Automatische Parallelisierung und parallele Gitterbasisreduktion

Dissertation
zur Erlangung des Grades
des Doktors der Naturwissenschaften
der Technischen Fakultät
der Universität des Saarlandes

von

Christian Heckler

Saarbrücken
1995

Tag des Kolloquiums:	30. März 1995
Dekan:	Prof. Dr. H. Bley
Vorsitzender:	Prof. Dr. H. Ruppersberg
1. Berichterstatter:	Prof. Dr. L. Thiele
2. Berichterstatter:	Prof. Dr. J. Buchmann

Die vorliegende Dissertation entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Mikroelektronik der Universität des Saarlandes, Saarbrücken.

Die Forschungsarbeiten fanden im Rahmen des Sonderforschungsbereiches (SFB) „VLSI Entwurfsmethoden und Parallelität“ der Deutschen Forschungsgemeinschaft (DFG), Teilprojekte B5 „Algorithmisch spezialisierte integrierte Schaltungen“ und D5 „Parallele und verteilte seminumerische Algorithmen“ statt.

Mein herzlichster Dank gilt Prof. Dr. Lothar Thiele, dem Leiter des Lehrstuhls und Projektleiter der Teilprojekte B5 und D5, der mir die in der Dissertation behandelten Themen und deren Grundlagen nahegebracht hat. Die Zusammenarbeit mit ihm war sehr angenehm und fruchtbar. Viele Diskussionen und Anregungen haben die Arbeit sehr gefördert und in dieser Form erst möglich gemacht. Ein wesentlicher Beweis ist überhaupt erst durch eine wochenlange intensive Zusammenarbeit entstanden.

Meinen Dank möchte ich auch Herrn Prof. Dr. Buchmann aussprechen, ebenfalls Projektleiter des Teilprojektes D5 des SFB, für die Bereitschaft, das Zweitgutachten meiner Dissertation zu übernehmen, und für die Betreuung dreier Diplomanden, die praktische Implementierungen durchgeführt haben.

Den Studenten Patrick Häusser, Stephan Kann, Andreas Kratz, Franz Muller und Kay Wiese, die im Rahmen des Projektes ihre Diplom- bzw. Studienarbeiten angefertigt haben, sei für die gute Zusammenarbeit gedankt.

Außerdem möchte ich mich bei dem „Labor für Paralleles Rechnen“ der Gesellschaft für Mathematik und Datenverarbeitung, Sankt Augustin, für die Möglichkeit bedanken, auf den Parallelrechnern CM-5 und CM-2 der Firma Thinking Machines zu rechnen, und bei den Systemverwaltern für die dabei geleistete, sehr gute und zuvorkommende Unterstützung. Nur so war es möglich, die theoretischen Ergebnisse auch praktisch umzusetzen und zu testen.

Meine Zeit am Lehrstuhl für Mikroelektronik war und ist eine der angenehmsten meines Lebens. Dafür danke ich allen Personen des Lehrstuhls. Umso trauriger ist es, daß der Kapitän das (nicht sinkende, sondern im Gegenteil sehr gut im Wind stehende) Schiff verlassen hat – ein menschlicher und wissenschaftlicher Verlust.

Schließlich danke ich meiner Partnerin Frau Diplom-Physikerin Christine Maier.

Meinen Eltern, die dies alles erst ermöglicht haben

Inhaltsverzeichnis

1 Einleitung **1**

2 Lineare Datenabhängigkeiten **3**

2.1 Einleitung und Literaturvergleich 3

2.2 Problemstellung 5

2.3 Struktur des Problems 10

2.3.1 Die lexikographische Ordnung 10

2.3.2 Struktur der Lösungen eines Unterfalles 12

2.4 Ein Spezialfall 15

2.4.1 Lösung im Spezialfall 15

2.4.2 Beispiele 17

2.4.3 Geltung des Spezialfalles 21

2.4.4 Anzahl der verschiedenen linearen Abhängigkeiten 22

2.5 Erweiterungen 23

2.5.1 Teilweise überlappende Schleifen 23

2.5.2 Das Problem der Überdeckung 24

2.5.3 Beispiele 25

2.6 Zusammenfassung 28

3 Parallele Gitterbasisreduktion **30**

3.1 Einleitung 30

3.2 Grundlagen der Basisreduktion in Gittern 32

3.2.1 Gitter und Basen 32

3.2.2 Reduzierte Basen 34

3.2.3 Allgemeine Struktur von Basisreduktionsalgorithmen 37

3.3 Das verwendete parallele Programmiermodell 42

3.4 Parallelisierung des Schnorr/Euchner Algorithmus 43

3.4.1 Parallele Berechnung der k-ten Spalte der Orthogonalisierung . . . 44

3.4.2 Parallele Größenreduktion der k-ten Spalte der μ-Matrix 46

3.5 Eine neue Variante des LLL - Reduktionsalgorithmus 49
3.5.1 Ein gut parallelisierbarer Algorithmus 49
3.5.2 Zeitkomplexität von Alltauschalgorithmen 50
3.5.3 Verwendung von Gleitkommaarithmetik 55
3.5.4 Fehleranalyse 58
3.5.5 Praktische Ergebnisse 65
3.6 Parallelisierung für n^2 Prozessoren 67
3.7 Parallelisierung für n Prozessoren 72
3.7.1 Parallelisierung von Alltauschalgorithmen 73
3.8 Größe der entstehenden Zahlen und binäre Komplexität 76
3.8.1 Abschätzung der Zahlengröße 76
3.8.2 Genauere Analyse der Zahlengröße bei der parallelen Größenreduktion 80
3.8.3 Binäre Komplexität 83
3.8.4 Diskussion 85
3.8.5 Die Methode von Joux zur Vermeidung großer Zahlen bei der parallelen Größenreduktion 86
3.9 Parallele Implementierungen und praktische Ergebnisse 88
3.9.1 Einleitung und Definitionen 88
3.9.2 Parallele Programmiermodelle 90
3.9.3 Ein Beispiel: Parallele Givens-Rotationen 92
3.9.4 Implementierung auf einem Workstationnetz mit LiPS 99
3.9.5 Implementierungen auf MIMD-Rechnern 103
3.9.6 Implementierung auf einem SIMD-Rechner 112
3.9.7 Überblick über die diskutierten Implementierungen 114
3.10 Zusammenfassung 115

A Unimodulare Matrizen **118**

B Lösungen ganzzahliger Gleichungssysteme **119**

C Das Rucksackproblem **121**

D Symbol- und Abkürzungsverzeichnis **122**

Tabellenverzeichnis

3.1 Parallele Größenreduktion einer Spalte mit Pipelining 48
3.2 Neuer serieller Algorithmus – unimodular transformierte Gitter 64
3.3 Schnorr/Euchner-Algorithmus – unimodular transformierte Gitter 64
3.4 Neuer serieller Algorithmus – „Rucksack-Gitter“ 64
3.5 Schnorr/Euchner-Algorithmus – „Rucksack-Gitter“ 64
3.6 Parallele Verknüpfungen 68
3.7 Parallele Givensrotationen auf der CM-2 – Effizienz 97
3.8 Parallele Givensrotationen auf der CM-2 – Wachtumsrate 98
3.9 Parallele Givensrotationen auf der CM-5 – Effizienz 98
3.10 LiPS-Algorithmus für „Rucksackgitter“ der Dimension 80 102
3.11 LiPS-Algorithmus für „Rucksackgitter“ der Dimension 100 102
3.12 LiPS-Algorithmus für unimodular transformierte Gitter der Dimension 80 102
3.13 LiPS-Algorithmus für unimodular transformierte Gitter der Dimension 100 102
3.14 Eine Phase mit paralleler Größenreduktion, CM-5 – unimodular 104
3.15 Parallele Größenreduktion, CM-5 – „Rucksack-Gitter“, 340 Binärstellen . . 104
3.16 Parallele Größenreduktion, CM-5 – „Rucksack-Gitter“, 510 Binärstellen . . 105
3.17 Eine Phase mit paralleler Größenreduktion, Paragon – unimodular 106
3.18 Parallele Größenreduktion, Paragon – „Rucksack-Gitter“, 510 Binärstellen 106
3.19 Eine parallele Phase, CM-5 – unimodular transformierte Gitter 107
3.20 Parallele Basisreduktion, CM-5 – „Rucksack-Gitter“, 510 Binärstellen . . . 108
3.21 Anzahl der Nullen bei Givens-Rotationen bei „Rucksackgittern“ 110
3.22 Eine parallele Phase, CM-2, n^2 Prozessoren, Zahlenlänge 160 113
3.23 Eine parallele Phase, CM-2, $8k$ und $16k$ Prozessoren, unimodular 113
3.24 Eine parallele Phase, CM-2, $8k$ und $16k$ Prozessoren, „Rucksackgitter“ . . 113
3.25 Parallele Reduktion, CM-2, n^2 Prozessoren, „Rucksackgitter“ 114
3.26 Überblick über die diskutierten Implementierungen 115

Abbildungsverzeichnis

2.1 Graphische Veranschaulichung zu Beispiel 2.1 (Iterationsraum) 7
2.2 Graphische Veranschaulichung zu Beispiel 2.3 (Datenabhängigkeiten) . . . 9

3.1 Parallelisierung der Größenreduktion der Elemente der Matrix M^T 71
3.2 Dateneingabe bei SIMD-Algorithmus für Givens-Rotationen (vgl. [21]) . . 95

Kapitel 1

Einleitung

Viele interessante Probleme der Computeralgebra benötigen zu ihrer Lösung eine lange Rechenzeit, da exakte oder symbolische Berechnungen durchgeführt werden müssen. Um solche Probleme trotzdem „schnell“ zu lösen, betrachtet man zunehmend parallele Programme, d.h. Programme, bei denen verschiedene Berechnungen gleichzeitig durchgeführt werden.

Grundsätzliches Problem bei der Erstellung eines parallelen Programms ist (vereinfacht gesprochen), Operationen zu finden, die unabhängig voneinander sind und somit parallel ausgeführt werden können. Dies bildet einerseits die Grundlage, automatisch aus sequentiell formulierten Programmen parallele zu generieren (s. z.B. [18]), und andererseits, automatisch Programme auf Schaltungen abzubilden (s. z.B. [10]).

Ein neues Verfahren zur Berechnung solcher Datenabhängigkeiten, das effizienter als bisher bekannte ist, wird in Kapitel 2 vorgestellt.

Der Vorteil einer automatischen Parallelisierung besteht darin, schnell parallele Programme zu erhalten. Zum einen stehen schon sehr viele sequentiell formulierte Programme in Programmbibliotheken zur Verfügung (z.B. numerische Programmpakete). Diese Programme sollten nicht alle neu entwickelt und implementiert werden müssen. Zum anderen ist die Formulierung eines sequentiellen Programms in einer bekannten, sequentiellen Programmiersprache einfacher und geht somit schneller.

Aber natürlich sind die Möglichkeiten der automatischen Parallelisierung begrenzt. Ein automatischer Parallelisierer erkennt in der Regel nur, welche Operationen parallel ausgeführt werden können. Die prinzipielle Struktur des Programms bleibt erhalten. Es ist aber möglich, daß ein anderer Algorithmus zur Lösung des Problems mehr parallel ausführbare Operationen aufweist.

Daher ist es oft sinnvoll, für ein gegebenes Problem spezielle Algorithmen zu entwerfen, die sich gut für eine parallele Ausführung eignen. Wenn ein bestimmter Algorithmus nicht gut parallelisierbar ist, heißt das noch nicht, daß eine effiziente parallele Lösung des Problems nicht möglich ist. Es ist auch sinnvoll, sich Gedanken über die parallele Komplexität eines Problems zu machen, d.h. zu untersuchen, wie schnell ein Problem parallel gelöst werden kann.

Unter diesen Gesichtspunkten wird in Kapitel 3 das interessante Problem der Gitterbasisreduktion, d.h. eine Basis eines Gitters mit „kurzen“ Vektoren zu finden, betrachtet. Die Gitterbasisreduktion hat viele wichtige Anwendungen (s. Kapitel 3).

Bekannte Verfahren erweisen sich als schlecht parallelisierbar. Daher wird eine Variante entwickelt, die besser für eine Parallelisierung geeignet ist, und für die die Verfahren aus Kapitel 2 prinzipiell anwendbar sind. Die parallele Komplexität der Gitterbasisreduktion wird betrachtet, und parallele Algorithmen, die auf verschiedenen Rechnerarchitekturen implementiert und getestet wurden, werden angegeben.

Kapitel 2

Bestimmung von linearen Datenabhängigkeiten in Schleifenprogrammen

2.1 Einleitung und Literaturvergleich

Dieses Kapitel beschäftigt sich mit der automatischen Parallelisierung sequentieller Programme. Ausgangspunkt sollen hierbei Schleifenprogramme sein, d.h. einfache, imperative Programme mit folgenden Einschränkungen:

- Als Datentypen werden nur Felder betrachtet (wobei Skalare degenerierte Felder sind).
- Als einzige Kontrollstruktur stehen for-Schleifen zur Verfügung (auch verschachtelt) mit der Schrittweite $+1$.[1]
- Die Schleifengrenzen sind affine Funktionen in den Schleifenvariablen der umschließenden Schleifen.
- Die Indexfunktionen der Feldzugriffe sind affine Terme in den Schleifenvariablen.

Zum einen liegen viele Programme z.B. aus der Numerik (RU-Zerlegung, SVD ...) und aus anderen wichtigen Anwendungsgebieten in dieser Form vor oder können in diese Form gebracht werden. Zum anderen ist die Behandlung bei allgemeineren Programmformen (z.B. while-Schleifen, nichtlineare Indexfunktionen) sehr viel schwieriger, wenn nicht gar unmöglich (s. [18]).

Ein wichtiges Problem bei der Parallelisierung solcher Algorithmen ist die Bestimmung von Datenabhängigkeiten. Dabei sind zwei Anweisungen abhängig voneinander, wenn beide auf dieselbe Speicherzelle zugreifen. Es liegt nahe, daß zwei solche Anweisungen nicht gleichzeitig durchgeführt werden können.

Viele Compilerbautechniken zur Parallelisierung von Schleifen beschäftigen sich mit der Bestimmung unabhängiger Anweisungen, um diese parallel ausführen zu können, bzw.

[1] Die Einschränkung der Schrittweite ist dabei nicht essentiell.

mit (evtl. nur hinreichenden) Abhängigkeitstests oder benutzen heuristische und ad hoc Verfahren zur Bestimmung von Abhängigkeiten und Unabhängigkeiten. Ein guter Literaturüberblick dazu wird in [9] angegeben (s. auch [1, 3, 7, 47, 76]).

Das Ziel der vorliegenden Arbeit ist hingegegen die exakte Berechnung der Abhängigkeiten durch Angabe von Bereichen des Iterationsraums, in denen eine feste lineare Datenabhängigkeit besteht. Aus dieser Darstellung läßt sich einfach ein semantisch äquivalenter „single assignment“ - Algorithmus ableiten, der als paralleles Programm betrachtet werden kann (s. z.B. [18]). Außerdem kann ein solcher „single assignment“ - Algorithmus als Eingabe für Verfahren zur automatischen Abbildung auf eine Schaltung dienen (s. [10]).

Systematische Verfahren zur Bestimmung von Datenabhängigkeitsvektoren werden in [49, 9, 33, 18, 39] betrachtet.

Kuhn berechnet in [33] Abhängigkeiten in Form von Relationen, die mit Hilfe konvexer Mengen dargestellt werden, berechnet die Abhängigkeitsvektoren aber nicht explizit.

In [9] wird die Struktur des entstehenden Abhängigkeitsgraphen genau untersucht. Es wird dabei davon ausgegangen, daß alle Instruktionen alle Schleifen gemeinsam haben, d.h. das Problem von nur teilweise verschachtelten Schleifen wird nicht betrachtet. Auch dort wird kein Verfahren zur exakten Berechnung der Abhängigkeitsvektoren angegeben. In Hinsicht auf eine Einbettung des Abhängigkeitsgraphen in ein „systolisches array“ wird dann untersucht, wie man konstante Abhängigkeitsvektoren erreichen kann (*Lokalisierung*).

In [49] wird eine exakte (aber nicht ganz korrekte) mathematische Formulierung zur Bestimmung eines Abhängigkeitsvektors zu einem Iterationspunkt angegeben. Diese Formulierung benutzt die Notation des Ordnungsvektors, um die Iterationspunkte in einer Reihenfolge anzuordnen. Bei dieser Formulierung müßte für jeden Iterationspunkt ein ganzzahliges lineares Programm gelöst werden, was aber in der Praxis zu aufwendig ist. Außerdem werden das Problem teilweise verschachtelter Schleifen und das Problem der Überdeckung nicht betrachtet, d.h. daß mehrere Anweisungen verändernd auf dasselbe Feld zugreifen und bestimmt werden muß, welcher Zugriff aktuell der letzte war. Es wird untersucht, wie man ein gegebenes Schleifenprogramm umformen kann, so daß konstante Abhängigkeitsvektoren entstehen.

In den beiden letztgenannten Arbeiten werden also konstante Abhängigkeitsvektoren betrachtet, während hier Abhängigkeitsvektoren untersucht werden, die linear vom betrachteten Iterationspunkt abhängen.

In der vorliegenden Arbeit wird eine Notation ähnlich der von [49] benutzt und von einer modifizierten Form der dortigen Problemformulierung ausgegangen, wobei sich die Verwendung von Ordnungsvektoren als nicht geeignet erwiesen hat.

Feautrier [17, 18] berechnet die linearen Datenabhängigkeiten mit einer von ihm entwickelten Methode zur Lösung parametrisierter ganzzahliger linearer Programme [16], was jedoch aufwendig und nicht ganz einfach zu implementieren ist. Es scheint, daß für praktische Anwendungen schnellere Methoden nötig sind [39].

In der vorliegenden Arbeit wird die Struktur des Problems genauer untersucht, wodurch eine Formulierung mit weniger Ungleichungen und Variablen erreicht wird und eine einfachere Lösung in Spezialfällen, die in der Praxis jedoch relativ häufig sind, möglich ist.

Die Lösung der Spezialfälle ist einfacher zu verstehen und es sind dabei insbesondere keine ganzzahligen linearen Programme zu lösen. In diesen Spezialfällen ist auch eine einfache Abschätzung der maximalen Anzahl der verschiedenen linearen Abhängigkeiten möglich. Die grundlegende Idee ist dabei, mit Hilfe der „Smith Normalform“ die Menge der ganzzahligen Lösungen eines ganzzahligen Gleichungssystemes zu bestimmen.

Unabhängig wurde ein ähnlicher Ansatz in [39] entwickelt, der ebenfalls auf der Lösung ganzzahliger linearer Gleichungssysteme beruht. Die Grundidee dort ist jedoch, zunächst sogenannte „unbenutzte Variable“ zu entfernen, d.h. Variable, die weder in der Indexfunktion des schreibenden Zugriffs auf das Feld noch in den Schleifengrenzen vorkommen. Dann wird das ganzzahlige Gleichungssystem gelöst. Ist diese Lösung eindeutig, wird daraus die Lösung des ursprünglichen Systems rekonstruiert. Im Gegensatz dazu werden in der vorliegenden Arbeit keine Variablen entfernt. In bestimmten Fällen kann jedoch eine Lösung gefunden werden, auch wenn die Lösung des ganzzahligen Gleichungssystems nicht eindeutig ist. Andernfalls kann die Methode von Feautrier angewendet werden. Damit ist das hier vorgestellte Verfahren allgemeiner.

Bei praktischen Anwendungen können evtl. Kombinationen der Methoden verwendet werden. Darauf wird an den entsprechenden Stellen eingegangen.

Das weitere Kapitel ist folgendermaßen gegliedert: Im nächsten Abschnitt wird das zu lösende Problem eingeführt und formal dargestellt. In Abschnitt 2.3 wird die Struktur des Problems genauer untersucht, indem u.a. das Lösen ganzzahliger linearer Gleichungssysteme betrachtet wird. Diese Struktur wird in Abschnitt 2.4 benutzt, um eine einfache Lösung in einem Spezialfall abzuleiten. Zur Verdeutlichung werden zwei Beispiele betrachtet. Außerdem wird untersucht, wann der günstige Spezialfall eintritt und wieviele verschiedene lineare Abhängigkeiten unter der Annahme des Spezialfalls auftreten können. In Abschnitt 2.5 werden wichtige Erweiterungen behandelt. Dies ist zum einen der Fall, daß die Schleifen nicht total ineinander verschachtelt sind und zum anderen, daß es mehrere Anweisungen gibt, die auf das Feld schreibend zugreifen. Ein weiteres Beispiel wird vorgestellt.

Weitere Beispiele, eine Ausarbeitung und Implementierungsdetails sind in der Studienarbeit von Stephan Kann beschrieben [30].

2.2 Problemstellung

Zunächst werden einfache Schleifenprogramme der folgenden Form betrachtet. Wichtige Erweiterungen werden in Abschnitt 2.5 behandelt.

$$
\begin{array}{l}
\text{for } i_1 = c_u \ldots c_o \text{ do} \\
\quad \text{for } i_2 = L_2^-(i_1) \ldots L_2^+(i_1) \text{ do} \\
\quad\quad \vdots \\
\quad\quad\quad \text{for } i_d = L_d^-(i_1, \ldots, i_{d-1}) \ldots L_d^+(i_1, \ldots, i_{d-1}) \text{ do} \\
\quad\quad\quad\quad \text{(s1)} \qquad \ldots = \ldots v[f_1^U(i_1, \ldots, i_d), \ldots, f_m^U(i_1, \ldots, i_d)] \ldots \\
\quad\quad\quad\quad \text{(s2)} \qquad v[f_1^M(i_1, \ldots, i_d), \ldots, f_m^M(i_1, \ldots, i_d)] = \ldots
\end{array}
$$

Es seien d Schleifen ineinander verschachtelt. Die oberen und unteren Schranken des Laufindex i_j der j-ten Schleife seien gegeben durch affine Funktionen L_j^+ bzw. L_j^- in den Laufvariablen $i_1, \ldots i_{j-1}$ der sie umgebenden Schleifen. Die Grenzen für den Index i_1 der äußersten Schleife seien Konstanten c_u, c_o. Im Schleifenrumpf werde ein m-dimensionales Feld v benutzt[2], zuerst lesend, dann schreibend. Die Zugriffe auf die einzelnen Dimensionen des Feldes seien wieder affine Funktionen in den Laufvariablen $i_1, \ldots i_d$ der umgebenden Schleifen.

Beispiel 2.1

for $i = 1, \ldots, 7$ do
  for $j = 1, \ldots, i$ do
    $\ldots = \ldots v[i, i + 2j + 1] \ldots$
    $v[i + j, 2i + 2j] = \ldots$

Eine Ausführung des Schleifenrumpfes ist eindeutig definiert durch die Werte, die die Laufvariablen zur Zeit der Ausführung besitzen.

Definition 2.1 (s. [9]) *Einen Punkt $I = (i_1, \ldots, i_d)^T \in \mathbf{Z}^d$ von Werten für die Laufvariablen, der bei der Ausführung der Schleife durchlaufen wird, nennt man Iterationspunkt, die Menge aller durchlaufenen Punkte* $\mathbf{I}$ *Iterationsraum.*

Der Iterationsraum für das obige Beispiel ist in Abbildung 2.1 graphisch dargestellt. Die Pfeile geben dabei die Reihenfolge an, in der die Iterationspunkte durchlaufen werden. Diese Reihenfolge ist durch das Schleifenprogramm eindeutig definiert. Ein Iterationspunkt x wird vor einem Iterationspunkt y durchlaufen, wenn für die erste Komponente i, auf der x und y nicht übereinstimmen, $x_i < y_i$ gilt. Daher ist die Ordnung auf dem Iterationsraum $\mathbf{I}$ durch die lexikographische Ordnung gegeben (s. auch Abschnitt 2.3.1). Da die Semantik obiger Schleifenprogramme hier nicht formal spezifiziert wurde, soll diese Tatsache als Definition angegeben werden.

Definition 2.2 (s. [18]) *Ein Iterationspunkt $x \in \mathbf{Z}^d$ wird genau dann vor einem Iterationspunkt $y \in \mathbf{Z}^d$ durchlaufen, wenn x lexikographisch kleiner als y ist ($x <_{lex} y$). Dabei ist x lexikographisch kleiner als y genau dann, wenn die erste Komponente von x, für die x nicht mit y übereinstimmt, kleiner als die entsprechende Komponente von y ist.*

Ein Schleifenprogramm dieser Art kann algebraisch schön formuliert werden [49, 18]:

Der Iterationsraum kann als Polytop betrachtet werden, das durch zwei untere Dreiecksmatrizen $A^{(1)}, A^{(2)} \in \mathbf{Z}^{d \times d}$ und zwei Vektoren $b^{(1)}, b^{(2)} \in \mathbf{Z}^d$ definiert ist.

$$\mathbf{I} = \{I \in \mathbf{Z}^d : A^{(1)} I \leq b^{(1)} \wedge A^{(2)} I \leq b^{(2)}\} \tag{2.1}$$

Dabei gilt $A^{(1)} I \leq b^{(1)}$ genau dann, wenn der Iterationspunkt die oberen Schranken der Laufvariablen erfüllt, und $A^{(2)} I \leq b^{(2)}$ genau dann, wenn die unteren Schranken erfüllt werden.

[2] Es soll angenommen werden, daß $m \neq 0$. Der Fall $m = 0$, d.h. skalare, kann ähnlich behandelt werden.

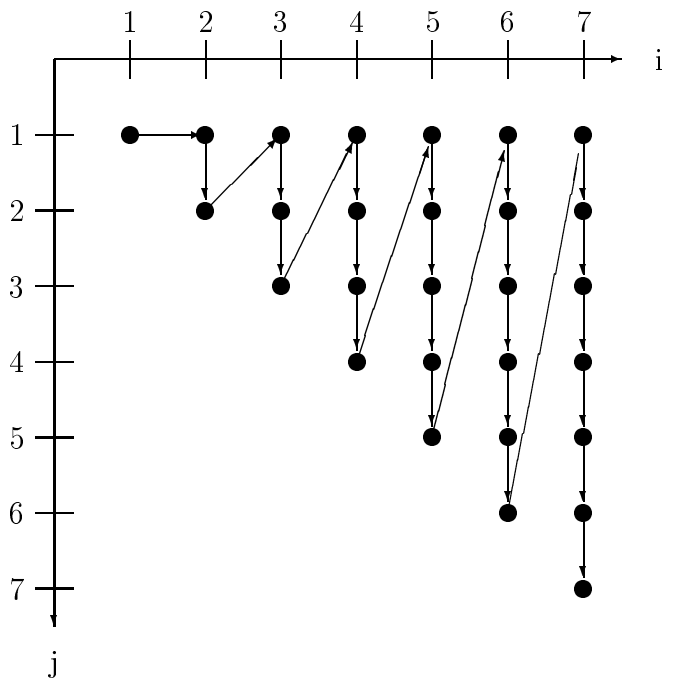

Abbildung 2.1: Graphische Veranschaulichung zu Beispiel 2.1 (Iterationsraum)

Eine affine Funktion zum Zugriff auf das Feld v kann mit Hilfe einer Matrix $M \in \mathbf{Z}^{m \times d}$ und eines Vektors $\mu \in \mathbf{Z}^m$ beschrieben werden, wobei m die Dimension des Feldes v ist, und d die Dimension des Iterationsraumes, d.h. die Anzahl der verschachtelten Schleifen, ist. Daher kann der Rumpf der Schleife auch folgendermaßen geschrieben werden [49].

$$\begin{aligned} \ldots &= \ldots v[UI + v] \ldots \\ v[MI + \mu] &= \ldots \end{aligned}$$

M, μ stehen dabei für *modifizierend* und entsprechend U, v für *used*.

Beispiel 2.2 [Fortführung von Beispiel 2.1] Die algebraische Darstellung des in Beispiel 2.1 angegebenen Schleifenprogramms ist die folgende:

$$\begin{aligned} d &= 2 \\ m &= 2 \\ M &= \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix} \\ U &= \begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix} \\ \mu = &= \begin{pmatrix} 0 \\ 0 \end{pmatrix} \end{aligned}$$

$$
\begin{aligned}
\upsilon = &= \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\
A^{(1)} &= \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \\
b^{(1)} &= \begin{pmatrix} 7 \\ 0 \end{pmatrix} \\
A^{(2)} &= \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \\
b^{(2)} &= \begin{pmatrix} -1 \\ -1 \end{pmatrix}
\end{aligned}
$$

Ziel der Datenabhängigkeitsanalyse ist das folgende: Betrachte einen Iterationspunkt $y \in \mathbf{I}$. Bei der Durchführung der Berechnungen für diesen Iterationspunkt y wird auf das Element $v[Uy + \upsilon]$ benutzend zugegriffen. Die Frage ist, wann, d.h. in welchem Iterationspunkt x, der entsprechende Wert berechnet wurde, d.h. gesucht ist der Iterationspunkt x, der vor y liegt, in dem auf das Feldelement generierend zugegriffen wird, das in y benutzt wird, und in dem dies zum letzten mal geschah. Man nennt dann y abhängig von x. Formal wird eine Abhängigkeit folgendermaßen definiert [49, 15, 18]:

Definition 2.3 *Ein Iterationspunkt $y \in \mathbf{I}$ ist abhängig von einem Iteratonspunkt $x \in \mathbf{I}$ genau dann, wenn die folgenden Bedingungen gelten:*

- $Uy + \upsilon = Mx + \mu$
- $x <_{lex} y$
- *Es gibt keinen Iterationspunkt $z \in \mathbf{I}$ mit $Uy + \upsilon = Mz + \mu$ und $x <_{lex} z <_{lex} y$*

Beispiel 2.3 [Fortführung von Beispiel 2.1] Sei in dem obigen Beispiel der betrachtete Iterationspunkt $y = (7, 3)^T$. In diesem Punkt wird auf das Element $v[7, 14]$ benutzend zugegriffen. In Abbildung 2.2 ist die Menge aller Iterationspunkte, in denen auf dieses Feldelement generierend zugegriffen wird, mit Hilfe einer Geraden eingezeichnet. Der gesuchte Iterationspunkt x ist der Punkt des Iterationsraumes auf der Geraden, der am nächsten an y liegt. In dem Beispiel ist dies der Punkt $x = (6, 1)^T$.

Für einen gegebenen Punkt $y \in \mathbf{I}$ läßt sich der gesuchte Punkt $x(y)$ als Lösung des folgenden Optimierungsproblems bestimmen:

Finde das lexikographisch maximale $x(y) \in \mathbf{Z}^d$ mit

- $Mx(y) = Uy + \upsilon - \mu$
- $x(y) <_{\text{lex}} y$
- $x(y) \in \mathbf{I}$

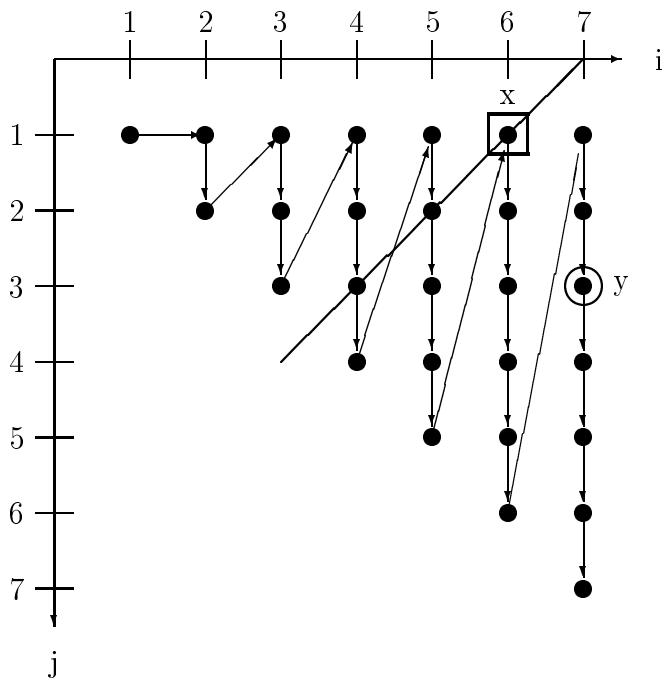

Abbildung 2.2: Graphische Veranschaulichung zu Beispiel 2.3 (Datenabhängigkeiten)

Ribas [49] beschreibt die lexikographische Ordnung mit Hilfe eines „Ordnungsvektors“. Das Optimierungsproblem bildet dann ein ganzzahliges lineares Programm. Wenn y nicht vorgegeben ist, entsteht ein parametrisiertes, ganzzahliges lineares Programm mit y als Parameter. Dieses könnte mit den Methoden aus [16] gelöst werden. Der Nachteil ist jedoch, daß dieses Verfahren aufwendig und in vielen Fällen nicht notwendig ist. Außerdem entstehen durch die Verwendung eines Ordnungsvektors relativ große Zahlen. Auch ist dieses Verfahren nicht anwendbar, wenn bei den Schleifengrenzen als konstante Terme nicht nur Zahlen, sondern sog. Programmparameter vorkommen, d.h. Werte, die erst im Programm gesetzt werden, aber während der Schleifenausführung konstant sind. Bei Matrizenrechnungen werden Schleifen z.B. oft die Form haben

for $i = 1, \ldots, n$

wobei n die Dimension der Matrix ist. Die hier angegebenen Methoden können leicht auf diesen Fall erweitert werden. Dies kommt sogar implizit in den hier vorgestellten Beispielen vor.

Die Problemstellung, die hier untersucht werden soll, ist nicht, die Abhängigkeit für einzelne Punkte zu berechnen, sondern Bereiche des Iterationsraumes, in denen die gleiche lineare Abhängigkeit für x von y besteht, d.h. Abhängigkeiten der Form $x(y) = Cy + \gamma$.

Beispiel 2.4 [Lineare Datenabhängigkeiten [62]] Das folgende Beispiel ist [62] (s. auch [30]) entnommen und zeigt die Aufspaltung des Iterationsraumes in zwei Bereiche mit zwei verschiedenen linearen Datenabhängigkeiten.

```
for i_1 = 0 ... n do
    for i_2 = 0 ... n do
        ... = ... v[0] ...
        v[i_1 - i_2] = ...
```

Berechnet man hier die linearen Abhängigkeiten, so gilt[3]:

$$\begin{aligned} x(y) &= \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} y \qquad \text{für } y \text{ mit} \quad (1,-1)y \leq -1 \\ x(y) &= \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} y + \begin{pmatrix} -1 \\ -1 \end{pmatrix} \qquad \text{für } y \text{ mit} \quad (-1,1)y \leq 0 \end{aligned}$$

Notation: Da bei den folgenden Rechnungen oft auf einzelne Komponenten eines Vektors und auf einzelne Spalten einer Matrix zugegriffen werden muß, wird zur Abkürzung folgende Notation eingeführt:

$$v_{(i\ldots j)}$$

für einen Vektor v bezeichnet einen Vektor, der aus den Komponenten $i, \ldots, j$ von v besteht. Analog beschreibt für eine Matrix M

$$M_{(i\ldots j)}$$

eine Matrix, die aus den Spalten $i, \ldots, j$ von M besteht.

2.3 Struktur des Problems

Es soll nun die Struktur des obigen Optimierungsproblems näher untersucht werden. Diese wird ausgenutzt, um (in manchen Fällen) eine einfache Lösung des Problems zu finden. Zunächst wird betrachtet, was lexikographische Ordnung bedeutet und wie sich dies auf die Lösung des Problems auswirkt. Dann wird angegeben, wie man die Struktur der möglichen Lösungen des ganzzahligen Gleichungssystems berechnen kann.

2.3.1 Die lexikographische Ordnung

Ein Iterationspunkt x ist lexikographisch kleiner als ein Punkt y, d.h. x wird bei der Schleifenausführung vor y durchlaufen, wenn einer der folgenden Fälle gilt (s. Definition 2.2):

- $x_1 < y_1$
- $x_1 = y_1$, $x_2 < y_2$
- $x_1 = y_1$, $x_2 = y_2$, $x_3 < y_3$

[3]Es ist eine gute Übung, dies für einzelne Punkte nachzurechnen!

- etc.

Daher müssen folgende, sogenannte „Unterfälle“ untersucht werden.

- $x_1 = y_1, \ldots, x_{d-1} = y_{d-1}, x_d < y_d$. Finde den maximalen Wert für x_d, so daß $x \in \mathbf{I}$.
- $x_1 = y_1, \ldots, x_{d-2} = y_{d-2}, x_{d-1} < y_{d-1}$. Finde das lexikographische Maximum für $x_{(d-1\ldots d)}$, so daß $x \in \mathbf{I}$.

  $\vdots$
- $x_1 = y_1, \ldots, x_{k-1} = y_{k-1}, x_k < y_k$. Finde das lexikographische Maximum für $x_{(k\ldots d)}$, so daß $x \in \mathbf{I}$.

  $\vdots$
- $x_1 < y_1$. Finde das lexikographische Maximum für x so daß $x \in \mathbf{I}$.

Diese Unterfälle können getrennt voneinander betrachtet werden. Dabei wird die oben angegebene Reihenfolge gewählt, da der Punkt x bestimmt werden soll, der y am nächsten liegt. Ein Iterationspunkt x liegt aber umso näher an dem Iterationspunkt y, je mehr Anfangskomponenten von x und y übereinstimmen. D.h. zunächst wird für jeden Unterfall das optimale x (als lineare Abhängigkeit von y) berechnet. Für ein gegebenes y muß dann die Lösung des Unterfalls gewählt werden, der als erstes eine gültige Lösung x besitzt. Je nach y können dies verschiedene Unterfälle sein.

Für jeden Unterfall entspricht das Finden des lexikographisch maximalen x dem Lösen eines ganzzahligen linearen Programms, das den Iterationspunkt y als Parameter enthält. Die zu beachtenden linearen Ungleichungen (bzw. Gleichungen) sind dabei die folgenden:

- Die ersten Komponenten von x sind mit denen von y gleichgesetzt. Die nächste Komponente von x ist echt kleiner als die entsprechende von y.
- Das Gleichungssystem $Mx = Uy + \nu - \mu$ muß erfüllt werden.
- Die Schleifengrenzenbedingungen $x \in \mathbf{I}$ müssen erfüllt sein.

Statt einer linearen Kostenfunktion soll der Zielvektor x lexikographisch maximiert werden. Feautrier [18] löst diese Optimierungsprobleme mit seinem Verfahren aus [16].

Dieses Vorgehen ist sehr allgemein und funktioniert für alle Programme, die der vorausgesetzten Form genügen. Das Lösen ganzzahliger linearer Programme ist aber relativ rechenintensiv und auch nicht einfach zu verstehen. Außerdem zeigen Untersuchungen [66], daß die Zahlengrößen im Verfahren nach [16] schon für sehr kleine Probleminstanzen problematisch groß werden und in der Praxis zum Abbrechen des Programms führen. Eine genauere Untersuchung zeigt, daß nicht immer das Lösen ganzzahliger parametrisierter Programme erforderlich ist. Die grundlegende Idee ist, daß der gesuchte Iterationspunkt x oft schon durch das Gleichungssystem, das er erfüllen muß, um auf das durch y spezifizierte Feldelement zuzugreifen, nahezu eindeutig definiert ist. Unabhängig von der vorliegenden Arbeit wurde diese Idee auch in [39] angewendet. Mit Hilfe der Smith-Normalform wird dort die Menge der Lösungen für x berechnet (dies wird im nächsten Abschnitt

vorgestellt). Ergibt dies zusammen mit den Schleifengrenzen, die x erfüllen muß, eine eindeutige Lösung, so hat man den gesuchten Punkt gefunden, ohne ein Optimierungsproblem lösen zu müssen. Dabei wird die Tatsache benutzt, daß bei einem Feldzugriff und bei der Definition der Schleifengrenzen viele Schleifenvariablen nicht benutzt werden. Zuerst werden diese entfernt, um auf den oben genannten Spezialfall zu kommen. Aus der dann gewonnenen Lösung wird die korrekte Lösung berechnet, indem die entfernten Schleifenvariablen wieder eingerechnet werden.

Im Gegensatz dazu ist die hier vorgestellte Methode etwas allgemeiner, da angegeben wird, wie man in gewissen Fällen eine Lösung finden kann, auch wenn die Lösung des ganzzahligen Gleichungssystems nicht eindeutig ist. Es werden keine Variablen entfernt. Es wird direkt die Menge der möglichen Lösungen berechnet und in die Schleifengrenzenbedingungen eingesetzt. Dadurch erhält man ein einfacheres System mit weniger Variablen und Ungleichungen als das ursprüngliche System. Es wird eine sehr einfache Bedingung angegeben, wann man das gesuchte x leicht berechnen kann, ohne parametrisierte ganzzahlige lineare Programme zu lösen. Falls dies nicht möglich ist, kann die Methode von Feautrier benutzt werden.

2.3.2 Struktur der Lösungen eines Unterfalles

Wie gesehen können die Unterfälle getrennt betrachtet werden. Es soll nun die Struktur aller möglichen Lösungen $x(y)$ in einem Unterfall untersucht werden.

Der gesuchte Iterationspunkt $x(y)$ muß zunächst das folgende ganzzahlige Gleichungssystem erfüllen.

$$Mx(y) = Uy + \nu - \mu \tag{2.2}$$

Dies gewährleistet, daß sowohl beim generierenden als auch beim benutzenden Vorkommen auf dasselbe Feldelement zugegriffen wird.

Außerdem ist ein Unterfall dadurch definiert, daß für ein bestimmtes k mit $1 \leq k \leq d$ die ersten $k-1$ Komponenten von x mit denen von y gleichgesetzt werden, d.h.

$$\begin{aligned} x_1 &= y_1 \\ &\vdots \\ x_{k-1} &= y_{k-1} \end{aligned} \tag{2.3}$$

Setzt man dies in (2.2) ein, so erhält man für die übrigen Komponenten $x_{(k\ldots d)}$ von x das folgende ganzzahlige Gleichungssystem.

$$\begin{aligned} M_{(1\ldots k-1)}y_{(1\ldots k-1)} + M_{(k\ldots d)}x_{(k\ldots d)} &= Uy + \upsilon - \mu \\ &= U_{(1\ldots k-1)}y_{(1\ldots k-1)} + U_{(k\ldots d)}y_{(k\ldots d)} + \\ &\quad + \upsilon - \mu \\ M_{(k\ldots d)}x_{(k\ldots d)} &= (U_{(1\ldots k-1)} - M_{(1\ldots k-1)})y_{(1\ldots k-1)} + \\ &\quad \underbrace{+U_{(k\ldots d)}y_{(k\ldots d)} + \upsilon - \mu}_{y'} \end{aligned}$$

Die Menge aller möglichen Lösungen dieses Systems läßt sich mit Hilfe der Smith Normalform (SNF)[4] [44] berechnen (s. auch Anhang B). Dabei wird die Matrix $M_{(k\ldots d)}$ mit Hilfe ganzzahliger (unimodularer) Operationen in eine Diagonalmatrix $\begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix}$ umgeformt. Die Berechnung einer Lösung eines „diagonalen" Systems ist aber einfach. Falls die Matrix $M_{(k\ldots d)}$ nicht vollen Rang hat, ist die Lösung nicht eindeutig bestimmt. Daher bildet die Menge aller Lösungen einen Unterraum des $\mathbf{Z}^{d-k+1}$. Dies entspricht auch der in Abschnitt 2.2 angegebenen Anschauung.

Die Menge aller Lösungen läßt sich also mit Hilfe der SNF folgendermaßen schreiben:

$$x_{(k\ldots d)}(y) = By' + V_2 w \qquad \text{für beliebige} \quad w \in \mathbf{Z}^{d-k+1-z} \tag{2.4}$$

wobei

- $z = \text{Rang}(M_{(k\ldots d)})$
- $WM_{(k\ldots d)}V = \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix}$ Smith Normalform von $M_{(k\ldots d)}$, wobei
- $D \in \mathbf{Z}^{z\times z}$ eine Diagonalmatrix ist und W und V unimodulare Transformationsmatrizen sind, die folgendermaßen aufgespalten werden:
- $W = \begin{bmatrix} W_1 \\ W_2 \end{bmatrix} \in \mathbf{Z}^{m\times m}$ unimodular, $W_1 \in \mathbf{Z}^{z\times m}$
- $V = [V_1 V_2] \in \mathbf{Z}^{d-k+1\times d-k+1}$ unimodular, $V_1 \in \mathbf{Z}^{d-k+1\times z}$, $V_2 \in \mathbf{Z}^{d-k+1\times d-k-z+1}$
- $B = V_1 D^{-1} W_1 \in \mathbf{Q}^{d-k+1\times m}$
- $y'' = Wy'$ and $y'' = \begin{pmatrix} y_1'' \\ y_2'' \end{pmatrix} \in \mathbf{Z}^m$ wobei $y_1'' \in \mathbf{Z}^z$ and $y_2'' \in \mathbf{Z}^{m-z}$

falls

- $D^{-1} y_1'' \in \mathbf{Z}^z$
- $y_2'' = (0, \ldots, 0)^t$

Wenn eine dieser beiden Bedingungen für ein gegebenes y nicht erfüllt ist, hat das ganzzahlige Gleichungssystem keine ganzzahlige Lösung. In diesem Fall kann für dieses y der Unterfall nicht auftreten und muß nicht weiter berücksichtigt werden. Daher werden diese beiden Bedingungen an y für den Unterfall gespeichert.

Man sieht, daß die Dimension des Lösungsraumes $d - k + 1 - z$ ist. Für Matrizen mit hohem Rang z ist dies relativ klein. In praktischen Fällen bewegt sich die Dimension fast nur im Bereich $0, \ldots, 2$. Dies kann man sich (wie im nächsten Abschnitt beschrieben) bei der Berechnung des gesuchten x zunutze machen.

[4] Für Implementierungszwecke ist die Benutzung der Hermite-Normalform (HNF) effizienter. Die Schreibweise mit der SNF ist aber etwas einfacher und wird daher hier benutzt.

