MAX-PLANCK-INSTITUT FÜR PLASMAPHYSIK

GARCHING BEI MÜNCHEN

Max-Planck-Institut für Plasmaphysik (IPP)

Abteilung Informatik (Prof. Dr. F. Hertweck)

Implementierung eines parallelisierten Buneman-Verfahrens für ein Multitransputersystem

R. Strunz

IPP R/44 März 1993

Die nachstehende Arbeit wurde im Rahmen des Vertrages zwischen dem Max-Planck-Institut für Plasmaphysik und der Europäischen Atomgemeinschaft über die Zusammenarbeit auf dem Gebiete der Plasmaphysik durchgeführt.

Implementierung eines parallelisierten Buneman-Verfahrens für ein Multitransputersystem

Rainer Strunz *
Max-Planck-Institut für Plasmaphysik
Abteilung Informatik (Prof. Hertweck)
Boltzmannstr. 2, W-8046 Garching, FRG

Abstract

Buneman's algorithm is considered one of the fastest sequential methods for solving linear systems of equations derived from the discretization of certain types of partial differential equations. According to papers presented recently, an efficient parallelization of this direct method can be achieved by replacing the recursive kernel of the algorithm by a parallel one, based on partial fraction expansion. Providing and analyzing an efficient implementation for a multiple transputer system using this superior ansatz, proves the Buneman algorithm to be an excellent parallel solution method either.

Zusammenfassung

Der Buneman-Algorithmus gehört mit zu den schnellsten sequentiellen Verfahren zur Lösung linearer Gleichungssysteme, die bei der Diskretisierung einer gewissen Klasse von partiellen Differentialgleichungen entstehen. Eine effiziente Parallelisierung dieser direkten Methode wurde aber erst durch die vor kurzem vorgestellte Möglichkeit der Ersetzung des verfahrensinternen rekursiv-multiplikativen Lösungsansatzes durch einen auf Partialbruchzerlegung basierenden additiv-parallelen Kern in Aussicht gestellt. Die detaillierte Vorführung einer effizienten praktischen Implementierung für ein Multitransputersystem zeigt die Wirksamkeit dieser Idee und belegt die Leistungsfähigkeit des Buneman-Verfahrens auch in seiner so parallelisierten Form.

Übersicht

* Die hier beschriebenen Arbeiten wurden bereits 1990/1991 durchgeführt.

Einführung

Eine der am häufigsten verwendeten Methoden, Differentialgleichungen numerisch zu lösen, besteht darin, eine (unter Umständen erst mit Hilfe eines Iterationsschemas linearisierte) Differentialgleichung über einem einfachen Grundgebiet (meist entsprechend dimensionale Gitter) zu diskretisieren, die auftretenden Ableitungen dort mittels abgebrochener Taylorreihen zu approximieren (in der Ebene üblicherweise mit einem 5-Stern) und die entstandenen Differenzengleichungen schließlich, zusammengefaßt zu einem linearen Gleichungssystem, mit geeigneten numerischen Verfahren zu lösen.

Dabei entscheiden der vorgegebene Differentialoperator und die Anordnung der Gleichungen darüber, welche Struktur und numerischen Eigenschaften die Systemmatrix des interessierenden Gleichungssystems besitzt. Je spezieller diese nun sind, umso speziellere und damit effizientere Algorithmen lassen sich zu seiner Lösung finden und einsetzen.

Im Falle einer ebenen elliptischen Differentialgleichung etwa, mit dem in der Magnetohydrodynamik gebräuchlichen Laplace-ähnlichen Operator Δ^* aus

$$\Delta^* \psi(r,z) = -\psi_{rr} - \psi_{zz} + 1/r\ \psi_r = -r\ (1/r\ \psi_r)_r - r\ (1/r\ \psi_z)_z$$

und M bzw. N Gitterpunkten in z- bzw. r-Richtung (zylindrische Koordinaten bei Rotationssymmetrie), besitzt die entsprechende Systemmatrix B bei geeigneter Anordnung der Unbekannten (r- vor z-Richtung, jeweils aufsteigend) eine M-dimensionale Blocktridiagonalgestalt mit quadratischen N-dimensionalen Matrizen als Komponenten: negative Einheitsmatrizen $-I_N$ als Nebendiagonal- und lauter identischen Tridiagonalmatrizen A als Hauptdiagonaleinträgen.

Partitioniert man die zu einem MN-Vektor p zusammengefaßten Unbekannten und die rechte Seite q entsprechend der Blockstruktur von B, so erhält man ein Gleichungssystem $Bp=q$, ausgeschrieben als

$$\begin{bmatrix} A & -I_N & & & & \\ -I_N & A & -I_N & & & \\ & \cdot & \cdot & \cdot & & \\ & & \cdot & \cdot & \cdot & \\ & & & -I_N & A & -I_N \\ & & & & -I_N & A \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ \cdot \\ \cdot \\ p_{M-2} \\ p_{M-1} \end{bmatrix} = \begin{bmatrix} q_0 \\ q_1 \\ \cdot \\ \cdot \\ q_{M-2} \\ q_{M-1} \end{bmatrix}.$$

Zur Lösung dieser Art von Gleichungssystemen existiert nun ein wie maßgeschneiderter, sehr schneller, direkter Algorithmus: Das Buneman-Verfahren, eine stabilisierte Version der Methode der blockorientierten zyklischen Reduktion. Damit lassen sich die MN Unbekannten der Lösung auf dem Gitter größenordnungsmäßig mit $O(M\,N\,\mathrm{ld}\,M)$ Rechenoperationen bestimmen.

1. Der sequentielle Algorithmus

Auf eine Herleitung des Verfahrens, sowohl der zyklischen Reduktion als auch den Verbesserungen von O. Buneman, sei hier verzichtet (eine ausgezeichnete Darstellung hierzu liefert etwa [1]). Die bekannte Formulierung des fertigen Verfahrens zur Lösung eines oben beschriebenen Gleichungssystems $Bp=q$ unter der (zumindest in Lehrbeispielen üblichen) Voraussetzung $M = 2^{s+1}-1$ sieht - in abstrahierter C-ähnlicher Scheibweise - so aus:

Sequentieller Buneman-Algorithmus:

```
/* Parameter:   M bzw. s, N, A, p, q */
/* Hilfsvektor: x                    */


/* Vorbesetzung: */
   /* M muß von der Form 2^(s+1)-1, s >= 0 sein,     */
   /* q wird im Laufe des Verfahrens modifiziert */

   for (j = 0; j < M; j++)
      p[j] = 0;                    /* Nullvektor */


/* Reduktionsphase: */

   for (r = 0; r < s; r++)
      for (j = 2^(r+1) - 1; j < 2^(s+1) - 2^(r+1); j += 2^(r+1)) {
         Solve  A^(r) x = q[j] + p[j-2^r] + p[j+2^r];
         p[j] = x + p[j];
         q[j] = 2*p[j] + q[j-2^r] + q[j+2^r];
      }


/* Substitutionsphase: */
   /* Vektoren mit Indizes <0 oder >M-1 werden "ignoriert" */

   for (r = s; r >= 0; r--)
      for (j = 2^r - 1; j < 2^(s+1) - 2^r; j += 2^(r+1)) {
         Solve  A^(r) x = q[j] + p[j-2^r] + p[j+2^r];
         p[j] = x + p[j];
      }
```

Zum leichteren Verständnis sei der Ablauf des Algorithmus am Beispiel M=7 verdeutlicht (siehe dazu auch Abb. 1a):

In der hier 2-stufigen Reduktionsphase werden die anfänglichen 7 Tridiagonalsysteme auf 3 (nur virtuell vorhandene) Pentagonalsysteme reduziert (r=0), und diese wiederum auf ein einziges vollbesetztes Gleichungssystem (r=1). In der 3-stufigen Substitutionsphase wird dieses dann rekursiv über seine Faktoren gelöst (r=2), wodurch der mittlere Teillösungsvektor p[3] entsteht. Weiter erhält man dann durch analoge Lösungen penta- (r=1) und schließlich tridiagonaler (r=0) Systeme die ungeradzahlig indizierten Teillösungsvektoren p[1] und p[5] und am Ende die geradzahlig indizierten partiellen Lösungen p[0], p[2], p[4] und p[6].

