Lösen großer dünnbesetzter Gleichungssysteme über endlichen Primkörpern

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Technischen Fakultät
der Universität des Saarlandes
von

Thomas Friedrich Denny

Saarbrücken
1997

Mein Dank gebührt Herrn Prof. Dr. Buchmann für die Vergabe dieses interessanten Themas sowie die motivierende Betreuung während meiner gesamten akademischen Ausbildung.

Mein Dank gilt auch Arjen K. Lenstra, der sich nicht nur während meines Besuchs in den USA als kompetenter und hilfsbereiter Ansprechpartner erwiesen hat.

Des weiteren bedanke ich mich bei Volker Müller für das Korrekturlesen der Arbeit und die freundschaftliche Zusammenarbeit.

Außerdem bedanke ich mich bei Damian Weber und Jörg Zayer für viele anregende Diskussionen und die Bereitstellung interessanter Anwendungsbeispiele. Für die angenehme Arbeitsatmosphäre am Lehrstuhl bedanke ich mich bei allen, die dazu beigetragen haben.

Franz-Dieter Berger danke ich recht herzlich für die langjährige freundschaftliche Unterstützung und Hilfe in vielen Situationen.

Mein besonderer Dank gilt meinen Eltern und meinem Geschwistern, die mein Studium ermöglichten und mich nach besten Kräften unterstützt haben.

Tag des Kolloquiums:	30.10.1997
Dekan:	Prof. Dr.-Ing. A. Koch
Berichterstatter:	Prof. Dr. J. Buchmann
	Prof. Dr. K. Mehlhorn

Zusammenfassung

Die kryptographische Sicherheit der relevantesten Public Key Verfahren hängt von der Schwierigkeit des Faktorisierungsproblems bzw. des Diskreten Logarithmen (DL) Problems in endlichen Primkörpern ab. Sowohl bei allgemeinen Faktorisierungsverfahren als auch bei sogenannten *Index-Calculus*-Verfahren zur Bestimmung diskreter Logarithmen stellt das Lösen eines großen dünnbesetzten Gleichungssystems einen wichtigen Teil der Berechnung dar. In dieser Arbeit wird ein Gleichungslöser vorgestellt, der erfolgreich beim Lösen großer dünnbesetzter Gleichungssysteme über endlichen Primkörpern eingesetzt wurde. Er wurde beim Aufstellen des aktuellen Weltrekords (einer DL-Berechnung in $\mathbb{F}_p, p \approx 10^{85}$) eingesetzt und besteht aus 3 Teilen:

- einer Modifikation der Double Large Prime Variante des Number Field Sieve,
- einem Preprocessingschritt, der aufbauend auf Ideen der strukturierten Gaußelimination die Laufzeit des Lanczos Verfahrens minimiert und
- einer (sequentiellen bzw. parallelen) Implementierung des Lanczos Verfahrens.

Abstract

There are numerous crypto-systems whose security is based on the difficulty of factoring integers and solving the discrete logarithm (DL) problem in finite fields. One of the main computational problems that occur in general factoring and DL-algorithms is to solve a sparse huge system of linear equations over a finite field. In this thesis a solver that was successfully used in a computation of discrete logarithms in $\mathbb{F}_p, p \approx 10^{85}$ which is the current world record. It consists of three parts:

- A modification of the Double Large Prime Variation for the Number Field Sieve.
- A preprocessing step which minimizes the running time of the Lanczos algorithm. It is based on the ideas of the Structured Gaussian Elimination.
- A sequential and a parallel implementation of the Lanczos algorithm.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Einleitung

Die kryptographische Sicherheit der relevantesten Public Key Verfahren hängt von der Schwierigkeit des Faktorisierungsproblems (RSA-Kryptosystem [35]) bzw. des diskreten Logarithmen Problems (DL-Problem) in endlichen Primkörpern (Diffie Hellman Schlüsselaustauschverfahren [10], ElGamal-Kryptosystem [13], Digital Signature Standard (DSS) [31]) ab. Sowohl bei allgemeinen Faktorisierungsverfahren als auch bei sogenannten *Index-Calculus*-Verfahren zur Bestimmung diskreter Logarithmen (siehe [27]) stellt das Lösen eines großen dünnbesetzten Gleichungssystems einen wichtigen Teil der Berechnung dar. Die vorliegende Arbeit beschäftigt sich mit diesem wichtigen Bestandteil der obigen Algorithmen und legt dabei den Schwerpunkt auf Gleichungssysteme, die vom *Number Field Sieve* Algorithmus erzeugt werden. Dieses 1990 zum Faktorisieren entwickelte ([26],[43]) und 1993 für diskrete Logarithmen Probleme erweiterte Verfahren ([16],[36],[40]) stellt unter gewissen plausiblen Annahmen den asymptotisch schnellsten bekannten Algorithmus für beide Probleme dar.

Generell sind Faktorisierungsverfahren wesentlich besser untersucht als Verfahren zur Berechnung diskreter Logarithmen in endlichen Primkörpern. Dies gilt auch für das Lösen der auftretenden Gleichungssysteme. Die Dimension dieser Gleichungssysteme wächst mit der Größe der zu faktorisierenden Zahl bzw. mit der Charakteristik des Körpers, in dem DL-Probleme gelöst werden. Unabhängig von der Größe der zu faktorisierenden Zahl müssen die Gleichungssysteme bei allgemeinen Faktorisierungsverfahren nur über dem Körper mit zwei Elementen gelöst werden. Aufbauend auf der Arbeit von P. Montgomery [29] und von O. Groß [19] konnte eine sehr schnelle Implementierung des Block-Lanczos Verfahrens über $\mathbb{F}_2$ implementiert werden, die jedoch in dieser Arbeit nicht besprochen wird. Die Arbeit konzentriert sich daher auf das Lösen von Gleichungssystemen für diskrete Logarithmen Probleme. Im Gegensatz zu Faktorisierungsverfahren muß bei der Berechnung diskreter Logarithmen in $\mathbb{F}_p$ das auftretende Gleichungssystem über dem Ring $(\mathbb{Z}/(p-1)\mathbb{Z})$ gelöst

werden. Dazu löst man das System für alle Primteiler von $p-1$ und generiert mit Hilfe von *Hensel Lifting* (bei Primzahlpotenzen) und des *Chinesischen Restsatzes* die gesuchte Lösung. Die Tatsache, daß die Gleichungssysteme für mehrere Primzahlen, die wesentlich größer als zwei sind, berechnet werden muß, erschwert dieses Problem deutlich. Somit hat die Charakteristik des zugrunde liegenden Körpers bei DL-Problemen einen sehr großen Einfluß auf die Schwierigkeit des Lösens der auftretenden Gleichungssysteme. Vor allem bei der Berechnung diskreter Logarithmen hängt deshalb die Praktikabilität des eingesetzten Algorithmus stark von der Effizienz des verwendeten Gleichungslösers ab. Löst man DL-Probleme über Primkörpern $\mathbb{F}_p$ mit kleiner Charakteristik ($p < 10^{15}$), so benutzt man den Algorithmus von Silver-Pohlig-Hellmann (vgl. [27]), wobei kein Gleichungssystem gelöst werden muß. Löst man DL Probleme in Primkörpern $\mathbb{F}_p$ mit größerer Charakteristik ($p > 10^{15}$), so können die Lösungen für kleine Primteiler q von $p-1$ ($q < 10^{15}$) auch mit dem Verfahren von Silver-Pohlig-Hellmann berechnet werden. Aus diesem Grund beschäftigt sich diese Arbeit hauptsächlich mit der Lösung von Gleichungssystemen über endlichen Primkörpern $\mathbb{F}_p$ mit großer Charakteristik ($p > 10^{15}$).

Wendet man Standardverfahren wie z.B. die Gaußelimination an, so stößt man sehr schnell auf Speicherplatzprobleme. Zur Speicherung einer Matrix der Dimension 100 000 mit Elementen aus $\mathbb{F}_p$, wobei $p \in \mathbb{P}, p \approx 10^{100}$, benötigt man ungefähr 125 Gigabyte Speicherplatz. Eine Variante der Gaußelimination, die sogenannte *strukturierte Gaußelimination* (vgl. [33],[32]), nutzt die Dünnbesetztheit der Matrizen aus. Sie reduziert die Dimension des ursprünglichen Systems auf ca. ein Drittel und somit den Speicherplatzbedarf auf ca. ein Zehntel des ursprünglichen Wertes. Beispiele in der obigen Größenordnung sind aber auch damit nicht zu lösen. Coppersmith und Odlyzko [32] haben um 1984 vorgeschlagen sogenannte *Krylow Subspace* Verfahren zur Lösung dieser Systeme zu verwenden. Sie benötigen im wesentlichen nur die Speicherung der dünnbesetzten Matrix sowie einiger Vektoren. Dabei handelt es sich unter anderem um das Lanczos Verfahren und das konjugierte Gradienten (CG) Verfahren ([24],[4],[32]). Beides sind Iterationsverfahren, die zur Bestimmung der Eigenwerte einer reellen Matrix und zur Lösung von reellen Gleichungssystemen um 1952 entwickelt wurden und eine in etwa vergleichbare Anzahl von Operationen benötigen. In der Literatur wird das Lanczos Verfahren vorgezogen. Bei der Anwendung beider Verfahren auf endliche Primkörper treten einige Probleme auf (vgl. [23]), deren Lösung jedoch zu keinen wesentlichen Veränderungen der Algorithmen führt. Diese Probleme treten bei dem Wiedemann Verfahren [42] (einem weiteren Krylow Subspace Algorithmus) nicht auf. Dieses Verfahren wurde 1986 speziell für endliche Primkörper entwickelt. Die Anzahl der benötigten Operationen ist hier jedoch deutlich höher als beim Lanczos oder CG-Verfahren.

Da das Lanczos Verfahren am schnellsten die Lösung der Gleichungssysteme berechnet, bildet es die Grundlage des in dieser Arbeit entwickelten Gleichungslösers. Die Anzahl der benötigten Operationen sowie der Speicherplatzbedarf des Lanczos Verfahrens konnte durch eine Verbesserung des Verfahrens reduziert werden. Aufbauend auf den Ideen der strukturierten Gaußelimination wird ein Preprocessingschritt entwickelt, der die Laufzeit des dann folgenden Lanczos Algorithmus um bis zu 80%

reduziert. Trotzdem erfordert die rechen- und speicherplatzintensive Anwendung die Nutzung eines Parallelrechners. Eine Parallelisierung des Lanczos Verfahrens, die fast linearen Speedup hat, legt den Grundstein für den erfolgreichen Einsatz des in dieser Arbeit entwickelten Gleichungslösers. Die Bemühungen zum Beschleunigen der Lösung der auftretenden Gleichungssysteme setzen bereits bei deren Generierung ein. Eine Modifikation der *Siebphase* (der ersten Phase der Index Calculus und der allgemeinen Faktorisierungsverfahren), die sogenannten *Double Large Prime* Variante (vgl. [25], [43]), führte 1993 zu einer deutlichen Beschleunigung dieser Phase. Gleichzeitig haben die mit dieser Variante generierten Gleichungssysteme jedoch deutlich mehr Einträge. Durch eine hier vorgestellte Verbesserung dieser Modifikation kann die Anzahl der Einträge wiederum bis zu 30% reduziert werden. Der entwickelte Gleichungslöser setzt sich also insgesamt aus drei Teilen zusammen:

- der Modifikation der Double Large Prime Variante (Generierung der Systeme),
- dem Preprocessingschritt, der aufbauend auf Ideen der strukturierten Gaußelimination die Laufzeit des Lanczos Verfahrens minimiert und
- der (sequentiellen bzw. parallelen) Implementierung des Lanczos Verfahrens.

Der Gleichungslöser wurde sowohl bei Faktorisierungsbeispielen von Jörg Zayer, der eine der weltweit ersten Number Field Sieve Implementierungen erstellt hat (vgl. [43]), als auch in DL Berechnungen von Damian Weber (vgl. [40]) eingesetzt. Aufbauend auf der Arbeit von Jörg Zayer hat Damian Weber eine COS-Implementierung (ein Index Calculus Algorithmus vgl. [4]) und die weltweit einzige Implementierung des Number Field Sieve für DL erstellt. Zusammen mit Damian Weber (vgl. [40], [9], [41]) konnte der Weltrekord von Odlyzko und LaMacchia von 1991 bei DL Berechnungen (Berechnung in $\mathbb{F}_p, p \approx 10^{58}$) mehrfach verbessert werden. Dabei betrug die Charakteristik des größten Körpers, in dem DL Berechnungen von uns durchgeführt wurden (also der aktuelle Weltrekord), ungefähr 10^{85}.

Die Arbeit gliedert sich in 6 Kapitel. Nach der Einleitung folgt ein Abschnitt mit den wichtigsten mathematischen Notationen und Grundlagen.
Im zweiten Kapitel wird dann das Lanczos Verfahren beschrieben. Der Herleitung der Idee folgt die Beschreibung für reelle Systeme. Dann wird die Anpassung des Algorithmus an endliche Primkörper beschrieben. Die Laufzeitanalyse einer verbesserten Variante gibt Aufschluß über die Faktoren, die die Laufzeit des Lanczos Verfahrens beeinflußen.
Mit der Struktur und den Eigenschaften der betrachteten Systeme beschäftigt sich Kapitel 3. Nach einer genauen Untersuchung ihrer Herkunft werden einige Eigenschaften der Systeme daraus abgeleitet.
Die Modifikation der Double Large Prime Variante, die bereits bei der Generierung der Systeme auf deren gute Lösbarkeit achtet, wird in Kapitel 4 beschrieben. Die Idee und deren Umsetzung in einen effizienten Algorithmus schließt sich an. Zahlreiche praktische Beispiele aus Faktorisierungs- und DL-Beispielen bilden den Abschluß dieses Kapitels.

Das fünfte Kapitel beschäftigt sich mit der Implementierung der sequentiellen und der parallelen Version des Lanczos Verfahrens sowie den damit verbundenen Fragen und Problemen. Dabei beschreibe ich noch eine Idee zur Beschleunigung der Berechnung der Lösungen, bevor eine genaue Laufzeitanalyse des Lanczos Verfahrens durchgeführt wird. Aus dieser Laufzeitanalyse entwickele ich ein Modell das Laufzeitvorhersagen ermöglicht und die Auswirkungen von Veränderungen des Systems auf die Laufzeit bestimmen kann.
Im abschließenden sechsten Kapitel wird der Kompaktifizierer beschrieben. Nachdem in Kapitel 5 das Modell zur Laufzeitvorhersage entwickelt wurde, können die Ideen der strukturierten Gaußelimination zur Minimierung der Laufzeit des Lanczos Verfahrens benutzt werden. Nach der Beschreibung der Idee und deren Umsetzung werden die praktischen Ergebnisse beschrieben. Das Fazit meiner Arbeit sowie ein Anhang mit einem Programm für Berechnungen im entwickelten Laufzeitmodell und zum Aufbau einer speziellen Kommunikationsstrukur bilden den Abschluß der Arbeit.

1.2 Mathematische Grundlagen

An dieser Stelle möchte ich Bezeichnungen einführen, die im weiteren Verlauf der Arbeit benutzt werden, dort aber nicht mehr explizit definiert werden. Als Quelle dienen hauptsächlich die Bücher von Lamprecht [22] und Golub und Loan [15].

Ich verwende zur Bezeichnung von Zahlbereichen folgende Symbole:

$\mathbb{N}$	die Menge der natürlichen Zahlen,
$\mathbb{P}$	die Menge der Primzahlen,
$\mathbb{Z}$	der Ring der ganzen Zahlen,
$\mathbb{Z}/n\mathbb{Z}$	der Restklassenring modulo n,
$(\mathbb{Z}/n\mathbb{Z})^*$	die multiplikative Gruppe des Restklassenringes modulo n,
$\mathbb{R}$	der Körper der reellen Zahlen,
$\mathbb{F}_p$	der Primkörper mit p Elementen, wobei $p \in \mathbb{P}$,
$\mathbb{K}$	ein beliebiger Körper.

Definition 1.1 (Matrix, Zeilenvektor, Spaltenvektor)

Ein rechteckiges Schema

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} = (a_{ij})_{\substack{1 \le i \le m \\ 1 \le j \le n}}$$

von Elementen aus einer beliebigen Menge M heißt eine **Matrix** *über M oder genauer eine $m \times n$-Matrix, das heißt eine Matrix mit m Zeilen und n Spalten. Eine*

$m \times 1$-Matrix

$$A = \begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix}$$

heißt auch **Spaltenvektor**. *Eine $1 \times n$-Matrix*

$$A = (a_{11}a_{12} \ldots a_{1n}) = (a_1, a_2, \ldots, a_n)$$

heißt auch **Zeilenvektor**.

Bemerkung 1.2

- Während der gesamten Arbeit handelt es sich, sofern nicht anders angegeben, bei Vektoren immer um Spaltenvektoren.
- Um eine $m \times n$-Matrix A als Konkatenation von n Spaltenvektoren

$$a_i = \begin{pmatrix} a_{1i} \\ a_{2i} \\ \vdots \\ a_{mi} \end{pmatrix} \quad (\, i \,=\, 1,\, \cdots,\, n\,)$$

zu definieren, schreiben wir

$$A = [a_1, \cdots, a_n].$$

Definition 1.3 (Matrizenmultiplikation)

Sind $m, n, r \in \mathbb{N}$ und $A \in \mathbb{K}^{m\times n}, B \in \mathbb{K}^{n\times r}$, so nennt man die gemäß

$$\begin{aligned} \mathbb{K}^{m\times n} \times \mathbb{K}^{n\times r} &\to \mathbb{K}^{m\times r} \\ (A, B) &\mapsto A \cdot B = C = (c_{ij}),\ \textit{mit} \\ & \qquad c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}, \quad (i = 1, \ldots, m;\ j = 1, \ldots, r). \end{aligned}$$

gebildete $m \times r$-Matrix C das **Produkt** *von A mit B.*

Definition 1.4 (Skalarprodukt)

Die Abbildung

$$\begin{aligned} \langle\, , \rangle : \mathbb{K}^n \times \mathbb{K}^n &\to \mathbb{K} \\ (x, y) &\mapsto \langle x, y\rangle = \sum_{k=1}^{n} x_k y_k = x^T y \\ & \qquad \textit{für } x = (x_1, \ldots, x_n)^T, y = (y_1, \ldots, y_n)^T \end{aligned}$$

heißt das **Skalarprodukt** *(bzw. innere Produkt) in $\mathbb{K}^n$.*

Bemerkung 1.5 [Vektor-Matrix-Multiplikation]

Ein Vektor kann als einzeilige beziehungsweise einspaltige Matrix aufgefaßt werden. Somit kann man mit Hilfe der Matrizenmultiplikation das Produkt zweier Vektoren (vgl. Skalarprodukt) sowie eines Vektors mit einer Matrix definieren. Sei $v \in \mathbb{K}^n$ ein Zeilenvektor ($v \in \mathbb{K}^{1\times n}$) und sei $M \in \mathbb{K}^{n\times r}$. Dann ist

$$w = v \cdot M$$

ein Zeilenvektor aus $K^{1\times r}$. Sei $v \in \mathbb{K}^n$ ein Spaltenvektor ($v \in \mathbb{K}^{n\times 1}$) und sei $M \in \mathbb{K}^{r\times n}$. Dann ist

$$w = M \cdot v$$

ein Spaltenvektor aus $\mathbb{K}^{r\times 1}$.

Definition 1.6 (Gewicht einer Matrix)

Das **Gewicht** ω *einer Matrix* $A = (a_{ij})$ *wird definiert als die Anzahl der von Null verschiedenen Einträge in* A*, also*

$$\omega \;=\; |\,\{\,a_{ij}\,|\,a_{ij} \neq 0\,\}\,| \;.$$

Diese Definition wird auch analog auf Vektoren angewendet.

Bemerkung 1.7 [Bedeutung des Gewichtes]

- Gilt für das Gewicht ω einer Matrix $A \in \mathbb{K}^{m\times n}, \omega \leq \min\{\sqrt{n}\cdot m, \sqrt{m}\cdot n\}$, so spreche ich von einer dünnbesetzten Matrix A.
- Durch das Gewicht ω einer Matrix $A \in \mathbb{K}^{m\times n}$ ist auch die Anzahl der Körperoperationen bei einer Matrix-Vektor-Multiplikation Ab für einen n-dimensionalen Spaltenvektor b bestimmt.

Definition 1.8 (obere und untere Dreiecksmatrix, Diagonalmatrix)

Eine quadratische Matrix A*, für deren Elemente* $a_{ij} = 0$ *für alle* $i > j$ *gilt, heißt* **obere Dreiecksmatrix***; gilt* $a_{ij} = 0$ *für alle* $i < j$*, so heißt* A **untere Dreiecksmatrix**. *Ist* $a_{ij} = 0$ *für alle* $i \neq j$*, so heißt* A **Diagonalmatrix**.

Definition 1.9 (Transponierte)

Sei

$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix}$$

eine $m \times n$-Matrix. Die **Transponierte** *von A ist die $n \times m$-Matrix*

$$A^T = \begin{pmatrix} a_{11} & \dots & a_{m1} \\ \vdots & & \vdots \\ a_{1n} & \dots & a_{mn} \end{pmatrix}.$$

Bemerkung 1.10 [Eigenschaften]

- Für eine Matrix $A = (a_{ij}) \in \mathbb{K}^{n\times n}$ gilt:

$$\langle x, Ay\rangle = \sum_{l,k=1}^{n} x_l\, a_{lk}\, y_k = \sum_{l=1}^{n} x_l \sum_{k=1}^{n} a_{lk}\, y_k = \sum_{k=1}^{n} y_k \sum_{l=1}^{n} a_{lk}\, x_l = \langle A^T x, y\rangle$$

- Gilt $A = A^T$ $(a_{lk} = a_{kl}\ \forall k, l)$ so sagt man: A ist **symmetrisch.**
- Für eine Matrix $B \in \mathbb{K}^{m\times n}$ ist die Matrix $B^T B \in \mathbb{K}^{n\times n}$ symmetrisch.

Definition 1.11 (A-orthogonal, A-konjugiert)

Für eine symmetrische Matrix $A \in \mathbb{K}^{n\times n}$ heißen zwei Spaltenvektoren $x, y \in \mathbb{K}^n$ **A-orthogonal** *(bzw.* **A-konjugiert***), falls $x^T Ay = 0$.*

Bemerkung 1.12 [Schreibweise linearer Gleichungssysteme]

Ein lineares Gleichungssystem

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & = & b_1 \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \cdots & + & a_{mn}x_n & = & b_m \end{array}$$

kann geschrieben werden als

$$\bigvee_{i=1,\cdots,m} \quad \sum_{j=1}^{n} a_{ij} \cdot x_j = b_i$$

oder auch als

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \text{ bzw. als } Ax = b.$$

Ist $b = 0$ ($b_i = 0$ für $i = 1, \cdots, m$), so spricht man von einem **homogenen** Gleichungssystem. Sonst redet man von einem **inhomogenen** Gleichungssystem.

Definition 1.13 (Krylow-Matrix, Krylow-Unterraum)

Für eine symmetrische Matrix $A \in \mathbb{K}^{n\times n}$ und einen Vektor $q \in \mathbb{K}^n$ ist die Krylow-Matrix $K(A,q,n)$ definiert durch

$$K(A,q,n) = [q, Aq, A^2q, \cdots, A^{n-1}q] \ .$$

Für ein $j \in \mathbb{N}$ bezeichnet man den von den ersten j Spalten der Krylow-Matrix aufgespannten Unterraum als Krylow-Unterraum

$$\mathcal{K}(A,q,j) = span\{q, Aq, A^2q, \cdots, A^{j-1}q\} \ .$$

Definition 1.14 (O-Notation)

Für zwei Funktionen $f, g : \mathbb{N} \to \mathbb{R}$ sagen wir:

$$f = O(g) \quad \Longleftrightarrow \quad \underset{n_0,c\in\mathbb{N}}{\exists} \ \underset{n\geq n_0}{\forall} \ f(n) \leq c \cdot g(n),$$

$$f = o(g) \quad \Longleftrightarrow \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \ .$$

Kapitel 2

Der Lanczos Algorithmus

In diesem Kapitel werde ich den Lanczos Algorithmus genauer vorstellen und eine Laufzeitabschätzung durchführen. Aufbauend auf [15] und [29] wird zuerst eine Herleitung der Idee vorgestellt. Ausgehend von der ursprünglichen Beschreibung für den Körper der reellen Zahlen (vgl. [24]) wird dann die Anpassung für endliche Körper (vgl. [23],[32]) beschrieben. Die dabei auftretenden Probleme und deren Lösung werden diskutiert. Es wird eine Verbesserung des bekannten Algorithmus beschrieben und eine Laufzeitanalyse durchgeführt.

Im folgenden gehe ich immer davon aus, daß ein lineares Gleichungssystem

$$A\ x = b \tag{2.1}$$

gelöst werden soll, wobei x ein gesuchter n-dimensionaler Vektor, A eine symmetrische $n \times n$ Matrix und b ein gegebener n-dimensionaler Spaltenvektor ist.

2.1 Herleitung der Idee

Angenommen, man findet eine Matrix $W = [w_0, \cdots, w_{n-1}]$ derart, daß $W^T A W = D$ gilt, wobei D eine Diagonalmatrix mit vollem Rang ist. Dann führt die Lösung des Gleichungssystems $Dy = W^T b$ zur Lösung von (2.1). Es gilt nämlich

$$Dy = W^T b \quad \Longleftrightarrow \quad W^T A W y = W^T b$$

und aus

$$Ax = b \quad \Longleftrightarrow \quad W^T A x = W^T b$$

folgt, daß $x = Wy$ eine Lösung von $Ax = b$ ist. Da D eine Diagonalmatrix ist, kann man die Lösung von $Dy = W^T b$ sehr schnell berechnen. Somit ist das Problem, eine Lösung von (2.1) zu bestimmen, auf das Problem der Bestimmung von W zurückgeführt. Beim Lanczos Algorithmus benutzt man die lineare Unabhängigkeit

der Spaltenvektoren der Krylow-Matrix $K(A,b,n)$, um daraus mit Hilfe einer modifizierten Version der Gram-Schmitt Orthogonalisierung eine A-orthogonale Basis zu konstruieren. Diese Basis baut dann die gesuchte Matrix $W = [w_0, \cdots, w_{n-1}]$ auf. Man setzt $w_0 = b$ und iteriert solange

$$w_i = Aw_{i-1} - \sum_{j=0}^{i-1} c_{ij} w_j, \qquad \text{wobei } c_{ij} = \frac{w_j^T A^2 w_{i-1}}{w_j^T A w_j}, \tag{2.2}$$

bis $w_m = 0$. Dabei ist zu beachten, daß (vgl. [29], §3)

$$w_j^T A w_i = 0 \quad (i \neq j) \qquad \text{gilt.} \tag{2.3}$$

Unter Benutzung von (2.3) kann man zeigen (vgl. [29], §3), daß für $j < i-2$ gilt:

$$\begin{aligned} w_j^T A^2 w_{i-1} &= (Aw_j)^T A w_{i-1} \\ &= \left(w_{j+1} + \sum_{k=0}^{j} c_{j+1,k} w_k \right)^T A w_{i-1} = 0 \end{aligned}$$

und damit auch $c_{ij} = 0$ für $j < i-2$.

Somit vereinfacht sich die Formel (2.2) zur Berechnung der w_i für $i \geq 2$:

$$w_i = Aw_{i-1} - c_{i,i-1} w_{i-1} - c_{i,i-2} w_{i-2} \ .$$

Stoppt die Iteration mit $w_m = 0$ ($w_i \neq 0$, für $0 \leq i < m$), dann gilt:

$$x = \sum_{j=0}^{m-1} \frac{w_j^T b}{w_j^T A w_j} w_j \tag{2.4}$$

ist die gesuchte Lösung von (2.1) ist. Der Beweis folgt ([29], §4).

Beweis: Sei $\mathcal{W} = < w_0, \cdots, w_{m-1} >$ der von den Iterationsvektoren aufgespannte Unterraum. Nach Konstruktion der Iterationsvektoren gilt:

$$A\mathcal{W} \subseteq \mathcal{W}, \tag{2.5}$$

$$Ax - b \in \langle w_0 (= b), Aw_0, Aw_1, \cdots Aw_{m-1} \rangle \subseteq \mathcal{W} \ . \tag{2.6}$$

Für alle $l = 0, \cdots, m-1$ gilt:

$$w_l^T A x \ = \ w_l^T A \left(\sum_{j=0}^{m-1} \frac{w_j^T b}{w_j^T A w_j} w_j \right) \overset{(2.3)}{=} w_l^T A \frac{w_l^T b}{w_l^T A w_l} w_l \ = \ w_l^T b \ . \tag{2.7}$$

Daraus folgt $\mathcal{W}^T (Ax - b) = \{0\}$. Aus der Symmetrie von A und (2.5) folgt:

$$\mathcal{W}^T A (Ax - b) = (A\mathcal{W})^T (Ax - b) \ \subseteq \ \mathcal{W}^T (Ax - b) = \{0\} \ .$$

Wegen (2.6) gibt es eine Darstellung

$$Ax - b = \sum_{i=0}^{m-1} c_i w_i \ .$$

Für alle $l = 0, \cdots m-1$ gilt wegen (2.7):

$$0 = w_l^T A\,(Ax - b) = w_l^T A \left(\sum_{i=0}^{m-1} c_i w_i \right) \overset{(2.3)}{=} w_l^T A\, c_l\, w_l \ .$$

Da für $0 \leq l < m$ gilt $w_l^T A w_l \neq 0$, folgt $c_l = 0$ und somit

$$Ax - b = 0 \ .$$

■

Man beachte, daß man die Lösung (2.4) bereits während der schrittweisen Berechnung der w_j partiell aufaddieren kann, da dazu in jeder Iteration nur w_j gebraucht wird.

2.2 Beschreibung des Lanczos Algorithmus

Nach der Herleitung des Verfahrens beschreibe ich nun den Algorithmus, wie er von C. Lanczos [24] für lineare Gleichungssysteme mit reellen Koeffizienten vorgestellt wurde. Die Beschreibung ist so angelegt, daß sie direkt zu einer effizienten Implementierung des Verfahrens führt.

Man setzt

$$w_0 = b \ , \tag{2.8}$$

$$v_1 = A\, w_0 \ , \tag{2.9}$$

$$w_1 = v_1 - \frac{\langle v_1, v_1 \rangle}{\langle w_0, v_1 \rangle}\, w_0 \tag{2.10}$$

und iteriert für $i \geq 1$ nach folgender Vorschrift

$$v_{i+1} = A\, w_i \ , \tag{2.11}$$

$$w_{i+1} = v_{i+1} - \frac{\langle v_{i+1}, v_{i+1} \rangle}{\langle w_i, v_{i+1} \rangle}\, w_i - \frac{\langle v_{i+1}, v_i \rangle}{\langle w_{i-1}, v_i \rangle}\, w_{i-1} \ . \tag{2.12}$$

Der Algorithmus stoppt, wenn ein w_m gefunden wird, das A-konjugiert zu sich selber ist ($\langle w_m, A w_m \rangle = 0$). Dies passiert für ein $m \leq n$. Falls $w_m = 0$ ist, dann ist

$$x = \sum_{i=0}^{m-1} \frac{\langle w_i, b \rangle}{\langle w_i, v_{i+1} \rangle}\, w_i \tag{2.13}$$

eine Lösung von Gleichung (2.1). Wird solch ein w_m nicht gefunden, dann liegt b nicht in dem von den Spalten von A erzeugten Unterraum.
Verwendet man den Lanczos Algorithmus zum Lösen linearer Gleichungssysteme über endlichen Primkörpern, so überträgt man die Formeln (2.9) - (2.13) auf die Situation in endlichen Körpern. Dabei muß man einige Probleme lösen:

- In endlichen Körpern folgt (im Gegensatz zur Situation in $\mathbb{R}$, wo A positiv definit ist) aus $w_j^T A w_j = 0$ nicht notwendig, daß $w_j = 0$ ist. Löst man Gleichungssysteme über Primkörpern $\mathbb{F}_p$ mit großer Charakteristik ($p >> n$), so stellt die sogenannte Selbstkonjugiertheit kein Problem dar. Mögliche Lösungen für dieses Problem in Körpern mit kleiner Charakteristik finden sich in [29] und [23].

- Ich bin an der Lösung von homogenen Gleichungssystemen interessiert. Da der Lanczos Algorithmus nur inhomogene Systeme lösen kann, wählt man folgendes Vorgehen:
  Angenommen, man will $Bx = 0$ lösen, wobei $B = [b_1, \cdots, b_n]$ und x ein n-dimensionaler Spaltenvektor ist. Dann löst man mit Hilfe des Lanczos Verfahrens $B'x' = b_n$, wobei $B' = [b_1, \cdots, b_{n-1}]$ und x' ein $(n-1)$-dimensionaler Spaltenvektor ist. Eine Lösung des homogenen Systems $Bx = 0$ erhält man durch $x_i = x'_i$ $(i = 1, \cdots, n-1)$ und $x_n = -1$, da aus

$$\begin{pmatrix} b_{11} & \cdots & b_{1,n-1} \\ \vdots & & \vdots \\ b_{m1} & \cdots & b_{m,n-1} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} b_{1n} \\ \vdots \\ b_{mn} \end{pmatrix}$$

  folgt

$$\begin{pmatrix} b_{11} & \cdots & b_{1,n-1} & b_{1n} \\ \vdots & & \vdots & \vdots \\ b_{m1} & \cdots & b_{m,n-1} & b_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_{n-1} \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix} .$$

- Die Gleichungssysteme, deren Lösung bei Faktorisierungen und Berechnung von diskreten Logarithmen benötigt werden, sind in der Regel nicht symmetrisch, jedoch dünnbesetzt. Gesucht wird eine Lösung für $Bx = u$, wobei x ein n-dimensionaler Vektor, B eine dünnbesetzte $m \times n$ Matrix ($n \geq m$) Matrix und b ein gegebener m-dimensionaler Vektor ist. Dabei muß u in dem von den Spalten von B erzeugten Unterraum liegen, damit eine Lösung gefunden wird. In der Praxis wird dann $A = B^T B$ und $b = B^T u$ gesetzt und eine Lösung für (2.1) gesucht. Diese Lösung ist dann auch eine Lösung von $Bx = u$ (vgl. [23]).

- Mit dem Lanczos Verfahren kann man nicht nur eine Lösung $Bx = u$, sondern auch mehrere Lösungen $Bx_j = u_j, \quad 1 \leq j \leq r$, berechnen. In [23] wird ein Verfahren beschrieben, das dies ermöglicht. Man berechnet $z_j = B^T u_j$ und startet den Algorithmus mit $w_0 = z_1$. Zur Berechnung der r verschiedenen Lösungen genügt es Schritt (2.13) durch

$$x_j = \sum_{l=0}^{m-1} \frac{\langle w_l, z_j \rangle}{\langle w_l, v_{l+1} \rangle} \, w_l \ , \quad 1 \leq j \leq r$$

zu ersetzen. Einen Beweis dafür findet sich in [23].

Beispiel:

Es wird in obiger Notation eine Lösung für:

$$Bx = \begin{pmatrix} 1 & 3 & -1 & 1 & 1 & 1 \\ 2 & 4 & 2 & 1 & -1 & -1 \\ 3 & 1 & -1 & -1 & 2 & 1 \\ 4 & -1 & 4 & 3 & -2 & -1 \\ 5 & -2 & -1 & 2 & 3 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} = u$$

in $\mathbb{F}_{29}$ gesucht. Der Lanczos Algorithmus löst:

$$B^T Bx = \begin{pmatrix} 26 & 0 & 11 & 22 & 12 & 3 \\ 0 & 2 & 2 & -1 & -3 & -1 \\ 11 & 2 & 23 & 12 & 13 & 20 \\ 22 & -1 & 12 & 16 & -2 & -2 \\ 12 & -3 & 13 & -2 & 19 & 9 \\ 3 & -1 & 20 & -2 & 9 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} 9 \\ 2 \\ -3 \\ 2 \\ 6 \\ 3 \end{pmatrix} = B^T u \ .$$

Dabei wird $W = [w_0, \cdots, w_4]$ berechnet als

$$W = \begin{pmatrix} 9 & 20 & 2 & 6 & 0 \\ 2 & 23 & 17 & 9 & 16 \\ 26 & 8 & 15 & 5 & 18 \\ 2 & 19 & 6 & 6 & 14 \\ 6 & 25 & 8 & 2 & 3 \\ 3 & 15 & 18 & 21 & 13 \end{pmatrix} \ .$$

Dann gilt

$$W^T B^T BWy = \begin{pmatrix} 19 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 15 & 0 & 0 \\ 0 & 0 & 0 & 28 & 0 \\ 0 & 0 & 0 & 0 & 24 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix} = \begin{pmatrix} 27 \\ 0 \\ 5 \\ 28 \\ 5 \end{pmatrix} = W^T B^T u \ .$$

Somit findet man als Lösungen

$$y = \begin{pmatrix} 6 \\ 0 \\ 10 \\ 1 \\ 28 \end{pmatrix} \quad \text{und damit} \quad Wy = x = \begin{pmatrix} 22 \\ 1 \\ 3 \\ 6 \\ 28 \\ 3 \end{pmatrix} .$$

2.3 Laufzeitabschätzung und Platzbedarf

Ich untersuche den Algorithmus zuerst so, wie er bisher in der Literatur beschrieben wird und in Abschnitt 2.2 vorgestellt wurde. Man stellt fest, daß in der i-ten Iteration ($i \geq 2$) die Vektoren $w_{i-1}, w_{i-2}, v_{i+1}$ und v_i sowie für jede zu berechnende Lösung der Vektor z_j und der Vektor x_j für die partiell aufaddierte Lösung benötigt wird. (Bei der Berechnung von w_i kann der Speicherbereich von w_{i-2} überschrieben werden.) Aus der vorherigen Iteration ist bereits der Vektor v_i sowie das innere Produkt $\langle w_{i-1}, v_i \rangle$ bekannt. Neu berechnet werden müssen die Vektoren v_{i+1} und w_{i+1}. Bei dieser Berechnung werden die 3 inneren Produkte $\langle v_{i+1}, v_{i+1} \rangle$, $\langle w_i, v_{i+1} \rangle$ und $\langle v_{i+1}, v_i \rangle$ benötigt.

Nutzt man die paarweise A-Orthogonalität der Vektoren w_i, w_j $(i \neq j)$ (vgl. (2.3)) aus, so kann man den Platzbedarf und die Anzahl der benötigten Operationen herabsetzen. Dazu benutzt man den folgenden Satz:

Satz 2.1 *In der Notation von Abschnitt 2.2 gilt:*

$$\forall\ i \geq 1 : \quad \langle v_{i+1}, v_i \rangle = \langle v_{i+1}, w_i \rangle \ .$$

Beweis:

Für $i = 1$ ist zu zeigen: $\langle v_2, w_1 \rangle = \langle v_2, v_1 \rangle$. Benutzt man die Symmetrie von A, so gilt:

$$\begin{aligned}
\langle v_2, w_1 \rangle &= v_2^T\ w_1 \ = \ (A\ w_1)^T\ w_1 \ = \ w_1^T\ A\ w_1 \\
&= w_1^T\ A\ \left(v_1 - \frac{\langle v_1, v_1 \rangle}{\langle w_0, v_1 \rangle}\ w_0 \right) \\
&= w_1^T\ A\ v_1 - \frac{\langle v_1, v_1 \rangle}{\langle w_0, v_1 \rangle}\ \left(\underbrace{w_1^T\ A\ w_0}_{=0} \right) \\
&= (A\ w_1)^T\ v_1 \ = \ v_2^T\ v_1 \\
&= \langle v_2, v_1 \rangle \ .
\end{aligned}$$

Für $i > 1$ erhalten wir auf ähnliche Weise

$$\begin{aligned}
\langle v_{i+1}, w_i \rangle &= v_{i+1}^T\, w_i \;=\; (A\, w_i)^T\, w_i \;=\; w_i^T\, A\, w_i \\
&= w_i^T\, A \left(v_i - \frac{\langle v_i, v_i \rangle}{\langle w_{i-1}, v_i \rangle}\, w_{i-1} - \frac{\langle v_i, v_{i-1} \rangle}{\langle w_{i-2}, v_{i-1} \rangle}\, w_{i-2} \right) \\
&= w_i^T\, A\, v_i - \frac{\langle v_i, v_i \rangle}{\langle w_{i-1}, v_i \rangle} \left(\underbrace{w_i^T\, A\, w_{i-1}}_{=0} \right) - \frac{\langle v_i, v_{i-1} \rangle}{\langle w_{i-2}, v_{i-1} \rangle} \left(\underbrace{w_i^T\, A\, w_{i-2}}_{=0} \right) \\
&= (A\, w_i)^T\, v_i \;=\; v_{i+1}^T\, v_i \\
&= \langle v_{i+1}, v_i \rangle
\end{aligned}$$

■

Man spart somit die Berechnung eines inneren Produktes $\langle v_{i+1}, v_i \rangle$ und der Vektor v_i wird nicht mehr in der i-ten Iteration gebraucht. Dadurch reduziert sich der Speicherplatzbedarf um einen Vektor. (Man beachte dabei, daß $\langle w_{i-1}, v_i \rangle$ aus der vorherigen Iteration bekannt ist.) Somit vereinfachen sich die Formeln (2.9) - (2.13) zu

$$w_0 = b\ , \tag{2.14}$$

$$v_1 = A\, w_0\ , \tag{2.15}$$

$$w_1 = v_1 - \frac{\langle v_1, v_1 \rangle}{\langle w_0, v_1 \rangle}\, w_0 \tag{2.16}$$

und für $i \geq 1$

$$v_{i+1} = A\, w_i\ , \tag{2.17}$$

$$w_{i+1} = v_{i+1} - \frac{\langle v_{i+1}, v_{i+1} \rangle}{\langle w_i, v_{i+1} \rangle}\, w_i - \frac{\langle w_i, v_{i+1} \rangle}{\langle w_{i-1}, v_i \rangle}\, w_{i-1}\ , \tag{2.18}$$

$$x_j = \sum_{l=0}^{m-1} \frac{\langle w_l, z_j \rangle}{\langle w_l, v_{l+1} \rangle}\, w_l\ , \quad 1 \leq j \leq r\ . \tag{2.19}$$

Der Algorithmus wird nun in verschiedene Phasen eingeteilt, deren Laufzeit dann ermittelt wird. Es werden jeweils nur die benötigten Körperoperationen gezählt, da im Moment nur interessant ist, von welchen Faktoren die Laufzeit des Verfahrens abhängt.

1. **Berechnung von $\boldsymbol{v_{i+1} = A\, w_i = B^T B\, w_i}$**
   Da B eine dünnbesetzte Matrix ist, jedoch B^TB in der Regel dichtbesetzt sein wird, berechnet man aus Speicherplatzgründen $A = B^TB$ nicht einmal vor, sondern berechnet in jeder Iteration immer $B^T(Bw) = v_{i+1}$. Sei ω das Gewicht von B (vgl. Def. 1.6). Dann benötigt man

   $$2 \cdot \omega \quad \text{Körperoperationen}$$

   pro Iteration zur Berechnung von v_{i+1}.

2. **Berechnung von w_{i+1}**

$$w_{i+1} = v_{i+1} - \frac{\langle v_{i+1}, v_{i+1}\rangle}{\langle w_i, v_{i+1}\rangle}\, w_i - \frac{\langle w_i, v_{i+1}\rangle}{\langle w_{i-1}, v_i\rangle}\, w_{i-1}$$

   (a) **Berechnung der inneren Produkte $\langle v_{i+1}, v_{i+1}\rangle, \langle w_i, v_{i+1}\rangle$**
   Bei der Berechnung der inneren Produkte müssen die einzelnen Vektorkomponenten multipliziert und aufaddiert werden. Da jeder Vektor n-dimensional ist, benötigt man pro Iteration

$$4 \cdot n \quad \text{Körperoperationen}$$

   zur Berechnung der beiden inneren Produkte $\langle v_{i+1}, v_{i+1}\rangle, \langle w_i, v_{i+1}\rangle$.

   (b) **Bildung von w_{i+1}**
   Es werden 2 Multiplikationen und 1 Invertierung zur Bestimmung der Skalaren $\langle v_{i+1}, v_{i+1}\rangle / \langle w_i, v_{i+1}\rangle$, $\langle w_i, v_{i+1}\rangle / \langle w_{i-1}, v_i\rangle$ benötigt (da aus der vorheriger Iteration $\langle w_{i-1}, v_i\rangle^{-1}$ schon bekannt ist). Dann müssen sämtliche Komponenten von w_i und w_{i-1} damit multipliziert werden und von der entsprechenden Komponente von v_{i+1} abgezogen werden. Somit benötigt man

$$4 \cdot n + 3 \quad \text{Körperoperationen}$$

   zur Bildung von w_{i+1}.

   Zusammen benötigt man also

$$8 \cdot n + 3 \quad \text{Körperoperationen}$$

   pro Iteration zur Berechnung von w_{i+1}.

3. **Aktualisierung einer partiellen Lösung**
   In der i-ten Iteration kann die j-te partielle Lösung als

$$x_j = x_j + \frac{\langle w_i, z_j\rangle}{\langle w_i, v_{i+1}\rangle}\, w_i$$

   berechnet werden. Aus der Berechnung von w_{i+1} ist $\langle w_i, v_{i+1}\rangle^{-1}$ schon bekannt. Es muß pro Lösung ein inneres Produkt bestimmt werden, ein Skalar ausgerechnet werden und ein Vielfaches der Vektorkomponenten zu den entsprechenden Vektorkomponenten von x_j addiert werden. Deshalb werden

$$4 \cdot n + 1 \quad \text{Körperoperationen}$$

   zur Aktualisierung einer partiellen Lösung pro Iteration benötigt.

Insgesamt werden also pro Iteration (bei einer berechneten Lösung)

$$2 \cdot \omega + 12 \cdot n + 4 \quad \text{Körperoperationen}$$

gebraucht. Da man höchstens n Iterationen durchführen muß, beträgt die Laufzeit des Lanczos Algorithmus

$$O(n^2 + \omega \cdot n) \quad \text{Körperoperationen.}$$

Die Laufzeit hängt also von der Dimension n des zu lösenden linearen Gleichungssystems und vom Gewicht ω der daraus resultierenden Matrix B ab.

