Parallele Algorithmen für $\mathcal{H}$-Matrizen

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Ronald Kriemann

Kiel
2004

Referent	Prof. Dr. Dr. h.c. Wolfgang Hackbusch
Koreferenten	Prof. Dr. Gerhard Zumbusch
	Prof. Dr. Peter Bastian
Datum der mündlichen Prüfung	15. April 2005
Zum Druck genehmigt	19. September 2005

Inhaltsverzeichnis

1 Einleitung

Bei der numerischen Behandlung von physikalischen, chemischen oder biologischen Problemen gelingt es in der Regel, diese durch Differential- oder Integralgleichungen zu beschreiben. Als Beispiele seien hier etwa der Wärmetransport, die Navier-Stokes-Gleichungen bei der Behandlung von Strömungen oder die in der Quantenmechanik wesentliche Schrödingergleichung genannt.

Bei vielen dieser Probleme führt der Lösungsprozess auf ein lineares Gleichungssystem der Form

$$Ax = y.$$

Hierbei enthält x die zu bestimmenden Koeffizienten der gesuchten Lösung, während die Matrix A die Beziehungen der einzelnen unbekannten Faktoren untereinander beschreibt. Der Vektor y bestimmt die zusätzlichen, äußeren Bedingungen an die Unbekannten.

Unter entsprechenden Voraussetzungen ist eine einfache und elegante Lösung für dieses Problem durch die lineare Algebra vorgegeben: die Inversion von A. Hierdurch genügt eine Multiplikation mit dem Vektor y, um den unbekannten Vektor x zu berechnen:

$$x = A^{-1}y.$$

Einzig offen bleibt die Frage nach der Bestimmung von A^{-1}, doch auch hier ist eine einfache und robuste Methode durch das *Gauß'sche Eliminationsverfahren* gegeben. Leider erweist sich dieser Algorithmus in der Praxis nur für kleine Probleme als tauglich, da der Aufwand im allgemeinen kubisch mit der Zahl der Unbekannten ansteigt. Auf der anderen Seite führt die Forderung nach einer hinreichend genauen Lösung der zugrundeliegenden Aufgabe häufig zu einer großen Zahl von Unbekannten. Zusammen mit der Gauß-Elimination nimmt die benötigte Ausführungszeit auf einem Computersystem somit schnell unannehmbare Werte an. Der einfache Ansatz über die Inversion von A kann deshalb oftmals nur für Spezialfälle eingesetzt werden.

Anstelle dessen wird auf Verfahren zurückgegriffen, welche die Multiplikation der Inversen mit einem Vektor erlauben. Hierzu ist die explizite Kenntnis der Matrix A^{-1} nicht zwingend erforderlich. Häufig kommen dabei *iterative* Methoden zum Einsatz, etwa die *Jacobi-* oder die *Gauß-Seidel-Iteration*. Der Aufwand pro Iterationsschritt hängt in diesen Fällen von der Zahl der Matrixkoeffizienten ab, wobei eine Differentialgleichung typischerweise zu einer linearen, eine Integralgleichung zu einer quadratischen Komplexität führt. Man spricht in diesem Zusammenhang auch von *schwach-* bzw. *vollbesetzten* Matrizen.

Durch den Einsatz einer iterativen Methode wird allerdings im allgemeinen nur eine Approximation der Lösung des obigen Problems bestimmt. Die exakte Berechnung von x ist in der Praxis nur selten möglich und nötig. Ein Grund hierfür liegt z.B. in kontinuierlichen Ausgangsproblemen, welche zu nicht-endlichen Gleichungssystemen führen. Um diese Systeme trotzdem numerisch auf einem endlichen Rechnersystem zu behandeln sind Einschränkungen notwendig, etwa die Beschränkung auf eine endliche Zahl von Punkten, für die die Lösung bestimmt wird. Der durch diese *Diskretisierung* verursachte Fehler definiert eine Schranke für die Bestimmung der Unbekannten im späteren Lösungsprozess.

Trotz der Beschränkung auf approximative Lösungen ist die Anzahl der Iterationen bei den erwähnten klassischen Verfahren häufig von der gleichen oder ähnlichen Ordnung wie die Zahl der Unbekannten, womit der Vorteil der einfachen Algorithmen im Vergleich zur Gauß-Elimination, insbesondere für vollbesetzte Matrizen, verschwindet. Aber auch eine quadratische Komplexität bei schwachbesetzten Systemen ist in der Regel für große Probleme nicht praktikabel.

Unter Ausnutzung weiterer Eigenschaften des Ausgangsproblems lassen sich verbesserte, approximative Methoden wie das *Mehrgitterverfahren* (siehe [Hac85]) einsetzen, welches unter geeigneten Voraussetzungen nur eine konstante Anzahl von Schritten benötigt. Für den Fall einer schwachbesetzten Matrix liefert dieser Ansatz somit bereits einen Gesamtaufwand von linearer Ordnung. Allerdings ist diese Methode nicht immer anwendbar.

Die beschriebene Fehlerschranke der Diskretisierung erlaubt es aber oftmals, auf eine exakte Darstellung der Systemmatrix zu verzichten. Insbesondere bei Problemen die zu vollbesetzten Matrizen führen, lassen sich häufig große Bereiche der Matrix sehr gut approximieren. Hierdurch sinkt der Aufwand zur Speicherung und zur Matrix-Vektor-Multiplikation typischerweise auf eine lineare bzw. linear-logarithmische Ordnung, womit ein vergleichbarer Aufwand wie bei schwachbesetzten Systemen erreicht wird.

Übliche Verfahren zur Komprimierung einer Matrix sind z.B. die *schnelle Multipol-Methode* (siehe [Rok85]) oder die *Paneel-Clusterung* (siehe [HN89]). Desweiteren kommen Algorithmen basierend auf *Wavelets* zum Einsatz. All diese Verfahren gestatten allerdings lediglich die Anwendung der Matrix-Vektor-Multiplikation zur Lösung des obigen Problems mittels Iterationsverfahren.

Eine weitere Möglichkeit bieten die hierarchischen Matrizen oder kurz $\mathcal{H}$-Matrizen (siehe [Hac99]), wie sie auch Gegenstand dieser Arbeit sind und in den Kapiteln 2 bzw. 3 näher definiert werden. Der wesentliche Unterschied zwischen den bisher genannten Algorithmen und den $\mathcal{H}$-Matrizen ist die Möglichkeit, die gesamte Algebra, d.h. neben der Matrix-Vektor-Multiplikation auch die Addition, Multiplikation und Inversion, in linear-logarithmischer Zeit durchzuführen. Die Arithmetik arbeitet hierbei allerdings nicht exakt, sondern führt ebenfalls zu approximativen Ergebnissen. Da diese aber, wie erwähnt, ausreichend für den praktischen Einsatz sind, bieten die $\mathcal{H}$-Matrizen somit ein mächtiges Werkzeug, um obige Probleme elegant zu lösen.

Trotz der fast-linearen Komplexität von $\mathcal{H}$-Matrix-Algorithmen ergibt sich für Probleme

mit einer großen Zahl von Unbekannten eine relativ hohe Laufzeit. Hier kommt deshalb sehr schnell der Wunsch nach einer Steigerung der Ausführungsgeschwindigkeit auf. Neben effizienteren sequentiellen Verfahren bildet die Parallelisierung, d.h. die Verteilung der auftretenden Aufgaben auf mehrere Prozessoren eines Rechnersystems, ein in diesem Zusammenhang probates Mittel. Dass ein solches Vorgehen nicht nur auf wenigen Spezialsystemen Vorteile bringt, zeigt die rasante Zunahme von Mehrprozessorsystemen z.B. im Bereich der Arbeitsplatzrechner. Diese Systeme sind häufig mit bis zu vier Prozessoren ausgestattet. Auch komplexere Rechner mit 16 oder 32 CPUs (*Central Processing Units*) sind als lokale Server weit verbreitet. Eine Alternative zu solchen, immer noch vergleichsweise teuren Systemen bietet ein sogenannter *Cluster*, bei welchem einfache und somit günstige Rechner durch ein schnelles Netzwerk miteinander verbunden sind. Hierdurch lassen sich Prozessorzahlen von mehreren Hundert oder gar Tausend erreichen.

In dieser Arbeit sollen verschiedene Techniken und Algorithmen, die bei der Parallelisierung der elementaren Verfahren im Kontext der $\mathcal{H}$-Matrizen zum Einsatz kommen, vorgestellt werden. Eine wesentliche Grundlage bilden hierbei die Lastbalancierungsalgorithmen in Kapitel 4. Da das eigentliche Verteilungsproblem auf die Prozessoren eines Rechners im allgemeinen NP-schwer ist, also mit vertretbarem Aufwand nicht lösbar, werden Verfahren vorgestellt, die eine gute Approximation einer optimalen Lösung bestimmen. Wichtig ist in diesem Zusammenhang auch die Anwendung dieser Verfahren auf die Datenstrukturen einer $\mathcal{H}$-Matrix. Die Art und Weise dieser Adaption entscheidet häufig darüber, ob der resultierende $\mathcal{H}$-Matrix-Algorithmus die verschiedenen Prozessoren ausnutzen kann oder nicht. Hier gelingt es z.B. mit Hilfe von raumfüllenden Kurven, effektive Verteilungen zu generieren.

Der Entwurf der eigentlichen parallelen Algorithmen wird auf entscheidende Art geprägt von dem Rechnersystem, auf dem das endgültige Programm schließlich ausgeführt werden soll. So haben die erwähnten Arbeitsplatzrechner oder die Serversysteme häufig einen gemeinsamen Arbeitsspeicher, d.h. jeder Prozessor kann uneingeschränkt auf alle Daten des Speichers zugreifen. Somit entfällt jegliche Kommunikation zwischen den einzelnen CPUs. Demgegenüber muss der Datenaustausch auf einem Cluster explizit durch den Programmierer gesteuert werden. Diese unterschiedlichen Eigenschaften führen daher auch zu verschiedenen Algorithmen für das gleiche Problem.

Um sich allerdings bei der Implementierung der Verfahren nicht explizit auf einen bestimmten Rechner festlegen zu müssen, werden in der Regel allgemeine Modelle zugrundegelegt. Diese Modelle enthalten die wesentlichen Eigenschaften eines Rechnersystems, etwa einen geteilten Speicher oder die Eckdaten des Kommunikationsnetzwerkes in einem Cluster. Durch die Verwendung einer solchen Beschreibung der Rechnerarchitektur lässt sich zudem die Laufzeit auf einem realen System anhand weniger Daten vorhersagen, wodurch die Angabe von Komplexitäten für die einzelnen parallelen Algorithmen ermöglicht wird. Die in dieser Arbeit genutzten Rechnermodelle und die jeweilige Art der Programmierung werden in Kapitel 5 vorgestellt.

Gegenstand von Kapitel 6 sind schließlich die eigentlichen parallelen Verfahren für $\mathcal{H}$-

Matrizen. Hierbei werden je nach Rechnermodell verschiedene Verfahren vorgestellt und deren Eigenschaften diskutiert. Die Algorithmen basieren zum Teil auf den jeweiligen sequentiellen Pendants, welche in Kapitel 2 diskutiert werden. Hierdurch lassen sich z.B. existierende Implementierungen auf einfache Weise parallelisieren. Allerdings führt ein solcher Ansatz nicht bei allen Verfahren zu einer optimalen parallelen Leistung. Deshalb werden in diesen Fällen auch alternative Ansätze vorgestellt.

Neben der direkten Parallelisierung der $\mathcal{H}$-Matrix-Arithmetik, d.h. der expliziten Verteilung der einzelnen Aufgaben, kommt in Kapitel 7 auch die Technik der Gebietszerlegung zur Sprache. Der Name leitet sich hierbei aus einer Aufteilung der zugrundeliegenden Daten ab, z.B. des Gebietes über dem das anfängliche Problem betrachtet wird. Dabei zerfällt auch das zu lösende Problem in Einzelaufgaben, welche mehr oder weniger unabhängig voneinander gelöst werden können. Die $\mathcal{H}$-Matrix-Algorithmen dienen in diesem Fall der Lösung dieser einzelnen Probleme. Welcher Aufwand hierbei auftritt und welche Aufgaben überhaupt mit dieser Technik effizient lösbar sind, soll in diesem Kapitel diskutiert werden.

Im Zusammenhang mit den $\mathcal{H}$-Matrizen treten häufig Probleme aus unerwarteten Bereichen auf. So benötigt z.B. eine entsprechend große $\mathcal{H}$-Matrix mitunter mehrere 10 Gigabyte an Arbeitsspeicher. Dieser Speicher ist hierbei in eine Vielzahl von kleineren Bereichen unterteilt und wird nicht als einheitlicher Speicherblock angefordert. Werden zusätzlich noch alle Einzelblöcke von verschiedenen Prozessoren verwaltet, so führt dies bei vielen Speicherverwaltungen zu einer sehr großen Laufzeit. Somit hat die Technik, die bei der Verwaltung von Arbeitsspeicher zum Einsatz kommt, einen wesentlichen Einfluss auf die zu beobachtende Komplexität der $\mathcal{H}$-Matrix-Algorithmen. Auf solche Speicherverwaltungstechniken soll in Kapitel 8 eingegangen werden. Dabei wird insbesondere eine speziell auf die $\mathcal{H}$-Matrix-Arithmetik zugeschnittene Implementierung vorgestellt.

Danksagung

Meinen Dank möchte ich Prof. Dr. Dr. h.c. W. Hackbusch für die Betreuung dieser Arbeit und die Möglichkeit aussprechen, währenddessen auch an vielen anderen Projekten mitwirken zu können, die einen positiven Einfluss auf dieses Werk hatten. Desweiteren gilt mein Dank in erster Linie Dr. Lars Grasedyck und Dr. Steffen Börm für viele intellektuelle Stupser in alternative und oftmals richtige Richtungen. Für weitere Anregungen dieser und anderer Art sowie für moralische Aufbauarbeit bin ich außerdem Dr. Maike Löhndorf und Dr. Kai Helms zu Dank verpflichtet.

Für ihr aufopferungsvolles Engagement bei der Korrektur dieser Arbeit danke ich in besonderem Maße Dr. Sabine Le Borne und Dr. Lars Grasedyck.

Und dafür, dass sie zahlreiche Entbehrungen während des Zeitraums der Erstellung dieser Arbeit auf sich nahm, möchte ich Nadine Rudel meinen Dank aussprechen.

2 $\mathcal{H}$-Matrizen

In diesem Kapitel sollen die grundlegenden Begriffe im Kontext der hierarchischen Matrizen eingeführt werden, wie sie ursprünglich in [Hac99] beschrieben wurden. Desweiteren dient dieser Abschnitt der Benennung von wesentlichen Eigenschaften der $\mathcal{H}$-Matrizen, etwa des Aufwands zur Speicherung.

Anhand zweier Beispiele aus den Bereichen der Integralgleichungen und der partiellen Differentialgleichungen wird außerdem die praktische Anwendung der $\mathcal{H}$-Matrizen demonstriert. Diese Beispiele werden in den späteren Kapiteln bei verschiedenen numerischen Experimenten genutzt.

2.1 Bäume

Eine wesentliche Grundlage der hierarchischen Matrizen bilden Bäume und deren Eigenschaften.

Definition 2.1.1 (Baum) *Sei V eine endliche Menge und $E \subset V \times V$ eine Relation über V. Die Elemente von V heißen* Knoten *und die Elemente von E* Kanten*. Die Knoten $v_0, v_1, \ldots, v_n$ der Kantenfolge $(v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)$ bilden einen* Weg *der Länge n, falls für alle $0 \leq i, j \leq n$ mit $i \neq j$ gilt: $v_i \neq v_j$.*

Das Paar $T = (V, E)$ heißt Baum*, wenn*

1. *es genau einen Knoten* $\mathrm{root}(T) \in V$ *gibt, so dass für alle Knoten $v \in V \setminus \{\mathrm{root}(T)\}$ gilt:* $(v, \mathrm{root}(T)) \notin E$ *und*

2. *für alle Knoten $v \in V \setminus \{\mathrm{root}(T)\}$ existiert genau ein Weg* $\mathrm{root}(T), \ldots, v$*.*

Der Knoten $\mathrm{root}(T)$ *wird als* Wurzel *von T bezeichnet.*

Direkt aus der Definition ergibt sich die folgende wichtige Eigenschaft von Bäumen.

Bemerkung 2.1.2 *In einem Baum $T = (V, E)$ lassen sich keine Knoten $u, v \in V$ mit $(v, u) \in E$ derart finden, dass $u, \ldots, v$ einen Weg bilden. Bäume sind somit* azyklisch*.*

In der folgenden Definition werden wichtige Begriffe für den späteren Umgang mit Bäumen vorgestellt.

Definition 2.1.3 *Sei $T = (V, E)$ ein Baum. Die Knoten- und Kantenmenge von T wird auch mit $V(T) = V$ bzw. $E(T) = E$ bezeichnet.*

1. *Für einen Knoten $v \in V$ werden die Elemente der Menge $\mathcal{S}(v) = \{u \,|\, (v, u) \in E\}$ als* Söhne *bezeichnet. Für $u \in \mathcal{S}(v)$ wird auch $v \to u$ geschrieben. Existiert für zwei Knoten $v, v' \in V$ ein Weg $v = v_0, \ldots, v_n = v'$ mit $v_i \to v_{i+1}$ für alle $0 \leq i < n$, so wird dies durch $v \xrightarrow{*} v'$ abgekürzt.*

2. *Der* Grad *eines Knotens $v \in V$ sei definiert als: $d(v) = |\mathcal{S}(v)|$.*

3. *Knoten $u \in V$ mit $d(u) = 0$ heißen* Blätter. *Die Menge $\mathcal{L}(T) = \{u \in V \,|\, \mathcal{S}(u) = \emptyset\}$ bezeichne die Menge aller Blätter von T.*

4. *Ein Knoten $u \in V \setminus \mathcal{L}(V)$ wird* innerer *Knoten von T genannt.*

5. *Mit $T^{(i)}$ wird die Menge aller Knoten $v \in V$ bezeichnet, für die der Weg $\mathrm{root}(T), \ldots, v$ die Länge i besitzt. Dabei heißt i auch* Stufe *von $T^{(i)}$. Die Einschränkung auf die Blätter der Stufe i definiert die Menge $\mathcal{L}(T, i) = T^{(i)} \cap \mathcal{L}(T)$.*

6. *Die Länge des längsten Weges $\mathrm{root}(T), \ldots, v$ für $v \in T$ heißt* Tiefe *$p(T)$ des Baumes.*

7. *Sei $v \in V$. Der Baum $T(v) = (V_v, E_v)$ mit $V_v = \left\{u \,|\, v \xrightarrow{*} u\right\}$ und $E_v = E \cap V_v \times V_v$ heißt der durch v induzierte* Teilbaum *von T.*

Wichtig für spätere Komplexitätsaussagen ist ein Zusammenhang zwischen der Tiefe eines Baumes und der Größe der Blattmenge. Um hierbei zu effizienten Verfahren zu gelangen werden in den folgenden Abschnitten üblicherweise nur Bäume betrachtet, bei denen die inneren Knoten mindestens zwei Söhne besitzen. Hierdurch ist die Baumtiefe an die Anzahl der Blätter gekoppelt und somit eine beliebig große Baumtiefe verhindert. Wichtige Teilklassen solcher Bäume charakterisiert die folgende Definition.

Definition 2.1.4 (Binär- und Quadbäume) *Sei $T = (V, E)$ ein Baum. Für den Fall, dass alle inneren Knoten von T den Grad 2 besitzen, wird T als* Binärbaum *bezeichnet. T heißt dagegen* Quadbaum*, falls alle inneren Knoten den Grad 4 haben.*

Weiterhin heißt T balanciert*, wenn für alle $v \in V$ und für alle Söhne $u, w \in \mathcal{S}(v)$ gilt: $|p(T(u)) - p(T(w))| \leq 1$.*

Wie das folgenden Lemma zeigt, wird die Zahl der Knoten in solchen Bäumen dominiert durch die Kardinalität der Blattmenge.

Lemma 2.1.5 *Sei $T = (V, E)$ ein Baum. Für alle inneren Knoten $v \in V \setminus \mathcal{L}(T)$ gelte: $d(v) \geq 2$. Dann folgt:*

$$|V \setminus \mathcal{L}(T)| < |\mathcal{L}(T)|. \tag{2.1.1}$$

Beweis: Es seien $n_i = |V \setminus \mathcal{L}(T)|$, $n_\ell = |\mathcal{L}(T)|$ und $n = n_i + n_\ell$. Da T ein Baum ist gilt (siehe Satz 2.4 in [CH94]): $|E| = n - 1$. Nach Voraussetzung gilt weiterhin: $|E| \geq 2n_i$. Damit folgt: $n - 1 = n_i + n_\ell - 1 \geq 2n_i$ bzw. $n_\ell \geq n_i + 1$ und somit die Behauptung. □

Aufgrund dieses Ergebnisses genügt es bei vielen der späteren Aufwandsabschätzungen in Bäumen nur die Blattmenge zu betrachten, wodurch sich die Analyse vereinfacht.

2.2 Clusterbaum und Blockclusterbaum

Analog zu den Bäumen werden im folgenden stets nur endliche Indexmengen betrachtet. Die effiziente Behandlung der gesamten Indexmenge ist in der Regel nicht möglich, da dieser Ansatz üblicherweise zu vollbesetzten Matrizen führt. Statt dessen werden einzelne Teilindexmengen betrachtet. Diese Teilmengen bilden disjunkte Überdeckungen der Indexmenge und sind hierarchisch aufgebaut. Zusammen definieren sie den *Clusterbaum*.

Definition 2.2.1 (Clusterbaum) *Sei I eine Indexmenge. Dann heißt $T = T(I) = (V, E)$ mit $V \subset \mathcal{P}(I)$ und $E \subseteq V \times V$* Clusterbaum *über I, falls*

1. *$root(T) = I$ und*
2. *für alle $v \in V \setminus \mathcal{L}(T(I))$ gilt: $v = \dot{\cup}_{u \in \mathcal{S}(v)} u$.*

In der Praxis ist es aus Effizienzgründen in der Regel sinnvoll, nicht zu kleine Blätter eines Clusterbaumes zuzulassen. Deshalb wird mittels eines $n_{\min} > 0$ eine Schranke für die Kardinalität von Knoten eingeführt, d.h. für alle $v \in T$ gilt: $|v| \geq n_{\min}$.

Das Analogon zu Clusterbäumen für die hierarchische Überdeckung einer Produktindexmenge $I \times J$ bilden *Blockclusterbäume*. Sie sind definiert als das Produkt zweier Clusterbäume.

Definition 2.2.2 (Blockclusterbaum) *Seien $T(I)$ und $T(J)$ Clusterbäume über den Indexmengen I, J. Dann heißt $T = T(I \times J) = T(T(I) \times T(J)) = (V, E)$ mit $V \subset \mathcal{P}(I) \times \mathcal{P}(J)$* Blockclusterbaum *bezüglich $T(I), T(J)$, falls für alle $v = (v_I, v_J) \in V$ gilt:*

1. *$v \in T^{(i)} \implies \left(v_I \in T(I)^{(i)} \ \wedge \ v_J \in T(J)^{(i)}\right)$ und*
2. *$u = (u_I, u_J) \in \mathcal{S}(v) \implies (u_I \in \mathcal{S}(v_I) \ \wedge \ u_J \in \mathcal{S}(v_J))$.*

Einige Eigenschaften der Blockclusterbäume leiten sich direkt aus Merkmalen der zugrundeliegenden Clusterbäume ab:

Bemerkung 2.2.3 *Für die Tiefe von $T(I \times J)$ gilt*

$$p(T(I \times J)) = \min \{p(T(I)), p(T(J))\}$$

Die Anzahl der Knoten auf der Stufe i ist beschränkt durch

$$|T(I \times J)^{(i)}| \leq |T(I)^{(i)}| \cdot |T(J)^{(i)}|.$$

Somit gilt auch für die Blätter im Blockclusterbaum:

$$|\mathcal{L}(T(I \times J))| \leq |\mathcal{L}(T(I))| \cdot |\mathcal{L}(T(J))|. \tag{2.2.1}$$

Ist die Größe der Knoten in $T(I)$ und $T(J)$ beschränkt durch $n_{\min}$, so folgt für alle $v \in T(I \times J)$: $|v| \geq n_{\min}^2$.

Für einen Clusterbaum mit einer Blattmenge der Kardinalität $|\mathcal{L}(T(I))| = n$ gilt somit, dass die Anzahl der Blätter von $T(I \times I)$ nur durch n^2 beschränkt ist. Diese Zahl führt aber zu einer vollbesetzten Arithmetik und ist deshalb zu groß. Aus diesem Grund wird ein weiteres Kriterium eingeführt, welches die Menge der Blätter des Blockclusterbaumes reduziert.

Definition 2.2.4 (Zulässigkeitsbedingung) *Sei $T = T(I \times J)$ ein Blockclusterbaum über den Indexmengen I und J. Eine Funktion $z : V(T) \to \mathbb{B} = \{\texttt{false}, \texttt{true}\}$ heißt* Zulässigkeitsbedingung*, falls*

$$\forall\, v \in V(T)\, \forall\, u \in \mathcal{S}(v) : z(v) = \texttt{true} \implies z(u) = \texttt{true}.$$

Ein Knoten $v \in V(T)$ heißt zulässig *bezüglich z oder z-*zulässig*, falls $z(v) = \texttt{true}$. Weiterhin seien*

$$\mathcal{L}_{\mathrm{z}}(T) = \{v \in \mathcal{L}(T) \,|\, z(v) = \texttt{true}\} \quad \textit{und} \quad \mathcal{L}_{\mathrm{nz}}(T) = \{v \in \mathcal{L}(T) \,|\, z(v) = \texttt{false}\} \tag{2.2.2}$$

die Mengen der zulässigen bzw. unzulässigen Blätter von T.

Die jeweils verwendete Zulässigkeitsbedingung ist abhängig vom betrachteten Problem. Beispiele hierfür sind in den Abschnitten 2.4.1 und 2.4.2 zu finden.

Die Beschränkung der Blattmenge auf zulässige Knoten mit nichtzulässigen Vätern führt zu den *minimal zulässigen* Blockclusterbäumen.

Definition 2.2.5 *Sei T ein Blockclusterbaum und z eine Zulässigkeitsbedingung. T heißt zulässig bezüglich z oder z-zulässig, falls*

$$\forall\, v \in \mathcal{L}(T) : z(v) = \texttt{true}.$$

T wird als minimal zulässig *bezüglich z bezeichnet, falls T z-zulässig ist und*

$$\forall\, v \in V(T) \setminus \mathcal{L}(T) : z(v) = \texttt{false}.$$

Aufgrund der hierarchischen Definition des Blockclusterbaumes ist die Konstruktion eines minimal zulässigen Baumes bezüglich einer gegebenen Zulässigkeitsbedingung z mit optimalem Aufwand möglich. Der entsprechende Algorithmus 2.2.1 benutzt hierzu eine Tiefensuche (siehe [Tar72]), um alle Knoten des Baumes zu erreichen und stoppt die Rekursion, sobald ein zulässiger Knoten gefunden wurde oder ein Blatt im Clusterbaum auftritt.

```
procedure bct_build( τ, σ, z )
    if z(τ, σ) = true ∨ S(τ) = ∅ ∨ S(σ) = ∅ then
        konstruiere das Blatt (τ, σ);
        return ;
    else
        for all τ' ∈ S(τ) do
            for all σ' ∈ S(σ) do
                konstruiere den inneren Knoten (τ, σ);
                bct_build( τ', σ' )
            endfor;
    endif;
end;
```

Algorithmus 2.2.1: Konstruktion eines minimal zulässigen Blockclusterbaumes

Je weniger Knoten ein Blockclusterbaum besitzt, umso *schwächer* ist er besetzt. Diese *Schwachbesetztheit* lässt sich durch eine Konstante charakterisieren.

Definition 2.2.6 (Schwachbesetztheit) *Seien $T(I)$ und $T(J)$ Clusterbäume über den Indexmengen I bzw. J und sei $T = T(T(I) \times T(J))$ ein Blockclusterbaum bezüglich $T(I)$ und $T(J)$. Dann heißt T* schwachbesetzt *zur Konstante c_{sp} („sp“ für „sparse“), falls gilt:*

$$\forall\, i \in \{0, \ldots, p(T(I))\}\ \forall\, v \in T(I)^{(i)} : \left|\left\{u \in V(T(J)) : (v,u) \in T^{(i)}\right\}\right| \leq c_{\mathrm{sp}},$$

$$\forall\, i \in \{0, \ldots, p(T(J))\}\ \forall\, u \in T(J)^{(i)} : \left|\left\{v \in V(T(I)) : (v,u) \in T^{(i)}\right\}\right| \leq c_{\mathrm{sp}}.$$

Die Schwachbesetztheit bzw. c_{sp} stehen in direktem Zusammenhang mit der Zulässigkeitsbedingung. Je restriktiver diese ist, umso größer ist c_{sp} und umso dichter ist der Blockclusterbaum besetzt. In Abbildung 2.2.1 sind Beispiele für Blockclusterbäume mit einer unterschiedlich großen Schwachbesetztheit angegeben, wobei die Blattmenge der einzelnen Bäume dargestellt ist.

Bemerkung 2.2.7 *Sei $T = T(I \times J)$ ein Blockclusterbaum mit der Schwachbesetztheitskonstante c_{sp} über den Clusterbäumen $T(I) = (V_I, E_I)$ und $T(J) = (V_J, E_J)$. Da es für jeden Knoten in $T(I)$ bzw. $T(J)$ nach Definition 2.2.6 maximal c_{sp} viele Knoten in T gibt, folgt:*

$$|T^{(i)}| \leq c_{\mathrm{sp}} \cdot \min\left\{|T(I)^{(i)}|, |T(J)^{(i)}|\right\} \quad \textit{und} \quad |V(T)| \leq c_{\mathrm{sp}} \cdot \min\{|V_I|, |V_J|\}\,. \tag{2.2.3}$$

Gilt außerdem $d(v) \geq 2$ für alle $v \in V(T)$, so folgt mit Lemma 2.1.5

$$|V(T)| = \mathcal{O}\left(c_{\mathrm{sp}} \cdot \min\{|I|, |J|\}\right), \tag{2.2.4}$$

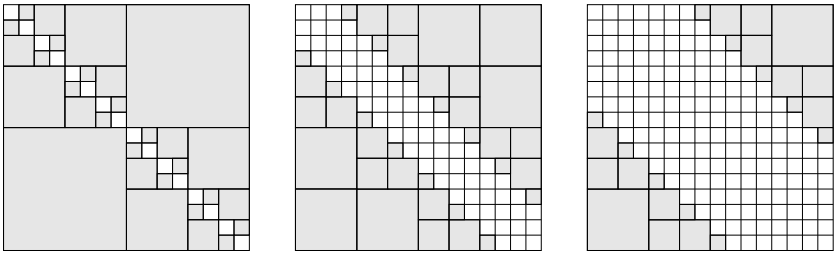


Abbildung 2.2.1: Beispiele für Blockclusterbäume mit wachsendem c_{sp}

da die Zahl der Blätter in $T(I)$ und $T(J)$ durch die Kardinalität der jeweiligen Indexmenge beschränkt ist.

Die Zahl der Knoten in einem Blockclusterbaum ist somit durch die Schwachbesetztheit und die Zahl der Knoten in den zugrundeliegenden Clusterbäumen bestimmt. Für eine im folgenden wichtige Klasse von Clusterbäumen, den binären Clusterbäumen, lassen sich die Abschätzungen (2.2.3) bzw. (2.2.4) präzisieren.

Bemerkung 2.2.8 (Blockclusterbaum über binären Clusterbäumen) *Sei I eine Indexmenge mit $n = |I|$ und sei $T(I)$ ein balancierter, binärer Clusterbaum über I. Dann gilt: $|T(I)^{(i)}| = 2^i$ für $0 \leq i \leq \log_2 n = p(T)$ und $|V(T(I))| = \mathcal{O}(n)$. Sei $T = T(I \times I)$ ein Blockclusterbaum über $T(I)$ schwachbesetzt bezüglich c_{sp}. Mit* (2.2.3) *und* (2.2.4) *folgt somit*

$$|T^{(i)}| \leq c_{\mathrm{sp}} \cdot 2^i \quad \textit{und} \quad |V(T)| = \mathcal{O}(c_{\mathrm{sp}} n)\,. \tag{2.2.5}$$

2.3 $\mathcal{H}$-Matrizen

Die Basis aller $\mathcal{H}$-Matrizen bilden Matrizen mit einem beschränkten Rang in einer speziellen Darstellung, die R-Matrizen.

Definition 2.3.1 (Rang-k-Matrizen) *Seien $\mathbb{K} \in \{\mathbb{R}, \mathbb{C}\}$ und I, J zwei Indexmengen mit $|I| = n$ und $|J| = m$. Eine Matrix $M \in \mathbb{K}^{n \times m}$ heißt $\mathcal{R}(k, n, m)$-Matrix bzw. $\mathcal{R}(k, I \times J)$-Matrix, falls M höchstens den Rang k besitzt.*

Besitzt M die Darstellung:

$$M = A \cdot B^T = \sum_{i=1}^{k} a_i b_i^T \tag{2.3.1}$$

mit $A = A_M \in \mathbb{K}^{n \times k}, B = B_M \in \mathbb{K}^{m \times k}$ und $a_i \in \mathbb{K}^n, b_i \in \mathbb{K}^m$, $1 \leq i \leq k$, so wird M als $\mathsf{R}(k, n, m)$-Matrix bezeichnet.

Die Eigenschaften der $\mathsf{R}(k)$-Matrizen bestimmen die wesentlichen Merkmale der Speicherkomplexität von $\mathcal{H}$-Matrizen und den Aufwand der Algorithmen bei der $\mathcal{H}$-Arithmetik. Insbesondere die Addition und Multiplikation von $\mathsf{R}(k)$-Matrizen sind dabei entscheidend.

Bemerkung 2.3.2 (Addition und Multiplikation von $\mathsf{R}(k)$-Matrizen) *Es seien M_1 und M_2 $\mathsf{R}(k,n,m)$-Matrizen und $M_3 \in \mathbb{K}^{m\times l}$. Dann gilt:*

1. *Die Summe $M_1 + M_2$ ist eine $\mathsf{R}(2\cdot k)$-Matrix.*
2. *Das Produkt $M_1 \cdot M_3$ ist eine $\mathsf{R}(k,n,l)$-Matrix. Somit besitzen $\mathcal{R}(k)$-Matrizen eine Idealeigenschaft bezüglich der Multiplikation.*

Zu den $\mathcal{H}$-Matrizen gelangt man schließlich, indem zulässige Blätter eines Blockclusterbaumes mit $\mathsf{R}(k)$-Matrizen assoziiert werden.

Definition 2.3.3 ($\mathcal{H}$-Matrizen) *Seien I, J Indexmengen, T ein Blockclusterbaum über I und J, $z : V(T) \to \mathbb{B}$ eine Zulässigkeitsbedingung und $k : \mathcal{L}(T) \to \mathbb{N}$. Sei weiterhin $M \in \mathbb{K}^{I\times J}$ eine Matrix.*

*Falls für alle z-zulässigen Knoten $v \in V$ der Matrixblock $M(v) = (M_{ij})_{(i,j)\in v}$ eine $\mathsf{R}(k(v))$-Matrix darstellt, heißt M $\mathcal{H}$-*Matrix *bezüglich T, z und k. Die Menge aller $\mathcal{H}$-Matrizen zu T, z und k wird mit $\mathcal{H}(T,k,z)$ bezeichnet.*

Die Matrix M_v heißt auch der zu v korrespondierende *Matrixblock von M. Umgekehrt wird mit $v(M) = (\tau,\sigma)$ der zu M* korrespondierende *Blockcluster bezeichnet. Weiterhin seien $\tau(M) = \tau$ und $\sigma(M) = \sigma$ die entsprechenden Knoten des zugehörigen Clusterbaumes.*

Ist der Rang oder die Zulässigkeitsbedingung aus dem Kontext ersichtlich oder spielt nur eine untergeordnete Rolle, so wird im folgenden auch die Schreibweise $\mathcal{H}(T,k)$ bzw. $\mathcal{H}(T)$ verwendet.

Definition 2.3.3 lässt offen, welches Format Matrixblöcke besitzen, die mit nichtzulässigen Knoten des Blockclusterbaumes korrespondieren. Je nach Anwendung kann dieses beliebig gewählt werden. Im folgenden wird dabei allerdings stets von einer vollbesetzten Darstellung oder D-*Darstellung* (von engl. *dense*) ausgegangen. Eine so repräsentierte Matrix heißt auch D-*Matrix*. Im Extremfall ist M selbst vollbesetzt falls $z \equiv$ `false` als Zulässigkeitsbedingung gewählt wird.

Neben den R- und den D-Matrizen spielen bei der Implementierung von $\mathcal{H}$-Matrizen auch jene Teilmatrizen eine Rolle, welche nicht mit Blättern des Blockclusterbaumes assoziiert sind. Für diese hat sich die Bezeichnung *Blockmatrizen* eingebürgert, da sie eine nichtleere Menge von Matrixblöcken enthalten. Allerdings ist die Generierung dieser Matrizen nur bei solchen Verfahren von Vorteil, welche explizit von der Hierarchie der $\mathcal{H}$-Matrizen Gebrauch machen.

Für den praktischen Umgang ist der Aufwand für die Speicherung von $\mathcal{H}$-Matrizen wichtig.

Lemma 2.3.4 (Speicheraufwand von $\mathcal{H}$-Matrizen) *Seien $T(I)$ und $T(J)$ Clusterbäume über den Indexmengen I und J mit einer minimalen Knotengröße von $n_{\min} > 0$. Desweiteren sei $T = T(I \times J)$ ein Blockclusterbaum über $T(I)$ und $T(J)$ mit der Zulässigkeitsbedingung z und einer Rangverteilung $k \equiv k'$, $k' \in \mathbb{N}$. Dann gilt für den Speicheraufwand $\mathcal{W}_{\mathcal{H},\mathrm{St}}$ einer $\mathcal{H}$-Matrix $M \in \mathcal{H}(T, k', z)$:*

$$\mathcal{W}_{\mathcal{H},\mathrm{St}}(M) = \mathcal{O}\left(c_{\mathrm{sp}}(T) \max\left\{k', n_{\min}\right\} p(T) \max\left\{|I|, |J|\right\}\right). \tag{2.3.2}$$

Beweis: [Siehe auch Beweis zu Lemma 5.11 in [Gra01]] Es gilt:

$$\begin{aligned}
\mathcal{W}_{\mathcal{H},\mathrm{St}}(M) &= \sum_{v \in \mathcal{L}(T)} \mathcal{W}_{\mathrm{St}}(v) = \sum_{v \in \mathcal{L}_{\mathrm{z}}(T)} \mathcal{W}_{\mathsf{R}(k),\mathrm{St}}(v) + \sum_{v \in \mathcal{L}_{\mathrm{nz}}(T)} \mathcal{W}_{\mathrm{dense},\mathrm{St}}(v) \\
&= \sum_{(\tau,\sigma) \in \mathcal{L}_{\mathrm{z}}(T)} k'(|\tau| + |\sigma|) + \sum_{(\tau,\sigma) \in \mathcal{L}_{\mathrm{nz}}(T)} |\tau||\sigma| \\
&\leq \sum_{(\tau,\sigma) \in \mathcal{L}_{\mathrm{z}}(T)} k'|\tau| + \sum_{(\tau,\sigma) \in \mathcal{L}_{\mathrm{z}}(T)} k'|\sigma| + \\
&\quad \sum_{(\tau,\sigma) \in \mathcal{L}_{\mathrm{nz}}(T)} |\tau| n_{\min} + \sum_{(\tau,\sigma) \in \mathcal{L}_{\mathrm{nz}}(T)} |\sigma| n_{\min} \\
&\leq \sum_{(\tau,\sigma) \in \mathcal{L}(T)} |\tau| \max\left\{k', n_{\min}\right\} + \sum_{(\tau,\sigma) \in \mathcal{L}(T)} |\sigma| \max\left\{k', n_{\min}\right\} \\
&= \max\left\{k', n_{\min}\right\} \left(\sum_{i=0}^{p(T)} \sum_{(\tau,\sigma) \in \mathcal{L}(T,i)} |\tau| + \sum_{(\tau,\sigma) \in \mathcal{L}(T,i)} |\sigma| \right) \\
&\leq \max\left\{k', n_{\min}\right\} \left(\sum_{i=0}^{p(T)} \sum_{\tau \in V^{(i)}(T(I))} c_{\mathrm{sp}}|\tau| + \sum_{\sigma \in V^{(i)}(T(J))} c_{\mathrm{sp}}|\sigma| \right) \\
&= c_{\mathrm{sp}} \max\left\{k', n_{\min}\right\} p(T)(|I| + |J|) \\
&= \mathcal{O}\left(c_{\mathrm{sp}} \max\left\{k', n_{\min}\right\} p(T) \max\left\{|I|, |J|\right\}\right)
\end{aligned}$$

□

Der Vergleich von (2.2.4) und (2.3.2) zeigt, dass der Aufwand für die Speicherung einer $\mathcal{H}$-Matrix zu einem wesentlichen Teil durch die Zahl der Knoten im zugrundeliegenden Blockclusterbaum bestimmt wird.

Beispiel 2.3.5 (Speicheraufwand bei binären Clusterbäumen) *Für den Blockclusterbaum T aus Beispiel 2.2.8 ergibt sich folgender Aufwand für die Speicherung einer $\mathcal{H}$-Matrix $M \in \mathcal{H}(T, k)$:*

$$\mathcal{W}_{\mathcal{H},\mathrm{St}}(M) = \mathcal{O}\left(c_{\mathrm{sp}}(T) \max\left\{k, n_{\min}\right\} n \log \frac{n}{n_{\min}}\right). \tag{2.3.3}$$

Man beachte in (2.3.3) den Einfluss von $n_{\min}$ auf den Speicheraufwand. Ein kleines $n_{\min}$ führt zu einer größeren Tiefe und erhöht damit unter Umständen indirekt die Komplexität.

Dagegen führt ein großes $n_{\min}$ direkt zu einem höheren Speicheraufwand. In der Praxis ist deshalb $n_{\min}$ applikationsabhängig anzupassen, um eine optimale Komplexität zu erzielen.

Definition 2.3.6 (Vektor- und Matrixrestriktionen) *Seien $T(I)$ und $T(J)$ über den Indexmengen I bzw. J definierte Clusterbäume. Für einen Knoten $\tau \in V(T(I))$ und Vektoren $v \in \mathbb{R}^I$ und $v' \in \mathbb{R}^\tau$ seien $\mathcal{R}_\tau$ und $\mathcal{P}_\tau$ definiert durch*

$$\mathcal{R}_\tau(v) := v|_\tau \quad \text{und} \quad \mathcal{P}_\tau(v') := \mathcal{R}_\tau^T(v').$$

Sei $T(I \times J)$ ein Blockclusterbaum über $T(I)$ und $T(J)$. Für Blockcluster $b, b' \in T(I \times J)$ mit $b' \subseteq b$ und Matrizen $M \in \mathbb{R}^b$ und $M' \in R^{b'}$ seien analog die Operatoren $\mathcal{R}_{b \to b'}$ und $\mathcal{P}_{b' \to b}$ definiert durch:

$$\mathcal{R}_{b \to b'}(M) := M|_{b'} \quad \text{und} \quad \mathcal{P}_{b' \to b}(M') := \mathcal{R}_{b \to b'}^T(M').$$

Man beachte, dass für eine $\mathcal{H}$-Matrix $M \in \mathcal{H}(T(I,J))$ und einen Block $b \subseteq I \times J$ die Matrix $\mathcal{R}_{\mathrm{root}(T(I,J)) \to b}(M)$ nicht Teil der $\mathcal{H}$-Matrix sein muss, d.h. kein entsprechender Matrixblock in M existiert. Dies ist nur der Fall, wenn $b \in T(I,J)$. Dann sind M_b und $\mathcal{R}_{\mathrm{root}(T(I,J) \to b}(M)$ identisch.

2.4 Beispiele für $\mathcal{H}$-Matrizen

In diesem Abschnitt sollen zwei Beispiele für die Definition von $\mathcal{H}$-Matrizen untersucht werden, die in den folgenden Kapiteln stets als Modellprobleme dienen. Dabei handelt es sich jeweils um ein Problem aus dem Bereich der Integralgleichungen sowie partiellen Differentialgleichungen. In beiden Fällen wird der Aufbau der Clusterbäume, die Definition der Blockclusterbäume mit entsprechender Zulässigkeitsbedingung und schlussendlich die eigentliche $\mathcal{H}$-Matrix diskutiert.

Neben den hier diskutierten Beispielen lassen sich die $\mathcal{H}$-Matrizen auch auf eine Vielzahl von weiteren Problemen (siehe z.B. [Lin02] oder [Bor03])) anwenden. Die Grundlage bildet hierbei stets eine geeignete Niedrigrangapproximation der exakten Matrix in Teilen des Blockclusterbaumes.

2.4.1 Integralgleichung

Gegenstand des in diesem Abschnitt diskutierten Beispiels ist die Fredholm-Integralgleichung

$$\lambda u + Ku = f, \tag{2.4.1}$$

über einem Hilbertraum V. Gesucht wird dabei bei vorgegebener rechter Seite f die Funktion u. Der Integraloperator K ist durch eine *Kernfunktion* κ wie folgt definiert:

$$Ku = \int_\Gamma \kappa(\cdot, y) u(y) \, \mathrm{d}y.$$

Das Integrationsgebiet wird durch eine 2-dimensionale Mannigfaltigkeit $\Gamma \subseteq \mathbb{R}^3$ bestimmt.

Zur Bestimmung von u wird das *Kollokationsverfahren* (siehe [Hac95]) genutzt. Hierbei wird die exakte Lösung durch Interpolation an disjunkten *Stützstellen* $\xi_0, \ldots, \xi_{n-1}$ angenähert, d.h. eine Funktion u_n mit

$$u_n(\xi_i) = u(\xi_i), \quad 0 \leq i < n$$

gesucht. Notwendig für die numerische Behandlung ist hierbei, V durch einen endlichdimensionalen Ansatzraum V_n mit $\dim V_n = n$ zu ersetzen. Eine Basis von V_n sei durch $\{\varphi_i\}_{i \in I}$ mit $I = \{0, \ldots, n-1\}$ gegeben.

Der Interpolationsansatz führt zu der modifizierten Aufgabe

$$\lambda u_n(\xi_i) + (K u_n)(\xi_i) = f(\xi_i), \quad i \in I,$$

bzw. mit dem Ansatz

$$u_n = \sum_{i \in I} \alpha_i \varphi_i$$

zur Bestimmung von $\{\alpha_i\}_{i \in I}$ durch das lineare Gleichungssystem

$$\lambda \sum_{j \in I} \alpha_j \varphi_j(\xi_i) + \sum_{j \in I} \alpha_j (K\varphi_j)(\xi_i) = f(\xi_i), \quad i \in I. \tag{2.4.2}$$

Durch die Matrizen

$$\begin{aligned} A = (a_{ij})_{i,j \in I} \quad &\text{mit} \quad a_{ij} = \varphi_j(\xi_i) \\ B = (b_{ij})_{i,j \in I} \quad &\text{mit} \quad b_{ij} = (K\varphi_j)(\xi_i) \end{aligned}$$

und die Vektoren $x = (\alpha_0, \ldots, \alpha_{n-1})^T$ bzw.

$$b = (b_i)_{i \in I} \quad \text{mit} \quad b_i = f(\varphi_i)$$

lässt sich (2.4.2) auch als

$$(\lambda A - B)x = b$$

darstellen.

Offen bleibt hierbei aber die Frage nach der Gestalt von V_n. Dessen Konstruktion erfolgt über eine *Triangulation* von Γ, d.h. eine Zerlegung von Γ in disjunkte Elemente t_j, so dass

$$\bigcup_j \overline{t_j} = \overline{\Omega}.$$

Für die Basis $\{\varphi_i\}$ selbst werden stückweise konstante Funktionen mit $\varphi_i(x) = 1$ für $x \in t_i$ und $\varphi_i(x) = 0$ für $x \in t_j, j \neq i$ genutzt (siehe auch Abbildung 2.4.1). Die Basisfunktionen φ_i bzw. deren Träger werden auch *Elemente* genannt. Da das Problem zusätzlich über dem

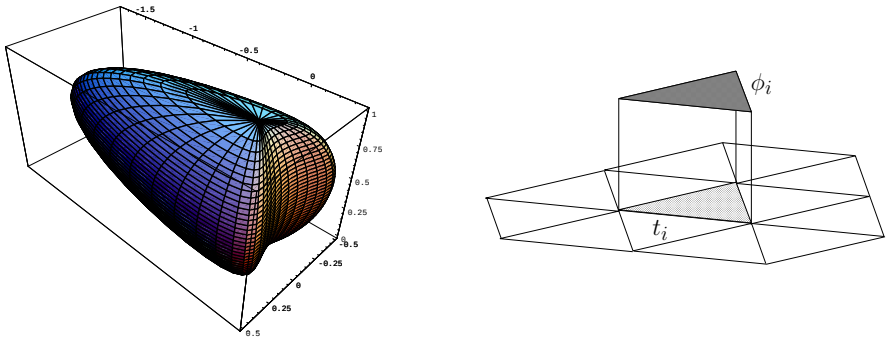


Abbildung 2.4.1: Integrationsgebiet Γ und Basisfunktionen φ_i

Rand Γ eines Gebietes definiert ist, wird dieses Verfahren auch als *Randelementmethode* bezeichnet. Die englische Bezeichnung „boundary element method“ führt zur Abkürzung BEM, welche im folgenden für dieses Beispiel genutzt wird.

Die Wahl der Stützstellen $\xi_0, \ldots, \xi_{n-1}$ fällt in diesem Beispiel auf den Schwerpunkt der Dreiecke t_i. Aufgrund der Definition der Basisfunktionen folgt deshalb $a_{ij} = \delta_{ij}$, womit A Diagonalgestalt hat. Auf B trifft dies allerdings nicht zu, da der Integraloperator typischerweise zu einer vollbesetzten Matrix führt. In diesem Beispiel wird im folgenden für den Kern κ des Integraloperators K das *Einfachschichtpotential*

$$\kappa(x, y) = \frac{1}{4\pi} \frac{1}{|x - y|} \tag{2.4.3}$$

verwendet. Das Einfachschichtpotential ist ein typischer Vertreter von Kernen, welche bis auf eine kleine Umgebung von $x = y$ gut approximiert werden können. Dies lässt sich ausnutzen, um B als $\mathcal{H}$-Matrix darzustellen.

2.4.1.1 Aufbau des Clusterbaumes

Entsprechend der Definition ist jeder Basisfunktion φ_i ein Index i zugeordnet, wobei die Position dieses Indizes durch den Stützpunkt ξ_i definiert sei. Diese Koordinaten bilden im folgenden die Grundlage für die Konstruktion des Clusterbaumes. Das Ziel soll hierbei ein balancierter Baum sein. Der dabei verwendete Algorithmus ist die *binäre Raumpartitionierung* (BSP, von engl. *binary space partitioning*).

Betrachtet wird hierfür die Menge $\Xi = \{\xi_0, \ldots, \xi_{n-1}\}$ der Koordinaten der Indizes. In jedem Schritt des BSP-Algorithmus werden zunächst die maximalen Ausdehnungen von Ξ entlang der Koordinatenachsen ermittelt. Anschließend erfolgt die Zerlegung von Ξ entlang der längsten Koordinate k in zwei disjunkte Teilmengen Ξ_0 und Ξ_1, so dass $(\xi)_k \leq (\xi')_k$ für alle $\xi \in \Xi_0$ und $\xi' \in \Xi_1$ gilt. Die Zerlegung wird nun mit den beiden Mengen Ξ_0 und Ξ_1 rekursiv fortgesetzt und stoppt, falls $\min\{|\Xi_0|, |\Xi_1|\} < n_{\min}$ erreicht ist. Abbildung 2.4.2 zeigt diese Konstruktion an einem Beispiel.

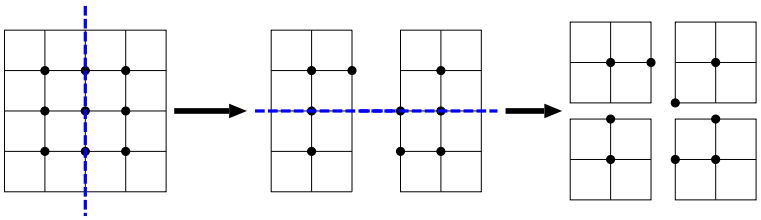


Abbildung 2.4.2: Beispiel für eine binäre Raumpartitionierung

Wird bei der Zerlegung in Ξ_0 und Ξ_1 die zusätzliche Bedingung $||\Xi_0| - |\Xi_1|| \leq 1$ eingeführt, so resultiert ein *kardinalitätsbalancierter* Clusterbaum. Für die Implementierung des BSP-Algorithmus ist es hierbei von Vorteil, wenn Ξ entsprechend der Koordinate k sortiert vorliegt. In diesem Fall erfolgt die Zerlegung in Ξ_0 und Ξ_1 durch einfaches Halbieren der betrachteten Sequenz.

Da jeder Koordinate ξ_i ein Index i zugewiesen ist, entspricht die räumliche Aufteilung von Ω durch den BSP-Algorithmus der hierarchischen Zerlegung der Indexmenge I und somit der Generierung des Clusterbaumes. Die Mengen Ξ_0 und Ξ_1 bilden hierbei die Söhne τ_0, τ_1 des Clusters τ, welcher durch Ξ definiert ist.

Bemerkung 2.4.1 *Der Aufwand für die Berechnung eines kardinalitätsbalancierten Clusterbaumes $T(I)$ durch den BSP-Algorithmus beträgt $\mathcal{O}(n \log n)$.*

Beweis: Die Sortierung von Ξ ist in $\mathcal{O}(n \log n)$ möglich, z.B. durch *Mergesort* (siehe [Sed90]). Es genügt dabei, die Sortierung einmal am Anfang des Algorithmus durchzuführen. Wegen der Balancierung bei der Zerlegung von Ξ in Ξ_0 und Ξ_1 ist die Rekursionstiefe von der Ordnung $\log n$. □

2.4.1.2 Blockclusterbaum und $\mathcal{H}$-Matrix

Notwendig für die Definition des Blockclusterbaumes ist die Angabe einer Zulässigkeitsbedingung z (siehe Abschnitt 2.2). Hierbei werden die Träger $\operatorname{supp} \varphi_i$ der einzelnen Basisfunktionen betrachtet und auf Cluster erweitert: $\chi(\tau) = \cup_{i \in \tau} \operatorname{supp} \varphi_i$.

Das Ziel der Zulässigkeitsbedingung ist es, Blockcluster (τ, σ) zu identifizieren, in denen die einzelnen Cluster τ und σ in Abhängigkeit von ihrer Größe hinreichend weit voneinander getrennt sind. Ein Grund hierfür liegt z.B. in der Möglichkeit, die Kernfunktion κ mittels Taylorentwicklung anzunähern und hierdurch eine Niedrigrangapproximation zu gewinnen. Hierfür ist es notwendig, die einzelnen Indizes in τ bzw. σ räumlich voneinander zu trennen, um hinreichend weit von der Singularität in κ entfernt zu sein. Für die Zulässigkeitsbedingung ergibt sich somit folgende Definition:

$$z(\tau, \sigma) = \texttt{true} \quad \Longleftrightarrow \quad \min\{\operatorname{diam}(\chi(\tau)), \operatorname{diam}(\chi(\sigma))\} \leq \eta \operatorname{dist}(\chi(\tau), \chi(\sigma)), \tag{2.4.4}$$

mit $\eta \geq 0$. Durch die Wahl von η lässt sich der Abstand der einzelnen Cluster und damit die Schwachbesetztheit des Blockclusterbaumes steuern, wobei kleinere Werte zu einem größeren c_{sp} führen. In Abbildung 2.4.3 sind hierfür Beispiele mit unterschiedlichem η angegeben.

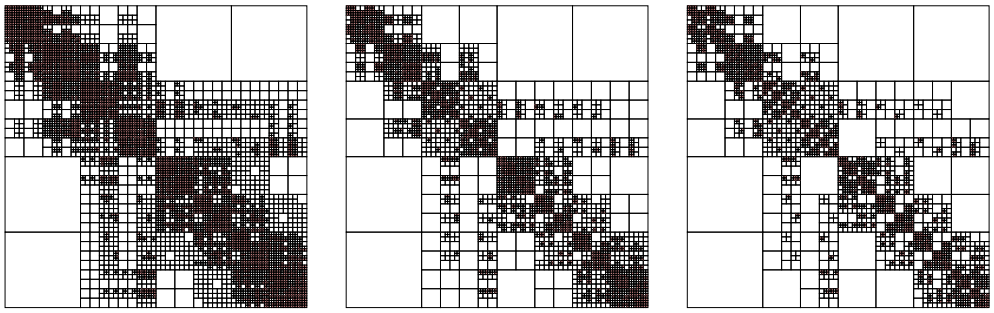


Abbildung 2.4.3: Blockclusterbaum für das BEM-Beispiel mit $\eta = 0.5$, $\eta = 1$ und $\eta = 2$

Für die eigentliche Konstruktion des Blockclusterbaumes wird Algorithmus 2.2.1 verwendet.

Um eine Niedrigrangrepräsentation in allen zulässigen Matrixblöcken zu generieren, stehen verschiedene Verfahren zur Verfügung, z.B. die *schnelle Multipol-Methode* (siehe [Rok85] oder [GR97]) oder das *Panel-Clustering* (siehe [HN89]). Bei diesen Verfahren erfordert der Matrixaufbau allerdings eine Modifikation der Kerns, etwa über die bereits angesprochene Taylorentwicklung.

Im Gegensatz hierzu kann bei der Verwendung der *adaptiven Kreuz-Approximation* (ACA, von engl: „adaptive cross approximation“) (siehe [Beb00]) die Ausgangsgleichung für die Darstellung der Matrixelemente beibehalten werden. Eine wichtige Voraussetzung für die Anwendbarkeit des ACA-Verfahrens ist die *asymptotische Glattheit* (siehe [Bra91]) der Kerns, welche im Fall von (2.4.3) erfüllt ist.

Die Grundidee des ACA-Algorithmus besteht darin, in jedem Iterationsschritt in der Matrix $M \in \mathbb{R}^{\tau \times \sigma}$ mit $(\tau, \sigma) \in \mathcal{L}_{\mathrm{z}}(T)$ eine geeignete Zeile u_i und eine Spalte v_i zu bestimmen und anschließend aus M zu entfernen: $M_{i+1} = M_i - u_i v_i^T$. Hierdurch ist es nicht notwendig, alle $|\tau| \cdot |\sigma|$ Koeffizienten von M zu berechnen. Weiterhin gestattet es der Algorithmus, neben einer Approximation zu einem vorgegebenen Rang, auch eine Näherung zu einer definierten Genauigkeit zu konstruieren. Der Aufwand für den ACA-Algorithmus ist in beiden Fällen durch $\mathcal{O}\left(k^2(|\tau| + |\sigma|)\right)$ bestimmt, wobei k der Rang der Approximation an M ist. Zu erwähnen ist hierbei, dass die Zahl der Kernauswertungen durch $k(|\tau| + |\sigma|)$ gegeben ist. Bei entsprechend teuren Funktionsaufrufen entspricht deshalb letzteres auch der beobachteten Komplexität.

In Abbildung 2.4.4 sind zu den Blockclusterbäumen aus Abbildung 2.4.3 $\mathcal{H}$-Matrizen dargestellt, die mit einer festen Genauigkeit von $\|M - M_k\| \leq 10^{-4}$ berechnet wurden. In den zulässigen Blöcken wurden hierbei die Singulärwerte auf einer logarithmischen Skala zusam-

men mit dem jeweils auftretenden Rang der R-Matrizen dargestellt. Deutlich sichtbar ist der exponentielle Abfall der einzelnen Werte.

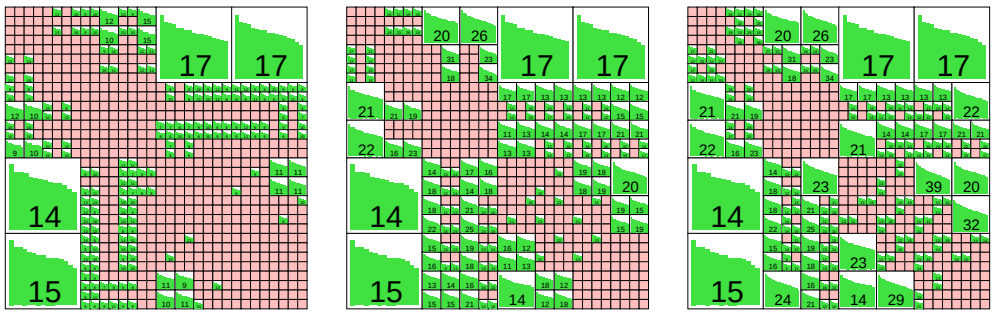


Abbildung 2.4.4: Beispiele für $\mathcal{H}$-Matrizen mit fester Genauigkeit

Die durch den ACA-Algorithmus berechnete R-Matrix ist typischerweise nicht optimal bezüglich der Datenmenge bei einer vorgegebenen Genauigkeit. Hier bietet sich ein nachträglicher Kürzungsschritt, wie er in Abschnitt 3.2.1 beschrieben wird, an, um den Rang bei konstanter Genauigkeit zu reduzieren.

2.4.2 Partielle Differentialgleichung

Betrachtet wird in diesem Anwendungsbeispiel die Poissongleichung

$$-\Delta u = f \quad \text{in} \quad \Omega =]0,1[^2 \tag{2.4.5}$$

mit der Dirichletrandbedingung

$$u = 0 \quad \text{auf} \quad \Gamma = \partial\Omega.$$

Sei V der Raum der einmal schwach differenzierbaren Funktionen mit Nullrandbedingung auf Γ. Multipliziert man (2.4.5) mit einer Testfunktion $v \in V$ und integriert anschließend über Ω, so erhält man

$$-\int_\Omega v\Delta u\,dx = \int_\Omega fv\,dx$$

Mit Hilfe der Greenschen Formel lässt sich das linke Integral umformulieren und als Bilinearform darstellen:

$$a(u,v) = \int_\Omega \nabla v \cdot \nabla u\,dx.$$

Wählt man für das rechte Integral die Darstellung

$$f(v) = \int_\Omega fv\,dx,$$

so lautet die zu lösende Aufgabe:

Finde $u \in V$ so, dass $a(u, w) = f(w)$ für alle $w \in V$ gilt.

Analog zum BEM-Beispiel wird der Raum V durch einen endlichdimensionalen Raum $V_n \subset V$ mit $\dim V_n = n$ ersetzt. Damit ergibt sich die modifizierte Aufgabe, ein $u_n \in V_n$ mit $a(u_n, w_n) = f(w_n)$ für alle $w_n \in V_n$ zu suchen.

Sei $I = \{0, \ldots, n-1\}$ die diesem Ansatz zugrundeliegende Indexmenge und $\{\varphi_i\}_{i \in I}$ eine Basis von V_n, etwa stetige, stückweise lineare Funktionen mit $\varphi_i(v_j) = \delta_{ij}$, wie sie in Abbildung 2.4.5 (rechts) dargestellt sind. Durch den Ansatz

$$u_h = \sum_{i=0}^{n-1} \alpha_i \varphi_i \tag{2.4.6}$$

lässt sich die Variationsaufgabe in ein lineares Gleichungssystem $Ax = b$ mit $x = (\alpha_0, \ldots, \alpha_{n-1})^T$ überführen, wobei die Matrix A und die rechte Seite b wie folgt definiert sind:

$$\begin{aligned} A = (a_{ij})_{i,j \in I} \quad &\text{mit} \quad a_{ij} = a(\varphi_i, \varphi_j) \\ &\text{und} \\ b = (b_i)_{i \in I} \quad &\text{mit} \quad b_i = f(\varphi_i). \end{aligned} \tag{2.4.7}$$

Auch bei dieser Lösungsmethode wird für die Basisfunktionen die Bezeichnung *Elemente* genutzt. Anders als im BEM-Beispiel zeichnet allerdings nicht die Form des Gebietes für die Namensgebung des Verfahrens verantwortlich, sondern die endliche Trägermenge. Hieraus ergibt sich die Bezeichnung *Finite-Elemente-Methode* oder kurz *FEM*.

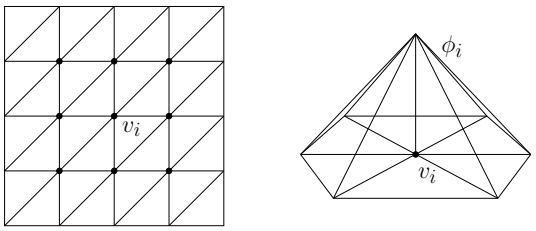


Abbildung 2.4.5: Triangulation mit Dreiecken und Basisfunktion

Die Konstruktion von V_n erfolgt wieder über eine Triangulation von Ω. Ein Beispiel für eine solche Zerlegung zeigt Abbildung 2.4.5 (links). Von dieser Triangulation wird auch im folgenden ausgegangen, wobei die Länge der Katheten durch $h > 0$ bestimmt ist. Für die Gesamtzahl der Basisfunktionen folgt somit

$$n = \left(\frac{1}{h} - 1\right)^2.$$

Auch in diesem Beispiel kommt das BSP-Verfahren zur Generierung des Clusterbaumes zum Einsatz. Die hierfür benötigten Koordinaten der Indizes sind durch die Basisfunktion

definiert und lauten:

$$x_{i+\sqrt{n}j} = \begin{pmatrix} ih \\ jh \end{pmatrix},\ 0 \leq i, j < \sqrt{n}.$$

Die Wahl der Zulässigkeitsbedingung für die Konstruktion des Blockclusterbaumes wird durch zwei Eigenschaften des FEM-Beispiels bestimmt. Zum einen sind die Träger der Basisfunktionen lokal, d.h. bei einem hinreichend großen Abstand zweier Cluster τ und σ, gilt

$$(a_{ij})_{i\in\tau, j\in\sigma} = 0.$$

Ein entsprechender Matrixblock besitzt damit Rang 0. Desweiteren lässt sich die Inverse des entsprechenden Differentialoperators als Integralgleichungsoperator formulieren (siehe auch [BH03]). Da ein Anwendungsgebiet von $\mathcal{H}$-Matrizen in der Invertierung der Systemmatrix liegt, ist somit ein Blockclusterbaum notwendig, welcher über eine Zulässigkeitsbedingung für die assoziierte Integralgleichung definiert ist. Dementsprechend wird auch in diesem Fall die Funktion (2.4.4) angewandt. In Abbildung 2.4.6 sind Beispiele für hierdurch definierte Blockclusterbäume mit wachsendem η angegeben.

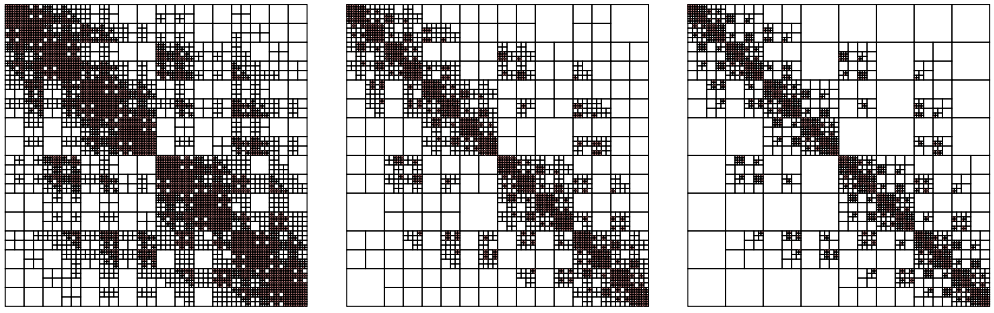


Abbildung 2.4.6: Beispiele für Blockclusterbäume mit $\eta = 0.5$, $\eta = 1$ und $\eta = 2$

Die eigentliche Berechnung der $\mathcal{H}$-Matrix erfolgt über (2.4.7). Wie erwähnt, besitzen alle R-Matrizen durch die verwendete Zulässigkeitsbedingung den Rang 0, weshalb der Berechnungsaufwand auf die nicht-zulässigen Blöcke beschränkt ist.

Die Darstellung von A im $\mathcal{H}$-Matrixformat liefert aufgrund der Schwachbesetztheit der Systemmatrix zunächst keinerlei Vorteile im Vergleich zu einem typischen Speicherformat für dünnbesetzte Matrizen. Allerdings ermöglicht die $\mathcal{H}$-Matrix-Arithmetik durch die Inversion einen natürlichen Zugang zur Lösung des entsprechenden Problems (2.4.5).

3 Sequentielle $\mathcal{H}$-Arithmetik

Gegenstand dieses Kapitels bilden die grundlegenden Algorithmen für die $\mathcal{H}$-Matrix-Arithmetik. Dazu gehören neben der Multiplikation einer $\mathcal{H}$-Matrix mit einem Vektor die Addition, Multiplikation sowie Inversion von $\mathcal{H}$-Matrizen. Desweiteren wird auf die LU-Zerlegung eingegangen. Interessant sind diese Verfahren insbesondere deshalb, da viele von ihnen die Basis für die parallelen Algorithmen in Kapitel 6 bilden.

Betrachtet werden hierbei stets Indexmengen I und J mit $|I| = n$ und $|J| = m$ und die korrespondierenden Clusterbäume $T(I)$ bzw. $T(J)$. Der darauf aufbauende Blockclusterbaum sei $T = T(I \times J)$.

3.1 Matrix-Vektor-Multiplikation

Anstelle der einfachen Matrix-Vektor-Multiplikation Ax mit der $\mathcal{H}$-Matrix $A \in \mathcal{H}(T)$ und dem Vektor $x \in \mathbb{K}^J$ wird im folgenden die erweiterte Operation

$$y := \beta y + \alpha A x \tag{3.1.1}$$

mit $y \in \mathbb{K}^I$ und $\alpha, \beta \in \mathbb{K}$ betrachtet. Diese Aktualisierung eines gegebenen Vektors y mit einem Produkt tritt in den üblichen numerischen Algorithmen sehr häufig auf und beinhaltet die Version $y = Ax$ als Spezialfall.

Bei der Multiplikation einer $\mathcal{H}$-Matrix mit einem Vektor ist es nicht erforderlich, die gesamte Hierarchie, die der Matrix zugrundeliegt, zu berücksichtigen. Anstelle dieser genügt es, die zu den Blättern des Blockclusterbaumes korrespondierenden Matrixblöcke zu betrachten. Hierbei wird für jeden Matrixblock $M = M(\tau, \sigma)$ eine Matrix-Vektor-Multiplikation mit dem restringierten Vektor $\mathcal{R}_\sigma(x)$ durchgeführt. Das Produkt wird anschließend auf y addiert: $y := y + \mathcal{P}_\tau(M\mathcal{R}_\sigma(x))$.

Damit lässt sich für (3.1.1) der Multiplikationsalgorithmus wie folgt formulieren:

```
procedure mv( α, A, x, β, y )
    y := βy;
    for all (τ, σ) ∈ L(T) do
        y := y + P_τ(αA(τ, σ)R_σ(x));
end;
```

Algorithmus 3.1.1: Matrix-Vektor-Multiplikation mit einer $\mathcal{H}$-Matrix

Man beachte, dass die Multiplikation mit α in jedem Matrixblock erfolgt. Weniger Operationen sind notwendig, wenn man die Berechnung des Produktes Ax von der Aktualisierung des

Vektors y trennt. Allerdings zeigen sich in der Praxis kaum Unterschiede zwischen beiden Varianten. Das angegebene Verfahren hat den Vorteil, dass das lokale Teilergebnis direkt addiert wird und damit kein zusätzlicher Speicherplatz benötigt wird.

3.1.1 Komplexität der Matrix-Vektor-Multiplikation

Zunächst gilt es, die Komplexität der einzelnen Multiplikationen mit Matrixblöcken in A zu beschreiben. Diese hängt ab von der Darstellung der Matrizen.

Bemerkung 3.1.1 *Für eine $\mathsf{R}(k)$-Matrix $R \in \mathbb{K}^{\tau\times\sigma}$ beträgt der Aufwand für die Matrix-Vektor-Multiplikation:*

$$\mathcal{W}_{\mathrm{MV},\mathsf{R}}(R) = \mathcal{W}_{\mathrm{MV},\mathsf{R}}(k,|\tau|,|\sigma|) = \mathcal{O}\left(k(|\tau|+|\sigma|)\right). \tag{3.1.2}$$

Der Aufwand für die Multiplikation mit einer vollbesetzten Matrix $D \in \mathbb{K}^{\tau\times\sigma}$ lautet:

$$\mathcal{W}_{\mathrm{MV},\mathsf{D}}(D) = \mathcal{W}_{\mathrm{MV},\mathsf{D}}(|\tau|,|\sigma|) = \mathcal{O}\left(|\tau|\cdot|\sigma|\right). \tag{3.1.3}$$

Weiterhin beobachtet man, dass für die Matrix-Vektor-Multiplikation von A jeder Koeffizient in den Matrixblöcken genau einmal verwendet wird. Somit ist die bei der Multiplikation aufgewendete Arbeit mindestens von der gleichen Größenordnung wie der Speicheraufwand für A.

Lemma 3.1.2 *Unter den Bedingungen von Lemma 2.3.4 beträgt der Aufwand für die Multiplikation einer $\mathcal{H}$-Matrix $M \in \mathcal{H}(T,k,z)$*

$$\mathcal{W}_{\mathcal{H},\mathrm{MV}}(M) = \mathcal{O}\left(c_{\mathrm{sp}}(T)\max\left\{k',n_{\min}\right\}p(T)\max\left\{|I|,|J|\right\}\right). \tag{3.1.4}$$

Beweis: [Siehe auch Beweis zu Lemma 5.13 in [Gra01]]

$$\begin{aligned}
\mathcal{W}_{\mathcal{H},\mathrm{MV}}(M) &= \sum_{(\tau,\sigma)\in\mathcal{L}(T)} \mathcal{W}_{\mathcal{H},\mathrm{MV}}(M(\tau,\sigma)) \\
&= \sum_{(\tau,\sigma)\in\mathcal{L}_{\mathrm{z}}(T)} \mathcal{W}_{\mathrm{MV},\mathsf{R}}(|\tau|,|\sigma|) + \sum_{(\tau,\sigma)\in\mathcal{L}_{\mathrm{nz}}(T)} \mathcal{W}_{\mathrm{MV},\mathsf{D}}(|\tau|,|\sigma|) \\
&= \mathcal{O}\left(\sum_{(\tau,\sigma)\in\mathcal{L}_{\mathrm{z}}(T)} k(|\tau|+|\sigma|) + \sum_{(\tau,\sigma)\in\mathcal{L}_{\mathrm{nz}}(T)} |\tau||\sigma|\right) \\
&= \mathcal{W}_{\mathcal{H},\mathrm{St}}(M).
\end{aligned}$$

□

Nicht eingerechnet wurde bei dieser Abschätzung der zusätzliche Aufwand, der durch die spezielle Repräsentation der Daten in einem Programm hervorgerufen wird. Tritt hierbei ein konstanter Aufwand pro Matrixblock auf, so sind die zusätzlichen Kosten durch die Anzahl der Blätter in T bestimmt und somit nach (2.2.4) von geringerer Größenordnung als $\mathcal{W}_{\mathcal{H},\mathrm{MV}}$.

Bemerkung 3.1.3 (Aufwand bei binären Clusterbäumen) *Für den Blockclusterbaum T aus Beispiel 2.2.8 folgt mit* (2.3.3) *und* (3.1.4) *für die Komplexität der Matrix-Vektor-Multiplikation mit einer $\mathcal{H}$-Matrix $M \in \mathcal{H}(T,k)$:*

$$\mathcal{W}_{\mathcal{H},\mathrm{MV}}(M) = \mathcal{O}\left(c_{\mathrm{sp}}(T)\max\{k,n_{\min}\}\, n \log \frac{n}{n_{\min}}\right). \tag{3.1.5}$$

Analog zum Speicheraufwand einer $\mathcal{H}$-Matrix gilt es auch bei der Matrix-Vektor-Multiplikation $n_{\min}$ in praktischen Anwendungen so zu wählen, dass der Aufwand möglichst gering wird.

3.2 Matrix-Addition

Betrachtet wird die Addition zweier $\mathcal{H}$-Matrizen $A, B \in \mathcal{H}(T)$. Analog zur Matrix-Vektor-Multiplikation ist auch in diesem Fall die Kenntnis der Blattmenge hinreichend für die Definition bzw. Implementierung der Addition. Eine notwendige Voraussetzung für diese Herangehensweise ist allerdings, dass A und B über identischen Blockclusterbäumen bzw. Zulässigkeitsbedingungen definiert sind.

Um eine größere Flexibilität bei der Anwendung der Addition in anderen Algorithmen zu erhalten, wird auch an dieser Stelle eine erweiterte Operation

$$A := \alpha A + \beta B, \tag{3.2.1}$$

mit $\alpha, \beta \in \mathbb{K}$ betrachtet. Der Additionsalgorithmus lautet hierfür:

```
procedure add( α, A, β, B )
    for all (τ, σ) ∈ L(T) do
        A(τ, σ) := αA(τ, σ) + βB(τ, σ);
end;
```

Algorithmus 3.2.1: Addition zweier $\mathcal{H}$-Matrizen

Bei der Addition zweier $\mathsf{R}(k)$-Matrizen ist allerdings zu beachten, dass das Ergebnis den Rang $2k$ besitzt (siehe Bemerkung 2.3.2). Somit folgt für die Summe $C = \alpha A + \beta B$: $C \in \mathcal{H}(2k)$. Die Komplexität der beschriebenen Addition lässt sich in diesem Fall wie folgt abschätzen.