Da aber die Matrizen $A^{(r)}$ vollständig in ein Produkt aus Linearfaktoren mit bekannten Nullstellen $\lambda_i^{(r)}$ zerfallen,

$$A^{(r)} = \prod_{i=1}^{2^r} A_i^{(r)} \qquad \text{mit } A_i^{(r)} = A - \lambda_i^{(r)} I_N ,$$

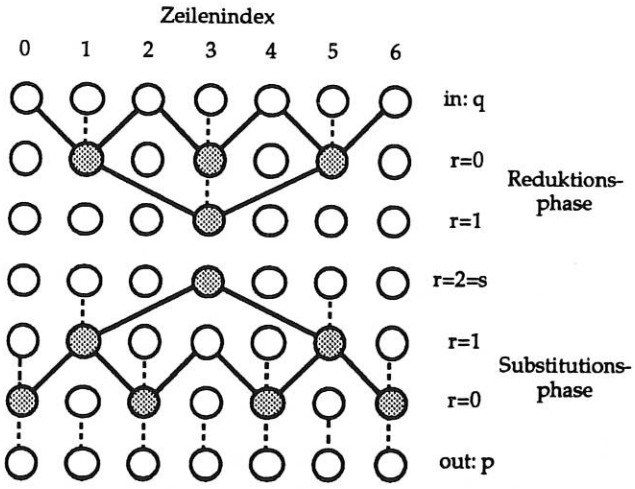
können Gleichungssysteme damit leicht und elegant rekursiv gelöst werden:

```
Solve  A(r) x = y :

    /* Parameter:      A, r, x, y */
    /* Hilfsvektoren: h           */

    h[0] = y;

    for (i = 1; i <= 2^r; i++) {
        θ_i^(r) = (2 * i - 1) * π / 2^(r+1);
        λ_i^(r) = 2 * cos(θ_i^(r));

        Trisolve  (A - λ_i^(r) I_N) h[i] = h[i-1];
    }

    x = h[2^r];
```

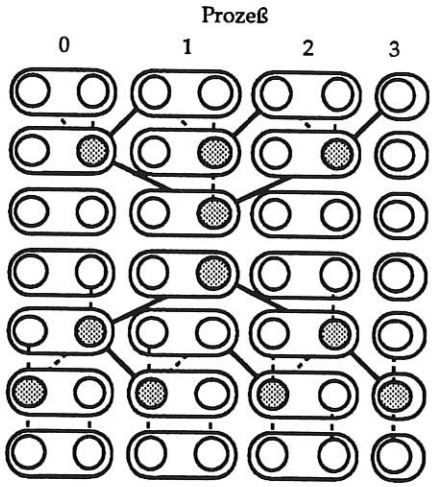
Die hier noch aufzulösenden Gleichungssysteme *(trisolve)* mit den Matrizen $A_i^{(r)}$ sind einfache skalare Tridiagonalgleichungssysteme und lassen sich daher mit einer der vielen existierenden Tridiagonallösemethoden angehen. Die schnellsten sequentiellen Verfahren beruhen auf einer Dreieckeckszerlegung der Systemmatrix und anschließenden linearen Rekursionen.

Abb. 1
a) Indexabhängigkeiten

Beispiel für M=7, also s=2



b) Prozeßkommunikationsstruktur



Vertikal von oben nach unten wird das aufeinanderfolgende Ablaufen der einzelnen Stufen beider Phasen zum Ausdruck gebracht (die r-Schleifen). Horizontal sind die pro Stufe (r fest) durchzuführenden Operationen (j-Schleifen) durch schraffierte Kreise (j fest) schematisiert. Durchgehende Linien geben fremde Datenabhängigkeiten (aus den Indizes $j \pm 2^r$ der p- und q-Vektoren) für den darunter liegenden Schleifenrumpf an, gestrichelte Linien Zugriffe auf lokal vorhandene Daten.

Da im gesamten Verlauf pro Stufe r jeweils $2^{s-r+1}-1$ Gleichungssysteme mit derselben Tridiagonalmatrix $A_i^{(r)}$, wovon es wiederum genau M verschiedene gibt, auftreten, empfiehlt es sich, den von den unterschiedlichen rechten Seiten unabhängigen Rechenaufwand, im wesentlichen also die Matrixfaktorisierung, schon im Voraus zu erledigen und so die Rechenbelastung für die eigentlichen, öfter wiederkehrenden Lösungsvorgänge auf ein Minimum zu reduzieren. Dies ist natürlich erst recht von Bedeutung, wenn das Verfahren für gleiche Gittergrößen mehr als einmal angewandt wird, wie es etwa für den eingangs erwähnten Fall von Linearisierungsiterationen bei nichtlinearen Differentialgleichungen zutrifft.

Ist nun

$$A = \begin{bmatrix} b_0 & c_0 & & & \\ a_1 & b_1 & c_1 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{N-2} & b_{N-2} & c_{N-2} \\ & & & a_{N-1} & b_{N-1} \end{bmatrix},$$

und legt man eine LU-Zerlegung der $A_i^{(r)}$, zugrunde, so lautet der zweigeteilte Algorithmus, der auf dem bekannten sog. Thomas-Algorithmus zur Lösung eines einzelnen Tridiagonalsystems (siehe dazu etwa [3]) basiert, folgendermaßen: [1)]

Trisolve $(A - \lambda I_N)\, x = y$:

```
/* Parameter:    A, λ, x, wbyc (indirekt), y                              */
/* Anmerkungen: x und y können (im Rahmen von Solve) identisch sein:     */
/*              die rechte Seite enthält also am Ende auch die Lösung;   */
/*              die Erstellung von wbyc ist nur einmal pro λ notwendig   */

/* einmalige Vorausberechnung (Faktorisierung): */
   wbyc[0] = 1 / (b[0] - λ);          /* Start */

   for (i = 1; i < N; i++)            /* Rekursion          */
      wbyc[i] = 1 / (b[i] - λ - a[i] * wbyc[i-1] * c[i-1]);


/* jeweilige Anwendung (Eliminationen): */
   x[0] = y[0] * wbyc[0];             /* Start */

   for (i = 1; i < N; i++)            /* Vorwärtsrekursion */
      x[i] = (y[i] - a[i] * x[i-1]) * wbyc[i];

   for (i = N-2; i >= 0; i--)         /* Rücksubstitution  */
      x[i] = x[i] - wbyc[i] * c[i] * x[i+1];
```

1) Durch eine aufwendigere LDU-Zerlegung der Tridiagonalmatrizen könnte noch eine Multiplikation im Anwendungsteil eingespart werden, wodurch statt einem Vektor (wbyc) allerdings drei (L, D und U repräsentierende) N-Vektoren notwendig werden, was den Speicherbedarf des gesamten Verfahrens fast verdoppelt.