Kapitel 3

Struktur der Systeme

Diese Arbeit beschäftigt sich mit dem Lösen linearer Gleichungssysteme, die bei der Faktorisierung ganzer Zahlen und bei der Lösung von diskreten Logarithmen (DL) Problemen auftreten. In diesem Kapitel wird die Struktur dieser Systeme analysiert. Dazu ist es notwendig, deren Herkunft genauer zu betrachten. Ich werde zunächst eine stark vereinfachte Sichtweise auf die verwendeten Faktorisierungs- und DL-Algorithmen geben. Für eine detailierte Betrachtung der einzelnen Algorithmen sind Verweise angegeben. Aufbauend auf der Herkunft der Gleichungssysteme werden einige Eigenschaften der Systeme abgeleitet.

3.1 Herkunft der Gleichungssysteme

Ich interessiere mich für Gleichungssysteme, die von den Faktorisierungsalgorithmen Quadratisches Sieb ([6],[7],[38]) bzw. Number Field Sieve (NFS) ([26],[43],[21]) oder von Algorithmen zur Lösung Diskreter Logarithmen (DL) Probleme in endlichen Primkörpern NFS für DL([16],[36],[39]) bzw. Index Calculus (IC) Algorithmen ([27],[32],[37]) aufgestellt werden. Als Vertreter der IC Verfahren wurde das Coppersmith-Odlyzko-Schroeppel Verfahren (COS) ([4]) untersucht.
Die oben genannten Algorithmen lassen sich grob in 3 Phasen gliedern. In der ersten Phase, der sogenannten *Siebphase*, werden Relationen gesammelt und daraus wird dann ein Gleichungssystem generiert. Das Lösen dieses Systems über einem endlichen Primkörper stellt dann die zweite Phase dar. In der dritten Phase werden die gefundenen Lösungen ausgewertet.
In der ersten Phase wählt man sich eine endliche Menge (FB) von Primzahlen, genannt *Faktorbasis*. In der Praxis beinhaltet diese Menge alle Primzahlen kleiner als eine Schranke FB_B, wobei typischerweise $FB_B \in [5 \cdot 10^3, 5 \cdot 10^5]$. Die Menge der gesammelten Relationen ($\mathcal{R}$) kann man dann als endliche Teilmenge der natürlichen

Zahlen betrachten, wobei

$$r_j \in \mathcal{R} \ : \ n_j = \prod_{p_i \in FB} p_i^{e_{ij}} \ . \tag{3.1}$$

Definition 3.1 (Exponentenvektor)

Für eine Relation r_j der Form

$$r_j \in \mathcal{R} \ : \ n_j = \prod_{p_i \in FB} p_i^{e_{ij}} \tag{3.2}$$

bezeichnen wir den Zeilenvektor e_{ij} als **Exponentenvektor** *der Relation r_j.*

Durch die Art und Weise, wie solche Relationen generiert und vor allem interpretiert werden, läßt sich eine Unterscheidung zwischen Faktorisierungsalgorithmen und dem NFS für DL auf der einen Seite und den IC Verfahren auf der anderen Seite treffen. Die unterschiedliche Vorgehensweise führt auch zu unterschiedlichen Gleichungssystemen, die auf verschieden Art und Weise gelöst werden müssen. Ich werde zuerst eine kurze Übersicht für IC Verfahren geben, bevor ich mich auf die Verfahren und Gleichungssysteme konzentriere, die von Faktorisierungsalgorithmen und dem NFS für DL generiert werden.

3.1.1 Index Calculus Verfahren

In Index Calculus Verfahren werden spezielle Relationen der Form (3.1) generiert. Angenommen, man möchte den Logarithmus zur Basis eines Erzeugers $g \in \mathbb{F}_p$ bestimmen. Dann sucht man Werte b_j $(1 \leq b_j < p)$, so daß

$$n_j \equiv g^{b_j} \bmod p \quad \text{und} \quad n_j = \prod_{p_i \in FB} p_i^{e_{ij}}$$

gilt. Dann gilt offensichtlich

$$\prod_{p_i \in FB} p_i^{e_{ij}} \equiv g^{b_j} \bmod p \ .$$

Logarithmiert man diese Relationen, so erhält man ein Gleichungssystem, in dem die Unbekannten den Logarithmen der Faktorbasiselemente zur Basis g entsprechen, d.h. es gilt für alle i

$$\sum_j e_{ij} \cdot \log_g(p_j) \equiv b_j \bmod (p-1) \ .$$

Wenn $n = |\mathcal{R}|$ und $m = |FB|$ ist, dann hat das zu lösende Gleichungssystem folgendes Aussehen:

$$\begin{pmatrix} e_{11} & \cdots & e_{1m} \\ \vdots & & \vdots \\ e_{n1} & \cdots & e_{nm} \end{pmatrix} \begin{pmatrix} \log_g(p_1) \\ \vdots \\ \log_g(p_m) \end{pmatrix} \equiv \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} \bmod (p-1) \ .$$

Die Zeilenvektoren werden also von den Exponentenvektoren der Relationen gebildet, und die einzelnen Faktorbasiselemente sind den Spalten zugeordnet. Man sucht soviele Relationen, bis ein überbestimmtes Gleichungssystem erstellt werden kann, dessen eindeutige Lösung die gesuchten Logarithmen liefert.

Bemerkung 3.2 Man beachte, daß eine Lösung modulo $p-1$ benötigt wird. Mit Hilfe des chinesischen Restsatzes und Hensel Liftings läßt sich eine solche Lösung aus Lösungen modulo q bestimmen, wobei q alle Primteiler von $p-1$ durchläuft.

3.1.2 Faktorisierungsverfahren und NFS für DL

Bei den Faktorisierungsverfahren und dem Number Field Sieve für DL ist das Ziel der zweiten Phase eine geeignete Kombination der Relationen zu finden. Man sucht für jede Relation $r_j \in \mathcal{R}$ eine geeignete Zahl λ_j, so daß für ein $q \in \mathbb{P}$ ein $H \in \mathbb{K}$ existiert mit

$$\prod_{r_j \in \mathcal{R}} n_j^{\lambda_j} = H^q \quad \text{und} \quad \exists\, \lambda_j \not\equiv 0 \bmod q \ .$$

Dabei sucht man für Faktorisierungsalgorithmen Quadrate ($q = 2$) und im Fall eines DL Problems in $\mathbb{F}_p$ muß man q-te Potenzen für alle Primteiler q von $p-1$ finden. Die Bestimmung der λ_j erfolgt durch Lösen eines Gleichungssystems. Man sieht leicht

$$\begin{aligned}
& \prod_{r_j \in \mathcal{R}} n_j^{\lambda_j} = H^q \\
\iff \quad & \prod_j \prod_{p_i \in FB} p_i^{e_{ij} \cdot \lambda_j} = H^q \\
\iff \quad & \prod_{p_i \in FB} p_i^{\sum_j e_{ij} \cdot \lambda_j} = H^q \\
\iff \quad & \forall i \quad \sum_j e_{ij} \cdot \lambda_j \equiv 0 \bmod q \ .
\end{aligned}$$

Wenn $n = |\mathcal{R}|$ und $m = |FB|$ ist, dann hat das zu lösende Gleichungssystem folgendes Aussehen:

$$\begin{pmatrix} e_{11} & \cdots & e_{1n} \\ \vdots & & \vdots \\ e_{m1} & \cdots & e_{mn} \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_n \end{pmatrix} \equiv \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix} \bmod q \ .$$

Die Spaltenvektoren werden also von den Exponentenvektoren der Relationen gebildet, und die einzelnen Faktorbasiselemente sind den Zeilen zugeordnet. Wir suchen

eine nichttriviale Abhängigkeit unter den Exponentenvektoren. Solch eine Abhängigkeit existiert immer, falls $n > m$ ist. Betrachtet man nur die Struktur der Systeme, so unterscheiden sich diese Gleichungssysteme von denen, die von Index Calculus Verfahren generiert werden, im wesentlichen nur durch die Vertauschung von Zeilen und Spalten. Somit gelten die weiteren Betrachtungen sinngemäß auch für Gleichungssysteme, die von Index Calculus Verfahren generiert wurden.

Einige Modifikationen dieser Grundalgorithmen haben auch starken Einfluß auf die Struktur der Systeme. So sind in den letzten Jahren Verbesserungen der Siebphase, die sogenannten *Large Prime Variations*, vorgestellt worden, die die Einsetzbarkeit der Algorithmen stark erweitert haben (vgl. [11],[25]). Dabei benutzt man sogenannte *partial relations*, d.h. Zerlegungen, die zusätzlich zu Primzahlen aus der Faktorbasis noch bis zu zwei Primzahlen (sog. *large primes*) enthalten, die nicht in der Faktorbasis sind. Um aus diesen partial relations nützliche Relationen zum Aufbau des Gleichungssystems zu bekommen, sucht man geeignete Kombinationen von partial relations, die nur large primes in q-ter Potenz oder Faktorbasiselemente enthalten. Je nachdem, ob man die Potenz der large primes in einer bisherigen Kombination erhöhen oder erniedrigen will, multipliziert man mit einer entsprechenden partial relation oder dividiert durch sie. Der Exponentenvektor einer kombinierten Relation entsteht dann also durch Addition bzw. Subtraktion der Exponentenvektoren der partial relations, die an ihrer Kombination beteiligt sind.

Bemerkung 3.3 Zur Unterscheidung von partial relations bezeichne ich Relationen der Form (3.2) als *full relations*. Eine Kombination von partial relations zu einer full relation heißt *combined relation*. Ist eine genaue Unterscheidung nicht notwendig, so rede ich allgemein von *relations*.

3.2 Eigenschaften der Gleichungssysteme

Im folgenden Absatz untersuche ich die Eigenschaften der oben beschriebenen Gleichungssysteme.

Bemerkung 3.4 Ich werde folgende Ergebnisse der Zahlentheorie benutzen:

- Die Anzahl der Primfaktoren einer zufällig gewählten Zahl $n \in \mathbb{N}$ liegt in $O(ln(ln(n)))$(vgl. [34] S. 156).
- Die Wahrscheinlichkeit für eine Primzahl $p \in \mathbb{P}$, eine zufällig gewählte Zahl $n \in \mathbb{N}$ zu teilen, ist ungefähr $1/p$.

Die Dünnbesetztheit der Systeme folgt damit aus ihrem Aufbau durch die Exponentenvektoren. Da die Spalten aus den Exponentenvektoren der Relationen aufgebaut

werden, hängt ihr Gewicht davon ab, ob es sich um eine full oder combined relation handelt bzw. wieviele relations an ihrem Aufbau beteiligt waren. Diesen Zusammenhang verdeutlicht Tabelle 3.1, die Daten aus der Berechnung eines DL-Problems in einem Primkörper mit ungefähr 10^{85} Elementen enthält (DL85). Die Berechnungen wurden mit dem Number Field Sieve für DL-Probleme durchgeführt. In diesem Beispiel hat die Faktorbasis 70 342 Elemente. Die erste Spalten der Tabelle enthält die Anzahl der am Aufbau der Relation beteiligten relations. Dann folgt jeweils die Anzahl der combined relations, das minimale, maximale und das durchschnittliche Gewicht einer Spalte, an deren Aufbau entsprechend viele relations beteiligt waren.

# rel.	num	min	max	avg	# rel.	num	min	max	avg
1	7482	15	28	21.60	11	12330	144	177	160.58
2	4322	28	44	36.26	12	13012	155	192	172.56
3	3911	43	63	52.66	13	13354	169	205	185.50
4	4241	57	77	66.39	14	13595	176	214	197.25
5	5064	70	93	81.33	15	12846	186	228	209.93
6	5935	76	107	94.45	16	11825	199	242	221.47
7	6963	96	121	108.72	17	9969	213	258	233.99
8	8312	106	136	121.27	18	7774	226	264	245.12
9	9680	120	150	135.04	19	4662	232	278	257.44
10	11046	129	163	147.36					

Tabelle 3.1: Gewichtsverteilung in Abhängigkeit der beteiligten relations

In diesem Beispiel wurden 175 000 combined relations, die aus bis zu 19 partial relations aufgebaut sind, zum Aufbau des Gleichungssystems benutzt. Die Tabelle zeigt, daß man aufgrund des Gewichtes einer Spalte nicht eindeutig auf die Anzahl der an ihrem Aufbau beteiligten relations schließen kann. Jedoch läßt sich in diesem Beispiel die Anzahl der beteiligten relations immer auf höchstens 4 aufeinanderfolgende Möglichkeiten eingrenzen.

Die Abbildung 3.1 verdeutlicht die Verteilung des Spaltengewichtes. Auf der X-Achse ist das Spaltengewicht, auf der Y-Achse die Anzahl der Spalten (relations) mit diesem Gewicht aufgetragen. Die Spitzen in der Abbildung tauchen an genau den Stellen auf, an dem das durchschnittlichen Gewicht der verschiedene Tabelleneinträgen liegt. Somit lassen sich die einzelnen "Hügel" den verschiedene Klassen der combined relations zuordnen. Dabei muß man beim Number Field Sieve noch eine technische Besonderheit beachten. Die erste Spitze resultiert von den sogenannten *free relations* (vgl. [26]). Diese free relations werden nicht durch die Siebphase gefunden, sondern man erhält sie aufgrund einer Besonderheit des Number Field Sieve, auf die ich hier nicht näher eingehen will. Man kann die free relations als full relations mit 5 Einträgen auffassen. Durch ihre Dünnbesetztheit werden diese Spalten später noch eine wichtige Rolle spielen. In Abbildung 3.1a wird jedem X-Wert das kumulierte Gewicht der x-dünnbesetztesten Spalten zugeordnet. Anhand dieser

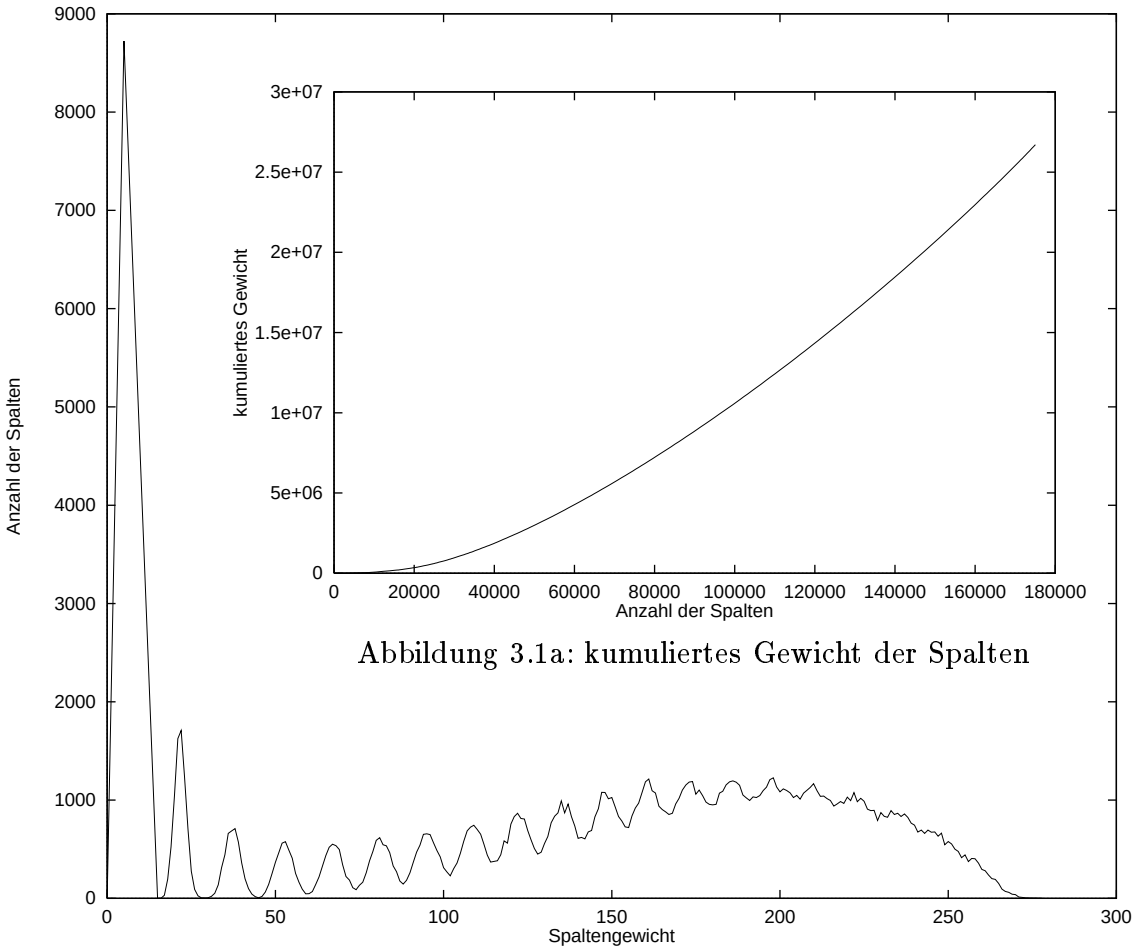


Abbildung 3.1a: kumuliertes Gewicht der Spalten

Abbildung 3.1: Verteilung des Spaltengewicht

Abbildung kann man sehen, wie stark das Gewicht der Matrix wächst, wenn man nicht genügend dünnbesetzte Spalten in der Siebphase gefunden hat.

Während sich das Gewicht der einzelnen Spalten nicht sehr stark voneinander unterscheidet ($5 \leq$ Gewicht einer Spalte ≤ 278), stellt man sehr große Unterschiede bei den Zeilen fest. Da es für kleine Faktorbasiselemente viel wahrscheinlicher ist, eine zufällig gewählte Zahl zu teilen, ist das Gewicht der Zeilen, die kleinen Faktorbasiselementen zugeordnet sind, viel höher als das Gewicht der Zeilen, die großen Faktorbasiselementen zugeordnet sind (vgl. Bemerkung 3.4). Es gibt ein paar Zeilen, bei denen fast alle Einträge ungleich Null sind, aber auch einige Zeilen, deren Gewicht 0 ist. Das Gewicht der 600 schwersten Zeilen des Beispiels ist in Abbildung 3.3b dargestellt. Praktische Untersuchungen haben gezeigt, daß in den ersten $\sqrt{m}$ Zeilen ($m = |FB|$) eines Systems ungefähr die Hälfte der von Null verschiedenen Einträge steht (vgl. [33]). Wenn also ω das Gewicht des Systems ist, dann gilt:

$$\sum_{i<\sqrt{m}} \text{Gewicht (Zeile } i) \approx \frac{\omega}{2} .$$

Anhand der Abbildung 3.2 läßt sich diese Verhalten sehr schön überprüfen. In diesem Beispiel würde das bedeuten, daß in den ersten $\sqrt{70\,342} \approx 265$ Zeilen bereits die Hälfte aller Einträge stehen. In diesem Beispiel benötigt man allerdings die 1515 schwersten Zeilen (was etwa 2% aller Zeilen entspricht) um 50% des Gewichtes der Matrix abzudecken.

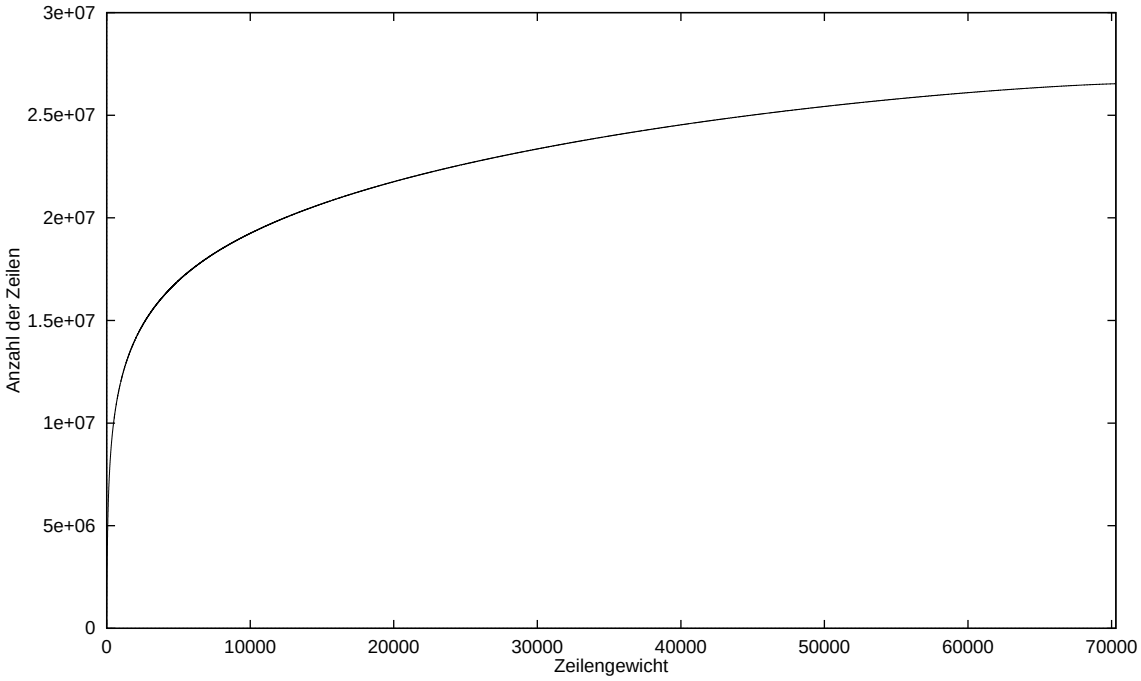


Abbildung 3.2: Kumuliertes Zeilengewicht

Die Tabelle 3.2 enthält für ausgesuchte Indizes i das kumulierte Zeilengewicht der i schwersten Zeilen. Die Verteilung des Zeilengewichtes ist in den Abbildungen 3.3 und 3.3a dargestellt. Dabei wurde der dargestellte Bereich des Zeilengewichtes variiert, jedoch wurde in beiden Fällen auf die Darstellung der schwersten Zeilen verzichtet.

Index	Gewicht	%	Index	Gewicht	%	Index	Gewicht	%
265	8 374 706	31.6	1 515	13 265 212	50.0	10 000	19 244 438	72.5
500	10 106 811	38.1	2 500	14 766 458	55.7	30 000	23 357 760	88.0
1 000	12 062 644	45.5	5 000	16 939 736	63.8	70 342	26 535 569	100

Tabelle 3.2: kumuliertes Zeilengewicht

In Abbildung 3.3a werden Zeilengewichte $\leq 10\,000$ dargestellt. Schränkt man das dargestellte Zeilengewicht noch stärker ein ($\leq 1\,000$), so erhält man Abbildung 3.3.

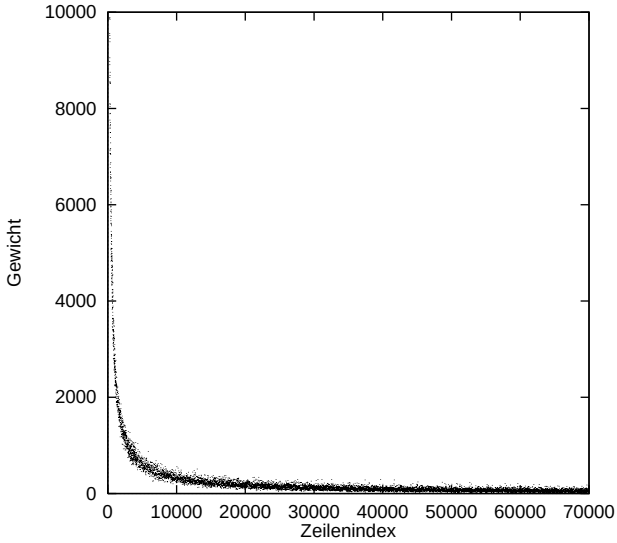


Abbildung 3.3a

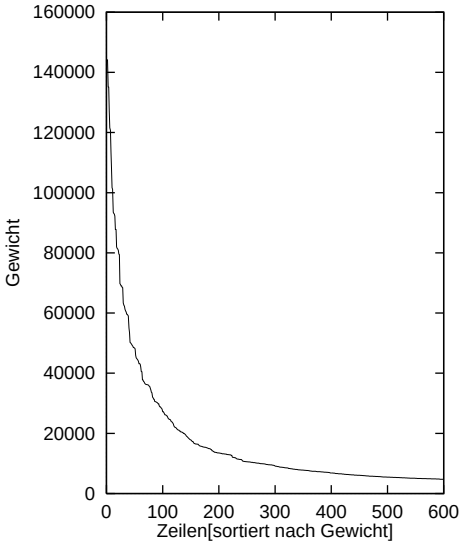


Abbildung 3.3b

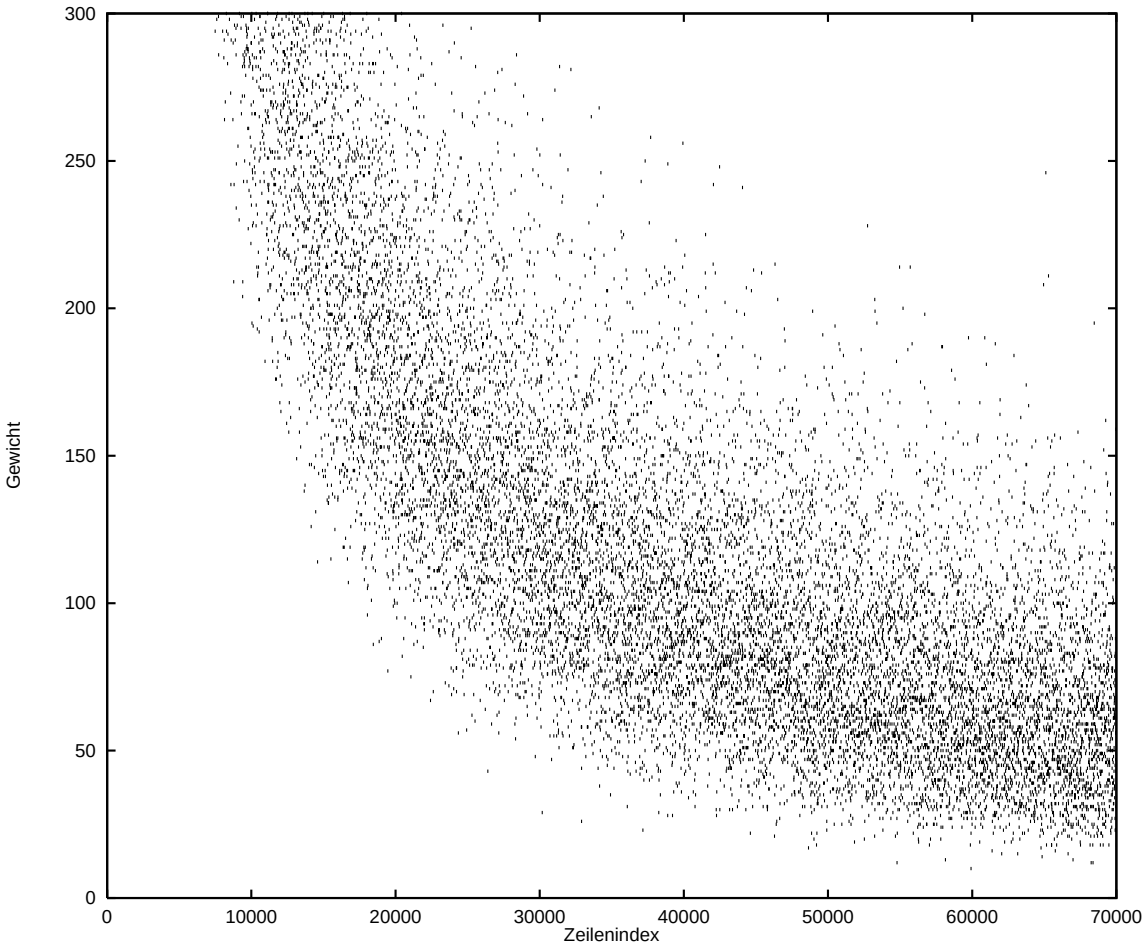


Abbildung 3.3: Verteilung des Zeilengewichtes (Ausschnitt)

In dieser Abbildung sieht man sehr gut eine Streuung im Zeilengewicht. Es gibt sehr viele Zeilen i, j mit Gewicht(i) > Gewicht(j), obwohl $i > j$ und somit $p_i > p_j$ ist. Dennoch läßt sich klar die Tendenz erkennen (vor allem in Abbildung 3.3a und 3.3b), daß Zeilen, die kleinen Faktorbasiselementen zugeordnet sind, ein wesentlich größeres Gewicht haben als die zu großen Faktorbasiselementen korrespondierenden Zeilen. In Tabelle 3.3 ist jedem Zeilengewicht kleiner als 31 die Anzahl der Zeilen mit diesem Gewicht zugeordnet. Dabei muß man beachten, daß in diesem Beispiel das Gleichungssystem aus etwa 2.5 mal mehr Spalten als Zeilen aufgebaut ist. Das läßt erwarten, daß auch das Gewicht der Zeilen um diesem Faktor höher ist als das Zeilengewicht bei einem quadratischen System.

Gewicht	Anzahl	Gewicht	Anzahl	Gewicht	Anzahl	Gewicht	Anzahl
7	1	13	5	19	64	25	130
8	1	14	16	20	58	26	130
9	1	15	31	21	70	27	140
10	3	16	21	22	82	28	177
11	9	17	29	23	102	29	149
12	10	18	47	24	102	30	182

Tabelle 3.3: Anzahl der dünnbesetzten Zeilen

Bisher wurde nur betrachtet, wie sich die von Null verschiedenen Einträge verteilen. Die Größe der Einträge ist jedoch auch wichtig. Bedenkt man, daß die Einträge Exponenten von Primzahlen in Primfaktorisierungen sind (full relation) bzw. aus einer Kombination (Addition, Subtraktion) von Exponenten entstanden sind, so erwartet man kleine Einträge. Die Tabelle 3.4 enthält für verschiedene Beispiele eine Aufstellung der Häufigkeit, in der Einträge in diesem System auftreten. Dabei unterteile ich die Einträge in +1-Einträge, −1-Einträge und sonstige Einträge. Es wird jeweils auch der prozentuale Anteil dieser Anträge an der Gesamtanzahl der Einträge angegeben. Außerdem ist jeweils der größte bzw. kleinste Eintrag in diesem System angegeben. Die Beispiele (DL75, DL85) stammen aus der Berechnung von

Einträge	DL75	DL85	McCurley
+1	3 658 635 (45.58 %)	11 790 588 (44.22 %)	2 666 710 (44.66 %)
−1	3 914 529 (48.77 %)	13 161 884 (49.37 %)	2 994 594 (50.16 %)
sonstige	452 894 (5.65 %)	1 708 994 (6.41 %)	309 192 (5.18 %)
Minimum	-16	-44	-57
Maximum	16	33	48

Tabelle 3.4: Anzahl der Einträge (DL85)

DL-Problemen in einem Primkörper mit ungefähr 10^{75} bzw. 10^{85} Elementen. Das Beispiel (McCurley) stammt aus der Lösung der ersten Phase der sog. McCurley Challenge (vgl. [27]).

Die Tabelle 3.5 listet für jeden Eintrag die Häufigkeit seines Auftretens im DL85 Beispiel auf.

Eintrag	Anzahl	Eintrag	Anzahl	Eintrag	Anzahl	Eintrag	Anzahl
-44	1	-18	339	-1	13161884	17	303
-35	1	-17	481	1	11790588	18	218
-34	2	-16	706	2	490485	19	125
-32	6	-15	942	3	124007	20	119
-31	6	-14	1356	4	51426	21	66
-30	5	-13	1882	5	27224	22	40
-29	2	-12	2756	6	16914	23	40
-28	13	-11	3759	7	11201	24	17
-27	15	-10	5366	8	7638	25	15
-26	13	-9	7538	9	5329	26	13
-25	21	-8	10730	10	3676	28	4
-24	42	-7	15416	11	2559	29	4
-23	58	-6	22863	12	1742	30	2
-22	98	-5	36238	13	1256	31	1
-21	121	-4	68465	14	857	32	1
-20	150	-3	166338	15	618	33	1
-19	232	-2	616857	16	425		

Tabelle 3.5: Verteilung der Einträge (DL85)

Kapitel 4

Reduzierung des Gewichtes

Durch eine Modifikation der Siebphase des Number Field Sieve Algorithmus, der sog. *Quadruple Large Prime Variation* (vgl. [11]), konnte die erste Phase (Sammeln der Relationen) erheblich beschleunigt werden. Obwohl diese Modifikation auch auf Index Calculus Verfahren angewandt werden kann, konzentriere ich mich hier auf Number Field Sieve Beispiele. Die Modifikation führt zu deutlich dichter besetzten Gleichungssystemen, die damit auch schwerer zu lösen sind. In diesem Kapitel werde ich eine Erweiterung dieser Modifikation vorstellen, die das Gewicht der zu lösenden Systeme um bis zu 30% reduziert. Dabei spielt es keine Rolle, ob das Number Field Sieve zum Faktorisieren ganzer Zahlen oder zur Berechnung von DL-Problemen eingesetzt wird. Da die Laufzeit des Lanczos Algorithmus auch von dem Gewicht des zu lösenden Systems abhängt (vgl. Kapitel 2), erhält man durch eine Reduzierung des Gewichtes schneller lösbare Gleichungssysteme. Wegen der insgesamt wesentlich größeren Laufzeit bei DL-Berechnungen, ist die Bedeutung der Gewichtsreduktion hier wesentlich höher als bei der Faktorisierung ganzer Zahlen. Die Laufzeit des hier vorgestellten Reduzierungsschrittes kann dabei, verglichen mit der zum Lösen des Systems benötigten Zeit (vor allem bei DL-Berechnungen), vernachlässigt werden.

4.1 Voraussetzungen und Notation

In diesem Abschnitt skizziere ich kurz die *Quadruple Large Prime Variation* (vgl. [11]) für das Number Field Sieve, soweit dies zur Erklärung der hier vorgestellten Erweiterung nötig ist. Außerdem werden einige Notationen und Begriffe eingeführt, die bei der Beschreibung der Erweiterung benötigt werden.
In der Quadruple Large Prime Variation werden mit einem Graphenalgorithmus Mengen von partial relations berechnet, die zu einer combined relation zusammengesetzt werden können. Dabei reduziert der Graphenalgorithmus das Problem der Bestimmung dieser Teilmengen auf das Finden aller unabhängigen Zyklen (einer

Basis der Zyklen) in einem ungerichteten Graphen. Die Knoten dieses Graphen werden von der Menge der vorkommenden large primes gebildet, die Kanten entsprechen den partial relations. Ich werde den Graphenalgorithmus hier nicht detailiert vorstellen. Eine genaue Beschreibung findet sich in ([25], [43]). Um möglichst dünnbesetzte Gleichungssysteme zu erhalten ist man an combined relations interessiert, die aus möglichst wenigen partial relations aufgebaut sind. Deshalb sucht man nach kürzesten Zyklen des Graphen. In Kapitel 3 wurde bereits auf den Zusammenhang zwischen dem Gewicht einer Spalte und der Anzahl der am Aufbau der entsprechenden combined relation beteiligten partial relations eingegangen (vgl. Tabelle 3.1, S. 31). Es existiert ein Algorithmus von Horten, der eine Basis bestimmt, die aus den kürzesten Zyklen des Graphen aufgebaut ist ([20]). Dessen Komplexität verhindert jedoch seinen Einsatz in praktischen Anwendungen dieser Größe (vgl. [25]). Die in der Praxis berechnete Basis ist also nicht optimal bzgl. der Anzahl der an ihrem Aufbau beteiligten Kanten. Der hier vorgestellte Algorithmus benutzt die von dem verwendeten Graphenalgorithmus berechneten Zyklen und führt Basistransformationen durch, die die Anzahl der beteiligten Kanten reduzieren.
Im folgenden gehe ich davon aus, daß die Menge der unabhängigen Zyklen $\mathcal{C}$ berechnet wurde. Die Menge der partial relations, die einen Zyklus i bilden, wird mit *edges(i)* bezeichnet. Die Menge aller partial relations, die in mindestens einem Zyklus vorkommen, bezeichne ich mit $\mathcal{E}$. Einzelne partial relations werden mit a, b, c, d bzw. mit e bezeichnet. Unter der Länge ($len(i)$) eines Zyklus i verstehe ich die Anzahl der Kanten (partial relations), die an seinem Aufbau beteiligt sind ($len(i) := |edges(i)|$). Die Summe der Längen aller Zyklen, die am Aufbau einer Basis beteiligt sind, bezeichne ich als Länge der Basis. Ich definiere nun einige Abbildungen, die im Laufe dieses Kapitels benötigt werden.

Definition 4.1 (Abbildungen cyc, h)

Um die Menge der Zyklen zu bestimmen, in denen eine partial relation vorkommt, definiere ich

$$\begin{aligned} cyc: \quad \mathcal{E} &\rightarrow \mathcal{C} \\ a &\mapsto \{i | a \in edges(i)\} \ . \end{aligned}$$

Die Abbildung

$$\begin{aligned} h: \quad \mathcal{E} &\rightarrow \mathbb{N} \\ a &\mapsto |cyc(a)| \end{aligned}$$

zählt für jede partial relation, in wievielen Zyklen sie vorkommt.

Ich teile die relations nach der Anzahl der Zyklen ein, in denen sie vorkommen.

Definition 4.2 (solitary und multiple relation)

Eine partial relation a, die nur in einem Zyklus auftaucht ($h(a) = 1$) bezeichne ich als **solitary relation**. *Sonst spreche ich von einer* **multiple relation**.

Für den Algorithmus ist es wichtig zu wissen, wieviele solitary relations in jedem Zyklus vorkommen. Deshalb definiere ich die Abbildung

Definition 4.3 (one_entry)

$$\begin{aligned} one\_entry: \quad \mathcal{C} &\rightarrow \mathbb{N} \\ i &\mapsto \left|\{a | a \in edges(i) \wedge h(a) = 1\}\right| . \end{aligned}$$

Dann gibt $one\_entry(i)$ die Anzahl der solitary relations in Zyklus i an. Die Anzahl der multiple relations in jedem Zyklus kann dann einfach als Differenz der Länge eines Zyklus und der Anzahl seiner solitary relations bestimmt werden.

Für zwei Zyklen (bzw. ihre Mengen von partial relations) definiere ich folgende Funktionen:

Definition 4.4 (common part, combination, remaining part)

$$\begin{aligned} common: \quad \mathcal{C} \times \mathcal{C} &\rightarrow \mathcal{E} \\ (i,j) &\mapsto edges(i) \cap edges(j), \end{aligned}$$

die den **common part** *von Zyklus i und j berechnet,*

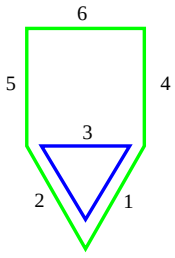
$$\begin{aligned} \bigoplus: \quad \mathcal{C} \times \mathcal{C} &\rightarrow \mathcal{E} \\ (i,j) &\mapsto (edges(i) \cup edges(j)) - common(i,j), \end{aligned}$$

die die **combination** *von Zyklus i und j ermittelt und*

$$\begin{aligned} rem: \quad \mathcal{C} \times \mathcal{C} &\rightarrow \mathcal{E} \\ (i,j) &\mapsto edges(i) - common(i,j), \end{aligned}$$

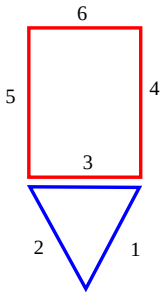
die den **remaining part** *von Zyklus i (wenn er mit Zyklus j kombiniert wird) berechnet.*

Beispiel:



Sei Zyklus i der blaue Zyklus und Zyklus j der grüne Zyklus. Dann gilt :

$$\begin{aligned} edges(i) = \{1,2,3\}, \quad edges(j) &= \{1,2,4,5,6\} \\ len(i) + len(j) &= 3 + 5 = 8 \\ rem(i,j) = \{3\}, \quad common(i,j) &= \{1,2\} \\ i \oplus j &= (\{1,2,3\} \cup \{1,2,4,5,6\}) \setminus \{1,2\} \\ &= \{3,4,5,6\} \end{aligned}$$



Der Zyklus $(i \oplus j)$ ist rot eingezeichnet. Es gilt: $len(i) + len(i \oplus j) = 7$.

Folgende Beobachtungen lassen sich leicht für alle Zyklen i, j überprüfen:

$$common(i,j) = common(j,i)\ , \tag{4.1}$$
$$edges(i) = common(i,j) \cup rem(i,j) \quad \text{und} \tag{4.2}$$
$$|common(i,j)| > |rem(i,j)| \iff 2 \cdot |common(i,j)| > len(i)\ . \tag{4.3}$$

4.2 Idee des Algorithmus

Untersucht man die Mengen von partial relations, die von dem bisher in der Praxis eingesetzten Graphenalgorithmus berechnet werden, so stellt man fest, daß diese Mengen sehr oft große Schnittmengen haben. Es sind also sehr viele partial relations am Aufbau mehrerer combined relations beteiligt. Dieser Abschnitt beschäftigt sich damit, wie man diese Beobachtung nutzen kann, um die Zyklen zu verkürzen.

4.2.1 Reduzierung der Anzahl der partial relations

Ich betrachte zuerst den Fall, daß man zwei Zyklen findet, die mehr als die Hälfte der partial relations von einem von ihnen gemeinsam haben. Angenommen, man findet Zyklus $i = abcde$ (schwarz) und Zyklus $j = abcr$ (blau), wobei r ein Pfad mit mindestens zwei partial relations ist. Diese Situation ist auf der linken Seite von Abbildung 4.1 dargestellt. Man kann jetzt in Zyklus j (blau) die Kanten abc $(= common(i,j))$ durch ed $(= rem(i,j))$ austauschen, ohne die Zyklus Eigenschaft zu zerstören. Die rechte Seite von Abbildung 4.1 zeigt die Situation nach dem Austauschen der Kanten (Zyklus $i = abcde$ (schwarz), Zyklus $j = der$ (blau)). In diesem Beispiel hat sich die Länge von Zyklus j um 1 verringert.

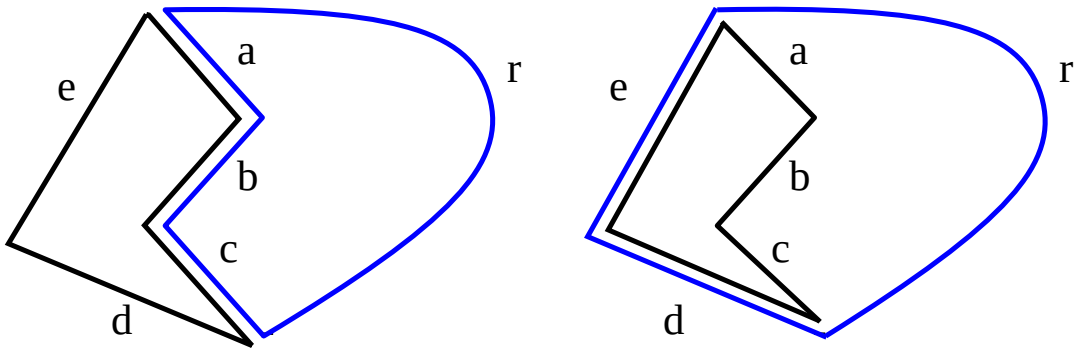


Abbildung 4.1: Reduzierung der Zykluslänge

Diese Beobachtung führt zu folgendem Vorgehen:
Angenommen, man findet ein Paar von Zyklen (i,j) mit $|common(i,j)| > |rem(i,j)|$, wobei o.B.d.A. $len(i) \leq len(j)$. Dann kann man den ursprünglichen Zyklus j durch die combination von Zyklus i und j ersetzen $(j = i \oplus j)$. Diese Operation hat den

Effekt, daß in $edges(j)$ die partial relations, die beide Zyklen gemeinsam hatten ($common(i,j)$), durch die partial relations ersetzt werden, die nur in Zyklus i vorkommen ($rem(i,j)$). Da nach Voraussetzung $|common(i,j)| > |rem(i,j)|$ gilt, ist Zyklus j kleiner geworden. Man sagt: *Zyklus i reduziert Zyklus j.*

4.2.2 Austausch von partial relations

Im letzten Abschnitt habe ich die Bedingungen untersucht, die zu einer Reduzierung der Zykluslänge führen. Dort mußten die Zyklen mehr als die Hälfte der partial relations von einem von ihnen gemeinsam haben. Im folgenden betrachte ich den Fall, daß sie genau die Hälfte der partial relations von einem von ihnen gemeinsam haben. Für zwei solche Zyklen i und j, wobei o.B.d.A. $len(i) \leq len(j)$ gilt also $|common(i,j)| = |rem(i,j)|$. In diesem Fall kann Zyklus i den Zyklus j nicht verkürzen. Erhöht ein Austausch von partial relations aber die Wahrscheinlichkeit, daß der "neue" Zyklus $j = i \oplus j$ reduziert werden kann, so ist dieser Austausch trotzdem nützlich. Als Maß für die Wahrscheinlichkeit, daß ein Zyklus reduziert werden kann, benutze ich die folgende Funktion:

$$\begin{aligned} prob: \quad \mathcal{C} &\rightarrow \mathbb{N} \\ i &\mapsto \sum_{e \in edges(i)} h(e) \ . \end{aligned}$$

Diese Funktion addiert für jeden Zyklus die Häufigkeit des Auftretens seiner partial relations in allen Zyklen auf. Als Grundlage für diese Funktion dient die Heuristik, daß die Wahrscheinlichkeit für einen Zyklus reduziert zu werden, mit der Anzahl der Zyklen steigt, mit denen er gemeinsame partial relations hat. Um zu entscheiden, ob Zyklus j durch $i \oplus j$ ersetzt werden soll (auch wenn keine Reduktion der Zykluslänge erzielt werden kann), vergleicht man $prob(j)$ mit $prob(i \oplus j)$. Um die Berechnung von

$$diff := prob(j) - prob(i \oplus j) = \sum_{e \in edges(j)} h(e) - \sum_{e \in edges(i \oplus j)} h(e) \qquad (4.4)$$

zu vereinfachen, benutzt man die Gleichungen

$$edges(j) = common(j,i) \cup rem(j,i) = common(i,j) \cup rem(j,i) \ , \qquad (4.5)$$

$$\begin{aligned} edges(i \oplus j) &= (edges(i) \cup edges(j)) - common(i,j) \\ &\overset{(4.5)}{=} (common(i,j) \cup rem(i,j) \cup rem(j,i)) - common(i,j) \\ &= rem(i,j) \cup rem(j,i) \end{aligned} \qquad (4.6)$$

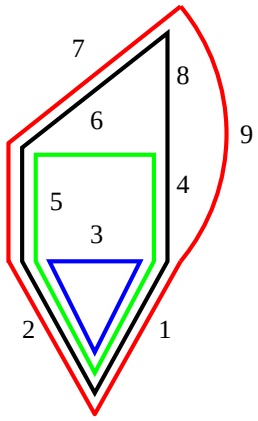
und berechnet

$$diff = \sum_{e \in common(i,j)} h(e) - \sum_{e \in rem(i,j)} h(e) \ .$$

Zur Berechnung von (4.4) muß man also lediglich $common(i,j)$ und $rem(i,j)$ kennen. Falls $diff < 0$ ist, ersetzt man Zyklus j durch $i \oplus j$.
Man sagt: *Zyklus i ändert Zyklus j.*

4.3 Beispiel

Ausgangspunkt dieses Beispiels ist eine Basis der unabhängigen Zyklen, die vom Graphenalgorithmus der Quadruple Large Prime Variation berechnet wurde. Die partial relations sind von 1 bis 9 durchnumeriert in der Reihenfolge, in der sie in den Graph eingefügt wurden. Der ursprüngliche Graphenalgorithmus berechnete daraus die 4 Zyklen (i, j, k und l). Die einzelnen Zyklen sind in den Abbildungen farbig dargestellt. Am Anfang sind insgesamt 19 Kanten am Aufbau der Basis beteiligt. An der Ausgangssituation kann man auch sehr schön die Beobachtung überprüfen, daß sehr viele Kanten in mehreren Zyklen vorkommen (vgl. Abschnitt 4.2).

