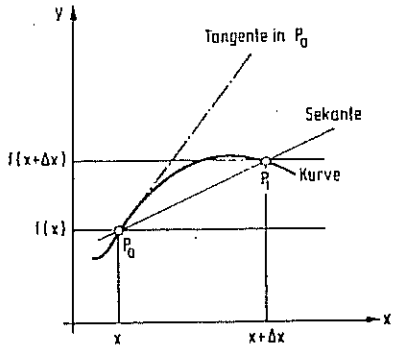


Bild 2. Beziehung zwischen Kurve, Tangente und Sekante einer Funktion

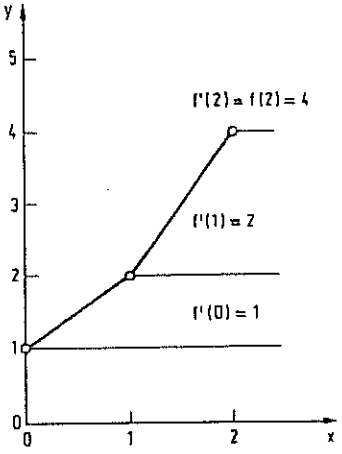


Bild 3. Lösung von f'(x) = f(x) mit Startwert f(0) = 1 (Euler-Verfahren)

Das Verfahren des letzten Abschnitts kann man nun verwenden, um Systeme von Newtongleichungen für Tausende von M_i's zu lösen [2]. Hat man bereits die Konfiguration des Gesamtsystems zum Zeitpunkt t approximiert, so kann man die Konfiguration zum Zeitpunkt t+Δt wie folgt berechnen:

1.1 Berechnung der Kräfte

Für alle i berechne K_{ij} aus dem Ort x_i von M_i und den Orten x_j der M_j. Hat man es mit Nahwirkungskräften zu tun, kann man sich hierbei auf M_j's beschränken, die gerade in der Nähe von M_i sind.

1.2 Berechnung der neuen Orte

Die neuen Ortsvektoren gewinnt man aus den alten Orts- und den alten Geschwindigkeitsvektoren. Geschwindigkeit ist Ableitung des Ortes nach der Zeit und ändert sich laufend. Man tut nun so, als bliebe die Geschwindigkeit zwischen den Zeitpunkten t und t+Δt konstant, und gewinnt für jedes i den neuen Ort von M_i, indem man zum alten Ort x_i einen Vektor der Länge $v_i * \Delta t$ in Richtung v_i hinzuaddiert.

1.3 Berechnung der neuen Geschwindigkeiten

Für jedes i gewinnt man aus der Kraft K_i den aktuellen Beschleunigungsvektor b_i, indem man K_i durch die Masse von M_i teilt. Beschleunigung ist die Ableitung der Geschwindigkeit nach der Zeit und ändert sich laufend. Man tut abermals so, als bliebe die Beschleunigung zwischen den Zeitpunkten t und t+Δt konstant und gewinnt für jedes i den neuen Geschwindigkeitsvektor, indem man zur alten Geschwindigkeit v_i einen Vektor der Länge $b_i * \Delta t$ in Richtung b_i hinzuaddiert.

Die Durchführung von (1.1) bis (1.3) für alle i nennt man einen *Iterationsschritt*.

Das folgende Beispiel ist besonders einfach [4]. Eine Saite sei zwischen den Punkten x=0 und x=L fest eingespannt. Es bezeichne U(x,t) die Auslenkung der Saite zum Zeitpunkt t an der Stelle x. Zum Zeitpunkt 0 sei die Saite an der Stelle x um U(x,0) = f(x) ausgelenkt und in Ruhe. Läßt man die Saite los, so erfüllt die Auslenkung U(x,t) der Saite zum Zeitpunkt t an der Stelle x die Wellengleichung

$$\delta^2U/\delta t^2 = \delta^2U/\delta x^2.$$

Zweite Ableitungen sind erste Ableitungen von ersten Ableitungen. Wenn man wieder so tut, als stimmten Sekante und Tangente überein, so erhält man die folgenden Approximationen:

$$\delta U(x,t)/\delta t = [\, U(x,t+\Delta t) - U(x,t) \,] / \Delta t$$

$$\delta U(x,t+\Delta t)/\delta t = [\, U(x,t+2\Delta t) - U(x,t+\Delta t) \,] / \Delta t$$

$$\delta^2U(x,t)/\delta t^2 = [\, \delta U(x,t+\Delta t)/\delta t - \delta U(x,t)/\delta t \,] / \Delta t$$
$$= [\, U(x,t+2\,\Delta t) - 2U(x,t+\Delta t) + U(x,t) \,] / (\Delta t)^2$$

$$\delta^2U(x,t)/\delta x^2 = [\, U(x+\Delta x,t+\Delta t) - 2U(x,t+\Delta t) + U(x-\Delta x,t+\Delta t) \,] / (\Delta x)^2$$