2. Inhärenter und implantierter Parallelismus

Die oben beschriebene effiziente sequentielle Formulierung des Algorithmus soll nun als Basis weiterer Untersuchungen dienen. Denn schon dort findet man als einen ersten Ansatzpunkt zur Parallelisierung inhärenten Parallelismus: Die innersten Schleifenrümpfe bzw. Kernstücke beider Phasen können offenbar nebeneinander ausgeführt werden. Zur Verdeutlichung der nun folgenden Überlegungungen sei einmal nur ein einzelner allgemeiner Rumpf (lokal, r und j fest) dargestellt, welcher sich in drei Abschnitte gliedern läßt:

Sequentieller Kern:

I. Aufstellung einer rechten Seite:
```
y = q[j] + p[j-2^r] + p[j+2^r];
```

II. Lösung eines linearen Gleichungssystems und Korrektur des lokalen p-Vektors:
```
Solve  A^(r)  x = y;
p[j] = x + p[j];
```

III. Korrektur des lokalen q-Vektors (nur in der Reduktionsphase):
```
q[j] = 2 * p[j] + q[j-2^r] + q[j+2^r];
```

In Abschnitt II sind keine anderen Indizes als j beteiligt. Für alle zulässigen j einer Stufe r können demnach die lokalen Lösungen x der dort auftretenden Gleichungssysteme vollkommen unabhängig voneinander bestimmt werden. Offensichtlich gilt dies auch für Abschnitt I. Zwar treten hier Indizes ungleich j auf, doch die zugehörigen p-Vektoren werden zu diesem Zeitpunkt nur gelesen. In Abschnitt III schließlich (der nur für die Reduktionsphase existiert) werden im Indexabstand 2^r benachbarte q-Vektoren gelesen und der eigene lokale q-Vektor damit korrigiert. Da zur gleichen Zeit aber für ein j kein anderes $j'=j\pm2^r$ aktiv ist (die nächsten korrigierend-aktiven j' sind $\pm2^{r+1}$ entfernt; siehe auch Abb. 1a), gibt es auch hier keine Konsistenzprobleme durch Aktualisierungsanomalien bei parallelem Zugriff.

Da pro Stufe r der Reduktions- bzw. Substitutionsphase jeweils $2^{s-r}-1$ bzw. 2^{s-r} Gleichungssysteme zu lösen sind, genügen zur parallelen Ausführung aller Schleifenrümpfe $2^s=(M+1)/2$ parallele Rechenprozesse. Ein solcher Prozeß, etwa mit der Identnummer k, hat dann die den Zeilenindizes 2k und 2k+1 zugeordneten Daten, ein sog. Zeilenpaar, bestehend aus p- und q-Vektoren, sowie passenden λ-Faktoren, zu betreuen. Nur der letzte Prozeß, mit der Nummer 2^s-1, spielt dahingegehend eine Ausnahmerolle, daß er lediglich für die einzelne letzte Zeile M-1 zuständig ist (siehe zu all dem Abb. 1b).

Was wäre nun eine solche parallele Implementierung zu leisten im Stande? Geht man einmal vom Idealfall unendlich schneller Kommunikation aus und ordnet jeden Prozeß einem Prozessor zu, so kann man durch simples Abzählen möglicher, nur numerisch relevanter, paralleler Operationen und Vergleich mit dem sequentiellen Pendant leicht Aussagen, wenn auch etwas grobe, über Obergrenzen für optimale Beschleunigungen (und daraus Effizienzen) gewinnen. Wie man nachrechnen und (in Diag. 2 an den untersten Kurven) sehen kann, bewegen sich diese Optimalwerte mit O(ld M) - Minsky läßt grüßen - im untersten Bereich dessen, was man an Beschleunigung wohl erwartet.

Dies liegt daran, daß im Vergleich von einer Stufe r zur nächsten r+1 (bzw. vorhergehenden r-1) immer weniger Prozessoren (phasenabhängig fast bzw. genau die Hälfte) immer mehr Arbeit (jeweils doppelt so viele skalare Tridiagonalsysteme) bewältigen müssen. Da das gesamte sequentielle Arbeitsaufkommen einer Stufe mit 2^s-2^r bzw. 2^s fast - na ja - bzw. ganz konstant ist (Arbeitseinheit ist dabei der Lösungsaufwand für ein Tridiagonalsystem), stellt sich die Frage, ob es nicht möglich ist, durch eine Zerlegung und Verteilung dieser Arbeit, also durch geeignete parallele Lösung eines jeden auftretenden Gleichungssystems $A^{(r)}x=y$, alle Prozessoren besser bzw. ganz auszulasten (ähnlich den Fällen r=0) und damit die Effizienz deutlich zu erhöhen.

Bei der Entwicklung parallelisierbarer Verfahren für ein vorgegebenes Problem ist es nun keineswegs eine neue Erkenntnis, daß ein Algorithmus dafür zwar sequentiell als effizient bezeichnet werden kann, aber parallelisiert enorme Leistungseinbußen erleidet. Bessere parallele Effizienz bieten in solchen Fällen oft unkonventionelle Lösungsansätze - weniger auf ganz neuen Ideen basierend, dafür mehr aus bekannten, aber ein wenig in Vergessenheit geratenen Bestandteilen neu zusammengestellt -, die aufgrund ihres Mehraufwandes sequentiell uninteressant sind, diesen aber parallelisiert mehr als wettmachen. [2)]

Das Problem einer brauchbaren parallelen Lösung der $A^{(r)}x=y$ stellt solch einen Fall dar. Aus der Funktionentheorie ist nämlich bekannt, daß formal für die zu $A^{(r)}$ inverse Matrix

$$(A^{(r)})^{-1} = \sum_{i=1}^{2^r} \gamma_i^{(r)} \, (A_i^{(r)})^{-1}$$

gilt (siehe [3], [4]), wo die γ-Faktoren, wie schon die λ-Dekremente zuvor, leicht herleitbar sind. Damit gilt für die gesuchte Lösung $x=(A^{(r)})^{-1}y$ natürlich sofort

$$x = \sum_{i=1}^{2^r} \gamma_i^{(r)} \, x_i \qquad \text{mit } x_i = (A^{(r)})^{-1} y \text{ und } \gamma_i^{(r)} = (-1)^{r-1} / 2^r * \sin \theta_i^{(r)} ,$$

was nichts anderes bedeutet, als die Lösungen x_i der 2^r Tridiagonalsysteme $A_i^{(r)} x_i = y$ zu bestimmen (genau soviel Aufwand war vorher schon nötig) und danach linear zu kombinieren (dieser Aufwand kommt jetzt noch hinzu). Die Summanden $\gamma_i^{(r)} x_i$ können nun jedoch völlig unabhängig voneinander bestimmt werden (vorher die entsprechenden Zwischenlösungen nur nacheinander), während die gesuchte Partialsumme x immerhin noch in (bestenfalls) r parallelen Schritten errechnet werden kann.

Gruppiert man etwa für eine Kettentopologie [3)] unter Berücksichtigung dieser Partialbruchzerlegung in einer Stufe $r>0$ die 2^{r-1}-1 linken und 2^{r-1} rechten Nachbarn, Arbeiter genannt, eines mit der gemeinschaftlichen Lösung eines Systems $A^{(r)}x=y$ beauftragten Prozessors, dem Vorarbeiter, mit diesem zu einer Mannschaft, einer sogenannten 2^r-Gruppe, so lassen sich in der Substitutionsphase jetzt alle 2^s und in der Reduktionsphase immerhin 2^s-2^r Prozessoren (die restlichen 2^r sind quasi arbeitlos) mit einzelnen skalaren tridiagonalen Gleichungssystemen beschäftigen. Damit zerfällt jeder alte sequentielle Kern in 2^r neue parallele Kernstücke (vergl. Abb. 2 u. 3):

2) Andere Beispiele hierfür sind etwa skalare zyklische Reduktion oder Recursive Doubling anstelle simpler Rekursionen, wie sie auch bei dem hier verwendeten Tridiagonallöser *trisolve* auftreten. Das Buneman-Verfahren könnte also in der Nichtreduktionsrichtung weiter vektorisiert/parallelisiert werden. Da die Parallelisierung tridiagonaler Systeme jedoch nichts Neues ist, sei hier nur noch einmal daran erinnert, daß damit theoretisch eine weitere Beschleunigung von bis zu O(N/ld N) möglich wird.

