Johann Wolfgang Goethe-Universität
Frankfurt
Fachbereich Informatik

Diplomarbeit

Gitteralgorithmen auf dem Vektorrechner

Markus Michalek
Amselweg 5
63674 Altenstadt-Waldsiedlung

Altenstadt, den 04.04.1995

Betreuer: Professor Dr. C.P. Schnorr

Inhaltverzeichnis

1. Einleitung

Die Anfänge der Gittertheorie reichen in das letzte Jahrhundert, wobei die wohl bekanntesten Ergebnisse auf Gauß, Hermite und Minkowski zurückgehen. Die Arbeiten sind jedoch zumeist in der Schreibweise der quadratischen Formen verfaßt; erst in den letzten Jahrzehnten hat sich die von uns verwendete Gitterschreibweise durchgesetzt. Diese ist zum einen geometrisch anschaulicher, zum anderen wurden in den letzten Jahren für diese Schreibweise effiziente Algorithmen entwickelt, so daß Probleme der Gittertheorie mittels Computer gelöst werden können. Ein wichtiges Problem ist, in einem Gitter einen kürzesten nicht verschwindenden Vektor zu bestimmen.
Den Grundstein für diese algorithmische Entwicklung legten A.K. Lenstra, H.W. Lenstra Jr. und L. Lovász mit ihrer Arbeit [LLL82]. In dieser führten sie einen Reduktionsbegriff ein, der durch einen Polynomialzeitalgorithmus erreicht werden kann. Ein weiterer Reduktionsbegriff, die Blockreduktion, geht auf Schnorr [S87] zurück.
Euchner hat im Rahmen seiner Diplomarbeit [E91] effiziente Algorithmen für diese beiden Reduktionsbegriffe auf Workstations implementiert und auch in Dimensionen > 100 erfolgreich getestet. Die Verbesserungen von Schnittechniken des in der Blockreduktion verwendeten Aufzählungsverfahrens und die Einführung einer geschnittenen Aufzählung über die gesamte Gitterbasis hat Hörner in seiner Diplomarbeit [H94] beschrieben.
Ziel der folgenden Arbeit war es nun, diese bereits auf sequentiellen Computern implementierten Algorithmen zu modifizieren, um auf parallelen Rechnern, speziell Vektorrechnern, einen möglichst hohen Geschwindigkeitsgewinn zu erzielen. Wie in den seriellen Algorithmen werden die Basisvektoren stets in exakter Darstellung mitgeführt, so daß das Endergebnis einer Berechnung nicht durch Rundungsfehler verfälscht wird.
Im folgenden Kapitel zwei werden zu Anfang Definitionen und Schreibweisen der Gittertheorie eingeführt. Anschließend werden Eigenschaften von reduzierten Gitterbasen beschrieben.
Kapitel drei behandelt allgemeine Rucksackprobleme.
Das vierte Kapitel unter der Überschrift Parallelrechner beginnt mit der Vorstellung verschiedener Arten der Klassifikation. Danach folgt eine Einführung in das Arbeitsprinzip der Vektorrechner. Zuletzt wird der im Rahmen dieser Arbeit verwendete Vektorrechner genauer vorgestellt.

Das fünfte Kapitel behandelt die verwendeten Reduktionsalgorithmen. Dabei werden zuerst die vorhandenen seriellen Versionen vorgestellt, anschließend wird erläutert, auf welche Art die Funktionen vektorisiert wurden.
Die Ergebnisse der Anwendung dieser Algorithmen auf allgemeine Rucksackprobleme werden im Kapitel sechs mit den entsprechenden aus [H94] verglichen. Weiterhin wird gezeigt, daß mit den gleichen Algorithmen das Kryptoschema von Chor und Rivest [CR84] erfolgreich angegriffen werden kann.

2. Gitter

2.1 Grundlegende Definitionen

Die in dieser Arbeit verwendeten Schreibweisen lehnen sich an das Skriptum „Gittertheorie und ganzzahlige Optimierung“ [S91] an. Ebenso finden sich dort die grundlegenden Sätze und zugehörigen Beweise, für die in diesem Kapitel keine separate Referenz angegeben ist.

Im folgenden wird die Existenz eines Skalarproduktes $\langle .,. \rangle : \mathbb{R}^n \times \mathbb{R}^n \longrightarrow \mathbb{R}$ vorausgesetzt. Dies kann eine beliebige positiv definite Bilinearform sein; wenn jedoch in dieser Arbeit nichts anderes angegeben ist, wird das Standardskalarprodukt

$$\langle x, y \rangle \quad = \quad \langle (x_1, \ldots, x_n), (y_1, \ldots, y_n) \rangle \quad := \quad \sum_{i=1}^{n} x_i y_i$$

verwendet. Desweiteren bezeichne

$$\|x\| \quad := \quad \sqrt{\langle x, x \rangle}$$

die Norm, die durch das Skalarprodukt induziert wird. Zwei weitere gebräuchliche Normen sind die Maximum- ($\|.\|_\infty$) und Eins-Norm ($\|.\|_1$), die durch

$$\|x\|_\infty := \max_{1 \le i \le n} |x_i| \qquad \|x\|_1 := \sum_{i=1}^{n} |x_i|$$

definiert sind.