Lemma 3.2.1 (Aufwand der exakten Addition) *Die exakte Addition* (3.2.1) *zweier $\mathcal{H}$-Matrizen $A, B \in \mathcal{H}(T,k)$ hat einen Aufwand von:*

$$\mathcal{W}_{\mathcal{H},\mathrm{MA}}(A) = \mathcal{O}\left(c_{\mathrm{sp}}(T)\max\{k,n_{\min}\}\, p(T)\max\{|I|,|J|\}\right). \tag{3.2.2}$$

Beweis:

$$\begin{aligned}\mathcal{W}_{\mathcal{H},\mathrm{MA}}(A) &= \mathcal{O}\left(\sum_{(\tau,\sigma)\in\mathcal{L}_{\mathrm{z}}(T)} 2k(|\tau|+|\sigma|) + \sum_{(\tau,\sigma)\in\mathcal{L}_{\mathrm{nz}}(T)} |\tau|\cdot|\sigma|\right)\\ &= \mathcal{W}_{\mathcal{H},\mathrm{St}}(A).\end{aligned}$$

□

Soll dagegen die Summe ebenfalls Rang k besitzen, ist ein nachfolgender *Kürzungsschritt* für die $\mathcal{H}$-Matrix erforderlich. Hierbei lässt sich ein effizientes Verfahren für die Singulärwertzerlegung von R-Matrizen ausnutzen, welches im folgenden Abschnitt beschrieben wird.

3.2.1 Singulärwertzerlegung von Rang-k-Matrizen

Die Singulärwertzerlegung (SVD, von engl. *singular value decomposition*) einer allgemeinen Matrix $M \in \mathbb{K}^{n\times m}$ ist definiert durch eine Zerlegung:

$$M = USV^T,$$

mit unitären $U \in \mathbb{K}^{n,n}$, $V \in \mathbb{K}^{m,m}$ und $S = \operatorname{diag}(s_0, s_1, \ldots, s_{l-1}, 0, \ldots, 0) \in \mathbb{K}^{n\times m}$, $l \leq \min\{n, m\}$. Die Zahlen $s_0 > s_1 > \cdots > s_{l-1}$ heißen auch *Singulärwerte* von M. Man beachte, dass M eine $\mathsf{R}(l)$-Matrix bildet:

$$M = \sum_{i=0}^{l-1} u_i (s_i v_i^T),$$

wobei u_i und v_i die i-ten Spaltenvektoren von U bzw. V darstellen.

Betrachtet man statt aller l nur die ersten $k < l$ Singulärwerte, so gelangt man zu folgendem Ergebnis.

Lemma 3.2.2 (Bestapproximation durch SVD) *Sei $k < l$ und $M' = \sum_{i=0}^{k-1} u_i s_i v_i^T$. Dann gilt:*

$$\min_{\operatorname{rang}(N)\leq k} \|M - N\|_2 \quad = \quad \|M - M'\|_2 = s_k$$

und

$$\min_{\operatorname{rang}(N)\leq k} \|M - N\|_{\mathrm{F}} \quad = \quad \|M - M'\|_{\mathrm{F}} = \sqrt{\sum_{i=k}^{l-1} s_i^2}.$$

Die Matrix M' stellt somit bezüglich der Spektral- und Frobeniusnorm eine Bestapproximation *an M in der Menge aller Rang-k-Matrizen dar.*

Beweis: Siehe [GL96, Theorem 2.5.3] und [Gra01, Satz 2.10]. □

Durch diese Aussage bietet sich eine Möglichkeit, den Rang bei der Addition von $2k$ auf k zu reduzieren und dabei die bestmögliche Approximation zu gewinnen. Notwendig ist hierfür die Berechnung der Singulärwertzerlegung einer $\mathsf{R}(k)$-Matrix $R \in \mathbb{K}^{n\times m}$. Offen bleibt allerdings zunächst, wie dies effizient berechnet werden kann, da der Algorithmus zur Bestimmung einer SVD einer vollbesetzten Matrix einen Aufwand von $\mathcal{O}\left(nm^2\right)$ besitzt (siehe [GL96, Abschnitt 5.4.5]).

Mit den (gekürzten) QR-Zerlegungen $Q_A R_A = A$ und $Q_B R_B = B$, mit $Q_A \in \mathbb{K}^{n\times k}, Q_B \in \mathbb{K}^{m\times k}$ von A bzw. B und $R_A, R_B \in \mathbb{K}^{k\times k}$ lässt sich die Matrix R umformulieren zu

$$\begin{aligned} R &= AB^T \\ &= Q_A R_A (Q_B R_B)^T \\ &= Q_A R_A R_B^T Q_B^T. \end{aligned}$$

Man beachte, dass das Produkt $T = R_A R_B^T$ eine $k\times k$-Matrix ist, deren Singulärwertzerlegung $T = U_T S_T V_T^T$ mit $\mathcal{O}\left(k^3\right)$ Operationen berechnet werden kann. Die obige Gleichungskette lässt sich anschließend fortsetzen zu

$$\begin{aligned} R &= Q_A R_A R_B^T Q_B^T \\ &= Q_A U_T S_T V_T^T Q_B^T \\ &= (Q_A U_T) S_T (Q_B V_T)^T, \end{aligned}$$

wodurch eine SVD von R definiert ist. Der folgende Algorithmus nutzt diese Darstellung für die Approximation einer gegebenen $\mathsf{R}(k)$-Matrix durch eine $\mathsf{R}(k')$-Matrix, wobei $k' \leq k$ ist.

procedure truncate($R = AB^T, k, k'$)
  { *Berechnung der (gekürzten) QR-Zerlegung von A und B* }
  $A = Q_A R_A$; $B = Q_B R_B$;
  $T := R_A R_B^T$;
  { *Berechnung der SVD von T* }
  $T = USV^T$;
  for $i = 0 \ldots k' - 1$ **do**
    $a'_i = (Q_A US)e_i$;
    $b'_i = (Q_B V)e_i$;
  endfor;
end;

Algorithmus 3.2.2: Kürzen einer $\mathsf{R}(k)$-Matrix auf einen festen Rang k'

Bemerkung 3.2.3 *Der Aufwand für die QR-Zerlegung von A bzw. B beträgt* $\mathcal{O}\left(k^2 n\right)$ *bzw.* $\mathcal{O}\left(k^2 m\right)$ *(siehe [GL96, Abschnitt 5.2.9]). Insgesamt folgt für die Komplexität von Algorithmus 3.2.2 somit:*

$$\mathcal{O}\left(k^2(n+m) + k^3\right). \tag{3.2.3}$$

Lemma 3.2.2 erlaubt auch einen alternativen Zugang zur Bestimmung einer Approximation an R mit einer festen Genauigkeit $\varepsilon > 0$. Hierbei finden nur die Singulärwerte mit $s_i \geq \varepsilon$ Verwendung:

$$M'' = \sum_{i=0, s_i \geq \varepsilon}^{k-1} u_i s_i v_i^T.$$

Für M'' gilt somit nach Lemma 3.2.2: $\|M - M''\|_2 \leq \varepsilon$. Algorithmus 3.2.2 ist in diesem Fall entsprechend abzuändern:

```
procedure truncate_eps( R = AB^T, k, ε )
    { Berechnung der (gekürzten) QR-Zerlegung von A und B }
    A = Q_A R_A; B = Q_B R_B;
    T := R_A R_B^T;
    { Berechnung der SVD von T }
    T = U S V^T;
    i := 0;
    while s_i ≥ ε do
        a'_i = (Q_A U S) e_i;
        b'_i = (Q_B V) e_i;
        i := i + 1;
    endwhile;
end;
```

Algorithmus 3.2.3: Kürzen einer $\mathsf{R}(k)$-Matrix auf eine Genauigkeit ε

Um die Berechnung einer gekürzten Summe während der Addition durchzuführen, ist der Kürzungsschritt in Algorithmus 3.2.1 einzuführen.

Definition 3.2.4 *Zu einem Blockcluster $(\tau, \sigma) \in T$ sei $M \in \mathbb{K}^{\tau\times\sigma}$. Weiterhin sei $M' \in \mathbb{K}^{\tau\times\sigma}$ eine Bestapproximation an M mit Rang k. Dann sei $\mathcal{T}_k : \mathbb{R}^{\tau\times\sigma} \to \mathcal{R}(k, \tau, \sigma)$ definiert durch $\mathcal{T}_k(M) = M'$.*

Reduziert man nach jeder Addition in Algorithmus 3.2.1 den Rang des Ergebnisses, entsteht ein modifiziertes Verfahren, wobei die resultierende $\mathcal{H}$-Matrix A blockweise den Rang k besitzt:

```
procedure add_trunc( α, A, β, B )
    for all (τ, σ) ∈ L(T) do
        A(τ, σ) := T_k(α A(τ, σ) + β B(τ, σ));
end;
```

Algorithmus 3.2.4: Addition zweier $\mathcal{H}$-Matrizen mit Kürzen

Mit (3.2.2) und (3.2.3) folgt für die Komplexität von Algorithmus 3.2.4:

Lemma 3.2.5 (Aufwand der Addition mit Kürzen) *Die Addition zweier $\mathcal{H}$-Matrizen $A, B \in \mathcal{H}(T, k)$ mit Kürzen hat einen Aufwand von:*

$$\mathcal{W}_{\mathcal{H},\mathrm{MA}}(A) = \mathcal{O}\left(c_{\mathrm{sp}}(T)(\max\{k, n_{\min}\} + k^3)p(T)\max\{|I|, |J|\}\right). \tag{3.2.4}$$

Beweis:

$$\begin{aligned}
\mathcal{W}_{\mathcal{H},\mathrm{MA}}(A) &= \mathcal{O}\left(\sum_{(\tau,\sigma)\in\mathcal{L}_{\mathrm{z}}(T)} 2k(|\tau| + |\sigma|) + k^2(|\tau| + |\sigma|) + k^3 + \sum_{(\tau,\sigma)\in\mathcal{L}_{\mathrm{nz}}(T)} |\tau| \cdot |\sigma|\right) \\
&\leq \mathcal{O}\left(\sum_{(\tau,\sigma)\in\mathcal{L}_{\mathrm{z}}(T)} k^2(|\tau| + |\sigma|) + \sum_{(\tau,\sigma)\in\mathcal{L}_{\mathrm{nz}}(T)} k|\tau| \cdot |\sigma| + \sum_{(\tau,\sigma)\in\mathcal{L}_{\mathrm{z}}(T)} k^3\right) \\
&\leq k\mathcal{W}_{\mathcal{H},\mathrm{St}}(A) + \mathcal{O}\left(c_{\mathrm{sp}}k^3p(T)\max\{|I|, |J|\}\right).
\end{aligned}$$

□

Soweit nicht anders beschrieben, wird im folgenden stets von einer $\mathcal{H}$-Arithmetik mit Kürzungsschritten ausgegangen. Dies ist insbesondere notwendig, um einen Speicheraufwand der $\mathcal{H}$-Matrizen zu bewahren, der linear-logarithmisch von der Größe der Indexmengen abhängt. Letzteres kann allerdings nur bei Kürzungen auf einen konstanten Rang, nicht aber auf eine konstante Genauigkeit garantiert werden. Praktische Experimente zeigen aber auch im letzteren Fall die gewünschte Komplexität der $\mathcal{H}$-Arithmetik.

Bemerkung 3.2.6 (Additionsaufwand für binäre Clusterbäume) *Sei T der Blockclusterbaum aus Beispiel 2.2.8 und $M \in \mathcal{H}(T,k)$. Dann folgt mit* (3.2.4)*:*

$$\mathcal{W}_{\mathcal{H},\mathrm{MA}}(M) = \mathcal{O}\left(c_{\mathrm{sp}}(T)(\max\{k, n_{\min}\} + k^3) n \log n\right). \tag{3.2.5}$$

3.3 Matrix-Multiplikation

Neben den bisher betrachteten Indexmengen I und J sei K eine weitere Indexmenge und $T(K)$ ein Clusterbaum über K. Weiterhin seien $T(I \times K)$ und $T(K \times J)$ Blockclusterbäume. In diesem Abschnitt soll die Multiplikation zweier $\mathcal{H}$-Matrizen $A \in \mathcal{H}(T(I \times K))$ und $B \in \mathcal{H}(T(K \times J))$ untersucht werden. Betrachtet wird dabei die Operation

$$C := \alpha AB + \beta C, \tag{3.3.1}$$

mit der $\mathcal{H}$-Matrix $C \in \mathcal{H}(T(I \times J))$ und $\alpha, \beta \in \mathbb{K}$. Bei einer exakten Durchführung der Multiplikation ergeben sich allerdings im allgemeinen viel zu hohe Kosten. Aus diesem Grund soll das Ergebnis der Operation ebenfalls in $\mathcal{H}(T(I \times J), k)$ vorliegen, d.h. dass während der Multiplikation Kürzungsschritte erfolgen, um den Rang der R-Matrizen in C konstant zu halten. Das Resultat approximiert mithin das exakte Produkt.

Aufgrund der Struktur der beteiligten Matrizen bildet der Algorithmus zur Multiplikation von Blockmatrizen die Grundlage für das hier vorgestellte Verfahren. Durch den hierarchischen Charakter der $\mathcal{H}$-Matrizen entsteht hierbei eine Rekursion, da die Teilmatrizen von A, B und C ebenfalls $\mathcal{H}$-Matrizen darstellen.

Ist eine der bei der Multiplikation auftretenden Matrizen keine Blockmatrix und korrespondiert somit mit einem Blatt im Blockclusterbaum, so endet die Rekursion zunächst. Dieser Fall tritt auf, da die Pfade in den beteiligten Blockclusterbäumen $T(I \times J), T(I \times K)$ und $T(K \times J)$ nicht notwendigerweise die gleiche Länge besitzen. Aus diesem Grund werden Blätter in den Bäumen bei unterschiedlichen Rekursionstiefen erreicht.

Um bei der Multiplikation eine optimale Komplexität zu erzielen, ist es notwendig, für die möglichen Kombinationen der Formate von A, B oder C spezielle Algorithmen zu verwenden. Diese Spezialroutinen sollen in den nächsten Abschnitten vorgestellt werden. Hierbei wird im wesentlichen nach dem Format von C unterschieden.

```
procedure mul ( α, A, B, β, C )
    procedure rec_mul ( α, A, B, C )
        if A, B und C sind Blockmatrizen then
            for all τ' ∈ S(τ(C)) do
                for all σ' ∈ S(σ(C)) do
                    for all τ'' ∈ S(τ(B)) do
                        rec_mul( α, A(τ', τ''), B(τ'', σ'), C(τ', σ') );
        else
            C := C + αAB;
    end;

    C := βC;
    rec_mul( α, A, B, C );
end;
```

Algorithmus 3.3.1: Rekursiver Matrix-Multiplikationsalgorithmus

Man beachte außerdem, dass die Blockclusterbäume, der in den beschriebenen Multiplikationsalgorithmen auftretenden Matrizen A, B, und C, bereits vorgegeben sind, d.h. die Struktur der Matrizen ist fest und soll auch nicht verändert werden. Somit gilt es lediglich, Routinen für die bei der Multiplikation auftretenden Fälle zu untersuchen und nicht ein optimales Format für das Produkt $A \cdot B$ zu finden (siehe etwa [Gra01]).

Die Analyse der Komplexität der im Anschluss beschriebenen Algorithmen führt zu dem folgenden Resultat für den Aufwand der $\mathcal{H}$-Matrix-Multiplikation.

Lemma 3.3.1 (Aufwand der Matrix-Multiplikation) *Seien*

$$\begin{aligned} c_{\mathrm{sp}} &= \max\{c_{\mathrm{sp}}(T(I\times J)), c_{\mathrm{sp}}(T(I\times K)), c_{\mathrm{sp}}(T(K\times J))\} \\ &\quad \textit{und} \\ p &= \min\{p(T(I\times J)), p(T(I\times K)), p(T(K\times J))\}. \end{aligned}$$

Dann besitzt die Matrix-Multiplikation (3.3.1) mit den Matrizen $A \in \mathcal{H}(T(I\times K), B \in \mathcal{H}(T(K\times J))$ und $C \in \mathcal{H}(T(I\times J))$ eine Komplexität von

$$\mathcal{W}_{\mathrm{MM}}(A,B,C) = \mathcal{O}\left(c_{\mathrm{sp}}^3 p^2 \max\{k, n_{\min}\}^2 \max\{|I|,|J|,|K|\}\right) \tag{3.3.2}$$

Beweis: Siehe [GH03, Lemma 2.10, 2.17 und 2.19]. □

Die Matrix-Multiplikation besitzt somit einen ähnlichen Aufwand wie die Matrix-Addition oder die Matrix-Vektor-Multiplikation (vergleiche (3.2.4) und (3.1.4)). Allerdings gehen die Schwachbesetztheit, die Baumtiefe und der Rang quadratisch bzw. kubisch in die Abschätzungen ein. In Kombination mit deutlich größeren Konstanten (siehe [Gra01]) ergibt sich damit in der Praxis eine wesentlich höhere Laufzeit als bei den vorherigen Operationen, wie auch die Beispiele in Abschnitt 6.5 belegen.

Bemerkung 3.3.2 *Für den Fall eines binären, kardinalitätsbalancierten Clusterbaumes (siehe Beispiel 2.2.8) folgt aus* (3.3.2)*:*

$$\mathcal{W}_{\mathrm{MM}}(A,B,C) = \mathcal{O}\left(c_{\mathrm{sp}}^3 \max\{k, n_{\min}\}^2 n \log^2 n\right)$$

Alternativ zu einem festen Rang während der Multiplikation lassen sich auch alle Kürzungsschritte mit einer festen Genauigkeit durchführen, wie diese in Abschnitt 3.2.1 beschrieben ist. In den Routinen der nachfolgenden Abschnitte sind die Kürzungsoperatoren hierfür entsprechend anzupassen.

3.3.1 C ist eine Rang-k-Matrix

Betrachtet wird die Situation bei der sowohl A als auch B Blockmatrizen darstellen. Ist dies nicht der Fall, kann die Multiplikation direkt ausgeführt oder z.B. auf die Matrix-Vektor-Multiplikation reduziert werden.

Die zuvor beschriebene Rekursion lässt sich bei gegebenen Blockmatrizen A und B fortsetzen. Die jeweils auftretenden Ergebnisse können dabei sukzessive auf C aufaddiert werden.

```
procedure mul_rank ( α, A, B, C ) ;
    if A, B sind Blockmatrizen then
        for all τ' ∈ S(τ(C)) do
            for all σ' ∈ S(σ(C)) do
                C' sei R(k, τ', σ')-Matrix;
                for all τ'' ∈ S(τ(B)) do
                    mul( α, A(τ', τ''), B(τ'', σ'), C' );
                C := C + P_{(τ',σ')→(τ(C),σ(C))}(C');
            endfor;
    else
        C := C + αAB;
    endif;
end;

mul_rank( α, A, B, C );
C := T_k(C);
```

Algorithmus 3.3.2: Rekursive Multiplikation bei einer R-Matrix

Durch den abschließenden Kürzungsschritt mittels Algorithmus 3.2.2 wird der Rang von C entsprechend auf k reduziert. Das Ergebnis ist nach Lemma 3.2.2 optimal. Die Komplexität dieses Verfahrens ist maßgeblich durch den Aufwand beim Kürzen definiert, wie das folgende Resultat zeigt:

Lemma 3.3.3 *Seien $\tau = \tau(C)$, $\sigma = \sigma(C)$ und $v = (\tau, \sigma)$. Der Rang k' von C nach Ausführung von Algorithmus 3.3.2 ist:*

$$k' = \mathcal{O}\left(c_{\mathrm{id}} \max\{c_{\mathrm{sp}}(T(v)) \cdot p(T(v)) \cdot k, n_{\min}\}\right). \tag{3.3.3}$$

Für den Aufwand der Multiplikation mittels Algorithmus 3.3.2 gilt:

$$\mathcal{O}\left(c_{\mathrm{id}}^3 c_{\mathrm{sp}}(T(v))^3 k^3 p(T(v))^3 \max\{|\tau|,|\sigma|,|\mathcal{L}(T(v))|\}\right). \tag{3.3.4}$$

Beweis: Siehe Satz 5.26 und Bemerkung 5.28 in [Gra01]. □

Die Konstante c_{id} in (3.3.3) ist abhängig von der Struktur des Blockclusterbaumes und wird in [Gra01, Abschnitt 5.5] ausführlich diskutiert.

Der Rang vor dem Kürzen ist abhängig von der Tiefe des Blockclusterbaumes und kann folglich sehr große Werte annehmen. Auch für die Komplexität der Multiplikation ergibt sich nach (3.3.3) eine quadratische bzw. kubische Abhängigkeit von der Tiefe. Um diesen Anteil zu reduzieren, lassen sich die, während der Rekursion berechneten Teilergebnisse auch hierarchisch addieren. Hierbei werden die Ergebnisse in jedem Rekursionsschritt zusammengefasst und der Rang der dabei entstehenden Matrix auf k gekürzt.

```
   procedure mul_rank_hier ( α, A, B, C ) ;
       if A, B sind Blockmatrizen then
           C' sei R(k, τ(C), σ(C))-Matrix;
           for all τ' ∈ S(τ(C)) do
               for all σ' ∈ S(σ(C)) do
                   C'' sei R(k, τ', σ')-Matrix;
                   for all τ'' ∈ S(τ(B)) do
                       mul( α, A(τ', τ''), B(τ'', σ'), C'' );
                   C' := C' + P_{(τ',σ')→(τ(C),σ(C))}(C'');
               endfor;
           C := C + C';
       else
           C := C + αAB;
       endif;
15:    C := T_k(C);
   end;
```

Algorithmus 3.3.3: Hierarchische Multiplikation bei einer R-Matrix

In jedem Rekursionsschritt werden in Algorithmus 3.3.3 $|\mathcal{S}(\tau(C))|\cdot|\mathcal{S}(\sigma(C))|$ Teilergebnisse C'' berechnet. Folglich ist der Rang von C vor dem jeweiligen Kürzungsschritt in Zeile 14 beschränkt durch

$$k(1+|\mathcal{S}(\tau(C))|\cdot|\mathcal{S}(\sigma(C))|).$$

Der Aufwand für die Reduktion des Ranges ist somit unabhängig von der Tiefe der betrachteten Blockclusterbäume. Für die Komplexität von Algorithmus 3.3.3 folgt damit:

Lemma 3.3.4 *Seien $\tau = \tau(C)$, $\sigma = \sigma(C)$ und $v = (\tau, \sigma)$. Der Aufwand für die Berechnung des Matrix-Produktes $C := \alpha AB + C$ mit Algorithmus 3.3.3 beträgt*

$$\mathcal{O}\left(c_{\mathrm{sp}}(T(v))^2 k^2 p(T(v))^2 \max\{|\tau|,|\sigma|\} + p(T(v))k^3|\mathcal{L}(T(v))|\right). \tag{3.3.5}$$

Beweis: Siehe Folgerung 5.27 in [Gra01]. □

Die Komplexität der hierarchischen Multiplikation entspricht (3.3.2), falls der lokale Teilbaum $T(\tau(C), \sigma(C))$ betrachtet wird. Im Vergleich zu (3.3.4) entfällt die kubische Abhängigkeit von der Baumtiefe. Das derart berechnete Produkt stellt allerdings im allgemeinen keine Bestapproximation an $\alpha AB + C$ mehr dar.

3.3.2 C ist eine Blockmatrix

Dieser Fall tritt auf, falls einer der beiden Faktoren A oder B einer Matrix entspricht, die mit einem Blatt in T korrespondiert. Anderenfalls würde die Rekursion in Algorithmus 3.3.1 fortgesetzt. Anstelle des rekursiven Aufrufs erfolgt hierbei die direkte Multiplikation der Faktoren. Die Addition des dabei entstehenden Produktes auf C wird im Anschluss getrennt durchgeführt.

Das Ergebnis $C' = \alpha AB$ liegt typischerweise zunächst im gleichen Format vor wie das des entsprechenden Blattes. Korrespondieren beide Faktoren mit Blättern, so ist eine R-Repräsentation vorzuziehen.

Sind die Matrizen A und B ausgeprägte Rechtecksmatrizen, d.h. $|\tau(A)| \gg |\sigma(A)|$ und $|\sigma(B)| \gg |\tau(B)|$, so kann sich ein hoher Aufwand für den Fall ergeben, dass weder A noch B eine R-Matrix sind. Allerdings ist der Rang der beteiligten Matrizen im Vergleich zur Dimension des Resultates klein. Vollbesetzte Matrizen können somit i.d.R. in das R-Format konvertiert werden. Bei kardinalitätsbalancierten Bäumen, wie sie in dieser Arbeit betrachtet werden, tritt dieser Fall allerdings nicht auf.

Die Addition auf C erfolgt nach der Multiplikation für jedes Blatt in $T(\tau(C), \sigma(C))$ einzeln:

for all $(\tau, \sigma) \in \mathcal{L}(T(\tau(C), \sigma(C)))$ **do**
  $C(\tau, \sigma) := C(\tau, \sigma) + \mathcal{R}_{(\tau(C),\sigma(C))\to(\tau,\sigma)}(C')$;
  if $(\tau, \sigma) \in \mathcal{L}_z(T)$ **then**
    $C(\tau, \sigma) := \mathcal{T}_k(C(\tau, \sigma))$;
endfor;

3.3.3 C ist eine vollbesetzte Matrix

Die Berechnung des Produktes $C := C + \alpha AB$ für den Fall einer vollbesetzten Matrix C erfolgt analog zu der Vorgehensweise bei einer R-Matrix, d.h. beim Auftreten zweier Blockmatrizen A und B wird die Multiplikation rekursiv fortgesetzt. Allerdings ist die hierarchische Addition der dabei auftretenden Teilergebnisse nicht notwendig, da die direkte Aktualisierung der Matrix C nicht mit Komplexitätseinbußen verbunden ist.

Das resultierende Verfahren entspricht somit Algorithmus 3.3.2, wobei der finale Kürzungsschritt entfällt. Die Berechnung des Produktes erfolgt hierbei ebenfalls direkt, falls einer der Faktoren mit einem Blatt korrespondiert.

Es sei allerdings bemerkt, dass ein rekursiver Aufruf der Multiplikationsroutine 3.3.4 nur selten auftritt, da bei vollbesetzten Matrizen typischerweise mindestens ein Cluster des korrespondierenden Knotens (τ, σ) ein Blatt im Clusterbaum darstellt. Gilt dies für τ, so sind nach Definition des Blockclusterbaumes alle Knoten $(\tau, \sigma') \in T$ ebenfalls Blätter. Gleiches gilt für den Fall, dass σ keine Söhne besitzt. Entsprechend ist mindestens ein Faktor A oder B eine D- oder eine R-Matrix.

```
procedure mul_dense ( α, A, B, C ) ;
    if A, B sind Blockmatrizen then
        for all τ' ∈ S(τ(C)) do
            for all σ' ∈ S(σ(C)) do
                C' sei R(k, τ', σ')-Matrix;
                for all τ'' ∈ S(τ(B)) do
                    mul( α, A(τ', τ''), B(τ'', σ'), C' );
                C := C + P_{(τ',σ')→(τ(C),σ(C))}(C');
            endfor;
    else
        C := C + αAB;
    endif;
end;

mul_dense( α, A, B, C );
```

Algorithmus 3.3.4: Rekursive Multiplikation bei einer D-Matrix

Lediglich für den, im allgemeinen ineffizienten Fall, dass sowohl τ als auch σ kein Blatt darstellen, ist ein rekursiver Aufruf in Algorithmus 3.3.4 möglich.

3.4 Matrix-Inversion

In diesem Abschnitt soll die Berechnung der Inversen A^{-1} einer $\mathcal{H}$-Matrix $A \in \mathcal{H}(T, k)$ untersucht werden. Auch in diesem Fall muss allerdings der Rang der zu berechnenden Matrix begrenzt werden, um den hierbei entstehenden Aufwand im gewünschten fast-linearen Rahmen zu belassen. Somit wird anstelle der exakten Inversen lediglich eine Approximation $C \in \mathcal{H}(T, k)$ bestimmt. Die Existenz von A^{-1} und eine Approximierbarkeit durch eine $\mathcal{H}$-Matrix werden dafür vorausgesetzt. Im allgemeinen ist diese Annahme nicht zwingend erfüllt und muss gegebenenfalls geprüft werden. Ein Existenzbeweis für das FEM-Beispiel aus Abschnitt 2.4.2 ist z.B. in [BH03] nachzulesen.

Diskutiert werden an dieser Stelle zwei verschiedene Verfahren. In Abschnitt 3.4.1 wird zunächst der Gauß'sche Eliminationsalgorithmus verwendet. Auf der Newton-Iteration basiert dagegen der zweite Algorithmus, welcher in Abschnitt 3.4.2 beschrieben wird.

3.4.1 Gauß-Elimination

Als Grundlage für den Inversionsalgorithmus dient die Gauß-Elimination. Wie bei der Matrix-Multiplikation wird auch in diesem Fall die Blockversion verwendet.

Um die Reihenfolge bei der Inversion zu beachten, seien die Söhne eines Clusters τ im folgenden angeordnet: $\mathcal{S}(\tau) = \{\tau_0, \tau_1, \ldots, \tau_{q-1}\}$. Desweiteren sei zu einer Matrix $M \in \mathbb{K}^{\tau\times\sigma}$ mit $\mathcal{S}(\tau,\sigma) \neq \emptyset$ die Teilmatrix M_{ij} definiert als

$$M_{ij} = M(\tau_i, \sigma_j).$$

Der Algorithmus zur Berechnung einer $\mathcal{H}$-Inversion lässt sich anschließend wie folgt formulieren:

```
procedure invert( A, C )
    if v(A) ∈ L_nz(T) then C = A^{-1};
    else
        { Elimination der unteren Dreiecksmatrix }
        for all i := 0, ..., |S(τ(A))| − 1 do
            { Skalierung der aktuellen Zeile }
            invert( A_ii, C_ii );
            for all j := i + 1, ..., |S(σ(A))| − 1 do A_ij := C_ii A_ij;
            for all j := 0, ..., i − 1 do C_ij := C_ii C_ij;
            { Elimination der Spalte }
            for all l := i + 1, ..., |S(τ(A))| − 1 do
                for all j := i + 1, ..., |S(σ(A))| − 1 do
                    A_lj := A_lj − A_li · A_ij;
            for all l := i + 1, ..., |S(τ(A))| − 1 do
                for all j := 0, ..., i do
                    C_lj := C_lj − A_li · C_ij;
        endfor;
        { Elimination der oberen Dreiecksmatrix }
        for all i := |S(τ(A))| − 1, ..., 0 do
            for all l := i − 1, ..., 0 do
                for all j := 0, ..., |S(σ(A))| − 1 do
                    C_lj := C_lj − A_li · C_ij;
    endif;
end;
```

Algorithmus 3.4.1: Inversion einer $\mathcal{H}$-Matrix

Man beachte, dass die Multiplikationen innerhalb von Algorithmus 3.4.1 alle vom Typ (3.3.1) sind. Somit können die entsprechenden Berechnungen mit Hilfe von Algorithmus 3.3.1 durchgeführt werden.

In der Praxis ist die Implementierung des allgemeinen Gauß-Algorithmus für die Inversion einer $\mathcal{H}$- bzw. Block-Matrizen in der Regel nicht notwendig, da häufig Binärbäume für den Clusterbaum genutzt werden (siehe Beispiel 2.2.8). In diesem Fall hat die Matrix A die

Gestalt

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}$$

Durch die Anwendung des obigen Eliminationsverfahrens folgt für die Inverse von A:

$$A^{-1} = \begin{pmatrix} A_{00}^{-1} + A_{00}^{-1} A_{01} S^{-1} A_{10} A_{00}^{-1} & -A_{00}^{-1} A_{01} S^{-1} \\ -S^{-1} A_{10} A_{00}^{-1} & S^{-1} \end{pmatrix},$$

mit dem *Schurkomplement*

$$S = A_{11} - A_{10} A_{00}^{-1} A_{01}.$$

Unter Ausnutzung dieser Darstellung und Verwendung der Funktion `mul` aus Algorithmus 3.3.1 vereinfacht sich Algorithmus 3.4.1 zu

```
procedure invert_bin ( A, C )
    invert_bin( A_00, C_00 );
    mul( 1, C_00, A_01, 0, T_01 );
    mul( 1, A_10, C_00, 0, T_10 );
    mul( -1, A_10, T_01, 1, A_11 );
    invert_bin( A_11, C_11 );
    mul( -1, T_01, C_11, 0, C_01 );
    mul( -1, C_11, T_10, 0, C_10 );
    mul( -1, T_01, C_10, 1, C_00 );
end;
```

Algorithmus 3.4.2: $\mathcal{H}$-Matrix-Inversion für Binärbäume

Vergleicht man die einzelnen Operationen in den Algorithmen 3.3.1 und 3.4.1, so erkennt man weiterhin, dass bei der Gauß-Inversion die gleichen Multiplikationen bzw. Additionen wie bei der Blockmultiplikation vorkommen. Lediglich bei den Diagonalelementen wird eine Rekursion anstelle einer Multiplikation durchgeführt. Besonders deutlich wird dies bei einem Vergleich der Matrix-Multiplikation für binäre Clusterbäume und Algorithmus 3.4.2. Per Induktion lässt sich aus dieser Eigenschaft das folgende Resultat für die Komplexität der Inversion mittels Gauß-Elimination zeigen.

Lemma 3.4.1 (Aufwand der Gauß-Elimination) *Sei $A \in \mathcal{H}(T, k)$. Dann beträgt der Aufwand zur Berechnung der Inversion mittels Algorithmus 3.4.1:*

$$\mathcal{W}_{\mathrm{MI,Gauß}}(A) = \mathcal{W}_{\mathrm{MM}}(A). \tag{3.4.1}$$

Beweis: Siehe Satz 5.29 in [Gra01]. □

Um eine gewünschte Genauigkeit bei der $\mathcal{H}$-Matrix-Inversion zu erreichen ist der gewählte Rang von entscheidender Bedeutung. Die Existenzaussage der Inversen von A im $\mathcal{H}$-Matrix-Format für das FEM-Beispiel in [BH03] gibt hierbei ebenfalls Aufschluss über die Abhängigkeit von k in Bezug auf die Approximationsgüte.

3.4.2 Newton-Iteration

Ein zur Gauß-Elimination alternativer Inversionsalgorithmus stellt die Newton-Iteration (siehe [HK00]) dar. Zu der gegebenen Matrix A wird dabei die Matrix-Gleichung

$$X^{-1} - A = 0$$

betrachtet, welche sich durch die Iterationsvorschrift

$$X_{i+1} = 2X_i - X_i A X_i \tag{3.4.2}$$

lösen lässt. Der Startwert X_0 der Iteration ist hierfür geeignet zu wählen.

Mit Hilfe der in Algorithmus 3.2.1 und Algorithmus 3.3.1 definierten Funktionen ergibt sich somit das nachfolgende Verfahren. Hierbei stellen $T_1 = X_i A$ und $T_2 = T_1 X_i$ temporäre Matrizen dar, da die Faktoren während der Multiplikation nicht verändert werden dürfen.

```
procedure invert_newton ( A, C, X_0, ε )
    X := X_0;
    while ‖I − AX‖ > ε do
        mul( 1, X, A, 0, T_1 );
        mul( 1, T_1, X, 0, T_2 );
        add( 2, X, −1, T_2 );
    endwhile;
end;
```

Algorithmus 3.4.3: Inversion mittels Newton-Iteration

In [Gra01] wurde die Konvergenz der Newton-Iteration untersucht, wobei sich im allgemeinen ein negatives Resultat bei einem zu kleinen Rang oder einem ungeeigneten Startwert X_0 ergibt. Ist k dagegen hinreichend groß und befindet sich X_0 im Konvergenzradius der Newton-Iteration, so strebt Algorithmus 3.4.3 mit einer quadratischen Konvergenzrate zum gesuchten Grenzwert.

Lemma 3.4.2 *Sei $A \in \mathcal{H}(T,k)$ invertierbar und $X_0 \in \mathcal{H}(T,k)$ mit*

$$\|A\|_F \|X_0 - A\|_F = q < \frac{1}{9}.$$

Weiterhin sei $B \in \mathcal{H}(T,k)$ die Bestapproximation von A^{-1} im Raum der $\mathcal{H}$-Matrizen. Es gelte für B:

$$\|A^{-1} - B\|_F \leq \varepsilon_A \leq \frac{1}{16}\|A\|_F^{-1}.$$

Dann gilt für die Iterierten X_i in (3.4.2)

$$\|A^{-1} - X_i\|_F \leq c q^{2^i} \|A\|_F^{-1} + 2\varepsilon_A,$$

mit $c > 0$.

Beweis: Siehe Beweis zu Lemma 4.22 in [Gra01]. □

Mit den Aufwandsabschätzungen für die Matrix-Addition und -Multiplikation ergibt sich ein fast optimales Ergebnis für die Komplexität der Newton-Iteration.

Lemma 3.4.3 (Aufwand der Newton-Iteration) *Unter den Annahmen von Lemma 3.4.2 besitzt die Berechnung der Inversen $A \in \mathcal{H}(T,k)$ bei einer vorgegebenen Genauigkeit von $\varepsilon > 0$ mittels Algorithmus 3.4.3 eine Komplexität von:*

$$\mathcal{W}_{\mathrm{MI,Newton}}(A) = \mathcal{W}_{\mathrm{MM}}(A)\mathcal{O}\left(\log\log\varepsilon^{-1}\right). \tag{3.4.3}$$

Bis auf den zusätzlichen schwach-logarithmischen Term stimmt der Aufwand der Newton-Iteration mit der Komplexität der Gauß-Elimination überein. In der Praxis sind die beiden Verfahren bezüglich der Laufzeit dagegen nur bedingt vergleichbar. Der Grund hierfür liegt in der Vielzahl von Multiplikationen, die typischerweise für die Newton-Iteration ausgeführt werden müssen. Dagegen entspricht der Aufwand für eine Gauß-Elimination im wesentlichen einer einzigen Multiplikation.

3.5 LU-Zerlegung

In vielen numerischen Algorithmen wird die explizite Repräsentation der Inversen eines Operators in der Regel nicht verlangt. Hier genügt es häufig, wenn einige wenige Operationen, z.B. die Matrix-Vektor-Multiplikation mit der Inversen durchgeführt werden können. In solchen Fällen bietet sich etwa die *LU-Zerlegung*

$$A = LU$$

mit der unteren Dreiecksmatrix L und der oberen Dreiecksmatrix U als Darstellung der Inversen an. In L wird die Diagonale dabei auf 1 normiert. Die Auswertung von $A^{-1}x = y$ geschieht in diesem Fall durch ein entsprechendes *Rückwärts-* bzw. *Vorwärtseinsetzen* der Faktoren L und U mit den betrachteten Vektoren.

Analog zu den bisherigen Algorithmen dient auch in diesem Abschnitt die Blockversion einer LU-Faktorisierung als Basis für die Berechnung der Matrizen L und U. Für die Bestimmung der Nichtdiagonalblöcke L_{ij}, U_{ij} mit $i \neq j$ besteht hierbei die Möglichkeit, auf die entsprechenden Inversionsalgorithmen für $\mathcal{H}$-Matrizen zurückzugreifen. Dieser Ansatz stellt sich in der Praxis aber als nicht effizient heraus. Aufgrund der Dreiecksgestalt der beteiligten Matrizen L_{ii} bzw. U_{ii} können allerdings spezielle Faktorisierungsalgorithmen genutzt werden, um so zu einem schnellen Verfahren zu gelangen.

```
   procedure lu( L, U, A )
       if v(A) ∈ L_nz(T) then
           berechne LU = A; return ;
       endif
       for all i = 0, ..., |S(τ(A))| − 1 do
6:         lu( L_ii, U_ii, A_ii );
           { Auflösen der i-ten Zeile und Spalte }
8:         for all j = i + 1, ..., |S(τ(A))| − 1 do löse L_ji U_ii = A_ji;
9:         for all j = i + 1, ..., |S(σ(A))| − 1 do löse L_ii U_ij = A_ij;
           { Aktualisieren der Restmatrix }
           for all l = i + 1, ..., |S(τ(A))| − 1 do
               for all j = i + 1, ..., |S(σ(A))| − 1 do
                   A_lj := A_lj − L_li U_ij
       endfor;
   end;
```

Algorithmus 3.5.1: LU-Zerlegung einer $\mathcal{H}$-Matrix

Betrachtet wird zunächst der Fall in Zeile 8, wobei die Gleichung $L_{ji}U_{ii} = A_{ji}, j > i$, gelöst werden soll. Die Matrix U_{ii} wurde bereits in Zeile 6 berechnet und hat obere Dreiecksgestalt. Das System hat damit die Form:

$$\begin{pmatrix} L_{00} & \cdots & L_{0,m-1} \\ \vdots & \ddots & \vdots \\ L_{m-1,0} & \cdots & L_{m-1,m-1} \end{pmatrix} \begin{pmatrix} U_{00} & \cdots & U_{0,m-1} \\ 0 & \ddots & \vdots \\ 0 & 0 & U_{m-1,m-1} \end{pmatrix} = \begin{pmatrix} A_{00} & \cdots & A_{0,m-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \cdots & A_{m-1,m-1} \end{pmatrix},$$

wobei der Einfachheit halber eine $m \times m$ Blockstruktur angenommen wurde. Man beachte, dass L in diesem Fall keine untere Dreiecksmatrix ist. Dieses System führt zu Gleichungen der Form

$$L_{ij}U_{jj} = A_{ij} - \sum_{l=0}^{i-1} L_{il}U_{lj},$$

die, beginnend mit der ersten Spalte, sukzessive gelöst werden können:

```
   procedure solve_L( L, U, A )
       if v(L) ∈ L_nz(T) then
           löse LU = A direkt; return ;
       endif
       for all j = 0, ..., |S(σ(L))| − 1 do
           { Auflösen in der j-ten Spalte }
           for all i = 0, ..., |S(τ(L))| − 1 do solve_L( L_ij, U_jj, A_ij );
           { Aktualisieren der rechten Restmatrix }
           for all l = j + 1, ..., |S(σ(L))| − 1 do
               for all i = 0, ..., |S(τ(L))| − 1 do
                   A_il := A_il − L_ij U_jl
       endfor;
   end;
```

Algorithmus 3.5.2: Lösen von $LU = A$ bei gegebenem U und A

Korrespondiert L mit einem Blatt, so kann die Gleichung $LU = A$ direkt für jede Zeile von L durch Vorwärtseinsetzen mit U^T gelöst werden.

Analog löst man Gleichung $L_{ii}U_{ij} = A_{ij}$ in Zeile 9 von Algorithmus 3.5.1, wobei in diesem Fall die Elemente U_{ij} zeilenweise, beginnend mit der ersten Zeile, berechnet werden.

```
procedure solve_U( L, U, A )
    if v(L) ∈ L_nz(T) then
        löse LU = A direkt; return ;
    endif
    for all i = 0, ..., |S(τ(U))| − 1 do
        { Auflösen in der i-ten Zeile }
        for all j = 0, ..., |S(σ(U))| − 1 do solve_U( L_ii, U_ij, A_ij );
        { Aktualisieren der unteren Restmatrix }
        for all l = i + 1, ..., |S(τ(U))| − 1 do
            for all j = 0, ..., |S(σ(L))| − 1 do
                A_lj := A_lj − L_li U_ij
    endfor;
end;
```

Algorithmus 3.5.3: Lösen von $LU = A$ bei gegebenem L und A

Die Gleichung kann hierbei direkt durch Vorwärtseinsetzen jeder Spalte von U in L gelöst werden, falls U als R- bzw. als vollbesetzte Matrix vorliegt.

Bemerkung 3.5.1 *Für den symmetrischen Fall $A = A^T$ genügt die Berechnung von L in Algorithmus 3.5.1, womit die* Cholesky*-Zerlegung*

$$A = LL^T$$

bestimmt wird (siehe auch [Lin02]). Die Aktualisierung der Restmatrix beschränkt sich hierbei auf die untere Dreiecksmatrix. Insgesamt ergibt sich ein halbierter Aufwand im Vergleich zur LU-Zerlegung. Die auftretenden Multiplikationen sollten sich hierbei ebenfalls auf die untere Dreiecksmatrix beschränken, wofür Algorithmus 3.3.1 entsprechend abzuändern ist.

Für den einfacheren Fall von binären Clusterbäumen lauten die entsprechenden Algorithmen:

```
procedure lu_bin( L, U, A )
    if v(A) ∈ L_z(T) then
        berechne LU = A; return ;
    endif;
    lu_bin( L_00, U_00, A_00 );
    solve_U_bin( L_00, U_01, A_01 );
    solve_L_bin( L_10, U_00, A_10 );
    mul( −1, L_10, U_01, 1, A_11 );
    lu_bin( L_11, U_11, A_11 );
end;
```

Algorithmus 3.5.4: LU-Zerlegung für binäre Clusterbäume

bzw.

```
procedure solve_L_bin( L, U, A )
    if v(L) ∈ L_z(T) then
        löse LU = A direkt; return ;
    endif;
    solve_L_bin( L_00, U_00, A_00 ); solve_L_bin( L_10, U_00, A_10 );
    mul( −1, L_00, U_01, 1.0, A_01 ); mul( −1, L_10, U_01, 1.0, A_11 );
    solve_L_bin( L_01, U_11, A_01 ); solve_L_bin( L_11, U_11, A_11 );
end;
```

Algorithmus 3.5.5: Auflösen nach L

und

```
procedure solve_U_bin( L, U, A )
    if v(L) ∈ L_z(T) then
        löse LU = A direkt; return ;
    endif;
    solve_U_bin( L_00, U_00, A_00 ); solve_U_bin( L_00, U_01, A_01 );
    mul( −1, L_10, U_00, 1.0, A_10 ); mul( −1, L_10, U_01, 1.0, A_11 );
    solve_U_bin( L_11, U_10, A_10 ); solve_U_bin( L_11, U_11, A_11 );
end;
```

Algorithmus 3.5.6: Auflösen nach U

wobei die Funktion `mul` aus Algorithmus 3.3.1 genutzt wurde.

Wie auch bei der $\mathcal{H}$-Matrix-Inversion bleibt bei der LU-Zerlegung die Frage nach der Existenz einer solchen Faktorisierung im $\mathcal{H}$-Matrix-Format unbeantwortet. Für die Komplexität ergibt sich dagegen folgendes Resultat:

Lemma 3.5.2 (Aufwand der LU-Zerlegung) *Sei $A \in \mathcal{H}(T,k)$. Dann beträgt der Aufwand zur Berechnung der LU-Zerlegung $A = LU$ mit $L, U \in \mathcal{H}(T,k)$ mittels Algorithmus 3.5.1:*

$$\mathcal{W}_{\mathrm{LU}}(A) = \mathcal{W}_{\mathrm{MI}}(A). \tag{3.5.1}$$

Beweis: Siehe [GHK04]. □

Obwohl die Ordnung der Komplexität identisch zur Matrixinversion ist, sind die auftretenden Konstanten häufig deutlich kleiner, wie auch die numerischen Beispiele in Abschnitt 6.7 zeigen.

4 Lastbalancierung

Die Grundlage effizienter Algorithmen auf parallelen System bildet eine adäquate Lastverteilung. In diesem Kapitel werden verschiedene Verfahren vorgestellt, die den Aufwand der im Kontext der $\mathcal{H}$-Matrix-Arithmetik entsteht, gleichmäßig auf alle Prozessoren verteilt. Diese Verteilungsverfahren bilden hiermit die Basis der eigentlichen parallelen Algorithmen, welche in Kapitel 6 beschrieben werden.

4.1 Zuteilung

Anstelle der eigentlichen Aufgaben aus dem Bereich der $\mathcal{H}$-Matrizen werden im folgenden allgemein n „Jobs“ $J = \{j_0, \ldots, j_{n-1}\}$ betrachtet. Jede der Aufgaben j_i benötige hierbei die Zeit $t(i) = t(j_i)$. Weiterhin werden p Prozessoren bzw. Maschinen $M = \{M_0, \ldots, M_{p-1}\}$ als gegeben vorausgesetzt.

Definition 4.1.1 (Zuteilung) *Eine* Zuteilung *oder* Scheduling *ist ein Paar* $\Sigma = (m, s)$ *von Funktionen* $m : J \to M$ *und* $s : J \to \mathbb{R}_{\geq 0}$*, die jeder Aufgabe* j_i *eine Maschine* $m(i) = m(j_i)$ *und eine Startzeit* $s(i) = s(j_i)$ *zuordnet, so dass für alle* $j_1, j_2 \in J$ *mit* $m(j_1) = m(j_2)$ *und* $s(j_1) < s(j_2)$ *gilt:* $s(j_2) \geq s(j_1) + t(j_1)$*. Die Gesamtzeit* $T(\Sigma)$*, die die parallele Maschine für die Bearbeitung benötigt, ist* $T(\Sigma) = \max_{i=0}^{n-1}(s(i) + t(i))$*.*

Die Aufgabe beim *Zuteilungs-* bzw. *Schedulingproblem* besteht im folgenden darin, eine Zuteilungsfunktion zu finden, so dass $T(\Sigma)$ minimal wird. Die einzelnen Maschinen M_i werden dabei als identisch angenommen. Leider ist dieses Problem im allgemeinen NP-schwer (siehe [GJ79]). Damit können, unter der Annahme, dass P≠NP, nur approximative Lösungen in effizienter, also polynomialer Zeit berechnet werden. Für die Verwendung im Kontext der $\mathcal{H}$-Matrizen besteht außerdem die Forderung nach Algorithmen mit linearem Aufwand, um für die Praxis taugliche Verfahren zu erhalten. Die nächsten Abschnitte stellen solche Algorithmen vor.

In den parallelen Verfahren der Kapitel 6 und 7 spielt der genaue Zeitpunkt, wann eine Aufgabe von einem Prozessor bearbeitet wird, nur eine untergeordnete Bedeutung. Wichtig ist in der Regel nur die Zuordnung der Aufgaben zu den einzelnen Prozessoren, also die Abbildung m. Deshalb beschränken sich die nachfolgenden Verteilungsalgorithmen auch auf die Angabe dieser Funktion.

Die Gesamtzeit $T(\Sigma)$ lässt sich in diesem Fall auch als

$$\max_{0 \le q < p} \sum_{j \in J_q} t(j) \quad \text{mit} \quad J_q = \{j : m(j) = q\} \tag{4.1.1}$$

formulieren. Anstelle der Laufzeiten $t(i)$ können somit auch äquivalent Kosten $c(i) = c(j_i)$ verwendet werden. Hierbei ist das Ziel des Zuteilungsproblems, die maximalen Kosten pro Prozessor zu minimieren. Je nach Verteilungsalgorithmus wird im folgenden auf eine der beiden Formulierung zurückgegriffen.

Da alle Maschinen als identisch angenommen wurden, ist nicht die Menge M, sondern lediglich die Anzahl der Prozessoren p entscheidend für das Scheduling. Entsprechend sei für die gegebene Aufgabenmenge J und p Maschinen

$$T_{\min} = T_{\min}(J, p)$$

die Zeit, die mit einer optimalen Zuteilung erreicht werden kann.

4.1.1 List-Scheduling

Ein einfacher, approximativer Algorithmus zur Berechnung einer Zuteilung ist *List-Scheduling*. Hierbei werden die Aufgaben in eine beliebige Reihenfolge gebracht und jeder freien Maschine der jeweils nächste Job der Liste zugewiesen.

```
procedure list_scheduling( J, M )
    s_0, ..., s_{p-1} = 0;
    for 0 ≤ i < n do
        sei j ∈ {0, ..., p-1} mit s_j = min_{l=0}^{p-1} s_l;
        setze m(j_i) = M_j, s(j_i) = s_j;
        s_j = s_j + t(j_i);
    endfor;
end;
```

Algorithmus 4.1.1: List-Scheduling

Die Analyse dieses Algorithmus ergibt:

Lemma 4.1.2 *Sei Σ_{LS} die vom List-Scheduling berechnete Zuteilung. Dann gilt:*

$$T(\Sigma_{\text{LS}}) \le \left(2 - \frac{1}{p}\right) T_{\min}. \tag{4.1.2}$$

Beweis: Siehe [Gra69]. □

Dass die obere Schranke aus (4.1.2) auch erreicht wird, zeigt das folgende Beispiel: Es seien $n = p(p-1)+1$ Jobs mit $t(0) = \cdots = t(n-2) = 1$ und $t(n-1) = p$ gegeben. Dann gilt: $T(\Sigma_{\text{LS}}) = (p-1) + p = 2p - 1$ und $T_{\min} = p$ (siehe Abbildung 4.1.1). Somit folgt: $T(\Sigma_{LS}) = 2p - 1 = (2 - \frac{1}{p})p = (2 - \frac{1}{p})T_{\min}$.

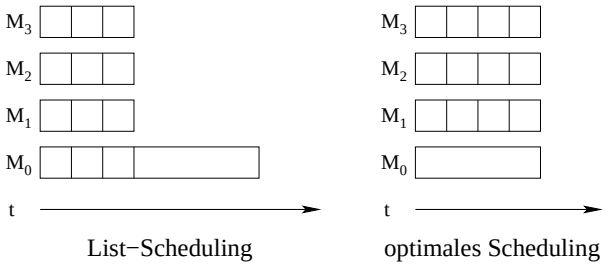


Abbildung 4.1.1: Scheduling für $p = 4$

Wie in diesem Beispiel deutlich wird, tritt die obere Schranke in (4.1.2) auf, falls sehr teure Aufgaben zuletzt abgearbeitet werden. Vermeidet man dies und verteilt die Jobs nicht in einer beliebigen Reihenfolge, sondern sortiert sie entsprechend der benötigten Zeit, beginnend mit dem teuersten, so ergibt sich eine effizientere Variante des List-Schedulings. Dieser modifizierte Algorithmus wird *Longest-processing-Time-* oder *LPT-* Scheduling genannt.

```
procedure lpt_scheduling( J, M )
    s_0, ..., s_{p-1} = 0;
    while J ≠ ∅ do
        sei i ∈ J mit t(i) = max_{l=0}^{n-1} t(j_l);
        sei j ∈ {0, ..., p-1} mit s_j = min_{l=0}^{p-1} s_l;
        setze m(i) := M_j, s(i) := s_j;
        s_j = s_j + t(j_i);
    endfor;
end;
```

Algorithmus 4.1.2: LPT-Scheduling

Die größere Effizienz zeigt sich beim LPT-Scheduling in Form einer besseren Schranke bei der Approximation einer optimalem Zuteilung:

Lemma 4.1.3 *Sei Σ_{LPT} die vom LPT-Scheduling berechnete Zuteilung. Dann gilt:*

$$T(\Sigma_{\mathrm{LPT}}) \leq \left(\frac{4}{3} - \frac{1}{3 \cdot p}\right) T_{\min}. \tag{4.1.3}$$

Beweis: Siehe [Gra69]. □

Bemerkung 4.1.4 *Die Komplexität von Algorithmus 4.1.1 beträgt $\mathcal{O}(pn)$. Bei einer einmaligen Sortierung vor der Lastbalancierung ist der Aufwand des LPT-Scheduling durch $\mathcal{O}(n \log n + np)$ bestimmt.*

4.1.2 Multifit-Scheduling

Ein anderer Zugang zum Zuteilungsproblem gelingt durch dessen duales Problem, dem (eindimensionalen) *Bin-Packing*.

Definition 4.1.5 (Bin-Packing-Problem) *Es seien* $J = \{j_0, \ldots, j_{n-1}\}$, $t : J \to \mathbb{R}_{\geq 0}$ *und* $t_{\max} > 0$. *Gesucht ist eine Aufteilung von* J *in eine* minimale *Anzahl von disjunkten Mengen* $B_0, \ldots, B_{p-1}$, *so dass* $\sum_{j \in B_i} t(j) \leq t_{\max}$, $0 \leq i < p$ *gilt.*

Das Problem besteht somit darin, die gegebenen Größen auf eine minimale Anzahl von Behältern („bin“) zu verteilen, wobei jeder der Behälter eine maximale Aufnahmekapazität von $t_{\max}$ besitzt. Angewandt auf das Zuteilungsproblem lautet die Frage, wieviele Prozessoren benötigt werden, um die gegebenen Aufgaben in einer Zeit $t_{\max}$ zu bearbeiten. Umgekehrt ließe sich bei Kenntnis der maximal zur Verfügung stehenden Zeit $t_{\max}$ für eine gegebene Anzahl von Prozessoren mittels eines Bin-Packing-Algorithmus eine entsprechend optimale Verteilung konstruieren. Leider ist auch das Bin-Packing-Problem NP-schwer (siehe [GJ79]), womit ein effizienter Lösungsalgorithmus nicht zur Verfügung steht.

Allerdings sind Approximationsalgorithmen bekannt, die das Bin-Packing-Problem in polynomialer Zeit bis auf einen konstanten Faktor lösen. Einer dieser Algorithmen ist *First-Fit-Decreasing* (*FFD*). FFD arbeitet ähnlich zum LPT-Scheduling, wobei sukzessive jede Größe $j \in J$ der ersten Menge B_i zugewiesen wird, die genügend Kapazität besitzt („first fit“). Die Menge der Aufgaben wird hierbei der Größe nach sortiert, wobei mit dem größten Element begonnen wird („decreasing“). Der Algorithmus lautet somit:

```
procedure ffd ( J, t, t_max )
    p := 0;
    for all j ∈ J do
        f := false; i := 0;
        while i < p ∧ f = false do
            if t_i + t(j) ≤ t_max then
                t_i := t_i + t(j); B_i := B_i ∪ {j};
                f := true;
            endif;
            i := i + 1;
        endwhile;
        if f = false then
            t_p := t(j); B_p := {j};
            p := p + 1;
        endif;
    endfor;
    return p;
end;
```

Algorithmus 4.1.3: First Fit Decreasing für das Bin-Packing-Problem

Die Güte der Approximation von FFD (und auch die Komplexität der entsprechenden Beweise) konnte sukzessive reduziert werden. Die folgende Bemerkung gibt hierbei den aktuellen Stand wieder (siehe auch [Hoc96]).

Bemerkung 4.1.6 *Sei p_{FFD} die Zahl von Mengen B_i, die von Algorithmus 4.1.3 für die Verteilung einer gegebenen Zahl von Aufgaben benötigt wird und $p_{\min}$ die minimal notwendige Anzahl. Dann gilt:*

$$p_{\mathrm{FFD}} \leq \frac{11}{9} p_{\min} + 1. \tag{4.1.4}$$

Beweis: Siehe [Yue91]. □

In [CGJ78] wird nun die Frage untersucht, wie groß $t_{\max}$ sein muss, damit p_{FFD} höchstens p ist. Dies würde eine Aussage über die Güte der hierdurch erzielten Zuteilung im Vergleich zur optimalen Zeit $T_{\min}$ liefern. Beantwortet wird die Frage im folgenden Lemma.

Lemma 4.1.7 *Sei für eine gegebene Menge J von Aufgaben und p Prozessoren*

$$t_{\max} \geq 1.22\, T_{\min}(J, p). \tag{4.1.5}$$

Dann gilt: $p_{\mathrm{FFD}} \leq p$.

Beweis: Siehe [CGJ78, Kapitel 4 und 5]. □

Allerdings lässt sich dieser Wert für $t_{\max}$ nur bei genauer Kenntnis von $T_{\min}(J, p)$ ausnutzen. Da letzteres im allgemeinen nicht zur Verfügung steht bzw. zu dessen Berechnung eine Lösung des Zuteilungsproblems notwendig wäre, ist dieser direkte Zugang nicht möglich. Anstelle dessen wird eine Binärsuche verwendet, um $t_{\max}$ zu minimieren. Das hierfür benötigte Suchintervall lässt sich wie folgt definieren.

Lemma 4.1.8 *Seien t_ℓ und t_u definiert als*

$$t_\ell = \max\left\{ \frac{1}{p} \sum_{j \in J} t(j)\,,\, \max_{j \in J} t(j) \right\} \quad \text{und} \quad t_u = \max\left\{ \frac{2}{p} \sum_{j \in J} t(j)\,,\, \max_{j \in J} t(j) \right\}. \tag{4.1.6}$$

Dann gilt:

$$t_{\max} < t_\ell \implies p_{\mathrm{FFD}} > p \quad \textit{und} \quad t_{\max} \geq t_u \implies p_{\mathrm{FFD}} \leq p.$$

Beweis: [CGJ78, Beweis zu Lemma 3.1 und 3.2]. □

Der endgültige *Multifit*-Scheduling-Algorithmus ist im folgenden aufgeführt, wobei d die Tiefe der Binärsuche beschreibt.

```
procedure multifit ( J, M, d )
    Setze t_ℓ und t_u entsprechend (4.1.6); i := 0;
    while i < d do
        t_max = (t_ℓ + t_u)/2;
        if ffd( J, t, t_max ) ≤ p then t_u = t_max;
        else                           t_ℓ = t_max;
        i := i + 1;
    endwhile;
    { Zuweisung der Zuteilung }
    for all B_i do
        s := 0;
        for all j ∈ B_i do
            m(j) := M_i; s(j) := s;
            s := s + t(j);
        endfor;
    endfor;
end;
```

Algorithmus 4.1.4: Multifit-Scheduling

Da $t_{\max}$ in diesem Fall lediglich approximiert wird, ist der Wert aus (4.1.5) in Abhängigkeit von d zu modifizieren. Das Resultat in [CGJ78] wurde in [Fri84] nach unten korrigiert und lautet:

Lemma 4.1.9 *Sei Σ_{MF} die mittels Multifit-Scheduling berechnete Zuteilung. Dann gilt:*

$$T(\Sigma_{\mathrm{MF}}) \leq 1.2\, T_{\min} + \frac{1}{2^d}. \tag{4.1.7}$$

Im Vergleich zum LPT-Scheduling lässt sich somit eine effizientere Zuteilung konstruieren. Die Laufzeit des Multifit-Scheduling-Verfahrens ist hierbei ähnlich zur der von Algorithmus 4.1.2 und in der Praxis im wesentlichen durch n bestimmt. Die Suchtiefe wird üblicherweise als konstant angenommen, wobei Werte für d von 5 bis 10 häufig eine hinreichende Genauigkeit garantieren.

4.1.3 Sequenzpartitionierung

Eine Modifikation des allgemeinen Zuteilungsproblems stellt die *Sequenzpartitionierung* dar. Hierbei wird anstelle einer ungeordneten Menge eine Anordnung, oder Sequenz, der einzelnen Aufgaben $j_0, \ldots, j_{n-1}$ betrachtet. Man beachte, dass in diesem Fall die Güte der resultierenden Zuteilung wesentlich von dieser Anordnung abhängt.

Das zu lösende Problem besteht im folgenden darin, diese Sequenz so aufzuteilen, dass die teuerste Teilsequenz die geringsten Kosten verursacht.

Definition 4.1.10 (Sequenzpartitionierung) *Sei $C = \{c_0, c_1, \ldots, c_{n-1}\}$ mit $c_i \in \mathbb{R}_{>0}$ und sei $p \leq n$. Mit $c(i,j) = \sum_{l=i}^{j} c_l$ seien die Kosten der Teilsequenz $c_i, \ldots, c_j$ bezeichnet.*

Desweiteren sei $R = \{r_0, \ldots, r_p\}$ *mit* $0 = r_0 \leq r_1 \leq \ldots \leq r_p = n$, $r_i \in \mathbb{N}, 0 \leq i \leq p$, *eine Menge von Abgrenzungen für die Teilsequenzen von* C. *Dann heißt* R Sequenzpartition *von* (C, p).

R *ist* optimal *bezüglich* (C, p), *falls es keine Partition* $R' = \{r'_0, \ldots, r'_p\}$ *von* (C, p) *gibt, so dass gilt*

$$\max_{0 \leq i < p} c(r'_i, r'_{i+1} - 1) < \max_{0 \leq i < p} c(r_i, r_{i+1} - 1).$$

Eine Partition in p *Teilsequenzen wird auch als* p*-Partition bezeichnet.*

Im folgenden wird für die Teilsequenz $c_i, \ldots, c_j$ auch die Intervallschreibweise $[i, j]$ genutzt.

Anders als beim allgemeinen Scheduling-Problem lässt sich eine optimale Sequenzpartition effizient berechnen. Da die Güte der Zuteilung aber anordnungsabhängig ist, lassen sich in diesem Fall keine allgemeinen Aussagen wie (4.1.2), (4.1.3) oder (4.1.5) formulieren.

Für eine Sequenz $C = \{c_0, c_1, \ldots, c_{n-1}\}$ sei $c_{\max}(i, k)$ der Aufwand für die teuerste Teilsequenz einer optimalen k-Partition des Intervalls $[i, n-1]$ von C. Bei Kenntnis von $c_{\max}(0, p)$ können die einzelnen Teilsequenzen einer optimalen Sequenzpartition bestimmt werden, indem man die Kosten c_i summiert und beim Überschreiten von $c_{\max}(0, p)$ ein neues Intervall beginnt:

```
procedure seq_part( C, c_max(0,p), p )
    s := 0; i := 1; r_0 := 0;
    for 0 ≤ j < n do
        if  s + c_j > c_max(0,p) then
            r_i := j; i := i + 1; s := 0;
        endif;
        s := s + c_j;
    endfor;
    r_{i+1} := ... := r_p := n;
end
```

Algorithmus 4.1.5: Sequenzpartitionierung

Die Lösung des Partitionierungsproblems ist somit im wesentlichen durch die Suche nach $c_{\max}(0, p)$ bestimmt. In [OM95] ist hierfür ein Algorithmus angegeben, mit dem $c_{\max}(0, p)$ effizient berechnet werden kann. Die Grundlage des Verfahrens bildet dabei die Rekursionsformel aus [AF91] für $c_{\max}(i, k)$:

$$c_{\max}(i, k) = \min_{i \leq j \leq n-k} \max \{c(i, j), c_{\max}(j+1, k-1)\}. \tag{4.1.8}$$

Beachtet man, dass für die Partitionierung in ein einzelnes Intervall

$$c_{\max}(i, 1) = c(i, n-1) \tag{4.1.9}$$

gilt und sich die Kosten der teuersten Teilsequenz bei einer maximalen Anzahl von nicht leeren Intervallen durch das maximale Element der Sequenz ergeben:

$$c_{\max}(i, n-i) = \max_{i \leq j < n} c(j, j), \tag{4.1.10}$$

so erlaubt (4.1.8) die Berechnung von $c_{\max}(0,p)$ mit einem Aufwand von $\mathcal{O}\left(n^2 p\right)$.

Die Komplexität der Berechnung kann reduziert werden, indem man die Monotonie von $c(i,j)$ und von $c_{\max}(i,k)$ ausnutzt:

Bemerkung 4.1.11 *Seien $i, i' \in \mathbb{N}$ mit $0 \leq i \leq i' \leq n$. Dann gilt für $1 \leq k \leq p$: $c_{\max}(i,k) \geq c_{\max}(i',k)$.*

Mit (4.1.8) und Bemerkung 4.1.11 lässt sich $c_{\max}(i,k)$ unter gewissen Bedingungen aus dem ersten Intervall $[i+1,j]$ einer optimalen Partition von $[i+1]$ und dem Wert von $c_{\max}(j+1,k-1)$ berechnen:

Lemma 4.1.12 *Sei $[i+1,j]$ das erste Intervall einer optimalen Sequenzpartition in $k>1$ Teilsequenzen. Falls $c(i,j) \leq c_{\max}(j+1,k-1)$, dann folgt:*

$$c_{\max}(i,k) = c_{\max}(j+1,k-1).$$

Beweis: Siehe Beweis zu Lemma 2 in [OM95]. □

Weiterhin existieren charakteristische Punkte, sogenannte *Ausgleichspunkte*, innerhalb einer Partition, die einen wesentlichen Einfluss auf die Kosten der Teilintervalle haben.

Definition 4.1.13 *Seien $i,k \in \mathbb{N}$, mit $0 \leq i < n$ und $2 \leq k \leq p$. Sei $s_{i,k} \in \{i,\ldots,n-1\}$. Falls $c(i,s_{i,k}-1) < c_{\max}(s_{i,k},k-1)$ und $c(i,s_{i,k}) \geq c_{\max}(s_{i,k}+1,k-1)$, dann nennt man $s_{i,k}$ einen* Ausgleichspunkt.

Aus der Monotonie von $c(i,j)$ und $c_{\max}(j,k)$ folgt, dass die Ausgleichspunkte wohldefiniert und eindeutig sind (siehe Lemma 4 in [OM95]). Weiterhin erlauben sie die Berechnung von $c_{\max}(i,k)$ für den Fall, dass sich Lemma 4.1.12 nicht anwenden lässt.

Lemma 4.1.14 *Sei $k \geq 2$. Dann gilt:*

$$c_{\max}(i,k) = \min\left\{c(i,s_{i,k}), c_{\max}(s_{i,k},k-1)\right\}.$$

Beweis: Siehe Beweis zu Theorem 5 in [OM95]. □

Die Kombination dieser beiden Lemmata erlaubt nun, den Wert von $c_{\max}(i,k)$ zu bestimmen. Für die Berechnung werden hierbei das erste Intervall einer optimalen k-Partition von $[i+1,n]$ und $c_{\max}(l,k-1)$ für $i < l \leq j+1$ benötigt. Durch die Verwendung des entsprechenden Intervalls ist somit auch dessen Berechnung während des Algorithmus notwendig.

Falls Lemma 4.1.12 anwendbar ist, d.h. $c(i,j) \leq c_{\max}(j+1,k-1)$, folgt die Position dieses ersten Intervalls direkt aus den Vorbedingungen. Das gleiche gilt für den Wert von $c_{\max}(i,k)$.

Anderenfalls gilt $c(i,j) > c_{\max}(j+1,k-1)$ und Lemma 4.1.14 wird zur Berechnung bemüht. Hierfür muss der entsprechende Ausgleichspunkt $s_{i,k}$ gesucht werden. Nach der Definition von $s_{i,k}$ genügt hierfür eine Suche im Intervall $[i,j]$. Dazu wird j solange verringert, bis $c(i,j-1) < c_{\max}(j,k-1)$ gilt und somit $s_{i,k} = j$ folgt. Die Position des ersten Intervalls in

einer optimalen k-Partition von $[i,n]$ ist $[i,j]$ falls $c(i,j) < c_{\max}(j,k-1)$ bzw. $[i,j-1]$ falls $c(i,j) \geq c_{\max}(j,k-1)$.

```
    procedure comp_c_max( C, p )
2:      for i = n-1, ..., p-1 do c_max(i,1) = c(i,n-1);
3:      for k = 2, ..., p do
4:          c_max(n-k,k) = max{c(n-k,n-k), c_max(n-k+1,k-1)};
            j = n-k;
6:          for i = n-k-1, ..., p-k do
7:              if c(i,j) ≤ c_max(j+1,k-1) then
                    c_max(i,k) = c_max(j+1,k-1);
                else
10:                 if c(i,i) ≥ c_max(i+1,k-1) then
                        c_max(i,k) = c(i,i); j = i;
                    else
13:                     while c(i,j-1) ≥ c_max(j,k-1) do j = j-1;
                        c_max(i,k) = min{c(i,j), c_max(j,k-1)};
                        if c_max(i,k) = c_max(j,k-1) then j = j-1;
                    endif;
                endif;
            endfor;
        endfor;
        return c_max(0,p);
    end;
```

Algorithmus 4.1.6: Berechnung von $c_{\max}(0,p)$

In Algorithmus 4.1.6 sind diese Betrachtungen umgesetzt. Zunächst werden hierbei in Zeile 2 die Startwerte der Rekursion festgelegt, wobei (4.1.9) ausgenutzt wird. Anschließend beginnt die Berechnung der Kosten für eine wachsende Anzahl von Teilsequenzen. Gleichung (4.1.10) wird dabei in Zeile 4 als Rekursionsformel benutzt.

Die Anwendbarkeit von Lemma 4.1.12 wird in Zeile 7 getestet und entsprechend verzweigt. Anderenfalls wird Lemma 4.1.14 genutzt und in Zeile 10 fortgefahren, wo nach dem Ausgleichspunkt gesucht wird. Zunächst wird hier getestet, ob $s_{i,k} = i$ gilt. Dies ist notwendig, da $c_{\max}(i,k-1)$ noch nicht berechnet wurde. Da aber stets $c(i,i) \leq c_{\max}(i,k-1)$ gilt, kann Lemma 4.1.14 auch ohne diesen Wert angewandt werden. Falls $i < s_{i,k}$, wird die Suche in Zeile 13 fortgesetzt.

Für die Analyse des Laufzeitverhaltens von Algorithmus 4.1.6 wird die Prozedur zunächst ohne die Schleife in Zeile 13 betrachtet. Die Initialisierung in Zeile 2 hat eine Komplexität von $\mathcal{O}(n-p)$. Dies gilt auch für die Schleife in Zeile 6. Zusammen mit der Iteration über die verschiedenen Werte von k in Zeile 3 ergibt sich somit eine Laufzeit von

$$\mathcal{O}(p \cdot (n-p)). \tag{4.1.11}$$

Nun zurück zur innersten Schleife in Zeile 13. Die Suche nach $s_{i,k}$ erfolgt nur im Intervall $[i,j]$. Dieses ist aber identisch mit $[p-k, n-k]$. Folglich wird die Bedingung in Zeile 13,

unabhängig von i, höchstens $n - p - 1$ mal erfüllt. Die Laufzeit der „while“-Schleife trägt damit nur additiv und nicht multiplikativ zur Komplexität der „for“-Schleife aus Zeile 6 bei, womit das Laufzeitverhalten insgesamt unverändert bleibt.

Bisher unbeachtet blieb die Berechnung von $c(i,j)$. Um die Komplexität von $\mathcal{O}(p(n-p))$ von Algorithmus 4.1.6 zu erreichen, dürfen die Kosten hierfür nur $\mathcal{O}(1)$ betragen. Dazu die folgende Bemerkung:

Bemerkung 4.1.15 *Sei $\mathcal{C}_i = c(0,i)$. Der Wert von $c(i,j)$ ergibt sich durch $c(i,j) = \mathcal{C}_j - \mathcal{C}_i + c_i$. Durch eine Speicherung der n Werte $\mathcal{C}_i$* vor *dem Aufruf von Algorithmus 4.1.6 kann die Berechnung somit in $\mathcal{O}(1)$ erfolgen.*

Ein anderer, approximativer Algorithmus zur Berechnung von $c_{\max}(0,p)$ ist in [Mr95] beschrieben. Die Grundlage für dieses alternative Verfahren bildet eine Bisektion. Durch

$$\left[\max_{0 \le i < n} c_i \, , \, c(0, n-1) \right] \tag{4.1.12}$$

ist hierbei das Intervall für die Suche definiert.

```
procedure seq_part_bisection( C, p, ε )
    c_u = c(0, n-1); c_l = max_{i=0}^{n-1} c_i;
    do
        c_m = (c_u + c_l)/2;
        seq_part( C, c_m, p ); { Algorithmus 4.1.5 }
        if max_{i=0}^{p-1} c(r_i, r_{i+1} - 1) ≤ c_m  then c_u = c_m;
        else                                          c_l = c_m;
    while c_l + ε ≥ c_u
end;
```

Algorithmus 4.1.7: Sequenzpartitionierung mittels Bisektion

Das jeweils nächste Suchintervall in Algorithmus 4.1.7 ist dadurch bestimmt, ob die Sequenzpartitionierung bezüglich c_m erfolgreich war, also p Teilsequenzen mit maximalen Kosten von c_m gefunden wurden. Waren die maximalen Kosten pro Teilintervall zu niedrig angesetzt, ist das letzte Intervall zu teuer. Bei zu hohen Kosten kann diese Teilsequenz leer bleiben ($r_{p-1} = r_p = n$).

Abgebrochen wird Algorithmus 4.1.7, falls der Abstand der untere Schranke zur oberen Schranke ein vorgegebenes $\varepsilon \in \mathbb{R}$ unterschreitet.

Die Laufzeit des Bisektionsalgorithmus ist unabhängig von der Anzahl der Teilsequenzen, dafür aber abhängig von den Kosten. Für $c_u = c(0,n-1)$ und $c_l = \max_{i=0}^{n-1} c_i$ beträgt die maximale Anzahl von Iterationen $\mathcal{O}(\log((c_u - c_l)/\varepsilon))$. Zusammen mit dem Aufwand für die Bestimmung einer p-Partition in Algorithmus 4.1.5 ergibt sich eine Gesamtkomplexität von:

$$\mathcal{O}\left(n \cdot \log\left(\frac{c_u - c_l}{\varepsilon} \right) \right). \tag{4.1.13}$$

Bemerkung 4.1.16 *Für den Fall von ganzzahligen Kosten $c_i \in \mathbb{N}$ berechnet Algorithmus 4.1.7 eine Lösung für das Sequenzpartitionierungsproblem in der Zeit*

$$\mathcal{O}\left(n \cdot \log(c_u - c_l)\right),$$

wobei für die Abbruchbedingung $\varepsilon = 1$ zu wählen ist.