3) Eine kettenartige Anordnung mehrerer Transputer dürfte in jeder Konfiguration möglich sein. In der Regel wird sie zum Booten eines Netzwerkes benutzt. Wünschenswert für dieses Verfahren wäre natürlich eine (binäre) Hypercube-Toplogie.

Parallele Kerne:

Arbeiter i: $i \in [j - 2^{r-1} + 1, j + 2^{r-1}] \setminus j$

IIa. Lösung eines Tridiagonalsystems:
`Trisolve` $A_i^{(r)} x_i = y;$

IIb. Bildung eines Lösungssummanden:
$x_i = \gamma_i^{(r)} x_i;$

Vorarbeiter j:

I. Aufstellung einer rechten Seite:
`y = q[j] + p[j-2^r] + p[j+2^r];`

IIa. Lösung eines Tridiagonalsystems:
`Trisolve` $A_j^{(r)} x_j = y;$

IIb. Bildung eines Lösungssummanden:
$x_j = \gamma_j^{(r)} x_j;$

IIc. Korrektur des lokalen p-Vektors:

$p[j] = \sum_{i=1}^{2^r} x_i + p[j];$

III. Korrektur des lokalen q-Vektors
(nur in der Reduktionsphase):
`q[j] = 2 * p[j] + q[j-2^r] + q[j+2^r];`

Abb. 2) Gruppenbildung am Beispiel M=15
bei paralleler Partialbruchzerlegung

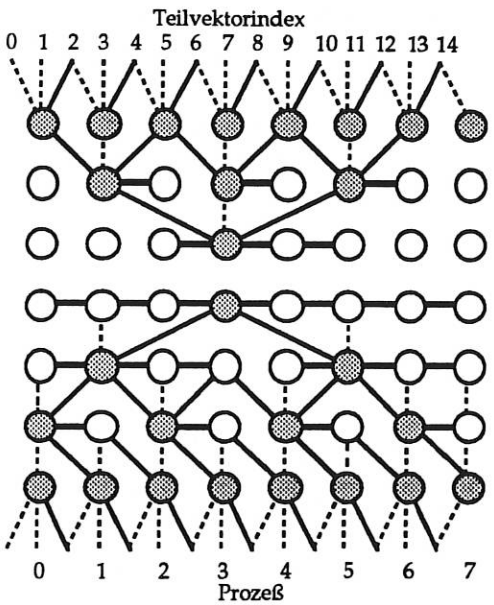
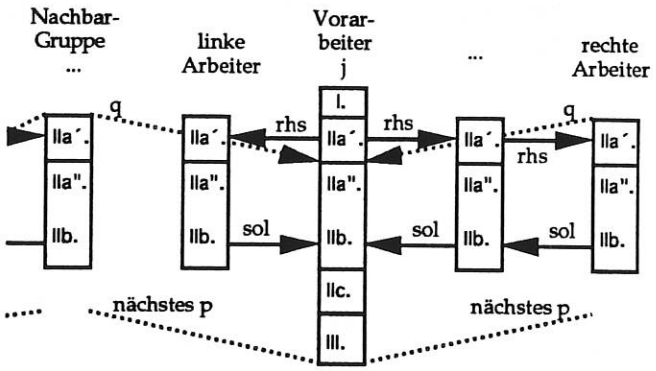


Abb. 3) Einzelne vollständige 2^r-Gruppe mit r=2
(etwa aus der Reduktionsphase aus Abb. 2)

Konfliktfreie Bewältigung der Gruppenaufgabe durch Spaltung des Tridiagonallöseabschnittes, da bis zum Beginn der Rücksubstitution (mit kombinierter Teillösungsaddition) die q-Vektorkommunikation abgeschlossen sein muß.



Die Kommunikationspartner der nächsten p-Vektoren sind phasenabhängig: Erst sind es nur die verschiedenenVorarbeiter, später dann bestimmte Gruppenmitglieder untereinander.

Nach gleichartiger Anwendung der ideal-theoretischen Ergebnisse dieser Kur kommt man nun auf Werte (Diag. 2a, jeweils die zweite Kurve der Legende), welche diejenigen des ersten Ansatzes bei weitem übertreffen und somit auch auf eine praktische effiziente Parallelisierung hoffen lassen: Die Beschleunigung landet mit O(M/ldM) bei bestmöglicher (logarithmischer statt hier benutzter "dualer") Parallelisierung der Summation der Teillösungen zwischen dem bisherigen logarithmischen und dem (hier unerreichbaren) maximal linearen Speedup. Dabei fällt auf (Vergleich der ersten beiden Kurven in Diag. 2a), daß bis in den Bereich von etwa M=31 hinein, was maximal 16 Prozessoren entspricht, statt einer Topologie mit logarithmischer Entfernung auch eine einfache lineare Topologie, also eine Kette oder Ring, ohne allzu großen Leistungsverlust verwendet werden kann.

3. Eine konkrete Implementierung: Occam und Transputer

Für eine konkrete Implementierung muß man natürlich den bisher vernachlässigten Aufwand für den notwendigen Datenaustausch möglichst gering halten, um dem beschriebenen Ideal auch nahe kommen zu können. Möglichkeiten zur Minimierung dieses Kommuniktionsoverheads hängen nun sehr stark sowohl von den Fähigkeiten der Hardware (Prozessoren und Netzwerk) als auch dem Ausdrucksvermögen der Implementierungssprache ab. Im hier vorliegenden Fall werden Transputer vom Typ T800 und die eigens für diese Klasse von Prozessoren entwickelte parallele Programmiersprache Occam eingesetzt. Verbunden sind diese Prozessoren im vorliegenden Fall (von der Anzahl her zweckmäßigerweise eine Zweierpotenz) Kettengliedern gleich linear, wodurch sich der den Aufwand eines Datenaustausches mitbestimmende Abstand zweier Prozessoren aus der absoluten Differenz ihrer Identifizierungsnummern ergibt. Da Kommunikation aber im allgemeinen um einiges teuerer als eine vergleichbare Rechenoperation ist, [4] würde eine naive oder mangels anderer Möglichkeiten erzwungene parallele Realisierung mittels *just in time* - Kommunikation nur einen Bruchteil der (in Diag. 2a-e, die Idealkurven bei linearer oder logarithmischer Konnektivität) prognostizierten Werte erreichen.

Einen Ausweg hierfür bietet die Fähigkeit der Transputer zur Überlappung von Kommunikations- und Rechenoperationen, und der Möglichkeit des Parallelbetriebs der vorhandenen vier Kommunikationskanalpaare, *links* genannt, untereinander und in beiden Richtungen. Der Einsatz dieser Eigenschaften soll zunächst für die Vektorkommunikation in der Reduktionsphase der alten rekursiv-multiplikativen Methode demonstriert werden:

Da die in Abschnitt I einer Stufe r benötigten p-Vektoren schon am Ende von Abschnitt II der vorausgegangenen Stufe r-1 fertiggestellt werden (anfangs, für r=0, sind alle p-Vektoren sowieso bekannt, nämlich noch gleich Null), kann die Zeitdauer von 3N Operationen des dazwischen liegenden Abschnittes III von r-1 (bzw. die entsprechende Leerlaufzeit gerade nicht rechnender Prozessoren) mit dazu genutzt werden, um beide p-Vektoren parallel im Voraus zu empfangen (bzw. zu schicken). Da der Abstand der kommunizierenden Prozessoren bei der angenommenen Kettentopologie exponentiell mit r wächst, kann man die p-Vektor-Kommunikation bei mit r steigenden Übertragungszeiten immer weniger gut verstecken.