Definitionen: Seien $b_1, \ldots, b_m \in \mathbb{R}^n$ linear unabhängige Vektoren; dann heißt die additive Gruppe

$$\sum_{i=1}^{m} \mathbb{Z} b_i \quad = \quad \left\{ \sum_{i=1}^{m} x_i b_i \,\middle|\, (x_1, \ldots, x_m) \in \mathbb{Z}^m \right\} \quad \subset \quad \sum_{i=1}^{m} \mathbb{R} b_i \quad \subset \quad \mathbb{R}^n$$

ein **Gitter**, das mit $L(b_1, \ldots, b_m)$ bezeichnet wird. Die Vektoren $b_1, \ldots, b_m$ nennen wir eine **Basis** des Gitters, m den **Rang** $R(L)$ bzw. die Dimension. Ein Gitter heißt **vollständig** oder volldimensional, wenn $m = n$, und **ganzzahlig**, wenn $L \subset \mathbb{Z}^n$ gilt. Ist die Reihenfolge der Basisvektoren

fest vorgegeben, so handelt es sich um eine **geordnete** Gitterbasis. Mit $span(b_1, \ldots, b_k)$ bezeichnen wir den von den Vektoren $b_1, \ldots, b_k$ aufgespannten Unterraum des $\mathbb{R}^n$. □

Satz: Jede additive diskrete Gruppe $G \subset \mathbb{R}^n$ ist ein Gitter. Ebenso gilt, daß jedes Gitter diskret ist. □

Satz: Seien $b_1, \ldots, b_m \in \mathbb{R}^m$ linear unabhängige Vektoren. $\bar{b}_1, \ldots, \bar{b}_m$ ist genau dann eine Basis von $L(b_1, \ldots, b_m)$, wenn eine Matrix $T \in Gl_m(\mathbb{Z})$ existiert mit

$$[\bar{b}_1, \ldots, \bar{b}_m] \quad = \quad [b_1, \ldots, b_m] \quad T.$$

Hierbei bezeichnet $Gl_m(\mathbb{Z})$ die Gruppe aller ganzzahligen, ganzzahlig invertierbaren $m \times m$ Matrizen. □

Eine Folgerung aus diesem Satz ist, daß jedes Gitter unendlich viele Gitterbasen hat. Diese erhält man, indem man eine gegebene Basismatrix mit einem der unendlich vielen Elemente von $Gl_m(\mathbb{Z})$ multipliziert.

Definition: Die **Determinante** $det(L)$ des Gitter $L(b_1, \ldots, b_m)$ ist definiert durch:

$$det(L) \quad := \quad vol_m \left\{ \sum_{i=1}^{m} t_i b_i \,\middle|\, 0 \leq t_i \leq 1 \qquad i = 1, \ldots, m \right\}$$

Hierbei bezeichnet vol_m das m-dimensionale Volumen, das über die sogenannte Grundmasche des Gitters genommen wird. □

Satz: Für jedes Gitter $L(b_1, \ldots, b_m)$ gilt:

$$det(L) \quad = \quad \sqrt{det[\langle b_i, b_j \rangle]_{1 \leq i,j \leq m}} \quad = \quad \sqrt{det(B^T B)}$$

□

Die Determinante ist von der Wahl der Basis B unabhängig, da für $T \in Gl_m(\mathbb{Z})$ und $B = \bar{B}T$ gilt:

$$\begin{aligned} det(L) &= \sqrt{det(B^T B)} \\ &= \sqrt{det(T^T \bar{B}^T \bar{B} T)} \\ &= \sqrt{det(T^T) det(\bar{B}\bar{B}^T) det(T)} \\ &= \sqrt{det(\bar{B}\bar{B}^T)} \\ \text{wegen } |det(T)| &= |det(T^T)| = 1 \end{aligned}$$

Sie bildet daher eine Gitterkonstante.

Definitionen: Sei $b_1, \ldots, b_m$ eine geordnete Gitterbasis und $\pi_i : \mathbb{R}^n \to span(b_1, \ldots, b_{i-1})^\perp$ die **orthogonale Projektion** bezüglich der Vektoren $b_1, \ldots, b_{i-1}$. Damit gilt:

$$\begin{aligned} \pi_i(b) &\in span(b_1, \ldots, b_{i-1})^\perp \\ b - \pi_i(b) &\in span(b_1, \ldots, b_{i-1}) \end{aligned} \qquad b \in \mathbb{R}^n$$

Die Vektoren $\hat{b}_1, \ldots, \hat{b}_m$, die durch die folgende Vorschrift

$$\begin{aligned} \hat{b}_1 &= \pi_1(b_1) = b_1 \\ \pi_i(b_j) &= b_j - \sum_{k=1}^{i-1} \mu_{i,k} \pi_k(b_k) \quad i = 2, \ldots, m \\ \hat{b}_i &= \pi_i(b_i) \end{aligned}$$

mit

$$\mu_{i,j} = \begin{cases} \frac{\langle b_i, \hat{b}_j \rangle}{\langle \hat{b}_j, \hat{b}_j \rangle} & j < i \\ 1 & j = i \\ 0 & j > i \end{cases}$$

erhalten werden, sind das **Orthogonalsystem** zu den Vektoren $b_1, \ldots, b_m$. Die $\mu_{i,j}$ heißen **Gram-Schmidt-Koeffizienten**. $\hat{b}_i$ heißt **i-te Höhe**. □
Daraus ergibt sich die Darstellung

$$[b_1, \ldots, b_m] = [\hat{b}_1, \ldots, \hat{b}_m][\mu_{i,j}]^T$$

sowie

$$detL(b_1, \ldots, b_m) = \prod_{i=1}^{m} \|\hat{b}_i\|.$$

Definition: Zwei Gitter L_1, L_2 heißen **isometrisch**, wenn es eine isometrische (skalarproduktserhaltende) Abbildung $V : span(L_1) \longrightarrow span(L_2)$ mit $V(L_1) = L_2$ gibt. □
Bemerkung: Isometrische Abbildungen verändern keine geometrische Aussagen wie Skalarprodukte, Volumina und daraus abgeleitete Größen. Jedoch können sich Normen ändern, die nicht durch ein Skalarprodukt induziert werden.