Die gefundene Lösung muß nun in das Optimierungsproblem eingesetzt werden. Die ersten $k-1$ Komponenten von x sind mit denen von y gleichgesetzt und erfüllen damit die Polytopgrenzenbedingung. Für die k-te Komponente muß gelten $x_k < y_k$, damit x lexikographisch kleiner als y ist. Also

$$(By' + V_2 w)_1 < y_k$$

Außerdem müssen die Komponenten $k, \ldots, d$ die Polytopgrenzen erfüllen. Die obere Schranke für die k-te Komponente von x ist schon durch obigen Ausdruck gegeben. Für die restlichen Komponenten muß gelten:

$$\begin{aligned}
(A^{(1)}x)_{(k+1,\ldots,d)} &\leq b^{(1)}_{(k+1,\ldots,d)} \\
\left(A^{(1)} \begin{pmatrix} y_1 \\ \vdots \\ y_{k-1} \\ By' + V_2 w \end{pmatrix} \right)_{(k+1,\ldots,d)} &\leq b^{(1)}_{(k+1,\ldots,d)} \\
(A^{(1)}_{(1,\ldots k-1)} y_{(1,\ldots k-1)} + A^{(1)}_{(k,\ldots,d)}(By' + V_2 w))_{(k+1,\ldots,d)} &\leq b^{(1)}_{(k+1,\ldots,d)}
\end{aligned}$$

Entsprechend berechnen sich die unteren Schranken, die die Komponenten $x_{(k,\ldots,d)}$ erfüllen müssen.

Insgesamt muß das lexikographische Maximum aller x bestimmt werden, das die obigen Bedingungen erfüllt. Daher ergibt sich das folgende Optimierungsproblem:

Bestimme $w \in \mathbf{Z}^{d-k+1-z}$, so daß $V_2 w$ lexikographisch maximal ist mit

$$\begin{aligned}
(V_2 w)_1 &\leq (y_k - (By')_1) - 1 && (2.5) \\
(A^{(1)}_{(k\ldots d)} V_2 w)_{(k+1\ldots d)} &\leq (b^{(1)} - A^{(1)}_{(1\ldots k-1)} y_{(1\ldots k-1)} - A^{(1)}_{(k\ldots d)} By')_{(k+1\ldots d)} && (2.6) \\
(A^{(2)}_{(k\ldots d)} V_2 w)_{(k\ldots d)} &\leq (b^{(2)} - A^{(2)}_{(1\ldots k-1)} y_{(1\ldots k-1)} - A^{(2)}_{(k\ldots d)} By')_{(k\ldots d)} && (2.7)
\end{aligned}$$

(2.5) stellt sicher, daß $x(y) <_{\text{lex}} y$ und (2.6) und (2.7) gewährleisten $x(y) \in \mathbf{I}$.

Abkürzend läßt sich dieses Bedingungssystem schreiben als

$$Ew \leq Fy + f \tag{2.8}$$

mit

$$\begin{aligned}
E &\in \mathbf{Z}^{2(d-k+1)\times d-k+1-z} \\
F &\in \mathbf{Q}^{2(d-k+1)\times d} \\
f &\in \mathbf{Z}^{2(d-k+1)}
\end{aligned}$$

Zeilen des Ungleichungssystems, bei denen in E nur Nullen stehen, sind wie die obigen Bedingungen Bedingungen an y für die Gültigkeit des Unterfalls. Falls ein gegebenes y diese Bedingungen nicht erfüllt, muß der Unterfall für dieses y nicht weiter verfolgt werden. Außerdem kann an dieser Stelle geprüft werden, ob es überhaupt ein $y \in \mathbf{Z}^d$ gibt, das folgendes erfüllt:

- die Polytopgrenzenbedingungen

- die Bedingung $y_2'' = (0, \ldots, 0)$
- die Bedingungen des Systems (2.8) für Nullzeilen von E

Dies kann z.B. mit Hilfe der (ganzzahligen)[5] linearen Programmierung oder mit Methoden, die in [65] beschrieben sind, geschehen. Kann es kein solches y geben, muß der Unterfall nicht weiter untersucht werden. Er kann nie auftreten. Eine entsprechende Information wird für den Unterfall abgespeichert.

Das eigentliche Ziel ist es nun, solche Bedingungen an y zu finden, so daß sich für w dieselbe lineare Abhängigkeit von y ergibt. Mit (2.3) und (2.4) ergibt sich dann die gewünschte lineare Abhängigkeit für den Iterationspunkt x vom Iterationspunkt y.

2.4 Ein Spezialfall

Das obige Optimierungssystem (2.8) für einen Unterfall kann einfach gelöst werden, falls jede Zeile von E maximal ein von Null verschiedenes Element besitzt. Diese Eigenschaft soll im folgenden „Zeilenbedingung“ genannt werden.

Definition 2.4 *Eine Matrix erfüllt die* Zeilenbedingung *genau dann, wenn jede Zeile der Matrix maximal ein Element ungleich Null enthält.*

In diesem Fall können die einzelnen Komponenten von w getrennt betrachtet werden. Andernfalls muß man die allgemeine Methode von Feautrier [16] anwenden.

Bemerkungen:

- Man kann allerdings nicht direkt die Methode von Feautrier anwenden, da dort der Zielvektor lexikographisch maximiert wird. Dies ist hier jedoch nicht der Fall, da nicht w, sondern $V_2 w$ lexikographisch maximiert werden soll.
- Bei der Implementierung von [30] wird eine Fehlermeldung ausgegeben, falls E nicht die Zeilenbedingung erfüllt.

2.4.1 Lösung im Spezialfall

E erfülle nun die Zeilenbedingung. Jede Zeile mit einem Element ungleich Null bildet eine obere oder untere Schranke für genau eine Komponente w_i. Daher können die einzelnen Komponenten von w getrennt berechnet werden. Da $V_2 w$ lexikographisch maximiert werden soll, muß die Komponente w_i von w maximiert werden, falls das erste Nicht-Null-Element der i-ten Spalte von V_2 positiv ist. Andernfalls muß w_i minimiert werden. Für die folgenden Überlegungen sei w_i o.B.d.A. zu maximieren.

Um das lexikographische Maximum zu finden, wird w_i nun nacheinander mit den oberen Schranken für w_i gleichgesetzt. Dabei muß beachtet werden, daß w_i ganzzahlig sein muß,

[5] Exakter wäre ein Test mit ganzzahliger linearer Programmierung. Für praktische Fälle reicht aber normale lineare Programmierung aus.

d.h. die entsprechende obere Schranke wird abgerundet. Alle anderen Ungleichungen für w_i stellen Bedingungen dar, ob diese Wahl von w_i für ein bestimmtes y möglich ist. Durch den Vergleich mit den anderen oberen Schranken wird gewissermaßen die kleinste obere Schranke für ein festes y bestimmt. Der Vergleich mit den unteren Schranken zeigt, ob die gewählte kleinste obere Schranke groß genug für die unteren Schranken ist.

Bemerkungen:

- Bei den entstehenden Bedingungen betreffend die oberen Schranken kann auf das Runden verzichtet werden, da für zwei rationale Zahlen a und b mit $a \leq b$ auch folgt $\lfloor a \rfloor \leq \lfloor b \rfloor$.

  Falls durch 1 dividiert wird, muß auch bei den unteren Schranken nicht gerundet werden. Dies gilt, weil $Fy + f$ ganzzahlig ist für zulässige y, da gefordert wird, daß $D^{-1}y_1''$ ganzzahlig sein muß (s. Berechnung der SNF). Damit sind aber auch $D^{-1}(Wy')_1$, $D^{-1}W_1y'$ und By' ganzzahlig.

  Dadurch entsteht ein einfacheres Bedingungssystem.

- Um das entstehende Bedingungssystem zu vereinfachen, kann auch hier überprüft werden, ob es überhaupt einen Punkt y des Iterationsraumes geben kann, der die Bedingungen erfüllt. Dazu wird folgendes System linearer Ungleichungen aufgestellt:

  - Polytopgrenzenbedingungen für y
  - Bedingungen an y für den Unterfall
  - Bedingungen betreffend die oberen Schranken (s.o.)
  - Bedingungen betreffend die unteren Schranken

    Für die Konsistenzprüfung können auch hier die Rundungen weggelassen werden. Wenn die Prüfung dann ergibt, daß es kein gültiges y gibt, gibt es erst recht keines, wenn man die Rundungen beachtet.

  Ob es ein y gibt, das dieses System erfüllt, kann wieder mit Hilfe linearer Programmierung bestimmt werden.

Beispiel 2.5 Seien z.B. die folgenden Schranken für w_i gegeben:

$$\begin{aligned} 2w_i &\leq y_1 + 3y_2 \\ w_i &\leq 2y_1 + 1 \\ 3w_i &\geq y_1 - y_2 \end{aligned}$$

Dann gilt:

- $w_i = \left\lfloor \frac{1}{2}y_1 + \frac{3}{2}y_2 \right\rfloor$ falls

$$\begin{aligned} \frac{1}{2}y_1 + \frac{3}{2}y_2 &\leq 2y_1 + 1 \\ 3\left\lfloor \frac{1}{2}y_1 + \frac{3}{2}y_2 \right\rfloor &\geq y_1 - y_2 \end{aligned}$$

- $w_i = 2y_1 + 1$ falls

$$\begin{aligned} 2y_1 + 1 &\leq \frac{1}{2}y_1 + \frac{3}{2}y_2 \\ 6y_1 + 3 &\geq y_1 - y_2 \end{aligned}$$

Die Kombination der verschiedenen Möglichkeiten der einzelnen Komponenten w_i führt dann zu einem Gesamtsystem von linearen Abhängigkeiten für x von y und den entsprechenden Bedingungen für den betrachteten Unterfall. Dabei kann wieder eine Konsistenzprüfung für die Kombination der Bedingungen der einzelnen Komponenten durchgeführt werden.

Wie vorher gesehen ergibt sich die *Gesamtlösung*, indem man die Unterfälle in der in Abschnitt 2.3.1 angegebenen Reihenfolge untersucht.

Das insgesamt entstehende Bedingungssystem hat also die folgende Form:

```
if (Bedingung für den ersten Unterfall) then
        if (Bedingung) then lineare Abhängigkeit
        ⋮
        if (Bedingung) then lineare Abhängigkeit
if (Bedingung für den zweiten Unterfall) then
        if (Bedingung) then lineare Abhängigkeit
        ⋮
        if (Bedingung) then lineare Abhängigkeit
⋮
```

Wie gesehen sind dabei die Bedingungen Bedingungen an den betrachteten Iterationspunkt y. Bedingungen für einen Unterfall sind lineare Bedingungen der Form $ay + c = 0$, $ay + c$ ganzzahlig und $ay \leq c$. Die Bedingungen für die einzelnen linearen Abhängigkeiten sind $\leq$-Bedingungen mit bzw. ohne Runden.

Es besteht auch die Möglichkeit, das Bedingungssystem als Bedingungsbaum zu beschreiben und abzuspeichern. In [18] wird dieser „QUAST“ (quasi-affine selection tree) und in [39] „LWT“ (last write tree) genannt.

Auch bei der Verknüpfung der Unterfälle ist ein Ansatzpunkt gegeben, das entstehende Bedingungssystem zu vereinfachen. Der zweite Unterfall wird nämlich nur dann untersucht, wenn keine lineare Abhängigkeit des ersten Unterfalls zutrifft. Man kann also überprüfen, ob die Bedingungen für eine lineare Abhängigkeit des zweiten Unterfalls konsistent sind mit der Bedingung, daß keine lineare Abhängigkeit des ersten Unterfalls zutrifft. Kann das nicht passieren, muß diese Bedingung nicht betrachtet werden.

2.4.2 Beispiele

Um die obigen, theoretischen Ausführungen zu erhellen, sollen nun zwei Beispiele betrachtet werden. Dabei ist das erste Beispiel sehr einfach. In der Praxis auftretende Probleme entsprechen sehr oft dieser Form. Das zweite Beispiel ist etwas künstlicher, aber komplizierter, um das Vorgehen auch für diesen Fall zu illustrieren.

Beispiel 2.6

$$
\begin{array}{l}
\text{for } k = 1 \ldots n-1 \\
\quad \text{for } i = k+1 \ldots m \\
\quad\quad \text{for } j = k+1 \ldots n \\
\quad\quad\quad \ldots = \ldots a[k,j] \ldots \\
\quad\quad\quad a[i,j] = \ldots
\end{array}
$$

Die algebraische Beschreibung ist:

$$
\begin{aligned}
d &= 3 \\
M &= \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
U &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
\mu = \nu &= \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\
A^{(1)} &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
b^{(1)} &= \begin{pmatrix} n-1 \\ m \\ n \end{pmatrix} \\
A^{(2)} &= \begin{pmatrix} -1 & 0 & 0 \\ 1 & -1 & 0 \\ 1 & 0 & -1 \end{pmatrix} \\
b^{(2)} &= \begin{pmatrix} -1 \\ -1 \\ -1 \end{pmatrix}
\end{aligned}
$$

Es müssen folgende Unterfälle betrachtet werden:

- $x_1 = y_1$, $x_2 = y_2$:
  $Mx = Uy$ führt zu

$$
\begin{pmatrix} 0 \\ 1 \end{pmatrix} x_3 = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} \tag{2.9}
$$

  Die erste Reihe beschreibt eine Bedingung für y ($y_1 = y_2$), die kein Punkt y des Iterationsraumes erfüllt, da hier gilt $y_2 > y_1$. Daher muß dieser Unterfall nicht weiter betrachtet werden.

- $x_1 = y_1$:
  $Mx = Uy$ führt zu

$$
\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} \tag{2.10}
$$

Ähnlich wie in dem ersten Unterfall ergibt die erste Reihe die Bedingung $x_2 = y_1 = x_1$. Dies kann aber nicht auftreten.

- $x_1 < y_1$:

  Die Lösungen des Gleichungssystems $Mx = Uy$ ergeben sich mit Hilfe der Smith-Normalform zu:

$$x = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} w \tag{2.11}$$

  Betrachtet man die Schranken für die Komponenten von x, so muß w maximiert werden mit $w \leq y_1 - 1$.

In diesem Beispiel gibt es also nur eine lineare Abhängigkeit:

$$x(y) = \begin{pmatrix} y_1 - 1 \\ y_1 \\ y_3 \end{pmatrix} \tag{2.12}$$

Es soll nun ein komplizierteres Beispiel betrachtet werden:

Beispiel 2.7

for $i_1 = 0 \ldots n$
  for $i_2 = -i_1 \ldots i_1$
    for $i_3 = 0 \ldots n$
      for $i_4 = 2i_1 - i_2 \ldots n$
        $\ldots = \ldots a[3i_1 - i_3 - i_4, i_1 + i_2] \ldots$
        $a[i_1 + 3i_4, 2i_2 + 2i_4] = \ldots$

Die formale Beschreibung ist:

$$\begin{aligned} d &= 4 \\ m &= 2 \\ M &= \begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 2 & 0 & 2 \end{pmatrix} \\ U &= \begin{pmatrix} 3 & 0 & -1 & -1 \\ 1 & 1 & 0 & 0 \end{pmatrix} \\ A^{(1)} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ b^{(1)} &= (n, 0, n, n)^T \\ A^{(2)} &= \begin{pmatrix} -1 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 2 & -1 & 0 & -1 \end{pmatrix} \\ b^{(2)} &= (0, 0, 0, 0)^T \end{aligned}$$

Die zu untersuchenden Unterfälle sind:

(1) $x_1 = y_1, x_2 = y_2, x_3 = y_3, x_4 < y_4$

(2) $x_1 = y_1, x_2 = y_2, x_3 < y_3$

(3) $x_1 = y_1, x_2 < y_2$

(4) $x_1 < y_1$

Aus Platzgründen sollen hier nur exemplarisch die Unterfälle (2) und (4) betrachtet werden. Eine ausführliche Berechnung findet sich in [30].

Unterfall (2): $x_1 = y_1, x_2 = y_2, x_3 < y_3$

Das zu betrachtende ganzzahlige Gleichungssystems ist:

$$\begin{pmatrix} 0 & 3 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 2 & 0 & -1 & -1 \\ 1 & -1 & 0 & 0 \end{pmatrix} y$$

Die Berechnung der Smith-Normalform führt zu

$$\begin{pmatrix} 1 & -1 \\ -2 & 3 \end{pmatrix} \begin{pmatrix} 0 & 3 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

womit sich die Menge aller möglichen Lösungen zu

$$x(y)_{(3\ldots4)} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & -1 & -1 \end{pmatrix} y + \begin{pmatrix} -1 \\ 0 \end{pmatrix} w \qquad \text{für beliebiges} \quad w \in \mathbf{Z}$$

ergibt, falls $D^{-1}y_1''$ ganzzahlig ist, was hier immer erfüllt ist, und falls $y_2'' = 0$, also

$$-y_1 - 3y_2 + 2y_3 + 2y_4 = 0 \tag{2.13}$$

Setzt man diese Lösung in die Schleifengrenzenbedingungen ein, so erhält man:

$$\begin{aligned} -w &< y_3 \\ y_1 + y_2 - y_3 - y_4 &\leq n \\ 0 &\leq -w \\ 2y_1 - y_2 &\leq y_1 + y_2 - y_3 - y_4 \end{aligned}$$

Um x lexikographisch zu maximieren, muß w minimiert werden. Somit erhält man

$$x(y) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & -1 & -1 \end{pmatrix} y + \begin{pmatrix} 0 \\ 0 \\ -1 \\ 0 \end{pmatrix}$$

falls

$$\begin{aligned} 0 &\leq y_3 - 1 \\ 0 &\leq -y_1 + 2y_2 - y_3 - y_4 \\ y_1 + y_2 - y_3 - y_4 &\leq n \\ -y_1 - 3y_2 + 2y_3 + 2y_4 &= 0 \end{aligned}$$

Unterfall (4): $x_1 < y_1$

Das zu betrachtende ganzzahlige Gleichungssystems ist:

$$\begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 2 & 0 & 2 \end{pmatrix} x = \begin{pmatrix} 3 & 0 & -1 & -1 \\ 1 & 1 & 0 & 0 \end{pmatrix} y$$

Die Berechnung der Smith-Normalform führt zu

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 2 & 0 & 2 \end{pmatrix} \begin{pmatrix} 1 & -3 & 0 & -3 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \end{pmatrix}$$

womit sich die Menge aller möglichen Lösungen zu

$$x(y) = \begin{pmatrix} \frac{3}{2} & -\frac{3}{2} & -1 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \end{pmatrix} y + \begin{pmatrix} 0 & -3 \\ 0 & -1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} \qquad \text{für beliebige} \quad w_1, w_2 \in \mathbf{Z}$$

ergibt, falls $D^{-1}y_1''$ ganzzahlig ist, d.h.

$$\begin{pmatrix} 1 & 0 \\ 0 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 3 & 0 & -1 & -1 \\ 1 & 1 & 0 & 0 \end{pmatrix} y \in \mathbf{Z}^2 \tag{2.14}$$

Setzt man diese Lösung in die Schleifengrenzenbedingungen ein, so erhält man:

$$\begin{aligned}
\tfrac{3}{2}y_1 - \tfrac{3}{2}y_2 - y_3 - y_4 - 3w_2 &\leq y_1 - 1 && (x_1 < y_1) \\
-w_2 &\leq \tfrac{3}{2}y_1 - \tfrac{3}{2}y_2 - y_3 - y_4 - 3w_2 && (x_2 \leq x_1) \\
w_1 &\leq n && (x_3 \leq n) \\
\tfrac{1}{2}y_1 + \tfrac{1}{2}y_2 + w_2 &\leq n && (x_4 \leq n) \\
0 &\leq \tfrac{3}{2}y_1 - \tfrac{3}{2}y_2 - y_3 - y_4 - 3w_2 && (0 \leq x_1) \\
-\tfrac{3}{2}y_1 + \tfrac{3}{2}y_2 + y_3 + y_4 + 3w_2 &\leq -w_2 && (-x_1 \leq x_2) \\
0 &\leq w_1 && (0 \leq x_3) \\
2(\tfrac{3}{2}y_1 - \tfrac{3}{2}y_2 - y_3 - y_4 - 3w_2) + w_2 &\leq \tfrac{1}{2}y_1 + \tfrac{1}{2}y_2 + w_2 && (2x_1 - x_2 \leq x_4)
\end{aligned}$$

Um x lexikographisch zu maximieren, muß w_1 maximiert und w_2 minimiert werden. Somit erhält man $w_1 = n$. Für w_2 gibt es vier obere und zwei untere Schranken. Die zwei unteren Schranken müssen gleichgesetzt werden (evtl. mit Aufrunden), wie in Beispiel 2.5 dargestellt.

2.4.3 Geltung des Spezialfalles

Es soll nun untersucht werden, wann der betrachtete Spezialfall auf jeden Fall eintritt.

- Wenn $\text{Rang}(M_{(k,\ldots,d)}) = d-k+1$ oder $\text{Rang}(M_{(k,\ldots,d)}) = d-k$ so hat die Matrix V_2, die sich aus der SNF ergibt, maximal eine Spalte. Der Spezialfall ist also auf jeden Fall für diesen Unterfall erfüllt.

  Gilt dies schon für die Ausgangsmatrix M, d.h. $\text{Rang}(M) = d$ oder $\text{Rang}(M) = d-1$, so gilt die Bedingung auch für jeden Unterfall. Dies ist also auf jeden Fall für ein- und zweidimensionale Schleifen gegeben. Nach [63] haben nur 8 % aller in praktischen Problemen auftretende Schleifen eine Dimension größer zwei.

- Falls $M_{(k,\ldots,d)}$ mindestens $d-k-z$ Nullspalten besitzt, wobei z der Rang von $M_{(k,\ldots,d)}$ ist, so erfüllt V_2 die Zeilenbedingung. Dies ist intuitiv folgendermaßen einsichtig: Die Dimension des Lösungsraums beträgt wie gesehen $d-k+1-z$, d.h. V_2 hat $d-k+1-z$ Spalten. Da die Matrix $d-k-z$ Nullspalten hat, ist die entsprechende Anzahl von Variablen durch das Gleichungssystem in keiner Weise festgelegt und damit absolut frei wählbar. Dies drückt sich in V_2 so aus, daß es $d-k-z$ Spalten gibt, in denen genau eine eins steht (in jeder Spalte an einer anderen Stelle), und daß in der verbleibenden Spalte von V_2 in den Zeilen, in denen schon eine eins steht, eine Null steht. Somit erfüllt V_2 die Zeilenbedingung.

  Diese Bedingung gilt auf jeden Fall dann für jeden Unterfall, wenn die Ausgangsmatrix M mindestens $d-z-1$ Nullspalten besitzt, wobei z der Rang von M ist. Kommt z.B. in jeder Dimension der Feldvariablen maximal eine Schleifenvariable vor, was nach [63] nicht unwahrscheinlich ist, so hat M sogar $d-z$ Nullspalten.

  Daher ist folgender Fall einfach zu lösen: M besitzt mindestens $d-z-1$ Nullspalten und der Iterationsraum formt ein Rechteck.

2.4.4 Anzahl der verschiedenen linearen Abhängigkeiten

Für verschiedene Anwendungen ist es interessant, wieviele verschiedene lineare Abhängigkeiten entstehen können. Für den hier betrachteten Spezialfall kann eine relativ leichte Abschätzung angegeben werden.

Für einen Unterfall k ($k = d, \ldots 1$) gilt:

- Die Dimension des betrachteten Iterationsraums ist $d_k = d-k+1$.
- Die Dimension der Menge aller in Frage kommender Punkte $x(y)$ ist $f_k = d-k+1-z$.

Dabei gilt $0 \leq f_k \leq d_k$.

Gemäß (2.8) hat das entstehende System $2d_k$ Ungleichungen. Diese Ungleichen zerfallen im Spezialfall in f Blöcke der Größe $g_1, \ldots g_f$, wobei jeder Block Ungleichungen für genau eine Komponente w_i beschreibt. Es gilt also $0 \leq g_i \leq 2d_k$ und $\sum_{i=1}^{f} g_i = 2d_k$.

Für eine Komponente w_i gibt es also im schlechtesten Fall g_i Ungleichungen, die gleichgesetzt werden können.[6] Daher können in dem gesamten Unterfall maximal $\prod_{i=1}^{f} g_i$ verschiedene lineare Abhängigkeiten entstehen.[7]

Das Produkt $\prod_{i=1}^{f} g_i$ mit $\sum_{i=1}^{f} g_i = 2d_k$ wird maximal, wenn alle g_i gleich groß sind.[8] Im

[6] Im allgemeinen gibt es weniger, da die Ungleichungen für w_i in untere und obere Schranken für w_i zerfallen. Man muß nur die oberen oder nur die unteren gleichsetzen. Für den Fall $A^{(1)} = E$ und $A^{(2)} = -E$ (Iterationsraum = Rechteck) gibt es z.B. $\frac{g_i}{2}$ obere und $\frac{g_i}{2}$ untere Schranken für w_i.

[7] In dem vorher beschriebenen Fall nur $\prod_{i=1}^{f} \frac{g_i}{2}$.

[8] Berechnung des Optimums nach der Methode aus [8, S. 287].

schlechtesten Fall gilt also für die Anzahl verschiedener linearer Abhängigkeiten in einem Unterfall:

$$\prod_{i=1}^{f} g_i = \prod_{i=1}^{f} \frac{2d_k}{f} = (\frac{2d_k}{f})^f$$

Durch Ableiten erhält man, daß dies maximal wird für $f = \frac{2d_k}{e}$.[9] In einem Unterfall können also maximal $e^{\frac{2d_k}{e}}$ verschiedene lineare Abhängigkeiten entstehen.

Addiert man dies über alle Unterfälle auf, so erhält man als obere Schranke für die Anzahl der verschiedenen linearen Abhängigkeiten:

$$\sum_{i=1}^{d} e^{\frac{2i}{e}}$$

Damit erhält man folgende Zahlen:

Dimension d	Maximale Anzahl
1	2
2	6
3	15
4	34
5	74

In praktischen Fällen werden natürlich durch die Konsistenzprüfungen weit weniger verschiedene Möglichkeiten entstehen.

2.5 Erweiterungen

Es sollen nun zwei wichtige Erweiterungen des einfachen Schleifenmodells aus Abschnitt 2.2 betrachtet werden. Dies ist zum einen, daß die Schleifen nicht vollständig geschachtelt sind, und zum anderen, daß es mehrere Anweisungen gibt, die schreibend auf das Feld zugreifen. Diese beiden Fälle werden auch in [18, 39] behandelt.

Ein Beispiel wird in Abschnitt 2.5.3 vorgestellt.

2.5.1 Teilweise überlappende Schleifen

Es werden Schleifen der folgenden Art betrachtet.

for i_1
  for i_2
    $\ddots$
      for i_l

[9] Dies ist natürlich nicht ganzzahlig. Aber zur Abschätzung einer oberen Schranke kann man damit weiterrechnen.

```
for i_{l+1}
    ⋱
        for i_d
            (s1)    A[MI + μ] = ...

for i_{l+1}
    ⋱
        for i_{d'}
            (s2)    ... = ... A[UI + υ] ...
```

Das generierende und das modifizierende Vorkommen haben l Schleifen gemeinsam und sind ansonsten noch in weitere, nicht gemeinsame Schleifen geschachtelt.

Dieser Fall unterscheidet sich von dem Fall vorher nur dadurch, daß die lexikographische Ordnung umdefiniert werden muß. Um einen Iterationspunkt x bzgl. (s1) und einen Iterationspunkt y bzgl. (s2) lexikographisch zu vergleichen, werden nur die ersten l Komponenten (also die Indizes der gemeinsamen Schleifen) betrachtet. Stimmen x und y auf den ersten l Komponenten überein, so entscheidet die Reihenfolge im Programmtext über die lexikographische Reihenfolge. Im obigen Fall wäre $x <_{\text{lex}} y$, da (s1) vor (s2) steht.

Entsprechend müssen auch nur die Unterfälle, die durch die ersten l Komponenten definiert sind, betrachtet werden:

- $x_1 = y_1, \ldots x_l = y_l$

  Maximiere lexikographisch $x_{(l+1 \ldots d)}$.

- $x_1 = y_1, \ldots x_l < y_l$

  Maximiere lexikographisch $x_{(l \ldots d)}$.

  $\vdots$

- $x_1 < y_1$

  Maximiere lexikographisch x.

Der erste Unterfall wird dabei nur dann betrachtet, wenn (wie im Beispiel) das generierende Vorkommen textuell vor dem benutzten Vorkommen steht.

2.5.2 Das Problem der Überdeckung

Falls es mehrere Anweisungen gibt, die modifizierend auf das betrachtete Feld zugreifen, so muß die Anweisung bestimmt werden, die den benutzten Wert tatsächlich berechnet hat. Die Werte, die von den anderen Vorkommen geschrieben wurden, nennt man *überdeckt*. Ein möglicher Ansatz zur Lösung des Problems ist der folgende:

Zuerst wird jedes generierende Vorkommen mit den bisher beschriebenen Methoden behandelt. Dann wird die korrekte Abhängigkeit ermittelt, indem „gleichwertige“ Unterfälle der generierenden Vorkommen miteinander verglichen werden. Zwei Unterfälle heißen

dabei gleichwertig, wenn gleich viele Komponenten gleich gesetzt werden. Es wird die Abhängigkeit des generierenden Vorkommens gewählt, die (für ein gegebenes y) zu dem lexikographisch maximalen[10] Iterationspunkt führt. Abhängig von y kann dies natürlich zu verschiedenen Ergebnissen führen. Die Unterfälle werden dabei wieder in der angegebenen Reihenfolge betrachtet. Es ist zu beachten, daß im Falle teilweise verschachtelter Schleifen verschiedene generierende Vorkommen verschiedene Unterfälle aufweisen können.

2.5.3 Beispiele

Mit Hilfe der angegebenen Verfahren können die linearen Abhängigkeiten u.a. für den Jakobi-SVD-Algorithmus, den Householder-Algorithmus, die Cholesky-Zerlegung und LU-Zerlegung berechnet werden. In allen diesen Algorithmen trifft der Spezialfall zu.

Als Beispiel soll hier der Jakobi-Algorithmus [22] betrachtet werden. Die Algorithmen, die bei der parallelen Gitterbasisreduktion in Kapitel 3 verwendet werden, weisen eine ähnliche Struktur auf.

Beispiel 2.8 [Jakobi-ähnliche Algorithmen zur SVD-Zerlegung]

$$
\begin{array}{l}
\text{for } p = 1, \ldots, n-1 \\
\quad \text{for } q = p+1, \ldots, n \\
\quad\quad l_{pp}, l_{pq}, l_{qp}, l_{qq} = f(a_{pp}, a_{pq}, a_{qp}, a_{qq}) \\
\quad\quad r_{pp}, r_{pq}, r_{qp}, r_{qq} = g(a_{pp}, a_{pq}, a_{qp}, a_{qq}) \\
\quad\quad \text{for } i = 1, \ldots, n \\
\quad\quad\quad h = a_{pi} \qquad\qquad (S) \\
\quad\quad\quad a_{pi} = h l_{pp} + a_{qi} l_{pq} \qquad (G_1) \\
\quad\quad\quad a_{qi} = h l_{qp} + a_{qi} l_{qq} \qquad (G_2) \\
\quad\quad \text{for } j = 1, \ldots, n \\
\quad\quad\quad h = a_{jp} \\
\quad\quad\quad a_{jp} = h r_{pp} + a_{jq} r_{qp} \qquad (G_3) \\
\quad\quad\quad a_{jq} = h r_{pq} + a_{jq} r_{qq} \qquad (G_4)
\end{array}
$$

In Anweisung (S) wird benutzend auf die Feldvariable a zugegriffen. Es gibt vier Anweisungen, in denen auf a verändernd zugegriffen wird. Die Anweisungen (G_1) und (G_2) haben drei gemeinsame Schleifen mit (S) und stehen textuell hinter (S). Daher müssen die folgenden Unterfälle betrachtet werden:

(1) $x_1 = y_1$, $x_2 = y_2$ und $x_3 < y_3$

(2) $x_1 = y_1$, $x_2 < y_2$

(3) $x_1 < y_1$

(G_3) und (G_4) haben hingegen nur zwei gemeinsame Schleifen mit (S). Es müssen also nur die folgenden Unterfälle betrachtet werden.

(2) $x_1 = y_1$, $x_2 < y_2$

(3) $x_1 < y_1$

[10] Bei teilweise verschachtelten Schleifen wird dabei die obige Definition für „lexikographisch maximal“ gewählt.

Generierendes Vorkommen (G_1)

Das zu lösende ganzzahlige Gleichungssystem $M_1 x = Uy$ ist

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} \tag{2.15}$$

Es sollen nun die angegebenen Unterfälle betrachtet werden:

(1) $x_1 = y_1$, $x_2 = y_2$ und $x_3 < y_3$

Die zweite Gleichung von (2.15) ist ein Widerspruch zu $x_3 < y_3$.

(2) $x_1 = y_1$, $x_2 < y_2$

Das resultierende Gleichungssystem ist

$$\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Die Menge aller möglichen Lösungen ist

$$x(y) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} w \qquad \text{für beliebiges} \quad w \in \mathbf{Z}$$

w muß maximiert werden mit

$$\begin{aligned} 1 \leq \;& y_1 \;& \leq n-1 \\ y_1 + 1 \leq \;& w \;& < y_2 \\ 1 \leq \;& y_3 \;& \leq n \end{aligned}$$

Daher gilt $w = y_2 - 1$ für y mit $y_1 + 1 \leq y_2 - 1$. Die gesamte lineare Abhängigkeit ist somit in diesem Fall:

$$x(y) = l_{2,1}(y) := \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

(3) $x_1 < y_1$

Dies ist aber ein Widerspruch zur ersten Gleichung von 2.15.

Anweisung (G_2) wird genauso behandelt.

Generierende Anweisung (G_3)

Das Gleichungssystem $M_3 x = Uy$ ist gegeben durch

$$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} \tag{2.16}$$

Wie erwähnt müssen hier nur die Unterfälle (2) und (3) untersucht werden.

(2) Das zu betrachtende Gleichungssystem ist

$$\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Für $y_1 = y_3$ ist die Menge aller möglichen Lösungen gegeben durch

$$x(y) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} w \qquad \text{für beliebiges} \quad w \in \mathbf{Z}$$

Daher muß w maximiert werden mit

$$\begin{aligned} 1 \leq \quad & y_1 \quad \leq n-1 \\ y_1 + 1 \leq \quad & w \quad < y_2 \\ 1 \leq \quad & y_1 \quad \leq n \end{aligned}$$

Die Lösung ist $w = y_2 - 1$ für y mit $y_1 = y_3$ und $y_1 + 2 \leq y_2$.

(3) Die Menge aller Lösungen von (2.16) ergibt sich aus:

$$x(y) = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} w \qquad \text{für beliebiges} \quad w \in \mathbf{Z}$$

w muß maximiert werden mit

$$\begin{aligned} 1 \leq \quad & y_3 \quad < y_1 \\ y_3 + 1 \leq \quad & w \quad \leq n \\ 1 \leq \quad & y_1 \quad \leq n \end{aligned}$$

Somit gilt $w = n$ für y mit $y_3 < y_1$.

Die Lösungen von (G_4) werden genauso behandelt.

Die Gesamtlösung

Die Lösungen der generierenden Vorkommen seien nun vollständig berechnet, wobei $l_{i,j}(y)$ die Lösung des i-ten Unterfalls des j-ten Vorkommens sein soll und $c_{i,j}(y)$ die Menge der Bedingungen, die y für diese Lösung erfüllen muß. Die obigen Überlegungen ergeben, daß Unterfall (1) für das erste generierende Vorkommen nicht auftreten kann. Das gilt auch für (G_2). Bei den anderen generierenden Vorkommen wird Unterfall (1) sowieso nicht betrachtet. Daher ist die Gesamtlösung:

(Unterfall (2))
if $l_{2,1}(y) \leq_{\text{lex}} l_{2,4}(y)$ and $l_{2,2}(y) \leq_{\text{lex}} l_{2,4}(y)$ and $l_{2,3}(y) \leq_{\text{lex}} l_{2,4}(y)$ and $c_{2,4}(y)$ then $x(y) = l_{2,4}(y)$
else if $l_{2,1}(y) \leq_{\text{lex}} l_{2,3}(y)$ and $l_{2,2}(y) \leq_{\text{lex}} l_{2,3}(y)$ and $c_{2,3}(y)$ then $x(y) = l_{2,3}(y)$
else if $l_{2,1}(y) \leq_{\text{lex}} l_{2,2}(y)$ and $c_{2,2}(y)$ then $x(y) = l_{2,2}(y)$
else if $c_{2,1}(y)$ then $x(y) = l_{2,1}(y)$
else
(Unterfall (3))
if $l_{3,1}(y) \leq_{\text{lex}} l_{3,4}(y)$ and $l_{3,2}(y) \leq_{\text{lex}} l_{3,4}(y)$ and $l_{3,3}(y) \leq_{\text{lex}} l_{3,4}(y)$ and $c_{3,4}(y)$ then $x(y) = l_{3,4}(y)$
else if $l_{3,1}(y) \leq_{\text{lex}} l_{3,3}(y)$ and $l_{3,2}(y) \leq_{\text{lex}} l_{3,3}(y)$ and $c_{3,3}(y)$ then $x(y) = l_{3,3}(y)$
else if $l_{3,1}(y) \leq_{\text{lex}} l_{3,2}(y)$ and $c_{3,2}(y)$ then $x(y) = l_{3,2}(y)$
else if $c_{3,1}(y)$ then $x(y) = l_{3,1}(y)$

Bemerkungen:

- Die Bedingung $\leq_{\text{lex}}$ zerfällt wieder in mehrere Unterbedingungen.
- Wie schon vorher gesehen können die entstehenden Bedingungssysteme vereinfacht werden.

2.6 Zusammenfassung

Ein wichtiges Problem bei der Parallelisierung verschachtelter Schleifen ist die Bestimmung linearer Datenabhängigkeiten. Feautrier [18] hat dafür ein allgemeines Verfahren angegeben, das die Lösung parametrisierter, ganzzahliger linearer Programme benutzt [16]. Hier wurde die Struktur des Problems genauer untersucht, wodurch für Spezialfälle, die in praktischen Anwendungen recht häufig scheinen, eine einfachere Lösung möglich ist. Die Hauptpunkte dabei waren (1) die Aufspaltung in Unterfälle (wie bei Feautrier) und (2), daß sich die Menge aller in Frage kommender Iterationspunkte x aus der Lösungsmenge eines ganzzahligen linearen Gleichungssystemes berechnen läßt. Ergibt diese Lösungsmenge zusammen mit der Form des Iterationsraumes ein „einfaches“ System, so kann die Lösung direkt bestimmt werden. In [39] wird ein ähnlicher Ansatz für Spezialfälle vorgestellt. Vielleicht ist eine effiziente Kombination aller drei Methoden möglich.

Das Verfahren (ohne Erweiterungen) wurde im Rahmen der Studienarbeit von Stephan Kann [30] implementiert. Im Sinne einer übersichtlichen Programmierung wurde die Programmiersprache „C++“ gewählt. Die rationalen Zahlen werden in exakter Darstellung verwaltet. Dadurch ist ein direkter Laufzeitvergleich mit dem Verfahren von Feautrier, der die Sprache „C“ und den Datentyp „double“ benutzt, nicht möglich. Außerdem könnte die Effizienz des Verfahrens verbessert werden, wenn statt der Smith-Normalform die Hermite-Normalform benutzt würde.

Kapitel 3

Parallele Algorithmen zur Basisreduktion in Gittern

3.1 Einleitung

Ein wichtiges Problem in dem Bereich der Computeralgebra und der Geometrie der ganzen Zahlen ist die Reduktion von Gitterbasen, d.h. das Problem, eine Basis eines Gitters zu finden mit möglichst kurzen Vektoren[1] („reduzierte Basis“). Ein Gitter ist dabei die Menge aller Punkte, die durch ganzzahlige Linearkombinationen der Basisvektoren berechenbar ist.[2] So erzeugen z.B. die beiden folgenden Matrizen dasselbe Gitter, wobei die Zahlen der zweiten Matrix wesentlich kleiner sind:

$$\begin{pmatrix} 60408853613420 & -3921463629539 & 25531978081724 \\ -214926455217486 & 13952032305640 & -90839292844591 \\ 62343913872 & -4047078798 & 26349836945 \end{pmatrix}$$

$$\rightarrow \begin{pmatrix} 2 & 7 & -1 \\ 2 & -5 & 0 \\ 0 & -1 & 8 \end{pmatrix}$$

Basen mit kurzen Vektoren finden z.B. Anwendung in der ganzzahligen Optimierung [44], bei der Faktorisierung von Polynomen [36], bei der Lösung von Rucksackproblemen [57, 48], beim „Knacken“ von darauf basierenden „public-key“ Kryptosystemen [40] und bei der Faktorisierung über algebraischen Zahlkörpern [35].

Das Problem, kurze Vektoren zu finden, ist schon lange Gegenstand der Forschung [28]. So gibt es auch verschiedene Definitionen von „Reduziertheit“. 1982 haben A.K. Lenstra, H.W. Lenstra und L. Lovasz [36] einen effizienten Algorithmus angegeben, der eine in einem gewissen Sinne „gute“ Basis findet. Solche Basen werden nach den Autoren „LLL-reduziert“ genannt. Seither wurden auch einige Varianten dieses Algorithmus entwickelt, z.B. [48, 58, 46, 55]. Um die Berechnung mit der langsamen, exakten ganzzahligen Arithmetik zu vermeiden, wurden Algorithmen implementiert, die, soweit möglich, auf Gleitkommazahlen [57] bzw. auf Festkommazahlen [56] arbeiten. In [56] wird eine exakte

[1] Wenn nichts anderes angegeben ist, sind die Begriffe „kurz“ und „lang“ auf die euklidische Norm bezogen. Ebenso bezeichnet $\|v\|$ die euklidische Norm eines Vektors v.

[2] Eine genaue Definition ist im nächsten Abschnitt angegeben.

Fehleranalyse durchgeführt, so daß die Korrektheit des Festkomma-Algorithmus bewiesen werden kann. Bei der Gleitkommavariante ist keine Fehleranalyse angegeben. In der Praxis ist der Festkomma-Algorithmus um 10 % langsamer als der Gleitkomma-Algorithmus [57].

Bei der Lösung großer Probleme, d.h. Probleme mit einer hohen Dimension (z.B. 100) oder mit sehr großen Zahlen (auftretende Zahlen können 100 Stellen haben und mehr), sind die Rechenzeiten trotz der Verwendung von Gleitkomma-Arithmetik sehr hoch (mehrere Tage bis hin zu Wochen). Daher interessiert man sich auch für parallele Algorithmen zur Basisreduktion.

Zeitgleich zu der Entstehung der vorliegenden Arbeit wurden von Roch und Villard parallele Algorithmen entwickelt [72, 50]. Diese weisen aber den oben erwähnten Nachteil auf, mit der langsamen, exakten Arithmetik zu arbeiten. Außerdem konnten Roch und Villard nicht formal beweisen, daß ihr Algorithmus bei der Verwendung von n^2 Prozessoren[3] effizient ist.

In der vorliegenden Arbeit wird die Effizienz des Verfahrens formal bewiesen. Außerdem wird eine bessere Analyse der Zahlengröße der entstehenden Zahlen durchgeführt. Die Zahlenlänge ist bei großen Zahlen bestimmend für die Laufzeit. Auch liegt das Hauptaugenmerk dieser Arbeit auf praktischen parallelen Algorithmen, die die schnellere Gleitkommadarstellung benutzen, und deren Implementierung. Im Gegensatz zu Berechnungen mit exakter Arithmetik entstehen dabei zusätzliche Probleme durch Rundungsfehler.

Ein erster Ansatz zu solchen Algorithmen besteht in der direkten Parallelisierung des Gleitkomma-Algorithmus von Schnorr und Euchner [57], die in Abschnitt 3.4 beschrieben wird. Dies ist unter der Verwendung von n Prozessoren möglich. Die Methode hat aber zwei Nachteile.

- Es entsteht ein hoher Kommunikationsaufwand.
- Die Methode ist nicht anwendbar für den Fall von n^2 Prozessoren.

Daher wurden grundsätzlich andere Methoden entwickelt, die im Fall von n Prozessoren weniger Kommunikationsaufwand erfordern, und die sich auch mit n^2 Prozessoren effizient parallelisieren lassen.

Unabhängig hat Joux [28, 27] einen parallelen Algorithmus für n^2 Prozessoren mit approximativer Arithmetik angegeben, für den er ähnlich zu Schnorr eine exakte Fehleranalyse durchführen konnte. Aber auch in der Arbeit von Joux konnte die Effizienz nicht bewiesen werden. Ein Schwerpunkt der Arbeit von Joux ist eine Methode, wie bei der parallelen Größenreduktion (ein Unterprogramm des LLL-Algorithmus) größere Zahlen als bei der seriellen Größenreduktion vermieden werden können. Parallele Implementierungen liegen jedoch nicht vor.

Das weitere Kapitel ist wie folgt gegliedert: Im nächsten Abschnitt werden die wichtigsten Grundlagen der Basisreduktion und serielle Algorithmen dargestellt. In 3.3 werden das parallele Programmiermodell, das den theoretischen Betrachtungen zugrunde liegt, und eine grundsätzliche Vereinbarung zur Messung der Zeitkomplexität der Algorithmen angegeben. Die direkte Parallelisierung des Schnorr-Euchner Algorithmus wird in Abschnitt

[3] n bezeichne die Dimension des Gitters

3.4 beschrieben. Dann wird in Abschnitt 3.5 ein neuer Algorithmus, der auch zur Parallelisierung mit n^2 Prozessoren geeignet ist, vorgestellt. Die Effizienz dieses Algorithmus wird formal bewiesen. Eine zugehörige Gleitkommavariante wird beschrieben und eine Fehleranalyse der entstehenden Rundungsfehler durchgeführt. Außerdem werden praktische Ergebnisse, die mit dem Algorithmus erzielt wurden, vorgestellt. In Abschnitt 3.6 wird die parallele Implementierung des neuen Algorithmus auf einem Prozessornetzwerk mit n^2 Prozessoren dargestellt. Außer der Parallelisierung des Schnorr-Euchner Algorithmus gibt es noch andere verschiedene Parallelisierungsmöglichkeiten bei der Verwendung von n Prozessoren. Diese werden in Abschnitt 3.7 diskutiert. In Abschnitt 3.8 wird die Größe der während der Berechnungen entstehenden Zahlen und deren Auswirkungen auf die Effizienz der Algorithmen berechnet. Dabei wird eine genauere Analyse als in [72] durchgeführt. Die Methode von Joux zur Vermeidung großer Zahlen bei der parallelen Größenreduktion wird kurz vorgestellt. Praktische parallele Implementierungen und Ergebnisse für verschiedene Parallelrechner und auf einem Workstation-Netz werden in Kapitel 3.9 dargestellt und diskutiert. Abschließend werden die wichtigsten Ergebnisse zusammengefaßt.