Etwas günstiger sieht es dagegen bei den q-Vektoren aus. Zu ihrer Kommunikation steht der mit (6 + (r+1)) N Operationen linear mit r wachsende Abschnitt II zur Verfügung, wodurch versteckte Kommunikation sogar mit proportional zur Pfadlänge wachsenden Übertragungskosten verwirklicht werden kann. Für die Substitutionsphase gilt dies alles leider nicht. Zwar müssen hier wegen des fehlenden Abschnittes III keine q-Vektoren verschickt werden, doch fehlen damit auch Rechenoperationen zum Verbergen der p-Vektor-Kommunikation, die somit zeitmäßig voll ins Gewicht fällt.

Die Anwendung dieser Übertragungsstrategie auf die effizientere additiv-parallele Lösungsmethode bringt nun folgende Änderungen einer Stufe r mit sich. Anstelle einer sequentiellen Lösung von $A^{(r)}x=y$ treten nun die verteilten Lösungen $A_i^{(r)}x_i=y$ mit entsprechender Kommunikation der rechten Seite y und Partialsummen von x auf: Da die von einem Vorarbeiter produzierte und für seine gesamte Gruppe gültige rechte Seite unmittelbar danach gebraucht wird, scheidet eine Überlappung der Verteilung von y mit sinnvoller Rechnung aus. Gleiches gilt für die Lösungen x_i, welche der Vorarbeiter sofort zur Korrektur seines lokalen p-Vektors benötigt. [5]

[4] Bei 10 MBit/s kostet die Übertragung eines Feldes (also eines Vektors) aus doppelt genauen Gleitkommazahlen (64 bit) zwischen zwei T800 etwa soviel wie 2-3 (in Occam definierte) Vektoroperationen (wie Vektor ± Vektor, Vektor * Skalar, o. ä.).

[5] Der Begriff des globalen Zeitpunktes bzw. -raumes beruht dabei auf der Vorstellung, daß sich alle Prozessoren zur gleichen Zeit im gleichen Abschnitt und damit in der schärfsten bzw. ungünstigsten Konkurrenzsituation befinden (können).

Für die q-Vektor-Kommunikation in der Reduktionsphase ist noch zu beachten, daß bei Nichtverwendung selbstidentifizierender Nachrichten, etwa Vektoren mit einer Zusatzinformation, die angibt, ob es sich um einen p-, q-, x- oder y-Vektor handelt [6], man diese zur Vermeidung von Verwechslungen nicht auch parallel zur Versendung der x-Vektoren durchführen darf, da sich q- wie x-Vektoren Richtung Vorarbeiter bewegen und Vorhersagen über die Reihenfolge des Eintreffens im Prinzip nicht möglich sind.

6) Auf solch eine Technik könnte man nicht verzichten, wenn in Occam nichtblockierendes Senden möglich wäre bzw. gemacht würde. Dann nämlich könnte es durchaus vorkommen, daß ein Vektor einen anderen quasi überholt und eine Identifizierung über den Eintreffzeitpunkt nicht mehr möglich wäre. Viel schlimmer wäre in so einem Scenario aber die Gefahr eines Deadlocks, welche die Einführung eines (nichtblockierenden) Kommunikationmodells unter Umständen (beispielsweise durch zusätzliche parallele Kommunikation bei pseudoparallelen Prozessen mit einem Kommunikationsserver pro Prozessor) mit sich bringen kann.

4. Stückchenweise Vektorkommunikation

Bis jetzt konnte man sich die Kommunikationsverbindungen der parallel arbeitenden Prozessoren ohne weiteres auch als direkte Leitungen vorstellen, die Knoten also als fest voll vermascht oder mit dynamisch variabler Toplogie versehen betrachten. In der Regel wird dies aber nicht gegeben sein, so daß der notwendige Datenfluß zwischen nicht benachbarten Prozessoren über geeignete dritte Knoten realisiert werden muß (wie es zum Beispiel bei einer Kette vorkommt). Dabei ist zu beachten, daß diese Zwischenknoten manchmal auch selbst am Inhalt der zu übermittelnden Nachricht interessiert sind, etwa bei der Verteilung einer rechten Seite und beim Aufsammeln der Teillösungen innerhalb einer 2^r-Gruppe, und manchmal nicht, wie bei der bloßen Weitergabe der p- und q-Vektoren.

Diese Versendung von Daten längs Pfaden uneinheitlicher Länge kann in unserem Falle aber noch optimiert werden. Denn die zu übertragenden Daten sind N-Vektoren, also aus vielen gleichen Komponenten zusammengesetze Gebilde. Statt nun so einen Vektor vom Start weg bis zum Ziel über alle, sagen wir k-1, dazwischen liegenden Stationen, also eine Weglänge k, zu befördern, was bei einer Startupzeit von α und einer Übertragungszeit von β pro Vektorkomponente insgesamt

$$t_{unsliced} = k * (\alpha + N * \beta)$$

Zeiteinheiten erfordert, nützt man die Fähigkeit der Transputer zur parallelen Kommunikation, indem man den Vektor in Päckchen *(slices)* der längs eines Weges einheitlichen Länge n zerhackt, und diese wie folgt verschickt: Während ein Päckchen i empfangen wird, kann parallel dazu das zuvor empfangene Päckchen i-1 weitergeleitet werden. Die vollständige Übertragung eines Vektors kostet auf diese *pipeline*-artigeWeise

$$t_{sliced} = (k + N/n - 1) * (\alpha + n * \beta)$$

Zeiteinheiten, was von der optimalen Packetgröße

$$n_{optimal} = [N / (k - 1) * (\alpha/\beta)]^{1/2}$$

minimiert wird. Die (in Diag. 1 dargestellte) theoretische Abhängigkeit der so errechneten Kommunikationsdauer von der Größe der Päckchen mit realistischen link-Parametern [7] ist relativ groß und läßt auch praktisch auf eine beträchtliche Verbesserung hoffen.

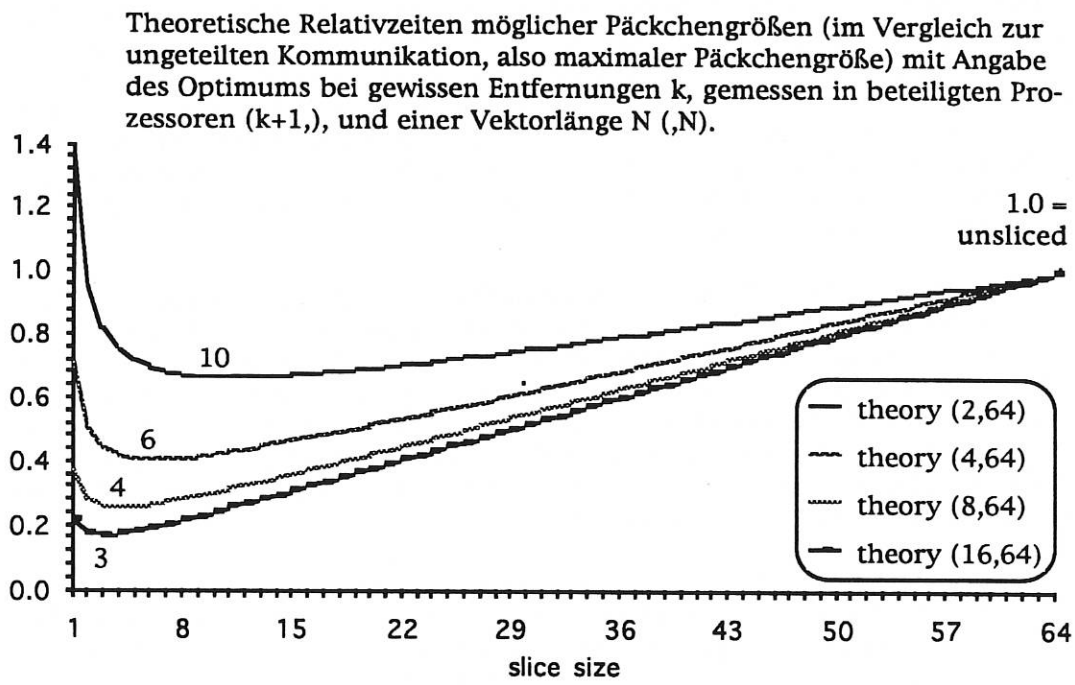
Die Päckchenmethode kann man nun aber nicht nur dazu verwenden, die bloße Übertragung der Vektordaten zu beschleunigen, sondern auch deren Verarbeitung. Nach der schon erwähnten Verteilung der einer 2^r-Gruppe gemeinsamen rechten Seite y wird diese nämlich im Zuge der individuellen Lösung eines Tridiagonalsystems (siehe *trisolve*) während der Vorwärtsrekursion von vorne nach hinten gelesen und damit, in gleihem Maße wachsend, ein temporärer Vektor (auf dem Platz der späteren lokalen Lösung x) erzeugt. Die Bildung dieses Vektors kann nun ebenfalls stückchenweise erfolgen: Während Päckchen i+1 der rechten Seite empfangen wird, kann parallel dazu nicht nur Päckchen i weitergeleitet, sondern auch gelesen und damit das Stückchen i des Zwischenvektors x gebildet werden.