2.2 Reduzierte Gitterbasen

Definition: Sei $L \subset \mathbb{R}^n$ ein Gitter vom Rang m. Die **sukzessiven Minima** $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_m$ sind wie folgt erklärt:

$$\lambda_i = \min \{r > 0 \quad | \quad \exists \text{ linear unabhängige } b_1, \ldots, b_i \in L \setminus \{0\} \quad \|b_j\| \leq r\}$$

□

Definition: Der **Orthogonalitätsdefekt** $\mathcal{OD}(b_1, \ldots, b_m)$ einer geordneten Gitterbasis ist definiert durch

$$\mathcal{OD}(b_1, \ldots, b_m) := \frac{\prod_{i=1}^{m} \|b_i\|^2}{\prod_{i=1}^{m} \|\hat{b}_i\|^2}$$

□

Bemerkung: Der Orthogonalitätsdefekt ist ein Maß für die Reduktionsgüte einer Gitterbasis; er ist stets ≥ 1. Ist er gleich 1, so stehen die Vektoren paarweise senkrecht aufeinander. Dann ist das Gitter bei geeigneter Numerierung der Basisvektoren bezüglich jeder bislang bekannten Reduktionsbedingung reduziert.

Satz: Sei $b_1, \ldots, b_m$ Gitterbasis, dann gilt für $i = 1, \ldots, m$:

$$\max_{j \leq i} \|b_j\| \geq \lambda_i \geq \min_{j \geq i} \|\hat{b}_j\|$$

□

Definition:

$$\lambda_{1,\infty}(L) := \min \{\|x\|_\infty \quad | \quad 0 \neq x \in L\}$$

□

Es gilt: $\lambda_{1,\infty} \leq \lambda_1 \leq \sqrt{n}\lambda_{1,\infty}$, wenn n die Dimension des Raumes ist.

Satz: Sei $L \subset \mathbb{R}^m$ ein vollständiges Gitter. Dann gilt:

$$\lambda_{1,\infty}(L) \leq (detL)^{\frac{1}{m}}$$

□

Diese Grenze ist scharf; z.B. für das Gitter der ganzen Zahlen $\mathbb{Z}^m$ gilt: $\lambda_{1,\infty} = 1 = (det\mathbb{Z}^m)^{\frac{1}{m}}$

Definition: (**Hermite-Konstante**)

$$\gamma_n = \sup_{R(L)=n} \lambda_1(L)^2 \, (detL)^{-\frac{2}{n}}$$

□

Bislang sind nur die Hermite-Konstanten bis zur Dimension 8 bekannt. Diese sind:

$$\begin{array}{lccccccc} n = & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \gamma_n^n = & \frac{4}{3} & 2 & 4 & 8 & \frac{2^6}{3} & \frac{2^7}{2} & 2^8 \end{array}$$

Die ersten vier Konstanten wurden bereits von Korkine und Zolotareff im letzten Jahrhundert gefunden, die restlichen drei 1934 von Blichfeldt. Für $n > 8$ kennt man nur den folgenden

Satz:

$$\frac{n}{2\pi e}(1+o(1)) \leq \gamma_n \leq \frac{1.744n}{2\pi e}(1+o(1))$$

□

Die obere Schranke geht auf Kabatiansky und Levenshtein (1979) zurück. Die Schranke $\gamma_n \leq \frac{n}{\pi e}(1+o(1))$ ist bereits seit Blichfeldt bekannt.
Aus dem folgenden Satz, angewandt auf eine Vollkugel, ergibt sich die untere Schranke für die Hermite-Konstanten.

Satz: ([C71]) Sei $S \subset \mathbb{R}^n$ mit Jordanvolumen < 1. Dann gibt es ein vollständiges Gitter $L \subset \mathbb{R}^n$ mit Gitterdeterminante 1 und $L \cap (S \setminus \{0\}) = \emptyset$. □

2.2.1 Einfache Reduktionsbegriffe

Definition: Eine geordnete Basis $b_1, b_2 \in \mathbb{R}^n$ ist **reduziert** im Sinne von Gauß, wenn gilt:

$$\|b_1\| \leq \|b_2\| \leq \|b_1 - b_2\| \leq \|b_1 + b_2\|$$

□

Bemerkung: Sei b_1, b_2 eine reduzierte Basis und $\mu = \langle b_2, b_1\rangle / \langle b_1, b_1\rangle$. Dann gelten die folgenden Äquivalenzen:

$$\begin{aligned} \mu = 0 &\iff b_1, -b_2 \text{ ist reduziert} \\ \mu = \frac{1}{2} &\iff b_1, b_2 - b_1 \text{ ist reduziert} \\ \|b_1\| = \|b_2\| &\iff b_2, b_1 \text{ ist reduziert} \\ \|b_1\| < \|b_2\|, \quad 0 < \mu < \frac{1}{2} &\iff b_1, b_2 \text{ ist die einzige reduzierte Basis} \end{aligned}$$