3.2 Grundlagen der Basisreduktion in Gittern

Das Problem, kurze Basen in Gittern zu finden, wird in der Literatur ausgiebig behandelt. Zur Grundlage werden z.B. die Bücher [44, 24, 59, 12, 11] und die Originalarbeit [36] empfohlen. In der vorliegenden Arbeit werden nur die Grundbegriffe dargestellt.

3.2.1 Gitter und Basen

Definition 3.1 *Seien $b_1, \ldots, b_n \in \mathbf{R}^n$ n linear unabhängige Vektoren[4]. Dann heißt L mit*

$$L = L(b_1, \ldots, b_n) = \left\{ \sum_{i=1}^{n} \lambda_i b_i | \lambda_i \in \mathbf{Z}, i = 1, \ldots, n \right\} \tag{3.1}$$

Gitter[5] mit der Basis $(b_1, \ldots, b_n)$.

Im folgenden werden Gitter mit ganzzahligen Basen, d.h. $b_1, \ldots, b_n \in \mathbf{Z}^n$, betrachtet.

Gitter finden z.B. Anwendungen bei ganzzahligen Problemen, z.B. bei der Lösung ganzzahliger Gleichungssysteme (siehe Anhang B).

Ziel der Basisreduktion ist es, kurze Vektoren des Gitters zu finden, bzw. eine Basis mit kurzen Vektoren. Dazu muß die Basis mit Operationen transformiert werden, die das erzeugte Gitter nicht verändern. Es ist leicht einzusehen, daß die folgenden Operationen diese Bedingung erfüllen:

- Multipliziere eine Spalte b_i mit -1
- Vertausche zwei Spalten b_i und b_j

[4] Die hier angegebenen Verfahren lassen sich auch leicht auf $m < n$ linear unabhängige Vektoren verallgemeinern.

[5] In der englischsprachigen Originalliteratur werden Gitter *lattice* genannt.

- Addiere auf Spalte i ein ganzzahliges Vielfaches einer anderen Spalte j, also

$$b_i \leftarrow b_i + \lambda b_j \qquad \text{für} \quad \lambda \in \mathbf{Z}$$

Allgemein können solche Spaltenoperationen auf der Matrix als Multiplikation der Basis mit einer Transformationsmatrix von rechts dargestellt werden. Beispielsweise kann die Multiplikation einer Spalte i mit -1 durch die Rechtsmultiplikation mit der Einheitsmatrix, wobei das i-te Diagonalelement -1 statt 1 ist, beschrieben werden. Die obigen Operationen haben die Eigenschaft, daß die zugehörgen Transformationsmatrizen ganzzahlig sind und Determinante ± 1 haben. Dies nennt man allgemein unimodulare Transformation.

Definition 3.2 *Eine Matrix $U \in \mathbf{Z}^{n\times n}$ heißt unimodular, wenn gilt*

$$|\det U| = 1$$

Die entscheidende Eigenschaft unimodularer Matrizen ist, daß die Inverse wiederum ganzzahlig (und damit unimodular) ist. Dies folgt leicht aus der Cramerschen Regel (s. Anhang A). Intuitiv gesprochen ändert sich daher durch eine Multiplikation mit einer unimodularen Matrix die Menge der ganzzahlig erreichbaren Punkte nicht. Formal gilt:

Satz 3.1 *Seien $(b_1, \ldots, b_n) \in \mathbf{Z}^{n\times n}$ und L das erzeugte Gitter. Dann gilt*

- *$(b_1, \ldots, b_n)U$ für eine unimodulare Matrix $U \in \mathbf{Z}^{n\times n}$ ist ebenfalls eine Basis von L, und*
- *ist $(b'_1, \ldots b'_n)$ eine Basis von L, dann gibt es eine unimodulare Matrix $U \in \mathbf{Z}^{n\times n}$ mit $(b_1, \ldots, b_n)U = (b'_1, \ldots b'_n)$,*

d.h. alle Basen von L entstehen genau durch unimodulare Transformationen auf der Basis $(b_1, \ldots, b_n)$.

Beweis:

- Sei L das von $(b_1, \ldots, b_n)$ erzeuge Gitter und L' das von $(b_1, \ldots, b_n)U$ erzeugte.
  - Sei $x \in L$.
    Dann gibt es ein $z \in \mathbf{Z}^n$ mit

$$\begin{aligned} (b_1, \ldots, b_n)z &= x \\ (b_1, \ldots, b_n)UU^{-1}z &= x \end{aligned}$$

    Da U^{-1} ganzzahlig ist, gibt es somit ein $z' = U^{-1}z$ mit

$$(b_1, \ldots, b_n)Uz' = x$$

    Daher ist x in L'.
  - Sei $x \in L'$.
    Setze analog $z' = Uz$, was ebenfalls ganzzahlig ist. Somit gilt $x \in L$.

- Seien $(b_1, \ldots, b_n)$ und $(b'_1, \ldots b'_n)$ zwei Basen von L. Da $(b_1, \ldots, b_n)$ eine Basis ist, gibt es eine ganzzahlige Matrix M mit $(b'_1, \ldots b'_n) = (b_1, \ldots, b_n)M$. Umgekehrt ist $(b'_1, \ldots b'_n)$ eine Basis, und somit gibt es eine ganzzahlige Matrix N mit $(b_1, \ldots, b_n) = (b'_1, \ldots b'_n)N$. Dann gilt aber

$$(b_1, \ldots, b_n) = (b'_1, \ldots b'_n)N = (b_1, \ldots, b_n)MN$$

  Damit ist MN die Einheitsmatrix, und M und N sind unimodular.

■

Der Betrag der Determinanten einer Basis eines Gitters ist somit unabhängig von der gewählten Basis und wird daher als Determinante des Gitters definiert.

$$\det L := |\det(b_1, \ldots, b_n)| \tag{3.2}$$

Geometrisch entspricht die Gitterdeterminante dem Volumen des von $b_1, \ldots, b_n$ aufgespannten Parallelotops.

Für die Größe der Basisvektoren gilt folgende untere Schranke (Hadamard):

$$\det L \leq \|b_1\| \cdot \ldots \cdot \|b_n\| \tag{3.3}$$

Dabei gilt Gleichheit, wenn die Vektoren paarweise senkrecht aufeinander stehen, d.h. orthogonale Basen sind in diesem Sinne optimal.

3.2.2 Reduzierte Basen

Grundsätzliche Idee ist der Gauß-Algorithmus für den zweidimensionalen Fall, d.h. Ausgangspunkt sind zwei zweidimensionale Vektoren [28]. Der kleinere Vektor wird vom größeren subtrahiert, bis der entstehende Vektor möglichst klein ist. Dies ist dann der Fall, wenn er möglichst senkrecht auf dem kleineren Vektor steht. Wenn der dadurch entstandene Vektor kürzer als der vorher kürzere Vektor ist, wird das Verfahren wiederholt, andernfalls ist man fertig.

Verallgemeinert man eine ähnliche Idee (siehe auch [12]) auf den mehrdimensionalen Fall, so erhält man folgende Grundideen:

1. Bei einer orthogonalen Basis gilt

$$\det L = \|b_1\| \cdot \|b_2\| \ldots \cdot \|b_n\|, \tag{3.4}$$

   d.h. die Hadamardsche Ungleichung (3.3) ist zu Gleichheit erfüllt. Intuitiv gesprochen ist somit eine Basis umso besser, je „orthogonaler“ sie ist. Daher betrachtet man die Gram-Schmidt Orthogonalisierung der Gitterbasis und versucht, die Gitterbasis der Gram-Schmidt Orthogonalisierung „anzupassen“. Die Gram-Schmidt Orthogonalisierung einer Matrix wird iterativ bestimmt. Seien $b^*_1, \ldots, b^*_i$ die ersten i Vektoren der Gram-Schmidt Orthogonalisierung. Der Vektor b^*_{i+1} ist dann die

„Komponente“ von b_{i+1}, die senkrecht auf dem von $b_1^*, \ldots, b_i^*$ aufgespannten Raum steht, oder formal

$$b_1^* = b_1 \tag{3.5}$$

$$b_i^* = b_i - \sum_{j=1}^{i-1} \mu_{i,j} b_j^* \qquad \text{für} \quad i = 2, \ldots, n \tag{3.6}$$

mit den Gram-Schmidt Koeffizienten $\mu_{i,j} = \frac{b_i^{\mathrm{T}} b_j^*}{||b_j^*||^2}$. Wenn man $\mu_{i,i} = 1$ und $\mu_{i,j} = 0$ für $i < j$ setzt und $M = (\mu_{i,j})_{1 \le i,j \le n}$ die Matrix der Gram- Schmidt Koeffizienten bezeichnet, kann man obige Beziehung in Matrixschreibweise schreiben:

$$(b_1, \ldots, b_n) = (b_1^*, \ldots, b_n^*) M^{\mathrm{T}}, \tag{3.7}$$

d.h. die Matrix $(b_1, \ldots, b_n)$ ist das Produkt einer Matrix, deren Vektoren paarweise senkrecht aufeinander stehen, und einer rechten oberen Dreiecksmatrix, auf deren Diagonalen nur Einsen stehen. Im folgenden bezeichne immer $(b_1, \ldots, b_n)$ die (aktuelle) Gitterbasis, $(b_1^*, \ldots, b_n^*)$ die dazugehörende Gram-Schmidt Orthogonalisierung, M die Matrix der Gram-Schmidt Koeffizienten und μ_j die j-te Spalte von M^{T}. Für die Gram-Schmidt Orthogonalisierung gilt:

$$\det L = |\det(b_1^*, b_2^*, \ldots, b_n^*)| = ||b_1^*|| \cdot ||b_2^*|| \cdot \ldots \cdot ||b_n^*|| \tag{3.8}$$

Die Basis $(b_1, \ldots, b_n)$ ist nun „möglichst orthogonal“, wenn die Gram-Schmidt Koeffizienten möglichst klein sind. Dies führt zu folgender Definition:

Definition 3.3 (Größenreduziert [57]) *Eine Basis $(b_1, \ldots, b_n)$ heißt größenreduziert (bzw. die Matrix M), wenn für die Gram-Schmidt Koeffizienten gilt:*

$$|\mu_{i,j}| \le \frac{1}{2} \qquad \textit{für alle} \quad 1 \le j < i \le n \tag{3.9}$$

Eine Basis kann mit Hilfe von unimodularen Spaltenoperationen sehr leicht größenreduziert werden:

Algorithmus 3.1

Größenreduktion der Basis $(b_1, \ldots, b_n)$ [36]

for $i = 2$ to n do
  Größenreduziere Spalte i der Basis und der Matrix M^{T}:
  for $j = i - 1$ downto 1 do
    Größenreduziere j-tes Element $\mu_{i,j}$ der Spalte i:
    Sei $\lceil \mu_{i,j} \rfloor$ die nächste ganze Zahl zu $\mu_{i,j}$
    $b_i \leftarrow b_i - \lceil \mu_{i,j} \rfloor b_j$
    $\mu_i \leftarrow \mu_i - \lceil \mu_{i,j} \rfloor \mu_j$
    Beachte: $\mu_{j,j} = 1$. Damit gilt jetzt: $|\mu_{i,j}| \le \frac{1}{2}$
  od
od

Bei der Größenreduktion wird die Gram-Schmidt Orthogonalisierung $(b_1^*, \ldots, b_n^*)$ nicht geändert.

2. Sei nun die Basis größenreduziert. Wie man an (3.5) und (3.6) sieht, ist die Gram-Schmidt Orthogonalisierung abhängig von der Ordnung der Vektoren der Basis. Vertauscht man zwei Vektoren b_i und b_{i+1} in der Basis und berechnet die Gram-Schmidt Orthogonalisierung neu, so ändern sich auch die Vektoren b_i^* und b_{i+1}^*. Dies soll nun genauer ausgeführt werden. Seien $(b_1, \ldots, b_n)$ eine größenreduzierte Gitterbasis und $(b_1^*, \ldots, b_n^*)$ die zugehörige Gram-Schmidt Orthogonalisierung und $(b_1^{*'}, \ldots, b_n^{*'})$ die Gram-Schmidt Orthogonalisierung nach dem Tausch von b_i und b_{i+1}. Dann gilt (siehe [24]):

   - Aus der Gram-Schmidt-Darstellung sieht man, daß sich in der Orthogonalbasis nur b_i^* und b_{i+1}^* ändern.
   - Das Produkt der Quadratnormen der Vektoren der Orthogonalbasis bleibt konstant (Quadrat der Gitterdeterminanten). Damit bleibt das Produkt $\|b_i^*\|^2 \cdot \|b_{i+1}^*\|^2$ unverändert.
   - Aus den Gram-Schmidt-Formeln folgt:

$$\begin{aligned} b_i^{*'} &= b_{i+1} - \sum_{j=1}^{i-1} \mu_{i+1,j} b_j^* \\ b_{i+1}^* &= b_{i+1} - \sum_{j=1}^{i-1} \mu_{i+1,j} b_j^* - \mu_{i+1,i} b_i^* \end{aligned}$$

     Und somit gilt:

$$b_i^{*'} = b_{i+1}^* + \mu_{i+1,i} \cdot b_i^*$$

Zusammenfassend gilt also nach einem Tausch von b_i und b_{i+1}:

$$b_j^{*'} = b_j^* \quad \text{für} \quad j \in \{1, \ldots, n\} \setminus \{i, i+1\} \tag{3.10}$$

$$b_i^{*'} = b_{i+1}^* + \mu_{i+1,i} \cdot b_i^* \tag{3.11}$$

$$||b_i^{*'}||^2 \cdot ||b_{i+1}^{*'}||^2 = ||b_i^*||^2 \cdot ||b_{i+1}^*||^2 \tag{3.12}$$

Gemäß den Formeln der Gram-Schmidt Orthogonalisierung (3.5) und (3.6) sind die Vektoren der Gitterbasis $(b_1, \ldots, b_n)$ umso kürzer, je kürzer die vorderen Vektoren in der Gram-Schmidt Orthogonalisierung sind. Es ist daher sinnvoll, zwei Vektoren b_i und b_{i+1} in der Basis zu tauschen, wenn dadurch der „linke“ Vektor b_i^* der Gram-Schmidt Orthogonalisierung kürzer wird.

Eine weitere Motivation, in der Gram-Schmidt Orthogonalisierung „kleine Vektoren nach links zu schaffen“, ist die folgende. Es gilt:

- Der kürzeste Vektor des Gitters ist stets größer als das Minimum der $\|b_i^*\|$ [24].
- $\|b_1\| = \|b_1^*\|$

Wenn man also eine Basis gefunden hat, so daß in der zugehörigen Gram-Schmidt Orthogonalisierung die Norm des ersten Vektors kleiner ist als die von allen anderen Vektoren der Gram-Schmidt Orthogonalisierung, dann ist der erste Vektor der Basis b_1 der kürzeste Vektor des Gitters.

Gilt nun

$$\|b_i^*\|^2 > 2\|b_{i+1}^*\|^2 \tag{3.13}$$

so gilt nach (3.9) und (3.11)

$$\|b_i^{*'}\|^2 \leq \frac{3}{4}\|b_i^*\|^2 \tag{3.14}$$

und somit ist es sinnvoll, b_i und b_{i+1} zu tauschen.

Eine wichtige Beobachtung für spätere Konvergenz- und Komplexitätsbeweise ist, daß bei einem Tausch von b_i und b_{i+1} das Produkt

$$\|b_1^*\|^2 \cdot \ldots \cdot \|b_i^*\|^2$$

um mindestens den Faktor $\frac{3}{4}$ fällt (s. (3.12) und (3.14)). Alle anderen Produkte bleiben unverändert.

Die obigen Überlegungen führen zu der folgenden Definition einer reduzierten Basis[6]:

Definition 3.4 (Reduzierte Basis [36, 59]) *Eine Gitterbasis* $(b_1, \ldots, b_n)$ *heißt* reduziert, *falls*

1. *sie größenreduziert ist und*
2. $\|b_i^*\|^2 \leq 2\|b_{i+1}^*\|^2$ *für alle* $1 \leq i < n$ [7]

Insbesondere hat man nach dem folgenden Satz einen kurzen Vektor des Gitters gefunden.

Satz 3.2 ([36]) *Für eine reduzierte Basis* $(b_1, \ldots, b_n)$ *gilt*

$$||b_1|| \leq 2^{\frac{n-1}{4}}(\det L)^{\frac{1}{n}} \tag{3.15}$$

Diese Schranke unterscheidet sich nur durch den Vorfaktor von einer optimalen bekannten Schranke (siehe [24]).

Reduzierte Basen besitzen noch andere Eigenschaften, die aber an dieser Stelle nicht interessant sind [36, 24], da der Schwerpunkt auf (parallele) Reduktionsalgorithmen gelegt werden soll. So kann z.B. eine obere Schranke für das Produkt $\|b_1\| \cdot \ldots \cdot \|b_n\|$ angegeben werden.

3.2.3 Allgemeine Struktur von Basisreduktionsalgorithmen

Die Überlegungen des letzten Abschnitts führen zu folgendem Grundalgorithmus:

Algorithmus 3.2

[6] Hier wird die Definition aus [59] benutzt, die der Originaldefinition aus [36] sehr ähnlich ist. Die Aussagen aus [36], die hier zitiert werden, sind auch mit der hier benutzten Definition richtig.

[7] Die Originalbedingung aus [36] ist $\|b_{i+1}^* + \mu_{i+1,i}b_i^*\|^2 \geq \delta\|b_i^*\|^2$ mit $\delta = \frac{3}{4}$.

Grundlegender Basisreduktionsalgorithmus (nach [59, 24])
Eingabe: Gitterbasis $b_1, b_2, \ldots, b_n$
Ausgabe: Reduzierte Basis $b_1, b_2, \ldots, b_n$
Methode:

tauschen $\leftarrow$ wahr
while (tauschen) do

Berechne die Gram-Schmidt Orthogonalisierung
$||b_1^*||^2, \ldots, ||b_n^*||^2$ und M

Berechne die Größenreduktion gemäß Algorithmus 3.1

tauschen $\leftarrow$ falsch
if (es gibt einen Index i mit $||b_i^*||^2 > 2||b_{i+1}^*||^2$) then
tauschen $\leftarrow$ wahr
Tausche für einen solchen Index i die Vektoren b_i und b_{i+1}, d.h.
betrachte die neue Basis $(b_1, \ldots, b_{i+1}, b_i, \ldots, b_n)$.
fi

od

Es ist sofort klar, daß die entstehende Gitterbasis reduziert ist, wenn der Algorithmus terminiert. Es muß also nur noch die Terminierung gewährleistet werden. Um zu beweisen, daß der Algorithmus terminiert und um gleichzeitig die Komplexität des Algorithmus abzuschätzen, wird eine obere Schranke für die Anzahl der durchzuführenden Tausche angegeben. Im folgenden sei B die größte Quadratnorm der Ausgangsbasis, d.h.

$$B = \max_{i=1\ldots n} \|b_i\|^2 \tag{3.16}$$

Nach der Konstruktion der Gram-Schmidt Orthogonalisierung gilt dann $\|b_i^*\|^2 \leq B$ für $i = 1, \ldots, n$. Für die Anzahl der Tausche gilt folgender Satz:

Satz 3.3 ([36]) *Algorithmus 3.2 terminiert nach höchstens $n^2 \log B$ Tauschoperationen*[8].

Beweis: Betrachte

$$D := ||b_1^*||^{2n} \cdot ||b_2^*||^{2(n-1)} \cdot \ldots \cdot ||b_n^*||^2$$

Zu Beginn gilt $D \leq B^{(n^2)}$. Wenn man von einer ganzzahligen Basis ausgeht, ist D ebenfalls ganzzahlig und bleibt während der Ausführung des Algorithmus ganzzahlig, d.h. insbesondere $D \geq 1$ (siehe [24]). Gemäß (3.14) und (3.12) fällt D bei jedem Tausch um einen Faktor kleiner $\frac{3}{4}$. Sei l die Anzahl der durchgeführten Tausche. Dann gilt

$$1 \leq \left(\frac{3}{4}\right)^l D \leq B^{(n^2)} \left(\frac{3}{4}\right)^l \Rightarrow l \leq n^2 \log B$$

■

Pro Tausch werden $O(n^3)$ arithmetische Operationen durchgeführt. Dies führt also zu folgender Komplexität:

[8] Im folgenden bezeichne log den Logarithmus zur Basis $\frac{4}{3}$.

Satz 3.4 *Algorithmus 3.2 benötigt $O(n^5 \log B)$ arithmetische Operationen, um eine reduzierte Basis zu finden.*

In Algorithmus 3.2 muß die Gram-Schmidt Orthogonalisierung nicht in jedem Schleifendurchgang neu berechnet werden. Es genügt, die Gram-Schmidt Orthogonalisierung am Anfang einmal zu berechnen und nach einem Tausch wiederherzustellen. Außerdem müssen zur Funktion des Algorithmus nicht die b_i^*, sondern nur die Quadratnormen $\|b_i^*\|^2$ verwaltet werden. Bei einem Tausch von b_i und b_{i+1} ändern sich diese Werte der Gram-Schmidt Orthogonalisierung gemäß (3.17) - (3.25) [36] (vergleiche auch (3.10) - (3.12)). In den Formeln sind mit $'$ die neuen Werte bezeichnet.

$$b_j^{*'} = b_j^* \quad \text{für} \quad j \in \{1,\ldots,n\} \setminus \{i, i+1\} \tag{3.17}$$

$$\|b_i^{*'}\|^2 = \|b_{i+1}^*\|^2 + \mu_{i+1,i}^2 \cdot \|b_i^*\|^2 \tag{3.18}$$

$$\|b_{i+1}^{*'}\|^2 = \frac{\|b_i^*\|^2 \|b_{i+1}^*\|^2}{\|b_i^{*'}\|^2} \tag{3.19}$$

$$\mu'_{i+1,i} = \frac{\mu_{i+1,i}\|b_i^*\|^2}{\|b_i^{*'}\|^2} \tag{3.20}$$

$$\mu'_{j,i} = \mu_{j,i}\mu'_{i+1,i} + \mu_{j,i+1}\frac{\|b_{i+1}^*\|^2}{\|b_i^{*'}\|^2} \quad \text{für alle} \quad j > i+1 \tag{3.21}$$

$$\mu'_{j,i+1} = \mu_{j,i} - \mu_{j,i+1}\mu_{i+1,i} \quad \text{für alle} \quad j > i+1 \tag{3.22}$$

$$\mu'_{i,j} = \mu_{i+1,j} \quad \text{für} \quad 1 \le j < i \tag{3.23}$$

$$\mu'_{i+1,j} = \mu_{i,j} \quad \text{für} \quad 1 \le j < i \tag{3.24}$$

$$\mu'_{k,j} = \mu_{k,j} \quad \text{für} \quad 1 \le j < k \le n \quad \text{und} \quad \{k,j\} \cap \{i+1, i\} = \emptyset \tag{3.25}$$

Die Reparierung der Gram-Schmidt Orthogonalisierung nach einer Phase mit diesen Formeln kostet $O(n^2)$ arithmetische Operationen statt $O(n^3)$ für eine Neuberechnung. Eine Verbesserung der asymptotischen Komplexität des Algorithmus ist damit aber nicht erreichbar, da die Komplexität einer Phase durch die Berechnung der Größenreduktion bestimmt wird.

Der LLL-Algorithmus [36] verfolgt zusätzlich eine bessere Strategie, die auch zu einer asymptotisch besseren Laufzeit führt. Er geht in Stufen vor. Pro Stufe k wird nur die Spalte k der Gram-Schmidt Orthogonalisierung und deren Größenreduktion berechnet. Falls die Spalten k und $k-1$ getauscht werden, wird k um eins zurückgesetzt, andernfalls um eins erhöht. Es gilt also als Invariante, daß die Basis bestehend aus den Vektoren $1,\ldots,k-1$ bereits reduziert ist. Die Grundstruktur ist in Algorithmus 3.3 dargestellt. Algorithmen, die nach diesem Schema vorgehen, sollen im folgenden *Stufenalgorithmen* genannt werden. Bei Stufenalgorithmen werden also nur noch $O(n^2)$ arithmetische Operationen pro Tausch durchgeführt. Damit ergibt sich die folgende Komplexität.

Satz 3.5 ([36]) *Stufenalgorithmen gemäß Algorithmus 3.3 benötigen $O(n^4 \log B)$ arithmetische Operationen, um eine reduzierte Basis zu finden.*

Eine parallele Version des LLL-Algorithmus ist in [72] beschrieben.

Algorithmus 3.3

Ein Stufenalgorithmus gemäß [36] zur Basisreduktion (nach [57])
Eingabe: Eine Basis $b_1, b_2, \ldots, b_n$
Ausgabe: Reduzierte Basis $b_1, b_2, \ldots, b_n$
Methode:

Initialisierung: $k \leftarrow 2$ (k bezeichnet die Stufe)
Berechne die Gram-Schmidt Koeffizienten M und $||b_i^*||^2$ für $i = 1, \ldots, n$.
while $(k \leq n)$ do
    Größenreduziere die Spalte k, d.h. b_k und μ_k
    gemäß Algorithmus 3.1

    if $(||b_{k-1}^*||^2 > 2||b_k^*||^2)$ then
        tausche b_{k-1} und b_k und verbessere die Gram-Schmidt
        Orthogonalisierung entsprechend
        $k \leftarrow \max\{k-1, 2\}$
    else
        $k \leftarrow k + 1$
    fi
od

Wenn man von einer ganzzahligen Gitterbasis ausgeht, treten während der Berechnung nur rationale Zahlen auf. Man kann also die gesamte Berechnung mit exakter ganzzahliger Arithmetik durchführen. Die Größe der dabei entstehenden ganzen Zahlen wird in Abschnitt 3.8 behandelt. Die Rechnung mit genauer ganzzahliger Arithmetik ist aber sehr langsam. Daher haben Schnorr und Euchner [57] eine Variante des LLL-Algorithmus entwickelt, die mit der schnelleren Gleitkomma-Arithmetik arbeitet. Nur noch die Gitterbasis selbst wird in exakter Darstellung gehalten, da kleinste Fehler in der Gitterbasis zu einem vollkommen anderen Gitter führen. Die Quadratnormen der Gram-Schmidt Orthogonalisierung $\|b_i^*\|^2$ und die Koeffizienten $\mu_{i,j}$ werden hingegen in Gleitkommadarstellung berechnet. Treten zu große oder zu kleine Zahlen auf, sind die entstehenden Rundungsfehler evtl. zu groß. Dann wird entweder exakt gerechnet oder der Index k zurückgesetzt. Die Grundstruktur des Schnorr-Euchner-Algorithmus sieht folgendermaßen aus:

Algorithmus 3.4

Gleitkomma-Basisreduktionsalgorithmus nach Schnorr/Euchner [57]
Eingabe: Gitterbasis $b_1, b_2, \ldots, b_n$
Ausgabe: Mit δ reduzierte Gitterbasis (siehe [57]) $b_1, b_2, \ldots, b_n$
Bezeichnungen: Gleitkommawerte werden mit $'$ bezeichnet,
    c_i entspricht $\|b_i^{*'}\|^2$,
    τ bezeichne die Anzahl der Präzisionsbits der Gleitkomma-Darstellung
Methode:

Initialisierungen:
$k \leftarrow 2$
for $(i = 1, \ldots, n)$ do $b_i' \leftarrow (b_i)'$ od

for $(i = 1, \ldots, n)$ do
  for $(j = 1, \ldots, n)$ do
    $\mu_{i,j} \leftarrow 0$
  od
od
for $(i = 1, \ldots, n)$ do $\mu_{i,i} \leftarrow 1$ od

while $(k \leq n)$ do

  (Berechnung der Spalte μ_k und der Norm c_k)
  if $(k = 2)$ then $c_1 \leftarrow \|b'_1\|^2$ fi
  $c_k \leftarrow \|b'_k\|^2$
  for $(j = 1, \ldots, k-1)$ do
    if $(|{b'_k}^{\mathrm{T}} b'_j| < 2^{\frac{-\tau}{2}} \|b'_k\| \|b'_j\|)$ then
      $s \leftarrow (b_k^{\mathrm{T}} b_j)'$
    else
      $s \leftarrow ({b'_k}^{\mathrm{T}} b'_j)$
    fi
    $\mu_{k,j} \leftarrow (s - \sum_{i=1}^{j-1} \mu_{j,i}\mu_{k,i}c_i)/c_j$
    $c_k \leftarrow c_k - \mu_{k,j}^2 c_j$
  od

  (Größenreduktion der Spalte b_k)
  $F_c \leftarrow F_r \leftarrow false$
  for $(j = k-1, \ldots, 1)$ do
    if $(|\mu_{k,j}| > \frac{1}{2})$ then
      $\mu \leftarrow \lceil \mu_{k,j} \rfloor$;
      $F_r \leftarrow true$;
      if $(|\mu| > 2^{\frac{\tau}{2}})$ then $F_c \leftarrow true$ fi;
      $\mu_k \leftarrow \mu_k - \mu\mu_j$
      $b_k \leftarrow b_k - \mu b_j$
    fi
  od
  if (F_r) then $b'_k \leftarrow (b_k)'$ fi
  if (F_c) then
    $k \leftarrow \max\{k-1, 2\}$
  else
    (Tausche oder inkrementiere k)
    if $(\delta c_{k-1} > c_k + \mu_{k,k-1}^2 c_{k-1})$ then
      tausche b_k und b_{k-1} und tausche b'_k und b'_{k-1}
      $k \leftarrow \max\{k-1, 2\}$
    else
      $k \leftarrow k+1$
    fi
  fi
od

3.3 Das verwendete parallele Programmiermodell

Es soll nun kurz das parallele Programmiermodell vorgestellt werden, das in den folgenden (eher theoretischen) Ausführungen benutzt wird (s. auch [5]). Praktische Programmiermodelle, Implementierungen und Ergebnisse werden in Kapitel 3.9 behandelt.

Ein paralleles System besteht aus einer Menge von p Berechnungseinheiten (Prozessoren). Diese können vollkommen unabhängig voneinander Berechnungen durchführen. Jede Berechnungseinheit verfügt über ihren eigenen (lokalen) Speicher, auf den nur sie zugreifen kann. Die Berechnungseinheiten sind über ein Netz verbunden, über das sie Nachrichten austauschen können. Dies kann z.B. dadurch geschehen, daß die Berechnungseinheiten linear angeordnet sind, wobei jede mit ihrer linken und rechten Nachbarin verbunden ist. Man spricht dann von einem *linearen Prozessorfeld*. Ist auch die erste Berechnungseinheit mit der letzten verbunden, so spricht man von einem *Ring*.

Wie bereits im letzten Abschnitt geschehen, werden im folgenden die Berechnungskosten der Algorithmen (seriell und parallel) mit Hilfe der Anzahl der im schlechtesten Fall durchgeführten arithmetischen Operationen gemessen. Bei parallelen Algorithmen gemäß dem obigen Modell müssen auch die Zeiten für den Nachrichtenaustausch beachtet werden. Dabei wird angenommen, daß zum Versenden einer Nachricht zunächst eine konstante Zeit benötigt wird, um das Versenden vorzubereiten. Diese Zeit nennt man „Startup-Zeit“ [5]. Die weitere Versendungszeit hängt dann linear[9] von der Länge der Nachrichten ab. Um diese Länge abzuschätzen wird im folgenden zunächst die Anzahl der zu versendenden Zahlen gezählt. Dabei gelte folgende Notation:

- α: Zeit für eine arithmetische Operation auf Zahlen in genauer Darstellung
- α': Zeit für eine arithmetische Operation auf Zahlen in Gleikommadarstellung
- β: Startup-Zeit beim Versenden einer Nachricht
- γ: Zeit zum Versenden einer Zahl in genauer Darstellung (ohne die Startup-Zeit)
- γ': Zeit zum Versenden einer Zahl in Gleikommadarstellung (ohne die Startup-Zeit)

Durch dieses Modell ist man bei der Analyse der Laufzeiten zunächst unabhängig davon, wie die arithmetischen Operationen (mit exakter Darstellung oder in Gleitkommadarstellung, seriell oder parallel (Hardware)) und die Kommunikation (Versenden von Nachrichten, hardwaremäßig über Busse) realisiert werden.

Bei der genauen Berechnung der („binären“) Kosten ist dann natürlich die Größe der entstehenden Zahlen zu beachten. So sind beispielsweise bei der Verwendung einer genauen, seriellen Standardarithmetik die Kosten einer Addition proportional zur Länge und die Kosten einer Multiplikation proportional zum Quadrat der Länge der binären Darstellung der Zahlen und die Kommunikationskosten proportional zur Länge, falls die Zeit zum Versenden einer Nachricht proportional zu deren Länge ist. Werden jedoch die arithmetischen Operationen parallel durchgeführt, so ist die Zeit proportional zum

[9] Die Annahme der Linearität ist in der Praxis nicht ganz richtig, stellt aber für ein theoretisches Modell eine gute Näherung dar [5].

Logarithmus der Länge, die Anzahl der Berechnungseinheiten jedoch hoch. Bei einer hardwaremäßigen Versendung von Nachrichten über Busse entsprechender Breite (Länge der binären Darstellung) kostet das Versenden einer Zahl konstant viel Zeit. Bei einer hardwaremäßigen Realisierung muß man aber die Größe der entstehenden Zahlen vorher kennen. Außerdem ist eine so große Zahl paralleler Leitungen unrealistisch.

Die Größe der Zahlen und die Auswirkungen auf die Laufzeit werden in Abschnitt 3.8 betrachtet (oder an entsprechender Stelle behandelt).

3.4 Parallelisierung des Schnorr/Euchner Algorithmus für n Prozessoren

Es wird nun eine Parallelisierung des Schnorr-Euchner Algorithmus für ein lineares Prozesssorfeld von $p \leq n$ Prozessoren[10] betrachtet. Ein lineares Feld hat den Vorteil, daß es eine sehr einfache Verbindungsstruktur besitzt und in viele Architekturen eingebettet werden kann, z.B. in Architekturen mit Hyperwürfeltopologie (s. z.B. [5]).

Die Grundstruktur des Schnorr-Euchner Algorithmus 3.4 sieht wie folgt aus.

Für einen Index k führe aus:

- Berechne die k-te Spalte der Gram-Schmidt-Koeffizienten und $\|b_k^*\|^2$.
- Größenreduziere diese Spalte.
- Tausche gegebenenfalls die Spalten k und $k-1$ und berechne k neu.

Im seriellen benötigen die ersten zwei Punkte $O(n^2)$ arithmetische Operationen. Diese Punkte sollen nun auf n Prozessoren (durchnumeriert von 1 bis n) parallelisiert werden. Dabei geschieht die Aufteilung der Matrix $(b_1, \ldots, b_n)$ der aktuellen Basis des Gitters und der Matrix M^{T} der Gram-Schmidt Koeffizienten *zeilenweise* auf die Prozessoren, d.h. Prozessor i übernimmt die Bearbeitung der i-ten Zeile. Außerdem kennt der Prozessor i die i-te Spalte der Matrix M^{T}. Diese wird für die Berechnung einer neuen Spalte benötigt. Genauer soll folgende Datenaufteilung gelten:

Invariante 3.1 *Vor Eintritt in die Stufe k des Algorithmus kennt der Prozessor i mit $1 \leq i \leq n$:*

- *den Wert des Index k*
- *die Reihe i der aktuellen Gitterbasis $(b_1, \ldots, b_n)$ in genauer Darstellung und in Gleitkommadarstellung $b'_1, \ldots, b'_n$*

Jeder Prozessor $i, i = 1, \ldots, k-2$ kennt darüberhinaus:

- *die Zeile i der bisher bekannten Matrix M^{T}, d.h. die Werte $\mu_{i+1,i}, \mu_{i+2,i}, \ldots, \mu_{k-1,i}$*

[10] Der Einfachheit halber wird angenommen, daß genau $p = n$ Prozessoren verfügbar sind. Andernfalls übernimmt ein Prozessor die Aufgabe von mehreren Prozessoren vorher.

- *die Norm des i-ten Vektors der Gram-Schmidt-Orthogonalisierung zum Quadrat (in Gleitkommadarstellung, s. Algorithmus 3.4):* c_i
- *den Wert* $||b'_i||$

Der Prozessor $k-1$ *kennt den Wert* c_{k-1}.

Jeder Prozessor $i, i = 1, \ldots, k-1$ *kennt die i-te Spalte der Matrix* M^{T}*, d.h. die Werte* $\mu_{i,1}, \ldots, \mu_{i,i-1}$.

3.4.1 Parallele Berechnung der k-ten Spalte der Orthogonalisierung

Es soll nun eine Parallelisierung der Berechnung der k-ten Spalte der Matrix M^{T} nach dem Schnorr/Euchner Algorithmus angegeben werden.

Algorithmus 3.5

$c_k \leftarrow \|b'_k\|^2$
for $(j = 1, \ldots, k-1)$ do
  if $(|({b'_k}^{\mathrm{T}} b'_j)| < 2^{\frac{-\tau}{2}} \|b'_k\| \|b'_j\|)$ then
    $s \leftarrow (b_k^{\mathrm{T}} b_j)'$
  else
    $s \leftarrow ({b'_k}^{\mathrm{T}} b'_j)$
  fi
  $\mu_{k,j} \leftarrow (s - \sum_{i=1}^{j-1} \mu_{j,i}\mu_{k,i}c_i)/c_j$
  $c_k \leftarrow c_k - \mu_{k,j}^2 \cdot c_j$
od

Dies geschieht in folgenden (parallelisierbaren) Schritten:

1. Prozessor j für $1 \leq j \leq k$ berechnet das Skalarprodukt $({b'_k}^{\mathrm{T}} b'_j)$. Prozessor $k-1$ berechnet zusätzlich $({b'_{k-1}}^{\mathrm{T}} b'_{k-1})$.
2. Verletzt ein Skalarprodukt die „Fehlerbedingung“, so berechne alle Skalarprodukte neu.
3. Prozessor j mit $1 \leq j \leq k-1$ berechnet $\mu_{k,j}$, $\mu_{k,j} \cdot c_j$ und $\mu_{k,j}^2 \cdot c_j$. Prozessor k berechnet c_k.

zu 1) Parallele Berechnung der Skalarprodukte

Die parallele Berechnung der Skalarprodukte kann wie eine parallele Matrix-Vektor-Multiplikation durchgeführt werden (s. [5, Seiten 73-76]). Sie geschieht in folgenden Schritten:

- Alle Prozessoren j mit $1 \leq j \leq n$ berechnen gleichzeitig die Produkte $b'_{k,j} \cdot b'_{1,j}, b'_{k,j} \cdot b'_{2,j}, \ldots, b'_{k,j} \cdot b'_{k,j}$ und $b'_{k-1,j} \cdot b'_{k-1,j}$. Dies kostet Berechnungszeit $k+1$ arithmetische Operationen.
- Die Aufsummierung der Werte entspricht einer „Multinode-Akkumulation“ und kann auf einem linearen Prozessorfeld in $n-1$ Schritten durchgeführt werden [5, Seite 44].

zu 2) Parallele Neuberechnung der Skalarprodukte

Von Prozessor k wird der Wert $||b'_k||^2$ zu den Prozessoren $k-1, \ldots, 1$ gesendet. Danach weiß Prozessor 1, ob ein Skalarprodukt neu berechnet werden muß. Diese Information wird wieder zu Prozessor n gesendet, so daß danach alle Prozessoren j für $1 \leq j \leq n$ die Information besitzen, ob eine Neuberechnung stattfinden muß. Ist dies der Fall, so wird Schritt 1 mit den exakten Werten der Basisvektoren des Gitters wiederholt. Andernfalls wird direkt Schritt 3 ausgeführt.

zu 3) Parallele Berechnung der $\mu_{k,j}$

Prozessor j mit $1 \leq j \leq k-1$ berechnet die Werte $\mu_{k,j}$, $\mu_{k,j} \cdot c_j$ und $\mu^2_{k,j} \cdot c_j$. Prozessor k berechnet c_k. Dabei gilt nach Schnorr/Euchner:

$$\begin{aligned} \mu_{k,j} &= ((b_k^{\mathrm{T}} b_j) - \sum_{i=1}^{j-1} \mu_{j,i}\mu_{k,i}c_i)/c_j \\ c_k &= ||b'_k||^2 - \sum_{j=1}^{k-1} \mu^2_{k,j} c_j \end{aligned}$$

Dabei ist zu beachten, daß gemäß der Invarianten 3.1 jeder Prozessor $j \leq k-1$ die Werte $\mu_{j,i}$ für $i < j$ kennt. Die obigen Berechnungen können dann folgendermaßen parallel durchgeführt werden.

Start: Prozessor 1 berechnet die Werte $\mu_{k,1}$, $\mu_{k,1} \cdot c_1$ und $\mu^2_{k,1} \cdot c_1$. und sendet sie an Prozessor 2.

Dann: Zu jedem Zeitpunkt t verfährt jeder Prozessor j mit $2 \leq j \leq k-1$ folgendermaßen:

- Erhält Prozessor j von Prozessor $j-1$ keine Nachricht, so tut er nichts.
- Erhält Prozessor j eine Nachricht von Prozessor $j-1$, die ursprünglich von Prozessor $i < j-1$ stammt, d.h. die die Werte $\mu_{k,i}$, $\mu_{k,i} \cdot c_i$ und $\mu^2_{k,i} \cdot c_i$ enthält, so macht er folgendes:
  - Berechne $\mu_{j,i} \cdot \mu_{k,i} \cdot c_i$ und subtrahiere diesen Wert von dem bisherigen Zwischenwert für $\mu_{k,j}$.
  - Sende die empfangene Nachricht an den Prozessor $j+1$.

- Von Prozessor $j-1$ kommt keine Nachricht mehr.
  - Prozessor j kann nun die Werte $\mu_{k,j}$, $\mu_{k,j} \cdot c_j$ und $\mu_{k,j}^2 \cdot c_j$ zu Ende berechnen.
  - Er sendet diese Werte als Nachricht an Prozessor $j+1$.

Prozessor k benutzt die empfangenen Werte zur Berechnung von c_k.

Laufzeit des Verfahrens:

Jeder oben beschriebene Schritt kostet konstant viel Zeit (Berechnung eines neuen Wertes und Versendung eines Paketes konstanter Größe.) Prozessor k muß zunächst $k-1$ Schritte auf die erste Nachricht warten. Er erhält dann nacheinander $k-1$ Nachrichten. Insgesamt werden also $2k-2$ Zeitschritte benötigt.

Damit ist die parallele Berechnung der k-ten Spalte der Gram-Schmidt Zerlegung abgeschlossen.

Zeitaufwand insgesamt zur Berechnung der k-ten Spalte auf einem linearen Feld (ohne Neuberechnung der Skalarprodukte):[11]

- *Berechnungszeit:* $n+6k$ arithmetische Operationen auf Gleitkommazahlen, d.h. Zeit $(n+6k)\alpha'$.
- *Kommunikationszeit:* $(n+6k)(\beta+\gamma')$

3.4.2 Parallele Größenreduktion der k-ten Spalte der μ-Matrix

Im nächsten Schritt des Schnorr-Algorithmus wird nun die neu berechnete Spalte μ_k größenreduziert, d.h. die Einträge werden betragsmäßig $\leq \frac{1}{2}$ gemacht. Der Basisvektor b_k wird entsprechend geändert (in der genauen Darstellung). Dabei merkt sich jeder Prozessor, ob überhaupt ein $|\mu| > \frac{1}{2}$ ist, d.h. ob eine Berechnung nötig ist, und ob bei der Berechnung ein zu großes $|\mu|$ im Sinne von Schnorr entstanden ist. Ebenso wird während der Größenreduktion jedem Prozessor mitgeteilt, ob die Tauschbedingung für die Spalten k und $k-1$ erfüllt ist.

Größenreduktion eines Elementes $\mu_{k,i}$ nach Schnorr [57]:

Algorithmus 3.6

if $(|\mu_{k,i}| > \frac{1}{2})$ then
  $\mu \leftarrow \lceil \mu_{k,i} \rfloor$;
  $F_r \leftarrow true$;
  if $(|\mu| > 2^{\frac{\tau}{2}})$ then $F_c \leftarrow true$ fi;
  $\mu_k \leftarrow \mu_k - \mu\mu_i$
  $b_k \leftarrow b_k - \mu b_i$
fi

[11] In den folgenden Angaben werden die konstanten Terme vernachlässigt.

Parallele Größenreduktion der Elemente $\mu_{k,i}$ für $i = k-1, \ldots, 1$:

Start: Prozessor k sendet an Prozessor $k-1$ den Wert c_k. Dieser macht dann folgendes:

- Er berechnet den reduzierten Wert von $\mu_{k,k-1}$.
- Er berechnet den neuen Wert von $b_{k,k-1}$.
- Er berechnet, ob die Tauschbedingung für die Spalten k und $k-1$ erfüllt ist.
- Er erstellt eine Nachricht bestehend aus dem (alten) Wert von $\lceil \mu_{k,k-1} \rfloor$ bzw. der Information, daß $\mu_{k,k-1}$ schon reduziert war und der Information, ob die Tauschbedingung erfüllt ist, und sendet diese an die Prozessoren $k-2$ und k.