4.2 Lastbalancierung im Blockclusterbaum

Bei vielen Verfahren im Kontext der $\mathcal{H}$-Matrizen sind die auftretenden Kosten mit den Knoten des Clusterbaumes bzw. Blockclusterbaumes verbunden, etwa bei der Matrix-Vektor-Multiplikation (siehe Abschnitt 3.1). Bisher wurden aber nur allgemeine Aufgaben mit Hilfe der verschiedenen Zuteilungsalgorithmen auf die p Prozessoren verteilt. In diesem Abschnitt sollen deshalb notwendige Anpassungen der Verteilungsverfahren beschrieben werden, die sich aus der Baumdarstellung ergeben oder aber für Methoden über Bäumen von Vorteil sind.

Eine wesentliche Eigenschaft bisheriger Verteilungsfunktionen war die explizite Zuweisung einer Aufgabe zu einem Prozessor. Bei der Verteilung eines Cluster- bzw. Blockclusterbaumes entspricht dies der Kopplung von Knoten und Maschinen. Im folgenden wird dies aus Gründen der Einfachheit, aber auch der Effizienz modifiziert, wobei es auch möglich sein soll, einen Knoten mit *allen* Prozessoren zu assoziieren. Für diesen Fall wird das Symbol $\perp$ genutzt. Allerdings ist diese erweiterte Kopplung beschränkt auf innere Knoten eines Baumes.

Definition 4.2.1 (Zulässige Verteilungsfunktion) *Eine Verteilungsfunktion $m : T \to \{\perp, 0, \ldots, p-1\}$ über einem Baum T heißt* zulässig*, falls für alle Blätter $v \in \mathcal{L}(T)$ gilt: $m(v) \neq \perp$.*

Die Bedingung der Zulässigkeit einer Verteilungsfunktion ist notwendig, damit die Arbeit, die üblicherweise an die Blätter gebunden ist, genau einem Prozessor zugewiesen wird.

In den folgenden Abschnitten wird zunächst auf die Anwendung der Scheduling-Algorithmen zur Verteilung der Blätter im Blockclusterbaum eingegangen. Anschließend werden Verfahren vorgestellt, die die Lokalität der einem Prozessor zugeordneten Knotenmenge erhöhen. Hierdurch sollen ungünstige Verteilungen bei parallelen Algorithmen, etwa der Matrix-Vektor-Multiplikation in Abschnitt 6.3, verhindert werden. Abschließend wird die Frage untersucht, wie die optimale Ausführungsgeschwindigkeit der parallelen Algorithmen im Kontext der $\mathcal{H}$-Matrizen von der Zahl der Prozessoren abhängt. In diesem Zusammenhang werden einige Eigenschaften der Verteilungsverfahren im praktischen Einsatz untersucht.

4.2.1 Lastbalancierung mittels List- und Multifit-Scheduling

Für die Verwendung des List-, LPT- und Multifit-Schedulings zur Lastbalancierung innerhalb des Blockclusterbaumes $T = T(I \times J)$ mit den Indexmengen I und J genügt es, die Knoten $v \in T$ mit Kosten zu assoziieren. Anschließend können die Algorithmen 4.1.1, 4.1.2 und 4.1.4 direkt aufgerufen werden, um eine entsprechende Verteilung zu berechnen.

In Abbildung 4.2.1 wurden die Blätter eines Blockclusterbaumes aus dem Beispiel in Abschnitt 2.4.2 mittels LPT-Scheduling auf unterschiedlich viele Prozessoren verteilt, wobei die dem ersten Prozessor zugewiesene Knotenmenge markiert ist.

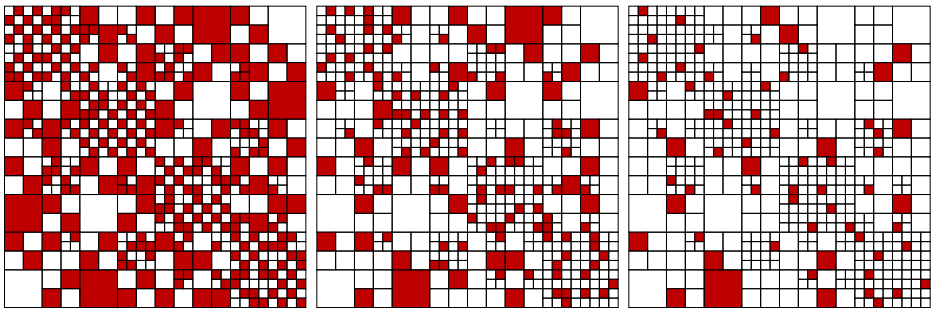


Abbildung 4.2.1: Knotenmenge eines Prozessors durch LPT-Scheduling mit 2 (links), 4 (mitte) und 8 (rechts) Prozessoren

Man erkennt deutlich, dass die prozessorlokalen Knoten im Blockclusterbaum nicht notwendigerweise benachbart sind. Diese fehlende „Kompaktheit“ oder Datenlokalität beim List-/LPT- bzw. Multifit-Scheduling ergibt sich aus der im wesentlichen zufälligen Verteilung der einzelnen Knoten auf die Prozessoren (siehe Abschnitt 4.1.1). Diese Eigenschaft kann unter Umständen zu einem ineffizienten Verhalten bei parallelen Algorithmen führen (siehe Abschnitt 6.3).

In Abschnitt 4.2.3 werden Algorithmen vorgestellt, die dem entgegenwirken und die Datenlokalität beim List-/LPT- und Multifit-Scheduling erhöhen.

4.2.2 Lastbalancierung mittels Sequenzpartitionierung

Notwendig für die Berechnung einer Sequenzpartitionierung war die Vorgabe einer Anordnung der einzelnen Aufgaben.

Betrachtet wird zunächst das Problem der Anordnung der Blätter eines Blockclusterbaumes. Sei $T(I)$ ein binärer Clusterbaum über der Indexmenge I. Hierbei wird für alle Knoten von $T(I)$ vorausgesetzt, dass die enthaltenen Indizes fortlaufend nummeriert sind. Durch die binäre Struktur von $T(I)$ ist ein hierüber definierter Blockclusterbaum $T(I \times I)$ ein Quadbaum. Die Söhne der Knoten $v \in V(T(I \times I)) \setminus \mathcal{L}(T(I \times I))$ seien $v_0, \ldots, v_3$ und wie folgt angeordnet:

v_0	v_1
v_2	v_3

(4.2.1)

Eine rekursive Fortsetzung dieser Anordnung auf die Söhne von $v_0, \ldots, v_3$ ergibt die Nummerierung aus Abbildung 4.2.2. Ordnet man schließlich die Blätter von $T(I \times I)$ lexikographisch entsprechend ihrer Nummerierung an, ergibt sich eine charakteristische Kurve, die sogenannte *Z-Kurve*, wie sie in Abbildung 4.2.3 (oben) dargestellt ist.

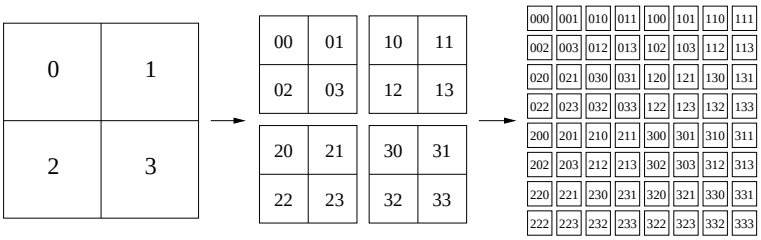


Abbildung 4.2.2: Rekursive Anordnung

Die Z-Kurve stellt eine *raumfüllende Kurve* dar. Ursprünglich wurden diese Kurven als Abbildungen von $[0, 1]$ nach $[0, 1]^2$ entworfen, wobei Peano (siehe [Pea90]) ein erstes Beispiel hierfür angab. Weitere Abbildungen stammen von Hilbert, Moore und Lebesgue (siehe auch [Sag94]). Ihren Namen verdanken die raumfüllenden Kurven der Eigenschaft, dass sie im Grenzfall jeden Punkt von $[0, 1]^2$ durchlaufen und somit die Fläche (bzw. Raum) *ausfüllen*. Die Anwendbarkeit der Kurven für die Lastbalancierung wurde zunächst bei der Lösung des N-Körper-Problems in der Physik demonstriert (siehe [WS93]).

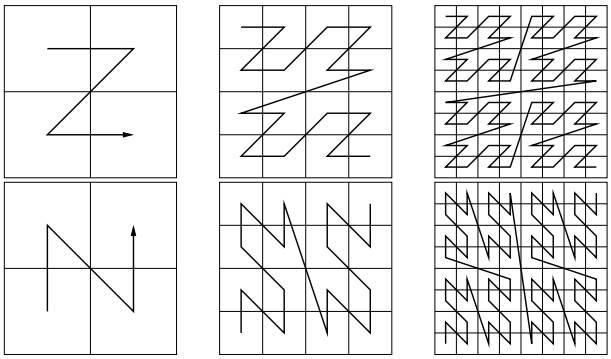


Abbildung 4.2.3: Z-Kurve (oben) und Lebesgue-Kurve (unten)

Analog zur Z-Kurve lassen sich auch alle anderen raumfüllenden Kurven verwenden, um eine Anordnung der Blätter des Blockclusterbaumes zu definieren. Im folgenden wird dies

für die Hilbert-, die Moore- und die Lebesgue-Kurve beschrieben. Die Peanokurve basiert auf einer 3×3 Zerlegung von $[0,1]^2$ und wird deshalb nicht betrachtet.

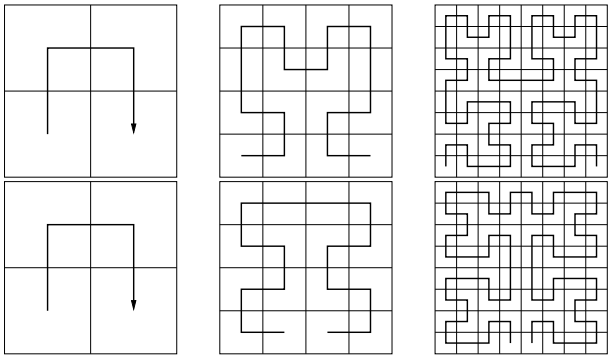


Abbildung 4.2.4: Hilbert-Kurve (oben) und Moore-Kurve (unten)

Die Definition der Anordnung der Blätter in T erfolgt konstruktiv, wobei das zugrundeliegende Verfahren eine Tiefensuche in T darstellt. Der wesentliche Unterschied zwischen den einzelnen Kurven betrifft die Reihenfolge, in der dabei die Söhne eines Knotens durchlaufen werden.

Im Anordnungsalgorithmus wird jedem $v \in T$ ein Typ $t(v) \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}\} = \mathcal{T}$ zugewiesen, wobei für die Wurzel stets $t(\text{root}(T)) = \mathcal{A}$ gilt. Während der Tiefensuche definiert der jeweilige Knotentyp neben der Reihenfolge der Söhne von v auch deren Typ. Sei durch $r : V(T) \times \mathcal{T} \to (V(T) \times \mathcal{T})^*$ mit $r(v, t(v)) = (v_i, t(v_i))_{i=0}^{m-1}, m = |\mathcal{S}(v)|$ eine solche Anordnung definiert. Dann lässt sich die Reihenfolge der Blätter in T rekursiv berechnen durch:

```
procedure order( v, t, S )
    for i = 0, ..., |S(v)| - 1 do
        (v', t(v')) := r(v, t)_i;
        S := (S, v');
        order( v', t(v'), S );
    endfor;
end;
```

Algorithmus 4.2.1: Anordnung von $\mathcal{L}(T)$ entsprechend raumfüllender Kurve

Hierbei bezeichne (S, v') das Anfügen des Knotens v' an die *Liste* S.

Für einen Quadbaum ergeben sich die im folgenden beschriebenen Anordnungen für die einzelnen Kurven, wobei stets für einen Knoten $v \in T$ von einer Anordnung der Söhne $v_0, \ldots, v_3 \in \mathcal{S}(v)$ wie in (4.2.1) ausgegangen wird.

Der einfachste und bereits beschriebene Fall ist die Z-Kurve:

$$r_{\text{Z}}(v, \mathcal{A}) \quad = \quad ((v_0, \mathcal{A}), (v_1, \mathcal{A}), (v_2, \mathcal{A}), (v_3, \mathcal{A})).$$

Da die Z-Kurve lediglich eine Variante der Lebesgue-Kurve darstellt, lässt sich letztere durch eine modifizierte Anordnung konstruieren. Die Markierungen bleiben dabei erhalten:

$$r_{\text{Lebesgue}}(v, \mathcal{A}) \quad = \quad ((v_2, \mathcal{A}), (v_0, \mathcal{A}), (v_3, \mathcal{A}), (v_1, \mathcal{A})).$$

Die Definitionen der Hilbert- bzw. Moore-Kurve sind komplizierter, da die Kurven in den einzelnen Teilgebieten von $[0,1]^2$ ein unterschiedliches Aussehen haben. Die Hilbert-Kurve erhält man mit

$$r_{\text{Hilbert}}(v, t) \quad = \quad \begin{cases} ((v_2, \mathcal{B}), (v_0, \mathcal{A}), (v_1, \mathcal{A}), (v_3, \mathcal{C})), & t = \mathcal{A} \\ ((v_2, \mathcal{A}), (v_3, \mathcal{B}), (v_1, \mathcal{B}), (v_0, \mathcal{D})), & t = \mathcal{B} \\ ((v_1, \mathcal{D}), (v_0, \mathcal{C}), (v_2, \mathcal{C}), (v_3, \mathcal{A})), & t = \mathcal{C} \\ ((v_1, \mathcal{C}), (v_3, \mathcal{D}), (v_2, \mathcal{D}), (v_0, \mathcal{B})), & t = \mathcal{D} \end{cases}.$$

Die Moore-Kurve lässt sich schließlich mit

$$r_{\text{Moore}}(v, t) \quad = \quad \begin{cases} ((v_2, \mathcal{B}), (v_0, \mathcal{B}), (v_1, \mathcal{C}), (v_3, \mathcal{C})), & t = \mathcal{A} \\ ((v_3, \mathcal{E}), (v_2, \mathcal{B}), (v_0, \mathcal{B}), (v_1, \mathcal{D})), & t = \mathcal{B} \\ ((v_0, \mathcal{D}), (v_1, \mathcal{C}), (v_3, \mathcal{C}), (v_2, \mathcal{E})), & t = \mathcal{C} \\ ((v_0, \mathcal{C}), (v_2, \mathcal{D}), (v_3, \mathcal{D}), (v_1, \mathcal{B})), & t = \mathcal{D} \\ ((v_3, \mathcal{B}), (v_1, \mathcal{E}), (v_0, \mathcal{E}), (v_2, \mathcal{C})), & t = \mathcal{E} \end{cases}$$

erzeugen.

In Abbildung 4.2.5 sind Beispiele für die Sequenzpartitionierung mit den vorgestellten Kurven aufgeführt. Hierbei wurden die Blätter von $T(I \times J)$ jeweils auf 8 Prozessoren verteilt. Die Kosten entsprechen dem Aufwand der Matrix-Vektor-Multiplikation (siehe Abschnitt 6.3).

Im Gegensatz zu den Verteilungen mittels List- bzw. LPT-Scheduling sind die Knoten auf jedem Prozessor im Blockclusterbaum fast immer benachbart. Dies ergibt sich durch die Nachbarschaftsbeziehung aneinander grenzender Teilintervalle der raumfüllenden Kurven und dem Sequenzpartitionierungsalgorithmus. Lediglich bei der Z- und bei der Lebesgue-Kurve kann es zu „Sprüngen“ kommen. Diese sind bedingt durch den Verlauf der Kurve (siehe Abbildung 4.2.3).

4.2.3 Lokalität der prozessorlokalen Knotenmenge

Die bereits im vorherigen Abschnitt beobachtete Nachbarschaftsbeziehung von prozessorlokalen Matrixblöcken ist bei vielen der in Kapitel 6 beschriebenen Algorithmen wünschenswert. Zum einen kann sich hierdurch eine geringere Komplexität der Verfahren ergeben (siehe etwa Abschnitt 6.3) und zum anderen kann die höhere Lokalität in der Praxis zu Effizienzvorteilen führen, da im Blockclusterbaum benachbarte Matrizen typischerweise auch im Arbeitsspeicher aneinander angrenzen. Diese Eigenschaft wird in vielen Prozessoren oder Programmen ausgenutzt und senkt damit die Ausführungszeit der Algorithmen.

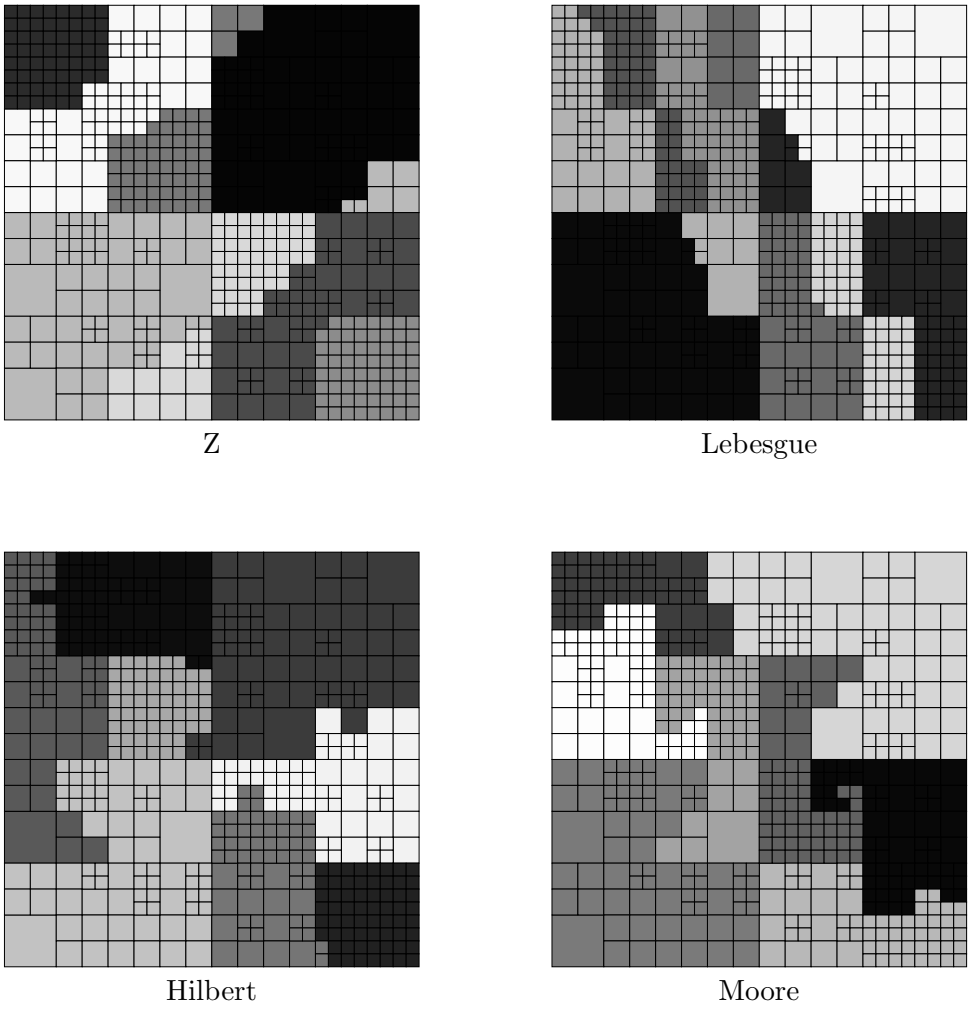


Abbildung 4.2.5: Sequenzpartitionierung mittels raumfüllender Kurven

Eine in diesem Zusammenhang sehr ungünstige Verteilung wird durch das List-/LPT- und Multifit-Scheduling berechnet, welche praktisch keine Lokalität aufweisen. Die Ursache hierfür liegt darin, dass innerhalb der Zuteilungsalgorithmen nur die Kosten, nicht aber die Position der Blöcke betrachtet wird. Das Ziel ist also, die Wahlmöglichkeiten einzuschränken ohne dabei die Güte der Zuteilung zu verschlechtern.

Als Motivation für die folgenden Algorithmen sei das folgende Beispiel betrachtet: $T(I \times I)$ sei ein vollständiger Quadbaum und $c : V(T) \to \mathbb{R}$ eine Kostenfunktion mit $c(v) = 1$ für alle $v \in \mathcal{L}(T)$. Für innere Knoten von T seien die Kosten definiert als die Summe der Kosten der Söhne:

$$c(v) = \begin{cases} 1 & , \quad v \in \mathcal{L}(T) \\ 1 + \sum_{w \in \mathcal{S}(v)} c(w) & , \quad \text{sonst} \end{cases}.$$

Damit gilt für alle Söhne $v_0, \ldots, v_3$ von $\text{root}(T)$: $c(v_0) = c(v_1) = c(v_2) = c(v_3)$. Ein optimales Scheduling für 4 Prozessoren besteht somit darin, die Knoten v_i auf jeweils einen Prozessor $0 \leq i \leq 3$ zu verteilen.

Anstatt die Blätter von $T(I \times I)$ zu betrachten, genügt es also, lediglich die Söhne von $\text{root}(T)$ in die Zuteilung einzubeziehen. Die Lastbalancierung kann somit auch direkt auf inneren Knoten aufsetzen. Da die kompletten Teilbäume mit den Wurzeln $v_0, \ldots, v_3$ jeweils auf einen Prozessor verteilt wurden, ergibt sich eine große Lokalität aller prozessorlokalen Matrizen.

In diesem Beispiel wurden die Knoten in $T^{(1)}$ betrachtet, um $\mathcal{L}(T)$ auf 4 CPUs zu verteilen. Für eine Verteilung für $p = 16$ genügt entsprechend ein Scheduling der Knoten in $T^{(2)}$, für $p = 64$ die Knoten in $T^{(3)}$ usw.

Dieses stufenweise Vorgehen lässt sich für beliebige Blockclusterbäume T und Prozessorzahlen p verallgemeinern. Der Start hierzu erfolgt stets auf der Stufe 0 von T. In jedem Schritt i des Algorithmus werden die Knoten in $T^{(i)}$ und die Blätter auf den Stufen $1 \leq j < i$ betrachtet. Ist die Anzahl der zu verteilenden Knoten mindestens so groß wie die Zahl der Prozessoren, wird ein Scheduling für die Knotenmenge berechnet. Hierfür können die Algorithmen aus den Abschnitten 4.1.1, 4.1.2 und 4.1.3 eingesetzt werden. Die Iteration wird solange fortgesetzt, bis die betrachtete Knotenmenge der Blattmenge von T entspricht oder eine hinreichende Lastbalancierung erreicht wurde. Hierzu lassen sich z.B. die kleinsten und die größten Kosten pro Prozessor, die die Zuteilung erzeugt, vergleichen:

$$C_{\min} := \min_{0 \leq q < p} \sum_{v \in V_q} c(v) \quad \text{und} \quad C_{\max} := \max_{0 \leq q < p} \sum_{v \in V_q} c(v)$$

mit $V_q = \{v \in T \mid m(v) = q\}$. Ist der relative Unterschied

$$1 - \frac{C_{\min}}{C_{\max}} \leq \varepsilon, \tag{4.2.2}$$

mit $\varepsilon \geq 0$, zwischen beiden Werte gering genug, wird der Algorithmus abgebrochen, anderenfalls die Iteration fortgesetzt. Das vollständige Verfahren ist in Algorithmus 4.2.2 aufgeführt.

Hierbei steht die Funktion „schedule“ stellvertretend für die Verteilungsalgorithmen aus Abschnitt 4.1.

```
procedure distribute_lvl()
    i = 0; S = root(T);
    while  i ≤ p(T)  do
        if  |S| ≥ p  then
            schedule( S );
            if (4.2.2) erfüllt then return ;
        endif;
        i = i + 1;
        S = T^(i) ∪ ⋃_{j=0}^{i-1} L(T, i);
    endwhile;
end;
```

Algorithmus 4.2.2: Stufenweise Lastbalancierung

Wann das Abbruchkriterium erreicht wird, hängt sehr stark der Anzahl der Prozessoren ab. Insbesondere bei einem großen p und einer niedrigen Stufe von T, d.h. einer kleinen Anzahl von zu verteilenden Knoten, wird üblicherweise nur ein ineffizientes Scheduling berechnet. Auch der Wert von ε spielt eine entscheidende Rolle. In den praktischen Experimenten wurde hierfür typischerweise ein Wert von 5 – 10% gewählt. Eine wesentlich kleinere Schranke ist in der Praxis nur selten sinnvoll, da häufig auch Parameter wie eine ungenaue Kostenfunktion (siehe Abschnitt 4.3.2) oder der Einfluss des Computersystems (siehe Kapitel 5) die Güte der Zuteilung bestimmen.

In Abbildung 4.2.6 ist ein Beispiel für die Anwendung von direktem LPT-Scheduling im Vergleich zum stufenweisem LPT-Scheduling dargestellt. Deutlich erkennbar ist die größere Lokalität der Blockcluster.

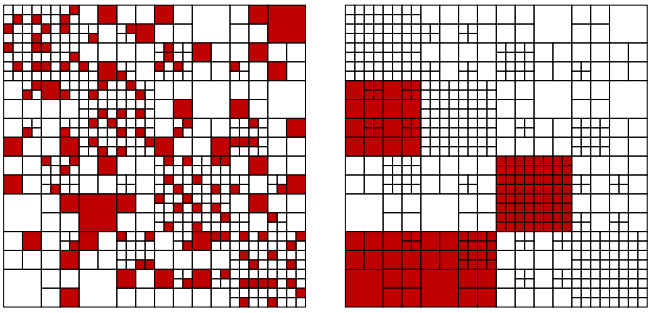


Abbildung 4.2.6: LPT-Scheduling und stufenweises LPT-Scheduling für $p = 4$

Bemerkung 4.2.2 *Im Iterationsschritt $p(T)$ von Algorithmus 4.2.2 gilt für die Menge S: $S = \mathcal{L}(T)$. Das erzeugte Scheduling ist somit identisch zu der Verteilung, die vom zugrundeliegenden Scheduling-Algorithmus berechnet wird. Dieser Fall tritt auch ein, falls ε in* (4.2.2)

zu klein gewählt wurde und somit auf den kleineren Stufen keine hinreichend gute Zuteilung berechnet werden konnte.

Bemerkung 4.2.3 *Nach Lemma 2.1.5 ist die Zahl der inneren Knoten durch die Anzahl der Blätter im Blockclusterbaum beschränkt. Somit ist auch der Aufwand von Algorithmus 4.2.2 bestimmt durch die Komplexität der Berechnung einer Verteilung der Blätter, d.h.* $\mathcal{O}\left(|\mathcal{L}(T)|\right)$. *In der Praxis ist die Ausführungszeit typischerweise geringer, da oft bereits auf kleinen Stufen ein hinreichend gutes Scheduling gefunden wird.*

Ein möglicher Nachteil von Algorithmus 4.2.2 besteht darin, dass keine Unterscheidung zwischen unterschiedlich teuren Knoten erfolgt, sondern während der Iteration stets alle inneren Knoten durch ihre Söhne ersetzt werden. In Algorithmus 4.2.3 wird demgegenüber ein adaptives Verfahren eingesetzt, bei dem lediglich der Knoten mit den größten Kosten betrachtet wird.

```
procedure distribute_ada()
    S = root(T);
    while S ≠ L(T) do
        if |S| ≥ p then
            schedule( S );
            if (4.2.2) erfüllt then return ;
        endif;
        Sei τ ∈ S \ L(T) mit c̃(τ) = max_{σ∈S\L(T)} c̃(σ);
        S = (S ∪ S(τ)) \ {τ};
    endwhile;
end;
```

Algorithmus 4.2.3: Adaptive Lastbalancierung

Die Anzahl der Iterationen in Algorithmus 4.2.3 ist allerdings durch die Zahl der Knoten bzw. Blätter in T bestimmt. Für die Komplexität ergibt sich damit $\mathcal{O}\left(|\mathcal{L}(T)|^2\right)$, womit das Verfahren im allgemeinen ungeeignet ist. Auch zeigten sich in praktischen Experimenten keine Vorteile einer solchen adaptiven Verteilung.

Wie bereits angedeutet erzeugen sowohl die stufenweise, als auch die adaptive hierarchische Lastbalancierung eine Verteilungsfunktion m, die jeweils komplette Unterbäume einem Prozessor zuweisen.

Definition 4.2.4 (Konsistente Verteilungsfunktion) *Eine Verteilungsfunktion* m *über einem Baum* T *heißt* konsistent, *falls für alle* $v, v' \in T$ *gilt:*

$$\left(m(v) \neq \perp \ \wedge \ v \xrightarrow{*} v'\right) \implies \left(m(v') = m(v)\right).$$

Eine konsistente Verteilungsfunktion führt somit auch zu der gewünschten Lokalität der prozessorlokalen Blockcluster. Diese Eigenschaft hat außerdem einen wesentlichen Einfluss auf die Arbeitsweise und die Komplexität vieler der parallelen Algorithmen in Kapitel 6.

4.3 Güte der Zuteilung

Gegenstand dieses Abschnittes sind zwei Probleme im Zusammenhang mit den betrachteten Lastbalancierungsverfahren. Zunächst wird die Frage untersucht, ob die Ausführungszeit $T_{\min}$ einer optimalen Zuteilung, welche die Grundlage in den bisherigen Abschätzungen bildete, bei $\mathcal{H}$-Matrix-Algorithmen von der Ordnung $\mathcal{O}(1/p)$ ist. Desweiteren soll auf das Problem von ungenauen Kostenfunktionen in realen Anwendungen der Zuteilungsverfahren eingegangen werden.

4.3.1 Optimale Zuteilung und optimale Komplexität

Die Abschätzungen (4.1.2) und (4.1.3) für das List- bzw. LPT-Scheduling bzw. (4.1.7) für das Multifit-Scheduling beziehen sich stets auf die Zeit, die minimal von einem parallelen Algorithmus benötigt wird, um die gegebene Menge von Aufgaben abzuarbeiten. In den späteren Komplexitätsabschätzungen ist es hierbei aber wichtig, dass diese Zeit von der Größenordnung $T_{\min}(J,1)/p$ ist, um die gewünschten Aussagen über die die parallele Laufzeit zu gewinnen.

Leider ist dieses Ergebnis im allgemeinen nicht möglich. Zum Beispiel ist bei gegebenen $n-1$ Aufgaben mit jeweils Kosten von $c=1$ und einer Aufgabe mit Kosten von $c=n' \gg n$, die Ausführungszeit stets von diesem teuren Job bestimmt, und zwar unabhängig von der Zahl der Prozessoren.

Das gleiche Resultat ergibt sich auch für eine Verteilung mittels Sequenzpartitionierung. Eine optimale Zerlegung in Teilsequenzen garantiert auch in diesem Fall keine optimale parallele Komplexität.

Allerdings variieren die Kosten im Kontext der $\mathcal{H}$-Matrizen nicht willkürlich, sondern sind durch die hierarchische Zerlegung bestimmt. Betrachtet man die Verteilung der zulässigen Knoten in einem typischen Blockclusterbaum, so nimmt der Aufwand pro Knoten typischerweise mit steigender Stufe im Blockclusterbaum um einen festen Faktor ab. Um den gleichen oder einen ähnlichen Faktor steigt dagegen die Anzahl der Aufgaben pro Stufe. In Abbildung 4.3.1 ist dies graphisch dargestellt. Ein entsprechendes Modellproblem soll im folgenden genutzt werden, um zu untersuchen, ob die Annahme

$$T_{\min}(J,p) = \mathcal{O}\left(\frac{T_{\min}(J,1)}{p}\right) \tag{4.3.1}$$

im Kontext der $\mathcal{H}$-Matrizen als erfüllt gelten kann.

Sei $L \geq 0$ und $n = 2^L$. Für $i \leq L$ sei J_i eine Menge von Aufgaben mit $|J_i| = 2^i$ und Kosten von $c(j) = 2^{L-i}$ für alle $j \in J_i$. Die Menge aller Aufgaben sei $J = \cup_{0 \leq i \leq L} J_i$. Man beachte, dass die Arbeit pro Menge J_i konstant ist: $c(J_i) = \sum_{j \in J_i} c(j) = n$ für alle $0 \leq i \leq L$. Weiterhin sei für die Prozessorzahl $p = 2^m$ mit $m \geq 0$ angenommen.

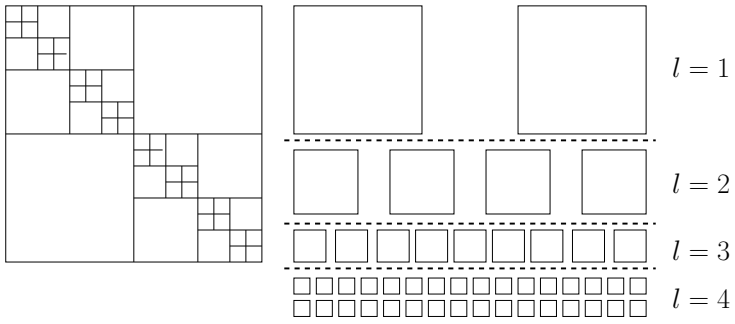


Abbildung 4.3.1: $\mathcal{H}$-Matrix und Matrizen bei der Matrix-Vektor-Multiplikation

Lemma 4.3.1 *Es gelte $t(i) = c(i)$. Die minimal notwendige Zeit zur Ausführung der Aufgaben in J auf einem parallelen Rechner lautet dann:*

$$T_{\min} = \max \left\{ \max_{0 \le i < n} t(i)\,,\ \frac{\sum_{i=0}^{n-1} t(i)}{p} \right\}. \tag{4.3.2}$$

Beweis: Zuerst der Fall $n \le 2^p$. Es gilt $L \le p$. Da der Aufwand pro Level n beträgt, lassen sich die $L \le p$ Mengen J_i auf L Prozessoren aufteilen, wobei $T(\Sigma) = n$ ist. Da J_0 mit einem Element bereits Aufwand n besitzt, ist auch keine kürzere Ausführungszeit möglich.

Für $n > 2^p$ ergibt sich aus der vorherigen Betrachtung, dass sich die Mengen J_i, $i \le p$ optimal auf die p Prozessoren verteilen lassen. Für die Mengen $J_{p+1}, \ldots, J_L$ gilt: $|J_i|$ ist ein ganzzahliges Vielfaches von p. Somit können diese Mengen jeweils optimal auf alle Prozessoren aufgeteilt werden. Insgesamt folgt somit die Behauptung. □

Für ein hinreichend großes n bzw. kleines p liefern das List-, das LPT- und das Multifit-Scheduling somit effiziente Lastbalancierungen. Allerdings ist die Bedingung $n \ge 2^p$ in der Praxis nur für kleine Prozessorzahlen realistisch. Während z.B. ein System mit 16 Prozessoren eine minimale Problemgröße von $n = 65536$ erfordert, führt ein Rechnersystem mit 128 Prozessoren bereits zu einem n in der Größenordnung von 10^{38}.

Bemerkung 4.3.2 *Die Verteilung im Beweis zu Lemma 4.3.1 erfolgt analog zur Vorgehensweise beim Multifit-Scheduling. Somit entspricht auch* (4.3.2) *der Zeit, die durch eine Lastbalancierung mittels Multifit-Scheduling benötigt wird.*

In dem Modellproblem wurde allerdings eine Eigenschaft der $\mathcal{H}$-Matrizen nicht berücksichtigt, die diese Schwierigkeit beseitigt. Es wurde angenommen, dass auf der Stufe 0 bereits Arbeit geleistet werden muss. Vergleicht man aber z.B. die Kosten der Matrix-Vektor-Multiplikation (siehe Abschnitt 3.1) für das Beispiel der $\mathcal{H}$-Matrix in Abbildung 4.3.1, so stellt man fest, dass die ersten zu berechnenden Probleme erst auf der Stufe $L_{\min} = 1$ auftreten. Der hierbei dargestellte Fall entspricht einer $\mathcal{H}$-Matrix zu einem 1-dimensionalen

Problem mit *schwacher Zulässigkeitsbedingung* (siehe [HKK03]). Bei üblicher Zulässigkeitsbedingung, wie sie auch in den Beispielen in Abschnitt 2.4.1 und 2.4.2 verwendet wurden, gilt sogar $L_{\min} = 2$ oder $L_{\min} = 3$.

Die Mengen J_i, $i \geq L_{\min}$ lassen sich somit in $p' = p/2^{L_{\min}}$ disjunkte Teilmengen J_i^q, $0 \leq q < p$ mit $|J_i^q| = \frac{1}{p'}|J_i|$ aufteilen (siehe Abbildung 4.3.2). Die hierdurch entstehenden Teilprobleme sind identisch mit dem reduzierten Ausgangsproblem für $n = n' = 2^{p'}$ und $p = p'$. Für das obige Beispiel mit $p = 128$ wird bei einem $L_{\min} = 3$ lediglich ein $n = 8 \cdot 2^{128/8} = 524\,288$ benötigt, um eine optimale Lastbalancierung zu erhalten.

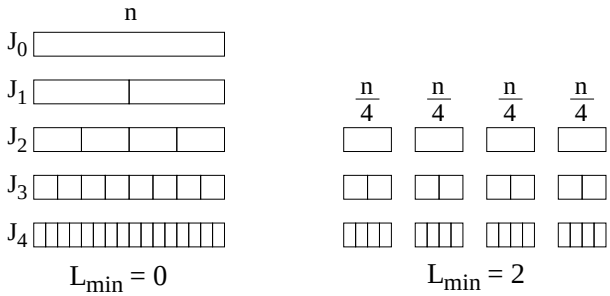


Abbildung 4.3.2: Modellproblem mit $L_{\min} = 0$ und $L_{\min} = 2$

Weiterhin nimmt die Schwachbesetztheitskonstante c_{sp} auf tieferen Stufen häufig größere Werte an als auf den oberen Stufen des Blockclusterbaumes. Dies führt im Modellproblem zu einem höheren Anteil von „kleinen“ Aufgaben, womit die Dominanz der großen Blöcke reduziert wird.

Wird außerdem die stufenweise oder adaptive Lastbalancierung (Algorithmus 4.2.2 bzw. 4.2.3) verwendet und eine Verteilung berechnet, die Kriterium (4.2.2) mit einem hinreichend kleinen ε erfüllt, ist die so ermittelte Zuteilung bis auf den Faktor $1 + \varepsilon$ optimal und ein darauf aufbauender paralleler Algorithmus somit effizient.

Diese Diskussion gibt somit Anlass zu der folgenden Vermutung, deren Gültigkeit in den nachfolgenden Kapiteln vorausgesetzt wird.

Vermutung 4.3.3 *Sei $\mathfrak{A}$ ein Algorithmus über einer $\mathcal{H}$-Matrix A. Die sequentielle Komplexität dieses Algorithmus sei mit $\mathcal{W}_{\mathfrak{A}}$ bezeichnet. Desweiteren lasse sich $\mathfrak{A}$ in eine Menge J voneinander unabhängiger Aufgaben mit $|J| \gg p$ zerlegen. Dann beträgt die Komplexität eines optimalen, zu $\mathfrak{A}$ äquivalenten, parallelen Algorithmus*

$$\mathcal{O}\left(\frac{\mathcal{W}_{\mathfrak{A}}}{p}\right). \tag{4.3.3}$$

Beispiele für solche Algorithmen sind der Aufbau, die Addition sowie Multiplikation von $\mathcal{H}$-Matrizen. Nicht dazu gehören dagegen die Matrix-Vektor-Multiplikation in der blockorientierten Form aus Abschnitt 3.1 (siehe Abschnitt 6.3.1) und die Matrix-Inversion mittels Gauß-Elimination (siehe Abschnitt 6.6.1).

4.3.2 Verhalten bei abweichenden Kosten

Bisher wurde bei der Lastverteilung stets davon ausgegangen, dass die verwendete Kostenfunktion exakt dem realen Aufwand entspricht. Diese Annahme ist in der Praxis aber nicht immer gegeben. Insbesondere bei der Verwendung von Softwarebibliotheken können nicht alle Aspekte der fremden Implementierung in der Kostenfunktion berücksichtigt werden. Desweiteren ergeben sich durch Cachehierarchien in modernen Prozessoren Laufzeiten, die nicht den theoretischen Vorhersagen entsprechen. Insbesondere kleine Probleme werden hierbei besonders schnell bearbeitet, sofern sie vollständig in einen CPU-nahen, und somit sehr schnellen Zwischenspeicher passen.

Aus diesem Grund ist der Einfluss einer solchen abweichenden Kostenfunktion auf die von einem Scheduling-Algorithmus berechnete Verteilung entscheidend für das parallele Verhalten. Das synthetische Beispiel aus dem vorherigen Kapitel wird im folgenden genutzt, um diesen Einfluss zu untersuchen. Dabei werden die angesetzten Kosten mit einem zufälligen Fehler versehen. Die resultierende Menge von Aufgaben wurde anschließend mittels List-, LPT-, Multifit-Scheduling und Sequenzpartitionierung auf die einzelnen Prozessoren verteilt, wobei der Lastbalancierung die originalen Kosten zugrundelagen.

Die folgende Tabelle zeigt die Laufzeiten für $p = 16$, $n = 524\,288$ und $L_{\min} = 3$. Daneben ist auch der prozentuale Unterschied zum jeweils besten Ergebnis angegeben.

Fehler	List		LPT		Multifit		Sequenzpart.	
0%	40.78 s	0.0%	40.95 s	0.4%	41.32 s	1.3%	43.16 s	5.8%
5%	41.88 s	0.0%	41.98 s	0.2%	42.76 s	2.1%	44.52 s	6.3%
10%	42.96 s	0.0%	43.01 s	0.1%	44.20 s	2.9%	45.88 s	6.8%
20%	45.13 s	0.1%	45.09 s	0.0%	47.08 s	4.4%	48.60 s	7.8%
50%	51.60 s	0.7%	51.23 s	0.0%	55.73 s	8.8%	56.76 s	10.8%

Bereits bei exakten Kosten ergeben sich leichte Unterschiede der Laufzeit. Insbesondere die Sequenzpartitionierung führt zu einer bemerkenswert hohen Dauer. Hierbei kommt die große Abhängigkeit dieses Algorithmus von der Anordnung der Aufgaben zum Tragen. Diese sollte deshalb entsprechend gewählt werden.

Mit wachsendem Unterschied zwischen angenommenen und realen Kosten vergrößert sich der relative Unterschied zwischen List- und LPT-Scheduling auf der einen und Multifit-Scheduling bzw. Sequenzpartitionierung auf der anderen Seite. Das LPT-Verfahren offenbart hierbei seine Nähe zum List-Scheduling und damit eine gewisse Unabhängigkeit von den Kosten.

In der Praxis sind neben diesen zufälligen Abweichungen allerdings häufiger systematische Fehler anzutreffen. Für diesen Fall wurden im folgenden Experiment die realen Kosten für kleine Aufgaben mit einem Aufwand unterhalb eines festen n_0 jeweils halbiert. Dies entspricht dem eingangs beschriebenen Verhalten in modernen Prozessoren. Die zu verteilenden Kosten blieben auch hierbei unverändert.

n_0	List		LPT		Multifit		Sequenzpart.	
4	37.89 s	0.0%	38.07 s	0.5%	41.31 s	9.0%	43.16 s	13.9%
8	36.91 s	0.0%	37.10 s	0.5%	41.31 s	11.9%	43.16 s	16.9%
16	36.76 s	1.8%	36.10 s	0.0%	41.33 s	14.5%	43.16 s	19.6%
32	36.76 s	4.7%	35.11 s	0.0%	41.32 s	17.7%	43.16 s	22.9%
64	36.76 s	7.8%	34.11 s	0.0%	41.32 s	21.1%	43.16 s	26.5%

Beim Multifit-Scheduling und bei der Sequenzpartitionierung verändert sich in diesem Beispiel die Laufzeit im gesamten Testbereich nicht. Auch List-Scheduling erreicht eine minimale Ausführungsdauer. Dagegen sinkt die Laufzeit beim LPT-Scheduling je mehr Aufgaben von der Kostenhalbierung betroffen sind.

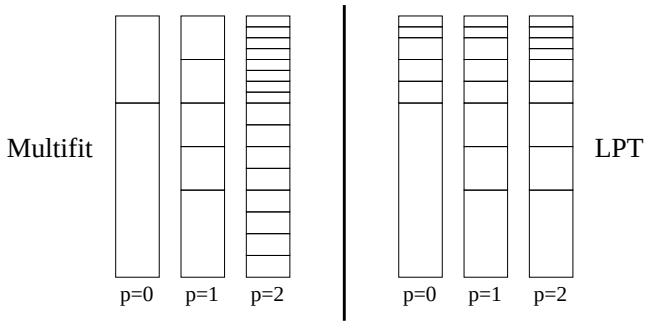


Abbildung 4.3.3: Unterschied zwischen Multifit- und LPT-Scheduling

Die Ursache für die Konstanz der Laufzeit ist eine Anhäufung von teuren Aufgaben auf einem Prozessor. Im Multifit-Scheduling wird dies durch den Algorithmus hervorgerufen, welcher zunächst Jobs so lange einer CPU zuweist, bis die Kosten einen gegebenen Betrag übersteigen (siehe Abbildung 4.3.3). Aus dem gleichen Grund können sich billige Aufgaben, deren Kosten reduziert werden, ebenfalls in einem Prozessor konzentrieren. Die Häufung innerhalb des List-Schedulings bzw. der Sequenzpartitionierung ist dagegen durch die gegebene Anordnung bestimmt.

Einzig LPT-Scheduling führt zu einer Reduktion der Laufzeit bei verminderten Kosten. Auch dies ist durch die Art der Verteilung innerhalb dieses Lastbalancierungsalgorithmus bestimmt. Hierbei werden die Aufgaben jeweils nacheinander dem nächsten Prozessor zugewiesen. Hierdurch wird die Anhäufung von Aufgaben einer bestimmten Größe auf einem Prozessor vermieden.

Insgesamt erscheint das LPT-Scheduling als das robusteste Verfahren, weshalb es im folgenden auch in der Regel genutzt wird. Lediglich in Einzelfällen, in denen bestimmte Eigenschaften der anderen Verteilungsalgorithmen von besonderem Nutzen sind, wird von dieser Wahl abgewichen.

5 Rechnerarchitekturen

Eine entscheidende aber oftmals wenig beachtete Größe beim Algorithmenentwurf stellt die genaue Funktionsweise des betrachteten Rechnersystems dar. Basiert hierbei ein Verfahren auf falschen Annahmen, so führt dies in der Regel zu unerwarteten Laufzeiten und Komplexitäten.

Aus diesem Grund sollen in diesem Kapitel Fragen des Entwurfs und der Implementierung von Programmen im Zusammenhang mit dem jeweils betrachten Rechnermodell diskutiert werden. Für einen umfassenden Überblick über die verschiedenen Architekturen sei auf [ST98], [Cam94] oder [MMT95] verwiesen.

5.1 Sequentielle Rechner

Die meisten sequentiellen Rechner folgen der *von-Neumann*-Architektur, welche bereits bei den ersten Computern in den 1940er Jahren entwickelt wurde. Hierbei ist der Rechner in ein zentrales Rechenwerk, die CPU (von engl. *Central Processing Unit*), ein Kontrollwerk, den Speicher und eine Ein-/Ausgabeeinheit unterteilt (siehe Abbildung 5.1.1).

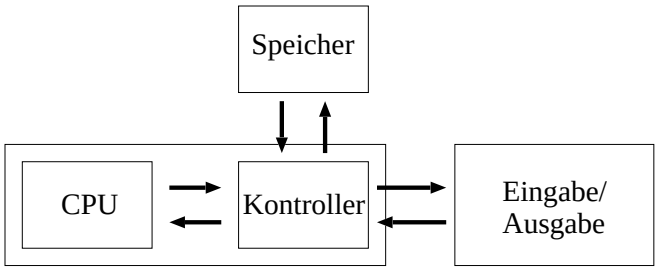


Abbildung 5.1.1: von-Neumann-Architektur

Eine wesentliche Eigenschaft dieser Architektur ist, dass im Speicher des Rechners Daten und Instruktionen gemeinsam gespeichert werden. Im Gegensatz dazu erfolgt die Speicherung von Daten und Instruktionen bei der *Harvard*-Architektur getrennt voneinander. Jeder Speichertyp besitzt hierbei eine eigene Schnittstelle zum Kontrollwerk. Obwohl moderne Rechner im wesentlichen dem von-Neumann-Modell folgen, findet in ihnen auch die Harvard-Architektur Anwendung, da innerhalb der CPU häufig getrennte Zwischenspeicher (*Caches*) für Daten und Instruktionen eingesetzt werden (siehe Abbildung 5.1.2).

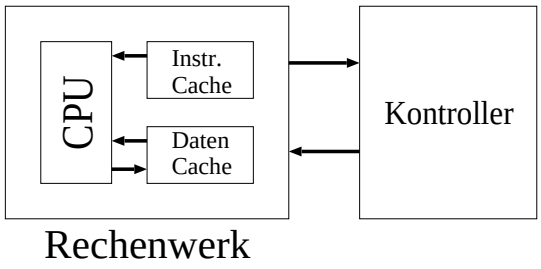


Abbildung 5.1.2: Harvard-Architektur in modernen Prozessoren

In der Theorie lassen sich beide Modelle durch eine *Random Access Machine* (RAM) (siehe [CR73]) beschreiben. Eine RAM besteht, ähnlich zu einer Turing-Maschine (siehe [Tur37]), aus einem unendlich großen Speicher. Allerdings ist bei einer RAM dieser Speicher direkt adressierbar, d.h. auf jede Speicherzelle kann in $\mathcal{O}(1)$ zugegriffen werden. Der Inhalt einer solchen Zelle besteht aus einem Wort über einem festen Alphabet. Weiterhin enthält die Maschine eine endliche Menge von *Registern*, in welche Daten, oder Adressen von Daten bzw. Instruktionen abgelegt werden. Die Menge der Instruktionen ist in diesem Modell ebenfalls endlich. Das Modell der RAM ist nicht mächtiger als das Turingmodell (siehe [HU79]), entspricht aber eher der Arbeitsweise eines modernen Rechners.

Die Beschreibung eines Rechnersystems durch das von-Neumann- bzw. das RAM-Modell erlaubt eine Implementierung oder Komplexitätsanalyse von Algorithmen, unabhängig vom speziellen Aufbau des Systems. Aufgrund der identischen Architektur der sequentiellen Rechner gelingt es somit, Programme zu entwerfen, für die sich die Laufzeit auf verschiedensten Systemen vorhersagen lässt. Wesentliche Parameter für solche Vorhersagen sind die Geschwindigkeit der CPU und des Speichersystems. Für die theoretische Komplexität der Algorithmen in Kapitel 3 bedeutet dies z.B., dass sie sich direkt auf die Laufzeit einer entsprechenden Implementierung auf einem realen Rechnersystem übertragen lässt.

5.2 Parallele Rechner

Ein einfaches und elegantes Modell für die Beschreibung des Rechners wie im sequentiellen Fall durch das von-Neumann-Modell steht für parallele Systeme bisher nicht zur Verfügung. Hierdurch ist die Implementierung und Analyse von parallelen Algorithmen derzeit stark abhängig von der Maschine, auf der die Implementierung erfolgt.

Einige der Versuche ein solches Modell zu definieren, sollen in diesem Abschnitt diskutiert werden. Insbesondere geht es dabei um die Anwendbarkeit bei der Algorithmenentwicklung und die Implementierung innerhalb der betrachteten Modelle.

5.2.1 PRAM

Die *Parallel Random Access Machine* (PRAM) (siehe [FW78]) ist die Erweiterung der normalen RAM auf mehrere Prozessoren. Eine PRAM besteht aus p Prozessoren, wobei jeder Prozessor eine RAM darstellt. Alle Prozessoren teilen sich einen gemeinsamen, globalen Speicher (siehe Abbildung 5.2.1). Sämtliche Kommunikation zwischen den Prozessoren findet über diesen Speicher statt. Wesentlich hierbei ist, dass der Zugriff auf alle Speicherzellen den gleichen Aufwand erfordert.

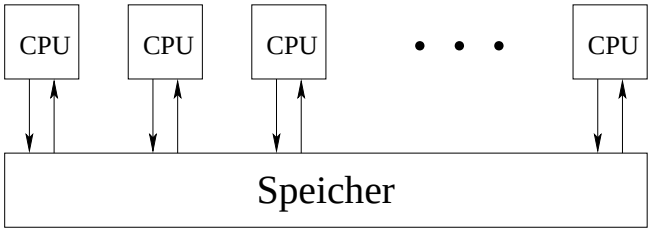


Abbildung 5.2.1: PRAM-Architektur

Ein Vorteil dieses Modells, sowohl beim Algorithmenentwurf als auch bei deren Implementierung, ist ihre Einfachheit. Die meisten Aspekte der sequentiellen Algorithmen können einfach auf die PRAM übertragen werden. Insbesondere fließt keine explizite Betrachtung der Kommunikation in den Algorithmenentwurf ein. Dies ermöglicht eine Herangehensweise, bei der ein hohes Maß der in dem Algorithmus vorhandenen Parallelität freigesetzt werden kann (siehe [MMT95]). Ebenso beschränkt sich die Analyse der Komplexität eines Algorithmus auf einer PRAM auf den Aufwand, der durch die Berechnung hervorgerufen wird. Terme, die die Kommunikation betreffen, sind dementsprechend nicht vorhanden.

Auf der anderen Seite lässt sich die Laufzeit derart entwickelter Algorithmen in der Praxis nur in seltenen Fällen abschätzen. Der Grund hierfür liegt darin, dass reale parallele Rechnersysteme durch eine PRAM nur unzureichend modelliert werden. Insbesondere die Kosten für die Kommunikation, die bei der PRAM nicht existent sind, gehen hierbei maßgeblich in die Komplexität der Algorithmen ein.

Am ehesten vergleichbar mit einer PRAM sind Rechnersysteme mit gemeinsamem Speicher. Bei diesen Systemen ist die Anzahl der Prozessoren vergleichsweise klein, mit Werten für p etwa zwischen 2 und 128. Hierdurch kann das Verbindungsnetzwerk, welches die Kommunikation zwischen den Prozessoren übernimmt, sehr effizient implementiert werden. Im Idealfall gelingt somit eine sehr gute Simulation einer PRAM. Allerdings zeigt die Praxis, dass dieser Idealfall nur von wenigen Computersystemen erreicht wird (siehe etwa [KBK03]).

Es existieren verschiedene Modifikationen des PRAM-Modells, die z.B. die gleichzeitigen Zugriffsmöglichkeiten auf Speicherzellen von verschiedenen Prozessoren aus behandeln. Im Einzelnen handelt es sich hierbei um exklusiven, beschränkten oder unbeschränkten Zugriff

beim Schreiben bzw. Lesen aus dem Speicher.

Diese Betrachtungen der Zugriffsmöglichkeiten sind insbesondere bei den schon erwähnten Rechnersystemen mit gemeinsamem Speicher wichtig. Bei modernen Prozessoren, die auch in solchen Rechnern zum Einsatz kommen, erfolgt der Speicherzugriff über unterschiedlich tiefe Cache-Hierarchien, wobei jeder Zelle in diesem Zwischenspeicher mit einer Zelle im globalen Speicher korrespondiert. Somit sind spezielle Protokolle notwendig, die die *Kohärenz* der Daten in den Zwischenspeichern auf verschiedenen Prozessoren sicherstellen. Verändert zum Beispiel ein Prozessor ein Datum in seinem lokalen Cache, muss das Verbindungsnetzwerk dafür Sorge tragen, dass diese Veränderung sowohl in der korrespondierenden Speicherzelle als auch in allen Prozessoren, die diese Speicherzelle in ihren Zwischenspeichern enthalten, bemerkt wird. Die hierfür notwendige Kommunikation verläuft unbemerkt vom Entwickler und kann entscheidenden Einfluss auf die Laufzeit eines parallelen Programms haben (siehe auch Abschnitt 8.1.1).

All diese Eigenschaften werden von einer PRAM nicht modelliert, wodurch die Tauglichkeit dieses Modells beschränkt ist. Trotzdem findet diese Architektur eine vergleichsweise breite Anwendung, nicht zuletzt auch durch seine Einfachheit. Wird insbesondere ein Computer mit gemeinsamem Speicher verwendet, so lassen sich in der Regel effiziente Algorithmen entwickeln und implementieren.

5.2.1.1 Programmieren von Rechnern mit gemeinsamem Speicher

Da sich, wie bereits beschrieben, das PRAM-Modell besonders gut für die Beschreibung von Rechnern mit gemeinsamem Speicher eignet und diese Systeme in der Praxis eine bedeutende Rolle spielen, etwa als Arbeitsplatzrechner mit 2 bis 4 Prozessoren oder als Server mit 16 oder 32 CPUs, soll in diesem Abschnitt auf die Programmierung dieser Art von Computern eingegangen werden.

Der gemeinsame Speicher, den diese Rechner allen Prozessoren zur Verfügung stellen, kann allerdings von mehreren Programmen bzw. *Prozessen*, die typischerweise jeweils nur eine CPU verwenden, nicht genutzt werden. Der Grund hierfür liegt in der Absicherung der Programme vor unerlaubter Veränderung ihrer Daten durch fremde Applikationen. Sollen die einzelnen Prozesse trotzdem Daten austauschen, so ist hierfür eine explizite Kommunikation notwendig.

Um auch innerhalb eines Programms mehrere Prozessoren zu nutzen und dabei einen gemeinsamen Speicher ansprechen zu können, wurde das Konzept von *Threads* eingeführt. Hierbei handelt es sich um parallele Ausführungspfade bzw. -fäden (von engl. *threads*) eines einzelnen Prozesses. Alle Threads teilen sich den gesamten Adressraum und somit den Speicher des Prozesses, d.h. jeder Thread kann uneingeschränkt auf alle Daten der anderen Threads zugreifen.

Für die meisten aktuellen Rechnersysteme haben sich die *POSIX* Threads oder kurz *PThreads* (siehe [But97]) als einheitlicher Standard für die Verwendung von Threads in

Programmiersprachen durchgesetzt. Allerdings ist die Benutzung der PThread-Funktionen, bis auf wenige einfache Beispiele, vergleichsweise umständlich. Der Grund hierfür liegt in der Fülle von Optionen, die die Verwaltung von Threads innerhalb eines Betriebssystems bietet. Beim praktischen Einsatz ist man aber an einer einfachen Schnittstelle interessiert, die die Konzentration auf das eigentliche Problem, etwa den parallelen Algorithmus, und nicht auf die Threadverwaltung erlaubt.

Ein Konzept, welches die niederen PThread-Funktionen vor dem Benutzer versteckt und mit einer kleinen Menge von eigenen Funktionen die Verwaltung von nebenläufigen Aufgaben ermöglicht, sind *Threadpools*. Ein Threadpool besteht aus einer festen Anzahl von Threads. Der Benutzer übergibt dem Threadpool eine bestimmte Aufgabe, welche einem freien Thread zugewiesen und anschließend ausgeführt wird. Steht kein freier Thread zur Verfügung, blockiert der Threadpool die aufrufende Prozedur solange, bis ein Thread die Bearbeitung einer Aufgabe beendet hat. Der Benutzer hat weiterhin die Möglichkeit, das Programm mit dem Beenden einer Aufgabe zu synchronisieren, d.h. die Ausführung so lange zu blockieren, bis eine vom Threadpool bearbeitete Aufgabe beendet wurde.

Die Schnittstelle eines solchen Threadpools kann auf die folgenden Funktionen beschränkt werden:

- `tp_init`(p) : initialisiert den Threadpool mit p Threads,
- `tp_run`(j) : führt die Aufgabe j im Threadpool aus,
- `tp_sync`(j) : synchronisiert das Programm mit dem Beenden der Aufgabe j und
- `tp_sync_all`(): synchronisiert das Programm mit dem Beenden aller ausgeführten Aufgaben.

Neben der einfacheren Benutzbarkeit eines Threadpools im Vergleich zu den PThread-Funktionen ergibt sich durch die Verwendung von festen Threads ein Geschwindigkeitsgewinn, da das Starten von Threads durch das Betriebssystem unter Umständen einen hohen Aufwand besitzt. Innerhalb des Threadpools ist der Aufwand dagegen nur durch die Kopplung von Thread und Aufgabe bestimmt. Hierdurch lassen sich Algorithmenkonzepte, wie etwa das *Online-Scheduling* realisieren.

Die Lastbalancierungsalgorithmen in Abschnitt 4 berechnen die Verteilungsfunktion für die einzelnen Aufgaben *vor* der eigentlichen Ausführung. Diese Vorgehensweise wird deshalb auch *Offline-Scheduling* genannt. Im Gegensatz hierzu erfolgt die Zuweisung der Aufgaben zu den Prozessoren beim Online-Scheduling *während* des Ablaufs des auszuführenden Algorithmus. Ein Verteilungsalgorithmus, welcher sich hierfür besonders gut eignet, ist das List-Scheduling (siehe Abschnitt 4.1.1). Die Grundidee dieser Art der Verteilung war, die erste noch nicht bearbeitete Aufgabe dem ersten freien Prozessor zuzuweisen. In einen Online-Scheduling-Algorithmus umgesetzt, ergibt sich für eine Menge von Aufgaben $J = \{j_0, \ldots, j_{n-1}\}$ somit folgendes Verfahren:

```
for all j ∈ J do
    sei q der erste freie Prozessor;
    berechne j auf q;
endfor;
```

Der Vorteil dieser Methode ist, dass kein Wissen über die Kosten der zu bearbeitenden Aufgaben für die Lastbalancierung notwendig ist. Das Ergebnis

$$T(\Sigma_{\mathrm{LS}}) \leq \left(2 - \frac{1}{p}\right) T_{\min}.$$

aus Lemma 4.1.2 für die Güte des List-Scheduling im Vergleich zu einer optimalen Verteilung bleibt dabei bestehen. Die Verwendung eines Threadpools vereinfacht hierbei lediglich die Identifikation eines freien Prozessors. In der Funktion `tp_run`() wird die übergebene Aufgabe dem ersten freien Thread zugewiesen bzw. die Ausführung solange blockiert, bis ein solcher zur Verfügung steht. Der Online-Scheduling-Algorithmus lässt sich somit wie folgt umformulieren:

```
for all j ∈ J do
    tp_run( j );
tp_sync_all();
```

Algorithmus 5.2.1: Online-Scheduling mittels Threadpool

Einige der in Abschnitt 6 vorgestellten parallelen Algorithmen machen von diesem einfachen Schema Gebrauch, um effiziente Verfahren zu implementieren.

Die vollständige Implementation eines Threadpools mit einer weitergehenden Diskussion der Vorteile gegenüber den nativen PThread-Funktionen ist z.B. in [Kri03] nachzulesen.

Wegen des gemeinsamen Speichers und des uneingeschränkten Zugriffs aller Threads auf eine Speicherzelle sind zusätzliche Vorkehrungen zu treffen, um die Sicherheit der Daten zu gewährleisten. Eine Möglichkeit besteht in der Verwendung von *Mutices*. Ein Mutex ist eine Variable, die sich entweder im Zustand `GESPERRT` oder `FREI` befindet. Hierdurch lassen sich Bereiche eines Programms vor gleichzeitigem Zugriff durch mehrere Threads schützen. Beim Betreten des kritischen Bereiches blockiert ein Thread solange, bis ein entsprechend vorhandener Mutex den Zustand `FREI` annimmt. Anschließend setzt der Thread den Zustand `GESPERRT`, so dass alle weiteren konkurrierenden Threads blockieren. Nachdem der entsprechende Bereich verlassen wurde, kann der Zustand des Mutex wieder in `FREI` geändert werden. Auch für Mutices sind Funktionen `lock`(m) und `unlock`(m) im PThread-Standard vorgesehen, die den Mutex m jeweils in den Zustand `GESPERRT` bzw. `FREI` überführen.

Eine alternativer Programmierzugang zu Rechnersystemen mit gemeinsamem Speicher gelingt über den *OpenMP* Standard (siehe [DM98]). Dieser definiert Erweiterungen von Sprachen wie *C* oder *Fortran*, um Programmteile parallel auszuführen. Hierbei spezifiziert der Programmierer die jeweiligen, parallel abzuarbeitenden Abschnitte und der Compiler generiert Verwaltungsstrukturen für die entsprechende Umsetzung auf einem Rechnersystem.

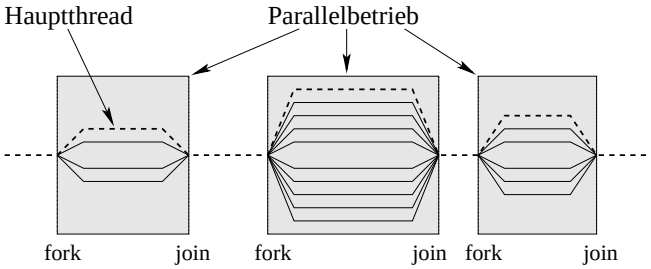


Abbildung 5.2.2: „fork-join“ Modell von OpenMP

Üblicherweise wird dabei auf Threads zurückgegriffen. Auf diese Weise vereinfacht sich der Umgang mit einem parallelen System entscheidend.

Allerdings basiert der OpenMP Standard auf einem sogenannte „fork-join“-Modell (siehe Abbildung 5.2.2), d.h. es wird davon ausgegangen, dass die einzelnen Threads zu einem bestimmten Zeitpunkt gestartet werden („fork“) und später gemeinsam beendet werden („join“). Der *Hauptthread*, welcher bereits beim Start des Programms aktiv ist, bleibt hierbei stets erhalten. Besonders gut ist diese Arbeitsweise für die Parallelisierung von langen Schleifen geeignet, welche etwa bei der Bearbeitung von großen Vektoren entstehen. Dagegen ist das Starten von einzelnen Threads zu beliebigen Zeitpunkten nicht möglich. Hierdurch ist die Umsetzung eines allgemeinen Online-Scheduling Verfahrens, bei welchem die zu bearbeitenden Aufgaben erst während des Algorithmus entstehen, schwieriger als bei direkter Verwendung von PThreads.

Ein Beispiel hierfür bildet eine Baumstruktur, bei welcher die Aufgaben an die Blätter gebunden sind. Ein Online-Scheduling Verfahren könnte den Baum mittels Tiefensuche durchlaufen und beim Antreffen eines Blattes den entsprechenden Job an einen freien Thread koppeln. Man beachte, dass der Baumdurchlauf sequentiell erfolgt. Bei Verwendung von OpenMP ist das Starten von Threads während eines sequentiellen Algorithmus nicht möglich. Hierfür sind die Aufgaben zunächst zu identifizieren. Dies erfolgt durch die erwähnte Tiefensuche. Anschließend können die Aufgaben dann innerhalb einer Schleife parallel bearbeitet werden. Zwar haben beide Ansätze den Baumdurchlauf gemein, allerdings tritt bei der OpenMP-Variante ein zusätzlicher sequentieller Aufwand für das Abarbeiten der Schleife auf.

Es sei allerdings darauf hingewiesen, dass in der Regel die Zeit zum Bearbeiten der einzelnen Aufgaben dominiert und somit sequentiellen Anteile vernachlässigt werden können. Deshalb gelingt auch mittels OpenMP ein Zugang zur effizienten Parallelisierung. In dieser Arbeit wurde dennoch auf die direkte Verwendung von PThreads mittels eines Threadpool zurückgegriffen, da sich Online-Scheduling Verfahren hierdurch einfacher implementieren lassen. Zudem muss der eingesetzte Compiler eine OpenMP-Unterstützung bereitstellen, was nicht auf allen Rechnersystemen der Fall war. Die Unterstützung von Threads ist dagegen

unabhängig vom Compiler.

5.2.1.2 Reale Rechnersysteme mit gemeinsamem Speicher

Das in dieser Arbeit für die numerischen Experimente von PRAM-basierten Algorithmen in Kapitel 6 eingesetzte System ist eine *HP 9000 Superdome* der Firma *Hewlett-Packard*. Dieser Rechner besteht aus einzelnen Platinen, welche jeweils 4 Prozessoren aufnehmen können. Der Zugriff dieser CPUs auf den Arbeitsspeicher sowie die Ein- und Ausgabe-Schnittstellen erfolgt über einen sogenannten *Cell Controller*. Hierbei wird zwischen *lokalem* Speicher, d.h. auf der gleichen Platine, und *entferntem* Speicher, also Adressen auf fremden Platinen unterschieden. Im letzteren Fall erfolgt die Verbindung zunächst über eine zusätzliche Logik, die *Crossbar*. Diese fasst maximal 4 Platinen zusammen. Bei mehr als 16 Prozessoren erfolgt die Verknüpfung über eine weitere Zwischenebene (siehe auch Abbildung 5.2.3). Der Maximalausbau beträgt hierbei 16 Platinen.

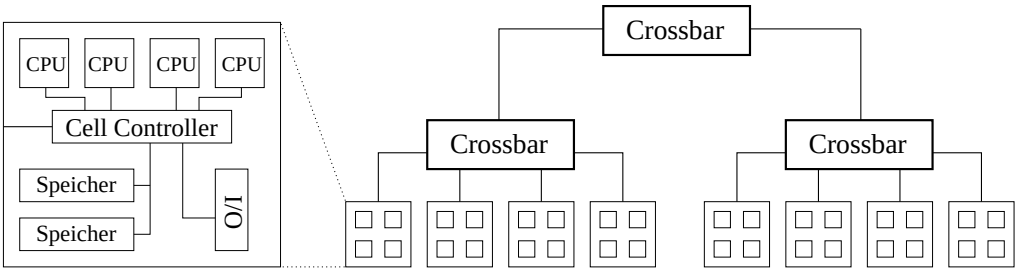


Abbildung 5.2.3: HP 9000 Superdome

Auf diese Weise erfolgt der Zugriff auf verschiedene Speicherzellen mit unterschiedlicher Geschwindigkeit. Für den lokalen Speicher beträgt die Zugriffszeit 174 ns, über eine *Crossbar* 240 ns und über zwei *Crossbars* 300 ns (siehe [Hew02]). Somit nähert dieses System eine PRAM lediglich an. Man spricht auch von *non-uniform memory access*. Allerdings zeigen sich die Auswirkungen dieser Architektur nur in einzelnen Fällen (siehe [KBK03]).

Die verwendeten Prozessoren sind vom Typ *PA-8700+* mit 875 MHz. Sie verfügen über einen Zwischenspeicher von 1.5 MB für Daten und 0.75 MB für Instruktionen und folgen damit der Harvard-Architektur.

Aufgrund der zur Verfügung stehenden Software wurde für die Experimente in Kapitel 8 ein anderes Rechnersystem eingesetzt. Dabei handelt es sich um eine *SunFire 6700* der Firma *Sun*. Der Aufbau dieses Rechers ist ähnlich zu dem der HP 9000 Superdome, allerdings wird lediglich eine *Crossbar* verwendet (siehe [Sun02]). Hierdurch ist der Unterschied in der Zugriffzeit zwischen lokalem Speicher (180 ns) und entferntem Speicher (240 ns) im Durchschnitt geringer. Auch fällt der lokale Zwischenspeicher der eingesetzten *UltrasparcIII*-Prozessoren mit 8 MB größer aus und wird sowohl für die Speicherung von Daten wie auch

Instruktionen genutzt. Die Geschwindigkeit der CPUs beträgt 900 MHz.

Insgesamt ist der Aufbau dieses Systems im Vergleich zur HP 9000 Superdome enger an der Definition einer PRAM ausgerichtet, wobei sich in der Praxis aber nur bedingt Vorteile ergeben.

5.2.2 Das BSP-Modell

Wie schon bei der Beschreibung der Systeme im letzten Abschnitt deutlich wurde, erfolgt die Kommunikation zwischen verschiedenen Prozessoren in realen Rechnern typischerweise nicht über einen gemeinsamen Speicher. Vielmehr sind diese Systeme aus einzelnen, eigenständigen Rechnereinheiten aufgebaut, die über ein Kommunikationsnetzwerk verbunden sind. Jede Rechnereinheit stellt hierbei eine RAM dar, d.h. sie verfügt über einen Prozessor und über einen *lokalen* Speicher (siehe Abbildung 5.2.4). Man spricht in diesem Zusammenhang auch von Rechnern mit *verteiltem* Speicher.

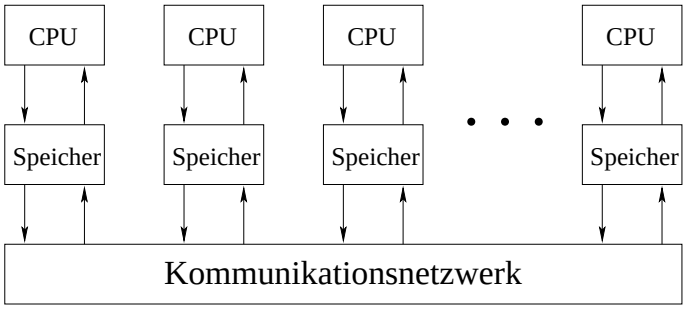


Abbildung 5.2.4: Rechnerarchitektur mit verteiltem Speicher

Im Gegensatz zu der PRAM-Architektur ist der Aufwand für den Zugriff auf *entfernten* Speicher, d.h. Speicher eines anderen Prozessors über das Kommunikationsnetzwerk, größer als der Zugriff auf lokalen Speicher. Somit ist es notwendig, die Kommunikation in das Rechnermodell mit einfließen zu lassen.

Ein solches Modell mit Kommunikation ist die *bulk-synchronous parallel machine* oder BSP-Maschine, wie sie in [Val90] beschrieben wird. Drei Parameter beschreiben eine solche BSP-Maschine: p, g und l. Die Anzahl der Prozessoren ist hierbei wieder durch p definiert. Der Parameter g beschreibt das Verhältnis der Gesamtzahl aller lokalen Operationen (Addition, Multiplikation etc.) in allen Prozessoren zu der Anzahl der Daten, die durch das Kommunikationsnetzwerk transferiert werden können. Er gibt somit ein Maß, wie hoch die Kosten für das Ausliefern eines Datums im Netzwerk sind. Die Zeit, die für eine globale Synchronisation aller Prozessoren benötigt wird, ist durch den letzten Parameter l definiert. Implizit existiert noch ein vierter Parameter s für die Geschwindigkeit, etwa die Anzahl von Fließkommaoperationen pro Sekunde, der einzelnen Prozessoren. Da aber g und l stets

bezüglich s normalisiert sind, kann dieser Parameter in der weiteren Betrachtung ignoriert werden.

Eine Berechnung innerhalb einer BSP-Maschine ist in einzelne Schritte unterteilt. Jeder Schritt ist gekennzeichnet durch eine Berechnungsphase, bei der nur auf Daten im lokalen Speicher zugegriffen wird und durch eine Kommunikationsphase, bei der Daten versendet und empfangen werden. Anschließend erfolgt eine globale Synchronisation, wobei sichergestellt wird, dass nach der Synchronisation alle für den nachfolgenden Schritt benötigten Daten auf den jeweiligen Prozessoren vorhanden sind. Eine solche Berechnung ist in Abbildung 5.2.5 graphisch dargestellt.

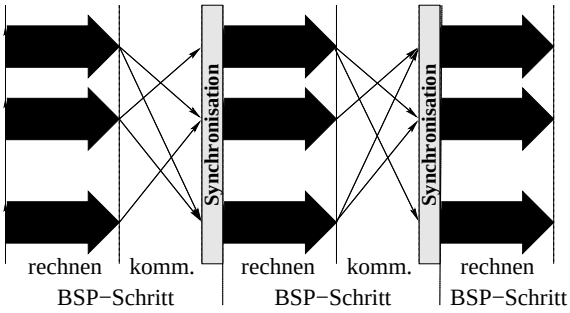


Abbildung 5.2.5: BSP-Berechnung durch einzelne Schritte

Der Vorteil dieser Art der parallelen Berechnung liegt in seiner Einfachheit. Innerhalb der Berechnungsphase eines BSP-Schrittes entspricht das parallele Programm seinem sequentiellen Pendant, da lediglich lokale Daten verarbeitet werden. Die Art der Berechnung innerhalb dieser Phase ist außerdem asynchron, d.h. jeder Prozessor kann unterschiedliche Algorithmen ausführen, ohne auf die Berechnung der anderen Prozessoren zu achten. Die Kommunikation und damit der schwer zu modellierende parallele Anteil wird dagegen von der Berechnung getrennt. Man beachte auch, dass in BSP-Algorithmen nicht die Gefahr eines *Dead-Locks* existiert, bei dem z.B. zwei CPUs gleichzeitig auf das Versenden eines Datums des jeweils anderen Prozessors warten.

Auf der anderen Seite lässt sich innerhalb des BSP-Modells eine Lastbalancierung mittels Online-Scheduling nur sehr schwer implementieren. Der Grund hierfür liegt in den erzwungenen Synchronisationspunkten. Beim List-Scheduling, als Beispiel für ein Online-Verfahren, existieren diese Stellen innerhalb des Algorithmus nicht und müssen somit künstlich definiert werden. Gleichzeitig ist es hierbei aber schwierig, die gewünschte parallele Komplexität aufrechtzuerhalten. Keine Probleme bereiten dagegen die Algorithmen, welche Offline-Scheduling verwenden, weshalb diese Methode im BSP-Modell favorisiert wird.