7) Tatsächlich sind α und β u.a. wiederum von der Zahl der gleichzeitig betriebenen links (welche sich nun erhöht) und ihrer Betreibungsrichtung (ob uni- oder bidirektional) abhängig (siehe [5]). Im Endeffekt kann man dann bytebezogen (also protokollos) von einem Verhältnis α/β von 10,41 und kompontenbezogen (Arrayprotokoll INT::[]REAL64) von 1,80 ausgehen.

Auch für die Rücksubstitution, welche die endgültige lokale Lösung x von hinten nach vorne bildet, die Multiplikation mit $\gamma_i^{(r)}$, Addition zu Partialsummen der anderen Mitarbeiter und die Versendung der neuen Summe Richtung Vorarbeiter kann die Päckchentechnik angewandt werden: Während der Bildung eines x-Päckchens i (gemäß IIa. kombiniert mit IIb.) kann ein benachbartes x-Päckchen i-1 empfangen werden. Dieses wird nun zum eigenen Päckchen i-1 addiert und Richtung Vorarbeiter weitergeschickt, welcher laut Abschnitt IIc nur noch zu seinem x-Päckchen i-1 die als x empfangenen "linke" und "rechte" Partialsumme i-1 addieren muß, um das vollständige Lösungspäckchen i-1 von x aus $A^{(r)}x=y$ zu erhalten.

Durch diese Technik muß man allerdings die q-Vektor-Kommunikation auf die Vorwärtsrekursion der Tridiagonalsystemlösung beschränken. Denn nur dort läuft während dieser Zeit kein Verkehr in der gleichen Richtung ab: die Verteilung der rechten Seite geschieht vom Vorarbeiter weg in entgegengesetzte Richtungen. Die Rücksubstitution ist dafür nicht geeignet, da sich dort weiterhin im gleichen Zeitraum und in derselben Richtung die akkumulierten x-Vektoren bewegen.

Diag. 1 Theoretische zeitliche Auswirkung der Scheibentechnik mit N=64 und $\alpha/\beta = 1{,}80$



5. Verallgemeinerungen

Nicht immer wird man genauso viele Prozessoren einsetzen können oder wollen, wie es der maximale Parallelisierungsgrad der Aufgabenstellung, hier 2^s, erlaubt. In Occam ließe sich das sehr einfach bewerkstelligen: man müßte jedem Prozessor möglichst balanciert nur mehrere logische Rechenprozesse zuteilen, um das Problem mit weniger Prozessoren zu lösen. Allerdings verhält sich die Einfachheit der Problemlösung in diesem Falle leider umgekehrt proportional zum Erfolg, dem erwarteten Speedup. Die theoretisch vorhandene Prozessormehrbelastung durch die Bearbeitung mehrerer Zeilenpaare läßt sich wegen der in Occam üblichen formalen Gleichbehandlung von Intra- und Interprozessorkommunikation nicht effizient genug nutzen. Statt nun eine Anpassung der bisher identischen Prozesse (individuelles Verhalten wird über Vergleiche mit der Identnummer gesteuert) diesbezüglich durchzuführen, geht man besser dazu über, die auf einem Prozessor ablaufenden Prozesse gleich selbst quasi zu einem Superprozeß zu sequentialisieren. Damit kann man bei mehr als einem Zeilenpaar pro Prozessor ohne weiteres sogar die in der letzten Stufe der Reduktionsphase nichtbeschäftigten (immerhin die Hälfte aller vorhandenen) Prozessoren durch gleichmäßige Aufteilung der stufenweit zu bewältigenden Arbeit voll beschäftigen.

Insgesamt läßt sich die notwendige Kommunikation mit diesen Mitteln so wirksam mit den numerischen Berechnungen überlappen, daß bei steigender Prozessorbelastung der tatsächliche Beschleunigungsfaktor dem Ideal immer näher kommt (siehe Diag. 2a-e, jeweils die zweite - eine ideale - und dritte - die entsprechende reale - Kurve). Weitere deutliche Leistungssteigerungen des so parallelisierten Buneman-Verfahrens sind praktisch nur noch über kürzere Kommunikationswege durch speziellere Topologien (etwa einen Hypercube oder gar ein voll vermaschtes Netzwerk), schnellere Datenleitungen (derzeit sind maximal 20 Mbit/s möglich) oder gar die Verwendung von maschinenspezifischen Tricks, vor allem zur Beschleunigung der Vektoroperationen (wie blockweises *loop unrolling*, schnelle *block moves*, geeignete Vektorplazierung; Genaueres findet man in [5]) möglich.

Einen weiteren nicht unbedeutenden Einflußfaktor auf die erreichbare Beschleunigung stellt die Dimension N der kommunizierten Vektoren dar. Zwar kann (zumindest für die *unsliced* Version) erwarten, daß sich die Startup-Anteile umso weniger bemerkbar machen, je größer N ist. Doch lassen reale Zeitvergleiche diesbezüglich nur für sehr große N (N>>M) eine gewisse Sättigung erkennen, wo (bei gleichbleibenden anderen Parametern) der Zeitaufwand etwa proportional mit N steigt. Für 'kleinere' N, praktisch also den Normalfall, fällt die Zeit unterproportional mit N. Benötigt also ein paralleles Programm zur Lösung eines Problems mit den Dimensionen (M,N) die Zeit T, so wird es zur Lösung von (M,N/2) länger als T/2 laufen, für (M,2N) aber weniger als 2T benötigen. [8)]

Für diejenigen realen Werte (Diag. 2a-e, kleine M) die überraschenderweise über ihren Idealen liegen, gibt es eine Reihe von Faktoren, die im Zusammenspiel durchaus solche Merkwürdigkeiten hervorrufen können. Neben der schon erwähnten Ungenauigkeit bei der Herleitung der Idealwerte können die Gründe hierfür im geänderten Code, das heißt seiner Größe, Plazierung bzw. Verteiltheit liegen, im totalen und teilweisen Wegfall von Schleifendurchläufen in der Parallelisierungsrichtung, usw. (mehr dazu in [6]).