In den *folgenden Zeitschritten* t machen die Prozessoren folgendes:

- *Prozessoren i mit $i < k-1$:*
  - Von Prozessor $i+1$ kommt eine Nachricht:
    Dann berechnet er, wenn nötig, die neuen Werte von $\mu_{k,i}$ und $b_{k,i}$ und sendet die erhaltene Nachricht an den Prozessor $i-1$ weiter.
  - Es kommt keine Nachricht mehr von Prozessor $i-1$:
    Dann reduziert er das Element $\mu_{k,i}$, falls nötig, und berechnet entsprechend den neuen Wert für $b_{k,i}$. Der Wert von $\lceil \mu_{k,i} \rfloor$ vor der Reduktion bzw. die Information, daß das Element $\mu_{k,i}$ nicht reduziert werden mußte, wird an die Prozessoren $i-1$ und $i+1$ gesendet.
- *Prozessoren i mit $i = 1, \ldots, n$:*
  Erhält Prozessor i von dem Prozessor $i-1$ eine Nachricht, so berechnet er mit deren Hilfe das Element $b_{k,i}$ neu, falls nötig, und sendet die empfangene Nachricht an den Prozessor $i+1$ weiter.

Beispiel 3.1 Der Datenfluß für den Fall $n = 5$ und $k = 4$ soll durch Tabelle 3.1 verdeutlicht werden. Jede Zeile der Tabelle entspricht einem Prozessor, der die entsprechende Zeile der Matrix verwaltet. Ein Pfeil bedeutet, daß der Prozessor in diesem Zeitschritt eine Nachricht in die entsprechende Richtung versendet. Der Buchstabe „g“ symbolisiert, daß der zugehörige Prozessor i mit der Größenreduktion des i-ten Elementes der Spalte k beginnt. Ein „b“ bedeutet, daß der Prozessor rechnet, aber nichts versendet.

Laufzeit für die Größenreduktion einer Spalte k:

Prozessor 1 muß zunächst $k-2$ Schritte auf die erste Nachricht von Prozessor 2 warten. Er erhält dann $k-2$ Nachrichten. Sodann schnürt er ein eigenes Paket und schickt es nach unten weiter. Nach weiteren $n-1$ Schritten wurde dieses auch von Prozessor n abgearbeitet. Das Verfahren dauert also $2(k-2)+n$ Schritte.

Nach der Reduktion „weiß“ jeder Prozessor:

P	$t=1$	$t=2$	$t=3$	$t=4$	$t=4$	$t=6$	$t=7$	$t=8$	$t=9$
1			b	b	g $\downarrow$				
2		$\uparrow$	g $\downarrow\uparrow$			$\downarrow$			
3	g$\downarrow\uparrow$			$\downarrow$			$\downarrow$		
4		$\downarrow$			$\downarrow$			$\downarrow$	
5			b			b			b

Tabelle 3.1: Parallele Größenreduktion einer Spalte mit Pipelining

- Ist ein zu großes μ aufgetreten?
- Wurde der Basisvektor b_k verändert?
- Gilt die Tauschbedingung?

Jeder Prozessor kann also folgendes durchführen:

- Setze $b'_{k,i} \leftarrow (b_{k,i})'$, falls nötig.
- Vertausche $b_{k,i}$ und $b_{k-1,i}$ und in Gleitkommadarstellung, falls die Tauschbedingung erfüllt ist, und kein zu großer Wert für μ aufgetreten ist.
- Berechne den neuen Wert von k.

Wurde k um eins erniedrigt, gilt die Invariante 3.1 auch für den neuen Wert von k. Anderfalls (k wurde um eins erhöht, insbesondere wurde nicht getauscht) muß noch die neu berechnete Spalte k (für den alten Wert von k) dem Prozessor k mitgeteilt werden, d.h. jeder Prozessor i mit $i < k$ sendet an den Prozessor k das neue Element $\mu_{k,i}$. Dies geht auf einem linearen Feld in Kommunikationszeit $k-1$. Danach gilt die Invariante 3.1 wieder für den neuen Wert von k, so daß mit der nächsten Stufe begonnen werden kann.

Für eine Gesamtstufe gilt somit folgende Komplexität:

Satz 3.6 *Eine Stufe des Schnorr-Euchner Algorithmus*[12] *kann auf einem linearen Prozessorfeld von n Prozessoren in folgender paralleler Zeit durchgeführt werden.*

- Berechnungszeit: $(3n + 10k)\alpha' + (2n + 2k)\alpha$
- Kommunikationszeit: $(2n + 9k)(\beta + \gamma')$

Korollar 3.7 *Unter der Verwendung eines linearen Feldes von n Prozessoren läßt sich für den Schnorr-Euchner Algorithmus größenordnungsmäßig eine Beschleunigung um den Faktor n gegenüber einer seriellen Ausführung erreichen. Dies gilt auch, wenn man die binäre Komplexität betrachtet.*

Beweis: Nach Satz 3.6 kann eine Stufe in paralleler Berechnungs- und Kommunikationszeit $O(n)$ durchgeführt werden gegenüber $O(n^2)$ im seriellen Fall. Dies gilt auch, wenn eine Neuberechnung der Skalarprodukte durchgeführt wird. Auch entstehen keine größeren Zahlen als bei einer seriellen Durchführung. ∎

[12] ohne die Neuberechnung der Skalarprodukte und unter Vernachlässigung konstanter Terme

3.5 Eine neue Variante des LLL - Reduktionsalgorithmus

Es wurde gezeigt, wie man den LLL-Algorithmus bzw. dessen Gleitkommavariante von Schnorr und Euchner mit Hilfe von n Prozessoren effizient parallelisieren kann. Eine Parallelisierung mit n^2 Prozessoren scheint hingegen nicht möglich zu sein, da diese Algorithmen stufenweise vorgehen. In jeder Stufe wird eine Spalte der Gram-Schmidt Orthogonalisierung berechnet, diese Spalte größenreduziert und evtl. ein Tausch durchgeführt. Auf Grund der Datenabhängigkeiten kostet eine solche Stufe immer $\Omega(n)$ parallele Schritte. n^2 Prozessoren können nur dann effizient eingesetzt werden, wenn auf der ganzen Matrix Operationen durchgeführt werden. Um einen hohen Grad an Parallelität zu erreichen, werden nun daher nicht mehr Stufenalgorithmen, sondern Algorithmen, die ähnlich dem Grundalgorithmus 3.2 arbeiten, betrachtet.

Zunächst werden wieder Algorithmen, die exakte Arithmetik benutzen, untersucht. Im nächsten Abschnitt wird ein solcher Algorithmus vorgestellt, der gut parallelisierbar ist. Dessen Effizienz wird in dem darauffolgenden Abschnitt bewiesen. In Abschnitt 3.5.3 wird diskutiert, wie man auch bei der neuen Variante schnelle Gleitkomma-Arithmetik benutzen kann. Eine Fehleranalyse wird durchgeführt. In Abschnitt 3.5.5 werden Ergebnisse, die mit dem Gleitkomma-Algorithmus erzielt wurden, vorgestellt.

3.5.1 Ein gut parallelisierbarer Algorithmus

Die grundsätzliche Idee ist, in einer Phase von Algorithmus 3.2 alle möglichen Tausche durchzuführen.[13] In einer Phase werden zwei Vektoren b_i und b_{i+1} getauscht für alle ungeraden Indizes i mit $\|b_i^*\|^2 > 2\|b_{i+1}^*\|^2$ und in der darauffolgenden Phase für alle geraden Indizes. Dies entspricht dem bekannten odd/even-Schema bei Matrixoperationen, s. z.B. [53]. Die Zusammenfassung zweier solcher Phasen wird im folgenden *„Alltauschphase“* genannt und ein Algorithmus, der nach dieser Strategie vorgeht, *„Alltauschalgorithmus“*. Die Effizienz dieses Vorgehens wird im nächsten Abschnitt bewiesen.

Algorithmus 3.7

Alltauschbasisreduktionsalgorithmus
Eingabe: Gitterbasis $b_1, b_2, \ldots, b_n$
Ausgabe: Reduzierte Basis $b_1, b_2, \ldots, b_n$
Methode:
Berechne die Gram-Schmidt Orthogonalisierung $\|b_1^*\|^2, \ldots, \|b_n^*\|^2$ und M
reihenfolge $\leftarrow$ ungerade

while (es kann noch getauscht werden) do

Berechne die Größenreduktion der Basis $(b_1, \ldots, b_n)$
(s. Algorithmus 3.1)

[13] Diese Idee hatten auch Roch und Villard [72, 50], aber sie konnten die Effizienz dieses Vorgehens nicht beweisen.

```
    if (reihenfolge = ungerade) then
       tausche b_i und b_{i+1} für jeden ungeraden Index i mit ||b_i^*||^2 > 2||b_{i+1}^*||^2
       reihenfolge ← gerade
    else
       tausche b_i und b_{i+1} für jeden geraden Index i mit ||b_i^*||^2 > 2||b_{i+1}^*||^2
       reihenfolge ← ungerade
    fi

    Repariere die Gram-Schmidt Orthogonalisierung
    (gemäß (3.17) - (3.25) )
od

Berechne die Größenreduktion der Basis (b_1, ..., b_n)
```

Alltauschalgorithmen in der Form von Algorithmus 3.7 sind unter der Verwendung von n^2 Prozessoren gut parallelisierbar. Eine Phase kann in Zeit $O(n)$ durchgeführt werden (s. [72, 50] und Kapitel 3.6).

3.5.2 Zeitkomplexität von Alltauschalgorithmen

Es soll nun die Effizienz von Alltauschalgorithmen untersucht werden. Wie gesehen werden beim LLL-Algorithmus maximal $n^2 \log B$ Vertauschungen durchgeführt, wobei ein Tausch $O(n^2)$ arithmetische Operationen benötigt. Dies ergibt eine Komplexität von $O(n^4 \log B)$ arithmetische Operationen (s. Abschnitt 3.2.3, Satz 3.5). Im Vergleich dazu wird jetzt berechnet, wieviele Alltauschphasen maximal durchgeführt werden müssen, um eine reduzierte Basis bzw. eine „gute“ Basis zu erhalten. Sei l die Anzahl der durchzuführenden Alltauschphasen. Da in einer Alltauschphase $O(n^3)$ arithmetische Operationen durchgeführt werden, ergibt sich eine Gesamtkomplexität von $l \cdot n^3$ arithmetischen Operationen. Um keinen Effizienzverlust gegenüber dem LLL-Algorithmus zu haben, sollten also $l = n \log B$ Alltauschphasen ausreichen. Umgekehrt formuliert sind Alltauschalgorithmen nur dann effizient im Vergleich zu Stufenalgorithmen, wenn in einer Alltauschphase viele Tausche gleichzeitig durchgeführt werden können.

Um das Problem besser zu verstehen, werden zunächst zwei Spezialfälle betrachtet. Dann wird ein Ergebnis für den allgemeinen Fall vorgestellt.

Spezialfall: Echter Tausch

Für die neuen Werte nach einem Tausch von b_i und b_{i+1} gelten die Formeln (3.10), (3.12) und (3.14). Würde immer gelten $\mu_{i+1,i} = 0$, so entspräche ein Tausch einem Tausch auf den *-Werten, d.h.

$$||b_i^{*'}||^2 = ||b_{i+1}^*||^2, \qquad ||b_{i+1}^{*'}||^2 = ||b_i^*||^2$$

Satz 3.8 *Unter der obigen Annahme hält der Algorithmus nach spätestens n Alltauschphasen.*

Beweis: Die Komplexität in diesem Fall entspricht der Komplexität von Odd-Even-Sort, s. z.B. [13]. ■

Spezialfall: $\frac{3}{4}$-Tausch

Der schlechteste Fall für die Abschätzung der Zahl der Tauschschritte bei Stufenalgorithmen ist, daß bei einem Tausch von b_i und b_{i+1} das Quadrat der Norm von b_i^* nur um den Faktor $\frac{3}{4}$ kleiner wird, also immer

$$||b_i^{*'}||^2 = \frac{3}{4}||b_i^*||^2$$

und

$$||b_{i+1}^{*'}||^2 = \frac{4}{3}||b_{i+1}^*||^2$$

gilt. Würde dies bei jedem Tausch gelten, wäre eine einfache Abschätzung für den parallelen Fall möglich.

Satz 3.9 *Gilt bei einem Tausch von b_i und b_{i+1} stets $||b_i^{*'}||^2 = \frac{3}{4}||b_i^*||^2$, dann gilt nach höchstens $n \log B$ Alltauschphasen*

$$||b_i^*||^2 \leq 2 \cdot \frac{16}{9}||b_{i+1}^*||^2 \qquad \textit{für} \quad 1 \leq i \leq n-1$$

d.h. die Bedingung für eine reduzierte Basis ist bis auf eine andere Konstante erfüllt.

Beweis: Die Beweisidee ist die folgende: Wenn an einer Stelle $i, i+1$ nicht getauscht wurde, so ist die Bedingung einer reduzierten Basis dort erfüllt. Sie kann nur dadurch wieder verloren gehen, daß an der Stelle $i-1, i$ oder $i+1, i+2$ getauscht wird. Da sich aber bei einem Tausch nach der Annahme die Werte nicht stark ändern, wird auch die Bedingung der Reduziertheit an der Stelle $i, i+1$ „nicht stark verletzt“. An dieser Stelle passiert daher nicht mehr viel.

Es muß also nur noch gezeigt werden, daß nach höchstens $n \log B$ Alltauschphasen an jeder Stelle mindestens einmal nicht getauscht wurde.

Der Beweis erfolgt also in zwei Schritten:

- Wurde an einer Stelle $i, i+1$ nicht getauscht, so gilt $||b_i^*||^2 \leq 2||b_{i+1}^*||^2$. Wird nun an der Stelle $i-1, i$ und/oder $i+1, i+2$ getauscht, so wird $||b_i^*||^2$ um den Faktor $\frac{4}{3}$ größer und/oder $||b_{i+1}^*||^2$ um den Faktor $\frac{3}{4}$ kleiner. Es gilt auf jeden Fall die Abschätzung des Satzes. Ist die Tauschbedingung an der Stelle $i, i+1$ auch jetzt nicht erfüllt, so gilt die Abschätzung auch nach der nächsten Phase (induktiv). Ist die Tauschbedingung erfüllt, so wird getauscht und somit $||b_i^*||^2$ um den Faktor $\frac{3}{4}$ kleiner und $||b_{i+1}^*||^2$ um den Faktor $\frac{4}{3}$ größer, wodurch die Tauschbedingung nach dieser Phase auf jeden Fall nicht mehr erfüllt ist. Damit ist auch die Abschätzung nach dieser Phase gültig.

- Nach höchstens $n \log B$ Alltauschphasen wurde an allen Stellen mindestens einmal nicht getauscht:

  Bei einem Tausch von b_i und b_{i+1} wird das Produkt $||b_1^*||^2 \cdot ||b_2^*||^2 \cdot \ldots \cdot ||b_i^*||^2$ um den Faktor $\frac{3}{4}$ kleiner (s. (3.12) und (3.14)). Dieses Produkt ist immer ganzzahlig [24], wenn man von einer ganzzahligen Basis ausgeht. Zu Beginn gilt $||b_1^*||^2 \cdot ||b_2^*||^2 \cdot \ldots \cdot ||b_i^*||^2 \leq B^i$ Daher kann an einer Stelle $i, i+1$ maximal $i \log B$ mal hintereinander getauscht werden.

■

Komplexität im allgemeinen Fall

Im allgemeinen können bei einem Tausch alle Möglichkeiten zwischen den beiden obigen Extremfällen auftreten. Daher ist eine allgemeine Abschätzung schwerer. Hier wird ein Resultat angegeben, das beweist, daß man nach $n \log B$ Alltauschphasen zumindest für den ersten Vektor b_1 der Basis einen guten Wert erhalten hat.

Die Idee dabei ist die folgende: Über die Größe der einzelnen Quadratnormen $\|b_i^*\|^2$ kann im allgemeinen keine Aussage getroffen werden, da $\|b_i^*\|^2$ sowohl bei einem Tausch an der Stelle $i-1, i$ als auch an der Stelle $i, i+1$ verändert wird. Dabei wird $\|b_i^*\|^2$ einmal vergrößert und einmal verkleinert. Eine verläßliche Aussage ist nur über die Produkte $\|b_1^*\|^2 \cdot \ldots \cdot \|b_i^*\|^2$ möglich. Das Produkt $\|b_1^*\|^2 \cdot \ldots \cdot \|b_i^*\|^2$ wird nur genau bei einem Tausch an der Stelle $i, i+1$ verändert. Es fällt dann um mindestens den Faktor $\frac{3}{4}$ (s. Formeln (3.10) - (3.12), (3.14) und auch Beweis zu Satz 3.3). Es wird nun berechnet, nach wievielen Alltauschphasen alle Produkte $\|b_1^*\|^2 \cdot \ldots \cdot \|b_i^*\|^2$ für $i = 1, \ldots, n$ den Wert erreicht haben, den sie in einer reduzierten Basis hätten.

Bei einer reduzierten Basis gilt

$$||b_i^*||^2 \leq 2||b_{i+1}^*||^2 \qquad \text{für} \quad 1 \leq i \leq n-1$$

Das Produkt

$$||b_1^*||^2 \cdot ||b_2^*||^2 \cdot \ldots \cdot ||b_n^*||^2 = (\det L)^2$$

ist immer konstant. Daher ist der schlechteste Fall, der für den ersten Vektor $b_1 = b_1^*$ einer reduzierten Basis auftreten kann:

$$||b_i^*||^2 = 2||b_{i+1}^*||^2 \qquad \text{für} \quad 1 \leq i \leq n-1$$

Dann gilt:

$$||b_i^*||^2 = 2^{\frac{n-2i+1}{2}} (\det L)^{\frac{2}{n}}$$

Für die Produkte der ersten i Quadratnormen gilt somit

$$||b_1^*||^2 \cdot ||b_2^*||^2 \cdot \ldots \cdot ||b_i^*||^2 = 2^{\frac{i(n-i)}{2}} (\det L)^{\frac{2i}{n}}$$

Im folgenden wird also abgeschätzt, wann diese Werte[14] erreicht sind.

[14] Aus beweistechnischen Gründen betrachtet man leicht modifizierte Werte.

Satz 3.10 *Nach höchstens $n \log B$ Alltauschphasen gilt die folgende Situation für alle i mit $1 \leq i \leq n$:*

$$||b_1^*||^2 \cdot ||b_2^*||^2 \cdot \ldots \cdot ||b_i^*||^2 \leq \tfrac{4}{3} \cdot c^{\frac{i(n-i)}{2}} \cdot (\det L)^{\frac{2i}{n}}$$

mit $c = \frac{32}{9}$.

Beweis: Es werden die folgenden Verhältnisse betrachtet.

$$v(i) := \frac{||b_1^*||^2}{c^{\frac{n-1}{2}}(\det L)^{\frac{2}{n}}} \cdot \frac{||b_2^*||^2}{c^{\frac{n-3}{2}}(\det L)^{\frac{2}{n}}} \cdot \ldots \cdot \frac{||b_i^*||^2}{c^{\frac{n-2i+1}{2}}(\det L)^{\frac{2}{n}}} = \frac{\prod_{j=1}^{i} ||b_j^*||^2}{c^{\frac{i(n-i)}{2}}(\det L)^{\frac{2i}{n}}} \tag{3.26}$$

Die Aussage des Satzes gilt, wenn für alle solche Verhältnisse $v(i), 1 \leq i \leq n$, gilt $v(i) \leq \frac{4}{3}$. Sei max das Maximum dieser Werte, d.h. $max = \max\{v(i) | 1 \leq i \leq n\}$ und max' das Maximum nach einer weiteren Alltauschphase.

Es wird gezeigt, daß nach jeder Alltauschphase der Wert dieses Maximums um mindestens den Faktor $\frac{3}{4}$ fällt, d.h. $max' \leq \frac{3}{4} max$, solange $max > \frac{4}{3}$. Zu Beginn gilt $max \leq B^n$, da $||b_i^*||^2 \leq B$ für alle $1 \leq i \leq n$. Daher sind alle Verhältnisse $v(i) \leq \frac{4}{3}$ nach höchstens $n \log B$ Alltauschphasen.

Behauptung 3.11 *Wenn $max > \frac{4}{3}$ und $v(i) > \frac{3}{4} max$, dann fällt in einer Alltauschphase das Produkt $||b_1^*||^2 \cdot ||b_2^*||^2 \cdot \ldots \cdot ||b_i^*||^2$ und damit $v(i)$ um einen Faktor kleiner gleich $\frac{3}{4}$. Alle anderen Verhältnisse ändern sich nicht oder werden ebenfalls kleiner.*

Beweis der Behauptung:

Wenn zwei Vektoren b_j und b_{j+1} getauscht werden, verringert sich das Produkt $||b_1^*||^2 \cdot ||b_2^*||^2 \cdot \ldots \cdot ||b_j^*||^2$ nach (3.14) um einen Faktor kleiner gleich $\frac{3}{4}$. Alle anderen Produkte bleiben nach (3.12) unverändert. Daher muß nur gezeigt werden, daß in einer Alltauschphase b_i und b_{i+1} getauscht werden für alle i mit $v(i) > \frac{3}{4} max$, falls $max > \frac{4}{3}$, d.h. nach der Tauschbedingung (3.13) muß gezeigt werden, daß $||b_i^*||^2 > 2||b_{i+1}^*||^2$, wenn $v(i) > \frac{3}{4} max$. Dabei ist zu beachten, daß, falls diese Tauschbedingung für einen geraden Index i gilt, sie auch nach einer „ungeraden Tauschphase“ gültig bleibt und umgekehrt (siehe (3.12) und (3.14)).

Falls für den letzten Index n gilt $v(n) > \frac{3}{4} max$, dann gilt $max < \frac{4}{3}$, da $v(n) = 1$, und man ist fertig.

Betrachte daher einen Index $i, 1 \leq i \leq n-1$, mit $v(i) > \frac{3}{4} max$. Es ist zu zeigen, daß die Vektoren b_i und b_{i+1} für diesen Fall getauscht werden.

Falls $v(i) > \frac{3}{4} max$ ist, gilt $v(i) > \frac{3}{4} v(i+1)$, und mit (3.26) erhält man:

$$\frac{\frac{4}{3}(\det L)^{\frac{2}{n}} c^{\frac{n-2i-1}{2}}}{||b_i^*||^2} > \frac{||b_{i+1}^*||^2}{||b_i^*||^2} \tag{3.27}$$

Ebenso gilt für $v(i) > \frac{3}{4} max$ und $i > 1$, daß $v(i) > \frac{3}{4} v(i-1)$ ist und damit

$$||b_i^*||^2 > \frac{3}{4} c^{\frac{n-2i+1}{2}} (\det L)^{\frac{2}{n}} \tag{3.28}$$

Dies gilt ebenfalls für $i = 1$, falls $max \geq 1$ und $v(1) > \frac{3}{4}max$.

Mit (3.27) und (3.28) folgt nun

$$\frac{\left(\frac{4}{3}\right)^2}{c} = \frac{1}{2} > \frac{||b_{i+1}^*||^2}{||b_i^*||^2} \tag{3.29}$$

und daher ist die Tauschbedingung für die Vektoren b_i und b_{i+1} erfüllt.

Damit ist die Behauptung und der Satz bewiesen. ■

Bemerkung: Im Gegensatz zu dem Beweis der Komplexität des LLL-Algorithmus wird hier die Ganzzahligkeit der Ausgangsbasis nicht ausgenutzt.

Insbesondere gilt also, daß nach $n \log B$ Phasen ein kurzer Vektor gefunden wurde.

Korollar 3.12 *Nach höchstens $n \log B$ Alltauschphasen gilt für den ersten Basisvektor*

$$\|b_1\| \leq \sqrt{\frac{4}{3}} c^{\frac{n-1}{4}} (\det L)^{\frac{1}{n}}$$

Beweis: Das Korollar folgt direkt aus Satz 3.10 für $i = 1$ und der Beziehung $b_1 = b_1^*$. ■

Dies ist fast dieselbe Schranke wie für den ersten Vektor einer reduzierten Basis (vgl. mit Satz 3.2).

Eine Anwendung, für die diese Schranke ausreichend ist, ist z.B. bei der „simultanen diophantischen Approximation“ gegeben (s. [59]).

Für die Komplexität von Alltauschalgorithmen gilt somit das folgende Korollar:

Korollar 3.13 *Ein kurzer Vektor des Gitters kann mit Alltauschalgorithmen mit $O(n^4 \log B)$ arithmetischen Operationen gefunden werden.*

Beweis: Eine Alltauschphase benötigt $O(n^3)$ arithmetische Operationen. Nach Korollar 3.12 müssen höchstens $n \log B$ Alltauschphasen durchgeführt werden. ■

Damit wird diesselbe Komplexität wie bei Stufenalgorithmen erreicht. Ein prinzipielles Problem bei der Benutzung von Alltauschalgorithmen soll jedoch an dieser Stelle nicht verschwiegen werden. Da die Gram-Schmidt-Zerlegung immer für die ganze Matrix berechnet wird und nicht nur für eine Spalte, werden auch noch nicht reduzierte Spalten bearbeitet. Dadurch können (zumindest theoretisch) größere Zahlen entstehen. Dieses Problem wird in Abschnitt 3.8 ausführlicher behandelt.

3.5.3 Verwendung von Gleitkommaarithmetik

Bisher wurden Alltauschalgorithmen nur bei der Verwendung von exakter Arithmetik betrachtet. Auch hier soll nun die Idee der Benutzung schneller Gleitkomma-Arithmetik untersucht werden. Wie bei dem Schnorr-Euchner Algorithmus 3.4 werden nur die Basisvektoren $b_1, \ldots, b_n$ in genauer ganzzahliger Darstellung gespeichert. Die Koeffizienten der Gram-Schmidt Orthogonalisierung (Matrix M) und die Quadratnormen der Vektoren der Gram-Schmidt Orthogonalisierung $\|b_1^*\|^2, \ldots, \|b_n^*\|^2$ sollen hingegen in Gleitkommadarstellung gehalten werden.

Es werden folgende Maßnahmen getroffen, um die Rundungsfehler möglichst gering zu halten:

- Die Gram-Schmidt Orthogonalisierung wird im Gegensatz zum LLL-Algorithmus in jeder Phase neu berechnet. So können sich die Rundungsfehler in der Matrix M nicht akkumulieren, da in jeder Phase von der exakten Gitterbasis ausgegangen werden kann.
- Die Gram-Schmidt Orthogonalisierung wird mit Givens-Rotationen berechnet (s. unten). Givens-Rotationen sind im Gegensatz zu der Original Gram-Schmidt-Prozedur numerisch stabil [22] und gut parallelisierbar mit n^2 Prozessoren (s. Abschnitt 3.6). In Abschnitt 3.5.4 wird eine Rundungsfehleranalyse durchgeführt und eine Bedingung hergeleitet, um falsche Tausche aufgrund von Rundungsfehlern zu vermeiden.

Der gundlegende Gleitkomma-Alltauschalgorithmus hat folgende Form:

Algorithmus 3.8

Gleitkomma-Alltauschbasisreduktionsalgorithmus
Eingabe: Gitterbasis $b_1, b_2, \ldots, b_n$ in exakter Darstellung
Ausgabe: Reduzierte Basis[15] $b_1, b_2, \ldots, b_n$
Methode:

```
reihenfolge ← ungerade
while (es kann noch getauscht werden) do
    MT ← Gleitkomma(b_1, ..., b_n)
    (MT entspricht der Koeffizientenmatrix M^T)
    Berechne die Gram-Schmidt Orthogonalisierung mit
    Givens-Rotationen auf MT (s. Algorithmus 3.9 unten)

    Berechne die Größenreduktion der Basis (b_1, ..., b_n)
    (s. Algorithmus 3.1)

    if (reihenfolge = ungerade) then
        tausche b_i und b_{i+1} für jeden ungeraden Index i mit ||b_i^*||^2 > 2||b_{i+1}^*||^2
        reihenfolge ← gerade
```

[15] Reduziert bis auf Rundungsfehler. Es kann sein, daß die resultierende Basis nicht exakt die Bedingungen einer reduzierten Basis erfüllt.

else
tausche b_i und b_{i+1} für jeden geraden Index i mit $||b_i^*||^2 > 2||b_{i+1}^*||^2$
reihenfolge $\leftarrow$ ungerade
fi
od
Berechne die Gram-Schmidt Orthogonalisierung und die Größenreduktion

Berechnung der Gram-Schmidt Orthogonalisierung mit Hilfe von Givens-Rotationen

Gemäß (3.7) kann die Gram-Schmidt Orthogonalisierung in Matrixform geschrieben werden:

$$(b_1, \ldots, b_n) = (b_1^*, \ldots, b_n^*)M^T$$

Dabei ist M^{T} eine obere Dreiecksmatrix mit Einsen auf der Hauptdiagonalen und $b_1^*, \ldots, b_n^*$ sind paarweise orthogonal. Für den Basisreduktionsalgorithmus müssen nur die Matrix M und die Quadratnormen $||b_1^*||^2, \ldots, ||b_n^*||^2$ berechnet werden. Um die Matrix $(b_1, \ldots b_n)$ zu triangularisieren, können Givens-Rotationen verwendet werden (s. z.B. [22]). Genauer gesagt kann man mit Givens-Rotationen eine orthogonale Matrix Q berechnen, so daß $Q(b_1, \ldots b_n) = R$, wobei R eine rechte obere Dreiecksmatrix ist. Da Q orthogonal ist, gilt

$$(b_1, \ldots, b_n) = Q^T R$$

Diese Gleichung kann umgeschrieben werden zu

$$(b_1, \ldots, b_n) = \underbrace{Q^T \begin{pmatrix} r_{11} & & 0 \\ & \ddots & \\ 0 & & r_{nn} \end{pmatrix}}_{(b_1^*, \ldots, b_n^*)} \underbrace{\begin{pmatrix} r_{11}^{-1} & & 0 \\ & \ddots & \\ 0 & & r_{nn}^{-1} \end{pmatrix} \begin{pmatrix} r_{11} & \cdots & r_{ij} \\ & \ddots & \vdots \\ 0 & & r_{nn} \end{pmatrix}}_{M^T}$$

Da Q orthogonal ist, haben die Spalten von Q die Norm 1. Mit obiger Beziehung können daher die Quadratnormen der Gram-Schmidt Orthogonalisierung folgendermaßen berechnet werden:

$$||b_i^*||^2 = r_{i,i}^2 \tag{3.30}$$

Die Matrix der Gram-Schmidt Koeffizienten M^{T} erhält man aus der Matrix R, indem jede Zeile i durch das Diagonalelement $r_{i,i}$ dividiert wird.

Insgesamt erhält man also folgenden Algorithmus zur Berechnung der Gram-Schmidt Orthogonalisierung:

Algorithmus 3.9

Berechnung der Gram-Schmidt Orthogonalisierung mit Givens-Rotationen

$MT \leftarrow$ Gleitkomma$(b_1, \ldots, b_n)$
(MT entspricht der Matrix M^{T})

Triangularisierung von MT mit Givens-Rotationen:

for $j = 1$ to $n-1$ do
  for $k = j+1$ to n do
    Berechne eine Givens Rotation auf den Reihen k und j
    von MT, so daß das Element $MT_{k,j}$ zu null rotiert wird:

    Bestimmung der orthogonalen Transformationsmatrix
    $G = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$:
    $c \leftarrow \frac{MT_{j,j}}{\sqrt{MT_{j,j}^2 + MT_{k,j}^2}}$
    $s \leftarrow \frac{MT_{k,j}}{\sqrt{MT_{j,j}^2 + MT_{k,j}^2}}$

    Anwendung der Transformation auf die Zeilen j und k, d.h.
    Multiplikation von G mit den Zeilen j und k, so daß
    $MT_{k,j} = 0$:
    for $i = j$ to n do
      $h_1 \leftarrow MT_{j,i}$; $h_2 \leftarrow MT_{k,i}$
      $MT_{j,i} \leftarrow c \cdot h_1 + s \cdot h_2$
      $MT_{k,i} \leftarrow -s \cdot h_1 + c \cdot h_2$
    od
  od
od

Berechnung der Gram-Schmidt Orthogonalisierung aus der
triangularisierten Matrix MT:
for $j = 1$ to n do
  $BN_j \leftarrow MT_{j,j} * MT_{j,j}$
  (BN_j entspricht $||b_j^*||^2$)
  for $i = n$ downto j do
    $MT_{j,i} \leftarrow MT_{j,i}/MT_{j,j}$
  od
od

Aus numerischen Gründen ist es sinnvoll, die Parameter c und s in einer anderen als der oben angegebenen Reihenfolge zu berechnen (s. [22]).

Die Berechnung der Gram-Schmidt Orthogonalisierung mit Hilfe von Givens Rotationen kann auch *exakt* durchgeführt werden, wenn man von einer ganzzahligen Basis $(b_1, \ldots, b_n)$ ausgeht. Dazu werden wurzel- und divisionsfreie Verfahren angewendet [23]. Bei der Berechnung von M und $\|b_i^*\|^2$ aus der transformierten Matrix fallen die Wurzeln weg, so daß diese nie berechnet werden müssen. Alle anderen Berechnungen können exakt durchgeführt werden. Dabei können allerdings sehr große Zahlen entstehen, so daß diese Methode bei exakter Rechnung in der Praxis nicht angewendet werden sollte.

Wurzel- und divisionsfreie Verfahren auf Gleitkommadarstellung können auch verwendet werden, um schnelle VLSI-Realisierungen zu erhalten.

3.5.4 Fehleranalyse

Wie vorher beschrieben werden die Normen $\|b_i^*\|^2$ der Gram-Schmidt Orthogonalisierung in Gleitkomma-Arithmetik mit Hilfe von Givens-Rotationen (s. Algorithmus 3.9) berechnet. Somit ist eine genaue Auswertung der Tauschbedingung $\|b_i^*\|^2 > 2\|b_{i+1}^*\|^2$ nicht möglich. Es soll nun eine Bedingung abgeleitet werden, um falsche Tausche zu vermeiden, da falsche Tausche zu großen Zahlen führen können. Dazu werden im nächsten Abschnitt die entstehenden Rundungsfehler bei der Orthogonalisierung abgeschätzt. Dann wird berechnet, wann diese Rundungsfehler zu groß sein können.

Rundungsfehler bei der Orthogonalisierung

Der entstehende Fehler wird mit Hilfe der von Wilkinson [74] eingeführten Fehleranalyse berechnet, s. auch z.B. [19, 75]. Dabei wird davon ausgegangen, daß Rundungsfehler dadurch entstehen, daß das (korrekte) Ergebnis einer Operation auf die nächste darstellbare Gleitkommazahl gerundet werden muß. Die Rundungsoperation soll im folgenden mit $fl(.)$ bezeichnet werden. Der bei einer Rundung entstehende Fehler kann folgendermaßen abgeschätzt werden:

Sei x eine reelle Zahl. Dann gilt:

$$fl(x) = x(1+\epsilon)$$

wobei $|\epsilon| \leq u$ gilt mit der „Maschinenzahl“ $u = \frac{1}{2}b^{-r+1}$, wobei b die Basis der Zahlendarstellung und r die Anzahl der Mantissenstellen bezeichnet.

Anwendung einer Givensrotation: Es soll nun der Fehler untersucht werden, der bei der Anwendung einer Givens-Rotation entstehen kann. Sei a ein (exakter) Vektor und G eine (exakte) Givens-Rotationsmatrix gemäß Algorithmus 3.9. Mit $'$ seien die gestörten Größen bezeichnet. Der exakt transformierte Vektor sei $b = Ga$. Tatsächlich berechnet wird jedoch $b' = fl(G'a)$. Eine Fehleranalyse nach Wilkinson [75] liefert dann für den entstehenden Fehler

$$\|\delta_b\| = \|b' - b\| = \|fl(G'a) - Ga\| \leq 7u\|a\| \tag{3.31}$$

Diese Analyse soll im folgenden durchgeführt werden.

Der entstehende Fehler läßt sich aufspalten in einen Fehler, der bei der Berechnung der Transformationsparameter entsteht, und einen bei der Durchführung der Transformation:

$$\begin{aligned} \delta_b &= fl(G'a) - Ga \\ &= fl(G'a) - G'a + \\ &\quad G'a - Ga \end{aligned} \tag{3.32}$$

Fehler bei der Berechung der Transformationsparameter: Bei der Anwendung einer Givenstransformation werden zwei Zeilen verändert. Daher wird zunächst angenommen, daß der Vektor a nur zweielementig ist. Zur Berechnung der Transformationsmatrix

$G = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$ werden die Parameter c und s berechnet:

$$c = \frac{a_1}{\sqrt{(a_1^2 + a_2^2)}} \tag{3.33}$$

$$s = \frac{a_2}{\sqrt{(a_1^2 + a_2^2)}} \tag{3.34}$$

Fehler bei der Nennerberechnung:

$$\begin{aligned} (a_1^2 + a_2^2)' &= (a_1^2(1+\epsilon_1) + a_2^2(1+\epsilon_2))(1+\epsilon_3) \qquad \text{mit} \quad |\epsilon_i| \leq u \\ &\approx (a_1^2 + a_2^2)(1+\epsilon_4) \qquad \text{mit} \quad |\epsilon_4| \leq 2u \end{aligned}$$

Damit gilt bei der Berechnung von c':

$$\begin{aligned} c' &= \frac{a_1}{\sqrt{(a_1^2 + a_2^2)(1+\epsilon_4)}(1+\epsilon_5)}(1+\epsilon_6) \\ c' &= c\frac{(1+\epsilon_6)}{\sqrt{(1+\epsilon_4)}(1+\epsilon_5)} \end{aligned}$$

mit $|\epsilon_5|, |\epsilon_6| \leq u$. Der Fehler läßt sich für $u \ll 1$ folgendermaßen abschätzen

$$\begin{aligned} \frac{(1+\epsilon_6)}{\sqrt{(1+\epsilon_4)}(1+\epsilon_5)} &\leq \frac{(1+u)}{(1-2u)^{\frac{1}{2}}(1-u)} \\ &\approx \frac{(1+u)}{(1-u)^2} \\ &= \frac{(1+u)(1+u)^2}{((1-u)(1+u))^2} \\ &= \frac{(1+u)(1+u)^2}{(1+u^2)^2} \\ &\leq (1+u)(1+u)^2 \\ &\approx (1+3u) \end{aligned}$$

da $(1-2u) \approx (1-u)^2$ für $u \ll 1$. Außerdem gilt $(1+2u) \approx (1+u)^2$ und somit

$$\begin{aligned} \frac{(1+\epsilon_6)}{\sqrt{(1+\epsilon_4)}(1+\epsilon_5)} &\geq \frac{(1-u)}{(1+2u)^{\frac{1}{2}}(1+u)} \\ &\approx \frac{(1-u)}{(1+u)^2} \\ &\approx \frac{(1-u)}{(1+2u)} \\ &= \frac{(1-u)(1-2u)}{(1+2u)(1-2u)} \\ &\approx (1-u)(1-2u) \\ &\approx (1-3u) \end{aligned}$$

Für s' gilt dieselbe Abschätzung und daher gilt für die fehlerhaften Rotationsparameter:

$$c' = c(1+\delta_c) \qquad \text{mit} \quad |\delta_c| \leq 3u \tag{3.35}$$

$$s' = s(1+\delta_s) \qquad \text{mit} \quad |\delta_s| \leq 3u \tag{3.36}$$

Für die fehlerbehaftete Transformationsmatrix gilt

$$G' = G + \begin{pmatrix} c\delta_c & s\delta_s \\ -s\delta_s & c\delta_c \end{pmatrix} \tag{3.37}$$

$$= G + \delta_G \tag{3.38}$$

mit

$$\|\delta_G\| \leq \sqrt{2} \cdot 3u \tag{3.39}$$

da

$$\begin{aligned} \|\delta_G\| &= (c^2\delta_c^2 + s^2\delta_s^2 + c^2\delta_c^2 + s^2\delta_s^2)^{\frac{1}{2}} \\ &\leq [(c^2+s^2) + (c^2+s^2)]^{\frac{1}{2}} 3u \\ &= 3 \cdot \sqrt{2} \cdot u \end{aligned}$$

Fehler bei der Anwendung der Transformation: Es sei $b' = fl(G'a)$. Dann gilt für die erste Komponente des transformierten Vektors b'_1 für $u \ll 1$:

$$\begin{aligned} b'_1 &= (c'a_1(1+\epsilon_1) + s'a_2(1+\epsilon_2))(1+\epsilon_3) \\ &\approx c'a_1 + c'a_1\epsilon_1 + s'a_2 + s'a_2\epsilon_2 + c'a_1\epsilon_3 + s'a_2\epsilon_3 \\ &= c'a_1(1+\epsilon_1+\epsilon_3) + s'a_2(1+\epsilon_2+\epsilon_3) \\ &= c'a_1(1+\epsilon_4) + s'a_2(1+\epsilon_5) \end{aligned}$$

mit $|\epsilon_1|, |\epsilon_2|, |\epsilon_3| \leq u$ und $|\epsilon_4|, |\epsilon_5| \leq 2u$. Damit gilt für den Fehler in dem transformierten Vektor b':

$$\begin{aligned} b' - b &= \begin{pmatrix} \epsilon_4 c' & \epsilon_5 s' \\ -\epsilon_6 s' & \epsilon_7 c' \end{pmatrix} a \qquad \text{mit} \quad |\epsilon_i| \leq 2u \\ \|b' - b\| &\leq \left\| \begin{pmatrix} \epsilon_4 c' & \epsilon_5 s' \\ -\epsilon_6 s' & \epsilon_7 c' \end{pmatrix} \right\| \|a\| \\ &\leq [(c'^2 + s'^2 + c'^2 + s'^2)\epsilon_8^2]^{\frac{1}{2}} \|a\| \qquad \text{mit} \quad |\epsilon_8| \leq 2u \\ &\leq 2\sqrt{2}u\|a\| \end{aligned} \tag{3.40}$$

Gesamtfehler bei der Anwendung einer Transformation: Für den Gesamtfehler δ_b gilt nach (3.32) und (3.39) und (3.40):

$$\begin{aligned} \|\delta_b\| &\leq \|fl(G'a) - G'a\| + \|G'a - Ga\| \\ &\leq 2\sqrt{2}u\|a\| + 3\sqrt{2}u\|a\| \\ &\approx 7u\|a\| \end{aligned} \tag{3.41}$$

Folge von Givensrotationen [20]: Um eine Matrix A in Dreiecksgestalt umzuformen, wird eine Folge von Givenstransformationen auf die Matrix angewendet (s. Algorithmus 3.9). Zur Fehlerabschätzung können die Fehler, die auf den einzelnen Spalten $a_i, 1 \leq i \leq n$, der Matrix A entstehen, getrennt betrachtet werden. Seien $b_i, 1 \leq i \leq n$, die Spalten der transformierten Matrix.

Dann gilt bei einer Folge von Givensrotationen:

$$\begin{aligned} b_i &= T_s T_{s-1} \ldots T_1 a_i \\ b'_i &= [(T_s T_{s-1} \ldots T_1) a_i]' \end{aligned}$$

Nach (3.41) gilt für eine Transformation

$$fl(T_1 a_i) = T_1 a_i + \delta_1$$

mit $\|\delta_1\| \leq 7u\|a_i\|$. Bei der Anwendung einer weiteren Transformation gilt

$$\begin{aligned} fl(T_2 T_1 a_i) &= T_2(T_1 a_i + \delta_1) + \delta_2 \\ &= T_2 T_1 a_i + T_2 \delta_1 + \delta_2 \end{aligned}$$

mit

$$\begin{aligned} \|\delta_2\| &\leq 7u\|T_1 a_i + \delta_1\| \\ &\leq 7u\|T_1 a_i\| + 7u\|\delta_1\| \\ &\leq 7u(1+7u)\|a_i\| \end{aligned}$$

da T_1 orthogonal ist und damit $\|T_1 a_i\| = \|a_i\|$.

Wendet man dies iterativ an, so gilt:

$$\|b_i' - b_i\| \leq 7u \cdot s \cdot (1+7u)^{s-1}\|a_i\| \tag{3.42}$$

Transformationen, die verschiedene Zeilen von a_i verknüpfen, verursachen keinen größeren Fehler als eine Transformation. Daher können diese in der obigen Abschätzung zusammengefaßt werden [20]. Bei der Triangularisierung einer $n \times n$-Matrix werden maximal $s = 2n-3$ „zusammengefaßte“ Transformationen durchgeführt.

Somit gilt für den Fehler in einem Spaltenvektor b_i:

$$\|b_i' - b_i\| \leq 7u(2n-3)(1+7u)^{2n-4}\|a_i\| \tag{3.43}$$

Für $7u \ll 1$ kann der Term $(1+7u)^{2n}$ durch $1 + 14 \cdot u \cdot n$ abgeschätzt werden. Daher gilt die folgende Abschätzung:

$$\begin{aligned} \|b_i' - b_i\| &\leq 7u(2n-3)(1+7u)^{2n-4}\|a_i\| \\ &\leq 7u(2n-3)(1+7u)^{2n}\|a_i\| \\ &\approx 7 \cdot u(2n-3)(1+14 \cdot u \cdot n)\|a_i\| \\ &\leq (14 \cdot u \cdot n + (14 \cdot u \cdot n)^2)\|a_i\| \\ &\approx 14 \cdot u \cdot n\|a_i\| \end{aligned}$$

für $n \cdot u \ll 1$.

Diese Abschätzung gilt insbesondere für ein einzelnes Element der Vektoren b_i, d.h. es gilt das folgende Lemma.

Lemma 3.14 *Nach der Triangularisierung einer Matrix mit Hilfe einer Folge von Givensrotationen gilt für den Rundungsfehler in jedem Matrixelement $b_{i,j}$ die folgende Abschätzung:*

$$|b_{i,j}' - b_{i,j}| \leq 14 \cdot u \cdot n\|a_i\| \qquad \textit{für} \quad j = 1, \ldots, n \tag{3.44}$$

Auswirkungen der Fehler auf die Tauschbedingung

Es werden nun die möglichen Rundungsfehler bei der Berechnung der Tauschbedingung $||b_i^{*'}||^2 > 2||b_{i+1}^{*'}||^2$ berechnet.