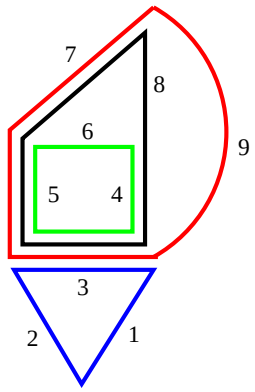


Ausgangssituation:

Zyklus	partial relations	Länge	Farbe
i	$\{1, 2, 3\}$	3	blau
j	$\{1, 2, 4, 5, 6\}$	5	grün
k	$\{1, 2, 4, 5, 7, 8\}$	6	schwarz
l	$\{1, 2, 5, 7, 9\}$	5	rot

Zyklus i kann alle anderen Zyklen reduzieren.

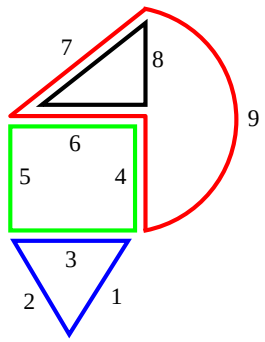
Die durchgeführten Operationen $j = i \oplus j, k = i \oplus k$ und $l = i \oplus l$ führen zu einer Reduzierung der Länge der Basis von 19 auf 16.



Nach Reduzierung durch Zyklus i:

Zyklus	partial relations	Länge	Farbe
i	$\{1, 2, 3\}$	3	blau
j	$\{3, 4, 5, 6\}$	4	grün
k	$\{3, 4, 5, 7, 8\}$	5	schwarz
l	$\{3, 5, 7, 9\}$	4	rot

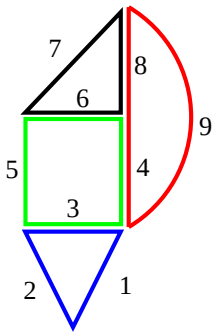
Zyklus j reduziert Zyklus k und ändert Zyklus l.



Nach $k = j \oplus k$ und $l = j \oplus l$:

Zyklus	partial relations	Länge	Farbe
i	$\{1, 2, 3\}$	3	blau
j	$\{3, 4, 5, 6\}$	4	grün
k	$\{6, 7, 8\}$	3	schwarz
l	$\{4, 6, 7, 9\}$	4	rot

Die durchgeführte Änderung $l = j \oplus l$ ist eine notwendige Voraussetzung für den letzten Reduzierungsschritt.



Nach $l = k \oplus l$:

Zyklus	partial relations	Länge	Farbe
i	$\{1, 2, 3\}$	3	blau
j	$\{3, 4, 5, 6\}$	4	grün
k	$\{6, 7, 8\}$	3	schwarz
l	$\{4, 8, 9\}$	3	rot

Am Ende der Reduzierung ist die Länge der Basis von 19 auf 13 geschrumpft.

4.4 Der Algorithmus

Nachdem ich die Idee des Algorithmus beschrieben habe, gehe ich nun auf Probleme ein, die bei der Umsetzung zu einer effizienten Implementierung auftauchen.

4.4.1 Vorberechnungen

Zuerst beschäftige ich mich mit einer Klassifikation der Zyklen. Um die Laufzeit des Algorithmus zu minimieren, ist es erforderlich, diejenigen Zyklen zu erkennen, die nicht mehr reduziert werden können. Ich werde diese Zyklen als *inaktiv* bezeichnen und sie aus der Menge $\mathcal{C}$ löschen. Das Löschen der inaktiven Zyklen hat Auswirkungen auf die Funktionswerte von *cyc* und h (vgl. Definition 4.1) für alle partial relations, die in inaktiven Zyklen vorkommen. So kann es zu mehr solitary relations in anderen Zyklen kommen, was zu besseren Entscheidungen beim Ändern von Zyklen führt, da multiple relations von inaktiven Zyklen nicht mehr berücksichtigt werden (vgl. Berechnung von *prob* in Abschnitt 4.2.2). Im allgemeinen ist es schwierig zu entscheiden, ob ein Zyklus reduziert wird oder nicht. Trotzdem helfen die folgenden Beobachtungen, einige inaktive Zyklen zu finden und eine Klassifizierung aller Zyklen anzugeben:

- Ein Zyklus, der nur aus solitary relations besteht, ist inaktiv.
- Ein Zyklus, der mehr solitary als multiple relations enthält, kann keinen Zyklus reduzieren oder ändern.
- Ein Zyklus, der nur eine multiple relations enthält, kann nicht mehr reduziert werden und kann auch keinen anderen Zyklus reduzieren. Er kann höchstens geändert werden oder einen anderen Zyklus ändern. Dies hätte jedoch keine großen Auswirkungen auf die restlichen Zyklen.

Mit Hilfe dieser Beobachtungen kann man für jeden Zyklus seinen Status berechnen, der besagt, ob der Zyklus inaktiv ist (Status 0), nur reduziert oder geändert werden kann (Status 1) oder auch selber reduzieren oder ändern kann (Status 2). (Man beachte dabei, daß ein Zyklus mit Status 2 auch selber reduziert oder geändert werden kann.) Am Anfang wird der Status für jeden Zyklus nach folgender Vorschrift berechnet:

$$status(i) = \begin{cases} 0 & \text{falls } len(i) - one\_entry(i) \leq 1 \ , \\ 2 & \text{falls } len(i) - one\_entry(i) > 1 \ \wedge \ 2 \cdot one\_entry(i) \leq len(i) \ , \\ 1 & \text{sonst .} \end{cases}$$

Dabei wird benutzt, daß die Anzahl der multiple relations als Differenz der Länge eines Zyklus und der Anzahl der solitary relations berechnet werden kann ($len(i) - one\_entry(i)$). Weiterhin kann getestet werden, ob ein Zyklus mehr solitary relations als multiple relations enthält, indem man testet, ob $2 \cdot one\_entry(i) \leq len(i)$ ist. Der Status eines Zyklus hängt also nur von seiner Länge und der Anzahl seiner solitary relations ab. Somit muß der Status eines Zyklus auch nur dann neu berechnet werden, wenn dieser Zyklus an einer Reduzierung oder Änderung beteiligt war, d.h. wenn sich seine Länge oder die Anzahl seiner solitary relations geändert haben kann. Als ersten Schritt (Vorberechnung) ermittelt der Algorithmus die Strukturinformationen der Zyklen und berechnet deren Status. Er liest die Zyklen ein und bestimmt ihre Länge (Schritt $1-3$). Dann wird gezählt, in wievielen Zyklen jede partial relation auftaucht (Schritt $4-6$). Mit Hilfe dieser Information kann dann für jeden Zyklus die Anzahl seiner solitary relations und damit auch sein Status bestimmt werden (Schritt $10-12$). Um sehr schnell testen zu können, ob ein Zyklus einen anderen Zyklus ändern oder reduzieren kann, ermittelt die Vorberechnungsphase auch für jede multiple relation a die Menge von Zyklen ($cyc(a)$), in denen sie auftaucht (Schritt $7-9$).

Vorberechnungen

(1) **foreach** (Zyklus i) **do**
(2) Lese Zyklus i ein und ermittele $len(i), edges(i)$;
(3) **od**
(4) **foreach** (partial relation a) **do**
(5) Berechne $h(a)$;
(6) **od**
(7) **foreach** (multiple relation a) **do**
(8) Bestimme $cyc(a)$;
(9) **od**
(10) **foreach** (Zyklus i) **do**
(11) Berechne $one\_entry(i) = |\{a | a \in edges(i) \wedge h(a) = 1\}|$ und $status(i)$;
(12) **od**

4.4.2 Test der Reduzierungsbedingungen

Nach den Vorberechnungen sucht der Algorithmus nach Zyklen, die andere Zyklen reduzieren oder ändern, bis es keine Zyklen mit Status 2 mehr gibt. Aus Abschnitt 4.2.1 ist bekannt, daß Zyklus i den Zyklus j reduziert (bzw. ändern kann), falls

$$|common(i,j)| \geq |rem(i,j))| \quad (\text{und } len(i) \leq len(j)).$$

Da die Länge jedes Zyklus bekannt ist und wir die Gleichungen

$$\begin{aligned} len(i) &= |common(i,j)| + |rem(i,j)| \quad \text{für alle Zyklen } i,j \text{ und} \\ len(i) - 2 \cdot |common(i,j)| \leq 0 &\iff |common(i,j)| \geq |rem(i,j))| \end{aligned} \tag{4.7}$$

benutzen, braucht man nur den common part von Zyklus i und j (bzw. dessen Mächtigkeit) zu berechnen, um zu entscheiden, ob Zyklus i Zyklus j reduziert (oder umgekehrt). In Abhängigkeit von der Situation benutzt die hier vorgestellte Implementierung zwei verschiedene Methoden zur Berechnung des common part. Die erste Version (**check** (i,j)) berechnet zu zwei gegebenen Zyklen i,j die Anzahl der partial relations, die in beiden Zyklen vorkommen. Dabei ist die Menge der partial relations eines Zyklus i ($edges(i)$) als sortiertes Array implementiert. Aus der Mächtigkeit des common parts bestimmt der Algorithmus dann $len(i) - 2 \cdot |common(i,j)|$ (vgl. (4.7)).

check (i,j)

EINGABE: Zyklus i und j.
AUSGABE: $len(i) - 2 \cdot |common(i,j)|$.

```
(1)  len1 = len(i);   len2 = len(j);
(2)  value = len(i);
(3)  ptr1 = edges[i];   ptr2 = edges[j];          /* Pointer auf relations */
(4)  while ((len1 > 0) ∧ (len2 > 0)) do
(5)    if (*ptr1 < *ptr2) then
(6)      len1--;   ptr1++;
(7)    else
(8)      if (*ptr1 > *ptr2) then
(9)        len2--;   ptr2++;
(10)     else
(11)       len1--;   ptr1++;
(12)       len2--;   ptr2++;
(13)       value -= 2;
(14)     fi
(15)   fi
(16) od
(17)  return (value)
```

Im allgemeinen ist man jedoch an allen Zyklen interessiert, die von einem Zyklus i reduziert oder geändert werden können. Deshalb berechnet die zweite Version (**test(i)**) mit Hilfe der Menge $cyc(a)$, die in den Vorberechnungen ermittelt wurde, die Anzahl der partial relations, die jeder Zyklus mit einem bestimmten Zyklus i gemeinsam hat. Bei dieser Berechnung genügt es, die multiple relations von Zyklus i zu betrachten, da solitary relations nicht im common part vorkommen können. Für jede multiple relation a enthält die Menge $cyc(a)$ alle Zyklen, in denen a vorkommt. Möchte man die Mächtigkeit des common parts aller Zyklen mit einem bestimmten Zyklus i bestimmen, so initialisiert man nun zuerst für jeden Zyklus einen Zähler mit 0. Dann durchläuft man für alle multiple relations a des Zyklus i die Menge $cyc(a)$ und inkrementiert den Zähler jedes vorkommenden Zyklus. Hat man dies getan, so enthält der Zähler für alle Zyklen j ($j \neq i$) die Anzahl der partial relations, die in Zyklus i und j vorkommen (also $|common(i,j)|$).

test (i)

EINGABE: Zyklus i.
AUSGABE: $cand_{reduce}$, $cand_{change}$

```
(1)  len1 = len(i);
(2)  foreach (Zyklus j) do
(3)     counter[j] = 0;
(4)  od
(5)  ptr = edges[i];                          /* Pointer auf relation  */
(6)  cand_change = ∅; cand_reduce = ∅;
(7)  while (len1-- > 0) do
(8)     if (h(*ptr) > 1) then                 /* multiple relation ?   */
(9)        foreach (Zyklus j, j ∈ cyc(*ptr)) do
(10)          counter[j]++;
(11)       od
(12)    fi
(13)    ptr++;
(14) od
(15) foreach (Zyklus j, j ≠ i) do
(16)    if (len(i) − 2 · counter[j] == 0) then
(17)          cand_change+={j};
(18)    else
(19)       if (len(i) − 2 · counter[j] < 0) then
(20)             cand_reduce+={j};
(21)       fi
(22)    fi
(23) od
(24)  return (cand_change, cand_reduce)
```

Diese Information wird dann zum Aufbau der Menge $cand_{change}$ und der Menge $cand_{reduce}$ benutzt. Dabei enthält die Menge $cand_{reduce}$ alle Zyklen, die von Zyklus i reduziert werden können, und die Menge $cand_{change}$ alle Zyklen, die von Zyklus i geändert werden können.

4.4.3 Variationen des Algorithmus

Durch die beiden im letzten Abschnitt vorgestellten Berechnungen des common part sind fast alle kritischen Phasen für die Effizienz des Algorithmus bereits beschrieben. In diesem Abschnitt beschäftige ich mich hauptsächlich mit der Reihenfolge, in der Reduzierungen durchgeführt werden sollen. Außerdem muß sichergestellt werden, daß der Algorithmus terminiert.

Um keine unnötigen Tests zu machen und die Terminierung sicherzustellen, ist es notwendig, den Status von Zyklen zu ändern, mit denen bereits getestet wurde, ob sie andere Zyklen reduzieren können. Wir erweitern die Statusdefinition (vgl. Seite 44) um den Status 3 für Zyklen, die bereits getestet wurden. Immer wenn der Aufbau eines Zyklus verändert wird, aktualisiert der Algorithmus seine Anzahl von solitary relations und pflegt die Mengen *cyc* (vgl. Definition 4.1). Wird ein Zyklus j durch $i \oplus j$ ersetzt, kann es sein, daß ein Status 3 Zyklus (ein Zyklus, der bereits getestet wurde) den "neuen" Zyklus reduzieren oder ändern kann. Ein schematisches Beispiel dafür zeigt die Abbildung 4.2. In der Ausgangssituation (1) kann Zyklus k (blau dargestellt) weder Zyklus i (grün dargestellt) noch Zyklus j (schwarz dargestellt) reduzieren. Deshalb erhält er nach dem Test den Status 3. Zyklus i kann aber Zyklus j reduzieren, was zur Situation führt, wie sie in Bild 2 dargestellt ist. Nun kann der bereits getestet Zyklus k den "neuen" Zyklus j reduzieren, was dann zur Situation in Bild 3 führt. Damit man nicht auf diese Reduzierung verzichten muß oder den Status

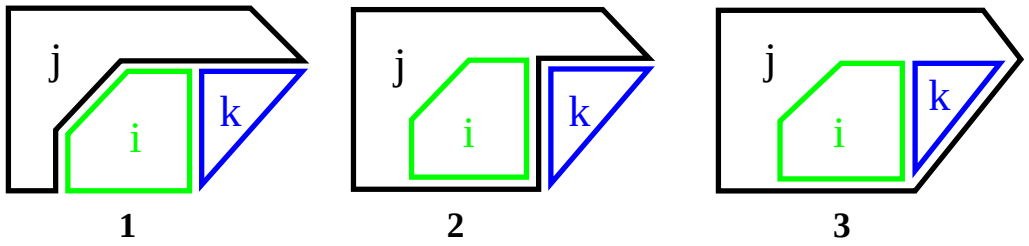


Abbildung 4.2: Beispiel für rekursive Effekte

aller Zyklen mit Status 3, die eine gemeinsame partial relation mit einem "neuen" Zyklus haben, zurücksetzen muß, untersuche ich die notwendigen Bedingungen für eine Reduzierung in diesem Fall genauer. Man betrachtet die Bedingungen, unter denen ein Zyklus k einen Zyklus $i \oplus j$ reduziert (oder ändert) und den Zyklus j nicht. In dieser Situation gilt:

$$2 \cdot |edges(k) \cap edges(i \oplus j)| \geq len(k) \text{ und } 2 \cdot |edges(k) \cap edges(j)| < len(k),$$

woraus folgt, daß

$$|edges(k) \cap edges(i \oplus j)| \quad > \quad |edges(k) \cap edges(j)| \ .$$

Mit (4.5), (4.6) erhält man

$$|edges(k) \cap (rem(i,j) \cup rem(j,i))| > |edges(k) \cap (common(j,i) \cup rem(j,i))| \ .$$

Da die Schnittmengen

$$rem(i,j) \cap rem(j,i) = (edges(i) - common(i,j)) \cap (edges(j) - common(j,i))$$

und

$$common(j,i) \cap rem(j,i) = common(j,i) \cap (edges(j) - common(j,i))$$

leer sind und

$$|edges(k) \cap (rem(i,j) \cup rem(j,i))| = |edges(k) \cap rem(i,j)| + |edges(k) \cap rem(j,i)| \ ,$$

$$\begin{aligned} |edges(k) \cap (common(j,i) \cup rem(j,i))| \quad &= \quad |edges(k) \cap common(j,i)| + \\ &\quad\ |edges(k) \cap rem(j,i)| \end{aligned}$$

gilt, folgt damit, daß in dieser Situation

$$|edges(k) \cap rem(i,j)| \quad > \quad |edges(k) \cap common(j,i)| \quad \geq \quad 0 \ \text{ ist} \ .$$

Deshalb kann man eine schärfere Aussage über die Eigenschaften eines Zyklus k machen, der einen Zyklus $i \oplus j$ reduziert (oder ändert) und den Zyklus j nicht. Es genügt nur diejenigen Status 3 Zyklen, die mindestens eine partial relation aus dem remaining part von i und j haben, nochmals zu testen. In der sogenannten *rekursiven Variante* des Algorithmus wird dann die Funktion $check(k, i \oplus j)$ benutzt, um zu testen, ob der "neue" Zyklus von Zyklus k reduziert (oder geändert) werden kann. Die praktischen Erfahrungen mit dem Algorithmus haben gezeigt, daß man nur minimal bessere Ergebnisse mit der rekursiven Variante erzielen kann, jedoch bis zu 4 mal mehr Rechenzeit benötigt. Dabei hängt der Mehraufwand an Rechenzeit sehr stark von der Anzahl der Zyklen ab, die die Basis bilden.

Ein weiterer kritischer Punkt ist die Bestimmung der Reihenfolge, in der die möglichen Reduzierungen durchgeführt werden sollen, so daß die erzielte Reduzierung maximal wird. Um die rekursiven Effekte und damit auch die Laufzeit zu minimieren, beginnt der Algorithmus mit den Reduzierungen, die wahrscheinlich zu den kleinsten Veränderungen der Basis führen. Dies führt sehr schnell zu einem Anhaltspunkt, welche Reduzierung man erreichen kann und wird als Variante I bezeichnet. Man kann minimal bessere Ergebnisse erzielen, wenn man mit denjenigen Reduzierungen startet, die zu den größten Veränderungen der Basis führen. Dies wird im folgenden als Variante II bezeichnet. Um eine Vorhersage über die Änderung der Basis durch einen Reduzierungsschritt zu machen, benutze ich die Differenz von

multiple und solitary relations. Diese Differenz kann als Differenz der Länge des Zyklus und der doppelten Anzahl von solitary relations berechnet werden, da

$$\Big|\{a|a \in edges(i) \wedge h(a) > 1\}\Big| - \Big|\{a|a \in edges(i) \wedge h(a) = 1\}\Big| = len(i) - 2 \cdot one\_entry(i)$$

gilt. Will man den Effekt auf die Basis minimieren (maximieren), den eine Reduzierung (bzw. Änderung) mit einem Zyklus hat, so muß diese Differenz so klein (groß) wie möglich sein. Der Algorithmus benutzt eine Schranke, die entscheidet, mit welchen Status 2 Zyklen der Test ($test(i)$) durchgeführt wird. In Abhängigkeit von der Variante, die benutzt wird, setzt man diese Schranke auf 0 oder die maximale Länge aller Zyklen. Dann werden alle Status 2 Zyklen getestet, deren Differenz der multiple und solitary relations höchstens (mindestens) so groß wie diese Schranke ist. Gibt es keine Status 2 Zyklen mit dieser gewünschten Eigenschaft, so wird die Schranke erhöht (erniedrigt), bis es überhaupt keine Status 2 Zyklen mehr gibt.

Der gesamte Algorithmus für Variante I sieht dann folgendermaßen aus:

Algorithmus cycle_reduce
Der Algorithmus berechnet eine Basis, deren Länge kleiner ist als die der ursprünglich berechneten Basis.

EINGABE: Menge $\mathcal{C}$, die von Graphenalgorithmus berechnet wurde.
AUSGABE: Modifizierte Menge $\mathcal{C}$.

SCHRITT 1: VORBERECHNUNGEN

(1) Algorithmus Vorberechnungen, Seite 44

SCHRITT 2: HAUPTTEIL

```
(2)  bound = 0;
(3)  while (∃ Zyklus i mit status(i) = 2) do
(4)    foreach (Zyklus i mit status(i) = 2 ∧
                  len(i) − 2 · one_entry(i) ≤ bound) do
(5)      test(i);
(6)      foreach (Zyklus j ∈ cand_reduce) do
(7)        j := i ⊕ j;
(8)      od
(9)      foreach (Zyklus j ∈ cand_change) do
(10)       if ( Σ_{e∈rem(i,j)} h(e) > Σ_{e∈com(i,j)} h(e)) then
(11)         j := i ⊕ j;
(12)       fi
(13)     od
```

```
(14)       status(i)= 3;
(15)    od
(16)    bound++;
(17) od

SCHRITT 3: AUSGABE

(18) foreach (Zyklus i) do
(19)       Ausgabe von edges(i);
(20) od
```

4.5 Praktische Ergebnisse

In diesem Abschnitt beschreibe ich praktische Ergebnisse, die mit der Implementierung der Erweiterung erzielt wurden. Es wurden sowohl Beispiele zur Reduzierung des Gewichtes von Matrizen des Number Field Sieve fürs Faktorisieren als auch zur Lösung von diskreten Logarithmen Problemen gerechnet.

Ich beginne mit Beispielen aus Faktorisierungsproblemen. Hier spielt die zweite Phase (das Lösen des Gleichungssystems) gegenüber dem Sammeln der Relationen (erste Phase) eine untergeordnete Rolle. Die Daten stammen aus Berechnungen von Jörg Zayer. Genauere Informationen darüber befinden sich in seiner Doktorarbeit [43]. Sie wurden durchgeführt, um einen Eindruck zu geben, wie lange es dauert, Zahlen in diesen Größenordnungen mit dem Number Field Sieve zu faktorisieren. Sobald die erste Phase genügend Relationen zum Aufbau des Gleichungssystems gesammelt hatte, wurde sie abgebrochen. Läßt man die erste Phase etwas länger laufen, so führt das zu dünner besetzten und überbestimmten Systemen, die leichter gelöst werden können. Wie bereits erwähnt spielt das Lösen der Gleichungssysteme bei der Lösung Diskreter Logarithmen Probleme eine wesentlich wichtigere Rolle als bei der Faktorisierung von ganzen Zahlen. Die Beispiele für DL-Probleme stammen von Damian Weber's Implementierung des Number Field Sieve für DL-Berechnungen ([39], [41]). Zu jedem Beispiel gebe ich die Anzahl der Zyklen, die von dem ursprünglichen Graphenalgorithmus berechnet wurden, die Länge dieser Basis sowie die maximale Länge eines Zyklus an. In Klammern werden diese Informationen für die minimale Anzahl von Zyklen aufgelistet, die zum Aufbau eines minimal überbestimmten Gleichungssystems benötigt werden. Die Tabellen geben dann die benötigten Laufzeiten für die verschiedenen Variationen des Algorithmus an, die in Abschnitt 4.4.3 beschrieben sind. Dabei werden die Zeiten ohne rekursive Variante (**I, II**) und mit ihr angegeben (**I rec., II rec.**). Die erzielten Reduzierungen werden für die Menge aller Zyklen und die Menge, der minimal für den Aufbau des Gleichungssystems benötigten Zyklen, angegeben. Beide Angaben sind wichtig, da Methoden existieren, die die Überbestimmtheit der Systeme ausnutzen, um diese schneller zu lösen.

Damit beschäftigt sich Kapitel 6. Alle Berechnungen wurden auf einer Sparc 20 mit 128 MB Hauptspeicher durchgeführt.

4.5.1 Beispiele aus Faktorisierungen

In diesem Abschnitt befinden sich Daten von durchgeführten Faktorisierungen einer 30-stelligen, einer 55-stelligen, einer 65-stelligen und einer 75-stelligen Zahl. Die Faktorisierung dieser Zahlen wurde mit der Quadruple Large Prime Variante des Number Field Sieve durchgeführt (siehe [43]).

Stellen	# Zyklen	$\sum$ Zykluslängen	Max. Zykluslänge
30	4 355 (1 304)	361 402 (36 572)	420 (47)

Variante	# eliminierte relations		Max. Zykluslänge		Laufzeit
	gesamt	benötigt	gesamt	benötigt	
I	69 896 (19.34 %)	5 960 (16.29 %)	384	37	0:41 min
II	132 836 (36.76 %)	7 876 (21.53 %)	298	33	3:06 min
I rec.	95 339 (26.38 %)	6 981 (19.08 %)	332	35	3:04 min
II rec.	136 651 (37.81 %)	7 960 (21.76 %)	306	33	13:02 min

Dieses Beispiel stellt eine Besonderheit dar. Es ist der einzige durchgeführte Test, bei dem mit rekursiver Variante deutlich bessere Ergebnisse als ohne sie erzielt wurden. Die folgenden Daten stammen von der Faktorisierung einer 55-stelligen Zahl. Hier wurde die Siebphase solange verlängert, bis mehr als dreimal mehr relations gefunden wurden als zum Aufbau des Systems unbedingt benötigt wurden. Eine Verlängerung der Siebphase führt zu deutlich kürzeren Zyklen. In diesem Beispiel konnte ein überbestimmte Gleichungssystem allein mit den Zyklen aufgestellt werden, die aus höchstens 4 relations aufgebaut sind. Diese Zyklen konnten mit Hilfe der hier vorgestellten Erweiterung nicht mehr reduziert werden.

Stellen	# Zyklen	$\sum$ Zykluslängen	Max. Zykluslänge
55	59 823 (13 859)	495 493 (40 784)	63 (4)

Variante	# eliminierte relations		Max. Zykluslänge		Laufzeit
	gesamt	benötigt	gesamt	benötigt	
I	37 973 (7.66 %)	0 (0.00 %)	42	4	6:46 min
II	38 122 (7.69 %)	0 (0.00 %)	41	4	10:31 min
I rec.	38 061 (7.68 %)	0 (0.00 %)	42	4	7:01 min
II rec.	38 140 (7.69 %)	0 (0.00 %)	41	4	10:01 min

Die Tabelle 4.1 zeigt die Auswirkungen der hier vorgestellten Erweiterung auf die Zyklen, die aus höchstens 15 relations aufgebaut sind. Zuerst wird (sortiert nach der Zykluslänge) die Anzahl der Zyklen aufgelistet, die vom ursprünglichen Graphenalgorithmus berechnet wurden. Die Spalten, die den einzelnen Varianten zugeordnet sind, geben die Anzahl der zusätzlichen Zyklen an, die durch eine Reduzierung der Zykluslänge entstanden sind. Da zum Aufbau des Gleichungssystems keine Zyklen

Zykluslänge	Anzahl der Zyklen				
	Original	**I**	**II**	**I rec.**	**II rec.**
2	4631	0	0	0	0
3	5390	0	0	0	0
4	5445	161	161	161	161
5	5754	383	383	383	383
6	5689	323	323	323	323
7	5281	487	488	489	489
8	4832	389	386	387	387
9	3996	429	434	430	434
10	3379	288	293	295	291
11	2771	154	158	154	159
12	2259	45	44	43	44
13	1862	-1	1	-1	0
14	1578	-181	-175	-178	-175
15	1202	-111	-111	-109	-110

Tabelle 4.1: Reduzierung der Zyklen (Faktorisierung, 55 Stellen)

mit größerer Länge als 4 benutzt wurden, und keine Reduzierung auf diesen Zyklen erzielt werden konnte, wurde die Länge der Basis, die zum Aufbau benutzt wurde, nicht verkürzt. Anhand dieser Tabelle kann man auch die kleinen Unterschiede zwischen den Varianten mit und ohne rekursiver Behandlung beobachten. Erst bei Zyklen der Länge 7 produziert die rekursive Variante 2 Zyklen mehr.

Stellen	# Zyklen	$\sum$ Zykluslängen	Max. Zykluslänge
65	24 692 (21 717)	495 596 (316 208)	214 (41)

Variante	# eliminierte relations		Max. Zykluslänge		Laufzeit
	gesamt	benötigt	gesamt	benötigt	
I	73 948 (14.93 %)	33 608 (10.62 %)	156	33	1:24 min
II	75 064 (15.16 %)	33 809 (10.69 %)	142	33	2:50 min
I rec.	74 532 (15.05 %)	33 750 (10.67 %)	142	33	1:34 min
II rec.	75 090 (15.16 %)	33 822 (10.69 %)	142	33	6:07 min

Im Gegensatz zu den bisherigen Beispielen, die als untypisch bezeichnet werden können, findet sich in den Beispielen der Faktorisierung einer 65 und 75-stelligen Zahl eine Situation, wie sie in fast allen untersuchten Fällen vorgefunden wurde. Hier benötigen die rekursiven Varianten deutlich mehr Zeit und führen zu nur leicht besseren Resultaten.

Stellen	# Zyklen	$\sum$ Zykluslängen	Max. Zykluslänge
75	55 457 (33 224)	7 004 103 (2 115 372)	853 (134)

Variante	# eliminierte relations		Max. Zykluslänge		Laufzeit
	gesamt	benötigt	gesamt	benötigt	
I	1 981 756 (28.29 %)	434 433 (20.53 %)	654	97	0:18:08 h
II	2 115 652 (30.21 %)	444 066 (20.99 %)	468	96	1:25:12 h
I rec.	2 058 057 (29.38 %)	441 641 (20.87 %)	545	97	0:53:57 h
II rec.	2 120 191 (30.27 %)	444 771 (21.02 %)	468	96	4:38:06 h

Die Tabelle 4.2 stellt für alle Beispiele die Gesamtzahl der verschiedenen relations ($|\mathcal{E}|$), die in den Zyklen vorkommen, der Anzahl der multiple relations gegenüber.

	30d	55d	65d	75d
# relations	14 760	135 928	99 907	334 863
# mult. relations	7 011	52 781	49 676	186 485
	47.5 %	38.8 %	49.7 %	55.7 %

Tabelle 4.2: Anzahl multiple relations

Eine mögliche Erklärung für die relativ geringen Reduzierungen in dem 55-stelligen Faktorisierungsbeispiel kann die Tatsache sein, daß die Anzahl der partial relations pro Zyklus vergleichsweise gering war. Im Durchschnitt sind in diesem Beispiel 8.28 relations am Aufbau eines Zyklus beteiligt. In den anderen Beispielen war diese Anzahl deutlich größer (82.98, 20.05 bzw. 126.29). Diese Erklärung wird auch durch ein weiteres Beispiel unterstützt. Betrachtet man eine Faktorisierung einer 65-stelligen Zahl, die nur mit Double Large Prime Variante und nicht mit der Quadruple Large Prime Variante faktorisiert wurde, so beträgt die durchschnittliche Anzahl von relations, die am Aufbau eines Zyklus beteiligt sind, nur 2.61. In diesem Beispiel waren die erzielten Reduzierungen sehr gering. In nur wenigen Sekunden ermittelte die hier vorgestellte Erweiterung, daß nur 0.0693% der relations eliminiert werden konnten. Daraus kann man schließen, daß der ursprüngliche Graphenalgorithmus für die Double Large Prime Variante relativ gut arbeitet und daß die hier vorgestellte Erweiterung erst nützlich bei Anwendung der Quadruple Large Prime Variante des Number Field Sieve ist.

4.5.2 Beispiele aus DL-Berechnungen

In diesem Abschnitt zeige ich ein paar Beispiele aus Berechnungen von DL-Problemen, die mit der Quadruple Large Prime Variante des Number Field Sieve zur Berechnung diskreter Logarithmen erstellt wurden. Unter den Beispielen befinden sich einige DL-Berechnungen, die Weltrekorde darstellten bzw. darstellen (vgl.[9]). Außerdem wurde der erste Schritt zur Lösung der sogenannten McCurley Challenge (vgl. [27]) durchgeführt.

Im ersten Beispiel werden die Daten einer DL-Berechnung modulo einer 50-stelligen Primzahl vorgestellt. Die bei dieser Berechnung gewonnenen Erkenntnisse flossen bereits in die Berechnungen modulo einer 65-stelligen Primzahl ein. Diese Berechnung stellte die erste Verbesserung des Weltrekords von Odlyzko und LaMacchia dar.

Stellen	# Zyklen	$\sum$ Zykluslänge	Max. Zykluslänge	mult. relat.
50	16 344 (3 491)	365 842 (22 398)	145 (10)	43.20 %

Variante	# eliminierte relations		Max. Zykluslänge		Laufzeit
	gesamt	benötigt	gesamt	benötigt	
I	102 168 (27.93 %)	788 (3.52 %)	107	9	0:48 min
II	105 355 (28.80 %)	795 (3.55 %)	84	9	1:57 min
I rec.	104 116 (28.46 %)	792 (3.54 %)	86	9	1:11 min
II rec.	105 645 (28.88 %)	795 (3.55 %)	84	9	2:12 min

Stellen	# Zyklen	$\sum$ Zykluslängen	Max. Zykluslänge	mult. relat.
65	23 759 (20 000)	828 336 (630 030)	55 (50)	47.66 %

Variante	# eliminierte relations		Max. Zykluslänge		Laufzeit
	gesamt	benötigt	gesamt	benötigt	
I	116 083 (14.01 %)	93 541 (14.84 %)	55	42	1:57 min
II	117 865 (14.22 %)	94 802 (15.04 %)	55	42	4:14 min
I rec.1	117 091 (14.14 %)	94 336 (14.97 %)	55	42	2:30 min
II rec.	118 110 (14.25 %)	94 984 (15.08 %)	55	42	5:42 min

Im nächsten Beispiel (einer DL-Berechnung modulo einer 75-stelligen Primzahl) möchte ich die Auswirkungen der Verlängerung der ersten Phase (Siebphase) auf den hier vorgestellten Algorithmus und das Gleichungssystem darstellen.
Es wurden 3 verschiedene Zeitpunkte während der Siebphase betrachtet. In allen drei Fällen waren bereits genügend relations zum Aufbau eines überbestimmten

Gleichungssystems gefunden. Trotzdem wurde weiter gesiebt, um kürzere Zyklen zu erhalten. An den drei verschiedenen Zeitpunkten wurden alle Zyklen untersucht, die aus weniger als 244 (185 bzw. 100) partial relations aufgebaut sind. Die Anzahl der untersuchten Zyklen betrug 62 702 (79 016 bzw. 75 832). Betrachtet man die maximale Zykluslänge eines Zyklus, der am Aufbau des Gleichungssystems beteiligt war, so sinkt sie bereits ohne Benutzung der Erweiterung von 140 beim ersten Testpunkt, über 94 beim zweiten auf 54 beim letzten Test.

Stellen	# Zyklen	$\sum$ Zykluslängen	Max. Zykluslänge	Anteil mul.rel.
75 I	62 702 (25 000)	9 446 618 (2 237 474)	243 (140)	50.10 %
75 II	79 016 (25 000)	9 096 547 (1 5110 69)	184 (94)	48.56 %
75 III	75 832 (25 000)	4 851 865 (884 067)	99 (54)	46.97 %

Man sieht deutlich, daß je kürzer die Zyklen selber werden, desto weniger Reduzierung kann mit der hier vorgestellten Erweiterungen erzielt werden. Dennoch kann selbst im letzten Test noch eine Reduzierung um fast 20% (bzw. 25%) erzielt werden.

Variante	# eliminierte relations		Max. Zykluslänge		Laufzeit
	gesamt	benötigt	gesamt	benötigt	
I	2 682 924 (28.40 %)	587 200 (26.24 %)	235	97	46:37 min
II	3 057 916 (32.37 %)	637 659 (28.50 %)	235	92	275:03 min
I rec.	2 873 269 (30.42 %)	619 561 (27.69 %)	235	94	181:43 min
II rec.	3 073 731 (32.54 %)	640 131 (28.61 %)	235	92	1393:45 min
I	2 626 785 (28.87 %)	359 337 (23.78 %)	181	66	46:59 min
II	2 882 224 (31.68 %)	377 902 (25.01 %)	181	64	273:30 min
I rec.	2 758 644 (30.32 %)	372 248 (24.63 %)	181	65	154:53 min
II rec.	2 894 857 (31.82 %)	379 001 (25.08 %)	181	64	1087:10 min
I	1 171 619 (24.14 %)	168 123 (19.02 %)	99	41	24:14 min
II	1 211 516 (24.97 %)	171 371 (19.38 %)	99	40	100:57 min
I rec.	1 193 356 (24.60 %)	170 567 (19.29 %)	99	41	46:38 min
II rec.	1 215 628 (25.05 %)	171 791 (19.43 %)	99	40	216:55 min

Das nächste Beispiel entstand beim Versuch, das von Kevin McCurley formulierte DL-Problem zu lösen [27]. Dabei handelt es sich um eine DL-Berechnung modulo einer 129-stelligen Primzahl, wobei es zu beachten gilt, daß diese Primzahl eine

spezielle Form hat, die bei den Berechnungen ausgenutzt werden kann. Für Details verweise ich auf [41]. Die Schwierigkeit dieses Problems entspricht etwa dem Problem diskrete Logarithmen modulo einer zufällig gewählten 85-stelligen Zahl zu berechnen. Auch hier wurden wieder zu drei verschiedenen Zeitpunkten der Siebphase Tests durchgeführt. Als Eingabe für die Erweiterung wurden alle Zyklen benutzt, die vom Graphenalgorithmus berechnet wurden und aus weniger als 150 (70, 42) partial relations aufgebaut sind.

Stellen	# Zyklen	$\sum$ Zykluslängen	Max. Zykluslänge	Anteil mul.rel.
126 I	79 286 (40 000)	6 067 318 (1 664 276)	149 (78)	53.76 %
126 II	64 383 (40 000)	2 549 535 (1 113 781)	69 (48)	51.18 %
126 III	94 180 (40 000)	2 330 832 (567 592)	41 (23)	47.17 %

Variante	# eliminierte relations		Max. Zykluslänge		Laufzeit
	gesamt	benötigt	gesamt	benötigt	
I	1 581 437 (26.06 %)	339 603 (20.41 %)	145	57	23:38 min
II	1 624 037 (26.76 %)	343 864 (20.66 %)	145	57	98:17 min
I rec.	1 604 705 (26.44 %)	342 959 (20.61 %)	145	57	40:30 min
II rec.	1 627 594 (26.82 %)	344 513 (20.70 %)	145	57	175:44 min
I	439 351 (17.23 %)	169 907 (15.25 %)	69	38	11:01 min
II	443 318 (17.38 %)	170 938 (15.34 %)	69	38	31:58 min
I rec.	441 343 (17.31 %)	170 552 (15.31 %)	69	38	12:57 min
II rec.	443 761 (17.40 %)	171 074 (15.36 %)	69	38	35:03 min
I	338 445 (14.52 %)	52 680 (9.28 %)	41	20	21:02 min
II	339 677 (14.57 %)	52 832 (9.31 %)	41	20	47:11 min
I rec.	339 289 (14.56 %)	52 790 (9.30 %)	41	20	23:06 min
II rec.	339 909 (14.58 %)	52 841 (9.31 %)	41	20	54:25 min

Je größer die Beispiele (und damit auch die Gleichungssysteme) werden, desto wichtiger wird es auch weiter zu sieben, auch wenn bereits überbestimmte Gleichungssysteme aufgestellt werden könnten. Aus dem aktuellen Weltrekord beim Berechnen DL in endlichen Primkörpern kommt das nächste Beispiel. Es handelt sich dabei um die Lösung eines DL-Problems modulo einer 85-stelligen Primzahl (DL85). In diesem Beispiel wurde wegen des enormen Platz- und Zeitbedarfs auf die Verwendung von partial relations mit vier large primes verzichtet. Als Eingabe für den originalen Graphenalgorithmus wurden nur partial relations mit höchstens drei large primes verwendet. An diesem Beispiel sieht man auch deutlich, wie der Anteil der multiple relations durch das Verlängern der Siebphase immer mehr zurück geht (vgl. auch vorherige Beispiele). Gleichzeitig sinkt auch die Anzahl der eliminierten relations.

Stellen	# Zyklen	∑ Zykluslängen	Max. Zykluslänge	Anteil mul.rel.
85 I	96 085 (60 000)	5 485 572 (1 918 703)	149 (66)	51.84 %
85 II	158 883 (60 000)	1 957 575 (428 330)	19 (11)	39.86 %

Variante	# eliminierte relations		Max. Zykluslänge		Laufzeit
	gesamt	benötigt	gesamt	benötigt	
I	1 085 907 (19.80 %)	266 696 (13.90 %)	146	53	27:28 min
II	1 097 684 (20.01 %)	267 687 (13.95 %)	146	52	91:50 min
I rec.	1 604 705 (26.44 %)	342 959 (20.61 %)	145	57	40:30 min
II rec.	1 627 594 (26.82 %)	344 513 (20.70 %)	145	57	175:44 min
I	110 357 (5.64 %)	11 468 (2.68 %)	19	11	57:58 min
II	110 450 (5.64 %)	11 467 (2.68 %)	19	11	81:31 min
I rec.	110 408 (5.64 %)	11 474 (2.68 %)	19	11	59:12 min
II rec.	110 468 (5.64 %)	11 476 (2.68 %)	19	11	78:00 min

Als letzten Test wurde für eine DL-Berechnung modulo einer 65-stelligen Primzahl das Gleichungssystem mit und ohne die hier vorgestellte Erweiterung aufgebaut. Dieser Test dient dazu nachzuweisen, daß mit der Anzahl der am Aufbau des Gleichungssystems beteiligten partial relations auch das Gewicht des Systems zurückgeht. Dabei wurde jeweils ein deutlich überbestimmtes System durch Löschen von schweren Spalten in ein knapp überbestimmtes System überführt. Es wurde danach die Anzahl der Einträge gezählt und verglichen. Dabei habe ich unterschieden zwi-

	ohne Erweiterung	mit Erweiterung
Dimension vorher	19 958 × 66 529	19 958 × 66 552
Dimension LGS	19 935 × 19 945	19 922 × 19 932
# +1-Einträge	3 054 508	2 504 303 (−18.01%)
# −1-Einträge	3 235 976	2 712 207 (−16.21%)
# *longs*	179 986	147 940 (−17.80%)

Tabelle 4.3: Gewichtsvergleich mit und ohne Erweiterung

schen Einträgen, die gleich +1 ,−1 oder vom Betrag her kleiner als 2^{31} sind (und somit mit Hilfe des Datentyps *long* dargestellt werden können). Die Reduktion auf der Menge der partial relations, die zum Aufbau benötigt werden, betrug ungefähr 19%. Bedenkt man, daß zum Aufbau des Gleichungssystems auch die free und full relations benutzt werden, bei denen ja keine Reduzierung durchgeführt werden kann, fallen die Rückgänge in den Einträgen wie erwartet aus (vgl. Tabelle 4.3).

Die Beispiele haben gezeigt, daß die Anzahl der am Aufbau eines Gleichungssystems beteiligten relations deutlich mit Hilfe der in diesem Kapitel vorgestellten Erweiterung der Quadruple Large Prime Variante reduziert werden kann. Als Beleg dafür, daß mit der Reduzierung der Einträge auch eine Reduzierung der Laufzeit des Lanczos Verfahrens verbunden ist, betrachte ich die DL-Berechnung modulo einer 75-stelligen Primzahl. Die Anzahl der relations, die am Aufbau des Systems beteiligt sind, konnte mit der Erweiterung um 22.5% reduziert werden. Ein Vergleich der Laufzeiten zum Lösen des Systems mit und ohne die Erweiterung bei 7 berechneten Lösungen zeigt eine Reduzierung um 18% . (Dabei muß man beachten, daß die Laufzeit nicht allein durch die Anzahl der Einträge bestimmt wird (vgl. Abschnitt 2.3).)

Die benötigte Rechenzeit für die Erweiterung ist im Vergleich zur benötigten Zeit zum Lösen des Gleichungssystems relativ gering (vor allem bei DL-Berechnungen). Leider kann man keine Aussagen darüber machen, wie gut der hier vorgestellte Algorithmus ist, d.h. wie optimal die von ihm berechnete Basis ist. Gerade beim Lösen von DL-Problemen hat sich die Erweiterung jedoch als äußerst nützlich erwiesen.