Ein weiterer Vorteil ist die Analyse des Laufzeitverhaltens eines BSP-Algorithmus, welche sich als vergleichsweise einfach gestaltet. Es genügt, jeden Schritt einzeln zu untersuchen. Hierbei sei w_i^j die Anzahl der Operationen, die Prozessor i in einem Schritt S_j ausführt und

h_i^j die Menge an Daten, die i in S_j empfängt bzw. sendet. Dann lauten die Kosten für den Schritt S_j

$$\max_{0 \le i < p} w_i^j + g \cdot \max_{0 \le i < p} h_i^j + l.$$

Summiert man die Kosten für jeden einzelnen Schritt, so ergibt sich für den gesamten Algorithmus ein Aufwand der Form:

$$W + g \cdot H + l \cdot S$$

mit

$$W = \sum_{j=0}^{S-1} \max_{0 \le i < p} w_i^j, \quad H = \sum_{j=0}^{S-1} \max_{0 \le i < p} h_i^j$$

und der Anzahl der Schritte S. Somit lässt sich allein durch den Vergleich der Parameter g und l realer Rechnersysteme die Laufzeit eines bestimmten Algorithmus vorhersagen. Der Algorithmenentwurf ist dementsprechend nicht abhängig von einem bestimmten Computer.

Das BSP-Modell beschreibt leider nicht alle Eigenschaften eines realen Rechners. So wurde z.B. im BSP*-Modell (siehe [BDadH95]) ein zusätzlicher Parameter eingefügt, welcher eine minimale Datengröße angibt, ab der das Kommunikationsnetzwerk die dem Parameter g entsprechende Leistung liefert. Dies entspricht der Eigenschaft, dass ein großes Datenpaket typischerweise schneller versandt werden kann, als viele kleine Datenblöcke. In [SHM97] wird eine Lösung für das gleiche Problem über eine Modifikation von g beschrieben und getestet.

In der Praxis zeigen sich solche Phänomene allerdings eher selten, da die eigentliche Kommunikation, d.h. das Verschicken der Nachrichten, durch eine BSP-Laufzeitumgebung erfolgt. Hierbei werden häufig kleine Nachrichten automatisch zu einem großen Datenpaket zusammengefasst.

Beispiele für verschiedene numerische und andere BSP-Algorithmen sind z.B. in [McC95b], [McC95a], [GV94] und [Bis04] zu finden.

5.2.2.1 Programmieren von BSP-Maschinen

Aufgrund der Arbeitsweise einer BSP-Maschine werden im wesentlichen 4 Funktionen benötigt:

- `bsp_sync`(): führt die globale Synchronisation durch und beendet damit einen BSP-Schritt,
- `bsp_send`(i, m) : sendet die Nachricht m an den Prozessor i,
- $m :=$ `bsp_recv`(): empfängt eine Nachricht m, die bei der letzten Synchronisation an den jeweiligen Prozessor übertragen wurde und
- `bsp_nmsgs`(): liefert die Anzahl der übertragenen, aber noch nicht vom Benutzer empfangenen Nachrichten zurück.

Der eigentliche Empfang der Nachrichten erfolgt allerdings bereits im vorherigen Schritt. Somit dient die Funktion `bsp_recv`() lediglich der Überführung der Daten von der BSP-Laufzeitumgebung zum Benutzer. Die Funktion `bsp_nmsgs`() ist im allgemeinen nicht notwendig. Allerdings hat sie sich bei der praktischen Verwendung als sehr vorteilhaft erwiesen.

Es existieren verschiedene Softwarebibliotheken, die diese Funktionalität bieten und somit das Programmieren eines BSP-Programms unterstützen, z.B. `BSPlib` ([HMS+98]), `PUB` ([BJvOR03]) und `GreenBSP`. Diese Bibliotheken wurden für eine Vielzahl von unterschiedlichen Rechnersystemen implementiert und bieten damit die Möglichkeit eines breiten Einsatzspektrums für das jeweilige Programm.

Einige Implementierungen (`BSPlib` und `PUB`) ermöglichen neben einer auf Nachrichten basierenden Funktionalität auch den direkten Zugriff auf Daten in entfernten Speichern. Da sich diese Kommunikationsform aber auch mittels Nachrichten simulieren lässt und außerdem besondere Sorgfalt notwendig ist, um die Konsistenz der Daten zu gewährleisten, soll hierauf nicht näher eingegangen werden. Die späteren BSP-Algorithmen verwenden stets nur die eingangs beschriebenen Funktionen.

Man beachte, dass auch eine PRAM eine BSP-Maschine darstellt. Somit kann bei der Programmierung von Rechnern mit gemeinsamem Speicher anstelle des Thread-Ansatzes aus Abschnitt 5.2.1.1 auch eine BSP-Bibliothek verwendet werden. Die genannten Softwarebibliotheken unterstützen in der Regel diese Möglichkeit.

Als Beispiel für einen BSP-Algorithmus wird im folgenden auf einer BSP-Maschine mit p Prozessoren die Summation von p Zahlen, mit jeweils einer Zahl pro Prozessor durchgeführt. Das Ergebnis der Addition soll nach Beendigung des Verfahrens auf allen Prozessoren vorliegen. In der einfachsten Variante senden alle Prozessoren das lokale Datum an Prozessor 0, welcher anschließend die Summation durchführt und das Ergebnis an alle anderen CPUs versendet. Der Aufwand dieses Ansatzes beträgt

$$\mathcal{O}\left(p + g \cdot p + l \cdot 3\right).$$

Ein solches Verfahren ist somit weder in Bezug auf den Berechungs- noch auf den Kommunikationsaufwand optimal. Lediglich die Zahl der BSP-Schritte ist konstant.

Alternativ lassen sich die Zahlen auch baumartig aufsummieren (siehe Abbildung 5.2.6). Das Ergebnis befindet sich am Ende der Summation ebenfalls in Prozessor 0. Das Versenden erfolgt im Anschluss an das Aufaddieren, wobei jede CPU, welche das Datum bereits empfangen hat, dieses an einen weiteren Prozessor sendet. Die hierdurch entstehenden Kosten lauten

$$\mathcal{O}\left(\lceil \log_2 p \rceil (1 + l + g)\right)$$

und sind somit bezüglich der Berechnung und der Kommunikation geringer als im direkten Fall. Allerdings ist die Anzahl der BSP-Schritte nun abhängig von p.

Der BSP-Algorithmus zur baumartigen Summation lässt sich wie folgt formulieren:

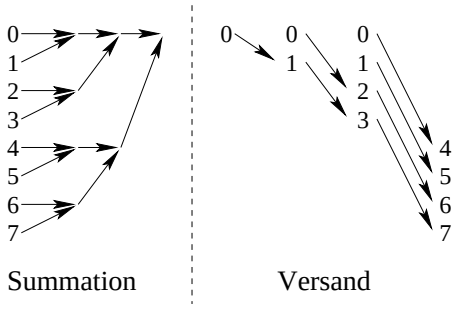


Abbildung 5.2.6: Baumartiges Aufsummieren und Versenden

```
procedure tree_sum ( i, d )
    s := 2; a := true;
    for i = 0, ..., ⌈log2 p⌉ do
        if a = true then
            { Summation der empfangenen Daten }
            while bsp_nmsgs() > 0 do d := d + bsp_recv();
            { Versenden des lokalen Datums }
            if i mod s ≠ 0 then
                bsp_send( ⌊i ÷ s⌋ · s, d ); a := false;
            endif;
        endif;
        bsp_sync(); s := 2 · s;
    endfor;
end;
```

Algorithmus 5.2.2: Summation eines Datums

Hierbei sind in jedem Schritt nur die Prozessoren aktiv ($a =$ `true`), die das jeweilige lokale Ergebnis noch nicht versendet haben.

Das entsprechende Pendant zum Verteilen des Ergebnisses lautet:

```
procedure tree_broadcast ( i, d )
    m := 1;
    while m < p do
        if i < m ∧ m + i < p then
            bsp_send( m + i, d )
        bsp_sync();
        if bsp_nmsgs() > 0 then
            d := bsp_recv();
        m := 2 · m;
    endwhile;
end;
```

Algorithmus 5.2.3: Versenden des Ergebnisses

Beide Algorithmen werden in den späteren BSP-Verfahren genutzt, um eine effiziente Summation durchzuführen. Da in diesen Fällen in der Regel große Daten addiert werden, ist die baumartige Summation aufgrund der günstigeren Komplexität dem direkten Verfahren vorzuziehen.

5.2.2.2 Reales Rechnersystem mit verteiltem Speicher

Als Beispiel für ein System mit verteiltem Speicher, welches auch für die numerischen Experimente dieser Arbeit genutzt wurde, dient ein Rechner bestehend aus 16 PCs. Jeder einzelne Knoten verfügt über einen Prozessor vom Typ *Athlon* der Firma *AMD*, welcher mit 900 MHz getaktet wird. Weiterhin stehen den CPUs jeweils 768 MB an Arbeitsspeicher zur Verfügung.

Die Verbindung zwischen den Knoten erfolgt über ein *FastEthernet*-Netzwerk mit einer maximalen Transferrate von 100 MBit pro Sekunde. Dabei beträgt die minimale Zeit für das Senden eines Datenelementes etwa 60 bis 80 μs und liegt somit etwa um den Faktor 200 bis 400 höher als bei den Rechnersystemen mit gemeinsamem Speicher (siehe Abschnitt 5.2.1.2).

5.2.3 Das *LogP*-Modell

Ausgehend von einer BSP-Maschine versucht das *LogP*-Modell (siehe [CKP+93]) reale Rechnersysteme noch genauer zu beschreiben. Hierbei wird insbesondere die Modellierung der Kommunikation im Gegensatz zum BSP-Modell verfeinert. Der Parameter L beschreibt hierbei die Latenzzeit des Netzwerks, d.h. die Zeit, die das Versenden und Empfangen einer sehr kleinen Nachricht benötigt. Dies ist vergleichbar mit dem Parameter l im BSP-Modell, allerdings wird im *LogP*-Modell keine globale Synchronisation durchgeführt.

Desweiteren beschreibt o die Zeit, die das Einfügen bzw. Empfangen von Nachrichten aus dem Kommunikationsnetzwerk benötigt, d.h. den durch jeden Kommunikationsaufruf verursachten zusätzlichen Aufwand oder „Overhead“. Die minimale Zeit zwischen aufeinanderfolgenden Übertragungen wird mit g bezeichnet. Der Kehrwert von g entspricht damit der Kommunikationsbandbreite, die das Netzwerk jedem Prozessor bietet.

Vervollständigt wird das Modell (und der Name) durch den Parameter P, welcher die Anzahl der Prozessoren im Rechnersystem bestimmt.

Eine weitere wichtige Eigenschaft im *LogP*-Modell ist eine beschränkte Kapazität des Netzwerks. Diese ist durch $\lceil L/g \rceil$ definiert. Überschreitet ein Prozessor bei der Nachrichtenübertragung diese Schranke, so blockiert er so lange, bis genügend Kapazität im Netzwerk zur Verfügung steht.

Das *LogP*-Modell lenkt somit beim Algorithmenentwurf eine größere Aufmerksamkeit auf die Parameter des eigentlichen Rechnersystems, womit realistischere Vorhersagen über die Laufzeit als beim BSP-Modell möglich sind. Asymptotisch sind die beiden Modelle allerdings äquivalent, wie in [BHP+99] gezeigt wird. Desweiteren ist die Programmierung von BSP-Algorithmen durch die größere Abstraktion des Kommunikationsnetzwerks und durch die Synchronisation deutlich leichter als die von *LogP*-Programmen.

5.2.4 Direkte Parallelisierung mit MPI oder PVM

Die in den letzten Jahren in der Praxis am häufigsten eingesetzten Schnittstellen für die Parallelisierung basieren auf dem direkten Austausch von Nachrichten. Die bekanntesten Vertreter sind das *Message Passing Interface* (MPI, siehe [GNL98]) und die *Parallel Virtual Machine* (PVM, siehe [GBD+94]).

Im Gegensatz zu den bisherigen Architekturbeschreibungen handelt es sich hierbei nicht um Rechnermodelle, sondern um Definitionen von Funktionsschnittstellen für das Übertragen von Daten. Hierin enthalten sind Funktionen für Punkt-zu-Punkt-Kommunikation, kollektive Kommunikation, globale Synchronisation und viele höhere Methoden für die direkte Verarbeitung von Daten. Diese Standards sind damit mächtiger als das PRAM-, BSP- oder *LogP*-Modell. Die BSP-Softwarebibliotheken BSPlib und PUB (siehe Abschnitt 5.2.2.1) bieten etwa die Möglichkeit, die Laufzeitumgebung für BSP-Programme auf MPI-Funktionen aufzusetzen.

Allerdings ist der Entwurf von Programmen, z.B. im Vergleich zu BSP-Algorithmen, deutlich komplizierter und anfällig gegenüber *Dead-Locks* (siehe Abschnitt 5.2.2). Auch ist die Analyse der Laufzeit von Programmen im allgemeinen schwieriger, da Kommunikation und Berechnung typischerweise parallel erfolgen.

Anhand des Skalarprodukts zwischen zwei Vektoren $x, y \in \mathbb{R}^n$ soll der grundsätzliche Unterschied zwischen einem BSP-Programm und einer Implementierung mittels des MPI-Standards verdeutlicht werden. In beiden Programmen hält jeder der p Prozessoren n/p Koeffizienten von x bzw. y. Diese lokalen Vektoren seien mit x_i und y_i bezeichnet. Das Ergebnis der Berechnung soll am Ende des Algorithmus in allen Prozessoren zur Verfügung stehen. Die hierbei verwendeten Funktionen `MPI_send` und `MPI_recv` senden eine Nachricht an bzw. empfangen eine Nachrichten von einem entsprechend angegebenen Prozessor.

BSP-Programm	MPI-Programm

procedure skp_bsp(i, x_i, y_i)
  $f = \langle x_i, y_i \rangle$;
  `bsp_send`($0, f$);
  `bsp_sync`();
  if $i = 0$ **then**
  $g := 0$;
  while `bsp_nmsgs`() > 0 **do**
  $f :=$ `bsp_recv`();
  $g := g + f$;
  endwhile;
  for $i = 1, \ldots, p-1$ **do**
  `bsp_send`(i, g);
  endif;
  `bsp_sync`();
  if $i > 0$ **then** $g :=$ `bsp_recv`();
end;
	procedure skp_mpi(i, x_i, y_i)
	  $f = \langle x_i, y_i \rangle$;
	  if $i > 0$ **then**
	  MPI_send(0, f);
	  MPI_recv(0, g);
	  else
	  $g := f$;
	  for $i = 1, \ldots, p-1$ **do**
	  MPI_recv(i, f);
	  $g := g + f$;
	  endfor;
	  for $i = 1, \ldots, p-1$ **do**
	  MPI_send(i, g);
	  endif
	end;

Typisch für ein MPI-Programm ist die enge Kopplung zwischen dem Senden und Empfangen einer Nachricht. Hierbei wurde die *blockierende* Kommunikation von MPI verwendet, d.h. der sendende Prozessor kann die Funktion `MPI_send` erst verlassen, nachdem die Routine `MPI_recv` auf der Empfangsseite aufgerufen wurde. Durch diese Bedingung ist die Gefahr eines *Dead-Locks* sehr groß, falls eine zyklische Abhängigkeit zwischen den Prozessoren besteht. Im BSP-Programm wird die Abhängigkeit dagegen aufgelöst. Dies entspricht der *nicht-blockierenden* Kommunikation von MPI, d.h. das Senden einer Nachricht wird von einem Prozessor initiiert, ohne dass auf die Empfangsbestätigung gewartet wird. Diese wird erst zu einem späteren Zeitpunkt geprüft. Innerhalb des BSP-Programms ist dieser Zeitpunkt äquivalent mit der Synchronisation des BSP-Rechners.

Ein großer Vorteil von MPI im Vergleich zu BSP besteht darin, dass für viele Rechnersysteme optimierte Implementierungen erhältlich sind, wodurch insbesondere die Effizienz von Applikationen gesteigert werden kann. Durch eine Simulation von BSP mittels MPI lassen sich somit häufig die positiven Eigenschaften der beiden Modelle verbinden.

6 Parallele $\mathcal{H}$-Arithmetik

In diesem Kapitel sollen zu den sequentiellen $\mathcal{H}$-Matrix-Algorithmen aus Kapitel 3 parallele Verfahren eingeführt werden. Die einzelnen Verfahren werden dabei sowohl für Rechnersysteme mit verteiltem Speicher in Form einer BSP-Maschine als auch für Computer mit gemeinsamem Speicher vorgestellt, wie sie im letzten Kapitel beschrieben wurden (Abschnitt 5.2.2 bzw. 5.2.1). Dabei werden die verschiedenen Eigenschaften beider Systeme explizit ausgenutzt, um eine hohe oder maximale parallele Leistung (siehe Abschnitt 6.1) zu erhalten.

Daneben gilt ein besonderes Augenmerk Algorithmen, die sich auf einfache Art und Weise aus den sequentiellen Verfahren ableiten, wodurch ein Großteil der bisher verwendeten Funktionen erhalten bleibt.

Analog zur Konvention in Kapitel 3 seien auch im folgenden I, J Indexmengen mit $|I| = n$ und $|J| = m$. Weiterhin seien $T(I)$ und $T(J)$ Clusterbäume über I und J. Mit $T = T(I \times J)$ wird der Blockclusterbaum über $T(I)$ und $T(J)$ bezeichnet. Um genügend Aufgaben für eine Parallelisierung zur Verfügung zu haben, und damit die Voraussetzung von Vermutung 4.3.3 zu erfüllen, sei desweiteren neben $n, m \gg p$ auch $|\mathcal{L}(T)| \gg p$ angenommen.

6.1 Parallele Effizienz

Wesentlich für die Bewertung eines parallelen Verfahrens ist die Abhängigkeit der Laufzeit von der Zahl der Prozessoren. Im Idealfall, d.h. bei einem vollständig parallelen Programm ohne sequentielle Anteile und einer optimalen Lastverteilung, stellt sich dabei ein $1/p$-Verhalten ein. Reale Algorithmen weichen hiervon allerdings oftmals ab. Für eine genauere Quantifizierung lassen sich die beiden folgenden Begriffe verwenden.

Definition 6.1.1 *Sei $t(p)$ die Zeit, die ein paralleler Algorithmus $\mathfrak{A}$ mit p Prozessoren benötigt. Dann bezeichnen*

$$S(p) = \frac{t(1)}{t(p)} \quad \textit{und} \quad E(p) = \frac{S(p)}{p} = \frac{t(1)}{p \cdot t(p)} \tag{6.1.1}$$

die parallele Skalierung *oder* Speedup *bzw. die* parallele Effizienz *von* $\mathfrak{A}$.

Der Speedup ist somit bei dem beschriebenen Idealfall identisch zu p, während die parallele Effizienz einen Wert von 100% annimmt. In der Praxis sind diese beiden Werte in der Regel kleiner, da etwa nicht parallelisierte Anteile oder im Vergleich zum sequentiellen Verfahren zusätzlich durchzuführende Berechnungen eine perfekte Skalierung verhindern.

In Einzelfällen kann aber auch, bedingt durch Eigenschaften des Rechnersystems, wie etwa Cache-Architekturen, die im Mehrprozessorbetrieb besser ausgenutzt werden können, eine Effizienz von mehr als 100% erreicht werden. Hierbei spricht man auch von einem *superlinearen* Speedup oder *Superskalierung*.

Untersucht man den Einfluss des sequentiellen Anteils auf das Skalierungsverhalten, so stellt sich ein pessimistisches Resultat ein, welches als Gesetz von Amdahl (siehe [Amd67]) bekannt ist:

$$S(p) = \frac{1}{c_s + \frac{1-c_s}{p}}. \tag{6.1.2}$$

Hierbei beschreibt $0 \leq c_s \leq 1$ den nicht parallelisierten Anteil am Gesamtaufwand. Für wachsendes p ergibt sich somit ein beschränkter Speedup bzw. eine gegen Null tendierende parallele Effizienz des Verfahrens. Der Einsatz von weiteren Prozessoren führt in diesem Fall zu keiner wesentlichen Reduktion der Laufzeit.

Sowohl der sequentielle Anteil eines Verfahrens, als auch ein zusätzlicher Aufwand im parallelen Algorithmus, hervorgerufen z.B. durch Kommunikation, lassen sich allgemein als *Overhead* zusammenfassen. Formal ist dieser definiert als die Differenz der parallelen Laufzeit aller Prozessoren und des sequentiellen Aufwands:

$$t_o(p) = pt(p) - t(1).$$

Hiermit lässt sich die parallele Effizienz wie folgt umschreiben:

$$E(p) = \frac{1}{1 + \frac{t_o(p)}{t(1)}} \tag{6.1.3}$$

Desweiteren gelingt es, den Overhead über die Effizienz auszudrücken:

$$t_o(p) = \left(\frac{1}{E(p)} - 1\right) t(1)$$

Lässt sich nun zu einer gegebenen Effizienz $0 < E \leq 1$ stets ein Paar aus Problemgröße, und damit sequentieller Laufzeit, und Anzahl von Prozessoren finden, so dass das Verhältnis $t_o(p)/t(1)$ den Wert $(1/E) - 1$ annimmt, so spricht man von *Isoeffizienz* (siehe auch [KR87] und [GGKK03]). Dieses Konzept erlaubt eine allgemeinere Charakterisierung von parallelen Algorithmen. Dabei wird ein Verfahren *effizient* bzw. *skalierbar* genannt, falls ein solcher konstanter Quotienten in (6.1.3) erzielt werden kann.

Als Konsequenz aus dieser Definition folgt für alle skalierbaren Algorithmen ein Anwachsen der parallelen Effizienz bei steigender Problemgröße und konstanter Prozessoranzahl. Bei Verfahren mit einer Komplexität in $\mathcal{O}(1/p)$ ist diese Eigenschaft offensichtlich erfüllt. Aber auch Algorithmen mit sequentiellen Anteilen können effizient sein, falls letztere bei steigender Problemdimension geringer werden.

Für die Beurteilung der Skalierbarkeit erfolgt in den nachfolgenden numerischen Beispielen stets die Angabe der parallelen Effizienz für gegebene Paare von Prozessoranzahl und

Problemdimension. Bei einem Rechnersystem mit verteiltem Speicher lassen sich einige Probleme allerdings nur mit einer hinreichend großen Anzahl von Prozessoren und dem damit zur Verfügung stehenden Speicher lösen. In solchen Fällen gelingt es somit, in Ermangelung von Ergebnissen für $p = 1$, nicht, eine geeignete parallele Effizienz bzw. Skalierung zu definieren. Auf der anderen Seite lässt sich die Definition von $S(p)$ und $E(p)$ verallgemeinern, indem als Ausgangswert nicht die sequentielle Zeit, sondern ein $t(p')$ für ein $p' < p$ verwendet wird. Der Quotient

$$E(p', p) = \frac{p' \cdot t(p')}{p \cdot t(p)}. \tag{6.1.4}$$

bezeichne dann die parallele Effizienz *bezüglich* p'. Die Untersuchung der Isoeffizienz erfolgt hierbei analog.

6.2 Aufbau der Matrix

Eine Parallelisierung des Aufbaus der $\mathcal{H}$-Matrix ist aus verschiedenen Gründen sinnvoll. Zum einen ist z.B. die Konstruktion der Matrixblöcke im Fall von BEM-Anwendungen sehr aufwändig, da Auswertungen von komplexen Kernfunktionen oder Quadraturformeln notwendig sind (siehe Abschnitt 2.4.1). Zum anderen lässt sich durch eine Verteilung der $\mathcal{H}$-Matrix in der Aufbauphase eine für spätere Algorithmen, wie z.B. der Matrix-Vektor-Multiplikation, optimale Lastverteilung realisieren.

Die Art und Weise, in der die $\mathcal{H}$-Matrix aufgebaut wird, hängt zudem von den später verwendeten Algorithmen ab. So genügt es z.B. für die Matrix-Vektor-Multiplikation, wenn lediglich die Matrixblöcke konstruiert werden, die mit Blättern im Blockclusterbaum korrespondieren. Hingegen empfiehlt sich für die Matrix-Inversion der Aufbau der gesamten Hierarchie, d.h. auch von Blockmatrizen für die inneren Knoten.

Im ersten Fall bildet die sequentielle Vorschrift

for all $b \in \mathcal{L}(T)$ **do**
    konstruiere den Matrixblock zu b;

die Basis für ein paralleles Verfahren. Der Vergleich mit Algorithmus 5.2.1 offenbart eine große Ähnlichkeit zu einer Online-Scheduling-Methode, weshalb diese Variante unter Verwendung eines Threadpools beschrieben wird.

Der allgemeine Fall entspricht dagegen einem Graphendurchlauf durch den Blockclusterbaum T, wobei für jeden Knoten eine entsprechende Matrix erzeugt wird. In beiden Fällen ist die Generierung der Matrizen zu einem gegebenen Knoten in der Regel unabhängig von anderen Matrixblöcken. Dies bedeutet, dass beim Aufbau der $\mathcal{H}$-Matrix keinerlei Kommunikation notwendig ist.

6.2.1 Allgemeiner BSP-Algorithmus

Aufgrund der Arbeitsweise eines BSP-Rechners (siehe Abschnitt 5.2.2) empfiehlt sich die Verwendung eines Offline-Scheduling-Verfahrens. Die hierfür notwendige Verteilungsfunktion über dem Blockclusterbaum m sei als zulässig und konsistent vorausgesetzt (siehe Abschnitt 4.2). Für die Definition von m kann jeder der in Abschnitt 4 beschriebenen Algorithmen zur Lastbalancierung genutzt werden.

Da für die Konstruktion einer $\mathcal{H}$-Matrix mit konstanter Genauigkeit keine Kenntnisse über die auftretenden Kosten vorliegen, beschränkt sich die folgende Diskussion auf den Aufbau von $\mathcal{H}$-Matrizen mit einem festen Rang k. Die Kosten sind hierbei bestimmt durch den Aufwand zur Berechnung einer Matrix über einem Knoten v:

$$c_{\mathrm{MB}}(v,k) = \begin{cases} c_{\mathrm{MB,z}}(v,k), & v \in \mathcal{L}_{\mathrm{z}} \\ c_{\mathrm{MB,nz}}(v), & v \in \mathcal{L}_{\mathrm{nz}} \\ c, & \text{sonst} \end{cases}.$$

Die Funktionen $c_{\mathrm{MB,z}}$ und $c_{\mathrm{MB,nz}}$ stellen applikationsabhängige Kostenfunktionen zur Berechnung eines Matrixblockes über zulässigen bzw. nichtzulässigen Blockclustern dar. Für das BEM-Beispiel aus Abschnitt 2.4.1 ergeben sich z.B. unter Verwendung des ACA-Algorithmus:

$$c_{\mathrm{MB,z}}(\tau,\sigma,k) = c_1 k^2(|\tau| + |\sigma|) \quad \text{und} \quad c_{\mathrm{MB,nz}}(\tau,\sigma) = c_2|\tau| \cdot |\sigma|.$$

Die Konstante $c \in \mathbb{R}_{\geq 0}$ repräsentiert den Aufwand zum Aufbau der Datenstrukturen einer Blockmatrix zur Darstellung von inneren Knoten in der $\mathcal{H}$-Matrix. Abhängig von der jeweiligen Implementierung und dem verwendeten Rechnersystem sind außerdem die Konstanten c_1 und c_2 entsprechend anzupassen.

Der Graphendurchlauf zur Konstruktion der $\mathcal{H}$-Matrix erfolgt rekursiv und entspricht einer Tiefensuche. Die Rekursion stoppt, sobald ein Knoten erreicht wird, welcher nicht dem lokalen Prozessor zugewiesen wurde. Durch die Konsistenzeigenschaft der Verteilungsfunktion ist hierbei sichergestellt, dass alle lokalen Matrizen erzeugt werden.

```
procedure build (v, q)
    if m(v) ≠ ⊥ ∧ m(v) ≠ q then
        return ;
    if v ∈ L(T) then
        konstruiere Matrixblock zu v;
    else
        konstruiere Blockmatrix zu v;
        for all v' ∈ S(v) do
            build( v', q );
    endif;
    bsp_sync();
end;
```

Algorithmus 6.2.1: Paralleler $\mathcal{H}$-Matrixaufbau mit Hierarchie

Für die Komplexität dieses Algorithmus ergibt sich mit obiger Kostenfunktion das folgende Resultat:

Lemma 6.2.1 *Sei $M \in \mathcal{H}(T,k)$ und sei $\mathcal{W}_{\mathcal{H},\mathrm{MB}}(M)$ der sequentielle Aufwand zur Konstruktion von M. Dann lautet die Komplexität zur parallelen Konstruktion mit Hierarchie von M durch Algorithmus 6.2.1:*

$$\mathcal{W}_{\mathcal{H},\mathrm{MB}}(M,p) = \mathcal{O}\left(|V(T)| - |\mathcal{L}(T)|\right) + \frac{\mathcal{W}_{\mathcal{H},\mathrm{MB}}(M)}{p} + l. \tag{6.2.1}$$

Beweis: Für $\mathcal{W}_{\mathcal{H},\mathrm{MB}}(M)$ gilt:

$$\begin{aligned}
\mathcal{W}_{\mathcal{H},\mathrm{MB}}(M) &= \sum_{v \in V(T)} c_{\mathrm{MB}}(v,k) \\
&= \sum_{v \in V(T)\setminus\mathcal{L}(T)} c_{\mathrm{MB}}(v,k) + \sum_{v \in \mathcal{L}(T)} c_{\mathrm{MB}}(v,k) \\
&= c \cdot |V(T) \setminus \mathcal{L}(T)| + \sum_{v \in \mathcal{L}(T)} c_{\mathrm{MB}}(v,k).
\end{aligned}$$

Da jeder Prozessor höchstens für alle inneren Knoten von T Blockmatrizen generieren muss, folgt ein Gesamtaufwand von

$$\frac{1}{p}\left(p\,\mathcal{O}\left(|V(T)| - |\mathcal{L}(T)|\right) + \sum_{v \in \mathcal{L}(T)} c_{\mathrm{MB}}(v,k)\right) \leq \mathcal{O}\left(|V(T)| - |\mathcal{L}(T)|\right) + \frac{\mathcal{W}_{\mathcal{H},\mathrm{MB}}(M)}{p}.$$

□

Das Ergebnis (6.2.1) impliziert einen sequentiellen Anteil des parallelen Algorithmus, womit ein optimaler paralleler Speedup nicht möglich ist. Allerdings ist c im Vergleich zum Aufwand für die Berechnung eines Matrixblocks verschwindend klein, weshalb die parallele Effizienz von Algorithmus 6.2.1 praktisch optimale Werte erreicht.

6.2.1.1 Numerische Beispiele

Die Basis für die Beispiele in diesem Abschnitt bildet das BEM-Beispiel aus Abschnitt 2.4.1. Der Parameter η der Zulässigkeitsbedingung (2.4.4) besitzt dabei zunächst den Wert $\eta = 1$. Aufgrund des begrenzten Speichers des Parallelrechners konnte bei der größten Problemdimension nur mit minimal 4 CPUs gerechnet werden. In diesem Fall ist die parallele Effizienz bezüglich $p = 4$ angegeben.

Matrix-Aufbau, konstanter Rang $k = 10$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	14.1	3.6	98.1	1.8	96.0	1.3	93.7	1.0	91.5
7 920	34.7	8.9	97.8	4.4	97.6	3.1	94.1	2.4	92.2
19 320	103.8	26.4	98.3	13.3	97.4	9.2	94.3	7.1	91.7
43 680	278.4	70.2	99.1	36.5	95.4	24.5	94.7	18.9	91.9
89 400	655.4	168.8	97.1	83.6	98.0	56.8	96.1	44.6	91.9
184 040	–	390.3	–	198.3	*98.4*	132.9	*97.9*	104.2	*93.7*

Obwohl eine leichte Abhängigkeit von der Prozessoranzahl festzustellen ist, gelingt eine effiziente Lastbalancierung. Somit stimmt die gewählte Kostenfunktion mit dem in der Praxis auftretenden Aufwand im wesentlichen überein. Der Verlust an paralleler Effizienz bei steigendem p ist zum einen auf kleine Abweichungen der Kosten für den Aufbau vollbesetzter bzw. R-Matrizen zurückzuführen, etwa durch eine Abhängigkeit von c_1 und c_2 von der Blockgröße (siehe auch Abschnitt 4.3.2). Zum anderen ergibt sich durch das gewählte Abbruchkriterium (4.2.2) aus Abschnitt 4.2.3 eine geringe Ineffizienz. Der Aufwand zum Aufbau der Hierarchie ist dagegen vernachlässigbar.

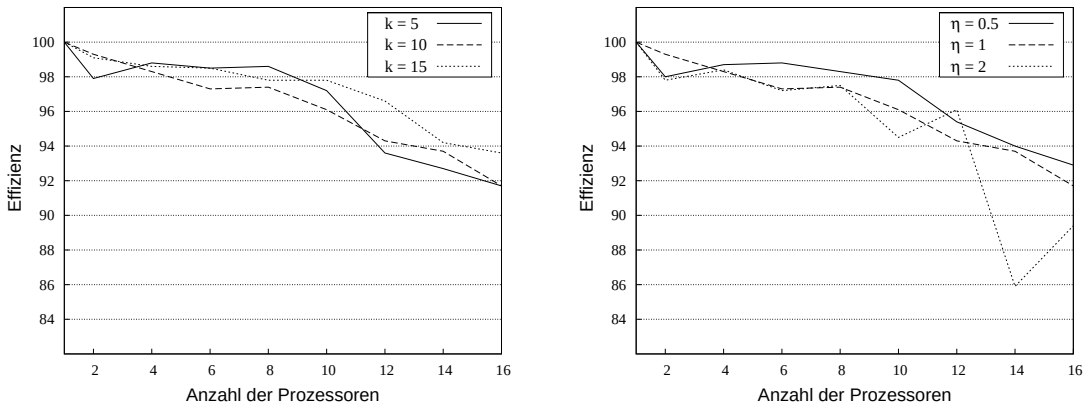


Abbildung 6.2.1: Abhängigkeit der parallelen Effizienz von k (links) und η (rechts)

In Abbildung 6.2.1 ist parallele Effizienz des Matrixaufbaus bei einer Problemdimension von $n = 19\,320$ für verschiedene Ränge bzw. Werte von η aufgetragen. Aufgrund von Beschränkungen des verwendeten Rechnersystems konnte für diesen Test keine größere Problemdimension genutzt werden.

Man erkennt nur eine geringe Abhängigkeit der Effizienz von diesen beiden Variablen, wobei die Schwankungen im wesentlichen durch Eigenschaften des Parallelrechners hervorgerufen werden. Insbesondere bei $\eta = 2$ führt die kleine Problemgröße zu sehr geringen Laufzeiten und damit zu entsprechenden Messungenauigkeiten.

6.2.2 Aufbau einer $\mathcal{H}$-Matrix mittels Threadpool

Wie eingangs bereits angedeutet, eignet sich das Online-Scheduling besonders gut zur Konstruktion einer $\mathcal{H}$-Matrix ohne Hierarchie. Unter Verwendung der in Abschnitt 5.2.1.1 definierten Funktionen lautet der entsprechende parallele Algorithmus:

```
procedure build_tp( T )
    procedure build_leaf ( v )
        konstruiere Matrixblock zu v;

    for all v ∈ L(T) do
        tp_run( build_leaf( v ) );
    tp_sync_all();
end;
```

Algorithmus 6.2.2: Paralleler Aufbau einer $\mathcal{H}$-Matrix ohne Hierarchie

Die Komplexität dieses Algorithmus ist nach der Argumentation zu Lemma 6.2.1 bestimmt durch

$$\mathcal{W}_{\mathcal{H},\mathrm{MB}}(M,p) = \frac{\mathcal{W}_{\mathcal{H},\mathrm{MB}}(M)}{p}$$

und somit optimal.

Online-Scheduling eignet sich aber auch zur Konstruktion einer $\mathcal{H}$-Matrix mit Hierarchie. Da hierbei allerdings die Verteilungsfunktion nicht explizit gegeben ist, kann eine entsprechende Rekursion nur bei den Blättern von T beendet werden, d.h. der Aufbau der Hierarchieinformation erfolgt sequentiell. Im allgemeinen führt dies zu einem, im Vergleich zu Algorithmus 6.2.1 leicht erhöhten Aufwand. Insgesamt läßt sich die Komplexität dieses Verfahrens aber weiterhin durch (6.2.1) beschreiben.

Der so definierte, parallele Matrixaufbau mit Hierarchie ist in Algorithmus 6.2.3 zusammengefaßt.

```
procedure build_tp_hier( T )
    procedure build_leaf ( v )
        konstruiere Matrixblock zu v;
    end;

    procedure build ( v )
        if v ∈ L(T) then tp_run( build_leaf( v ) );
        else
            konstruiere Blockmatrix zu v;
            for all v' ∈ S(v) do build( v' );
        endif;
    end;

    build( root(T) );
    tp_sync_all();
end;
```

Algorithmus 6.2.3: Paralleler Aufbau einer $\mathcal{H}$-Matrix mit Hierarchie

6.2.2.1 Numerische Beispiele

Das gleiche Beispiel wie im Falle eines BSP-Rechners wurde auch für die Berechnung auf einer PRAM gewählt. Durch das Online-Scheduling und die damit verbundene Unabhängigkeit von einer Kostenfunktion besteht allerdings die Möglichkeit, auch mit variablem Rang, d.h. fester Genauigkeit zu rechnen.

Zunächst sind aber in der nachfolgenden Tabelle die Laufzeiten des Matrixaufbaus für einen festen Rang von $k = 10$ und dem Zulässigkeitsparameter $\eta = 1$ dargestellt.

Matrix-Aufbau, konstanter Rang $k = 10$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	10.3	2.6	97.9	1.4	95.0	0.9	95.7	0.7	95.0
7 920	25.3	6.4	98.3	3.2	97.7	2.2	96.8	1.7	95.9
19 320	75.5	19.4	97.5	9.7	97.6	6.5	97.1	4.9	96.3
43 680	202.3	51.0	99.1	25.7	98.5	17.2	97.8	13.0	97.6
89 400	481.7	120.0	100.3	61.7	99.8	41.2	99.1	30.6	98.4
184 040	1146.4	282.1	101.6	141.5	101.3	94.8	100.7	71.6	100.1

Bei allen Problemdimensionen ist eine sehr hohe parallele Effizienz zu beobachten. Desweiteren ist nur eine sehr kleine Abhängigkeit der Effizienz von der Prozessoranzahl, verursacht durch den sequentiellen Aufbau der Hierarchie, festzustellen. In einzelnen Fällen tritt sogar ein superlinearer Speedup auf.

Die Ergebnisse für das gleiche Beispiel bei Verwendung einer konstanten Genauigkeit von $\varepsilon = 10^{-4}$ lauten:

Matrix-Aufbau, konstante Genauigkeit $\varepsilon = 10^{-4}$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	14.6	3.7	97.9	1.9	97.1	1.3	96.6	1.0	96.1
7 920	35.6	9.1	97.9	4.6	97.0	3.1	97.0	2.3	96.8
19 320	109.0	27.8	98.0	13.9	97.8	9.3	97.5	7.1	95.4
43 680	299.8	76.0	98.6	38.2	98.0	25.6	97.5	19.5	96.2
89 400	707.5	179.1	98.8	89.9	98.3	60.1	98.1	45.3	97.6
184 040	1747.6	426.8	102.4	213.9	102.1	143.1	101.8	108.1	101.0

Hierbei ergibt sich ein ähnliches Bild wie im Falle eines konstanten Ranges. Insbesondere die hohe parallele Effizienz und die Unabhängigkeit von der Anzahl der Prozessoren wird auch mit einer festen Genauigkeit erreicht.

Der Einfluss des Ranges und der Genauigkeit auf die Effizienz des Algorithmus wird in Abbildung 6.2.2 dargestellt. Die Problemdimension beträgt hierbei $n = 43\,680$.

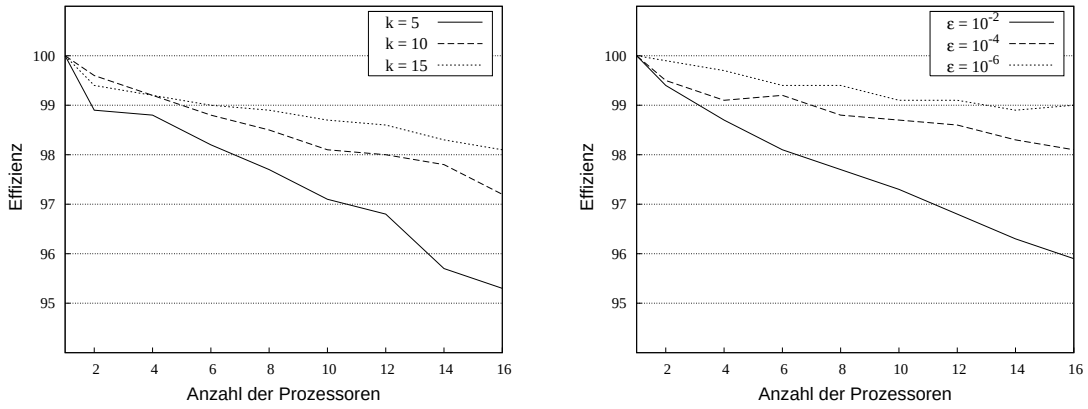


Abbildung 6.2.2: Abhängigkeit der parallelen Effizienz von k (links) und ε (rechts)

In beiden Fällen ergeben sich ähnliche Ergebnisse, wobei mit steigendem Rang bzw. höherer Genauigkeit auch die Effizienz wächst. Allerdings sind die Unterschiede vergleichsweise gering.

6.3 Matrix-Vektor-Multiplikation

In der Praxis wird die Matrix-Vektor-Multiplikation $y = Ax$ mit $A \in \mathcal{H}(T)$ und $x \in \mathbb{R}^J$, $y \in \mathbb{R}^I$ häufig in Verbindung mit einer Vektoraddition und Vektorskalierung benutzt. Aus diesem Grund wird, wie auch im sequentiellen Fall im folgenden die erweiterte Matrix-Vektor-Multiplikation

$$y := \alpha \cdot A \cdot x + \beta \cdot y \tag{6.3.1}$$

mit $\alpha, \beta \in \mathbb{R}$ betrachtet. Um Vektoroperationen wie eine Addition oder ein Skalarprodukt mit einer optimalen parallelen Komplexität berechnen zu können, wird für die Vektoren x und y eine gleichmäßige Verteilung über den p Prozessoren angenommen. Die entsprechenden Anteile von x und y, die Prozessor q zugewiesen werden, seien durch die Indexmengen

$$J_q = \left\{ \frac{qm}{p}, \ldots, \frac{(q+1)m}{p} - 1 \right\} \quad \text{und } I_q = \left\{ \frac{qn}{p}, \ldots, \frac{(q+1)n}{p} - 1 \right\} \tag{6.3.2}$$

definiert. Der Einfachheit halber wird hierbei angenommen, dass n bzw. m Vielfache von p sind. Die lokalen Vektoren seien mit $x_q = x|_{J_q}$ und $y_q = y|_{I_q}$ bezeichnet.

In den folgenden Kapiteln werden verschiedene Algorithmen für die Matrix-Vektor-Multiplikation vorgestellt. Den Anfang macht dabei in Abschnitt 6.3.1 ein Verfahren, welches den sequentiellen Algorithmus als Ausgangspunkt verwendet. Obwohl die resultierende parallele Methode keine optimale parallele Effizienz garantiert, ist sie für viele Anwendungszwecke hinreichend. Außerdem gestattet der Algorithmus die Verwendung einer existierenden sequentiellen Implementierung.

Ein anderer Zugang wird in Abschnitt 6.3.2 gewählt, um die Schwierigkeiten des ersten Verfahrens zu überwinden und einen Algorithmus zu konstruieren, der eine optimale Effizienz ermöglicht.

In beiden Algorithmen kommen Offline-Scheduling-Verfahren bei der Lastbalancierung zum Einsatz. Deshalb werden die Methoden primär für den allgemeineren Fall eines BSP-Computers beschrieben. Die Angabe von Algorithmen für eine PRAM erfolgt nachträglich als entsprechend modifizierte Verfahren.

Aufgrund des Offline-Schedulings wird für alle vorgestellten Algorithmen eine Kostenfunktion benötigt. Hierbei bilden die in Abschnitt 2.3 beschriebenen Darstellungen der Niedrigrangmatrizen als $\mathsf{R}(k)$-Matrix und die D-Repräsentation für vollbesetzte Matrizen die Grundlage. Damit ergibt sich die folgende Funktion für den Aufwand der Matrix-Vektor-Multiplikation in jedem Matrixblock:

$$c_{\mathrm{MV}}(\tau, \sigma, k) = \begin{cases} c_1 k \cdot (|\tau| + |\sigma|), & (\tau, \sigma) \in \mathcal{L}_{\mathrm{z}}(T) \\ c_2 |\tau| \cdot |\sigma|, & (\tau, \sigma) \in \mathcal{L}_{\mathrm{nz}}(T) \end{cases}. \tag{6.3.3}$$

Auch hierbei gilt es, die Konstanten c_1 und c_2 je nach Rechnersystem anzupassen.

Bei hierarchischen Verteilungsalgorithmen, wie sie in Abschnitt 4.2.3 vorgestellt wurden, ist diese Funktion ebenfalls für Blockmatrizen zu definieren:

$$c_{\mathrm{MV},h}(\tau, \sigma, k) = \begin{cases} c_{\mathrm{MV}}(\tau, \sigma, k), & (\tau, \sigma) \in \mathcal{L}(T) \\ \sum_{b \in \mathcal{S}(\tau,\sigma)} c_{\mathrm{MV}}(b, k), & \text{sonst} \end{cases}. \tag{6.3.4}$$

6.3.1 Matrix-Vektor-Multiplikation ohne Blockaufteilung

Der erste und zugleich einfachere Algorithmus geht auf die vollbesetzte Matrix-Vektor-Multiplikation zurück, wie sie in [McC95b] beschrieben wird. Hierbei ist die Matrix $D \in \mathbb{R}^{I \times J}$

in p Blöcke der Dimension $\frac{n}{\sqrt{p}} \times \frac{m}{\sqrt{p}}$ aufgeteilt, wobei $J_{q,\mathrm{mat}}$ und $I_{q,\mathrm{mat}}$ die Blockindexmenge von Prozessor q definieren. Die jeweiligen Matrixblöcke seien mit $D_q = D|_{I_{q,\mathrm{mat}} \times J_{q,\mathrm{mat}}}$ bezeichnet.

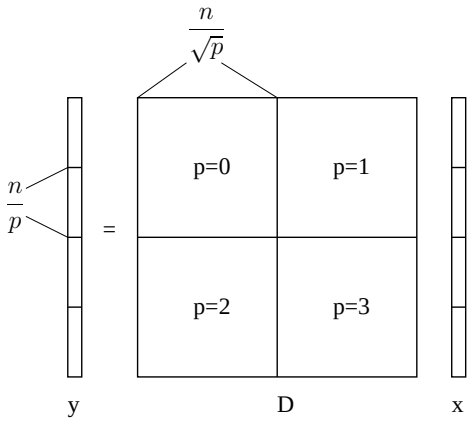


Abbildung 6.3.1: Aufteilung von x, y und D bei der vollbesetzten Multiplikation

Die Multiplikation erfolgt in 3 BSP-Schritten. Zunächst wird y_i, $0 \leq i < p$, mit β multipliziert und x_i an alle Prozessoren gesendet, die diesen für die lokale Matrix-Vektor-Multiplikation benötigen. Die Kosten für diesen Schritt betragen $\mathcal{O}(n/p)$ für die Skalierung von y, $g \cdot n/p$ für das Versenden und $g \cdot m/\sqrt{p}$ für Empfangen von x.

Im zweiten BSP-Schritt erfolgt die eigentliche Multiplikation mit einen Aufwand von $\mathcal{O}(nm/p)$. Die Elemente des lokalen Ergebnisvektor y_i' werden anschließend an die jeweiligen Prozessoren versandt. Hierbei entstehen $g \cdot n/\sqrt{p}$ Kosten.

Der letzte BSP-Schritt besteht darin, die lokalen Ergebnisse in den einzelnen Prozessoren auf y_i aufzuaddieren. Für jedes Element von y_i ergeben sich hierbei $\sqrt{p}$ Summanden, womit Kosten von $\mathcal{O}\left(n/\sqrt{p}\right)$ resultieren.

Bemerkung 6.3.1 *Die Berechnung des Produktes* (6.3.1) *mit einer vollbesetzten Matrix* $A \in \mathbb{R}^{I \times J}$ *auf einem BSP-Rechner mit p Prozessoren besitzt eine Komplexität von*

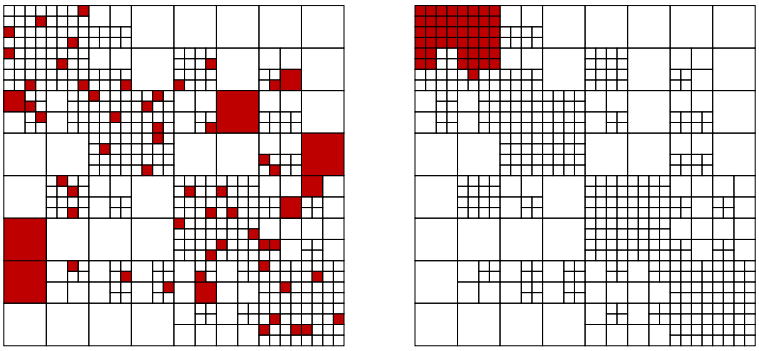
$$\mathcal{W}_{\mathrm{MV,d}}(A,p) = \mathcal{O}\left(\frac{nm}{p}\right) + g \cdot \mathcal{O}\left(\frac{\max\{n,m\}}{\sqrt{p}}\right) + l \cdot 3. \tag{6.3.5}$$

Der gleiche Algorithmus lässt sich für die Matrix-Vektor-Multiplikation von $\mathcal{H}$-Matrizen nutzen. Der einzige Unterschied besteht in der Verteilung der Matrix, da die $\frac{n}{\sqrt{p}} \times \frac{m}{\sqrt{p}}$ Blockaufteilung für $\mathcal{H}$-Matrizen im allgemeinen nicht optimal bezüglich der Kosten (6.3.3) bzw. (6.3.4) ist.

```
procedure dense_mul( i, α, D_i, x_i, β, y_i )
    { Schritt 1 }
    y_i := β · y_i;
    for all 0 ≤ q < p mit J_i ∩ J_{q,mat} ≠ ∅ do
        bsp_send( q, x_i|_{J_i ∩ J_{q,mat}} );
    bsp_sync();
    { Schritt 2 }
    while bsp_nmsgs() > 0 do
        x_q := bsp_recv(); X_i := X_i ∪ {x_q};
    endwhile;
    x'_i := x_i + Σ_{x_q ∈ X_i} x_q;
    y'_i := α D_i x'_i;
    for all 0 ≤ q < p mit I_i ∩ I_{q,mat} ≠ ∅ do
        bsp_send( q, y'_i|_{I_i ∩ I_{q,mat}} );
    bsp_sync();
    { Schritt 3 }
    while bsp_nmsgs() > 0 do
        y_q := bsp_recv(); Y_i := Y_i ∪ {y_q};
    endwhile;
    y_i := y_i + Σ_{y_q ∈ Y_i} y_q;
    bsp_sync();
end;
```

Algorithmus 6.3.1: Vollbesetzte Matrix-Vektor-Multiplikation

Außerdem ist die Verteilung der $\mathcal{H}$-Matrix entscheidend für die Kommunikationskosten während der Multiplikation. Als Beispiel seien die beiden Verteilungen in Abbildung 6.3.2 genannt. Beide Darstellungen zeigen die Matrixblöcke, die einem Prozessor zugewiesen wurden und bzgl. der Multiplikation identische Kosten verursachen.



(I) **(II)**

Abbildung 6.3.2: Unterschiedliche Verteilungen für $\mathcal{H}$-Matrizen

Der wesentliche Unterschied zwischen beiden Verteilungen besteht in der „Kompaktheit“ der Menge der lokalen Matrizen. Während in der Verteilung (I) Matrizen aus allen Teilen der

Produktindexmenge $I \times J$ auftreten, bilden die Matrizen in der Verteilung (II) eine kompakte Menge. Dies hat Auswirkungen auf die bei der Multiplikation auftretende Kommunikation. Im Fall (I) ist üblicherweise der lokale Vektor x_i an alle anderen Prozessoren zu senden. Ähnliches gilt für den Ergebnisvektor y'_i, welcher im allgemeinen eine Dimension von $\mathcal{O}(n)$ hat. Die Summation im letzten Schritt von Algorithmus 6.3.1 erfolgt schließlich über $\mathcal{O}(p)$ Summanden und erzeugt damit einen Aufwand von $\mathcal{O}(pn/p) = \mathcal{O}(n)$. Die Gesamtkomplexität der Multiplikation beläuft sich damit auf

$$\mathcal{W}_{\mathrm{MV}}(A,p) = \mathcal{O}\left(\frac{\mathcal{W}_{\mathrm{MV}}(A)}{p} + n\right) + g \cdot \mathcal{O}(\max\{n,m\}) + l \cdot 3.$$

Hierbei wurde von Vermutung 4.3.3 Gebrauch gemacht, da die eigentliche Multiplikation in den Matrixblöcken völlig unabhängig voneinander stattfinden kann, wodurch sich ein Aufwand von $\mathcal{W}_{\mathrm{MV}}(A)/p$ ergibt. Die Gesamtkomplexität der Matrix-Vektor-Multiplikation ist allerdings weder bezüglich der Berechnung noch der Kommunikation optimal.

Bei der Verteilung (II) ist der Anteil des Vektors x, der für die lokale Multiplikation benötigt wird, wesentlich geringer. Gleiches gilt für die Dimension des lokalen Ergebnisvektors und somit für die Anzahl der Summanden im finalen Schritt. Folglich ist auch die Komplexität der Matrix-Vektor-Multiplikation kleiner als bei Verteilung (I).

6.3.1.1 Teilungsgrad und Verteilungsdurchmesser

Der bisher nur anschaulich definierte Begriff der Kompaktheit von Knotenmengen soll im folgenden präzisiert werden. Dies bezieht sich dabei insbesondere auf die prozessorlokalen Knotenmengen, wie sie durch verschiedene Verteilungen bestimmt sind.

Betrachtet man einen einzelnen Index $i \in I$, so ist der Aufwand für das Versenden des Vektors x, das Empfangen der Vektoren y'_q und die Vektorsummation bzgl. i definiert durch die Anzahl von Prozessoren, die diesen Index in einem Matrixblock beinhalten. Dies gibt Anlass zu folgender Definition.

Definition 6.3.2 *Es bezeichne $\mathcal{L}(T,q)$ die Blätter von T, die Prozessor q zugewiesen wurden. Dann ist der* Teilungsgrad $d_{\mathrm{sh}}(i)$ *des Index $i \in I \cup J$ wie folgt definiert:*

$$d_{\mathrm{sh}}(i) = \max\{d^{\mathrm{z}}_{\mathrm{sh}}(i), d^{\mathrm{s}}_{\mathrm{sh}}(i)\}, \tag{6.3.6}$$

mit

$$\begin{aligned} d^{\mathrm{z}}_{\mathrm{sh}}(i) &= |\{q \,|\, \exists(\tau,\sigma) \in \mathcal{L}(T,q) : i \in \tau\}| \quad \textit{und} \\ d^{\mathrm{s}}_{\mathrm{sh}}(i) &= |\{q \,|\, \exists(\tau,\sigma) \in \mathcal{L}(T,q) : i \in \sigma\}|. \end{aligned}$$

Weiterhin sei $d_{\mathrm{sh}} = \max_{i \in I} d_{\mathrm{sh}}(i)$ der Teilungsgrad der Indexmenge I.

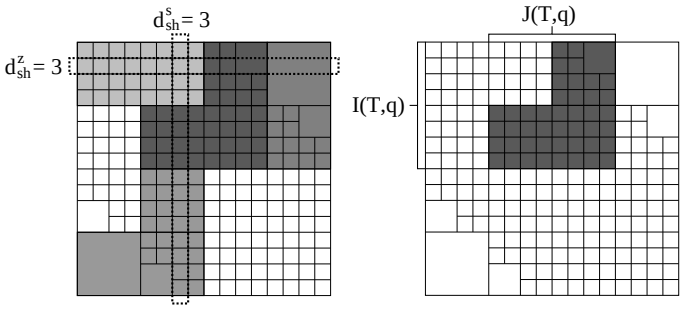


Abbildung 6.3.3: Teilungsgrad und Verteilungsdurchmesser

Im Falle einer vollbesetzten Matrix ergibt sich ein Wert von $d_{\mathrm{sh}} = \sqrt{p}$, womit sich die Komplexität (6.3.5) umformulieren lässt zu

$$\mathcal{W}_{\mathrm{MV,d}}(A,p) = \mathcal{O}\left(\frac{nm}{p} + \frac{d_{\mathrm{sh}} n}{p}\right) + g \cdot \mathcal{O}\left(\frac{d_{\mathrm{sh}} \max\{n,m\}}{p}\right) + l \cdot 3.$$

Bemerkung 6.3.3 *Bei einer schwachbesetzten Matrix, wie sie im FEM-Beispiel in Abschnitt 2.4.2 auftritt, ist die Anzahl von Matrixkoeffizienten pro Index beschränkt durch eine Konstante c. Somit gilt in diesem Fall auch für den Teilungsgrad:* $d_{\mathrm{sh}} \leq c$.

Für $\mathcal{H}$-Matrizen ist dagegen die Angabe von d_{sh} nicht ausreichend, um die Komplexität vollständig zu formulieren, da sich die Größe des empfangenen Anteils von x und des lokalen Ergebnisvektors hierdurch nicht beschreiben lässt. Beide Größen sind ausschließlich von der Menge der prozessorlokalen Blöcke abhängig. Dies führt zurück zu dem oben genannten Kompaktheitsbegriff, welcher nun präzisiert wird.

Definition 6.3.4 *Für einen Blockclusterbaum* $T = T(I \times J)$ *und einen Prozessor* q, $0 \leq q < p$, *seien* $I(T,q)$ *bzw.* $J(T,q)$ *definiert als:*

$$\begin{aligned} I(T,q) &= \{i \in I \,|\, \exists (\tau,\sigma) \in \mathcal{L}(T,q) : i \in \tau\} \quad \text{und} \\ J(T,q) &= \{i \in J \,|\, \exists (\tau,\sigma) \in \mathcal{L}(T,q) : i \in \sigma\}. \end{aligned}$$

Seien weiterhin

$$\mathrm{diam_z}(T) = \max_{0 \leq q < p} |I(T,q)| \quad \text{und} \quad \mathrm{diam_s}(T) = \max_{0 \leq q < p} |J(T,q)|.$$

Dann wird $\mathrm{diam}(T) = \max\{\mathrm{diam_z}(T), \mathrm{diam_s}(T)\}$ Verteilungsdurchmesser *von* T *genannt. Die Werte* $\mathrm{diam_z}(T)$ *und* $\mathrm{diam_s}(T)$ *heißen auch* Zeilen- *bzw.* Spaltendurchmesser *von* T.

Der Spaltendurchmesser entspricht der Dimension des Anteils von x, der für die Multiplikation notwendig ist. Die Größe des lokalen Ergebnisvektors y_i' wird dagegen durch den

Zeilendurchmesser definiert. Der Aufwand für die Kommunikation, d.h. das Empfangen von x und Senden von y_i im zweiten Schritt von Algorithmus 6.3.1 ist somit durch $\mathrm{diam}(T)$ bestimmt.

Mit Hilfe von Definition 6.3.4 lässt sich Algorithmus 6.3.1 an die veränderten Indexmengen anpassen. Das Ergebnis ist der parallele Matrix-Vektor-Multiplikations-Algorithmus 6.3.2 für $\mathcal{H}$-Matrizen. Dessen Komplexität kann durch $d_{\mathrm{sh}}(T)$ und $\mathrm{diam}(T)$ wie folgt formuliert werden:

Bemerkung 6.3.5 *Die Matrix-Vektor-Multiplikation* (6.3.1) *mit einer $\mathcal{H}$-Matrix $A \in \mathcal{H}(T)$ mittels Algorithmus 6.3.2 auf p Prozessoren hat einen Aufwand von*

$$\mathcal{W}_{\mathrm{MV}}(A,p) = \mathcal{O}\left(\frac{\mathcal{W}_{\mathrm{MV}}(A)}{p} + \frac{d_{\mathrm{sh}} n}{p}\right) + g \cdot \mathcal{O}\left(\frac{d_{\mathrm{sh}} \max\{n,m\}}{p} + \mathrm{diam}(T)\right) + l \cdot 3. \quad (6.3.7)$$

Das Ziel eines Algorithmus zur Lastbalancierung der Matrix-Vektor-Multiplikation muss es somit sein, die Werte von $d_{\mathrm{sh}}(T)$ und $\mathrm{diam}(T)$ zu minimieren.

```
procedure mv_mul( i, α, A_i, x_i, β, y_i )
    { Schritt 1 }
    y_i := β · y_i;
    for all 0 ≤ q < p mit J_i ∩ J(T,q) ≠ ∅ do
        bsp_send( q, x_i|_{J_i ∩ J(T,q)} );
    bsp_sync();
    { Schritt 2 }
    while bsp_nmsgs() > 0 do
        x_q := bsp_recv(); X_i := X_i ∪ {x_q};
    endwhile;
    x'_i := x_i + Σ_{x_q ∈ X_i} x_q;
    y'_i := α A_i x'_i;
    for all 0 ≤ q < p mit I_i ∩ I(T,q) ≠ ∅ do
        bsp_send( q, y'_i|_{I_i ∩ I(T,q)} );
    bsp_sync();
    { Schritt 3 }
    while bsp_nmsgs() > 0 do
        y_q := bsp_recv(); Y_i := Y_i ∪ {y_q};
    endwhile;
    y_i := y_i + Σ_{y_q ∈ Y_i} y_q;
    bsp_sync();
end;
```

Algorithmus 6.3.2: Parallele Matrix-Vektor-Multiplikation für $\mathcal{H}$-Matrizen

6.3.1.2 Lastbalancierung ohne Kommunikation

Für die Lastbalancierung der Matrix-Vektor-Multiplikation wird zunächst davon ausgegangen, dass die Kosten der eigentlichen Multiplikation gegenüber dem Kommunikationsaufwand dominieren. Die verwendeten Algorithmen berücksichtigen deshalb nicht direkt den

Aufwand für das Versenden oder Empfangen eines Vektorelements. Das Hauptaugenmerk liegt somit auf der Reduktion von $d_{\mathrm{sh}}(T)$.

Eine Bedingung für die Minimierung von $d_{\mathrm{sh}}(T)$ sind möglichst „kompakte“ Mengen $\mathcal{L}(T, q)$, d.h. eine große Nähe der prozessorlokalen Matrizen (vgl. Verteilung (II) in Abbildung 6.3.2). Dies erinnert an die Eigenschaften von raumfüllenden Kurven im Zusammenhang mit Sequenzpartitionierung, wie diese in Abschnitt 4.2.2 diskutiert wurden. Dort kam die Eigenschaft raumfüllender Kurven zum Tragen, bei der aneinandergrenzende Teilintervalle im allgemeinen auf benachbarte Flächenelemente abgebildet werden. Die Teilsequenzen, die sich durch die Sequenzpartitionierung ergeben, führten damit auch zu kompakten Mengen von Blockclustern.

Der sich hierdurch ergebende Vorteil für $d_{\mathrm{sh}}(T)$ ist in Abbildung 6.3.4 dargestellt. Hierbei wurden sowohl LPT-Scheduling als auch Sequenzpartitionierung genutzt, um eine Lastbalancierung zu berechnen. Dabei führt das LPT-Scheduling zur Verteilung (I) in Abbildung 6.3.2.

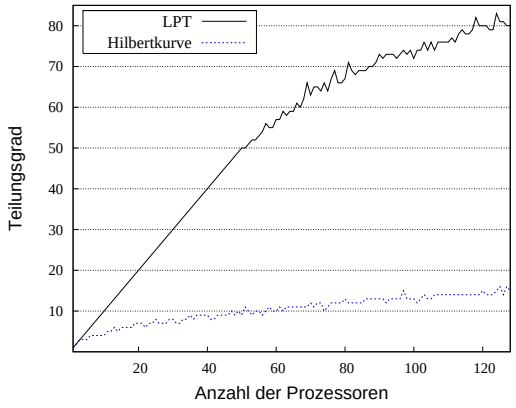


Abbildung 6.3.4: Teilungsgrad für LPT-Scheduling und Hilbert-Kurve

Deutlich erkennbar ist die lineare Abhängigkeit des Teilungsgrades von der Anzahl der Prozessoren bei Verwendung des LPT-Algorithmus. Lediglich für vergleichsweise große Werte von p reduziert sich die Ordnung von $d_{\mathrm{sh}}(T)$. Der Grund hierfür liegt in den mit wachsendem p kleiner werdenden Mengen $\mathcal{L}(T, q)$. Hierdurch sinkt die Wahrscheinlichkeit, dass ein Index von allen Prozessoren geteilt wird. Dagegen nimmt der Teilungsgrad der mittels Sequenzpartitionierung bestimmten Verteilung bereits bei kleinem p deutlich reduzierte Werte an.

Eine genauere Untersuchung des Teilungsgrades für die verschiedenen, in Abschnitt 4.2.2 vorgestellten Kurven ergibt ein $\mathcal{O}\left(\sqrt{p}\right)$-Verhalten für $d_{\mathrm{sh}}(T)$. Die jeweils gefitteten Ergebnisse für die einzelnen Kurven sind in den Abbildungen 6.3.5 und 6.3.6 dargestellt. Hierbei ergibt sich ein leichter Vorteil bei Verwendung der Hilbert- bzw. der zu dieser ähnlichen Moore-Kurve. Die Ursache hierfür ist in der größeren Lokalität der jeweils abgebildeten In-

tervalle bei diesen Kurventypen zu finden, welche bei der Z- und der Lebesgue-Kurve durch Sprünge unterbrochen wird (vergleiche Abbildung 4.2.3 bzw. Abbildung 4.2.4).

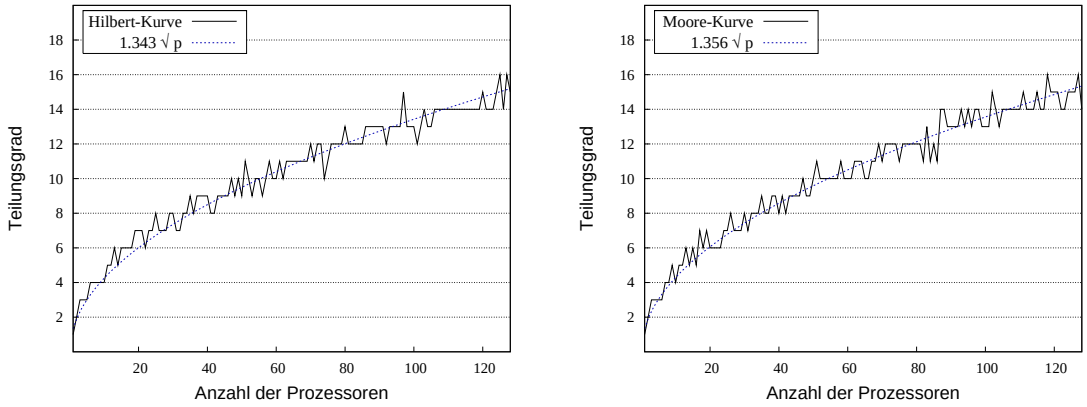


Abbildung 6.3.5: Teilungsgrad für Hilbert- und Moore-Kurve

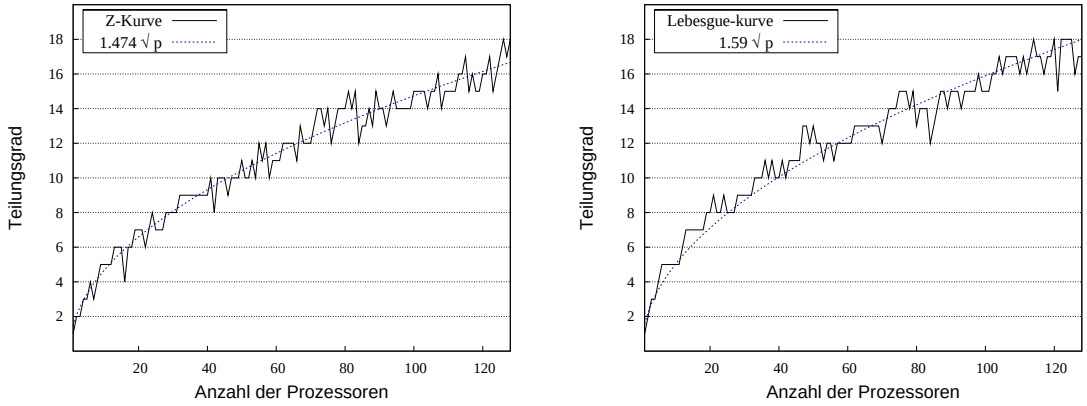


Abbildung 6.3.6: Teilungsgrad für Z- und Lebesgue-Kurve

Leider führt Sequenzpartitionierung mit raumfüllenden Kurven nicht gleichzeitig auch zu einer Reduktion des Durchmessers $\operatorname{diam}(T)$. Der Grund hierfür liegt darin, dass stets ganze Blockcluster bzw. Matrizen auf die einzelnen Prozessoren verteilt werden. Somit ist $\operatorname{diam}(T)$ durch die Dimension des größten zulässigen Blockes beschränkt, wobei bei den $\mathcal{H}$-Matrizen üblicherweise Blockgrößen der Ordnung n angestrebt werden. Diese Eigenschaft ist prinzipieller Natur und liegt somit bei allen Algorithmen zur Lastbalancierungen vor, sofern die Matrizen selbst nicht geteilt werden.

Der durch $\operatorname{diam}(T)$ induzierte Kommunikationsaufwand von $\max\{n, m\}$ dominiert folglich den entsprechenden Term in (6.3.7). Werden desweiteren die experimentell ermittelten Werte

für d_{sh} verwendet, so lässt sich die Komplexität der parallelen Matrix-Vektor-Multiplikation wie folgt angeben:

Bemerkung 6.3.6 *Unter der Annahme, dass $d_{\mathrm{sh}} \sim \sqrt{p}$, besitzt Algorithmus 6.3.2 für $\mathcal{H}$-Matrizen $A \in \mathcal{H}(T)$ eine Komplexität von*

$$\mathcal{W}_{\mathrm{MV}}(A,p) = \mathcal{O}\left(\frac{\mathcal{W}_{\mathrm{MV}}(A)}{p} + \frac{n}{\sqrt{p}}\right) + g \cdot \mathcal{O}\left(\max\{n,m\}\right) + l \cdot 3. \tag{6.3.8}$$

6.3.1.3 Lastbalancierung mit Kommunikation

In dem Beispiel in Abbildung 6.3.7 wurde eine Verteilung der Matrizen für die Matrix-Vektor-Multiplikation mittels Sequenzpartitionierung und Hilbert-Kurven berechnet. Dabei ergibt sich durch die stark variierenden Kosten von Matrizen nahe der Diagonalen und von Blöcken entfernt davon ein Zeilendurchmesser, welcher im wesentlichen n gleicht. Analog lassen sich Beispiele konstruieren, in denen der Spaltendurchmesser m entspricht. Auch wenn, wie im letzten Abschnitt beschrieben, sich die Ordnung des Verteilungsdurchmessers nicht verringern lässt, so besteht doch die Möglichkeit, den absoluten Wert von $\mathrm{diam}(T)$ zu reduzieren.

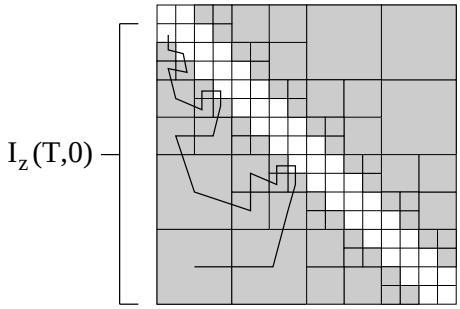


Abbildung 6.3.7: Zeilendurchmesser für Sequenzpartitionierung mit Hilbert-Kurven

Im letzten Abschnitt wurden die Kommunikationskosten für das Versenden eines Vektorelementes nicht bei der eigentlichen Lastbalancierung berücksichtigt. Die Annahme war, dass sie im Vergleich zur Multiplikation vernachlässigbar sind. Trifft dies jedoch nicht zu, führt die ungleichmäßige Verteilung wie in Abbildung 6.3.7 zu einem sehr großen Aufwand bei der Kommunikation. Gesucht ist deshalb ein Algorithmus, welcher die beiden Aspekte gleichzeitig berücksichtigt.

Eine Möglichkeit hierzu besteht in einem modifizierten Sequenzpartitionierungsalgorithmus. Die Grundlage bildet dabei das Bisektionsverfahren 4.1.7 aus Abschnitt 4.1.3, da der Lösungsalgorithmus 4.1.6 für das Sequenzpartitionierungsproblem nicht auf die hier vorgestellte Art verändert werden kann.

Während des Ablaufs des Bisektionsalgorithmus 4.1.7 werden Blockcluster entsprechend der Reihenfolge, wie sie die verwendete raumfüllende Kurve definiert, zu $\mathcal{L}(T,q)$ hinzugefügt.

Eine neue Sequenz wird begonnen, sobald $\sum_{b\in\mathcal{L}(T,q)} c(b)$ die Kosten der maximalen Teilsequenz der Partition übersteigen. Die Modifizierung besteht nun darin, in diesen Vergleich auch den Aufwand für die Kommunikation mit einzubeziehen.

Sei im folgenden $c_{\text{com}} > 0$ der Aufwand für das Versenden und Empfangen eines Vektorelements. Die Kommunikationskosten eines einzelnen Prozessors q sind somit durch

$$c_{\text{com}} \cdot (\text{diam}_{\text{z}}(T,q) + \text{diam}_{\text{s}}(T,q))$$

bestimmt. Die Gesamtkosten der Matrix-Vektor-Multiplikation, die Prozessor q verursacht, lauten entsprechend

$$c_{\text{com}} \cdot (\text{diam}_{\text{z}}(T,q) + \text{diam}_{\text{s}}(T,q)) + \sum_{b\in\mathcal{L}(T,q)} c(b).$$

Dieser Aufwand definiert damit auch das Abbruchkriterium innerhalb des Bisektionsverfahrens. Die vollständige Partitionierung ist im folgenden Algorithmus dargestellt.

```
procedure partition_com( L(T), c_max, c_com )
    c := 0; p := 0;
    for all (τ,σ) ∈ L(T) do
        c' := c_MV,k(τ,σ);
        if c + c' + c_com · (diam_z(T,p) + diam_s(T,p)) > c_max then
            p := p + 1; c := c';
        else
            c := c + c';
        L(T,p) := L(T,p) ∪ {(τ,σ)};
    endfor;
    return p;
end;
```

Algorithmus 6.3.3: Sequenzpartitionierung mit Kommunikation

Notwendig für die Bisektion ist weiterhin die Angabe des Suchintervalls. Nach unten ist dieses Intervall nur durch 0 beschränkt. In diesem Fall verrichtet ein Prozessor keinerlei Arbeit. Wird dagegen die gesamte Matrix-Vektor-Multiplikation auf nur einem einzigen Prozessor ausgeführt, ergeben sich die nachfolgend beschriebenen minimalen und maximalen Kosten. Letztere setzen sich hierbei aus der Multiplikation und dem Versenden des Ergebnisses an alle anderen Prozessoren zusammen.

$$c_{\min} = 0 \quad \text{und} \quad c_{\max} = \sum_{b\in\mathcal{L}(T)} c(b) + c_{\text{com}} \cdot \max\{n,m\}\,. \tag{6.3.9}$$

Der vollständige Algorithmus für die Sequenzpartitionierung mit Kommunikation lautet damit:

```
procedure greedy_seq_part_com( L(T), p, c_com )
    Setze c_min und c_max entsprechend zu (6.3.9);
    while  c_min ≠ c_max  do
        c_avg = 1/2 (c_min + c_max);
        p' = partition_com( L(T), c_avg, c_com );
        if  p' < p  then c_max = c_avg;
        else             c_min = c_avg;
        if  p' = p and |1 − c/C| < ε then return
    endwhile;
end;
```

Algorithmus 6.3.4: Bisektion mit Kommunikation

Algorithmus 6.3.4 kann ebenfalls innerhalb der hierarchischen Verteilungsalgorithmen aus Abschnitt 4.2 verwendet werden.

Der Aufwand für das ursprüngliche Bisektionsverfahren ist bestimmt durch den Unterschied der Kosten für das maximale Element der Sequenz und die Gesamtkosten (siehe (4.1.13)). In der modifizierten Variante gehen dagegen allein die maximalen Kosten $c_{\max}$ ein. Da $c_{\max}$ dem Aufwand der Matrix-Vektor-Multiplikation plus Kommunikation entspricht, ergibt sich das folgende Resultat.