Sei $(b_1, \ldots, b_n)$ die Gitterbasis und $R = Q(b_1, \ldots, b_n)$ die transformierte Basis. Nach (3.30) gilt die folgende Beziehung zwischen der Quadratnorm $\|b_i^*\|^2$ der Gram-Schmidt Orthogonalisierung und der transformierten Matrix R:

$$||b_i^*||^2 = r_{ii}^2 \leq B$$

Nach (3.44) gilt für den Rundungsfehler in den Elementen $r_{i,j}$ der transformierten Matrix R:

$$|r'_{i,j} - r_{i,j}| \leq 14 \cdot n \cdot u \cdot \|b_j\|$$

oder

$$r'_{i,j} = r_{i,j} + \epsilon$$

mit $|\epsilon| \leq 14 \cdot n \cdot u \cdot ||b_j||$.

Ein Tausch zweier Basisvektoren b_j und b_{j+1} soll stattfinden, wenn für die exakten Werte gilt:

$$\|b_{j+1}^*\|^2 < \frac{1}{2}\|b_j^*\|^2$$

Dabei ist der Fakor $\frac{1}{2}$ willkürlich. Ein Tausch bringt nach (3.11) eine echte Verbesserung, wenn die obige Beziehung für einen beliebigen Faktor zwischen 0 und $\frac{3}{4}$ gültig ist.

Für die Fehleranalyse ist es günstiger, die Formel ohne Quadrierung zu betrachten, d.h. es soll getauscht werden, wenn

$$\|b_{j+1}^*\| < \sqrt{\frac{1}{2}}\|b_j^*\|$$

wobei der Faktor $\sqrt{\frac{1}{2}}$ zwischen 0 und $\sqrt{\frac{3}{4}}$ variieren kann.

Ist diese Tauschbedingung für die gerundeten Werte $r'_{i,j}$ erfüllt, so gilt für die exakten Werte:

$$|r_{j+1,j+1}| + \epsilon_1 < \sqrt{\frac{1}{2}}(|r_{j,j}| + \epsilon_2)(1 + \epsilon_3)$$

mit

$$\begin{aligned} |\epsilon_1| &\leq 14 \cdot n \cdot u \cdot \|b_{j+1}\| \\ |\epsilon_2| &\leq 14 \cdot n \cdot u \cdot \|b_j\| \\ |\epsilon_3| &\leq 2u \end{aligned}$$

Mit ϵ_3 wird der Fehler abgeschätzt, der bei der Berechnung von $\sqrt{\frac{1}{2}}$ und der Multiplikation entsteht.

Unter Vernachlässigung von $\sqrt{\frac{1}{2}}\epsilon_2\epsilon_3$ erhält man:

$$|r_{j+1,j+1}| + \underbrace{\epsilon_1 - \sqrt{\frac{1}{2}}\epsilon_2 - \sqrt{\frac{1}{2}}|r_{j,j}|\epsilon_3}_{f} < \sqrt{\frac{1}{2}}|r_{j,j}|$$

Der „Fehlerterm“ f läßt sich folgendermaßen abschätzen. Es gelte $\|b_i\|^2 \leq A$ für alle i. Dann gilt nach der Konstruktion der Gram-Schmidt-Beziehung auch $\|b_i^*\|^2 \leq A$. Damit gilt für f:

$$\begin{aligned} |f| &\leq 14 \cdot n \cdot u \cdot \sqrt{A} + \sqrt{\frac{1}{2}} \cdot 14 \cdot n \cdot u \cdot \sqrt{A} + \sqrt{\frac{1}{2}} \cdot |r_{jj}| \cdot 2u \\ &\leq 28 \cdot n \cdot u \cdot \sqrt{A} + 2u|r_{jj}| \\ &\leq 30 \cdot n \cdot u \cdot \sqrt{A} \end{aligned}$$

Gilt nun

$$|f| \leq \frac{1}{10}|r_{jj}|$$

so gilt für die exakten Werte

$$\|b_{j+1}^*\| < a\|b_j^*\|$$

wobei der Faktor a in dem oben angegebenen Toleranzbereich liegt. Damit ist ein Tausch sinnvoll.

Der folgende Satz faßt das Ergebnis zusammen:

Satz 3.15 *Sei A das Maximum der Quadratnormen der aktuellen Basis, d.h. $A = \max_{i=1,\ldots,n} \|b_i\|^2$. Unter der Verwendung des Fehlermodells von Wilkinson [74] und der Vernachlässigung höherwertiger Terme führt ein Tausch der Basisvektoren b_j und b_{j+1} zu einer besseren Gitterbasis, wenn die Tauschbedingung $||b_j^{*'}|| > \sqrt{2}||b_{j+1}^{*'}||$ für die gerundeten Werte gilt und*

$$const \cdot n \cdot u \cdot \sqrt{A} \leq |r_{j,j}|$$

mit $const = 300$.

Bemerkungen:

- Zur Berechnung von A:

  Während der gesamten Ausführung des Algorithmus gilt immer

  $$A \leq nB$$

  wobei B die maximale Quadratnorm der Ausgangsbasis ist (s. Abschnitt 3.8).

  Es ist jedoch besser, A in jeder Phase neu zu berechnen, da dieser Wert während der Ausführung des Algorithmus kleiner werden sollte, wodurch sich eine weniger harte Fehlerbedingung ergibt.

  Noch besser ist die Wahl

  $$A = \max\{\|b_j\|^2, \|b_{j+1}\|^2\}$$

- Sollte aufgrund der Fehlerbedingung kein Tausch möglich sein, so muß eine Gram-Schmidt Orthogonalisierung exakt berechnet werden, um festzustellen, ob die Basis tatsächlich reduziert ist.

n	Stellen	Phasen	Tausche	Gr.-Red.	dZl	dT	G/EP (max)	Z/EP	Zeit [sec]
16	164 / 8	680	4 586	44 790	34	13	65 (120)	0,01	8,4
32	105 / 5	740	10 043	209 346	20	27	282 (496)	0,08	57,6
64	205 / 6	4 376	118 483	5 491 880	34	54	1 255 (2 016)	0,67	2 936,6
80	254 / 7	7 668	257 350	15 573 392	42	67	2 030 (3 160)	1,30	9 987,1
100	305 / 8	14 054	589 006	46 462 524	54	83	3 306 (4 950)	2,70	38 475,2
110	358 / 8	18 556	861 471	76 140 917	62	92	4 103 (5 995)	4,01	74 465,7

Tabelle 3.2: Neuer serieller Algorithmus – unimodular transformierte Gitter

n	Stellen	Phasen	Tausche	Gr.-Red.	eS	dZl	Z/EP	Zeit [sec]
16	164 / 8	19 151	9 707	14 795	0	21	0,0005	8,8
32	105 / 5	63 970	32 071	105 082	42	10	0,0014	93,9
64	205 / 6	612 168	306 192	2 643 446	18	19	0,0039	2 411,7
80	254 / 10	1 128 625	564 349	6 530 510	0	25	0,0075	8 514,7
100	305 / 8	2 377 053	1 188 473	19 171 009	14	30	0,0104	24 810,3
110	358 / 8	3 180 622	1 590 102	29 005 913	44	32	0,0172	54 666,9

Tabelle 3.3: Schnorr/Euchner-Algorithmus – unimodular transformierte Gitter

n	Stellen	Phasen	Tausche	Gr.-Red.	dZl	dT	G/EP (max)	Z/EP	Zeit [sec]
32	339 / 13	432	5 063	75 538	12	23	174 (496)	0,05	23,4
32	506 / 18	562	7 066	109 334	20	25	194 (496)	0,06	34,6
64	341 / 8	632	12 436	449 975	8	39	711 (2 016)	0,34	215,5
64	506 /10	910	18 055	669 958	12	39	736 (2 016)	0,39	355,3
128	342 / 7	840	17 092	1 875 092	4	40	2 232 (8 128)	1,58	1 330,6
128	507 / 9	1252	31 399	3 595 814	6	50	2 872 (8 128)	2,23	2 789,7
256	354 / 8	750	15 799	3 998 355	2	42	5 331 (32 640)	5,92	4 441,0
256	509 / 9	1358	32 052	9 551 183	3	47	7 033 (32 640)	8,45	11 478,0

Tabelle 3.4: Neuer serieller Algorithmus – „Rucksack-Gitter“

n	Stellen	Phasen	Tausche	Gr.-Red.	eS	dZl	Z/EP	Zeit [sec]
32	339 / 13	20 012	9 981	28 532	5	19	0,0007	14,7
32	506 / 18	30 227	15 064	41 978	2	28	0,0008	23,8
64	341 / 8	47 573	23 495	148 374	304	15	0,0026	123,9
64	506 /11	73 339	36 232	231 892	77	21	0,0028	205,0
128	342 / 8	57 417	27 617	297 957	6 491	34	0,0071	407,4
128	507 / 9	108 821	52 705	654 392	3 268	38	0,0086	935,0
256	345 / 8	62 691	28 337	671 820	32 389	66	0,0234	1 468,2
256	509 / 9	119 372	54 623	1 342 880	35 449	88	0,0278	3 320,6

Tabelle 3.5: Schnorr/Euchner-Algorithmus – „Rucksack-Gitter“

3.5.5 Praktische Ergebnisse

Der neue Algorithmus 3.8 und der Schnorr/Euchner-Algorithmus 3.4 [57] wurden in der Programmiersprache „C“ implementiert. Bei beiden Algorithmen wurde die Originaltauschbedingung des LLL-Algorithmus mit $\delta = 0,9$ gewählt. Bei Algorithmus 3.8 wurde die Fehlerbedingung aus Satz 3.15 weggelassen und die Größenreduktion in der Reihenfolge durchgeführt, die bei parallelen Algorithmen verwendet werden muß (vgl. Abschnitt 3.6). Für die ganzzahlige Arithmetik wurde die Bibliothek „thalesII“ [45] verwendet. Funktionen zur Umwandlung einer langen Zahl in eine „double“-Zahl und umgekehrt wurden von Kay Wiese übernommen [73]. Die Tests wurden auf einer Sun Sparcstation 10 durchgeführt für Gitter, die aus Matrizen mit kleinen Einträgen durch iteratives Multiplizieren mit zufälligen unimodularen 2×2-Matrizen erzeugt wurden (Tabellen 3.2 und 3.3), und für Gitter, die bei der Lösung von „Rucksackproblemen“ Verwendung finden und gemäß [57] erzeugt wurden (Tabellen 3.4 und 3.5). Das Rucksackproblem und die Struktur der zur Lösung verwendeten Gitter sind in Anhang C kurz dargestellt.

Die Spalten der Tabellen sind folgendermaßen bezeichnet:

n	Dimension des Gitters
Stellen	Maximale Anzahl der Binärstellen der Einträge der Matrix zu Beginn bzw. Ende
Phasen	Anzahl der Einzelphasen
Tausche	Anzahl der Tausche
Gr.-Red.	Gesamtzahl der durchgeführten Größenreduktionen
eS	Anzahl der exakten Skalarprodukte beim Schnorr/Euchner-Algorithmus
dZl	Durchschnittliche binäre Zahlenlänge der Elemente während der Größenreduktion
dT	Durchschnittliche Zahl der Tausche pro Alltauschphase
G/EP (max)	Zahl der Größenreduktionen pro Einzelphase (maximal mögliche Zahl)
Z/EP	Durchschnittliche Zeit pro Einzelphase
Zeit [sec]	Benötigte CPU-Zeit in Sekunden

Eine Einzelphase ist dabei eine ODD- oder EVEN-Phase.

Die Erbebnisse lassen sich folgendermaßen zusammenfassen.

- Da bei dem neuen Algorithmus nicht an jeder Stelle getauscht werden kann, ist die Gesamtzahl der durchzuführenden Größenreduktionen höher als bei dem Schnorr/Euchner-Algorithmus. Diese bestimmt im wesentlichen die Laufzeit, da nur bei der Größenreduktion exakt gerechnet wird. Dafür werden bei der neuen Variante insgesamt weniger Tausche ausgeführt. Diese Beobachtung ist auch in [72] getroffen worden. Insgesamt sind die Laufzeiten des neuen Algorithmus bei den unimodular transformierten Gittern bis zu dem Faktor $1,6$ und bei den Rucksackgittern bis zu dem Faktor $3,5$ höher als die des Schnorr-Euchner-Algorithmus. Dies liegt daran, daß bei den unimodular transformierten Gittern tatsächlich viele Tausche parallel ausgeführt werden können, während dies bei den Rucksackgittern nicht in diesem Maße der Fall ist.

- Bei den unimodular transformierten Gittern ist, wie theoretisch berechnet, die Anzahl der Einzelphasen des neuen Algorithmus im wesentlichen linear im Produkt von Dimension und Bitzahl.

- Für beide Algorithmen ist bei den unimodular transformierten Gittern die Gesamtzahl der Tausche ungefähr proportional zu Dimension^2 $\cdot$ Bitzahl.

- Bei den unimodular transformierten Gittern werden im Gegensatz zu den Rucksackgittern viele Größenreduktionen pro Phase durchgeführt. Diese Beobachtung ist erst wichtig, wenn parallele Versionen des Algorithmus betrachtet werden.

- Die Größe der auftretenden Zahlen wird theoretisch ausführlich in Abschnitt 3.8 diskutiert. Als Ergebnis der hier praktisch durchgeführten Tests bleibt festzuhalten:

  - Bei beiden Algorithmen entstehen in der Basis während des Algorithmus keine wesentlich größeren Zahlen als in der Ausgangsbasis.

  - Die durchschnittliche Zahlengröße ist wesentlich kleiner.

  - Bei den Rucksackgittern ist die durchschnittliche Zahlengröße der exakten Zahlen bei der Größenreduktion für den neuen Algorithmus kleiner als bei dem Schnorr/Euchner-Algorithmus. Dies erklärt, warum der neue Algorithmus nur um den Faktor $3,5$ langsamer ist, obwohl wesentlich mehr Größenreduktionen durchgeführt werden. Andererseits ist bei den Rucksackgittern die Zeit für die Orthogonalisierung gegenüber der Zeit für die Größenreduktion nicht mehr vernachlässigbar. Dadurch verliert der neue Algorithmus wieder etwas, obwohl die entstehenden ganzen Zahlen kleiner sind.

    Bei den unimodular transformierten Gittern ist die durchschnittliche Zahlenlänge für den Schnorr/Euchner-Algorithmus bis um den Faktor 2 kleiner.

  - Die durchschnittliche binäre Länge der zu größenreduzierenden Elemente der Koeffizientenmatrix ist bei den unimodular transformierten Gittern für beide Algorithmen klein in der Größenordnung von 2. Dies gilt auch bei den Rucksackgittern für den neuen Algorithmus. Für den Schnorr/Euchner-Algorithmus wächst jedoch diese Länge mit der Dimension von 31 ($n = 32$) bis 154 ($n = 256$). Dies ist wohl dadurch erklärbar, daß bei dem Schnorr/Euchner-Algorithmus nur dort größenreduziert wird, wo aktuell große Zahlen stehen, während bei der neuen Variante gleichmäßig auf der ganzen Matrix gearbeitet wird.

Bemerkung: Bei den Rucksackgittern reicht die Zahlengröße, die von dem Datentyp „double“ zur Verfügung gestellt wird, ab einer Ausgangszahlengröße von ungefähr 600 Binärstellen nicht mehr aus. Ab dann benötigt man einen Datentyp für Gleitkommazahlen, bei dem die Exponentengröße und die Mantissengröße frei eingestellt werden können. Ein solcher Datentyp erhöht natürlich die Rechenzeit (vgl. [4]), so daß sich hier eine Parallelisierung erst recht lohnt. In der vorliegenden Arbeit wurden jedoch nur Beispiele mit dem Datentyp „double“ betrachtet.

3.6 Parallelisierung für n^2 Prozessoren

Es sollen nun Möglichkeiten angegeben werden, wie die neue Basisreduktionsvariante 3.8 parallelisiert werden kann. Dabei wird auch ein Satz über die parallele Zeitkomplexität hergeleitet. Die Ausführungen dieses und des nächsten Kapitels sind mehr theoretischer und allgemeiner Art. Konkrete Implementierungen und praktische Ergebnisse sind in Kapitel 3.9 beschrieben.

Zunächst soll eine parallele Implementierung für ein gitterförmig verbundenes Prozessorfeld von $n \times n$ Prozessoren angegeben werden. Eine Phase von Algorithmus 3.8 besteht im wesentlichen aus zwei Schritten:

- Orthogonalisierung mit Hilfe von Givens-Rotationen (s. Algorithmus 3.9)
- Größenreduktion (s. Algorithmus 3.1)

Beide Schritte sind auf einem $n \times n$-Gitter in Zeit $O(n)$ durchführbar, s. z.B. Schwiegelshohn und Thiele [60]. Dort werden Algorithmen der folgenden Form betrachtet:

Algorithmus 3.10

```
for i = 1 to n − 1 do
        for j = i + 1 to n do
                A ← f_ij(A)
        od
od
```

wobei f_{ij} eine Funktion der folgenden Form ist

$$\begin{aligned} a_{ik} &\leftarrow p_1 a_{ik} + p_2 a_{jk} \\ a_{jk} &\leftarrow p_3 a_{ik} + p_4 a_{jk} \\ a_{ki} &\leftarrow p_5 a_{ki} + p_6 a_{kj} \\ a_{kj} &\leftarrow p_7 a_{ki} + p_8 a_{kj} \end{aligned}$$

für $k = 1, \ldots, n$, wobei die Parameter p_l $(l = 1, \ldots, 8)$ mit Hilfe der Matrixelemente $a_{ii}, a_{ij}, a_{ji}, a_{jj}$ berechnet werden.

Die folgenden Ideen sind entscheidend für eine parallele Ausführung des Algorithmus:

- Die Funktion f_{ij} bearbeitet die Zeilen i und j und die Spalten i und j. Es ist klar, daß z.B. die Bearbeitung der Spalten (Zeilen) 1 und 4 einerseits und 2 und 3 andererseits parallel ausgeführt werden können. Dies sieht man leicht anhand einer analog zu der in Kapitel 2 beschriebenen Datenabhängigkeitsanalyse. Daher kann die folgende parallele Version von Algorithmus 3.10 betrachtet werden:

Algorithmus 3.11

for $k = 3$ to $2n - 1$ do
  for $i = \max\{1, k - n\}$ to $(k - 1)/2$ pardo
    $j = k - i$
    $A \leftarrow f_{ij}(A)$
  odrap
od

Beispiel 3.2 In Tabelle 3.6 sind die durchzuführenden Verknüpfungen (i,j) für den Fall $n = 6$ dargestellt. Dabei werden die Verknüpfungen, die in einer Zeile der Tabelle stehen, parallel durchgeführt.

k	Parallele Verknüpfungen (i, j)				
3	$(1,2)$				
4	$(1,3)$				
5	$(1,4)$	$(2,3)$			
6	$(1,5)$	$(2,4)$			
7	$(1,6)$	$(2,5)$	$(3,4)$		
8		$(2,6)$	$(3,5)$		
9			$(3,6)$	$(4,5)$	
10				$(4,6)$	
11					$(5,6)$

Tabelle 3.6: Parallele Verknüpfungen

- Die Bearbeitung zweier Zeilen (bzw. Spalten) kann wiederum parallelisiert werden. Dazu werden Pipelining-Methoden verwendet (s. z.B. [5]).
- Bei einer parallelen Implementierung auf einem Prozessornetz müssen die Zeilen und Spalten so ausgetauscht werden, daß gerade zu bearbeitende Zeilen und Spalten benachbart sind. Außerdem müssen die Elemente, mit denen die Parameter der Funktion berechnet werden, in einem Prozessor verfügbar sein und zwar so, daß die Parameter leicht über die ganze Zeile bzw. die ganze Spalte versendet werden können. Diese Punkte sind in dem in [60] angegebenen Schema gewährleistet.

Mit diesen Ideen ist eine parallele Implementierung von Algorithmen der Form 3.10 auf einem Prozessorfeld mit n^2 Prozessoren möglich, die Zeit $O(n)$ benötigt statt $O(n^3)$ wie bei der seriellen Ausführung.

Sowohl die Berechnung der Orthogonalisierung mit Hilfe von Givens-Rotationen als auch die Größenreduktion lassen sich in der Form von Algorithmus 3.10 angeben[16]. Daher läßt sich Algorithmus 3.8 wie in Algorithmus 3.12 angegeben parallelisieren. Eine Implementierung auf dem in [60] angegebenen Prozessornetzwerk ist dann einfach möglich.

[16]Bei der Größenreduktion wird die umgekehrte Reihenfolge gewählt, die analog parallelisierbar ist, s.u.

Algorithmus 3.12

Paralleler Gleitkomma-Alltauschbasisreduktionsalgorithmus
Eingabe: Gitterbasis $b_1, b_2, \ldots, b_n$
Ausgabe: Reduzierte Basis $b_1, b_2, \ldots, b_n$
Methode:

```
const odd=0
const even=1
reihenfolge ← odd
while (es kann noch getauscht werden) do
    MT ← Gleitkomma(b_1, ..., b_n)
    (MT entspricht der Matrix M^T von Abschnitt 3.2)

    Berechne die Gram-Schmidt Orthogonalisierung mit
    Givens Rotationen auf MT:
    for e = 3 to 2n − 1 do
        for j = max{1, e − n} to (e − 1)/2 pardo
            k = e − j
            Berechne eine Givens Rotation auf den Reihen k und j
            von MT, so daß das Element MT_{k,j} zu null rotiert wird
        odrap
    od
    for j = 1 to n pardo
        BN_j ← MT_{j,j} * MT_{j,j}
        (BN_j entspricht ||b*_j||^2)
        for i = j + 1 to n pardo
            MT_{j,i} ← MT_{j,i}/MT_{j,j}
        odrap
    odrap

    Berechne die Größenreduktion von (b_1, ..., b_n) und MT:
    for e = 2n − 1 downto 3 do
        for j = max{1, e − n} to (e − 1)/2 pardo
            k = e − j
            Bearbeite die Spalten k und j der Matrizen MT und
            (b_1, ..., b_n) zur Größenreduktion des Elementes MT_{j,k}:
            m ← ⌈MT_{j,k}⌋
            b_k ← b_k − m · b_j
            μ_k ← μ_k − m · μ_j
            (μ_k entspricht der k-ten Spalte von MT)
            if ((j = k − 1) and (k modulo 2=reihenfolge) and
            (BN_{k−1} > 2BN_k) and
            (Fehlerbedingung aus Satz 3.15) ) then
                tausche b_{k−1} und b_k and
                tausche μ_{k−1} und μ_k
            fi
        odrap
    od
```

```
        if (reihenfolge = odd) then
            reihenfolge ← even
        else
            reihenfolge ← odd
        fi
    od
    Führe noch eine Größenreduktion der Basis durch
```

Bemerkungen:

- Zur Parallelisierung der Größenreduktion wird das in [60] angegebene Schema in der umgekehrten Reihenfolge (aber auf demselben Prozessorfeld) durchgeführt. Die parallele Durchführung der Größenreduktion entspricht der seriellen Reihenfolge:

  ```
  for k = n downto 2 do
      for j = k − 1 downto 1 do
          Größenreduziere μ_{k,j} durch Bearbeitung der Spalten k und j
          Sei ⌈μ_{k,j}⌋ die nächste ganze Zahl zu μ_{k,j}
          b_k ← b_k − ⌈μ_{k,j}⌋ b_j
          μ_k ← μ_k − ⌈μ_{k,j}⌋ μ_j
      od
  od
  ```

  Dies unterscheidet sich von der ursprünglichen Reihenfolge bei der Größenreduktion (s. Algorithmus 3.1). Bei der Größenreduktion einer Spalte wird die Spalte also mit noch nicht reduzierten Spalten verarbeitet. Dies kann theoretisch zu größeren Zahlen führen[17], ist aber für eine parallele Ausführung unumgänglich. Die Datenabhängigkeiten sind so geartet, daß eine parallele Ausführung der Größenreduktion mit n^2 Prozessoren, so daß mit nur bereits größenreduzierten Spalten verknüpft wird, unmöglich ist. Dies ist in Bild 3.1 verdeutlicht. Ein Punkt (i, j) bedeutet dabei die Größenreduktion des Elementes (i, j) der Matrix M^T. Die Pfeile von unten nach oben verdeutlichen die Reihenfolge innerhalb einer Spalte, da die Elemente einer Spalte „von unten nach oben“ reduziert werden müssen. Elemente in der gleichen Spalte und Elemente in der gleichen Zeile dürfen nicht gleichzeitig reduziert werden. Außerdem kann ein Element (i, j) nicht gleichzeitig mit einem Element der Spalte i reduziert werden, da zur Reduzierung des Elementes (i, j) die Spalten i und j bearbeitet werden.[18] Damit ergeben sich die in Abbildung 3.1 angegebenen Parallelisierungsebenen, d.h. die Elemente auf einer Linie werden gleichzeitig größenreduziert.

[17] Das Problem der Größe der Zahlen wird in Abschnitt 3.8 ausführlich behandelt.

[18] Es sei denn, es existiert eine Kopie der Spalte i, da diese bei der Größenreduktion des Elementes (i, j) nicht geändert wird. Wenn man lokale Kopien der Spalten der Matrix anlegt, sind auch andere, einfachere Parallelisierungsschemata möglich (s. Abschnitt 3.9.4 und 3.9.5). Dies ändert aber natürlich nichts an dem Problem der großen Zahlen, da in diesem Fall die Kopien der Spalten auch noch nicht größenreduziert sind. D.h. man verwendet auch hier zur Größenreduktion noch nicht größenreduzierte Spalten.

Die Größenreduktionsreihenfolge für die Parallelisierung werde im folgenden *„parallele Reihenfolge“* genannt und die ursprüngliche Reihenfolge aus Algorithmus 3.1 *„serielle Reihenfolge“*.

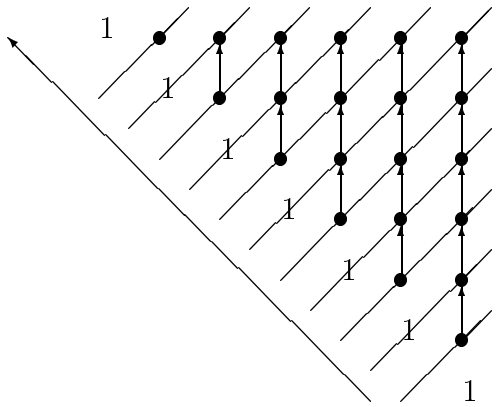

Abbildung 3.1: Parallelisierung der Größenreduktion der Elemente der Matrix M^T

- Eine exakte Berechnung der Gram-Schmidt Orthogonalisierung mit wurzel- und divisionsfreien Givens-Rotationen ist natürlich auf dieselbe Weise parallelisierbar. Damit ist auch der exakte Überprüfungsschritt, wenn auf Grund von möglichen Rundungsfehlern kein Tausch durchgeführt werden kann, parallel ausführbar.

- Die Speicherung der Normen BN_j kann so erfolgen, daß bei der Bearbeitung der Spalten $k-1$ und k die Tauschbedingung an der richtigen Stelle berechnet werden kann. Die Information, ob getauscht werden muß, wird dann zusammen mit $\lceil MT_{k-1,k} \rfloor$ über die gesamte Spalte versendet.

- Das Durchführen von Tauschen während der Größenreduktion stört den weiteren Reduktionsprozess nicht. Betrachte z.B. den Tausch der Spalten $i-1$ und i. Wenn diese Spalten getauscht werden, gilt:

  - Das Element $\mu_{i,i-1}$ wurde gerade größenreduziert.
  - Die Elemente $\mu_{i-1,j}$ und $\mu_{i,j}$ für $j = i-2, \ldots, 1$ sind noch nicht größenreduziert.

  D.h. bei beiden Spalten sind noch die gleichen Elemente größenzureduzieren. Außerdem wurden bereits alle Elemente $\mu_{j,i-1}$ mit $j = i, \ldots, n$ größenreduziert, d.h. während des weiteren Größenreduktionsprozesses ist es nicht störend, daß an der Stelle $(i-1, i-1)$ keine eins mehr steht. Zur Verdeutlichung dieser Aussagen betrachte man Abbildung 3.1, in der die parallele Größenreduktionsreihenfolge dargestellt ist.

- Die Terminierungsbedingung und die Bedingung für eine exakte Rechnung kann in der nächsten Phase zwischen den Prozessoren ausgetauscht werden.

- Die Neuberechnung des Wertes $\max_{i=1,\ldots,n} \|b_i\|^2$ für die Fehlerbedingung ist ebenfalls einfach parallelisierbar.

Insgesamt gilt also der folgende Satz[19]:

Satz 3.16 *Betrachtet man die Anzahl der durchzuführenden arithmetischen Operationen und die Anzahl der Kommunikationen von Zahlen, so kann eine Phase des Alltauschalgorithmus 3.8 auf einem $n \times n$-Gitter in Zeit $O(n)$ durchgeführt werden.*

Beweis: Betrachte Algorithmus 3.12 und die parallele Implementierung von [60]. ∎

Satz 3.16 gilt sowohl bei der Verwendung von Gleitkomma-Arithmetik als auch bei exakter Rechnung, da die Givens-Rotationen wie erwähnt auch exakt durchgeführt werden können. Bei exakter Arithmetik ist jedoch aufgrund praktischer Erwägungen eine parallele Version von Algorithmus 3.7 vorzuziehen. Aber auch hier gilt, daß eine Phase in Zeit $O(n)$ parallel durchgeführt werden kann [72, 50].

Insgesamt erhält man folgendes Korollar:

Korollar 3.17 *Ein kurzer Vektor eines Gitters kann in Zeit $O(n^2 \log B)$ auf einem Prozessorfeld von n^2 Prozessoren gefunden werden. Insbesondere läßt sich also eine Beschleunigung von $c \cdot n^2$ für eine Konstante c gegenüber dem seriellen LLL-Algorithmus erreichen.*

Beweis: Die Behauptung folgt mit Korollar 3.12 und Satz 3.16 für einen Algorithmus mit exakter Arithmetik. Die Komplexität des LLL-Algorithmus ist in Satz 3.5 angegeben. ∎

3.7 Parallelisierung für n Prozessoren

Es sollen nun parallele Algorithmen für den Fall untersucht werden, daß nur n Prozessoren zur Verfügung stehen, die linear oder ringförmig verbunden sind. Roch und Villard [50, 72] haben eine parallele Version des LLL-Algorithmus mit exakter Arithmetik angegeben. Hier werden weitere Parallelisierungsmöglichkeiten, insbesondere in Hinblick auf die Verwendung von Gleitkomma-Arithmetik, dargestellt.

Bei der Verwendung von n Prozessoren gibt es prinzipiell zwei Möglichkeiten:

- Parallelisierung von Stufenalgorithmen, z.B. des Schnorr-Euchner-Algorithmus (s. Abschnitt 3.4). Bei Stufenalgorithmen wird immer nur eine Spalte bearbeitet. Diese Bearbeitung kann parallelisiert werden.
- Parallelisierung von Alltauschalgorithmen, die stets auf der gesamten Matrix arbeiten.

Die Vorteile der ersten Möglichkeit sind dabei:

- Die serielle Version des Schnorr-Euchner Algorithmus arbeitet in der Praxis sehr gut.

[19] Villard [72] kommt unter der Verwendung von exakter Arithmetik und anderer Methoden auf ein ähnliches Ergebnis für die parallele Zeit für eine Phase.

- Es entstehen (auch theoretisch) keine größeren Zahlen als im Seriellen.

Die Parallelisierung des Schnorr-Euchner Algorithmus weist aber auch Nachteile auf:

- Der Korrekturschritt bei evtl. zu großen Gleitkommafehlern ist aufwendig. Wenn ein Skalarprodukt exakt neu berechnet werden muß, werden bei der parallelen Version gleich alle Skalarprodukte neu berechnet (s. Abschnitt 3.4.1). Es kann also nicht ausgenutzt werden, daß evtl. nur ein oder wenige Skalarprodukte exakt berechnet werden müssen.

- Die Kommunikationskosten betragen pro Stufe $11n(\beta+\gamma')$ (s. Satz 3.6). Insgesamt muß man also $O(n^3 \log B)$ mal die hohe Startup-Zeit beim Versenden einer Nachricht investieren. Da hier mit schneller Gleitkomma-Arithmetik gearbeitet werden soll, sind die Kommunikationskosten im Gegensatz zum LLL-Algorithmus nicht vernachlässigbar gegenüber den Berechnungskosten.

Die Parallelisierung des Schnorr-Euchner Algorithmus wurde schon ausführlich in Abschnitt 3.4 diskutiert. Daher soll nun die zweite Möglichkeit näher untersucht werden.

3.7.1 Parallelisierung von Alltauschalgorithmen

Die Verwendung von Alltauschalgorithmen hat den prinzipiellen Nachteil, daß theoretisch größere Zahlen entstehen können, als bei Stufenalgorithmen. Zusätzlich können bei einer Ausführung der Größenreduktion in der parallelen Reihenfolge größere Zahlen entstehen als bei einer Ausführung in der seriellen Reihenfolge. In der Praxis scheint dieses Problem der Zahlengröße jedoch keine bedeutende Rolle zu spielen (s. Abschnitt 3.5.5 und Abschnitt 3.8).

Bei der Parallelisierung von Alltauschalgorithmen für n Prozessoren gibt es verschiedene Möglichkeiten. Die beiden Hauptschritte, Berechnung der Orthogonalisierung und der Größenreduktion, sind auf zwei Arten parallelisierbar. Diese beiden Arten sollen am Beispiel der Größenreduktion erklärt werden:

- Wie bei Stufenalgorithmen werden die Spalten *nacheinander* größenreduziert. Die Größenreduktion einer Spalte wird parallel ausgeführt wie bei der Parallelisierung des Schnorr-Euchner Algorithmus (s. Abschnitt 3.4 und Tabelle 3.1). Auch das Tauschen kann wie dort beschrieben behandelt werden. Insbesondere kann die Größenreduktion in der seriellen Reihenfolge durchgeführt werden. Dazu wird die Matrix *zeilenweise* auf die Prozessoren verteilt, d.h. jeder Prozessor verwaltet eine Zeile. Da die parallele Reduktion einer Spalte mit Pipelining erfolgt, soll dieses Verfahren *„Pipelining-Methode“* genannt werden.

- Verschiedene Spalten werden *gleichzeitig* miteinander verknüpft.

  Dazu wird die Matrix *spaltenweise* auf die Prozessoren verteilt, d.h. ein Prozessor verwaltet je zwei Spalten. Verschiedene Verknüpfungen von je zwei Spalten zum Zwecke der Größenreduktion eines Elementes $\mu_{k,j}$ können nun parallel ausgeführt werden. Dabei übernimmt je ein Prozessor die vollständige Verknüpfung zweier Spalten gemäß der Formel $b_k \leftarrow b_k - \lceil \mu_{k,j} \rfloor b_j$ zur Größenreduktion.

Dies geschieht nach dem Schema, das schon bei der Parallelisierung für n^2 Prozessoren angewendet wurde (s. Algorithmus 3.12 von Seite 69 und Abbildung 3.1 von Seite 71):

for $e = 2n - 1$ downto 3 do
  for $j = \max\{1, e - n\}$ to $(e - 1)/2$ pardo
    $k = e - j$
    Bearbeite die Spalten k und j der Matrizen MT und
    $(b_1, \ldots, b_n)$ zur Größenreduktion des Elementes $MT_{j,k}$
    $m \leftarrow \lceil MT_{j,k} \rfloor$
    $b_k \leftarrow b_k - m \cdot b_j$
    $\mu_k \leftarrow \mu_k - m \cdot \mu_j$
    (μ_k entspricht der k-ten Spalte von MT)
  odrap
od

Damit jeder Prozessor immer die richtigen zu verknüpfenden Spalten besitzt, müssen diese jeweils nach einem parallelen Schritt ausgetauscht werden. Nach dem in [60] vorgeschlagenen Schema geschieht dies zyklisch. Daher wird diese Methode *„zyklische Methode“* genannt. Hier muß wie bei der Parallelisierung für n^2 Prozessoren die parallele Reduktionsreihenfolge verwendet werden (s. die ausführliche Diskussion aus Abschnitt 3.6). Das Durchführen der Tausche kann wie dort während der Größenreduktionsphase geschehen.

Die Möglichkeiten zur Parallelisierung der Orthogonalisierung mit Hilfe von Givens-Rotationen sind prinzipiell dieselben, nur werden hier Zeilen und nicht Spalten miteinander verarbeitet.

Bei der Verwendung von nur n Prozessoren könnte man auch die Formeln von Schnorr und Euchner (s. Algorithmus 3.4) zur parallelen Berechnung der Orthogonalisierung verwenden. Dies soll hier aber nicht weiter ausgeführt werden.

Es sollen nun anhand der Größenreduktion Abschätzungen für den Zeitaufwand der verschiedenen Möglichkeiten angegeben werden.[20]

Zyklische Größenreduktion

Bei der Durchführung der Größenreduktion der Matrix mit Hilfe zyklischer Spaltenoperationen sind $\frac{n}{2}$ Prozessoren erforderlich. Ein Prozessor verwaltet je zwei Spalten. Die parallele Größenreduktion besteht bei dem obigen zyklischen Algorithmus aus $2n-3$ Stufen. In jeder Stufe größenreduziert ein Prozessor ein Element $\mu_{k,j}$ durch die Bearbeitung der Spalten j und k. Dies erfordert gemäß obiger Formeln $2n + 1$ arithmetische Operationen auf Zahlen in genauer Darstellung und $2n$ arithmetische Operationen auf Zahlen in Gleitkommadarstellung. Dies ergibt eine *Gesamtberechnungszeit* von $4n^2\alpha + 4n^2\alpha'$.

[20] Der Einfachheit halber sei angenommen, daß die entsprechende Anzahl von Prozessoren zur Verfügung steht. Ist dies nicht der Fall, muß ein Prozessor die Aufgabe von mehreren Prozessoren übernehmen. Bei den Angaben der Zeiten werden die konstanten Terme vernachlässigt.

Nach der Durchführung einer Stufe wird eine Spalte der Basis (in genauer Darstellung) und eine Spalte der Matrix M^{T} versendet. Dies ergibt als *Gesamtkommunikationszeit* $2n\beta + 2n^2\gamma + 2n^2\gamma'$.

- Vorteil: Die hohe Startup-Zeit erfolgt nur n-mal. Es werden nur wenige, aber große Pakete versendet.
- Nachteil: Es können theoretisch große Zahlen entstehen, was in den praktischen Untersuchungen aber nicht beobachtet werden konnte (s. Abschnitt 3.5.5).

Größenreduktion mit Pipelining

Zur Durchführung der Größenreduktion mit Pipelining werden n Prozessoren benutzt. Jeder Prozessor verwaltet eine Zeile der Matrix. Nach der Analyse des parallelen Schnorr-Euchner-Algorithmus dauert die Größenreduktion der k-ten Spalte $n+2k-4$ Stufen. Pro Stufe muß ein Prozessor maximal die folgenden Operationen durchführen:

- Berechnung von 2 genauen und 2 ungenauen arithmetischen Operationen
- Versendung von 2 Paketen mit einer konstanten Anzahl von Zahlen in Gleitkommadarstellung.

Zusätzlich werden insgesamt n genaue arithmetische Operationen durchgeführt (Berechnung der Werte $\lceil \mu_{i,j} \rfloor$).

Damit ergeben sich also folgende Abschätzungen:

- *Berechnungszeit*: $7n^2\alpha + 6n^2\alpha'$
- *Kommunikationszeit*: $6n^2(\beta + \gamma')$
- Nachteil: Die hohe Startup-Zeit muß n^2-mal aufgewendet werden, da viele kleine Pakete verschickt werden müssen.
- Vorteil: Es entstehen keine größeren Zahlen als bei der seriellen Reihenfolge. Es werden nur Gleitkommazahlen verschickt, d.h. die Versendungszeit ist nicht abhängig von der Größe der entstehenden Zahlen.

Totalaustausch-Schritt

Soll die Orthogonalisierung mit Givensrotationen zyklisch auf den Zeilen und die Größenreduktion zyklisch auf den Spalten durchgeführt werden, so ist nach der Orthogonalisierung ein Totalaustausch-Schritt (s. [5]) zur Umorganisation der Matrix von einer zeilenweisen auf eine spaltenweise Aufteilung erforderlich.

Die Kommunikationskosten betragen hierbei auf einem Prozessorring $\frac{n^2}{32}(\beta + 2\gamma + 2\gamma')$. Es muß also wieder n^2 mal die Startup-Zeit aufgebracht werden.

Steht hingegen eine Hyperwürfel-Architektur zur Verfügung (s. z.B. [5]) kann ein Totalaustausch schneller durchgeführt werden mit Kommunikationskosten

$$\frac{n}{4}(\beta + 2\gamma + 2\gamma') \tag{3.45}$$

bzw.

$$\frac{n}{4}\log\frac{n}{2}(\beta+2\gamma+2\gamma'), \tag{3.46}$$

wenn in einem Zeitschritt nur über eine ausgehende Kante eine Nachricht versendet werden kann.

3.8 Größe der entstehenden Zahlen und binäre Komplexität

Beginnt man mit einer ganzzahligen Gitterbasis, sind alle Zahlen während der Berechnung rational. Es kann also mit genauer ganzzahliger Arithmetik gerechnet werden. Bisher wurden die Berechnungszeiten der Algorithmen mit Hilfe der Anzahl der durchzuführenden arithmetischen Operationen und die Kommunikationszeiten mit Hilfe der Anzahl der zu versendenden Zahlen angegeben. Zur Berechnung der tatsächlichen Zeiten müssen auch die Zeiten für eine arithmetische Operation und zum Versenden einer Zahl betrachtet werden. Diese sind (bei Verwendung einer genauen Arithmetik) abhängig von der Größe der Zahlen.

Zunächst werden bekannte Abschätzungen für die Zahlengröße bei den verschiedenen Verfahren angegeben. Dann wird in Abschnitt 3.8.2 eine neue, bessere Analyse für den parallelen Fall vorgestellt. In Abschnitt 3.8.3 werden die sich daraus ergebenden binären Komplexitäten betrachtet. Es folgen eine kurze Diskussion und eine Beschreibung des Verfahrens von Joux zur Vermeidung großer Zahlen bei der parallelen Größenreduktion.

3.8.1 Abschätzung der Zahlengröße

Die Abschätzungen für die Stufenalgorithmen gemäß dem LLL-Algorithmus stammen aus [36, 29], die Abschätzungen für die Alltauschalgorithmen und für den parallelen Fall aus [72].

Zunächst wird die Größe der Quadratnormen der Gram-Schmidt Orthogonalisierung $\|b_i^*\|^2$ abgeschätzt.

Lemma 3.18 ([36]) *Während des gesamten Reduktionsprozesses gilt*[21]

$$\|b_i^*\|^2 \le B \qquad \text{für} \quad i=1,\ldots,n \tag{3.47}$$

Beweis: Nach der Konstruktion der Gram-Schmidt Orthogonalisierung gilt die Behauptung am Anfang. Die Gram-Schmidt Orthogonalisierung wird nur durch einen Tausch verändert. Gemäß den Formeln (3.10) – (3.14) für die neue Gram-Schmidt Orthogonalisierung nach einem Tausch wird das Maximum aller $\|b_i^*\|^2$ nicht größer. ∎

Lemma 3.19 ([72]) *Für die Gram-Schmidt Orthogonalisierung der Ausgangsbasis gelte*

$$\frac{1}{\beta} \le \|b_i^*\|^2 \qquad \text{für} \quad i=1,\ldots,n \tag{3.48}$$

für ein $\beta \in \mathbf{Z}$. Dann gilt dies während des gesamten Reduktionsprozesses.

[21] Zur Erinnerung: $B = \max_{i=1,\ldots,n}\|b_i\|^2$, wobei die b_i die Vektoren der Ausgangsbasis sind.

Beweis: Analog zu obigem Beweis gilt, daß das Minimum der $\|b_i^*\|^2$ nach einem Tausch nicht kleiner wird. ∎

Es können nun die entstehenden Nenner abgeschätzt werden [36]. Dazu werden die folgenden Größen betrachtet für $i = 1, \ldots, n$:

$$d_i = \det((b_j^{\mathrm{T}} \cdot b_l))_{1 \le j,l \le i} \tag{3.49}$$

$$= \prod_{j=1}^{i} \|b_j^*\|^2 \tag{3.50}$$

Somit sind diese Werte ganzzahlig.

Lemma 3.20 ([36]) *Ausgehend von einer ganzzahligen Basis gilt für eine Gram-Schmidt Orthogonalisierung* $(b_1, \ldots, b_n) = (b_1^*, \ldots, b_n^*)M^{\mathrm{T}}$*:*

- *Die Basisvektoren* b_i *sind immer ganzzahlig.*
- $\|b_i^*\|^2 = \frac{d_i}{d_{i-1}}$
- $d_j \mu_{i,j}$ *ist ganzzahlig*

Lemma 3.21 ([36]) *Die entstehenden rationalen Zahlen können so abgespeichert werden, daß die Nenner kleiner als* B^n *sind.*

Beweis:

- Die Gram-Schmidt Beziehung $(b_1, \ldots, b_n) = (b_1^*, \ldots, b_n^*)M^{\mathrm{T}}$ bleibt während der Ausführung des gesamten Algorithmus bestehen, bzw. wird nach der Durchführung von Tauschen wieder repariert, so daß sie danach wieder gilt. Also gelten die Aussagen von Lemma 3.20 während der gesamten Ausführung des Algorithmus.
- Die Werte d_i werden abgespeichert. Sie können gemäß Lemma 3.20 als Nenner benutzt werden. Genauer gesagt werden die Werte $\mu_{i,j}$ in der Form von Zähler und Nenner gespeichert, wobei nicht gekürzt wird. Erst wenn der Nenner bei einer Berechnung größer als d_j wird, wird der Zähler durch ($d_j \cdot$ Zähler)/Nenner ersetzt und der Nenner durch d_j. Damit kann also erreicht werden, daß die Nenner kleiner als d_j bleiben. Die Werte $\|b_i^*\|^2$ werden analog behandelt. Die Aussage folgt dann mit (3.47).

