Der Block-Huang-Algorithmus

Udo Schendel Joachim Tews

Forschungsprojekt *Parallele Numerik*
Fachbereich Mathematik und Informatik
Freie Universität Berlin 1999

Vorwort

Lineare Gleichungssysteme gehören zu den elementaren und essentiellen Problemen der Natur-/ Geisteswissenschaften. Es ist unmöglich, alle Anwendungsgebiete aufzuzählen; einige Beispiele werden in [Schendel(5)] und in [Duff] beschrieben.

Diese Arbeit befaßt sich mit parallelen Verfahren zur Lösung von linearen Gleichungssystemen. Sie entstand in dem Forschungsprojekt "Parallele Numerik" unter Leitung von Herrn Prof. Dr. Schendel.

Im Text wird weitestgehend die vor dem 1. Juli 1999 in Berlin geltende deutsche Rechtschreibung verwendet.

Die Querverweise der Form K.A.N**S**S innerhalb dieses Textes enthalten die Kapitelnummer K, die Abschnittsnummer A, eine fortlaufende Nummer N und die Seitenzahl S.
Die Angabe der Seitenzahl am Ende der Referenz sollte das Auffinden erleichtern. Sofern diese Arbeit als "pdf-Datei" vorliegt, kann man durch Mausklick auf diese Seitenzahl S zur entsprechenden Seite S "springen".

Udo Schendel und Joachim Tews im Oktober 1999

Inhaltsverzeichnis

1 Zusammenfassung

Diese Arbeit beschreibt den Block-Huang-Algorithmus, der sich zur Lösung von linearen Gleichungssystemen auf massiv-parallelen Rechnern eignet.

Ausgehend vom **ursprünglichen Huang-Algorithmus** werden der **modifizierte Huang-Algorithmus**, der **Block-Huang-Algorithmus**, der **modifizierte Block-Huang-Algorithmus**, der **Sparse-Huang-Algorithmus**, der **Sparse-Block-Huang-Algorithmus**, der **parallele Block-Huang-Algorithmus** und der **parallele Sparse-Block-Huang-Algorithmus** hergeleitet und untersucht.

Es werden sowohl theoretische Erkenntnisse, als auch praktische Tests präsentiert. Insbesondere die parallele Programmierung wird ausführlich erläutert. Es ist ein Anliegen, zum besseren Verständnis die Programme mitanzugeben.

In den Ergänzungen wird außerdem das **modifizierte Block-Schmidt-Orthogonalisierungsverfahren** definiert.

Die Querverweise der Form K.A.N**S**S innerhalb dieses Textes enthalten die Kapitelnummer K, die Abschnittsnummer A, eine fortlaufende Nummer N und die Seitenzahl S.
Die Angabe der Seitenzahl am Ende der Referenz sollte das Auffinden erleichtern. Sofern diese Arbeit als "pdf-Datei" vorliegt, kann man durch Mausklick auf diese Seitenzahl S zur entsprechenden Seite S "springen".

2 Serieller Block-Huang-Algorithmus

2.1 Ursprünglicher/Modifizierter Huang-Algorithmus

Der ursprüngliche Huang-Algorithmus (siehe [Huang]) stellt ein Verfahren zur Lösung von linearen Gleichungssystemen dar.

Definition 2.1.1 (Ursprünglicher Huang-Algorithmus) :

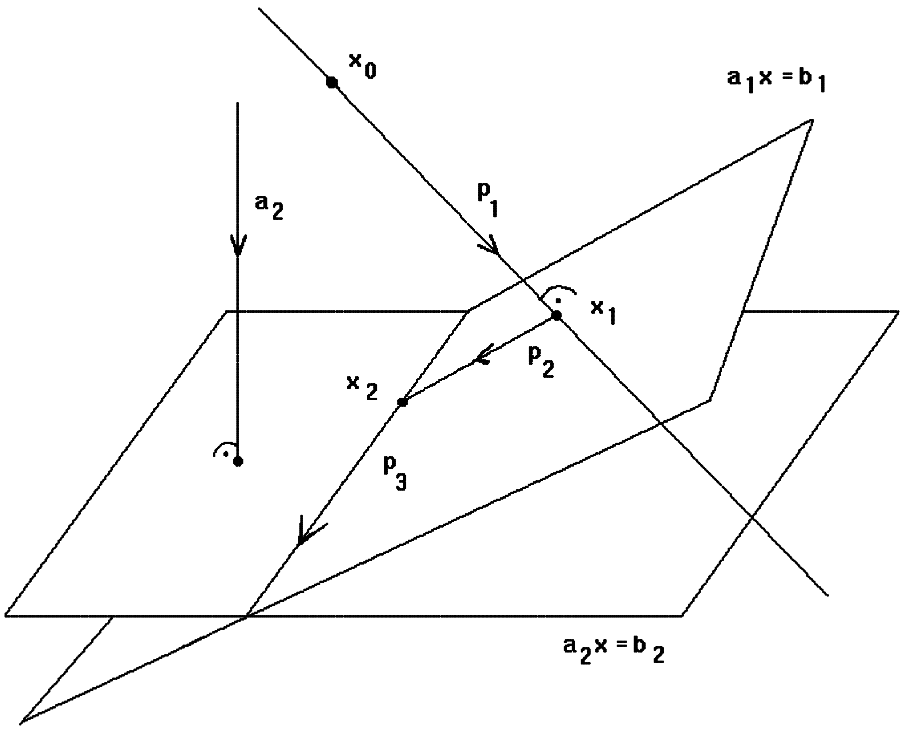
Sei das lineare Gleichungssystem $Ax = b = (b_1, b_2, \ldots, b_n)^T$
mit $x, b \in \mathbb{R}^n$ und $A \in \mathbb{R}^{n\times n}$ gegeben, A sei invertierbar,
I_n sei die n-dimensionale Einheitsmatrix,
$a_i = (a_{i1}, a_{i2}, \ldots, a_{in})$ bezeichne die i. Zeile von A und
$r_i(x)$ die i. Komponente des Residuums: $r_i(x) := a_i x - b_i$.
Definiere ferner $x_0 := 0 \in \mathbb{R}^n, H_0 := I_n$
und für $i = 1, 2, \ldots, n$:
$p_i := H_{i-1} a_i^T, \ x_i := x_{i-1} - \dfrac{r_i(x_{i-1})}{a_i p_i} p_i,$
$H_i := H_{i-1} - \dfrac{1}{a_i p_i} p_i p_i^T.$

Der Huang-Algorithmus gehört zur Klasse der Projektionsmethoden und besitzt die folgende geometrische Interpretation (siehe Abbildung 2.1.2S3):
Der Startpunkt x_0 wird in Richtung von p_1 auf die Hyperebene $a_1 x = b_1$ projiziert. Weil a_1 nicht senkrecht auf p_1 steht($a_1 p_1 \neq 0$) und a_1 senkrecht auf der Hyperebene steht, schneidet die Gerade durch den Punkt x_0 mit dem Richtungsvektor p_1 die Hyperebene $a_1 x = b_1$ in einem im Endlichen liegenden Punkt x_1. Diese Vorstellung läßt sich für die Punkte $x_1, \ldots, x_{n-1}$ wiederholen.
Insbesondere ist:

$$p_2 = H_1 a_2^T = (H_0 - \frac{1}{a_1 p_1} p_1 p_1^T) a_2^T = (I_n - \frac{1}{a_1 a_1^T} a_1^T a_1) a_2^T = a_2^T - \frac{a_1 a_2^T}{a_1 a_1^T} a_1^T$$

Hierin erkennt man einen Schritt des Schmidtschen Orthogonalisierungsverfahrens wieder. Wesentlich ist nun, daß dank der Projektionseigenschaften der Matrizen H_i ($H_{i-1} a_j^T \neq 0$ und linear unabhängig für $i \leq j \leq n$ und $H_i a_j^T = H_i p_j = 0$ für $1 \leq j \leq i$) die Richtungsvektoren p_i und die Punkte x_{i-1} in der Schnittmenge der Hyperebenen $a_1 x = b_1, \ldots, a_{i-1} x = b_{i-1}$ liegen. Der Schnittpunkt x_n aller Hyperebenen ist die gesuchte Lösung des linearen Systems.

Abbildung 2.1.2 (Geometrische Interpretation) :



Obgleich der **Huang-Algorithmus** schon sehr robust ist, haben die numerischen Erfahrungen von [Abaffy/Spedicato] gezeigt, daß die folgende Modifikation zusätzliche Vorteile besitzt.

Definition 2.1.3 (Modifizierter Huang-Algorithmus) :

Der modifizierte Huang-Algorithmus unterscheidet sich vom Huang-Algorithmus aus Def. 2.1.1S2 nur in der Defintion der Suchrichtungen p_i, der Aufdatierung der x_i und der Projektionsmatrizen H_i :

$$p_i := H_{i-1}(H_{i-1}a_i^T), \; x_i := x_{i-1} - \frac{r_i(x_{i-1})}{p_i^T p_i} p_i,$$

$$H_i := H_{i-1} - \frac{1}{p_i^T p_i} p_i p_i^T.$$

Der **modifizierte Huang-Algorithmus** ist dem **Huang-Algorithmus** in exakter Arithmetik äquivalent; dies erkennt man durch Betrachtung der Eigenschaften: $H_{i-1}^2 = H_{i-1}$, d.h. H_{i-1} ist idempotent und $a_i p_i = p_i^T p_i > 0$.

2.2 Block-Huang-Algorithmus

Der Block-Huang-Algorithmus stellt eine Weiterentwicklung des ursprünglichen Huang-Algorithmus dar. Die Idee, durch Blockbildung zu Algorithmen mit besonderen Eigenschaften zu gelangen, ist nicht neu (vergleiche z.B. [BrandSchen], [Schendel(1)] bis [Schendel(4)], die dort angegebene Literatur oder [DufEriRei]).
Die Blockbildung erfolgt hier ausschließlich zeilenweise, d.h. die Koeffizientenmatrix und die rechte Seite werden in mehrere gleich große Zeilenblöcke unterteilt:

$$Ax = \begin{pmatrix} \tilde{A}_1 \\ \tilde{A}_2 \\ \dots \\ \tilde{A}_{\frac{n}{m}} \end{pmatrix} x = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \dots \\ \tilde{b}_{\frac{n}{m}} \end{pmatrix} = b \Longleftrightarrow \forall\ 1 \le i \le \frac{n}{m} : \tilde{A}_i x = \tilde{b}_i$$

Algorithmus 2.2.1 (Block-Huang-Algorithmus) :

Problem: $Ax = b, \qquad x \in \mathbb{R}^{n\times q}$ gesucht.

Vor.: $m, n, q \in \mathbb{N}, \quad b \in \mathbb{R}^{n\times q}$,
$A \in \mathbb{R}^{n\times n},$ Rang $A = n$.

Vor. (*): m Teiler von n.

Lösung: $x_0 = 0 \in \mathbb{R}^{n\times q},\ H_0 = I_n \in \mathbb{R}^{n\times n}$.
Für $k = 1, \dots, \frac{n}{m}$:

$$\begin{aligned} p_k &= H_{k-1}\tilde{A}_k^T && \in \mathbb{R}^{n\times m} \\ \gamma_k &= p_k^T p_k && \in \mathbb{R}^{m\times m} \\ \alpha_k &= \tilde{A}_k x_{k-1} - \tilde{b}_k && \in \mathbb{R}^{m\times q} \\ x_k &= x_{k-1} - p_k(\gamma_k^{-1}\alpha_k) && \in \mathbb{R}^{n\times q} \\ H_k &= H_{k-1} - p_k(\gamma_k^{-1}p_k^T) && \in \mathbb{R}^{n\times n} \end{aligned}$$

Bem.: $\tilde{b}_k \ \in \mathbb{R}^{m\times q}$ enthält die k. m Zeilen von b,
$\tilde{A}_k \ \in \mathbb{R}^{m\times n}$ enthält die k. m Zeilen von A :

$$\tilde{A}_k = \begin{pmatrix} a_{(k-1)m+1} \\ a_{(k-1)m+2} \\ \dots \\ a_{(k-1)m+m-1} \\ a_{km} \end{pmatrix} \text{ mit } a_j\ j\text{. Zeile von } A.$$

Für $m = q = 1$ ergibt sich der ursprüngliche Huang-Algorithmus.
Bitte x_k nicht mit der k. Komponente von x verwechseln !
$x_{\frac{n}{m}} = A^{-1}b$ (siehe Satz 2.3.1S5).
Im letzten Schritt muß die Matrix $H_{\frac{n}{m}}$ nicht berechnet werden, kann aber zur Kontrolle dienen, weil $H_{\frac{n}{m}} = 0$ ist.
In der Praxis werden die Produkte $\gamma_k^{-1}\alpha_k$ und $\gamma_k^{-1}p_k^T$ simultan durch
1.) Cholesky-Zerlegung von $p_k^T p_k = LL^T$ mit $L \in \mathbb{R}^{m\times m}$ Dreiecksmatrix
2.) Löse $L(L^T W) = (p_k^T|\alpha_k) \in \mathbb{R}^{m\times(n+q)}$
bestimmt.

2.3 Eigenschaften

Der folgende Satz stellt sicher, daß der Block-Huang-Algorithmus anwendbar ist, d.h. für beliebige (eindeutig lösbare) Gleichungssysteme in endlich vielen Schritten die (eindeutige) Lösung liefert.

Satz 2.3.1 (Anwendbarkeit des Block-Huang-Algorithmus) :

Vor.: Sei der Block-Huang-Algorithmus nach Def. 2.2.1S4 gegeben.

Beh.: Er liefert nach $\frac{n}{m}$ Schritten die Lösung des Systems.

Bew.: Zunächst führen wir die Konvention ein, daß Größen ohne Index den Index k besitzen sollen. Ebenso steht der Index $-$ für $k-1$.
Sodann definieren wir noch die Matrizen p^{T*} und z.
Ein Schritt des Algorithmus lautet nun:

$(S1)\quad p = H_-\tilde{A}^T$
$(S2)\quad \alpha = \tilde{A}x_- - \tilde{b}$
$(S3)\quad p^{T*} = p(p^Tp)^{-1} \in \mathbb{R}^{n\times m}$
$(S4)\quad z = p^{T*}p^T = pp^{T*T}$
$(S5)\quad H = H_- - z$
$(S6)\quad x = x_- - p^{T*}\alpha$

Wir bemerken noch, daß p^{T*} die Moore-Penrose-Inverse zu p^T ist, benutzen diese Eigenschaft aber nicht explizit.
Die Behauptung des Satzes wird mit beschränkter Induktion bewiesen.
Der Induktionsanfang ist trivial.
Im Induktionsschritt setzen wir voraus:

$a)\quad H_-H_- = H_- = H_-^T$
$b)\quad \forall\ 1 \le j \le (k-1)m:\ a_jx_- = b_j \wedge a_jH_- = 0$
$c)\quad \forall\ (k-1)m < j \le n:\ H_-a_j^T \ne 0$ und linear unabhängig

Wir vollziehen den Induktionsschritt in 8 Teilschritten:

1. Rang $p = m$, also ist (S3) mit p^{T*} definiert.
2. $H_-z = zH_- = zz = z = z^T$
3. $HH = H = H^T$
4. $\tilde{A}x = \tilde{b}$
5. $\forall\ 1 \le j \le (k-1)m:\ a_jx = b_j$
6. $\forall\ 1 \le j \le (k-1)m:\ a_jH = 0$
7. $\tilde{A}H = 0$
8. $\forall\ km < j \le n:\ Ha_j^T \ne 0$ und linear unabhängig

1. ist eine Folgerung aus Teil c), weil die Spalten von p nach c) linear unabhängig sind.
3. ergibt den Induktionsschritt zu Teil a).
4.-7. ergeben Teil b) und 8. ergibt Teil c).
Im einzelnen sehen wir ein:

2. $H_-z = H_-p(p^Tp)^{-1}p^T = H_-H_-\tilde{A}^T(p^Tp)^{-1}p^T =$
$H_-\tilde{A}^T(p^Tp)^{-1}p^T = p(p^Tp)^{-1}p^T = z$
$zH_- = p^{T*}p^TH_- = p^{T*}(H_-\tilde{A}^T)^TH_- = p^{T*}\tilde{A}H_-^TH_- =$
$p^{T*}\tilde{A}H_- = p^{T*}\tilde{A}H_-^T = p^{T*}p^T = z$
$zz = p(p^Tp)^{-1}p^Tp(p^Tp)^{-1}p^T = p(p^Tp)^{-1}p^T = z$
$z^T = z$ klar.
3. $HH = (H_- - z)(H_- - z) = H_-H_- - zH_- - H_-z + zz =$
$H_- - z = H$ nach Teilschritt 2. und Vor. a).
$H^T = H_-^T - z^T = H_- - z = H$
4. $I_m = (\tilde{A}H_-\tilde{A}^T)(\tilde{A}H_-\tilde{A}^T)^{-1} = \tilde{A}H_-\tilde{A}^T(\tilde{A}H_-^TH_-\tilde{A}^T)^{-1} =$
$\tilde{A}H_-\tilde{A}^T((H_-\tilde{A}^T)^TH_-\tilde{A}^T)^{-1} = \tilde{A}p(p^Tp)^{-1}$,
also $\tilde{A}p^{T*} = I_m = I_m^T = (p^{T*T}\tilde{A}^T)^T = p^{T*T}\tilde{A}^T \Longrightarrow$
$0 = \alpha - \tilde{A}p^{T*}\alpha = \tilde{A}x_- - \tilde{b} - \tilde{A}p^{T*}\alpha =$
$\tilde{A}(x_- - p^{T*}\alpha) - \tilde{b} = \tilde{A}x - \tilde{b}$
5. Sei a_j eine Zeile von A, die in vorhergehenden Blockschritten berücksichtigt wurde.
Nach Induktionsvor. b) gilt:
$a_jH_- = 0 \wedge a_jx_- - b_j = 0 \Longrightarrow$
$0 = a_jx_- - b_j - a_jH_-\tilde{A}^T(p^Tp)^{-1}\alpha = a_jx - b_j$.
6. Gelte $1 \leq j \leq (k-1)m$.
$a_jH = a_jH_- - a_jz = a_jH_- - a_jpp^{T*T} =$
$a_jH_- - a_jH_-\tilde{A}^Tp^{T*T} = a_jH_-(I_n - \tilde{A}^Tp^{T*T}) = 0$
wegen Induktionsvor. b).
7. $\tilde{A}z = \tilde{A}p^{T*}p^T = I_mp^T = p^T \Longrightarrow$
$\tilde{A}H = \tilde{A}H_- - \tilde{A}z = p^T - p^T = 0$,
also $a_jH = 0$ für $(k-1)m < j \leq km \Longrightarrow$
$H\tilde{A}^T = 0$, d.h. 8. ist nur für $j > km$ sinnvoll !
8. Sei $km < j \leq n$.
Ziel: $Ha_j^T \neq 0$.
$Ha_j^T = H_-a_j^T - \Sigma$ mit $\Sigma := H_-\tilde{A}^Tw,\ w := (p^Tp)^{-1}p^Ta_j^T$
$H_-\tilde{A}^T = (H_-\tilde{A}_1^T, \ldots, H_-\tilde{A}_m^T)$ mit $\tilde{A}_\mu^T$ ist μ. Spalte von $\tilde{A}^T$.
$\Sigma = w_1H_-\tilde{A}_1^T + \ldots + w_mH_-\tilde{A}_m^T$ ist Linearkombination von
$H_-a_\nu^T,\ \nu \in \{(k-1)m+1, \ldots, km\} \Longrightarrow$
Σ ist linear unabhängig von $H_-a_j^T$ nach Induktionsvor. $\diamondsuit$
Die lineare Unabhängigkeit der Ha_j^T ergibt sich so:
$Ha_j^T = 0 \Longrightarrow 1 \leq j \leq km$.
Allora: Kern $H \subset \mathcal{M} :=$ span $\{a_1^T, \ldots, a_{km}^T\}$.
Teilschritt 6. und 7. bedeuten: $\mathcal{M} \subset$ Kern H,
insgesamt: Kern $H = \mathcal{M}$.
Nun gilt $\mathbb{R}^n =$ Bild $H \oplus$ Kern H,
weil H Projektion im endlichdimensionalen Vektorraum ist.
Die Einschränkung $H|_{\text{Bild } H}$ ist Isomorphismus.
$(a_j^T)_{km<j\leq n}$ ist Basis von Bild H.
Somit sind auch Ha_j^T für $km < j \leq n$ linear unabhängig.

Nach Vollzug der Induktion folgt die Beh. aus Teil b).$_{\text{qed}}$

2.4 Modifizierter Block-Huang-Algorithmus

Die Idee, durch Reprojektion (wiederholte Projektion) eine verbesserte Stabilität des usprünglichen Huang-Algorithmus zu erreichen, läßt sich auf den Block-Huang-Algorithmus übertragen.

Definition 2.4.1 (Modifizierter Block-Huang-Algorithmus) :

Der modifizierte Block-Huang-Algorithmus unterscheidet sich vom Block-Huang-Algorithmus 2.2.1S4 nur durch die Definition der Matrizen p_k:

$p_k = H_{k-1}\tilde{A}_k^T$ wird ersetzt durch

$p_k = H_{k-1}(H_{k-1}\tilde{A}_k^T)$.

Der "neue" Algorithmus ist dem "alten" in exakter Arithmetik äquivalent, weil die Matrizen H_k Projektoren sind.

Man kann auf die Einschränkung verzichten, die verlangt, daß die Blockgröße m ein Teiler der Dimension n ist. Hierzu kann man den Beweis zu Satz 2.3.1S5 dahingehend verallgemeinern, daß in jedem Schritt eine neue Blockgröße gewählt wird. Man nutzt dies aus und erweitert den Block-Huang-Algorithmus im einfachsten Fall so:

Algorithmus 2.4.2 (Erweiterung des Block-Huang-Algorithmus) :

Man betrachte den Block-Huang-Algorithmus 2.2.1S4.

1. Die Schleife $k = 1, \ldots, \frac{n}{m}$ wird
   durch $k = 1, \ldots, \text{Ganzteil}(\frac{n}{m})$ ersetzt.
2. Der folgende Teil wird an den Algorithmus "angehängt":
   $\tilde{m} = \text{Rest}(\frac{n}{m})$
   Wenn $\tilde{m} \neq 0$, dann:
   $$\begin{aligned}
   k &= \text{Ganzteil}(\tfrac{n}{m}) + 1 & \\
   p_k &= H_{k-1}\tilde{A}_k^T & \in \mathbb{R}^{n\times\tilde{m}} \\
   \gamma_k &= p_k^T p_k & \in \mathbb{R}^{\tilde{m}\times\tilde{m}} \\
   \alpha_k &= \tilde{A}_k x_{k-1} - \tilde{b}_k & \in \mathbb{R}^{\tilde{m}\times q} \\
   x_k &= x_{k-1} \quad - p_k(\gamma_k^{-1}\alpha_k) & \in \mathbb{R}^{n\times q}
   \end{aligned}$$

Hierbei enthält $\tilde{A}_k$ die letzten noch fehlenden Zeilen von A.

Der modifizierte Block-Huang-Algorithmus läßt sich natürlich ebenso erweitern.

Wenn man nicht kontrollieren möchte, ob $H_{\frac{n}{m}}$ numerisch gleich der Nullmatrix ist, dann kann man auf die letzte Aufdatierung verzichten. Außerdem kann man in den vorgestellten Algorithmen berücksichtigen, daß H_0 gleich der Identität ist und somit unnötige Operationen einsparen.

2.5 Programmierung

Der Block-Huang-Algorithmus läßt sich, ebenso wie der ursprüngliche Huang-Algorithmus, programmieren, indem man schön brav drei Schleifchen für jede Matrix-Matrix-Multiplikation entweder in C oder in Fortran programmiert. Man sollte jedoch stattdessen besser eine Linear-Algebra-Library verwenden, dies erhöht zunächst die Lesbarkeit des Codes. Außerdem stellt sich heraus (siehe Abschnitt "Tests" ab Seite 9), daß selbst die Verwendung einer nicht rechnerspezifisch optimierten Bibliothek einen Performance Vorteil einbringt.

Die BLAS-Library (Basic-Linear-Algebra-Software) aus dem Lapack (siehe [Lapack]) hat sich bewährt und als Standard durchgesetzt, an dem man nicht mehr vorbeikommt. Sie ist auch in einer C-Version verfügbar (CBLAS). Mit Hilfe dieses CLAPACK haben wir den Block-Huang-Algorithmus auf einem PC (Borland-C) und auf einer Sun-Workstation getestet.

Leider steht keine rechnerspezifisch optimierte BLAS-Version im Institut zur Verfügung. Es gibt mit Sicherheit eine, weil inzwischen alle Hersteller das Lapack-Projekt unterstützen und bemüht sind, eine maximale Performance im Bereich der linearen Algebra dadurch nachzuweisen, daß sie speziell "getunte" BLAS-Routinen anbieten.

Lapack hat sich nicht nur auf seriellen Rechnern bewährt, sondern erobert mittlerweile auch die parallelen Computer. Lapack liegt in einer optimierten Version für Cray-Vektorrechner vor. Der modifizierte Huang-Algorithmus wurde bereits mit Erfolg vektorisiert (siehe [Bertocchi/Spedicato]). Bemerkenswert ist, das man auch mit den modernen, vektorisierenden Compilern nicht die Perfomance erreicht wie durch die Verwendung geeigneter Bibliotheken. "Handtuning" ist also immer noch vonnöten.

Hätte man die Tests des Block-Huang-Algorithmus zunächst auf einem PC durchgeführt, dann hätte man sich nicht weiter für diesen Algorithmus interessiert, weil sich herausstellt, daß der modifizierte Block-Huang-Algorithmus im seriellen Fall bei Verwendung einer nicht optimierten BLAS-Version keine Geschwindigkeits-Vorteile gegenüber dem modifizierten Huang-Algorithmus bietet. Dies kann man sich auch dadurch verdeutlichen, daß man die Rechenoperationen zählt und somit theoretisch feststellt, daß der Block-Huang-Algorithmus für m=1 am schnellsten ist (siehe hierzu Tabelle 4.1.2S28).

Ganz anders liegt der Fall jedoch auf der Cray T3E. Wenn man nur einen Prozessor benutzt, dann ist der modifizierte Block-Huang-Algorithmus, aufgrund der optimierten BLAS-Routinen, deutlich schneller als der modifizierte Huang-Algorithmus (siehe Abschnitt "Tests" ab Seite 9).

2.6 Tests

Um den modifizierten Block-Huang-Algorithmus intensiv zu prüfen, werden spezielle lineare Gleichungssysteme erzeugt und anschließend gelöst. Man kann dann die exakte Lösung mit der Näherung vergleichen.

Hierzu werden vollbesetzte Matrizen und Lösungsvektoren mit ganzzahligen Pseudo-Zufallszahlen erzeugt und die dazugehörigen rechten Seiten berechnet. Strenggenommen weiß man nicht, ob das System eindeutig lösbar ist, aber im Falle eines geringen Abstands von der Näherung zum erzeugten Lösungsvektor kann man ziemlich sicher sein.

Man erhält nun die in den folgenden Grafiken dargestellten Ergebnisse, die nicht weiter kommentiert werden müssen (vergleiche Abschnitt "Programmierung"). Beachtenswert sind insbesondere die Resultate zur Genauigkeit, die zeigen, daß der modifizierte Block-Huang-Algorithmus besser als der modifizierte Huang-Algorithmus sein kann. Essentiell ist hierbei jedoch die schon am ursprünglichen Algorithmus angebrachte Modifikation der doppelten Multiplikation (Reprojektion) mit der Projektionsmatrix H.

Erläuterung der Tabelle:

In der folgenden Tabelle werden benötigte Zeit und erzielte Genauigkeit für ein Gleichungssystem der Dimension 1801 angegeben.
Die Berechnungen werden auf dem massiv-parallelen Rechner Cray T3E (nur ein Prozessor !) und auf dem Vektorrechner Cray J932 durchgeführt.
Die Maschinengenauigkeit beträgt ca. 10^{-15}.

Rechner	Verfahren	Zeit in s	Genauigkeit
T3E	modifizierter Alg. mit Level 1/2 Blas	2.570527e+02	9.138024e-12
T3E	mod. Block-Algorithmus, Blockgröße 63	7.080302e+01	1.812595e-11
J932	modifizierter Alg. mit Level 1/2 Blas	1.580191e+02	1.888719e-08
J932	mod. Block-Algorithmus, Blockgröße 63	1.890716e+02	1.918715e-08

Legende zu den Abbildungen:

Die vorgestellten Daten beziehen sich auf ein repräsentatives Gleichungssystem der Dimension 513.
Die Blockgröße wird auf der x-Achse annähernd logarithmisch dargestellt. Insbesondere bedeuten:

- 1a modifizierter Huang-Algorithmus, mit Schleifen programmmiert.
- 1b modifizierter Huang-Algorithmus, mit Level 1/2 Blas.
- 1c mod. Block-Huang-Algorithmus, Blockgröße 1, mit Level 2/3 Blas.

Die Zeit bzw. Genauigkeit wird auf der y-Achse linear dargestellt.

Abbildung 2.6.1 (PC / Borland C) :

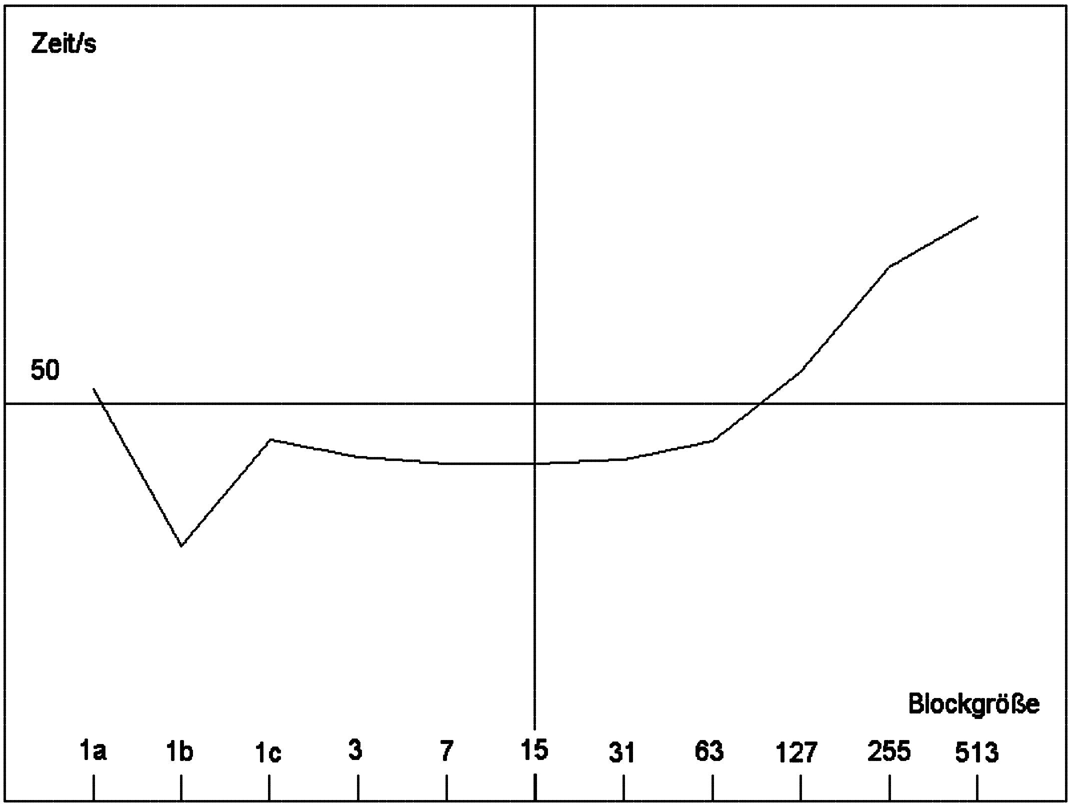


Abbildung 2.6.2 (Cray Vektorrechner) :