Einsetzen in die Differentialgleichung und Auflösen nach U(x,t+2Δt) liefert:

$$U(x,t+2\Delta t) = 2U(x,t+\Delta t) - U(x,t) + \mu^2 [U(x+\Delta x,t+\Delta t) - 2U(x,t+\Delta t) + U(x-\Delta x,t+\Delta t)]$$
$$\text{mit } \mu = \Delta t/\Delta x \qquad (2.5)$$

Daß die Saite anfangs in Ruhe ist, kann durch U(x,0) = U(x,Δt) = f(x) beschrieben werden. Zu den Zeitpunkten t = 0 und t = Δt kennt man damit U(x,t). Hat man U an den Stellen x = Δx, 2Δx, ..., L−Δx für die Zeiten t und t+Δt bereits approximiert, kann man durch Auswerten von (2.5) U für die gleichen x-Werte zum Zeitpunkt t+2Δt bestimmen. Das Auswerten von (2.5) für einen t-Wert und alle genannten x-Werte nennt man wieder einen Iterationsschritt.

2 Parallelisierung

In der Natur läuft vieles gleichzeitig ab. Bei Simulationen spiegelt sich das darin wider, daß man verschiedene Teile der Natur, z.B. verschiedene Dezimeter einer Saite, auf verschiedenen Rechnern gleichzeitig simulieren kann. Die Standardmethode hierfür ist, den zu simulierenden Teil der Natur in Räume aufzuteilen. Diese Einteilung ist in Bild 1 bereits angedeutet. Simuliert man das Weltall, werden die Räume 3-dimensional sein. Simuliert man Moleküle in einer ebenen Fläche, werden die Räume 2-dimensional sein, und simuliert man eine Saite, werden sie 1-dimensional sein. Man teilt die Natur in soviele Räume ein, wie man Rechner hat, jeder Rechner ist für einen Raum zuständig und kennt den Teil der Natur in seinem Raum. Im Folgenden wird skizziert, wie in den einzelnen Beispielen ein Iterationsschritt auf einem Parallelrechner durchgeführt werden kann.

Eine 1 m lange Saite werde etwa von einem 10-Prozessor-Rechner simuliert. Es sei $\Delta x = 1$ mm. Dann ist jeder Raum 1 dm lang und für $i = 0, ..., 9$ simuliert Prozessor P_i die Saite an den Millimetern 100 i, ..., 100 i + 99. Ein Iterationsschritt wird in drei Runden durchgeführt:

1: Für $i \neq 9$ sendet jeder Prozessor P_i seinem Nachbarn P_{i+1} zur Rechten die Auslenkung $U(100i + 99, t+\Delta t)$ der Saite am rechtesten Millimeter seines Raums nach dem letzten Iterationsschritt.

2: Für $i \neq 0$ sendet jeder Prozessor P_i seinem Nachbarn P_{i-1} zur Linken die Auslenkung $U(100i,t+\Delta t)$ der Saite am linkesten Millimeter seines Raums nach dem letzten Iterationsschritt.

3: Nun kann jeder Prozessor die Gleichung (2.5) für die Millimeter in seinem Raum auswerten.

In Schritt drei führt jeder Prozessor im wesentlichen das Programm aus, das ein einzelner Prozessor ausführen würde, wenn er allein die ganze Saite zu simulieren hätte. Da die Laufzeit hier proportional zur Anzahl der simulierten Millimeter ist und da jeder der 10 Prozessoren nur einen Dezimeter zu simulieren hat, gewinnt man an dieser Stelle gegenüber einem einzelnen Rechner einen Faktor 10 an Geschwindigkeit. Ist C die Zeit, die ein einzelner Prozessor für eine Iteration braucht und K die Dauer der Schritte zwei und drei, so dauerte eine Iteration auf einem 10-Prozessor-Rechner hier C/10 + K.

Simuliert man die Saite statt an 1000 Punkten an n Punkten und statt 10 Prozessoren mit p Prozessoren, so ist $C = O(n)$ und eine Iteration auf dem Parallelrechner dauert C/p + K. Mehr als ein Faktor p an Geschwindigkeit kann man durch Einsatz von p Prozessoren gegenüber einem einzelnen Prozessor nicht gewinnen. Damit der tatsächliche Gewinnfaktor nah an der theoretischen Schranke liegt, muß K klein sein im Vergleich zu C/p. Die Kommunikationskosten K kann man konstant halten, wenn man die Rechner auf die naheliegende Weise verbindet (P_i mit P_{i-1} und P_{i+1}). C/p wird groß, indem man die Prozessorzahl klein macht im Verhältnis zur Problemgröße n *(Coarse Grain Parallelismus)*.

Eine Iteration bei der Simulation von n Molekülen mit Nahwirkungskräften durch p Prozessoren kann man wie folgt organisieren: Man macht die Räume quadratisch, ordnet sie in einem Schachbrettmuster an und indiziert sie mit einem Doppelindex (x,y). Prozessor P_{xy} simuliert zu jedem Zeitpunkt diejenigen M_i's, die sich gerade im Raum (x,y) befinden. Man macht durch Wahl des Verhältnisses von n zu p die Seitenlänge der Räume groß im Vergleich zur Reichweite der Nahwirkungskräfte. Jede Iteration verläuft nun in fünf Runden.