Lemma 6.3.7 *Sei $T(I \times J)$ ein Blockclusterbaum und für alle $v \in V(T) \setminus \mathcal{L}(T)$ gelte: $d(v) \geq 2$. Dann besitzt Algorithmus 6.3.4 eine Komplexität von*

$$\mathcal{O}\left(c_{\mathrm{sp}} \max\{n, m\} \log\left(c_{\mathrm{sp}}(T) p(T) \max\{n, m\}\right)\right).$$

Beweis: Nach Bemerkung (2.1.1) gilt für die Kardinalität der Knotenmenge in T: $|V(T)| = \mathcal{O}\left(c_{\mathrm{sp}} \max\{n, m\}\right)$. Die Länge des Suchintervalls wird durch die maximalen Kosten beschränkt, welche nach Lemma 3.1.2 durch $\mathcal{O}\left(c_{\mathrm{sp}}(T) p(T) \max\{n, m\}\right)$ gegeben sind. Damit folgt die Behauptung. □

6.3.1.4 Matrix-Vektor-Multiplikation mittels Threadpool

Unter Verwendung eines Threadpools (siehe Abschnitt 5.2.1.1) vereinfacht sich Algorithmus 6.3.2, da die Kommunikation auf einem Rechner mit gemeinsamem Speicher entfällt. Hierdurch können die ersten beiden BSP-Schritte zusammengefasst werden. Lediglich die Summation der lokalen Teilergebnisse muss in einem einzelnen Schritt erfolgen, da hierfür zunächst die Berechnung sämtlicher Matrix-Vektor-Produkte notwendig ist.

```
procedure mv_mul_tp( α, A, x, β, y )
    procedure mat_vec ( i, β, y_i, A_i, x )
        y_i := β · y_i; y'_i := α A_i x;
    end;

    procedure sum ( i, y_i )
        Y_i := { y'_j | I_i ∩ I(T, j) ≠ ∅ };
        y_i := y_i + Σ_{y'_j ∈ Y_i} y'_j|_{I_i};
    end;

    for 0 ≤ i < p do tp_run( mat_vec( i, β, y_i, A_i, x_i ) );
    tp_sync_all();
    for 0 ≤ i < p do tp_run( sum( i, y_i ) );
    tp_sync_all();
end;
```

Algorithmus 6.3.5: Parallele Matrix-Vektor-Multiplikation mit Threadpool

Da keine Kommunikation bei der Analyse von Algorithmus 6.3.5 zu beachten ist, lässt sich auch die Abschätzung der Komplexität entsprechend vereinfachen.

Bemerkung 6.3.8 *Die Berechnung des Matrix-Vektor-Produktes* (6.3.1) *für eine $\mathcal{H}$-Matrix $A \in \mathcal{H}(T)$ auf einem Rechner mit gemeinsamem Speicher und p Prozessoren hat einen Aufwand von*

$$\mathcal{W}_{\mathrm{MV}}(A,p) = \mathcal{O}\left(\frac{\mathcal{W}_{\mathrm{MV}}(A)}{p} + \frac{d_{\mathrm{sh}} n}{p}\right). \tag{6.3.10}$$

Mit dem experimentellen Ergebnis $d_{\mathrm{sh}} \sim \sqrt{p}$ ergibt sich damit

$$\mathcal{W}_{\mathrm{MV}}(A,p) = \mathcal{O}\left(\frac{\mathcal{W}_{\mathrm{MV}}(A)}{p} + \frac{n}{\sqrt{p}}\right).$$

Da die Anzahl der Prozessoren auf einem Rechnersystem mit gemeinsamem Speicher begrenzt ist (siehe Abschnitt 5.2.1), dominiert üblicherweise der erste Term in dieser Abschätzung. Aus diesem Grund besteht in der Praxis die Möglichkeit, dass eine andere, bessere Lastverteilung, etwa mittels LPT- oder Multifit-Scheduling zu einer geringeren Laufzeit führt, obwohl der Teilungsgrad von größerer Ordnung ist.

6.3.1.5 Numerische Beispiele

Das Modellproblem für die nachfolgenden numerischen Tests bildet das BEM-Beispiel aus Abschnitt 2.4.1, wobei zunächst in der Zulässigkeitsbedingung (2.4.4) $\eta = 1$ gewählt wird. Da bei der auftretenden Systemmatrix die einzelnen Matrixkoeffizienten typischerweise ungleich 0 sind, entspricht die angenommene Kostenfunktion (6.3.3) den realen Kosten.

In der folgenden Tabelle sind die Laufzeiten und die parallele Effizienz von Algorithmus 6.3.2 mittels der in Abschnitt 6.3.1.2 beschriebenen Lastverteilung, d.h. ohne explizite Berücksichtigung der Kommunikationskosten, angegeben.

100× Matrix-Vektor-Multiplikation, konstanter Rang $k = 10$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	6.7	2.2	74.7	1.5	57.4	1.3	44.4	0.9	44.8
7 920	16.3	5.1	80.2	3.4	59.4	2.4	56.0	2.0	51.1
19 320	48.0	14.1	84.9	8.0	74.8	6.2	64.2	5.1	58.4
43 680	128.0	35.9	89.1	21.6	73.9	16.4	65.0	13.5	59.1
89 400	303.6	86.5	87.7	45.6	83.3	34.5	73.4	28.0	67.9
184 040	–	198.2	–	113.0	*87.7*	81.8	*80.8*	63.0	*78.7*

Deutlich erkennbar ist die große Abhängigkeit der Effizienz von der Zahl der Prozessoren, im wesentlichen hervorgerufen durch den Kommunikationsaufwand. Insbesondere bei kleinen Problemdimensionen ergibt sich hierdurch eine sehr geringe parallele Skalierung. Aufgrund der wachsenden und schließlich dominierenden Kosten der eigentlichen Multiplikation wächst die Effizienz aber mit n, was auf einen skalierbaren Algorithmus hindeutet (siehe Abschnitt 6.1).

Motiviert durch den beobachteten Einfluss der Kommunikation auf die parallele Effizienz stellt sich die Frage nach einer Optimierung der Multiplikation unter einer expliziten Einbeziehung der Kommunikationskosten wie sie mittels Algorithmus 6.3.3 vorgenommen wird.

100× Matrix-Vektor-Mult. ohne/mit Kommunikation, $n = 184\,040$						
	$p = 8$		$p = 12$		$p = 16$	
k	$c_{\text{com}} = 0$	$c_{\text{com}} = 20$	$c_{\text{com}} = 0$	$c_{\text{com}} = 20$	$c_{\text{com}} = 0$	$c_{\text{com}} = 20$
1	46.9	41.6	33.5	31.0	30.5	28.2
3	57.7	54.2	44.8	39.5	36.4	31.1
5	76.6	71.2	57.8	51.6	44.3	39.8
7	90.2	88.5	66.4	62.8	51.8	46.0
10	113.0	111.5	81.8	77.5	63.0	58.6
15	152.9	147.6	109.1	105.0	82.3	77.0

Je nach Rang und Prozessorzahl sind teilweise deutliche Reduktionen der Laufzeit und damit Steigerungen der Effizienz zu verzeichnen. Allerdings lässt sich keine allgemeine Aussage darüber treffen, unter welchen Bedingungen der Effekt am größten ist. Aus theoretischen Überlegungen heraus sollte sich bei einem kleinen Rang die größte Effizienzsteigerung zeigen, da der Anteil der eigentlichen Multiplikation am Gesamtaufwand in diesem Fall am geringsten ist.

In Abbildung 6.3.8 ist die Effizienz bei unterschiedlichem Rang und Zulässigkeitsparameter dargestellt. Um die Skalierung bezüglich einem Prozessor berechnen zu können, wurde die Problemdimension hierbei bei einem vergleichsweise kleinen Wert von $n = 19\,320$ festgesetzt. Aus den Diagrammen ist ersichtlich, dass ein kleiner Rang und ein großes η, und damit ein

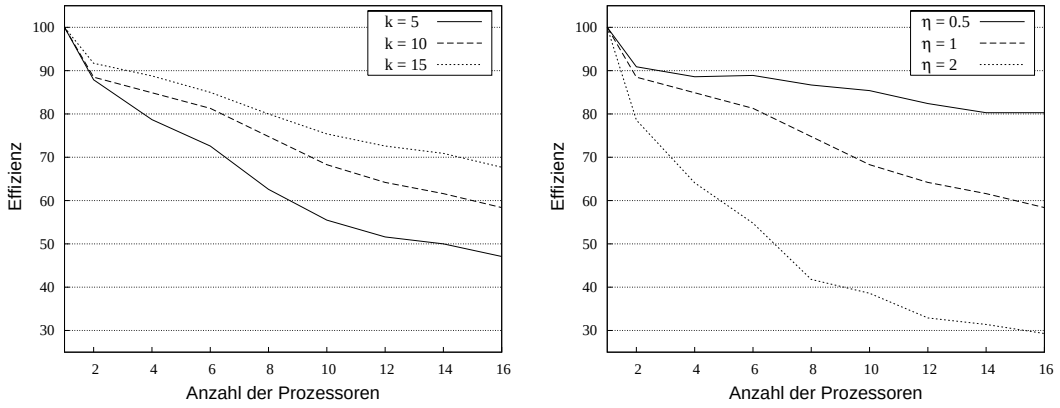


Abbildung 6.3.8: Parallele Effizienz in Abhängigkeit von k (links) bzw. η (rechts)

kleiner Wert für c_{sp}, zu einer relativ geringen parallelen Effizienz führen. Hierbei sei allerdings erwähnt, dass die Laufzeiten in diesen Fällen bei dem gewählten Beispiel sehr kurz sind und somit die Kommunikation dominiert. Ist die $\mathcal{H}$-Matrix dagegen datenstark, d.h. sie besitzt einen großen Rang oder ein großes c_{sp}, so tritt die Kommunikation in den Hintergrund und die Effizienz steigt an.

Die Berechnung des gleichen Problems erfolgt nun auf einem Rechnersystem mit gemeinsamem Speicher. Die Laufzeiten und parallele Effizienz des hierfür genutzten Algorithmus 6.3.5 sind in der folgenden Tabelle aufgeführt.

100× Matrix-Vektor-Mult., konstanter Rang $k = 10$ (PRAM)									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	4.4	1.3	82.4	0.7	75.5	0.5	69.7	0.4	63.2
7 920	10.3	3.0	85.2	1.6	81.2	1.1	79.0	0.9	73.4
19 320	29.4	8.2	89.7	4.4	84.3	3.0	80.9	2.4	76.9
43 680	74.5	20.7	89.8	10.8	85.9	7.5	83.3	5.8	80.0
89 400	168.1	47.3	88.8	24.2	86.7	16.6	84.5	13.1	80.2
184 040	382.6	105.1	91.0	54.6	87.6	37.6	84.7	29.6	80.7

Im Falle einer PRAM ergibt sich ein zu einer BSP-Berechnung ähnliches Bild der Matrix-Vektor-Multiplikation. Die parallele Effizienz steigt mit der Problemdimension an. Allerdings ergeben sich ob der fehlenden Kommunikation bessere Effizienzwerte.

Neben der Multiplikation mit einer $\mathcal{H}$-Matrix bei konstantem Rang besteht auf einer PRAM ebenfalls die Möglichkeit, die $\mathcal{H}$-Matrix mit konstanter Genauigkeit und somit variablem Rang aufzubauen. Die Ergebnisse für diesen Fall sind in der nachfolgenden Tabelle zusammengestellt.

100× Matrix-Vektor-Mult., konstante Genauigkeit $\varepsilon = 10^{-4}$ (PRAM)									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	5.8	1.7	84.1	1.0	76.6	0.7	68.3	0.6	65.0
7 920	13.9	4.0	88.1	2.2	80.9	1.5	75.3	1.2	71.9
19 320	39.2	11.2	87.2	5.9	83.4	4.3	76.4	3.3	74.8
43 680	101.9	29.3	86.9	15.2	83.9	10.8	78.9	8.2	77.7
89 400	231.9	65.6	88.3	34.4	84.1	23.6	81.9	18.7	77.7
184 040	534.4	147.1	90.8	76.6	87.2	53.2	83.7	41.7	80.0

Die parallele Effizienz entspricht dabei im wesentlichen den Werten, die mit einem konstanten Rang erzielt wurden. Die Abweichungen lassen sich durch den Einfluss des eingesetzten Zuteilungsverfahrens bzw. des Abbruchkriteriums (4.2.2) erklären.

Ebenfalls untersucht wurde der Einfluss des Ranges und der Genauigkeit auf die parallele Effizienz. Die Ergebnisse für eine Problemdimension von $n = 43\,680$ sind in Abbildung 6.3.9 dargestellt.

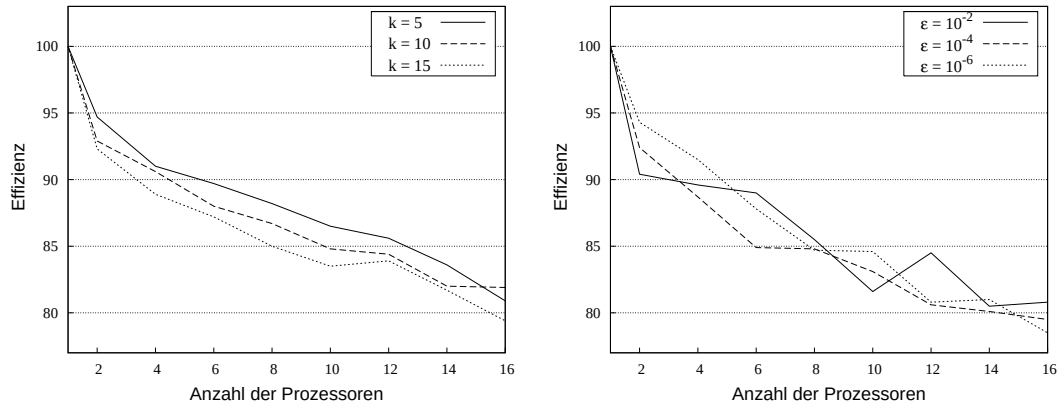


Abbildung 6.3.9: Parallele Effizienz in Abhängigkeit von k (links) bzw. ε (rechts)

Man erkennt deutlich, dass die Effizienz der parallelen Matrix-Vektor-Multiplikation auf einer PRAM praktisch unabhängig von k und ε ist. Bis auf kleine Schwankungen, die durch das Rechnersystem verursacht sind, zeigen alle Kurven das gleiche Verhalten.

6.3.2 Matrix-Vektor-Multiplikation mit Blockaufteilung

Im letzten Abschnitt war die Effizienz des Matrix-Vektor-Multiplikationsalgorithmus beschränkt durch die maximale Größe der Blätter im Blockclusterbaum. Weiterhin skaliert der Algorithmus nur bzgl. $\sqrt{p}$, da der Teilungsgrad $d_{\mathrm{sh}}(T)$ nicht konstant ist.

In diesem Abschnitt wird ein alternativer Algorithmus vorgestellt, der diese Beschränkungen überwindet. Im Gegensatz zur blockorientierten Verteilung wird die Last in diesem Fall bezüglich der Indexmengen bzw. der Clusterbäume balanciert.

Definition 6.3.9 *Sei I eine Indexmenge und $T(I)$ ein Clusterbaum über I und sei $m : \mathcal{L}(T(I)) \to \{0, \ldots, p-1\}$ eine Verteilungsfunktion bezüglich der Blätter in $T(I)$. Dann sei*

$$I(q) = I(m, q) = \bigcup_{\substack{\tau \in \mathcal{L}(T(I)) \\ m(\tau) = q}} \tau$$

die durch $m(q)$ induzierte *Indexmenge.*

Anstelle der Blätter in $T(I)$ lassen sich die Indizes auch direkt verteilen. Man beachte aber, dass für $n_{\min} = 1$ die jeweils induzierten Mengen $I(q)$ identisch sind.

Eine gegebene Verteilung der Indizes lässt sich auch auf die Blätter des Blockclusterbaumes übertragen.

Definition 6.3.10 *Sei $T = T(I \times J)$ ein Blockclusterbaum über $T(I)$ und $T(J)$ und m_I und m_J Verteilungsfunktionen bzgl. $\mathcal{L}(T(I))$ bzw. $\mathcal{L}(T(J))$. Dann seien die Mengen $\mathcal{L}_z(T, q)$ und $\mathcal{L}_s(T, q)$ wie folgt definiert:*

$$\begin{aligned} \mathcal{L}_z(T, q) &= \{(\tau, \sigma) \in \mathcal{L}(T) \,|\, \tau \cap I(m_I, q) \neq \emptyset\} \quad \text{und} \\ \mathcal{L}_s(T, q) &= \{(\tau, \sigma) \in \mathcal{L}(T) \,|\, \sigma \cap J(m_J, q) \neq \emptyset\} \end{aligned}$$

Die Indizes z und s stehen hierbei für *Zeilen* bzw. *Spalten* und deuten die zeilen- bzw. spaltenorientierte Verteilung von $\mathcal{L}(T)$ an. Ein Beispiel hierfür ist in Abbildung 6.3.10 zu sehen. Dargestellt sind dort die einzelnen Teilmatrizen der R- bzw. D-Matrizen, die einem Prozessor zugeordnet werden.

6.3.2.1 Einfacher Fall ohne Blockteilung

Zunächst soll eine einfache Version des Algorithmus dazu dienen, die wesentlichen Techniken zu illustrieren. Betrachtet wird hierzu die $\mathcal{H}$-Matrix aus Abbildung 6.3.10. Für die Vektoren x und y findet die bereits beschriebene Aufteilung in Blöcke der Dimension n/p Verwendung. Die gleiche Verteilung wird für die Indizes genutzt, wodurch sich die Indexmengen

$$I(q) = \left\{ \frac{qn}{p}, \ldots, \frac{(q+1)n}{p} - 1 \right\} \quad \text{und} \quad J(q) = \left\{ \frac{qm}{p}, \ldots, \frac{(q+1)m}{p} - 1 \right\}$$

ergeben.

Der Einfachheit halber wird zunächst davon ausgegangen, dass kein Blockcluster auf zwei verschiedenen Prozessoren vorhanden ist und somit $\mathcal{L}_z(T, q) \cap \mathcal{L}_z(T, q') = \emptyset$ für $q \neq q'$ gilt. Analoges sei für $\mathcal{L}_s(T, q)$ vorausgesetzt. In der Praxis ist dies nur bei kleinen Prozessorzahlen erfüllt. Für die Anzahl von Blockclustern in den Mengen $\mathcal{L}_z(T, q)$ und $\mathcal{L}_s(T, q)$ ergibt sich unter dieser Annahme das folgende, optimale Ergebnis.

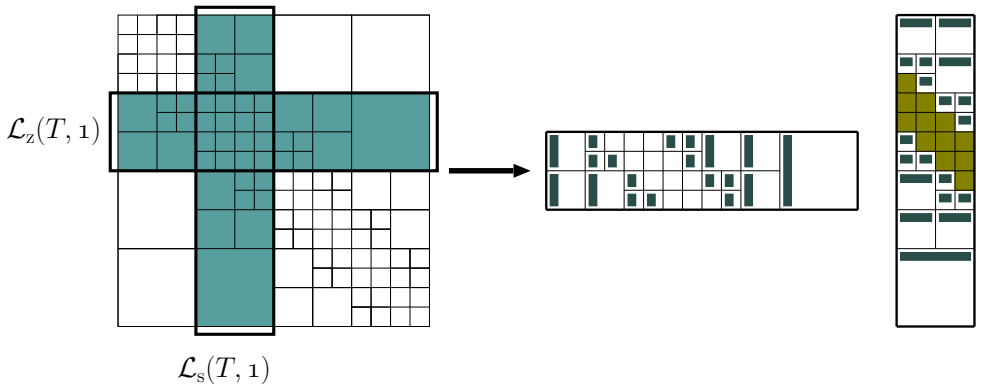


Abbildung 6.3.10: $\mathcal{H}$-Matrix (links) und Datenblöcke auf Prozessor 1 (rechts)

Bemerkung 6.3.11 *Da $|I(q)| = n/p$ und $|J(q)| = m/p$ gilt und somit die Zahl der Knoten in den Clusterbäumen bzgl. $I(q)$ und $J(q)$ durch $\mathcal{O}(n/p)$ bzw. $\mathcal{O}(m/p)$ bestimmt ist, folgt:*

$$|\mathcal{L}_z(T,q)| = \mathcal{O}\left(\frac{c_{\mathrm{sp}} n}{p}\right) \quad \text{und} \quad |\mathcal{L}_s(T,q)| = \mathcal{O}\left(\frac{c_{\mathrm{sp}} m}{p}\right).$$

Wichtigstes Merkmal des folgenden Multiplikationsalgorithmus ist die Aufspaltung der Multiplikation $x = Ry$ mit $\mathsf{R}(k,\tau,\sigma)$-Matrizen $R = A_R B_R^T \in \mathbb{R}^{\tau\times\sigma}$ in die Produkte $y' = B_R^T y$ und $x = A_R y'$. Hierfür wird R nicht vollständig auf einem Prozessor gespeichert, sondern die Matrix A_R wird q mit $(\tau,\sigma) \in \mathcal{L}_s(T,q)$ zugewiesen. Analog wird B_R auf Prozessor q' gespeichert, falls $(\tau,\sigma) \in \mathcal{L}_z(T,q')$. Letzteres gilt auch für vollbesetzte Matrizen $D \in \mathbb{R}^{\tau\times\sigma}$.

Die eigentliche Matrix-Vektor-Multiplikation erfolgt in zwei Schritten. Zunächst werden die Produkte $B_R^T x|_\sigma$ berechnet. Das Ergebnis sind Vektoren y'_q der Dimension k. Außerdem werden alle Produkte mit vollbesetzten Matrixblöcken ermittelt, jeweils mit Ergebnisvektoren der Größe $n_{\min}$.

Anschließend erfolgt der Austausch der Daten. Das Produkt $y' = B_R^T x|_\sigma$ wird an den Prozessor q' mit $\tau \in I_{q'}$ gesendet. Analoges gilt für das Produkt $Dx|_\sigma$. Nach der anschließenden globalen Synchronisation des BSP-Rechners erfolgt die Berechnung der Produkte Ay' mit den zu Blockclustern $(\tau,\sigma) \in \mathcal{L}_z(T,q)$ korrespondierenden $\mathsf{R}(k,\tau,\sigma)$-Matrizen. Die hierbei entstehenden Vektoren können direkt auf den lokalen Vektor y_i addiert werden.

Der vollständige BSP-Algorithmus lautet somit:

```
procedure mv_mul2( A, x, y, q )
    { Schritt 1 }
    y_i := β · y_i;
    for all b = (τ, σ) ∈ L_s(T, q) do
        if M(b) ist R(k, b)-Matrix then
            y(b) = B(M(b))^T x_i|_σ;
        else
            y(b) = M(b) x_i|_σ;
        bsp_send( y(b) zu q mit τ ∈ I_q );
    endfor;
    bsp_sync();
    { Schritt 2 }
    for all b = (τ, σ) ∈ L_z(T, q) do
        if M(b) ist R(k, b)-Matrix then
            y_i|_τ := y_i|_τ + αA(M(b))y(b);
        else
            y_i|_τ := y_i|_τ + αy(b);
    endfor;
    bsp_sync();
end;
```

Algorithmus 6.3.6: Algorithmus zur Matrix-Vektor-Multiplikation

Der große Vorteil der beschriebenen Aufteilung liegt darin, dass bzgl. der Vektoren x und y keine Kommunikation notwendig ist. Im ersten Schritt wird lediglich der lokale Anteil x_i benötigt, während die Ergebnisse des zweiten Schrittes direkt auf y_i addiert werden können. Die eigentliche Kommunikation ist dagegen nur von der Zahl der lokalen Blockcluster und vom Rang bzw. der minimalen Blockgröße $n_{\min}$ abhängig.

Für einfache $\mathcal{H}$-Matrizen, bei denen die Kosten der eigentlichen Matrix-Vektor-Multiplikation für jeden Prozessor gleich sind, ergibt sich folgende Abschätzung der Komplexität von Algorithmus 6.3.6.

Lemma 6.3.12 *Seien $C_s(q)$ und $C_z(q)$ die Kosten pro Prozessor q, die durch die Matrix-Vektor-Multiplikation der zu $\mathcal{L}_s(T,q)$ bzw. $\mathcal{L}_z(T,q)$ korrespondierenden Matrizen hervorgerufen werden. Es gelte: $C_s(q) = C_s(q')$ und $C_z(q) = C_z(q')$ für $q \neq q'$. Dann folgt für die Komplexität von Algorithmus 6.3.6:*

$$\begin{aligned} \mathcal{W}_{\mathrm{MV}}(A,p) &= \mathcal{O}\left(\frac{\mathcal{W}_{\mathrm{MV}}(A)}{p}\right) \\ &+ g \cdot \mathcal{O}\left(\frac{c_{\mathrm{sp}} \max\{k, n_{\min}\} \max\{n,m\}}{p}\right) \\ &+ l \cdot 2. \end{aligned} \tag{6.3.11}$$

Beweis: Die Berechnung der Vektoren $y(b)$ für $b \in \mathcal{L}_s(T,q)$ in Schritt 1 entspricht einer Matrix-Vektor-Multiplikation und besitzt den Aufwand (3.1.5). Dies gilt ebenfalls für die Berechnung im zweiten Schritt.

Nach Bemerkung 6.3.11 werden in Schritt 1 $\mathcal{O}\left(c_{\mathrm{sp}} \max\{n,m\}/p\right)$ Vektoren $y(b)$ der Dimension k, falls $b \in \mathcal{L}_{\mathrm{z}}(T)$, bzw. $n_{\min}$, falls $b \in \mathcal{L}_{\mathrm{nz}}(T)$ erzeugt. Damit folgt ein Kommunikationsaufwand von $g \cdot \mathcal{O}\left(\frac{1}{p} c_{\mathrm{sp}} \max\{k, n_{\min}\} \max\{n,m\}\right)$. Analog empfängt jeder Prozessor maximal $\mathcal{O}\left(\frac{1}{p} c_{\mathrm{sp}} \max\{k, n_{\min}\} \max\{n,m\}\right)$ viele Daten.

Die Summation der einzelnen Kosten ergibt schließlich (6.3.11). □

Bei einer gleichmäßigen Verteilung der Multiplikationskosten über die Indexmenge I zeigt Algorithmus 6.3.6 somit eine optimale Komplexität und parallele Effizienz.

6.3.2.2 Der allgemeine Fall

Nachdem im vorigen Abschnitt die grundlegenden Techniken anhand eines Spezialfalles eingeführt wurden, soll nun der allgemeine Fall untersucht werden, bei dem die Kosten der Multiplikation nicht notwendigerweise gleichmäßig über I bzw. J verteilt sind und ein beliebiges p erlaubt ist. Allerdings wird weiterhin $\min\{n,m\} \gg p$ vorausgesetzt.

Betrachten wir zunächst Schritt 1 von Algorithmus 6.3.6, d.h. die Berechnung des Matrix-Vektor-Produktes aller vollbesetzten Matrizen und von $B^T x$ aller R-Matrizen. Für die Aufteilung der Multiplikation wird die Menge $\mathcal{L}(T(J))$ betrachtet, welche für $n_{\min} = 1$ der Indexmenge J entspricht. Sei m_J eine konsistente Verteilungsfunktion über $T(J)$ (siehe Abschnitt 4.2).

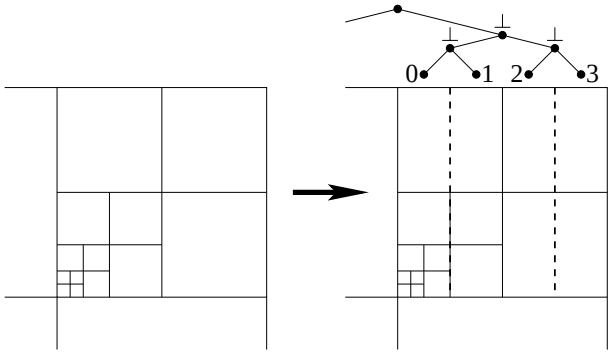


Abbildung 6.3.11: Verfeinerung der Blattmenge bezüglich $T(J)$ und m_J

Die Aufteilung von $T(J)$ durch m_J führt im allgemeinen zu einer Spaltung größerer Blockcluster in T, d.h. ein Blockcluster $(\tau,\sigma) \in \mathcal{L}(T)$ mit $m_J(\sigma) = \perp$ wird mindestens zwei Prozessoren zugewiesen. Dies würde dazu führen, dass die Multiplikation eines solchen Blockes von allen beteiligten Prozessoren durchgeführt wird. Um die damit korrespondierende parallele Ineffizienz zu vermeiden werden die Blätter in T bezüglich $T(J)$ so lange verfeinert, bis der entsprechende Cluster auf genau einen Prozessor verteilt ist. In Abbildung 6.3.11 ist

dies in einem Beispiel dargestellt. Die kleinste so entstehende Menge ist definiert durch

$$\mathcal{L}^s(T) = \left\{ (\tau,\sigma) \;\middle|\; \begin{array}{ll} m_J(\sigma) \neq \perp \quad \wedge & \\ (\exists\, \sigma' \in T(J) \;:\; & (\tau,\sigma') \in \mathcal{L}(T) \;\wedge\; \sigma' \xrightarrow{*} \sigma \;\wedge \\ & \neg\exists\, \sigma'' \in T(J) \;:\; (m_J(\sigma'') \neq \perp \;\wedge\; \sigma'' \rightarrow \sigma)) \end{array} \right\}.$$

Aufgrund der Definition ist die Menge $\mathcal{L}^s(T)$ eindeutig. Die Berechnung von $\mathcal{L}^s(T)$ erfolgt analog zur Konstruktion des Blockclusterbaumes mittels Algorithmus 2.2.1 in Abschnitt 2.2, wobei in diesem Fall nur der Cluster σ verfeinert wird:

```
procedure build_L_s( τ, σ, m_J )
    if (τ,σ) ∈ L(T) ∧ m_J(σ) ≠ ⊥ then return ;
    else
        for all σ' ∈ S(σ) do
            S(τ,σ) = S(τ,σ) ∪ {(τ,σ')};
            build_L_s( τ, σ', m_J );
        endfor;
    endif;
end;
```

Algorithmus 6.3.7: Aufbau von $\mathcal{L}^s(T)$

Man beachte, dass im allgemeinen nur zulässige Blockcluster aufgeteilt werden, da nichtzulässige Blöcke üblicherweise nur auf der untersten Stufe des Blockclusterbaumes auftreten und somit die atomaren Elemente der Clusterbaum- bzw. Indexverteilung darstellen.

Die Matrix-Vektor-Multiplikation im ersten Schritt von Algorithmus 6.3.6 erfolgt bezüglich der Knoten in $\mathcal{L}(T)^s$, d.h. für eine $\mathsf{R}(k)$-Matrix $R = AB^T \in \mathbb{R}^{\tau\times\sigma}, (\tau,\sigma) \in \mathcal{L}(T)$, mit $\sigma \cap J(q) \neq \emptyset$, wird das Produkt $(Bx|_\sigma)|_{\sigma\cap J(q)}$ berechnet. Ein Beispiel für die Aufteilung der Daten durch die Verfeinerung der Blattmenge ist in Abbildung 6.3.12 zu sehen. Man beachte, dass sich die Menge der Daten durch die Aufteilung nicht ändert, der Aufwand zur Speicherung der Matrix somit konstant bleibt.

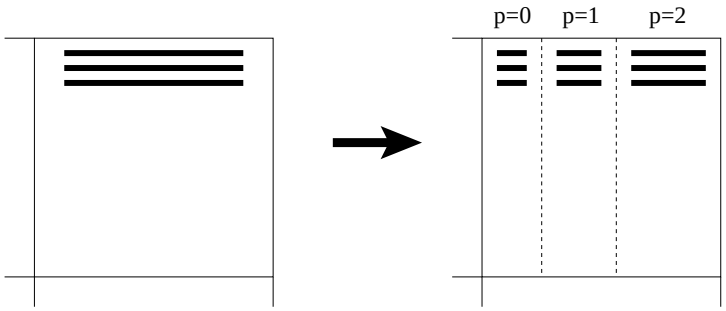


Abbildung 6.3.12: Beispiel für die Aufteilung der Daten von R-Matrizen

Analog zur Aufteilung in Schritt 1 erfolgt die Verteilung der Blöcke im zweiten Schritt von Algorithmus 6.3.6 bezüglich einer konsistenten Verteilungsfunktion m_I über $T(I)$. Die

Menge der verfeinerten Blätter ist hierbei:

$$\mathcal{L}^z(T) = \left\{ (\tau,\sigma) \;\middle|\; \begin{array}{ll} m_I(\tau) \neq \bot \quad \wedge & \\ (\exists\, \tau' \in T(I) : & (\tau',\sigma) \in \mathcal{L}(T) \;\wedge\; \tau' \xrightarrow{*} \tau \;\wedge \\ & \neg \exists\, \tau'' \in T(I) : (m_I(\tau'') \neq \bot \;\wedge\; \tau'' \to \tau) \end{array} \right\}.$$

Der Konstruktion von $\mathcal{L}^z(T)$ und die Matrix-Vektor-Multiplikation in Schritt 2 erfolgen analog zu Algorithmus 6.3.7 bzw. dem ersten Schritt.

Im letzten Abschnitt basierte die Komplexitätsangabe (6.3.11) auf der Kardinalität der lokalen Blattmengen $\mathcal{L}_s(T,q)$ und $\mathcal{L}_z(T,q)$ aus Bemerkung 6.3.11. Dort konnte ausgenutzt werden, dass nur Blockcluster unterhalb einer Größe von $(n/p)^2$ einem Prozessor zugewiesen wurden. Im allgemeinen Fall ist diese Größe dagegen durch $\max_{q=0}^{p-1} \{|I(q)|, |J(q)|\}$ bestimmt. Weiterhin müssen Blockcluster oberhalb dieser Grenze dazugerechnet werden, da auch größere Blöcke durch Aufteilung einen Beitrag zum lokalen Ergebnis leisten können.

Lemma 6.3.13 *Sei $n_{\max} = \max_{q=0}^{p-1} \{|I(q)|, |J(q)|\}$. Dann gilt für die Mächtigkeit der Mengen $\mathcal{L}_z(T,q)$ und $\mathcal{L}_s(T,q)$:*

$$|\mathcal{L}_z(T,q)|, |\mathcal{L}_s(T,q)| = \mathcal{O}\left(c_{\mathrm{sp}} p(T) n_{\max}\right).$$

Beweis: Die Zahl von Knoten mit einer Größe unterhalb von $n_{\max}$ lässt sich nach (2.2.4) mit $\mathcal{O}\left(c_{\mathrm{sp}} n_{\max}\right)$ abschätzen. Für jedes Blatt dieser Menge ist die Zahl der Vorgänger durch $p(T)$ beschränkt. Somit folgt für die Gesamtzahl der Knoten, die größer als $n_{\max}$ sind eine obere Schranke in $\mathcal{O}\left(c_{\mathrm{sp}} p(T) n_{\max}\right)$. Zusammen ergibt sich die Behauptung. □

Für den optimalen Fall $n_{\max} \sim \frac{\max\{n,m\}}{p}$ folgt demnach eine Knotenzahl von

$$|\mathcal{L}_z(T,q)| = \mathcal{O}\left(c_{\mathrm{sp}} p(T) \frac{\max\{n,m\}}{p}\right) \quad \text{und} \quad |\mathcal{L}_s(T,q)| = \mathcal{O}\left(c_{\mathrm{sp}} p(T) \frac{\max\{n,m\}}{p}\right).$$

Zusammenfassend lässt sich nun die Komplexität der Matrix-Vektor-Multiplikation 6.3.6 für den allgemeinen Fall beschreiben, wobei sich ein ähnliches Resultat zu Lemma 6.3.12 ergibt.

Bemerkung 6.3.14 *Nach Lemma 6.3.13 folgt für die Komplexität von Algorithmus 6.3.6:*

$$\begin{aligned} \mathcal{W}_{\mathrm{MV}}(A,p) &= \mathcal{O}\left(\frac{c_{\mathrm{sp}}(T) p(T) \max\{k, n_{\min}\} \max\{n,m\}}{p}\right) \\ &+ g \cdot \mathcal{O}\left(c_{\mathrm{sp}} \max\{k, n_{\min}\} p(T) n_{\max}\right) \\ &+ l \cdot 2. \end{aligned} \tag{6.3.12}$$

Das Ergebnis (6.3.12) ist sehr stark von der Verteilung der Last innerhalb der $\mathcal{H}$-Matrix abhängig, wobei die jeweiligen Extremfälle durch $n_{\max} \sim \max\{n,m\}$ und $n_{\max} \sim \frac{\max\{n,m\}}{p}$ gegeben sind.

6.3.2.3 Lastbalancierung

Bisher offen blieb die Definition der Funktion m, durch die die Verteilung der einzelnen Matrizen der $\mathcal{H}$-Matrix bestimmt ist. Bei allen bisherigen Betrachtungen wurde dabei stets implizit angenommen, dass die Mengen $I(q)$ und $J(q)$ zusammenhängend sind, um die Kompaktheit der daraus resultierenden Blockclustermengen zu gewährleisten und somit die Anzahl der zu teilenden Blockcluster so gering wie möglich zu halten. Mit einem Zuteilungsalgorithmus auf Basis der Sequenzpartitionierung aus Abschnitt 4.1.3 kann diese Eigenschaft erreicht werden.

Notwendig für die Anwendbarkeit der Sequenzpartitionierung ist eine Kostenfunktion für die Blätter des Clusterbaumes. Hierbei wird jeweils eine Funktion für die unterschiedlichen Phasen der Multiplikation angegeben, wobei zunächst der erste Schritt von Algorithmus 6.3.6 diskutiert wird.

Die Grundlage bildet dabei die Kostenfunktion (6.3.3). Eingeschränkt auf die beteiligten Matrixanteile folgt für den Aufwand:

$$c_s((\tau,\sigma),k) = \begin{cases} |\tau| \cdot |\sigma|, & (\tau,\sigma) \in \mathcal{L}_{\mathrm{nz}}(T) \\ k \cdot |\sigma|, & (\tau,\sigma) \in \mathcal{L}_{\mathrm{z}}(T) \end{cases}.$$

Die Summation aller Beiträge der einzelnen Blockcluster pro Knoten des Clusterbaumes ergibt schließlich die eigentliche Kostenfunktion:

$$c_{\mathrm{MV},s}(\sigma) = \sum_{(\tau,\sigma')\in\mathcal{L}(T),\sigma\subseteq\sigma'} c_s((\tau,\sigma'),k)\frac{|\sigma|}{|\sigma'|}.$$

Sei $\tau_0,\tau_1,\ldots,\tau_t$ die Liste der Blätter von $T(I)$ entsprechend der Reihenfolge, wie sie bei einer klassischen Tiefensuche entsteht. Die von Algorithmus 4.1.5 berechnete Partition $R = \{r_0,\ldots,r_p\}$ der Sequenz $c_{\mathrm{MV},s}(\tau_0), c_{\mathrm{MV},s}(\tau_1),\ldots,c_{\mathrm{MV},s}(\tau_t)$ definiert dann die Verteilungsfunktion m, d.h.

$$m(\tau_i) = q \Longleftrightarrow r_q \leq i < r_{q+1}.$$

In Abbildung 6.3.13 ist ein Beispiel für das Resultat dieser Form der Lastbalancierung dargestellt.

Für den zweiten Schritt erhält man analog

$$c_z((\tau,\sigma),k) = \begin{cases} n_{\min}, & (\tau,\sigma) \in \mathcal{L}_{\mathrm{nz}}(T) \\ k \cdot |\tau|, & (\tau,\sigma) \in \mathcal{L}_{\mathrm{z}}(T) \end{cases},$$

wobei in diesem Fall für vollbesetzte Blöcke nur die Kosten zum Aufaddieren des Ergebnisvektors aus dem ersten Schritt entstehen. Die endgültige Kostenfunktion lautet:

$$c_{\mathrm{MV},z}(\tau) = \sum_{(\tau',\sigma)\in\mathcal{L}(T),\tau\subseteq\tau'} c_z((\tau',\sigma),k)\frac{|\tau|}{|\tau'|}.$$

Die Verteilung bzgl. der angeordneten Blätter von $T(I)$ erfolgt ebenfalls analog zum ersten Fall.

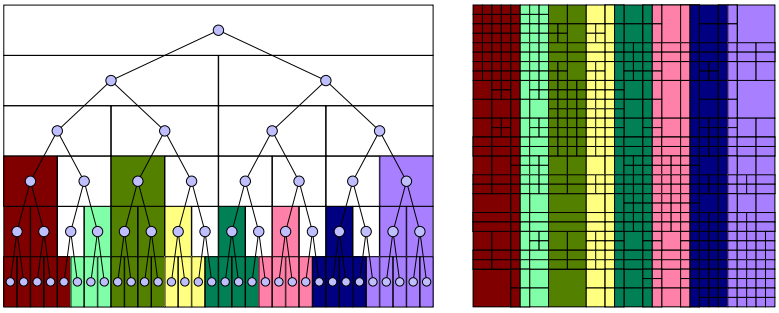


Abbildung 6.3.13: Lastbalancierung im Clusterbaum mittels Sequenzpartitionierung

6.3.2.4 Matrix-Vektor-Multiplikation mittels Threadpool

Auch in diesem Fall lässt sich der Multiplikationsalgorithmus vereinfachen, wenn man die Eigenschaften einer PRAM für die Implementierung zugrundelegt. Zum einen steht der Vektor x allen Prozessoren zur Verfügung, womit die Zerlegung bezüglich $T(J)$ überflüssig ist, und zum anderen ist der Transfer der Produkte $B^T x|_\sigma$ für alle $\mathsf{R}(k)$-Matrizen $AB^T \in \mathbb{R}^{\tau\times\sigma}$ nicht notwendig.

Somit genügt die Verteilung bezüglich $T(I)$ und auch die Verfeinerung von T hinsichtlich der hierdurch definierten Menge $\mathcal{L}^z(T)$. Eine weitere Vereinfachung betrifft Matrizen M, deren korrespondierender Blockcluster (τ,σ) nicht verfeinert wurde, d.h. $(\tau,\sigma) \in \mathcal{L}(T)$. Da keine Kommunikation stattfindet, kann das Produkt $Mx|_\sigma$ direkt ausgeführt werden, unabhängig davon, ob M eine R-Matrix oder eine D-Matrix ist.

Für alle R-Matrizen AB^T über einem Blockcluster $(\tau,\sigma) \in \mathcal{L}^z(T) \setminus \mathcal{L}(T)$ ist dagegen das Produkt wieder getrennt zu behandeln. Allerdings liegt der Grund hierfür nicht in der Kommunikation des Teilergebnisses $B^T x|_\sigma$, sondern in einer optimalen Verteilung der Kosten. Sei $(\tau',\sigma') \in \mathcal{L}(T)$ der kleinste Blockcluster mit $(\tau,\sigma) \in (\tau',\sigma')$ und $M' = M'(\tau',\sigma') = A'B'^T$ der korrespondierende Matrixblock. Da lediglich bezüglich $T(I)$ balanciert wurde und somit nur die Multiplikation von A' verteilt wurde, tritt das Produkt $B^T x|_\sigma$ bei mehr als einem Prozessor auf (siehe Abbildung 6.3.14).

Um hierbei eine Verteilung der Kosten zu gewährleisten, muss die Menge aller entsprechenden Produkte $B^T x$ ebenfalls verteilt werden. Da aber in diesem Fall keine zusätzlichen Eigenschaften gefordert werden, etwa Lokalität der Blockcluster, genügt ein LPT- oder Multifit-Scheduling (siehe Abschnitt 4.1.1 bzw. 4.1.2).

Der Grund, warum nicht alle R-Matrizen auf diese Art und Weise behandelt werden, liegt darin, dass üblicherweise nur wenige Matrizen von einer Verfeinerung betroffen sind und sich in der Praxis Vorteile bei der Laufzeit, insbesondere den Verwaltungsaufwand betreffend gezeigt haben. Unabhängig davon, ob die partielle Multiplikation aller oder nur der jeweils verfeinerten R-Matrizen verteilt werden, ergibt sich die gleiche Komplexität. Diese ist nach

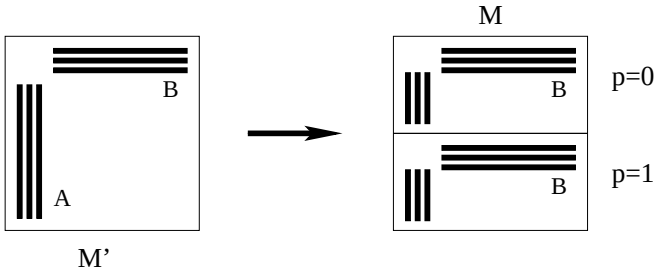


Abbildung 6.3.14: Verteilung der Matrizen A und B einer R-Matrix

(6.3.12) durch

$$\mathcal{W}_{\mathrm{MV}}(A,p) = \mathcal{O}\left(\frac{c_{\mathrm{sp}}(T)p(T)\max\{k,n_{\min}\}\max\{n,m\}}{p}\right)$$

definiert ist und somit optimal bezüglich der Anzahl der Prozessoren.

Für den Algorithmus sei $\mathcal{P}_B = \{M(b) \,|\, b \in \mathcal{L}^z(T) \setminus \mathcal{L}(T) \ \wedge \ M(b) \text{ ist } \mathsf{R}-\text{Matrix}\}$ die Menge aller R-Matrizen über verfeinerten Blockclustern und $\mathcal{P}_{B,q} \subseteq \mathcal{P}_B$ die Menge der Matrizen, die Prozessor q zugeteilt wurde. Dann lässt sich die parallele Matrix-Vektor-Multiplikation in zwei Schritten durchführen:

```
procedure mv_mul ( i, α, A, x, β, y )
    procedure step_1 ( i, β, y, A )
        y_i = β · y_i;
        for all M = AB^T ∈ P_{B,q} do berechne B^T x|_{σ(M)};
    end;

    procedure step_2 ( i, α, A, x, y )
        y_i = y_i + α A x_i;
    end;

    for 0 ≤ i < p do tp_run( step_1( i, β, y, A ) );
    tp_sync_all();
    for 0 ≤ i < p do tp_run( step_2( i, α, A, x, y ) );
    tp_sync_all();
end;
```

Algorithmus 6.3.8: Parallele Matrix-Vektor-Multiplikation mittels Threadpool

Bei der Multiplikation in Schritt 2 von Algorithmus 6.3.8 werden die in Schritt 1 berechneten Teilergebnisse genutzt.

6.3.2.5 Numerische Beispiele

Auch für den Test des Multiplikationsalgorithmus in diesem Abschnitt dient das BEM-Beispiel aus Abschnitt 2.4.1. Zunächst soll hierbei die BSP-Variante der Matrix-Vektor-

Multiplikation untersucht werden. Die folgende Tabelle gibt die Ergebnisse für Berechnungen mit einem konstanten Rang von $k = 10$ und $\eta = 1$ in (2.4.4) wieder.

100× Matrix-Vektor-Mult., konstanter Rang $k = 10$ (BSP)									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	6.7	2.9	58.4	2.1	39.1	2.3	24.6	2.1	20.0
7 920	16.3	6.0	67.5	5.1	40.2	4.2	32.3	3.4	30.0
19 320	48.0	16.2	74.0	11.8	50.9	9.7	41.1	9.2	32.7
43 680	128.0	41.3	77.4	26.6	60.1	20.4	52.2	22.3	35.9
89 400	303.6	89.8	84.5	57.4	66.1	46.8	54.1	45.7	41.5

Im Vergleich zur parallelen Effizienz von Algorithmus 6.3.2 sind deutlich geringere Werte zu beobachten. Der Grund hierfür liegt in dem wesentlich höheren Verwaltungsaufwand, der bei der Matrix-Vektor-Multiplikation 6.3.6 notwendig ist. Verdeutlicht man zusätzlich die geringe Ausführungsdauer einer einzelnen Multiplikation, z.B. von weniger als 0.5 Sekunden bei $n = 89\,400$ und $p = 16$, so wird der große Einfluss des hierfür benötigten Aufwandes deutlich.

Auf der anderen Seite zeigen die Werte für die Effizienz eine einheitliche Tendenz. So ist mit steigender Problemgröße eine Zunahme der parallelen Skalierung zu erwarten. Allerdings konnte dies auf dem zur Verfügung stehenden Rechnersystem nicht überprüft werden.

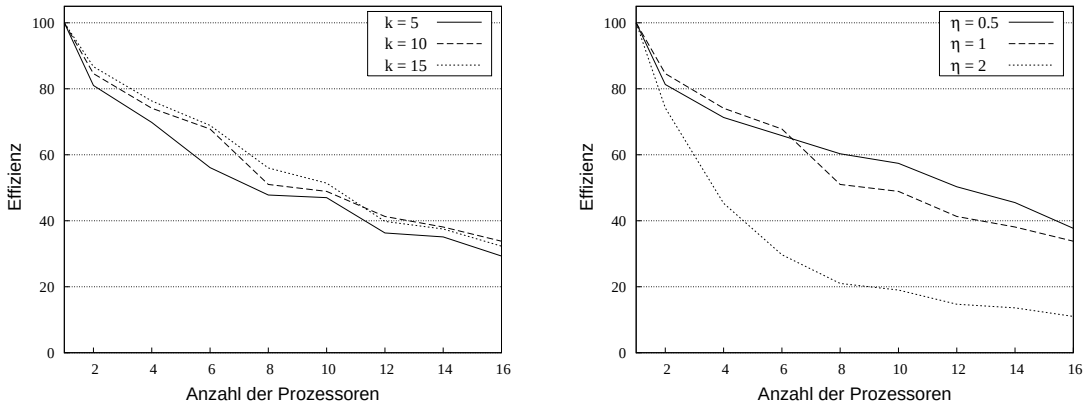


Abbildung 6.3.15: Abhängigkeit der parallelen Effizienz von k (links) und η (rechts)

Abbildung 6.3.15 zeigt einen Vergleich der parallelen Effizienz bei unterschiedlichem Rang und Zulässigkeitsbedingung respektive Schwachbesetztheit der $\mathcal{H}$-Matrix für eine Problemdimension von $n = 19\,320$. Während hierbei die Abhängigkeit der Effizienz von k nur gering ist, wirkt sich ein kleiner Wert von c_{sp}, hervorgerufen durch ein großes η, stark auf die parallele Skalierung aus. In diesem Fall scheint allerdings der Gesamtaufwand weitestgehend

durch die Kommunikationskosten, insbesondere die Zeit für die globale Synchronisation des BSP-Rechners, dominiert zu sein, da die Dauer einer einzelnen Multiplikation weniger als 0.1 Sekunden beträgt. Für größere Problemdimensionen ist hierbei ein besseres, paralleles Verhalten zu erwarten.

Insgesamt lässt sich feststellen, dass die parallele Effizienz von Algorithmus 6.3.6 unbefriedigend ist und hinter den Erwartungen, die durch (6.3.12) hervorgerufen wurden, zurückbleibt. Insbesondere der hohe Verwaltungsaufwand verhindert ein effizientes Verfahren. Der Anstieg der parallelen Effizienz bei wachsender Problemdimension demonstriert allerdings, dass die Multiplikation insgesamt skalierbar ist. Entsprechend ist für ein hinreichend großes n mit ähnlichen Effizienzwerten zu rechnen, wie im Falle von Algorithmus 6.3.2.

Das gleiche Beispiel dient auch der Untersuchung von Algorithmus 6.3.8 auf einem System mit gemeinsamem Speicher. Auch hier erfolgen die Berechnungen zunächst mit einem festen Rang und $\eta = 1$. Die Laufzeiten und jeweilige parallele Effizienz sind in der nachfolgenden Tabelle aufgeführt.

100× Matrix-Vektor-Mult., konstanter Rang $k = 10$ (PRAM)									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	4.5	1.2	87.7	0.7	81.2	0.5	72.5	0.4	64.7
7 920	10.3	2.9	89.7	1.5	87.8	1.1	78.3	0.9	73.4
19 320	29.4	7.9	93.0	4.2	87.9	2.9	83.9	2.3	79.9
43 680	74.4	20.5	90.9	10.6	88.2	7.4	84.1	5.7	82.1
89 400	168.1	45.3	92.7	23.4	89.7	16.2	86.6	12.8	82.3
184 040	382.6	100.6	95.1	53.3	89.8	35.9	88.9	27.9	85.8

Die in diesem Fall gemessenen Werte zeigen eine wesentlich höhere Effizienz als im ersten Rechenbeispiel. Offensichtlich konnten die Vereinfachungen, die aufgrund des geänderten Rechnermodells möglich waren, den Verwaltungsaufwand reduzieren. Die resultierende Effizienz ist von ähnlicher Ordnung wie bei Algorithmus 6.3.5. Teilweise zeigen sich sogar bessere Werte, wie die folgende Tabelle verdeutlicht. In ihr sind die Laufzeiten beider Verfahren für $n = 184\,040$ gegenübergestellt.

	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
	t [s]	E	t [s]	E	t [s]	E	t [s]	E
Algorithmus 6.3.5	105.1	91.0	54.6	87.6	37.6	84.7	29.6	80.7
Algorithmus 6.3.8	100.6	95.1	53.3	89.8	35.9	88.9	27.9	85.8

Auch in diesem Fall ermöglicht der gemeinsame Speicher Berechnungen mit einer konstanten Genauigkeit. Die hierbei gemessenen Werte enthält die folgende Tabelle.

100× Matrix-Vektor-Mult., konstante Genauigkeit $\varepsilon = 10^{-4}$ (PRAM)									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	10.4	2.9	89.4	1.8	72.0	1.3	65.5	1.0	62.9
7 920	27.0	7.7	88.1	4.3	79.4	2.8	79.2	2.4	69.2
19 320	79.8	22.2	90.0	12.0	83.1	8.2	80.7	6.7	72.5
43 680	219.9	58.6	93.9	21.0	88.7	23.0	79.5	17.0	81.0
89 400	539.7	143.5	94.0	73.9	91.3	51.4	87.6	41.5	81.2
184 040	1337.7	352.5	94.9	183.3	91.2	122.1	91.3	95.8	87.3

Das sich hierbei abzeichnende Bild entspricht dem Verhalten bei einem konstanten Rang. Insbesondere die Werte der parallelen Effizienz sind von gleicher Größenordnung.

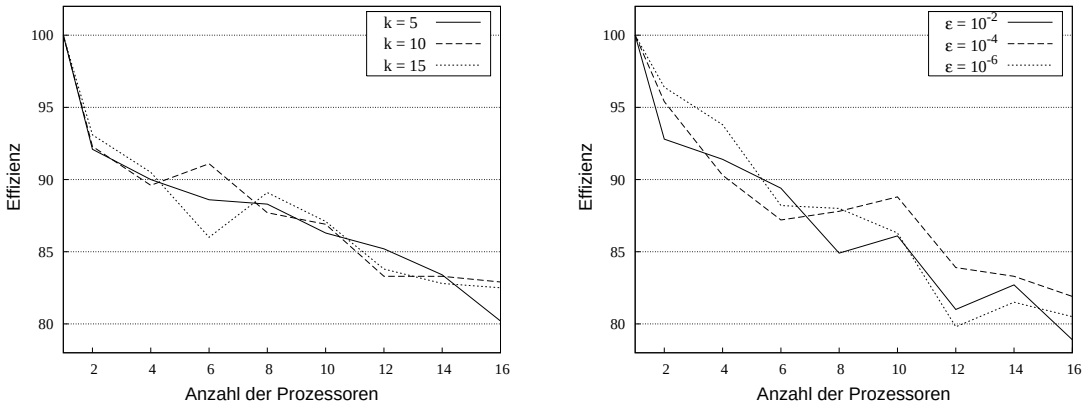


Abbildung 6.3.16: Abhängigkeit der parallelen Effizienz von k (links) und ε (rechts)

Der Einfluss des Ranges und der Genauigkeit auf die Effizienz von Algorithmus 6.3.8 bei einer konstanten Problemgröße von 43 680 ist in Abbildung 6.3.16 dargestellt. Analog zu Verfahren 6.3.5 ergibt sich auch hierbei nur eine geringe Abhängigkeit von beiden Variablen.

6.4 Matrix-Addition

In diesem Abschnitt wird, wie schon bei der sequentiellen Addition in Abschnitt 3.2, die Operation

$$A := \alpha A + \beta B,$$

mit $\mathcal{H}$-Matrizen $A, B \in \mathcal{H}(T, k)$ und $\alpha, \beta \in \mathbb{R}$, betrachtet. Die parallele $\mathcal{H}$-Matrix-Addition basiert hierbei direkt auf dem sequentiellen Algorithmus 3.2.1. Somit gilt es, die Addition der einzelnen Matrixblöcke zu verteilen.

Zunächst wird die Addition auf einem allgemeinen BSP-Rechner diskutiert. Anschließend wird die gleiche Operation auf einer PRAM mittels Threadpool betrachtet.

6.4.1 Ein BSP-Algorithmus zur Matrix-Addition

Für das auf einem BSP-Rechner notwendige Offline-Scheduling wird eine Kostenfunktion und ein darauf aufbauender Verteilungsalgorithmus benötigt. Die Kostenfunktion leitet sich hierbei direkt aus dem Aufwand der Addition pro Matrixblock ab (siehe auch Lemma 3.2.1). Für eine Matrix-Addition ohne Kürzen ergibt sich somit die Funktion:

$$c_{\mathrm{MA}}(\tau,\sigma,k) = \begin{cases} |\tau|\cdot|\sigma|, & (\tau,\sigma)\in\mathcal{L}_{\mathrm{nz}}(T) \\ 2k(|\tau|+|\sigma|), & (\tau,\sigma)\in\mathcal{L}_{\mathrm{z}}(T) \end{cases}.$$

Ist ein entsprechender Kürzungsschritt gewünscht, so sind die Kosten für die Addition zweier $\mathsf{R}(k)$-Matrizen zu modifizieren:

$$c_{\mathrm{MA,tr}}(\tau,\sigma,k) = \begin{cases} |\tau|\cdot|\sigma|, & (\tau,\sigma)\in\mathcal{L}_{\mathrm{nz}}(T) \\ k^2(|\tau|+|\sigma|)+k^3, & (\tau,\sigma)\in\mathcal{L}_{\mathrm{z}}(T) \end{cases}.$$

Die Verteilung der Blockcluster auf die einzelnen Prozessoren geschieht mittels der in Abschnitt 4 beschriebenen Verfahren, etwa LPT- oder Multifit-Scheduling. Eine Verteilungsfunktion m über der Blattmenge von T ist dabei hinreichend.

Während der parallelen Addition berechnet jeder Prozessor für die Menge der lokalen Blockcluster $\mathcal{L}(T,q) = \{v\in\mathcal{L}(T)\,|\,m(v)=q\}$ die Summe der assoziierten Matrizen. Somit wird jeder zusätzliche Aufwand, wie z.B. ein Durchlaufen der Hierarchie, vermieden.

```
procedure par_matrix_add( q, α, A, β, B )
    for all v ∈ L(T,q) do
        A(v) := T_k(αA(v) + βB(v));
    bsp_sync();
end;
```

Algorithmus 6.4.1: Parallele Matrix-Addition

Man beachte weiterhin, dass keinerlei Kommunikation zwischen den einzelnen Prozessoren notwendig ist. Die Komplexität von Algorithmus 6.4.1 ergibt sich somit direkt aus dem Aufwand (3.2.2) der sequentiellen Addition und der Güte der eingesetzten Zuteilung (siehe Abschnitt 4.3).

Bemerkung 6.4.1 *Die parallele Addition zweier $\mathcal{H}$-Matrizen $A,B\in\mathcal{H}(T,k)$ mit Algorithmus 6.4.1 hat eine Komplexität von:*

$$\mathcal{W}_{\mathrm{MA}}(A,p) = \frac{\mathcal{W}_{\mathrm{MA}}(A)}{p} + l. \tag{6.4.1}$$

6.4.2 Numerische Beispiele

Die $\mathcal{H}$-Matrix in diesem Beispiel ist definiert durch das BEM-Problem aus Abschnitt 2.4.1. Auch hierbei ergibt sich das Problem des begrenzten Speichers bei hohen Dimensionen und einer kleinen Prozessoranzahl, weshalb in diesen Fällen die Effizienz bezüglich des kleinsten p angegeben wird, auf welchem die Addition durchgeführt werden konnte. Die angegebenen Laufzeiten entsprechen dabei 10 Matrixadditionen. Die Verteilung der Addition erfolgte mittels stufenweisem LPT-Scheduling (siehe Abschnitt 4.2.3).

Die folgende Tabelle enthält die Laufzeit und parallele Effizienz der Matrix-Addition mit Kürzen für einen konstanten Rang von $k = 10$ und $\eta = 1$ in der Zulässigkeitsbedingung (2.4.4).

10× Matrix-Addition, konstanter Rang $k = 10$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	21.4	5.6	96.4	3.0	89.6	2.1	85.9	1.6	84.2
7 920	52.8	13.5	97.7	7.3	90.6	4.8	91.1	3.8	86.6
19 320	167.5	45.4	92.2	22.0	95.4	16.6	84.3	11.8	88.8
43 680	476.9	122.2	97.6	62.3	95.6	44.7	89.0	34.6	86.1
89 400	–	368.0	–	179.4	*102.6*	116.7	*105.1*	90.7	*101.5*
184 040	–	–	–	447.8	–	311.3	*95.9*	256.8	*87.2*

Die Daten zeigen eine geringfügige Ineffizienz des parallen Verfahrens, welche abhängig von der Anzahl der Prozessoren ist. Hierbei lässt sich allerdings kein eindeutiger Trend ausmachen. Vielmehr kommt es zu leichten Schwankungen der Effizienz. Ein Grund hierfür sind etwa leichte Abweichungen der Kostenfunktion vom realen Aufwand, da nicht alle Aspekte der verwendeten Softwarebibliothek zur QR- und SVD-Zerlegung berücksichtigt werden können (siehe auch Abschnitt 4.3.2). Aufgrund der Kürze der Laufzeiten können somit bereits kleine Abweichungen von den realen Kosten zu entsprechenden Einbußen bei der parallelen Effizienz führen.

Unabhängig von diesen Schwierigkeiten erreicht das Verfahren auch für kleine Problemdimensionen und große Prozessorzahlen eine hinreichende Skalierung.

Den Einfluss des Ranges bzw. der Schwachbesetztheitsstruktur der $\mathcal{H}$-Matrix auf die Effizienz ist in Abbildung 6.4.1 dargestellt. Die hierbei verwendete Problemgröße beträgt $n = 19\,320$.

Man erkennt nur einen geringen Einfluss von k auf das parallele Verhalten der Addition. Dagegen führt eine sehr datenschwache Matrixstruktur zu einer vergleichsweise kleinen Effizienz. Hierbei sei allerdings erwähnt, dass die Laufzeiten in diesem Fall sehr klein sind und somit die erwähnten Abweichungen von den exakten Kosten besonders großen Einfluss haben.

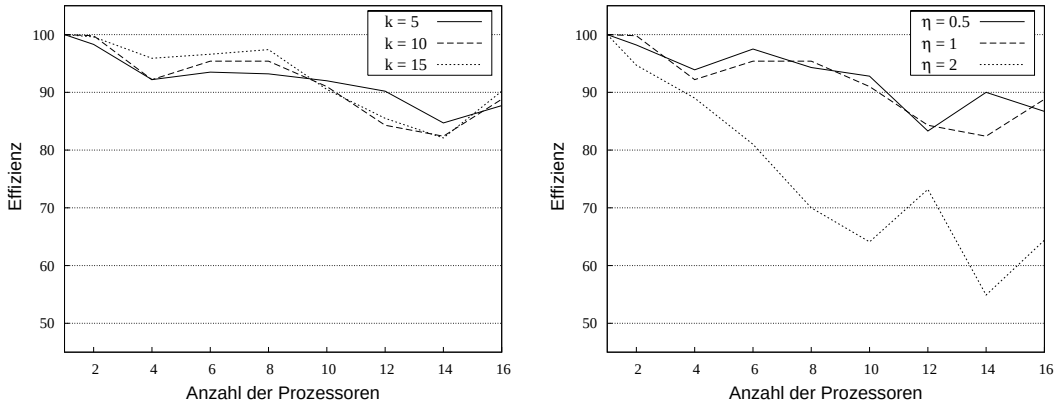


Abbildung 6.4.1: Parallele Effizienz in Abhängigkeit von k (links) und η (rechts)

6.4.3 Matrix-Addition mittels Threadpool

Der sequentielle Algorithmus 3.2.1 lässt sich weiterhin verwenden, um ein Online-Scheduling-Verfahren wie das List-Scheduling einzusetzen. Der parallele Algorithmus addiert hierzu die zu den Blättern von T assoziierten Matrizen in A und B, wobei der erste freie Prozessor die jeweils nächste Addition ausführt. Letztere ist hierbei gegebenenfalls durch einen Kürzungsschritt abzuschließen.

```
procedure add_tp ( α, A, β, B )
    procedure add_mat ( α, A, β, B )
        A(τ,σ) := T_k(αA(τ,σ) + βB(τ,σ));
    end;

    for all v ∈ L(T) do
        tp_run( add_mat( α, A(v), β, B(v) ) );
    tp_sync_all();
end;
```

Algorithmus 6.4.2: Parallele Matrix-Addition mittels Threadpool

Entsprechend der Abschätzung (4.1.2) für das List-Scheduling ist die Komplexität von Algorithmus 6.4.2 identisch zu der von Algorithmus 6.4.1. Dies betrifft allerdings nur die reine Berechnung. Nicht beachtet wird dabei der als konstant anzunehmende Aufwand für das Durchlaufen der Blattmenge, die z.B. in Form einer verketteten Liste vorliegen kann. Da dies sequentiell erfolgt, muss die Aufwandsabschätzung entsprechend angepasst werden.

Lemma 6.4.2 *Der Aufwand für die Matrix-Addition mittels Algorithmus 6.4.2 lautet:*

$$\mathcal{W}_{\mathrm{MA,tp}}(A,p) = \mathcal{O}\left(|\mathcal{L}(T)|\right) + \frac{\mathcal{W}_{\mathrm{MA,tp}}(A)}{p}. \tag{6.4.2}$$

In der Praxis ist der Verwaltungsaufwand allerdings zu vernachlässigen, womit auch Algorithmus 6.4.2 eine optimale Effizienz zeigt.

Bemerkung 6.4.3 *Verwendet man List-Scheduling bei der Verteilung der Blockcluster in Abschnitt 6.4.1, so lässt sich das Online-Scheduling von Algorithmus 6.4.2 simulieren. Hierbei wird außerdem der Verwaltungsaufwand verteilt, wodurch die Effizienz im allgemeinen besser ist.*

6.4.4 Numerische Beispiele

Wie auch im Falle eines BSP-Rechners dient das BEM-Beispiel aus Abschnitt 2.4.1 für die numerischen Tests der Matrix-Addition. Aufgrund des flexibleren Algorithmus können allerdings auch Additionen mit einer konstanten Genauigkeit durchgeführt werden.

Zunächst aber die Ergebnisse für 10 Matrix-Additionen mit einem festen Rang von $k = 10$ und $\eta = 1$.

10× Matrix-Addition, konstanter Rang $k = 10$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	18.0	4.6	98.2	2.3	97.6	1.6	96.9	1.2	96.3
7 920	40.1	10.2	98.3	5.1	97.9	3.4	97.5	2.6	97.2
19 320	109.1	27.7	98.5	13.9	98.0	9.3	97.8	7.0	97.4
43 680	256.6	64.9	98.8	32.6	98.4	21.8	98.0	16.4	97.7
89 400	584.2	147.9	99.1	74.2	98.7	49.7	98.4	37.3	98.2
184 040	1365.5	345.3	98.9	172.8	98.8	115.6	98.5	87.2	97.9

Man erkennt eine durchgehend hohe parallele Effizienz. Diese ist praktisch unabhängig von der Zahl der Prozessoren und erreicht schon bei kleinen Problemdimensionen einen maximalen Wert. Im Vergleich zum BSP-Algorithmus und dem dort verwendeten Lastbalancierungsverfahren wird der Vorteil des Online-Scheduling deutlich, bei dem keine Kostenfunktion nötig ist.

Wie erwähnt, erfolgt die Matrix-Addition im gewählten Beispiel auch mit variablem Rang, d.h. konstanter Genauigkeit. In der nachfolgenden Tabelle sind die Zeiten für $\varepsilon = 10^{-4}$ aufgeführt.

10× Matrix-Addition, konstante Genauigkeit $\varepsilon = 10^{-4}$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	21.2	5.4	97.4	2.7	98.3	1.8	97.4	1.4	97.2
7 920	46.5	12.0	96.8	5.9	98.4	4.0	97.6	3.0	96.9
19 320	132.9	33.3	99.7	16.7	99.5	11.2	98.5	8.5	97.4
43 680	355.6	89.5	99.3	44.9	99.1	30.3	97.8	23.1	96.0
89 400	876.4	216.5	101.2	108.4	101.1	73.3	99.7	56.0	97.9
184 040	2411.1	589.1	102.3	295.6	102.0	207.6	96.8	171.9	87.6

Es zeigt sich ein zum Fall eines konstanten Ranges analoges Bild. Auch mit einer konstanten Genauigkeit wird bereits bei kleinen Problemdimensionen eine sehr hohe Effizienz erzielt. Einzig bei der größten Problemdimension ergibt sich bei 16 Prozessoren ein reproduzierbarer, unerwartet geringer Wert. Eine mögliche Erklärung hierfür könnte im Aufbau des Rechners liegen (siehe Abschnitt 5.2.1.2), da bei Experimenten auf anderen Systemen dieses Verhalten nicht auftritt.

Abschließend wird der Einfluss des Ranges und der Genauigkeit auf die parallele Effizienz untersucht. Hierzu wird die Addition bei einer festen Problemgröße von $n = 43\,680$ mit verschiedenen Werten für k bzw. ε durchgeführt. Die Ergebnisse sind in Abbildung 6.4.2 dargestellt.

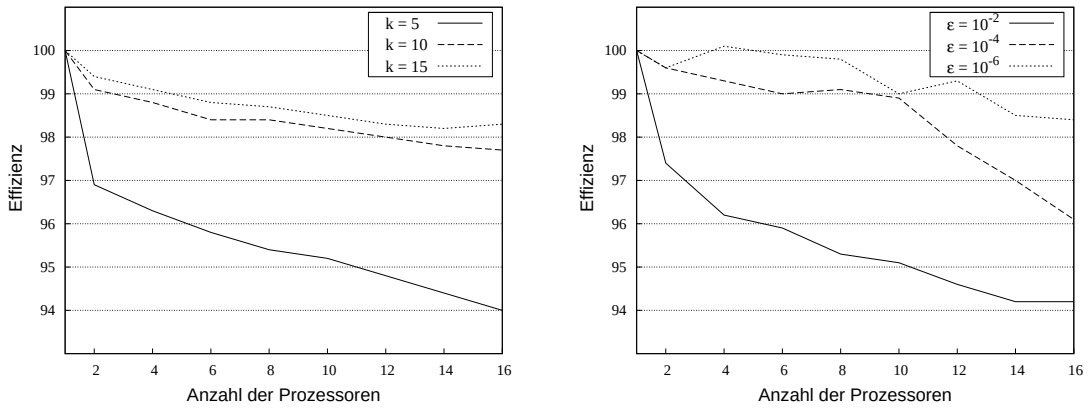


Abbildung 6.4.2: Abhängigkeit der parallelen Effizienz von k (links) und ε (rechts)

Man erkennt eine reduzierte Effizienz bei einem kleinen Rang bzw. geringer Genauigkeit. Allerdings befindet sich diese weiterhin auf einem sehr hohen Niveau. Auch die Abhängigkeit von der Anzahl der Prozessoren ist in diesen Fällen geringfügig stärker ausgeprägt.

6.5 Matrix-Multiplikation

Die Algorithmen zur parallelen Matrix-Multiplikation, wie sie Gegenstand dieses Abschnitts sind, setzen als paralleles Rechnermodell eine PRAM (siehe Abschnitt 5.2.1) voraus. Dabei werden zwei unterschiedliche Ansätze diskutiert. Zunächst erfolgt in Abschnitt 6.5.1 die Konstruktion eines Verfahrens mit Hilfe von Online-Scheduling-Algorithmen. Dabei wird vom Schema des rekursiven Verfahrens, wie es im sequentiellen Fall verwendet wurde, abgewichen. Eine Erweiterung des sequentiellen Multiplikationsalgorithmus mittels Offline-Scheduling-Methoden erfolgt dagegen in Abschnitt 6.5.2.

6.5.1 Matrix-Multiplikation mittels Online-Scheduling

Eine einfache Online-Scheduling-Variante von Algorithmus 3.3.1 besteht darin, jede Multiplikation, die nicht Teil der Rekursion ist, auf einem freien Prozessor auszuführen. Der resultierende parallele Algorithmus, formuliert mit den Funktionen des in Abschnitt 5.2.1.1 beschriebenen Threadpools, lautet damit:

```
procedure mul ( α, A, B, C )
    procedure mul_block( α, A, B, C )
        C := C + αAB;
    end;

    procedure rec_mul ( α, A, B, C )
        if A, B und C sind Blockmatrizen then
            for all τ' ∈ S(τ(C)) do
                for all σ' ∈ S(σ(C)) do
                    for all τ'' ∈ S(τ(B)) do
                        rec_mul( α, A(τ', τ''), B(τ'', σ'), C(τ', σ') );
        else
            tp_run( mul_block( α, A, B, C ) );
        endif;
    end;

    C := βC;
    rec_mul( α, A, B, C );
    tp_sync_all();
end;
```

Algorithmus 6.5.1: Parallele Matrix-Multiplikation mit Online-Scheduling

Bei diesem einfachen Verfahren ist allerdings die Gefahr einer Kollision durch ein gleichzeitiges Schreiben der Daten einer Matrix sehr groß. Als Beispiel sei hierzu die Multiplikation

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} := \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} + \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

betrachtet. Für die Matrix C_{00} ergeben sich die Operationen

$$C_{00} := C_{00} + A_{00}B_{00} \quad \text{und } C_{00} := C_{00} + A_{01}B_{10}.$$

Werden beide Multiplikationen gleichzeitig von unterschiedlichen Prozessoren ausgeführt, ist der schreibende Zugriff auf C_{00} aus Gründen der Datenkonsistenz nur jeweils einem Prozessor zu gestatten. Der entsprechend andere, nicht schreibberechtigte Prozessor wird hierdurch blockiert.

Die Reihenfolge, in der die Prozessoren die kritischen, d.h. Daten modifizierenden Bereiche eines Programms erreichen, ist im allgemeinen nichtdeterministisch. Somit lassen sich auch für die parallele Effizienz keine verlässlichen Vorhersagen treffen. Aus diesem Grund sollte das Blockieren eines Prozessors innerhalb des Verfahren möglichst verhindert werden.

In dem betrachteten Beispiel besteht eine Möglichkeit hierfür darin, alle Operationen, die eine bestimmte Zielmatrix, etwa C_{00}, betreffen, genau einem Prozessor zuzuweisen. Hierdurch arbeiten alle Prozessoren unabhängig, und ein Blockieren ist nicht möglich. Dieses Verfahren soll im folgenden auch auf die gesamte Matrix-Multiplikation einer $\mathcal{H}$-Matrix angewendet werden.

Zur Identifikation aller Produkte, die zu einem Matrixblock C beitragen, kann ein zu Algorithmus 3.3.1 ähnliches Verfahren genutzt werden. Die Grundlage bildet auch hierbei die Rekursion, wie sie bei der Matrix-Multiplikation auftritt. Auch das Stoppkriterium ist identisch, d.h. sobald ein Faktor oder die Zielmatrix mit einem Blatt des Blockclusterbaumes korrespondiert, wird die Rekursion beendet. Die hierbei auftretenden Matrizen A und B werden anschließend der Zielmatrix C als Faktorenpaar zugewiesen:

```
procedure sim ( A, B, C )
    if A, B und C sind Blockmatrizen then
        for all τ' ∈ S(τ(C)) do
            for all σ' ∈ S(σ(C)) do
                for all τ'' ∈ S(τ(B)) do
                    sim( A(τ', τ''), B(τ'', σ'), C(τ', σ') );
    else
        P_C := P_C ∪ {(A, B)};
        L_MM := L_MM ∪ {C};
    endif;
end;
```

Algorithmus 6.5.2: Bestimmung der Faktoren pro Matrixblock

Nach Beendigung von Algorithmus 6.5.2 enthält P_C alle Paare von Faktoren, die während der Matrix-Multiplikation einen Beitrag zu C liefern. In der Menge $\mathcal{L}_{\mathrm{MM}}$ befinden sich desweiteren alle Matrizen, die Ziel einer Multiplikation sind. Dabei ist zu beachten, dass sich $\mathcal{L}_{\mathrm{MM}}$ im allgemeinen von $\mathcal{L}(T)$ unterscheidet, da auch Matrizen enthalten sind, die mit keinem Blatt in T korrespondieren. Der Grund hierfür liegt im bereits erwähnten Abbruchkriterium der Rekursion in Algorithmus 6.5.2. Die explizite Speicherung dieser Menge ist von Vorteil für den anschließenden Multiplikationsvorgang.