Satz: Für jede reduzierte Basis gilt:

$$\|b_1\| = \lambda_1 \qquad \|b_2\| = \lambda_2$$

□

Folgender Algorithmus führt auf zwei Vektoren eine Reduktion im Sinne von Gauß durch:

Eingabe: $b_1, b_2 \in \mathbb{R}^n$ linear unabhängig

1. $\mu := \langle b_2, b_1 \rangle / \langle b_1, b_1 \rangle$
2. $b_2 := b_2 - \lceil \mu \rfloor b_1$

IF $\|b_1\| > \|b_2\|$ THEN
    vertausche b_1 und b_2
    gehe zu 1.
ENDIF

3. $b_2 := b_2 * sign(\langle b_1, b_2 \rangle)$

Ausgabe: b_1, b_2 reduziert im Sinne von Gauß

Satz: Der obige Algorithmus zur Reduktion im Sinne von Gauß bricht bei Eingabe von $b_1, b_2 \in \mathbb{R}^n$ nach höchstens $\lceil \log_{1+\sqrt{2}}(\|b_2\|/\lambda_2) \rfloor$ Iterationen ab. □

Eine Verallgemeinerung der Reduktion von Gauß auf mehr als zwei Vektoren ist die Gewichtsreduktion.

Definition: Eine geordnete Basis $b_1, \ldots, b_m$ ist **gewichtsreduziert**, wenn gilt:

$$|\langle b_i, b_j \rangle \|b_j\|^{-2}| \leq \frac{1}{2} \qquad 1 \leq i < j \leq m$$

$$\|b_1\| \leq \|b_2\| \leq \ldots \leq \|b_m\|$$

□

Nachfolgender Algorithmus transformiert eine Gitterbasis in eine gewichtsreduzierte Basis:

Eingabe: $b_1, \ldots, b_m \in \mathbb{R}^n$

```
F := true
WHILE F DO
    Ordne b_1, ..., b_m so, daß ||b_1|| ≤ ||b_2|| ≤ ...||b_m||
    F := false
    FOR i = 1, ..., m DO
        FOR j = 1, ..., i - 1 DO
            r := ⟨b_i, b_j⟩ / ⟨b_j, b_j⟩
            IF |r| > 1/2 THEN
                b_i := b_i - ⌈r⌋ b_j /* Reduktionsschritt */
                F := true
            ENDIF
        ENDFOR
    ENDFOR
ENDWHILE
```

Ausgabe: gewichtsreduzierte Basis $b_1, \ldots, b_m$

Korrektheit des Algorithmus: Es ist offensichtlich, daß nach Verlassen der äußeren Schleife eine gewichtsreduzierte Basis vorliegt. Da der Reduktionsschritt jeweils nur einen Vektor verkleinert und die restlichen unverändert läßt, bricht das Verfahren für ganzzahlige Eingaben auch ab. Eine Polynomialzeitschranke in der Anzahl der Eingabevektoren kann man jedoch nicht beweisen.

Definition: Eine geordnete Basis $b_1, \ldots, b_m$ ist **längenreduziert**, wenn gilt:

$$|\mu_{i,j}| \leq \frac{1}{2} \quad 1 \leq j < i \leq m$$

□

Für Basisvektoren b_i, b_k längenreduzierter Gitterbasen gilt

$$i < k \quad \Rightarrow \quad \|b_i\| \leq \|b_i + t b_k\| \quad \forall t \in \mathbb{Z}$$

Folgender Algorithmus führt auf einer Basis eine Längenreduktion durch:

Eingabe: $b_1, \ldots, b_m \in \mathbb{R}^n$
FOR $i = 1, \ldots, m$ DO
  FOR $j = i-1, \ldots, 1$ DO
    $b_i := b_i - \lceil \mu_{i,j} \rfloor \, b_j$ /* Reduktionsschritt */
  ENDFOR
ENDFOR
Ausgabe: längenreduzierte Basis $b_1, \ldots, b_m$

Korrektheit des Algorithmus: Der Reduktionsschritt bewirkt, daß

$$-\frac{1}{2} \;\leq\; \mu_{i,k}^{neu} \;:=\; \mu_{i,k}^{alt} - \lceil \mu_{i,j} \rfloor \, \mu_{j,k} \;\leq\; \frac{1}{2} \quad \text{für} \quad k = 1, \ldots, j$$

ist. Ferner bleiben die $\mu_{i,p}$ für $p > j$ und $\mu_{p,q}$ für $p < i$ unverändert. □
Offensichtlich besitzt der Algorithmus die Zeitkomplexität von $O(m^2 n)$. Da bei dem Vefahren der Längenreduktion keine Vertauschung der Vektoren erfolgt, verändern sich die Höhen und somit auch die Länge des Vektors b_1 nicht. Daher ist diese Reduktionseigenschaft so schwach, daß sie nicht allein verwendet wird; sie ist aber Teil der Reduktionseigenschaften, die in den nächsten Abschnitten vorgestellt werden.