Kapitel 5

Implementierung des Lanczos Verfahren

In diesem Kapitel beschreibe ich zuerst die sequentielle Version meiner Lanczos Implementierung. Darauf aufbauend beschreibe ich die parallele Implementierung auf einer MIMD (Multiple Instruction Multiple Data) Maschine, einer Intel Paragon XP/S 10. Dabei stelle ich diesen Rechner auch kurz vor. Im Rahmen des HLRZ-Forschungsprojekts Diskrete Logarithmen war es mir möglich, auf diesem Rechner zu arbeiten. Die Maschine gehört zum Forschungszentrum in Jülich.
Nach einer kurzen Darstellung und Motivation der Ziele, die mit diesen Implementierungen verfolgt wurden, werden die Laufzeiten für grundlegende Rechenoperationen in endlichen Primkörpern untersucht. Dabei wird auf sog. Langzahlarithmetiken aufgebaut, die das Rechnen mit Zahlen aus $\mathbb{Z}$ (sog. *langen Zahlen*) auch dann ermöglichen, wenn diese wegen ihrer Größe nicht in elementaren Datentypen abgespeichert werden können. Dann werden die grundlegenden Operationen, die zur Implementierung des Lanczos Verfahrens benötigt werden, sowie deren Realisierung beschrieben. Bevor ich eine genaue Laufzeitanalyse durchführe, stelle ich noch eine Idee zur Beschleunigung der Berechnung der (partiellen) Lösungen vor. Aufbauend auf der Laufzeitanalyse entwickele ich dann ein Modell, das Vorhersagen zum Laufzeitverhalten des Verfahrens ermöglicht. Dieses Modell wird auch dazu benutzt, bei Veränderungen des zu lösenden Systems deren Auswirkung auf das Laufzeitverhalten des Verfahrens vorherzusagen. Das Modell wird mit der sequentiellen Version anhand einiger Berechnungen getestet. Dann folgt die Beschreibung der parallelen Implementierung. Nach einer kurzen Beschreibung des benutzten Parallelrechners stelle ich dann meinen Ansatz zur Parallelisierung des Lanczos Verfahrens vor. Dabei stellt die gleichmäßige Auslastung der Knoten und die Realisierung des Austauschs von Daten einen Schwerpunkt dar. Die erreichte komplette Überlagerung der Kommunikation durch Berechnungen stellt eine wichtige Voraussetzung für den linearen Speedup der parallelen Implementierung dar. Das entwickelte Laufzeitmodell wird dann auch mit der parallelen Implementierung getestet. Das Ende dieses Kapitels bilden einige technische Anmerkungen zu den Implementierungen.

5.1 Ziel der Implementierung

Aufbauend auf den Arbeiten von Peter Montgomery [29] und der Coppersmith Block-Lanczos Implementierung von Olaf Groß [19] wurde eine sehr schnelle Implementierung für $\mathbb{F}_2$ (Montgomery Block-Lanczos) entwickelt, auf die hier aber nicht näher eingegangen wird. Damit ist das Lösen von Gleichungssystemen behandelt, die bei Faktorisierungen ganzer Zahlen auftreten. Im Fall des Lösens von DL-Problemen über Primkörpern $\mathbb{F}_p$ mit kleiner Charakteristik ($p < 10^{15}$) benutzt man den Algorithmus von Silver-Pohlig-Hellmann (vgl. [27]), wobei kein Gleichungssystem gelöst werden muß. Somit war das Ziel dieser Implementierung, Gleichungssysteme über endlichen Primkörpern $\mathbb{F}_p$ mit großer Charakteristik ($p > 10^{15}$) zu lösen. Die hier vorgestellten Implementierungen wurden erfolgreich zum Lösen der Gleichungssysteme eingesetzt, die bei den ersten Vorberechnungen zum Lösen der McCurley Challenge [27] und einigen Verbesserungen des Weltrekords bei der Berechnung diskreter Logarithmen in endlichen Primkörpern aufgetreten sind. Dabei lag der Schwerpunkt wegen der hohem Rechenleistung der Paragon auf der parallelen Version. Die teilweise schlechte Verfügbarkeit und das limitierte Rechenzeitkontingent des Parallelrechners führten jedoch auch zur Weiterentwicklung der sequentiellen Version.

5.2 Rechnen in endlichen Primkörpern

Die grundlegenden Rechenoperationen auf ganzen Zahlen werden von Langzahlarithmetiken bereitgestellt. Darauf aufbauend werden die Rechenoperationen in endlichen Primkörpern realisiert, indem, falls es nötig ist, nach Ausführung einer Operation das Ergebnis reduziert wird. Die sequentielle Version meiner Lanczos Implementierung verwendet die von Ralf Dentzer entwickelte *libI*. Darin ist ein Assembler Kernel für SPARC RISC Prozessoren enthalten. Die Assembler Routinen beschleunigen die Operationen deutlich gegenüber der reinen C Version. Die von Arjen Lenstra entwickelte Bibliothek *LIP* ist in reinem C geschrieben, jedoch sehr effizient implementiert und universell einsetzbar. Verglichen mit der Assembler Version der *libI* auf SPARC 20 Maschinen ist sie langsamer, auf der Paragon jedoch ist sie schneller als die reine C Version der *libI* und wird deshalb auch dort eingesetzt.
Da die Multiplikation modulo eines festen ungeraden Moduls mit Hilfe von Montgomery Arithmetik [30] auf Zahlen in Montgomerydarstellung schneller als die "normale" Multiplikation mit anschließender Reduktion durchgeführt werden kann, ist es sinnvoll, Montgomerydarstellung von Zahlen zu verwenden. Die von den Implementierungen benutzten Langzahlarithmetiken haben beide Funktionen für die Montgomeryarithmetik eingebaut. Zusätzlich wurden jeweils spezielle neue Funktionen entwickelt, die für den Spezialfall, in dem sie in meiner Implementierung eingesetzt werden, optimiert wurden.
Beginnend mit der *libI*-Bibliothek liste ich die Laufzeiten für Operationen in endlichen Primkörpern ($\mathbb{F}_p,\ p \in \mathbb{P}$) auf. Die Zeiten wurden in Sekunden auf einer SPARC 20 gemessen. Die Tabelle 5.1 enthält für unterschiedlich große Moduli p

($p \approx 10^{65}, 10^{75}, 10^{85}, 10^{126}$) die Zeiten, die für eine Addition, Subtraktion, Multiplikation zweier Elemente aus $\mathbb{F}_p$ benötigt werden, sowie die Zeit, die für die Invertierung eines Elements aus $\mathbb{F}_p$ benötigt wird. Außerdem sind die Zeiten angegeben, die für die Multiplikation (*kmult*) eines Elements aus $\mathbb{F}_p$ mit einer Zahl kleiner als 2^{31} (die somit in den elementaren Datentyp *long* paßt). Das Ergebnis ist wiederum eine Zahl aus $\mathbb{F}_p$. Dabei wird für die Multiplikation jeweils die Laufzeit bei Benutzung von Montgomerydarstellung und "normaler" Berechnung mit anschließender Reduktion verglichen. Zum Vergleich sind die Zeiten für die Addition und Subtraktion derselben Zahlen (jedoch nicht in Montgomerydarstellung) in $\mathbb{Z}$ angegeben. Auf

Operation	**Anzahl**	**65**	**75**	**85**	**126**
Addition (Montgomery)	$12 \cdot 10^6$	17.43	21.10	22.65	27.60
Addition in $\mathbb{Z}$	$12 \cdot 10^6$	15.17	16.46	17.54	23.69
Subtraktion(Montgomery)	$12 \cdot 10^6$	15.25	16.15	18.47	23.67
Subtraktion in $\mathbb{Z}$	$12 \cdot 10^6$	15.25	16.50	17.62	23.53
Multiplikation (Montgomery)	$6 \cdot 10^5$	11.09	14.07	17.33	34.48
Multiplikation ("Normal")	$6 \cdot 10^5$	20.17	25.97	27.92	48.03
Invertierung ("Montgomery")	$2 \cdot 10^4$	19.81	25.25	31.76	64.72
kmult (Montgomery)	$1.2 \cdot 10^6$	14.92	19.93	16.90	23.01
kmult ("Normal")	$1.2 \cdot 10^6$	12.66	15.64	13.93	20.13

Tabelle 5.1: Laufzeiten Montgomerydarstellung (SPARC20 mit libI)

der Paragon in Jülich wurden dieselben Tests mit der *LIP* gemacht. Die Tabelle 5.2 enthält die gemessenen Zeiten in Sekunden. In der *LIP* gibt es zusätzlich noch eine Funktion zum Quadrieren von Zahlen.

Operation	**Anzahl**	**65**	**75**	**85**	**126**
Addition (Montgomery)	$28 \cdot 10^5$	26.74	24.06	29.70	43.43
Addition in $\mathbb{Z}$	$28 \cdot 10^5$	16.78	17.68	22.68	27.55
Subtraktion (Montgomery)	$28 \cdot 10^5$	30.87	30.62	34.90	51.32
Subtraktion in $\mathbb{Z}$	$28 \cdot 10^5$	21.71	22.34	27.21	32.09
Multiplikation (Montgomery)	$15 \cdot 10^4$	16.79	19.88	23.13	44.99
Multiplikation ("Normal")	$15 \cdot 10^4$	24.26	27.85	34.11	60.03
Quadrieren (Montgomery)	$15 \cdot 10^4$	15.77	18.44	21.22	39.69
Quadrieren ("Normal")	$15 \cdot 10^4$	23.17	27.41	30.74	53.59
Invertierung	$5 \cdot 10^3$	14.02	16.02	18.43	29.01
kmult (Montgomery)	$4 \cdot 10^5$	15.85	17.11	18.49	23.19

Tabelle 5.2: Laufzeiten Montgomerydarstellung (Paragon mit LIP)

Bei der Addition und Subtraktion in $\mathbb{F}_p$ in Montgomeryarithmetik werden die beiden Zahlen in Montgomerydarstellung über $\mathbb{Z}$ miteinander addiert bzw. subtrahiert

und dann wird getestet, ob eine Subtraktion (bzw. Addition) des Moduls erforderlich ist. Dies kann dazu führen, daß eine Addition (Subtraktion) in $\mathbb{F}_p$ fast zweimal so lange dauern kann als dieselbe Operation über $\mathbb{Z}$. Ein Vorteil der Montgomerydarstellung, auf den ich später noch genauer eingehen werde, ist jedoch die Tatsache, daß nur mit nicht negativen Zahlen gerechnet wird. Über $\mathbb{Z}$ muß jedesmal überprüft werden, welches Vorzeichen die Argumente haben. Die teilweise geringen Unterschiede zwischen den Operationen in $\mathbb{Z}$ und in Montgomerydarstellung beruhen auf der Ausnutzung dieser Informationen sowie auf der Tatsache, daß nicht bei jeder Operation eine Reduktion notwendig ist. In beiden Langzahlarithmetiken sieht man den deutlichen Vorteil der Montgomerydarstellung bei der Multiplikation. Dabei muß man jedoch auch beachten, daß eine Umwandlung einer Zahl in Montgomerydarstellung ungefähr die Kosten einer Multiplikation hat. Deshalb sollte nicht zu oft zwischen Montgomerydarstellung und "normaler" Zahldarstellung gewechselt werden. Die beiden Langzahlarithmetiken unterscheiden sich auch in der Realisierung der langen Zahlen, die für einen Benutzer normalerweise keine Rolle spielt. Da die genaue Realisierung der langen Zahlen für die parallele Version jedoch wichtig ist, stelle ich sie im Fall der *LIP* kurz dar. In der *LIP* werden lange Zahlen als Pointer auf einen zusammenhängenden Speicherbereich (Array) realisiert. Dieser Bereich enthält einen Informationsteil (Vorzeichen, Länge des allokierten Speicherbereichs und der Zahl) und einen Datenteil zum Abspeichern der Zahl.

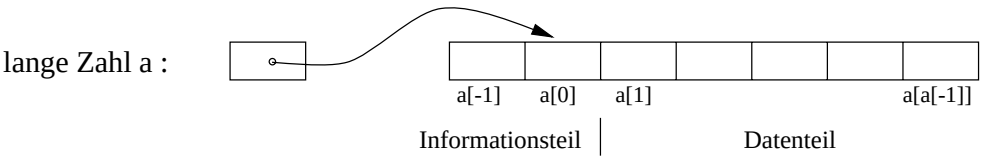


Abbildung 5.1: Aufbau einer langen Zahl (LIP)

Eine lange Zahl a repräsentiert die Zahl

$$\sum_{i=1}^{|a[0]|} a[i] \cdot RADIX^{i-1}$$

wobei a das Vorzeichen von $a[0]$ hat und $RADIX$ (in unserem Fall) gleich 2^{26} ist. In $a[-1]$ wird die Länge des allokierten Speicherbereiches und in $a[0]$ die Länge des davon zur Darstellung der Zahl benötigten Bereiches abgespeichert.
Ein Vektor von langen Zahlen ist dementsprechend als Array von Pointern auf jeweils zusammenhängende Speicherbereiche implementiert.

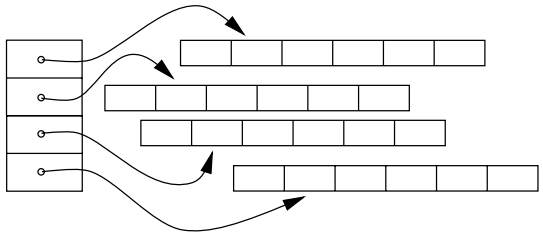


Abbildung 5.2: Aufbau eines Vektors von langen Zahlen (LIP)

5.3 Grundaufgaben des Lanczos Verfahrens

Die Tests des vorherigen Abschnitts haben gezeigt, daß es von Vorteil ist, Montgomerydarstellung zu benutzen, falls man modulo einer festen ungeraden Zahl rechnet und multiplizieren muß. Die Laufzeitabschätzung des Lanczos Verfahrens aus Kapitel 2.3 hat gezeigt, daß diese Voraussetzungen erfüllt sind. Dabei muß man aber beachten, daß auch Additionen durchgeführt werden, die in Montgomerydarstellung teuerer sind. In diesem Abschnitt werden die benötigten Grundaufgaben im Lanczos Verfahren sowie deren Implementierung behandelt. Dabei gehe ich auch auf das Problem der teueren Additionen in Montgomerydarstellung ein und stelle eine schnelle Lösung dafür vor.
Sei $p \in \mathbb{P}, \alpha, \beta \in \mathbb{F}_p, v, w \in \mathbb{F}_p^n$ und $B \in \mathbb{F}_p^{n \times n}$ (gilt in der Praxis nur näherungsweise; normalerweise ist $B \in \mathbb{F}_p^{m \times n}$, wobei $|m - n| < 20$). Zu den Grundaufgaben im Lanczos Verfahren gehören dann:

- **Berechnung des inneren Produktes zweier Spaltenvektoren $\langle v, w \rangle$**
  Für $v, w \in \mathbb{F}_p^n$ ist das innere Produkt definiert als

  $$\alpha = \langle v, w \rangle = v^T w = \sum_{k=1}^{n} v_k w_k \quad \text{für } v = (v_1, \ldots, v_n)^T, w = (w_1, \ldots, w_n)^T \ .$$

  Diese Formel kann direkt umgesetzt werden, indem man die Produkte der einzelnen Komponenten bestimmt und diese dann aufaddiert. Das Ergebnis ist dann ein Element aus $\mathbb{F}_p$. Somit sind die Kosten der Berechnung eines inneren Produktes n Multiplikationen und n Additionen (sowie die Initialisierung $\alpha = 0$). Gilt $w = v$, so ersetzt man die Multiplikationen durch Quadrierungen.

- **Vektorupdate $w = w - \alpha \cdot v$**
  Für $v, w \in \mathbb{F}_p^n$, wobei $v = (v_1, \ldots, v_n)^T, w = (w_1, \ldots, w_n)^T$, berechnet man $w = w - \alpha \cdot v$ mit Hilfe folgender Formeln für $i = 1, \ldots, n$

  $$w_i = w_i - \alpha \cdot v_i \ .$$

  Somit wird zur Berechnung eines Vektorupdate n Multiplikationen und n Subtraktionen benötigt. Der Anteil der Subtraktionen an der Gesamtlaufzeit ist so gering, daß eine Umformung $w_i = w_i + (-\alpha) \cdot v_i$ kaum spürbare Auswirkungen hätte.

- **Matrix-Vektormultiplikation $v = B^T B w$**
  Bei der Matrix B handelt es sich um eine dünnbesetzte Matrix, die zu über 90% aus Einträgen mit Betrag 1 besteht (vgl. Kapitel 3). Da $A = B^T B$ im allgemeinen nicht mehr dünnbesetzt ist, kann man A nicht vorberechnen und in jeder Iteration $v = Aw$ berechnen, sondern berechnet immer $v = B^T(Bw)$. Eine effiziente Implementierung dieser Operation erfordert eine geeignete Speicherung der dünnbesetzten Matrix B, so daß möglichst wenig Speicherplatz verbraucht wird und ein schneller Zugriff möglich ist. Die Matrix B bleibt

während der gesamten Berechnung unverändert, und es muß immer nur auf die gesamte Matrix und nicht auf einzelne Elemente zugegriffen werden. Diese Voraussetzungen führten zu folgender Datenstruktur:

- Die Einträge der Matrix werden in drei Klassen eingeteilt. Ich unterscheide zwischen $+1, -1$ und sonstigen Einträgen.
- Die Matrix B wird kontinuierlich spaltenweise abgelegt, wobei die Einträge in die drei Klassen getrennt werden.
- Bei einem Zugriff auf die Matrix B werden die Klassen nacheinander abgearbeitet. Zuerst die Klasse mit den 1-Einträgen, dann die -1-Einträge und am Schluß die sonstigen Einträge.

Getrennt nach den drei Klassen wird pro Spalte zuerst die Anzahl der entsprechenden Einträge dieser Spalte abgespeichert. Man beachte dabei, das es in einer Spalte auch keine Einträge für eine Klasse geben kann. Nach der Anzahl der entsprechenden Einträge folgen dann die benötigten Daten dieser Einträge. In der $+1$ und -1 Klasse ist das die Zeilennummer, in der der Eintrag steht. In der Klasse der sonstigen Einträge benötigt man jeweils die Zeilennummer und den Wert des Eintrags. Die Relationen werden in der Reihenfolge free, full und partial relations abgespeichert. Da free relations nur aus 1-Einträgen aufgebaut sind und full relations keine negativen Einträge haben, führt dies zu Null Einträgen am Anfang der Datenstruktur für die -1 und sonstigen Einträge. Diese führenden Nullen läßt man weg und führt zwei zusätzliche Variablen ein, die den Index der ersten Spalte enthalten, in der entsprechende Einträge auftauchen. Diese Variablen werden mit *mone_start* (-1-Einträge) bzw. mit *rest_start* bezeichnet. Die Abbildung 5.3 enthält die schematische Darstellung der Datenstruktur für dünnbesetzte Matrizen.

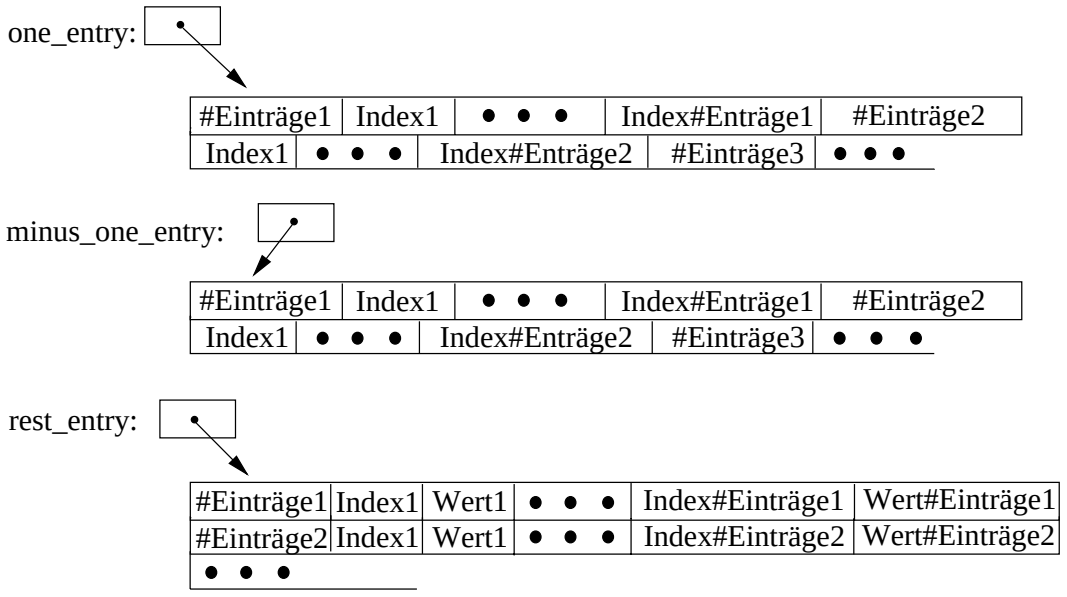


Abbildung 5.3: Datenstruktur dünnbesetzte Matrix

Beispiel:

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

one_entry ==>

3	1	5	10	**3**	2	3	10	**3**	5	6	10	**4**	1
4	9	10	**3**	5	7	9	**2**	2	3	**2**	1	9	**2**
4	7	**3**	3	6	9	**2**	8	9	**3**	1	9	10	–

minus_one_entry ==>

1	2	**3**	1	6	10	**1**	2	**0**	**1**	2

rest_entry ==>

1	8	2	**1**	3	2	**2**	5	2	8	3	**3**
3	2	6	−2	7	3	**1**	3	3	**1**	7	−2
3	1	2	3	−2	4	3	**2**	5	2	8	−3

In diesem Beispiel ist $mone\_start = 7$ und $rest\_start = 4$. Die Anzahl der jeweiligen Einträge wurden fett geschrieben. Die Numerierung der Zeilenindizes beginnt mit 1. In der Implementierung werden die drei Klassen hintereinander in einem Array abgespeichert. Dies ermöglicht beim Zugriff auf die Matrix B einen kontinuierlichen Zugriff auf Speicherbereiche.

Zur Berechnung von $Bw = zwi \in \mathbb{F}_p^n$ benutzt man Definition 1.3 und berechnet die einzelnen Komponenten (zwi_i) mit Hilfe der Formel

$$zwi_i = \sum_{k=1}^{n} b_{ik} w_k \quad (i = 1, \ldots, n) \; . \tag{5.1}$$

Zum kompletten Berechnen einer i-ten Summe benötigt man also alle Einträge der i-ten Zeile der Matrix. Da B spaltenweise abgelegt ist, wird mit jeder Spalte, die durchlaufen wird, für jede Komponente von zwi ein neuer Summand berechnet und aufaddiert. Mit der letzten durchlaufenen Spalte von B werden dann also auch die letzten Summanden jeder Komponente berechnet und somit auch der komplette Vektor zwi.
Die Aufteilung von B in drei Teile ermöglicht eine weitere Vereinfachung der Formel (5.1). So ist im ersten Teil (one_entry) $b_{ij} = 1$ und somit müssen nur die entsprechenden Komponenten w_j zu zwi_i addiert werden. Im zweiten Teil ($minus\_one\_entry$) wird auch keine Multiplikation bei der Ermittlung der Summanden benötigt, da $b_{ij} = -1$ ist. Hier muß dann nur w_j von zwi_i

abgezogen werden. Nur für den kleinsten Teil der Matrix B (*rest_entry*), der ungefähr 5% der Einträge ausmacht, muß zur Ermittlung der Summanden multipliziert werden. Da alle Einträge von B in den elementaren Datentyp *long* passen (vgl. Kapitel 3), kann dabei jedoch auf eine sog. kurze Multiplikation (Multiplikation einer langen Zahl mit einem *long*) zurückgegriffen werden.
Die Berechnung von $B^T zwi = v$ führt man analog durch, wobei gilt:

$$v_i = \sum_{k=1}^{n} b_{ki} zwi_k \quad (i = 1, \ldots, n) \; .$$

Man berechnet also mit den Werten einer Spalte sämtliche Summanden einer Komponente. Während der gesamten Berechnung von $B^T Bw$ wird also keine Multiplikation zweier allgemeiner Zahlen aus $\mathbb{F}_p$ benötigt. Die Montgomeryarithmetik bietet hier also keinen Vorteil gegenüber dem Rechnen in endlichen Primkörpern ohne Montgomerydarstellung. Generell kann es beim Rechnen in endlichen Primkörpern $\mathbb{F}_p$ nötig sein über $\mathbb{Z}$ berechnete Ergebnis modulo p zu reduzieren. Während bei der kurzen Multiplikation eine Division mit Rest zum Reduzieren notwendig ist, werden die Ergebnisse der Addition bzw. Subtraktion reduziert, indem man den Modul bei Bedarf entweder addiert oder subtrahiert. Die durchgeführte Reduktion verlangsamt die Operationen beträchtlich, ist aber für die Benutzung der schnellen Montgomery Multiplikation nach der Berechnung von $B^T Bw$ notwendig. Benutzt man bei der Matrix-Vektormultiplikation die Operationen über $\mathbb{Z}$ und reduziert die Werte der Komponenten nur jeweils am Ende der Matrix-Vektormultiplikation, so kann man bis zu 40% der Rechenzeit sparen.Betrachtet man z.B. die Situation im Beispiel der McCurley Challenge, so hatte die Matrix dort $2.66 \cdot 10^6$ 1-Einträge, $2.99 \cdot 10^6$ -1-Einträge und $3 \cdot 10^5$ sonstige Einträge. Die Dimension des Systems betrug ca. 40 000. Alleine bei den kurzen Multiplikationen spart man somit $6 \cdot 10^5$ Reduktionen pro Matrix-Vektormultiplikation ($B^T Bw$).
Die Tabelle 5.3 betrachtet die Auswirkungen der hier vorgestellten Variante auf die Laufzeit der Matrix-Vektormultiplikation. Sie vergleicht die benötigte Laufzeit bei Verwendung von Montgomeryarithmetik und bei Verwendung der benutzten Variante (Montgomerydarstellung, jedoch keine Reduktion nach jeder Operation). In diesem Beispiel war die Laufzeit der alten Version ca. 40% größer als die benötigte Zeit mit der jetzt dargestellten Methode.

Laufzeit	**alte Version**	**neue Version**
1-Einträge	14.050 s	10.470 s
-1-Einträge	13.220 s	11.080 s
long-Einträge	6.170 s	1.700 s
Reduzierung	0 s	0.580 s
Gesamt	33.440 s	23.830 s (-40.32 %)

Tabelle 5.3: Vergleich beider Vorgehensweisen (SPARC20)

Im folgenden wird die genaue Vorgehensweise und die sich daraus ergebenden Konsequenzen erläutert. Daß die Operationen in $\mathbb{Z}$ schneller sind als entsprechende Operationen in $\mathbb{F}_p$ wurde bereits in Abschnitt 5.2 gezeigt. Zur weiteren Laufzeitreduzierung wurde in beide Arithmetiken zusätzlich spezielle Additions- und Subtraktionsfunktionen in $\mathbb{Z}$ eingebaut, die die Tatsache ausnutzen, daß das eine Argument immer positiv ist (reduzierte Werte der Vektoren (w_k bzw. zwi_k)) und das zumindest bei den Additionen beide Argumente immer nicht negativ sind (die 1-Einträge werden zuerst bearbeitet). Außerdem wurde ausgenutzt, daß das Ergebnis immer eines der beiden Argumente ist (vgl. Formel 5.1). Um möglichst aussagekräftige Laufzeittests durchzuführen, die die einzelnen Operationen miteinander vergleichen, wurden bei den Additions- und Subtraktionsoperationen das Schema des Algorithmus **Additions- bzw. Subtraktionstest** verwendet. Dieses Schema ist dem Einsatz der Operationen in der Matrix-Vektormultiplikation sehr stark angepaßt. So wurde auch bei den Tests als Startwert der Addition immer 0 verwendet, da dies auch in der Matrix-Vektormultiplikation immer der Fall ist. Über den Startwert beim Einsatz der Subtraktionsoperation lassen sich keine genauen Angaben machen. Die Ergebnisse des Subtraktionstests hängen aber sehr stark von dem verwendeten Startwert ab, wie der folgende Test beweist.

Additions- bzw. Subtraktionstest

(1) wähle $0 \leq \alpha, \beta \leq p$ zufällig ;
(2) wähle *startwert*;
(3) **for** ($i = 0$ to $30\,000$) **do**
(4) $\quad erg = startwert$;
(5) $\quad$ **for** ($j = 0$ to 200) **do**
(6) $\quad\quad erg \mathrel{+}= \alpha$; (bzw. $erg \mathrel{-}= \alpha$)
(7) $\quad\quad erg \mathrel{+}= \beta$; (bzw. $erg \mathrel{-}= \beta$)
(8) $\quad$ **od**
(9) **od**

In den Zeilen der Tabelle 5.4 werden die Laufzeiten des Subtraktionstests aufgelistet für $startwert = 0$ (Ergebnis wird immer negativ), einen zufällig gewählten Startwert mit der Eigenschaft $0 \leq startwert \leq p \cdot 20$ (Ergebnis wird im Laufe der Berechnung negativ) und einen Startwert $startwert = 2^{31} \cdot p$ (Ergebnis bleibt immer positiv). Dieser Test wurde wieder für die verschiedene Moduli durchgeführt. Der zufällig gewählte Startwert kommt der praktischen Anwendung am nächsten.
Die Tabelle 5.5 vergleicht die verschiedenen Additions- und Subtraktionszeiten auf der SPARC20 mit der *libI*-Bibliothek. Die Laufzeit der Reduktion wurde auch in der Tabelle angegeben. Als Spezialfall wurde zusätzlich zur *libI* die Multiplikation (*kmult*) einer langen Zahl mit einer Zahl implementiert, die

Startwert	Anzahl	65	75	85	126
0	$12 \cdot 10^6$	13.32	14.59	15.86	21.27
$0 \leq startwert \leq p \cdot 20$	$12 \cdot 10^6$	13.15	14.42	15.72	21.47
$startwert = 2^{31} \cdot p$	$12 \cdot 10^6$	11.68	12.81	13.93	18.51

Tabelle 5.4: Subtraktionszeiten bei wechselnden Startwerten

kleiner als 2^{31} ist (und somit in den elementaren Datentyp *long* paßt). Diese Funktion ist ungefähr fünf mal schneller als die Funktion, die in der *libI* für Zahlen in Montgomeryarithmetik enthalten ist. Dabei muß man aber beachten, daß in der zusätzlichen Funktion nicht reduziert wird.

Operation	Anzahl	65	75	85	126
Addition (Montgomery)	$12 \cdot 10^6$	17.43	21.10	22.65	27.60
Addition in $\mathbb{Z}$	$12 \cdot 10^6$	15.17	16.46	17.54	23.69
Addition in $\mathbb{Z}$ (Spezial)	$12 \cdot 10^6$	12.50	13.75	15.05	20.80
Subtraktion (Montgomery)	$12 \cdot 10^6$	15.25	16.15	18.47	23.67
Subtraktion in $\mathbb{Z}$	$12 \cdot 10^6$	15.25	16.50	17.62	23.53
Subtraktion in $\mathbb{Z}$ (Spezial)	$12 \cdot 10^6$	13.15	14.42	15.72	21.47
kmult (Montgomery)	$1.2 \cdot 10^6$	14.92	19.93	16.90	23.01
kmult (Spezial)	$6 \cdot 10^6$	13.94	15.38	16.70	23.37
Reduktion	$2 \cdot 10^6$	10.65	11.25	11.93	20.12

Tabelle 5.5: Additions- und Subtraktionszeiten (SPARC20 mit *libI*)

Die erreichte Laufzeitreduzierung bei Verwendung der speziellen Operationen auf der Paragon mit der *LIP*-Bibliothek ist noch höher. Dies veranschaulicht die Tabelle 5.6, die die Laufzeiten der verschiedenen Additions- und Subtraktionsmethoden auf der Paragon einander gegenüberstellt. In der *LIP* existiert außerdem eine spezielle *kmult* Funktion, die eine lange Zahl mit einer positiven Zahl kleiner als 2^{13} sehr schnell multipliziert. Diese Funktion läuft verglichen mit der allgemeinen Funktion doppelt so schnell. Deshalb findet auf der Paragon eine weitere Aufteilung der sonstigen Einträge in positive und negative Einträge statt.

Wird nicht nach jeder Operation eine Reduktion durchgeführt, so muß man jedoch beachten, daß die Zwischenergebnisse größer werden und deshalb auch etwas mehr Speicherplatz benötigen können. Um die Zunahme des benötigten

Operation	Anzahl	65	75	85	126
Addition (Montgomery)	$28 \cdot 10^5$	26.74	24.06	29.70	43.43
Addition in $\mathbb{Z}$	$28 \cdot 10^5$	16.78	17.68	22.68	27.55
Addition in $\mathbb{Z}$ (Spezial)	$28 \cdot 10^5$	11.59	14.65	18.90	23.82
Subtraktion (Montgomery)	$28 \cdot 10^5$	30.87	30.62	34.90	51.32
Subtraktion in $\mathbb{Z}$	$28 \cdot 10^5$	21.71	22.34	27.21	32.09
Subtraktion in $\mathbb{Z}$ (Spezial)	$28 \cdot 10^5$	14.76	17.71	20.87	26.88
kmult (positiv)	$2 \cdot 10^6$	13.77	15.14	15.50	19.96
kmult (allgemein)	$1 \cdot 10^6$	14.76	17.13	19.33	22.46
Reduktion	$5 \cdot 10^5$	15.96	17.11	18.24	21.22

Tabelle 5.6: Additions- und Subtraktionszeiten (Paragon mit *LIP*)

Speicherplatzes abschätzen zu können, betrachtet man den Platzbedarf von

$$\sum_{k=1}^{n} b_{ik} w_k < \sum_{k=1}^{n} 2^{32} \cdot p \ = n \cdot 2^{32} \cdot p \quad (i = 1, \ldots, m) \ ,$$

wobei p der verwendete Modul und n die Dimension des Systems ist. Größer als dieser Wert kann keine Summe (vgl. Formel (5.1)) werden, da für alle $i, j \leq n$ gilt $b_{ij} < 2^{32}$. Da ohne echte Einschränkung für die Praxis immer $n < 2^{31}$ und außerdem auch

$$log(n \cdot 2^{32} \cdot p) = log(n \cdot 2^{32}) + log(p)$$

gilt, kann der Platzbedarf der Komponenten nie um mehr als zwei *longs* größer sein als der Platz, der zum Abspeichern des Moduls benötigt wird. Die Auswirkungen der etwas längeren Operanden auf die benötigte Laufzeit kann dabei vernachlässigt werden und ist in der Testsituation (vgl. Tabelle 5.5 und Tabelle 5.6) berücksichtigt worden. Zusammenfassend kann die Anzahl der benötigten Operationen zur Berechnung von $v = B^T B w$ mit $2 \cdot \omega_1$ speziellen Additionen, $2 \cdot \omega_2$ speziellen Subtraktionen, $2 \cdot \omega_3$ kurzen Multiplikationen und $2 \cdot n$ Reduktionen angegeben werden, wobei ω_1 die Anzahl der 1-Einträge ist, ω_2 die Anzahl der -1-Einträge ist und ω_3 die Anzahl der restlichen Einträge ist. Es folgt der Algorithmus zum Berechnen von $zwi = Bw$.

Bemerkung 5.1 Im allgemeinen führen while-Schleifen zu effizienterem Code als for-Schleifen. Trotzdem wurde als äußere Schleife jeweils eine for-Schleifen gewählt, da der gcc-Compiler bei bekannter Dimension n effizienten Code generieren kann (vgl. loop-unrolling [12]). Bei Benutzung einer while-Schleife findet diese Optimierung leider keine Anwendung.

Der Algorithmus berechnet $zwi = Bw$.
Auf die einzelnen Vektorkomponenten wird indiziert zugegriffen.

INITIALISIERUNG

```
(1)  h := 0;
(2)  for (i = 0; i < n; i++) do
(3)     zwi[i] := 0;
(4)  od
```

BEARBEITE (1)-EINTRÄGE

```
(5)  for (i = 0; i < n; i++) do
(6)     j := one_entry[h++];              /* # Einträge in Spalte i */
(7)     while (j-- > 0) do
(8)        zwi[one_entry[h++]] += w[i];
(9)     od
(10) od
```

BEARBEITE (-1)-EINTRÄGE

```
(11) for (i = mone_start; i < n; i++) do
(12)    j := one_entry[h++];              /* # Einträge in Spalte i */
(13)    while (j-- > 0) do
(14)       zwi[one_entry[h++]] -= w[i];
(15)    od
(16) od
```

BEARBEITE RESTLICHE EINTRÄGE

```
(17) for (i = rest_start; i < n; i++) do
(18)    j := one_entry[h++];              /* # Einträge in Spalte i */
(19)    while (j-- > 0) do
(20)       zwi[one_entry[h]] += w[i] · one_entry[h + 1];
(21)       h += 2;
(22)    od
(23) od
```

REDUZIERE DEN VEKTOR

```
(24) for (i = 0; i < n; i++) do
(25)    zwi[i] := zwi[i] mod p;
(26) od
```

Mit leichten Veränderungen kann der Algorithmus auch zur Berechnung von $B^T v = zwi$ benutzt werden. Man ändert

$$\begin{aligned} (8) \quad & \text{in} \quad zwi[i] \mathrel{+}= v[one\_entry[h{+}{+}]]; \\ (14) \quad & \text{in} \quad zwi[i] \mathrel{-}= v[one\_entry[h{+}{+}]]; \\ (20) \quad & \text{in} \quad zwi[i] \mathrel{+}= v[one\_entry[h]] \cdot one\_entry[h+1]; \ . \end{aligned}$$

Die Tabelle 5.7 enthält zusammenfassend die Anzahl der elementaren Operationen, die für die einzelnen Grundoperationen benötigt werden. Dabei steht **Mult** für Montgomery Multiplikationen, **Square** für Montgomery Quadrierungen, **Add_m** für Montgomery Additionen, **kmult** für die Multiplikation einer langen Zahl mit einem *long* über $\mathbb{Z}$, **Add** für Additionen über $\mathbb{Z}$, **Sub** für Subtraktionen über $\mathbb{Z}$ und **Red** für Reduktionen einer Zahl aus $\mathbb{Z}$ modulo p.

Operation	**Mult**	**Square**	**Sub_m**	**Add_m**	**kmult**	**Add**	**Sub**	**Red**
$\langle w, v\rangle$	n			n				
$\langle v, v\rangle$		n		n				
$w \mathrel{+}= \alpha \cdot v$	n			n				
$w \mathrel{-}= \alpha \cdot v$	n		n					
$B^T B w$					$2\omega_3$	$2\omega_1$	$2\omega_2$	$2n$

Tabelle 5.7: Elementare Operationen der Grundoperationen

5.4 Beschleunigung der Lösungsberechnung

Bevor ich zu einer genauen Laufzeitanalyse des Lanczos Verfahrens komme, stelle ich noch eine weitere Idee zur Beschleunigung des Algorithmus vor. Bei der Berechnung von $r\,(r \geq 1)$ Lösungen $Bx_j = u_j \quad (1 \leq j \leq r)$ bestimmt man $z_j = B^T u_j$ und startet den Algorithmus mit $w_0 = z_1$ (vgl. Abschnitt 2.2). Während der Berechnung der r verschiedenen Lösungen bestimmt man

$$x_j = \sum_{l=0}^{m-1} \frac{\langle w_l, z_j\rangle}{\langle w_l, v_{l+1}\rangle}\, w_l \ , \quad 1 \leq j \leq r \ .$$

Bei dieser Berechnung benötigt man die inneren Produkte $\langle w_l, z_j\rangle$. Da $z_j = B^T u_j$ ist, kann man in unseren Beispielen (B dünnbesetzt, mehr als 90% Einträge mit Betrag 1 und maximaler Betrag kleiner 100) direkt folgern, daß alle Komponenten der z_j betragsmäßig kleiner als 2^{31} sind. Deshalb wird bei dieser inneren Produktberechnung keine Montgomerymultiplikation benötigt, sondern es ist ausreichend, die kurze Multiplikation zu benutzen. Eine geschickte Wahl der u_j (die als Spaltenvektoren Exponentenvektoren von Relationen sind) kann die Anzahl der benötigten Operationen noch weiter reduzieren. Sind möglichst viele der Komponenten von z_j

gleich Null, so kann die Anzahl der Summanden, die zur inneren Produktberechnung ermittelt werden müssen, minimiert werden. Im folgenden betrachte ich der Übersichtlichkeit wegen den Fall, daß nur eine Lösung berechnet wird. Sollen mehrere Lösungen berechnet werden, so kann man analog vorgehen. Aus

$$z = B^T u \quad \text{und somit} \quad z_i = \sum_{k=1}^{n} b_{ki} u_k \quad (i = 1, \ldots, n)$$

folgt, daß es wegen $z_i \neq 0$ mindestens ein k gibt, so daß $b_{ki} \neq 0$ und $u_k \neq 0$. Deshalb wählt man als rechte Seite u eine Relation, die mit möglichst wenigen Relationen gemeinsame Einträge hat. Als Kandidaten dafür kommen die free relations in Frage (vgl. Kapitel 3). Zur Bestimmung der Relation, die als rechte Seite benutzt werden soll, bestimmt man für jede Spalte k

$$cum\_weight(k) := \sum_{i=1}^{n} Gewicht(Zeile_i) \cdot b_{ik},$$

also die Summe der Zeilengewichte, in denen diese Einträge ungleich Null hat. Die Spalte, in der diese Summe minimal ist, wird als rechte Seite gewählt. Benötigt man r Lösungen, so wählt man die r Spalten mit den kleinsten Werten. Die Tabelle 5.8 listet für die 10 berechneten Lösungen des DL85-Beispiels jeweils die berechnete Summe *cum_weight* sowie die Anzahl der Komponenten von $z = B^T u$ mit den verschiedenen Einträgen (0, 1, −1 oder sonstige). Zum Vergleich wurden auch die 5 schwersten Spalten in die Tabelle aufgenommen, für die jedoch kein *cum_weight* Wert berechnet wurde.

rechte Seite	**cum_weight**	**Eintrag**			
		0	1	−1	**sonstige**
Lösung 1	843	51 201	394	435	5
Lösung 2	921	51 127	435	473	2
Lösung 3	964	51 118	441	507	5
Lösung 4	986	51 061	473	499	2
Lösung 5	995	51 053	446	532	4
Lösung 6	1000	51 044	493	494	4
Lösung 7	1005	51 047	473	498	17
Lösung 8	1006	51 045	474	513	3
Lösung 9	1011	51 037	439	558	1
Lösung 10	1022	51 035	471	527	2
Vergleich 1		861	534	542	50 098
Vergleich 2		1 637	1 313	1 262	47 823
Vergleich 3		1 003	650	640	49 744
Vergleich 4		1 416	987	1 048	48 584
Vergleich 5		1441	1 055	977	48 562

Tabelle 5.8: Einträge der leichten rechten Seiten ($B^T u$) in DL85

Somit reduzieren sich die Kosten der Bestimmung des inneren Produktes bei der Lösungsberechnung auf wenige Additionen. Bei den sonstigen Einträgen der Lösungen handelt es sich ausschließlich um Einträge mit Betrag 2. Daß die Summe der 1, -1 und sonstigen Einträge nicht genau mit *cum_weight* übereinstimmt, hängt an der Auslöschung von Einträgen und an der Tatsache, daß in *cum_weight* das Gewicht aller rechten Seiten auch berücksichtigt ist.

5.5 Genaue Laufzeitanalyse

In diesem Abschnitt wird eine genaue Laufzeitanalyse des Lanczos Verfahrens durchgeführt. Dabei wird die Anzahl der Operationen ermittelt, die pro Iteration ausgeführt werden müssen. Da diese Laufzeitanalyse als Grundlage für exakte Laufzeitvorhersagen dienen soll, zähle ich nicht wie im Abschnitt 2.3 nur die Körperoperationen, sondern teile die Operationen wie in Abschnitt 5.3 ein. Auf die dort erzielten Ergebnisse wird zurückgegriffen, sofern dies möglich ist.

Bemerkung 5.2 [Bezeichnungen] Im folgenden steht **Inv** für eine Invertierung einer Zahl aus $\mathbb{F}_p$, wobei p der verwendete Modul ist. Weiterhin steht **Mult** für Montgomery Multiplikationen, **Square** für Montgomery Quadrierungen, **Add_m** für Montgomery Additionen, **kmult** für die Multiplikation einer langen Zahl mit einem *long* über $\mathbb{Z}$ (für die Paragon bezeichnet **kmult_N** die normale kurze Multiplikation und **kmult_F** die schnelle Variante), **Add** für Additionen über $\mathbb{Z}$, **Sub** für Subtraktionen über $\mathbb{Z}$ und **Red** für Reduktionen einer Zahl aus $\mathbb{Z}$ modulo p. Außerdem bezeichnet r die Anzahl der berechneten Lösungen, n die Dimension des Systems, ω_1 die Anzahl der 1-Einträge, ω_2 die Anzahl der -1-Einträge und ω_3 die Anzahl der restlichen Einträge. Mit $\omega_{3.1}$ wird die Anzahl der negativen restlichen Einträge und mit $\omega_{3.2}$ die Anzahl der positiven restlichen Einträge auf der Paragon bezeichnet. Mit $T(Operation)$ wird die Laufzeit einer Operation bezeichnet.

Die Beschreibung des Lanczos Verfahrens

$$w_0 = b \ , \tag{5.2}$$

$$v_1 = B^T B \ w_0 \ , \tag{5.3}$$

$$w_1 = v_1 - \frac{\langle v_1, v_1 \rangle}{\langle w_0, v_1 \rangle} \ w_0 \tag{5.4}$$

und für $i \geq 1$

$$v_{i+1} = B^T B \ w_i \ , \tag{5.5}$$

$$w_{i+1} = v_{i+1} - \frac{\langle v_{i+1}, v_{i+1} \rangle}{\langle w_i, v_{i+1} \rangle} \ w_i - \frac{\langle w_i, v_{i+1} \rangle}{\langle w_{i-1}, v_i \rangle} \ w_{i-1} \ , \tag{5.6}$$

$$x_j = \sum_{l=0}^{m-1} \frac{\langle w_l, z_j \rangle}{\langle w_l, v_{l+1} \rangle} \ w_l \ , \quad 1 \leq j \leq r \tag{5.7}$$

legt eine Einteilung des Algorithmus in Phasen nahe. Zur Bestimmung der Laufzeit betrachte ich lediglich die Formeln für den Fall $i \geq 1$.

1. **Berechnung von $v_{i+1} = B^T B\, w_i$**
   Die benötigten Operationen wurden bereits in Abschnitt 5.3 ermittelt. Pro Iteration werden folgende Operationen benötigt:

Inv	Mult	Square	Sub_m	Add_m	kmult	Add	Sub	Red
					$2\omega_3$	$2\omega_1$	$2\omega_2$	$2n$

2. **Berechnung von w_{i+1}**

$$w_{i+1} = v_{i+1} - \frac{\langle v_{i+1}, v_{i+1}\rangle}{\langle w_i, v_{i+1}\rangle}\, w_i - \frac{\langle w_i, v_{i+1}\rangle}{\langle w_{i-1}, v_i\rangle}\, w_{i-1}$$

   Zur Berechnung der inneren Produkte $\langle v_{i+1}, v_{i+1}\rangle, \langle w_i, v_{i+1}\rangle$ werden pro Iteration folgende Operationen benötigt:

Inv	Mult	Square	Sub_m	Add_m	kmult	Add	Sub	Red
	n	n		$2n$				

   Dann werden 2 Multiplikationen und 1 Invertierung zur Bestimmung der Skalaren $\langle v_{i+1}, v_{i+1}\rangle / \langle w_i, v_{i+1}\rangle, \langle w_i, v_{i+1}\rangle / \langle w_{i-1}, v_i\rangle$ benötigt. Durch zwei Vektorupdates wird dann w_{i+1} berechnet (vgl. Tabelle 5.7).