1: Jeder Prozessor P_{xy} sendet seinem Nachbarn $P_{x,y+1}$ im Norden die neuen Positionen x_i von denjenigen M_i's, die sich vor der letzten Iteration in der Nähe der Nordkante seines Raums befanden.

2: Jeder Prozessor P_{xy} sendet seinem Nachbarn im Süden $P_{x,y-1}$ die neuen Positionen x_i von denjenigen M_i's, die sich vor der letzten Iteration in der Nähe der Südkante seines Raums befanden.

3: Jeder Prozessor P_{xy} sendet seinem Nachbarn $P_{x+1,y}$ im Osten die neuen Positionen x_i von denjenigen M_i's die sich vor der letzten Iteration in der Nähe der Ostkante seines Raumes oder der Nordostecke des Raums seines Nachbarn im Süden oder der Südostecke seines Nachbarn im Norden befanden.

4: Jeder Prozessor P_{xy} sendet seinem Nachbarn $P_{x-1,y}$ im Westen die neuen Positionen x_i von denjenigen M_i's, die sich vor der letzten Iteration in der Nähe der Westkante seines Raumes oder der Südwestecke des Raums seines Nachbarn im Norden oder der Nordwestecke des Raums seines Nachbarn im Süden befanden.

Hat sich in irgendeiner dieser Runden ein M_i vom Raum (x,y) in einen benachbarten Raum bewegt, so sendet P_{xy} zusätzlich die Geschwindigkeit v_i, und führt fortan M_i nicht mehr in seinem Raum. Hat sich ein M_i von einem benachbarten Raum in Raum (x,y) bewegt, so führt P_{xy} M_i fortan in seinem Raum.

5: Jeder Prozessor führt eine Iteration für die M_i's in seinem Raum durch.

Sei C wieder die Laufzeit, die ein einzelner Rechner für eine Iteration braucht. Dann ist $C = O(r^2n)$, wobei r die Reichweite der Nahwirkungskräfte ist. Die Moleküle tendieren dazu, sich gleichmäßig über die Räume zu verteilen. Somit hat jeder Raum stets etwa n/p Moleküle, und die Laufzeit für Schritt 5 ist ziemlich genau C/p. Sei S eine Kante eines Raums mit Kantenlänge L. Dann hat im Mittel der r/L-te Teil der Moleküle im Raum einen Abstand von r oder weniger von S. Typischerweise sendet ein Prozessor also während einer Runde $O(rn/Lp)$ Daten. Dies ist zu vergleichen mit $C/p = O(r^2 n/p)$. Die Dauer einer Runde wird allerdings von dem Prozessor mit den meisten Daten am Rand seines Raums diktiert.

Die naheliegende Vernetzung der Prozessoren ist hier ein 2-dimensionales Gitter. Studiert man ein ähnliches 3-dimensionales Problem, ist die naheliegende Vernetzung ein 3-dimensionales Gitter.

Simuliert man ein Problem mit Fernwirkungskräften, so genügt es nicht mehr, nur die Prozessoren in benachbarten Räumen über die neuen Positionen der M_i's in einem Raum zu informieren. Vielmehr müssen die neuen Positionen eines jeden Raums in p-1 Runden von Kommunikation an jeden anderen Raum weitergegeben werden. Dort kann dann ihr Beitrag zu den jewei-

ligen Kräftekalkulationen berücksichtigt werden. Sind die M_i's gleichmäßig verteilt, so befinden sich wieder in jedem Raum etwa n/p M_i's. Man hat es also in einer Iteration mit p-1 Runden der Dauer O(n/p) zu tun. Nach jeder Runde innerhalb einer Iteration werden allerdings in jedem Prozessor $O(n^2/p)$ Summen gebildet. Der Overhead für die Kommunikation wird also auch hier asymptotisch schnell klein.

Damit dies funktioniert, ist es übrigens gleichgültig, wie man die Prozessoren den Räumen zuordnet, und es genügt, die Prozessoren ringförmig zu vernetzen [5].

Die Diskussion dieses Abschnitts hat gezeigt, daß sich mit Coarse-Grain Parallelismus die betrachteten Anwendungen so implementieren lassen, daß der Overhead für Kommunikation zwischen den Rechnern gering ist im Vergleich zur totalen Rechenzeit. Damit ist allerdings nicht nachgewiesen, daß man mit Coarse-Grain Parallelismus bei diesen Anwendungen wirklich billiger fährt als mit Fine-Grain Parallelismus. Sind die Fine-Grain Prozessoren nämlich nur billig genug im Vergleich zu den Coarse-Grain Prozessoren, so kann man es sich auch leisten, die Hälfte der Rechenzeit mit Kommunikation zu vertun.