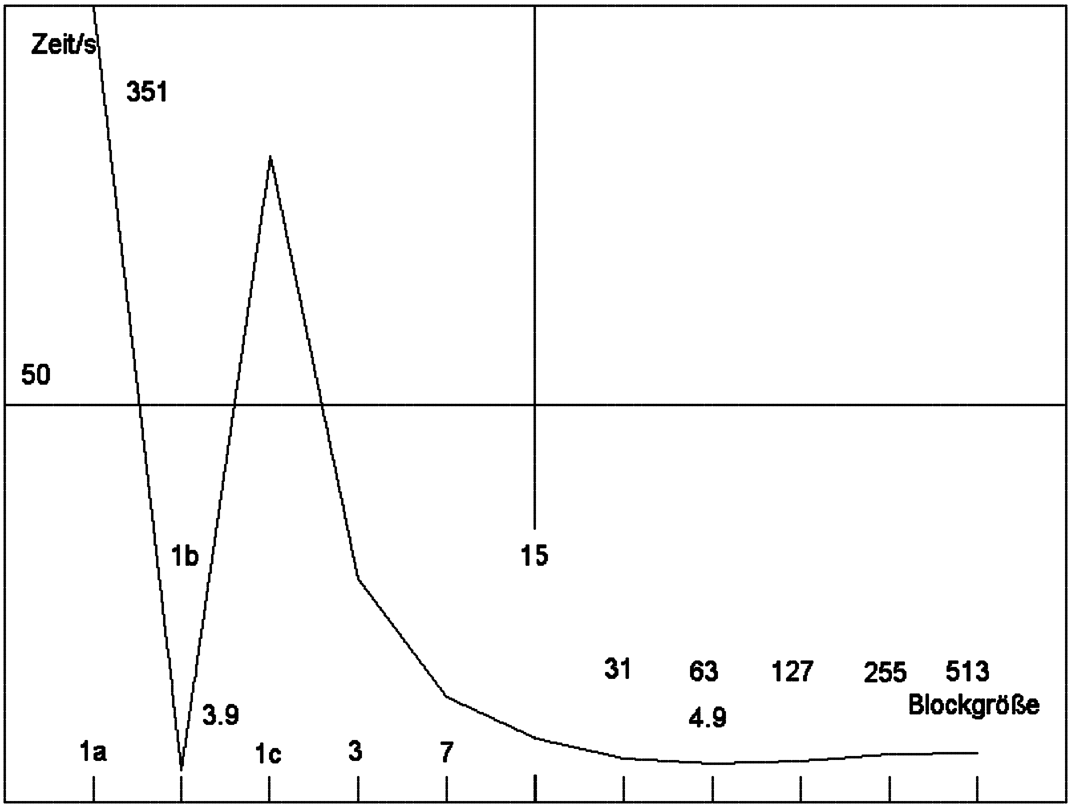


Abbildung 2.6.3 (Cray T3E) :

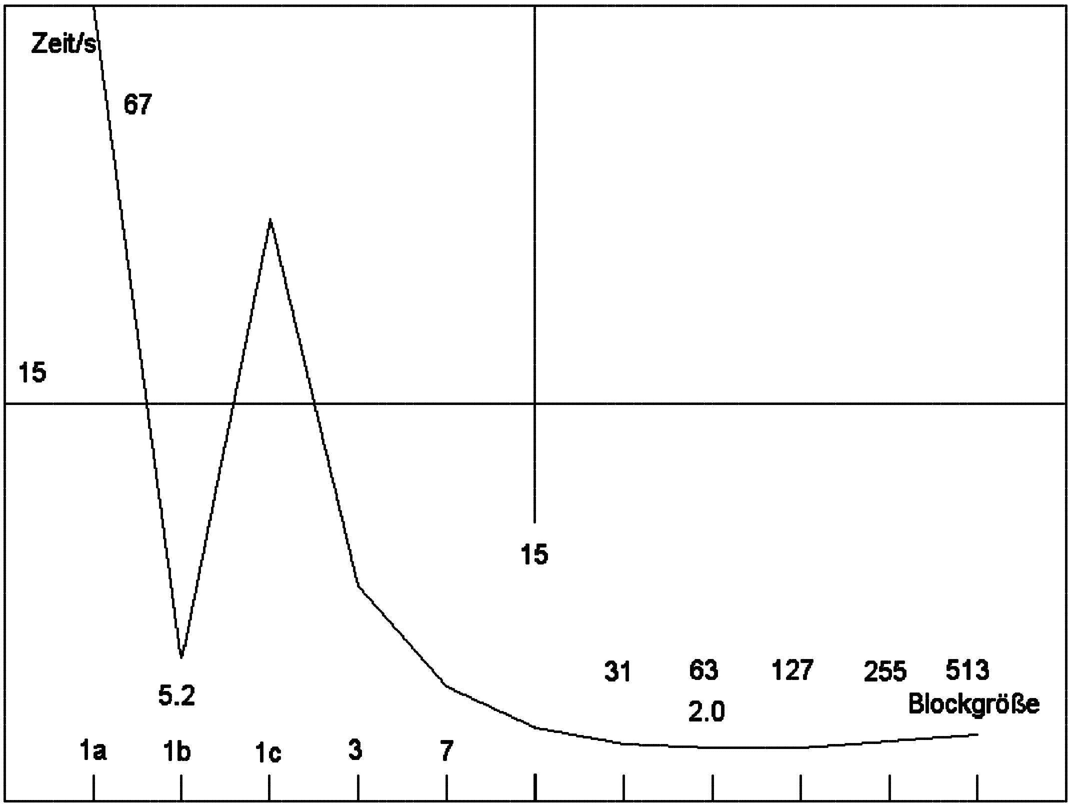


Abbildung 2.6.4 (PC / Borland C Genauigkeit) :

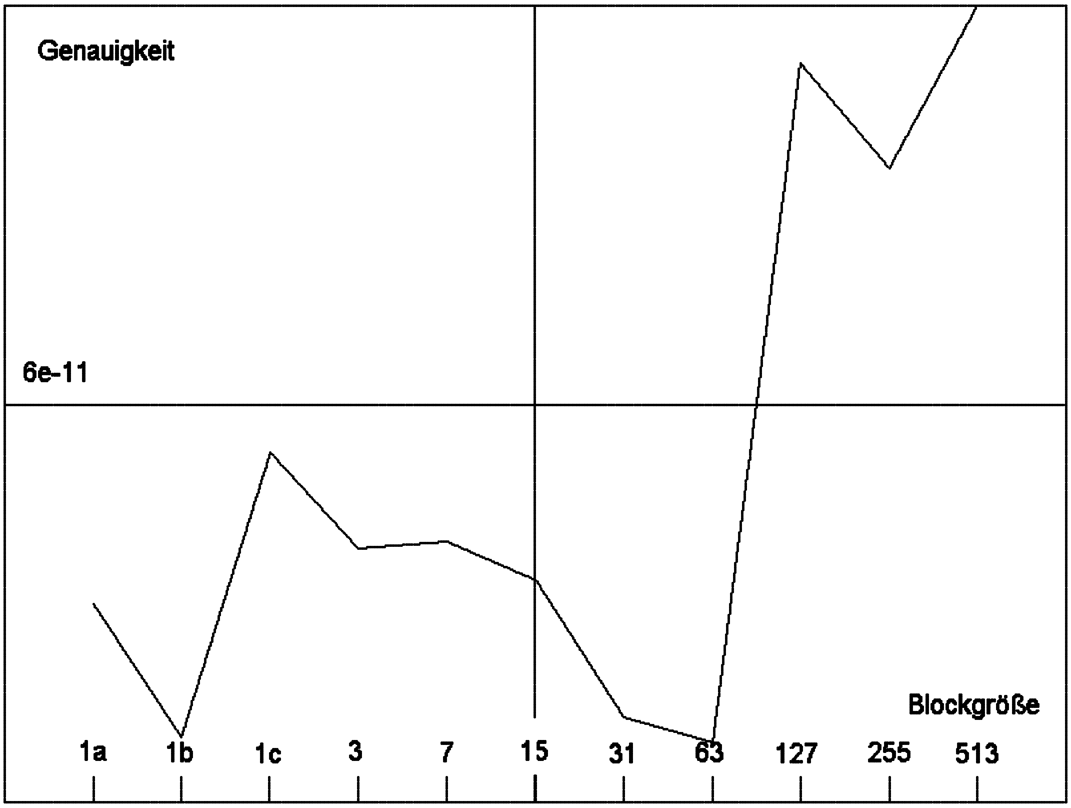


Abbildung 2.6.5 (Cray Vektorrechner Genauigkeit) :

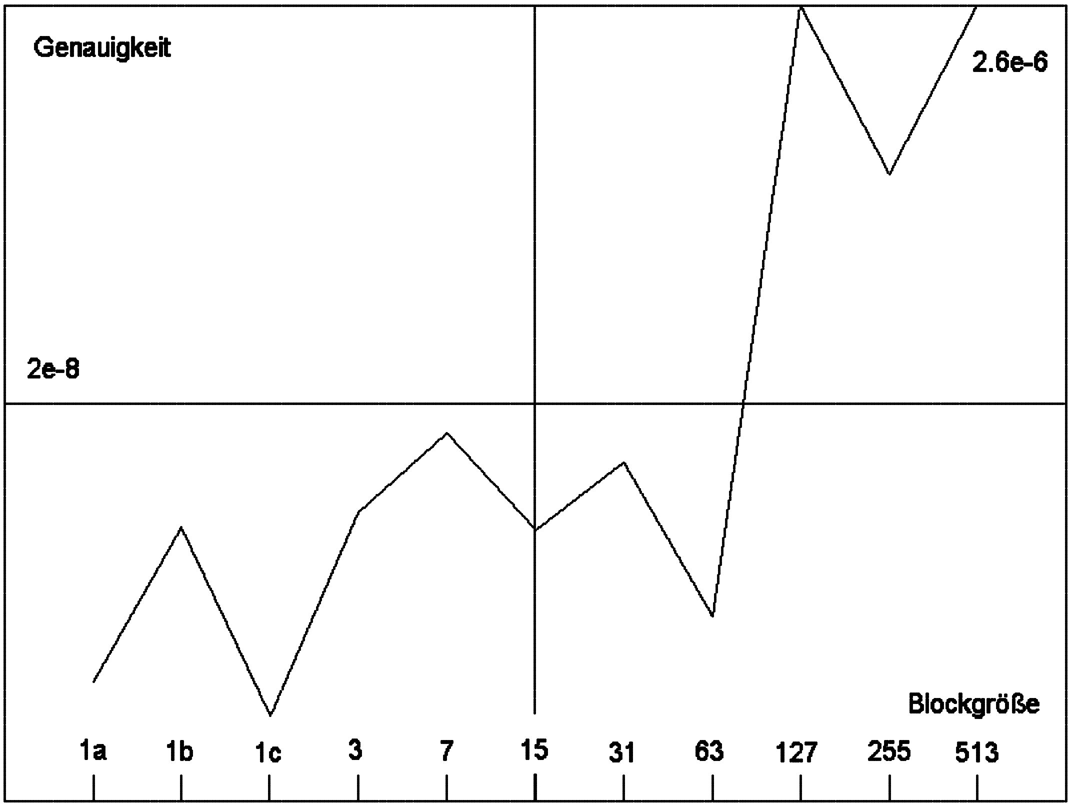
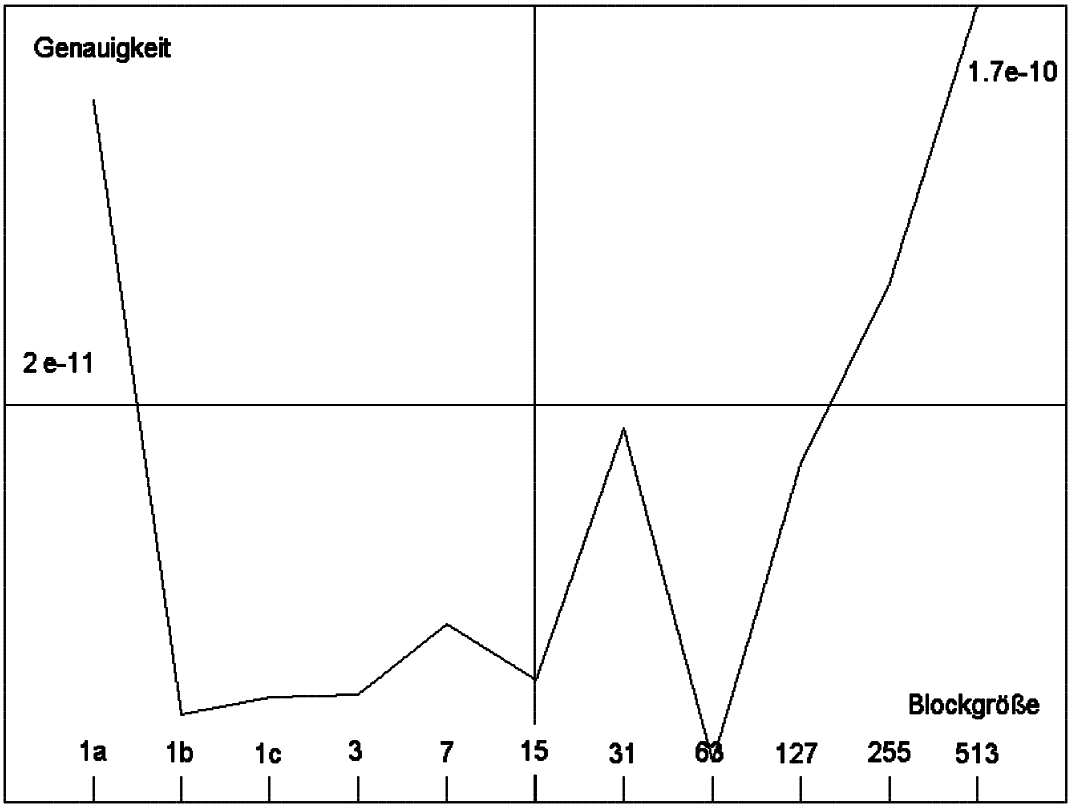


Abbildung 2.6.6 (Cray T3E Genauigkeit) :



3 Serieller Sparse-Block-Huang-Algorithmus

3.1 Grundlagen

Die in den Anwendungen auftretenden Matrizen sind oftmals dünn besetzt (**sparse**), d.h. sehr viele Elemente sind Null (siehe [Schendel(6)] oder [Duff]).

Definition 3.1.1 (Besetztheitsgrad Bg) :

Der Besetztheitsgrad Bg einer $m \times n$-Matrix ist definiert als der Quotient $\text{Bg} = \dfrac{\text{Anzahl der Nichtnullelemente}}{m * n}$.
Er wird oft in % angegeben.

Das Ziel einer effizienten Behandlung von linearen Gleichungssystemen mit Sparse-Matrizen besteht in einer wesentlichen Reduktion der notwendigen Rechenoperationen. Wir messen dies mittels

Definition 3.1.2 (Sparse-Speedup SS) :

Seien ein lineares Gleichungssystem mit Sparse-Matrix, ein die Dünnbesetztheit berücksichtigender Algorithmus und ein Algorithmus für vollbesetzte (**dense**) Matrizen zur Lösung gegeben.
T_A bezeichne die Rechenzeit oder die Anzahl der Rechenoperationen von Algorithmus A.
Das Sparse-Speedup ist definiert als der Quotient $\text{SS} = \dfrac{T_{\text{dense}}}{T_{\text{sparse}}}$.

Algorithmen, die Matrix-Vektor-Produkte mit A bzw. A^T beinhalten und keine Änderungen an der Koeffizientenmatrix A vornehmen, sind zur Lösung von Sparse-Systemen geeignet, weil man nur die Nichtnullelemente bei der Multiplikation berücksichtigen muß.

Diese wesentliche Erkenntnis einer guten Eigenschaft führt direkt zur Definition entsprechender Algorithmen.

Definition 3.1.3 (MV-Algorithmus) :

Ein Algorithmus zur Lösung von linearen Gleichungssystemen soll MV-Algorithmus genau dann genannt werden, wenn

1. Die Koeffizientenmatrix A kommt <u>nur</u> in Form von Matrix-Vektor-Produkten Ax bzw. A^Tx im Algorithmus vor, d.h. u.a. daß sie nicht verändert wird.
2. Der Aufwand zur Berechnung der Matrix-Vektor-Produkte überwiegt gegenüber den übrigen Anteilen des Algorithmus, sofern die Matrix A unstrukturiert, vollbesetzt und hinreichend groß ist.