Inv	Mult	Square	Sub_m	Add_m	kmult	Add	Sub	Red
1	$2n+2$		$2n$					

   Zusammen benötigt man also pro Iteration zur Berechnung von w_{i+1}

Inv	Mult	Square	Sub_m	Add_m	kmult	Add	Sub	Red
1	$3n+2$	n	$2n$	$2n$				

3. **Aktualisierung der r partiellen Lösungen**
   In der i-ten Iteration kann die j-te partielle Lösung als

$$x_j = x_j + \frac{\langle w_i, z_j\rangle}{\langle w_i, v_{i+1}\rangle}\, w_i$$

   berechnet werden. Aus der Berechnung von w_{i+1} ist $\langle w_i, v_{i+1}\rangle^{-1}$ schon bekannt. In Abschnitt 5.4 wurde gezeigt, daß das innere Produkt mit (vernachlässigbar) wenigen Operationen bestimmt werden kann. Deshalb muß nur ein Skalar ausgerechnet werden und ein Vektorupdate durchgeführt werden. Man benötigt zur Aktualisierung der r partiellen Lösungen pro Iteration

Inv	Mult	Square	Sub_m	Add_m	kmult	Add	Sub	Red
	$r(n+1)$			rn				

Insgesamt werden also pro Iteration (bei r berechneten Lösungen)

Inv	Mult	Square	Sub_m	Add_m	kmult	Add	Sub	Red
1	$(3+r)(n+1)-1$	n	$2n$	$(2+r)n$	$2\omega_3$	$2\omega_1$	$2\omega_2$	$2n$

Operationen gebraucht. Da man höchstens n Iterationen durchführen muß, beträgt die Geamtlaufzeit des Lanczos Algorithmus

Inv	Mult	Square	Sub_m	Add_m	kmult	Add	Sub	Red
n	$n(3+r)(n+1)-n$	n^2	$2n^2$	$(2+r)n^2$	$2n\omega_3$	$2n\omega_1$	$2n\omega_2$	$2n^2$

Die Anzahl der benötigten Operationen pro Iteration und die Laufzeiten der einzelnen Operationen dienen als Grundlage für ein Modell zur relativ genauen Angabe der Zeiten, die eine Iteration bzw. die gesamte Berechnung benötigen. In diesem Modell muß aber auch noch berücksichtigt werden, daß in der realen Anwendung die Daten eventuell erst in die Register geladen werden müssen bzw. sich nicht mehr im Cache der Maschine befinden. Das Testprogramm, das die Laufzeiten der einzelnen Operationen ermittelt, greift immer auf Zahlen im selben Speicherbereich zu und vernachlässigt diese Tatsache. Der Effekt läßt sich besonders gut bei der sequentiellen Version beobachten. Dort muß z.B. während den Additionen und Subtraktionen der Matrix-Vektormultiplikation (je nach Größe des Beispiels) ungefähr 30 MB geladen werden. Dasselbe Beispiel auf der Paragon mit 64 Knoten benötigt nur noch einen Anteil von weniger als 1 MB. Bei der Ermittlung der realen Laufzeit wird diese Tatsache durch Faktoren berücksichtigt, mit denen die Laufzeiten der einzelnen Operationen multipliziert werden. Praktische Tests haben gezeigt, daß man auf der SPARC20 Faktoren für die Additionen und Subtraktionen der Matrix-Vektoroperationen (*cache1*), für die dort durchgeführten Reduktionen und kurzen Multiplikationen (*cache3*) und für die reinen Vektoroperationen (*cache2*) bestimmen muß. Die ermittelten Werte sind für vier Beispiele, an denen das Modell überprüft wird, in Tabelle 5.9 aufgeführt.

Faktor	DL85	COS85	DL65.1	DL65.2
cache1	2.00	2.20	1.80	1.70
cache2	1.04	1.04	1.04	1.04
cache3	1.30	1.50	1.50	1.50

Tabelle 5.9: Ermittelten Faktoren für die Testbeipiele

Aus diesen Faktoren und der Anzahl der benötigten Operationen folgt direkt eine Formel, die bei Angabe der Laufzeiten der einzelnen Operationen sowie einigen charakteristischen Daten des zu lösenden Systems $(\omega_1, \omega_2, \omega_3, n, r)$, die benötigte Gesamtzeit der Berechnung ermitteln kann.

Setzt man

$$\begin{aligned}
\omega &= cache1 \cdot (\omega_1 \cdot T(Add) + \omega_2 \cdot T(Sub)) + cache3 \cdot \omega_3 \cdot T(kmult) \\
t_1 &= cache2 \cdot (T(Inv) + (2+r) \cdot T(Mult)) \\
t_2 &= cache2 \cdot (T(Square) + 2 \cdot T(Sub_m) + (2+r) \cdot T(Add_m) \\
&\quad +(3+r) \cdot T(Mult)) + 2 \cdot cache3 \cdot T(Red)
\end{aligned}$$

so erhält man für die Gesamtlaufzeit des Verfahrens in Abhängigkeit von der Dimension n, dem Gewicht ω und der Anzahl der benötigten Lösungen r

$$\mathcal{T}(n, \omega, r) = n^2 \cdot t_2 + n \cdot (2 \cdot \omega + t_1) \ . \tag{5.8}$$

Mit Hilfe dieser Formel kann man auch ermitteln, wie lange eine Iteration benötigt $(\mathcal{T}(n, \omega, r)/n)$. Außerdem können die Analysen in diesem Abschnitt dazu benutzt werden, die Laufzeit pro Iteration den verschiedenen Teilen der Berechnung zuzuordnen. Dieses Modell kann auch dazu benutzt werden, um die Auswirkungen von Veränderungen in den charakteristischen Daten eines Systems oder den Laufzeiten der einzelnen Operationen vorherzusagen. Im Anhang A befindet sich ein Skript mit dem Modell, das die Laufzeitvorhersagen durchführt. Das so entwickelte Modell wird mit vier Beispielen überprüft. Die Tabelle 5.10 vergleicht die gemessenen Laufzeiten für die einzelnen Teile der Berechnung und die berechneten Laufzeiten der beiden Systeme COS85 und DL85 miteinander.

Operation	**Laufzeit COS85**		**Laufzeit DL85**	
	gemessen	berechnet	gemessen	berechnet
Additionen (Bw)	6.82 s	6.45 s	9.82 s	9.71 s
Additionen $(B^T zwi)$	6.03 s	6.45 s	9.03 s	9.71 s
Subtraktionen (Bw)	8.53 s	7.66 s	11.63 s	11.19 s
Subtraktionen $(B^T zwi)$	7.45 s	7.66 s	11.13 s	11.19 s
kmult (Bw)	0.46 s	0.45 s	0.55 s	0.61 s
kmult $(B^T zwi)$	0.42 s	0.45 s	0.57 s	0.61 s
Reduktion (Bw)	0.47 s	0.47 s	0.44 s	0.41 s
Reduktion $(B^T zwi)$	0.48 s	0.47 s	0.43 s	0.41 s
Berechnung $B^T(Bv)$	30.70 s	30.20 s	43.63 s	43.85 s
update Lösung(en)	1.67 s	1.68 s	16.85 s	16.83 s
Berechnung $w_i + 1$	3.26 s	3.35 s	3.270 s	3.36 s
Berechnung $\langle w_i, v_{i+1} \rangle$	1.69 s	1.68 s	1.70 s	1.68 s
Berechnung $\langle v_{i+1}, v_{i+1} \rangle$	1.68 s	1.65 s	1.65 s	1.68 s
Gesamtiteration	38.67 s	38.45s	67.55 s	67.42 s

Tabelle 5.10: Vergleich gemessene und berechnete Laufzeiten COS85, DL85

Dabei wird in der Berechnung von $B^T zwi$ und Bw jeweils die gleiche Anzahl von

Rechenoperationen durchgeführt. Beide Berechnungen unterscheiden sich im wesentlichen nur durch die unterschiedliche Art und Weise, wie sie auf den Speicherbereich zugreifen. Der hohe Faktor *cache1* sowie der Unterschied in diesen beiden Berechnungen ($B^T zwi$ und Bw) verdeutlichen die Bedeutung des effizienten Speicherzugriffs in der Matrix-Vektormultiplikation bei der Benutzung der sequentiellen Implementierung. Als weiteren Test für das Modell wurden zwei verschiedene Systeme (DL65.1, DL65.2) untersucht, die bei der Lösung eines DL-Problems modulo einer 65-stelligen Primzahl aufgetreten sind. Den Vergleich der berechneten und gemessenen Zeiten für die beiden DL65 Beispiele enthält Tabelle 5.11.

Operation	**Laufzeit DL65.1**		**Laufzeit DL65.2**	
	gemessen	berechnet	gemessen	berechnet
Additionen (Bw)	0.58 s	0.59 s	0.85 s	0.80 s
Additionen ($B^T zwi$)	0.55 s	0.59 s	0.69 s	0.80 s
Subtraktionen (Bw)	0.41 s	0.40 s	1.02 s	0.90 s
Subtraktionen ($B^T zwi$)	0.40 s	0.40 s	0.81 s	0.90 s
kmult (Bw)	0.05 s	0.04 s	0.08 s	0.08 s
kmult ($B^T zwi$)	0.04 s	0.04 s	0.08 s	0.08 s
Reduktion (Bw)	0.13 s	0.14 s	0.07 s	0.07 s
Reduktion ($B^T zwi$)	0.13 s	0.14 s	0.07 s	0.07 s
Berechnung $B^T(Bv)$	2.32 s	2.32 s	3.67 s	3.71 s
update Lösung(en)	4.25 s	4.15 s	2.13 s	2.08 s
Berechnung $w_i + 1$	0.82 s	0.83 s	0.41 s	0.42 s
Berechnung $\langle w_i, v_{i+1} \rangle$	0.43 s	0.42 s	0.20 s	0.21 s
Berechnung $\langle v_{i+1}, v_{i+1} \rangle$	0.41 s	0.42 s	0.22 s	0.21 s
Gesamtiteration	8.23 s	8.14s	6.62 s	6.62 s

Tabelle 5.11: Vergleich gemessene und berechnete Laufzeiten DL65

Die Tabelle 5.12 enthält die charakteristischen Daten aller Testbeispiele.

Daten	**DL85**	**COS85**	**DL65.1**	**DL65.2**
1-Einträge	3 872 070	2 339 061	313 462	453 380
-1-Einträge	4 228 111	2 631 285	197 823	475 681
long-Einträge	168 561	106 115	12 108	25 789
Dimension	52 035	51 855	19 819	9 919
Lösungen	10	1	10	10

Tabelle 5.12: Charakteristische Daten der Testbeispiele

Die Tests haben gezeigt, daß man mit Hilfe des Modells die Laufzeiten relativ genau vorhersagen kann. Das Modell ist auch nützlich, um zu sehen, in welchen Teilen der Berechnung viel Rechenzeit verbraucht wird. Im Kapitel 6 werden noch weitere Anwendungsmöglichkeiten dieses Modells besprochen.

5.6 Parallele Implementierung

In diesem Abschnitt beschreibe ich meine parallele Implementierung des Lanczos Verfahrens auf einer MIMD-Maschine der Intel Paragon XP/S 10. Dazu stelle ich den Rechner zuerst kurz vor. Eine genauere Beschreibung der Maschine in Jülich findet sich in [1]. Dann beschreibe ich meinen Ansatz zur Parallelisierung des Verfahrens. Dabei stellt die gleichmäßige Auslastung der Knoten und die Realisierung des Austauschs von Daten einen Schwerpunkt dar. Zu diesem Zweck werden die Kommunikationsbefehle der Paragon genau untersucht, sowie die sich daraus ergebenden Möglichkeiten zur Übertragung von Daten. Danach werden die Möglichkeiten zur Überlagerung der Kommunikation mit Berechnungen analysiert. Abschließend wird das entwickelte Laufzeitmodell mit der parallelen Implementierung überprüft.

Die Ziele, die mit der Parallelisierung angestrebt werden, sind:

- ein skalierbares Programm (d.h. das Programm läuft prinzipiell auf beliebig vielen Knoten),
- ein (annähernd) linearer Speedup (d.h. bei steigender Anzahl von Knoten k soll das Produkt $T_k \cdot k$ (annähernd) konstant bleiben, wobei T_k die Laufzeit des Programms bei k beteiligten Knoten ist).

Bevor die Realisierung dieser Ziele angesprochen wird, möchte ich generelle Probleme ansprechen, die beim Parallelisieren fast immer auftauchen. Beim Design eines parallelen Programms sollte man die folgenden (möglichen) Probleme immer beachten:

- alle Knoten sollen (annähernd) gleich ausgelastet sein,
- es gibt Synchronisationspunkte, an denen Ergebnisse unter den Knoten ausgetauscht werden müssen,
- der Kommunikationsaufwand steigt in der Regel mit der Anzahl der beteiligten Knoten.

Da die Maschine, auf der das parallele Programm ablaufen soll, das Design des Algorithmus stark beeinflußt, werde ich im nächsten Abschnitt zuerst eine kleine technische Beschreibung der Paragon anschließen. Danach werde ich meinen Ansatz zum Parallelisieren des Lanczos Verfahrens vorstellen.

5.6.1 Beschreibung der Intel Paragon XP/S 10

Der Rechner Intel Paragon ist ein Parallelrechner mit verteiltem Speicher. Die Knotenrechner der Paragon basieren auf dem i860XP RISC Prozessor. Sie sind mit einem zweidimensionalen Gitter mit hoher Bandbreite verbunden. Die zwischen zwei Knoten angegebene theoretische Bandbreite beträgt 175 MB/s in beide Richtungen. Die Knoten der Paragon werden (je nach ihrer Aufgabe und Ausstattung) in drei Klassen unterteilt. Es gibt:

compute Knoten : auf ihnen laufen die parallelen Programme ab.

service Knoten : bieten die Möglichkeiten eines UNIX Systems sowie Compiler und Programmentwicklungstools.

I/O Knoten : als Schnittstelle zu Netzwerken und Massenspeichern.

Dabei besteht jeder Knoten aus 2 Prozessoren. Zusätzlich zu dem Rechenprozessor (RP), auf dem das Programm abläuft, gibt es noch einen Kommunikationsprozessor (KP), der sämtliche Aufgaben beim Empfangen und Versenden von Nachrichten (und dazu gehört auch I/O) selbstständig ausführt. Somit wird die Kommunikation zumindest teilweise von dem Rechenprozessor entkoppelt. Sofern dies nicht unbedingt notwendig ist, findet zwischen dem Rechenprozessor und dem kompletten Rechenknoten keine Unterscheidung statt.

Die Leistung eines Rechenknotens des Paragon liegt erfahrungsgemäß bei 5 bis 20 MFLOPS. Das System Intel Paragon in der KFA hat derzeit 138 Rechenknoten (vgl. Abbildung 5.4 (Quelle: KFA-Jülich)). Der gesamte Speicher ist auf die einzelnen Knoten aufgeteilt. Auf jedem Knoten stehen für das Benutzerprogramm ca. 26 MB Hauptspeicher zur Verfügung. Da es keinen globalen Adreßraum gibt, wird diese Maschine hauptsächlich nach dem Programmiermodell des Message Passing genutzt, d.h. der Zugriff auf Daten eines anderen Rechenknotens erfolgt durch explizit zu programmierenden Nachrichtenaustausch. An Plattenplatz (RAID-Systeme) hat der Rechner insgesamt 28,8 GB, die I/O-Leistung ist aber im Vergleich zu anderen Parallelrechner gering. Weitere Informationen zur Intel Paragon:

Prozessoren (i860XP)	138 Compute Knoten, 13 Service Knoten
Leistung	75 MFLOPS/Prozessor
Overall Peak Performance	10,5 GFLOPS
Speicher	32 (26) MB/Prozessor
Betriebssystem	Paragon OSF/1

Tabelle 5.13: Technische Daten Intel Paragon XP/S 10

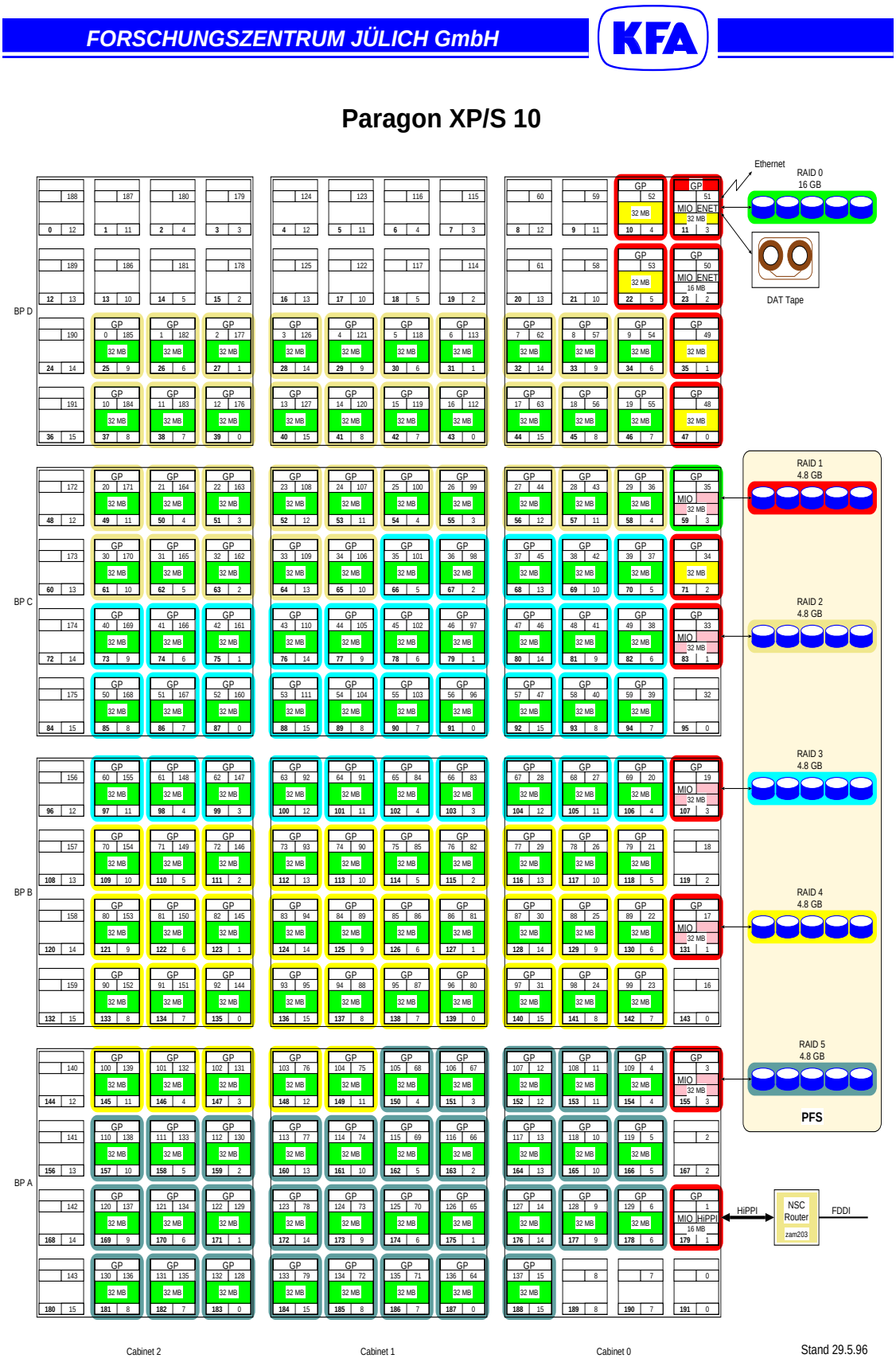


Abbildung 5.4: Aufbau der Paragon in Jülich

5.6.2 Ansatz zur Parallelisierung

Betrachtet man die Laufzeitergebnisse der sequentiellen Version, so sieht man, daß der größte Teil der Laufzeit in der Matrix-Vektormultiplikation und je nach Anzahl der benötigten Lösungen in deren Berechnung benötigt wird. Deshalb beginne ich mit dem Ansatz der Parallelisierung des Lanczos Verfahrens bei der Matrix-Vektormultiplikation $Bw = zwi$, wobei

$$zwi_i = \sum_{k=1}^{n} b_{ik} w_k \quad (i = 1, \ldots, n) \; .$$

Die Parallelisierung dieser Operation ist für dichtbesetzte Matrizen bereits gut untersucht worden (vgl. z.B. Kapitel 2 in [2]). Prinzipiell können die dort gewonnenen Erkenntnisse auch auf dünnbesetzte Matrizen angewandt werden. So kann man zwischen der spaltenweisen und zeilenweisen Aufteilung der Matrix B unterscheiden. Die Abbildung 5.5 stellt die beiden Möglichkeiten jeweils für drei Knoten schematisch dar. Im Fall a sind die Zeilen von B in drei Teile aufgeteilt, die den einzelnen Knoten zugeordnet sind. Bei zeilenweiser Aufteilung müssen alle Knoten den kompletten Vektor w haben. Sie berechnen dann einen Teil des Ergebnisvektors zwi komplett, d.h. jede Komponente des Vektors zwi wird nur von einem Knoten berechnet (vgl. Abschnitt 5.3). Teilt man die Spalten von B auf die einzelnen Knoten

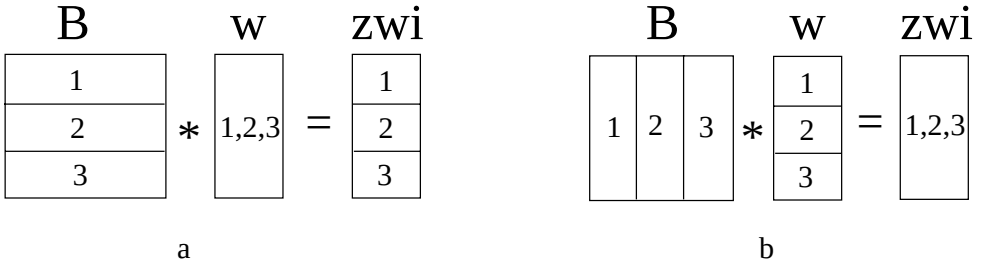


Abbildung 5.5: Aufteilungsmöglichkeiten der Matrix-Vektormultiplikation

auf (Fall b), so benötigt jeder Knoten nur einen Teil des Vektors w. Jeder Knoten berechnet dann Anteile jeder Komponente des Ergebnisvektors zwi. Damit die Knoten mit dem Ergebnisvektor zwi weitere Berechnungen durchführen können, muß die komplette Komponente auf einem Knoten bekannt sein. Ein Knoten müßte also von allen anderen Knoten den von ihnen berechneten Anteil geschickt bekommen, um eine Komponente von zwi komplett zu berechnen. Um den Kommunikationsaufwand möglichst gering zu halten und da es in den nachfolgenden Schritten von Vorteil ist, wenn alle Knoten den kompletten Vektor w kennen, habe ich mich für die erste Variante entschieden und teile die Zeilen der Matrix B auf die Knoten auf. Damit im Anschluß an die Berechnung von $Bw = zwi$ die Berechnung von $v = B^T zwi = B^T Bw$ erfolgen kann, findet nach der Berechnung von zwi ein Kommunikationsschritt statt, der dazu führt, daß der gesamte Vektor zwi allen Knoten bekannt ist. Auf die Realisierung der Kommunikationsschritte werde ich später eingehen. Die Berechnung von $v = B^T zwi$ erfolgt analog zur Berechnung von $zwi = Bw$. Somit wird bei der parallelen Implementierung im Unterschied zur sequentiellen Version, bei der nur B spaltenweise abgespeichert wird, B und B^T zeilenweise abgespeichert.

Zu den weiteren Grundaufgaben in Lanczos Verfahren (vgl. Abschnitt 5.3) gehört die Bestimmung von inneren Produkten und der Vektorupdate. Bei der Bestimmung von inneren Produkten werden die Komponenten der einzelnen Vektoren auf die Knoten verteilt. Das innere Produkt ($\alpha \in \mathbb{F}_p$) zweier Vektoren $v, w \in \mathbb{F}_p^n$ ist definiert als

$$\alpha = \langle v, w \rangle = v^T w = \sum_{k=1}^{n} v_k w_k \quad \text{für } v = (v_1, \ldots, v_n)^T, w = (w_1, \ldots, w_n)^T \ .$$

Berechnet man das innere Produkt auf k Knoten, so wird von jedem Knoten das partielle innere Produkte seiner Komponenten der Vektoren bestimmt. Diese Werte müssen dann in einem weiteren Kommunikationsschritt aufaddiert werden. Dazu sammelt jeder Knoten seine berechneten partiellen inneren Produkte in einem Vektor. Während einer Iteration muß das innere Produkt von $\langle v, w \rangle$ und $\langle v, v \rangle$ sowie für jede der r berechneten Lösungen $\langle w, z_j \rangle$ für $j = 1, \ldots, r$ ermittelt werden. In Abbildung 5.6 ist die Berechnung der inneren Produkte sowie der Aufbau des Vektors zum Sammeln der partiellen inneren Produkte schematisch dargestellt.

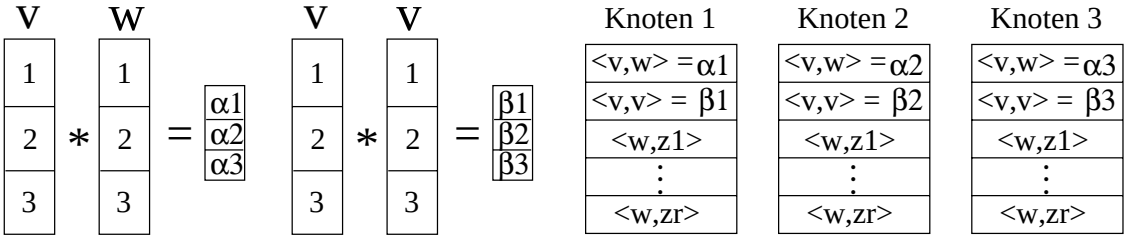


Abbildung 5.6: Parallele Berechnung der inneren Produkte

Bei der Bestimmung von w_{i+1}

$$w_{i+1} = v_{i+1} - \frac{\langle v_{i+1}, v_{i+1} \rangle}{\langle w_i, v_{i+1} \rangle} \, w_i - \frac{\langle w_i, v_{i+1} \rangle}{\langle w_{i-1}, v_i \rangle} \, w_{i-1}$$

müssen Vektorupdates durchgeführt werden. Nach der Bestimmung der globalen inneren Produkte kann jeder Knoten die Skalaren $\gamma = \langle v_{i+1}, v_{i+1} \rangle / \langle w_i, v_{i+1} \rangle$ und $\delta = \langle w_i, v_{i+1} \rangle / \langle w_{i-1}, v_i \rangle$ berechnen. Jeder Knoten berechnet dann die ihm zugeteilten Komponenten des neuen Vektors w_{i+1} mit Hilfe der entsprechenden Komponenten der Vektoren v_{i+1}, w_i sowie w_{i-1}. Die Abbildung 5.7 stellt die Vorgehensweise noch einmal schematisch dar.

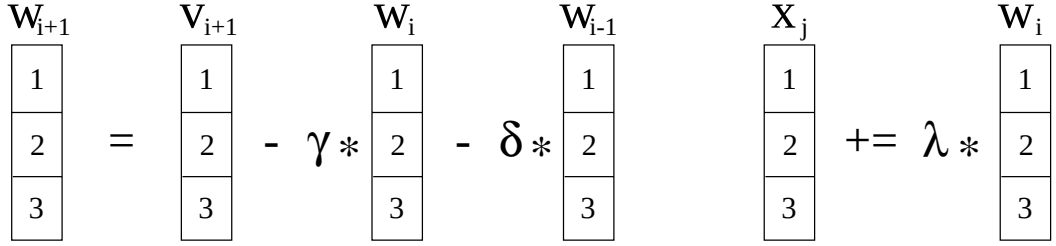


Abbildung 5.7: Parallele Berechnung des Vektors w_{i+1} und der Lösungen x_j

Da in der nächsten Iteration w_{i+1} wieder von allen Knoten komplett benötigt wird, muß wieder ein Kommunikationsschritt zum Austausch dieses Vektors durchgeführt werden. Da es sich bei der partiellen Berechnung der Lösungen auch um ein Vektorupdate handelt, wird die parallele Berechnung analog zur Bestimmung des Vektor w_{i+1} durchgeführt. Es wird kein Kommunikationsschritt zum Austausch von Lösungsvektoren in jeder Iteration benötigt. Am Ende der Berechnung (also nachdem die Lösungen vollständig berechnet wurden) schickt jeder Knoten seine Komponenten der Lösungsvektoren an einen Knoten. Dieser sammelt die Komponenten und gibt sie dann sortiert aus. Die Abbildung 5.7 enthält auch exemplarisch für eine Lösung x_j die schematische Darstellung deren Berechnung.

In der parallelen Version besteht eine Iteration also insgesamt aus 8 Schritten:

Schritt 1: (*Berechnung*) Jeder Knoten berechnet seine Komponenten von $zwi = Bw$.

Schritt 2: (*Kommunikation*) Austausch der Komponenten des Vektors zwi .

Schritt 3: (*Berechnung*) Jeder Knoten berechnet seine Komponenten von $v_{i+1} = B^T zwi$.

Schritt 4: (*Berechnung*) Jeder Knoten berechnet seine partiellen inneren Produkte.

Schritt 5: (*Kommunikation*) Ermittlung der globalen inneren Produkte.

Schritt 6: (*Berechnung*) Jeder Knoten berechnet seine Komponenten von w_{i+1}.

Schritt 7: (*Kommunikation*) Austausch der Komponenten des Vektors w_{i+1}.

Schritt 8: (*Berechnung*) Jeder Knoten berechnet seine Komponenten der Lösungen x_j $(j = 1, \ldots, r)$.

Bei der Parallelisierung fallen 2 Probleme auf. Es gibt 3 Schritte im Algorithmus, an denen ein Austausch von Ergebnissen nötig ist (Schritt 2,5,7). Zu diesem Zeitpunkt findet eine Synchronisation der einzelnen Knoten statt, da Knoten auf Ergebnisse anderer Knoten warten müssen, falls diese noch nicht berechnet und verschickt wurden. Auf die Kommunikationsmöglichkeiten und deren Ausnutzung werde ich später eingehen. Ein weiteres Problem stellt die gleichmäßige Auslastung der Knoten zwischen diesen Synchronisationspunkten dar. Dies ist eine notwendige Voraussetzung für den angestrebten linearen Speedup.

5.6.3 Gleichmäßige Auslastung der Knoten

Die gleichmäßige Auslastung der Knoten in den Schritten 4,6 und 8 ist durch die Aufteilung der Vektoren auf die Knoten sichergestellt. Jeder Knoten erhält annähernd

die gleiche Anzahl von Komponenten. Somit muß noch sichergestellt werden, daß die Berechnungen der Knoten in den Schritten 1 und 3 annähernd gleich lange dauern. Die Laufzeit dieser Schritte hängt von der Anzahl und der Art der Einträge in den Spalten und Zeilen der Matrix B ab, die dem jeweiligen Knoten zugeteilt werden. Die Anzahl der Komponenten eines Knotens legt auch die Anzahl der ihm zugeteilten Zeilen von B und B^T fest. Somit benötigt man einen Algorithmus, der eine Aufteilung der Zeilen und Spalten (bzw. der Zeilen von B^T) von B auf die einzelnen Knoten vornimmt, die eine gleiche Auslastung der Knoten gewährleistet. Da die zyklische Verteilung der Vektorkomponenten (und damit auch der Zeilen und Spalten von B) einige programmiertechnische Vorteile hat (z.B. Einfachheit des Codes, leichte Berechnung der Indizes ohne zusätzliche Daten, etc.), ist das Ziel des Algorithmus eine Umnumerierung der Zeilen und Spalten zu berechnen, so daß deren zyklische Aufteilung die geforderten Eigenschaften hat. Bei diesem Problem handelt es sich um ein $\mathcal{NP}$ vollständiges Problem (Scheduling Task Problem)(vgl. [28], [14]). Somit ist im allgemeinen die Bestimmung einer optimalen Verteilung der Zeilen und Spalten sehr zeitaufwendig. Man erzielt sehr oft bei Problemen dieser Art sehr gute Näherungen mit Greedy Algorithmen (vgl. Kapitel 17 in [5]). Der von mir vorgestellte Algorithmus zur Aufteilung der Spalten und Zeilen der Matrix B gehört in die Klasse der Greedy Algorithmen. In Kapitel 3 wurde die Gewichtsverteilung der Spalten und Zeilen untersucht. (Man beachte dabei, daß bei COS-Systemen Zeilen und Spalten vertauscht sind.) Die Tabelle 5.14 enthält für das DL85 und das COS85 Beispiel die maximalen und minimalen Spalten- und Zeilengewichte der einzelnen Zeilen und Spalten des gesamten Systems sowie der kumulierten Spalten- und Zeilengewichte der den Knoten zugeordneten Blöcke. Dieses Gewicht bezeichne ich im folgenden als kumuliertes Zeilengewicht der Knoten. Dabei wurden die Zeilen und Spalten des gesamten Systems zyklisch auf 64 Knoten aufgeteilt.

Beispiel	Gesamtsystem				Knoten			
	Spalten		Zeilen		Spalten		Zeilen	
	min.	max.	min.	max.	min.	max.	min.	max.
DL85	5	274	6	95 889	146 715	150 523	118 339	226 704
COS85	3	59 537	16	450	64 718	209 980	81 178	86 603

Tabelle 5.14: Extremwerte der zyklischen Spalten- und Zeilenaufteilung

Danach scheint das Problem, eine geeignete Zeilenaufteilung zu finden, wegen der größeren Unterschiede im Gewicht der Zeilen schwieriger zu sein. Im folgenden beschreibe ich meinen Algorithmus, der die Zeilen und Spalten auf eine Anzahl (k) von Knoten aufteilt anhand der Aufteilung der Zeilen. (Falls nicht anders angegeben, gilt $k = 64$.) Der Algorithmus bestimmt zuerst ein leicht modifiziertes Zeilengewicht jeder Zeile. In dem modifizierten Zeilengewicht wird auch die Art der Einträge berücksichtigt und nicht nur deren Anzahl. Das Gewicht $gew[i]$ einer Zeile i wird

definiert als

$$gew[i] := cache1 \cdot \left(\omega_1 + \omega_2 \cdot \frac{T(Sub)}{T(Add)}\right) + cache3 \cdot \left(\omega_{3.1} \cdot \frac{T(kmult_N)}{T(Add)} + \omega_{3.2} \cdot \frac{T(kmult_F)}{T(Add)}\right) \ ,$$

wobei die Bezeichnungen von Abschnitt 5.5 sinngemäß benutzt werden (vgl. Bemerkung 5.2 (Seite 73)). Der Algorithmus bestimmt am Anfang das Gewicht $gew[i]$ jeder Zeile, das Gesamtgewicht aller Zeilen sowie das maximale und minimale Zeilengewicht. Für jeden Knoten wird die Anzahl der Zeilen ermittelt und gepflegt, die ihm noch zugeteilt werden müssen. Außerdem wird das Gewicht der Zeilen, die ihm bereits zugeteilt sind, gespeichert. Zusammen mit dem durchschnittlichen Gewicht, das jedem Knoten insgesamt zugeteilt werden muß, kann man so zu jedem Zeitpunkt das durchschnittliche Gewicht berechnen, das diejenigen Zeilen haben sollen, die diesem Knoten noch zugeteilt werden. Dann beginnt der Algorithmus mit der Aufteilung der Zeilen. Als erstes werden die k schwersten Zeilen auf die einzelnen Knoten verteilt, so daß ein Knoten mit einer höheren Nummer eine schwerere Zeile erhält. (Diese Reihenfolge ist von Vorteil, da bei einer zyklischen Aufteilung die Knoten mit kleiner Nummer eventuell eine Zeile mehr haben können als die anderen Knoten.) Die Tabelle 5.15 listet für jeden Knoten, die von 0 an durchnumeriert sind, sein kumuliertes Zeilengewicht, nachdem die erste Zeile zugeordnet wurde. Die Daten stammen aus dem COS85 Beispiel.

Nr.	Gewicht	Nr.	Gewicht	Nr.	Gewicht	Nr.	Gewicht	Nr.	Gewicht
63	59537	62	55059	61	55041	60	51565	59	50354
58	47959	57	44176	56	43493	55	33631	54	33617
53	29247	52	28913	51	27468	50	25350	49	23928
48	22494	47	22290	46	20949	45	20790	44	20742
43	19791	42	17762	41	17660	40	16668	39	16095
38	14909	37	14079	36	13885	35	12732	34	12662
33	12469	32	12059	31	12054	30	12031	29	11903
28	11667	27	10906	26	10744	25	10190	24	10018
23	9863	22	9855	21	9808	20	9795	19	9645
18	9215	17	8932	16	8808	15	8384	14	8227
13	8141	12	8023	11	7983	10	7966	9	7731
8	7726	7	7599	6	7359	5	7349	4	7185
3	7184	2	7072	1	6942	0	6649		

Tabelle 5.15: Startwerte der Zeilenaufteilung bei COS85

Das System hat insgesamt 51 907 Zeilen, so daß bis auf die ersten 3 Knoten allen Knoten 811 Zeilen zugeordnet werden. Dann wird eine Schleife solange durchlaufen, bis alle Zeilen aufgeteilt sind. In jedem Durchlauf wird denjenigen Knoten eine Zeile

zugeordnet, denen noch am meisten Zeilen zugeordnet werden müssen. Dabei werden in jedem Durchlauf zwei Ziele verfolgt:

- versuche mit möglichst vielen Knoten das durchschnittlichen Gewicht aller Knoten zu erreichen und
- versuche möglichst viele schwere Zeilen Knoten zuzuordnen.

Diese Ziele führen in jedem Schleifendurchlauf zu einer Einteilung der Knoten in 3 Gruppen, in denen auf unterschiedliche Art und Weise den Knoten eine Zeile zugeordnet wird.

- Dem Knoten mit dem kleinsten kumulierten Zeilengewicht der ihm zugeordneten Zeilen wird die Zeile mit dem maximalen Zeilengewicht zugeordnet.
- Den Knoten, deren kumuliertes Zeilengewicht unter dem aktuellen Durchschnittsgewicht aller Knoten liegt, wird eine Zeile mit dem entsprechenden Gewicht zugeordnet, so daß sein kumuliertes Zeilengewicht möglichst dicht an dem Durchschnittsgewicht liegt.
- Den Knoten, die mit ihrem Zeilengewicht über dem Durchschnittsgewicht liegen, werden Zeilen mit dem noch durchschnittlich benötigten Gewicht zugeordnet, so daß dem Knoten am Ende der Berechnung das geforderte Gewicht zugeordnet wurde.

Die Tabelle 5.16 stellt die Extremwerte des Zeilengewichtes, das den Knoten zugeordnet wurde sowie die Extremwerte des Zeilengewichtes, der noch nicht zugeordneten Zeilen während deren Berechnung dar. Werden auf die oben angegebene Weise alle Zeilen des Systems den 64 Knoten zugeordnet, so erhält man sehr gute Verteilungen.

zugeordnete	**Knoten**			**Restsystem**	
Zeilen	Minimum	Maximum	Mittel	Minimum	Maximum
1	6649	59537	18505	3	6625
10	32711	64181	35964	3	858
100	57340	66341	57452	3	127
400	72541	73659	72574	3	81
811	83901	83912	83909	0	0

Tabelle 5.16: Extremwerte der Zeilenaufteilung

Im diesem Beispiel liegt das maximale Gewicht, das einem Knoten zugeordnet wird, nur 0.0035% über dem Mittelwert und das minimale Gewicht nur 0.0095% unter dem

Mittelwert. Somit kann von einer gleichen Auslastung der Knoten ausgegangen werden. Die Abbildung 5.8 stellt das kumulierte Zeilengewicht der einzelnen Knoten dar, nachdem verschieden viele Zeilen aufgeteilt wurden. Nach Zuteilung der schwersten Zeilen (Startverteilung (grüne Linie)) sind die Unterschiede zwischen den einzelnen Knoten noch ziemlich groß. Werden mit obiger Strategie weitere 9 Zeilen zugeordnet (blaue Linie), so ist das kumulierte Gewicht von über 50 Knoten schon fast gleich. Diese Tendenz, daß immer mehr Knoten das gleiche Gewicht zugeordnet wird, verstärkt sich (rote Linie) bis schließlich am Schluß allen Knoten annähernd das gleiche Gewicht zugeteilt wird (schwarze Linie).

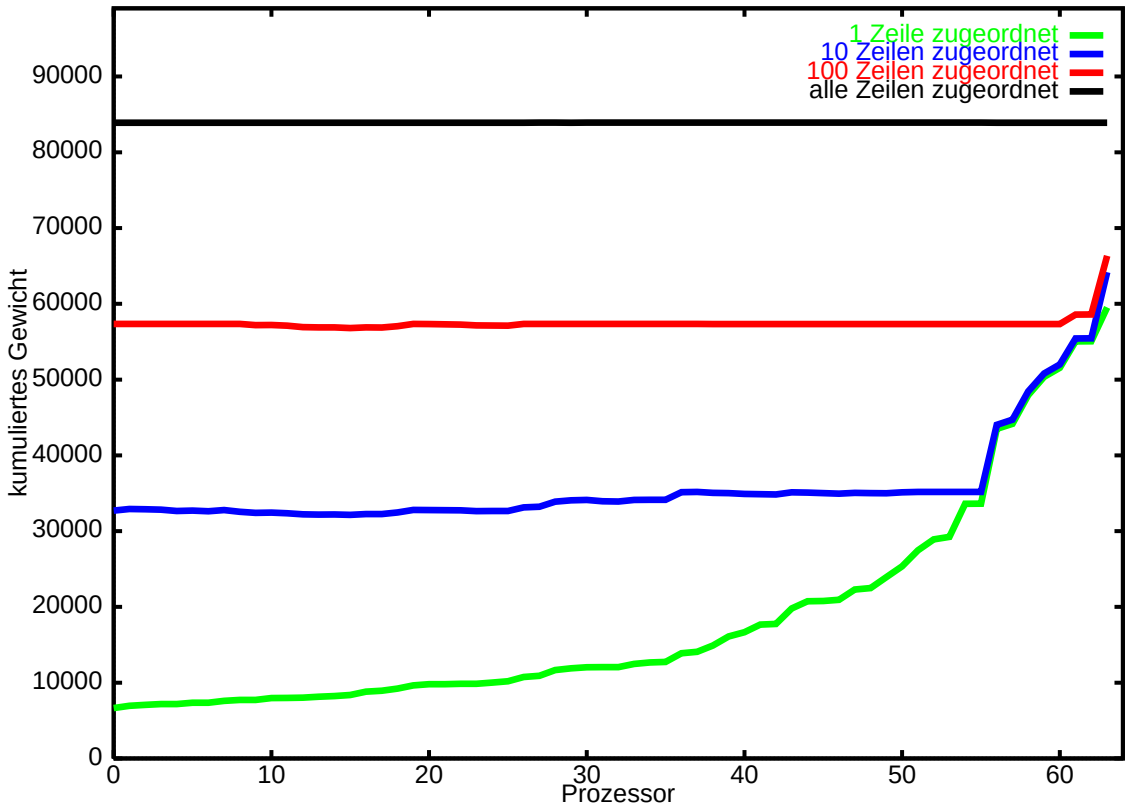


Abbildung 5.8: Verteilung des zugeteilten Zeilengewichtes

Praktische Erfahrungen haben gezeigt, daß man sehr gute Ergebnisse mit diesem Aufteilungsverfahren erzielen kann, solange das mittlere Gewicht, das einem Knoten zugeteilt werden soll, größer ist als das maximale Zeilengewicht im System. Hat eine Zeile des Systems alleine schon ein Gewicht, das größer ist als das Gewicht, das einem Knoten insgesamt zugeteilt werden soll, so kann man natürlich keine optimale Verteilung finden. In diesem Fall müßte die Spalte auf mehrere Knoten aufgeteilt werden. Dies stellt einen nicht unerheblichen Programmier- und Organisationsaufwand dar, der nicht im Verhältnis zur erzielten Beschleunigung durch Benutzung von 10 oder 20 zusätzlichen Knoten steht. Die Aufteilung einer Spalte auf mehrere Knoten wurde deshalb nicht durchgeführt, stellt aber eine mögliche Erweiterung meiner parallelen Implementierung dar.

Die Tabelle 5.17 stellt die Güte der Verteilung der Zeilen auf die einzelnen Knoten dar. Für verschiedene Anzahlen von Knoten wurde jeweils die Verteilung berechnet und der maximale, minimale und durchschnittlichen Wert des zugeteilten Gewichtes ermittelt. In jedem Fall sind auch die Abweichungen der Extremwerte von dem durchschnittlichen Wert in Prozent angegeben. Im COS85 Beispiel kann bis zu einer Anzahl von 80 Knoten eine gute Verteilung berechnet werden. Danach werden die Unterschiede zwischen dem Mittelwert und dem Maximalwert sehr groß. Bei der Anzahl der Knoten ist in Klammern jeweils noch die Anzahl der Knoten angegeben, die mehr als 0.1% über dem Mittel liegen.

Anzahl	**zugeteiltes Gewicht**			**Abweichung**	
Knoten	Minimum	Maximum	Mittel	untere	obere
64 (0)	83901	83912	83909	0.0095 %	0.0035 %
80 (1)	67121	67309	67127	0.0089 %	0.27 %
100 (5)	52725	66814	53702	1.81 %	24.41 %
128 (8)	40159	65479	41949	4.26 %	56.09 %

Tabelle 5.17: Verteilungsergebnisse bei variabler Knotenanzahl (COS85)

Führt man den selben Test mit dem DL85 Beispiel durch, so erhält man sehr gute Verteilungen bis zu einer Knotenanzahl von 100 Knoten. Erst dann werden die Abweichungen vom Mittelwert so groß, daß man nicht mehr mit einer gleichmäßigen Auslastung der Knoten rechnen kann. Die Tabelle 5.18 enthält die Daten der Verteilungen für das DL85 Beispiel.

Anzahl	**zugeteiltes Gewicht**			**Abweichung**	
Knoten	Minimum	Maximum	Mittel	untere	obere
64 (0)	141203	141233	141215	0.0084 %	0.0127 %
80 (0)	133437	133457	133446	0.0067 %	0.0082 %
100 (0)	106746	106771	106757	0.0103 %	0.0131 %
110 (2)	95614	104543	97052	1.481 %	7.718 %
120 (4)	86459	104681	88964	2.815 %	17.666 %
128 (4)	79951	104249	83404	4.140 %	24.992 %

Tabelle 5.18: Verteilungsergebnisse bei variabler Knotenanzahl (DL85)

Die mit dem Algorithmus ermittelten Aufteilungen führen zu einer fast ausgeglichenen Auslastung der Knoten während der Berechnung von $Bw = zwi$ und $B^T zwi = v$.