Zum Abschluß dieses Abschnitts noch einige Beobachtungen:

1: Die parallelen Algorithmen, die beschrieben wurden folgen einem ganz simplen Schema. Es gibt zwei Sorten von Runden: *Compute*-Runden, in denen alle Rechner autonom rechnen, bis der letzte fertig ist, und *Communicate*-Runden, In denen alle Rechner anderen Rechnern Daten senden und von anderen Rechnern Daten empfangen. Welche Rechner in welcher Runde miteinander kommunizieren, ist von vornherein bekannt und kann durch ein Gitter oder einen Ring beschrieben werden. Auch Communicate Runden dauern, bis der letzte Rechner seine Daten empfangen hat.

2: Das oben genannte Schema beschreibt nicht nur die genannten drei Anwendungen, sondern alle den Autoren zur Zeit bekannte numerische Anwendungen auf irgendwelchen Parallelrechnern lassen sich ohne nennenswerten Effizienzverlust in einen Computer-Communicate Algorithmus umformen.

3: Die Kontrolle des Ablaufs der einzelnen Runden ist trivial. Jeder Prozessor Pi setzt ein Signal f auf 1, wenn er mit seiner Arbeit in dieser Runde fertig ist. Ist das logische UND dieser Signale gleich 1, wird ein Signal „nächste Runde" an alle Rechner gesendet.

4: Beim Start der Kommunikationsrunden laufen keine Protokolle zwischen Sendern und Empfängern ab. Man erhält Startup-Zeiten von einigen Zig Mikrosekunden.

3 Kommunikationsnetze

In diesem Abschnitt werden zwei Kommunikationsnetze beschrieben, die in flexibler Weise in Gitter und Ringe konfiguriert werden können.

3.1 Hypercube

Seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ ungerichtete Graphen. Das Kreuzprodukt $G_1 \times G_2$ der Graphen G_1 und G_2 hat Knotenmenge $V_1 \times V_2$ und Kantenmenge

$$\{((v_1,v_2),(v_1,v_2')); (v_2,v_2')EE_2, v_1EV_1\} \cup$$
$$\{((v_1,v_2),(v_1',v_2)); (v_1,v_1')EE_1, v_2EV_2\}$$

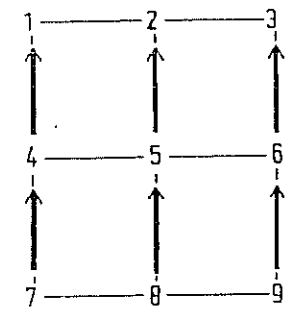
Beispielsweise ist ein NxM-Gitter das Produkt eines Kantenzugs aus N-1 Kanten mit einem Kantenzug aus M-1 Kanten. Das Produkt eines Kantenzugs mit einem Kreis gibt einen Zylinder, und das Produkt eines n-dimensionalen Würfels mit einem m-dimensionalen Würfel gibt einen (n+m) – dimensionalen Würfel. Würfel mit Dimension größer als 3 nennt man auch Hypercubes. Man beweist leicht durch Induktion über n, daß man Kantenzüge der Länge 2^n-1 in n-dimensionale Würfel einbetten kann. Bezeichnet man in der üblichen Weise die Knoten des Würfels mit den 0/1-Folgen der Länge n und folgt man dann diesem Kantenzug, so durchläuft man die Knoten gerade in der Reihenfolge des Gray-Codes.

Nun folgt sofort durch Induktion über k: Sind $n_1, ..., n_k$ ganze Zahlen und ist $n = n_1 + ... + n_k$, so kann man ein $2^{n_1} \times ... \times 2^{n_k}$ Gitter in einen n-dimensionalen Würfel einbetten.

Hypercubes kommen als Kommunikationsnetz in vielen Parallelrechnern vor, zum Beispiel im CAL-TECH Cosmic Cube, in der T-Serie von FPS und in der Connection Machine.

3.2 Permutationsnetze

Ein n-Permutationsnetz hat Ein- und Ausgänge für n Prozessoren $P_1, ..., P_n$. Zu gegebener Permutation Π der Zahlen von 1 bis n muß man das Netz so konfigurieren können, daß für alle i gleichzeitig Prozessor P_i zu Prozessor $P_{\Pi(i)}$ Daten senden kann. In Bild 4 ist an einem kleinen Beispiel dargestellt, wie man Permutationsnetze zur Simulation von Gittern verwenden kann. Will man etwa in diesem Beispiel von jedem Prozessor aus Daten an den Nachbarn des Prozessors im Norden senden, so hat man das Kommunikationsnetz nur auf irgendeine Permutation einzustellen, die 4 auf 1, 5 auf 2, 6 auf 3, 7 auf 4, 8 auf 5 und 9 auf 6 abbildet. Stellt man das Netzwerk nacheinander auf die Permutationen Norden, Süden, Osten und Westen, so hat dadurch jeder Prozessor die Möglichkeit, jedem seiner Nachbarn im Gitter Daten zu senden.