Das cg-Verfahren (siehe [Schendel(4)]) ist ein bekanntes Beispiel für derartige Algorithmen. Für MV-Algorithmen läßt sich das erzielbare Sparse-Speedup annähernd angeben:

Lemma 3.1.4 (Sparse-Speedup für MV-Algorithmen) :

Vor.: Seien ein MV-Algorithmus (siehe 3.1.3S13) und ein Algorithmus für Dense-Matrizen gegeben.

Beh.: Das Sparse-Speedup (siehe 3.1.2S13) beträgt $SS \approx \frac{1}{\mathrm{Bg.}}$

Bg bezeichnet den Besetztheitsgrad (siehe 3.1.1S13).

Bew.: klar. $\diamondsuit$

Lemma 3.1.4S14 gibt uns einen Hinweis darauf, welche Erwartungen wir hinsichtlich des Sparse-Speedup im allgemeinen Fall hegen dürfen. Deshalb definieren wir eine Sparse-Efficiency, die allerdings nicht kleiner als eins sein muß.

Definition 3.1.5 (Sparse-Efficiency SE) :

Die Sparse-Efficiency ist definiert als das Produkt SE = SS * Bg.
Hierbei bezeichnen SS das Sparse-Speedup (siehe 3.1.2S13) und
Bg den Besetztheitsgrad (siehe 3.1.1S13).

Sparse-Efficiencies nahe oder sogar größer als eins besagen, daß der Sparse-Algorithmus relativ zum Dense-Algorithmus effizient arbeitet.

Bei der Entwicklung des Sparse-Block-Huang-Algorithmus beschränken wir uns auf den Fall von Block-Tridiagonal-Matrizen.

Definition 3.1.6 (Homogene m-Block-Tridiagonal-Matrix) :

Eine $n \times n$-Matrix A ist genau dann eine homogene m-Block-Tridiagonal-Matrix, wenn

1. $m \in \mathbb{N}$ mit $\mathrm{Rest}(\frac{n}{m}) = 0$ bzw. $l := \frac{n}{m} \in \mathbb{N}$
2. $A = \begin{pmatrix} B_1 & C_1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ A_2 & B_2 & C_2 & 0 & \cdots & 0 & 0 & 0 \\ 0 & A_3 & B_3 & C_3 & \cdots & 0 & 0 & 0 \\ \cdots \\ 0 & 0 & 0 & 0 & \cdots & A_{l-1} & B_{l-1} & C_{l-1} \\ 0 & 0 & 0 & 0 & \cdots & 0 & A_l & B_l \end{pmatrix}$ mit

   $0, A_\nu, B_\nu, C_\nu \in \mathbb{R}^{m \times m}$.

Weil wir nur homogene Block-Tridiagonal-Matrizen betrachten, lassen wir den Zusatz homogen weg.
Für $m = 1$ ergibt sich eine gewöhnliche Tridiagonalmatrix.
Weil $m = n$ zugelassen ist, kann man jede Matrix als homogene n-Block-Tridiagonal-Matrix auffassen.

3.2 Sparse-Huang-Algorithmus

Man kann den **Huang-Algorithmus** speziell für Bandmatrizen hinsichtlich des Speicher- und Rechenbedarfs optimieren; hierzu sei auf Theorem 11.7 von [Abaffy/Spedicato] auf Seite 180 hingewiesen.

Aber auch im allgemeinen Fall einer unstrukturierten, dünnbesetzten Matrix A kann man zumindest Rechenzeit dadurch einsparen, daß man die Multiplikation von H_i mit a_i^T geeignet durchführt:
Man berücksichtigt nur die Nichtnullelemente von A.

3.3 Sparse-Block-Huang-Algorithmus

Aufgrund der Identität $\tilde{A}p = \tilde{A}H\tilde{A}^T = \tilde{A}H^T H\tilde{A}^T = p^T p$ kann man auch bei Verwendung des Block-Huang-Algorithmus Nullelemente berücksichtigen, wenn man γ alternativ berechnet.

Definition 3.3.1 (Sparse-Block-Huang-Algorithmus) :

Wir setzen voraus, daß m ein Teiler von n ist.
Sei ein lineares Gleichungssystem mit m-Block-Tridiagonal-Matrix (siehe 3.1.6S14) gegeben.
Wenn wir zur Lösung den Block-Huang-Algorithmus mit Blockgröße m verwenden, $\gamma_k = \tilde{A}_k p_k$ alternativ berechnen und nur die Nichtnullblöcke in $\tilde{A}_k$ berücksichtigen, dann sprechen wir von der Anwendung des Sparse-Block-Huang-Algorithmus.

3.4 Programmierung

Unter Berücksichtigung der mit dem Block-Huang-Algorithmus gesammelten Erfahrungen (siehe Abschnitt "Tests") wird der Sparse-Block-Huang-Algorithmus auf der Cray T3E in Fortran programmiert. Um einen fairen Vergleich zu ermöglichen, wird der Dense-Block-Huang-Algorithmus ebenfalls noch einmal in alternativer Form in Fortran programmiert. Die Programme nutzen die Lapack-Library (siehe [Lapack]) sehr intensiv. Auf den folgenden Seiten werden die Quelltexte anstelle der Algorithmen wiedergegeben, damit eine möglichst exakte Darstellung der Testbedingungen vorliegt.

Der reprogrammierte Block-Huang-Algorithmus für vollbesetzte Matrizen:

```
PN ist die Dimension n
PM ist die Blockgroesse m
B  ist die rechte Seite
A  ist die Koeffizientenmatrix
X  ist die Loesung
subroutine HUANG (PN,PM,B,A,PMM,H,P,X,ALPHA,BETA,GAMMA,DELTA)
implicit none
integer ii, jj, I, J, L, LL, M, N, JS, K
character chl, cht, chn
integer MM, NP1, MQ
integer ione
real*8 one, nil, mone
integer ppp, hhh, del, aa, aaa
integer PN, PM, PMM, N1, M1, MQ1
real*8 B,A,H,P,X,ALPHA,BETA,GAMMA,DELTA
dimension B(PN), H(PN*PN), P(PN*PM), X(PN), A(PN*PN)
dimension ALPHA(PM), BETA(PM), GAMMA(PM*PM), DELTA(PM*(PN+1))
MM = 0
N = PN
M = PM
MQ = M * M
N1 = N - 1
M1 = M - 1
MQ1 = MQ - 1
L   = N / M
LL  = L - 1
NP1 = N + 1
chl = 'L'
cht = 'T'
chn = 'N'
ione = 1
one = 1.0E0
nil = 0.0E0
mone = -1.0E0
```

```
do I=0,N1
   X(I+1) = nil
enddo
do I=0,LL
   do J=0,I
      hhh = 1 + (I*L+J) * MQ
      do ii=0,MQ1
         H(hhh+ii) = nil
      enddo
   enddo
   hhh = 1 + (I*L+I) * MQ
   do ii=0,M1
      H(hhh+M*ii+ii) = one
   enddo
enddo
do K=0,LL
   aa = 1 + K * L * MQ
   JS = 1 + K * M
   do J=0,M1
      BETA(J+1) = B(JS+J)
   enddo
   if(K > 0) then
      do J=0,LL
         hhh = 1  + J * L * MQ
         aaa = aa;
         ppp = 1  +     J * MQ;
         call SGEMM(chn,cht,PM,PM,PM,one,H(hhh),PM,A(aaa),PM,nil,P(ppp),PM)
         do I=1,LL
            if(I<=J) then
               hhh = 1  + (J*L+I) * MQ
               aaa = aa +       I * MQ
               call SGEMM(chn,cht,PM,PM,PM,one,H(hhh),PM,A(aaa),PM,one,P(ppp),PM)
            else
               hhh = 1  + (I*L+J) * MQ
               aaa = aa +       I * MQ
               call SGEMM(cht,cht,PM,PM,PM,one,H(hhh),PM,A(aaa),PM,one,P(ppp),PM)
            endif
         enddo
      enddo
   else
```

```
   do J=0,LL
      ppp = 1  + J * MQ
      aaa = aa + J * MQ
      do ii=0,M1
         do jj=0,M1
            P(ppp+ii*M+jj) = A(aaa+jj*M+ii)
         enddo
      enddo
   enddo
endif
aaa = aa
hhh = 1
call SGEMV(chn,PM,PM,one,A(aaa),PM,X(hhh),ione,nil,ALPHA,ione)
do I=1,LL
   aaa = aa + I * MQ
   hhh =  1 + I * M
   call SGEMV(chn,PM,PM,one,A(aaa),PM,X(hhh),ione,one,ALPHA,ione)
enddo
aaa = aa
ppp = 1
call SGEMM(chn,chn,PM,PM,PM,one,A(aaa),PM,P(ppp),PM,nil,GAMMA,PM)
do I=1,LL
   aaa = aa + I * MQ
   ppp =  1 + I * MQ
   call SGEMM(chn,chn,PM,PM,PM,one,A(aaa),PM,P(ppp),PM,one,GAMMA,PM)
enddo
call SAXPY(PM,mone,ALPHA,ione,BETA,ione)
do J=0,LL
   ppp = 1 + J * MQ
   del = 1 + J * MQ
   do ii=0,M1
      do jj=0,M1
         DELTA(del+ii*M+jj) = P(ppp+jj*M+ii)
      enddo
   enddo
enddo
call SCOPY(PM,BETA,ione,DELTA(1+M*N),ione)
call SPOSV(chl,PM,NP1,GAMMA,PM,DELTA,PM,MM)
if(MM == 0) then
   MM = 0
else
   MM = -2
   goto 600
endif
```

```
    call SCOPY(PM,DELTA(1+M*N),ione,BETA,ione)
    do J=0,LL
       ppp = 1 + J * MQ
       hhh = 1 + J * M
       call SGEMV(chn,PM,PM,one,P(ppp),PM,BETA,ione,one,X(hhh),ione)
    enddo
    if(K < LL) then
       do I=0,LL
          do J=0,I
             ppp = 1 +       I * MQ
             del = 1 +       J * MQ
             hhh = 1 + (I*L+J) * MQ
             call SGEMM(chn,chn,PM,PM,PM,mone,P(ppp),PM,DELTA(del),PM,one,H(hhh),PM)
          enddo
       enddo
    endif
 enddo
 600 PMM = MM
 call SCOPY(PN,X,ione,B,ione)
 return
 end
```

Der Sparse-Block-Huang-Algorithmus:

```
PN   ist die Dimension n
PM   ist die Blockgroesse m
B    ist die rechte Seite
alow ist die Sub  -Block-Diagonale der Block-Tridiagonal-Matrix
adia ist die       Block-Diagonale der Block-Tridiagonal-Matrix
aupp ist die Super-Block-Diagonale der Block-Tridiagonal-Matrix
X    ist die Loesung
subroutine HUANG (PN,PM,B,alow,adia,aupp,PMM,H,P,X,ALPHA,BETA,GAMMA,DELTA)
implicit none
integer ii, jj, I, J, L, LL, M, N, JS, K
character chl, cht, chn
integer MM, NP1, MQ
integer ione
real*8 one, nil, mone
integer hhh, ppp, del, alo, adi, aup
integer M1, MQ1
integer PN,PM,PMM
real*8 B,alow,adia,aupp,H,P,X,ALPHA,BETA,GAMMA,DELTA
dimension B(PN), H(PN*PN), P(PN*PM), X(PN)
dimension alow(PN*PM), adia(PN*PM), aupp(PN*PM)
dimension ALPHA(PM), BETA(PM), GAMMA(PM*PM), DELTA(PM*(PN+1))
MM = 0
N = PN
M = PM
MQ = M * M
M1 = M - 1
MQ1 = MQ - 1
chl = 'L'
cht = 'T'
chn = 'N'
ione = 1
one = 1.0E0
nil = 0.0E0
mone = -1.0E0
L   = N / M
LL  = L - 1
NP1 = N + 1
```

```
do I=0,N
   X(I) = nil
enddo
do I=0,LL
   do J=0,I
      hhh = 1 + (I*L+J) * MQ
      do ii=0,MQ1
         H(hhh+ii) = nil
      enddo
   enddo
   hhh = 1 + (I*L+I) * MQ
   do ii=0,M1
      H(hhh+M*ii+ii) = one
   enddo
enddo
do K=0,LL
   alo = 1 + K * MQ
   adi = 1 + K * MQ
   aup = 1 + K * MQ
   JS  = 1 + K * M
   do J=0,M1
      BETA(J+1) = B(JS+J)
   enddo
   if( K > 0 ) then
      do J=0,LL
         ppp = 1 + J * MQ
         if( J >= K ) then
            hhh = 1 + (J*L+K) * MQ
            call SGEMM(chn,cht,PM,PM,PM,one,H(hhh),PM,adia(adi),PM,nil,P(ppp),PM)
         else
            hhh = 1 + (K*L+J) * MQ
            call SGEMM(cht,cht,PM,PM,PM,one,H(hhh),PM,adia(adi),PM,nil,P(ppp),PM)
         endif
         if( J >= K-1 ) then
            hhh = 1 + (J*L+K-1) * MQ
            call SGEMM(chn,cht,PM,PM,PM,one,H(hhh),PM,alow(alo),PM,one,P(ppp),PM)
         else
            hhh = 1 + ((K-1)*L+J) * MQ
            call SGEMM(cht,cht,PM,PM,PM,one,H(hhh),PM,alow(alo),PM,one,P(ppp),PM)
         endif
```

```
      if( K < LL ) then
         if( J >= K+1 ) then
            hhh = 1 + (J*L+K+1) * MQ
            call SGEMM(chn,cht,PM,PM,PM,one,H(hhh),PM,aupp(aup),PM,one,P(ppp),PM)
         else
            hhh = 1 + ((K+1)*L+J) * MQ
            call SGEMM(cht,cht,PM,PM,PM,one,H(hhh),PM,aupp(aup),PM,one,P(ppp),PM)
         endif
      endif
   enddo
else
   do J=0,LL
      ppp = 1 + J * MQ
      if( J == 0 ) then
         do ii=0,M1
            do jj=0,M1
               P(ppp+ii*M+jj) = adia(adi+jj*M+ii)
            enddo
         enddo
      endif
      if( J == 1 ) then
         do ii=0,M1
            do jj=0,M1
               P(ppp+ii*M+jj) = aupp(aup+jj*M+ii)
            enddo
         enddo
      endif
      if( J > 1 ) then
         do ii=0,MQ1
            P(ppp+ii) = 0.0
         enddo
      endif
   enddo
endif
call SGEMV(chn,PM,PM,one,adia(adi),PM,X(JS),ione,nil,ALPHA,ione)
if( K > 0 ) then
   call SGEMV(chn,PM,PM,one,alow(alo),PM,X(JS-M),ione,one,ALPHA,ione)
endif
if( K < LL ) then
   call SGEMV(chn,PM,PM,one,aupp(aup),PM,X(JS+M),ione,one,ALPHA,ione)
endif
ppp = 1 + K * MQ
call SGEMM(chn,chn,PM,PM,PM,one,adia(adi),PM,P(ppp),PM,nil,GAMMA,PM)
```

```
   if( K > 0 ) then
      ppp = 1 + (K-1) * MQ
      call SGEMM(chn,chn,PM,PM,PM,one,alow(alo),PM,P(ppp),PM,one,GAMMA,PM)
   endif
   if( K < LL ) then
      ppp = 1 + (K+1) * MQ
      call SGEMM(chn,chn,PM,PM,PM,one,aupp(aup),PM,P(ppp),PM,one,GAMMA,PM)
   endif
   call SAXPY(PM,mone,ALPHA,ione,BETA,ione)
   do J=0,LL
      ppp = 1 + J * MQ
      del = 1 + J * MQ;
      do ii=0,M1
         do jj=0,M1
            DELTA(del+ii*M+jj) = P(ppp+jj*M+ii)
         enddo
      enddo
   enddo
   call SCOPY(PM,BETA,ione,DELTA(1+M*N),ione)
   call SPOSV(chl,PM,NP1,GAMMA,PM,DELTA,PM,MM)
   if( MM == 0 ) then
      MM = 0
   else
      MM = -2
      goto 600
   endif
   call SCOPY(PM,DELTA(1+M*N),ione,BETA,ione)
   do J=0,LL
      ppp = 1 + J * MQ
      hhh = 1 + J * M
      call SGEMV(chn,PM,PM,one,P(ppp),PM,BETA,ione,one,X(hhh),ione)
   enddo
   if( K < LL ) then
      do I=0,LL
         do J=0,I
            ppp = 1 +       I * MQ
            del = 1 +       J * MQ
            hhh = 1 + (I*L+J) * MQ
            call SGEMM(chn,chn,PM,PM,PM,mone,P(ppp),PM,DELTA(del),PM,one,H(hhh),PM)
         enddo
      enddo
   endif
enddo
600 PMM = MM
call SCOPY(PN,X,ione,B,ione)
return
end
```