2.2.2 LLL-reduzierte Gitterbasen

Der folgende Algorithmus geht auf Lenstra, Lenstra und Lovász zurück, die ihn 1982 in [LLL82] vorstellten. Zusätzlich zur Längenreduktion wird hierbei versucht, die Höhen im vorderen Bereich der Basis zu minimieren. Mit dem wählbaren Parameter δ steigen die Reduktionsgüte und die Schranken für die Laufzeit an. Wenn $\delta < 1$ ist, kann man für dieses Verfahren eine polynomiale Laufzeit beweisen.

Definition: Sei $\delta \in (\frac{1}{4}, 1]$. Eine geordnete Gitterbasis $b_1, \ldots, b_m$ ist mit δ **LLL-reduziert**, wenn gilt:

$$|\mu_{i,j}| \leq \frac{1}{2} \qquad 1 \leq i < j \leq m$$

$$\delta \|\hat{b}_{k-1}\|^2 \leq \|\pi_{k-1}(b_k)\|^2 = \|\hat{b}_k + \mu_{k,k-1}\hat{b}_{k-1}\|^2 \qquad k = 2, \ldots, m$$

□

Der folgende Algorithmus führt auf einer Gitterbasis eine LLL-Reduktion durch:

Eingabe: $b_1, \ldots, b_m \in \mathbb{Z}^n$, $\delta \in (\frac{1}{4}, 1]$

$k := 2$
berechne $\mu_{i,j} \quad 1 \leq i < j \leq m$ und $\|\hat{b}_k\|^2 \quad 1 \leq k \leq m$
`WHILE` $k \leq m$ `DO`
  längenreduziere b_k und korrigiere $\mu_{k,j} \quad j = 1, \ldots, k-1$
    $\mu_{i,k} \quad i = k+1, \ldots, m$
  `IF` $\delta \|\hat{b}_{k-1}\|^2 > \|\hat{b}_k\|^2 + \mu_{k,k-1}^2 \|\hat{b}_{k-1}\|^2$ `THEN`
    vertausche b_k und b_{k-1}
    korrigiere $\mu_{k,i}, \mu_{k-1,i}, \mu_{j,k-1}, \mu_{j,k}$ $i = 1, \ldots, k-1, \quad j = k, \ldots, m$
    berechne $\|\hat{b}_k\|^2$ und $\|\hat{b}_{k-1}\|^2$
    $k := \max(k-1, 2)$
  `ELSE`
    $k := k + 1$
  `ENDIF`
`ENDWHILE`

Ausgabe: $b_1, \ldots, b_m$ mit δ LLL-reduziert

Der Beweis der partiellen Korrektheit erfolgt durch Induktion über die Iterationen der While-Schleife. Es wird gezeigt, daß beim Eintritt in die Schleife und nach dem Austritt aus der Schleife die Vektoren $b_1, \ldots, b_{k-1}$ mit δ LLL-reduziert sind. Daraus folgt unmittelbar, daß nach Beendigung des Algorithmus die gesamte Basis mit δ LLL-reduziert ist.
Zu Beginn ist nichts zu zeigen, da $k = 2$ ist.
In der Schleife wird der Vektor b_k längenreduziert, und die betroffenen Gram-Schmidt-Koeffizienten werden neu berechnet. Daher ist die erste Reduktionsbedingung erfüllt. Ist die zweite ebenfalls erfüllt, wird k erhöht, da nun die Vektoren $b_1, \ldots, b_k$ LLL-reduziert sind. Sonst wird b_k mit b_{k-1} vertauscht, k erniedrigt, und die betroffenen Gram-Schmidt-Koeffizienten werden korrigiert. Da die Vektoren $b_1, \ldots, b_{k-2}$ im Schleifendurchlauf nicht verändert wurden, sind diese weiterhin LLL-reduziert.
Ferner muß gezeigt werden, daß der Algorithmus terminiert. Dazu betrachtet man die Anzahl der Iterationen.
Da bei jedem Schleifendurchlauf ohne Vertauschung die Stufe k inkrementiert wird, gilt für die Anzahl der Iterationen:

$$\#\text{ Iterationen } \leq m - 1 + 2 * \#\text{ Vertauschungen}$$

Um die Anzahl der Vertauschungen abzuschätzen, betrachten wir das Produkt der Determinantenquadrate folgender Teilgitter:

$$D_i := (det\ L(b_1, \ldots, b_i))^2 = \prod_{j=1}^{i} \|\hat{b}_j\|^2$$

$$D := \prod_{j=1}^{m-1} D_j$$

Da die Gitterbasis ganzzahlig ist, sind es auch die Teildeterminanten D_j; desweiteren sind sie positiv. Wird eine Vertauschung von b_k mit b_{k-1} durchgeführt, so führt dies zu der Veränderung $\|\hat{b}_{k-1}^{neu}\|^2 < \delta \|\hat{b}_{k-1}^{alt}\|^2$ und damit $D_{k-1}^{neu} < \delta D_{k-1}^{alt}$. Da sich die übrigen Teilgitter $L(b_1, \ldots, b_j) \quad j \neq k-1$ nicht verändern, folgt daraus $D^{neu} < \delta D^{alt}$. Daher gilt:

$$1 \leq D^{Ende} < \delta^{\#\text{ Vertauschungen}} D^{Start}$$

$$\text{bzw. } \#\text{ Vertauschungen } < \log_{\frac{1}{\delta}} D^{Start}$$

Im weiteren Verlauf dieses Abschnittes sei stets $M := \max_i \|b_i\|^2$ das maximale Normquadrat der Eingabevektoren und $\alpha := \left(\delta - \frac{1}{4}\right)^{-1}$. Die folgenden Sätze zur Beschränkung der Zahlengrößen und der Laufzeit finden sich in [LLL82]. Dort wurde für δ der Wert $\frac{3}{4}$ verwendet, die Beweise lassen sich jedoch entsprechend verallgemeinern (siehe auch [S91]).