TT_N (1, 2, 3, 4, 5, 6, 7, 8, 9)
= (7, 8, 9, 1, 2, 3, 4, 5, 6)

Bild 4. Norden-Permutation im 2-dimensionalen Gitter

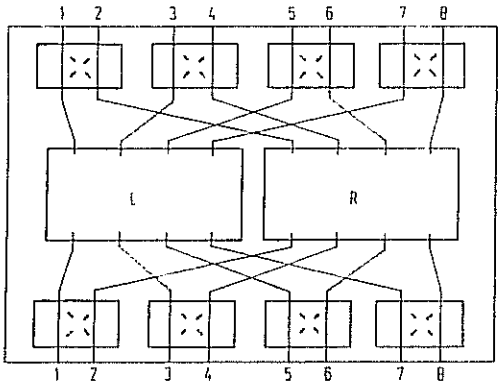


Bild 5. Konstruktion eines 8-Benes-Netzwerkes aus zwei 4-Benes-Netzwerken und zweimal vier 2-Benes-Netzwerken

Fällt im obigen Beispiel irgendein Prozessor – etwa Prozessor 5 – aus, und man hat noch einen Reserveprozessor 10 am Netz, so ist es sehr leicht, das System vom letzten Checkpoint aus warm zu starten. Prozessor 10 übernimmt die Arbeit von Prozessor 5. Zur Kontrolle des Netzwerks sind nur in jeder der 4 Permutationen zwei Werte zu ändern. In der „Nord"-Permutation zum Beispiel wird nun 8 auf 10 und 10 auf 2 abgebildet. Nach dieser Korrektur, die automatisch durchgeführt werden kann, läuft das System mit unverminderter Leistung weiter.

Die klassische rekursive Konstruktion von n-Permutationsnetzen aus zwei (n/2)-Permutationsnetzen L und R und zweimal n/2 2-Permutationsnetzen nach Benes [3] ist in Bild 5 zu finden. Für den Nachweis der Korrektheit dieser Konstruktion hat man zu beliebiger Permutation Π und zu jedem Eingang i anzugeben, ob man den Weg von Eingang i nach Ausgang $\Pi(i)$ über L oder über R legt. Dabei muß man genau die Hälfte der Wege über R und die andere Hälfte über L legen, und man muß die folgenden Restriktionen erfüllen:

4.1 Ist i ungerade und j=i+1, so dürfen die Wege von den Eingängen i und j nicht beide über L und nicht beide über R gehen, da sie sonst an einem Eingang von L bzw. R kollidieren würden.

4.2 Ist $\Pi(i)$ ungerade und $\Pi(j) = \Pi(i)+1$, so dürfen die Wege von den Eingängen i und j nicht beide über L und nicht beide über R gehen, da sie sonst an einem Ausgang von L bzw. R kollidieren würden.

Aus 4.1 und 4.2 gewinnt man wie folgt den Konfliktgraphen zur Permutation Π. Knoten des Graphen sind die Eingänge 1,...,n. Dürfen Eingänge i und j nicht beide über L und nicht beide über R gehen wegen (4.1), so verbindet man sie mit einer roten Kante. Ist dies wegen (4.2) verboten, so verbindet man sie mit einer schwarzen Kante. Dann endet in jedem Punkt des Konfliktgraphen genau eine rote und genau eine schwarze Kante. Also hat jeder Knoten Grad 2 und der Graph besteht aus einem oder mehreren disjunkten Kreisen. Diese Kreise haben gerade Länge, da die Kanten abwechselnd rot und schwarz gefärbt sind. Man routet, indem man auf jedem Kreis mit einem beliebigen Punkt i beginnt und den von i ausgehenden Weg beliebig über R oder L legt. Dann marschiert man in beliebiger Richtung über den Kreis, auf dem i liegt, und legt die Wege, die von den Eingängen j auf diesem Kreis ausgehen, abwechselnd über L und über R. Wenn man wieder bei i ankommt, kriegt man keinen Widerspruch zum ursprünglichen Routing des von i ausgehenden Wegs, da der Kreis gerade Länge hat.

Die Tiefe des oben angegebenen Permutationsnetzes ist $2\log_2 n - 1$. Für n = 256 ergibt dies eine Tiefe von 15, was bei realen Netzen der Startup-Zeit nicht gut tut und – schlimmer noch – zu einer großen Menge von Leitungen führt. Glücklicherweise funktioniert die obige Divide-and-Conquer-Konstruktion nicht nur beim Teilen des Problems in k = 2 Teilprobleme, sondern für beliebiges k [8]. Man benutzt dann einen kxk-Crossbar als elementares Schaltelement und erhält ein Netzwerk mit Tiefe $2\log_k n - 1$. Für n = 256 und k = 16 ergibt sich das 3-stufige Netz aus Bild 6.