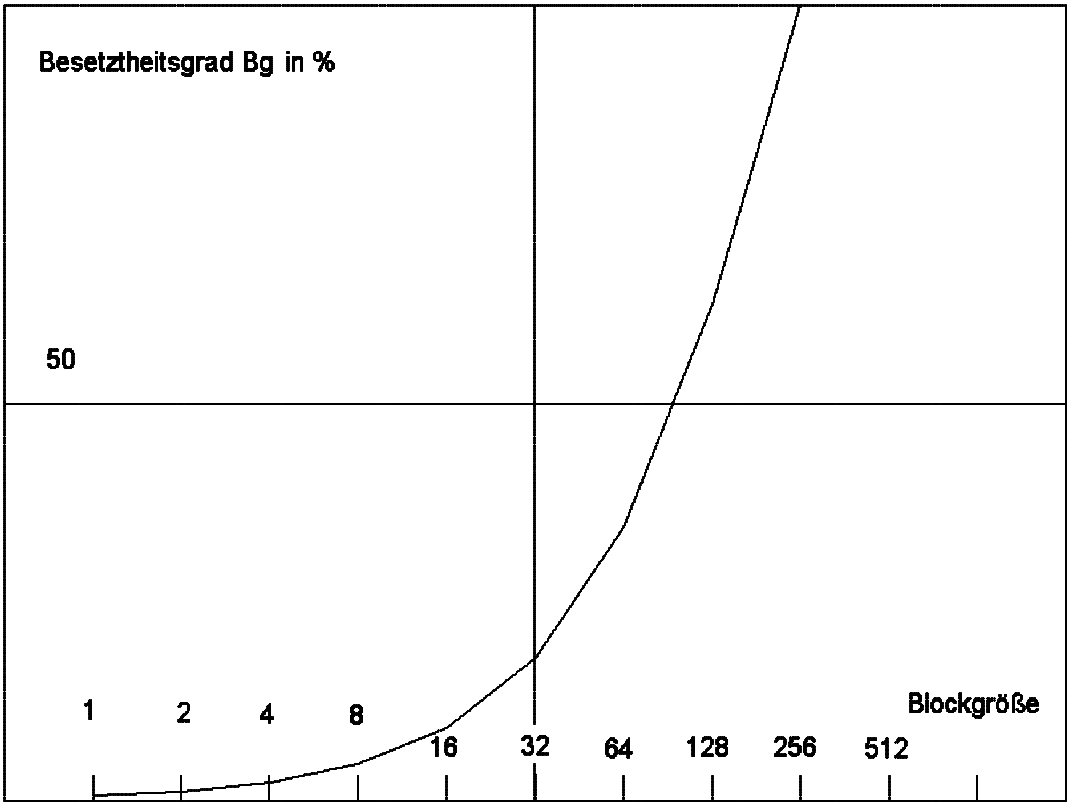
3.5 Tests

Um den Sparse-Block-Huang-Algorithmus intensiv zu prüfen, werden spezielle lineare Gleichungssysteme der Dimension $n = 512$ erzeugt und anschließend gelöst:
Die zufällig erzeugten $n \times n$-Matrizen sind m-block-tridiagonal (siehe 3.1.6S14) mit ganzzahlig-vollbesetzten Nichtnullblöcken.
Der Besetztheitsgrad (siehe 3.1.1S13) von derartigen Matrizen errechnet sich gemäß:

$$Bg = \frac{m}{n} \left(3 - 2\frac{m}{n} \right)$$

und wird in der folgenden Abbildung für $n = 512$ dargestellt.

Abbildung 3.5.1 (Besetztheitsgrad, $n = 512$) :



Um den Sparse-Block-Huang-Algorithmus mit dem Dense-Block-Huang-Algorithmus vergleichen zu können, geben wir sowohl das Sparse-Speedup (siehe 3.1.2S13), als auch die Sparse-Efficieny (siehe 3.1.5S14) an. Die Tests werden auf dem Rechner Cray-T3E (ein proccessing element PE) durchgeführt. Es wird jeweils dreimal gemessen und der Mittelwert gebildet.

Abbildung 3.5.2 (T3E 1 PE Sparse-Speedup) :

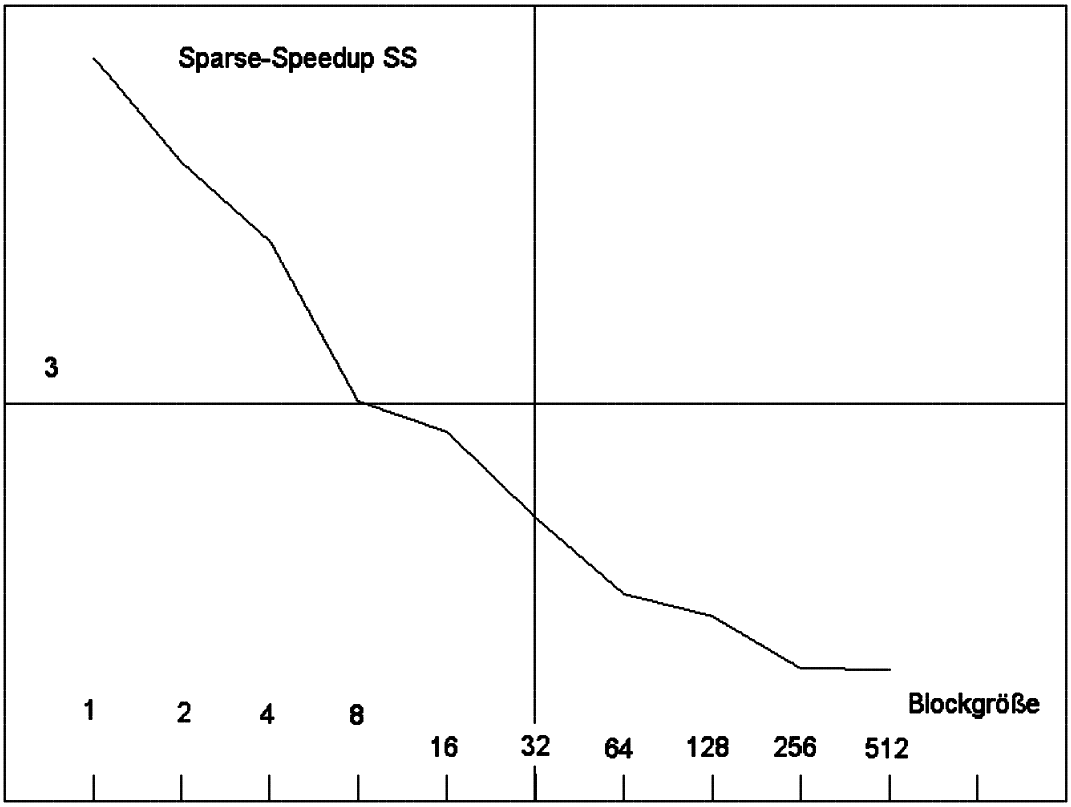
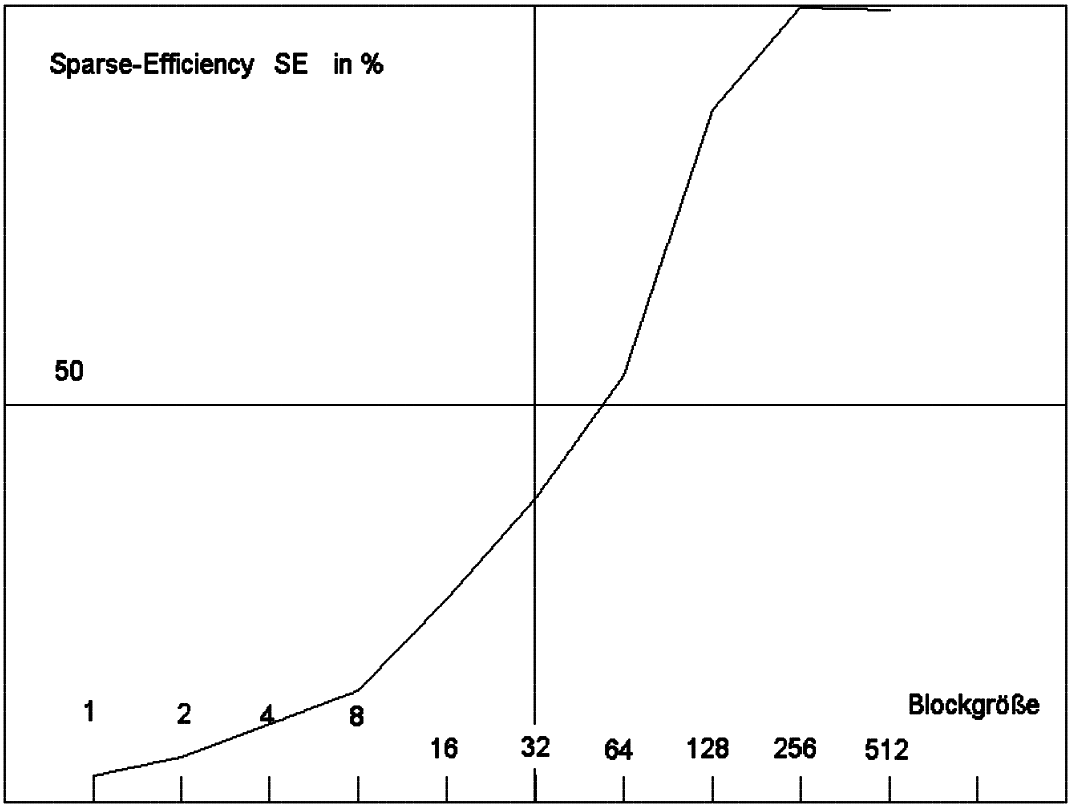


Abbildung 3.5.3 (T3E 1 PE Sparse-Efficiency) :



3.6 Auswertung

Zunächst sieht man, daß für hinreichend dünn besetzte Matrizen eine deutliche Geschwindigkeitssteigerung erzielt werden kann, wenn man den Sparse-Block-Huang-Algorithmus verwendet. Erwartungsgemäß nimmt das Sparse-Speedup ab, wenn sich die Blockgröße der Dimension nähert, also die Matrix nahezu vollbesetzt ist.

Die Darstellung der Sparse-Efficiency zeigt uns:

1. Mit einer hinreichend groß gewählten Blockgröße (z.B. ab $m = 32$ für $n = 512$) läßt sich eine annehmbare Sparse-Efficiency (> 38 %) erzielen.

2. Der Sparse-Block-Huang-Algorithmus ist eigentlich kein "echter" Sparse-Algorithmus; sonst würde man gerade für extrem dünn besetzte Matrizen (z.B. Tridiagonal-Matrizen) eine gute Sparse-Efficiency erzielen.

Der Sparse-Block-Huang-Algorithmus besitzt eine entscheidende Schwäche:

- Er benötigt Speicherplatz für die volle, symmetrische Projektionsmatrix.

Dadurch läßt sich die Abbildung 3.5.3S25 vollständig erklären.

Der Sparse-Block-Huang-Algorithmus kann eingesetzt werden, wenn

1. genügend Speicherplatz vorhanden ist; üblicherweise bedeutet das, daß die Dimension im mittleren Bereich liegt.

2. die Blockgröße möglichst groß gewählt werden kann oder muß (z.B. für mäßig besetzte Matrizen).

4 Paralleler Block-Huang-Algorithmus

4.1 Eigenschaften

Der Block-Huang-Algorithmus 2.2.1S4 kann als serieller Algorithmus betrachtet werden, indem man die einzelnen Teilschritte (insbesondere die Matrix-Matrix-Multiplikationen und die Lösung der symmetrisch, positiv-definiten Hilfssysteme) seriell bewältigt.
Er kann aber auch als paralleler Algorithmus betrachtet werden, wenn man parallele Teilalgorithmen verwendet.
Die Hauptschleife ($k = 1, \ldots, \frac{n}{m}$) läßt sich jedoch nicht parallelisieren, ohne zu einem völlig neuen Gesamtalgorithmus zu gelangen. Dieser Ansatz wurde von [Dekker] verfolgt.

Es ist nun interessant zu sehen, welches Speedup theoretisch erzielt wird, wenn man die Teilschritte parallelisiert. Hierbei benutzen wir eine Struktureigenschaft des Speedups:

Lemma 4.1.1 (Zusammengesetztes Speedup) :

Vor.: Seien die auf p Prozessoren lauffähigen Algorithmen A_p gegeben.
Sie setzen sich aus den Teilalgorithmen $A_{p,i},\ i = 1, \ldots, l$
zusammen, die jeweils $t_{p,i}$ Zeitschritte auf p Prozessoren
benötigen.

Beh.: Das Gesamtspeedup S ist ein gewichtetes, harmonisches Mittel
der Einzelspeedups S_i.

Bew.: $t_p = \sum_{i=1}^{l} t_{p,i},\ f_i := \frac{t_{1,i}}{t_1}$ mit $\sum_{i=1}^{l} f_i = 1.$

$$S = \frac{t_1}{t_p} = \frac{1}{\sum_{i=1}^{l} \frac{t_{p,i}}{t_1}} = \frac{1}{\sum_{i=1}^{l} \frac{t_{p,i}}{t_{1,i}} * \frac{t_{1,i}}{t_1}} = \frac{1}{\sum_{i=1}^{l} f_i * \frac{1}{S_i}}. \diamond$$

Die Komplexitätsanalyse des parallelen Block-Huang-Algorithmus gliedert sich in

1. die serielle Analyse, d.h. in das Abzählen der Rechenoperationen und die Bestimmung der Gewichte f_i aus Lemma 4.1.1S27.

2. die Bestimmung der Einzelspeedups für die Teilalgorithmen unter Berücksichtigung der Kommunikation.

Wir verwenden ein vereinfachtes Modell der Komplexitätsanalyse:

- Addition, Subtraktion, Multiplikation und Division benötigen einen Zeitschritt.

- Alle anderen Operationen (z.B. Kommunikation, Speicherzugriffe) werden vernachlässigt.

Auf diese Art und Weise erhält man eine obere Schranke für das reale Speedup, weil:

- die Kommunikation als serieller Anteil der parallelen Algorithmen betrachtet werden muß.

- nach Amdahls Gesetz ein geringeres Speedup bei wachsendem seriellen Anteil erwartet werden muß.

Ohne Herleitung führen wir die einzelnen Teilschritte zusammen mit der benötigten Anzahl an Operationen tabellarisch auf (die Anzahl q der rechten Seiten ist stets 1, jeder Teilschritt wird $\frac{n}{m}$ mal berechnet) :

Tabelle 4.1.2 (Zeitschritte des seriellen Algorithmus) :

	Teilschritt	Zeitschritte (näherungsweise !)	Gewicht für $n = 1536$, $m = 64$ ca.
1	$p = H_{-}\tilde{A}^T$	$2n^2m$	0,6309 $\approx$ 63 %
2	$\gamma = p^Tp$	$2nm^2$	0,0263 $\approx$ 3 %
3	$\alpha = \tilde{A}x - \tilde{b}$	$2nm$	0,0004 $\approx$ 0 %
4	$\gamma \longrightarrow LL^T$	$\frac{1}{3}m^3$	0,0002 $\approx$ 0 %
5	$L(L^TW) = (p^T\|\alpha)$	$2nm^2$	0,0263 $\approx$ 3 %
6	$x = x - pW_{n+1}$	$2nm$	0,0004 $\approx$ 0 %
7	$H = H_{-} - pW_{1\ldots n}$	n^2m	0,3155 $\approx$ 31 %
	Gesamt:	$\frac{n}{m} \cdot (3n^2m + 4nm^2 + 4nm + \frac{1}{3}m^3)$	

Wir wenden uns nun dem parallelen Block-Huang-Algorithmus zu.