Satz: Im LLL-Verfahren gilt bei Eintritt in Stufe k

$$\|b_i^{neu}\|^2 \leq \frac{m+3}{4} M \qquad i = 1, \ldots, m$$

$$|\mu_{i,j}|^2 \leq \frac{m+3}{4} M \alpha^{j-1} \qquad j < k, \quad i = 1, \ldots, m$$

□

Satz: Im Verlauf von Stufe k gilt stets

$$|\mu_{k,j}| \leq \frac{m+3}{4} M \left(\frac{9\alpha}{4}\right)^{k-1}$$

□

Im Gegensatz dazu können die Gram-Schmidt-Koeffizienten $\mu_{i,j}$ für $j > k$ sehr groß werden; für sie gilt nur die folgende Schranke:

$$|\mu_{i,j}|^2 \leq \frac{m+3}{4} M^j$$

Das obige LLL-Verfahren mit iterativer Orthogonalisierung vermeidet die Berechnung dieser Gram-Schmidt-Koeffizienten. Auf Stufe k werden nur

Algorithmus LLL-Reduktion mit iterativer Orthogonalisierung

Eingabe: $b_1, \ldots, b_m \in \mathbb{Z}^n$, $\delta \in (\frac{1}{4}, 1]$

$k := 2$

$c_1 := \|b_1\|^2$

Bei Eintritt in Stufe k liegen vor:

$c_i = \|\hat{b}_i\|^2$ für $i = 1, \ldots, k-1$

$\mu_{i,j}$ für $1 \leq j < i < k$

`WHILE` $k \leq m$ `DO`

  `IF` $k = 2$ `THEN`

    $c_1 := \|b_1\|^2$

  `ENDIF`

  `FOR` $j = 1, \ldots, k-1$ `DO`

    $\mu_{k,j} := \langle b_k, b_j \rangle - \sum_{i=1}^{j-1} \mu_{j,i}\mu_{k,i}c_i/c_j$

  `ENDFOR`

  $c_k := \langle b_k, b_k \rangle - \sum_{j=1}^{k-1} \mu_{k,j}^2 c_j$

  Führe Längenreduktion von b_k durch und korrigiere $\mu_{k,1}, \ldots, \mu_{k,k-1}$

  `IF` $\delta\|\hat{b}_{k-1}\|^2 > \|\hat{b}_k\|^2 + \mu_{k,k-1}^2\|\hat{b}_{k-1}\|^2$ `THEN`

    vertausche b_k und b_{k-1}

    $k := \max(k-1, 2)$

  `ELSE`

    $k := k + 1$

  `ENDIF`

`ENDWHILE`

Ausgabe: $b_1, \ldots, b_m$ mit δ LLL-reduziert

---

die $\mu_{i,j}$ mit $i \leq k$ benötigt. Die $\mu_{k,j}$ werden neu berechnet, die $\mu_{i,j}$ mit $i < k$ stammen aus vorhergehenden Stufen.

Satz: Bei Eingabe von $b_1, \ldots, b_m \in \mathbb{Z}^n$ führt der LLL-Algorithmus mit iterativer Orthogonalisierung höchstens

$$O(m^2 n(1 + m \log M)) \text{ arithmetische Operationen aus}$$

auf

$$\text{Zahlen der Größe } O\left(m \log\left(M + \frac{3\sqrt{\alpha}}{2}\right)\right).$$

□

Satz: Sei $b_1, \ldots, b_m$ mit δ LLL-reduziert. Dann gilt

$$\|\hat{b}_i\|^2 \leq \alpha^{j-i}\|\hat{b}_j\|^2 \qquad 1 \leq i \leq j \leq m$$

□

Satz: Die Basis $b_1, \ldots, b_m$ sei mit δ LLL-reduziert. Dann gilt für $j = 1, \ldots, m$:

$$\alpha^{1-j} \leq \frac{\|\hat{b}_j\|^2}{\lambda_j^2} \leq \frac{\|b_j\|^2}{\lambda_j^2} \leq \alpha^{m-1}$$

□

Die vorhergehenden Sätze geben theoretische Schranken für Laufzeit des LLL-Algorithmus und Orthogonalitätsdefekt der LLL-reduzierten Gitterbasen an, die in der Praxis zumeist unterboten werden. Dennoch reicht die Reduktionsgüte in vielen Fällen, insbesondere bei höheren Dimensionen, nicht aus. Ein schon aus dem letzten Jahrhundert stammender Reduktionsbegriff geht auf Hermite, Korkine und Zolotareff zurück.