Die Tabelle 5.19 enthält für ausgewählte Knoten die gemessenen Zeiten für die einzelnen Phasen von B^TBw im DL85 Beispiel. Dabei sind die einzelnen Operationen in der Reihenfolge aufgelistet, in der sie auch im Algorithmus durchgeführt werden. Aus Effizienzgründen wird dabei die Reduzierung der Komponenten jeweils direkt am Ende der Schleife zur Berechnung von B_kmult_N und $B^T\_kmult\_N$ durchgeführt, direkt nachdem die Komponenten berechnet wurden. Zu diesem Zeitpunkt befinden sich die Daten noch in den Registern und müssen später nicht mehr geladen werden. Aus dem selben Grund wird auch die Berechnung der inneren Produkte $\langle w, v\rangle$ und $\langle v, v\rangle$ jeweils nach der Berechnung und Reduzierung der entsprechenden Komponente von v in $B^T\_kmult\_N$ durchgeführt. Nach Berechnung einer Komponente von v werden die bisher berechneten partiellen inneren Produkte um das Produkt der aktuell berechneten Komponente mit sich selbst ($\langle v, v\rangle$) bzw. der entsprechenden Komponente von w erhöht ($\langle w, v\rangle$). Dadurch liegt die Zeit zur Berechnung von B^Tzwi ungefähr um $0.23s$ höher als die Zeit zur Berechnung von Bw.

Operation	**Knoten 0**	**Knoten 54**	**Knoten 60**	**Knoten 62**	**Knoten 63**
B_Add	0.454	0.393	0.275	0.223	0.225
B_kmult_F	0.018	0.053	0.126	0.157	0.155
B_Sub	0.632	0.546	0.387	0.320	0.320
B_kmult_N	0.073	0.180	0.379	0.464	0.464
B_gesamt	1.177	1.172	1.167	1.164	1.164
B^T_Add	0.380	0.383	0.381	0.382	0.384
B^T_kmult_F	0.035	0.035	0.036	0.036	0.036
B^T_Sub	0.587	0.583	0.588	0.588	0.584
B^T_kmult_N	0.397	0.391	0.395	0.394	0.393
B^T_gesamt	1.399	1.392	1.400	1.400	1.397

Tabelle 5.19: Laufzeit von B^TBw auf der Paragon

Bei der Aufteilung der Spalten von B (Zeilen von B^T) gibt es keine großen Unterschiede bei den verschiedene Einträgen auf den einzelnen Knoten und somit auch bei den Rechenzeiten der einzelnen Schritte (vgl. untere Hälfte der Tabelle). Bei der Aufteilung der Zeilen sind dagegen deutliche Unterschiede zu erkennen. Während der Knoten 0 fast ausschließlich mit Einträgen mit Betrag 1 beschäftigt ist (B_Add, B_Sub), verbringen die Knoten mit den höheren Nummern die meiste Zeit mit den Einträgen mit Betrag größer als 1. Trotzdem benötigen alle Knoten fast die gleiche Zeit um ihre Komponenten des Vektors $zwi = Bw$ zu berechnen (vgl. B_gesamt).

Damit ist das Problem der gleichmäßigen Auslastung der Knoten (mit einer kleinen Einschränkung in der Anzahl der Knoten) gelöst. Als letztes Problem der Parallelisierung müssen die Kommunikationsschritte (Schritt 2, 5 und 7, vgl. Seite 83) realisiert werden.

5.6.4 Kommunikationsschritte

Bei der parallelen Implementierung des Lanczos Verfahrens müssen drei Kommunikationsschritte durchgeführt werden. Dabei handelt es sich bei den zwei zeitintensivsten Schritten um die Zusammensetzung eines Vektors, zu dem jeder Knoten einen Teil der Komponenten beiträgt und die Verteilung dieses Vektors auf alle Knoten. Bei der Reallisierung dieser Schritte spielt die gewählte Datenstruktur eine wichtige Rolle.
Verwendet man die Speicherallokierungsfunktionen der langen Arithmetiken, so kann man immer nur einzelne Komponenten der Vektoren austauschen bzw. verschicken, da die einzelnen Komponenten eines Vektors nicht notwendigerweise aufeinanderfolgende Speicherbereiche belegen (vgl. Abbildung 5.2 (Seite 62)). Verschickt man die Komponenten einzeln, so erhält man viele sehr kleine Nachrichten, deren Übertragung ineffizient ist. Sehr viel schneller kann ein einzelner aber dementsprechend größerer Speicherbereich übertragen werden. Man faßt die Komponenten eines Vektors deshalb in einem Block zusammen und erzeugt die notwendige Struktur des Speicherbereichs (vgl. Abbildung 5.9). Jetzt können alle Komponenten eines Vektors in einem Speicherblock verschickt werden. Da der maximal benötigte Speicherplatz pro Komponente bekannt ist, kann ausgeschlossen werden, daß während der Berechnungen eine Nachallokation (durch Systemfunktionen) notwendig wird, die zu einer Zerstörung der aufgebauten Speicherstruktur führen würde.

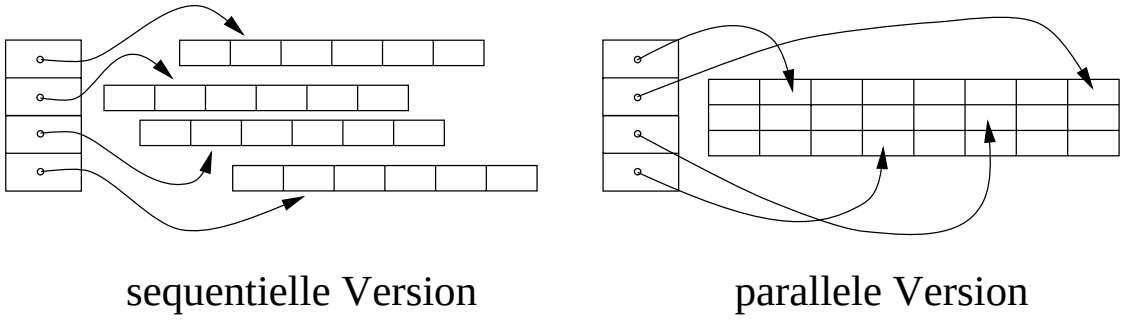


Abbildung 5.9: Vergleich Speicherung eines Langzahlvektors

Während einer Iteration muß ein Knoten auf alle Komponenten eines verteilten Vektors zugreifen können (Matrix-Vektormultiplikation) und er soll gleichzeitig einen schnellen Zugriff auf seine eigenen Komponenten haben (partielle innere Produktbestimmung). Dies wird durch die Tatsache erschwert, daß die einzelnen Komponenten zyklisch auf die einzelnen Knoten zugeordnet werden. Die Abbildung 5.10 enthält die Datenstruktur eines verteilten Vektors in der parallelen Version am Beispiel eines Vektors mit 12 Komponenten, die zyklisch auf 4 Knoten verteilt sind. Durch die einmalige Zuordnung der Komponenten in die einzelnen Blöcke kann man indiziert auf die Komponenten zugreifen. Den Beginn der Speicherblöcke, die die Komponenten eines Knoten enthalten, erhält man aus den Speicheradressen der ersten k Vektorkomponenten. Davon ausgehend kann man mit Hilfe des Speicherbedarfs einer Komponente schnell die Speicheradresse der nächsten Komponenten bestimmen.

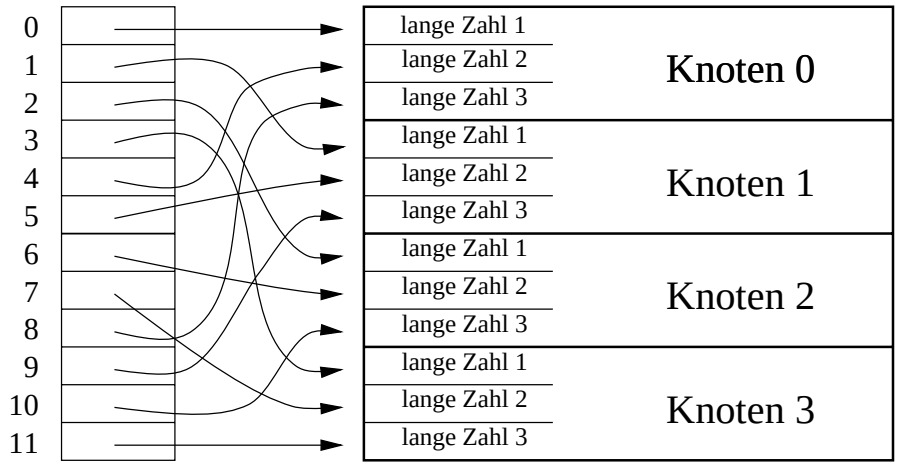


Abbildung 5.10: Aufbau verteilter Vektor

Beim Austausch der Vektoren muß also lediglich der aktuelle Speicherblock des entsprechenden Knotens ausgetauscht werden. Der folgende Algorithmus erzeugt die gewünschten Verweise. Zuerst werden einige Parameter (Anzahl der Knoten, Anzahl der Vektorkomponenten und Speicherplatzbedarf einer langen Zahl) bestimmt. Dann wird in Schritt 4 Speicherplatz für den gesamten Vektor allokiert. Dann werden nacheinander alle Blöcke in der äußeren Schleife durchlaufen. Die innere Schleife (Schritt 8 - 12) ordnet dann die Komponenten des aktuellen Blockes zu.

Aufbau der Datenstruktur eines verteilten Vektors

```
(1)  Bestimme Anzahl der Knoten (anz_nodes);
(2)  Bestimme Anzahl der Komponenten (anz_komp);
(3)  Bestimme Speicherplatz einer langen Zahl (next_num);
(4)  ptr = MALLOC(next_num · anz_komp);
(5)  i = 0;
(6)  while (i < anz_nodes) do
(7)     j = i;
(8)     while (j < anz_komp) do
(9)        vektor[j] = ptr;
(10)       ptr += next_num;
(11)       j += anz_nodes;
(12)    od
(13)    i ++ ;
(14) od
```

Mit Hilfe der vorgestellten Konstruktion ist die Voraussetzung für den Austausch

der verteilten Vektoren geschaffen. Ein weiterer Kommunikationsschritt ist die Berechnung der globalen inneren Produkte aus den partiellen inneren Produkten der einzelnen Knoten. Durch die Datenstruktur zur Speicherung der partiellen inneren Produkte (vgl. Abbildung 5.6 (Seite 82)) ist hier bereits sichergestellt, daß die Daten blockweise übertragen werden können. Bevor ich die genaue Realisierung der Kommunikationsschritte vorstelle, gehe ich auf die zwei prinzipiellen Möglichkeiten zur Durchführung von Kommunikation ein. Außerdem werden einige spezielle Kommunikationsbefehle aus dem Sprachumfang der Paragon erläutert.

Prinzipiell kann man zwischen zwei Arten von Kommunikation unterscheiden. Man spricht von *synchroner* Kommunikation, wenn der Sender bzw. der Empfänger einer Nachricht wartet, bis die Nachricht empfangen wurde. Von *asynchroner* Kommunikation spricht man, falls der Sender bzw. der Empfänger nicht durch das Verschicken bzw. Empfangen einer Nachricht blockiert wird. Bei synchroner Kommunikation ist die Übertragungsgeschwindigkeit normalerweiser höher, jedoch kann der Rechenprozessor während der Kommunikation keine anderen Berechnungen ausführen. Zum besseren Verständnis beider Kommunikationsarten erläutere ich das prinzipielle Vorgehen anhand der Abbildung 5.11. Der Knoten 1 möchte seinen Speicherbereich 1 an Knoten 2 verschicken, der ihn in Speicherbereich 2 empfangen will.

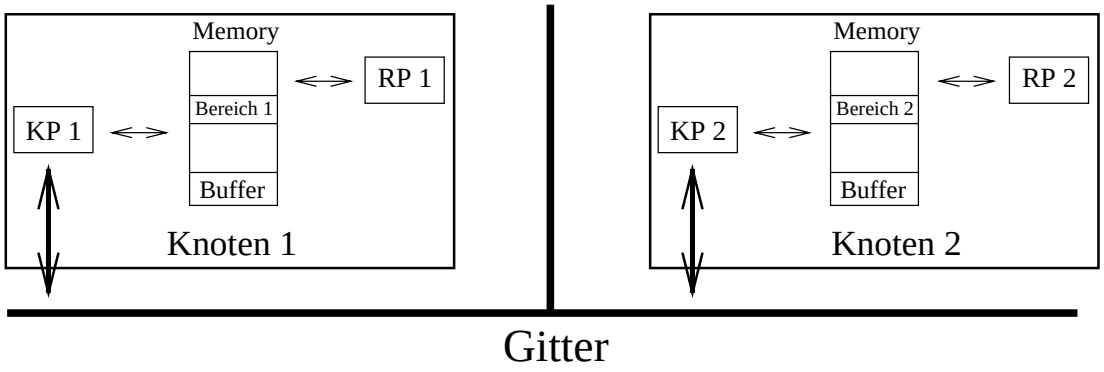


Abbildung 5.11: Abbildung zur Kommunikation

Ablauf asynchroner Kommunikation: Der Rechenprozessor RP 1 startet die asynchrone Kommunikation (*isend*), indem er dem Kommunikationsprozessor KP 1 den Speicherbereich (Beginn und Länge) mitteilt, den er an Knoten 2 verschicken will. Danach kann RP 1 mit den weiteren Berechnungen fortfahren. KP 1 schickt die Daten aus dem angegebenen Speicherbereich an KP 2. Wurde von RP 2 ein asynchrones Empfangskommando (*irecv*) für die entsprechende Nachricht abgesandt, so schickt KP 2 die empfangenen Daten direkt zum angegebenen Speicherbereich. Ist dies nicht der Fall, so speichert er sie in einen Systempuffer (*Buffer*). Erreicht KP 2 dann später das entsprechende Empfangskommando von RP 2, so wird die Nachricht in den nun bekannten Speicherbereich 2 kopiert. Beide Rechenprozessor können also weiterarbeiten und haben über sog. *Message-ID's* die Möglichkeit, auf die Beendigung

der Nachrichtenübertragung blockierend zu warten oder zu testen, ob die Datenübertragung komplett abgeschlossen wurde.

Ablauf synchroner Kommunikation: RP 1 startet die synchrone Kommunikation (*csend*), indem er wiederum KP 1 den Speicherbereich und den Adressaten der Nachricht mitteilt. RP 1 wartet, bis KP 1 ihm mitteilt, daß die Datenübertragung komplett abgeschlossen wurde. KP 1 teilt KP 2 mit, daß er eine Nachricht verschicken will. Die Ausführung eines synchronen Empfangsbefehls (*crecv*) von RP 2 führt dazu, das KP 2 die zu verschickende Nachricht von KP 1 empfängt und direkt in dem angegebenen Speicherbereich zur Verfügung stellt. Sind alle Daten korrekt übertragen worden, teilen die jeweiligen Kommunikationsprozessoren dies ihren Rechenprozessor mit. Nun können diese ihre weiteren Berechnungen ausführen.

Beide Formen der Kommunikation können auch miteinander gemischt werden (synchrones senden + asynchrones empfangen bzw. umgekehrt). Der Vorteil der synchronen Datenübertragung ist deren leichtere Benutzbarkeit und die Sicherheit, daß die übertragenen Daten (ohne den Umweg über den Puffer) direkt zur Verfügung stehen, wenn weitergerechnet wird. Der Vorteil der asynchronen Datenübertragung liegt in der Nebenläufigkeit der Kommunikation. Beide Rechenprozessoren können weiterarbeiten, während gleichzeitig Daten übertragen werden. Dies setzt jedoch voraus, daß diese Daten nicht direkt zur Weiterberechnung benötigt werden. Diese Betrachtungen führen auf der Paragon zu folgenden Regeln:

Bemerkung 5.3

- Nach Möglichkeit sollte asynchrone Kommunikation verwendet werden.
- Die asynchronen Empfangsbefehle sollten möglichst früh ausgeführt werden, damit der Umweg über den Systempuffer vermieden werden kann. Dabei muß man jedoch beachten, daß sich der Speicherbereich, in den ein asynchroner Empfang stattfindet, vom Zeitpunkt des Startens des Befehls bis zum vollständigen Abschluß der Kommunikation in einem undefinierten Zustand befindet.
- Stellt man sicher, daß keine Speicherseiten ausgeswappt werden (-plk switch bei Programmstart), so vereinfacht sich das Kommunikationsprotokoll der Kommunikationsprozessoren der Paragon, was auch einen spürbaren Performancegewinn zur Folge hat.

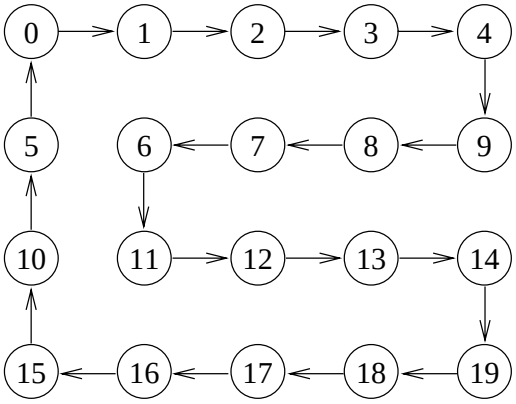
Nachdem nun die beiden prinzipiellen Möglichkeiten zur Kommunikation beschrieben wurden, werden noch zwei spezielle Kommunikationsbefehle erläutert. Sie gehören zu einer Familie von Kommunikationsbefehlen, die sehr effizient auf der Paragon implementiert wurden und die Beteiligung aller Knoten voraussetzen. Der jeweilige Befehl muß auf allen Knoten aufgerufen werden und die weiteren Berechnungen werden blockiert, bis der Befehl beendet wurde. Wird der Befehl nur von einem Knoten nicht aufgerufen, so wird die Berechnung aller anderen Knoten blockiert.

gcolx (Speicherstelle 1, Pointer auf Array, Speicherstelle 2)
Der Befehl konkateniert von allen Knoten Speicherbereiche bekannter Länge zu einem Speicherbereich. Die Länge der einzelnen Speicherbereiche der Knoten steht im Array, auf das das zweite Argument zeigt. Die Speicherbereiche, die von den einzelnen Knoten zusammengefügt werden, beginnen ab Speicherstelle 1. Der konkatenierte Speicherbereich ist auf allen Knoten nach Beendigung des gcolx Befehls ab Speicherstelle 2 vorhanden.

gopf (Speicherstelle 1, Länge, Speicherstelle 2, Pointer auf Funktion)
Es wird eine assoziative und kommutative benutzerdefinierte Funktion, auf die ein Pointer übergeben wird, auf allen Knoten aufgerufen. Dabei müssen zwei Speicherbereiche angegeben werden, in dem die Ausgangsdaten der Funktion stehen (Speicherstelle 1) und ein Zwischenspeicher für die Daten anderer Knoten (Speicherstelle 2). Das Ergebnis der Ausführung der globalen Operation wird auf allen Knoten ab Speicherstelle 1 zur Verfügung gestellt. Das zweite Argument enthält die Länge der Speicherbereiche, auf denen gearbeitet wird.

Zur Berechnung der globalen inneren Produkte wird die Funktion *gopf* benutzt. Die kommutative und assoziative Funktion, die in *gopf* benutzt wird, berechnet jeweils aus den partiellen inneren Produkten, die auf den zwei Knoten berechnet wurden, das partielle innere Produkt der beiden Knoten. Dazu müssen also lediglich die entsprechenden Teilsummen aufaddiert werden. Nach Beendigung des *gopf* Aufrufs sind dann alle globalen inneren Produkte auf allen Knoten verfügbar.
Der Befehl *gcolx* kann zum Austausch der Vektoren (vgl. Schritt 2,5 auf Seite 83) benutzt werden. Ein Nachteil der speziellen Kommunikationsbefehle ist ihre synchrone Ausführung. Um eine asynchrone Vergleichsmöglichkeit zu haben, wird eine zweite Möglichkeit zum Austausch der Vektoren entwickelt. Dazu wird aufbauend auf den Ideen in [18] ein Ring aufgebaut, der nach Möglichkeit immer zwei physikalisch benachbarte Knoten miteinander verbindet. Dazu wird jedem Knoten genau ein Nachfolger zugeordnet (vgl. Abbildung 5.12). (Im Anhang B befindet sich ein Programm zum Aufbau des Rings, das aufbauend auf der Arbeit von [3] entwickelt wurde.) Soll nun ein Vektor unter k Knoten ausgetauscht werden, so geschieht dies in $k-1$ Runden. Zuerst schickt jeder Knoten seinem Nachfolger eine Nachricht mit den eigenen Komponenten. In den darauffolgenden Runden schickt jeder Knoten dann die Komponenten weiter, die er in der letzten Runde empfangen hat. Es werden Tests mit zwei verschiedenen Nachfolgerfunktionen durchgeführt. Die erste Nachfolgerfunktion (Nf 1) führt zum Aufbau des dargestellten Rings (vgl. [18]). Damit ein solcher Ring aufgebaut werden kann, müssen die Knoten in einer rechteckförmigen Topologie mit mindestens einer Kanten mit gerader Anzahl von Knoten angeordnet sein. Dies stellt eine Einschränkung dar, da zum Teil auf die Topologie der Knoten kein Einfluß genommen werden kann. Deshalb wird auch noch eine zweite sehr einfach zu berechnende und universell einsetzbare Nachfolgerfunktion (Nf 2) angegeben, die jedoch zu einem für die Kommunikation vermeintlich schlechteren Ring führt. Sie ist folgendermaßen definiert:

$$\text{Nf } 2(i) := \begin{cases} i+1 & \text{falls } i \leq k-1 \ , \\ 0 & \text{falls } i = k-1 \ . \end{cases}$$



Nr.	Nf 1	Nr.	Nf 1	Nr.	Nf 1
0	1	7	6	14	19
1	2	8	7	15	10
2	3	9	8	16	15
3	4	10	5	17	16
4	9	11	12	18	17
5	0	12	13	19	18
6	11	13	14		

Abbildung 5.12: Aufbau eines Kommunikationsrings

Damit die Kommunikation asynchron ablaufen kann, benötigt man noch zusätzliche Kommunikationsbefehle. Man muß eine Nachricht asynchron empfangen können und danach dann das Weiterschicken der gerade empfangenen Nachricht anstoßen können. Im Befehlsumfang der Paragon existiert der Befehl *hrecv*, der eine Nachricht asynchron empfängt, das aufrufende Programm unterbricht, sobald die Nachricht empfangen wurde und einen benutzerdefinierten Handler aufruft. Sobald der Handler gestartet wurde, kann er mit dem aufrufenden Programm konkurrierend ablaufen. Der Handler und das Hauptprogramm benutzen auch den gleichen Speicher. In diesem Handler kann dann überprüft werden, ob die Nachricht weitergeschickt werden muß oder ob es sich um die Nachricht des eigenen Nachfolgers handelt. (Dieser benötigt seine eigenen Komponenten natürlich nicht noch einmal.) Der Algorithmus **Asynchroner Austausch eines verteilten Vektors (1)** stellt ein Beispiel für die Benutzung des *hrecv* Befehles dar. Anhand eines Zählers (*verschickt*), der im Hauptprogramm mit Eins initialisiert wird (Schritt 5) und im Handler pro empfangener Nachricht inkrementiert wird (Schritt 16), kann das aufrufende Programm feststellen, ob alle benötigten Nachrichten empfangen wurden (Schritt 28 - 36). Im Hauptprogramm wird in Schritt 23 die Funktion aufgerufen, die die Handler zum Empfangen und Weiterschicken der Komponenten der anderen Knoten startet. Nach dem Berechnen der eigenen Komponenten (Schritt 24) werden diese an den jeweiligen Nachfolger weitergeschickt, und damit die Verbreitung der eigenen Komponenten über dem Kommunikationsring gestartet. Danach kann der Knoten weiterrechnen. Treffen Komponenten von seinem Vorgänger auf dem Ring ein, wird der Handler (**send_part()**) gestartet, der gegebenenfalls die Komponenten zu dem Nachfolger weiterschickt. Sind die Berechnungen, die der Knoten ohne den verteilten Vektor durchführen konnte, abgeschlossen, so wartet der Knoten in der While-Schleife (Schritt 28 - 31) auf den Abschluß der Kommunikation. Dazu wird der **flick** Befehl benutzt. Er gibt die Kontrolle über die Zuteilung von Rechenzeit an den Scheduler für längstens 10 Millisekunden zurück, so daß das Hauptprogramm kaum Zeit verbraucht und andere Prozesse diese Zeit komplett nutzen können. Wird vom Handler der Wert für die empfangenen Komponenten (*verschickt*) auf die Anzahl der Knoten gesetzt, so verläßt der Knoten die Schleife und kann mit dem verteilten Vektor weitere Berechnungen durchführen.

Asynchroner Austausch eines verteilten Vektors (1)

```
(1)  Bestimme Anzahl der Knoten (anz_nodes);
(2)  Bestimme eigene Knotennummer (me);
(3)  Berechne Nachfolgerfunktion (nachf[me]);
```

FUNKTION: STARTEN DES HANDLERS

```
(4)  void start_hrecv()
(5)  verschickt = 1;
(6)  i = 0;
(7)  while (i < anz_nodes) do
(8)    if (i ≠ me) then
(9)      /* starte hrecv für Nachricht mit Komponenten von Knoten
         i. Benutze Handler send_part */
(10)     hrecv(400 + i,Speicherstelle,Länge,send_part);
(11)   fi
(12)   i ++ ;
(13) od
```

FUNKTION: DER HANDLER

```
(14) void send_part(type,count,node,pid)
(15) long type, count, node, pid;
(16) verschickt ++;
(17) /* Enthält Nachricht die Komponenten meines Nachfolgers ? */
(18) if (type ≠ nachf[me] + 400) then
(19)   /* schicke empfangene Nachricht weiter */
(20)   csend(type,Speicherstelle,Länge,nachf[me], 0);
(21) fi
```

HAUPTPROGRAMM

```
(22) void main()
(23) start_hrecv();
(24) Berechne eigene Vektorkomponenten;
(25) /* Verschicke eigenen Block an Nachfolger */
(26) csend(400 + me,Speicherstelle,Länge,nachf[me], 0);
(27) Führe weitere Berechnungen aus (ohne verteilten Vektor);
(28) while (verschickt ≠ anz_nodes) do
(29)   /* Warte bis alle Komponenten empfangen wurden */
(30)   flick();/* Gibt Kontrolle an den Scheduler zurück */
(31) od
(32) Führe weitere Berechnungen aus (mit verteilten Vektor);
```

Bemerkung 5.4 Bei der Benutzung des Handlerkonzepts muß man immer beachten, daß die beiden Prozesse gleichzeitig laufen können und auf dem selben Hauptspeicher arbeiten. Dies ist vorteilhaft zum Austausch von Informationen untereinander, kann aber auch zu ungewollten Effekten führen. Es existiert zwar ein Mechanismus, der das Hauptprogramm vor der Unterbrechung durch einen Handler schützt, dieser kann jedoch nicht dazu benutzt werden, um sicherzustellen, daß nicht gleichzeitig zwei Handler laufen. Während der Entwicklung traten Probleme auf, wenn der Handler und das Hauptprogramm die Nachrichten asynchron verschickten. Dabei wird jeder Nachricht vom Betriebssystem eine sogenannte *Message-ID* zugeordnet. Die dafür benötigte Datenstruktur gehört zum gemeinsamen Speicher und kann von jedem Handler verändert werden. Je nach dem zeitlichen Eintreffen der Nachrichten, kam es dann zur Zerstörung dieser Struktur durch einen Handler und den zeitgleich aktiven Hauptprozeß, der seine Startnachricht verschickte. Deshalb werden in der Variante 1 alle Startnachrichten synchron verschickt. Will man die Startnachricht asynchron versenden, so muß man das Codesegment zum Verschicken der asynchroner Nachrichten vor Unterbrechung schützen. Möchte man auch mit asynchroner Übertragung durch die Handler arbeiten, so muß man sicherstellen, daß die Handler nicht gleichzeitig aktiv sind. Dazu kann man die einzelnen *hrecv* Kommandos nacheinander ausführen, damit immer nur ein Handler aktiviert werden kann (Variante 2). Jeder Handler startet nach dem Verschicken der gerade empfangenen Komponenten das *hrecv* Kommando zum Empfang der nächsten Komponente, bis alle Komponenten empfangen wurden. Benutzt man die Ringkommunikation und schützt die asynchrone Startnachricht vor der Unterbrechung durch einen Handler, so haben praktische Erfahrungen gezeigt, daß man bei gleichmäßiger Auslastung der Knoten davon ausgehen kann, daß es zu keinen Überlagerungen durch die Handler kommt. (Alle Nachrichten, die bei einem Knoten eintreffen, werden von dem selben Knoten abgeschickt.) Zum asynchronen Verschicken der Nachrichten im Handler wird Zeile 20 ersetzt durch

$$\text{mess\_id} = \text{isend}(type, \text{Speicherstelle}, \text{Länge}, nachf[me], 0); \text{msgignore}(\text{mess\_id});$$

wobei *mess_id* eine lokale Variable von Typ *long* ist und der Befehl *msgignore* die benutzte Message-ID wieder frei gibt. Dies führt zur Variante 3, in der sowohl im Hauptprogramm als auch im Handler asynchrone Kommunikation benutzt wird. Da keine Laufzeitunterschiede zwischen den verschiedenen Nachfolgefunktionen festgestellt werden konnten, wird in Zukunft wegen der universellen Einsetzbarkeit immer Nf 2 benutzt.

Die Tabelle 5.20 enthält die Laufzeit der Operation *gopf*, *gcolx* und der hier vorgestellten asynchronen Varianten des Vektoraustauschs. Dabei wurden die Tests für verschiedene Anzahl von Knoten durchgeführt. Bei den Messungen betrug die Gesamtlänge des verteilten Vektors jeweils 3 072 496 Bytes, was einem Vektor mit 54 866 Komponenten zur Speicherung von 10^{85}-stelligen Zahlen entspricht. Es wurde angenommen, daß 10 Lösungen berechnet werden, so daß bei der inneren Produktberechnung (*gopf*) zur Speicherung der partiellen inneren Produkte jeweils 672 Bytes pro Knoten gebraucht wurden. In diesen Tests wurde nur auf die Beendigung der

Kommunikation gewartet, d.h. auch bei den asynchronen Varianten wurden keine Berechnungen während dieser Zeit durchgeführt.

Anzahl	synchron		asynchroner Vektoraustausch		
Knoten	gopf	gcolx	Variante 1	Variante 2	Variante 3
16	0.00065 s	0.08777 s	0.1485 s	0.2550 s	0.0913 s
36	0.00088 s	0.10509 s	0.2093 s	0.2782 s	0.1117 s
64	0.00103 s	0.09558 s	0.2475 s	0.3429 s	0.1260 s
80	0.00113 s	0.08851 s	0.2768 s	0.3497 s	0.1325 s
128	0.00125 s	0.12753 s	0.3925 s	0.4859 s	0.2033 s

Tabelle 5.20: Laufzeit der Kommunikationsbefehle

Der synchrone Kommunikationsbefehl *gcolx* führt die Konkatenation eines verteilten Vektors sehr schnell durch. Dabei paßt sich der Befehl mit seiner Kommunikationsstruktur der Topologie der allokierten Knoten an (vgl. [18]), was seine relativ stabilen Laufzeiten bei mehreren Knoten erklärt. Betrachtet man die Laufzeiten seiner asynchronen Alternativen, so wächst bei Variante 1 und 2 die Laufzeit wesentlich stärker in Abhängigkeit der Anzahl der beteiligten Knoten. Die Zeitdifferenz zwischen Variante 1 und 2 liegt wohl daran, daß Empfangskommandos auch nach dem eigentlichen Empfang der Nachricht gestartet werden können. Bei Variante 1 und 3 wurde sichergestellt, das alle Empfangskommandos vor dem Verschicken der ersten Nachricht gestartet wurden. Da selbst die asynchrone Variante 3 fast doppelt so lange dauert als der synchrone Austausch mit *gcolx*, ist es nur sinnvoll, die asynchrone Variante zu benutzen, wenn man längere Berechnungen ohne die zu übertragenden Vektoren durchführen kann, während die Vektoren übertragen werden. Ist dies nicht möglich, verzichtet man auf die Überlagerung der Kommunikation durch Berechnungen und führt den Austausch der Vektoren mit dem *gcolx* Befehl durch. Anhand der Laufzeit zur Berechnung der globalen inneren Produkte (*gopf*) wird deutlich, daß hier nicht nach einer asynchronen Alternative gesucht werden muß.

Bevor ich auf die Frage nach der Möglichkeit zur Überlagerung der Kommunikation eingehe, möchte ich noch eine untersuchte Alternative zur vorgestellten asynchronen Variante beschreiben. Die Hauptidee beim Aufbau des Kommunikationsrings ist die Benutzung von disjunkten Kanten zwischen den einzelnen Knoten, die möglichst kurz sind. Dabei hofft man, daß es keine Wechselwirkungen zwischen den verschickten Nachrichten gibt. Diese Wechselwirkungen kann man beobachten, wenn man den Austausch des Vektors durch *Broadcasts* (jeder Knoten schickt an alle anderen Knoten seine Komponenten) zeitgleich ausführen läßt. Betrachtet man die Beschreibung der Paragon, so fällt auf, das rein technisch gesehen die Kommunikation zwischen zwei Knoten in beide Richtungen unabhängig voneinander möglich ist. Daraus folgt, daß man untersuchen sollte, ob die Kommunikation in dem vorgestellten Ring auch

durch Verschicken der Nachrichten in beide Richtungen beschleunigt werden kann. Dazu wird für jeden Knoten zusätzlich noch sein Vorgänger im Kommunikationsring benötigt (vgl. Abbildung 5.13). Die roten Kanten stellen die Vorgängerverweise und die blauen Kanten die Nachfolgerverweise dar.

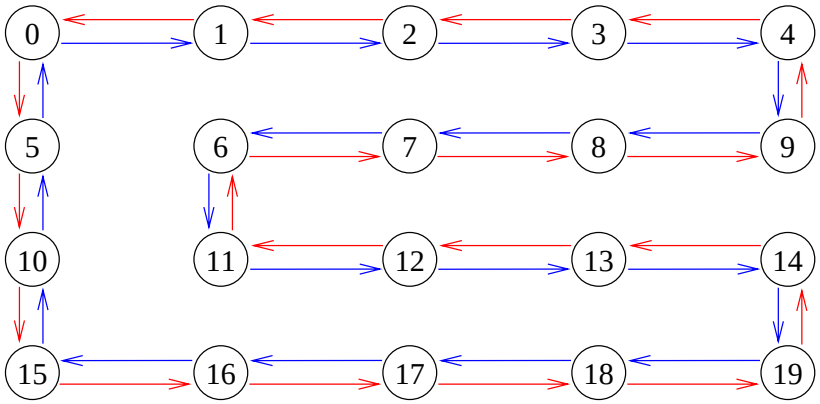


Abbildung 5.13: Aufbau eines doppelten Kommunikationsrings

Gestartet wird der Austausch eines Vektors jetzt dadurch, daß jeder Knoten seine Komponenten zu seinem Vorgänger und Nachfolger schickt. Die Vorgehensweise beim Empfang von Nachrichten wird komplizierter, da nicht jede Nachricht einfach weitergeschickt wird. Sowohl der Vorgänger als auch der Nachfolger können die Komponenten schon von ihrem Vorgänger oder Nachfolger erhalten haben. Es werden insgesamt drei Handler benötigt, die die empfangenen Komponenten zum Nachfolger, Vorgänger oder gar nicht weiterschicken. Die Funktion zum Starten der Handler (**start_hrecv**) muß entscheiden, welche Komponenten mit welchen Handler empfangen werden. Zum Aufbau des Rings wird ein Array *Feld* angelegt, das die logischen Nummern der Knoten in der Reihenfolge ihres Auftretens im Ring enthält (vgl. Anhang B). Dieses Array wird auch in der Funktion zum Starten der Handler benutzt, um den einzelnen Nachrichten die richtigen Handler zuzuordnen. Dabei sucht zuerst jeder Knoten seine eigene Position im Ring. Von dort kann die Reihenfolge der Nachrichten ermittelt werden, die von Vorgänger und von Nachfolger eingehen. Diese Nachrichten werden solange an den Nachfolger bzw. den Vorgänger weitergeschickt, bis jeder Knoten $k-1$ Nachrichten verschickt hat. Man beachte dabei, daß jeder Knoten an Anfang seine Komponenten zweimal verschickt. Deshalb dürfen zwei empfangene Nachrichten nicht mehr weitergeschickt werden, da sonst diese Nachrichten doppelt bei einem Knoten eintreffen würden. Die Funktion zum Starten der Handler stellt auch sicher, daß jede Komponente bei jedem Knoten eintrifft. Prinzipiell erhält ein Knoten jetzt die Hälfte der Komponenten von seinem Vorgänger und die Hälfte der Komponenten von seinem Nachfolger, so daß eine Halbierung der Anzahl der Kommunikationsrunden stattfindet. Die Programmierung der Handler wird einfacher, da beim Starten der Handler bereits eine Einteilung der Nachrichten erfolgt.

FUNKTION: STARTEN DES HANDLERS

```
(1)  void start_hrecv()
(2)  verschickt = 1;
(3)  mess = 2;
(4)  /* suche eigne Position in Ring */
(5)  i = 0;
(6)  while (Feld[i] != me) do
(7)     i ++ ;
(8)  od
(9)  j = i;
(10) while (mess < anz_nodes − 1) do
(11)    j ++ ;
(12)    if (j == anz_nodes) then
(13)       j = 0;
(14)    fi
(15)    /* starte hrecv für Nachricht vom Nachfolger */
(16)    hrecv(400 + Feld[j],Speicherstelle,Länge,send_part_vorg);
(17)    mess ++ ; i −− ;
(18)    if (i == −1) then
(19)       i += anz_nodes;
(20)    fi
(21)    if (mess < anz_nodes − 1) then
(22)       /* starte hrecv für Nachricht vom Vorgänger */
(23)       hrecv(400 + Feld[i],Speicherstelle,Länge,send_part_nachf);
(24)    else
(25)       /* starte nur Empfang der Nachricht */
(26)       hrecv(400 + Feld[i],Speicherstelle,Länge,send_part);
(27)    fi
(28)    mess ++ ;
(29) od
(30) j ++ ; i −− ;
(31) if (j == anz_nodes) then
(32)    j = 0;
(33) fi
(34) /* starte nur Empfang der Nachricht */
(35) hrecv(400 + Feld[j],Speicherstelle,Länge,send_part);
(36) if (i == −1) then
(37)    i += anz_nodes;
(38) fi
(39) if (j ≠ i) then
(40)    /* starte nur Empfang der Nachricht */
(41)    hrecv(400 + Feld[i],Speicherstelle,Länge,send_part);
(42) fi
```

FUNKTION: DIE HANDLER

HANDLER FÜR NACHRICHTEN VOM NACHFOLGER

```
(1)  void send_part_vorg(type,count,node,pid)
(2)  long type, count, node, pid;
(3)  verschickt ++;
(4)  /* schicke empfangene Nachricht an Vorgänger weiter */
(5)  csend(type,Speicherstelle,Länge,vorg[me],0);
```

HANDLER FÜR NACHRICHTEN VOM VORGÄNGER

```
(6)  void send_part_nachf(type,count,node,pid)
(7)  long type, count, node, pid;
(8)  verschickt ++;
(9)  /* schicke empfangene Nachricht an Nachfolger weiter */
(10) csend(type,Speicherstelle,Länge,nachf[me],0);
```

HANDLER NUR ZUM EMPFANGEN VON NACHRICHTEN

```
(11) void send_part(type,count,node,pid)
(12) long type, count, node, pid;
(13) verschickt ++;
```

Das Hauptprogramm kann fast unverändert bleiben. Es muß lediglich an Anfang noch der Vorgänger bestimmt werden und nach Zeile 26 muß noch

$$\textbf{csend}(400 + me, \text{Speicherstelle,Länge,}vorg[me], 0);$$

eingefügt werden, damit auch der Vorgänger die Komponenten des Knotens erhält. Die Tabelle 5.21 enthält die empfangenen und verschickten Nachrichten für die ersten drei Knoten (0,1, und 2) aus den Ring (vgl. Abbildung 5.12).

Dabei wird jeweils unterschieden, ob an den Nachfolger (**N**) oder den Vorgänger (**V**) eine Nachricht geschickt wird bzw. von ihm eine Nachricht empfangen wird. Da Knoten 1 der Nachfolger von Knoten 0 ist, müssen die vierte Spalte von Knoten 0 (an den Nachfolger verschickte Nachrichten) und die erste Spalte von Knoten 1 (von dem Vorgänger empfangene Nachrichten) identisch sein. Die Nachrichten, die von den Knoten nicht weitergeschickt werden, sind Nachricht 12 und 13 für Knoten 0 und 1, und die Nachrichten 14 und 19 für Knoten 2. Bei allen anderen Nachrichten schicken die Knoten, die von ihren Vorgänger empfangenen Nachrichten an ihren Nachfolger weiter und umgekehrt. Dies kann man an der ersten und vierten Spalte sowie der zweiten und dritten Spalte eines Knotens schön beobachten.

Die gemessenen Laufzeiten mit der doppelten Ringkommunikation sind jedoch deutlich schlechter als die Zeiten für die Kommunikation im einfachen Ring (Variante

Knoten 0				Knoten 1				Knoten 2			
empfängt		verschickt		empfängt		verschickt		empfängt		verschickt	
V	N	V	N	V	N	V	N	V	N	V	N
5	1	0	0	0	2	1	1	1	3	2	2
10	2	1	5	5	3	2	0	0	4	3	1
15	3	2	10	10	4	3	5	5	9	4	0
16	4	3	15	15	9	4	10	10	8	9	5
17	9	4	16	16	8	9	15	15	7	8	10
18	8	9	17	17	7	8	16	16	6	7	15
19	7	8	18	18	6	7	17	17	11	6	16
14	6	7	19	19	11	6	18	18	12	11	17
13	11	6	14	14	12	11	19	19	13	12	18
12		11			13		14	14		13	

Tabelle 5.21: Empfangene und verschickte Nachrichten

3) (vgl. Tabelle 5.20 S. 98). In diesem Test wurde bei der doppelten Ringkommunikation wieder mit beiden Nachfolgerfunktionen experimentiert. In der Tabelle sind die Zeiten für die erste Nachfolgerfunktion angegeben. Die Zeiten für die zwei Nachfolgerfunktionen unterschieden sich jedoch (wie auch schon bei der einfachen Ringkommunikation) nicht stark voneinander.

Anzahl Knoten	Kommunikationsring	
	einfach	doppelt
16	0.0913 s	0.1025 s
36	0.1117 s	0.1339 s
64	0.1260 s	0.1617 s
80	0.1325 s	0.1688 s

Tabelle 5.22: Vergleich einfacher und doppelte Ringkommunikation

Obwohl bei der doppelten Ringkommunikation im Schnitt nur ungefähr die Hälfte Runden benötigt werden, liegen die Übertragungszeiten mit doppelter Ringkommunikation deutlich über den Zeiten bei einfacher Ringkommunikation. Nach Rücksprache mit den Systemfachleuten in Jülich kann dieses Ergebnis dadurch erklärt werden, daß es nur einen Kommunikationsprozessor gibt, dessen Cache zusätzlich auch nicht für beide Nachrichten ausreicht. Außerdem wird das Kommunikationsprotokoll deutlich komplizierter, wenn ein Knoten mit mehr als einem Knoten kommuniziert.

Als Alternative zum synchronen *gcolx* steht also nur die asynchrone Kommunikation im Ring in eine Richtung zur Verfügung. Wegen der sehr langen Übertragungszeiten kann Variante 2 als mögliche Alternative ausgeschlossen werden. Obwohl in den bisherigen Berechnungen mit Variante 3 noch nie Probleme aufgetreten sind, wird auch Variante 1 weiterhin als Alternative untersucht. Bisher dienten alle Tests nur der Ermittlung der reinen Kommunikationszeit. Es wurden im Gegensatz zum praktischen Einsatz keine Berechnungen während des Vektoraustauschs durchgeführt. Wird der Handler aktiviert, so läuft er mit dem Hauptprogramm konkurrierend auf der CPU des Knotens. Deswegen ist mit kleineren Auswirkungen auf die Laufzeit der Berechnungen, die parallel zur Übertragung von Nachrichten durchgeführt werden, zu rechnen. Untersucht man das Verhalten der ersten und dritten Variante bei gleichzeitiger Berechnung und asynchroner Kommunikation, so erhält man zum Teil überraschende Ergebnisse. Die Tabelle 5.23 listet für verschiedene Anzahl von Knoten die Zeit, die nach der Berechnung im Mittel gewartet werden mußte (mittl. Warten) bei variierender Berechnungszeit (Berechnung). Außerdem wurde auf Knoten 0 immer noch die Summe der Berechnungszeit und der Wartezeit ermittelt. Reichte die Berechnungszeit nicht aus, um die komplette Kommunikation durchzuführen, so wird diese Summe jeweils in der dritten Spalte aufgelistet (Gesamt).

Anzahl	**Variante 1**			**Variante 3**		
Knoten	Berechnung	mittl. Warten	Gesamt	Berechnung	mittl. Warten	Gesamt
16	0.0450 s	0.1095 s	0.1545 s	0.0133 s	0.0775 s	0.0908 s
	0.0654 s	0.1156 s	0.1810 s	0.0179 s	0.0732 s	0.0911 s
	0.1003 s	0.1097 s	0.2100 s	0.0225 s	0.0677 s	0.0902 s
	0.1386 s	0.1136 s	0.2522 s	0.0343 s	0.0569 s	0.0911 s
32	0.0935 s	0.1719 s	0.2654 s	0.0264 s	0.0773 s	0.1037 s
	0.1323 s	0.1642 s	0.2965 s	0.0382 s	0.0656 s	0.1038 s
	0.1880 s	0.1645 s	0.3525 s	0.0579 s	0.0539 s	0.1038 s
	0.2772 s	0.1594 s	0.4365 s	0.0733 s	0.0282 s	0.1020 s
64	0.1846 s	0.2652 s	0.4497 s	0.0568 s	0.0672 s	0.1253 s
	0.2596 s	0.2408 s	0.5004 s	0.0837 s	0.0395 s	0.1225 s
	0.3815 s	0.2310 s	0.6124 s	0.1113 s	0.0117 s	0.1230 s
	0.5528 s	0.2276 s	0.7804 s	0.1585 s	0.0000 s	
	0.8255 s	0.2212 s	1.0467 s			
	1.2426 s	0.1867 s	1.4293 s			

Tabelle 5.23: Laufzeit der asynchronen Kommunikation (bei Berechnungen) I

Die Tabelle 5.24 listet die selben Daten für eine größere Anzahl von Knoten. Während bei der dritten Variante die Summe aus Berechnungszeit und Wartezeit immer sehr dicht an der gemessenen Übertragungszeit ohne Berechnungen liegt, scheint

die Wartezeit bei der ersten Variante kaum von der Berechnungszeit beeinflußt zu werden. Laufen Berechnungen auf der CPU, so scheint der Handler kaum Kommunikation mit dem synchronen *csend* Befehl durchführen zu können. Somit bleiben nur die beiden Alternativen, den Vektor mit dem synchronen *gcolx* Befehl zu übertragen oder die asynchrone Variante 3 zu benutzen.