Mit den erwähnten Tricks zur Steigerung der effektiven Rechenleistung - mit Ausnahme der Plazierung kritischer Vektoren, welche bei der Parallelversion mit der Zahl gehaltener Zeilenpaare repliziert auftreten und so den schnellen Speicher im allgemeinen Fall (Occam kennt keine dynamischen Arrays) sprengen - wurden alle hier direkt oder indirekt vorgebrachten Zeitmessungen durchgeführt, beziehen sich also auf möglichst gleichartige (durch die Verwendung einheitlicher Prozeduren, Übersetzungsmodi, etc.) und damit vergleichbare Programmversionen. Allen Vergleichen liegt außerdem die beschriebene fähigste sequentielle Version zugrunde, und nicht etwa eine zwangssequentialisierte Parallelversion.

8) Tatsachen wie diese nähren Kontroversen um Art und Aussage von Beschleunigungsmessungen. Leider gilt in unserem Fall, daß sich die besten Werte bei maximaler Ausnutzung des Rechners, genauer des verteilten Speichers, ergeben und nicht schon für oft gebräuchlichere kleinere Größenordnungen. Bei dem im Institut anstehenden Experiment *Asdex Upgrade* etwa benutzt man zur Bestimmung der magnetischen Flußdiche aus technischen Gründen eine Auflösung von nur 63 x 40 Meßpunkten.

6. Resultate

Diag. 2a-e Vergleich von Beschleunigungen diverser paralleler Varianten

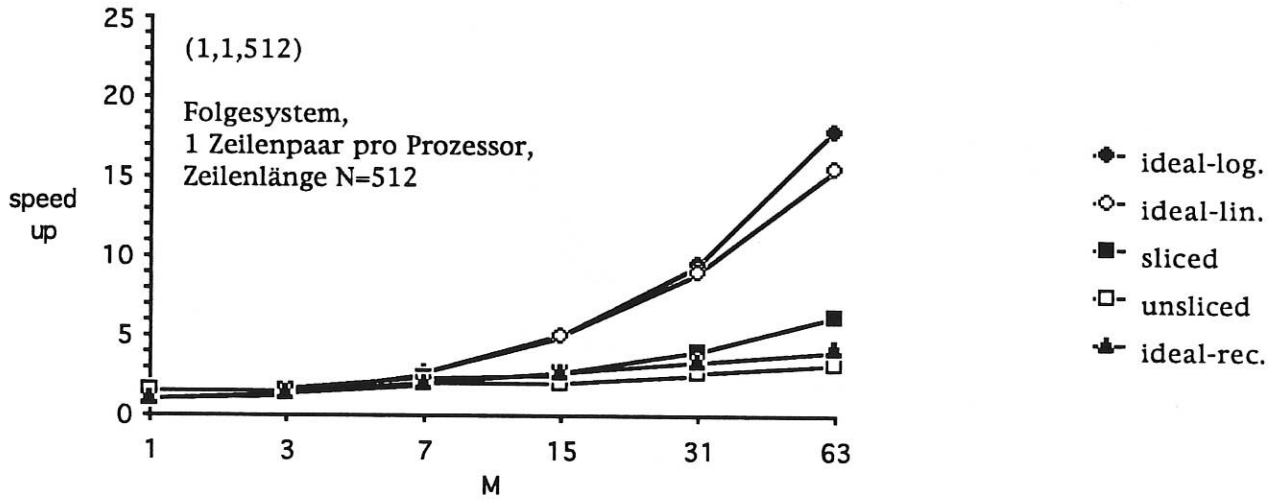
Folgesysteme (ohne Matrizenfaktorisierungen) *(1)*
unterschiedlicher Belastung *(1, 2, 4 , 8 oder 16 Zeilenpaar(e) pro Prozessor)*
bei jeweils größtmöglicher Zeilenlänge *(512, 256, 128, 64 bzw. 32)*

Verglichen sind jeweils obere Schranken für *ideale* Beschleunigungen
- Partialbruchzerlegung mit *log*arithmischer Vektorkombination (etwa Hypercube)
- Partialbruchzerlegung mit *lin*earer Vektorkombination (etwa Kettentopologie)
- alte rekursiv-multiplikative Methode (*rec*ursive)

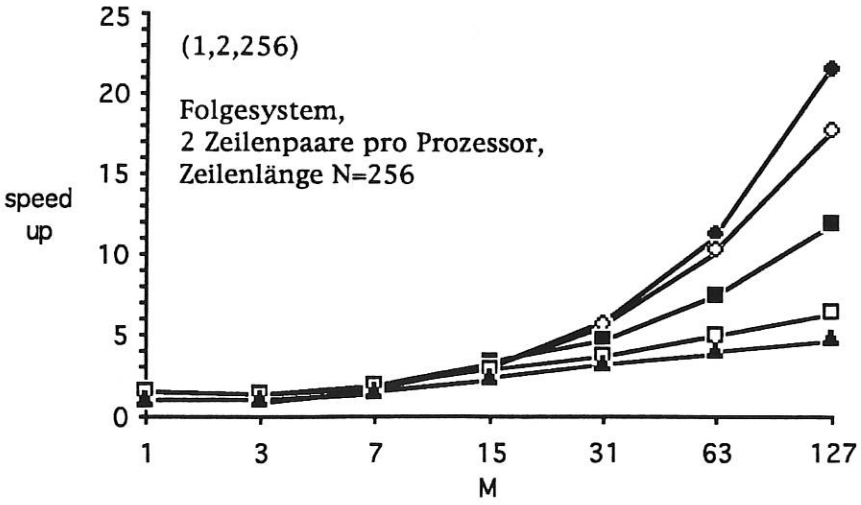
mit tatsächlich auf einem T800 mit 4MB bzw. 32 T800 á 128 KB erreichten Werten
- optimal gestückelte Vektorübertragung bzw. -bearbeitung *(sliced)*
- konventionelle Vektorübertragung *(unsliced)*

Anmerkung: Die Anzahl der jeweils verwendeten Prozessoren ergibt sich aus $\lceil (M+1)/2 \;/\; \#\text{Zeilenpaare pro Prozessor} \rceil$; für das größte M je Diagramm sind dies immer 32, das zweitgrößte 16, usw., mindestens jedoch 1.