2.2.3 Korkine-Zolotareff-reduzierte Gitterbasen

Definition: Eine geordnete Gitterbasis $b_1, \ldots, b_m$ ist **Korkine-Zolotareff-reduziert**, wenn gilt:

$$|\mu_{i,j}| \leq \frac{1}{2} \qquad 1 \leq i < j \leq m$$
$$\|\hat{b}_i\| = \lambda_1(\pi_i(L)) \qquad i = 1, \ldots, m$$

□

Die erste Bedingung gewährleistet die Längenreduziertheit. Die zweite besagt, daß jeder Basisvektor b_i in der Projektion auf den von den Vektoren $b_1, \ldots, b_{i-1}$ aufgespannten Unterraum der kürzeste Gittervektor ist. Insbesondere ist b_1 der kürzeste nichttriviale Gittervektor, was sich auch aus dem folgenden Satz ergibt ([LLS90]):

Satz: Für jede Korkine-Zolotareff-reduzierte Basis $b_1, \ldots, b_m$ eines Gitters L gilt:

$$\frac{4}{i+3} \leq \frac{\|b_i\|}{\lambda_i(L)} \leq \frac{i+3}{4} \qquad i = 1, \ldots, m$$

□

Die bislang bekannten Verfahren zur Korkine-Zolotareff-Reduktion haben eine in der Dimension exponentielle Laufzeit, so daß für Gitterbasen mit Dimensionen deutlich größer als 30 die Reduktion nicht mehr in vernünftiger Zeit durchführbar ist. Gesucht wurde daher ein Reduktionsbegriff, der zwar stärker als der der LLL-Reduktion ist, aber schwächer als die Korkine-Zolotareff-Reduktion. Ein solcher wurde von Schnorr [S87] eingeführt.

2.2.4 Block-Korkine-Zolotareff-reduzierte Gitterbasen

Definition: Eine geordnete Basis $b_1, \ldots, b_m$ ist **Block-Korkine-Zolotareff-reduziert** mit Blockweite $\beta \in [2, m] \cap \mathbb{N}$ und $\delta \in \left(\frac{1}{4}, 1\right]$, falls

$$|\mu_{i,j}| \leq \frac{1}{2} \qquad 1 \leq i < j \leq m$$
$$\delta \|\hat{b}_i\|^2 \leq \lambda_1 \left(L\left(\pi_i(b_i), \ldots, \pi_i(b_{\min(i+\beta-1,m)})\right)\right) \qquad i = 1, \ldots, m$$

□

Auch hierbei sichert die erste Bedingung die Längenreduziertheit. Die zweite besagt, daß jeder Basisvektor die bis auf einen konstanten Faktor kleinste Höhe aus einem Block von β Vektoren besitzt. Für $\beta = m$ und $\delta = 1$ erhält man eine Korkine-Zolotareff-reduzierte Basis.

Satz: Eine Basis $b_1, \ldots, b_m$ ist genau dann BKZ-reduziert mit Blockweite $\beta = 2$ und $\delta \in \left(\frac{1}{3}, 1\right]$, wenn sie mit δ LLL-reduziert ist. □

Definition:

$$\alpha_\beta := \sup \frac{\|b_1\|^2}{\|\hat{b}_\beta\|^2}$$

wobei das Supremum über alle KZ-reduzierte Basen $b_1, \ldots, b_\beta$ gebildet wird.

Satz: [S92]

$$\begin{aligned} \alpha_m &\leq \gamma_m^{\frac{m}{m-1}} \gamma_{m-1}^{\frac{1}{m-2}} \cdots \gamma_2^{\frac{1}{1}} \\ &\lesssim \left(\frac{m}{e\pi}\right)^{\frac{m}{m-1} + \frac{m-1}{m-2} + \ldots + \frac{1}{1}} \\ &\lesssim \left(\frac{m}{e\pi}\right)^{1+\ln m} \end{aligned}$$

□

Satz: [S87],[S92] Sei $b_1, \ldots, b_m$ eine β-BKZ-reduzierte Basis mit $\delta = 1$ und $m = 1 \mod (\beta - 1)$. Dann gilt:

$$\|b_1\|^2 \leq \alpha_\beta^{\frac{m-1}{\beta-1}-1} \lambda_1^2$$

□

3. Rucksackprobleme

Das Rucksackproblem hat folgende Gestalt: Gegeben sei $(n, s, a_1, \ldots, a_n) \in \mathbb{N}^{n+2}$. Entscheide, ob ein Vektor $(x_1, \ldots, x_n) \in \{0,1\}^n$ existiert mit

$$\sum_{i=1}^{n} x_i a_i = s.$$

Dabei bezeichne $d := \frac{n}{\log_2 \max a_i}$ die Dichte des Problems. Das Rucksackproblem läßt sich in ein Gittervektorproblem transformieren: Gegeben seien $(n, s, a_1, \ldots, a_n) \in \mathbb{N}^{n+2}$. Entscheide, ob das von den Spaltenvektoren der Matrix B_1

$$B_1 = \begin{array}{c} \begin{array}{cccccc} b_0 & b_1 & b_2 & \cdots & b_{n-1} & b_n \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ 1 & 2 & 0 & \cdots & \cdots & 0 \\ 1 & 0 & 2 & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 1 & \vdots & & \ddots & 2 & 0 \\ 1 & 0 & \cdots & \cdots & 0 & 2 \\ cs & ca_1 & ca_2 & \cdots & ca_{n-1} & ca_n \end{pmatrix} \end{array} \in \mathbb{Z}^{(n+1)\times(n+2)}$$