Tabelle 4.1.3 (Zeitschritte des parallelen Algorithmus) :

Teilschritt	Zeitschritte (näherungsweise !)	Speedup
$p = H_{-}\tilde{A}^T$	$\frac{1}{p}(2n^2m)$	p, d.h. maximal
$H = H_{-} - pW_{1\ldots n}$	$\frac{1{,}0526}{p}(n^2m)$	$0,95p$
Rest	$\frac{1{,}1111}{p}$(serielle Zeit)	$0,9p$

Vereinfachend geht man davon aus, daß die Matrix-Matrix-Multiplikation optimal und die übrigen Teilschritte schlechter parallelisierbar sind; dies deckt sich mit den Erfahrungen aus der Praxis.

Satz 4.1.4 (Speedup des Block-Huang-Algorithmus) :

Vor.: Sei der parallele Block-Huang-Algorithmus gegeben.

Beh.: Speedup $S \approx \dfrac{1}{\frac{0{,}63}{S_1} + \frac{0{,}31}{S_7} + \frac{0{,}06}{S_{\text{Rest}}}}$
$\approx 62{,}6$ für $p = m = 64,\ n = 1536.$

Bew.: offensichtlich nach Lemma 4.1.1S27.$_{\text{qed}}$

Bemerkung 4.1.5 (Berücksichtigung der Symmetrie) :

Wenn man die Symmetrie der Projektionsmatrix im Update-Schritt $H = H_{-} - pW_{1\ldots n}$ nicht berücksichtigen kann, dann entfallen (in sehr grober Näherung) ca. je die Hälfte der Rechenzeit auf den Update-Schritt und auf das Matrix-Matrix-Produkt $p = H_{-}\tilde{A}^T$; hierbei setzen wir $n \gg m$ voraus.
Beispielsweise gibt es keine "symmetrische" Update-Prozedur in der BLAS-Bibliothek !
Auch in diesem Fall ergibt sich ein ähnlich optimistischer Wert für das Speedup.

4.2 Programmierung

Die Programmierung des parallelen Block-Huang-Algorithmus läßt sich mit Hilfe von PBLAS/Scalapack bewältigen. Dies trifft auf alle Programme zu, die die Bibliotheken BLAS/Lapack (siehe [Lapack]) intensiv nutzen.
PBLAS baut auf BLAS und BLACS (=parallele Kommunikationsroutinen) auf. Eigentlich sollte die parallele Programmierung ganz einfach sein. Theoretisch muß man nur die Aufrufe der BLAS-Routinen durch die entsprechenden Aufrufe der PBLAS-Routinen ersetzen, einige Parameter ändern oder ergänzen und die Matrix-Descriptoren definieren, die festlegen, wie eine Matrix auf die Prozessoren verteilt wird.

Praktisch ist die parallele Programmierung aufgrund der Übergabe von weiteren (die parallele Struktur betreffenden) Parametern deutlich aufwendiger als die serielle; man vergleiche einmal das Programmlisting des Dense-Block-Huang-Algorithmus ab Seite 16 mit dem des parallelen Block-Huang-Algorithmus. Dadurch ist man gezwungen eine umfangreichere Fehlersuche durchzuführen. Diese wird erheblich erschwert, weil kein funktionstüchtiger Debugger zur Verfügung steht !
Beispielsweise bewirkt die Übergabe einer zu großen Blockgröße an die **NUMROC**-Funktion nicht etwa eine Fehlermeldung zur Laufzeit des Programms, stattdessen kann das gleiche Programm

- abstürzen,
- eine falsche Lösung liefern,
- mit einer Fehlermeldung, die von einer <u>anderen</u> Prozedur geliefert wird, abbrechen oder
- sogar die richtige Lösung liefern (insbesondere dann, wenn man nur auf einem Prozessor arbeitet) !

Beim "Hand-Debugging" arbeitet man vorzugsweise auf zwei bis vier Prozessoren und verwendet die althergebrachten Techniken aus den Zeiten, aus denen noch keine Debugger bekannt waren !
Beispielsweise ist es nützlich, sich die Schleifenindizes und Grenzwerte per **print**-Befehl anzeigen zu lassen. Hierbei muß man darauf achten, daß die Prozessornummer mit ausgegeben wird, damit man die einzelnen Bildschirmausgaben auseinander halten kann.
Schwierigkeiten, die auf die Kommunikation zurückzuführen sind, kann man allerdings mit diesen Methoden nicht finden, weil sie "zufallsabhängig" sein können !

Bei der parallelen Programmierung tritt der Aspekt der Kommunikation in den Vordergrund. Wenn man PBLAS/Scalapack verwendet, dann sollte man sich um die Kommunikation eigentlich nicht kümmern müssen, weil sie implizit mit Hilfe der BLACS stattfindet. Wie bereits angedeutet, muß man sich leider doch damit befassen. Man muß die Kommunikation prinzipiell als seriellen Overhead des Programms betrachten, der nach Amdahls Gesetz zu einem geringeren Speedup und einer geringeren Effizienz führt. Die PBLAS-Routinen sind theoretisch hinsichtlich dieser Erfordernisse optimiert und sollten zumindest bei der ausschließlichen Verwendung von Matrix-Matrix-Operationen gute Effizienzen ermöglichen.

Auf den folgenden Seiten befindet sich das Programmlisting des parallelen Block-Huang-Algorithmus.

```
C
C       huangcra.f
C       Joachim Tews 25.07.99
C       with special from Carlo Cavazzoni
C       CRAY T3E only
C
        SUBROUTINE HUANG (N, M, B, A, MM)
C
        IMPLICIT NONE
        INTEGER I, J, K, L, N, M, MM, IOLD, II, JJ
        INTEGER N1, MYR, MYC, NP, NQ, MP, MQ, N1Q
        INTEGER NPA, NQA, MPA, MQA, N1QA, IP, JP
        INTEGER DAA, DH, DP, DX, DZ, DALPHA, DBETA
        INTEGER DGAMMA, DDELTA, NB, MB, ICTXT
        INTEGER NPSQRT, NPR, NPC, NBP, DIM
        INTEGER NUMROC, INDXL2G, INDXG2L, INDXG2P, MAX, MIN
        REAL*8 A, B, SALPHA, SBETA, SGAMMA, SDELTA, SEPS
        REAL*8 ABS
C
        PARAMETER (NPSQRT=1,NPR=NPSQRT,NPC=NPSQRT,NBP=64,DIM=9)
C
        REAL*8, allocatable :: AA(:,:), H(:,:), P(:,:)
        REAL*8, allocatable :: X(:), ALPHA(:)
C        REAL*8, allocatable :: X(:), Z(:,:), ALPHA(:)
        REAL*8, allocatable :: BETA(:), GAMMA(:,:), RDELTA(:,:)
C
        DIMENSION A(N,N), B(N)
        DIMENSION DAA(DIM), DH(DIM), DP(DIM), DX(DIM), DZ(DIM)
        DIMENSION DALPHA(DIM), DBETA(DIM), DGAMMA(DIM)
        DIMENSION DDELTA(DIM)
C
C
C
        MM = 0
        IF (N.LT.1) THEN
           MM = 99
           CALL JBSTOP(-1,MM)
        ENDIF
        IF (M.LT.0) THEN
           MM = 98
           CALL JBSTOP(-1,MM)
        ENDIF
```

```
      IF (N.EQ.1) THEN
         SALPHA = A(1,1)
         IF (ABS(SALPHA).LE.1.0E-10) THEN
            MM = -3
            CALL JBSTOP(-1,MM)
         ENDIF
         SALPHA = -B(1) / SALPHA
         B(1) = SALPHA
         RETURN
      ENDIF
      CALL BLACS_GRIDINIT(ICTXT,'R',NPR,NPC)
      CALL BLACS_GRIDINFO(ICTXT,NPR,NPC,MYR,MYC)
      CALL BLACS_QUEUE(20)
      IOLD = 0
      IF (M.EQ.0) THEN
         M = 1
         IOLD = 1
      ENDIF
      NB = NBP
      MB = NBP
C      NB = MIN(NBP,MAX(1,N/MAX(NPR,NPC)))
C      MB = MIN(NBP,MAX(1,M/MAX(NPR,NPC)))
      print *, 'NB = ', NB, ' ; MB = ', MB
      N1 = N + 1
      IF (MYR.LE.-1 .OR. MYR.GE.NPR .OR. MYC.LE.-1
     $    .OR. MYC.GE.NPC) THEN
         MM = 1
         CALL JBSTOP(ICTXT,MM)
      ENDIF
      NP  = NUMROC( N,NB,MYR,0,NPR)
      NQ  = NUMROC( N,NB,MYC,0,NPC)
      N1Q = NUMROC(N1,NB,MYC,0,NPC)
      MP  = NUMROC( M,MB,MYR,0,NPR)
      MQ  = NUMROC( M,MB,MYC,0,NPC)
      IF (NP.LT.0 .OR. NQ.LT.0 .OR. N1Q.LT.0 .OR.
     $    MP.LT.0 .OR. MQ.LT.0) THEN
         MM = 2
         CALL JBSTOP(ICTXT,MM)
      ENDIF
      NPA  = MAX(NP, 1)
      NQA  = MAX(NQ, 1)
      N1QA = MAX(N1Q,1)
      MPA  = MAX(MP, 1)
      MQA  = MAX(MQ, 1)
```

```
C       allocate memory for distributed AA, H, P, X, Z,
C                           ALPHA, BETA, GAMMA, RDELTA
        allocate(AA(NPA,MQA))
        allocate( H(NPA,NQA))
        allocate( P(NPA,MQA))
        allocate( X(NPA))
C        allocate( Z(NPA,MQA))
        allocate( ALPHA(MPA))
        allocate(  BETA(MPA))
        allocate( GAMMA(MPA,MQA))
        allocate(RDELTA(MPA,N1QA))
C       init array descriptors
        CALL DESCINIT(DAA,N,M,NB,MB,0,0,ICTXT,NPA,I)
        IF (I.NE.0) THEN
           MM = 3
           CALL JBSTOP(ICTXT,MM)
        ENDIF
        CALL DESCINIT(DH,N,N,NB,NB,0,0,ICTXT,NPA,I)
        IF (I.NE.0) THEN
           MM = 4
           CALL JBSTOP(ICTXT,MM)
        ENDIF
        CALL DESCINIT(DP,N,M,NB,MB,0,0,ICTXT,NPA,I)
        IF (I.NE.0) THEN
           MM = 5
           CALL JBSTOP(ICTXT,MM)
        ENDIF
        CALL DESCINIT(DX,N,1,NB,1,0,0,ICTXT,NPA,I)
        IF (I.NE.0) THEN
           MM = 6
           CALL JBSTOP(ICTXT,MM)
        ENDIF
C        CALL DESCINIT(DZ,N,M,NB,MB,0,0,ICTXT,NPA,I)
C        IF (I.NE.0) THEN
C           MM = 7
C           CALL JBSTOP(ICTXT,MM)
C        ENDIF
        CALL DESCINIT(DALPHA,M,1,MB,1,0,0,ICTXT,MPA,I)
        IF (I.NE.0) THEN
           MM = 8
           CALL JBSTOP(ICTXT,MM)
        ENDIF
        CALL DESCINIT(DBETA,M,1,MB,1,0,0,ICTXT,MPA,I)
        IF (I.NE.0) THEN
           MM = 9
           CALL JBSTOP(ICTXT,MM)
        ENDIF
```

```
      CALL DESCINIT(DGAMMA,M,M,MB,MB,0,0,ICTXT,MPA,I)
      IF (I.NE.0) THEN
         MM = 10
         CALL JBSTOP(ICTXT,MM)
      ENDIF
      CALL DESCINIT(DDELTA,M,N1,MB,NB,0,0,ICTXT,MPA,I)
      IF (I.NE.0) THEN
         MM = 11
         CALL JBSTOP(ICTXT,MM)
      ENDIF
C     set distributed H to identity
      DO I=1,NQ
         DO J=1,NP
            H(J,I) = 0.0E0
         ENDDO
      ENDDO
      DO I=1,N
         CALL PSELSET(H,I,I,DH,1.0E0)
      ENDDO
C     set distributed X to zero
      DO I=1,NP
         X(I) = 0.0E0
      ENDDO
      CALL BLACS_BARRIER(ICTXT,'A')
C     old method ???
C
      IF (IOLD.EQ.0) THEN
C
      L  = N / M
      IF (M*L.NE.N) THEN
         MM = 77
         CALL JBSTOP(ICTXT,MM)
      ENDIF
      DO K=1,L
         DO J=1,MP
C           copy repeated part of B to distributed BETA
            JJ = INDXL2G(J,MB,MYR,0,NPR)
            JJ = (K - 1) * M + JJ
            BETA(J) = B(JJ)
         ENDDO
```

```
          DO J=1,MQ
C            copy repeated part of A to distributed AA
             JJ = INDXL2G(J,MB,MYC,0,NPC)
             JJ = (K - 1) * M + JJ
             DO I=1,NP
                II = INDXL2G(I,NB,MYR,0,NPR)
                AA(I,J) = A(II,JJ)
             ENDDO
          ENDDO
          IF (K.EQ.1) THEN
             CALL SCOPY(NPA*MQA,AA,1,P,1)
          ELSE
             CALL PSSYMM('L','L',N,M,1.0E0,H,1,1,DH,AA,1,1,DAA,
     $          0.0E0,P,1,1,DP)
C             CALL PSSYMM('L','L',N,M,1.0E0,H,1,1,DH,AA,1,1,DAA,
C     $          0.0E0,Z,1,1,DZ)
C             CALL PSSYMM('L','L',N,M,1.0E0,H,1,1,DH,Z ,1,1,DZ ,
C     $          0.0E0,P,1,1,DP)
          ENDIF
          CALL PSGEMV('T',N,M,1.0E0,AA,1,1,DAA,X,1,1,DX,1,
     $       0.0E0,ALPHA,1,1,DALPHA,1)
          CALL PSGEMM('T','N',M,M,N,1.0E0,P,1,1,DP,P,1,1,DP,
     $       0.0E0,GAMMA,1,1,DGAMMA)
          CALL PSAXPY(M,-1.0E0,ALPHA,1,1,DALPHA,1,BETA,1,1,DBETA,1)
C         better : transpose procedure !!!
          CALL PSTRAN(M,N,1.0E0,P,1,1,DP,0.0E0,RDELTA,1,1,DDELTA)
          CALL PSCOPY(M,BETA,1,1,DBETA,1,RDELTA,1,N1,DDELTA,1)
          CALL PSPOSV('L',M,N1,GAMMA,1,1,DGAMMA,
     $       RDELTA,1,1,DDELTA,MM)
          IF (MM.NE.0) THEN
             MM = -1
             CALL JBSTOP(ICTXT,MM)
          ENDIF
          CALL PSCOPY(M,RDELTA,1,N1,DDELTA,1,BETA,1,1,DBETA,1)
          CALL PSGEMV('N',N,M,1.0E0,P,1,1,DP,BETA,1,1,DBETA,1,
     $       1.0E0,X,1,1,DX,1)
          IF (K.NE.N) THEN
             CALL PSGEMM('N','N',N,N,M,-1.0E0,P,1,1,DP,
     $          RDELTA,1,1,DDELTA,1.0E0,H,1,1,DH)
          ENDIF
       ENDDO
C      else part of IF (IOLD.EQ.0) THEN
       ELSE
C      else part of IF (IOLD.EQ.0) THEN
```

```
C       old version
        DO K=1,N
C          copy repeated column of A to distributed AA
           DO I=1,NP
              II = INDXL2G(I,NB,MYR,0,NPR)
              AA(I,1) = A(II,K)
           ENDDO
           IF (K.EQ.1) THEN
              CALL PSCOPY(N,AA,1,1,DAA,1,P,1,1,DP,1)
           ELSE
              CALL PSSYMV('L',N,1.0E0,H,1,1,DH,AA,1,1,DAA,1,
     $           0.0E0,P,1,1,DP,1)
           ENDIF
C           CALL PSSYMV('L',N,1.0E0,H,1,1,DH,AA,1,1,DAA,1,
C     $        0.0E0,Z,1,1,DZ,1)
C           CALL PSSYMV('L',N,1.0E0,H,1,1,DH, Z,1,1, DZ,1,
C     $        0.0E0,P,1,1,DP,1)
           CALL PSDOT(N,SALPHA,AA,1,1,DAA,1,X,1,1,DX,1)
           CALL BLACS_BARRIER(ICTXT,'A')
           CALL PSDOT(N,SGAMMA, P,1,1, DP,1,P,1,1,DP,1)
           CALL BLACS_BARRIER(ICTXT,'A')
C          same if 2 times ! no barrier neccessary
           IF ( (MYR.EQ.0) .AND. (MYC.EQ.0) ) THEN
              CALL SGEBS2D(ICTXT,'A','H',1,1,SGAMMA,1)
           ELSE
              CALL SGEBR2D(ICTXT,'A','H',1,1,SGAMMA,1,0,0)
           ENDIF
           IF ( (MYR.EQ.0) .AND. (MYC.EQ.0) ) THEN
              CALL SGEBS2D(ICTXT,'A','H',1,1,SALPHA,1)
           ELSE
              CALL SGEBR2D(ICTXT,'A','H',1,1,SALPHA,1,0,0)
           ENDIF
           IF (SGAMMA.LE.1.0E-10) THEN
              MM = -2
              CALL JBSTOP(ICTXT,MM)
           ENDIF
           SBETA  = SALPHA - B(K)
           SDELTA = -1.0E0 / SGAMMA
           SEPS   = SBETA * SDELTA
           CALL PSAXPY(N,SEPS,P,1,1,DP,1,X,1,1,DX,1)
           IF (K.NE.N) THEN
              CALL PSSYR('L',N,SDELTA,P,1,1,DP,1,H,1,1,DH)
           ENDIF
        ENDDO
C       end of IF (IOLD.EQ.0) THEN
        ENDIF
C       end of IF (IOLD.EQ.0) THEN
```

```
C
C       the following special is due to
C       Carlo Cavazzoni
C
C       clear and copy distributed X to local B
        DO I=1,N
           B(I) = 0.0E0
           IP = INDXG2P(I,NB,MYR,0,NPR)
           JP = INDXG2P(1, 1,MYC,0,NPC)
           IF( (IP.eq.MYR) .and. (JP.eq.MYC) ) THEN
              J = INDXG2L(I,NB,MYR,0,NPR)
              B(I) = X(J)
           ENDIF
        ENDDO
C       parallel sum of B
C       result: copy of distributed X
        CALL SGSUM2D(ICTXT,'A','H',N,1,B,N,-1,-1)
        CALL BLACS_GRIDEXIT(ICTXT)
        RETURN
        END
C       HUANG()
C
C
        SUBROUTINE JBSTOP(ICTXT,I)
        INTEGER I,ICTXT,NPR,NPC,MYR,MYC
C
        CALL BLACS_GRIDINFO(ICTXT,NPR,NPC,MYR,MYC)
        print *, 'MYR = ', MYR, ',  MYC = ', MYC, ',  error = ', I
        CALL BLACS_ABORT(ICTXT,I)
        RETURN
        END
C       JBSTOP()
C
C       end huangcra.f
C
```