Bemerkung 6.5.1 *Die Bestimmung der Faktorenpaare für eine Matrix-Multiplikation $C = AB$ mit $A, B, C \in \mathcal{H}(T)$ durch Algorithmus 6.5.2 hat einen Aufwand von:*

$$\mathcal{W}_{\text{sim,MM}}(C) = \mathcal{O}\left(c_{\text{sp}}(T)|V(T)|\right). \tag{6.5.1}$$

Die eigentliche Multiplikation der Matrizen erfolgt durch ein Online-Scheduling auf $\mathcal{L}_{\text{MM}}$. Jede Zielmatrix C wird dabei einem freien Prozessor zugeordnet, wobei jeweils die in der Menge P_C enthaltenen Produkte ausgeführt werden.

```
procedure mul_tp( α, β, C, L_MM )
    procedure mul_block( α, C )
        for all (A, B) ∈ P_C do
            C := C + αAB;
    end;

    C := βC;
    for all C' ∈ L_MM do
        tp_run( mul_block( α, C' ) );
    tp_sync_all();
end;
```

Algorithmus 6.5.3: Parallele Matrix-Multiplikation

Allerdings kann es auch bei diesem Verfahren zu einem gleichzeitigen Schreibzugriff von verschiedenen Prozessoren auf einen bestimmten Matrixblock kommen. In dem Beispiel in Abbildung 6.5.1 ergibt sich unter anderem für den Block C_{00} das Produkt $A_{01}B_{10}$. Teil von C_{00} ist die Matrix $(C_{00})_{01}$, für die die Paare $((A_{00})_{00}, B_{00})_{01})$ und $((A_{00})_{01}, B_{00})_{10})$ während der Simulation ermittelt werden. Bei gleichzeitiger Ausführung der Produkte für C_{00} und $(C_{00})_{01}$ besteht die Gefahr eines konkurrierenden Zugriffs auf die Daten in $(C_{00})_{01}$.

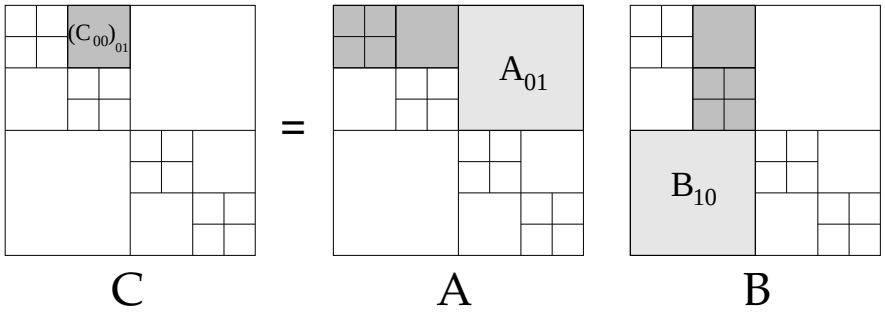


Abbildung 6.5.1: Gleichzeitiges Schreiben verschiedener Prozessoren

Um auch in diesem Fall ein gleichzeitiges Schreiben zu verhindern, könnten alle Teilergebnisse der Multiplikation, die auf einen Matrixblock addiert werden, zunächst in eine Menge eingefügt werden. Der mit der Matrix assoziierte Prozessor wäre dann in der Lage, alle Produkte aus dieser Menge sukzessive auf den Zielblock aufzuaddieren. Allerdings ist dies mit

einem erhöhten Speicherverbrauch des Algorithmus verbunden, da die Zwischenergebnisse möglicherweise nicht sofort bearbeitet werden können.

Dieser gleichzeitige Zugriff trat in praktischen Experimenten allerdings relativ selten auf. Deshalb wurde, wie bereits eingangs diskutiert, der Schutz des kritischen Bereichs durch einen einer Matrix C zugeordneten Mutex m_C (siehe Abschnitt 5.2.1.1) realisiert. In Algorithmus 6.5.3 ist die Funktion `mul_block` hierfür entsprechend anzupassen:

```
procedure mul_block( α, C )
    for all (A, B) ∈ P_C do
        T := αAB;
        lock(m_C);
        C := C + T;
        unlock(m_C);
    endfor;
end;
```

Durch diese Art des Schutzes können alle Teilergebnisse sofort auf die Zielmatrix addiert werden, wodurch kein erhöhter Speicheraufwand entsteht.

Bemerkung 6.5.2 *Die Reihenfolge, in der die Daten eines Matrixblockes geändert werden, ist in diesem Fall nichtdeterministisch. Hierdurch kommt es bei verschiedenen Durchläufen des Algorithmus bzw. bei unterschiedlichen Prozessorzahlen zu geringen Abweichungen des Endergebnisses der Multiplikation im Bereich der Maschinengenauigkeit.*

6.5.1.1 Numerische Beispiele

Die $\mathcal{H}$-Matrix, welche in den folgenden Berechnungen verwendet wird, ist durch das BEM-Beispiel in Abschnitt 2.4.1 definiert. Zunächst erfolgt dabei die Approximation mittels eines konstanten Ranges von $k = 10$. Bei der Matrix-Multiplikation wird dieser Rang nicht verändert, d.h. es werden Kürzungen vorgenommen. Für die Zulässigkeitsbedingung (2.4.4) wurde ein Wert von $\eta = 1$ gewählt. Die folgende Tabelle zeigt die Zeit und die parallele Effizienz von Algorithmus 6.5.3.

Matrix-Multiplikation, konstanter Rang $k = 10$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	59.2	15.2	97.4	7.7	96.4	5.2	95.1	3.9	94.4
7 920	160.5	41.1	97.7	20.7	96.7	14.4	92.8	10.8	93.1
19 320	523.8	133.2	98.3	67.2	97.4	45.7	95.5	35.2	93.1
43 680	1445.8	364.4	99.2	183.5	98.5	125.5	96.0	94.1	96.0
89 400	3704.3	928.5	99.7	466.1	99.3	315.2	97.9	238.0	97.3
184 040	9498.6	2371.6	100.1	1193.0	99.5	804.5	98.4	609.4	97.4

Die Ergebnisse weisen nur eine geringe Abhängigkeit der Effizienz von der Anzahl der Prozessoren auf, wobei diese bei sehr kleinen Problemgrößen am deutlichsten zu sehen ist. Der sequentielle Anteil, der durch die Bestimmung der Faktorpaare für jeden Matrixblock hervorgerufen wird, führt dagegen nur zu einem geringen Verlust an paralleler Effizienz. Als Beispiel sei hierbei eine Ausführungszeit von weniger als 2 Sekunden für Algorithmus 6.5.2 bei $n = 184\,040$ genannt, welche damit lediglich 0.3% der Gesamtzeit der Multiplikation ausmacht. Die verbleibenden Effizienzeinbußen und Superskalierungseffekte lassen sich durch den Einfluss der parallelen Maschine, welche eine PRAM lediglich approximiert (siehe Abschnitt 5.2.1), sowie durch das verwendete Scheduling erklären.

Ungeachtet dieser Bemerkungen erreicht das Verfahren selbst bei einer kleinen Problemgröße eine Effizienz von deutlich über 90% und zeigt damit ein sehr gutes paralleles Verhalten.

In der nächsten Tabelle sind die Resultate der gleichen Berechnungen mit $\mathcal{H}$-Matrizen bei einer konstanten Genauigkeit von $\varepsilon = 10^{-4}$ aufgeführt.

Matrix-Multiplikation, konstante Genauigkeit $\varepsilon = 10^{-4}$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	67.6	17.4	97.1	8.8	95.8	6.0	94.4	4.5	93.8
7 920	181.4	46.8	96.8	23.7	95.9	16.2	93.2	12.5	90.4
19 320	622.3	159.6	97.5	80.5	96.7	54.9	94.4	42.1	92.4
43 680	1861.4	472.0	98.6	237.5	98.0	160.5	96.6	122.7	94.8
89 400	4871.3	1223.8	99.5	614.9	99.0	414.7	97.9	314.9	96.7
184 040	13261.1	3299.5	100.5	1657.2	100.0	1118.1	98.8	851.5	97.3

Die parallele Effizienz im Fall einer konstanten Genauigkeit ist vergleichbar mit den Ergebnissen für einen konstanten Rang. Dies gilt auch für die Superskalierung, welche bei der größten Problemdimension auftritt.

In Abbildung 6.5.2 ist für beide Varianten der Einfluss des Ranges bzw. der Genauigkeit auf die parallele Effizienz dargestellt. Die Problemgröße wurde hierbei konstant bei $n = 43\,680$ belassen. Sowohl mit einem wachsenden Rang als auch bei einer höheren Genauigkeit nimmt die Effizienz zu. Dies lässt sich durch die wachsende Datenmenge in beiden Fällen erklären. Hierdurch steigt z.B. der Anteil der eigentlichen Berechnung im Vergleich zu sequentiellen Programmteilen, etwa dem durch den Threadpool verursachten Aufwand.

6.5.2 Matrix-Multiplikation mittels Offline-Scheduling

Um die bei der Matrix-Multiplikation anfallende Arbeit bereits im voraus zu verteilen, ist zunächst deren genaue Kenntnis notwendig. Leider sind die in Abschnitt 3.3 genannten Abschätzungen für den Aufwand der verschiedenen Teilroutinen für diese Aufgabe nicht geeignet, da hier nur einzelne Multiplikationen betrachtet werden und nicht die Gesamtheit aller Produkte, die zu einer Matrix beitragen. Desweiteren haben sich die Angaben in der

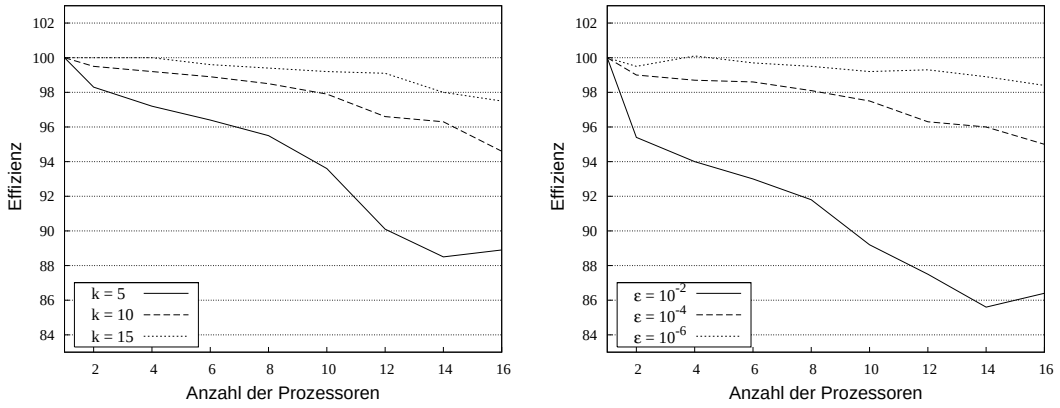


Abbildung 6.5.2: Parallele Effizienz in Abhängigkeit von k (links) bzw. ε (rechts)

Praxis als zu grob erwiesen, da einzelne Optimierungen innerhalb des Algorithmus, welche entscheidenden Einfluss auf die auftretenden Konstanten haben, nicht berücksichtigt werden.

Um eine möglichst genaue Angabe der jeweils auftretenden Arbeit zu erhalten, wurde deshalb ein ähnlicher Zugang gewählt, wie er schon beim Online-Scheduling zum Einsatz kam: durch Simulation der Multiplikation. Im Gegensatz zu Algorithmus 6.5.2 genügt es hierbei allerdings nicht, die Rekursion bei Antreffen eines Blattes zu beenden, sondern die Teilroutinen bis zu den atomaren Operationen weiterzuverfolgen. Der jeweils ermittelte Aufwand wird in diesem Fall der entsprechenden Zielmatrix zugewiesen, wobei die Summe aller auf diese Art und Weise bestimmten Kosten die Funktion $c_{\mathrm{MM}} : \mathcal{L}(T) \to \mathbb{R}_{\geq 0}$ definieren. Man beachte, dass zur Simulation der Matrix-Multiplikation und somit auch zur Berechnung der Kostenfunktion c_{MM} lediglich die Blockclusterbäume und die Kenntnis des auftretenden Ranges benötigt werden, das Vorhandensein der $\mathcal{H}$-Matrizen ist dagegen nicht notwendig.

Leider ist diese Form der Simulation vergleichsweise teuer, insbesondere, da praktisch die gesamte Multiplikation verfolgt wird. Der somit resultierende Aufwand ist vergleichbar mit (3.3.2), wobei die Terme für das Kürzen entfallen.

Bemerkung 6.5.3 *Die Simulation der gesamten Matrix-Multiplikation hat einen Aufwand von*

$$\mathcal{W}_{\mathrm{sim,MM}}(C) = \mathcal{O}\left(c_{\mathrm{sp}}(T)^2 p(T)^2 \max\{|I|, |J|\}\right). \tag{6.5.2}$$

Die Lastbalancierung kann anschließend durch einen der in Abschnitt 4.2 beschriebenen Algorithmen erfolgen, wobei eine zulässige Verteilungsfunktion vorausgesetzt wird. Letztere sei mit m_{MM} bezeichnet.

Der eigentliche Algorithmus für die Matrix-Multiplikation entspricht im wesentlichen dem sequentiellen Verfahren 3.3.1. Einzig eine Abfrage, ob die jeweils betrachtete Matrix dem lokalen Prozessor zugeordnet wurde, ist in den Algorithmus einzufügen.

Ein Schutz vor einem gleichzeitigen, schreibenden Zugriff wie in Algorithmus 6.5.3 ist in diesem Fall nicht notwendig, da eine zulässige Verteilungsfunktion berechnet wurde. Somit sind alle Blätter einer Blockmatrix dem gleichen Prozessor zugeordnet. Allerdings kann der Fall eintreten, dass eine Blockmatrix, die Ziel einer Multiplikation ist, allen Prozessoren zugewiesen wurde. Dies tritt typischerweise bei Matrizen auf niedrigen Stufen im Baum auf, etwa der Matrix C_{00} in dem Beispiel in Abbildung 6.5.1. In diesem Fall ist es aber jedem Prozessor möglich, die Berechnung der Multiplikation auf die lokalen Matrizen zu beschränken. Hierdurch werden konkurrierende, schreibende Zugriffe ebenfalls verhindert.

Insgesamt ergibt sich unter Verwendung der Threadpool-Schnittstellen damit folgendes Verfahren:

```
procedure mul ( α, A, B, β, C )
    procedure rec_mul ( i, α, A, B, C )
        if m_MM(C) ≠ i then return ;
        if A, B und C sind Blockmatrizen then
            for all τ' ∈ S(τ(C)) do
                for all σ' ∈ S(σ(C)) do
                    for all τ'' ∈ S(τ(B)) do
                        rec_mul( i, α, A(τ', τ''), B(τ'', σ'), C(τ', σ') );
        else
            C := C + αAB;
        endif;
    end;

    C := βC;
    for i = 0, ..., p − 1 do tp_run( rec_mul( i, α, A, B, C ) );
    for i = 0, ..., p − 1 do tp_sync( i );
end;
```

Algorithmus 6.5.4: Parallele Matrix-Multiplikation

Dabei wurde angenommen, dass die Skalierung von C parallel erfolgt. Hierfür ist eine eigene Verteilung mittels entsprechender Kostenfunktion notwendig.

6.5.2.1 Numerische Beispiele

Wie im Fall der Matrix-Multiplikation mittels Online-Scheduling wird der parallele Algorithmus anhand der $\mathcal{H}$-Matrix aus dem BEM-Beispiel in Abschnitt 2.4.1 getestet. Da die Bestimmung der Kosten nur für eine Multiplikation mit einem konstanten Rang der Matrix möglich ist, wird auf die Berechnung mit einer konstanten Genauigkeit verzichtet. Die nachfolgende Tabelle gibt die Ergebnisse für einen Rang von $k = 10$ und $\eta = 1$ wieder.

Matrix-Multiplikation, konstanter Rang $k = 10$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	59.2	15.6	95.1	8.2	90.5	5.6	88.6	4.1	90.7
7 920	160.5	41.9	95.8	21.5	93.5	14.8	90.4	11.1	90.0
19 320	523.8	133.9	97.8	69.8	93.8	47.1	92.7	35.8	91.4
43 680	1445.8	370.3	97.6	188.4	95.9	127.8	94.3	100.4	90.0
89 400	3704.3	946.7	97.8	476.0	97.3	322.2	95.8	256.9	90.1
184 040	9498.6	2436.7	97.5	1227.7	96.7	843.4	93.8	648.9	91.5

Die Laufzeiten belegen eine sehr hohe Effizienz der parallelen Multiplikation. Allerdings ist diese geringfügig kleiner als bei der Matrix-Multiplikation mittels Online-Scheduling. Auch beobachtet man Schwankungen der Effizienz, sodass sich nur sehr schwach ein Trend abzeichnet. Dies verdeutlicht die Schwierigkeit bei der Bestimmung der realen Kosten während der Multiplikation (siehe Abschnitt 4.3.2).

Desweiteren ist der Aufwand für die Lastbalancierung in den obigen Zeiten nicht mit eingerechnet. Die nachfolgende Tabelle zeigt diese zusätzlich entstehenden Kosten auf, wobei ein stufenweises LPT-Scheduling genutzt wurde (siehe Abschnitt 4.1.1 sowie 4.2.3). In der Praxis zeigt sich hierbei lediglich eine Abhängigkeit von n, nicht aber von p (vergleiche Bemerkung 4.1.4).

Zeit für die Lastbalancierung						
n	3 968	7 920	19 320	43 680	89 400	184 040
t [s]	0.3	0.7	2.1	5.4	13.1	31.7

Durch einen Vergleich mit der Laufzeit der eigentlichen Multiplikation werden die vergleichsweise hohen Kosten der Lastbalancierung deutlich. Somit eignet sich dieses Verfahren insbesondere dann, wenn mehrere Multiplikationen mit den gleichen Matrizen oder Matrizen gleichen Aufwandes durchzuführen sind.

Ebenfalls interessant ist die Abhängigkeit der parallelen Effizienz der Multiplikation vom Rang der $\mathcal{H}$-Matrix und von der Schwachbesetztheit des zugrundeliegenden Blockclusterbaumes. Die Ergebnisse dieses Vergleichs sind in Abbildung 6.5.3 dargestellt.

Während der Einfluss des Ranges vergleichsweise gering ist, führt eine schwächere Struktur der $\mathcal{H}$-Matrix bedingt durch ein größeres η zu einer Abnahme der Effizienz. In diesem Fall scheinen die ermittelten Kosten nur bedingt mit dem realen Aufwand übereinzustimmen.

6.6 Matrix-Inversion

Gegenstand der folgenden Betrachtungen ist die parallele Matrixinversion. Dabei werden sowohl die Gauß-Elimination, als auch das Newton-Verfahren, wie sie in Abschnitt 3.4 vorgestellt wurden, genutzt und entsprechend parallele Pendants entwickelt. Wie schon bei der

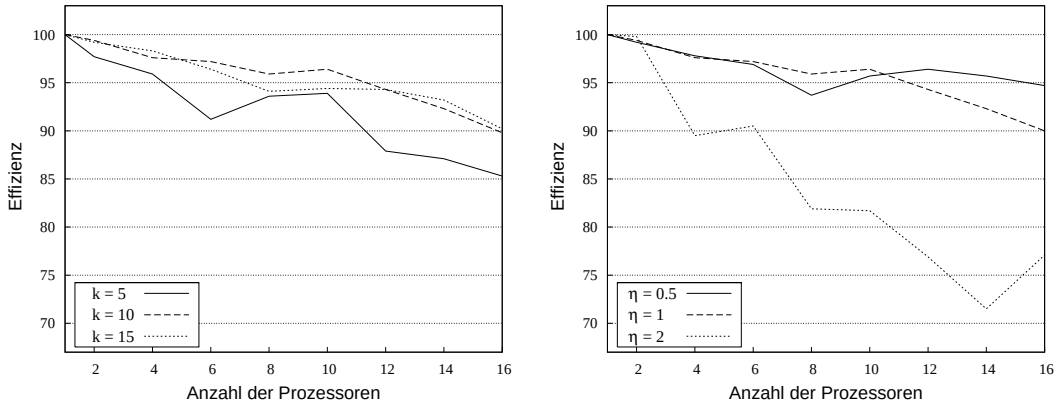


Abbildung 6.5.3: Parallele Effizienz in Abhängigkeit von k (links) bzw. η (rechts)

parallelen Matrixmultiplikation beschränkt sich das betrachtete Rechnermodell allerdings auf eine PRAM.

Der Grund für diese Beschränkung auf das PRAM-Modell liegt in einer gemeinsamen Eigenschaft beider Algorithmen: sie beruhen im wesentlichen auf der Matrixmultiplikation. Damit ergibt sich auch bereits der Ansatzpunkt für die Parallelisierung, indem die Algorithmen aus dem vorherigen Abschnitt genutzt werden. Je nach Inversionsverfahren ergeben sich hierbei unterschiedliche Ergebnisse für die parallele Effizienz.

6.6.1 Gauß-Elimination

Das Gauß'sche Eliminationsverfahren zur Inversion einer $\mathcal{H}$-Matrix, wie es in Abschnitt 3.4.1 beschrieben wird, erfordert ein strikt sequentielles Abarbeiten der Diagonalen. Somit ist mit diesem Algorithmus keine optimale parallele Effizienz möglich. Auf der anderen Seite besteht der Algorithmus, neben den beiden rekursiven Aufrufen, nur aus Matrix-Multiplikationen entsprechend (3.3.1). Letztere lassen sich aber, wie im vorangegangenen Abschnitt beschrieben, optimal verteilen.

Ein Ansatz für einen parallelen Inversionsalgorithmus besteht somit darin, alle Multiplikationen von $\mathcal{H}$-Matrizen während der Gauß-Elimination zu verteilen. Im allgemeinen lässt sich hierbei eine effiziente Parallelisierung der Matrix-Multiplikation nicht bis zu den Blättern des Blockclusterbaumes durchführen, so ist z.B. die parallele Multiplikation von Blättern in Algorithmus 3.3.1 nicht vorgesehen. Aus diesem Grund wird mit $\ell_0 = \ell_0(T)$ die Stufe $p(T) - \ell_0$ von T bestimmt, ab der die Multiplikation sequentiell ausgeführt wird. Mit wachsender Prozessoranzahl muss ℓ_0 allerdings angepasst werden, wobei sich durch den minimalen Grad von 2, der für alle Knoten des Blockclusterbaumes angenommen wurde (siehe Abschnitt 2.1), ein logarithmisches Wachstum ergibt:

$$\ell_0 \sim \log_b p.$$

Die Basis b des entsprechenden Logarithmus ist durch den jeweiligen Knotengrad bestimmt.

Insgesamt ergibt sich der nachfolgende, parallele Algorithmus für binäre Clusterbäume. Auf analoge Weise lässt sich auch der allgemeine Algorithmus 3.4.1 modifizieren.

```
procedure invert_bin ( p, A, C, ℓ, ℓ_0 )
    if ℓ ≥ p(T) − ℓ_0 then invert_bin( A, C );
    else
        invert_bin( p, A_00, C_00, ℓ + 1, ℓ_0 );
        mul( p, 1, C_00, A_01, 0, T_01 );
        mul( p, 1, A_10, C_00, 0, T_10 );
        mul( p, −1, A_10, T_01, 1, A_11 );
        invert_bin( p, A_11, C_11, ℓ + 1, ℓ_0 );
        mul( p, −1, T_01, C_11, 0, C_01 );
        mul( p, −1, C_11, T_10, 0, C_10 );
        mul( p, −1, T_01, C_10, 1, C_00 );
    endif;
end;
```

Algorithmus 6.6.1: Parallele $\mathcal{H}$-Matrix-Inversion

Für die Analyse dieses Verfahrens sei zunächst der Einfachheit halber vorausgesetzt, dass $T(I)$ und $T(J)$ kardinalitätsbalancierte Clusterbäume sind, d.h. alle Cluster der gleichen Stufe im wesentlichen eine gleiche Größe aufweisen (siehe Abschnitt 2.4.1.1). Entsprechend besitzen alle Wege, die von einem Knoten zu seinen Söhnen ausgehen, die gleiche Länge. Hierdurch lässt sich anstelle von ℓ_0 äquivalent ein n_0 definieren, so dass Matrizen mit einer Dimension kleiner als $n_0 \times n_0$ sequentiell behandelt werden. Innerhalb der Invertierung ergeben sich folglich $\mathcal{O}(n/n_0)$ Blöcke, die jeweils zu einem Aufwand von $\mathcal{O}\left(n_0^3\right)$ führen. Der Aufwand, der durch die übrigen, größeren Matrizen entsteht, kann durch $\mathcal{W}_{\mathrm{MI}}(A)$ abgeschätzt werden. Zusammen mit dem Ergebnis der parallelen Matrix-Multiplikationsalgorithmen erhält man somit für die Komplexität der Gauß-Inversion auf p Prozessoren:

$$\mathcal{W}_{\mathrm{MI}}(A,p) = \frac{\mathcal{W}_{\mathrm{MI}}(A,1)}{p} + \mathcal{O}\left(nn_0^2\right).$$

Nicht beachtet wurde hierbei die Abhängigkeit von ℓ_0 und damit auch von n_0 von der Anzahl der Prozessoren. Durch die Kardinalitätsbalancierung und die damit verbundene Verdopplung der Clustergröße bei wachsendem ℓ_0 ergibt sich in diesem Fall

$$n_0 = n_{\min} 2^{\ell_0} = n_{\min} 2^{\log_b p}.$$

Damit folgt auch hier eine große Abhängigkeit von n_0 von der Basis b des Logarithmus bzw. des Knotengrades im Blockclusterbaum. Für einen vollständigen Quadbaum mit $b = 4$ gilt: $n_0 = n_{\min}\sqrt{p}$. Ähnelt der Blockclusterbaum dagegen einem Binärbaum mit $b = 2$, so folgt: $n_0 = n_{\min} p$. Insgesamt folgt für die Komplexität der Gauß-Elimination:

$$\mathcal{W}_{\mathrm{MI}}(A,p) = \frac{\mathcal{W}_{\mathrm{MI}}(A,1)}{p} + \mathcal{O}\left(nn_{\min}^2 p^{2\beta}\right) \tag{6.6.1}$$

mit $\beta \in [0.5, 1]$.

Entscheidend für die Einsatzmöglichkeit in der Praxis sind desweiteren die Konstanten die implizit vor den Termen in (6.6.1) vorhanden sind. Insbesondere das Verhältnis des sequentiellen zum parallelen Anteil gibt Auskunft über die erreichbare parallele Skalierung des Verfahrens. Wie in Abschnitt 6.1 beschrieben, ist die maximale parallele Skalierung durch den nicht parallelisierten Anteil $0 \leq c_s \leq 1$ am Gesamtaufwand über das Gesetz von Amdahl bestimmt:

$$S(p) = \frac{1}{c_s + \frac{1-c_s}{p}},$$

Im Falle der Gauß-Elimination treten zwei jeweils entgegengesetzte Faktoren auf die c_s bestimmen. Zum einen führt eine größere Zahl von Prozessoren durch ein kleineres ℓ_0 bzw. größeres n_0 zu einem höheren sequentiellen Anteil und zum anderen sinkt c_s mit wachsender Problemgröße, da der Aufwand für die Inversion nach (3.4.1) und (3.3.2) logarithmisch-linear steigt. Insbesondere die letztgenannte Eigenschaft sollte insgesamt zu einem isoeffizienten (siehe Abschnitt 6.1) und damit skalierbaren Verfahren führen.

Auch für den allgemeinen Fall lässt sich die Stufe ℓ_0 durch eine Blockgröße ausdrücken, wodurch sich eine identische Abschätzung des Aufwandes ergibt. Notwendig ist hierbei aber auch eine zusätzliche Bedingung an die Größe der Diagonalblöcke, um die Zunahme von n_0 mit wachsendem p zu beschränken. Da eine unbeschränkte Diagonalblockgröße allerdings zu einer kubischen Gesamtkomplexität führen kann, besitzt eine Begrenzung auf ein Vielfaches von $n_{\min}$ keine Auswirkungen auf den praktischen Einsatz.

Lemma 6.6.1 *Sei $c > 0$ und sei T ein Blockclusterbaum derart, dass für alle $(\tau, \tau) \in \mathcal{L}(T)$ gelte: $|\tau| \leq cn_{\min}$. Desweiteren sei $\beta \in (0, 1]$. Dann besitzt die parallele Inversion einer $\mathcal{H}$-Matrix $A \in \mathcal{H}(T, k)$ mittels Algorithmus 6.6.1 auf p Prozessoren hat eine Komplexität von:*

$$\mathcal{W}_{\mathrm{MI}}(A, p) = \frac{\mathcal{W}_{\mathrm{MI}}(A)}{p} + \mathcal{O}\left(n(n_{\min}^2 p^{2\beta})\right). \tag{6.6.2}$$

Beweis: Sei ℓ_0 die Stufe sequentiellen Abarbeitens für ein festes p und sei

$$n_0 = \max\left\{|\tau| \,|\, (\tau, \tau) \in T \ \wedge \ p(T(\tau, \tau)) = \ell_0\right\}.$$

Die Menge der Diagonalknoten mit Größe n_0 ist dann durch n/n_0 begrenzt. Somit folgt für die Diagonale ein Aufwand von nn_0^2.

Zu zeigen bleibt: $n_0 = \mathcal{O}\left(p^\beta n_{\min}\right)$. Seien hierzu $T(I), T(J)$ die T zugrundeliegenden Blockclusterbäume. Dann gilt für alle Knoten $\tau \in T(I) \cup T(J)$: $|V(T(\tau))| \geq |\tau|/(cn_{\min})$. Der Beweis erfolgt durch Induktion. Für den Induktionsanfang überträgt sich die vorausgesetzte Eigenschaft von T auf $T(I)$ bzw. $T(J)$. Im Induktionsschluss gelte die Bedingung für die Söhne $\tau_0, \ldots, \tau_{m-1}$ von τ. Dann folgt: $|V(T(\tau))| = \sum_{i=0}^{m-1} |V(T(\tau_i))| \geq \sum_{i=0}^{m-1} |\tau_i|/(cn_{\min}) = |\tau|/(cn_{\min})$.

Somit gilt aber auch für einen Knoten $(\tau, \tau) \in T$: $|V(T(\tau))| \geq |\tau|/(cn_{\min})$. Da $|V(T(\tau))|$ dem Grad der Parallelität und somit dem maximal erlaubten p entspricht folgt mit $\beta = 1$: $n_0 \leq pcn_{\min}$. □

Wie schon im kardinalitätsbalancierten Fall richtet sich die Konstante β nach dem durchschnittlichen Knotengrad des Blockclusterbaumes. Je komplexer die Struktur dabei ist, desto kleiner ist β und desto geringer fällt der sequentielle Anteil der Inversion aus.

Bemerkung 6.6.2 *Wie in [Tis02] gezeigt wurde, ist Algorithmus 6.6.1 optimal für vollbesetzte Matrizen, d.h. die Komplexität beträgt $\mathcal{O}\left(n^3/p\right)$ für eine Matrix $A \in \mathbb{R}^{n \times n}$.*

In Abschnitt 6.5 wurden zwei verschiedene Parallelisierungsansätze für die Multiplikation beschrieben. Diese beiden Algorithmen lassen sich auch für die Matrix-Inversion benutzen, wobei die einfachere Variante das Online-Scheduling-Verfahren 6.5.3 darstellt, welches direkt in Algorithmus 6.6.1 aufgerufen werden kann.

Verwendet man dagegen Algorithmus 6.5.4, so ist eine analoge Vorgehensweise nicht möglich, da hierfür zunächst die Berechnung der Kostenfunktion für jede Matrix-Multiplikation erforderlich ist. Die Bestimmung der Kosten ist allerdings vergleichsweise aufwändig (siehe Bemerkung 6.5.3), womit die Berechnung vor jedem Multiplikationsaufruf, zu einer verminderten parallelen Effizienz der Inversion führt.

Wie bereits in Abschnitt 3.4.1 beschrieben wurde, sind die Matrix-Operationen, die bei der Gauß-Elimination vorkommen, ebenfalls Bestandteil der Matrix-Multiplikation. Damit folgt aber auch, dass bei der Simulation der Multiplikation $A := A \cdot A$, bereits alle Einzeloperationen der Inversion $C := A^{-1}$ auftreten. Durch einen Simulationschritt vor der Inversion lassen sich auf diese Weise alle Informationen über die auftretenden Kosten während der Gauß-Elimination bestimmen. Die Lastbalancierung beschränkt sich anschließend darauf, die gesammelten Kosten pro Matrixblock auf die jeweils involvierten Blockcluster bzw. Matrizen zu beschränken.

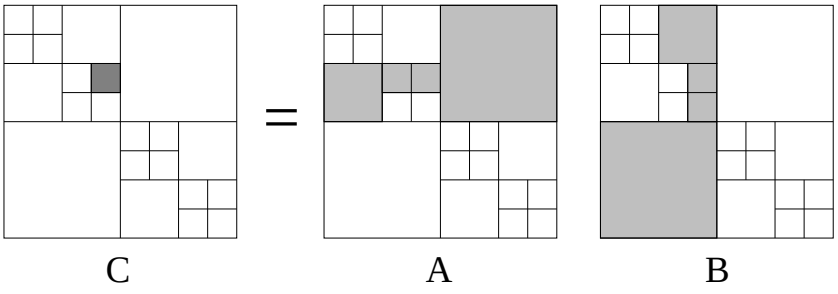


Abbildung 6.6.1: Kostenbestimmung bei der Gauß-Elimination

Für einen Matrixblock C' enthalte die Menge $\mathcal{C}(C') = \left\{(b_A^i, b_B^i, c^i)\right\}_{i=0}^{n_{C'}}$ mit $b_A^i, b_B^i \in V(T)$ und $c^i \in \mathbb{R}_{\geq 0}$ die Paare von Blockclustern und Kosten, wie sie durch die vollständige Simu-

lation der Matrix-Multiplikation bestimmt wurden. In Abbildung 6.6.1 sind die entsprechenden Paare für eine Zielmatrix beispielhaft angegeben. Für die Multiplikation $C'' := A'B'$ mit $v(C') \subseteq v(C'')$ ist die Kostenfunktion c_{MM} somit definiert durch

$$c_{\mathrm{MM}}(C') = \sum_{\substack{(b_A,b_B,c)\in\mathcal{C}(C'),\\ b_A\subseteq v(A') \wedge b_B\subseteq v(B')}} c \quad .$$

Die Verteilung des Produktes $C_{00} := A_{00}B_{00}$ in dem Beispiel in Abbildung 6.6.1 führt zu den in Abbildung 6.6.2 dargestellten Blockclustern, welche für die Berechnung der Kosten für den Matrixblock C' betrachtet werden müssen.

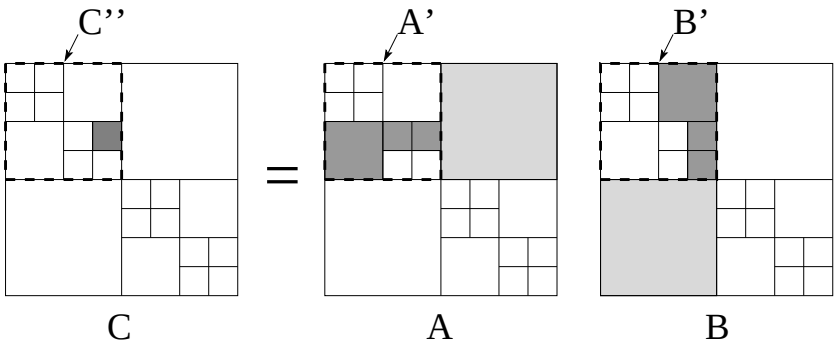


Abbildung 6.6.2: Reduktion der Blockclusterpaare

Auf diese Weise kann der Aufwand für die Lastbalancierung erheblich reduziert werden. Lediglich die Bestimmung der zulässigen Verteilungsfunktion vor jeder Multiplikation wirkt sich weiterhin negativ auf die parallele Effizienz der Gauß-Elimination aus. Allerdings ist deren Anteil vergleichsweise gering, was auch die nachfolgenden numerischen Beispiele zeigen.

6.6.1.1 Numerische Beispiele

Zunächst wird die Online-Scheduling-Variante der Gauß-Elimination betrachtet. Die zu invertierende $\mathcal{H}$-Matrix ist dabei durch das FEM-Beispiel in Abschnitt 2.4.2 definiert.

Die erste Tabelle zeigt die Laufzeit bzw. die Effizienz für den Fall einer Approximation mit einem konstanten Rang von $k = 10$. Für η in der Zulässigkeitsbedingung (2.4.4) wurde der Wert $\eta = 1$ gewählt.

Gauß-Elimination (online), konstanter Rang $k = 10$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
4 096	22.3	6.2	89.9	3.6	77.7	2.8	66.5	2.5	56.3
8 100	83.0	22.6	92.0	12.9	80.4	10.1	68.5	8.5	61.2
16 384	172.3	46.3	93.0	25.6	84.1	19.3	74.5	17.0	63.4
32 761	578.1	153.9	93.9	86.5	83.5	65.2	73.9	55.8	64.8
65 536	1126.5	295.8	95.2	162.6	86.6	121.6	77.2	103.7	67.9
131 044	3563.9	930.1	95.8	519.8	85.7	392.1	75.7	334.2	66.6
262 144	7362.3	1899.6	96.9	1026.6	89.6	779.3	78.7	667.2	69.0

Deutlich erkennbar ist die Abhängigkeit der parallelen Effizienz von der Anzahl der Prozessoren, welche durch die sequentielle Inversion der Diagonalen von A hervorgerufen wird. Allerdings steigt die Effizienz mit wachsender Problemgröße, womit das Verfahren insgesamt als skalierbar anzusehen ist.

Verwendet man (6.1.2) und bestimmt den sequentiellen Anteil c_s aus den vorliegenden Daten, so erhält man Werte zwischen 1% und 5%. Hierbei bestätigt sich die erwähnte Abhängigkeit von p und n.

Im nächsten Beispiel erfolgen die gleichen Berechnungen mit einer konstanten Genauigkeit von $\varepsilon = 10^{-6}$. Die ermittelten Werte zeigt die folgende Tabelle.

Gauß-Elimination (online), konstante Genauigkeit $\varepsilon = 10^{-6}$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
4 096	8.7	2.6	84.5	1.6	70.1	1.3	57.5	1.1	47.6
8 100	26.1	7.5	87.1	4.4	74.6	3.4	64.3	3.0	55.4
16 384	73.7	20.6	89.6	12.0	76.8	9.2	66.4	8.1	56.7
32 761	197.2	54.2	90.9	31.5	78.4	24.2	67.8	21.2	58.2
65 536	491.2	133.4	92.1	76.4	80.4	59.4	68.9	51.1	60.1
131 044	1243.0	334.4	92.9	194.5	79.9	150.5	68.8	129.9	59.8
262 144	3003.1	796.9	94.2	448.4	83.7	342.0	73.2	296.1	63.4

Es zeigt sich eine ähnliches Effizienzverhalten wie im Falle eines konstanten Ranges. Lediglich die absoluten Werte sind geringfügig kleiner. Vergleicht man die Laufzeiten in beiden Fällen, so ist ein möglicher Grund hierfür die geringere Datenmenge der Matrix bei konstanter Genauigkeit. Interessant ist deshalb der Einfluss des Ranges bzw. der Genauigkeit auf die parallele Effizienz. Die Ergebnisse dieser Untersuchung sind in Abbildung 6.6.3 dargestellt. Die Problemdimension wurde dabei bei $n = 65\,536$ belassen.

Insbesondere bei einem kleinen Rang bzw. einem großen Wert für ε ergibt sich eine geringe Effizienz der parallelen Gauß-Elimination. In diesen Fällen ist der Einfluss der Diagonalen und damit der sequentielle Anteil am Gesamtaufwand stärker ausgeprägt. Mit wachsendem

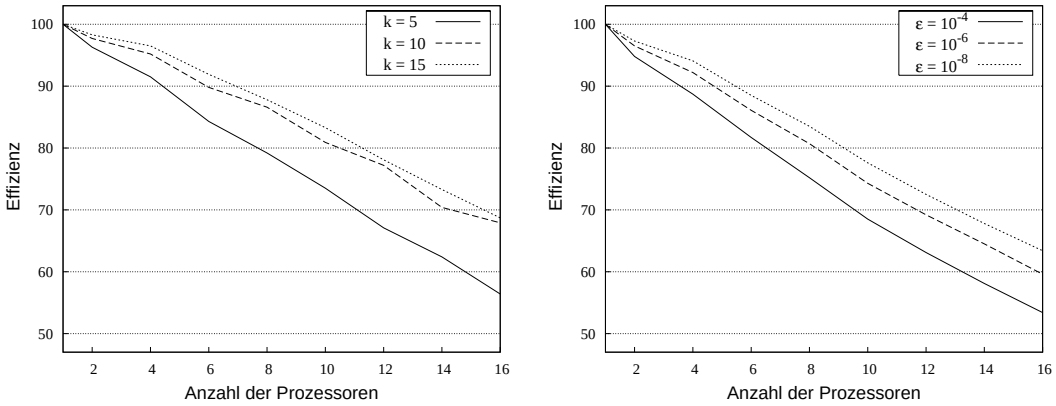


Abbildung 6.6.3: Parallele Effizienz in Abhängigkeit von k (links) bzw. ε (rechts)

Rang bzw. Genauigkeit sinkt allerdings dieser Einfluss, und entsprechend steigt die parallele Effizienz an. Vergleicht man diese Beobachtung mit Bemerkung 6.6.2, so ergäbe sich für $\varepsilon \to 0$ eine maximale Effizienz. Allerdings ist ε in der Praxis typischerweise durch die Maschinengenauigkeit begrenzt und die dabei beobachtete Komplexität entspricht nicht dem kubischen Aufwand für vollbesetzte Matrizen. Somit bleibt das Resultat (6.6.2) weiterhin gültig.

Im folgenden wird die Matrix-Multiplikation mittels Offline-Scheduling innerhalb des Inversionsalgorithmus genutzt. Hierbei beschränken sich die Beispiele allerdings auf den Fall eines konstanten Ranges, da nur hierbei eine Vorhersage über die auftretenden Kosten erfolgen kann. Die Ergebnisse für den Fall von $k = 10$ lauten:

Gauß-Elimination (offline), konstanter Rang $k = 10$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
4 096	22.3	6.4	87.1	3.7	76.0	2.9	65.1	2.5	55.0
8 100	83.0	24.0	86.7	13.8	75.4	10.2	67.7	8.6	60.3
16 384	172.3	49.0	87.9	28.0	77.0	20.9	68.6	17.5	61.7
32 761	578.1	163.9	88.2	91.7	78.8	67.7	71.1	56.4	64.0
65 536	1126.5	316.7	88.9	178.4	78.9	133.9	70.1	111.5	63.2
131 044	3563.9	1001.1	89.0	565.1	78.8	413.0	71.9	354.9	62.8
262 144	7362.3	2075.1	88.7	1187.6	77.5	908.2	67.6	758.4	60.7

Die Effizienz der parallelen Gauß-Elimination mit Offline-Scheduling entspricht im wesentlichen der Online-Scheduling-Variante. Allerdings ist das Verhalten bezüglich der Problemdimension verschieden. Während bei der letzteren Verteilungsmethode die Effizienz mit n ansteigt, bleibt sie in diesem Fall relativ konstant. Die Schwierigkeit einer exakten Kosten-

bestimmung führt in diesem Fall offensichtlich zu einer Beeinträchtigung des Skalierungsverhaltens.

Aufgrund der vergleichbaren Laufzeiten zwischen dem Offline- und dem Online-Scheduling wird deutlich, dass der Aufwand für die Lastbalancierung während der Inversion auf einen minimalen Wert reduziert werden konnte. Zum Vergleich sind in der folgenden Tabelle die entsprechenden Zeiten für die Matrix-Inversion mittels schneller und normaler Lastbalancierung bei einer Problemgröße von $n = 65\,536$ gegenübergestellt.

Kostenbestimmung	$p = 4$	$p = 8$	$p = 12$	$p = 16$
vor Inversion	316.7 s	178.4 s	133.9 s	111.5 s
während Inversion	342.6 s	271.8 s	290.7 s	283.5 s

Insbesondere für größere Prozessorzahlen wird der hohe Aufwand, der durch die Bestimmung der Kostenfunktion bei der normalen Verteilung verursacht wird, deutlich. Die Zeiten für die Bestimmung der Einzelkosten vor der Inversion stimmen mit den in Abschnitt 6.5.2 angegebenen Werten überein und wurden wie dort nicht bei den aufgeführten Laufzeiten berücksichtigt.

Abschließend erfolgt eine Untersuchung des Ranges und der Schwachbesetztheit auf die parallele Effizienz der Offline-Inversion. Die Resultate hierfür bei einer Problemdimension von $n = 65\,536$ sind in Abbildung 6.6.4 dargestellt.

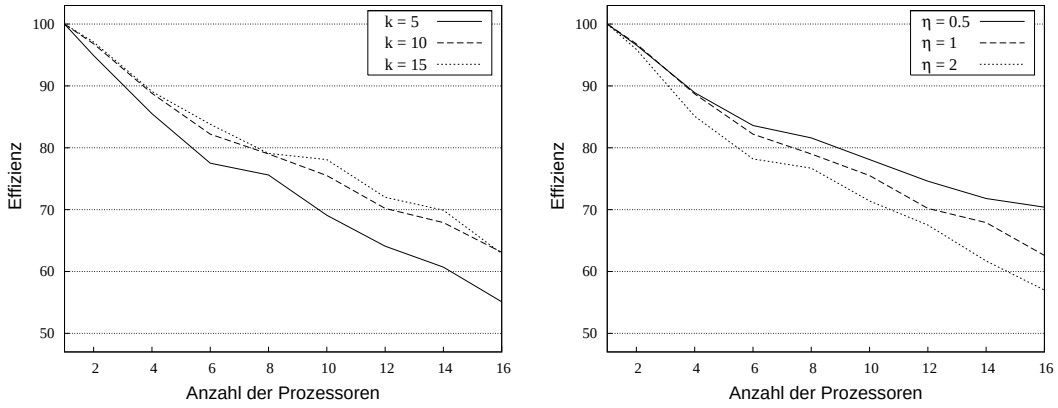


Abbildung 6.6.4: Parallele Effizienz in Abhängigkeit von k (links) bzw. η (rechts)

Wie auch schon bei der Verwendung des Online-Scheduling-Verfahrens steigt die parallele Effizienz mit dem Rang an. Das gleiche Verhalten ergibt sich bei einer Verringerung der Schwachbesetztheit, d.h. einem größeren Wert von c_{sp}, bedingt durch ein kleineres η. Der Grenzfall $\eta = 0$ führt zu einer vollbesetzten Matrix und somit nach Bemerkung 6.6.2 zu einer maximalen Effizienz.

6.6.2 Newton-Iteration

Da die Newton-Iteration zur Berechnung der Inversen von A aus Algorithmus 3.4.3 vollständig auf der Addition und Multiplikation von $\mathcal{H}$-Matrizen basiert, kann die Parallelisierung auf die jeweiligen parallelen Algorithmen 6.4.2 und 6.5.3 bzw. 6.5.4 zurückgeführt werden. Für die Komplexität ergibt sich hierbei ein optimales Resultat.

Bemerkung 6.6.3 *Bei gegebenem Startwert besitzt die parallele Berechnung der approximierten Inversen einer $\mathcal{H}$-Matrix bis zu einer Genauigkeit von $\varepsilon > 0$ mittels Newton-Iteration einen Aufwand von*

$$\mathcal{W}_{\mathrm{MI,Newton}}(A,p) = \mathcal{W}_{\mathrm{MM}}(A,p)\mathcal{O}\left(\log\log\varepsilon^{-1}\right). \tag{6.6.3}$$

Die Komplexität der parallelen Newton-Iteration ist damit asymptotisch besser als die der Gauß-Elimination. Allerdings ist, wie auch im sequentiellen Fall der letztere Algorithmus im allgemeinen deutlich schneller. Zudem wurde hierbei die Berechnung des Startwertes ausgenommen. Falls hierfür z.B. das Gauß'sche Eliminationsverfahren verwendet wird, entspricht auch der theoretische Gesamtaufwand der Newton-Iteration diesem Verfahren. Aufgrund der Vielzahl an Multiplikationen während des Iterationsprozesses dominiert aber auch in diesem Fall die optimale Komplexität der parallelen Matrix-Multiplikation.

6.6.2.1 Numerische Beispiele

Um die letzte Aussage zu untermauern, sind in der folgenden Tabelle Ausführungszeit und parallele Effizienz für die Inversion einer $\mathcal{H}$-Matrix aus dem FEM-Beispiel in Abschnitt 2.4.2 dargestellt. Dabei beschränkt sich das Beispiel auf die Arithmetik mit konstantem Rang. Neben der Laufzeit und der parallelen Effizienz ist auch die Anzahl der Iterationen aufgeführt.

Als Startwert X_0 der Iteration wurde eine Approximation der Inversen mittels Gauß-Elimination berechnet. Der hierbei verwendete Rang betrug $k = 3$. Die anschließende Newton-Iteration wurde bis zu einem Fehler von $\|I - XA\| \leq 10^{-4}$ durchgeführt, wobei X die genäherte Lösung in jedem Schritt darstellt.

Newton-Iteration, konstanter Rang $k = 10$										
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$		It.
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E	
1 024	14.1	4.0	89.0	2.3	76.3	1.8	67.1	1.6	55.8	2
2 116	115.2	30.6	94.0	16.6	86.6	11.9	80.9	9.7	74.2	3
4 096	331.5	87.6	94.6	46.5	89.2	33.7	82.1	28.3	73.3	4
8 100	2772.4	724.5	95.7	373.3	92.8	267.3	86.4	209.5	82.7	8
16 384	10809.7	2775.8	97.4	1424.5	94.9	1005.1	89.6	737.2	91.7	16

Die Effizienz der Newton-Iteration ist, insbesondere für große p, deutlich höher als bei der Gauß-Elimination und erreicht mit wachsender Problemdimension die erwarteten, optimalen

Werte. Man beachte hierbei, dass die Zeit für die Berechnung des Startwertes ebenfalls in der Laufzeit enthalten ist.

Dagegen nimmt die Anzahl der Iteration deutlich schneller zu, als es die Theorie voraussagt. Der Grund hierfür ist in einem zu kleinen Rang zu suchen, welcher zu einem entsprechend schlechten Startwert führt. Für wachsende Problemgrößen ist dieser entsprechend anzupassen.

Abschließend lässt sich durch einen direkten Vergleich der Ausführungszeit mit den Werten in Abschnitt 6.6.1 die geringe Nutzbarkeit der Newton-Iteration für den praktischen Gebrauch belegen.

6.7 LU-Zerlegung

Ähnlich dem Gauß'schen Eliminationsverfahren 3.4.1 besteht auch Algorithmus 3.5.1 zur Berechnung der LU-Zerlegung einer $\mathcal{H}$-Matrix aus Rekursionen und Matrix-Multiplikationen. Somit bietet sich die gleiche Herangehensweise zur Parallelisierung an, die schon bei der Matrix-Inversion verwendet wurde: die Verteilung der Multiplikationen.

Im Gegensatz zur Gauß-Elimination werden die Außerdiagonalblöcke allerdings nicht durch Matrix-Multiplikationen bestimmt, sondern wie die Diagonale rekursiv. Betrachtet man die Algorithmen 3.5.4 und 3.5.5 bzw. 3.5.6, so ergeben sich die folgenden Rekursionsformeln für die Komplexität der LU-Zerlegung im Falle eines binären Clusterbaumes:

$$\mathcal{W}_{\mathrm{LU}}(n,p) = \mathcal{W}_{\mathrm{LU}}\left(\frac{n}{2},p\right) + 2\mathcal{W}_{\mathrm{solve}}\left(\frac{n}{2},p\right) + \mathcal{W}_{\mathrm{MM}}\left(\frac{n}{2},p\right)$$

und

$$\mathcal{W}_{\mathrm{solve}}(n,p) = 4\mathcal{W}_{\mathrm{solve}}\left(\frac{n}{2},p\right) + 2\mathcal{W}_{\mathrm{MM}}\left(\frac{n}{2},p\right).$$

Besonders deutlich wird der hohe Anteil der Rekursionen gegenüber den Multiplikationen. Da durch die rekursiven Funktionen auch der sequentielle Aufwand bestimmt wird, ist kein effizientes paralleles Verfahren möglich. So führt der Einsatz von mehreren Prozessoren nur zu einer Reduktion der auftretenden Konstanten.

Löst man die obigen Rekursionsformeln auf, so gelangt man zu folgendem, negativen Ergebnis für den Aufwand der LU-Zerlegung.

Lemma 6.7.1 *Die parallele LU-Zerlegung einer $\mathcal{H}$-Matrix $A \in \mathcal{H}(T,k)$ auf p Prozessoren hat eine Komplexität von:*

$$\mathcal{W}_{\mathrm{LU}}(A,p) = \mathcal{W}_{\mathrm{LU}}(A,1). \tag{6.7.1}$$

6.7.1 Numerische Beispiele

Trotz des wenig hoffnungsvollen Komplexitätsresultats soll dennoch der Einfluss der Prozessoranzahl auf den Aufwand der LU-Zerlegung untersucht werden. Die Basis bildet hierfür

wieder das BEM-Beispiel aus Abschnitt 2.4.1. Zunächst erfolgt die LU-Faktorisierung bei einem konstanten Rang von $k = 10$ und $\eta = 1$.

LU-Zerlegung, konstanter Rang $k = 10$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	19.3	6.0	79.8	4.1	58.5	3.9	41.0	3.2	37.2
7 920	54.8	17.1	80.1	11.4	59.9	9.9	46.4	8.6	39.6
19 320	181.3	55.3	82.0	37.1	61.2	32.0	47.2	27.5	41.2
43 680	506.0	153.4	82.5	100.4	63.0	86.2	48.9	73.3	43.2
89 400	1307.6	390.6	83.7	253.6	64.4	216.8	50.3	182.7	44.7
184 040	3376.5	1002.4	84.2	649.8	65.0	549.2	51.2	467.3	45.2

Deutlich sichtbar ist der starke Abfall der parallelen Effizienz mit einer zunehmenden Anzahl von Prozessoren. Hierdurch ist dieses parallele Verfahren bei mehr als 8 CPUs nur noch bedingt einsetzbar. Auch nimmt der Anstieg der Effizienz bei wachsender Problemdimension deutlich ab, was das beschränkte Skalierungsverhalten bestätigt.

Auch hierbei lässt sich der sequentielle Anteil am Gesamtaufwand mittels (6.1.2) zurückrechnen. Dieser beträgt in den Beispielen etwa 6% bis 10% und liegt damit deutlich über den Werten für die Gauß-Elimination. Damit ist der Anteil der Multiplikationen zwar noch vergleichsweise groß, allerdings die Möglichkeiten der Parallelisierung auch stark begrenzt.

Das gleiche Beispiel wurde auch mit einer konstanten Genauigkeit von $\varepsilon = 10^{-6}$ gerechnet. Die sich daraus ergebenden Werte sind in der folgenden Tabelle zusammengefasst.

LU-Zerlegung, konstante Genauigkeit $\varepsilon = 10^{-6}$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
3 968	34.2	10.4	82.1	7.4	57.6	6.5	44.0	5.8	37.0
7 920	109.8	34.3	80.0	24.6	55.9	21.6	42.3	19.5	35.3
19 320	404.0	123.6	81.7	87.1	58.0	80.8	41.7	73.6	34.3
43 680	1275.7	369.4	86.3	289.3	55.1	270.6	39.3	250.3	31.8
89 400	3943.3	1225.9	80.4	824.1	59.8	736.3	44.6	669.5	36.8
184 040	11002.8	3459.2	79.5	2320.0	59.3	2072.0	44.3	1869.9	36.8

Das sich ergebende Bild ist ähnlich, aber nicht identisch zur LU-Zerlegung mit einem konstanten Rang. Als wesentlichen Unterschied läßt sich eine Stagnation der parallelen Effizienz bei wachsender Problemdimension und damit das aus der Theorie folgende, nichtskalierende Verhalten ausmachen.

In Abbildung 6.7.1 wurde die parallele Effizienz für verschiedene Werte von k und ε bei einer konstanten Problemgröße von $n = 43\,680$ aufgetragen.

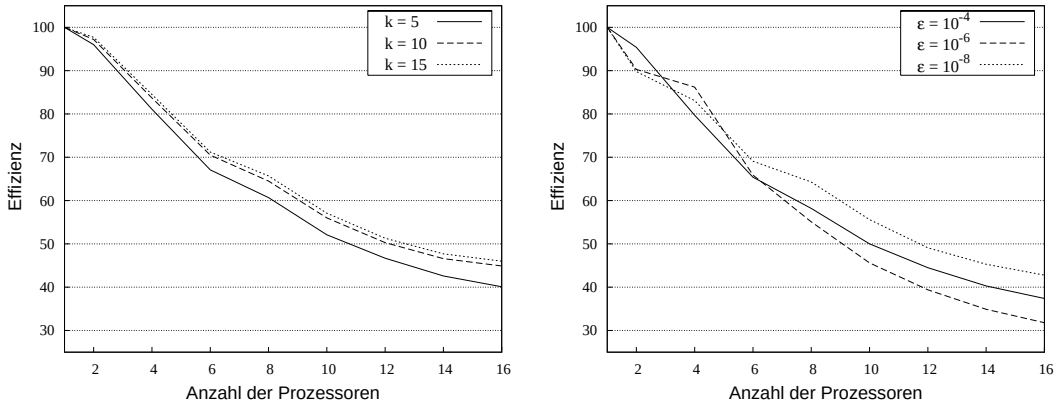


Abbildung 6.7.1: Parallele Effizienz in Abhängigkeit von k (links) bzw. ε (rechts)

Während eine Erhöhung des Ranges zu einer leichten Zunahme der Effizienz führt, ergibt sich bei einer Änderung der Genauigkeit keine solch eindeutige Tendenz. In beiden Fällen belegen die dargestellten Werte aber das schlechte Skalierungsverhalten der parallelen LU-Zerlegung.

7 Gebietszerlegung

Betrachtet wird in diesem Kapitel die Methode der Gebietszerlegung. Wie der Name andeutet, wird hierbei bei dem zu lösenden Problem das Gebiet Ω in einzelne Teilgebiete Ω_i aufgeteilt. Üblicherweise zerfällt dabei auch die eigentliche Aufgabe in Teilprobleme, welche mehr oder weniger unabhängig voneinander gelöst werden können. Im Gegensatz zu den Verfahren in Kapitel 6 erfolgt hierbei die Parallelisierung nicht direkt, sondern indirekt über die parallele Lösung der einzelnen Teilprobleme.

Ein Vorteil dieser Methode ist, dass häufig sequentielle Verfahren bei der Lösung der Teilprobleme angewendet werden können, d.h. keine Reimplementierung von Algorithmen notwendig ist. Dies gilt auch für $\mathcal{H}$-Matrizen und deren Arithmetik.

Eine einfache Anwendung der $\mathcal{H}$-Matrizen gelingt über die Nutzung der *nichtüberlappenden* Gebietszerlegung. Dabei wird die globale Indexmenge I in disjunkte Teilmengen $(I_i)_{i=0}^{m-1}$ zerlegt. Die Anzahl m der Teilgebiete ist dabei üblicherweise an die Anzahl der Prozessoren gekoppelt, muss mit dieser aber nicht übereinstimmen. Für jede Indexmenge I_i kann anschließend unabhängig ein Clusterbaum $T(I_i)$ konstruiert werden. Die Definition der Blockclusterbäume $T(I_i \times I_j)$ bzw. der $\mathcal{H}$-Matrizen über diesen erfolgt dann entsprechend. Man beachte hierbei die Ähnlichkeit dieser Konstruktion mit einem Verfahren zur Bestimmung eines üblichen Clusterbaumes, z.B. der binären Raumpartitionierung (siehe Abschnitt 2.4.1.1), bei welchem ebenfalls die Indexmenge in disjunkte Teilmengen zerlegt wird.

In den folgenden Abschnitten wird für die Beispiele aus Abschnitt 2.4.2 und 2.4.1 der Zugang über die Technik der Gebietszerlegung diskutiert. Dabei führen die verschiedenen Eigenschaften der beiden Probleme zu unterschiedlichen Ansätzen bei der Wahl der Zerlegung. Desweiteren ist die Auswahl von effizienten Algorithmen verschieden.

Die alternative Technik der Gebietszerlegung mit *überlappenden* Teilgebieten ist ebenfalls möglich, allerdings mittels $\mathcal{H}$-Matrizen schwieriger zu implementieren. Wie der Name andeutet, sind die einzelnen Teilmengen I_i in diesem Fall nicht disjunkt. Durch das Vorhandensein einzelner Indizes in verschiedenen Clusterbäumen bzw. Blockclusterbäumen ist im Kontext der $\mathcal{H}$-Matrizen zusätzlicher Aufwand nötig, um arithmetische Operationen durchzuführen.

Für einen allgemeinen Überblick über die Gebietszerlegungsmethode mit einer Diskussion der unterschiedlichen Eigenschaften von überlappenden und nichtüberlappenden Zerlegung sei z.B. auf [CM94] verwiesen.

7.1 Gebietszerlegung für die Methode der finiten Elemente

In diesem Abschnitt soll Gebietszerlegung auf die Methode der finiten Elemente angewendet werden. Als Ausgangsproblem dient hierbei das in Abschnitt 2.4.2 vorgestellte Beispiel einer partiellen Differentialgleichung. Eine besondere Eigenschaft der dort definierten Systemmatrix ist deren Schwachbesetztheit. Diese ergab sich aus den lokalen Trägern der Basisfunktionen, die bei der Diskretisierung zum Einsatz kommen. Innerhalb der Gebietszerlegung gestattet es diese Eigenschaft, die Teilgebiete so zu wählen, dass sie durch einen *inneren Rand*, auch *Kopplungsrand* genannt, voneinander getrennt sind. Die Träger der Basisfunktionen der einzelnen Gebiete schneiden sich somit nicht (siehe Abbildung 7.1.1). Damit ergibt sich für die Systemmatrix A aus (2.4.7) eine Blockaufteilung, die sehr effiziente parallele Algorithmen erlaubt.

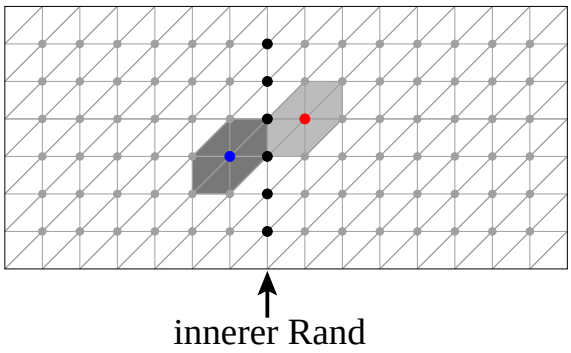


Abbildung 7.1.1: Entkopplung der Teilgebiete durch inneren Rand

Zunächst aber zur verwendeten Notation. Sei I wieder die globale Indexmenge mit $n = |I|$ und der Zerlegung $I_0, \ldots, I_{p-1}, I_\Sigma$ mit $I = \left(\dot{\cup}_{i=0}^{p-1} I_i\right) \cup I_\Sigma$. Die Indexmengen I_i, $0 \leq i < p$, entsprechen dabei den jeweiligen Teilgebieten, während I_Σ die Indizes des inneren Kopplungsrandes enthält. Mit n_i wird die Kardinalität von I_i benannt, wobei $n_{\max} = \max_{i=0}^{p-1} n_i$ sei. Analog dazu sei $n_\Sigma = |I_\Sigma|$. In der Praxis sollte der innere Rand im Vergleich zu den Teilgebieten vergleichsweise klein sein, um eine hinreichende parallele Effizienz zu erhalten. Im folgenden sei deshalb

$$n_{\max} \geq n_\Sigma \tag{7.1.1}$$

angenommen.

Ordnet man die Indizes in $I_0, \ldots, I_{p-1}$ und I_Σ entsprechend an, so ergibt sich für A die

erwähnte Blockdarstellung

$$A = \begin{pmatrix} A_{00} & 0 & \dots & 0 & A_{0,\Sigma} \\ 0 & A_{11} & \dots & 0 & A_{1,\Sigma} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & A_{p-1,p-1} & A_{p-1,\Sigma} \\ A_{\Sigma,0} & A_{\Sigma,1} & \dots & A_{\Sigma,p-1} & A_{\Sigma,\Sigma} \end{pmatrix}, \tag{7.1.2}$$

mit $A_{ii} \in \mathbb{R}^{I_i \times I_i}, A_{i,\Sigma} \in \mathbb{R}^{I_i \times I_\Sigma}, A_{\Sigma,i} \in \mathbb{R}^{I_\Sigma, I_i}$ und $A_{\Sigma,\Sigma} \in \mathbb{R}^{I_\Sigma \times I_\Sigma}$.

Mit den Clusterbäumen $T(I_i)$ und $T(I_\Sigma)$ und entsprechenden Blockclusterbäumen $T_{ii} = T(I_i \times I_i), T_{i,\Sigma} = T(I_i \times I_\Sigma), T_{\Sigma,i} = T(I_\Sigma \times i)$ und $T_{\Sigma,\Sigma} = T(I_\Sigma \times I_\Sigma)$ kann jede der auftretenden Matrizen als $\mathcal{H}$-Matrix dargestellt werden.

Interessant ist hierbei vor allen Dingen die Inversion von A, die in Abschnitt 7.1.1 diskutiert wird. Alternativ lässt sich die LU-Zerlegung verwenden, die Gegenstand von Abschnitt 7.1.2 ist. Zur Lösung mittels Iterationsverfahren wird weiterhin die Matrix-Vektor-Multiplikation aus Abschnitt 7.1.3 benötigt.

Die folgende Bemerkung beschreibt einen Spezialfall für die Größen der Teilgebiete und des inneren Randes. Aufgrund der Relevanz für die Praxis wird dieser Spezialfall in den folgenden Abschnitten bei der Komplexitätsanalyse stets als Beispiel verwendet.

Bemerkung 7.1.1 *Aus Gründen einer guten Lastbalancierung sei angenommen, dass $n_i = n_j$ für $i \neq j$. Somit folgt: $n = pn_i + n_\Sigma$. Für einen hinreichend kleinen Kopplungsrand erhält man demnach:*

$$n_i = \mathcal{O}\left(\frac{n}{p}\right). \tag{7.1.3}$$

Sei weiterhin I_Σ von minimaler Ordnung bzgl. der Raumdimension d, d.h. $n_\Sigma = n_i^{(d-1)/d}$. Dann folgt:

$$\mathcal{O}(n_\Sigma) = \mathcal{O}\left(pn_i^{(d-1)/d}\right) = \mathcal{O}\left(p\left(\frac{n}{p}\right)^{(d-1)/d}\right) = \mathcal{O}\left(p^{1/d}n^{(d-1)/d}\right). \tag{7.1.4}$$

Ein Verfahren, um eine Aufteilung von I entsprechend zu Bemerkung 7.1.1 zu berechnen, ist in [Zum03] beschrieben. Dabei kommen die in Abschnitt 4.2.2 beschriebenen raumfüllenden Kurven zum Einsatz.

7.1.1 Matrix-Inversion

Verwendet man das Gauß'sche Eliminationsverfahren, wie es bereits in Abschnitt 3.4.1 genutzt wurde, so lässt sich die spezielle Struktur der Matrix A ausnutzen. Der hier beschriebene Ansatz geht dabei zurück auf [Hac03].

Zunächst wird der erste Schritt betrachtet, d.h. die Elimination der ersten Blockspalte von A. Als Resultat ergeben sich

$$A^1 = \begin{pmatrix} I & 0 & \dots & 0 & A_{00}^{-1}A_{0,\Sigma} \\ 0 & A_{11} & \dots & 0 & A_{1,\Sigma} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & A_{p-1,p-1} & A_{p-1,\Sigma} \\ 0 & A_{\Sigma,1} & \dots & A_{\Sigma,p-1} & A_{\Sigma,\Sigma} - A_{\Sigma,0}A_{00}^{-1}A_{0,\Sigma} \end{pmatrix}$$

und

$$C^1 = \begin{pmatrix} A_{00}^{-1} & 0 & \dots & 0 & 0 \\ 0 & I & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & I & 0 \\ -A_{\Sigma,0}A_{00}^{-1} & 0 & \dots & 0 & I \end{pmatrix}.$$

Die vollständige Elimination der unteren Dreiecksmatrix von A führt schließlich zu

$$A^{p+1} = \begin{pmatrix} I & 0 & \dots & 0 & A_{00}^{-1}A_{0,\Sigma} \\ 0 & I & \dots & 0 & A_{11}^{-1}A_{1,\Sigma} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & I & A_{p-1,p-1}^{-1}A_{p-1,\Sigma} \\ 0 & 0 & \dots & 0 & I \end{pmatrix}$$

und

$$C^{p+1} = \begin{pmatrix} A_{00}^{-1} & 0 & \dots & 0 & 0 \\ 0 & A_{11}^{-1} & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & A_{p-1,p-1}^{-1} & 0 \\ -S^{-1}A_{\Sigma,0}A_{00}^{-1} & -S^{-1}A_{\Sigma,1}A_{11}^{-1} & \dots & -S^{-1}A_{\Sigma,p-1}A_{p-1,p-1}^{-1} & S^{-1} \end{pmatrix},$$

mit dem Schurkomplement

$$S = A_{\Sigma,\Sigma} - \sum_{i=0}^{p-1} A_{\Sigma,i}A_{ii}^{-1}A_{i,\Sigma}. \tag{7.1.5}$$

Die Elimination der oberen Dreiecksmatrix führt anschließend zur gesuchten Lösung

$$C^{2p-1} = A^{-1} = \begin{pmatrix} \ddots & \vdots & \dots & \vdots \\ \vdots & \delta_{ij}A_{ii}^{-1} + A_{ii}^{-1}A_{i,\Sigma}S^{-1}A_{\Sigma,j}A_{jj}^{-1} & \vdots & -A_{ii}^{-1}A_{i,\Sigma}S^{-1} \\ & \vdots & \ddots & \vdots \\ \dots & -S^{-1}A_{\Sigma,j}A_{jj}^{-1} & \dots & S^{-1} \end{pmatrix}. \tag{7.1.6}$$

Die Inverse (7.1.6) besitzt leider nicht mehr die schwache Struktur von A. Insbesondere müssen $(p+1)^2$ viele Matrizen gespeichert werden, falls keine zusätzlichen Berechnungen erwünscht sind. Eine alternative Darstellung von A^{-1}, bei der wie in (7.1.2) ebenfalls lediglich $3p+1$ Matrizen notwendig sind, ergibt sich durch eine additive Aufspaltung in

$$\begin{aligned} A^{-1} &= \begin{pmatrix} A_{00}^{-1} & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & A_{p-1,p-1}^{-1} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \\ &+ \begin{pmatrix} A_{00}^{-1} A_{0,\Sigma} \\ \vdots \\ A_{p-1,p-1}^{-1} A_{p-1,\Sigma} \\ -I \end{pmatrix} \begin{pmatrix} S^{-1} A_{\Sigma,0} A_{00}^{-1} & \cdots & S^{-1} A_{\Sigma,p-1} A_{p-1,p-1}^{-1} & -S \end{pmatrix}. \end{aligned} \tag{7.1.7}$$

Allerdings ist die Form (7.1.7) nur eingeschränkt für die Verwendung von Matrixalgorithmen, etwa der Matrix-Multiplikation, verwendbar. Bei der Matrix-Vektor-Multiplikation ergeben sich dagegen keinerlei Probleme, wie in Abschnitt 7.1.3 gezeigt wird.

Aufbauend auf der Darstellung (7.1.7) erhält man folgenden Algorithmus zur Berechnung der Inversen von A:

```
   procedure dd_invert( i, A )
       { Schritt 1: Inversion im Teilgebiet }
       invert( A_ii, C_ii );
       mul( A_Σ,i, C_ii, 0, T_i );
       mul( T_i, A_i,Σ, 0, S_i );
       { Schritt 2 ... ⌈log2 p⌉ + 1: Summation des Schurkomplements }
7:     summiere S_i in allen Teilgebieten;
       bsp_sync();
       { Schritt ⌈log2 p⌉ + 2: Inversion des Schurkomplements }
10:    if i = 0 then
           invert( ( A_Σ,Σ − Σ_{i=1}^{p} S̃_i ), S̃ );
       { Schritt ⌈log2 p⌉ + 3 ... 2⌈log2 p⌉ + 3: Versenden des Schurkomplements }
13:    bsp_send( S );
       bsp_sync();
       { Schritt 2⌈log2 p⌉ + 4 }
       mul( C_ii, A_i,Σ, 0, C_i,Σ );
       mul( S̃, T_i, 0, C_Σ,i );
       bsp_sync();
   end;
```

Algorithmus 7.1.1: Berechnung der Inversen einer Gebietszerlegungsmatrix

Die in Algorithmus 7.1.1 berechneten Teilmatrizen von C sind durch (7.1.7) wie folgt

definiert:

$$C = \mathrm{diag}(C_{00}, \ldots, C_{p-1,p-1}, 0) + \begin{pmatrix} C_{0,\Sigma} \\ \vdots \\ C_{p-1,\Sigma} \\ -I \end{pmatrix} \begin{pmatrix} C_{\Sigma,0} & \cdots & C_{\Sigma,p-1} & \tilde{S} \end{pmatrix}. \tag{7.1.8}$$

Die Summation des Schurkomplements bzw. das Versenden von $\tilde{S}$ innerhalb des Inversionsalgorithmus erfolgen hierbei durch die in Abschnitt 5.2.2.1 beschriebenen Verfahren 5.2.2 und 5.2.3, wofür jeweils $\lceil \log_2 p \rceil$ Teilschritte erforderlich sind.

Durch die in Kapitel 3 angegebenen Komplexitäten für die Matrix-Multiplikation und -Inversion folgt die nachfolgende Aussage über den Aufwand von Algorithmus 7.1.1.

Lemma 7.1.2 *Die Inversion der Gebietszerlegungsmatrix A aus* (7.1.2) *mittels Algorithmus 7.1.1 hat eine Komplexität von*

$$\begin{aligned} \mathcal{W}_{\mathrm{MI}}(A,p) &= \mathcal{O}\left(\max_{0 \le i < p} \mathcal{W}_{\mathrm{MI}}(A_{ii}) + \log(p) \mathcal{W}_{\mathrm{MA}}(A_{\Sigma,\Sigma}) \right) \\ &+ g \cdot \mathcal{O}\left(\log(p) \mathcal{W}_{\mathrm{St}}(A_{\Sigma,\Sigma}) \right) + l \cdot \mathcal{O}\left(\log p \right). \end{aligned} \tag{7.1.9}$$

Beweis: Aufgrund von (7.1.1) dominiert $\mathcal{W}_{\mathrm{MI}}(A_{ii})$ die Kosten der Berechnung im ersten und letzten BSP-Schritt. Das gleiche gilt für die Berechnung der Inversen des Schurkomplements. Die Summation der lokalen Anteile S_i in $\log p$ Schritten komplettiert den ersten Term. Die Größe des Schurkomplements ist durch $\mathcal{W}_{\mathrm{St}}(A_{\Sigma,\Sigma})$ gegeben. Da jeder Prozessor höchstens 2 Matrizen empfängt bzw. versendet (siehe Algorithmus 5.2.2), folgen die Kommunikationskosten in (7.1.9). □

Für das Beispiel aus Bemerkung 7.1.1 können diese Angaben konkretisiert werden:

Bemerkung 7.1.3 *Es gelten* (7.1.3) *und* (7.1.4) *für die Größe der Teilgebiete bzw. des inneren Randes. Weiterhin seien die verwendeten Clusterbäume binär und kardinalitätsbalanciert. Dann folgt für die Komplexität von Algorithmus 7.1.1:*

$$\begin{aligned} \mathcal{W}_{\mathrm{MI}}(A,p) &= \mathcal{O}\left(\frac{n}{p} \log^2\left(\frac{n}{p}\right) + \log(p) p^{1/d} n^{(d-1)/d} \log\left(p^{1/d} n^{(d-1)/d} \right) \right) \\ &+ g \cdot \mathcal{O}\left(p^{1/d} n^{(d-1)/d} \log\left(p^{1/d} n^{(d-1)/d} \right) \log(p) \right) \\ &+ l \cdot \mathcal{O}\left(\log p \right). \end{aligned} \tag{7.1.10}$$

Ignoriert man die logarithmischen Anteile in (7.1.10) und betrachtet den vereinfachten Term

$$\mathcal{O}\left(\frac{n}{p} + p^{1/d} n^{(d-1)/d} \right) + g \cdot \mathcal{O}\left(p^{1/d} n^{(d-1)/d} \right) + l \cdot \mathcal{O}\left(1 \right),$$

so lässt sich für ein konstantes n ein für Algorithmus 7.1.1 optimales p näherungsweise durch $\mathcal{O}\left(d n^{1/(d+1)} \right)$ beschreiben. Das Verfahren hat in diesem Fall also eine beschränkte parallele

Skalierbarkeit. Andererseits folgert man für ein festes p, dass die Effizienz mit n wächst, da der Term $\frac{n}{p}\log^2\frac{n}{p}$ in (7.1.10) schließlich dominiert. Somit führt ein im Vergleich zur Anzahl der Prozessoren hinreichend großes Problem zu einem effizienten Invertierungsverfahren.

7.1.1.1 Numerische Beispiele

Das Gebiet $\Omega = [0,1]^2$ aus dem Beispiel in Abschnitt 2.4.2 wird für die numerischen Tests in p annähernd gleich große Teilgebiete aufgeteilt, wie es auch in Abbildung 7.1.2 dargestellt ist. Der Kopplungsrand ist hierbei durch die jeweiligen Indizes hervorgehoben.