Anzahl	Variante 1			Variante 3		
Knoten	Berechnung	mittl. Warten	Gesamt	Berechnung	mittl. Warten	Gesamt
80	0.2329 s	0.2917 s	0.5246 s	0.0809 s	0.0573 s	0.1381 s
	0.3256 s	0.2886 s	0.6142 s	0.1156 s	0.0232 s	0.1388 s
	0.4725 s	0.2846 s	0.7572 s	0.1419 s	0.0017 s	0.1435 s
	0.6919 s	0.2623 s	0.9542 s	0.1958 s	0.0000 s	
100	0.2809 s	0.3509 s	0.6317 s	0.0971 s	0.0574 s	0.1545 s
	0.4031 s	0.3442 s	0.7474 s	0.1435 s	0.0165 s	0.1600 s
	0.5883 s	0.3286 s	0.9169 s	0.1504 s	0.0155 s	0.1658 s
	0.8668 s	0.3120 s	1.1788 s	0.1834 s	0.0000 s	
128	0.3568 s	0.4026 s	0.7593 s	0.1298 s	0.0733 s	0.2031 s
	0.5175 s	0.3815 s	0.8990 s	0.1924 s	0.0131 s	0.2054 s
	0.7553 s	0.3726 s	1.1278 s	0.2316 s	0.0000 s	
	1.1083 s	0.3572 s	1.4656 s			

Tabelle 5.24: Laufzeit der asynchronen Kommunikation (bei Berechnungen) II

5.6.5 Überlagerung der Kommunikation mit Berechnungen

Die Untersuchungen des letzten Abschnitts haben gezeigt, daß die Berechnung der globalen inneren Produkte sehr schnell geht. Außerdem wurde gezeigt, daß für den Austausch der verteilt berechneten Vektoren zwei Alternativen zur Verfügung stehen. Wählt man die synchrone Variante (*gcolx*), so warten alle Knoten, bis der Vektor übertragen wurde. Die dafür benötigte Zeit ist kleiner als die Zeit, die zum Übertragen des Vektors mit der asynchronen Ringkommunikation in eine Richtung benötigt wird (Variante 3) (vgl. Tabelle 5.20 auf Seite 98). Der Vorteil der asynchronen Kommunikation ist aber, daß während der Übertragungszeit jeder Knoten weiter rechnen könnte. Es muß jetzt untersucht werden, ob es eine Anordnung der Rechenoperationen gibt, so daß die Knoten während der asynchronen Übertragung lange genug rechnen können. Dabei ist es nicht nötig, daß die Knoten die komplette Übertragungszeit rechnen können. Für die Gesamtlaufzeit ist wichtig, wann mit den Berechnungen begonnen werden kann, die den verteilt berechneten Vektor benutzen.

Untersucht man die Bedingung, unter der es sinnvoll ist, die asynchrone Variante einzusetzen, so findet man, daß (in Formeln ausgedrückt) gelten muß

$$\mathcal{T}(gcolx) + \mathcal{T}(Berechnung) \;\geq\; max\left\{\mathcal{T}(asynchron), \mathcal{T}(Berechnung)\right\} \;, \quad (5.9)$$

wobei

$$\begin{aligned} \mathcal{T}(gcolx) &= \text{Zeit zum synchronen Austausch} \\ \mathcal{T}(Berechnung) &= \text{Zeit für die unabhängigen Berechnungen} \\ \mathcal{T}(asynchron) &= \text{Zeit zum asynchronen Austausch .} \end{aligned}$$

Da jeder Knoten mindestens die Initialisierungen einiger Hilfsvektoren unabhängig von der Übertragung der Vektoren durchführen kann, sind alle Zeiten positiv. Somit kann mit Hilfe einer Fallunterscheidung die Bedingung in Formel (5.9) äquivalent umgeformt werden in

$$\mathcal{T}(gcolx) + \mathcal{T}(Berechnung) \;\geq\; \mathcal{T}(asynchron) \;. \quad (5.10)$$

Die Summe der Laufzeiten der synchronen Übertragung und der unabhängigen Berechnungen muß größer sein als die asynchrone Übertragungszeit, damit der Einsatz der asynchronen Variante sinnvoll ist. In Abbildung 5.14 sind die beiden möglichen Fälle schematisch dargestellt. Im ersten Fall ist es sinnvoll, die asynchrone Variante zu benutzen und im zweiten Fall nicht, da die unabhängigen Berechnungen nicht lange genug dauern.

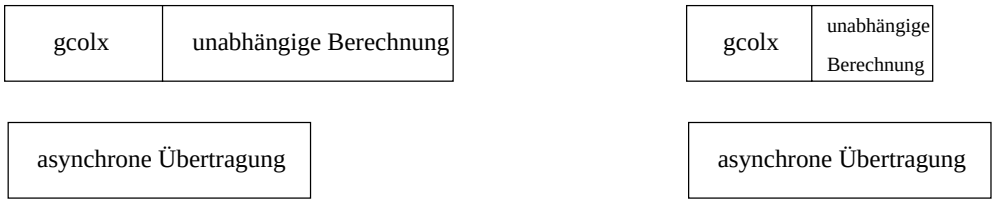


Abbildung 5.14: Einsatz asynchroner Variante

Untersucht man die Rechenoperationen des Lanczos Verfahrens, so findet man, daß (bei geschickter Anordnung der Operationen) die Berechnung der partiellen Lösungen x_j $(j = 1 \ldots r)$ und der partiellen inneren Produkte $\langle w, z_j \rangle$ unabhängig von den zu übertragenden Vektoren durchgeführt werden können. Der Einsatz der asynchronen Variante ist also sehr stark mit der Anzahl der berechneten Lösungen verbunden. Werden ausreichend viele Lösungen berechnet, so kann die komplette Kommunikation beim Austausch der verteilt berechneten Vektoren mit Berechnungen überlagert werden. Man teilt die Lösungen in zwei Hälften auf und überträgt die beiden Vektoren zwi und w während des Berechnens der Lösungen. Eine schematische Darstellung der Berechnung ist in Abbildung 5.15 dargestellt. Dabei wird der asynchronen Variante (schwarz) die synchrone Variante (blau) gegenübergestellt. Die Einteilung der Operationen sieht dann folgendermaßen aus:

Schritt 1: Jeder Knoten berechnet seine Komponenten der Lösungen x_j $(j = 1, \ldots, \lceil \frac{r}{2} \rceil)$. (zeitgleich (*asynchrone Kommunikation*) Austausch der Komponenten des Vektors w_{i+1}).

Schritt 2: Jeder Knoten testet, ob die asynchrone Kommunikation abgeschlossen ist und wartet gegebenenfalls auf die Beendigung.

Schritt 3: Jeder Knoten berechnet seine Komponenten von $zwi = Bw_{i+1}$.

Schritt 4: Jeder Knoten berechnet seine Komponenten der Lösungen x_j $(j = \lceil \frac{r}{2} \rceil + 1, \ldots, r)$ sowie der partiellen inneren Produkte $\langle w_{i+1}, z_j \rangle$ $(j = 1, \ldots, r)$. (zeitgleich (*asynchrone Kommunikation*) Austausch der Komponenten des Vektors zwi).

Schritt 5: Jeder Knoten testet, ob die asynchrone Kommunikation abgeschlossen ist und wartet gegebenenfalls auf die Beendigung.

Schritt 6: Jeder Knoten berechnet seine Komponenten von $v_{i+2} = B^T zwi$.

Schritt 7: Jeder Knoten berechnet die partiellen inneren Produkte $\langle w_{i+1}, z_j \rangle$ und $\langle v_{i+2}, v_{i+2} \rangle$.

Schritt 8: (*Kommunikation*) Ermittlung der globalen inneren Produkte.

Schritt 9: Jeder Knoten berechnet seine Komponenten von w_{i+2}.

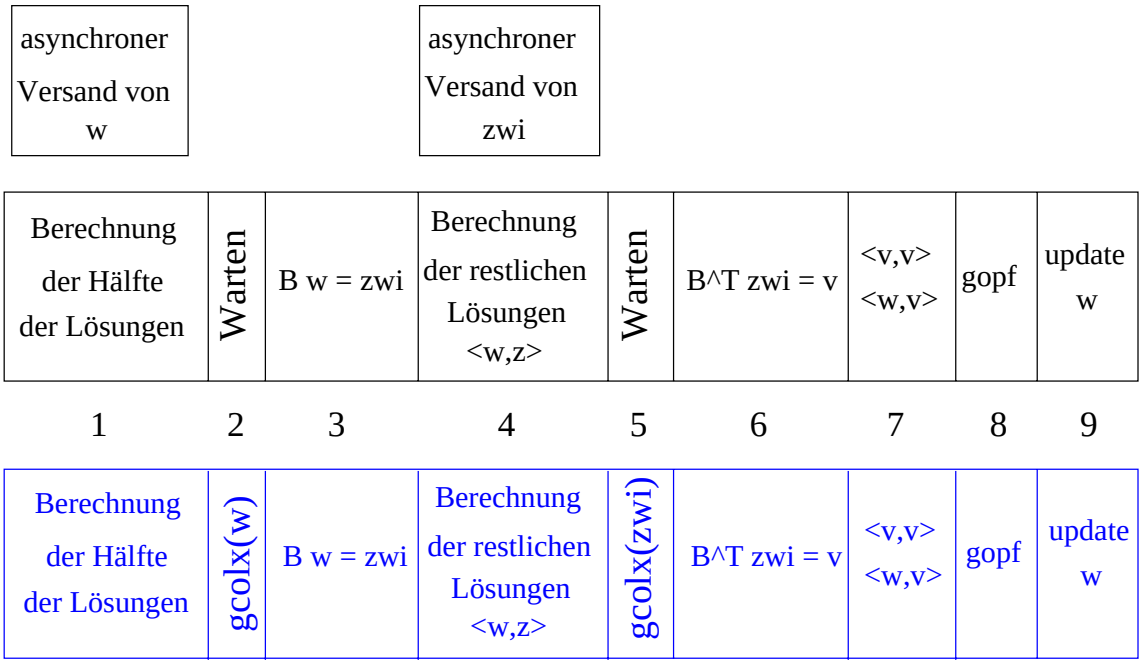


Abbildung 5.15: Schema paralleler Lanczos (asynchrone Kommunikation)

Anhand des folgenden Tests wurde das Kriterium zum Einsatz der asynchronen Kommunikation getestet. Gleichzeitig wurde das in Abschnitt 5.5 entwickelte Modell zur Laufzeitberechnung für die parallele Version getestet. Dabei wurden folgende Werte ermittelt: $cache1 = 1.1$, $cache2 = 1.00$ und $cache3 = 1.19$. In dem Test wurde die asynchrone Variante (Variante 3) und die Variante mit synchroner Kommunikation auf das DL85 Beispiel angewendet und die Zeiten für die einzelnen Berechnungs- und Kommunikationsschritte gemessen. In diesem Beispiel wurden insgesamt neun Lösungen berechnet. Bei den Zeiten in Klammern handelt es sich nicht um gemessene bzw. berechnete Daten, sondern um Kombinationen von gemessenen bzw. berechneten Daten. Außerdem enthält die Spalte mit den berechneten Werten für die Kommunikationsschritte keine Werte. Die Daten für die synchrone (sync) und die asynchrone (async) Variante enthält Tabelle 5.25.

Operation		gemessen	berechnet
update Lösungen 1 (sync)	**(Schritt 1)**	0.551 s	
update Lösungen 1 (async)	**(Schritt 1)**	0.600 s	
gcolx (w_i)	**(Schritt 2)**	0.104 s	(0.104 s)
Warten (w_i)	**(Schritt 2)**	0.000 s	(0.000 s)
Berechnung ($Bw = zwi$)	**(Schritt 3)**	1.141 s	1.141 s
update Lösungen 2 (sync)	**(Schritt 4)**	0.688 s	
update Lösungen 2 (async)	**(Schritt 4)**	0.740 s	
update Lösungen (sync)	**(Schritte 1+4)**	(1.239 s)	1.232 s
update Lösungen (async)	**(Schritte 1+4)**	(1.340 s)	1.232 s
gcolx (zwi)	**(Schritt 5)**	0.118 s	(0.118 s)
Warten (zwi)	**(Schritt 5)**	0.000 s	(0.000 s)
Berechnung $B^T zwi = v$	**(Schritt 6)**		1.141 s
Berechnung $\langle w_i, v_{i+1}\rangle$, $\langle v_{i+1}, v_{i+1}\rangle$	**(Schritt 7)**		0.263 s
kombinierte Berechnung	**(Schritte 6+7)**	1.379 s	(1.403 s)
gopf (sync)	**(Schritt 8)**	0.005 s	(0.005 s)
gopf (async)	**(Schritt 8)**	0.030 s	(0.030 s)
Berechnung w_{i+1}	**(Schritt 9)**	0.282 s	0.283 s
Gesamtiteration (sync)		4.268 s	4.287 s
Gesamtiteration (async)		4.180 s	4.090 s

Tabelle 5.25: Vergleich gemessene und berechnete Laufzeiten DL85

Die gemessenen Daten stimmen mit den berechneten Daten überein. Die gemessene Laufzeit für die Kombination von Schritt 6 und 7 liegt unter der Summe der berechneten Laufzeit für die beiden Einzelschritte. Dies liegt an dem gesparten Laden der Daten, da die einzelnen Komponenten direkt nach ihrer Berechnung benutzt werden. In den Laufzeiten der synchronen Kommunikationsbefehle (*gcolx* und *gopf*)

sind auch die Zeiten integriert, die zur Synchronisation der Knoten benötigt wird. (Nicht alle Knoten rufen den Befehl genau zur gleichen Zeit auf.) Dadurch erklärt sich auch die unterschiedliche Laufzeit der beiden *gcolx*-Aufrufe. Da während der Berechnung der Lösungen im asynchronen Fall der Handler konkurrierend aktiv ist, sind die gemessenen Laufzeiten von Schritt 1 und 4 bei der asynchronen Variante etwas größer. Dieser Effekt wird bei der berechneten Gesamtlaufzeit der asynchronen Variante nicht berücksichtigt. In diesem Beispiel ist die asynchrone Variante schneller.
Als letzten Aspekt betrachte ich das Laufzeitverhalten meiner parallelen Lanczos Implementierung bei wachsender Anzahl von beteiligten Knoten. Die Tabelle 5.26 enthält die Laufzeit einer Iteration des DL85 Beispiels. Außerdem wurde jeweils noch das Produkt aus der Anzahl der Knoten und der benötigten Laufzeit berechnet. Basierend auf diesem Wert für das Beispiel mit 32 Knoten wurde auch die Iterationszeit (Zielzeit) für alle anderen Knotenanzahlen ermittelt, die aus linearem Speedup resultiert hätte. Die Abbildung 5.16 verdeutlicht zusätzlich, daß fast linearer Speedup erreicht wird. Dabei muß man beachten, daß nur eine optimale Aufteilung der Zeilen bei weniger als 100 Knoten berechnet werden kann (vgl. Tabelle 5.18 (Seite 88)).

Knoten	Laufzeit	Koeffizient	Zielzeit
32	8.15 s	260.8	8.15 s
40	6.55 s	262.0	6.52 s
50	5.31 s	265.5	5.22 s
64	4.18 s	267.5	4.07 s
80	3.39 s	271.2	3.26 s
100	2.79 s	279.0	2.61 s
120	2.47 s	296.4	2.17 s

Tabelle 5.26: Laufzeiten DL85 bei variabler Anzahl Knoten

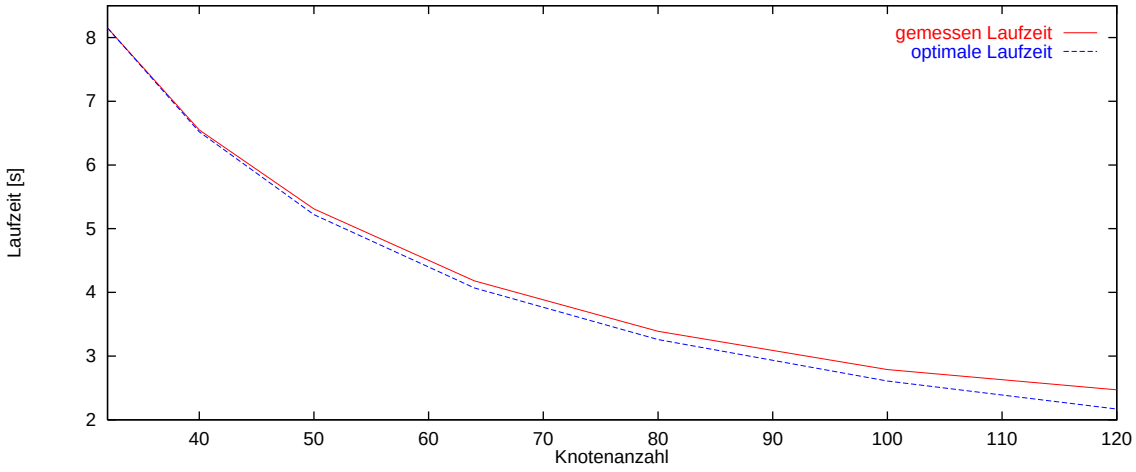


Abbildung 5.16: Vergleich erreichte und optimale Laufzeit

5.7 Technische Anmerkungen

In diesem Abschnitt werden einige zusätzliche technische Details der Implementierung besprochen.

- Bei Berechnungen mit dem Number Field Sieve befinden sich aus technischen Gründen (vgl. [40]) bei jeder Gleichung (in Abhängigkeit des Grads des verwendeten Polynoms) bis zu 5 Einträge (sog. *Degree*-Einträge), die aus langen Zahlen aufgebaut sind. Diese Einträge werden aus allen Gleichungen gelöscht und dafür werden entsprechend mehr Lösungen des restlichen Systems berechnet. Somit liegt die Anzahl der berechneten Lösungen bei NFS-Beispielen normalerweise über 7. Nach der Berechnung der Lösungen wird dann ein sehr kleines Gleichungssystem (z.B. 7×5 System) aufgebaut, das für jede Lösung die entsprechenden Werte der Degree-Einträge enthält. Dieses System kann sehr schnell mit normaler Gaußelimination gelöst werden. Die Lösungen dieses kleinen Systems stellen dann die Lösungen des ursprünglichen Gesamtsystems dar. Diese Vorgehensweise vereinfacht nicht nur die Implementierung, sondern hat auch Laufzeitvorteile. Die Berechnung einer zusätzlichen Lösung kann schneller durchgeführt werden als die Berücksichtigung einer dichtbesetzten Zeile mit langen Zahlen bei der Matrix-Vektormultiplikation. Außerdem wird die schnellere Lösungsberechnung (vgl. Abschnitt 5.4) erst durch die Elimination aller Degree-Einträge ermöglicht. Die genaue Laufzeitabschätzung (vgl. Formel 5.8 auf Seite 76) ermöglicht auch Aussagen darüber, ob die Elimination einer weiteren Spalte bei gleichzeitiger Berechnung einer zusätzlichen Lösung zu Laufzeitvorteilen führt. Stellt man die Formel (5.8)

$$\mathcal{T}(n,\omega,r) = n^2 \cdot t_2 + n \cdot (2 \cdot \omega + t_1)$$

um, wobei

$$\begin{aligned}
\omega &= cache1 \cdot (\omega_1 \cdot T(Add) + \omega_2 \cdot T(Sub)) + cache3 \cdot \omega_3 \cdot T(kmult) \\
t_1 &= cache2 \cdot (T(Inv) + (2+r) \cdot T(Mult)) \\
t_2 &= cache2 \cdot (T(Square) + 2 \cdot T(Sub_m) + (2+r) \cdot T(Add_m) \\
&\quad + (3+r) \cdot T(Mult)) + 2 \cdot cache3 \cdot T(Red) \ ,
\end{aligned}$$

so kann man ermitteln, wie stark das Gewicht (ω) beim Löschen einer Spalte mindestens abnehmen muß, damit die Gesamtlaufzeit beim Lösen des Gleichungssystems abnimmt. Man ermittelt also Δ (> 0), so daß

$$\mathcal{T}(n,\omega,r) \ - \ \mathcal{T}(n-1,\omega-\Delta,r+1) \ \geq \ 0 \ .$$

Durch elementare Umformungen erhält man

$$\mathcal{T}(n,\omega,r) \ - \ \mathcal{T}(n-1,\omega+\Delta,r) \ \geq \ 0$$

$$\iff \ \Delta \ \geq \ \frac{(n-1)^2 \cdot t_3 - (2 \cdot n - 1) \cdot t_2 - 2 \cdot \omega - t_4}{2 \cdot (n-1)} \ ,$$

wobei

$$t_3 = cache2 \cdot (T(Mult) + T(Add\_m))$$
$$t_4 = cache2 \cdot T(Inv) + (3 + r - n) \cdot cache2 \cdot T(Mult) \ .$$

In allen Beispielen lag dieser Wert jedoch weit oberhalb des Gewichtes der schwersten Spalte, so daß das Löschen einer weiteren Spalte bei gleichzeitiger Berechnung einer zusätzlichen Lösung zu keinen Laufzeitvorteilen geführt hätte.

- In beiden Implementierungen (sequentiell und parallel) existieren jeweils 2 Programme zur Berechnung der Lösungen, die nacheinander aufgerufen werden. Mit der *start*-Version der Lanczos Implementierung wird das Gleichungssystem aus einer Datei eingelesen und die benötigten Datenstrukturen werden aufgebaut. Außerdem wird die erste Iteration berechnet und ein Checkpoint erstellt. Nach Beendigung des ersten Programms sind charakteristische Daten wie z.B. die Anzahl der Zeilen, Spalten und Einträge in der Matrix B, der Speicherplatzbedarf einer Komponente etc. bekannt. Diese Informationen werden vom Compiler benutzt, um effizienten Code für die restlichen Iterationen zu generieren. Da die Anzahl der Schleifendurchläufe bekannt ist, kann Loopunrolling (vgl. [12]) eingesetzt werden. Die Datenstruktur zur Speicherung der Einträge von B kann statisch allokiert werden, da ihr Speicherplatzbedarf nach dem ersten Programmlauf bekannt ist. Im Gegensatz zur dynamischen Allokierung kennt der Compiler die Adresse der Speicherung, was zur Vereinfachung der Adreßrechnung beim Zugriff auf diese Elemente benutzt wird. Das erste Programm (*start*-Version) erzeugt ein File mit *#define* Anweisungen, das in den Sourcecode des zweiten Programms includet wird.

- In beiden Versionen ist ein Aufsetzmechanismus eingebaut. In der sequentiellen Version kann eingestellt werden, nach wievielen Iterationen ein Checkpoint geschrieben wird. Wird die Berechnung unterbrochen, so kann dann auf dem letzten Checkpoint aufgesetzt werden. Die parallele Version wird mit der Anzahl der Iterationen gestartet, die bei diesem Lauf berechnet werden sollen. Außerdem ist ein Signal Handler eingebaut, der, sobald das Programm ein bestimmtes Signal erhält, einen Checkpoint schreibt und das Programm beendet. Dieses Signal wird im Batchbetrieb der parallelen Maschine automatisch fünf Minuten vor Ablauf der Rechenzeit des Batchjobs an das Programm geschickt und ermöglicht somit das Erstellen eines Aufsetzpunkts.

- Bei der Berechnung der Matriz-Vektormultiplikation findet eine Umwandlung eines rest-Eintrags mit Betrag 2 in zwei entsprechende Einträge mit Betrag 1 statt, falls dadurch Laufzeitvorteile erzielt werden können. (Ist $T(kmult) > 2 \cdot T(Add)$ (bzw. $T(Sub)$) führt die Umwandlung zu einer Reduzierung der Laufzeit.) Der dadurch erzielte Effekt ist jedoch relativ unbedeutend.

Kapitel 6

Der Kompaktifizierer

In diesem Kapitel werden die in Abschnitt 5.5 durchgeführte genaue Laufzeitanalyse des Lanczos Verfahrens und die Ideen der strukturierten Gaußelimination (vgl. [33]) dazu benutzt, die Laufzeit zum Lösen eines Gleichungssystems mit dem Lanczos Verfahren zu minimieren. Dazu wird das gegebene Gleichungssystem in ein anderes Gleichungssystem mit kleinerer Dimension jedoch eventuell mehr Einträgen transformiert. Dieses System kann bis zu 80% schneller gelöst werden, und aus seiner Lösung läßt sich sehr schnell eine Lösung des ursprünglichen Systems generieren. Ein weiterer Aspekt ist die Reduzierung des Hauptspeicherbedarfs der Vektoren, der im wesentlichen von der Dimension des Gleichungssystems bestimmt wird. Hilfreich bei der Umsetzung dieser Ideen war ein Forschungsaufenthalt (1993) bei Prof. A.K. Lenstra (Bellcore) in den USA, mit dem zusammen die strukturierte Gaußelimination für endliche Primkörper (vgl. [33], [32]) implementiert und untersucht wurde. Nach einer Vorstellung der prinzipiellen Idee der strukturierten Gaußelimination stelle ich das daraus resultierende Vorgehen für das Lanczos Verfahren an Hand der sequentiellen Version dar. Die Vorgehensweise des Kompaktifizierers für die parallele Lanczos Version unterscheidet sich prinzipiell nicht von der sequentiellen Version. Da in der LIP jedoch spezielle Funktionen implementiert wurden, ist im parallelen Fall die Ermittlung des Gewichtes einer Zeile komplizierter. Auf die sich daraus ergebenden Unterschiede wird bei den einzelnen Schritten noch genauer eingegangen. Nach der Darstellung der Idee gehe ich auf einige Details der Implementierung ein und gebe einige Beispiele für die mit meiner Implementierung erzielten Resultate. Im folgenden werden die Unbekannten eines Systems mit den Spalten und die Gleichungen mit den Zeilen gleichgesetzt.

6.1 Idee

Bei der strukturierten Gaußelimination wird das Lösen des Gleichungssystems in zwei Phasen eingeteilt. In der zweiten Phase wird normale Gaußelimination einge-

setzt, deren Laufzeit nur von der Dimension des zu lösenden Systems abhängt. Die erste Phase versucht, die Laufzeit der zweiten Phase zu minimieren. Dabei geht man prinzipiell so vor, daß man die Elimination der Spalten bei den dünnbesetzten Spalten beginnt. Da diese Spalten nur wenige Einträge haben, ist sehr wenig Arbeit zur Elimination dieser Spalten nötig. Die Spalten des Systems werden dabei in einen *light*-Part (dünnbesetzte Spalten) und in einen *heavy*-Part (dichtbesetzte Spalten) eingeteilt. Dabei startet man mit sehr wenigen Spalten im heavy Part und erlaubt den Wechsel von Spalten aus den light in den heavy Part. Die erste Phase eliminiert alle Spalten im light Part, wobei nur Operationen erlaubt sind, die keine neuen Einträge im light Part erzeugen. Die durchgeführten Operationen werden nur auf dem light Part ausgeführt und mitprotokolliert. Dadurch ist die erste Phase sehr schnell. Sind alle Spalten des light Parts eliminiert, so endet die erste Phase. Dann wird der heavy Part anhand der mitprotokollierten Operationen aufgebaut und das daraus resultierende Gleichungssystem gelöst. Anhand der mitprotokollierten Operationen läßt sich aus der Lösung des heavy Parts sehr schnell eine Lösung des ursprünglichen Gleichungssystems berechnen. Der Effekt der ersten Phase ist eine Reduzierung der Dimension auf ca. ein Drittel und eine starke Zunahme der Einträge, da der heavy Part am Ende der ersten Phase fast dichtbesetzt ist. Bei der strukturierten Gaußelimination hängt die Laufzeit der zweiten Phase nur von der Dimension ab, so daß das Ziel der ersten Phase allein in der Reduzierung der Dimension liegt. Die Laufzeit des Lanczos Verfahrens hängt zusätzlich zu der Dimension auch noch von der Anzahl der Einträge ab (vgl. Abschnitt 2.3). Deshalb muß bei der Elimination einer Spalte auch die Zunahme der Einträge im gesamten System beachtet werden. Die genaue Laufzeitabschätzung (vgl. Formel 5.8 auf Seite 76) erlaubt Aussagen über die Veränderung der Laufzeit bei Veränderung der Dimension und der Anzahl der Einträge in dem zu lösenden Gleichungssystem. Stellt man die Formel (5.8)

$$\mathcal{T}(n,\omega,r) = n^2 \cdot t_2 + n \cdot (2 \cdot \omega + t_1)$$

um, wobei

$$\begin{aligned}
\omega &= cache1 \cdot (\omega_1 \cdot T(Add) + \omega_2 \cdot T(Sub)) + cache3 \cdot \omega_3 \cdot T(kmult) \\
t_1 &= cache2 \cdot (T(Inv) + (2+r) \cdot T(Mult)) \\
t_2 &= cache2 \cdot (T(Square) + 2 \cdot T(Sub_m) + (2+r) \cdot T(Add_m) \\
&\quad + (3+r) \cdot T(Mult)) + 2 \cdot cache3 \cdot T(Red),
\end{aligned}$$

so kann man ermitteln, wie stark das Gewicht (ω) bei der Elimination einer Spalte höchstens zunehmen darf, damit die Gesamtlaufzeit beim Lösen des Gleichungssystems nicht zunimmt. Man ermittelt also Δ (> 0), so daß

$$\mathcal{T}(n,\omega,r) \; - \; \mathcal{T}(n-1,\omega+\Delta,r) \; \geq \; 0 \; .$$

Durch elementare Umformungen erhält man

$$\mathcal{T}(n,\omega,r) \; - \; \mathcal{T}(n-1,\omega+\Delta,r) \; \geq \; 0$$

$$\iff \; n^2 \cdot t_2 + n \cdot (2 \cdot \omega + t_1) - ((n-1)^2 \cdot t_2 + (n-1) \cdot (2 \cdot (\omega + \Delta) + t_1)) \; \geq \; 0$$

$$\iff \qquad \Delta \leq \frac{(2 \cdot n - 1) \cdot t_2 + 2 \cdot \omega + t_1}{2 \cdot (n-1)} .$$

In praktischen Beispielen erhält man so für Δ Werte zwischen 100 und 1200. Bedenkt man die Struktur der Systeme (vgl. Kapitel 3), so kann man auch beim Lanczos Verfahren auf eine Reduzierung der Laufzeit hoffen, wenn man die dünnbesetzten Spalten in einem Preprocessingschritt eliminiert.

Damit man die Elimination einer Spalte nicht mehr rückgängig machen muß, falls das Gewicht dabei zu stark zugenommen hat, benötigt man eine relativ genaue Vorhersage, wie stark das Gewicht des Systems bei einer Spaltenelimination zunehmen kann. Dazu betrachte ich ein Beispiel. Die Abbildung 6.1 enthält die drei Zeilen (i, j und k) eines Beispiels, die als einzige Zeilen in der letzten Spalte einen von Null verschiedenen Eintrag haben. Zur Elimination dieser Spalte wird ein geeignetes

i	-1	1	0	3	0	0	1	0	0	0	0	1	0	1
j	2	1	1	5	0	-3	0	1	1	0	-1	1	0	2
k	1	2	-1	0	-2	0	1	0	1	-2	0	0	1	-1

Abbildung 6.1: Beispiel vor Elimination der letzten Spalte

Vielfaches von Zeile i zu Zeile j und k addiert. Dadurch verschwinden deren Einträge in der letzten Spalte. Da Zeile i nun als einzige Zeile Einträge in dieser Spalte hat, kann diese Zeile und damit auch die letzte Spalte aus dem zu lösenden System gelöscht werden, wodurch sich das Gewicht des Systems um das Gewicht von Zeile i verringert. Die Situation nach der Elimination der letzten Spalte ist in Abbildung 6.2 dargestellt. Als weitere Beobachtung stellt man fest, daß die Einträge der ur-

j-2i	4	-1	1	-1	0	-3	-2	1	1	0	-1	-1	0	0
k+i	0	3	-1	3	-2	0	2	0	1	-2	0	1	1	0

Abbildung 6.2: Beispiel nach Elimination der letzten Spalte

sprünglichen Zeilen j und k nur an den Stellen geändert werden, an denen auch Zeile i Einträge hat. Somit spielt die Anzahl der Einträge der Zeile, mit der eine Spalte eliminiert werden soll, eine wichtige Rolle. Im folgenden wird diese Zeile auch als *eliminierende Zeile* bezeichnet. Wie stark das Gewicht einer Zeile zunimmt, bei der eine Spalte eliminiert wird, hängt aber nicht nur von der Anzahl der Einträge der eliminierenden Zeile ab, sondern auch sehr stark von der Art der Einträge ab, die die Zeilen in den Spalten haben, in denen auch die eliminierende Zeile Einträge hat.

Im folgenden untersuche ich die Veränderung des Gewichts einer Zeile, zu der eine Zeile addiert wird, in Abhängigkeit von der Art der beteiligten Einträge. Ausgehend von den Einträgen der addierten Zeile wird untersucht, welche Auswirkungen die möglichen Einträge der anderen Zeile haben.

- **Eintrag der addierten Zeile ist 1.**
  - Eintrag der Ausgangszeile ist 1.

    Der ursprüngliche 1-Eintrag wird zu einem neuen *rest*-Eintrag (2). (Man beachte dabei, daß *rest*-Einträge (2) in zwei entsprechende 1-Einträge geändert werden, falls die kurze Multiplikation mehr als doppelt so lange dauert wie eine Addition.) Somit gilt für die Gewichtsänderung δ:

    $$\delta = T(kmult) - T(Add) \leq T(Add) .$$

  - Eintrag der Ausgangszeile ist -1.

    Der -1-Eintrag wird durch den 1-Eintrag eliminiert und das Gewicht der Zeile nimmt ab. Für die Gewichtsänderung δ gilt: $\delta = -T(Sub) < 0$.
  - Eintrag der Ausgangszeile ist ein *rest*-Eintrag.

    Handelt es sich bei dem Eintrag um eine -2, so wird der neue Eintrag ein -1-Eintrag und für die Gewichtsänderung δ gilt dann:

    $$\delta = T(Sub) - T(kmult) < 0 .$$

    In allen anderen Fällen ändert sich nur der Wert, aber nicht die Art des Eintrags und das Gewicht der Zeile ändert sich nicht. Es gilt dann:

    $$\delta = 0 .$$

  - Ausgangszeile hatte keinen Eintrag (bzw. Eintrag ist 0).

    Die Zeile erhält einen zusätzlichen 1-Eintrag. Somit erhöht sich das Gewicht um $\delta = T(Add)$.

  In allen betrachteten Fällen gilt für die Gewichtsänderung

  $$\delta \leq T(Add) .$$

- **Eintrag der addierten Zeile ist -1.**
  - Eintrag der Ausgangszeile ist 1.

    Der 1-Eintrag wird durch den -1-Eintrag eliminiert und das Gewicht der Zeile nimmt ab. Für die Gewichtsänderung δ gilt: $\delta = -T(Add)$.
  - Eintrag der Ausgangszeile ist -1.

    Der ursprüngliche -1-Eintrag wird zu einem neuen *rest*-Eintrag (-2). (Man beachte dabei, daß *rest*-Einträge (-2) in zwei entsprechende -1-Einträge geändert werden, falls die kurze Multiplikation mehr als doppelt so lange dauert wie eine Subtraktion.) Somit gilt für die Gewichtsänderung δ:

    $$\delta = T(kmult) - T(Sub) \leq T(Sub) .$$

– Eintrag der Ausgangszeile ist ein *rest*-Eintrag.

Handelt es sich bei dem Eintrag um eine 2, so wird der neue Eintrag ein 1-Eintrag und für die Gewichtsänderung δ gilt dann:

$$\delta = T(Add) - T(kmult) \ < \ 0 \ .$$

In allen anderen Fällen ändert sich nur der Wert aber nicht die Art des Eintrags und das Gewicht der Zeile ändert sich nicht. Es gilt dann:

$$\delta = \ 0 \ .$$

– Ausgangszeile hatte keinen Eintrag (bzw. Eintrag ist 0).

Die Zeile erhält einen zusätzlichen -1-Eintrag. Somit erhöht sich das Gewicht um $\delta = T(Sub)$.

In allen betrachteten Fällen gilt für die Gewichtsänderung

$$\delta \leq T(Sub) \ .$$

- **Eintrag der addierten Zeile ist *rest*-Eintrag.**

– Eintrag der Ausgangszeile ist 1.

Handelt es sich bei dem *rest*-Eintrag um eine -2, so wird der neue Eintrag ein -1-Eintrag und für die Gewichtsänderung δ gilt dann:

$$\delta = T(Sub) - T(Add) \ .$$

In allen anderen Fällen wird aus dem 1-Eintrag ein *rest*-Eintrag. Es gilt dann:

$$\delta = \ T(kmult) - T(Add) \ .$$

– Eintrag der Ausgangszeile ist -1.

Handelt es sich bei dem *rest*-Eintrag um eine 2, so wird der neue Eintrag ein 1-Eintrag und für die Gewichtsänderung δ gilt dann:

$$\delta = T(Add) - T(Sub) \ .$$

In allen anderen Fällen wird aus dem -1-Eintrag ein *rest*-Eintrag. Es gilt dann:

$$\delta = \ T(kmult) - T(Sub) \ .$$

– Eintrag der Ausgangszeile ist ein *rest*-Eintrag.

Sind die Beträge beider Einträge gleich, aber die Vorzeichen verschieden, so verschwindet der *rest*-Eintrag. Dann gilt für die Gewichtsänderung

$$\delta = \ -T(kmult) \ .$$

Unterscheiden sich die Beträge um eins und die Vorzeichen sind verschieden, so wird aus dem *rest*-Eintrag ein 1 bzw. −1-Eintrag. Für die Gewichtsänderung gilt dann:

$$\delta < 0 ,$$

da entweder $\delta = T(Add) - T(kmult)$ oder $\delta = T(Sub) - T(kmult)$ gilt, in Abhängigkeit vom Vorzeichen des betragsmäßig größten Eintrags.
In allen anderen Fällen ändert sich nur der Wert, aber nicht die Art des Eintrags und das Gewicht der Zeile ändert sich nicht. Es gilt dann:

$$\delta = 0 .$$

– Ausgangszeile hatte keinen Eintrag (bzw. Eintrag ist 0).
Die Zeile erhält einen zusätzlichen *rest*-Eintrag. Somit erhöht sich das Gewicht um

$$\delta = T(kmult) .$$

In allen betrachteten Fällen gilt für die Gewichtsänderung

$$\delta \leq T(kmult) .$$

Die Tabelle 6.1 enthält eine Zusammenfassung der Ergebnisse.

	Einträge der addierten Zeile		
	1	**-1**	**long**
1	$\delta \leq T(Add)$	$\delta = -T(Add)$	$\delta \leq T(Add)$
-1	$\delta = -T(Sub)$	$\delta \leq T(Sub)$	$\delta \leq T(Sub)$
long	$\delta \leq 0$	$\delta \leq 0$	$\delta \leq 0$
0	$\delta = T(Add)$	$\delta = T(Sub)$	$\delta = T(kmult)$

Tabelle 6.1: Änderung des Gewichts in Abhängigkeit der Einträge

Die Untersuchungen haben gezeigt, daß eine genaue Vorhersage der Gewichtszunahme nur möglich ist, wenn bekannt ist, ob die zwei beteiligten Zeilen gemeinsam in Spalten Einträge haben und um welche Einträge es sich dabei handelt. Haben die beiden Zeilen nur in verschiedene Spalten Einträge, so nimmt das Gewicht der Zeile in der die Spalte eliminiert wird (und somit das Gewicht des Systems) um das Gewicht der eliminierenden Zeile zu (vgl. letzte Spalte von Tabelle 6.1). Dies stellt gleichzeitig eine obere Schranke für die Gewichtszunahme dar, da gemeinsame Einträge in Spalten höchstens zu einer gleichgroßen Gewichtszunahme führen. Für die modifizierte Gewichtsfunktion der parallelen Lanczos Implementierung (vgl. Abschnitt 5.6.3) und die damit verbundene genauere Unterscheidung der *rest*-Einträge führen

die selben Untersuchungen zu prinzipiell den gleichen Ergebnissen. Im Algorithmus wird das Gewicht der eliminierenden Zeile benutzt, um die Gewichtszunahme bei der Elimination einer Spalte abzuschätzen. Dabei wird berücksichtigt, daß bei der Elimination einer Spalte jede beteiligte Zeile mindestens einen 1-Eintrag mit der eliminierenden Zeile gemeinsam hat und daß die eliminierende Zeile nach der Elimination aus dem zu lösenden System entfernt wird. Definiert man das Gewicht einer Zeile j (wobei die Bezeichnungen von Abschnitt 5.5 benutzt werden (vgl. Bemerkung 5.2 (Seite 73))) als

$$gew[j] := cache1 \cdot \left(\omega_1 + \omega_2 \cdot \frac{T(Sub)}{T(Add)} \right) + cache3 \cdot \omega_3 \cdot \frac{T(kmult)}{T(Add)} ,$$

so wird als obere Schranke (*bound*) für die Gewichtszunahme bei der Elimination einer Spalte mit n Einträgen durch Zeile i

$$bound := (gew[i] - cache1) \cdot (n - 2)$$

benutzt. Diese Abschätzung dient lediglich dazu, mögliche Kandidaten zur Elimination einer Spalte zu finden. In dieser Abschätzung ist weder die Anzahl der sonstigen gemeinsamen Einträge noch die Tatsache berücksichtigt, daß bei der Elimination einer Spalte nicht immer nur die eliminierende Zeile einfach addiert werden muß. Die spezielle Struktur der Systeme:

- 95% der Einträge haben Betrag 1,
- es gibt ungefähr gleich viele 1- und -1-Einträge und
- Einträge mit Betrag größer als 1 treten fast nie in dünnbesetzten Spalten auf,

sorgt jedoch dafür, daß in der Praxis mit dieser Abschätzung eine schnelle Vorauswahl durchgeführt werden kann. Eine genauere Abschätzung, die auch die Einträge in der zu eliminierenden Spalte berücksichtigt, wird dann benutzt, um zu entscheiden, ob die Spaltenelimination durchgeführt wird. Man eliminiert jetzt solange Spalten, bis keine Reduzierung der Laufzeit des zu lösenden Systems mehr möglich ist. Könnten nur noch Eliminationen durchgeführt werden, die zu zu großer Zunahme des Gewichts führen würden, so wird das noch zu lösende System mit dem Lanczos Verfahren gelöst. Die berechnete Lösung kann dann aufgrund der mitprotokollierten Operationen in eine Lösung des ursprünglich zu lösenden Systems umgewandelt werden.

6.2 Implementierung

Nachdem im vorherigen Abschnitt die prinzipielle Idee und Vorgehensweise des Kompaktifizierers beschrieben wurde, gehe ich nun auf einige technische Details und Probleme bei der Implementierung ein.

- **Überbestimmtheit**
  In der Praxis ist es von Vorteil, bei der Lösung der dünnbesetzten Gleichungssysteme mehr Gleichungen als Unbekannte zu haben. Dadurch gewinnt man einige Wahlmöglichkeiten und kann diejenigen Gleichungen auswählen, die am schnellsten zu einer Lösung des Systems führen. Da man alle Operationen auf den kompletten Zeilen wirklich durchführen und nicht nur wie bei der strukturierten Gaußelimination mitprotokollieren muß, bekommt man bei der Umsetzung der hier vorgestellten Idee sehr schnell Speicherplatzprobleme. Die Implementierung erlaubt deshalb die Anzahl der überschüssigen Gleichungen zu beschränken. Existieren mehr Gleichungen als gewünscht, so werden die schwersten Gleichungen direkt gelöscht. Außerdem werden während der Elimination diejenigen Zeilen gelöscht, deren Gewicht so stark angewachsen ist, daß ihre Aufnahme in das eigentlich zu lösende System sehr unwahrscheinlich erscheint. Für die Entscheidung, ob eine Spalte eliminiert werden kann, sind jedoch nur die Auswirkungen auf das zu lösende Gleichungssystem wichtig und nicht auf alle Gleichungen des Systems. Damit das Gewicht der Spalten im eigentlichen System (das nach dem Kompaktifizierungsschritt gelöst wird) bestimmt werden kann, muß der Kompaktifizierungsalgorithmus zu jedem Zeitpunkt in der Lage sein, die Auswahl der Gleichungen zu treffen, die dieses System aktuell aufbauen. Es werden jeweils die Gleichungen mit dem kleinsten Gewicht ausgewählt, so daß ein System entsteht, das etwa 10 Gleichungen mehr enthält als Unbekannte. Dazu wird eine Datenstruktur aufgebaut, die jedem Gewicht die Anzahl der Zeilen mit diesem Gewicht zuordnet. Danach kann man bestimmen, bis zu welchem Gewicht eine Zeile in das zu lösende System aufgenommen wird. Das Gewicht einer Zeile reicht dann um zu entscheiden, ob die Zeile am Aufbau des zu lösenden Systems beteiligt ist. Diese Datenstruktur wird während der Elimination von Spalten fortlaufend aktualisiert.