4.3 Tests

Um den parallelen Block-Huang-Algorithmus intensiv zu prüfen, werden spezielle lineare Gleichungssysteme erzeugt und anschließend gelöst:
Hierzu werden vollbesetzte Matrizen und Lösungsvektoren mit ganzzahligen Pseudo-Zufallszahlen erzeugt und die dazugehörigen rechten Seiten berechnet.

Alle Probleme besitzen die Dimension $n = 1536$.
Es wird jeweils dreimal gemessen und der Mittelwert gebildet.
Es werden Tests mit unterschiedlichen Blockgrößen, Prozessoranzahlen und Prozessorgitttern durchgeführt. Wir stellen hier nur die Ergebnisse für die Blockgröße 64 dar.
Zum Vergleich betrachten wir ebenfalls den parallelen Gauß-Algorithmus aus der Scalapack-Bibliothek (Prozedur **PSGESV**).

Abbildung 4.3.1 (Rechenzeit PSGESV-Prozedur) :

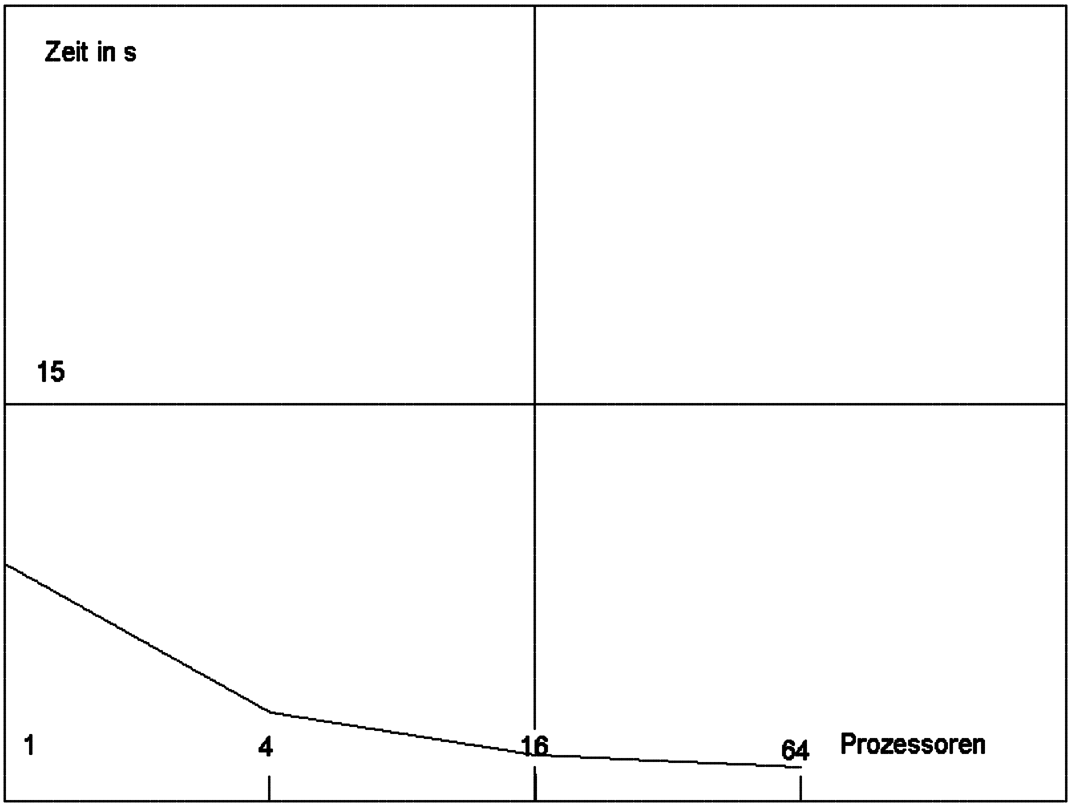


Abbildung 4.3.2 (Rechenzeit Block-Huang-Algorithmus) :

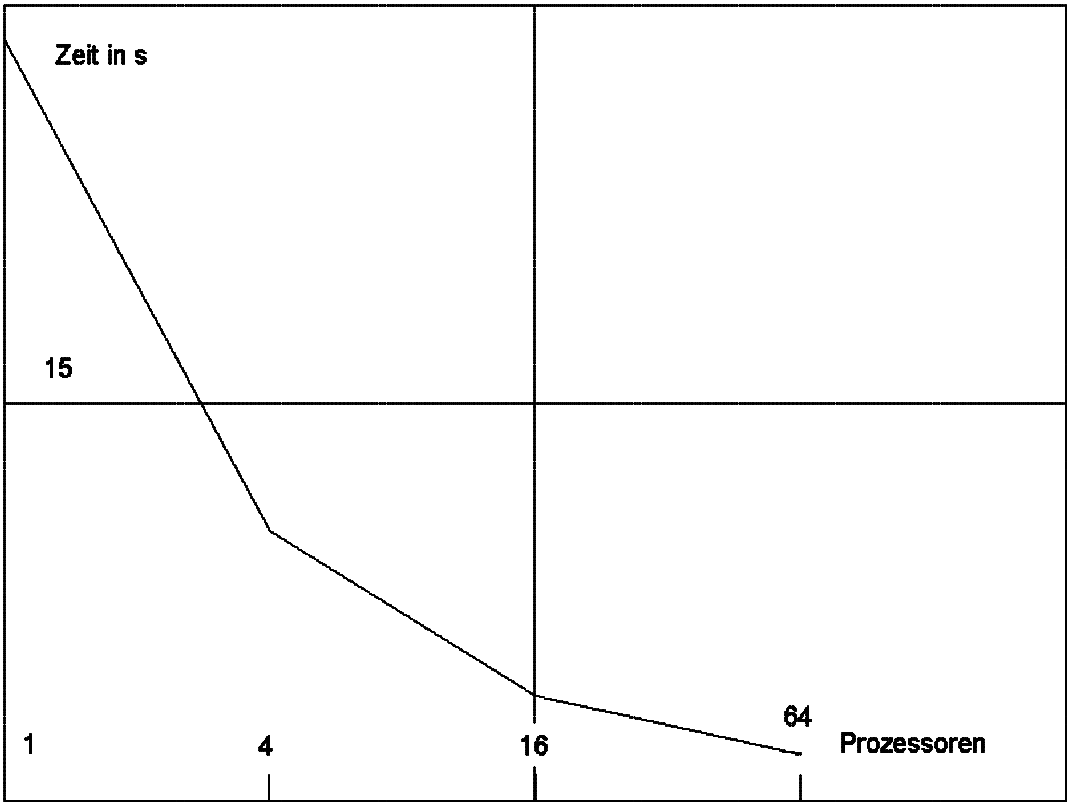


Abbildung 4.3.3 (Faktor Huang zu Gauß) :

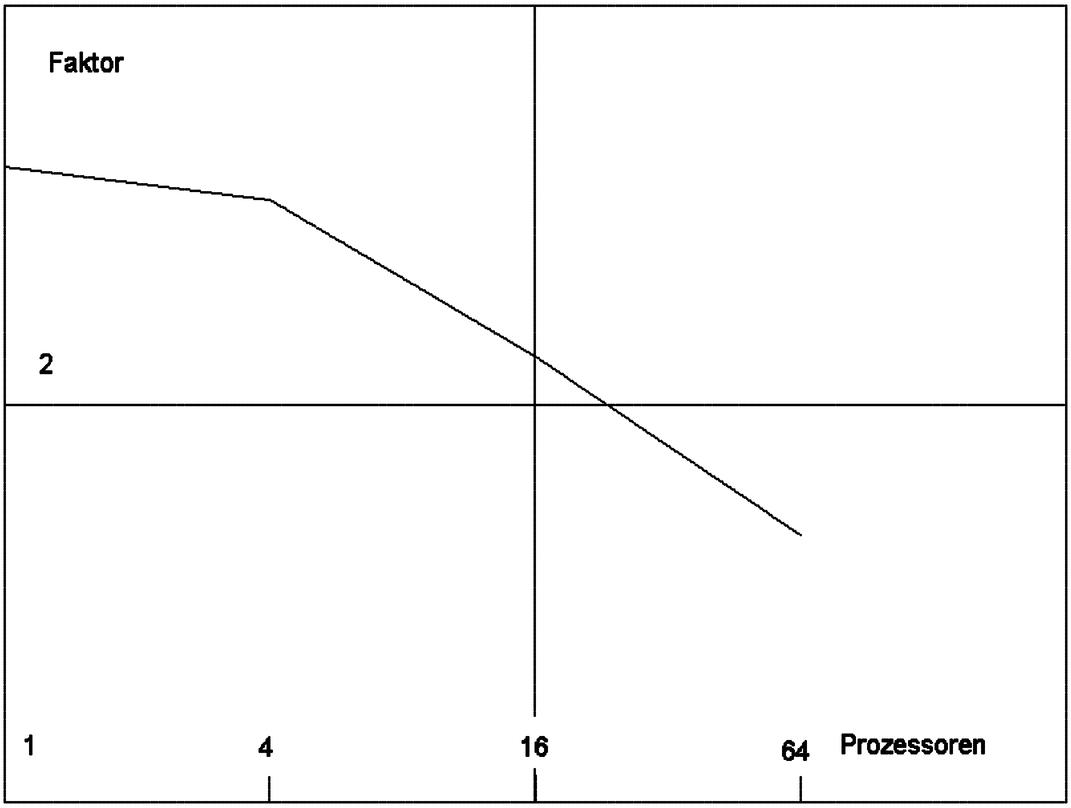


Abbildung 4.3.4 (Paralleles Speedup Block-Huang-Algorithmus) :

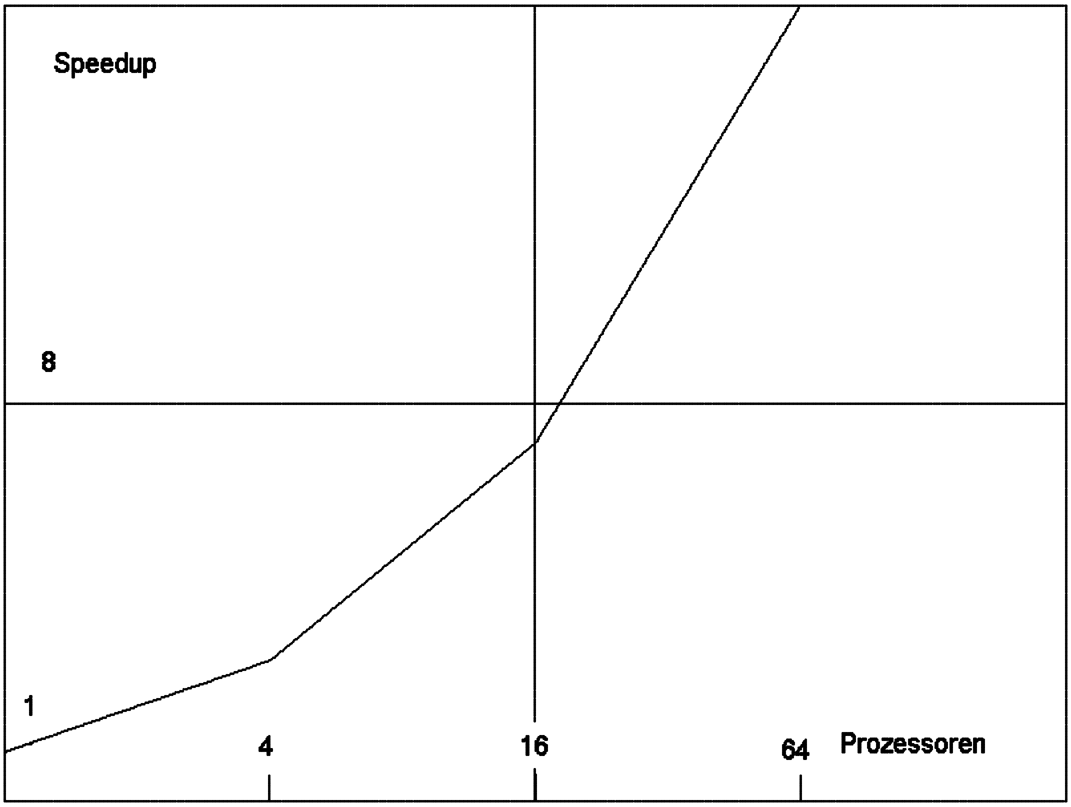
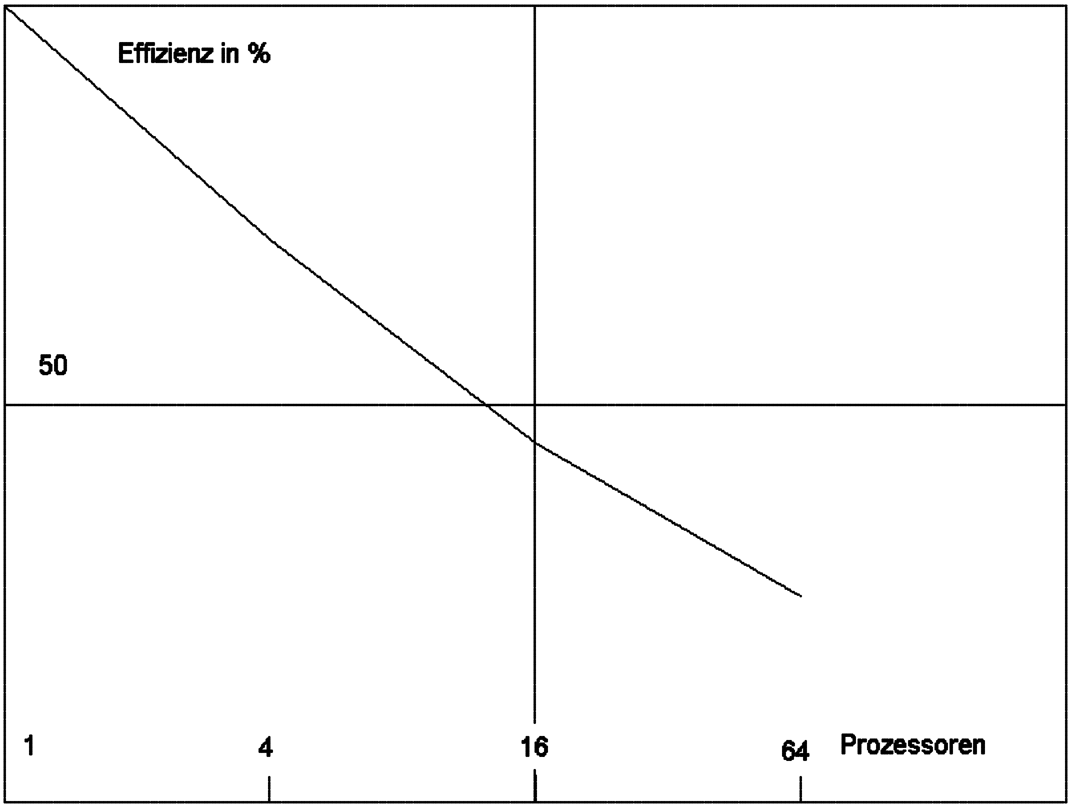