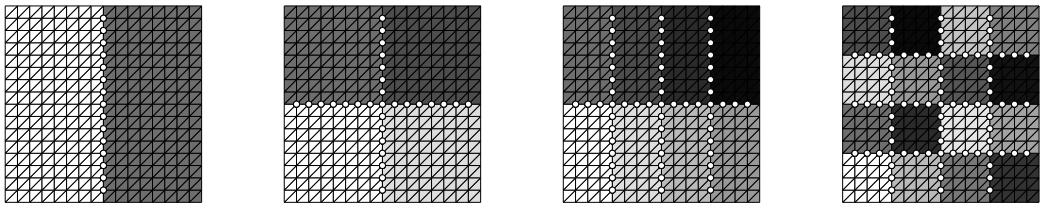


Abbildung 7.1.2: Gebietszerlegung von $\Omega = [0,1]^2$ für 2, 4, 8 und 16 Prozessoren

Aufgrund der Beschränkungen beim Speicherplatz pro Prozessor, wie sie sich bei einem Rechnersystem mit verteiltem Speicher ergeben, können Probleme mit einer großen Anzahl an Unbekannten nur mit mehreren CPUs gerechnet werden. Deshalb fehlen die entsprechenden Laufzeiten für hohe Problemdimensionen. Desweiteren wird in diesen Fällen die modifizierte parallele Effizienz aus Abschnitt 6.1 angegeben, welche sich auf die Laufzeit mit dem kleinsten p bezieht, für welches das Problem gerechnet werden konnte.

Matrix-Inversion, konstanter Rang $k = 10$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
4 069	27.9	3.6	195.0	1.3	270.5	1.4	167.4	2.0	89.5
8 100	101.8	13.2	193.0	4.3	293.1	3.6	236.9	3.5	183.3
16 384	238.2	43.9	135.6	13.7	217.5	10.7	184.8	9.5	156.1
32 761	–	114.0	–	38.5	*148.1*	31.1	*122.0*	21.8	*130.7*
65 536	–	347.0	–	112.9	*153.7*	86.9	*133.1*	61.0	*142.1*
131 044	–	–	–	377.7	–	229.3	*109.8*	144.0	*131.2*

Bei der Interpretation der Ergebnisse sind zwei unterschiedliche Faktoren zu berücksichtigen. Zum einen ist dies der Kommunikationsaufwand, welche insbesondere bei kleinen

Problemen die gesamten Berechnungskosten dominieren kann. Dabei besitzt die Zeit für die globale Synchronisation der BSP-Maschine einen besonders hohen Anteil. Deshalb sinkt die Laufzeit der Inversion etwa bei $n = 4\,096$ nicht unter eine Sekunde.

Zum anderen hat die Komplexität der $\mathcal{H}$-Matrix-Inversion einen wesentlichen Einfluss auf die parallele Effizienz. Durch die logarithmischen Terme in (3.4.1) folgt, dass die Berechnung zweier Probleme der Dimension $n/2$ weniger Aufwand erfordert, als die Behandlung eines Problems der Dimension n. Insbesondere im präasymptotischen Bereich der Inversion ergibt sich somit bei der Gebietszerlegung ein superlinearer Speedup. Je größer die Problemdimension ist, desto kleiner zeigen sich allerdings die Auswirkungen dieses Effektes.

Matrix-Inversion, konstante Genauigkeit $\varepsilon = 10^{-6}$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
4 069	8.8	1.1	192.3	0.6	179.7	0.8	87.0	0.9	63.0
8 100	27.4	3.4	198.8	1.4	237.5	1.5	148.1	1.3	127.6
16 384	78.0	13.4	145.9	3.8	254.5	3.9	168.3	3.2	154.7
32 761	218.7	33.8	161.7	10.6	257.9	10.2	178.2	6.2	219.1
65 536	–	119.0	–	33.3	*178.4*	29.1	*136.3*	16.8	*177.1*
131 044	–	273.0	–	88.2	*154.8*	77.0	*118.2*	39.0	*174.8*
262 144	–	–	–	258.3	–	200.0	*86.1*	108.2	*119.4*

Bei der Matrix-Inversion mit einer konstanten Genauigkeit ergeben sich analoge Ergebnisse wie im Fall eines konstanten Ranges. Erwähnenswert sind hierbei die Werte für 12 Prozessoren im Vergleich zu der Effizienz bei $p = 8$. Hierbei kann die zusätzliche Kommunikation im Vergleich zu 8 Prozessoren nicht durch eine entsprechende Reduktion des Berechnungsaufwandes ausgeglichen werden, wodurch die parallele Effizienz sinkt.

Die Inversion der Gebietszerlegungsmatrix wurde auch für den Fall $d = 3$ durchgeführt. Das zugrundeliegende Gebiet bildet hierbei der Einheitswürfel $[0, 1]^3$, wobei die Triangulation mittels Tetraederelementen erfolgt. Zu beachten ist, dass die Schwachbesetztheitskonstante im dreidimensionalen Fall vergleichsweise hohe Werte annimmt (siehe [GH03]), wodurch der Speicheraufwand für die $\mathcal{H}$-Matrix bei gleicher Problemgröße im Vergleich zu obigem Beispiel stark ansteigt. Hierdurch lassen sich auf dem zur Verfügung stehenden Rechnersystem nur relative kleine Probleme berechnen.

Die folgende Tabelle zeigt die Ergebnisse der entsprechenden Berechnungen für eine Gebietszerlegungsmatrix mit konstantem Rang von $k = 10$.

Matrix-Inversion 3D, konstanter Rang $k = 10$							
	$p = 1$	$p = 4$		$p = 8$		$p = 12$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E
4 069	290.1	10.9	663.0	8.5	424.6	12.0	201.5
8 100	1072.5	65.8	407.3	31.4	426.8	71.4	125.2
16 384	–	258.6	–	170.3	*76.0*	160.2	*53.8*
32 761	–	–	–	775.0	–	701.7	*73.6*

Besonders markant ist der Sprung von der sequentiellen zur parallelen Ausführung. Bedingt ist dies durch das starke Wachstum der Komplexität der $\mathcal{H}$-Matrix-Inversion im $\mathbb{R}^3$ bei kleinen Problemgrößen. Allerdings kann die hierdurch erzielte hohe Effizienz nicht für alle Prozessorzahlen aufrecht erhalten werden. Insbesondere für $p = 12$ ist ein entsprechender Abfall zu beobachten. Der Grund hierfür liegt in der geringen Größe der behandelten Probleme, wodurch kein effizientes Verfahren möglich ist.

Für die gleiche Geometrie wurden die Berechnungen auch bei einer konstanten Genauigkeit durchgeführt. Die hierbei ermittelten Laufzeiten und die zugehörige parallele Effizienz sind in der folgenden Tabelle zusammengefasst.

Matrix-Inversion 3D, konstante Genauigkeit $\varepsilon = 10^{-6}$							
	$p = 1$	$p = 4$		$p = 8$		$p = 12$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E
4 069	144.5	10.2	356.0	11.4	158.6	17.4	69.3
8 100	580.4	41.7	348.0	44.6	162.6	56.6	85.4
16 384	–	167.5	–	108.5	*77.2*	191.3	*29.2*
32 761	–	637.2	–	424.7	*75.0*	413.0	*51.4*
64 000	–	–	–	1426.6	–	1299.4	*73.2*

Das sich abzeichnende Bild entspricht dem bei konstantem Rang beobachtete Verhalten. Auch in diesem Fall führt die geringe Problemgröße nur bei kleinen Prozessorzahlen zu einem effizienten Invertierungsalgorithmus.

7.1.2 LU-Zerlegung

In diesem Abschnitt soll die LU-Faktorisierung der Gebietszerlegungsmatrix A aus (7.1.2) untersucht werden. Die einzelnen Faktoren L und U sind hierbei analog zu Abschnitt 3.5 definiert durch

$$A = \begin{pmatrix} L_{00} & & & \\ \vdots & \ddots & & \\ L_{p-1,0} & \dots & L_{p-1,p-1} & \\ L_{\Sigma,0} & \dots & L_{\Sigma,p-1} & L_{\Sigma,\Sigma} \end{pmatrix} \begin{pmatrix} U_{00} & \dots & U_{0,p-1} & U_{0,\Sigma} \\ & \ddots & \vdots & \vdots \\ & & U_{p-1,p-1} & U_{p-1,\Sigma} \\ & & & A_{\Sigma,\Sigma} \end{pmatrix}, \tag{7.1.11}$$

mit $L_{ij}, U_{ij} \in \mathcal{H}(T(I_i \times I_j))$, $L_{\Sigma,i} \in \mathcal{H}(T(I_\Sigma \times I_i))$ und $U_{i,\Sigma} \in \mathcal{H}(T(I_i \times I_\Sigma))$.

Im folgenden sei angenommen, dass A_{ii} nicht verschwindende LU-Faktoren $L_{ii}U_{ii}$ besitzt. Durch die Gleichungen $L_{00}U_{0i} = 0$ und $L_{i0}U_{00} = 0$ für $0 < i < p$, ergibt sich somit: $U_{0i} = 0$ bzw. $L_{i0} = 0$. Analoges folgert man für L_{ij} und U_{ij} mit $i \neq j$. Die Faktorisierung (7.1.11) lässt sich hierdurch vereinfachen zu

$$A = \begin{pmatrix} L_{00} & & & \\ & \ddots & & \\ & & L_{p-1,p-1} & \\ L_{\Sigma,0} & \dots & L_{\Sigma,p-1} & L_{\Sigma,\Sigma} \end{pmatrix} \begin{pmatrix} U_{00} & & & U_{0,\Sigma} \\ & \ddots & & \vdots \\ & & U_{p-1,p-1} & U_{p-1,\Sigma} \\ & & & U_{\Sigma,\Sigma} \end{pmatrix}. \tag{7.1.12}$$

Die sich aus dieser Zerlegung ergebenden Gleichungen

$$\begin{aligned} L_{ii}U_{ii} &= A_{ii}, \\ L_{ii}U_{i,\Sigma} &= A_{i,\Sigma}, \\ L_{\Sigma,i}U_{ii} &= A_{\Sigma,i} \\ &\text{und} \\ L_{\Sigma,\Sigma}U_{\Sigma,\Sigma} &= A_{\Sigma,\Sigma} - \sum_{i=0}^{p-1} L_{\Sigma,i}U_{i,\Sigma} \end{aligned}$$

führen direkt zum BSP-Algorithmus zur LU-Faktorisierung von A. Hierbei werden die sequentiellen Methoden zur LU-Zerlegung genutzt, wie sie in Abschnitt 3.5 eingeführt wurden.

```
procedure dd_lu( i, A )
    { Schritt 1: Faktorisierung im lokalen Teilgebiet }
    LU_decompose( A_ii );
    solve_U( L_ii, A_i,Σ );
    solve_L( U_ii, A_Σ,i );
    T_i := L_Σ,i U_i,Σ;
    { Schritt 2 ... ⌈log2 p⌉ + 1: Summation von T_i }
    summiere T_i in allen Teilgebieten;
    bsp_sync();
    { Schritt ⌈log2 p⌉ + 2: Faktorisieren von A_Σ,Σ }
    if i = 0 then
        A_Σ,Σ := A_Σ,Σ − Σ_{i=1}^{p} L_Σ,i U_i,Σ;
        LU_decompose( A_Σ,Σ );
    endif;
    { Schritt ⌈log2 p⌉ + 3 ... 2⌈log2 p⌉ + 3: Versenden von L_Σ,Σ und U_Σ,Σ }
    bsp_send( L_Σ,Σ ); bsp_send( U_Σ,Σ );
    bsp_sync();
end;
```

Algorithmus 7.1.2: Berechnung der LU-Zerlegung einer Gebietszerlegungsmatrix

Die Summation der lokalen Produkte $L_{\Sigma,i}U_{i,\Sigma}$ und das Verteilen der Faktoren $L_{\Sigma,\Sigma}$ bzw. $U_{\Sigma,\Sigma}$ erfolgt ebenfalls in $\lceil \log p \rceil$ Schritten. Analog zu Lemma 7.1.2 ergibt sich mit (3.5.1) aus Lemma 3.5.2:

Lemma 7.1.4 *Die Berechnung einer LU-Zerlegung der Gebietszerlegungsmatrix* (7.1.2) *mittels Algorithmus 7.1.2 besitzt eine Komplexität von*

$$\begin{aligned} \mathcal{W}_{\mathrm{LU}}(A,p) &= \mathcal{O}\left(\max_{0\le i<p} \mathcal{W}_{\mathrm{LU}}(A_{ii}) + \log(p)\mathcal{W}_{\mathrm{MA}}(A_{\Sigma,\Sigma})\right) && (7.1.13)\\ &+ g\cdot\mathcal{O}\left(\log(p)\mathcal{W}_{\mathrm{St}}(A_{\Sigma,\Sigma})\right) && (7.1.14)\\ &+ l\cdot\mathcal{O}\left(\log p\right). \end{aligned}$$

Das Ergebnis aus Bemerkung 7.1.3 folgt identisch für die LU-Zerlegung. Somit gilt auch in diesem Fall, dass n im Vergleich zu p hinreichend groß sein muss, um ein effizientes paralleles Verfahren zu erhalten.

7.1.2.1 Numerische Beispiele

Wie im Fall der Matrix-Inversion dient auch bei der LU-Zerlegung das FEM-Beispiel aus Abschnitt 2.4.2 als Grundlage für die numerischen Experimente. Zunächst erfolgen die Berechnungen mit einem konstanten Rang von $k = 10$. Die Laufzeit und die parallele Effizienz für die einzelnen Tests sind in der folgenden Tabelle zusammengefasst.

LU-Zerlegung, konstanter Rang $k = 10$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
4 069	1.4	0.5	68.1	0.6	29.0	0.8	14.5	0.9	10.1
8 100	5.4	1.5	89.0	1.0	69.0	1.3	34.2	1.5	22.5
16 384	13.7	5.5	62.1	2.1	83.7	2.9	39.7	2.9	29.2
32 761	47.4	14.9	79.7	5.1	116.4	5.6	70.3	5.4	55.4
65 536	–	34.6	–	12.6	*137.0*	13.1	*87.9*	11.7	*73.7*
131 044	–	–	–	37.4	–	30.3	*82.4*	22.5	*83.0*
262 144	–	–	–	–	–	92.7	–	53.1	*131.0*

Die niedrige Effizienz der LU-Zerlegung bei kleinen Problemdimensionen ist bedingt durch die große Dominanz der Kommunikationskosten. Dass auch bei größeren Problemen nicht die hohen Effizienzwerte der Matrix-Inversion erreicht werden, liegt dagegen an der Faktorisierung von $A_{\Sigma,\Sigma}$. Während in den Matrizen A_{ii} nur die nichtzulässigen Blöcke Einträge verschieden von Null enthalten, führt die Summation vor der Bestimmung von $L_{\Sigma,\Sigma}$ und $U_{\Sigma,\Sigma}$ auch zu einem Auffüllen der R-Matrizen in $A_{\Sigma,\Sigma}$. Hierdurch ist der Berechnungsaufwand der LU-Zerlegung von $A_{\Sigma,\Sigma}$ trotz der deutlich kleinerer Problemgröße im Vergleich zu A_{ii} von ähnlicher Dauer. Zusammen mit der Kommunikation bei der Summation ergibt sich

ein dominanter Anteil an den Gesamtkosten und somit eine geringe Effizienz. Im Vergleich zur Matrix-Inversion muss der Wert von n mithin deutlich größer gewählt werden, um ein effizientes Verfahren zu erhalten.

Bei der Berechnung mit konstanter Genauigkeit ergab sich ein deutlich reduzierter Speicherbedarf der $\mathcal{H}$-Matrizen. Somit konnten auch größere Problemdimensionen bei den numerischen Beispielen genutzt werden. Für $\varepsilon = 10^{-6}$ lauten die dabei erhaltenen Ergebnisse wie folgt:

LU-Zerlegung, konstante Genauigkeit $\varepsilon = 10^{-6}$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
16 384	5.8	3.0	49.0	1.1	68.2	1.5	32.1	1.6	22.9
32 761	16.8	5.4	77.7	2.3	91.1	2.7	53.0	2.6	41.1
65 536	39.5	20.5	48.2	5.9	83.1	5.4	61.1	4.6	53.9
131 044	112.0	39.0	71.7	13.5	104.1	11.5	81.5	8.7	80.9
262 144	–	120.4	–	35.2	*171.1*	25.6	*156.6*	18.8	*159.9*
524 176	–	246.4	–	81.4	*151.5*	61.6	*133.3*	40.4	*152.5*
1 048 576	–	–	–	235.2	–	141.8	*110.5*	89.3	*131.6*

Das beobachtete Verhalten der parallelen Effizienz stimmt im wesentlichen mit den Ergebnissen im Fall eines konstanten Ranges überein. Allerdings zeigt sich für die größten Probleme eine höhere Effizienz und auch die erwartete superlineare Skalierung. Die Verringerung dieses Superskalierungseffektes folgt hierbei dem sich zunehmend durchsetzendem asymptotischen Verhalten der $\mathcal{H}$-Matrix-Arithmetik.

Neben den zweidimensionalen Problemen erfolgte auch die Untersuchung des Verhaltens der LU-Zerlegung für eine Gebietszerlegungsmatrix bei drei Raumdimensionen. Analog zur Matrix-Inversion ist das zugrundeliegende Gebiet durch den Einheitswürfel $[0, 1]^3$ definiert, welcher mittels einer auf Tetraedern basierenden Triangulation in einzelne Elemente unterteilt ist.

LU-Zerlegung 3D, konstanter Rang $k = 10$							
	$p = 1$	$p = 4$		$p = 8$		$p = 12$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E
4 069	22.5	3.7	150.5	4.1	68.3	7.4	25.3
8 100	88.2	16.4	134.3	12.3	89.9	26.4	27.8
16 384	–	45.7	–	43.4	*52.7*	57.9	*26.3*
32 761	–	–	–	194.1	–	205.7	*62.9*

Wie schon bei der Inversion zeigt sich auch in diesem Fall, dass die Problemgrößen kein effizientes Verfahren zulassen. Lediglich für $p = 4$ zeigt sich eine praktikable Effizienz. In allen anderen Fällen ist die Anzahl von Unbekannten auf dem inneren Randes zu groß, wodurch

der Gesamtaufwand von der Kommunikation und den Berechnungen im Zusammenhang mit $A_{\Sigma,\Sigma}$ dominiert wird.

LU-Zerlegung 3D, konstante Genauigkeit $\varepsilon = 10^{-6}$							
	$p=1$	$p=4$		$p=8$		$p=12$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E
4 069	17.3	3.5	123.4	5.5	39.2	9.3	15.6
8 100	65.0	13.6	119.3	18.4	44.2	28.3	19.2
16 384	232.7	39.9	145.7	40.0	72.7	84.6	22.9
32 761	796.1	147.7	134.8	155.1	64.1	189.7	35.0
64 000	–	483.2	–	443.4	*54.5*	576.8	*27.9*

Analoge Schlussfolgerungen ergeben sich durch die Resultate für eine konstante Genauigkeit. Auch in diesem Fall kann die parallele LU-Zerlegung bei den berechenbaren, kleinen Problemgrößen nicht durch eine entsprechend große Prozessoranzahl profitieren.

7.1.3 Matrix-Vektor-Multiplikation

Zunächst wird die Matrix-Vektor-Multiplikation mit der Matrix A aus (7.1.2) untersucht. Die Vektoren $x \in \mathbb{R}^J$ und $y \in \mathbb{R}^I$ seien hierbei entsprechend der Aufteilung der Indexmenge I in die Vektoren $x_i \in \mathbb{R}^{J_i}$ und $x_\Sigma \in \mathbb{R}^{J_\Sigma}$ bzw. $y_i \in \mathbb{R}^{I_i}$ und $y_\Sigma \in \mathbb{R}^{I_\Sigma}$ zerlegt. Die Teilvektoren x_Σ und y_Σ sind in diesem Fall auf allen Prozessoren vorhanden.

procedure mv_mul_dd ($i, \alpha, A, x, \beta, y$)
  { *Schritt 1* }
  $y_i := \beta y_i$; $y_\Sigma := \beta y_\Sigma$;
  $y_i := y_i + \alpha A_{ii} x_i + \alpha A_{i,\Sigma} x_\Sigma$;
  $y'_{\Sigma,i} := A_{\Sigma,i} x_i$;
  { *Schritt* $2, \ldots, \lceil \log_2 p \rceil + 1$ }
  Summiere $y'_{\Sigma,i}$
  `bsp_sync`();
  { *Schritt* $\lceil \log_2 p \rceil + 2$ }
  if $i = 0$ **then** $y'_\Sigma := \alpha \left(\sum_{i=0}^{p-1} y'_{\Sigma,i} + A_{\Sigma,\Sigma} x_\Sigma \right)$;
  { *Schritt* $\lceil \log_2 p \rceil + 3, \ldots, 2\lceil \log_2 p \rceil + 3$ }
  Verteile y_Σ auf alle Prozessoren;
  `bsp_sync`();
end;

Algorithmus 7.1.3: Matrix-Vektor-Multiplikation einer Gebietszerlegungsmatrix

Die Analyse von Algorithmus 7.1.3 liefert das folgende Resultat für den Aufwand der Matrix-Vektor-Multiplikation:

Lemma 7.1.5 *Die Multiplikation* (3.1.1) *mit der Gebietszerlegungsmatrix* (7.1.2) *durch Algorithmus 7.1.3 besitzt einen Aufwand von*

$$\begin{aligned} \mathcal{W}_{\mathrm{MV}}(A,p) &= \mathcal{O}\left(\max_{0\le i<p} \mathcal{W}_{\mathrm{MV}}(A_{ii}) + \log(p) n_\Sigma\right) \\ &+ g\cdot\mathcal{O}\left(\log(p) n_\Sigma\right) \\ &+ l\cdot\mathcal{O}\left(\log p\right). \end{aligned} \tag{7.1.15}$$

Bemerkung 7.1.6 *Es gelten die Bedingungen aus Bemerkung 7.1.3. Dann folgt:*

$$\begin{aligned} \mathcal{W}_{\mathrm{MV}}(A,p) &= \mathcal{O}\left(\frac{n}{p}\log\left(\frac{n}{p}\right) + \log(p) p^{1/d} n^{(d-1)/d}\right) \\ &+ g\cdot\mathcal{O}\left(\log(p) p^{1/d} n^{(d-1)/d}\right) \\ &+ l\cdot\mathcal{O}\left(\log(p)\right). \end{aligned}$$

Das in Abschnitt 7.1.1 näherungsweise bestimmte, optimale p der Ordnung $dn^{1/(d+1)}$ für ein festes n gilt somit auch bei der Matrix-Vektor Multiplikation. Folglich ergibt sich auch in diesem Fall die Bedingung eines genügend großen n für ein paralleles Verfahren mit einer hinreichend praktikablen Skalierung.

Für die Matrix-Vektor-Multiplikation der zu A inversen Matrix empfiehlt sich die Darstellung (7.1.7) bzw. (7.1.8), da weniger Einzeloperationen auszuführen sind als bei der vollständigen Form (7.1.6). Der Multiplikationsalgorithmus ist in diesem Fall anzupassen.

```
procedure mv_mul_dd_inv ( i, α, C, x, β, y )
    { Schritt 1 }
    y_i := β y_i; y_Σ := β y_Σ;
    y_{Σ,i} := C_ii x_i;
    { Schritt 2, ..., ⌈log p⌉ + 1 }
    summiere y'_{Σ,i};
    { Schritt ⌈log p⌉ + 2 }
    if i = 0 then y'_Σ := α ( S̃ x_Σ + Σ_{i=0}^{p-1} y'_{Σ,i} );
    { Schritt ⌈log p⌉ + 3, ..., 2⌈log p⌉ + 3 }
    versende y'_Σ;
    { Schritt 2⌈log p⌉ + 4 }
    y_i := C_{i,Σ} y'_Σ;
    y_Σ := y_Σ − y'_Σ;
    bsp_sync();
end;
```

Algorithmus 7.1.4: Matrix-Vektor-Multiplikation mit Matrix (7.1.8)

Aus einem Vergleich der beiden Multiplikationsalgorithmen folgt, dass sich die Komplexitätsresultate aus den Lemmata 7.1.5 und 7.1.6 auf die Multiplikation mit der inversen Matrix übertragen lassen.

7.1.4 Numerische Beispiele

Für die in diesem Abschnitt durchgeführten numerischen Tests wird abermals das FEM-Beispiel aus Abschnitt 2.4.2 bemüht. Aufgrund des identischen Aufwands zur Multiplikation der Gebietszerlegungsmatrix (7.1.2) und der entsprechenden Inversion (7.1.7) wird lediglich letztere für die folgenden Berechnungen genutzt. Die jeweiligen Matrizen stimmen hierbei mit den in Abschnitt 7.1.1.1 bestimmten Approximationen überein.

Zunächst erfolgt die Zeitmessung für $\mathcal{H}$-Matrizen mit einem konstanten Rang.

100× Matrix-Vektor-Mult., konstanter Rang $k = 10$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
4 069	2.9	1.0	73.2	1.3	27.0	1.0	23.0	1.6	11.4
8 100	8.1	2.2	90.0	2.5	41.0	2.5	27.2	2.4	21.0
16 384	16.5	5.3	77.4	5.4	38.5	5.6	24.7	4.2	24.9
32 761	–	25.9	–	9.9	*130.1*	10.2	*84.7*	8.0	*80.9*
65 536	–	50.0	–	30.8	*81.0*	27.8	*60.0*	15.7	*79.6*
131 044	–	–	–	59.9	–	50.8	*78.7*	31.0	*96.5*

Bei kleinen Problemgrößen und einer großen Prozessoranzahl dominiert die Kommunikation und der mit dem inneren Rand verbundene Berechnungsaufwand die Matrix-Vektor-Multiplikation, wodurch die Effizienz stark eingeschränkt ist. Mit zunehmender Größe der $\mathcal{H}$-Matrizen ergeben sich allerdings bessere Werte und insgesamt ein effizientes Verfahren. Interessant ist hierbei wieder der Übergang von $p = 8$ auf $p = 12$, bei welchem die Anzahl der Unbekannten auf dem Kopplungsrand stark ansteigt, ohne dass gleichzeitig die zusätzlichen Prozessoren diesen erhöhten sequentiellen Anteil kompensieren könnten.

Auch sind die Superskalierungseffekte nur begrenzt zu beobachten. Der Grund hierfür liegt im asymptotischen Verhalten der Matrix-Vektor-Multiplikation, welches bereits bei relativ kleinen Problemgrößen erreicht wird.

Die gleichen Berechnungen mit $\mathcal{H}$-Matrizen bei einer konstanten Genauigkeit von $\varepsilon = 10^{-6}$ führen zu den folgenden Resultaten:

100× Matrix-Vektor-Mult., konstante Genauigkeit $\varepsilon = 10^{-6}$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
4 069	2.2	0.6	95.7	0.5	51.4	0.8	24.7	1.1	12.7
8 100	7.2	1.1	158.3	0.8	110.1	1.2	50.6	1.5	30.1
16 384	14.0	3.3	106.4	1.8	98.3	3.1	37.8	2.7	31.9
32 761	32.9	7.9	104.1	4.3	96.9	3.9	72.2	5.0	41.2
65 536	–	18.4	–	9.1	*101.5*	9.7	*62.9*	8.3	*55.4*
131 044	–	55.0	–	19.8	*138.8*	16.2	*113.0*	14.4	*95.7*
262 144	–	–	–	55.9	–	66.0	*56.5*	28.3	*98.9*

Die Ergebnisse entsprechenden im wesentlichen denen bei einem konstanten Rang. Auch die Ineffizienz bei 12 Prozessoren ist hier zu beobachten. Dagegen belegen die Effizienzwerte für $p = 16$ die obige Feststellung, dass eine hinreichend große Zahl von Unbekannten zu einem effizienten Verfahren führt.

In der folgenden Tabelle sind die Laufzeiten und entsprechende parallele Effizienz für die Matrix-Vektor-Multiplikation mit den bereits in Abschnitt 7.1.1.1 beschriebenen, auf einem dreidimensionalen Problem basierenden $\mathcal{H}$-Matrizen angeben. Zunächst auch hier der Fall eines konstanten Ranges.

100× Matrix-Vektor-Mult. 3D, konstanter Rang $k = 10$							
	$p = 1$	$p = 4$		$p = 8$		$p = 12$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E
4 069	5.5	7.0	19.7	4.4	15.5	3.6	12.8
8 100	15.8	15.4	25.6	10.4	19.0	13.4	9.0
16 384	–	47.9	–	16.5	*145.5*	22.3	*71.6*
32 761	–	–	–	50.8	–	56.7	*59.8*

Es zeigt sich deutlich, dass die betrachteten Problemgrößen praktisch keinerlei Effizienz des parallelen Verfahren erlauben. Der Aufwand wird hierbei im wesentlichen durch die Kommunikation und den sequentiellen Anteil bestimmt.

100× Matrix-Vektor-Mult. 3D, konst. Genauigkeit $\varepsilon = 10^{-6}$							
	$p = 1$	$p = 4$		$p = 8$		$p = 12$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E
4 069	11.9	3.3	90.9	3.0	50.0	3.9	25.2
8 100	42.3	9.2	115.2	6.5	81.1	8.9	39.8
16 384	–	17.0	–	16.8	*50.8*	15.7	*36.1*
32 761	–	54.0	–	33.9	*79.5*	30.7	*58.7*
64 000	–	–	–	117.9	–	110.7	*71.0*

Deutlich bessere Werte ergeben sich bei $\mathcal{H}$-Matrizen mit einer konstanten Genauigkeit. In diesem Fall ergibt sich durch die veränderte Datenmenge in den beteiligten Matrizen ein reduzierter sequentieller Anteil. Allerdings ist die Problemgröße weiterhin zu klein, um eine Effizienz wie im zweidimensionalen Fall zu erhalten.

7.2 Gebietszerlegung für die Randelementmethode

Die Grundlage für das in diesem Abschnitt betrachtete Problem bildet das Beispiel einer Integralgleichung aus Abschnitt 2.4.1. Wie dort bereits beschrieben, sind die hierbei zu diskretisierenden Operatoren im allgemeinen nicht lokal, d.h. die Systemmatrix ist vollbesetzt. Aus diesem Grund gelingt es auch nicht, A in das schwache Format (7.1.2) zu zerlegen.

Deshalb ist die Definition eines inneren Randes nicht sinnvoll. Die Indexmenge I wird stattdessen disjunkt in die Mengen $I_i \subset I, 0 \leq i < p$ zerlegt, welche den jeweiligen Teilgebieten entsprechen. Über diesen Indexmengen seien die Clusterbäume $T(I_i)$ definiert und aus der Kombination zweier Clusterbäume folgen die Blockclusterbäume $T(I_i \times I_j)$.

Unter der Voraussetzung einer Anordnung der Indizes entsprechend den Teilmengen I_i ergibt sich die nachfolgende Gestalt von A:

$$A = \begin{pmatrix} A_{00} & \dots & A_{0,p-1} \\ \vdots & \ddots & \vdots \\ A_{p-1,0} & \dots & A_{p-1,p-1} \end{pmatrix}, \tag{7.2.1}$$

mit $A_{ij} \in \mathcal{H}(T(I_i \times I_j)$.

Auch in diesem Fall stellt sich die Frage nach effizienten Algorithmen für die übliche Matrixarithmetik. Für die Matrix-Vektor-Multiplikation ergibt sich hierbei ein positives Resultat, welches in Abschnitt 7.2.1 vorgestellt wird. Dagegen lässt sich die Matrix-Inversion aufgrund der vollbesetzten Blockstruktur nicht mittels der im FEM-Fall genutzten Gauß-Elimination effizient parallelisieren.

Das Beispiel aus Bemerkung 7.1.1 lässt sich analog auch auf diesen Fall übertragen, d.h. es wird im folgenden für praktische Berechnungen

$$n_i = \mathcal{O}\left(\frac{n}{p}\right) \tag{7.2.2}$$

angenommen.

Wie bereits eingangs erwähnt, lassen sich die Clusterbäume $T(I_i), 0 \leq i < p$, auch als Knoten eines globalen Clusterbaumes $T(I)$ über I betrachten. Hierbei ergeben sich allerdings für verschiedene Prozessorzahlen im allgemeinen jeweils unterschiedliche Bäume $T^p(I)$ bzw. Blockclusterbäume $T^p = T^p(I \times I)$ über $T^p(I)$, welche aber vergleichbare Eigenschaften besitzen.

Im Unterschied zu einem üblichen Blockclusterbaum treten in T^p zulässigen Knoten allerdings frühestens auf der Stufe $\lfloor \log p \rfloor$ auf. Folglich ist $c_{\text{sp}}(T^p)$ von der Ordnung $\log p$. Die Abhängigkeit der Schwachbesetztheit von p zeigt sich allerdings erst bei vergleichsweise großen Prozessorzahlen, da die Maximalwerte von $c_{\text{sp}}(T^p)$ im üblichen auf relativ hohen Stufen auftreten. Um in diesem Zusammenhang die Unterschiede bei den jeweiligen Blockclusterbäumen T^p zu begrenzen, sei im folgenden angenommen, dass ein $c_{\text{sp}}^{\max} > 0$ existiert, so dass

$$c_{\text{sp}}(T^p) \leq c_{\text{sp}}^{\max} \log p \tag{7.2.3}$$

gilt. Dies ist z.B. durch eine von p unabhängige Wahl des Algorithmus zum Clusterbaumaufbau bzw. der Zulässigkeitsbedingung erreichbar. Aufgrund der schwachen Abhängigkeit von p sind unter dieser Voraussetzung damit häufig auch die Werte von $c_{\text{sp}}(T^p)$ für verschiedene Prozessorzahlen praktisch identisch.

7.2.1 Matrix-Vektor-Multiplikation

Betrachtet wird wie in Abschnitt 7.1.3 die Multiplikation

$$y := \alpha Ax + \beta y \tag{7.2.4}$$

mit $y \in \mathbb{R}^I$, $x \in \mathbb{R}^J$ und $\alpha, \beta \in \mathbb{R}$. Die Vektoren x und y seien wieder in die jeweiligen Anteile $x_i \in \mathbb{R}^{J_i}$ und $y_i \in \mathbb{R}^{I_i}$ pro Teilgebiet zerlegt, wobei x_i bzw. y_i Prozessor i zugewiesen werden.

Verwendet man den gleichen Ansatz für die Matrix-Vektor-Multiplikation wie in Abschnitt 7.1.3, so gelangt man zu einem ineffizienten Verfahren. Der Grund hierfür liegt in der vollbesetzten Blockgestalt (7.2.1) von A. Nimmt man z.B. eine Verteilung der Matrix A an, bei der die Blockzeile i Prozessor i zugewiesen wird (Abbildung 7.2.1 links), so ist der Vektor x_i an alle Prozessoren $j \neq i$ zu versenden. Dies resultiert in Kommunikationskosten der Ordnung n. Wird dagegen eine spaltenorientierte Verteilung vorgenommen (Abbildung 7.2.1 rechts), so produziert jeder Prozessor einen lokalen Ergebnisvektor der Dimension n, welcher einen identischen Kommunikationsaufwand verursacht. Allgemein führt somit jede Form der Verteilung entweder zu einer nichtbalancierten Multiplikation oder einer ineffizienten Kommunikation.

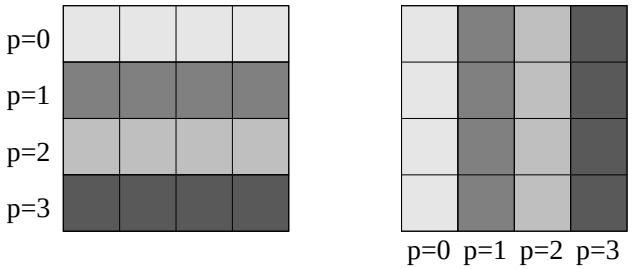


Abbildung 7.2.1: Zeilen- und spaltenorientierte Blockverteilung

Aus diesem Grund wird auf die in Abschnitt 6.3.2 genutzte Technik zurückgegriffen und für alle Nichtdiagonalblöcke $A_{ij}, i \neq j$, die Multiplikation von $\mathsf{R}(k)$-Matrizen $M = AB^T$ aufgetrennt in die Produkte $x' = B^T x|_{\sigma(M)}$ und Ax'.

Die Matrix-Vektor-Multiplikation zerfällt damit wieder in zwei Phasen. Zunächst erfolgt die Multiplikation mit den Diagonalblöcken A_{ii} und die Berechnung des Produktes $y(b) := B^T x|_{\sigma(M)}$ für alle R-Matrizen $M = AB^T \in \mathbb{R}^b$. In diesem Schritt wird die spaltenorientierte Verteilung aus Abbildung 7.2.1 genutzt, d.h. die Matrix A_{ij} wird Prozessor j zugewiesen. Die Ergebnisse $y(b)$ werden anschließend an die Prozessoren versendet, welche das endgültige Produkt Ay' berechnen.

Letzteres erfolgt in der zweiten Phase der Matrix-Vektor-Multiplikation, wobei eine zeilenorientierte Verteilung verwendet wird, d.h. A_{ij} ist mit Prozessor i assoziiert. Das Ergebnis

kann hierbei direkt auf den lokalen Vektor y_i addiert werden, womit keine zusätzliche Kommunikation notwendig ist.

```
procedure ddbem_mv ( i, α, A, x_i, β, y_i )
    { Schritt 1 }
    y_i := β y_i + α A_ii x_i;
    for all 0 ≤ j ≠ i < p do
        for all b ∈ L(T(I_j × I_i)) do
            if M(b) ist R-Matrix then y(b) := B(M(b))^T x_i|_σ(M(b));
            else                      y(b) := M(b) x_i|_σ(M(b));
            bsp_send( j, y(b) );
        endfor;
    bsp_sync();
    { Schritt 2 }
    for all 0 ≤ j ≠ i < p do
        for all b ∈ L(T(I_i × I_j)) do
            if M(b) ist R-Matrix then y_i := y_i + P_{I_i}(A(M(b)) y(b));
            else                      y_i := y_i + P_{I_i}(y(b));
        endfor;
    bsp_sync();
end;
```

Algorithmus 7.2.1: Matrix-Vektor-Multiplikation einer BEM-Gebietszerlegungsmatrix

Durch Summation der jeweils auftretenden Kosten ergibt sich für den allgemeinen Fall die folgenden Aussage über die Komplexität von Algorithmus 7.2.1:

Bemerkung 7.2.1 *Die Matrix-Vektor-Multiplikation* (7.2.4) *mit der Matrix A aus* (7.2.1) *mittels Algorithmus 7.2.1 besitzt eine Komplexität von*

$$\begin{aligned} \mathcal{W}_{\mathrm{MV}}(A,p) &= \mathcal{O}\left(\max_{0\le i<p}\left\{\sum_{j=0}^{p-1}\mathcal{W}_{\mathrm{MV}}(A_{ij}),\sum_{j=0,j\neq i}^{p-1}\mathcal{W}_{\mathrm{MV}}(A_{ji})\right\}\right) \\ &+ g\mathcal{O}\left(\max\{k,n_{\min}\}\max_{0\le i<p}\sum_{j=0,j\neq i}^{p-1}|\mathcal{L}(T_{ji})|\right) \\ &+ 2l. \end{aligned} \tag{7.2.5}$$

Für Abschätzungen der Laufzeit lässt sich das Ergebnis (7.2.5) allerdings nur bedingt verwenden. Hierfür sei der für die Praxis relevante Fall (7.2.2) angenommen. Nach Bemerkung 2.2.8 ist die Zahl von Knoten in $T(I_i)$ somit durch $\mathcal{O}(n/p)$ bestimmt und folglich ergibt sich $\mathcal{O}(c_{\mathrm{sp}}(T(I_i\times I_j))n/p)$ für die Knotenzahl in $T_{ij}, 0\le j<p$. Die letztere Abschätzung ist allerdings zu pessimistisch:

Bemerkung 7.2.2 *Sei $\tau \in T^p(I_i), i < p$. Dann gilt mit* (7.2.3)

$$\sum_{j=0}^{p-1} |\{(\sigma,\tau) \in T^p(I \times I) \,|\, \sigma \in T(I_j)\}| \leq c_{\text{sp}}^{\max} \log p.$$

Somit folgt für die Zahl der Blätter in der Blockspalte $T_{ji}, 0 \leq j < p$:

$$\sum_{j=0}^{p-1} |\mathcal{L}(T_{ji})| = \mathcal{O}\left(\frac{c_{\text{sp}}^{\max} n \log p}{p}\right). \tag{7.2.6}$$

Setzt man (7.2.6) in (7.2.5) ein, so gelangt man zu dem nachfolgenden, konkreteren Ergebnis.

Lemma 7.2.3 *Sei $n_i = \mathcal{O}(n/p)$. Dann folgt für die Komplexität der Matrix-Vektor-Multiplikation mittels Algorithmus 7.2.1*

$$\begin{aligned} \mathcal{W}_{MV}(A,p) &= \mathcal{O}\left(c_{\text{sp}}^{\max} \frac{n}{p} \log \frac{n}{p} \log p\right) \\ &+ g \cdot \mathcal{O}\left(c_{\text{sp}}^{\max} \max\{k, n_{\min}\} \frac{n}{p} \log p\right) \\ &+ 2 \cdot l. \end{aligned} \tag{7.2.7}$$

Beweis: Nach Bemerkung 7.2.2 ist die Zahl der Knoten in T von der Ordnung $c_{\text{sp}}^{\max} n/p$. Analog zum Beweis zu Lemma 3.1.2 bzw. nach Bemerkung 3.1.3 folgt damit ein Aufwand von $\mathcal{O}\left(c_{\text{sp}}^{\max} n/p \log n/p\right)$ für die eigentliche Multiplikation pro Prozessor. Da auch der Kommunikationsaufwand proportional zur Knotenzahl ist, ergibt sich eine Komplexität von $g\mathcal{O}\left(c_{\text{sp}}^{\max} \max\{k, n_{\min}\} \frac{n}{p}\right)$ und somit die Behauptung. □

Sowohl die Berechnung als auch die Kommunikation sind in diesem Fall, bis auf einen logarithmischen Term, optimal, womit ein in der Theorie effizientes paralleles Verfahren für die Matrix-Vektor-Multiplikation gegeben ist. Ob sich dieses Verhalten auch in der Praxis zeigt, sollen numerische Experimente im nächsten Abschnitt zeigen.

7.2.1.1 Numerische Resultate

Anstelle des in Abschnitt 2.4.1 definierten Gebietes für das BEM-Beispiel wird in den folgenden Experimenten auf das schon im FEM-Fall genutzte Einheitsquadrat $[0,1]^2$ zurückgegriffen, da hierbei die Zerlegung in Teilgebiete einfacher ist. Das Integrationsgebiet lässt sich dabei auch als Teilfläche des Einheitswürfels im $\mathbb{R}^3$ betrachten.

Aufgrund der identischen Gebietszerlegung stimmen die Teilgebiete mit denen aus Abbildung 7.1.2 überein. Lediglich der innere Rand entfällt.

Zunächst wurde die Laufzeit der Matrix-Vektor-Multiplikation mit konstantem Rang gemessen. Die entsprechenden Werte sind in der folgenden Tabelle aufgeführt.

100× Matrix-Vektor-Mult., konstanter Rang $k = 10$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
7 938	20.3	6.1	83.7	3.7	68.4	2.6	64.9	2.2	58.6
15 842	46.6	13.1	88.9	7.8	75.0	5.3	73.0	4.7	62.1
32 258	116.3	33.0	88.2	17.8	81.8	11.1	87.4	10.0	72.4
64 082	258.2	71.0	90.9	37.0	87.2	23.3	92.5	22.5	71.8
130 050	–	159.4	–	85.3	*93.4*	50.8	*104.7*	46.4	*85.8*
260 642	–	–	–	–	–	111.3	–	101.6	*82.1*

Die parallele Effizienz von Algorithmus 7.2.1 ist vergleichbar mit der in Abschnitt 6.3.1 ermittelten Leistung der blockorientierten Matrix-Vektor-Multiplikation, wobei die absoluten Werte eine geringfügig höhere parallele Leistung belegen. Allerdings fällt auch in diesem Fall die Effizienz mit wachsender Prozessorzahl, wächst aber mit der Problemdimension.

Im Gegensatz zu Algorithmus 6.3.6 aus Abschnitt 6.3.2 führt die Aufteilung der Blöcke zu einem, auch bei kleinen Problemgrößen effizienten Verfahren. Dies ist darin begründet, dass im Gebietszerlegungsfall keine Blockcluster geteilt werden müssen. Hierdurch steigt der Verwaltungsaufwand nur unwesentlich an.

Im nächsten Experiment wird die $\mathcal{H}$-Matrix-Arithmetik mit konstanter Genauigkeit durchgeführt.

100× Matrix-Vektor-Mult., konstante Genauigkeit $\varepsilon = 10^{-4}$									
	$p = 1$	$p = 4$		$p = 8$		$p = 12$		$p = 16$	
n	t [s]	t [s]	E	t [s]	E	t [s]	E	t [s]	E
7 938	19.9	6.0	83.3	3.7	68.0	2.8	58.7	2.2	55.9
15 842	47.1	13.3	88.6	8.0	73.8	5.8	68.2	4.8	61.0
32 258	114.5	31.2	91.7	17.6	81.1	12.4	76.8	9.9	72.6
64 082	259.5	69.8	93.0	37.6	86.3	25.6	84.3	22.1	73.4
130 050	–	155.8	–	84.3	*92.5*	56.4	*92.1*	45.7	*85.2*
260 642	–	–	–	–	–	122.7	–	99.6	*92.4*

Das Verhalten der Matrix-Vektor-Multiplikation ist dem im Fall eines konstanten Ranges sehr ähnlich. Auch hierbei steigt die Effizienz mit wachsender Problemdimension.

Wie die numerischen Experimente zeigen, stellt der Gebietszerlegungsansatz auch im BEM-Fall eine Alternative zur direkten Parallelisierung der $\mathcal{H}$-Matrix-Arithmetik dar. Im Unterschied zu letzterer ergibt sich eine optimale Abhängigkeit von der Prozessorzahl und eine hohe Effizienz bereits bei kleinen Problemgrößen.

8 Dynamische Speicherverwaltung

Bei anfänglichen numerischen Tests der $\mathcal{H}$-Matrix-Arithmetik ergaben sich zunächst Laufzeiten, die nicht mit den theoretischen Vorhersagen vereinbar waren. Insbesondere bei großen Problemgrößen traten fast quadratische Komplexitäten zutage. Nach eingehenden Untersuchungen und dem Ausschluss von Fehlern bei der Implementierung blieb eine Ursache für dieses Verhalten übrig: die Speicherverwaltung.

Bei den meisten heutigen Programmiersprachen erfolgt die Verwaltung des für Daten genutzten Speichers dynamisch, d.h. es besteht die Möglichkeit während der Ausführung Speicher zu allozieren und wieder freizugegeben. Beim Algorithmendesign werden diese Aspekte in der Regel nicht beachtet und konstante Laufzeiten für die Allokation bzw. Freigabe von Speicher angenommen. In der Praxis zeigt sich dagegen ein anderes Bild. Viele gängige Speicherverwaltungen sind z.B. nicht für die Verwendung von sehr großen Datenmengen konzipiert oder nicht in der Lage, gleichzeitig Anfragen von mehreren Prozessoren zu beantworten. Diese Aspekte sind aber insbesondere bei $\mathcal{H}$-Matrizen wesentlich für eine Übertragung der theoretischen Vorhersagen in praktische Resultate.

Da auch alternative, freie Speicherverwaltungen wie *PTmalloc* (siehe Abschnitt 8.3.1) oder *Hoard* (siehe Abschnitt 8.3.3) nicht zu einer Besserung des eingangs geschilderten Verhaltens führten, wurden die Anforderungen, wie sie die $\mathcal{H}$-Matrix-Arithmetik stellte, als Ausgangspunkt für eine Neuimplementierung genutzt. Diese Anforderungen werden im einzelnen in Abschnitt 8.1 diskutiert.

Grundlegende Techniken, die die Basis einer Speicherverwaltung bilden, sind Gegenstand von Abschnitt 8.2. Bestehende Implementierungen werden anschließend in Abschnitt 8.3 vorgestellt, bevor in Abschnitt 8.4 die Beschreibung der bei den $\mathcal{H}$-Matrizen in dieser Arbeit zum Einsatz kommenden Verwaltung folgt. Numerische Tests in Abschnitt 8.5 sollen die Entscheidungen beim Design rechtfertigen.

8.1 Anforderungen

Die folgende Aufzählung gibt die gewünschten Eigenschaften einer, nicht nur für die $\mathcal{H}$-Arithmetik idealen Speicherverwaltung wieder:

- $\mathcal{O}(1)$-Komplexität für die Allokation und Freigabe von Speicher,
- maximale parallele Effizienz und
- minimaler Speicherverbrauch.

Allerdings sind diese Eigenschaften in der Praxis nur schwer zu erreichen. Auch schließen sich einige der Anforderungen gegenseitig aus. So lässt sich mit einigen der in Abschnitt 8.2 beschriebenen Techniken z.B. die Antwortzeit bei der Allokation auf das gewünschte Niveau senken. Dies ist dann aber mit einem deutlich höheren Speicherverbrauch verbunden und umgekehrt.

Durch den gemeinsamen Adressraum beim PRAM-Modell (siehe Abschnitt 5.2.1) und insbesondere beim Einsatz von Threads müssen zusätzliche Maßnahmen getroffen werden, um *Sicherheit* und *Skalierbarkeit* zu garantieren.

Dabei bedeutet Sicherheit, dass alle Threads gleichzeitig Speicher allozieren und deallozieren dürfen, ohne dass es zu undefinierten Zuständen des Programms kommt, etwa durch das Überschreiben von Speicherblöcken. Eine mögliche Lösung hierfür ist z.B. die Verwendung eines Mutices, um kritische Bereiche zu sichern.

Die Skalierbarkeit wiederum verlangt ein nichtblockierendes, gleichzeitiges Aufrufen der dynamischen Speicherverwaltung in allen Threads. Ein Mutex-basiertes System ist in diesem Fall aber nur dann effizient, wenn die Speicheranfragen relativ selten erfolgen und sich die Threads dementsprechend nicht blockieren.

8.1.1 False-Sharing

Ein weiteres Problem, dessen Ursache im gemeinsamem Adressraum von Programmen mit mehreren Threads liegt und zu einer geringen parallelen Effizienz führen kann, ist *False-Sharing*. Dabei handelt es sich um ein Phänomen, welches in modernen Computerarchitekturen mit mehreren Prozessoren und Speichercaches auftritt. Das Zwischenspeichern von Daten aus dem Hauptspeicher erfolgt in der Praxis nicht bezüglich eines einzelnen Bytes, sondern durch die Bildung größerer Gruppen, sogenannter *Cache-Zeilen*. Diese sind bei aktuellen Prozessoren üblicherweise 64 Byte lang.

Um den Inhalt der einzelnen Zwischenspeicher pro Prozessor konsistent zu halten, werden spezielle Protokolle benutzt, die Zustandsänderungen einer Cache-Zeile mit entsprechenden Aktionen verknüpfen. Typischweise treten dabei die folgenden Zustände auf:

- *Verändert*: die Cache-Zeile befindet sich nur im lokalen Zwischenspeicher und wurde geändert, d.h. das geänderte Datum ist noch nicht in den Hauptspeicher zurückgeschrieben worden,

- *Exklusiv*: die Cache-Zeile ist nur im lokalen Zwischenspeicher enthalten und identisch mit der Kopie im Hauptspeicher,

- *Geteilt*: die Cache-Zeile ist im Zwischenspeicher des lokalen und mindestens eines anderen Prozessors vorhanden, wobei das enthaltene Datum mit dem im Hauptspeicher übereinstimmt und

- *Ungültig*: die Cache-Zeile ist nicht im lokalen Zwischenspeicher vorhanden oder das dort befindliche Datum ungültig.

Die Anfangsbuchstaben der englischen Begriffe dieser Zustände, „Modified“, „Exklusive“, „Shared“ und „Invalid“, geben dem hierauf basierenden Protokoll seinen Namen: *MESI* (siehe [PP84]).

Das Problem besteht nun darin, dass das Schreiben eines Datums einer geteilten Cache-Zeile diese in allen Zwischenspeichern ungültig macht. Die korrespondierenden Prozessoren müssen somit die gesamte Zeile erneut aus dem Arbeitsspeicher einlesen, selbst wenn kein lokal benötigtes Datum geändert wurde (siehe Abbildung 8.1.1). Dieser unnötige Speicherzugriff führt unter Umständen zu einer Laufzeit des parallelen Programms, welche größer ist als die der sequentiellen Variante.

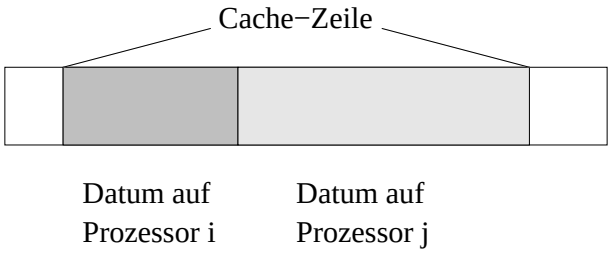


Abbildung 8.1.1: False-Sharing: Daten auf unterschiedlichen Prozessoren benutzen die gleiche Cache-Zeile

Ein durch die Speicherverwaltung induziertes False-Sharing ergibt sich z.B. dadurch, dass alle Speicheranfragen der unterschiedlichen Threads aus einem gemeinsamen Speicherbereich bedient werden. Dieses Verhalten wird *aktives* False-Sharing genannt.

Demgegenüber verursacht eine Speicherverwaltung *passives* False-Sharing, falls freigegebene Speicherbereiche eines Threads für Speicheranfragen eines anderen Threads genutzt werden. Die mit diesem Block verbundene Cache-Zeile wird aber möglicherweise auch von anderen Daten des ersten Threads verwendet.

Beide Fälle von False-Sharing sollten durch die Speicherverwaltung möglichst vermieden werden.

8.1.2 Speicherverbrauch und Fragmentierung

Allgemein wird durch Fragmentierung die Unfähigkeit bezeichnet, freien Speicher innerhalb der Speicherverwaltung wiederzuverwenden (siehe auch [WJNB95]). Abhängig von der jeweiligen Situation unterscheidet man hierbei zwischen

- *externer* Fragmentierung: alle freien Blöcke der Speicherverwaltung sind zu klein für eine Speicheranfrage und

- *interner* Fragmentierung: bei einer Speicheranfrage wird mehr Speicher zurückgegeben, als hierfür notwendig wäre.

Um die Fragmentierung in der Praxis zu untersuchen, gibt es verschiedene Möglichkeiten der Messung (siehe etwa [JW99]). In dieser Arbeit werden externe und interne Fragmentierung zusammengefasst und der gesamte Speicherverbrauch ermittelt.

Definition 8.1.1 (Fragmentierungsgrad) *Sei M_a die maximale Menge von Speicher, die vom Speichermanagement benutzt wird und M_u die maximale Menge Speicher, die das Programm tatsächlich benötigt. Dann bezeichne*

$$F = \frac{M_a}{M_u}$$

den Grad der Fragmentierung.

Um die Speicherkomplexität von Algorithmen innerhalb eines Programms nicht zu verändern, ist ein konstanter Fragmentierungsgrad auch hinsichtlich der Gesamtzahl der Prozessoren wichtig.

Bemerkung 8.1.2 *Der Speicherverbrauch eines optimalen, sequentiellen Algorithmus zur Speicherverwaltung liegt in $\mathcal{O}(M_u \log M_b)$, wobei M_b die Größe des größten angeforderten Speicherblockes ist (siehe [Rob74]).*

8.2 Verschiedene Techniken der Speicherverwaltung

In diesem Abschnitt soll ein kurzer Überblick über typische Techniken bei der Speicherverwaltung gegeben werden. Besondere interessant sind in diesem Zusammenhang die Eigenschaften hinsichtlich der oben genannten Anforderungen für die $\mathcal{H}$-Matrizen bzw. deren Arithmetik.

Neben den hier angegebenen Verfahren existieren weitere Algorithmen zur dynamischen Speicherverwaltung, z.B. *Buddy-Systems* oder *Indexed-Fits* . Eine ausführliche Beschreibung dieser und anderer Methoden ist z.B. in [WJNB95] zu finden.

8.2.1 Sequential Fits

Eine einfache Form des Speichermanagements kann durch eine Liste freier Blöcke realisiert werden. Die freien Blöcke entsprechen dabei den vom Programm freigegebenen Speicherbereichen. Für eine Allokation eines neuen Speicherblocks wird zunächst nach einem passenden Block in der Liste gesucht und bei einem negativen Suchresultat vom Betriebssystem Speicher angefordert.

Die Art und Weise wie die Suche in der Liste erfolgt, hat einen entscheidenden Einfluss auf die Eigenschaften der Speicherverwaltung. Man unterscheidet hierbei typischerweise 3 Typen:

1. *First-Fit*: Der erste freie Block, der groß genug ist, die Speicheranfrage zu beantworten, wird benutzt.

2. *Best-Fit*: Der kleinste Block in der Liste, der groß genug ist, die Speicheranfrage zu beantworten, wird genutzt.

3. *Next-Fit*: Dieses Verfahren arbeitet wie „First Fit“, beginnt die Suche aber stets nach dem letzten gefundenen Block.

Die First-Fit- und Next-Fit-Strategie ist typischerweise schneller als Best-Fit, da im allgemeinen nicht die gesamte Liste durchsucht werden muss. Außerdem ist der Speicherverbrauch und somit der Fragmentierungsgrad bei Best-Fit theoretisch am größten (vergleiche [Rob77]), obwohl dieses Verhalten in der Praxis selten zu beobachten ist.

Weitere Techniken beeinflussen die jeweilige Strategie. Zum Beispiel kann ein durch Teilung entstehender Restblock an den Anfang der Liste gestellt werden oder an der Position des alten Blockes verbleiben. Im ersten Fall besteht bei einer *First-Fit*- oder *Best-Fit*-Suche die Möglichkeit zur Erhöhung der Datenlokalität, falls weitere Anfragen ebenfalls diesen Block nutzen können.

Verbunden hiermit ist die Reihenfolge, in der neue, freie Blöcke in die Liste eingefügt werden. Denkbar ist das Platzieren am Anfang bzw. Ende der List oder eine Sortierung entsprechend der Größe bzw. der Speicheradresse der freien Blöcke. Für die *Best-Fit*-Suche ist es hierbei günstig, eine Sortierung nach der Blockgröße vorzunehmen, da in diesem Fall die Suche nach einem passenden Speicherbereich nur bis zum ersten, hinreichend großen Block erfolgen muss.

Entscheidend für den Fragmentierungsgrad der Speicherverwaltung ist der Umgang mit freien aber nichtpassenden Speicherblöcken. Liefert man einen zu großen Block als Antwort auf eine Anfrage zurück, erhöht sich mithin die interne Fragmentierung. Um dem entgegenzuwirken, kann der Block geteilt und der nicht benötigte Bereich wieder in die Liste eingefügt werden. Bei freien, aneinandergrenzenden Bereichen des Speichers besteht umgekehrt die Möglichkeit, diese zu einem größeren Block zusammenzufügen, um die externe Fragmentierung zu verringern.

Für alle drei Techniken ergibt sich in jedem Fall eine Komplexität, abhängig von der Anzahl der freien Speicherblöcke. Mittels Suchbäumen gelingt es zwar, diese auf eine logarithmische Abhängigkeit zu reduzieren, eine konstante Ausführungsgeschwindigkeit lässt sich allerdings nicht realisieren. Somit sind diese Techniken nur bedingt für sehr große Speichermengen und vergleichsweise kleine Blockgrößen nutzbar.

Weiterhin erlauben sie keine parallele Ausführung, womit auch die Effizienz bei gleichzeitiger Allokation in verschiedenen Prozessoren nicht gegeben ist.

Ein Vorteil der erwähnten Techniken ist dagegen die vergleichsweise einfache Implementierung. Für eine kleine und möglicherweise begrenzte Anzahl von Blöcken ist die Speicherverwaltung somit sehr schnell zu realisieren.

8.2.2 Segregated Free Lists

Bei den „Sequential Fits“ wurden alle Speicherblöcke in einer einzigen Liste gespeichert. Hierdurch war die Zahl der bei einer Allokation untersuchten Speicherblöcke sehr groß. Um die Geschwindigkeit zu erhöhen, kann die Anzahl der Listen erhöht werden, wobei jede Liste für unterschiedliche Blockgrößen genutzt wird. Diese Technik der Speicherverwaltung nennt man *Segregated Free Lists*.

Aus Effizienzgründen werden Klassen $S_0 < S_1 < \ldots < S_m \in \mathbb{N}$ für die verschiedenen Blockgrößen eingeführt. Eine Speichergröße $s \in \mathbb{N}$ gehört zur Klasse S_i, falls

$$S_{i-1} < s \leq S_i.$$

Typischerweise wird für die Größenklassen eine exponentielle Folge verwendet: $S_i = S_0 b^i$ mit $b > 1$. Die Speicherung der Listen erfolgt in einem Array, wobei sich der Index aus der Größenklasse ergibt. Zu einer gegebenen Größe s lässt sich die zugehörige Größenklasse durch eine Binärsuche ermitteln, womit der Zugriff proportional zu $\log m$ ist.

Bemerkung 8.2.1 *Sei s die Speichergröße eines angeforderten Speicherbereichs und s' die Größe des von der Speicherverwaltung hierfür benutzten, nicht geteilten Blockes. Dann gilt: $s' < bs$. Man spricht in diesem Fall auch von* Good-Fit. *Die interne Fragmentierung ist entsprechend durch b begrenzt.*

Die letzte Bemerkung erlaubt eine einfache Optimierungsmaßnahme, um die Zeit für die Suche nach einem passenden Block innerhalb der Größenklasse zu reduzieren. Rundet man alle Speicheranfragen einer Klasse S_i auf den Wert S_i auf, so beträgt der Aufwand für die Suche $\mathcal{O}(1)$. Da in diesem Fall die interne Fragmentierung durch b beschränkt ist, sollte dessen Wert relativ klein gewählt werden.

Durch die Klassifizierung der Speichergrößen ist die Skalierung dieser Verwaltungstechnik gegenüber dem verwendeten Speicherplatz deutlich besser, als bei einer einzelnen Liste. Auch die parallele Effizient wird gesteigert, falls konkurrierende Anfragen verschiedener Prozessoren verschiedene Größenklassen betreffen. In diesem Fall können sämtliche Zugriffe gleichzeitig erfolgen, ohne die Sicherheit der Daten zu kompromittieren. Lediglich beim parallelen Zugriff auf die gleiche Größenklasse kommt es zu einem Blockieren einzelner Prozessoren.

8.2.2.1 Simple Segregated Storage

In der einfachsten Version einer Speicherverwaltung, die mit „Segregated Free Lists“ arbeitet, wird auf das Teilen und Zusammenfügen von benachbarten Speicherblöcken völlig verzichtet. Die Beantwortung von Speicheranfragen erfolgt entweder aus der Liste der korrespondierenden Größenklasse, oder es wird ein neuer Block vom Betriebssystem angefordert.

Wieder freigegebene Blöcke fügt man anschließend in die Listen ein, aus welcher sie bei einer erneuten Speicheranfrage ohne zusätzlichen Aufwand wieder entnommen werden können.

Treten in der Applikation nur Gruppen von Speicherblöcken gleicher Größe auf, so bietet sich hiermit eine schnelle und einfache Speicherverwaltung an. Variieren die Größen leicht, so kann das oben beschriebene Aufrunden der Speichergrößen innerhalb einer Klasse genutzt werden.

Der Nachteil dieser Art des Speichermanagements ist eine nicht beschränkte externe Fragmentierung. Als Beispiel sei hier ein Programm genannt, welches n Blöcke einer Größenklasse S_i anfordert, anschließend freigibt und nie wieder benutzt. Die externe Fragmentierung ist in diesem Fall durch $\mathcal{O}(nS_i)$ bestimmt.

8.2.2.2 Segregated Fits

Der wesentliche Unterschied dieses Verfahrens zum „Simple Segregated Storage“ besteht darin, dass Speicherblöcke geteilt und wieder zusammengefügt werden. Bei einer Speicheranfrage s erfolgt dabei zunächst eine Suche nach einem passenden Block in der Liste der entsprechenden Größenklasse S_i. Findet sich dort kein Block der angeforderten Größe, durchsucht man anschließend sukzessive alle Größenklassen S_j, $j > i$. Die damit verbundene Suche hat einen Aufwand von $\mathcal{O}(m)$.

Beim Antreffen eines zu großen Speicherbereiches, kann dieser geteilt und der Restblock in die entsprechende Liste einsortiert werden. Andernfalls fordert man einen neuen Speicherblock vom Betriebssystem an. Bei der Speicherfreigabe besteht dann die Möglichkeit, den geteilten Block wieder zusammenzufügen, sofern der Restblock nicht bereits verwendet wird.

Insbesondere für den Fall, dass viele Speicherblöcke der gleichen Größe angefordert und freigegeben werden, lässt sich diese Strategie optimieren, indem man das Zusammenfügen der freien Blöcke hinauszögert. Die freien Blöcke werden stattdessen in einer speziellen Liste zwischengespeichert, um bei einer erneuten Anfrage schneller auf sie zugreifen zu können. Hierdurch lassen sich die Kosten für das Teilen und Zusammenfügen von Speicherbereichen einsparen. Um die externe Fragmentierung zu begrenzen, werden diese zwischengespeicherten Blöcke an die normale Verwaltung übergeben, falls die summierte Speichergröße in dieser speziellen Liste eine gewisse Maximalgröße überschreitet.

8.2.2.3 Container

Eine Technik, bei der „Simple Segregated Storage“ und „Segregated Fits“ kombiniert werden, ist die Verwendung von *Speichercontainern*. Unter einem Container versteht man dabei einen Speicherbereich fester Größe $n_c > 0$, der in einzelne, gleich große Speicherblöcke aufgeteilt wird (siehe Abbildung 8.2.1). Ein Container in der Größenklasse S_i bietet damit n_c/S_i Blöcken Platz.

An die Stelle der Liste von Blöcken tritt eine Liste von Containern. Die Verwaltung der Speicherblöcke wird desweiteren an die jeweiligen Container übertragen. Gibt man einen Speicherblock frei, so genügt hierzu lediglich eine entsprechende Markierung innerhalb des Containers. Hierdurch lässt sich eine sehr schnelle Form des Teilens und Zusammenfügens

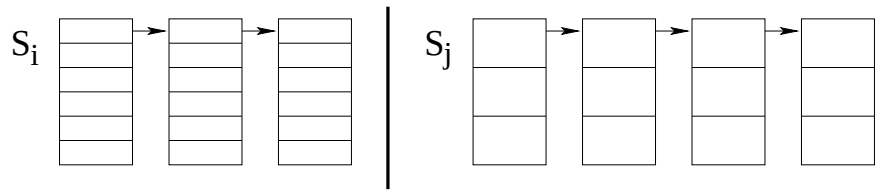


Abbildung 8.2.1: Verwendung von Containern

realisieren. Außerdem ermöglicht die identische Größe aller Blöcke innerhalb eines Containers einen Zugriff auf einen freien Block in konstanter Zeit.

Ist ein Container unbenutzt, d.h. alle Speicherblöcke in diesem Container frei, so kann er aufgrund der festen Größe von einer beliebigen Größenklasse wiederverwendet werden, wodurch man die externe Fragmentierung reduziert.

Ein weiterer Vorteil der Verwendung von Containern ist eine größere parallele Effizienz. Konkurrierende Speicheranfragen, auch in der gleichen Größenklasse, lassen sich simultan beantworten, sofern verschiedene Container betroffen sind.

Leider kann man diese Technik nicht für beliebige Speichergrößen verwenden, da die externe Fragmentierung bei zu hohen Containergrößen in absoluten Werten zu groß wird. Typischerweise haben die Container eine Größe von wenigen Kilobyte. Somit ist diese Technik nur eingeschränkt für den Einsatz bei $\mathcal{H}$-Matrizen geeignet, bei denen mehrere Gigabyte verwaltet werden müssen.

8.2.3 Multiple Speicherverwaltungen

Bei allen bisherigen Techniken wurde stets nur eine Speicherverwaltung für alle Prozessoren verwendet. Obwohl einige Formen dabei einen gleichzeitigen Zugriff ermöglichen, ist keine der aufgeführten Techniken für einen vollkommen parallelen Einsatz geeignet. Insbesondere verhindert keine der genannten Speicherverwaltungen False-Sharing.

Der optimale Parallelbetrieb lässt sich realisieren, indem man für jeden Prozessor eine eigene, *private* Speicherverwaltung, auch *Heap* genannt, verwendet. Alle Speicheranfragen auf einem Prozessor i werden in diesem Fall stets von der assoziierten Verwaltung $\mathcal{V}_i$ beantwortet. Analog erfolgt die Freigabe von Speicherblöcken in $\mathcal{V}_i$. Dies erlaubt eine gleichzeitige Speicherallokation bzw. -deallokation auf allen Prozessoren.

Dieser Ansatz führt allerdings zu einem unbeschränkten Fragmentierungsgrad. In einem entsprechenden Beispielprogramm wird auf einem Prozessor i ein Datenblock alloziert und an Prozessor j übergeben, welcher den Block freigibt. Bei einer anschließenden Wiederholung des Vorgangs ist es stets notwendig, neuen Speicher vom Betriebssystem anzufordern, da bei der Speicheranfrage auf Prozessor i der freie Datenblock nicht mehr zur Verfügung steht.

Weiterhin lässt sich kein False-Sharing verhindern. Insbesondere kann passives False-

Sharing (siehe Abschnitt 8.1.1) induziert werden, da die auf einem Prozessor allozierten Speicherblöcke während der Programmausführung auf andere Speicherverwaltungen migrieren können.

Um dies zu vermeiden, ist es notwendig, alle Speicherblöcke b fest mit der Speicherverwaltung $V(b)$ zu verknüpfen, in denen sie alloziert wurden. Unabhängig vom Prozessor, auf dem die Freigabe von b erfolgt, übergibt man den Block somit stets an $V(b)$. Hierdurch lässt sich aber nicht verhindern, dass innerhalb des Programms ein Austausch von Speicherbereichen zwischen verschiedenen Prozessoren stattfindet.

Durch die feste Kopplung wird der unbeschränkte Fragmentierungsgrad unterbunden. Dieser ist allerdings nicht identisch mit einer sequentiellen Speicherverwaltung, sondern abhängig von p (siehe [BL94]). Betrachtet sei hierfür ein Programm, in dem zu verschiedenen Zeitpunkten auf den einzelnen Prozessoren ein Block der Größe s alloziert und anschließend wieder freigegeben wird. Bei einer sequentiellen Ausführung genügt lediglich ein Datenblock, wohingegen man $p \cdot s$ Speicher bei multiplen Speicherverwaltungen benötigt.

Da jeder Block b in der assoziierten Speicherverwaltung V_i freigegeben wird, besteht außerdem die Möglichkeit, dass Prozessoren blockieren, falls sie gleichzeitig auf V_i zugreifen, z.B. Datenblöcke freigeben. Die parallele Effizienz ist in diesem Fall nicht optimal. Durch die im letzten Abschnitt beschriebenen Techniken lässt sich die Wahrscheinlichkeit eines Blockierens aber reduzieren, da man nicht V_i insgesamt vor einem gleichzeitigen Zugriff schützen muss, sondern lediglich die einzelnen Listen bzw. Container.

8.3 Existierende Speicherverwaltungen

In diesem Abschnitt sollen einige Vertreter von parallelen Speicherverwaltungen vorgestellt werden. Hierbei handelt es sich in erster Linie um bekannte Implementierungen, die auch in der Praxis weit verbreitet sind.

8.3.1 PTmalloc

PTmalloc (siehe [Glo]) ist eine Modifikation der sequentiellen Speicherverwaltung *DLmalloc*. DLmalloc nutzt die Technik der „Segregated Fits“ (siehe Abschnitt 8.2.2.2). Für die 128 Größenklassen ergibt sich hierbei die folgende Einteilung:

$$S_i := \begin{cases} S_{i-1} + 8, & i \leq 64 \\ S_{i-1} + 64, & 64 < i \leq 96 \\ S_{i-1} + 512, & 96 < i \leq 112 \\ S_{i-1} + 4096, & 112 < i \leq 120 \\ S_{i-1} + 32768, & 120 < i \leq 124 \\ S_{i-1} + 262144, & 124 < i \leq 126 \end{cases},$$

mit $S_0 = 0$. Die übrigen Größen sind der Klasse S_{127} zugeordnet. Freie Blöcke in den Listen einer Größenklasse werden entsprechend der Größe sortiert. Verbunden mit einer „Best-Fit"-Strategie erfolgt somit die Suche pro Liste in optimaler Zeit.

Das Zusammenfügen von benachbarten freien Speicherblöcken wird nicht sofort bei der Freigabe vorgenommen, sondern hinausgezögert (siehe Abschnitt 8.2.2.2) und die freien Blöcke zwischengespeichert.

Um eine hinreichende parallele Effizienz zu gewährleisten, verwendet PTmalloc mehrere Speicherverwaltungen. Hierbei erfolgt aber keine feste Verknüpfung zwischen einem Prozessor und einem Heap. Bei der ersten Speicheranfrage auf Prozessor i sucht PTmalloc stattdessen innerhalb einer Liste nach einer *zur Zeit* unbenutzten Speicherverwaltung. Steht eine solche nicht zur Verfügung, wird ein neuer Heap angelegt. Diesen freien bzw. neuen Heap V und den entsprechenden Prozessor verknüpft PTmalloc anschließend. Bei einer nachfolgenden Speicheranfrage wird zunächst V auf eine mögliche aktuelle Benutzung hin geprüft. Ist V frei, kann die Anfrage aus V beantwortet werden. Anderenfalls erfolgt erneut eine Suche nach einem unbenutzten Heap bzw. es wird eine neue Verwaltung angelegt. Auf diese Art und Weise ist die Anzahl der Heaps gleich dem maximalen Grad der Parallelität während der Allokation von Speicher innerhalb des Programms. Hierdurch reduziert sich auch der Fragmentierungsgrad.

Da allerdings bei diesem Verfahren die Möglichkeit besteht, dass mehrere Prozessoren die gleiche Speicherverwaltung nutzen, kann aktives False-Sharing (siehe 8.1.1) in PTmalloc nicht ausgeschlossen werden.

Eine weitere Eigenschaft von PTmalloc betrifft die Allokation von Speicher vom Betriebssystem. Um die Anzahl der vergleichsweise teuren Systemaufrufe zu reduzieren, wird neben dem benötigten Speicherplatz eine bestimmte Menge zusätzlich angefordert. Dieser Zusatzspeicher dient dazu, zukünftige Anfragen ohne einen Systemaufruf beantworten zu können. Analog wird freier Speicher nur dann an das Betriebssystem zurückgegeben, falls eine definierte Menge an freien Blöcken in der Speicherverwaltung zur Verfügung steht.

8.3.2 LKmalloc

LKmalloc (siehe [LK99]) arbeitet ähnlich zu PTmalloc. Es basiert ebenfalls auf der „Segregated Fits"-Technik und verwendet die gleichen Größenklassen. Anders als PTmalloc, werden die Listen pro Klasse aber nicht sortiert, sondern Blöcke jeweils an das Ende der Liste angefügt. Weiterhin erfolgt eine sofortige Vereinigung von freigegebenen Speicherbereichen mit angrenzenden, freien Blöcken.

Für eine gute parallele Effizienz sorgen auch in diesem Fall mehrere Heaps, die den Prozessoren zugewiesen werden. Beim Start des Programms legt der Benutzer hierfür die Anzahl der Speicherverwaltungen fest. Die Zuordnung von Heaps zu den einzelnen Prozessoren erfolgt anschließend über eine Hashfunktion, d.h. eine Abbildung, die die Prozessoren möglichst gleichmäßig auf die Menge der Speicherverwaltungen verteilt. Um hierbei Kollisionen zu

vermeiden, ist die Empfehlung der Autoren, die Anzahl der Heaps größer als die Zahl der Prozessoren zu setzen. Bei dieser Vorgehensweise besteht aber weiterhin die Möglichkeit, dass mehrere Prozessoren auf den gleichen Heap abgebildet werden. Eine automatische Anpassung der Heapanzahl an die Zahl der Prozessoren wie bei PTmalloc erfolgt nicht.

Um bei der Freigabe von Speicher in einem fremden Heap ein mögliches Blockieren eines Prozessors zu minimieren, erfolgt der Zugriffsschutz bezüglich einzelner Listen und nicht der gesamten Speicherverwaltung.

Der vom Betriebssystem allozierte Speicher wird bei LKmalloc in Bereiche von jeweils 4 MB aufgeteilt, die den einzelnen Speicherverwaltungen zugeordnet werden. Hierdurch gelingt es, analog zu PTmalloc, die Anzahl der Systemaufrufe zu reduziert. Desweiteren lässt sich der gesamte Block an das Betriebssystem zurückgegeben, falls innerhalb eines solchen Bereiches kein von der Applikation benutzter Speicherblock vorhanden ist. Eine Wiederverwendung in einer anderen Speicherverwaltung ist nicht vorgesehen. Diese festen Speicherblöcke sind nicht mit der Benutzung von Containern (siehe Abschnitt 8.2.2.3) zu vergleichen, da im Gegensatz zu diesen Anfragen aus allen Größenklassen aus den 4 MB großen Speicherbereichen beantwortet werden.

8.3.3 Hoard

Das Ziel bei der Implementierung von Hoard (siehe [BMBW00]) war ein beschränkter Fragmentierungsgrad und gleichzeitig eine hohe parallele Effizienz. Erreicht wurde dies durch eine Kombination von Containertechnik (siehe Abschnitt 8.2.2.3) und multiplen Heaps (siehe Abschnitt 8.2.3).