∎

Damit folgt für β in (3.48):

$$\beta \le B^n \tag{3.51}$$

Zur Abschätzung der Zählergrößen reicht es nun, die Größe der Zahlen abzuschätzen. Der Zähler ist dann kleiner als Zahlengröße × Nennergröße. Die Größe der $\|b_i^*\|^2$ wurde bereits angegeben. Es werden nun also die Werte von $\|b_i\|$ und $\mu_{i,j}$ betrachtet.

Größe nach der Größenreduktion

Lemma 3.22 ([36]) *Nach der Größenreduktion gilt*

$$|\mu_{i,j}| \leq \frac{1}{2} \quad \text{für} \quad 1 < j < i \leq n \tag{3.52}$$

$$\|b_i\|^2 \leq iB \quad \text{für} \quad 1 \leq i \leq n \tag{3.53}$$

Beweis: (3.52) ist nach der Konstruktion der Größenreduktion erfüllt, s. auch Definition 3.3. Nach der Gram-Schmidt Formel (3.7) gilt

$$\|b_i\|^2 = \sum_{j=1}^{i} \mu_{i,j}^2 \|b_j^*\|^2$$

Mit (3.52) und (3.47) folgt (3.53). ∎

Größe nach der Gram-Schmidt Berechnung

Die Basis $(b_1, \ldots, b_n)$ wird bei der Berechnung der Gram-Schmidt Orthogonalisierung nicht verändert, d.h. (3.53) gilt weiterhin. Für die Gram-Schmidt Koeffizienten $\mu_{i,j}$ gilt folgendes Lemma:

Lemma 3.23 ([36, 29]) *Nach der Berechnung der Gram-Schmidt Orthogonalisierung gilt:*

- *Bei Stufenalgorithmen:* $\mu_{i,j}^2 \leq n2^n B$
- *Bei Alltauschalgorithmen:* $\mu_{i,j}^2 \leq i\beta B$

Beweis: Nach der Gram-Schmidt Orthogonalisierung (3.6) gilt:

$$\begin{aligned} \mu_{i,j}^2 &= \frac{(b_i^{\mathrm{T}} \cdot b_j^*)^2}{\|b_j^*\|^4} \\ &\leq \frac{\|b_i\|^2 \|b_j^*\|^2}{\|b_j^*\|^4} \\ &= \frac{\|b_i\|^2}{\|b_j^*\|^2} \end{aligned} \tag{3.54}$$

Für Alltauschalgorithmen gilt mit (3.53) und (3.48) die Behauptung. Für Stufenalgorithmen kann eine bessere untere Schranke für $\|b_j^*\|^2$ angegeben werden [29]. Da die Basis bis zu der betrachteten Stufe k bereits reduziert ist, gilt:

$$\|b_j^*\|^2 \geq \frac{\|b_1^*\|^2}{2^{j-1}}$$

Mit $\|b_1^*\| = \|b_1\| \geq 1$ folgt nun die Behauptung. ∎

Größe während der Reparierung der Gram-Schmidt Berechnung

Während der Neuberechnung der Werte von $\|b_i^*\|^2$ und $\|b_{i+1}^*\|^2$ und der entsprechenden Koeffizienten entstehen keine größeren Zwischenwerte (s. Formeln (3.17) - (3.25)).

Größe während der Größenreduktion

Während der Größenreduktion können größere Zahlen entstehen. Wie in Abschnitt 3.6 gesehen, muß man zwischen den Reduktionsreihenfolgen bei einer seriellen und einer parallelen Implementierung unterscheiden, da hier unterschiedliche Größen entstehen können. Bei der „seriellen Reihenfolge“ werden die Spalten „von links nach rechts“ und bei der „parallelen Reihenfolge“ „von rechts nach links“ größenreduziert, d.h. bei der seriellen Reihenfolge wird eine Spalte bei ihrer Größenreduktion nur mit bereits größenreduzierten Spalten verknüpft, während dies bei der parallelen Reihenfolge nicht der Fall ist.

Lemma 3.24 ([36, 72]) *Vor Beginn der Größenreduktion gelte $\mu_{i,j}^2 \leq M$ für $1 \leq j < i \leq n$. Dann gilt während der Größenreduktion in der seriellen Reihenfolge*

$$\mu_{i,j}^2 \leq 4^n M \quad \text{für} \quad 1 \leq j < i \leq n \tag{3.55}$$

$$\|b_i\|^2 \leq nB4^n M \quad \text{für} \quad 1 \leq i \leq n \tag{3.56}$$

Bei der Durchführung in der parallelen Reihenfolge gilt:

$$\mu_{i,j}^2 \leq 4^n M^n \quad \text{für} \quad 1 \leq j < i \leq n \tag{3.57}$$

$$\|b_i\|^2 \leq nB4^n M^n \quad \text{für} \quad 1 \leq i \leq n \tag{3.58}$$

Beweis: Betrachte die Größenreduktion einer Spalte k. Bei der Größenreduktion des Elementes $\mu_{k,k-1}$ wird berechnet $\mu_{k,i} \leftarrow \mu_{k,i} - \lceil \mu_{k,k-1} \rfloor \mu_{k-1,i}$ für $i = 1, \ldots, k-1$. Bei der seriellen Reihenfolge gilt $|\mu_{k-1,i}| \leq \frac{1}{2}$, da die Spalte $k-1$ bereits größenreduziert ist, und $|\mu_{k,i}|, |\mu_{k,k-1}| \leq \sqrt{M}$. Damit gilt für den neuen Wert von $|\mu_{k,i}| \leq 2\sqrt{M}$. Induktiv gilt dann die Behauptung nach der Größenreduktion der Elemente $\mu_{k,j}$ für $j = k-2, \ldots, 1$.

Bei der parallelen Reduktionsreihenfolge gilt jedoch nur $|\mu_{k-1,i}| \leq \sqrt{M}$, da hier die Spalte $k-1$ noch nicht größenreduziert ist. Und somit gilt für den neuen Wert von $|\mu_{k,i}| \leq 2M$. Induktiv folgt auch hier die Behauptung.

Die Abschätzungen für $\|b_i\|^2$ ergeben sich aus der Gram-Schmidt Beziehung (3.7), die während der gesamten Größenreduktion ihre Gültigkeit behält

$$\|b_i\|^2 = \sum_{j=1}^{i} \mu_{i,j}^2 \|b_j^*\|^2$$

und (3.47). ∎

Insgesamt ergibt sich also folgende Situation:

	Stufenalg.	Alltausch (seriell)	Alltausch (parallel)
$\mu_{i,j}^2$ zwischen den Schritten	$n2^n B$	$n\beta B$	$n\beta B$
$\|b_i\|^2$ zwischen den Schritten	nB	nB	nB
$\mu_{i,j}^2$ während Größenreduktion	$n2^{3n} B$	$4^n n\beta B$	$(4n\beta B)^n$
$\|b_i\|^2$ während Größenreduktion	$n^2 2^{3n} B^2$	$4^n n^2 \beta B^2$	$nB(4n\beta B)^n$

Dabei gilt $\beta \leq B^n$ bei beliebigen Gittern.

Bei Gittern, die zur Lösung von Rucksackproblemen betrachtet werden, kann eine bessere Abschätzung für β angegeben werden [72]:

Lemma 3.25 *[72] Bei der Basisreduktion von Gittern, die gemäß [72] für die Lösung von Rucksackproblemen verwendet werden, gilt*

$$\beta \leq n^2 B^4 \tag{3.59}$$

Damit erhält man folgenden Satz:

Satz 3.26 *Bei der Basisreduktion von Rucksackgittern entstehen größenordnungsmäßig bei der Verwendung von Alltauschalgorithmen mit der seriellen Größenreduktionsreihenfolge keine größeren Zahlen als bei der Verwendung von Stufenalgorithmen.*

3.8.2 Genauere Analyse der Zahlengröße bei der parallelen Größenreduktion

Ist für β nur die Abschätzung $\beta \leq B^n$ möglich, so könnten nach den obigen Abschätzungen bei der parallelen Größenreduktion Zahlen in der Größenordnung $B^{(n^2)}$ auftreten.

Dies liegt daran, daß sowohl bei der Berechnung der Orthogonalisierung als auch bei der parallelen Größenreduktion eine „Zahlenexplosion“ stattfinden kann.

Für beide Fälle soll nun ein Beispiel angegeben werden.

Beispiel 3.3 Betrachte folgende $(n \times n)$-Basis mit $D > 1 \in \mathbf{Z}$:

$$\begin{pmatrix} D & D+1 & \cdots & \cdots & D+1 & 0 \\ 0 & D & D+1 & \cdots & D+1 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & D & D+1 & 0 \\ 1 & \cdots & 1 & 1 & 1 & 1 \\ 0 & \cdots & \cdots & 0 & 0 & 1 \end{pmatrix}$$

Für die Gram-Schmidt-Orthogonalisierung gilt, daß $\|b^*_{n-1}\|$ in der Größenordnung von $1/D^n$ liegt und $\mu_{n,n-1}$ somit in der Größenordnung von D^n.

Zum Beweis betrachte man die Berechnung von $\|b^*_{n-1}\|^2$ und $\mu_{n,n-1}$ mit Hilfe von Givens-Rotationen. Seien $m_{k,j}$ die Elemente der obigen Matrix. Dann müssen die Elemente $m_{n-1,1}, \ldots, m_{n-1,n-2}$ zu Null rotiert werden. Bezeichne $m^{(0)}_{k,j}$ die Matrixelemente zu Beginn und $m^{(i)}_{k,j}$ das Element $m_{k,j}$ nach der Rotierung von $m_{n-1,i}$ zu Null. Dann gilt:

$$\|b^*_{n-1}\|^2 = {m^{(n-2)}_{n-1,n-1}}^2 \tag{3.60}$$

$$\mu_{n,n-1} = \frac{m^{(n-2)}_{n-1,n}}{m^{(n-2)}_{n-1,n-1}} \tag{3.61}$$

Behauptung 3.27 *Es gilt*

$$m_{n-1,j}^{(i)} = \frac{(-1)^i}{\sqrt{D^{2i} + D^{2(i-1)} + \ldots + 1}}$$

für $0 \leq i < n-1$ *und* $i < j \leq n-1$ *und*

$$m_{n-1,n}^{(i)} = \frac{D^i}{\sqrt{D^{2i} + D^{2(i-1)} + \ldots + 1}}$$

für $0 \leq i < n-1$.

Beweis: Die Behauptung gilt für $i = 0$. Unter der Annahme, daß sie für $i \geq 0$ gilt, wird nun das Element $m_{n-1,i+1}^{(i)}$ zu Null rotiert. Unter der Induktionsannahme für $m_{n-1,i+1}^{(i)}$ ergeben sich als Rotationsparameter

$$c = \frac{D}{\sqrt{D^2 + \frac{1}{D^{2i}+D^{2(i-1)}+\ldots+1}}} = \frac{D\sqrt{D^{2i} + D^{2(i-1)} + \ldots + 1}}{\sqrt{D^{2(i+1)} + D^{2i} + \ldots + 1}}$$

und

$$s = \frac{(-1)^i}{\sqrt{D^{2i} + D^{2(i-1)} + \ldots + 1}\sqrt{D^2 + \frac{1}{D^{2i}+D^{2(i-1)}+\ldots+1}}} = \frac{(-1)^i}{\sqrt{D^{2(i+1)} + D^{2i} + \ldots + 1}}$$

Damit gilt für alle $m_{n-1,j}^{(i+1)}$ mit $i+1 < j \leq n-1$

$$\begin{aligned} m_{n-1,j}^{(i+1)} &= -s(D+1) + cm_{n-1,j}^{(i)} \\ &= -\frac{(-1)^i(D+1)}{\sqrt{D^{2(i+1)} + D^{2i} + \ldots + 1}} + \frac{D(-1)^i}{\sqrt{D^{2(i+1)} + D^{2i} + \ldots + 1}} \\ &= (-1)^{i+1}\frac{1}{\sqrt{D^{2(i+1)} + D^{2i} + \ldots + 1}} \end{aligned}$$

Für $m_{n-1,n}^{(i+1)}$ gilt:

$$\begin{aligned} m_{n-1,n}^{(i+1)} &= -s \cdot 0 + cm_{n-1,n}^{(i)} \\ &= \frac{D\sqrt{D^{2i} + D^{2(i-1)} + \ldots + 1}}{\sqrt{D^{2(i+1)} + D^{2i} + \ldots + 1}}\frac{D^i}{\sqrt{D^{2i} + D^{2(i-1)} + \ldots + 1}} \\ &= \frac{D^{i+1}}{\sqrt{D^{2(i+1)} + D^{2i} + \ldots + 1}} \end{aligned}$$

■

Also gilt:

$$\begin{aligned} \mu_{n,n-1} &= \frac{m_{n-1,n}^{(n-2)}}{m_{n-1,n-1}^{(n-2)}} && (3.62) \\ &= (-1)^{n-2}D^{n-2} && (3.63) \end{aligned}$$

Beispiel 3.4 Betrachte folgende Basis mit $D > 1 \in \mathbf{Z}$:

$$\begin{pmatrix} 1 & D & \cdots & D \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & D \\ 0 & \cdots & 0 & 1 \end{pmatrix}$$

Die Matrix M^{T} der Gram-Schmidt-Koeffizienten sieht dann genauso aus. Bei der Größenreduktion der Spalte n entstehen Zwischenwerte der Größenordnung D^n.

In diesem Abschnitt soll nun gezeigt werden, daß beide Effekte aber nicht gleichzeitig auftreten können. Insgesamt bleiben die Zahlen in der Größenordnung von B^n.

Dies hat folgenden Grund: Bei der Orthogonalisierung entstehen nur dann sehr große Zahlen, wenn $\|b_i^*\|^2$ sehr klein ist. Dann sind die Zahlen der i-ten Zeile der Matrix M^{T} groß. Dies kann aber nicht für alle $\|b_i^*\|^2$ gelten, da $\|b_1^*\|^2 \cdot \ldots \cdot \|b_n^*\|^2$ ganzzahlig ist und somit ≥ 1. Bei der Größenreduktion entstehen nur Produkte von Gram-Schmidt-Koeffizienten μ *verschiedener* Zeilen. Daher findet keine Akkumulation sehr großer Werte statt. Dies soll in folgendem Lemma bewiesen werden.

Lemma 3.28 *Bei der Größenreduktion der Spalte k, $2 \leq k \leq n$, d.h. der Elemente $\mu_{k,k-1}, \ldots, \mu_{k,1}$, wurden nach der Größenreduktion des Elementes $\mu_{k,k-i}$ mit $1 \leq i \leq k-2$ von jedem Koeffizienten $\mu_{k,k-j}$, $j > i$, $2^i - 1$ Produkte von Gram-Schmidt-Koeffizienten subtrahiert. In jedem Produkt kommen nur (gerundete) Koeffizienten verschiedener Zeilen der Matrix M^{T} vor und nur Koeffizienten der Zeilen $k-1, k-2, \ldots, k-i$ und $k-j$.*

Beweis: Der Beweis erfolgt durch Induktion über i:

$\mu_{x,y}^{(0)}$ bezeichne $\mu_{x,y}$ vor der Größenreduktion der Spalte k und $\mu_{x,y}^{(i)}$ bezeichne $\mu_{x,y}$ nach der Größenreduktion von $\mu_{k,k-i}$.

$i = 1$: Nach der Größenreduktion von $\mu_{k,k-1}$ gilt für $j > 1$:

$$\mu_{k,k-j}^{(1)} = \mu_{k,k-j}^{(0)} - \lfloor \mu_{k,k-1}^{(0)} \rceil \mu_{k-1,k-j}^{(0)}$$

Damit gilt die Behauptung für $i = 1$.

Die Behauptung gelte nun für i. Nach der Größenreduktion von $\mu_{k,k-i-1}$ gilt für $j > i$:

$$\mu_{k,k-j}^{(i+1)} = \mu_{k,k-j}^{(i)} - \lfloor \mu_{k,k-i-1}^{(i)} \rceil \mu_{k-i-1,k-j}^{(0)}$$

Mit der Induktionsannahme für $\mu_{k,k-j}^{(i)}$ und $\mu_{k,k-i-1}^{(i)}$ gilt die Behauptung auch für $i+1$. ■

Damit folgt für die Zahlengröße bei der parallelen Größenreduktion:

Satz 3.29 *Für die bei der parallelen Größenreduktion auftretenden Zahlen gilt:*

$$\mu_{i,j}^2 \leq n^n (2B)^{2n} \quad \textit{für} \quad 1 \leq j < i \leq n \tag{3.64}$$

$$\|b_i\|^2 \leq nBn^n (2B)^{2n} \quad \textit{für} \quad 1 \leq i \leq n \tag{3.65}$$

Beweis: Nach dem vorherigen Lemma treten bei der Rechnung nur Summen mit höchstens 2^n Summanden auf. Jeder Summand ist das Produkt von Koeffizienten verschiedener Zeilen. Nach (3.54) gilt:

$$\prod \mu_{i,j}^2 \leq \prod \frac{\|b_i\|^2}{\|b_j^*\|^2}$$

Nach (3.53) gilt $\|b_i\|^2 \leq nB$ und somit folgt

$$\prod \|b_i\|^2 \leq (nB)^n$$

Da $\|b_1^*\|^2 \cdot \ldots \cdot \|b_n^*\|^2 \geq 1$ und $\|b_i^*\|^2 \leq B$ für alle $1 \leq i \leq n$ und in dem Produkt nur verschiedene b_j^* vorkommen, gilt:

$$\prod \|b_j^*\|^2 \geq \frac{1}{B^n}$$

Damit gilt für jedes auftretende Produkt

$$\begin{aligned} \prod \mu_{i,j}^2 &\leq n^n B^{2n} \\ |\prod \mu_{i,j}| &\leq n^{n/2} B^n \end{aligned}$$

Da in einer Summe maximal 2^n Produkte auftreten, gilt für alle auftretenden Zwischenwerte der Koeffizienten

$$\begin{aligned} |\mu| &\leq n^{n/2}(2B)^n \\ \mu^2 &\leq n^n(2B)^{2n} \end{aligned}$$

Die Behauptung für die auftretenden Zwischenwerte der b_i folgt wieder aus der Gram-Schmidt-Beziehung, die während der gesamten Größenreduktionsphase gültig ist. ∎

Falls für β keine andere Abschätzung als $\beta \leq B^n$ möglich ist, ergibt sich also insgesamt folgende Situation:

	Stufenalg.	Alltausch (seriell)	Alltausch (parallel)
$\mu_{i,j}^2$ zwischen den Schritten	$n2^nB$	nB^{n+1}	nB^{n+1}
$\|b_i\|^2$ zwischen den Schritten	nB	nB	nB
$\mu_{i,j}^2$ während Größenreduktion	$n2^{3n}B$	$n4^nB^{n+1}$	$n^n(2B)^{2n}$
$\|b_i\|^2$ während Größenreduktion	$n^22^{3n}B^2$	$n^24^nB^{n+2}$	$nBn^n(2B)^{2n}$

Bemerkung: Bereits in [27, 28] wird als Zahlenlänge $O(n \log B)$ angegeben jedoch ohne jeglichen Kommentar, Analyse oder gar Beweis. In [72] wird die Zahlenlänge mit $O(n^2 \log B)$ angegeben.

3.8.3 Binäre Komplexität

Nach der obigen Analyse der Größe der entstehenden Zahlen ergeben sich die folgenden binären Zeitkomplexitäten für Algorithmen, die nur mit genauer ganzzahliger, serieller Standardarithmetik arbeiten. Unter der Verwendung von Standardarithmetik benötigt eine Multiplikation von zwei l-stelligen Zahlen binäre Zeit $O(l^2)$, Addition und Subtraktion $O(l)$ [31].

Bei der asymptotischen Abschätzung wird $n < B$ angenommen.

Satz 3.30 (Binäre Zeitkomplexität des LLL Algorithmus [29]) *Der LLL - Algorithmus hat die binäre Zeitkomplexität* $O(n^5(\log B)^3 + n^6(\log B)^2)$.

Beweis: Die Reparierung der Gram-Schmidt Orthogonalisierung einer Spalte kostet $O(n)$ arithmetische Operationen auf ganzen Zahlen der binären Länge $O(n \log B)$ nach Lemma 3.21. Eine arithmetische Operation auf zwei ganzen Zahlen der Länge k kostet i.a. $O(k^2)$. Da maximal $n^2 \log B$ Phasen durchgeführt werden, kostet die Reparierung der Gram-Schmidt Orthogonalisierung insgesamt $O(n^5(\log B)^3)$ binäre Zeit.

Bei der Größenreduktion kann bei der Berechnung der b_i modulo $\sqrt{nB}$ gerechnet werden wegen (3.53). Bei den Werten der M-Matrix haben die Nenner Länge $n \log B$ und die Zähler $O((n + \log B) + (n \log B))$ nach Lemma 3.24. Nach Kaltofen [29] können die benötigten arithmetischen Operationen dann in binärer Zeit $O((n + \log B)n \log B)$ ausgeführt werden. Bei der Größenreduktion einer Spalte werden $O(n^2)$ arithmetische Operationen durchgeführt. Bei $n^2 \log B$ Phasen ergibt sich die Behauptung. ∎

Satz 3.31 *Die asymptotische binäre Laufzeit von Alltauschalgorithmen mit der seriellen Größenreduktionsreihenfolge ist bei der Reduktion von Rucksackgittern gleich der binären Laufzeit des LLL-Algorithmus.*

Beweis: Die Anzahl der arithmetischen Operationen und die Zahlengrößen sind asymptotisch gleich. Es kann die Analyse aus [29] angewendet werden. ∎

Satz 3.32 (Binäre Zeitkomplexität Alltauschalgorithmen) *Serielle Alltauschalgorithmen haben bei genauer Rechnung die binäre Zeitkomplexität* $O(n^6(\log B)^3)$. *Dies gilt sowohl bei der seriellen Reihenfolge als auch bei der parallelen Reihenfolge für die Größenreduktion.*

Beweis: Nach Korollar 3.13 werden $O(n \log B)$ Phasen durchgeführt. Bei einer genauen Rechnung muß die Gram-Schmidt Orthogonalisierung nicht immer berechnet werden, sondern kann mit $O(n^2)$ arithmetischen Operationen repariert werden. Bei der Gram-Schmidt Orthogonalisierung hat man also insgesamt $n^3 \log B$ arithmetische Opertionen. Bei der Größenreduktion werden insgesamt $n^4 \log B$ arithmetische Operationen durchgeführt. Nach den Abschätzungen des letzten Abschnitts haben die entstehenden Zahlen die binäre Länge $O(n \log B)$. ∎

Für praktische Anwendungen wurden Gleitkomma-Algorithmen betrachtet. Dort wird nur die Gitterbasis in genauer ganzzahliger Arithmetik berechnet. Die Werte der M-Matrix und die Werte $\|b_i^*\|^2$ werden in Gleitkommadarstellung verwaltet. Die Zeit für eine arithmetische Operation auf Gleitkommazahlen kann als konstant angenommen werden.

Um die binäre Laufzeit solcher Algorithmen ungefähr vergleichen zu können, wird der Begriff der *heuristischen* binären Laufzeit eingeführt.

Definition 3.5 *Die* heuristische *Laufzeit ergibt sich, wenn man die entstehenden Rundungsfehler vernachlässigt.*

Satz 3.33 *Die binäre heuristische Laufzeit von seriellen Gleitkomma - Alltauschalgorithmen (sowohl bei der seriellen als auch bei der parallelen Reihenfolge für die Größenreduktion) ist* $O(n^5(\log B)^2)$.

Beweis: Die einzige genaue Rechnung des Gleitkomma-Alltauschalgorithmus aus Abschnitt 3.5 ist

$$b_i = b_i - \lceil \mu_{i,j} \rfloor b_j$$

während der Größenreduktion. Sei l die Länge der Zahlen der Vektoren b_i. Dann können die arithmetischen Operationen in Zeit $O(l)$ ausgeführt werden. Nach Korollar 3.13 werden $O(n^4 \log B)$ arithmetische Operationen durchgeführt. Die Laufzeiten des Satzes ergeben sich somit aus den möglichen Zahlengrößen für $\|b_i\|^2$. ∎

Satz 3.34 *Die binäre heuristische Laufzeit des Schnorr - Euchner Algorithmus ist* $O(n^5 \log B + n^4(\log B)^2)$.

Beweis: Es werden $O(n^4 \log B)$ Operationen der Form $b_i = b_i - \lceil \mu_{i,j} \rfloor b_j$ durchgeführt, wobei die maximale Zahlenlänge $O(n + \log B)$ beträgt. ∎

3.8.4 Diskussion

Wie gesehen können bei Alltauschalgorithmen (theoretisch) größere Zahlen entstehen als bei Stufenalgorithmen. Außerdem können bei der parallelen Größenreduktionsreihenfolge größere Zahlen entstehen als bei der seriellen.

Praktische Ergebnisse aus Abschnitt 3.5.5 und [72] zeigen, daß die entstehenden Zahlengrößen bei parallelen Alltauschalgorithmen und Stufenalgorithmen vergleichbar sind. Es findet keine „Zahlenexplosion“ bei dem parallelen Algorithmus statt.

Aus *theoretischer Sicht* sind die Ergebnisse für die binären Laufzeiten nicht ganz befriedigend. Ein gutes theoretisches Ergebnis ist erreichbar, wenn man nur die *schnellste erreichbare, parallele Zeit* betrachtet, ohne die Hardwarekosten zu beachten. Dann können auch die exakten arithmetischen Operationen parallelisiert werden. Die Länge der Zahlen geht in diesem Fall nur noch logarithmisch in die Zeit für eine exakte arithmetische Operation ein, so daß die Größe der Zahlen nur noch eine untergeordnete Rolle spielt (s. auch [50]). Eine parallele Versendung einer exakten Zahl kostet nur konstante Zeit. Entscheidend für die (theoretische) parallele Komplexität ist das in der vorliegenden Arbeit bewiesene Korollar 3.12.

Joux [27, 28] hat ein Verfahren angegeben, wie bei der parallelen Größenreduktion größere Zahlen gegenüber der seriellen Größenreduktion vermieden werden können. Er hat diese Methode auch benutzt, um einen parallelen Algorithmus zu erhalten mit approximativer Arithmetik mit beweisbar vernachlässigbaren Rundungsfehlern ähnlich dem seriellen Algorithmus von Schnorr [56]. Dieses Verfahren soll der Vollständigkeit halber in dem folgenden Abschnitt kurz vorgestellt werden.

3.8.5 Die Methode von Joux zur Vermeidung großer Zahlen bei der parallelen Größenreduktion

Die Idee von Joux ist, größere Zahlen bei der parallelen Größenreduktionsreihenfolge zu vermeiden, so daß eine „Zahlenexplosion“ wie in Beispiel 3.4 nicht auftreten kann. Mit dieser Methode ist es möglich, für die parallele Größenreduktionsreihenfolge dieselben Schranken für die Zahlengrößen herzuleiten wie bei der seriellen Größenreduktionsreihenfolge. Falls für β aber keine bessere Abschätzung als $\beta \leq B^n$ angegeben werden kann, ist auch mit dieser Methode keine bessere Analyse als in Satz 3.29 angegeben möglich.

Für die Methode von Joux muß man sich noch einmal den Reduktionsalgorithmus und die Rolle der Größenreduktion vergegenwärtigen.

Der Reduktionsalgorithmus würde auch korrekt arbeiten, wenn nur die $\mu_{i+1,i}$ für $i = 1, \ldots, n-1$ in der Gram-Schmidt Orthogonalisierung größenreduziert würden. Auch dann würde das Konvergenzkriterium, daß bei einem Tausch von b_i und b_{i+1} das Produkt $\|b_1^*\|^2 \cdot \ldots \cdot \|b_i^*\|^2$ um mindestens $\frac{3}{4}$ fällt, erfüllt (siehe Gleichung (3.11)).

Die Größenreduktion der übrigen $\mu_{i,j}$ erfolgt nur, um eine entsprechende Schranke über die Größen der $\|b_i\|^2$ nach der Größenreduktion zeigen zu können (s. Gleichungen (3.52) und (3.53)).

Diese Tatsache kann dazu benutzt werden, um bei der parallelen Größenreduktion nur die $\mu_{i+1,i}$ gemäß dem normalen Algorithmus größenzureduzieren. Es müssen nur Maßnahmen getroffen werden, daß die $\|b_i\|$ nicht zu groß werden.

Das Problem bei der parallelen Größenreduktionsreihenfolge ist, daß bei der Größenreduktion Spalten der Koeffizientenmatrix M^{T} mit noch nicht größenreduzierten Spalten verarbeitet werden, d.h. mit Spalten, deren Elemente $\mu_{i,j}$ betragsmäßig nicht kleiner gleich $\frac{1}{2}$ sind, und so theoretisch immer größere Zahlen entstehen können (s. Beweis zu Lemma 3.24).

Der Ansatz von Joux besteht darin, nur die Überdiagonalelemente $\mu_{i+1,i}$ „normal“ größenzureduzieren, die übrigen $\mu_{i,j}$ jedoch gegen eine sogenannte „lokale Basis“. Die lokalen Basen werden so verwaltet, daß sie immer größenreduziert sind. Somit ist sichergestellt, daß die Werte der Koeffizientenmatrizen der lokalen Basen klein sind. Obwohl die Basis des Ursprungsgitters dann nicht größenreduziert ist gemäß der bisherigen Definition, lassen sich über die Größenreduktion zu den lokalen Basen für die $\|b_i\|$ dieselben oberen Schranken ableiten wie vorher. Eine globale Größenreduktion der Basis des Ursprungsgitters am Schluß gewährleistet, daß die Basis bei Beendigung des Algorithmus tatsächlich nach der normalen Definition größenreduziert ist und somit insgesamt reduziert nach Definition 3.4.

Es soll nun der Begriff der „lokalen Basen“ genauer erläutert werden. Mit jedem Vektor b_i der Gitterbasis wird das Untergitter L_{i-1}, das durch die Basisvektoren $b_1, \ldots, b_{i-1}$ erzeugt wird, assoziiert. Für dieses Gitter wird eine lokale Basis $b_{i,1}, \ldots, b_{i,i-1}$ mit der entsprechenden Gram-Schmidt Orthogonalisierung $b_{i,1}^*, \ldots, b_{i,i-1}^*$ verwaltet.

Gespeichert und verwaltet werden also die eigentliche Basis $b_1, \ldots, b_n$ des Gitters, ihre Gram-Schmidt Orthogonalisierung $b_1^*, \ldots, b_n^*$ und zu jedem Basisvektor b_i die lokale Basis und ihre Orthogonalisierung. Die Koeffizienten $\mu_{i,j}$ der Gram-Schmidt Orthogonalisierungen werden bei Bedarf mit Hilfe der Gram-Schmidt-Formel berechnet.

Während der Ausführung des Reduktionsalgorithmus soll nun die folgende *Invariante* aufrecht erhalten werden. Für $i = 2, \ldots, n$ gilt, daß die lokale Basis zu zu dem Basisvektor b_i (also die Basis bestehend aus den Vektoren $b_{i,1}, \ldots, b_{i,i-1}$) eine Basis des Untergitters L_{i-1} und größenreduziert ist.

Zu Beginn läßt sich die Invariante sehr leicht herstellen. Es wird die Gram-Schmidt Orthogonalisierung der Ursprungsbasis berechnet und eine normale Größenreduktion wird durchgeführt. Die lokale Basis und die Orthogonalisierung zu L_{i-1} werden mit $b_1, \ldots, b_{i-1}$ bzw. $b_1^*, \ldots, b_{i-1}^*$ initialisiert.

Bei einem Tausch von b_{i-1} und b_i wird nur das Untergitter L_{i-1} verändert. Die anderen Untergitter bleiben unverändert, und somit gilt für diese auch weiterhin die Invariante. Die neue lokale Basis für b_i (also das ehemalige b_{i-1}) erhält man, indem an die lokale Basis des ehemaligen b_{i-1} der neue Vektor b_{i-1} (also das ehemalige b_i) angehängt wird. Für die Orthogonalisierung der neuen lokalen Basis wird ebenfalls die Orthogonalisierung der lokalen Basis des ehemaligen b_{i-1} genommen und zusätzlich $b_{i,i-1}^* \leftarrow b_{i-1}^*$ gesetzt. Die Orthogonalisierung $b_1^*, \ldots b_n^*$ der neuen Gesamtbasis $b_1, \ldots, b_n$ wird dabei wie bisher mit Givens-Rotationen berechnet. b_i wird nun gegen seine lokale Basis größenreduziert, so daß die Invariante erhalten bleibt.

Insgesamt sieht der Algorithmus für einen Tauschschritt also folgendermaßen aus:

1. Es wird berechnet, ob b_{i-1} und b_i getauscht werden müssen.

2. Wenn ja, wird b_i gegen b_{i-1} wie gewohnt größenreduziert, d.h. das Element $\mu_{i,i-1}$ wird folgendermaßen größenreduziert.

$$b_i \leftarrow b_i - \lfloor \mu_{i,i-1} \rceil b_{i-1}$$

3. Die Basisvektoren b_{i-1} und b_i werden getauscht und wie beschrieben wird die neue lokale Basis zu dem neuen Basisvektor b_i berechnet. Es werden also die folgenden Operationen durchgeführt.

   Für $j = 1, \ldots, i-2$ tue:

$$\begin{aligned} b_{i,j} &\leftarrow b_{i-1,j} \\ b_{i,j}^* &\leftarrow b_{i-1,j}^* \end{aligned}$$

   und

$$b_{i,i-1} \leftarrow b_i$$

   Tausche b_i und b_{i-1}.

4. Das ehemalige b_i und jetzige b_{i-1} wurde durch die Größenreduktion aus Schritt 2 verändert. Um eine Aussage über seine Größe zu erhalten, wird es nun gegen seine lokale Basis größenreduziert.

   Für $j = i-2, \ldots, 1$:

$$\begin{aligned} r &\leftarrow \left\lfloor \frac{b_{i-1} b_{i-1,j}^*}{\|b_{i-1,j}^*\|^2} \right\rceil \\ b_{i-1} &\leftarrow b_{i-1} - r b_{i-1,j} \end{aligned}$$

5. Mit Givens Rotationen wird die Orthogonalisierung für die neue Gesamtbasis (also nach dem Tausch von b_{i-1} und b_i) berechnet.

6. $b^*_{i,i-1} \leftarrow b^*_{i-1}$

7. Der Basisvektor b_i wird gegen seine lokale Basis größenreduziert (zur Bewahrung der Invarianten).

Bei einer Parallelisierung entsprechen dabei die Punkte 1-3,6 einer lokalen Kommunikation benachbarter Spalten, Punkte 4 und 7 sind lokale Operationen innerhalb einer Spalte und Punkt 5 ist wie bisher eine globale Operation.

Damit gilt folgender Satz:

Satz 3.35 ([27]) *Es gibt einen Basisreduktionsalgorithmus, bei dem eine Phase effizient mit n^2 Prozessoren parallelisierbar ist, und bei dem asymptotisch keine gößeren Zahlen entstehen, als bei einem seriellen Alltauschalgorithmus.*

3.9 Parallele Implementierungen und praktische Ergebnisse

Nachdem gezeigt wurde, daß theoretisch eine gute Parallelisierung der Basisreduktion erreicht werden kann, sollen nun verschiedene parallele Implementierungen der Basisreduktion und die Ergebnisse praktischer Untersuchungen beschrieben werden.

Während Roch und Villard [50, 72] nachgewiesen haben, daß eine gute parallele Implementierung möglich ist unter der Benutzung exakter Arithmetik, ist die Aufgabe hier schwieriger, da mit dem seriellen Algorithmus von Schnorr und Euchner [57] ein sehr effizienter serieller Algorithmus zur Verfügung steht, und bei der Verwendung von Gleitkomma-Arithmetik die Kommunikationskosten im allgemeinen gegenüber den Rechenzeiten nicht mehr vernachlässigt werden können. Die Kommunikation stellt aber oft einen entscheidenden Geschwindigkeitsverlust bei parallelen Implementierungen dar. Außerdem müssen Maßnahmen zur Fehlerminimierung getroffen werden, die Roch und Villard nicht beachten müssen.

Nach einer kurzen Einleitung, in der die wichtigsten Begriffe eingeführt und prinzipielle Bermerkungen zur parallelen Basisreduktion getroffen werden, werden die verschiedenen Rechnerarchitekturen und parallelen Programmiermodelle vorgestellt. Als Beispiel werden danach Givens-Rotationen betrachtet (s. auch Kapitel 3.5.3 und Algorithmus 3.9). Danach werden die Algorithmen und Ergebnisse für die verschiedenen Implementierungen der Basisreduktion beschrieben. Den Abschluß bildet ein Überblick über die verschiedenen diskutierten Implementierungen.

3.9.1 Einleitung und Definitionen

Das Hauptziel bei der Entwicklung paralleler Algorithmen ist, ein Problem möglichst schnell zu lösen (möglichst schneller als mit einem „normalen“ sequentiellen Programm

auf einem „normalen“ Rechner). Das Maß für den Erfolg ist die *Beschleunigung*[22] [5]:

$$S_p(m) = \frac{T^*(m)}{T_p(m)}$$

Dabei ist m die Eingabegröße, p die Prozessorzahl, T^* die sequentielle Zeit und T_p die parallele Zeit bei der Verwendung der p Prozessoren.

Der Vergleich parallel - seriell ist nur fair, wenn für die sequentielle Zeit $T^*(m)$, die Zeit des „besten“ sequentiellen Algorithmus gewählt wird. Ist diese nicht bekannt, so nimmt man im allgemeinen die Zeit des besten *bekannten* Algorithmus. Man kann aber auch die Zeit des parallelen Algorithmus zu Grunde legen, die dieser auf *einem* Prozessor benötigt (also $T_1(m)$).

Die Auslastung der Prozessoren wird mit Hilfe der Effizienz angegeben [5]:

$$E_p(m) = \frac{T^*(m)}{p \cdot T_p(m)}$$

Ist die Effizienz 1, so werden alle Prozessoren optimal ausgenutzt, und der Aufwand des parallelen Algorithmus ist insgesamt nicht größer als der des Seriellen.

Wenn auch eine Effizienz von 1 kaum erreichbar ist, so sollte sie doch konstant, d.h. unabhängig von der Eingabegröße m und der Prozessorzahl p, sein[23]. In diesem Falle ist der Algorithmus gut skalierbar, d.h. eine Verdoppelung der Prozessorzahl führt zu einer Halbierung der Laufzeit. Auch wenn ein paralleler Algorithmus diese Eigenschaft theoretisch erfüllt, so muß dies in der Praxis noch lange nicht der Fall sein, da mit der Zunahme der Prozessorzahl auch der Verwaltungsaufwand (z.B. Kommunikationskosten) zunimmt. Es lohnt sich daher, diese Eigenschaft bei einer parallelen Implementierung experimentell zu testen.

Eine andere Sichtweise ergibt sich, wenn man die Wachstumsrate des parallelen Algorithmus bei wachsender Eingabegröße betrachtet unter der Annahme, daß immer optimal viele Prozessoren zur Verfügung stehen. Dies ist auch bei sequentiellen Algorithmen die übliche Vorgehensweise. So ist ein Algorithmus mit linearer Wachstumsrate ab einer gewissen Problemgröße Algorithmen mit einer quadratischen Wachstumsrate überlegen. Diese Betrachtungsweise ist bei parallelen Algorithmen zwar problematischer, da ab einer gewissen Problemgröße nicht mehr die optimale Prozessorzahl zur Verfügung steht, ist aber dennoch ein Maß für die Güte eines parallelen Algorithmus und dessen Implementierung. So ist zum Beispiel die Laufzeit für eine Phase der Basisreduktion im sequentiellen Fall kubisch in der Dimension n der Basis und im parallelen Fall nur linear, falls n^2 Prozessoren zur Verfügung stehen (s. Kapitel 3.6).

Bei den praktischen Untersuchungen in diesem Kapitel sollen, falls möglich, beide Aspekte untersucht werden, d.h. die Effizienz einerseits und die Wachstumsrate andererseits.

Prinzipielle Bemerkungen zur Effizienz von parallelen Basisreduktionsalgorithmen

Die Effizienz von parallelen Implementierungen von Algorithmus 3.8 im Vergleich zu seriellen Stufenalgorithmen hängt von zwei Punkten ab.

[22] Der englische Fachbegriff ist „speedup“.

[23] Natürlich nur für die Prozessorzahlen p, die der Algorithmus überhaupt nutzen kann.

- Anzahl der Phasen
- Effizienz einer Phase

Für die Anzahl der Phasen wurde eine obere Schranke hergeleitet, die im Vergleich zu der Anzahl der Phasen von seriellen Stufenalgorithmen optimal ist. Außerdem wurde mit einer seriellen Implementierung gezeigt, daß viele Tausche parallel durchgeführt werden können. Allerdings muß beachtet werden, daß nicht an allen Stellen getauscht werden kann und in einer Einzelphase („Odd/Even“) maximal an jeder zweiten Stelle. Wie oft tatsächlich parallel getauscht werden kann, hängt auch von der Eingabebasis ab.

Wenn man davon ausgeht, daß die Anzahl der Phasen (fast) optimal ist, muß nur noch die Laufzeit für eine Phase des Algorithmus untersucht werden, um eine Aussage über die Effizienz der parallelen Implementierungen zu erhalten. Daher werden an geeigneter Stelle ohne wesentlichen Informationsverlust nur die Laufzeiten für einzelne Phasen der Basisreduktion betrachtet.

Ein Problem bei der parallelen Ausführung einer Phase ist die Zahl der Größenreduktionen. Je weniger Elemente in einer Phase größenreduziert werden müssen, desto ineffizienter ist der parallele Algorithmus. Wenn ein Element nicht größenreduziert werden muß, macht man im sequentiellen Fall nichts, im parallelen Fall muß aber evtl. auf einen Prozessor gewartet werden, der eine Größenreduktion durchführt (s. auch [73]). Die praktischen Ergebnisse des seriellen Falles zeigen aber, daß zumindest bei den unimodular transformierten Gittern viele Größenreduktionen pro Phase durchgeführt werden müssen. Bei den Rucksackgittern bereitet dieses Problem größere Schwierigkeiten.

Zusammenfassend bleibt festzuhalten, daß die Effizienz von parallelen Basisreduktionsalgorithmen prinzipiell durch die folgenden Punkte negativ beeinflußt wird.

- Es kann nicht an jeder Stelle getauscht werden.
- In einer Phase muß nicht jedes Element größenreduziert werden.

3.9.2 Parallele Programmiermodelle

Es soll hier nur ganz kurz auf verschiedene Parallelrechnerarchitekturen und parallele Programmiermodelle eingegangen werden, so weit es zum Verständnis der hier benutzten Implementierungen wichtig ist. Für eine genauere Darstellung sei z.B. auf [2, 7] verwiesen.

Das Wesen eines Parallelrechners besteht darin, über mehrere Berechnungselemente („Prozessoren“) zu verfügen, die parallel Berechnungen durchführen können. Diese greifen entweder auf einen gemeinsamen Speicher zu, oder jeder verfügt über seinen eigenen lokalen Speicher[24]. Besitzen die Prozessoren keinen gemeinsamen Speicher, so können sie über ein Verbindungsnetzwerk Nachrichten austauschen.

Bei sogenannten *MIMD*-Rechnern (Multiple Instruction, Multiple Data) handelt es sich bei den Berechnungselementen um Prozessoren, die vollkommen unabhängig voneinander je ein eigenes Programm ausführen können. Man nennt dies daher auch *asynchrone*

[24] Man spricht dann von „verteiltem Speicher“.

Parallelität. Ein Informationsaustausch findet entweder über den evtl. vorhandenen gemeinsamen Speicher oder über das Austauschen von Nachrichten statt.

Das Modell aus Kapitel 3.3, das den theoretischen Betrachtungen zugrunde gelegt wurde, entspricht also einem MIMD-Rechner mit verteiltem Speicher.

Der Vorteil dieses Modells ist, daß durch die Unabhängigkeit der Prozessoren eine flexible Programmierung möglich ist. Der Nachteil von MIMD-Architekturen besteht jedoch darin, daß nur eine realtiv kleine Anzahl von Prozessoren eingesetzt werden kann. So verfügt der Intel iPSC/860 über maximal 128 Prozessoren und die Intel Paragon über maximal 512 [7].

Dieser Nachteil wird bei *SIMD*-Maschinen (Single Instruction, Multiple Data) dadurch umgangen, daß sehr einfache Prozessoren verwendet werden, die über keinen eigenen Befehlsdekodierer verfügen. Die parallelen Prozessoren werden durch einen Steuerrechner[25] angesprochen, der allen Prozessoren den auszuführenden Befehl mitteilt. Diesen führen dann alle Prozessoren auf ihren eigenen Daten aus. Die einzige Möglichkeit, daß verschiedene Prozessoren verschiedenes tun, besteht darin, daß Prozessoren „abgeschaltet“ werden können. Ein Prozessor führt also den globalen Befehl aus, oder er macht nichts. Dies führt zu einer Sequentialisierung von „if-then-else“-Strukturen. Diese Art der Parallelität nennt man auch *synchrone Parallelität*, *massive Parallelität* oder *Datenparallelität*. Der Vorteil besteht darin, daß sehr viele Prozessoren realisiert werden können (z.B. bis zu 65.536 bei der CM-2 der Firma Thinking Machines [7]) und daß die Kommunikationszeit kleiner als bei den MIMD-Maschinen ist, wird aber durch eine umständliche Programmierung erkauft. Außerdem sind die Prozessoren oft langsamer bei arithmetischen Operationen.

Eine weitere Möglichkeit, parallel zu rechnen, besteht darin, mehrere Workstations zu benutzen, die über einen bestimmten Mechanismus Nachrichten austauschen. Dies entspricht im wesentlichen dem MIMD-Modell mit verteiltem Speicher.