Ein Gate-Array mit einem 16×16 Crossbar wurde am Lehrstuhl des zweiten Autors von J. Sauermann entworfen. Zur Zeit werden Prototypen dieses Chips gefertigt. Je 3*16 Chips bilden eine Bitebene des geplanten Netzwerks. Es sind 16 Daten-Bitebenen, 2 Kontroll-Bitebenen (data valid und transmission complete) sowie eine Parity-Bitebene geplant. Man braucht für 256 Prozessoren insgesamt 19*48 = 912 dieser Chips im Netzwerk, d.h. ca. 4 Chips pro Prozessor. Das Netz soll mit ca. 12 MHz betrieben werden. Man erhält eine Bandbreite pro Prozessor von 12*(16/8) = 24 Mbyte/s pro Prozessor zum Senden und gleichzeitig weitere 24 Mbyte/s pro Prozessor zum Empfangen von Daten. Das Netz wird nach [10] so aufgebaut, daß man auf jeder Bitebene nach Ausfall eines Netzwerk-Chips das Routing so ändern kann, daß das Netz mit voller Leistung weiterbetrieben werden kann. Hierbei nutzt man die Freiheiten beim Routing aus, die einem der Korrektheitsbeweis für das Benes-Netzwerk läßt.

Es ist geplant, an dem eben beschriebenen Netz ca. 100 Mikrocomputer mit MC68020/68881-Prozessoren zu betreiben, und zwar zunächst in einem leicht verallgemeinerten Compute-Communicate Modus. Im gleichen Modus arbeitet der Parallelrechner im SPARK-Projekt bei IBM-research in San Jose [1,2], aus dem das Saarbrükker Projekt hervorgegangen ist und mit dem weiter kooperiert wird. In San Jose wird zur Zeit eine Maschine mit 10 allerdings sehr schnellen Bit-Slice Prozessoren mit Gleitkomma-Pipeline aufgebaut. Weitere Maschinen mit Permutationsnetz sind GF(11) bei IBM-research in Yorktown Heights und die Rechner von Butterfly Machines.

4 Programmierung

Zum Formulieren von parallelen Algorithmen, die dem Compute-Communicate-Schema folgen, sollte es genügen, Fortran um einige ganz einfache und leicht compilierbare Konstrukte zu erweitern. Das wird ungefähr so aussehen [9]:

1. Netzwerk Deklarationen. Sie erlauben es dem Benutzer, virtuelle Netze zu definieren, die dann vom Compiler in ein reales Netz eingebettet bzw. auf einem realen Netz geroutet werden müssen. Netzwerk-Deklarationen

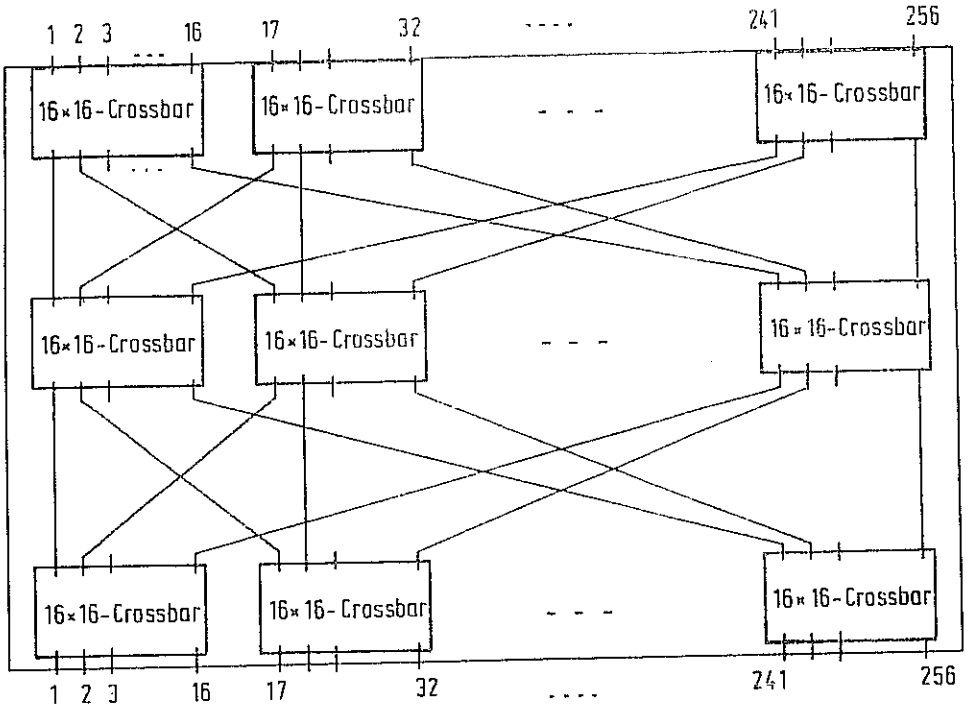


Bild 6. 3-stufiges 256-Permutationsnetz aus 48 16×16-Crossbars

spezifizieren den Namen des Netzes, die Menge der Knoten des Netzes, die Menge der Richtungen, in denen im Netz Kanten von Knoten ausgehen können, sowie, für jeden Knoten v und jede Richtung d im Netz, den Nachbarn von v in Richtung d. Die Deklaration eines 16 × 16 Gitters mit Name GRID sieht dann etwa so aus:

```
DEFNET GRID(i=0:15;j=0:15)
        /*Definition von Name und Knotenmenge */
DIR = (N,S,O,W)          /*Definition der Richtungen*/
NEIGHBOR(i,j) = (N: if j < 15 then (i, j+1) else NIL
                          /*Definition der Nachbarn*/
                 S: if j > 0 then (i, j−1) else NIL
                 O: if i < 15 then (i+1,j) else NIL
                 W: if i > 0 then (i−1,j) else NIL
ENDEF
```