Jeder Prozessor wird ähnlich zu LKmalloc in Hoard auf einen Heap abgebildet. Innerhalb dieser Speicherverwaltungen erfolgt die Speicherung der Blöcke in Containern. Die Einteilung der hierbei verwendeten Größenklassen ist abhängig von der jeweils verwendeten Version von Hoard, wobei diese in neueren Implementierungen vergleichbar mit der Einteilung bei PTmalloc ist.

Zusätzlich zu den Prozessorheaps existiert ein sogenannter *globaler* Heap. Über diesen erfolgt der Austausch von freien Containern. Letztere werden von einem Prozessorheap V_i an den globalen Heap übergeben, falls der Anteil des freien Speichers in V_i einen gewissen Wert überschreitet. Sei hierbei u_i die Größe des Speichers in V_i, der zur Zeit von der Anwendung genutzt wird und a_i, die Größe des gesamten allozierten Speichers in V_i. Seien weiterhin $f \in [0,1)$ und $K \in \mathbb{N}$. Die genaue Bedingung für den Austausch von Containern lautet dann:

$$u_i < fa_i \quad \text{und} \quad u_i < a_i - K \cdot s_c,$$

wobei s_c die Größe der Container angibt. Da umgekehrt stets $u_i \geq fa_i$ und $u_i \geq a_i - Ks_c$ gilt, kann ein beschränkter Fragmentierungsgrad in Hoard garantiert werden. Der von Hoard benötigte Speicher ist somit von der gleichen Ordnung, wie der des ausgeführten Programms.

Ist der übergebene Container frei, kann er für die Allokation von Speicher in einer beliebigen Größenklasse wiederverwendet werden. Anderenfalls ist die Wiederverwendung auf die gleiche Größenklasse beschränkt.

Bei einer Speicheranfrage in einem Prozessorheap versucht Hoard zunächst, diese aus den lokal zur Verfügung stehenden Containern zu beantworten. Ist dies nicht möglich, wird, sofern vorhanden, einen Container vom globalen Heap transferiert. Ist dort kein passender Container zu finden, erfolgt die Allokation eines neuen Containers vom Betriebssystem.

Durch den hierbei möglichen Austausch von teilweise benutzten Containern zwischen verschiedenen Prozessoren, kann passives False-Sharing auftreten.

Weiterhin erfolgt keine Anforderung von zusätzlichem Speicher, z.B. in Form von freien Containern. Somit ist die Zahl der Systemaufrufe bei der Allokation von großen Speichermengen sehr hoch. Dies wird verstärkt durch die geringe Größe der Container ($s_c = 8$ kB).

8.4 Rmalloc

Wie bereits erwähnte, konnte keiner der beschriebenen freien Allokatoren (PTmalloc und Hoard) das ineffiziente Verhalten der $\mathcal{H}$-Matrix-Arithmetik verbessern oder stand nicht als verfügbares Programm zur Verfügung (LKmalloc). Auch waren die bisherigen Speicherverwaltungen nur bedingt auf die großen Speichermengen im Kontext der $\mathcal{H}$-Matrizen ausgelegt. Desweiteren sollte die Verwendung von objektorientierten Programmiertechniken mit vielen kleinen Speicherblöcken keinen negativen Einfluss auf die Laufzeit des Programms haben. Aus diesen Gründen wurde eine neue Speicherverwaltung implementiert, die speziell auf diese Anforderungen eingeht. Anwendung fand die im folgenden vorzustellende Verwaltung bereits bei den numerischen Tests in den Kapiteln 6 und 7.

Ursprünglich war *Rmalloc* als Speicherverwaltung basierend auf der „Simple Segregated Storage“-Technik (siehe Abschnitt 8.2.2.1), kombiniert mit multiplen Heaps (Abschnitt 8.2.3) implementiert. Motiviert wurde dies durch die Beobachtung, dass sich die bei der $\mathcal{H}$-Matrix-Arithmetik mit konstantem Rang auftretenden Speicheranfragen üblicherweise in eine beschränkte Zahl von identischen Blockgrößen aufteilen lassen. In diesem Fall ist die Wahrscheinlichkeit für die Wiederverwendung eines freigegebenen Blockes sehr groß. Hieraus leitet sich auch der Name „Rmalloc“ („R“ für *recycled*, wiederverwerten) ab. In diesem Kontext ist die verwendete Technik sehr effizient, wie die numerischen Beispiele in Abschnitt 8.5 zeigen.

Wird dagegen kein fester Rang, sondern eine feste Genauigkeit und somit ein adaptiver Rang verwendet, so versagt dieser einfache Ansatz, da die auftretenden Speicheranfragen eine beliebige Größe haben können.

Aus diesem Grund wurde die Technik der „Segregated Fits“ verwendet und mit der Containertechnik kombiniert. Letztere dient der Beschleunigung von Speicheranfragen bei kleinen Blockgrößen. Die Klassifizierung der Speichergrößen erfolgt durch die Größenklassen

$$S_i := \begin{cases} S_{i-1} + 8, & i \leq 32 \\ S_{i-1} + 32, & 32 < i \leq 64 \\ S_{i-1} + 128, & 64 < i \leq 88 \\ S_{i-1} + 512, & 88 < i \leq 112 \\ S_{i-1} + 2048, & 112 < i \leq 128 \\ S_{i-1} + 8192, & 128 < i \leq 144 \\ S_{i-1} + 32768, & 144 < i \leq 160 \end{cases}$$

und $S_i = b \cdot S_{i-1}$ mit der Basis $b = 1.1$ für die Klassen $161 \leq 222$. Die Abstände zwischen den Klassen sind hierbei geringer als bei PTmalloc und LKmalloc. Alle Speicheranfragen oberhalb von $S_{\max} := S_{223} := 256$ Megabyte werden direkt an das Betriebssystem weitergegeben. Diese Grenze wurde derart gewählt, dass der entsprechende Fall sehr selten oder nie auftritt. Die Einteilung lässt sich desweiteren vom Benutzer anpassen, sodass auch zukünftige Problemgrößen behandelt werden können.

Da alle Speicherblöcke, die von Rmalloc an das Programm zurückgegeben werden, auf eine Adresse ausgerichtet sind, welche ein Vielfaches von 8 darstellt, sind Speicheranfragen in den ersten 32 Größenklassen exakt. Dies ist auch der Bereich, in dem die Containertechnik angewendet wird.

Für die übrigen Größenklassen findet die klassische „Segregated Fits“-Technik Anwendung. In diesem Größenbereich arbeitet Rmalloc analog zu LKmalloc.

In den nächsten Abschnitten werden die jeweiligen Algorithmen genauer beschrieben. Es wird dabei auch auf die Unterschiede zwischen LKmalloc bzw. PTmalloc und Rmalloc eingegangen.

8.4.1 Allgemeine Speicherallokation und -Freigabe

In diesem Abschnitt werden die von der Applikation sichtbaren Funktionen zur Speicherallokation bzw. Speicherfreigabe vorgestellt. Die wichtigste Aufgabe der dabei verwendeten Prozeduren `malloc` und `free` ist eine grobe Klassifikation in *kleine*, *mittlere* und *große* Größen, nach der die verschiedenen Spezialroutinen aufgerufen werden.

Zunächst zur Allokation von Speicher in der Funktion `malloc`. Das vollständige Verfahren ist in Algorithmus 8.4.1 dargestellt.

Um eine spätere Identifizierung des von Rmalloc an die Applikation zurückgegebenen Speicherblockes zu ermöglichen, werden diesem zusätzliche Datenfelder vorangestellt. Im einzelnen sind dies die Größe des Blockes und ein Zeiger auf den Heap bzw. Container, aus dem der Block entnommen wurde.

```
procedure malloc( s )
    s := s+ PREFIX_SIZE;
    if s > S_max then return sys_alloc( s ) ;
    if kein Prozessorheap vorhanden then
        if kein unbenutzter Heap vorhanden then V := neuer Heap;
        else                                    V := unbenutzter Heap;
        Setze V als Prozessorheap;
    else
        V := Prozessorheap;
    endif;
    lock( V );
    if s ≤ S_small then b := small_alloc( V, s );
    else
        s := s+ POSTFIX_SIZE;
        b := middle_allocV, s;
    endif;
    unlock( V );
    return b ;
end;
```

Algorithmus 8.4.1: Allokation in Rmalloc

Um Platz für diese Daten zu erhalten, wird deshalb die Größe der Speicheranfrage zunächst dementsprechend korrigiert. Übersteigt die resultierende Speichergröße $S_{\max}$, erfolgt die Beantwortung durch das Betriebssystem. Dieser Fall führt direkt zum Aufruf der entsprechenden Systemroutine. Da hierbei kein Heap zum Einsatz kommt, sind Schutzmaßnahmen nicht erforderlich.

Erfolgt die Bearbeitung der Anfrage dagegen von Rmalloc, wird zunächst der lokale Heap des Prozessors ermittelt (siehe hierzu Abschnitt 8.4.4). Ist kein Prozessorheap vorhanden, dieser Aufruf von `malloc` somit der erste auf diesem Prozessor, wird in einer Liste nach einem freien, von keinem Prozessor benutzten Heap gesucht. Existiert keine freie Speicherverwaltung, erfolgt das Anlegen eines neuen Heaps. Anschließend wird die Speicherverwaltung fest mit dem Thread gekoppelt.

Nachdem der lokale Heap bestimmt wurde, erfolgt der Schutz vor parallelem Zugriff durch andere Prozessoren. In Rmalloc wird hierbei der gesamte Heap geschützt. Obwohl ein Listenbasierter Schutz potentiell effizienter ist, hat dieser Ansatz den Vorteil, dass z.B. statistische Daten des Heaps verändert werden können, ohne einen zusätzlichen Schutz einzuführen. Auch haben Tests in diesem Zusammenhang keine Beeinträchtigung der parallelen Effizienz gezeigt. Der Schutz selbst erfolgt mittels Mutices (siehe Abschnitt 5.2.1.1).

Abhängig von der Größe des angeforderten Speicherbereiches verzweigt der Algorithmus anschließend und ruft die entsprechenden Funktionen auf. „Kleine“ Blockgrößen werden dabei durch eine Konstante $S_{\text{small}} > 0$ von „mittleren“ Größen abgegrenzt. Für letztere ist zusätzlicher Speicher für Verwaltungsdaten nötig (siehe Abschnitt 8.4.3), welcher an dieser Stelle hinzugefügt wird.

Die Freigabe von Speicher ist in dem nachfolgenden Algorithmus aufgeführt.

```
procedure free( b )
    if kein Heap oder Container mit b assoziiert then
        sys_dealloc( b );
    else
        if size(b) ≤ S_small then small_free( b );
        else
            lock( V(b) );
            middle_free( V(b), b );
            unlock( V(b) );
        endif;
    endif;
end;
```

Algorithmus 8.4.2: Speicherfreigabe in Rmalloc

Zunächst wird überprüft, ob Rmalloc oder das Betriebssystem den erhaltenen Block verwalten. Letzteres ist der Fall, falls weder ein Heap, noch ein Container mit dem Block assoziiert sind.

Über die im Speicherblock festgehaltene Größe kann anschließend bestimmt werden, ob es ein Block kleiner oder mittlerer Größe ist und an welche Funktion die Verwaltung zu übergeben ist.

8.4.2 Verwaltung kleiner Größenklassen

Insbesondere für kleine Speicherblöcke, wie sie in objektorientierten Sprachen sehr häufig vorkommen, etwa in Listen, als Iteratoren usw., sollte die Speicherverwaltung möglichst wenig Zeit bei der Allokation bzw. Freigabe benötigen. Aus diesem Grund wurde für solche Blöcke das Containerkonzept (siehe Abschnitt 8.2.2.3) verwendet.

Die dabei auftretenden Größenklassen sind in Rmalloc allerdings eingeschränkt auf Vielfache von 8. Zum einen ist hierdurch die Berechnung der Größenklasse zu einer gegebenen Speicheranfrage bzw. -freigabe in $\mathcal{O}(1)$ möglich, andererseits hat ein freier Block in der entsprechenden Klasse stets eine optimale Größe. Da zudem alle Blöcke der Klasse identisch groß sind, kann die gesamte Anfrage in $\mathcal{O}(1)$ bearbeitet werden.

Weiterhin sind keine zusätzlichen Verwaltungsdaten für das Zusammenfügen und Teilen von Speicherbereichen notwendig (siehe auch Abschnitt 8.4.3), womit die interne Fragmentierung lediglich durch zwei Datenfelder definiert ist: die Größe des Blockes und einen Zeiger auf den Container, aus dem der Block stammt. Das Abspeichern der Größeninformation ist notwendig, um die Größenklasse bei der Freigabe ermitteln zu können. Der genaue Aufbau eines Containers ist in Abbildung 8.4.1 dargestellt.

Ist ein Container frei, also keiner der enthaltenen Blöcke in Benutzung, so wird er nicht sofort freigegeben, sondern in einer speziellen Liste zwischengespeichert. Bei Bedarf ist hierdurch die direkte Wiederverwendung dieser Container möglich. Steht kein freier Contai-

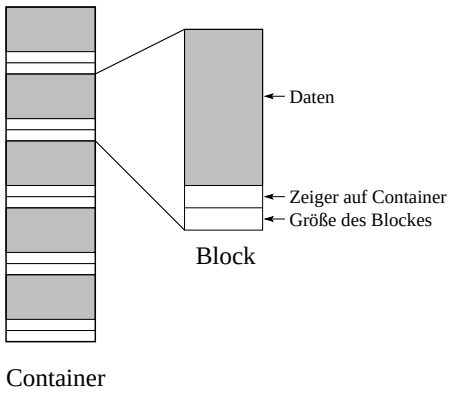


Abbildung 8.4.1: Aufbau eines Containers in Rmalloc

ner zur Verfügung, wird über die Funktion `middle_alloc` (siehe Abschnitt 8.4.3) ein neuer Speicherblock angefordert.

Um die Suche nach einem nicht-leeren Container in konstanter Zeit zu bewältigen, werden in Rmalloc in jeder Größenklasse zwei Listen von Containern geführt, wobei zwischen *nicht-leeren* und *vollen* Containern unterschieden wird. In letzteren sind sämtliche enthaltene Blöcke in Benutzung. Somit kann man durch einen Zugriff auf die Liste der nicht-leeren Container bestimmen, ob die Speicheranfrage direkt oder durch die Allokation eines neuen Containers beantwortet werden kann.

Der vollständige Algorithmus für die Allokation kleiner Speicherblöcke lautet:

```
procedure small_alloc( s )
    S_i := Größenklasse von s;
    bestimme nicht-leeren Container in S_i;
    if kein nicht-leerer Container in S_i vorhanden then
        if alter Container vorhanden then C := alter Container;
        else C := middle_alloc( CONTAINER_SIZE );
    else
        C := nicht-leerer Container in S_i;
    endif;
    b := unbenutzter Block in C;
    if C ist voll then
        verschieben von C in Liste voller Container;
    return b ;
end;
```

Algorithmus 8.4.3: Allokation von Speicherblöcken kleiner Größe

Wird ein Block kleiner Größe freigegeben, ist es über den gespeicherten Zeiger möglich, den korrespondierende Container direkt zu ermitteln. Nach der Freigabe ist der entsprechende Container entweder nicht-leer oder leer. Je nach Zustandsänderung wird er anschließend in

die entsprechenden Listen einsortiert.

Überschreitet die Menge der freien Container einen festen Wert, so erfolgt die Freigabe einer dieser Container an Rmalloc über die Funktion `middle_free` (siehe Abschnitt 8.4.3).

```
procedure small_free( b )
    S_i := Größenklasse von b;
    C := Container von b;
    gib b in C frei;
    if C war voller Container then
        verschiebe C in Liste nicht-leerer Container;
    else if C ist leer then
        entferne C aus S_i;
end;
```

Algorithmus 8.4.4: Freigabe von Speicherblöcken kleiner Größe

8.4.3 Verwaltung mittlerer Größenklassen

Bei der Allokation eines Speicherbereichs mit einer *mittleren* Größe $S_{\text{small}} < s \leq S_{\max}$ wird zunächst die entsprechende Größenklasse bestimmt. Anschließend erfolgt eine „First-Fit"-Suche in der Liste freier Blöcke dieser Klasse. Ist in dieser kein passender Block vorhanden, dehnt Rmalloc die Suche auf die nachfolgenden Größenklassen aus.

Konnte ein freier Block b mit der Größe $s_b \geq s$ gefunden werden und überschreitet die Größe $s_b - s$ des Restblockes b' den Wert S_{small}, so erfolgt die Teilung von b und das Einfügen von b' in Rmalloc. Andernfalls wird b als Ganzes verwendet und der Restspeicher als interne Fragmentierung angesehen.

Die Verwendung von S_{small} ist notwendig, da die Verwaltung kleinerer Größen in Rmalloc anders als bei mittleren Größen erfolgt und ein entsprechender Restblock nicht in einen Container eingefügt werden kann.

War die Suche nach einem Speicherblock mit einer hinreichenden Größe in Rmalloc erfolglos, wird vom Betriebssystem ein neuer Block alloziert. Hierbei findet eine ähnliche Technik wie bei LKmalloc (Abschnitt 8.3.2) Anwendung. Allerdings alloziert Rmalloc keine Blöcke einer festen Größe, sondern, ähnlich zu PTmalloc, zusätzlich zu s einen Bereich der Größe $s_p > 0$.

Dies hat mehrere Gründe. Zum einen kann bei einer nachfolgenden Allokation eines Blockes der Größe s' mit $s' < s_p$ vom Betriebssystem der Speicher ohne einen teuren Systemaufruf geliefert werden. Zum anderen sind die Speicherbereiche, die vom Betriebssystem pro Heap zurückgegeben werden, nicht notwendigerweise benachbart (siehe auch Abbildung 8.4.2). Der Grund hierfür ist die Anforderung von Speicher in konkurrierenden Heaps in Rmalloc. Wird stets nur Speicher der Größe s angefordert, ist somit das Zusammenfügen zu einem größeren Block im allgemeinen unmöglich, da die benachbarten Speicherbereiche in anderen Heaps verwaltet werden. Dies hat unter Umständen einen negativen Einfluss auf den Fragmentierungsgrad.

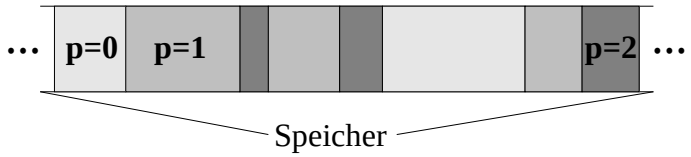


Abbildung 8.4.2: Lage von Speicherbereichen durch das Betriebssystem

Anders als bei PTmalloc wird in Rmalloc s_p aber nicht fest vorgegeben, sondern an die Größe A_i des bisher in Heap V_i allozierten Speicherbereiches angepasst, z.B. $s_p = 0.1 \cdot A_i$. Hierdurch kann bei einer Applikation mit einem geringen Speicherbedarf die interne Fragmentierung in Maßen gehalten werden ($\leq s_p$). Umgekehrt nimmt bei einem Programm mit einem sehr großen Verbrauch an Speicher die Zahl der Systemaufrufe nicht linear zu.

Speicher wird auch nicht direkt vom Betriebssystem angefordert, sondern die Anfrage an eine Zwischenschicht weitergeleitet. Diese Schicht besteht aus einer Liste von vom Betriebssystem allozierten Blöcken. Erst falls keiner dieser Speicherblöcke genügend Platz für die Speicheranfrage bietet, wird ein Systemaufruf benutzt. Der nicht genutzte Speicherbereich der Blöcke in dieser Schicht heißt auch *Wildnis*, da der Speicher dort nicht verwaltet und somit „wild“ ist.

Nachdem ein Speicherblock b für die Anfrage gefunden bzw. angelegt wurde, erfolgt die Markierung von b als benutzter Speicherblock. Durch diese Markierung haben die angrenzenden Speicherbereiche die Möglichkeit, zu testen, ob ein Zusammenfügen mit b möglich ist.

Die Ausführungen über die Arbeitsweise der Allokation sind in dem nachfolgenden Algorithmus zusammengefasst.

```
procedure middle_alloc( V, s )
    i := Größenklasse von s;
    for j := i, ..., m do
        for all b ∈ S_j do
            if size(b) ≥ s then
                if size(b) − s > S_small then
                    trenne b und füge Restblock in Rmalloc ein;
                markiere b als benutzt;
                return b;
            endif;
    b := neuer Block { neuer Block vom Betriebssystem }
    markiere b als benutzt;
    return b;
end;
```

Algorithmus 8.4.5: Allokation von Speicherblöcken mittlerer Größe

Ein freigegebener Speicherblock b wird in Rmalloc sofort mit benachbarten, nichtbenutzten

Bereichen zusammengefügt. Der Nachbarblock mit der kleineren Adresse wird hierbei als *Vorgänger*, der Nachbarblock mit der größeren Adresse als *Nachfolger* von b bezeichnet. Je nach Lage von b können beim Zusammenfügen verschiedene Fälle auftreten:

1. Block b besitzt keinen Nachfolger, der Speicherbereich nach b ist somit Teil der Wildnis: b wird, soweit möglich, mit seinem Vorgänger vereinigt und *nicht* in Rmalloc eingefügt, sondern der Wildnis angegliedert.

2. b hat einen Nachfolger: Soweit möglich, wird b mit seinem Vorgänger und Nachfolger vereinigt und in die entsprechenden Größenklasse von Rmalloc eingefügt.

Für die schnelle Bestimmung des Vorgängers werden sogenannte *Boundary Tags* verwendet (siehe [Knu73]). Wie in Abbildung 8.4.3 zu sehen, wird hierbei die Größe eines Blockes b sowohl am Anfang als auch am Ende von b abgelegt. Man beachte, dass die Größeninformation nur für den Fall eines freien b notwendig ist. Anderenfalls genügt eine Markierung im Nachfolger von b. Auf diese Weise lässt sich die Größe im eigentlichen Datenbereich von b ablegen wenn b freigegeben wird. Dies reduziert die interne Fragmentierung, da kein zusätzlicher Speicher für Verwaltungsinformationen notwendig ist.

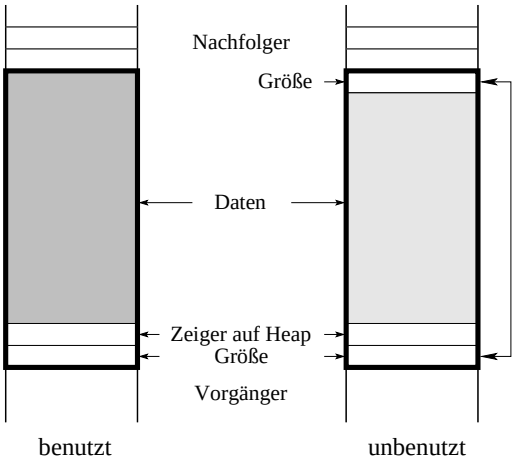


Abbildung 8.4.3: *Boundary Tags* in Speicherblöcken

An dieser Stelle sei ein technisches Detail erwähnt, welches die Speicherung der benötigten Markierungen betrifft. Typischerweise werden dabei boolsche Variablen genutzt, die lediglich ein Bit benötigen. Hier lässt sich die Ausrichtung aller Speicherblöcke auf eine durch 8 teilbare Adresse ausnutzen, da in diesem Fall die unteren drei Bit des Größenfeldes stets Null sind. Auch diese Technik reduziert die interne Fragmentierung, da gerade bei kleinen Speicherblöcken zusätzliche Verwaltungsdaten zu einer erheblichen Zunahme des Fragmentierungsgrades führen können. In Abbildung 8.4.4 ist die Verwendung der einzelnen Bits in Rmalloc dargestellt.

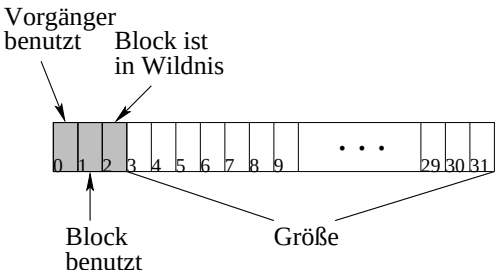


Abbildung 8.4.4: Speicherung von Markierungen auf 32 Bit Systemen

Der endgültige Algorithmus für die Freigabe eines mittleren Speicherblockes lautet:

```
procedure middle_free( V, b )
    if  b hat keinen Nachfolger  then
        if  Vorgänger b' von b ist unbenutzt  then
            entferne b' aus Rmalloc;
            b' := b' + b; b := b';
        endif
    else
        if  Vorgänger b' von b ist unbenutzt  then
            entferne b' aus Rmalloc;
            b' := b' + b; b := b';
        endif
        if  Nachfolger b'' von b ist unbenutzt  then
            entferne b'' aus Rmalloc;
            b := b + b'';
        endif;
        bestimme Größenklasse von b;
        füge b in Rmalloc ein;
    endif
end;
```

Algorithmus 8.4.6: Freigabe von Speicherblöcken mittlerer Größe

8.4.4 Zuordnung von Heaps zu Threads

Rmalloc verwendet PThreads (siehe Abschnitt 5.2.1.1) für das Ansprechen der Threadfunktionalität. Dies erlaubt die Verwendung der Speicherverwaltung auf einer großen Menge von unterschiedlichen Betriebssystemen, da es sich um einen POSIX-Standard handelt und somit weit verbreitet ist.

In der Definition von PThreads enthalten ist das Konzept von *Thread-spezifischen* Daten. Hierbei handelt es sich um Daten, die fest an einen Thread gebunden sind und mittels eines Schlüssels angesprochen werden können. Mit dem gleichen Schlüssel lässt sich an die Daten

auch eine Destruktionsfunktion binden, die beim Beenden des Threads aufgerufen wird, um die gespeicherten Daten zu entfernen.

Diese Funktionalität wird in Rmalloc genutzt, um Heaps dynamisch mit Threads zu assoziieren, wobei ein Zeiger auf einen Heap in den privaten Daten des Threads gespeichert wird. Bei einer Speicheranfrage werden vom aktuellen Thread die privaten Daten abgerufen, um den dort gespeicherten Heap zu erhalten. Für weitere Speicheranfragen wird hierdurch stets der gleiche Heap verwendet. Gleichzeitig kann kein anderer Thread auf diesen Heap zugreifen, wodurch aktives False-Sharing ausgeschlossen ist.

Wird der Thread beendet, erfolgt der Aufruf der zuvor registrierten Destruktionsfunktion, welche den mit dem Thread verbundenen Heap freigibt. Anschließend kann dieser Heap in Speicheranfragen kommender Threads genutzt werden.

Auf diese Weise ist die Zahl der Speicherverwaltungen in Rmalloc höchstens so groß, wie die Anzahl von parallel ausgeführten Threads. Hierdurch wird allerdings auch der Einsatzbereich von Rmalloc eingeschränkt, da die Threadanzahl im allgemeinen nicht an die Zahl der Prozessoren gekoppelt ist. Als Beispiel sei hier eine Client-Server-Anwendung betrachtet, in der ein Server auf Anfragen von beliebig vielen Client-Systemen reagieren muss. Die Verbindungen mit den verschiedenen Clients kann hierbei von permanenter Dauer sein, wobei der Server lediglich sporadisch Arbeit pro Client verrichtet. In diesem Szenario ist die Anzahl der Heaps so groß wie die Zahl der Clients und nicht gekoppelt an die Zahl der Prozessoren.

Besser geeignet für die Verwendung von Rmalloc ist dagegen eine Applikation, die analog zu dem in Abschnitt 5.2.1.1 beschriebenen Threadpool arbeitet. Hierbei ist die Zahl der Threads vergleichbar bzw. identisch mit der Prozessoranzahl.

8.5 Numerische Tests

In diesem Abschnitt sollen die in den vorangegangenen Kapiteln beschriebenen Eigenschaften der verschiedenen Speicherverwaltungen in praktischen Tests geprüft werden. Dabei werden sowohl rein synthetische Tests zur gezielten Untersuchung einzelner Eigenschaften, als auch praktische Experimente mit typischen Programmen durchgeführt.

Die in diesen Versuchen genutzten Speicherverwaltungen sind *SUNmalloc*, PTmalloc (Abschnitt 8.3.1), LKmalloc (Abschnitt 8.3.2), Hoard (Abschnitt 8.3.3), *MTmalloc* und Rmalloc (Abschnitt 8.4). SUNmalloc ist die übliche malloc-Implementierung unter dem Betriebssystem *Solaris* der Firma Sun. Es handelt sich hierbei um eine sequentielle Speicherverwaltung, wobei die Multithreadsicherheit durch einen globalen Mutex erfolgt. MTmalloc wird unter Solaris als Alternative für den Einsatz von mehreren Threads bereitgestellt.

Bei LKmalloc konnte auf keine frei verfügbare Implementierung zurückgegriffen werden, weshalb eine modifizierte Version von Rmalloc zum Einsatz kam, die das Verhalten von LKmalloc soweit wie möglich simuliert.

Bei Hoard ist außerdem anzumerken, dass dieser Allokator nicht in der Lage war, mehr

als 2 GByte zu verwalten, weshalb die Ergebnisse für einige Tests fehlen.

8.5.1 Synthetische Tests

In [BMBW00] wurden verschiedene Testprogramme verwendet, um die Skalierbarkeit und das False-Sharing-Verhalten von Hoard im Vergleich zu anderen Speicherverwaltungen zu zeigen. In diesem Abschnitt werden ähnliche Programme verwendet, um die gleichen Eigenschaften von Rmalloc zu überprüfen.

Im Einzelnen sind dies:

- *thralloc-eq*: Hierbei werden kleine, gleichgroße Speicherblöcke in jedem Thread angefordert und sofort wieder freigegeben. Die Gesamtzahl der allozierten Blöcke ist dabei stets gleich.
- *thralloc-rand*: Dieses Programm arbeitet ähnlich wie *thralloc-eq*. Es alloziert aber Blöcke von unterschiedlicher, zufälliger Größe. Die Zufallszahlen sind dabei bei allen Programmabläufen identisch.
- *active-false*: In diesem Test wird überprüft, ob aktives False-Sharing auftritt. Dazu wird in jedem Thread ein kleiner Speicherblock angelegt, beschrieben und wieder freigegeben. Dieser Vorgang wird mehrmals wiederholt.
- *passive-false*: Um passives False-Sharing zu testen, wird hierbei vom Hauptprogramm zunächst für jeden Thread ein Speicherblock alloziert und diesem übergeben. Jeder Thread gibt den Block daraufhin sofort frei und arbeitet anschließend wie *active-false*.

Zunächst wird mittels `thralloc-eq` für jede Speicherverwaltung die Zeit zur Beantwortung einer Speicheranfrage in Abhängigkeit von der Blockgröße getestet. Insbesondere der Einfluss der Containertechnik für kleine Blöcke in Rmalloc soll hierbei untersucht werden. In Abbildung 8.5.1 sind die gemessenen Ergebnisse dargestellt.

Rmalloc zeigt hierbei ein sehr gutes Verhalten. Dieses ist vergleichbar mit Hoard, welches ebenfalls Container verwendet. Deutlich erkennbar ist der Wechsel der Verwaltungstechnik bei 256 Bytes. Da für größere Blöcke Speicherbereiche getrennt bzw. zusammengefügt werden, erhöht sich auch der Aufwand. Außerdem ist die Berechnung der Größenklasse in diesem Bereich teurer. Insbesondere die direkte Berechnung der Größenklasse im Bereich unterhalb von 512 Byte führt bei LKmalloc zu einer vergleichsweise geringen Varianz der Antwortzeiten. Die Art der Verwaltung, vergleichbar mit Rmalloc bei mittleren Speichergrößen, bedingt dagegen eine größere Ausführungszeit als bei Rmalloc für Anfragen kleiner als 256 Byte.

Steigt die Blockgröße und damit auch der Speicherverbrauch, führt die adaptive Präallokation von Speicher (siehe Abschnitt 8.4.3) zu einer geringeren Laufzeit als bei LKmalloc, bei welchem jeweils Blöcke konstanter Größe vom Betriebssystem angefordert werden.

Bemerkenswert ist auch das Verhalten von MTmalloc, bei welchem zwischen zwei Zweierpotenzen konstante Zeiten ermittelt wurden. Die Erklärung hierfür liefert Abbildung 8.5.2.

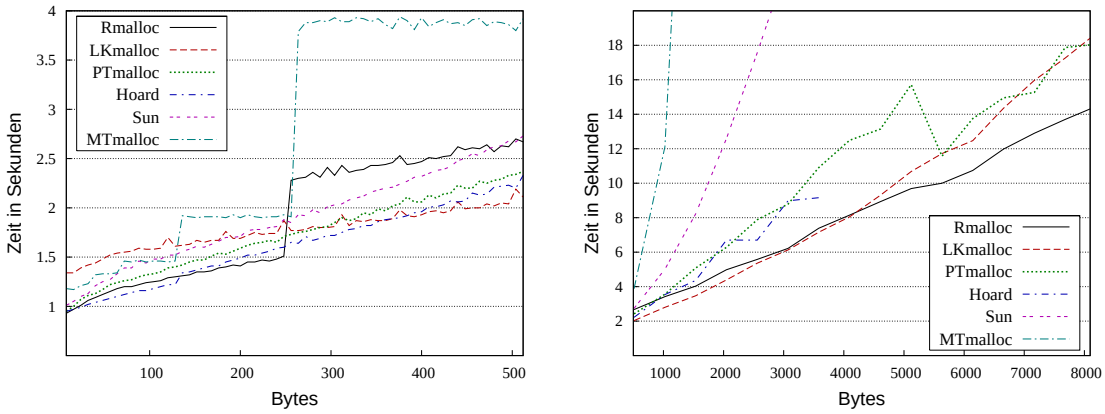


Abbildung 8.5.1: Einfluss der Blockgröße auf die Beantwortungszeit

Dort ist der Speicherverbrauch der einzelnen Allokatoren bei diesem Test aufgetragen. Man erkennt, dass MTmalloc offensichtlich die Blockgröße der Anfrage auf die nächste Zweierpotenz rundet. Wie in Abschnitt 8.2.2 beschrieben, vermeidet man hierdurch die Suche nach einem passenden Block innerhalb der Größenklasse. Allerdings gelingt es MTmalloc nicht, dies in entsprechende geringe Laufzeiten umzusetzen. Insbesondere bei größeren Speicheranfragen nimmt die Antwortzeit überproportional zu und erreicht bei der größten gemessenen Blockgröße einen etwa 90mal höheren Wert als Rmalloc.

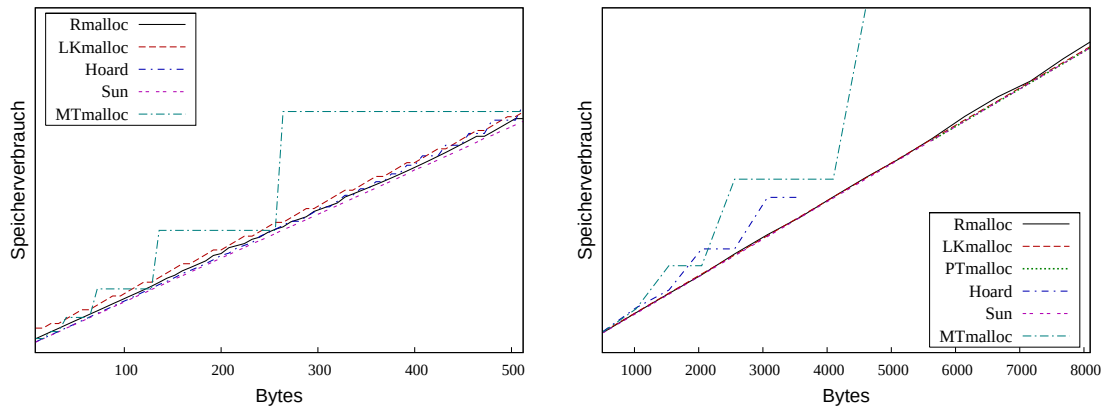


Abbildung 8.5.2: Speicherverbrauch in Abhängigkeit von der Blockgröße

Die verbleibenden Speicherverwaltungen besitzen einen sehr ähnlichen Speicherverbrauch. Bei PTmalloc war die Messung bei kleinen Blockgrößen nicht möglich, da der Speicher bei der Freigabe wieder an das Betriebssystem zurückgegeben wird und aufgrund der Implemen-

tierung der Verbrauch erst nach Beenden des Tests ermittelt werden konnte. Lediglich für vergleichsweise große Speicherblöcke, bei denen PTmalloc direkt das Betriebssystem bemüht, zeigt sich ein analoger Wert. Die hierdurch definierte Grenze führt bei PTmalloc auch zu einer Reduktion der Laufzeit, etwa auf das Niveau von LKmalloc. Bei Hoard zeigten sich Probleme mit der Stabilität, weshalb Messungen oberhalb einer Blockgröße von 4096 Byte nicht möglich waren.

Im folgenden werden die genannten Testprogramme genutzt, um die parallele Effizienz der einzelnen Speicherverwaltungen zu überprüfen. Für einen absoluten Vergleich ist hierfür zunächst in der folgenden Tabelle die sequentielle Laufzeit angegeben.

Sequentielle Laufzeit [s]						
	Rmalloc	LKmalloc	PTmalloc	Hoard	SUNm.	MTmalloc
`thralloc-eq`	30.3	37.9	33.4	28.8	47.3	127.7
`thralloc-rand`	121.2	172.4	125.8	122.3	139.1	226.8
`afalse`	88.4	137.5	95.6	199.1	76.9	102.4
`pfalse`	83.5	150.8	76.6	209.8	73.4	101.9

Im Test `thralloc-eq` beweisen sowohl Rmalloc als auch LKmalloc eine durchgehend gute parallele Effizienz, wie in Abbildung 8.5.3 (links) zu sehen ist. Obwohl ein leichter Rückgang bei einer Erhöhung von p zu beobachten ist, sinkt $E(p)$ nicht unter 90%. Anders verhält es sich bei PTmalloc und Hoard. Beide Allokatoren zeigen einen deutlichen Verlust der parallelen Skalierbarkeit, welche bei PTmalloc zudem nicht kontinuierlich ist. Keinerlei parallele Effizienz ist bei SUNmalloc und MTmalloc festzustellen. Innerhalb von SUNmalloc verursacht der Mutex, der die sequentielle Verwaltung schützt, diese schlechte Leistung. Bei MTmalloc scheint das Verhalten durch häufige Systemaufrufe bedingt zu sein.

Bei einer zufälligen Blockgröße ergibt sich ein analoges Verhalten (Abbildung 8.5.3 rechts). Die unregelmäßige parallele Effizienz von PTmalloc zeigt sich aber in einer wesentlich ausgeprägteren Form. Hierdurch sind praktisch keine Vorhersagen über Programmlaufzeiten möglich.

Ein geändertes Bild erhält man bei einem Test bezüglich False-Sharing. In Abbildung 8.5.4 ist die parallele Effizienz für die einzelnen Speicherverwaltungen dargestellt. PTmalloc scheint hierbei besonders anfällig zu sein. Eine Erklärung für dieser Verhalten bietet die Zuweisung von Heaps zu Threads, welche in PTmalloc nicht fest, sondern dynamisch erfolgt. Hierdurch ist es möglich, dass sich ein oder mehrere Threads einen Heap teilen und aktives False-Sharing induziert wird. In Kombination mit der eingeschränkten parallelen Skalierung von PTmalloc ergibt sich ein sehr schlechtes Abschneiden in dieser Disziplin.

Ein ähnliches Zuweisungsverhalten der Heaps führt bei LKmalloc ebenfalls zu einer reduzierten parallelen Effizienz, welche aber durch das generell bessere parallele Verhalten nicht so ausgeprägt wie bei PTmalloc ist.

Durch das Aufrunden der Speichergrößen in MTmalloc wird die Wahrscheinlichkeit, dass

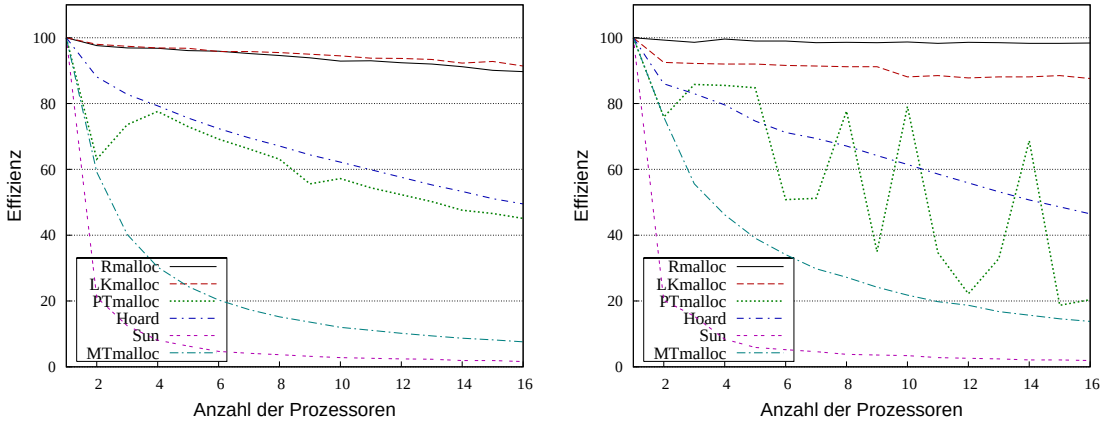


Abbildung 8.5.3: Parallele Effizienz bei `thralloc-eq` und `thralloc-rand`

sich zwei Blöcke eine Cache-Zeile teilen, reduziert bzw. diese Möglichkeit völlig ausgeschlossen. Hierdurch zeigt sich eine optimale Effizienz von MTmalloc in diesem Test.

8.5.2 Numerische Algorithmen

Nachdem im letzten Abschnitt einige grundlegende Eigenschaften der verschiedenen Speicherverwaltungen anhand von synthetischen Programmen getestet wurden, sollen im folgenden reale numerische Verfahren genutzt werden. Der Schwerpunkt liegt dabei bei der $\mathcal{H}$-Matrix-Arithmetik, aber auch andere Algorithmen werden hierzu herangezogen.

Die Grundlage für die $\mathcal{H}$-Matrix-Tests bildet das Beispiel aus Abschnitt 2.4.2. Hierbei wird die Zeit für eine Matrix-Inversion bzw. eine Matrix-Multiplikation mit fester Genauigkeit und unterschiedlichen Prozessorzahlen gemessen. Der Fall eines konstanten Ranges ist für den Vergleich der Speicherallokatoren ungeeignet, da praktisch der gesamte Speicher vor der Inversion bzw. Multiplikation angefordert wird und während der Berechnung nur wenige Aufrufe der Speicherverwaltung erfolgen.

Für die Matrix-Inversion mit fester Genauigkeit von $\varepsilon = 10^{-6}$ ergeben sich im sequentiellen Fall ($p = 1$) die folgenden Zeiten bei den unterschiedlichen Allokatoren:

Matrix-Inversion, konstante Genauigkeit $\varepsilon = 10^{-6}$, [s]							
n	Rmalloc	LKmalloc	PTmalloc	Hoard	SUNm.	MTmalloc	Speicher
4 096	12.9	12.7	12.9	17.2	13.0	13.0	44 MB
16 384	114.8	114.0	122.4	212.1	115.8	117.8	220 MB
65 536	800.7	798.3	1268.9	3068.9	818.0	924.6	1098 MB
262 144	5003.5	5043.8	30556.7	–	5257.6	8488.3	5393 MB

Aufgrund der vergleichbaren Implementierung erhält man bei Rmalloc und LKmalloc ähn-

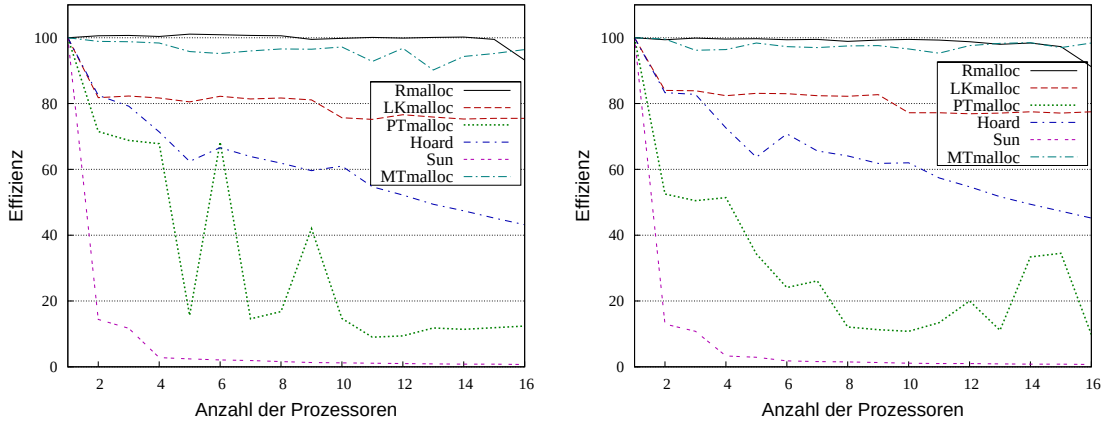


Abbildung 8.5.4: Parallele Effizienz bei `afalse` und `pfalse`

liche Ergebnisse. Lediglich bei der größten Problemdimension scheint die adaptive Präallokation von Speicher in Rmalloc einen leichten Laufzeitvorteil zu erbringen. Die mittels SUNmalloc produzierten Zeiten sind geringfügig höher, spiegeln aber eine identische Komplexität wieder.

Demgegenüber entspricht das Laufzeitverhalten von Hoard praktisch einer quadratischen Komplexität und somit in keiner Weise den theoretischen Vorhersagen aus Abschnitt 3.4.1. Eine mögliche Erklärung für dieses Verhalten besteht darin, dass Hoard nur Blöcke von wenigen Kilobyte direkt verwaltet. Größere Speicherbereiche werden hingegen an das Betriebssystem übergeben, wodurch viele teure Systemaufrufe auftreten. Bei PTmalloc zeigt sich eine solche Abkehr von der theoretischen Komplexität erst ab einem größeren Speicherverbrauch. Der wahrscheinlichste Grund für die Zunahme in diesem Fall liegt in der Sortierung der freigegebenen Speicherblöcke. Nicht ganz so ausgeprägt ist die Zunahme der Komplexität bei MTmalloc, wobei hierfür das Aufrunden der Speichergrößen verantwortlich zeichnet.

In Abbildung 8.5.5 ist die parallele Effizienz der Matrix-Inversion für die verschiedenen Speicherallokatoren dargestellt.

Völlig unerwartet sind die Ergebnisse von Hoard, welches ab 14 Prozessoren sogar höhere Laufzeiten als im sequentiellen Fall hervorruft. Dagegen ist die eingeschränkte Effizienz von SUNmalloc im Rahmen der Möglichkeiten, die eine sequentielle Implementierung bietet.

Bei Hoard und SUNmalloc führt die sequentielle Implementierung zu einer stark eingeschränkten parallelen Skalierung. Die übrigen Speicherverwaltungen zeigen bei $n = 16\,384$ dagegen ein im wesentlichen homogenes Verhalten. Die Situation ändert sich allerdings bei $n = 262\,144$, bei welchem die Ergebnisse deutlich differieren.

Durch die Verteilung der Speichermenge auf mehrere Heaps treten die negativen Folgen der Verwaltungstechnik in PTmalloc bei mehreren Prozessoren nicht so deutlich hervor.

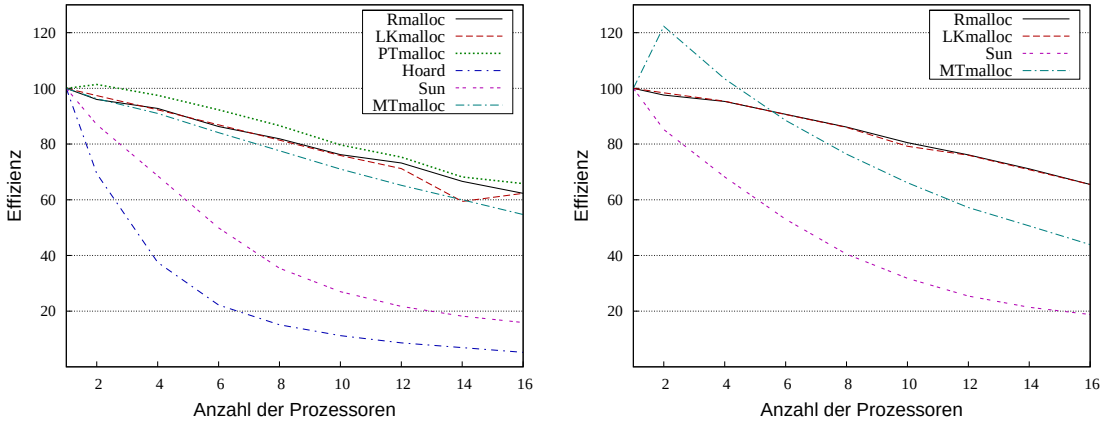


Abbildung 8.5.5: Parallele Effizienz bei Matrix-Inversion mit konstanter Genauigkeit bei $n = 16384$ und $n = 262144$

Hierdurch ergibt sich allerdings eine parallele Effizienz von weit über 100%, welche nicht mit dem realen Verhalten übereinstimmt und dementsprechend nicht dargestellt wurde. Aus dem gleichen Grund treten bei MTmalloc Superskalierungseffekte auf, wenngleich auch diese nicht so ausgeprägt sind.

Trotz einer vergleichbaren parallelen Effizienz von LKmalloc und Rmalloc ist die absolute Laufzeit von Rmalloc insbesondere bei hohem Speicherverbrauch auch im parallelen Fall geringfügig besser, wie die nachfolgende Tabelle zeigt:

	$p = 4$	$p = 8$	$p = 12$	$p = 16$
Rmalloc	1313.1 s	726.3 s	547.9 s	477.6 s
LKmalloc	1322.1 s	733.4 s	552.9 s	481.2 s

Neben den Eigenschaften der Speicherverwaltungen im Parallelbetrieb ist der Speicherverbrauch, ausgedrückt im Fragmentierungsgrad, wesentlich für die Abbildung der theoretischen Vorhersagen in praktischen Berechnungen. Allerdings ist aufgrund der Komplexität des Programms der minimal benötigte Speicher nicht bekannt. Deshalb wird als Ausgangsgröße der Speicherverbrauch zugrundegelegt, der von allen Allokatoren minimal benötigt wurde. Die Ergebnisse für die beiden in Abbildung 8.5.5 verwendeten Problemdimensionen sind in Abbildung 8.5.6 zu sehen.

Die geringste Abhängigkeit von der Anzahl der Prozessoren zeigt SUNmalloc. Dieses Verhalten war allerdings aufgrund der sequentiellen Implementierung zu erwarten. Nicht überraschen ist ebenfalls der hohe Fragmentierungsgrad von MTmalloc. Bei den übrigen parallelen Allokatoren wird PTmalloc seinem Ruf gerecht und zeigt eine vergleichsweise kleine Fragmentierung, welche bei $n = 262144$ aber nur geringfügig besser als der entsprechende Grad von LKmalloc und Rmalloc ist. Bei der kleinen Problemdimension und somit geringerem

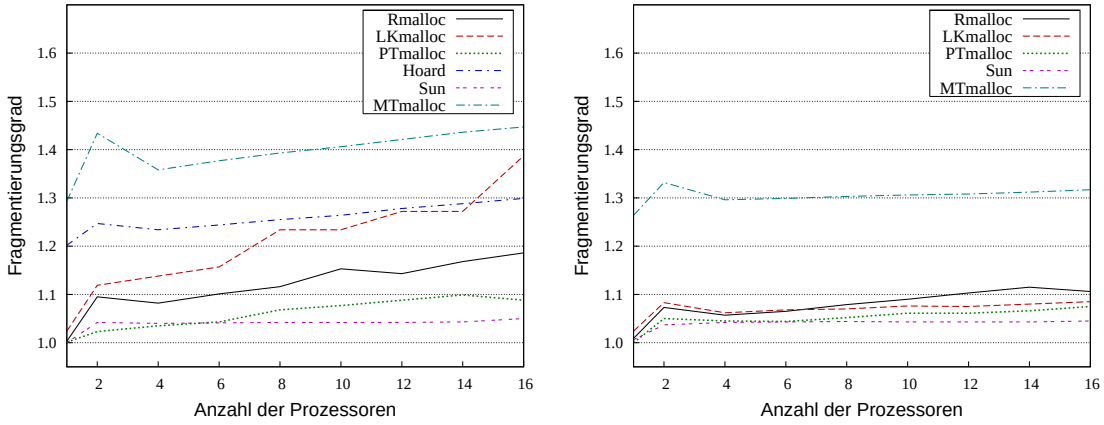


Abbildung 8.5.6: Fragmentierungsgrad bei Matrix-Inversion mit konstanter Genauigkeit bei $n = 16384$ und $n = 262144$

Speicherverbrauch ist bei LKmalloc eine relativ große Abhängigkeit von p zu beobachten, welche durch die feste Blockgröße von 4 MB bei Speicheranfragen an das Betriebssystem begründet ist.

Ebenfalls mit einer konstanten Genauigkeit von $\varepsilon = 10^{-6}$ wurde die Matrix-Multiplikation durchgeführt, wie sie in Abschnitt 3.3 beschrieben ist. Die Laufzeiten beim Einsatz der verschiedenen Speicherverwaltungen sind in der nachfolgenden Tabelle zusammengefasst.

Matrix-Multiplikation, konstante Genauigkeit $\varepsilon = 10^{-6}$, [s]							
n	Rmalloc	LKmalloc	PTmalloc	Hoard	SUNm.	MTmalloc	Speicher
4 096	45.0	44.7	45.4	61.6	45.5	45.1	68 MB
16 384	377.1	374.8	412.9	728.4	379.2	383.0	365 MB
65 536	2507.6	2508.0	4160.4	–	2527.7	2953.5	1930 MB
262 144	15111.9	15301.4	185079.4	–	15305.2	34244.0	9704 MB

Es ergibt sich ein analoges Bild zur Matrix-Inversion. Allerdings ist das Verhalten von PTmalloc noch ausgeprägter. Deutlich erkennbar ist auch der Vorteil der adaptiven Präallokation bei einem hohen Speicherverbrauch durch die geringere Laufzeit von Rmalloc im Vergleich zu LKmalloc.

Auch die parallele Effizienz, welche für die Multiplikation in Abbildung 8.5.7 dargestellt ist, zeigt ein ähnliches Verhalten wie bei der Inversion. Bemerkenswert ist die schon bei den numerischen Tests in Abschnitt 6.5.1 festgestellte Superskalierung bei fast allen Speicherverwaltungen außer SUNmalloc. Bei $n = 262144$ führt die hohe sequentielle Laufzeit von PTmalloc und MTmalloc wie bei der Inversion zu einer unrealistischen Effizienz von weit über 300% bzw. 200%, weshalb auf die Darstellung verzichtet wurde.

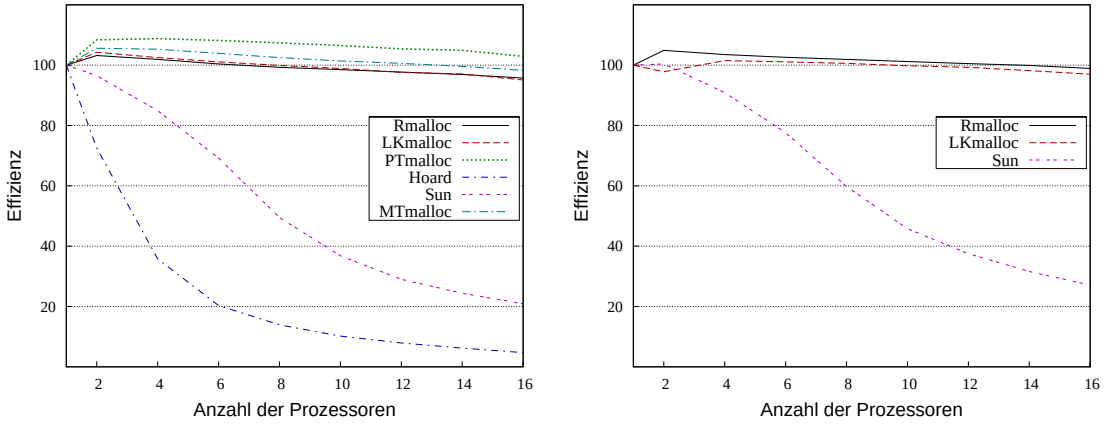


Abbildung 8.5.7: Parallele Effizienz bei Matrix-Multiplikation mit konstanter Genauigkeit bei $n = 16384$ und $n = 262144$

Der Fragmentierungsgrad, in Abbildung 8.5.8 zu sehen, zeigt im wesentlichen ein ähnliches Ergebnis wie bei der Inversion. Allerdings ist die Fragmentierung insgesamt größer. Interessant ist auch der besonders hohe Fragmentierungsgrad von Rmalloc, LKmalloc und MTmalloc bei $p = 2$. In diesem Fall scheint der zusätzliche Speicher, der aufgrund des parallelen Betriebs benötigt wird, zu einer hohen externen Fragmentierung in den beiden jeweils verwendeten Heaps zu führen.

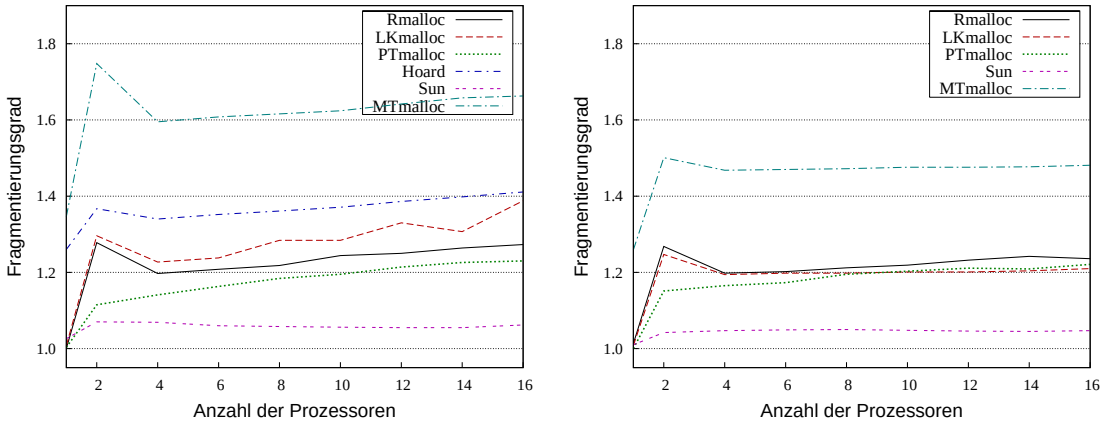


Abbildung 8.5.8: Fragmentierungsgrad bei Matrix-Multiplikation mit konstanter Genauigkeit bei $n = 16384$ und $n = 262144$

Im folgenden Beispiel wird das Programm *CD2D3D* (siehe [HKBM01]) genutzt. Es beinhaltet Algorithmen zur Lösung von Konvektions-Diffusions-Gleichungen unter Verwendung

von verschiedenen Anordnungsalgorithmen, z.B. *Reverse-Cuthill-McKee* (siehe [DER86]), konzentrischen Zyklen oder dem *Feedback-Vertex-Set-Problem* (siehe jeweils [Hac97]). Durch das iterative Lösungsverfahren dient es außerdem als Beispiel für klassische FEM-Applikationen. Das Programm ist in der Sprache C++ geschrieben und macht intensiven Gebrauch von Listen, Iteratoren und dynamischen Arrays, d.h. es werden sehr viele kleine Datenblöcke angefordert und freigegeben.

In der folgenden Tabelle sind die Zeiten für den Aufbau der Triangulation, die Diskretisierung, die Berechnung einer Anordnung mittels konzentrischen Zyklen (CYC), Reverse-Cuthill-McKee (RCM) und Feedback-Vertex-Set (FVS), sowie die Dauer des Lösungsschrittes mittels GMRES (siehe [SS86]) angegeben.

CD2D3D, $n = 1\,046\,529$, [s]					
Operation	Rmalloc	LKmalloc	PTmalloc	SUNmalloc	MTmalloc
Triang.	19.6	36.8	19.0	21.4	36.9
Diskr.	61.7	63.7	179.7	62.9	74.7
CYC	34.4	46.1	202.3	39.8	51.1
RCM	39.2	56.1	–	45.5	66.0
FVS	107.2	132.6	–	113.9	136.0
GMRES	163.5	186.5	180.2	177.7	169.1

Insbesondere der Vergleich zwischen Rmalloc und LKmalloc offenbart den Vorteil der Containertechnik bei dem verwendeten Programmierstil. Die Laufzeit kann hierdurch teilweise beträchtlich reduziert werden. Auch in diesem Beispiel zeigt sich das ineffiziente Verhalten von PTmalloc bei einem größeren Speicherverbrauch. Bei RCM und FVS wurde die Berechnung einer Anordnung abgebrochen, da die Laufzeit 15 Minuten deutlich überschritten hatte. PTmalloc ist also auch bei diesem Beispiel nicht geeignet, um theoretische Vorhersagen von Algorithmen zu testen.

SUNmalloc und MTmalloc zeigen zu den bisherigen Tests vergleichbare Ergebnisse, wobei bei MTmalloc durch das Aufrunden der Speichergrößen wieder eine höhere Laufzeit resultiert.

9 Fazit

Durch die Verwendung von $\mathcal{H}$-Matrizen lassen sich die Eingangs beschriebenen Probleme auf eine elegante und zugleich effiziente Art und Weise lösen. Die in dieser Arbeit vorgestellten Verfahren demonstrieren zudem, dass hierbei auch mehrere Prozessoren gleichzeitig genutzt werden können, um die Arithmetik der $\mathcal{H}$-Matrizen zu beschleunigen.

Die Parallelisierung führt dabei auf einem System mit gemeinsamem Speicher (siehe Abschnitt 5.2.1), wie es oftmals bei Arbeitsplatzrechnern oder kleinen Serversystemen anzutreffen ist, zu optimalen Resultaten für den Aufbau (Abschnitt 6.2), die Matrix-Vektor-Multiplikation (Abschnitt 6.3), die Addition (Abschnitt 6.4) sowie die Multiplikation (Abschnitt 6.5) einer $\mathcal{H}$-Matrix. Die Inversion mittels Gauß-Elimination, dem hierfür geeignetsten Verfahren, erfolgt dagegen nicht mit maximaler paralleler Effizienz, führt aber dennoch zu einer skalierbaren Methode und zeigt bei moderaten Prozessorzahlen gute Ergebnisse. Stärkere Restriktionen treffen allerdings auf die parallele LU-Zerlegung (Abschnitt 6.7) zu, welche nur wenige Prozessoren auszunutzen in der Lage ist.

Bedingt durch die Verfügbarkeit des gesamten Speichers für Prozessoren lassen sich zudem Algorithmen verwenden, die sehr eng an ihre sequentiellen Pendants angelehnt sind. Außerdem kann in der Regel ein Online-Scheduling-Verfahren (siehe Abschnitt 5.2.1.1) eingesetzt werden, d.h. es ist keine explizite Lastbalancierung notwendig. Zusammen gelingt es hierdurch, eine bestehende Implementierung der $\mathcal{H}$-Matrix-Arithmetik auf einfache Weise zu modifizieren, um mehrere Prozessoren zu nutzen.

In diesem Zusammenhang wurde auch der große Einfluss der eingesetzten Speicherverwaltung (siehe Kapitel 8) untersucht. Aufgrund des speziellen Aufbaus einer $\mathcal{H}$-Matrix aus einer Vielzahl von kleinen Blöcken, ergeben sich besondere Schwierigkeiten für viele, verbreitete Verwaltungsalgorithmen, wodurch die Komplexität der $\mathcal{H}$-Matrix-Algorithmen insbesondere für große Problemdimensionen leidet. Werden zusätzlich mehrere Prozessoren gleichzeitig genutzt, so resultiert unter Umständen eine Laufzeit, die größer als die des sequentiellen Verfahrens ist. Durch eine speziell angepasste Speicherverwaltung (Abschnitt 8.4) konnten diese Probleme gelöst werden.

Insbesondere für Rechnersysteme mit einer hohen Zahl von CPUs haben sich Maschinen mit einem verteilten Speicher (siehe Abschnitt 5.2.2) etabliert, da der Realisierungsaufwand in diesem Fall deutlich geringer als bei Rechnern mit gemeinsamem Speicher ist. Für diese Systeme wurden Verfahren für den Aufbau, die Matrix-Vektor-Multiplikation und die Addition beschrieben. Gemein ist all diesen Verfahren die Notwendigkeit einer expliziten Lastbalancierung. Wie diese für $\mathcal{H}$-Matrizen vorgenommen werden kann, ist in Kapitel 4

dargestellt. Dabei reicht die Palette von „klassischen“ Verfahren wie dem List-Scheduling (Abschnitt 4.1.1) bis zur Sequenzpartitionierung (Abschnitt 4.1.3), welche allerdings auf raumfüllende Kurven (Abschnitt 4.2.2) angewiesen ist.

Für den Aufbau und die Addition wurden optimal skalierende, parallele Algorithmen vorgestellt, wobei von einem fehlenden Kommunikationsbedarf profitiert werden konnte. Bei der Adaption der blockorientierten, sequentiellen Matrix-Vektor-Multiplikation (Abschnitt 6.3.1) zeigt sich dagegen ein negatives Resultat für den Kommunikationsaufwand, welcher sich durch mehr Prozessoren nicht verringern lässt. Trotz dieses Verhaltens ergeben sich in numerischen Tests annähernd optimale Ergebnisse, falls eine hinreichend große Problemdimension gewählt wird.

Ein alternativer Ansatz für die Matrix-Vektor-Multiplikation (Abschnitt 6.3.2) umgeht diese Schwierigkeit und liefert ein Verfahren, welches sowohl bezüglich der eigentlichen Berechnung, als auch der Kommunikation die gewünschte, optimale Komplexität zeigt. Allerdings spiegeln die praktischen Resultate nicht das theoretische Verhalten wieder, da ein erheblicher zusätzlicher Verwaltungsaufwand notwendig ist. Hier sind weitere Modifikationen notwendig, um die Problemgröße für den Übergang in den asymptotischen und damit effizienten Bereich zu reduzieren.

Desweiteren fehlen für diese Rechnersysteme Algorithmen für die Multiplikation, die Inversion und die LU-Zerlegung einer $\mathcal{H}$-Matrix. Allerdings ist für die letzten beiden Fälle der Einsatz der Gebietszerlegungsmethode (siehe Kapitel 7) möglich, wobei diese hierbei auf Probleme aus dem Bereich der partiellen Differentialgleichungen beschränkt ist. Durch den alternativen Zugang, bei welchem schon das Ausgangsproblem in einzelne Teilprobleme zerlegt wird, ergeben sich unterschiedliche Verfahren, welche bei hinreichend großen Problemen und einer entsprechenden Lastverteilung effiziente $\mathcal{H}$-Matrix-Algorithmen darstellen. Insbesondere für die Inversion (Abschnitt 7.1.1) und die LU-Zerlegung (Abschnitt 7.1.2) erlaubt diese Methode den Einsatz von vielen Prozessoren.

Neben den Differentialgleichungen konnte auch für Integralgleichungen die Matrix-Vektor-Multiplikation mittels Gebietszerlegung parallelisiert werden (Abschnitt 7.2.1), wobei der gleiche alternative Zugang wie schon bei der direkten Parallelisierung Verwendung fand. Das Ergebnis ist auch hier ein hocheffizientes Verfahren.

Literaturverzeichnis

[AF91] S. Anily and A. Federgruen. Structured partitioning problems. *Operations Research*, 13:130–149, 1991.

[Amd67] G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings*, pages 483–485, 1967.

[BDadH95] A. Bäumker, W. Dittrich, and F. Meyer auf der Heide. Truly Efficient Parallel Algorithms: c-Optimal Multisearch for an Extension of the BSP Model. In *Proc. of European Symposium on Algorithms*, pages 17–30, 1995.

[Beb00] M. Bebendorf. Approximation of boundary element matrices. *Numerische Mathematik*, 86:565–589, 2000.

[BH03] M. Bebendorf and W. Hackbusch. Existence of $\mathcal{H}$-Matrix Approximants to the Inverse FE-Matrix of Elliptic Operators with L^∞-Coefficients. *Numerische Mathematik*, 95:1–28, 2003.

[BHP+99] G. Bilardi, K.T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP versus LogP. *Algorithmica*, 24:405–422, 1999. Special Issue on Coarse Grained Parallel Algorithms.

[Bis04] R. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI.* Oxford University Press, 2004.

[BJvOR03] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) Library. *Parallel Computing*, 29(2):187–207, February 2003.

[BL94] R.D. Blumofe and C.E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, Santa Fe, New Mexico, 1994.

[BMBW00] E.D. Berger, K.S. McKinley, R.D. Blumofe, and P.R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, Cambridge, MA, November 2000.

[Bor03] S. Le Borne. $\mathcal{H}$-matrices for convection-diffusion problems with constant convection. *Computing*, 70:261–274, 2003.

[Bra91] A. Brandt. Multilevel computations of integral transforms and particle interactions with oscillatory kernels. *Computer Physics Comm.*, 65:24–38, 1991.

[But97] D.R. Butenhof. *Programming with POSIX Threads*. Addison Wesley, 1997.

[Cam94] D.K.G. Campbell. *CLUMPS: A Candidate Model Of Efficient, General Purpose Parallel Computation*. PhD thesis, University of Exeter, 1994.

[CGJ78] E.G. Coffman, M.R. Garey, and D.S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM J. Comput.*, 7(1):1–17, 1978.

[CH94] J. Clark and D.A. Holton. *Graphentheorie*. Spektrum Akademischer Verlag, Heidelberg, 1994.

[CKP+93] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[CM94] T.F. Chan and T.P. Mathew. Domain decomposition algorithms. *Acta Numerica*, 3:61–143, 1994.

[CR73] S. Cook and R. Reckhow. Time Bounded Random Access Machines. *Journal of Computer and Systems Sciences*, 7:354–375, 1973.

[DER86] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford Science Publications, Oxford, 1986.

[DM98] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1), January/March 1998.

[Fri84] D.K. Friesen. Tighter bounds for the multifit processor scheduling algorithm. *SIAM J. Comput.*, 13(1):170–181, 1984.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM Press, 1978.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[GGKK03] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison Wesley, 2003.

[GH03] L. Grasedyck and W. Hackbusch. Construction and Arithmetics of $\mathcal{H}$-Matrices. *Computing*, 70:295–334, 2003.

[GHK04] L. Grasedyck, W. Hackbusch, and R. Kriemann. Parallel Rank-Adaptive Arithmetics of $\mathcal{H}$-Matrices. Technical report, MPI Leipzig, 2004. Preprint.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[GL96] G.H. Golub and C.F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 3rd edition, 1996.

[Glo] W. Gloger. ptmalloc. http://www.malloc.de.

[GNL98] W. Gropp, B. Nitzberg, and E. Lusk. *MPI: The Complete Reference*. MIT Press, 1998.

[GR97] L. Greengard and V. Rokhlin. A new vetrsion of the fast multipole method for the Laplace in three dimensions. *Acta Numerica*, pages 229–269, 1997. Cambridge University Press.

[Gra69] R.L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[Gra01] L. Grasedyck. *Theorie und Anwendungen Hierarchischer Matrizen*. Dissertation, University of Kiel, 2001.

[GV94] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, 1994.

[Hac85] W. Hackbusch. *Multi-grid methods and applications*. Springer, 1985.

[Hac95] W. Hackbusch. *Integral equations: Theory and numerical treatment*, volume 128 of *ISNM*. Birkhäuser, Basel, 1995.

[Hac97] W. Hackbusch. On the Feedback Vertex Set Problem for a Planar Graph. *Computing*, 58(2):129–155, 1997.

[Hac99] W. Hackbusch. A sparse matrix arithmetic based on $\mathcal{H}$-matrices. I. Introduction to $\mathcal{H}$-matrices. *Computing*, 62(2):89–108, 1999.

[Hac03] W. Hackbusch. Direct Domain Decomposition using the Hierarchical Matrix Technique. In *Proceedings of the 14th International Conference on Domain Decomposition Methods*, pages 38–50, Cocoyoc, Mexico, June 2003.

[Hew02] Hewlett-Packard. *Meet the HP Superdome Servers*, May 2002. White Paper.

[HK00] W. Hackbusch and B.N. Khoromskij. A sparse $\mathcal{H}$-matrix arithmetic: general complexity estimates. *J. Comput. Appl. Math.*, 125:479–501, 2000.

[HKBM01] W. Hackbusch, R. Kriemann, S. Le Borne, and J.-F. Maitre. CD2D3D - a package to solve convection dominated problems employing ordering techniques. *Notes on Numerical Fluid Mechanics*, 75:34–48, 2001.

[HKK03] W. Hackbusch, B.N. Khoromskij, and R. Kriemann. Hierarchical Matrices based on a Weak Admissibility Criterion. Preprint nr. 2/2003, MPI Leipzig, 2003.

[HMS+98] J. Hill, B. McColl, D. Stefanescu, M. Goudreau, K. Lang, S. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, December 1998.

[HN89] W. Hackbusch and Z.P. Nowak. On the Fast Matrix Multiplication in the Boundary Element Method by Panel Clustering. *Numer. Math.*, 54:463–491, 1989.

[Hoc96] D.S. Hochbaum. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, Boston, MA, 1996.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

[JW99] M.S. Johnstone and P.R. Wilson. The Memory Fragmentation Problem: Solved? *ACM SIGPLAN Notices*, 34(3):26–36, 1999.

[KBK03] R. Kriemann, J. Burmeister, and R. Kleinrensing. Benchmarking a Shared Memory System. Technical Report and Documentations 1/2003, Max-Planck-Institute for Mathematics in the Sciences, Leipzig, 2003.

[Knu73] D.E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 1973.

[KR87] V. Kumar and V.N. Rao. Parallel Depth-First Search on Multiprocessors Part II: Analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[Kri03] R. Kriemann. Implementation and Usage of a Thread Pool based on POSIX Threads. Technical Report and Documentations 2/2003, Max-Planck-Institute for Mathematics in the Sciences, Leipzig, 2003.

[Lin02] M. Lintner. *Lösung der 2D Wellengleichung mittels hierarchischer Matrizen*. Dissertation, Technische Universität München, 2002.

[LK99] P.-Å. Larson and M. Krishnan. Memory allocation for long-running server applications. *ACM SIGPLAN Notices*, 34(3):176–185, 1999.

[McC95a] W. McColl. A BSP Realisation of Strassen's Algorithm, 1995.

[McC95b] W. F. McColl. Scalable Computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000, pages 46–61. Springer-Verlag, 1995.

[MMT95] B. M. Maggs, L. R. Matheson, and R. E. Tarjan. Models of parallel computation: a survey and synthesis. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS)*, volume 2, pages 61–70, 1995.

[Mr95] F. Manne and T. Sørevik. Optimal Partitioning of Sequences. *J. Algorithms*, 19(2):235–249, 1995.

[OM95] B. Olstad and F. Manne. Efficient Partitioning of Sequences. *IEEE Transactions on Computers*, 44(11):1322–1326, 1995.

[Pea90] G. Peano. Sur une courbe qui remplit toute une aire plane. *Mathematische Annalen*, (36):157–160, 1890.

[PP84] M.S. Papamarcos and J.H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th annual international symposium on Computer architecture*, pages 348–354. ACM Press, 1984.

[Rob74] J.M. Robson. Bounds for Some Functions Concerning Dynamic Storage Allocation. *Journal of the ACM*, 21(3):491–499, July 1974.

[Rob77] J.M. Robson. Worst case fragmentation of the first fit and best fit storage allocation strategies. *Computer Journal*, 20(3):242–244, 1977.

[Rok85] V. Rokhlin. Rapid solution of integral equations of classical potential theory. *Journal of Computational Physics*, 60:187–207, 1985.

[Sag94] H. Sagan. *Space-Filling Curves*. Springer-Verlag, Berlin, Heidelberg, New York, 1994.

[Sed90] R. Sedgewick. *Algorithms*. Addision Wesley, 2nd edition, 1990.

[SHM97] D.B. Skillicorn, J.M.D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, Fall 1997.

[SS86] Y. Saad and M.H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Comput.*, 7(3):856–869, 1986.

[ST98] D.B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, 1998.

[Sun02] Sun. *Sun Fire^{TM} 3800-6800 Servers – Computing for Lower Total Cost of Ownership (TCO)*, Feb. 2002. Technical White Paper.

[Tar72] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2), 1972. 146–160.

[Tis02] A. Tiskin. Bulk-synchronous parallel Gaussian elimination. *Journal of Mathematical Sciences*, 108(6):977–991, 2002.

[Tur37] A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. In *Proc. Lond. Math. Soc.*, volume 42 of *2*, pages 230–265, 1936-37.

[Val90] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[WJNB95] P.R. Wilson, M.S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), 1995.

[WS93] M.S. Warren and J.K. Salmon. A parallel hashed oct-tree N-body algorithm. In *Supercomputing '93*, pages 12–21, Los Alamitos, 1993. IEEE Comp. Soc.

[Yue91] M. Yue. A simple proof of the inequality $\mathrm{FFD}(l) \leq \frac{11}{9}\mathrm{OPT}(l) + 1 \,\forall\, l$, for the FFD bin-packing algorithm. *Acta Math. App. Sinica*, 7:321–331, 1991.

[Zum03] G. Zumbusch. *Parallel Multilevel Methods.* Advances in Numerical Mathematics. B.G. Teubner Verlag, 2003.

Index