Bei der Programmierung gibt es prinzipiell zwei Möglichkeiten:

- Verwendung einer höheren parallelen Programmiersprache
- Verwendung einer sequentiellen Programmiersprache (C, Fortran ...) und einer Funktionsbibliothek mit Funktionen zum Nachrichtenaustausch (MIMD) bzw. zum Ansteuern des parallelen Prozessorfeldes (SIMD)

Bei der Basisreduktion wurden die folgenden parallelen Plattformen untersucht:

- Workstationnetz (Kapitel 3.9.4)
- MIMD mit verteiltem Speicher unter Verwendung der Programmiersprache „C“ und Nachrichtenaustausch (Kapitel 3.9.5)
- SIMD mit verteiltem Speicher unter Verwendung der datenparallelen Programmiersprache „C*“ (Kapitel 3.9.6)

Bevor aber die parallelen Implementierungen der Basisreduktion beschrieben werden, wird als Beispiel die Implementierung paralleler Givensrotationen betrachtet.

[25] engl. „Host“ oder „front end“

3.9.3 Ein Beispiel: Parallele Givens-Rotationen

Als Beispiel soll nun die Implementierung paralleler Givens-Rotationen beschrieben werden. Die Gründe hierfür sind die folgenden:

- Parallele Givens-Rotationen werden auch bei der parallelen Basisreduktion benötigt.
- Das Schema zur Parallelisierung ist sehr allgemein und kann auch bei anderen Algorithmen verwendet werden (z.B. Gauß-Reduktion) [60].
- Auch der zweite Teil einer Phase der Basisreduktion, die Größenreduktion, kann gemäß diesem Schema parallelisiert werden. Bei der Größenreduktion sind zwar etwas einfachere Varianten möglich (vgl. Kapitel 3.9.4 und 3.9.5), aber die Grundprinzipien sind dieselben. Daher wird in den dortigen Kapiteln nicht mehr so ausführlich auf die Algorithmen eingegangen.
- Givens-Rotationen sind einfach zu implementieren und zu testen.

Parallele Zeilenverknüpfungen

Es sei eine $n \times n$-Matrix zu triangularisieren[26]. Die Grundstruktur des Algorithmus ist wie in Algorithmus 3.10 beschrieben. Wie in den Kapiteln 3.6 und 3.7 angegeben, können verschiedene Zeilenverknüpfungen parallel ausgeführt werden. Dafür gibt es viele verschiedene Möglichkeiten (s. auch [5, 14, 25, 60, 73]). Während in [60] ein „zyklisches Schema“ beschrieben wird, d.h. ein Algorithmus, bei dem die Zeilen zyklisch über die Prozessoren ausgetauscht werden, ist unten in Algorithmus 3.13 eine Möglichkeit angegeben, bei der die Zeilen der Matrix „von oben nach unten durch ein lineares Prozessorfeld fallen“. Eine genauere Beschreibung des Algorithmus und eine exaktere Analyse finden sich in der Diplomarbeit [25].

Es seien p Prozessoren in einem linearen Prozessorfeld gegeben, wobei p die Matrixdimension n teile, und die Zeilen der Matrix gleichmäßig auf die Prozessoren verteilt seien. Der MIMD-Algorithmus könnte dann wie in Algorithmus 3.13 dargestellt aussehen. Dabei bedeuten:

nr	Prozessornummer im linearen Prozessorfeld
p	Anzahl der Prozessoren im linearen Feld
erste	Nummer der ersten Zeile, die ein Prozessor verwaltet
letzte	Nummer der letzten Zeile, die ein Prozessor verwaltet
anzahl	Anzahl der Zeilen, die ein Prozessor verwaltet

Diese Werte können einmal zu Programmstart von den Prozessoren berechnet werden.

Algorithmus 3.13

for $i = 1 \ldots erste - 1$ do
    Empfange von Prozessor $nr - 1$ eine Reihe.
    Dies ist Reihe i der Gesamtmatrix.

[26] Die hier angegebenen Verfahren sind leicht auf $m \times n$-Matrizen erweiterbar.

```
        for j = erste ... letzte do
            Verknüpfe Zeilen i und j
        od

        if (nr < p)
            Sende (veränderte) Reihe i an Prozessor nr + 1
        else
            Speichere Reihe i als Reihe i der Endmatrix
        fi
    od

    for i = erste ... letzte − 1 do
        for j = i + 1 ... letzte do
            Verknüpfe Zeilen i und j
        od

        if (nr < p)
            Sende (veränderte) Reihe i an Prozessor nr + 1
        else
            Speichere Reihe i als Reihe i der Endmatrix
        fi
    od

    if (nr < p)
        Sende Reihe letzte an Prozessor nr + 1
    else
        Speichere Reihe letzte als letzte Reihe der Endmatrix
    fi
```

Eine einfache Analyse zeigt, daß Algorithmus 3.13 $n + \frac{n^2}{p}$ parallele Zeilenverknüpfungen durchführt (s. auch [25]). Alle n/p Schritte wird eine Zeile versendet, also finden insgesamt $\leq p + n$ Zeilenversendungen statt. Für eine theoretische Analyse können diese somit zu einem parallelen Berechnungsschritt hinzugezogen werden.

Für $p = n$ finden $2n$ parallele Schritte statt. Nach [14] sind $2n + o(n)$ Schritte optimal, wobei $f(x) = o(x)$ gilt, wenn $f(x)/x \to 0$ für $x \to \infty$.

Sequentiell werden $n(n-1)/2$ Zeilenverknüpfungen durchgeführt, womit sich folgende Effizienz ergibt:

$$\frac{n(n-1)}{2p(n + \frac{n^2}{p})} \approx \frac{n}{2(p+n)} \to \frac{1}{2} \qquad \text{für} \quad n \to \infty$$

Dies ist nicht optimal, aber gut, wenn man die Einfachheit des Algorithmus betrachtet. Ohne Beachtung der Kommunikation kann eine asymptotische[27] Effizienz von 1 erreicht werden für $p = \lceil n/(2+\sqrt{2}) \rceil$ Prozessoren (PRAM-Modell) [14].

Bemerkungen:

[27] d.h. für $n \to \infty$

- Es gibt viele andere Schemata für die verschiedenen parallelen Modelle (SIMD, MIMD, verteilter bzw. gemeinsamer Speicher) und für verschiedene Anwendungen (quadratische/rechteckige Matrizen), s. z.B. [37, 54, 14, 41]. So ist auch für MIMD-Rechner eine bessere Effizienz als die oben angegebene für $p \leq n/2$ Prozessoren erreichbar, z.B. 1/2 für ein zyklisches Schema nach [60].

  Diese werden aber aus folgenden Gründen hier nicht betrachtet:

  - Einige bessere Schemata sind auf Givens-Rotationen zugeschnitten. Algorithmus 3.13 ist allgemein anwendbar, z.B. auch für die Größenreduktion oder die Gauß-Elimination.
  - Algorithmus 3.13 hat einen geringen und einfachen Kommunikationsaufwand.
  - Da bei der Basisreduktion nur in der Größenreduktionsphase exakt gerechnet wird, ist diese dominant bei der Laufzeit. Für die Größenreduktion sind effizientere Varianten leicht möglich (s. Kapitel 3.9.4 und 3.9.5).
  - Für manche Parallelrechner existieren effiziente Givens-Rotationen in Bibliotheken.

- Eine Verbesserung ist auch durch den Gebrauch wurzelfreier Verfahren möglich [23, 41, 37].

- Ein weiterer Effizienzverlust entsteht dadurch, daß im seriellen Fall die zu verknüpfenden Zeilen immer kürzer werden. Im parallelen Fall bestimmt aber der Prozessor, der die längsten Zeilen zu verknüpfen hat, die Laufzeit. In [37], wo ein Algorithmus für einen MIMD-Rechner mit gemeinsamen Speicher für $n/2$ Prozessoren beschrieben wird, ist dies schon eingerechnet. Dort wird eine Effizienz von 4/9 erreicht.

Parallelisierung mit n^2 Prozessoren

Da nur SIMD-Maschinen genug Prozessoren für einen Algorithmus zur Verfügung stellen, der n^2 Prozessoren nutzen kann, wurde der Algorithmus in der SIMD-Sprache C* geschrieben, die für die „Connection Machines“ der Firma „Thinking Machines“ entwickelt wurde [68, 67].

Beschreibung des Algorithmus Es wird wieder die Triangularisierung einer $n \times n$-Matrix betrachtet gemäß eines Algorithmus der Form von Algorithmus 3.10. Zur Parallelisierung auf einem $n \times n$-Gitter werden die in Kapitel 3.6 beschriebenen Ideen verwendet (das hier verwendete Schema ist aus [21], s. auch [5, 7, 34]), d.h. verschiedene Zeilenverknüpfungen werden gleichzeitig durchgeführt und mit Hilfe von Pipelining-Methoden parallelisiert.

Es wird angenommen, daß die zu triangulisierende Matrix „von oben in das Prozessorfeld fällt“ (s. auch Abbildung 3.2), wobei jede Spalte gegenüber ihrer Vorgängerin um einen Schritt verschoben ist. In einem realen Programm wird dies dadurch erreicht, daß in einem Vorbereitungsschritt die Daten zyklisch über das Feld verteilt werden (nach einer Idee aus [7] für einen Algorithmus zur Matrixmultiplikation).[28]

[28] Praktische Tests zeigen, daß die Zeit für die Vorbereitungsphase vernachlässigbar ist.

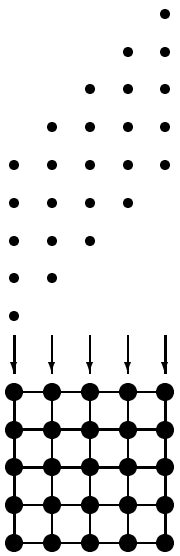

Abbildung 3.2: Dateneingabe bei SIMD-Algorithmus für Givens-Rotationen (vgl. [21])

In dem Programm kann davon ausgegangen werden, daß tatsächlich n^2 Prozessoren zur Verfügung stehen. Ist dies physikalisch nicht der Fall, übernimmt der Compiler die Aufteilung der Berechnungen.

Ein *SIMD-Algorithmus* hat dann folgende Struktur:

Algorithmus 3.14

Alle Prozessoren befinden sich im Zustand „wartend“
for $i = 1$ to $3n - 1$ do
Alle Prozessoren empfangen von oben ein Element

Prozessoren im Zustand „rechnend“:
{
Diagonalprozessoren:
{
Berechnung der Rotationsparameter mit Hilfe des
eigenen Elementes und des empfangenen Elementes
}

Verknuepfen des eigenen Elements mit dem empfangenen

}

Prozessoren im Zustand „wartend“
{
Falls das empfangene Element die Zeilennummer besitzt, die
auch der Prozessor besitzt, so speichert der Prozessor
das Element und geht in den Zustand „rechnend“

}

Alle Prozessoren empfangen von links (neue) Rotationsparameter
od

Effizienz Da die Verteilungen der Berechnung auf die Prozessoren automatisch stattfindet, wird näherungsweise angenommen, daß sich die Anzahl der parallelen Schritte folgendermaßen ergibt:

$$\text{Anzahl der Schritte im parallelen C*-Programm} \cdot \frac{n^2}{p}$$

Es werden $n(n-1)/2$ Elemente eliminiert, wobei jeweils zwei Zeilen verknüpft werden. Da die Zeilen immer weniger Elemente ungleich Null besitzen, werden in einem sequentiellen Programm insgesamt ungefähr $n^3/3$ Elementverknüpfungen durchgeführt. Die $n(n-1)/2$ Berechnungen der Verknüpfungsparameter werden bei der folgenden Analyse vernachlässigt.

Im parallelen Programm finden, wie oben gesehen, $3n$ Schleifendurchläufe statt. Nach [6], wo ein anderes paralleles Schema verwendet wird, ist dies (bis auf eine kleine Konstante) optimal.

Die Effizienz ergibt sich also zu[29]

$$\frac{n^3}{3 \cdot p \cdot 3n \cdot \frac{n^2}{p}} = \frac{1}{9}$$

Eine weitere Effizienzverschlechterung ergibt sich daraus, daß bei dem SIMD-Modell alle Prozessoren die gleiche Operation ausführen oder nichts machen. Daher benötigt jeder Schleifendurchlauf soviel Zeit, als ob immer die Rotationsparameter berechnet werden müßten (mit sämtlichen Fallunterscheidungen!). Während im sequentiellen Fall die Verknüpfung zweier Elemente 4 Multiplikationen benötigt, werden in einem Schleifendurchlauf des parallelen Programms 2 Multiplikationen, 2 Divisionen und eine Wurzelberechnung zur Parameterberechnung und 4 Multiplikationen für die Verknüpfung benötigt auf dem Datentyp „double“.[30]

Insgesamt ergibt sich eine Effizienz von $\frac{1}{9}e$, mit

$$e = \frac{4t_{sm}}{8t_{pm} + t_{pw}}$$

wobei t_{sm} die Zeit für eine Multiplikation des seriellen Programms, t_{pm} die Zeit für eine Multiplikation / Division und t_{pw} für eine Wurzelberechnung auf dem parallelen Prozessorfeld ist.

Für $t_{sm} = t_{pm} = t_{pw}$ ergäbe sich also eine Effizienz von $1/20,25$.

Folgende Verbesserungen sind denkbar:

[29] Eigentlich benötigt der Algorithmus kein rechteckiges Prozessorfeld, sondern nur ein dreieckiges. Dies ist in C* aber nicht möglich. Die theoretische Effizienz wäre dann 2/9.

[30] Additionen, Subtraktionen und Zuweisungen werden vernachlässigt.

n	$8k$	$16k$	seriell
128	28,8	14,6	
	6,1	5,3	3,2
256	227,8	114,5	
	17,1	12,9	33,3
512	1813,4	906,9	
	80,1	48,1	289,3

Tabelle 3.7: Parallele Givensrotationen auf der CM-2 – Effizienz

- Bei wurzelfreien Verfahren (z.B. [41]) ist der Aufwand, den ein Diagonalprozessor zur Parameterberechnung aufbringen muß, geringer. Damit ist der Wert von e besser.
- Wenn statt des SIMD-Modells die Verwendung des MIMD-Modells möglich ist, kann für e der Wert $\frac{4t_{sm}}{6t_{pm}+t_{pw}}$ erreicht werden (vgl. mit [6]).
- Die Anzahl der Prozessoren kann verringert werden. Übernimmt z.B. jeder Prozessor (außer den Diagonalprozessoren) zwei Spalten der Matrix, so reichen $n^2/4$ Prozessoren aus. Für e und für die Effizienz für den Fall $t_{sm} = t_{pm} = t_{pw}$ ergeben sich denn die folgenden Werte
  - SIMD-Modell: $e = \frac{4t_{sm}}{4t_{pm}+t_{pw}+8t_{pm}}$, Effizienz $\approx \frac{1}{7,3}$
  - MIMD-Modell: $e = \frac{4t_{sm}}{\max\{6t_{pm}+t_{pw},8t_{pm}\}}$, Effizienz $\approx \frac{1}{4,5}$
- In [38] ist ein Verfahren angegeben, das $n^2/2$ Prozessoren in einer rechteckigen Anordnung nutzen kann. Mit diesem Schema ist also auch die Nutzung von nur $n^2/2$ Prozessoren in C* möglich. Allerdings benötigt das Verfahren $9n/2$ statt $3n$ Schritte. Ähnliches trifft auch auf des Schema aus [60] zu, das nur $n^2/4$ Prozessoren in einer rechteckigen Anordnung nutzt. Das Verfahren benötigt $6n$ Schritte mit $e = \frac{4t_{sm}}{4t_{pm}+t_{pw}+8t_{pm}}$. Allerdings ist dieses Schema allgemeiner, da immer komplette Zeilen und komplette Spalten verknüpft werden können. Außerdem können drei unabhängige Probleme „verzahnt“ gelöst werden.

Praktische Ergebnisse

Algorithmus 3.14 wurde in C* implementiert und auf der SIMD-Maschine CM-2 [68] und der CM-5 [71] der Firma Thinking Machines getestet.[31] Die CM-5 ist keine SIMD-Maschine, unterstützt aber durch entsprechende Hardware den SIMD-Programmierstil. Daher existiert auch für die CM-5 ein C*-Compiler (näheres dazu in [25, 71]).

Ergebnisse auf der CM-2 In Tabelle 3.7 sind die Ergebnisse für verschiedene Prozessorzahlen dargestellt. Dabei hat man nur die Wahl zwischen $8k = 8192$ und $16k = 16384$ Prozessoren. Die Laufzeiten sind in CPU-Sekunden angegeben, wobei die erste Zahl die Zeit auf dem parallelen Prozessorfeld (Rechen- und Kommunikationszeit) und die zweite

[31] Die Benutzung der Rechner wurde durch die Gesellschaft für Mathematik und Datenverarbeitung (GMD) ermöglicht.

n	Zeit
16	1,80
	0,65
32	3,62
	1,30
64	7,27
	3,05
128	14,58
	5,33

Tabelle 3.8: Parallele Givensrotationen auf der CM-2 – Wachtumsrate

n	16	32	64	seriell
64	0,51	0,33	0,26	
	0,75	0,53	0,44	0,12
128	2,93	1,64	1,02	
	3,33	1,97	1,41	0,95
256	20,91	10,86	5,89	
	22,04	11,74	6,65	10,49
512	160,92	81,35	41,92	
	163,46	83,84	43,99	97,64

Tabelle 3.9: Parallele Givensrotationen auf der CM-5 – Effizienz

auf dem Steuerrechner darstellt. Interessant ist, daß die Kommunikationszeiten gegenüber den Rechenzeiten vernachlässigbar sind. Die serielle Zeit wurde mit einem sequentiellen C-Programm auf dem Steuerrechner gemessen.

Die wichtigsten Ergebnisse lassen sich folgendermaßen zusammenfassen.

- Eine Verdoppelung der Prozessoren führt im wesentlichen zu einer Halbierung der Laufzeit. Gemäß den Ausführungen in Kapitel 3.9.1 hat man also eine konstante Effizienz.
- Mit einem einfachen Testprogramm wurde für e der Wert 1/6900 ermittelt. Dies liegt daran, daß es sich bei den Prozessoren der CM-2 um sehr einfache 1-Bit-Prozessoren handelt. Die theoretische Effizienz ist damit 1/62100. Dies stimmt im wesentlichen mit der praktisch gemessenen Effizienz von ungefähr 1/56500 überein.

In Tabelle 3.8 sind die Ergebnisse für den Fall dargestellt, daß immer die maximal mögliche Prozessorzahl von n^2 zur Verfügung steht. Man sieht, daß tatsächlich die theoretisch erwartete lineare Wachstumsrate vorliegt.

Ergebnisse auf der CM-5 In Tabelle 3.9 sind die Ergebnisse für die CM-5 angegeben. Die erste Zahl gibt wieder die Zeit auf den parallelen Knoten (Kommunikations- und Rechenzeit) und die zweite auf dem Steuerrechner (hier „partition manager“) an. Bei

kleinen Dimensionen und großen Prozessorzahlen ist die Kommunikationszeit etwa in der Größenordnung der Rechenzeit. Bei $p = 64$ und $n = 64$ sind die Zeiten beispielsweise ungefähr gleich.

- Eine Verdoppelung der Prozessoren führt auch hier ab der Dimension 128 fast zu einer Halbierung der Laufzeit. Bei kleineren Dimension macht sich wahrscheinlich der Einfluß der Kommunikationszeit bemerkbar.

- Praktische Messungen zeigen, daß auf der CM-5 die Zeit für eine arithmetische Operation auf dem parallelen Prozessorfeld von dem Verhältnis der Anzahl der virtuellen Prozessoren zu der Anzahl der physikalischen Prozessoren abhängt. Je größer dieses Verhältnis, desto geringer die Zeit für eine einzelne Operation.

  Für die hier vorliegenden Größenordnungen ergeben Messungen für e Werte zwischen 1 und 1/3.

  Die gemessene Effizienz beträgt für größere Dimensionen ungefähr 1/32.

3.9.4 Implementierung auf einem Workstationnetz mit LiPS

LiPS ist ein System, das verteilte Anwendungen auf einem Workstationnetz mittels Nachrichtenaustausch ermöglicht [52, 61, 51]. Es wurde am Lehrstuhl Prof. Buchmann der Universität des Saarlandes entwickelt. Die Idee ist, die (meistens) ungenutzte Rechenkapazität der Workstations (z.B. nachts) für rechenintensive Aufgaben zu nutzen. Um die Benutzer der Rechner nicht zu stark zu beeinträchtigen, können Bedingungen angegeben werden, wann eine LiPS-Anwendung auf einem Rechner laufen soll. So ist es z.B. möglich, Wochentage und Tageszeiten festzulegen, an denen die Anwendung nicht laufen darf, oder festzulegen, daß die Anwendung ab einer gewissen Benutzerzahl auf dem Rechner nicht aktiv sein darf.

Bei der Implementierung eines (synchronen) parallelen Basisreduktionsalgorithmus auf einem (asynchronen) Workstationnetz ergeben sich folgende Probleme:

- Im Netz können sich verschieden schnelle Rechner befinden. Der langsamste bestimmt die Geschwindigkeit. Bei einem Parallelrechner arbeiten alle Prozessoren i.a. gleich schnell.

- Unterschiedliche Rechenzeiten auf den Rechnern können auch durch stark unterschiedliche Auslastungen der Rechner entstehen.

- Die Kommunikation findet nicht über direkte Verbindungen zwischen zwei Rechnern statt, sondern über das Netz. Tauschen je zwei Rechner Nachrichten aus, ist das gesamte Netz belastet, während dies bei einem Parallelrechner, bei dem eine direkte Verbindung zwischen den kommunizierenden Prozessoren besteht, nicht der Fall ist.

- Die Kommunikationszeiten sind abhängig von der allgemeinen Netzbelastung.

- Die Kommunikationszeiten sind höher als bei einem Parallelrechner.

Die grundsätzlich Idee ist also, die Kommunikation zu minimieren und jeden Rechner so lange unabhängig rechnen zu lassen, wie es geht. Die Hauptzeit einer Phase wird bei der Größenreduktion verbraucht, da nur hier exakt gerechnet wird, während die Orthogonalisierung schnell mit Gleitkommazahlen berechnet wird. Daher ist folgender Ansatz zur Implementierung auf einem Workstation-Netz sinnvoll: Jeder beteiligte Rechner verwaltet die gesamte Basismatrix für sich und berechnet die Orthogonalisierung. Dann größenreduziert er mit Hilfe der gespeicherten Matrix eine vorher festgelegte Menge von Spalten. Dabei wird ausgenutzt, daß die Spalten unabhängig voneinander größenreduziert werden können. Dann tauschen die Rechner die größenreduzierten Spalten aus, so daß danach jeder Rechner wieder die korrekte neue Basismatrix besitzt. Jeder Rechner kann nun für sich die Tausche ausführen und von vorne beginnen.

Ohne Beachtung der Kommunikationskosten ist somit bei geschickter Wahl der zu reduzierenden Spalten pro Rechner *die Effizienz für die Größenreduktionsphase* 1 für $p \leq n/2$ Rechner.

Für den Austausch der größenreduzierten Spalten gibt es verschiedene Möglichkeiten, z.B. über einen Nachrichtenaustausch mit Hilfe einer Ringstruktur oder über Dateien. Werden die Spalten ringförmig ausgetauscht, so läßt sich theoretisch auch unter Beachtung der Kommunikationskosten eine *konstante Effizienz* für die Größenreduktionsphase erreichen.

Dieser Ansatz wurde in einer Diplomarbeit von Andreas Kratz untersucht [32] unter der Verwendung der Programmiersprache „C“ und der ganzzahligen Arithmetik „thalesII“ [45]. Es wurden verschiedene Algorithmen zum Spaltenaustausch implementiert:

- Jeder Rechner sendet seine neuberechneten Spalten zu allen anderen Rechnern.
- Der Spaltenaustausch findet über eine Ringstruktur statt.
- Der Spaltenaustausch findet über Dateien statt, d.h. jeder Rechner schreibt „seine“ Spalten in eine Datei und liest die Dateien der anderen Rechner. Nur die Synchronisation der Rechner wird mit LiPS durchgeführt.

Dabei wurde insbesondere auf einen modularen Aufbau geachtet, so daß die verschiedenen Kommunikationsfunktionen leicht ausgewechselt oder in anderen Programmen verwendet werden können. Außerdem ist es leicht möglich, die einzelnen Versende-Operationen durch andere zu ersetzen, oder ein anderes Kommunikationssystem (z.B. PVM oder MPI) zugrundezulegen.

Die erzielten Ergebnisse lassen sich folgendermaßen zusammenfassen:

- Falls eine Menge ungefähr gleich schneller Rechner eingesetzt wird (auch unter der Beachtung der Auslastung), ist die Größenreduktion tatsächlich gut parallelisierbar.
- Die Parallelisierung bringt einen Gewinn, wenn das Verhältnis von Rechenzeit zu Kommunikationszeit hoch genug ist. Bei Gittern, die bei der Lösung von Rucksackproblemen eingesetzt werden (s. Abschnitt 3.5.5 und Anhang C), ist das beispielsweise nicht der Fall.

Bei der Erreichung schneller Kommunikationszeiten gibt es zwei Probleme:

- Die zugrundeliegende Netzstruktur ist physikalisch kein Ring. Bei der Spaltenaustauschphase beträgt also die „Netzbelastung“ $O(n^3)$, was der Laufzeit des seriellen Algorithmus entspricht. Außerdem ist das Kommunikationsnetz auch durch andere Anwendungen belastet.

- Die Kommunikation mit LiPS ist relativ langsam. Dies hat folgende Gründe:

  - Für jede Kommunikation wird eine temporäre Verbindung aufgebaut.
  - Die Nachrichtengröße ist auf 1024 Byte beschränkt. Daher können nicht komplette Spalten, sondern nur einzelne Elemente übertragen werden.
  - Es können keine binären Daten übertragen werden. Daher muß eine lange Zahl für die Versendung zunächst in (beispielsweise) Dezimaldarstellung umgewandelt werden und nach dem Empfangen wieder zurück.

  Die letzten beiden Punkte können umgangen werden, indem der Spaltenaustausch über (binäre) Dateien durchgeführt wird. Jeder Rechner schreibt seine neu berechneten Spalten in eine Datei und liest die der anderen Rechner aus deren Dateien. LiPS ist dann nur noch für die Synchronisation zuständig. Dieser Ansatz bringt für unimodular transformierte Gitter etwa ab der Dimension 80 einen Gewinn.

Dabei ist allerdings folgendes zu bemerken:

- FDDI-Glasfasernetze sind sehr schnell und stellen physikalisch einen Ring dar. Falls das verwendete Kommunikationssystem diese physikalische Struktur ausnutzen kann, ist die Kommunikation effizient über den Ring durchführbar.

- LiPS wurde ursprünglich für verteilte Algorithmen entworfen, die wenig Kommunikationsaufwand aufweisen. Daher wurde mehr Aufwand auf die komfortable Verwaltung gelegt, wie z.B. die Möglichkeit, Zeiten für die Anwendungsprogramme festzulegen. Für die Basisreduktion wäre vielleicht ein Kommunikationssystem mit schnellerer Kommunikation sinnvoll, z.B. PVM oder MPI.

- Einige Nachteile von LiPS werden in neuen Version behoben sein bzw. sind bereits verbessert. So wird z.B. der Verbindungsaufbau überarbeitet. Außerdem soll es möglich sein, größere Nachrichten in binärem Format zu versenden.

Exemplarisch sind die Ergebnisse für $n = 80$ und $n = 100$ für Rucksackgitter und für unimodular transformierte Matrizen bei der Verwendung von vier Rechnern (2 Sparcstations 10 und 2 Sparcstations 2) angegeben (Tabellen 3.10, 3.11, 3.12 und 3.13). Die Kommunikation wurde über Dateien durchgeführt. Eine Kommunikation über LiPS ist für die Basisreduktion zu langsam. Bei LiPS übernimmt immer ein Rechner die Koordination der Anwendung. Dieser Rechner hat also besonders viel zur Synchronisation zu kommunizieren und ist in den Tabellen mit einem „*“ markiert. Die stark unterschiedlichen LiPS-Kommunikationszeiten bei der Synchronisation sind dadurch erklärbar, daß schnellere Rechner auf langsamere warten müssen. Gemessen wurde jeweils die Zeit für die ersten zehn Phasen. Die Zeiten sind in CPU-Sekunden angegeben. Weitere Meßergebnisse sind in [32] zu finden.

Falls unterschiedlich schnelle Rechner eingesetzt werden, kann eine Lastbalancierung im folgenden Sinne durchgeführt werden [42]: Es gibt einen Rechner (am besten der schnellste

$n = 80$	s10a (*)	s2m	s2c	s10b	Seriell (s10a)
Gram-Schmidt:	0,33	0,38	0,33	0,35	0,32
Größenreduktion:	0,17	0,08	0,12	0,07	0,43
Datei-Kommunikation:	8,27	17,58	14,92	8,23	–
LiPS-Kommunikation:	16,42	0,13	4,52	4,93	–
Gesamt:	25,18	18,18	19,88	13,65	0,75

Tabelle 3.10: LiPS-Algorithmus für „Rucksackgitter“ der Dimension 80

$n = 100$	s10a (*)	s2m	s2c	s10b	Seriell (s10a)
Gram-Schmidt:	0,52	0,53	0,60	0,43	0,47
Größenreduktion:	0,20	0,23	0,13	0,18	0,70
Datei-Kommunikation:	13,15	27,23	22,70	12,68	–
LiPS-Kommunikation:	21,07	0,12	1,87	9,93	–
Gesamt:	34,93	28,12	25,30	23,23	1,18

Tabelle 3.11: LiPS-Algorithmus für „Rucksackgitter“ der Dimension 100

$n = 80$	s10a (*)	s2m	s2c	s10b	Seriell (s10a)
Gram-Schmidt:	4,30	5,25	4,97	4,25	4,30
Größenreduktion:	11,78	9,42	9,65	11,82	45,37
Datei-Kommunikation:	10,82	20,02	18,07	10,18	–
LiPS-Kommunikation:	22,73	0,00	2,02	5,58	–
Gesamt:	49,63	34,68	34,70	31,83	49,67

Tabelle 3.12: LiPS-Algorithmus für unimodular transformierte Gitter der Dimension 80

$n = 100$	s10a (*)	s2m	s2c	s10b	Seriell (s10a)
Gram-Schmidt:	7,82	9,08	9,10	7,85	7,80
Größenreduktion:	28,62	23,87	23,38	28,77	112,96
Datei-Kommunikation:	17,32	31,62	28,20	15,57	–
LiPS-Kommunikation:	44,45	0,08	2,72	6,30	–
Gesamt:	98,20	64,65	63,40	58,48	120,76

Tabelle 3.13: LiPS-Algorithmus für unimodular transformierte Gitter der Dimension 100

im Netz), der die Indizes der zu größenreduzierenden Spalten verwaltet. Sobald irgendein Rechner im Netz nichts mehr zu tun hat, fordert er den Index der nächsten Spalte an und größenreduziert sie.

3.9.5 Implementierungen auf MIMD-Rechnern

Ein Algorithmus mit paralleler Größenreduktion

Ein erster Ansatz zur Implementierung auf einem MIMD-Rechner ist, die Idee des vorherigen Abschnitts anzuwenden, d.h. nur die Größenreduktion parallel auszuführen. Jeder Prozessor größenreduziert eine vorher festgelegte Menge von Spalten. Danach werden die neu berechneten Spalten über eine Ringstruktur ausgetauscht, so daß nach dem Austausch jeder Prozessor die gesamte neue Matrix kennt.

Die Vorteile dieses Ansatzes sind:

- Der Algorithmus ist einfach zu implementieren.
- Die Kommunikationsstruktur ist einfach. Es findet lediglich ein ringförmiger Datenaustausch nach der Größenreduktionsphase statt. Damit ist das Programm auch leicht portierbar.
- Das Verfahren zur parallelen Größenreduktion hat eine sehr gute Effizienz.

Jedoch weist dieser Algorithmus bei der Verwendung eines Parallelrechners auch Nachteile auf:

- Jeder Prozessor verwaltet die gesamte Matrix. Dadurch entsteht ein hoher Speicheraufwand. Das Ziel bei der Verwendung eines Parallelrechners ist es jedoch auch oft, den Speicherplatz, den jeder Prozessor benötigt, zu verringern.
- Der sequentielle Anteil, den jeder Prozessor gleichermaßen durchführen muß, ist (zu) hoch. Dieser sequentielle Anteil begrenzt die zu erreichende Beschleunigung. Die Zeit für eine Phase kann bestenfalls auf die Zeit einer sequentiellen Orthogonalisierung gedrückt werden.

Dieser parallele Ansatz wurde auf dem Parallelrechner CM-5 der GMD implementiert. Dabei wurde die Programmiersprache „C“, die ganzzahlige Arithmetik „thalesII“ [45] und die Standardkommunikationsbibliothek „CMMD“ der CM-5 [69] benutzt. Außerdem wurde der Algorithmus auf die Intel Paragon der Universität Stuttgart portiert. Dort wurde die „nx“-Kommunikationsbibliothek verwendet [26].

In den Tabellen 3.14, 3.15 und 3.16 sind die Ergebnisse für unimodular transformierte Gitter und für Rucksackgitter[32] auf der CM-5 dargestellt. Die entsprechenden Ergebnisse für die Paragon finden sich in den Tabellen 3.17 und 3.18.[33] Während bei der CM-5 der GMD nur 16, 32 oder 64 Prozessoren verwendet werden können, kann bei der Paragon eine beliebige Prozessorzahl der 72 vorhandenen Prozessoren belegt werden. Außerdem besitzt

[32] vgl. Anhang C

[33] Aus Platzgründen und wegen mangelnder zusätzlicher Information sind für die Paragon weniger Ergebnisse dargestellt.

n	Prozessoren:	1	16	32	64
32	Gesamt:	0,36	0,11	0,11	–
	Orthogonalisierung:	0,06	0,06	0,06	–
	Größenreduktion:	0,29	0,04	0,02	–
	Kommunikation:	–	0,02	0,04	–
64	Gesamt:	3,46	0,68	0,58	0,58
	Orthogonalisierung:	0,34	0,32	0,32	0,32
	Größenreduktion:	3,12	0,27	0,12	0,14
	Kommunikation:	–	0,09	0,15	0,12
100	Gesamt:	17,84	2,75	2,19	1,85
	Orthogonalisierung:	1,00	1,00	1,00	1,00
	Größenreduktion:	16,84	1,36	0,80	0,49
	Kommunikation:	–	0,39	0,40	0,37
140	Gesamt:	72,43	8,70	6,45	5,34
	Orthogonalisierung:	2,62	2,63	2,61	2,62
	Größenreduktion:	69,81	5,22	3,10	1,95
	Kommunikation:	–	0,85	0,73	0,86

Tabelle 3.14: Eine Phase mit paralleler Größenreduktion, CM-5 – unimodular transformierte Gitter

n	Prozessoren:	1	16	32	64
32	Gesamt:	53,3	27,8	28,7	–
	Orthogonalisierung:	19,1	17,8	18,0	–
	Größenreduktion:	34,1	3,0	1,9	–
	Kommunikation:	–	7,0	8,8	–
64	Gesamt:	329,3	140,1	131,8	139,4
	Orthogonalisierung:	95,3	95,3	95,2	92,1
	Größenreduktion:	233,9	17,6	10,3	6,2
	Kommunikation:	–	27,1	26,2	40,9
128	Gesamt:	1 053,9	375,6	347,3	326,6
	Orthogonalisierung:	270,8	257,3	256,2	230,8
	Größenreduktion:	782,8	53,1	28,4	15,8
	Kommunikation:	–	65,1	62,5	79,8
256	Gesamt:	3 579,4	1 119,4	1 013,3	1 009,3
	Orthogonalisierung:	724,5	717,2	714,7	722,3
	Größenreduktion:	2 854,6	193,4	103,6	56,3
	Kommunikation:	–	208,4	194,7	230,3

Tabelle 3.15: Parallele Größenreduktion, CM-5 – „Rucksack-Gitter“ mit Zahlen mit 340 Binärstellen

n	Prozessoren:	1	16	32	64
32	Gesamt:	84,4	41,3	41,1	–
	Orthogonalisierung:	26,6	26,3	25,0	–
	Größenreduktion:	57,7	5,0	3,1	–
	Kommunikation:	–	10,0	12,9	–
64	Gesamt:	666,4	255,0	230,1	253,3
	Orthogonalisierung:	179,6	173,0	166,6	187,3
	Größenreduktion:	486,6	38,4	23,2	14,0
	Kommunikation:	–	43,5	40,1	51,8
128	Gesamt:	2 444,6	769,8	640,4	640,3
	Orthogonalisierung:	490,2	521,2	462,9	463,3
	Größenreduktion:	1 954,0	139,4	78,2	48,3
	Kommunikation:	–	108,9	99,0	128,2
256	Gesamt:	8 613,8	2 559,4	2 295,6	2 276,3
	Orthogonalisierung:	1 693,8	1 678,3	1 671,1	1 689,5
	Größenreduktion:	6 919,4	491,1	272,8	155,2
	Kommunikation:	–	389,4	351,0	431,0

Tabelle 3.16: Parallele Größenreduktion, CM-5 – „Rucksack-Gitter“ mit Zahlen mit 510 Binärstellen

man bei der Paragon einen exklusiven Zugriff auf die reservierten Prozessoren, während bei der CM-5 mehrere Benutzer gleichzeitig auf den parallelen Prozessoren rechnen können.

Bei den Rucksackgittern handelt es sich bei den angegebenen Zeiten um die Zeiten, die zur Reduktion des Gitters benötigt wurden (in CPU-Sekunden).[34] Aus Zeitgründen wurde bei den unimodular transformierten Gittern nur die Zeit für die ersten zehn Phasen gemessen. Die angegebenen Zeiten sind dabei die Mittelwerte für eine Phase.[35] Nach den vorher getroffenen Aussagen ist die Messung der Zeit für eine Phase durchaus aussagekräftig.

Gemessen wurden die Zeiten für die Orthogonalisierung, die Größenreduktion, die Kommunikation[36] und die Gesamtzeit. Die serielle Zeit wurde gemessen, indem jeder Knoten die komplette Reduktion durchführte. Die Zeiten für die Größenreduktion variieren natürlich von Prozessor zu Prozessor, da jeder Prozessor andere Spalten größenreduziert. Die Summe aus Größenreduktionszeit und Kommunikationszeit ist jedoch bei allen Prozessoren ungefähr gleich, da die Kommunikation vollständig synchron abläuft, d.h. Prozessoren, die schnell bei der Größenreduktion sind, müssen bei der Kommunikation entsprechend lange warten. Es stellt sich auch heraus, daß die Zeiten für die Größenreduktionen der einzelnen Prozessoren nicht zu stark voneinander abweichen, was für eine gute Lastverteilung des Algorithmus spricht. Bei den angegebenen Zeiten handelt es sich um die Zeiten des Prozessors, der für die Größenreduktion die meiste Zeit

[34] Dabei wurde die Tauschbedingung aus Definition 3.4 benutzt.

[35] Daher sind die angegebenen Zeiten nicht direkt mit denen des seriellen Algorithmus vergleichbar, da dort der Durchschnitt über alle Phasen gebildet wurde. Die Zeit für eine Phase nimmt aber während der Laufzeit ab, da die Zahlen immer kleiner werden.

[36] In der Kommunikationszeit ist auch die Zeit enthalten, die benötigt wird, die zu versendenden Spalten in einen Versendepuffer zu packen und wieder auszupacken.

n	Prozessoren:	1	4	8	16
64	Gesamt:	1,55	0,69	0,56	0,49
	Orthogonalisierung:	0,31	0,31	0,31	0,31
	Größenreduktion:	1,25	0,35	0,19	0,11
	Kommunikation:	–	0,04	0,06	0,07
140	Gesamt:	28,16	8,94	5,72	4,09
	Orthogonalisierung:	1,94	1,95	1,95	1,95
	Größenreduktion:	26,22	6,45	3,39	1,74
	Kommunikation:	–	0,54	0,38	0,40

Tabelle 3.17: Eine Phase mit paralleler Größenreduktion, Paragon – unimodular transformierte Gitter

n	Prozessoren:	1	4	8	16
128	Gesamt:	1 320,9	591,5	468,3	411,6
	Orthogonalisierung:	283,2	281,0	281,1	279,3
	Größenreduktion:	1 037,6	269,0	139,2	74,0
	Kommunikation:	–	41,5	47,9	58,2
256	Gesamt:	5 472,3	2 284,7	1 751,2	1 484,5
	Orthogonalisierung:	1 007,3	1 006,3	1 007,1	1 006,1
	Größenreduktion:	4 464,9	1 138,0	585,7	305,8
	Kommunikation:	–	140,4	158,3	172,6

Tabelle 3.18: Parallele Größenreduktion, Paragon – „Rucksack-Gitter“ mit Zahlen mit 510 Binärstellen

benötigt. Damit ist eine faire Betrachtungsweise der Effizienz der Parallelisierung der Größenreduktion möglich.

Die erzielten Ergebnisse lassen sich folgendermaßen zusammenfassen:

- Bei Benutzung eines Parallelrechners mit schneller Kommunikation ist im Gegensatz zu einem Workstation-Netz der sequentielle Programmteil, der durch die Orthogonalisierung gegeben ist, der begrenzende Faktor bei der zu erreichenden Beschleunigung. Die Zeit für eine Phase kann größenordnungsmäßig auf die Zeit für die sequentielle Orthogonalisierung gesenkt werden.

- Eine effiziente Parallelisierung der Größenreduktionsphase ist (auch unter Beachtung der Kommunikationskosten) tatsächlich möglich.

- Bei der Reduktion von Rucksackgittern werden nur vergleichsweise wenig Größenreduktionen durchgeführt. Bei den gemessenen Dimensionen ist die Zeit, die für die Größenreduktion benötigt wird, nur etwa 3 bis 4 mal so groß wie die für die Orthogonalisierung. Daher wird nur eine maximale Beschleunigung von ungefähr 4 erreicht. Im Vergleich mit dem seriellen Schnorr/Euchner-Algorithmus läßt sich daher kein großer Gewinn erzielen (vgl. Abschnitt 3.5.5).

n	Prozessoren:	1	16	32	64
100	Gesamt:	18,72	2,11	1,56	1,24
	Orthogonalisierung:	0,87	0,34	0,35	0,41
	Givens:	0,86	0,25	0,25	0,26
	Matrixkonvertierung:	0,28	0,02	0,01	0,01
	Größenreduktion:	17,57	1,42	0,84	0,54
	Spaltenaustausch:	–	0,33	0,36	0,32
140	Gesamt:	72,61	6,70	4,45	3,51
	Orthogonalisierung:	2,43	0,71	0,68	0,74
	Givens:	2,41	0,54	0,48	0,48
	Matrixkonvertierung:	0,57	0,04	0,02	0,01
	Größenreduktion:	69,62	4,90	2,85	1,84
	Spaltenaustausch:	–	1,00	0,91	0,91

Tabelle 3.19: Eine parallele Phase, CM-5 – unimodular transformierte Gitter

- Bei den unimodular transformierten Gittern sind sehr gute Ergebnisse erreichbar, da sie einen hohen parallelen Anteil aufweisen. Es werden viele Größenreduktionen durchgeführt. Dies gilt auch im Verleich zu dem Schnorr/Euchner-Algorithmus, da nach Abschnitt 3.5.5 die serielle Variante des neuen Algorithmus kaum schlechter als der Schnorr/Euchner-Algorithmus ist. Mit 16 Prozessoren ist eine Beschleunigung von ungefähr 8 erreichbar.

- Insgesamt ist dieser parallele Ansatz sinnvoll für Parallelrechner mit wenig Prozessoren (2 bis 16) für Gitter, bei denen viele Größenreduktionen durchgeführt werden müssen.

Parallele Orthogonalisierung und parallele Größenreduktion

Es ist nun eine naheliegende Idee, obigen Algorithmus so abzuändern, daß auch die Orthogonalisierung parallel ausgeführt wird. Dies ist relativ einfach möglich. Jeder Prozessor verwaltet die gesamte Matrix. Die parallele Berechnung der Givens-Rotationen geschieht gemäß Algorithmus 3.13 aus Kapitel 3.9.3. Am Ende dieser Phase besitzt der letzte Prozessor die triangularisierte Matrix. Diese wird nun an alle anderen Prozessoren versendet. Mit Hilfe der triangularisierten Matrix können die Prozessoren die Gram-Schmidt-Orthogonalisierung berechnen und fahren wie bisher fort.

Der serielle Programmanteil wird auch dadurch verringert, daß nun die Umwandlung der Matrix in der exakten Darstellung in eine „double“-Matrix, die zur Orthogonalisierung benötigt wird, parallel erfolgen kann.

Zusammenfassend hat dieser Algorithmus also folgende Eigenschaften:

- Es wird wiederum viel Speicherplatz benötigt, da jeder Prozessor die gesamte Matrix verwaltet.