- **Aufbau der Spalteninformation**
  Bei der Elimination einer Spalte müssen die Einträge von allen Zeilen in dieser Spalte eliminiert werden. Deshalb benötigt der Algorithmus das Gewicht dieser Spalte im Gesamtsystem sowie alle Zeilen mit Einträgen in dieser Spalte. Wegen des sehr großen Speicherplatzbedarfs wird diese Information nicht für alle Spalten berechnet. (Dies würde einer Transponierung des gesamten Gleichungssystems entsprechen.) Lediglich für die dünnbesetzten Spalten, die als Kandidaten zum Eliminieren in Frage kommen können, wird ermittelt, welche Zeilen von Null verschiedene Einträge in dieser Spalte haben. Zur Ermittlung des Vielfachen, das bei der Elimination dieser Spalte zu jeder Zeile addiert werden muß, wird auch jeweils der Wert des Eintrags gespeichert. Der Funktion zum Aufbau dieser Spalteninformation wird als Parameter eine obere Schranke mitgegeben. Für alle Spalten, deren Gewicht im zu lösenden System kleiner als die mitgegebene Schranke ist, wird die Spalteninformation aufgebaut. Obwohl während der gesamten Berechnung die Spalteninformation stets aktualisiert wird, ist es nötig, von Zeit zu Zeit die Auswahl der Spalten, für die diese Information aufgebaut wird zu überprüfen. Bei der Elimination von Spalten kann

es zu starken Schwankungen des Gewichts anderer Spalten kommen. Deshalb kann eine Zahl von Spalteneliminationen angegeben werden, nach denen die Spalteninformation komplett neu aufgebaut wird. Der Algorithmus startet mit einem Wert von $\Delta \cdot 0.05$ als Schranke für das Gewicht der Spalten, für die die Information berechnet wird. Können damit keine Spalteneliminationen mehr durchgeführt werden, weil die benötigten Informationen fehlen, so wird der Wert der Schranke auf $\Delta \cdot 0.1$ verdoppelt. Man beachte dabei, daß der Wert von Δ während der Berechnung nicht konstant ist, sondern leicht anwächst.

- **Bestimmung eliminierender Zeilen**
  Um diejenigen Zeilen zu finden, die Spalten eliminieren können, bestimmt man für jede Zeile, das Minimum der Spaltengewichte, in denen sie von Null verschiedene Einträge hat. Dabei wird wiederum für jede Spalte nur das Gewicht benutzt, das sich durch diejenigen Zeilen ergibt, die aktuell am Aufbau des zu lösenden Gleichungssystems beteiligt sind. Nachdem durch die obige Abschätzung dann eine Spalte ermittelt wurde, die eliminiert werden kann, bestimmt der Algorithmus diejenige Zeile, mit dem kleinsten Gewicht und einem Eintrag mit Betrag 1 in dieser Spalte. Diese Zeile wird als eliminierende Zeile gewählt. Durch die Veränderung des Systems bei Spalteneliminationen verändern sich auch die Informationen, die zur Auswahl der Spalten führen, die eliminiert werden können (vgl. Abschnitt **Aufbau der Spalteninformationen**). Aus diesem Grund wird nach einem Neuaufbau der Spalteninformationen auch für jede Zeile die Information über ihre dünnbesetzteste Spalte aktualisiert. In der Praxis hat sich gezeigt, das der Neuaufbau der Spalteninformation nach der Elimination von 1000 Spalten zu guten Ergebnissen bei vertretbarem Aufwand führt.

- **Genauere Abschätzung der Gewichtszunahme**
  Bevor endgültig entschieden wird, ob die Elimination einer Spalte durchgeführt wird, ermittelt der Algorithmus aufgrund der Spalteninformationen eine genauere Abschätzung der Gewichtszunahme. Dabei wird berücksichtigt, daß eine Zeile des eigentlich zu lösenden Systems, deren Gewicht durch Elimination einer Spalte zu stark anwächst, durch eine leichtere Zeile ersetzt werden kann, die vorher nicht zum Aufbau des zu lösenden Systems benutzt wurde. Außerdem wird für jede Zeile, die aktuell im zu lösenden System ist, an Hand der Art der Einträge in der zu eliminierenden Spalte eine genauere Abschätzung für ihre Gewichtszunahme ermittelt. Die Anzahl und Art der sonstigen gemeinsamen Einträge wird jedoch auch in dieser Abschätzung nicht berücksichtigt. Eine genaue Analyse der Art und der Anzahl der sonstigen gemeinsamen Einträge würde zu lange dauern. Statt dessen ermöglicht die Implementierung eine Steuerung über einen Multiplikator, mit dem die Abschätzung multipliziert wird, bevor der Wert mit Δ verglichen wird. Ein Wert des Multiplikators kleiner als 1 führt eventuell zu einer größeren Gewichtszunahme als Δ, ermöglicht jedoch die Berücksichtigung potentieller gemeinsamer Einträge. Auf der anderen Seite soll durch die Elimination einer Spalte eine Reduktion der Laufzeit erzielt werden, deshalb kann auf diejenigen Eliminationen verzichtet werden, die nur durch Gewichtszunahmen in der Nähe von Δ erzielt würden.

- **Zeilenaddition in dünnbesetzter Darstellung**
  Hat man eine Zeile gefunden, die eine Spalte auf alle Fälle ohne Überschreitung der zulässigen Gewichtszunahme reduziert, so addiert man die geeigneten Vielfachen dieser Zeile zu allen anderen Zeilen, die Einträge in dieser Spalte haben. Aus Speicherplatzgründen wird zur Abspeicherung einer Zeile das Format zur Abspeicherung dünnbesetzter Matrizen (vgl. Abbildung 5.3 auf Seite 64) benutzt, d.h. pro Zeile werden die Einträge nach ihrer Art getrennt. Durch diese spezielle Struktur wird die Addition zweier Spalten schwieriger durchführbar als bei dichtbesetzter Darstellung. Man könnte vor jeder Addition die Zeilen in ein dichtbesetztes Format überführen und nachher das Ergebnis wieder in das dünnbesetzte Format zurückwandeln. Dieses Vorgehen hätte den Vorteil der einfachen Implementierung der Addition, ist jedoch sehr zeitaufwendig. Im folgenden beschreibe ich eine schnellere Variante zur Addition des *multi*-fachen einer Zeile i zu einer Zeile j. Dabei sind beide Zeilen im dünnbesetzten Format abgespeichert. Mit der Beschreibung des Vorgehens wird gleichzeitig ein Beispiel mitgerechnet. Die Ausgangssituation ist in Abbildung 6.3 dargestellt. Die beiden Zeilen i und j stammen aus dem vorherigen Beispiel (vgl. Abbildung 6.1), wobei in der Abbildung 6.1 die Spalten, in denen keine der drei Zeilen einen Eintrag hat, weggelassen wurden. Die Spaltenindizes der dargestellten Spalten im Gesamtsystem sind: (der Reihenfolge nach aufgelistet) 0, 3, 11, 23, 51, 117, 293, 501, 1391, 5182, 9321, 18009, 23701 und 26103. Die Datenstruktur, die das Ergebnis (die neue Zeile j) speichern soll, enthält noch keine Einträge. In dem Beispiel ergibt sich die neue Zeile j als Summe von Zeile

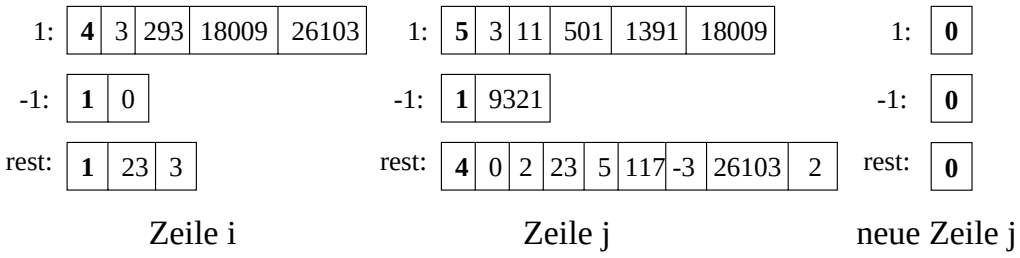


Abbildung 6.3: Ausgangssituation des Beispiels

j und dem (-2)-fachen von Zeile i. Als Vorberechnung bei der Elimination einer Spalte wird berechnet, in welchen Spalten die eliminierende Zeile Einträge hat. Diese Informationen wird in einem Array gespeichert, so daß während den Zeilenadditionen der Elimination schnell festgestellt werden kann, ob Zeile i (die eliminierende Zeile) in einer bestimmten Spalte einen Eintrag hat.

Zuerst werden die Einträge von Zeile j in zwei Mengen aufgeteilt. Handelt es sich um einen Eintrag in einer Spalte, in der Zeile i keinen Eintrag hat, so wird der Eintrag in die temporäre Struktur zum Aufbau der "neuen" Zeile j kopiert. (Diese Einträge bleiben ja unverändert.) Die Werte der anderen Einträge von Zeile j werden indiziert mit ihren Spaltenindizes in ein Array (*zwischen*) abgespeichert und dann später mit den Einträgen der Zeile i ver-

arbeitet. Die Situation, nachdem die Einträge von Zeile j verarbeitet wurden, enthält die Abbildung 6.4. Jetzt werden die Einträge von Zeile i verarbeitet.

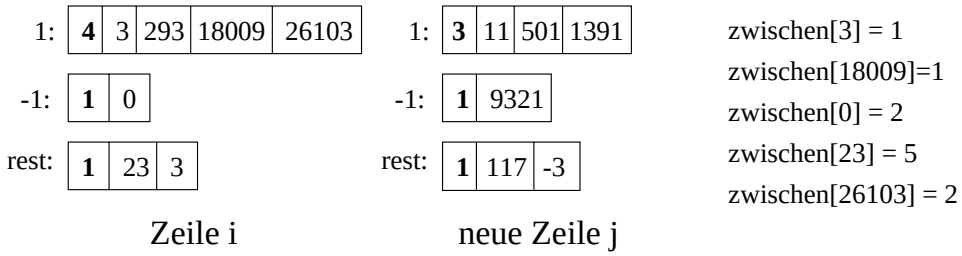


Abbildung 6.4: Situation nach Bearbeitung der Einträge von Zeile j

Dabei wird zuerst das *multi*-fache der Einträge bestimmt und zu den entsprechenden Einträgen des *zwischen*-Arrays addiert. Das Ergebnis wird dann in die Datenstruktur zum Aufbau der neuen Zeile j eingebaut. Den ersten 1-Eintrag hat Zeile i in Spalte 3. Da $multi = -2$ und $zwischen[3] = 1$ ist wird der neue -1-Eintrag in Spalte 3 aufgenommen ($1 * (-2) + 1 = -1$). Nach der Ermittlung des neuen Wertes werden die Werte des entsprechenden *zwischen*-Arrays auf Null gesetzt. Der nächste 1-Eintrag von Zeile i befindet sich in Spalte 293. Da $zwischen[293] = 0$ gibt es einen neuen *rest*-Eintrag -2 in Spalte 293. Der letzte 1-Eintrag von Zeile i ist in Spalte 26103. Da $zwischen[26103] = 2$ ist, ist der neue Wert in dieser Spalte Null und braucht nicht mehr in die Datenstruktur aufgenommen zu werden. Die neue Zeile j hat wie beabsichtigt keine Einträge mehr in dieser Spalte. Werden auch die -1-Einträge und die *rest*-Einträge in der obigen Art und Weise verarbeitet, so erhält man die Datenstruktur der neuen Zeile j die in Abbildung 6.5 dargestellt ist. Die ursprünglich vorhandene Sortierung nach aufsteigenden Indizes geht

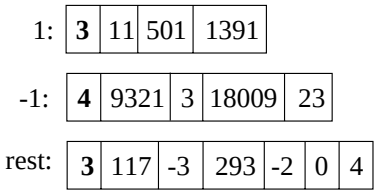


Abbildung 6.5: Ergebnis der Zeilenaddition

während der Zeilenaddition verloren. Für die Additionen wird diese Sortierung auch nicht gebraucht und am Ende des Kompaktifizierungschritts werden die Einträge wieder nach aufsteigenden Indizes sortiert und ausgegeben.

Damit sind alle Teile des Programms besprochen. Der folgende Algorithmus faßt die einzelnen Teile zusammen. Dabei wurde auf eine Duplizierung der Schritte 2 -

21 mit erhöhter Schranke bei der Berechnung der Spalteninformation vor Schritt 22 verzichtet. Nachdem keine weitere Laufzeitreduzierung durch Spalteneliminationen erzielt werden kann, bestimmt das Programm die optimalen rechten Seiten für die Lösungsberechnung (vgl. Abschnitt 5.4 (Seite 71)) sowie eine Numerierung der Zeilen und Spalten, die zu einer gleichmäßigen Auslastung der Knoten führt (vgl. Abschnitt 5.6.3 (Seite 83)). Das Programm erlaubt außerdem die Eingabe eines Zeilenbereichs, der nicht aus dem Gleichungssystem entfernt werden darf. Dies ist nötig, falls bestimmte Gleichungen in den Lösungen auftauchen müssen.

Kompaktifizierer

EINGABE: Anzahl der Lösungen, Anzahl der zusätzlichen Gleichungen
Anzahl der Knoten, Zeilenbereich, der im System verbleiben soll
AUSGABE: kompaktifiziertes System

```
(1)  Lese Gleichungssystem ein;
(2)  Berechne Spalteninformation;
(3)  Berechne Zeileninformation;
(4)  while (Spalten können eliminiert werden) do
(5)    zähler = 0;
(6)    for (i = 0; i < Anzahl Zeilen; i++) do
(7)      if (Zeile i kann eine Spalte eliminieren) then
(8)        Bestimme optimale eliminierende Zeile;
(9)        Bestimme genaue Abschätzung;
(10)       if ( ausgewählte Zeile kann Spalte eliminieren) then
(11)         Eliminiere Spalte;
(12)         zähler++;
(13)       fi
(14)       if (zähler = 1000) then
(15)         Berechne Spalteninformation;
(16)         Berechne Zeileninformation;
(17)         zähler = 0;
(18)       fi
(19)     fi
(20)   od
(21) od
(22) Bestimme optimale rechte Seiten;
(23) Bestimme Zeilen- und Spaltennumerierung;
(24) Ausgabe des kompaktifizierten Systems;
```

6.3 Praktische Tests und Ergebnisse

In diesem Abschnitt wird der Einfluß der verschiedenen Parameter untersucht, die im vorherigen Abschnitt angesprochen wurden. Außerdem wird der Einfluß der Anzahl der berechneten Lösungen auf die erzielte Laufzeitreduzierung anhand von Beispielen belegt.

6.3.1 Einfluß der Anzahl der berechneten Lösungen

Im ersten Test wird der Einfluß der Anzahl der berechneten Lösungen untersucht. Das Beispiel stammt aus der Berechnung individueller Logarithmen in einem Primkörper mit ungefähr 10^{65} Elementen. Dabei war es nötig, 41 Lösungen zu berechnen. Zum Vergleich wurde der Kompaktifizierungsschritt auch für 7 Lösungen durchgeführt. Die Tabelle 6.2 enthält die charakteristischen Daten des Gleichungssystems. Es wurden fast fünfmal mehr Gleichungen (Zeilen) als Unbekannte (Spalten) gesammelt. Die Tabelle enthält die Anzahl der Einträge für das komplette System und für ein System, das nur etwa 10 Gleichungen mehr als Unbekannte enthält.

Dimension		**Gesamtsystem**			**zu lösendes System**		
Spalten	Zeilen	**1**	**-1**	**long**	**1**	**-1**	**long**
26 159	129 067	1 816 900	2 885 791	338 532	372 856	118 979	29 400

Tabelle 6.2: Daten DL65

Die Tabelle 6.3 enthält einige Daten und Ergebnisse der beiden Kompaktifizierungsläufe. Es wurde jeweils der Wert der zulässigen Gewichtszunahme bei Elimination einer Spalte (Δ) zu Beginn und am Ende der Berechnung ermittelt. Dann folgt die Anzahl der Unbekannten, die nicht eliminiert werden konnte sowie die zur Kompaktifizierung benötigte Laufzeit des Algorithmus auf einer Sparc20. Außerdem wurde mit Hilfe der genauen Laufzeitanalyse von Kapitel 5.5 die Laufzeit des Lanczos Verfahrens vor und nach der Kompaktifizierung ermittelt. Die Zeiten wurden umgerechnet in Additionen. In beiden Fällen konnte eine deutliche Reduzierung dieser Laufzeit erreicht werden. Bei 7 Lösungen wird nur noch etwa ein Drittel der Laufzeit benötigt. Müssen 41 Lösungen berechnet werden, so kann die Gesamtlaufzeit des Lanczos Verfahrens auf unter ein Fünftel der ursprünglich benötigten Zeit reduziert werden. In beiden Fällen wurden auch jeweils einige Iterationen des Lanczos Verfahrens durchgeführt. Die dabei gemessenen Laufzeiten stimmen genau mit den rechnerisch aus dem Modell ermittelten Laufzeiten überein. Die Anzahl der Einträge im kompaktifizierten System sind ebenfalls in die Tabelle aufgenommen. Durch die deutlich größere Anzahl von Lösungen im zweiten Fall kann die Dimension des System stärker reduziert werden. Dies führt jedoch zu einer stärkeren Zunahme

	7 Lösungen	41 Lösungen
Δ(Start)	274.05	950.76
Δ(Ende)	366.08	1 161.82
Spalten	11 148	9 561
Laufzeit	16:16 min	41:50 min
# Additionen (Start)	$2.15 \cdot 10^{11}$	$6.87 \cdot 10^{11}$
# Additionen (Ende)	$6.19 \cdot 10^{10}$ ($\approx 28.79\%$)	$1.29 \cdot 10^{11}$ ($\approx 19.02\%$)
Zeit pro Iteration	6.11 s	14.84 s
1-Einträge	348 213	564 747
-1-Einträge	358 634	579 009
rest-Einträge	48 797	83 382

Tabelle 6.3: Ergebnis in Abhängigkeit der Anzahl der berechneten Lösungen

des Gesamtgewichts. Trotzdem kann bei größerer Anzahl von Lösungen ein deutlich besseres Ergebnis erzielt werden.

6.3.2 Einfluß der Anzahl der zusätzlichen Gleichungen

In diesem Test werden die Ergebnisse des Kompaktifizierers bei variierender Anzahl von zusätzlichen Gleichungen miteinander verglichen. Dabei benutze ich wieder das DL65 Beispiel mit 7 berechneten Lösungen (vgl. Tabelle 6.2). Als Eingabe wird jeweils das Gleichungssystem mit dem geringsten Gesamtgewicht und der angegebenen Anzahl von Gleichungen (Zeilen) benutzt.

	Beispiel 1	Beispiel 2	Beispiel 3	Beispiel 4
Zeilen (Start)	129 067	78 484	39 245	26 176
Spalten (Ende)	11 148	11 550	12 202	14 093
Laufzeit	16:16 min	10:56 min	7:13 min	5:00 min
Ergebnis	$\approx 28.79\%$	$\approx 30.74\%$	$\approx 35.07\%$	$\approx 47.29\%$
1-Einträge	348 213	370 289	411 815	484 161
-1-Einträge	358 634	359 046	393 229	463 724
rest-Einträge	48 797	49 330	53 149	61 572
Lanczos	18.92 h	20.20 h	23.04 h	31.07

Tabelle 6.4: Ergebnis in Abhängigkeit der Anzahl der zusätzlichen Gleichungen

Wird der Kompaktifizierer nur mit wenigen zusätzlichen Gleichungen gestartet (Beispiel 4), so erzielt er deutlich schlechtere Ergebnisse. Es kann kein Austausch von Zeilen vorgenommen werden, die durch die Elimination einer Spalte zu schwer geworden sind. Die Laufzeit des Kompaktifizierers ist im Beispiel 1 jedoch dreimal höher als in Beispiel 4 und der Hauptspeicherbedarf des Algorithmus steigt in ähnlicher Art und Weise an. Dennoch überwiegen die Vorteile von zusätzlichen Gleichungen deutlich, wie man an der Zeile (*Ergebnis*) mit der erzielten Laufzeitreduzierung deutlich erkennen kann. Aus diesem Grund werden immer möglichst viele zusätzliche Gleichungen aufgenommen, solange der Hauptspeicherbedarf die Grenzen der benutzten Maschine nicht übersteigt. Die letzte Spalte der Tabelle enthält die Laufzeiten des Lanczos Verfahrens für das jeweilig resultierende Gleichungssystem. Ohne den Kompaktifizierungsschritt hätte die Berechnung der 7 Lösungen 65.71 h auf einer Sparc20 gedauert.

6.3.3 Einfluß der Blockgröße

Da sich auch das Gewicht einiger anderer Spalten bei der Elimination einer Spalte ändert, muß die Spalten- und Zeileninformation im Algorithmus von Zeit zu Zeit neu aufgebaut werden (vgl. Abschnitt 6.2). Der Algorithmus steuert den Neuaufbau dieser Daten in Abhängigkeit der Anzahl der durchgeführten Spalteneliminationen. Die Tabelle 6.5 listet für das DL65 Beispiel für verschiedene Blockgrößen (Anzahl der Eliminationen, bevor ein Neuaufbau statt findet) die benötigte Laufzeit sowie die erreichte Reduktion in Prozent von der Ausgangssituation.

	Beispiel 1	**Beispiel 2**	**Beispiel 3**	**Beispiel 4**	**Beispiel 5**
Anzahl	25	50	100	500	1 000
Spalten	11 183	11 175	11 186	11 101	11 043
Laufzeit	1:26 h	48:14 min	27:10 min	11:27 min	10:45 min
Ergebnis	$\approx 28.65\%$	$\approx 28.78\%$	$\approx 28.76\%$	$\approx 28.65\%$	$\approx 28.56\%$

Tabelle 6.5: Ergebnis in Abhängigkeit der Blockgröße

Die Ergebnisse zeigen einen deutlichen Unterschied in der Laufzeit, jedoch nur sehr kleine Unterschiede in der reduzierten Laufzeit durch größere Blockgrößen. Der Neuaufbau der Spalteninformation dient also hauptsächlich zur Beschränkung des Speicherplatzbedarfs, da für Spalten, deren Gewicht durch andere Spalteneliminationen zu stark angewachsen ist, bis zu einem Neuaufbau der Spalteninformation ihre Einträge weitergepflegt werden. Steigt das Gewicht einer Spalte, für die diese Informationen aktualisiert wird, jedoch zu stark an, so stoppt der Algorithmus die Aktualisierung und entfernt die Daten dieser Spalte aus der Datenstruktur zum Aufbau der Spalteninformation. Mit diesem Zusatz kann eine Blockgröße von 1000 (Spalten) verwendet werden.

6.3.4 Vergleich der Ergebnisse für Paragon und Sparc20

In diesem Abschnitt vergleiche ich die erzielte Reduktion in der Laufzeit auf den beiden benutzten Plattformen (Paragon und Sparc20). Durch die Unterschiede in der verwendeten Langzahlarithmetik und dem Speicherzugriff ergeben sich sehr unterschiedliche Voraussetzungen für dem Kompaktifizierungsalgorithmus. Aufgrund der besseren Verteilung der Daten auf den vielen Knoten der Paragon im Gegensatz zur Speicherung der gesamten Daten auf einem Sparc Prozessor ergibt sich ein wesentlich besseres Speicherzugriffverhalten bei der Matrix-Vektormultiplikation auf der Paragon (vgl. *cache1*-Werte in Tabelle 5.9 (S. 75) und in Abschnitt 5.6.5). Außerdem können mit der *LIP* auf der Paragon im Verhältnis zur *libI* auf der Sparc20 zwei langen Zahlen besser addiert als multipliziert werden. Dies führt zu einer größeren Bedeutung der Matrix-Vektormultiplikation auf der Sparc20 und somit zu schlechteren Voraussetzungen für den Kompaktifizierungsalgorithmus. Ein weiterer Punkt, der zu besseren Ergebnissen des Kompaktifizierers für die Paragon führen sollte, ist die spezielle *kmult*-Funktion für positive Zahlen in der *LIP* (vgl. Tabelle 5.6 (S. 69)).

Die praktischen Tests beginnen mit COS-Systemen. Bei diesen Systemen wird jeweils nur eine Lösung benötigt, da diese zu den diskreten Logarithmen aller Faktorbasiselemente führt (vgl. Kapitel 3). Wegen ihres speziellen Aufbaus eliminiert man zuerst Unbekannte, die höchstens einmal vorkommen, und entfernt überflüssige Gleichungen, bis ein quadratisches System entsteht. Erst jetzt wendet man den Kompaktifizierer an. Der Kompaktifizierer wurde mit zwei Beispielen getestet, die sich deutlich in der Wahl der Faktorbasisgröße unterscheiden. Das erste Beispiel (COS58) ist der alte Weltrekord von Odlyzko und LaMacchia von 1990. Es wurde die Originalgröße der Faktorbasis verwendet. Das zweite Beispiel (COS85) stellt den aktuellen Weltrekord dar, der zusammen mit Damian Weber gehalten wird (vgl. [40]). Hier hat die Verwendung des Lanczos Verfahrens zu einer drastischen Reduzierung der Faktorbasisgröße geführt. Die Tabelle 6.6 enthält die Ausgangsdaten und die Daten nachdem ein quadratisches System erstellt wurde.

	COS58	**COS85**
Unbekannte (Start)	180 000	69 985
Gleichungen (Start)	250 000	119 951
Dimension (Ende)	73 004	68 894
Laufzeit (Ende)	10.14 Tage	26.03 Tage

Tabelle 6.6: Ausgangsdaten COS Systeme

Durch die wesentlich größere Faktorbasis des COS58 Beispiels kann dort eine sehr große Reduzierung der Dimension bereits bei der Aufstellung eines quadratischen Systems erzielt werden. Danach wurde der Kompaktifizierer auf beide Systeme mit Parametern für die Paragon und die Sparc20 angewendet. Die Tabelle 6.7 enthält

die Dimension der reduzierten Systeme sowie die berechnete Laufzeit in Prozent der ursprünglichen Laufzeit vor dem Kompaktifizierungsschritt. Außerdem wird die Laufzeit des Kompaktifizierers angegeben.

	COS58	COS85
Dimension (Paragon)	27 078	53 364
Dimension (Sparc20)	29 159	55 148
Reduktion (Paragon)	28.17 %	82.9 %
Reduktion (Sparc20)	35.86 %	89.5 %
Laufzeit (Paragon)	59:54 min	61:43 min
Laufzeit (Sparc20)	49:24 min	45:24 min

Tabelle 6.7: Ergebnisse des Kompaktifizierers bei COS Systemen

Wie erwartet fallen die Reduzierungsergebnisse für die Paragon höher aus. Die Faktorbasisgröße im COS58 Beispiel sorgt für extrem dünnbesetzte Systeme, die mit dem Kompaktifizierungsalgorithmus sehr stark reduziert werden können. Im COS85 Beispiel fällt die Reduzierung mit fast 20% deutlich niedriger aus. Dank des Kompaktifizierungsschrittes spart man jedoch immer noch ca. 4 Tage bei einem Einsatz von einer knappen Stunde Rechenzeit.

Im Gegensatz zu den COS-Beispielen benötigt man bei NFS-Beispielen immer mehr als 5 Lösungen (vgl. Abschnitt 5.7). Mehrere Lösungen bedeutet aber gleichzeitig auch einen höheren Wert für Δ, und somit größere Möglichkeiten zur Reduktion (vgl. Abschnitt 6.3.1). Im folgenden werden die Ergebnisse des Kompaktifizierungslaufs für einige Gleichungssysteme aus Berechnungen mit dem Number Field Sieve für Diskrete Logarithmen vorgestellt. Dabei bestätigt sich die Tendenz, daß es bei relativ klein gewählter Faktorbasis kaum Möglichkeiten zur Reduktion gibt. Vergleicht man ein älteres DL75 Beispiel (als die Faktorbasis mangels Kompaktifizierer noch klein gewählt werden mußte) mit einer aktuelleren DL65 Berechnung, so kann man den Unterschied deutlich sehen. In beiden Fällen wurde die Größe der Faktorbasis etwa gleich groß gewählt, obwohl in der Theorie die Faktorbasis bei 10 Dezimalstellen Unterschied mindestens verdoppelt werden sollte. Die relativ kleine Faktorbasis im DL75 Beispiel führt zu einem deutlich dichteren System. Obwohl das DL75 Beispiel etwa ein Drittel weniger Gleichungen als das DL65 Beispiel hat, existieren zu Beginn des Kompaktifizierungslaufs etwa 10 mal mehr Einträge im DL75 System als im DL65 System. Somit ist das DL75 System fast 15 mal dichter besetzt. Die Ergebnisse des Kompaktifizierungslaufs, der wieder für beide Plattformen durchgeführt wurde, dieser beiden extremen Beispiele enthält Tabelle 6.8. In beiden Beispielen werden jeweils 7 Lösungen berechnet.

Abschließend gebe ich noch die Ergebnisse für die beiden größten berechneten Beispiele in Tabelle 6.9 an. Es handelt sich dabei um den aktuellen Weltrekord einer

	DL65	DL75
Unbekannte (Start)	26 160	25 061
Gleichungen (Start)	129 067	89 089
Dimension (Paragon)	9 321	24 094
Dimension (Sparc20)	10 037	24 392
Reduktion (Paragon)	23.47 %	96.1 %
Reduktion (Sparc20)	30.12 %	98.8 %
Laufzeit (Paragon)	14:57 min	20:08 min
Laufzeit (Sparc20)	10:07 min	19:28 min
1-Einträge (vorher)	1 816 900	18 881 343
-1-Einträge (vorher)	2 885 791	19 881 099
rest-Einträge (vorher)	338 532	2 489 409

Tabelle 6.8: Vergleich DL65 und DL75

Diskreten Logarithmen Berechnung in endlichen Primkörpern (DL85) und der ersten Phase zur Berechnung der McCurley Challenge (DL126) (siehe [27]).

	DL85	DL126
Unbekannte (Start)	70 342	39 995
Gleichungen (Start)	175 046	90 971
Dimension (Paragon)	49 070	33 010
Dimension (Sparc20)	52 009	34 057
Reduktion (Paragon)	69.3 %	85.0 %
Reduktion (Sparc20)	77.2 %	89.8 %
Laufzeit (Paragon)	65:20 min	30:45 min
Laufzeit (Sparc20)	67:26 min	31:42 min

Tabelle 6.9: Vergleich DL85 und DL126

Die Tests haben gezeigt, daß man mit dem Kompaktifizierer gute Reduzierungen der Laufzeit des abschließenden Lanczos Verfahrens erzielen kann. Dabei kann die Laufzeit der parallelen Version auf der Paragon immer stärker reduziert werden als die Laufzeit der sequentiellen Version.

Fazit

In dieser Arbeit wurde ein Gleichungslöser vorgestellt, der erfolgreich beim Lösen großer dünnbesetzter Gleichungssysteme über endlichen Primkörpern eingesetzt wurde.

Bereits mit der Generierung der Systeme beginnend wurden deren spezielle Eigenschaften untersucht und ausgenutzt. Dies führte zu einem effizienten Gleichungslöser, dessen drei Teile aufeinander abgestimmt wurden. Dabei handelt es sich um:

- die Modifikation der Double Large Prime Variante (Generierung der Systeme),
- den Preprocessingschritt, der aufbauend auf Ideen der strukturierten Gaußelimination die Laufzeit des Lanczos Verfahrens minimiert und um
- die (sequentielle bzw. parallele) Implementierung des Lanczos Verfahrens.

Ein Schwerpunkt der Arbeit war dabei der Ansatz zum Parallelisieren des Lanczos Verfahrens und dessen Umsetzung in ein Programm. Trotz des hohen Kommunikationsbedarfs der Anwendung konnte die Rechenleistung des Parallelrechners optimal ausgenutzt werden. Dazu mußte die Kommunikation komplett mit Berechnungen überlagert, und die gleichmäßige Auslastung der Knoten garantiert werden. Die Verbesserungen am Lanczos Verfahren und die Entwicklung eines Laufzeitmodells stellten weitere wichtige Schritte dar.
Die Verbesserungen in der Leistungsfähigkeit des Gleichungslösers führten auch zu Anpassungen bei der Generierung der Systeme. So konnte durch eine möglich gewordene Vergrößerung der Faktorbasis die Siebphase beschleunigt werden. Im Laufe der Zusammenarbeit mit Damian Weber konnte der bestehende Rekord bei DL-Berechnungen um fast 30 Dezimalstellen verbessert werden. Dabei muß man beachten, daß sich der Rechenaufwand mit jeweils drei zusätzlichen Dezimalstellen mindestens verdoppelt. Im gleichen Zeitraum konnte der Rekord bei Faktorisierungen lediglich um 10 Dezimalstellen erhöht werden. Die dabei benötigte Rechenzeit wurde durch ein weltweites Projekt bereitgestellt, an dem mehr als 1000 Personen mit vielen Rechner teilnahmen, wohingegen die in dieser Arbeit vorgestellten Berechnungen auf einem Netz von Rechner der Universität des Saarlandes und dem Parallelrechner in Jülich durchgeführt wurden. Die praktische Differenz zwischen DL-Berechnungen und Faktorisierungen konnte also unter anderem mit Hilfe dieser Arbeit deutlich verringert werden.

Anhang A

LC-Programm zur Laufzeitbestimmung

```
#!/exil/lc/bin/lc.sparc8
int main()
{
// Rechnet mit den gegebenen Parametern (n, w* , SOLS,...) in dem Laufzeitmodell

cache1  = 2.0;
cache2 = 1.04;
cache3 = 1.3;
c = 15.05/12000000; // Zeitumrechnung Sekunden pro Addition
//-------------------------------------------------------------------------
// COS 85
//-------------------------------------------------------------------------
//n      = 51855;
//SOLS   = 1;
//w0_0 = 2339061;
//w0_1 = 2631285;
//w2 = 212230/2.0;
//w3 = 0;
//-------------------------------------------------------------------------
// DL 85
//-------------------------------------------------------------------------
n      = 52035;                 // Dimension des Systems
SOLS   = 10;                    // Anzahl der Loesungen
w0_0 = 3872070 ;                // Anzahl (+)Eins-Eintr"age
w0_1 = 4228111;                 // Anzahl (-)Eins-Eintr"age
w2 = 337122/2.0;                // Anzahl (-)long- Eintr"age bzw. alle long
w3 = 0;                         // Anzahl (+)long-Eintr"age, schneller verarbeitet
//-----------------------------------------------
// Wie oft langsamer als Addition ist
//-----------------------------------------------
mult    = 23.279642 ;
square  = 23.279642 ;
kmult   = 2.241163 ;
addmont = 1.523937 ;
submont = 1.228188 ;
sub     = 1.0551230 ;
red     = 4.806711 ;
inv     = 1277.181208 ;

n_mult = f_mult = kmult;
//-----------------------------------------------
// ENDE DER EINGABEDATEN
//-----------------------------------------------
print ("Loesungern = ", SOLS,"  DIM = ",n,"\n 1: ",w0_0,"   -1: ",w0_1,"   long:",w2,"\n");
```

```
omega = cache1 *(w0_1* sub + w0_0) + cache3*(w2*n_mult + w3*f_mult);
t1 = cache2 * (inv + (2 + SOLS)*mult);
t2 = cache2 *(square + 2 * submont+ (2 + SOLS)*addmont+(3+SOLS)* mult)+ 2 * red *cache3;
print ("ADD_CACHE = ",cache1," MUL_CACHE ",cache2,"cache3 = ",cache3,"\n");
print ("-------------------------------------------------\n");
print ("   ADD  : ", cache1 * w0_0 *c," Sekunden\n");
print ("   SUB  : ", cache1 *sub* w0_1 *c," Sekunden\n");
print ("   kMUn : ", cache3 * w2 * n_mult *c," Sekunden\n");
print ("   kMUf : ", cache3 * w3 * f_mult *c," Sekunden\n");
print ("   RED  : ", cache3 *n * red *c," Sekunden\n");
print ("   kMU+ : ", cache3 *(n * red +w2 * n_mult) *c," Sekunden\n");
print ("-------------------------------------------------\n");
print ("MAT_MULT_VEC     : ",2.0 * (x2 +n * red*cache3)* c," Sekunden\n");
print ("inneres Produkt(S) : ",cache2 *(square+ addmont)* n * c,"Sekunden\n");
print ("inneres Produkt(N) : ",cache2 *(mult+ addmont)* n * c," Sekunden\n");
print ("update w           : ",cache2 *(n*2*(mult+submont)+ mult *2+inv )*c," Sekunden\n");
print ("update SOLS        : ",cache2 *(SOLS*addmont*n+(SOLS* (n+1)*mult))*c," Sekunden\n");
print ("-------------------------------------------------\n");
print ("Iteration          : ", (t1 + 2 * omega + n *t2)*c," Sekunden\n");
print ("Gesamtarbeit       : ",n * (t1 + 2 * omega + n *t2), " Additionen\n");
print ("Gesamtarbeit       : ",n * (t1 + 2 * omega + n *t2)*c/3600, " Stunden\n");
print ("-------------------------------------------------\n");
//-----------------------------------------------------------------
// Berechnungen :
//-----------------------------------------------------------------
// Wie stark darf Gewicht anwachsen bei Elimination einer Spalte ?
//-----------------------------------------------------------------
delta = ((2 * n -1)* t2 + t1 + 2 * omega)/(2 * (n-1));
print ("Delta     (n-=1) : ",delta,"\n");
print ("1-Eintraege    += ", delta/cache1," \n");
print ("-1-Eintraege   += ", delta/(cache1 * sub)," \n");
print ("rest-Eintraege += ", delta/f_mult," \n");
print ("-------------------------------------------------\n");
//-----------------------------------------------------------------
// Wie stark muss Gewicht abnehmen beim Loeschen einer Spalte und
// Berechnung einer zusaetzlichen Loesung ?
//-----------------------------------------------------------------
t3 = cache2 * (mult+ addmont);
t4 = cache2* inv + (3 + SOLS -n)  * cache2 *  mult;
delta = ((n-1)^2 * t3 -  (2 * n -1) * t2 - 2 * omega - t4 ) / (2 * (n-1));
print ("Delta(SOLS += 1, n -= 1) : ",delta,"\n");
}
```

Anhang B

Programm zum Aufbau eines Kommunikationsrings

```
/*--+---+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--*/
/*  Aufbau eines Kommunikationsringes                             */
/*  basiert auf einem Algorithmus von Stefan Burkhardt            */
/*--+---+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--*/
/* Time-stamp: "96/12/03 14:17:54 denny"                          */
/*--+---+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--*/

long *Feld, *nachf, *vorg;
long cols,rows, anz_nodes;


void comp_folge()
{
    long     ct, startval, up;
    long     i, j;

    startval=0;
    ct=0;
    up=1;
    Feld[ct]=0;
    ct++;

    for (i=0; i<rows; i++)
    {
        for (j=0; j<cols-1; j++)
        {
            startval+=up;
            Feld[ct]=startval;
            ct++;
        }
        startval+=up;
        startval+=cols;
        up=up*-1;           /* Aendere die Laufrichtung mit jeder Zeile */
    }
    startval-=cols;
    for (i=0; i<rows-1; i++)
    {
        Feld[ct]=startval;
        startval-=cols;
        ct++;
    }

    for(i=0;i<anz_nodes -1;i++)
        nachf[Feld[i]] = Feld[i+1];
```

```
    nachf[Feld[anz_nodes - 1]] = Feld[0];

    for(i = 1; i < anz_nodes; i++)
        vorg[Feld[i]] = Feld[i-1];
    vorg[Feld[0]] = Feld[anz_nodes - 1];
}

main (argc, argv)
int argc;
char *argv[];
{

    register long i;

    rows =atoi(argv[1]);
    cols = atoi(argv[2]);
    if ((rows & 1 ) == 1 )
    {
        printf("Sorry!\nrows muss gerade sein\n");
        exit(0);
    }

    anz_nodes = cols * rows;  /* Anzahl der beteiligten Knoten */

    Feld  = (long *)malloc(anz_nodes * sizeof(long));
    nachf = (long *)malloc(anz_nodes * sizeof(long));
    vorg  = (long *)malloc(anz_nodes * sizeof(long));

    comp_folge();

    printf("Nr. | Feld | nachf | vorg \n");
    printf("--------------------------\n");
    for(i=0;i<anz_nodes;i++)
        printf("%3d | %4d | %5d | %3d\n",i,Feld[i],nachf[i], vorg[i]);
    return(0);
}
```

Literaturverzeichnis

[1] R. Berrendorf, H. Burg, U. Detert, R. Esser, M. Gerndt, R. Knecht, *Intel Paragon XP/S - Architecture, Software Environment, and Performance*, Interner Bericht KFA Jülich GMBH, KFA-ZAM-IB-9409,http://www.kfa-juelich.de/zam/docs/autoren94/berrendorf1.html

[2] D.P. Bertsekas, J.N. Tsitsiklis, *Parallel and Distributed Computations: Numerical Methods*, Prentice Hall, 1989

[3] S. Burkhardt, *Parallele Implementierung des Lanczos Verfahrens über endlichen Primkörpern*, Diplomarbeit, Universität des Saarlandes, Saarbrücken, 1997

[4] D. Coppersmith, A. Odlyzko, R. Schroeppel, *Discrete logarithms in GF(p)*, Algorithmica, **1** (1986), 1–6

[5] T.H. Cormen, *Introduction to Algorithms*, The MIT Press, 1990

[6] T. Denny, *Faktorisieren mit dem Quadratischen Sieb*, Diplomarbeit, Universität des Saarlandes, Saarbrücken, 1993

[7] T. Denny, B. Dodson, A.K. Lenstra, M.S. Manasse, *On the factorization of RSA-120*, Advances in Cryptology - Crypto '93, Lecture Notes in Computer Science, **773** (1993), 166–174

[8] T. Denny, V. Müller, *On the reduction of composed relations from the number field sieve*, Algorithmic Number Theory – ANTS II, Lecture Notes in Computer Science, **1122** (1996), 75–90

[9] T. Denny, D. Weber, *A 65-digit prime DL computation*, Message published on the NumberTheory Net, Oct. 1995

[10] W. Diffie, M.E Hellman, *New directions in cryptography*, IEEE Trans. Information Theory, **22** (1976), 644–654

[11] B. Dodson, A.K. Lenstra, *NFS with Four Large Primes: An Explosive Experiment*, Advances in Cryptology - Crypto '95, Lecture Notes in Computer Science, **963** (1995), 372–385

[12] K. Dowd, *High Performance tuning*, O'Reilly & Associates Inc., 1993

[13] T. ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, IEEE Trans. Information Theory **31** (1985), 469–472

[14] M.R. Garey, D.S. Johnson, *Computers and Intractability: A guide to the Theory on $\mathcal{NP}$ completeness*, W.H. Freemann and Company, 1979

[15] G.H. Golub, C.F. van Loan, *Matrix Computations*, second edition, John Hopkins University Press, Baltimore and London, 1989

[16] D.M. Gordon, *Discrete Logarithms in GF(p) using the Number Field Sieve*, SIAM J. Discrete Mathematics **6** (1993), 124–138

[17] D.M. Gordon, K. McCurley, *Massively parallel computation of discrete logarithms*, Advances in Cryptology - Crypto '92, Lecture Notes in Computer Science, **740** (1992), 310–323

[18] D.S. Greenberg, K.S. McCurley, *All-to-All Broadcast on Mesh Connected Parallel Computers*, Technical report, Sandia National Laboratories, 1992

[19] O. Groß, *Der Block Lanczos Algorithmus über GF(2)*, Diplomarbeit, Universität des Saarlandes, Saarbrücken, 1994

[20] J.D. Horton, *A polynomial-time algorithm to find the shortest cycle basis of a graph*, SIAM Journal of Computations **16** (1987) , 344–355

[21] R.M. Huizing, *An implementation of the number field sieve*, Technical report, CWI Report NM-R9511, July 1995

[22] E. Lamprecht, *Einführung in die Algebra*, Birkhäuser Verlag, Basel, 1978

[23] B.A. LaMacchia, A.M. Odlyzko, *Solving large sparse systems over finite fields*, Advances in Cryptology - Crypto '90, Lecture Notes in Computer Science, **537** (1991), 109–133

[24] C. Lanczos, *Solution of systems of linear equations by minimized iterations*, J. Res. Nat. Bureau of Standards **49** (1952), 33–53

[25] A.K. Lenstra, M.S. Manasse, *Factoring with two large primes*, Mathematics of Computation, **63** (1994), 72–82

[26] A.K. Lenstra, H.W. Lenstra, *The development of the number field sieve*, Springer Verlag, 1993

[27] K.S. McCurley, *The discrete logarithm problem, cryptology and computational number theory*, Proc. Symp. in Applied Mathematics, American Mathematical Society, 1990

[28] K. Mehlhorn, *Graph Algorithms and $\mathcal{NP}$ - Completeness*, Springer Verlag, 1984

[29] P.L. Montgomery, *A block Lanczos algorithm for finding dependencies over GF(2)*, Advances in Cryptology - Eurocrypt '95, Lecture Notes in Computer Science, **921** (1995), 106–120

[30] P.L. Montgomery, *Modular multiplication without trial division*, Mathematics of Compation, **44** (1985), 519–521

[31] National Institute of Standards and Technology (NIST), *Publication XX: Announcement and Specifications for a Digital Signature Standard (DSS)*, 1992

[32] A.M. Odlyzko, *Discrete logarithms in finite fields and their cryptographic significanse*, Advances in Cryptology - Eurocrypt '84, Lecture Notes in Computer Science, **209** (1985), 224–314

[33] C. Pomerance, J.W. Smith, *Reduction of huge sparce matrices over finite fields via created catastrophes*, Experimental Mathematics **1**, (1992), 89–94

[34] H. Riesel, *Prime Numbers and Computer Methods for Factorization*, Second Edition, Birkhäuser Verlag, Boston, 1994

[35] R. Rivest, A. Shamir, L. Adleman, *A new method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM **21**, (1978), 120–126

[36] O. Schirokauer, *Discrete logarithms and local units*, Phil. Trans. R. Soc. Land. A 345 (1993), 409–423

[37] O. Schirokauer, D. Weber, T. Denny, *Discrete Logarithms: The effectiveness of the index calculus method*, Algorithmic Number Theory – ANTS II, Lecture Notes in Computer Science, **1122** (1996), 337–361

[38] R.D. Silvermann, *The Multiple Polynomial Quadratic Sieve*, Mathematics of Computation, **48** (1987), 329–339

[39] D. Weber, *An Implementation of the General Number Field Sieve to Compute Discrete Logarithms mod p*, Advances in Cryptology - Eurocrypt '95, Lecture Notes in Computer Science, **921** (1995), 95–105

[40] D. Weber, *On the Computation of Discrete Logarithms in Finite Prime Fields*, Dissertation, Universität des Saarlandes, 1997

[41] D. Weber, *Computing Discrete Logarithms with the Number Field Sieve*, Algorithmic Number Theory – ANTS II, Lecture Notes in Computer Science **1122** (1996), 390–403

[42] D.H. Wiedemann, *Solving sparse linear equations over finite fields*, IEEE Trans. Information Theory, **32** (1986), 54–62

[43] J. Zayer, *Faktorisieren mit dem Number Field Sieve*, Dissertation, Universität des Saarlandes, 1995