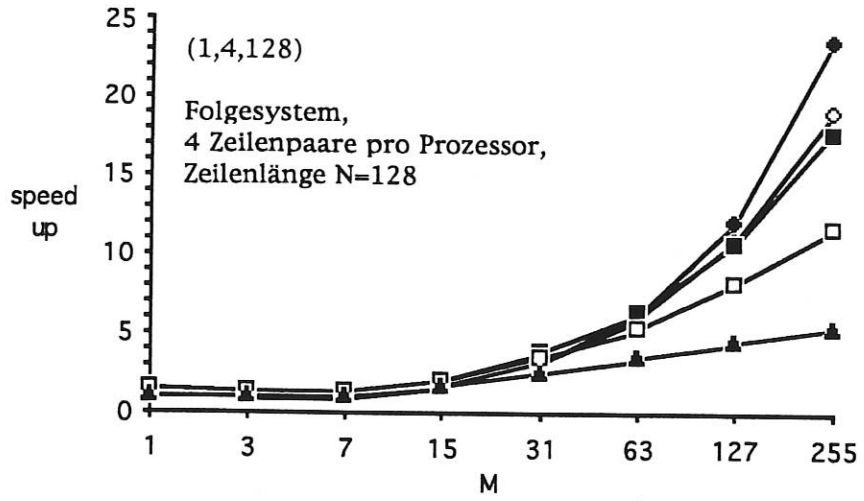
Diag. 2a



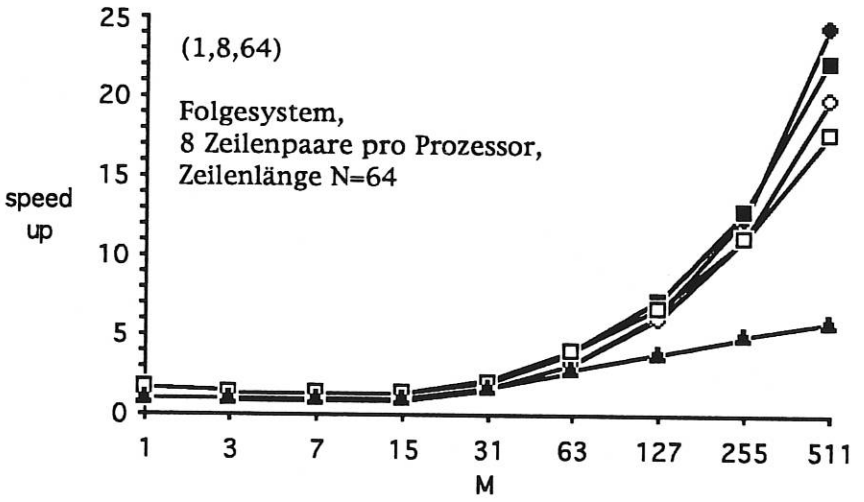
Diag. 2b



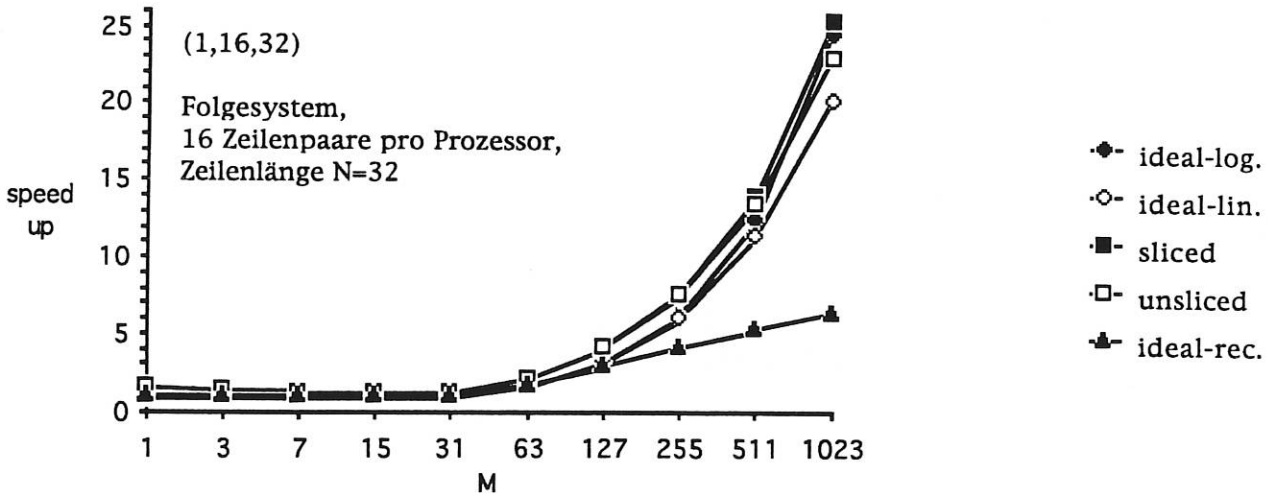
Diag. 2c



Diag. 2d



Diag. 2e



Literaturverzeichnis

1. Buzbee B., Golub G., Nielson C., On direct methods for solving Poisson's equations, SIAM J. Numer. Anal. 7 (No. 4), 1970, 627 - 656

2. Hockney R., Jesshope C., Parallel Computers 2, 1988, 2nd edition, Adam Hilger Ltd., Bristol

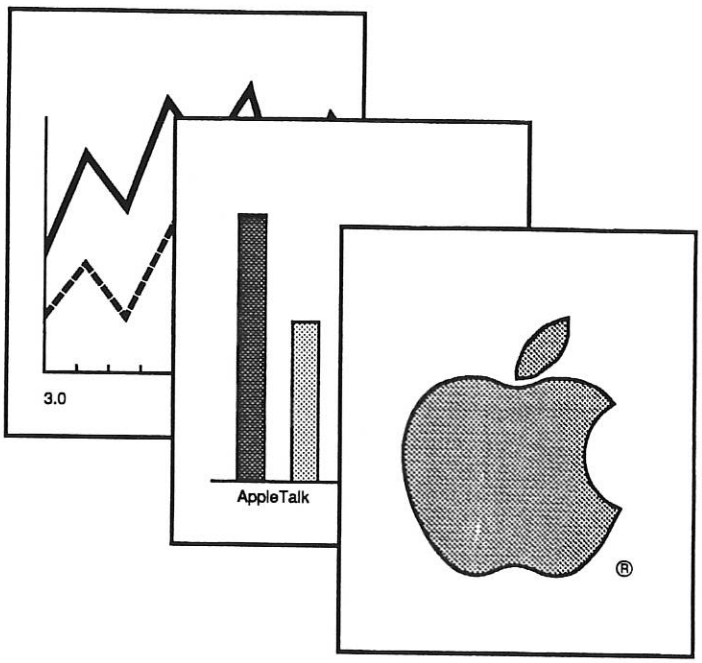
3. Sweet R., A parallel and vector variant of the cyclic reduction algorithm, SIAM J. Sci. Stat. Comput. 9 (No. 4), 1988, 761 - 765

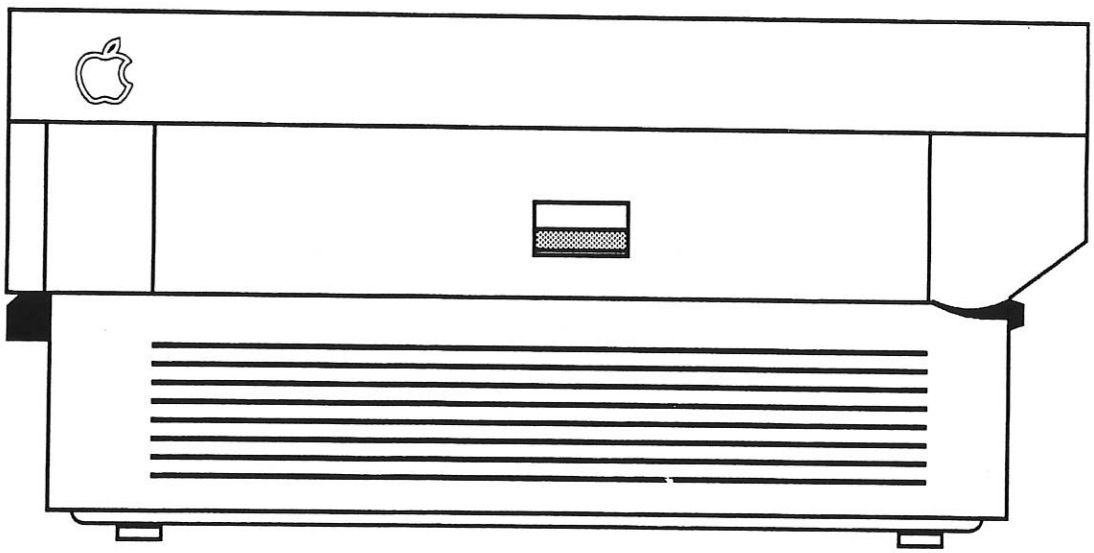
4. Gallopoulos E., Saad Y., A parallel block cyclic reduction algorithm for the fast solution of elliptic equations, Par. Comput. 10, 1989, 143 - 159

5. INMOS Limited, Transputer Technical Notes, 1989, Prentice Hall, New York - London - Toronto - Sydney - Tokyo

6. Boreddy J., Paulray A., On the performance of transputer arrays for dense linear systems, Par. Comput. 15, 1990, 107 - 117

LaserWriter Plus





31627