- Die theoretische Effizienz der Givensrotationen ist asymptotisch 1/2, die theoretische Effizienz der Größenreduktion ist 1.

n	Prozessoren:	1	16	32	64
64	Gesamt:	917,3	233,3	228,6	266,6
	Orthogonalisierung:	186,9	134,8	146,7	183,5
	Givens:	183,3	94,6	100,0	123,8
	Matrixkonvertierung:	80,6	6,9	4,1	1,5
	Größenreduktion:	649,4	45,0	29,1	20,3
	Spaltenaustausch:	–	49,2	50,2	61,7
128	Gesamt:	7 490,9	1 354,2	1 110,8	1 099,8
	Orthogonalisierung:	1 437,9	751,4	691,6	759,9
	Givens:	1 417,2	556,9	485,8	502,6
	Matrixkonvertierung:	362,4	35,9	19,9	8,3
	Größenreduktion:	5 691,2	386,6	217,5	129,6
	Spaltenaustausch:	–	209,6	197,9	209,2
256	Gesamt:	27 007,6	5 314,9	4 147,5	3 769,8
	Orthogonalisierung:	4 141,1	3 144,3	2 701,9	2 704,9
	Givens:	4 044,1	2 402,0	1 914,2	1 811,1
	Matrixkonvertierung:	1 003,5	138,0	78,5	34,8
	Größenreduktion:	21 862,4	1 430,9	744,1	400,9
	Spaltenaustausch:	–	720,5	741,9	691,4

Tabelle 3.20: Parallele Basisreduktion, CM-5 – „Rucksack-Gitter“ mit Zahlen mit 510 Binärstellen

- Folgende Aktionen werden noch seriell ausgeführt:
  - Berechnung der Gram-Schmidt Orthogonalisierung aus der triangularisierten Matrix
  - Tausch der Basisvektoren

  Diese beiden Punkte sind aber in der Laufzeit vernachlässigbar.
- Folgende Kommunikation muß ausgeführt werden:
  - Zeilenaustausch in Gleitkommadarstellung während der parallelen Givens-Rotation
  - Spaltenaustausch in exakter Darstellung nach der Größenreduktion
  - Zusätzlich: Verteilung der Gleitkomma-Matrix nach der Givens-Orthogonalisierung

Die Werte, die mit diesem Algorithmus ermittelt wurden, sind in den Tabellen 3.19 und 3.20 beschrieben.[37] Die Messungen wurden wie vorher durchgeführt.[38] Die Zeiten, die unter dem Punkt „Givens“ aufgeführt sind, sind die Zeiten, die zur Berechnung der triangularisierten Matrix mit Hilfe von Givens-Rotationen benötigt wurden. Die Zeiten zur Konvertierung der ganzzahligen Matrix in eine Gleitkomma-Matrix zur Orthogonalisierung sind ebenfalls aufgeführt. Mit „Spaltenaustausch“ ist die Zeit zum Austausch der Spalten nach der parallelen Größenreduktion gemeint.[39]

Die *Ergebnisse für unimodular transformierte Gitter* lassen sich folgendermaßen zusammenfassen:

- Bei 16 Prozessoren kann für die Berechnung der Givens-Rotationen eine Effizienz von ungefähr 1/4 erreicht werden.
- Die Effizienz bei der gesamten Orthogonalisierungsphase ist schlechter. Dies ist in der Kommunikationszeit bei der Matrixverteilung begründet.
- Insgesamt ist ein gutes Ergebnis erreichbar. Für 16 Prozessoren und $n = 140$ ist die Effizienz $0,68$. Dies gilt auch etwa im Vergleich zum seriellen Schnorr/Euchner-Algorithmus, da dieser für unimodular transformierte Gitter kaum schneller ist.

Bei *Rucksackgittern* werden insgesamt weniger Rechnungen ausgeführt (sowohl bei der Größenreduktion als auch bei der Orthogonalisierung), so daß sich die Zeiten für die seriellen Anteile und insbesondere die Kommunikationszeiten stärker negativ auf die zu erreichenden Effizienzwerte auswirken.

Zu Beginn des Algorithmus weisen die Rucksackgitter schon sehr viele Nullen auf, so daß der serielle Algorithmus sehr wenig zu rechnen hat, während beim parallelen Algorithmus

[37] Der Algorithmus wurde in Zusammenarbeit mit Patrick Häusser implementiert (s. auch [25]).

[38] Es wurde die Original-LLL-Tauschbedingung mit $\delta = 0.9$ verwendet.

[39] Die Zeiten für die Givens-Rotationen und für die Größenreduktion sind jeweils die maximalen Zeiten über alle Prozessoren. Daher ist die Summe der Einzelzeiten nicht unbedingt gleich der Gesamtzeit. Die Zeit für den Spaltenaustausch ist die Zeit des Prozessors, der für die Größenreduktion die meiste Zeit benötigte, da in der Kommunikationszeit der anderen Prozessoren die entsprechende Wartezeit enthalten ist.

n	binäre Zahlenlänge	zu eliminierende Elemente	davon schon Null
32	510	296 736	25 549
32	3 330	1 634 754	30 261
64	510	1 892 800	373 053
64	3 330	7 395 960	385 709
128	510	10 336 512	5 727 595
128	3 330	39 627 510	5 679 599
256	510	44 672 768	39 302 732

Tabelle 3.21: Anzahl der Nullen bei Givens-Rotationen bei „Rucksackgittern“

der Kommunikationsaufwand derselbe bleibt und Prozessoren, die nichts rechnen müssen auf diejenigen warten müssen, die Rechnungen durchführen müssen. Der Anteil der Nullen sinkt im Laufe des Algorithmus, jedoch ist das Verhältnis von Elementen, die bereits Null sind, zu der Gesamtzahl der zu Null zu rotierenden Elemente auch während der Ausführung des gesamten Algorithmus sehr hoch (s. Tabelle 3.21). Dieses Verhältnis steigt noch mit der Dimension. Daher ließ sich nur eine Effizienz von ca. $0,35$ erreichen. Dies ist auch deshalb nicht sehr gut, weil der serielle Schnorr/Euchner-Algorithmus ungefähr dreimal so schnell ist, wie der hier zu Grunde gelegte serielle Algorithmus.

Je größer aber die Ausgangszahlen sind, desto mehr Phasen werden durchgeführt. Dies gilt auch, wenn für δ bei der Tauschbedingung ein größerer Wert gewählt wird (z.B. $0,99$). Damit sinkt der Anteil der Elemente, die schon Null sind, im Verhältnis zu der Gesamtzahl der zu Null zu rotierenden Elemente (s. Tabelle 3.21). Daher ist bei sehr großen Zahlen mit einem Gewinn bei der parallelen Ausführung zu rechnen.

Hinzu kommt, daß bei sehr großen Zahlen der Datentyp „double“ nicht mehr ausreichend ist. Bei der Verwendung eines Multi Precision Datentyps steigt aber der Rechenaufwand, wodurch das Verhältnis von Rechenzeit zu Kommunikationszeit günstiger wird. Dies gilt natürlich auch für alle anderen Gitter.

Die Implementierung eines Algorithmus mit Multi Precsion Arithemtik ist zur Zeit Gegenstand einer Diplomarbeit von Patrick Häusser [25].

Mit weniger Prozessoren ließen sich bessere Effizienzwerte erreichen.

Weitere Alternativen

Bei den bisher vorgestellten Algorithmen verwaltet jeder Prozessor die gesamte Matrix. Will man den parallelen Algorithmus auch dazu nutzen, daß der Speicherbedarf gleichmässig auf alle Prozessoren verteilt wird, muß man einen Algorithmus wählen, bei dem die Zeilen bzw. Spalten der Matrix immer gleichmäßig über alle Prozessoren verteilt sind.

Zur *parallelen Ausführung der Givens-Rotationen* kann wieder ein Algorithmus der Form von Algorithmus 3.13 aus Kapitel 3.9.3 angewendet werden. Dieser Algorithmus verfügt über ein relativ einfaches Kommunikationsschema. Außerdem ist ein einfaches „Umschalten“ zwischen einer zeilenweisen und einer spaltenweisen Aufteilung zwischen Orthogonalisierung und Größenreduktion möglich. Givens-Rotationen sind oft in mathematischen

Funktionsbibliotheken enthalten. Dies gilt auch für die Parallelrechner CM-5 und CM-2 [70]. Damit ist auch eine Verwendung dieser parallelen Bibliotheksfunktionen möglich.

Die *parallele Ausführung der Größenreduktion* muß aber etwas modifiziert werden. Die Größenreduktion kann beispielsweise ebenfalls nach Algorithmus 3.13 parallelisiert werden. Hier ist jedoch auch eine etwas einfachere Variante des Algorithmus ähnlich zu oben möglich. Die Grundidee ist wieder, daß alle Spalten unabhängig voneinander größenreduziert werden können. Die Spalten werden gleichmäßig über die Prozessoren verteilt und abgespeichert. Zusätzlich werden die Spalten ringförmig über die Prozessoren versendet, wobei sie mit den Spalten, die in den Prozessoren gespeichert sind, größenreduziert werden. Die gespeicherten Spalten bleiben dabei unverändert. D.h. im ersten parallelen Schritt werden alle Spalten i mit den gespeicherten Spalten $i-1$ größenreduziert für $i \geq 2$. Im nächsten Schritt werden die Spalten i mit den Spalten $i-2$ größenreduziert und so weiter. Während Algorithmus 3.13 $2n$ parallele Schritte benötigt, sind es hier nur n, falls n Prozessoren zur Verfügung stehen. Da (maximal) $n^2/2$ Elemente größenreduziert werden müssen, beträgt die **Effizienz dann** $1/2$. Da zur Größenreduktion immer vollständige Spalten der Basismatrix verknüpft werden müssen, entsteht kein weiterer Effizienzverlust gegenüber dem seriellen Fall wie bei den Givensrotationen, bei denen die zu verknüpfenden Zeilen immer kürzer werden.

Insgesamt dürfte die Laufzeit dieses Algorithmus größer sein als die des obigen Algorithmus, da die Effizienz der Größenreduktionsphase schlechter ist und mehr Kommunikation durchgeführt werden muß.

Dieser Ansatz wird zur Zeit im Rahmen der Diplomarbeit von Patrick Häusser auf der CM-5 der GMD untersucht [25].

Ein weiteres Ziel dieser Arbeit besteht darin, wiederverwendbare Programme zu erhalten, d.h.

- Wiederverwendbarkeit einzelner Module in anderen parallelen Programmen und
- Wiederverwendbarkeit der Programme und Moduln auf anderen Parallelrechnern.

Ein *anderer Algorithmus*, bei dem sowohl die Größenreduktion als auch die Orthogonalisierung parallel durchgeführt wird und bei dem die Matrix über die Prozessoren verteilt ist, wurde von Kay Wiese auf einem Intel iPSC/860 Hypercube Parallelrechner des Rechenzentrums der Universität des Saarlandes implementiert [73]. Es handelt sich dabei um einen MIMD-Rechner mit verteiltem Speicher, wobei die Prozessoren durch ein Verbindungsnetzwerk in der Form eines Hyperwürfels miteinander verbunden sind. Die Programmierung erfolgte in „C“ mit Nachrichtenaustausch unter Verwendung der Bibliothek „thales“.

Es wurde ein zyklisches Schema gemäß [60] implementiert (s. Kapitel 3.6 und 3.7), das n Prozessoren effizient ausnutzen kann, d.h. Zeilen- (und Spalten-) Verknüpfungen werden in jedem Prozessor sequentiell durchgeführt, verschiedene Verknüpfungen aber gleichzeitig durch verschiedene Prozessoren.

Die mit diesem Algorithmus erzielten Ergebnisse waren aber nicht sehr gut. Dies liegt wohl in erster Linie an dem hohen Kommunikationsaufwand, der auch dadurch entsteht, daß bei dem zyklischen Schema mit dynamischer Datenaufteilung auch einige Steuerinformationen

versendet werden. Das Versenden der Steuerinformationen hätte vielleicht eingespart werden werden können.

In dieser Arbeit [73] sind grundsätzliche Techniken und Implementierungsdetails bei der Programmierung von Parallelrechnern im allgemeinen und des iPSC860 im speziellen beschrieben (z.B. die Einbettung von linearen Prozessorfeldern in Hyperwürfel, die Behandlung lokaler und globaler Indizes, die Versendung langer Zahlen), für die in der vorliegenden Arbeit kein Platz war. Außerdem sind hier die verschiedenen Parallelisierungsmöglichkeiten, die in Kapitel 3.6 und 3.7 nur angedeutet wurden, genauer erklärt.

Die Erfahrungen, die in dieser Arbeit gemacht wurden, haben dazu geführt, über einfachere parallele Schemata nachzudenken. Ergebnisse dieser Überlegungen sind die oben beschriebenen Ansätze.

3.9.6 Implementierung auf einem SIMD-Rechner

Im folgenden wird die Grundstruktur eines SIMD-Algorithmus, der n^2 Prozessoren effizient nutzen kann, beschrieben.

Für die parallelen Givens-Rotationen kann ein Algorithmus gemäß Algorithmus 3.14 benutzt werden.

Für die parallele Größenreduktion kann ein ähnlicher Algorithmus gewählt werden jedoch mit der Modifikation aus Kapitel 3.9.5 für die Größenreduktion, d.h. Kopien der Spalten werden zyklisch versendet und zur Größenreduktion der gespeicherten Spalten verwendet. Es ist zu beachten, daß bei der Größenreduktion Verknüpfungen sowohl exakt auf der Basismatrix als auch auf der Koeffizientenmatrix in Gleitkommadarstellung stattfinden. Insbesondere ist die Basismatrix nicht in Dreiecksform, wodurch im Gegensatz zu den Givens-Rotationen immer komplette Spalten verknüpft werden müssen. Im Programm wird dies dadurch erreicht, daß auch die Verknüpfungsparameter zyklisch versendet werden [43]. Dies funktioniert jedoch nur für quadratische Matrizen. Insgesamt müssen $3(n-1)$ Schleifendurchläufe durchgeführt werden. Im seriellen Fall müssen $n(n-1)/2$ Elemente größenreduziert werden. Damit werden $n^2(n-1)/2$ Multiplikationen auf exakten Zahlen durchgeführt. Die Effizienz auf der Ebene von Multiplikationen exakter Zahlen ist also 1/6 für die Größenreduktionsphase [43].

Durch das SIMD-Modell muß in jedem Schleifendurchlauf die Zeit für zwei exakte Multiplikationen aufgewendet werden. Damit ergibt sich eine Effizienz von 1/12.

Ein Nachteil bei der Verwendung eines „n^2-Algorithmus“ ist, daß nicht ausgenutzt werden kann, wenn Elemente bei der Orthogonalisierung bereits Null sind und nicht mehr zu Null rotiert werden müssen, bzw. wenn Elemente nicht größenreduziert werden müssen. Außerdem wird die Laufzeit einer exakten Multiplikation durch die längsten Zahlen bestimmt. Damit sind die Zeiten für eine Größenreduktionsphase bei gleicher maximaler Zahlenlänge gleich für alle Gitter. Dies wirkt sich negativ aus bei Gittern, bei denen weniger gerechnet werden muß oder bei denen stark unterschiedlich lange Zahlen auftreten (z.B. bei „Rucksackgittern“).

Bei der Implementierung einer ganzzahligen SIMD-Arithmetik muß die Zahlenlänge aller Prozessoren statisch zu Beginn eingestellt werden. Aktualisiert man diese Länge nach jeder Phase, ist eine Verbesserung erreichbar.

Eine genaue Beschreibung und Analyse des Algorithmus ist in [43] zu finden.

n	Orthogonalisierung	Größenreduktion
16	2,02	11,91
32	4,11	24,65
64	7,28	48,23
128	14,59	95,09

Tabelle 3.22: Eine parallele Phase, CM-2, n^2 Prozessoren, Zahlenlänge 160

n	$n^2 (\leq 8k)$	$16k$	seriell
16	13,93	–	0,07
32	28,76	–	0,43
64	76,23	–	3,93
128	456,13	233,63	50,71

Tabelle 3.23: Eine parallele Phase, CM-2, $8k$ und $16k$ Prozessoren, unimodular transf. Gitter

Der Algorithmus wurde von Franz Muller auf dem SIMD-Rechner CM-2 der GMD implementiert [43].

Zur Programmierung wurde die datenparallele Programmiersprache „C*“ gewählt [67]. Unter der Verwendung von „thales“-Funktionen wurde eine eigene SIMD-Ganzzahlarithmetik implementiert.

In Tabelle 3.22 sind die Zeiten auf dem parallelen Rechenfeld für eine Phase bei der Verwendung von n^2 Prozessoren für Gitter mit der Zahlenlänge 160 Bits angegeben (in CPU-Sekunden).

In Tabelle 3.23 werden die Zeiten für eine Phase des seriellen Algorithmus und des parallelen Algorithmus mit maximal $8k$ bzw. (wo sinnvoll) $16k$ Prozessoren für unimodular transformierte Gitter vorgestellt. Die seriellen Zeiten wurden dabei auf dem Hostrechner der CM-2 ermittelt. In Tabelle 3.24 sind die entsprechenden Zeiten für „Rucksackgitter“ mit Zahlen der Länge 340 Bit dargestellt. Ein Vergleich mit den seriellen Zeiten ist bei der Messung nur einer Phase aber nur sehr bedingt aussagekräftig, da gerade zu Beginn bei Rucksackgittern nur wenig gerechnet werden muß.

In Tabelle 3.25 sind die Zeiten für eine komplette Reduktion für sehr kleine Rucksackgitter dargestellt.

n	$8k$	$16k$	seriell
128	105,1	54,4	0,4
256	819,3	415,7	2,2
512	6 470,6	3 337,6	6,0

Tabelle 3.24: Eine parallele Phase, CM-2, $8k$ und $16k$ Prozessoren, „Rucksackgitter“

n	n^2 Proz.	seriell	Effizienz
8	196,7	0,12	0,031
16	4 934,0	9,00	0,023
32	11 942,0	62,70	0,016

Tabelle 3.25: Parallele Reduktion, CM-2, n^2 Prozessoren, „Rucksackgitter“

Die wesentlichen Ergebnisse, die mit dem SIMD-Algorithmus erzielt wurden, lassen sich folgendermaßen zusammenfassen:

- Der Algorithmus hat eine lineare Wachstumsrate in der Matrixdimension (für gleiche Zahlengrößen).
- Da eine Verdoppelung der Prozessorzahl zu einer Halbierung der Laufzeit führt, weist der Algorithmus eine konstante Effizienz auf.
- Rechnet man die Geschwindigkeit der parallelen Prozessoren im Verhältnis zur Geschwindigkeit des für Vergleichstests verwendeten sequentiellen Prozessors ein (Faktor ca. 3 200), ist die tatsächlich erzielte Effizienz für unimodular transformierte Gitter 1/25. Dies ist ein guter Wert, wenn man bedenkt, daß sehr viele Prozessoren eingesetzt werden können.
- Aus Zeitgründen lassen sich nur sehr kleine Beispiele komplett durchrechnen. Für Rucksackgitter der Dimension 8 (16, 32) beträgt die erzielte Effizienz unter Berücksichtigung der Geschwindigkeit der Prozessoren $0,03$ $(0,023,\ 0,016)$. Es ist zu erwarten, daß die Effizienz für größere Rucksackgitter sinkt, da der Rechenaufwand im Verhältnis zur Dimension geringer wird.

3.9.7 Überblick über die diskutierten Implementierungen

An dieser Stelle soll ein kurzer Überblick über die diskutierten Implementierungen gegeben werden.

Für eine parallele Implementierung der Orthogonalisierung wurden die folgenden Methoden betrachtet.

- Die Zeilen „fallen von oben in das Prozessorfeld“ („durchfallend“, s. Algorithmen 3.13 und 3.14).
- Die Zeilen werden zyklisch zwischen den Prozessoren ausgetauscht („zyklisch“, s. Abschnitt 3.6 und Abschnitt 3.7).

Folgende Möglichkeiten für die parallele Größenreduktion wurden vorgestellt.

- Jeder Prozessor verwaltet die komplette Matrix, d.h. der Algorithmus weist einen hohen Speicherbedarf auf. Jeder Prozessor größenreduziert eine festgelegte Menge von Spalten. Danach werden (z.B. über eine Ringstruktur) die größenreduzierten Spalten ausgetauscht, so daß jeder Prozessor über die größenreduzierte Basis verfügt („verteilt“, s. Abschnitt 3.9.4).

- Die Spalten sind gleichmäßig verteilt. Kopien der Spalten werden ringförmig versendet und gegen die gespeicherten Spalten größenreduziert („Ring“, s. Abschnitt 3.9.5).

- Die Spalten werden zyklisch zwischen den Prozessoren ausgetauscht („zyklisch“, s. Abschnitt 3.6 und Abschnitt 3.7).

In Tabelle 3.26 sind die verschiedenen Implementierungen dargestellt. In der mit P bezeichneten Spalte ist die von dem Algorithmus maximal nutzbare Prozessorzahl angegeben, wobei n die Matrixdimension ist.

Seite	P	Architektur	Modell	Orthogonalisierung	Größenreduktion
99	n	Rechnernetz	C + LiPS	seriell	verteilt
103	n	MIMD	C + Nachrichtentausch	seriell	verteilt
107	n	MIMD	C + Nachrichtentausch	durchfallend	verteilt
110	n	MIMD	C + Nachrichtentausch	durchfallend	Ring
111	n	MIMD	C + Nachrichtentausch	zyklisch	zyklisch
112	n^2	SIMD	SIMD-Sprache C*	durchfallend	Ring

Tabelle 3.26: Überblick über die diskutierten Implementierungen

3.10 Zusammenfassung

Es wurden verschiedene Möglichkeiten zur Parallelisierung von Algorithmen zur Basisreduktion in Gittern diskutiert.

Die *theoretischen Untersuchungen* lassen sich folgendermaßen zusammenfassen:

- Betrachtet man nur die Anzahl der arithmetischen Operationen und die Anzahl der zu versendenden Zahlen, läßt sich sowohl bei der Verwendung von n Prozessoren als auch bei der Verwendung von n^2 Prozessoren eine optimale Parallelisisierung erreichen. Für die Parallelisierung bei n^2 Prozessoren wurde eine neue Version eines Basisreduktionsalgorithmus vorgestellt, deren Effizienz formal bewiesen wurde.

- Außer bei der Parallelisierung des Schnorr-Euchner Algorithmus und ähnlicher Strategien für n Prozessoren können aber theoretisch größere Zahlen entstehen als bei seriellen Algorithmen. Eine genauere Analyse der Zahlengröße hat aber gezeigt, daß die bisher bekannte Schranke für die Zahlengröße von $O(n^2 \log B)$ aus [72] auf $O(n \log B)$ verbessert werden kann.

- Die parallelen Algorithmen können die effiziente Gleitkomma-Arithmetik benutzen. Eine Fehlerabschätzung wurde hergeleitet, so daß falsche Tausche vermieden werden können.

Praktische Untersuchungen haben ergeben, daß bei den parallelen Algorithmen für die hier betrachteten Beispiele keine wesentlich größeren Zahlen entstehen als bei den seriellen Algorithmen. Außerdem hat sich gezeigt, daß bei den untersuchten Beispielen[40] die Abfrage der Fehlerbedingung bei den Tauschen weggelassen werden kann.

Es wurden zwei Klassen von Gittern untersucht.

Bei der ersten Klasse („unimodular transformierte Gitter“) muß „viel gerechnet werden“ und es werden „viele“ Größenreduktionen durchgeführt. Die Zeit für die Orthogonalisierung ist vernachlässigbar gegenüber der Zeit für die Größenreduktion.

Bei der zweiten Klasse von Gittern, wie sie bei der Lösung von Rucksackproblemen entstehen („Rucksackgitter“), sind bei der Orthogonalisierung schon „sehr viele“ Elemente eliminiert, und es müssen nur „wenige“ Größenreduktionen durchgeführt werden. Die Zeit für die Orthogonalisierung ist zwar immer noch geringer als die Zeit für die Größenreduktion, ist aber nicht mehr vernachlässigbar. Es ist klar, daß diese „dünnbesetzten“ Matrizen bei einer Parallelisierung Probleme bereiten.

Folgende Implementierungen wurden betrachtet:

- Der neue parallele Algorithmus wurde seriell implementiert und mit dem seriellen Schnorr/Euchner - Algorithmus verglichen. Für unimodulare Matrizen ist der neue Ansatz nur bis zu dem Faktor $1,6$ und für Rucksackgitter bis zu dem Faktor $3,5$ langsamer.

- Eine Version, bei der nur die Größenreduktion parallel durchgeführt wird, wurde mit Hilfe des Systems „LiPS“ für ein Workstationnetz implementiert. Für Rucksackgitter ist der serielle Programmanteil (Orthogonalisierung) und die Kommunikationszeit im Verhältnis zur parallelen Rechenzeit zu hoch, so daß keine guten Ergebnisse erzielt werden können. Für unimodular transformierte Gitter können Gewinne erzielt werden, die aber auf Grund der Asynchronität des Rechnernetzes und der hohen Kommunikationszeiten eher bescheiden sind.

- Dieser Algorithmus wurde auch auf MIMD-Rechnern mit der Programmiersprache „C“ und Funktionen zum Nachrichtenaustausch implementiert. Dabei können für unimodular transformierte Gitter gute Ergebnisse erzielt werden. Bei Rucksackgittern ist jedoch der serielle Programmanteil, der durch die Orthogonalisierung gegeben ist, zu hoch für gute Ergebnisse.

  Daher wurde ein Algorithmus implementiert, bei dem auch die Orthogonalisierung parallel durchgeführt wird. Dieser Algorithmus weist für unimodular transformierte Gitter eine Effizienz von $0,68$ und für Rucksackgitter von $0,35$ auf.[41]

- Ein Algorithmus, der n^2 Prozessoren effizient nutzen kann, wurde auf dem SIMD-Rechner CM-2 in der datenparallelen Programmiersprache „C*“ implementiert. Es kann gezeigt werden, daß die theoretische Laufzeit $O(n)$ für eine Phase auch praktisch erreicht wird. Außerdem weist der Algorithmus eine konstante Effizienz auf, was sich durch eine Halbierung der Laufzeit bei einer Verdoppelung der Prozessorzahl zeigt.

[40] also für den Zahlenbereich, für den der Datentyp „double“ noch ausreichend ist

[41] im Vergleich zu einer seriellen Ausführung

Die parallelen Prozessoren dieses Rechners sind allerdings sehr langsam und nicht für numerische Berechnungen gedacht. Rechnet man dies ein, so erhält man für unimodular transformierte Gitter eine Effizienz von 1/25, was sehr gut ist, wenn man bedenkt, daß sehr viele Prozessoren eingesetzt werden können.

„Dünn besetzte“ Matrizen sind für „n^2-Algorithmen“ jedoch schlecht geeignet.

Insgesamt bleibt festzuhalten, daß „dünnn besetzte“ Matrizen insbesondere bei parallelen Algorithmen mit n^2 Prozessoren Schwierigkeiten bereiten. Dies bietet einen Ansatzpunkt für weitere Untersuchungen. Es ist z.B. zu erwarten, daß auch bei solchen Gittern für Probleme mit sehr großen Zahlen bei Algorithmen mit n Prozessoren gute Gewinne erzielt werden können.

Außerdem sollten weitere Anwendungen der Gitterbasisreduktion auf ihren Erfolg bei der Verwendung der parallelen Algorithmen hin untersucht werden.

Anhang A

Unimodulare Matrizen

Definition A.1 *Eine Matrix $U \in \mathbf{Z}^{n \times n}$ heißt unimodular, wenn gilt*

$$|\det U| = 1$$

Nach der Cramerschen Regel [64] berechnet sich die Inverse einer invertierbaren Matrix folgendermaßen.

$$A^{-1} = \frac{1}{\det A} \begin{pmatrix} \det A_{11} & \dots & \det A_{n1} \\ \vdots & & \vdots \\ \det A_{n1} & \dots & \det A_{nn} \end{pmatrix}$$

Dabei ist A_{ij} die Matrix, die aus A durch Streichen der i-ten Zeile und j-ten Spalte entsteht.

Daher gilt, daß die Inverse einer unimodularen Matrix ganzzahlig und somit wieder unimodular ist.

Unimodulare Transformationen sind damit bijektiv auf dem $\mathbf{Z}^n$, d.h.

$$\{x \in \mathbf{Z}^n\} = \{x | x = Uw, w \in \mathbf{Z}^n\}$$

für unimodulare Matrizen U.

Anhang B

Lösungen ganzzahliger Gleichungssysteme

Das Lösen ganzzahliger Gleichungssysteme wird unter anderem in [44, 59] behandelt.

Gesucht sei die Menge S der ganzzahligen $x \in \mathbf{Z}^n$ mit

$$Ax = b$$

für $A \in \mathbf{Z}^{m \times n}$ und $b \in \mathbf{Z}^m$. Es sei $z = \text{Rang}(A)$.

Um S zu bestimmen, wird A in die *Hermite-Normalform* umgewandelt [44]. Da jedoch hier A nicht unbedingt quadratisch ist und nicht vollen Rang besitzen muß, wird folgende (leicht einsehbare) Variante der Hermite-Normalform verwendet.

Satz B.1 *Sei $A \in \mathbf{Z}^{m \times n}$ beliebig. Dann gibt es unimodulare Matrizen (s. Anhang A) $U \in \mathbf{Z}^{m \times m}$ und $V \in \mathbf{Z}^{n \times n}$ mit*

$$UAV = \begin{pmatrix} H & 0 \\ K & 0 \end{pmatrix} \in \mathbf{Z}^{m \times n}$$

wobei $H \in \mathbf{Z}^{z \times z}$ eine linke untere Dreiecksmatrix und $K \in \mathbf{Z}^{(m-z) \times (z)}$ ist.

Es gilt nun für die Menge der Lösungen:

$$\begin{aligned} S &= \{x \in \mathbf{Z}^n | Ax = b\} \\ &= \{x \in \mathbf{Z}^n | UAx = Ub\} \\ &= \{x | x = Vw, UA(Vw) = Ub, w \in \mathbf{Z}^n\} \end{aligned}$$

Dies gilt, da $\{x \in \mathbf{Z}^n\} = \{x | x = Vw, w \in \mathbf{Z}^n\}$, weil V unimodular ist. Damit gilt weiter:

$$\begin{aligned} S &= \{x | x = Vw, \begin{pmatrix} H & 0 \\ K & 0 \end{pmatrix} w = Ub, w \in \mathbf{Z}^n\} \\ &= \{x | x = Vw, Hw_1 = U_1 b, Kw_2 = U_2 b, w_1 \in \mathbf{Z}^z w_2 \in \mathbf{Z}^{n-z}\} \end{aligned}$$

wobei U_1 die Matrix bestehend aus den ersten z Zeilen von U und U_2 die Matrix bestehend aus den restlichen Zeilen von U ist.

Damit ist S die leere Menge, d.h. das ganzzahlige Gleichungssystem hat keine Lösung, wenn

$$w_1 = H^{-1}U_1 b$$

nicht ganzzahlig ist, oder

$$KH^{-1}U_1 b \neq U_2 b$$

Andernfalls läßt sich die Lösungsmenge folgendermaßen beschreiben, wobei V_1 die Matrix bestehend aus den ersten z Spalten von V und V_2 die Matrix bestehend aus den restlichen Spalten von V ist.

$$\begin{aligned} S &= \{x | x = V_1 w_1 + V_2 w_2, w_1 = H^{-1}U_1 b, w_2 \in \mathbf{Z}^{n-z}\} \\ &= \{x | x = V_1 H^{-1}U_1 b + V_2 w_2, w_2 \in \mathbf{Z}^{n-z}\} \end{aligned}$$

Statt der Zerlegung in die Hermite-Normalform kann auch die Zerlegung in die *Smith-Normalform* gewählt werden [44]. Diese ist zwar aufwendiger zu berechnen, aber die Berechnung von D^{-1}, wobei D eine Diagonalmatrix ist, gestaltet sich einfacher, und der Test, ob das System eine ganzzahlige Lösung besitzt, ist einfacher.

Satz B.2 *Sei $A \in \mathbf{Z}^{m \times n}$ beliebig. Dann gibt es unimodulare Matrizen $U \in \mathbf{Z}^{m \times m}$ und $V \in \mathbf{Z}^{n \times n}$ mit*

$$UAV = \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} \in \mathbf{Z}^{m \times n}$$

wobei $D \in \mathbf{Z}^{z \times z}$ eine Diagonalmatrix ist.

Dann gilt analgo zu oben:

$$S = \{x | x = V_1 D^{-1}U_1 b + V_2 w_2, w_2 \in \mathbf{Z}^{n-z}\}$$

falls

- $D^{-1}U_1 b$ ganzzahlig ist und
- $U_2 b = 0$

Anhang C

Das Rucksackproblem

Das Rucksackproblem besteht darin, für gegebene $a_1, \ldots, a_n, s \in \mathbf{N}$ die Gleichung

$$\sum_{i=1}^{n} a_i x_i = s$$

mit $x_i \in \{0, 1\}$ für $i = 1, \ldots, n$ zu lösen (s. z.B. [57]).

Das Problem kann auf das Finden kurzer Vektoren in einem Gitter zurückgeführt werden. In [57] wird dazu das Gitter mit der Basis

$$b_1 = \begin{pmatrix} 2 \\ 0 \\ \vdots \\ 0 \\ na_1 \\ 0 \end{pmatrix}, \qquad b_2 = \begin{pmatrix} 0 \\ 2 \\ \vdots \\ 0 \\ na_2 \\ 0 \end{pmatrix}, \ldots, b_n = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 2 \\ na_n \\ 0 \end{pmatrix}, \qquad b_{n+1} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \\ ns \\ 1 \end{pmatrix}$$

betrachtet.

Findet man einen Vektor $z = (z_1, \ldots, z_{n+2})^{\mathrm{T}} \in L(b_1, \ldots, b_{n+1})$ mit

$$\begin{aligned} |z_{n+2}| &= 1 \\ z_{n+1} &= 0 \\ z_1, \ldots, z_n &\in \{1, -1\}, \end{aligned}$$

so erhält man die Lösung des Rucksackproblems

$$x_i = |z_i - z_{n+2}|/2$$

für $i = 1, \ldots, n$.

Anhang D

Symbol- und Abkürzungsverzeichnis

det	Determinante (einer Matrix)
ggT	Größter gemeinsamer Teiler
HNF	Hermite-Normalform (einer Matrix)
log	Logarithmus
$\mathbf{N}$	Menge der natürlichen Zahlen
$\mathbf{N}_0$	Menge der natürlichen Zahlen mit der Null
$O(f)$	$\{g : \mathbf{N}_0 \to \mathbf{N}_0; \exists c > 0, \exists n_0 \quad \text{mit} \quad g(n) \leq cf(n) \quad \forall n \geq n_0\}$ für $f : \mathbf{N}_0 \to \mathbf{N}_0$
$\Omega(f)$	$\{g : \mathbf{N}_0 \to \mathbf{N}_0; \exists c > 0, \exists n_0 \quad \text{mit} \quad g(n) \geq cf(n) \quad \forall n \geq n_0\}$ für $f : \mathbf{N}_0 \to \mathbf{N}_0$
$\mathbf{Q}$	Menge der rationalen Zahlen
$\mathbf{R}$	Menge der reellen Zahlen
SNF	Smith-Normalform (einer Matrix)
$\mathbf{Z}$	Menge der ganzen Zahlen
$\lvert a \rvert$	Betrag der Zahl a
$\lVert v \rVert$	Euklidische Norm des Vektors v
$\leq_{\text{lex}}$	Lexikographisch kleiner oder gleich
$<_{\text{lex}}$	Lexikographisch echt kleiner
M^{T}	Transponierte Matrix der Matrix M
$u^{\text{T}}v$	Skalarprodukt (inneres Produkt) der Vektoren u und v
$M_{(i\ldots j)}$	Matrix, die aus den Spalten i bis j der Matrix M besteht
$v_{(i\ldots j)}$	Vektor, der aus den Komponenten i bis j des Vektors v besteht

$\lfloor a \rfloor$	Größte ganze Zahl $\leq a$
$\lceil a \rceil$	Kleinste ganze Zahl $\geq a$
$\lceil a \rfloor$	Nächste ganze Zahl zu a
$\leftarrow$	Zuweisungsoperator in einem Algorithmus
$\sum$	Summe
$\prod$	Produkt
$p \mid n$	p teilt n
$\emptyset$	Leere Menge

Literaturverzeichnis

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilerbau.* Addison-Wesley Publishing Company, 1988.

[2] S.G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall International Editions, 1989.

[3] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[4] J. Bartholomes. Gitterbasisreduktion im Reellen. Diplomarbeit, Fachbereich 14 - Informatik, Universität des Saarlandes, Saarbrücken, Deutschland, 1993.

[5] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation - Numerical Methods.* Prentice Hall International Editions, 1989.

[6] A. Bojanczyk, R.P. Brent, and H.T. Kung. Numerically stable solution of dense systems of linear equations using mesh-connected processors. *SIAM J. Sci. Stat. Comput.*, 5(1):95–104, 1984.

[7] Thomas Bräunl. *Parallele Programmierung – Eine Einführung.* Vieweg, 1993.

[8] I.N. Bronstein and K.A. Semendjajew. *Taschenbuch der Mathematik.* Verlag Harri Deutsch, 1985.

[9] J. Bu. *Systematic Design of Regular VLSI Processor Arrays.* PhD thesis, Delft University of Technology, 1990.

[10] J. Bu, L. Thiele, and E. Deprettere. Systolic array implementation of nested loop programs. In *Application Specific Array Processors, Princeton*, pages 31–43, IEEE Computer Society Press, Sept. 1990.

[11] J. Buchmann. Algorithmische Zahlentheorie. Vorlesungsskript WS 1994/95, Universität Saarbrücken.

[12] J. Buchmann and V. Müller. Einführung in die Computeralgebra. Vorlesungsskript WS 1991/92, Universität Saarbrücken.

[13] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[14] M. Cosnard and E.M. Daoudi. Optimal algorithms for parallel Givens factorization on a coarse-grained PRAM. *Journal of the Association for Computing Machinery*, 41(2):399–421, 1994.

[15] P. Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing, St. Malo*, pages 429–441, 1988.

[16] P. Feautrier. Parametric integer programming. *Recherche operationelle / Operations Research*, 22(3):243–268, 1988.

[17] P. Feautrier. Data flow analysis of array and scalar references. Technical Report RR89/282, MASI, 1989.

[18] P. Feautrier. Data flow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[19] G. E. Forsythe, M. A. Malcom, and C. B. Moler. *Computer Methods for Mathematical Computations*. Prentice-Hall, Englewood Cliffs, 1977.

[20] W. M. Gentleman. Error analysis of QR decompositions by Givens transformations. *Linear Algebra and Its Applications*, 10:189–197, 1975.

[21] W.M. Gentleman and Kung H.T. Matrix triangularization by systolic arrays. In *Proc SPIE Vol. 298 Real-Time Signal Processing IV*, pages 19–26, 1981.

[22] G.H. Golub and C.F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, Maryland, 1983.

[23] J. Götze and U. Schwiegelshohn. A square root and division free Givens rotation for solving least squares problems on systolic arrays. *SIAM J. Sci. Stat. Comput.*, 12(4):800–807, 1991.

[24] M. Grötschel, L. Lovasz, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, 1988.

[25] P. Häusser. Parallele Gitterbasisreduktion auf MIMD-Parallelrechnern. Diplomarbeit, Fachbereich 14 - Informatik, Universität des Saarlandes, Saarbrücken, Deutschland, in Vorbereitung.

[26] Intel Corporation. *Paragon User's Guide*, 1994.

[27] A. Joux. A fast parallel lattice reduction algorithm. In *Proc Second Gauss Symposium (Munich)*, 1993.

[28] A. Joux. *La Réduction des Réseaux en Cryptographie*. PhD thesis, Laboratoire d'Informatique de L'Ecole Normale Superieure LIENS, Paris, France, 1993.

[29] E. Kaltofen. On the complexity of finding short vectors in integer lattices. In *Proc EUROCAL83 (London), LNCS 162*, pages 236–244. Springer-Verlag, 1983.

[30] S. Kann. Ein Algorithmus zur Bestimmung linearer Datenabhängigkeiten in verschachtelten Schleifen. Studienarbeit am Lehrstuhl für Mikroelektronik, Universität des Saarlandes, Saarbrücken, Deutschland, 1994.

[31] D.E. Knuth. *The Art of Computer Programming*, volume 2 / Seminumerical Algorithms. Addison-Wesley Publishing Company, second edition, 1981.

[32] A. Kratz. Implementierung eines parallelen Gitterbasisreduktionsalgorithmus mit Hilfe des verteilten Systems LiPS. Diplomarbeit, Institut für Mikroelektronik, Universität des Saarlandes, Saarbrücken, Deutschland, 1994.

[33] R.H. Kuhn. *Optimization and Interconnection Complexity for: Parallel Processors, Single-Stage Networks, and Decision Trees.* PhD thesis, University of Illinois at Urbana-Champaign, 1980.

[34] F.T. Leighton. *Introduction to parallel algorithms and architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, 1992.

[35] A.K. Lenstra. Factoring polynomials over algebraic number fields. In *Proc EUROCAL83 (London), LNCS 162*, pages 245–254. Springer-Verlag, 1983.

[36] A.K. Lenstra, H.W. Lenstra Jr., and L. Lovasz. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.

[37] R.E. Lord, J.S. Kowalk, and S.P. Kumar. Solving linear algebraic equations on an MIMD computer. *J. of the Association for Computing Machinery*, 30(1):103–117, 1983.

[38] F.T. Luk. A rotation method for computing the QR-decomposition. *SIAM J. Sci. Stat. Comput.*, 7(2):452–459, 1986.

[39] D. E. Maydan, S. P. Amarainghe, and M. S. Lam. Array dataflow analysis and its use in array privatization. In *Proc ACM Conf. on Principles of Programming Languages*, pages 2–15. Springer-Verlag, January 1993.

[40] R.C. Merkle and M.E. Hellmann. Hiding information and signatures in trap-door knapsacks. *IEEE Trans. Inf. Theory*, 24:525–530, 1978.

[41] J.J. Modi. *Parallel Algorithms and Matrix Computation.* Oxford Applied Mathematics and Computing Science Series. Clarendon Press, London, 1988.

[42] F. Muller. Persönliches Gespräch.

[43] F. Muller. Massiv parallele Gitterbasisreduktion auf SIMD-Parallelrechnern. Diplomarbeit, Fachbereich 14 - Informatik, Universität des Saarlandes, Saarbrücken, Deutschland, in Vorbereitung.

[44] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley and Sons, 1988.

[45] T. Papanikolaou. *thalesII Manual.* Lehrstuhl Prof. Buchmann, Universität des Saarlandes, 1993.

[46] M. Pohst. A modification of the LLL reduction algorithm. *J. Symbolic Computation*, 4:123–127, 1987.

[47] C. D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, 1988.

[48] S. Radziszowski and D. Kreher. Solving subset sum problems with the L^3 algorithm. *J. Combin. Math. Combin. Comput.*, 3:49–63, 1988.

[49] H.B. Ribas. Obtaining dependence vectors for nested-loop computations. In *1990 International Conference on Parallel Processing*, pages II–212 – II–219, 1990.

[50] J.L. Roch and G. Villard. Parallel gcd and lattice basis reduction. In *Proc CONPAR92 (Lyon), LNCS 632*, pages 557–564. Springer-Verlag, 1992.

[51] R. Roth. LiPS: Ein System für verteilte Anwendungen . Diplomarbeit, Fachbereich 14 - Informatik, Universität des Saarlandes, Saarbrücken, Deutschland, 1992.

[52] R. Roth and T. Setz. LiPS: a system for distributed processing on workstations. Technical Report SFB 124 TP D5, Universität des Saarlandes, December 1992.

[53] A.H. Sameh. On Jacobi and Jacobi-like algorithms for a parallel computer. *Math. Comp.*, 25:579–590, 1971.

[54] A.H. Sameh and D.J.Kuck. On stable parallel linear solvers. *Journal of the Association for Computing Machinery*, 25(1):81–91, 1978.

[55] C.P. Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical Computer Science*, 53:201–224, 1987.

[56] C.P. Schnorr. A more efficient algorithm for lattice basis reduction. *J. Algorithms*, 9:47–62, 1988.

[57] C.P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. In *Proceedings of the FCT'91 (Gosen, Germany), LNCS 529*, pages 68–85. Springer, 1991.

[58] A. Schönhage. Factorization of univariate integer polynomials by diophantine approximation and an improved basis reduction algorithm. In *Proceedings of the 11th ICALP (Antwerp, Belgium), LNCS 172*, pages 436–447. Springer, 1984.

[59] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, 1986.

[60] U. Schwiegelshohn and L. Thiele. A systolic array for cyclic-by-rows Jacobi algorithms. *Journal of Parallel and Distributed Computing*, 4:334–340, 1987.

[61] T. Setz. Integration einer Linda-orientierten Laufzeitumgebung in LiPS. Diplomarbeit, Fachbereich 14 - Informatik, Universität des Saarlandes, Saarbrücken, Deutschland, November 1991.

[62] Finanzierungsantrag 1992 - 1993 - 1994. Sonderforschungsbereich 124, VLSI Entwurfsmethoden und Parallelität, Universitäten Kaiserslautern und Saarbrücken, 1991.

[63] Z. Shen, Z. Li, and P. Yew. An empirical study of fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, 1990.

[64] G. Strang. *Linear Algebra and its Application.* Harcourt Brace Jovanovich, Inc., third edition, 1986.

[65] J. Teich. *A Compiler for Application-Specific Processor Arrays.* Dissertation, Institut für Mikroelektronik, Universität des Saarlandes, Saarbrücken, Deutschland, September 1993.

[66] L. Thiele. Persönliches Gespräch.

[67] Thinking Machines Corporation. *C* Programming Guide*, 1993.

[68] Thinking Machines Corporation. *CM User's Guide*, 1993.

[69] Thinking Machines Corporation. *CMMD Reference Manual*, 1993.

[70] Thinking Machines Corporation. *CMSSL for C**, 1993.

[71] Thinking Machines Corporation. *CM-5 User's Guide*, 1994.

[72] G. Villard. Parallel lattice basis reduction. In *International Symposium on Symbolic and Algebraic Computation, Berkeley California USA*, pages 269–277. ACM Press, 1992.

[73] K. Wiese. Parallelisierung von LLL - Algorithmen zur Gitterbasisreduktion - Eine Implementierung auf dem Intel iPSC/860 Hypercube. Diplomarbeit, Fachbereich 14 - Informatik, Universität des Saarlandes, Saarbrücken, Deutschland, 1994.

[74] J. H. Wilkinson. *Rounding Errors in Algebraic Processes.* Prentice-Hall, Englewood Cliffs, 1963.

[75] J. H. Wilkinson. *The Algebraic Eigenvalue Problem.* Clarendon Press, Oxford, 1965.

[76] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, Massachusetts, 1989.