Abbildung 4.3.5 (Parallele Effizienz Block-Huang-Algorithmus) :



4.4 Auswertung

Wie wir bereits gesehen haben, ist der modifizierte Block-Huang-Algorithmus, aufgrund der optimierten BLAS-Routinen, deutlich schneller als der modifizierte Huang-Algorithmus; zumindest dann, wenn man nur einen Prozessor auf der Cray T3E benutzt (siehe Abschnitt "Tests" ab Seite 9). Dies trifft auch dann zu, wenn man mehrere (4,16 oder 64) Prozessoren benutzt und an Stelle von BLAS/Lapack die parallele Version PBLAS/Scalapack verwendet.

Umfangreiche Tests zeigen, daß die optimale Rechenzeit in fast allen Fällen unter folgenden Bedingungen erzielt wird:

1. NB = MB = MM = 64, d.h.
   die Zeilenblockgröße der PBLAS-Prozeduren ist gleich der Spaltenblockgröße und ist gleich der Blockgröße des Huang-Algorithmus.

2. NPR = NPC = NPSQRT = Quadratwurzel(Prozessoranzahl), d.h.
   es wird ein (logisches) quadratisches Prozessorgitter innerhalb der PBLAS-Prozeduren verwendet.

Es spielt keine Rolle, ob das logische Prozessorgitter zeilenweise oder spaltenweise orientiert ist.
Diese Bedingungen gelten für die Dimension $n = 1536$.

Aus Abbildung 4.3.3S38 ist ersichtlich, daß der parallele Gauß-Algorithmus (Prozedur **PSGESV** aus Scalapack) für größere Prozessoranzahlen nicht wesentlich schneller ist als der Block-Huang-Algorithmus. Dies läßt sich dadurch erklären, daß das Problem (Dimension 1536) nicht groß genug ist, genauer:
Die Autoren von Scalapack empfehlen die Verwendung von $\frac{n \cdot n}{1000000}$ Prozessoren für Probleme der Dimension n.

Mit der gleichen Begründung (zu kleine Problemgröße) kann man nachvollziehen, warum die parallele Effizienz so gering ist (siehe Abbildung 4.3.5S39).
Die zu kleine Problemgröße bedeutet, daß die Kommunikation (d.h. der serielle Anteil) im Vergleich zur Komplexität der Rechenoperationen nicht mehr vernachläsigbar ist.
Weil wir uns jedoch für Probleme mittlerer Größe interessieren, präsentieren wir hier keine weiteren Testergebnisse für größere Dimensionen.

Die parallelen Tests auf der Cray T3E bestätigen die aus anderen Experimenten bekannte Regel, daß die Verwendung von Level 3 Blas (Matrix-Matrix-Operationen) die beste Performance bringt.
Insgesamt läßt sich festhalten, daß Algorithmen, die Block-Operationen enthalten, in der Praxis schneller sein können, obwohl sie es der Theorie nach nicht sind. Diese Aussage ist rechner- und/oder bibliotheks-spezifisch.

5 Paralleler Sparse-Block-Huang-Algorithmus

5.1 Besonderheiten

Der Sparse-Block-Huang-Algorithmus läßt sich in der ab Seite 20 präsentierten Form nicht so einfach parallelisieren wie der Dense-Block-Huang-Algorithmus. Eine genaue Betrachtung des Programmlistings zeigt, daß die Blockstruktur bereits explizit programmiert wurde und nicht mehr in den Lapack-Prozeduren "versteckt" ist. Dies bedeutet, daß man nicht einfach die seriellen Lapack-Prozeduren durch parallele Scalapack-Prozeduren ersetzen kann; stattdessen müßte man auf die BLACS-Prozeduren zur Kommunikation zurückgreifen und selbstständig für eine geeignete Verteilung der Blöcke auf die einzelnen Prozessoren sorgen. Wenn der begrenzte Speicher kein Problem darstellen würde, dann könnte man den Sparse-Block-Huang-Algorithmus in unveränderter Form beibehalten und für die Parallelisierung eine Sub-Block-Struktur der $m \times m$-Blöcke anstreben. Da aber die Blockgröße m sehr viel kleiner als die Dimension n ist, würde sich dies nur für extrem große Matrizen (z.B. $n = 100000$, $m = 1000$) lohnen, im Widerspruch zur erwähnten Speicherbegrenzung.

Der einzig sinnvolle Weg besteht darin, daß man versucht, sich von der Notwendigkeit der Speicherung einer vollen, symmetrischen Matrix zu befreien.

Problem 5.1.1 (Alternative Formen des Sparse-Block-Huang-Algorithmus) :

Finde eine alternative Form des Sparse-Block-Huang-Algorithmus, bei der auf die Speicherung der gesamten Projektionsmatrix verzichtet werden kann.

5.2 Diskussion

Wenn wir ein lineares Gleichungssystem mit dünn besetzter Koeffizientenmatrix auf einem Parallelrechner lösen möchten, dann sollten wir nach den in dieser Arbeit gesammelten Erfahrungen nicht auf den Sparse-Block-Huang-Algorithmus zurückgreifen,

- weil Speicherplatz für eine symmetrische Matrix der Problemdimension n zusätzlich benötigt wird,
- weil das erzielbare Speedup in der Größenordnung **Sparse-Speedup * (paralleles Speedup des Dense-Algorithmus)** nur geringen Anforderungen genügt und
- weil es erfolgreichere Iterations-Verfahren gibt (siehe z.B. [Schendel(3)]).

6 Ausblick

6.1 Ergänzungen

Variable Blockgrößen ermöglichen, zusammen mit Zeilenvertauschungen, einen adaptiven Algorithmus mit "Pivotisierungstechnik". Die Idee der Zeilenvertauschungen geht bereits auf [Huang] zurück und verbessert einerseits die Stabilität des Algorithmus und ermöglicht andererseits die Lösung von unter-/überbestimmten Systemen. Geeignete Strategien zur Wahl der Blockgröße oder zur "Block-Pivotisierung" sind den Autoren jedoch nicht bekannt. Man wird allerdings im parallelen Fall bestrebt sein, keine Pivotisierungen durchzuführen, weil sie wahrscheinlich serielle Elemente zusätzlich einführen.

Wenn eine möglichst hohe Genauigkeit verlangt wird, dann empfehlen wir die Verwendung der Reprojektionstechnik (**modifizierter Block-Huang-Algorithmus** 2.4.1S7) zusammen mit einer nicht begründbaren Blockgröße im Bereich $16 - 64$.

Der modifizierte Huang-Algorithmus ist in gewisser Weise zum modifizierten Schmidtschen Orthogonalisierungsverfahren äquivalent (siehe [Abaffy/Spedicato]). Möglicherweise führen Erkenntnisse über das (modifizierte) Block-Schmidt-Orthogonalisierungsverfahren auch zu neuen Erkenntnissen über den (modifizierten) Block-Huang-Algorithmus.

In der folgenden Definition wird das modifizierte Block-Schmidt-Orthogonalisierungsverfahren zur Lösung eines linearen Gleichungssystems in der Programmiersprache des
Euler for Windows 95 V 1.30 Mathematiklabors
(Freeware von
Dr. Rene Grothmann
Ahornweg 5a
D-85117 Eitensheim Germany
EMail: grothmam.ku-eichstaett.de)
widergegeben. Dieses Labor eignet sich hervorragend zum schnellen Testen von Algorithmen der linearen Algebra.

Definition 6.1.1 (mod. Block-Schmidt-Orthogonalisierungsverfahren) :

```
function modblockschmidt(m,a,b)
eps = 1e-16;
n = rows(b);
h = mod(n,m);
l = n/m;
x = b-b;
p = id(n);
q = p;
for k=1 to l;
   ks = (k-1)*m;
   t = a[ks+1:ks+m,:]';
   r = a[ks+1:ks+m,:].x-b[ks+1:ks+m];
   s = t;
   for j=1 to k-1;
      js = (j-1)*m;
      w = q[js+1:js+m,:].t;
      s = s+p[:,js+1:js+m].w;
   end;
   t = s;
   for j=1 to k-1;
      js = (j-1)*m;
      w = q[js+1:js+m,:].s;
      t = t+p[:,js+1:js+m].w;
   end;
   ttt = t'.t;
   v = t'|r;
   v = -ttt\v;
   p[:,ks+1:ks+m] = t;
   q[ks+1:ks+m,:] = v[:,1:n];
   x = x+t.v[:,n+1];
end;
return x;
endfunction;
```

6.2 Offene Fragen

Der Block-Huang-Algorithmus sollte hinsichtlich Stabilität, Fehlerfortpflanzung, etc. theoretisch mit dem ursprünglichen Huang-Algorithmus verglichen werden.

Wenn man es auch aus den präsentierten Testergebnissen nicht so offensichtlich ablesen kann, dennoch weisen umfangreiche Tests auf

Vermutung 6.2.1 (Optimale Blockgröße) :
Sowohl in Bezug auf die benötigte Rechenzeit, als auch in Bezug auf die erzielbare Genauigkeit lassen sich bei Verwendung des Block-Huang-Algorithmus optimale Blockgrößen angeben, die abhängen von

1. dem Problem (d.h. von der Matrix A und der rechten Seite b),
2. dem verwendeten Rechner (insbesondere der Fließkomma-Hardware),
3. der Programmiersprache / dem Compiler,
4. den verwendeten mathematischen Bibliotheken.

Ein Ziel besteht in einer Präzisierung dieser Aussage und der Gewinnung von Faustregeln.

Möglicherweise bringt eine algorithmus-spezifische Parallelisierung mit besonderer Berücksichtigung einer geeigneten Kommunikationsstruktur zusätzliche Vorteile gegenüber der straight-foward parallelization with PBLAS/Scalapack.

Zeilen-Projektionsmethoden

Schendel definiert in [Schendel(2)] die

Definition 6.2.2 (Zeilen-Projektionsmethoden) :
A row projektion method for solving $Ax = b$ is any algorithm, which requires the computation of the orthogonal projections $P_i x := A_i^T (A_i A_i^T)^{-1} A_i x$ of a vector x onto the range of A_i^T for all $i = 1, \ldots, l$.
Hierbei ist die Matrix A in l Zeilenblöcke A_i zerlegt, genau wie beim Block-Huang-Algorithmus.

Um den Block-Huang-Algorithmus einordnen zu können, sollte man u.a. die Verbindungen zu den Projektionsmethoden näher untersuchen.

Vermutung 6.2.3 (Block-Alg. ist keine row projection method) :
Der Block-Huang-Algorithmus ist keine Zeilen-Projektionsmethode nach Definition 6.2.2S44. Als Beleg dient die aus [Schendel(4)] stammende Aussage, daß Zeilen-Projektionsmethoden nicht nach endlich vielen Schritten die exakte Lösung liefern.
Man erkennt, daß im Block-Huang-Algorithmus anstelle von
$A_i^* = A_i^T (A_i A_i^T)^{-1}$ die Moore-Penrose-Inverse $(A_i H_{i-1})^*$ vorkommt.
Kann man den Block-Huang-Algorithmus als "skalierte" Zeilen-Projektionsmethode auffassen ?

Literatur

[Abaffy/Spedicato] J.Abaffy, E.Spedicato.
ABS projection algorithms:
Mathematical techniques for linear and nonlinear equations.
Halsted Press. New York 1989.

[Bertocchi/Spedicato] M.Bertocchi, E.Spedicato.
Vectorizing the modified Huang algorithm of the ABS class on the IBM 3090.
Report No. DMSIA 1988/23. University of Bergamo.

[BrandSchen] J. Brandenburger, U. Schendel.
Block-Verfahren.
Preprint Nr. 262/87, Serie A Fachbereich Mathematik,
Freie Universität Berlin, 1987.

[Dekker] E. Dekker.
Interaction between algorithms and architectures in parallel computation.
Proefschrift, Technische Universiteit Delft, 1995.

[Duff] Edited by Ian S. Duff.
Sparse Matrices and their Uses.
Academic Press. London 1981.

[DufEriRei] I.S. Duff, A.M. Erisman, J.K. Reid.
Direct Methods for Sparse Matrices.
Clarendon Press. Oxford 1986.

[Huang] H.Y.Huang.
A direct method for the general solution of a system of linear equations.
J. of optim. theory and appl. 16(5/6), pp 429-445, 1975.

[Lapack] Lapack.
Linear Algebra Package. Library for Linear Algebra Software.
Details see www.netlib.org/lapack.
Links to Scalapack and to BLAS, PBLAS, BLACS, CBLAS, CLAPACK.

[Schendel(1)] Udo Schendel.
Algorithmen mit Blockmatrizen zur Lösung von Sparse-Matrix-Problemen.
Zugabe zu einer Vorlesung von 1992 an der Freien Universität Berlin.

[Schendel(2)] Udo Schendel.
Projection Methods for Solving Large Linear Systems.
Preprint No. A-18/93, Serie A Fachbereich Mathematik,
Freie Universität Berlin, 1993.

[Schendel(3)] Udo Schendel.
Parallele numerische Projektionsmethoden zur Lösung unsymmetrischer Gleichungssysteme.
Preprint No. A-7/94, Serie A Fachbereich Mathematik,
Freie Universität Berlin, 1994.

[Schendel(4)] Udo Schendel.
Spezielle Projektionsmethoden für Sparse-Matrizen.
Preprint No. A-15/98, Serie A Mathematik, Fachbereich Mathematik und Informatik, Freie Universität Berlin, 1998.

[Schendel(5)] Udo Schendel.
Sparse Matrizen und Anwendungen.
Vorlesungsskript der Freien Universität Berlin 1992.

[Schendel(6)] Udo Schendel.
Sparse-Matrices - Numerical Aspects with Applications for Scientists and Engineers.
John Wiley & Sons, New York 1989.