2. Netzwerk-Aufrufe. Der Benutzer kann vom seriellen Teil des Programms aus spezifizieren, welches der zuvor spezifizierten Netze er benutzen will. Das oben deklarierte Netz kann man zum aktuellen Netz machen mit:

NET = GRID

Mit einem Netzwerk-Aufruf wird jedem Knoten v des Netzwerks ein Prozessor P(v) zugeordnet.

3. Parallele Teile des Programms. Sie werden nur vom seriellen Teil des Programms aufgerufen und werden nicht geschachtelt. Sie sind eingeschlossen in ein Klammerpaar PARBEGIN, PAREND. Ein paralleler Teil des Programms besteht aus einer Folge von Deklarationen und Anweisungen. Diese Folge wird bei Eintritt in einen parallelen Teil von sämtlichen Prozessoren P(v) gleichzeitig abgearbeitet.

3.1 Deklarationen innerhalb eines parallelen Teils erzeugen auf jedem Prozessor lokale Variablen oder Arrays. Nach Verlassen des parallelen Teils behalten die lokalen Variablen ihren Wert. Man kann auf sie vom seriellen Teil des Programms aus zugreifen, und man findet sie nach Wiedereintritt in einen parallelen Teil wieder vor.

3.2 In einem parallelen Teil können die Identifier, die zur Definition der Knotenmenge des aktuellen Netzes gedient haben (also im obigen Beispiel i und j) als Konstanten benutzt werden. Sie haben für jeden Prozessor einen anderen Wert, und zwar für Prozessor P(v) gerade den Wert v. Möchte man also beispielsweise alle Prozessoren des oben definierten Gitters außer dem Prozessor in der nordwestlichen Ecke ihre lokale Variable x um 1 erhöhen lassen, so schreibt man innerhalb eines parallelen Teils:

if (i,j) ≠ (0,0) then x = x+1

4. Kommunikationsrunden zwischen den parallelen Prozessoren. Sie werden in einem parallelen Teil durch SEND- und RECEIVE-Anweisungen eingeleitet. Bei einer SEND-Anweisung muß für jede Richtung d im Netzwerk ein lokales Array A(d) oder eine lokale Variable V(d) spezifiziert sein. Eine SEND-Anweisung muß von einer RECEIVE-Anweisung gefolgt sein. Bei einer RECEIVE-Anweisung muß ebenfalls für jede Richtung ein lokales Array A'(d) oder eine lokale Variable V'(d) spezifiziert werden.

Ein Paar von SEND/RECEIVE-Anweisungen wird erst dann ausgeführt, wenn alle Prozessoren P(v) eine SEND-Anweisung erreicht haben. Dann findet für jede Richtung d eine Communicate-Runde statt. Während der Runde für Richtung d verschickt jeder Prozessor sein lokales Array A(d) bzw. die Variable V(d) und speichert empfangene Daten in A'(d) bzw. V'(d). Erlaubt man in jedem parallelen Teil nur ein einziges Paar von SEND/RECEIVE-Anweisungen, so kann man zur Compilezeit feststellen, ob hierbei Fehler auftreten werden.

5. Zugriff vom seriellen Programm auf lokale Daten von Prozessoren P(v). Das serielle Programm kann auf

lokale Variablen V oder Arrays A von Prozessor P(v) durch Anhängen des Keywords OF gefolgt vom Namen v zugreifen. Die Anweisung

a := b OF (15,15)

kopiert beispielsweise die lokale Variable b von Prozessor P(15,15) des Gitters in die Variable a des seriellen Programms.

6. Broadcast eines Arrays oder einer Variablen an alle Prozessoren erreicht man durch ein Kommando

a OF ALL := b.

Hierdurch wird der Wert von b in die lokale Variable a eines jeden Prozessors kopiert.

7. Beim Aufruf eines neuen Netzes bleiben auf allen Prozessoren die alten lokalen Variablen und Arrays am Leben. Die Prozessoren bekommen nur neue Namen, und sie können nun über die Kanten eines neuen virtuellen Netzes miteinander kommunizieren. Dabei wird die Umbenennung vom Compiler willkürlich vorgenommen. Allerdings kann darauf durch eine IDENTIFY-Anweisung Einfluß genommen werden. Wird etwa das obige Netz GRID gegen ein Netz LINE mit Knoten (1,...,256) durch den Netzaufruf NET = LINE ausgetauscht, so kann man etwa dafür sorgen, daß der Prozessor an der nordwestlichen Ecke von Grid nun der Linksaußen von Line wird. Dazu schreibt man direkt nach dem Netzaufruf

IDENTIFY LINE(0) = GRID(0,0)

In dem folgenden abschließenden Beispiel eines in diesem Stil geschriebenen parallelen Programms wird das vorne beschriebene Verfahren zur Simulation einer schwingenden Saite implementiert.

PARALLELE PROGRAMMIERUNG DER SCHWINGENDEN SAITE

```
/*Durch geeignete Wahl der Maßeinheiten für L und T
ergibt sich Δx=1, Δt=1.
      INTEGER L, T, P, PROC
      REAL U, U0, MUE, MUEQUAD
      MUE = 1/10
      L = 1000 /*Anzahl Punkte auf der Saite
      T = 1000/MUE /*Anzahl Zeitintervalle
      P = 10 /*Anzahl Prozessoren
      MUEQUAD = MUE *MUE
      DIMENSION U (0:L-1, 0:T-1), U0(0:L-1)
      [U0-Feld mit Anfangswerten für t=0 belegen]
      DO 10 I = 0,T-1 /*Randwerte setzen
      U(0,I) = 0
10    U(L-1,I) = 0
      DO 20 I = 1,L-2 /*Anfangswerte setzen
      U(I,0) = U0(I)
20    U(I,1) = U0(I)
      DEFNET KETTE (PROC=0:P-1)/*Netzdeklar.
      DIR = (LINKS, RECHTS)
      NEIGHBOR(I) = (LINKS: IF (PROC.GT. 0)
      PROC-1 ELSE NIL
      RECHTS: IF (PROC.LT.P-1) PROC+1 ELSE NIL)
      ENDEF
      NET = KETTE /*Netzaufruf
      PARBEGIN
      INTEGER L, P, ML, MR
      REAL UP2, UP1, UP /*U für t=T+2dT, T+dT und T
      REAL MUQ
      L = 1000
      DIMENSIONUP2(0:L-1),UP1(0:L-1),UP(0:L-1)
      PAREND
      DO 25 I = 0, L-1
      UP OF ALL := U(I,0)
25    UP1 OF ALL := U(I,1)
      MUQ OF ALL := MUEQUAD
      P OF ALL := P
      PARBEGIN
      ML = L/P *PROC /*Nr. des linkesten Mill
      MR = L/P *(PROC+1)-1/*Nr. des rechtesten M.
      IF PROC.EQ.0 ML=1
      IF PROC.EQ.(P-1) MR = L-2
      PAREND
      DO 100 J = 2, T-1 /*Hauptschleife
      PARBEGIN
      SEND (LINKS: UP1(ML) /*Communicate-Runde
           RECHTS: UP1(MR))
      RECEIVE (LINKS: UP1(ML-1)
           RECHTS: UP1(MR+1))/*Compute Runde
      DO 30 I = ML,MR
        30  UP2(I) = 2*UP1(I)-UP(I)+MUQ*
            (UP1(I-1)-2*UP1(I)+UP1(I+1))
            DO 40 I = ML,MR
            UP(I) = UP1(I)
40    UP1(I) = UP2(I)
      PAREND
      IF J/50 .NE. TRUNC(J/50) GOTO 100
      DO 50 I = 1,L-2 /*Ausgabe der Auslenkung
50    U(I,J) = UP2(I) OF TRUNC(I*P/L)
           /*alle 50 Iterationen zum /*seriellen Teil
100   CONTINUE
```

Literatur

1 *D.J. Auerbach, A.F. Bakker, T.C. Chen, A.A. Munshi* and *W.J. Paul:* A Highly Parallel Computer For Molecular Dynamics Simulations. Mat. Res. Symp. Proc. Vol. 63. 1985 Materials Research Socyety.

2 *D.J. Auerbach, W. Paul, A.F. Bakker, C. Lutz, W.E. Rudge:* A Special Purpose Parallel Computer for Molecular Dynamics: Motivation, Design, Implementation and Application. IBM Research Report RJ 5431 (55560), 1986.

3 *V. Benes:* Mathematical Theory of Connecting Networks and Telephone Traffic. Academic Press, New York 1965.

4 *G.E. Forsythe* and *W.R. Wasow:* Finite-Difference Methods for Partial Differential Equations. John Wiley and Sons, Inc. New York, London, 1960.

5 *G.C. Fox* and *S.W. Otto:* Physics Today 37, No 5,53, 1984.

6 *Z. Galil* and *W.J. Paul:* An Efficient General-Purpose Parallel Computer. J. ACM 30, 360–387, 1983.

7 *W. Giloi* und *U. Trottenberg:* Private Kommunikation

8 *G. Lev, N. Pippenger* and *L.G. Valiant:* A fast parallel algorithm for routing in permutation networks. IEEE Trans. Comput. C-30, 93–100, 1981.

9 *A. Munshi* and *W. Paul:* A Language for Use with Spark and Barnburner. Manuscript. 1985.

10 *W. Paul* and *N. Pippenger:* Parallel Computers Coupling a Permutation Network. IBM Technical Disclosure Bulletin. Vol. 28, No. 7, 1985. (1060)