erzeugte Gitter einen Vektor $z = (z_0, \ldots, z_{n+1})$ enthält mit $\|z\|_\infty = 1$. Dabei kann $1 \neq c \in \mathbb{N}$ frei gewählt werden. Für jeden Gittervektor z mit $\|z\|_\infty = 1$ gilt:

$$\begin{array}{lll} (1) & z_0 = \pm 1 & \\ (2) & |z_i| = 1 & i = 1, \ldots, n \\ (3) & z_{n+1} = 0 & \end{array}$$

zu (1): Für $z_0 = 0$ ist z eine Linearkombination der Vektoren $b_1, \ldots, b_n$. Da deren Komponenten alle gerade oder durch c teilbar sind, ist mindestens ein z_i betragsmäßig größer eins.
zu (2): Da der Vektor b_0 genau einmal in z eingeht, müssen die Komponenten $z_1, \ldots, z_n$ ungerade und damit ± 1 sein.
zu (3): z_{n+1} ist durch die Konstruktion der Gitterbasis stets durch c teilbar. Aus $\|z\|_\infty = 1$ und $c > 1$ folgt damit die Behauptung.

Satz: Alle Vektoren z mit $\|z\|_\infty = 1$ sind Zeugen des Gitterproblems. Aus einem solchen Vektor z läßt sich eine Lösung des Rucksackproblems folgendermaßen ablesen:

$$x_i = \begin{cases} 1 & z_i = -z_0 \\ 0 & z_i = z_0 \end{cases}$$

Beweis: gegeben sei $z = (z_0, \ldots, z_{n+1}) \in \{\pm 1\}^{n+1} \times \{0\} \in L(b_0, \ldots, b_n)$, d.h. $z_j = \sum_{i=0}^{n} w_i b_{j,i} \quad j = 1, \ldots, n$. O.B.d.A. sei $z_0 = -1$.

i. $z_0 = -1 \quad \Rightarrow \quad w_0 = -1$

ii. $\{-1, 1\} \ni z_j = \sum_{i=0}^{n} w_i b_{j,i} = -1 + \sum_{i=1}^{n} w_i b_{j,i} = -1 + 2w_j \quad \Rightarrow$ $w_j \in \{0, 1\} \qquad j = 1, \ldots, n$

iii. $0 = z_{n+1} = \sum_{i=0}^{n} w_i b_{n+1,i} = -c\,(s - \sum_{i=1}^{n} w_i a_i) \quad \overset{c \neq 0}{\Rightarrow} \quad s = \sum_{i=1}^{n} a_i$

□

Verwendet man statt der Maximumnorm die euklidische Norm, dann liefert ein kürzester Gittervektor nicht immer eine Lösung des Rucksackproblems. Lagarias und Odlyzko bewiesen in [LO85]:

Satz: Für hinreichend große n liefert für fast alle Rucksackprobleme mit Dichte $d < 0.645$ ein kürzester Gittervektor des von B_2 erzeugten Gitters eine Lösung. Hierbei sind die Gewichte a_i unabhängig voneinander und über das Intervall $[1, 2^{n/d} - 1]$ gleichverteilt. □

Die Matrix B_2 wird mit $N = n^2$ aus den Gewichten a_i und der Summe s wie folgt erzeugt:

$$B_2 = \begin{array}{c} \begin{matrix} b_0 & b_1 & b_2 & \cdots & b_{n-1} & b_n \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 & \cdots & \cdots & 0 \\ 0 & 0 & 1 & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \vdots & & \ddots & 1 & 0 \\ 0 & 0 & \cdots & \cdots & 0 & 1 \\ Ns & -Na_1 & -Na_2 & \cdots & -Na_{n-1} & -Na_n \end{pmatrix} \end{array} \in \mathbb{Z}^{(n+1)\times(n+1)}$$

Mittels der Basismatrizen

$$B_3 = \begin{array}{c} \begin{matrix} b_0 & b_1 & b_2 & \cdots & b_{n-1} & b_n \end{matrix} \\ \begin{pmatrix} \frac{1}{2} & 1 & 0 & \cdots & \cdots & 0 \\ \vdots & 0 & 1 & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & & \ddots & 1 & 0 \\ \frac{1}{2} & 0 & \cdots & \cdots & 0 & 1 \\ Ns & -Na_1 & -Na_2 & \cdots & -Na_{n-1} & -Na_n \end{pmatrix} \end{array} \in \mathbb{Q}^{(n+1)\times(n+1)}$$

bzw.

$$B_4 = \begin{array}{c} \begin{matrix} b_0 & b_1 & \cdots & b_{n-1} & b_n \end{matrix} \\ \begin{pmatrix} n+1 & -1 & \cdots & \cdots & -1 \\ -1 & n+1 & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ -1 & \vdots & \ddots & n+1 & -1 \\ -1 & -1 & \cdots & -1 & n+1 \\ Ns & -Na_1 & \cdots & -Na_{n-1} & -Na_n \end{pmatrix} \end{array} \in \mathbb{Z}^{(n+1)\times(n+2)}$$

wurde diese Schranke in [CJLOSS92] von $d < 0.645$ auf $d < 0.9408$ verschärft.
Ist die Anzahl der Gewichte, die in die Summe eingehen, bekannt, so kann die Basismatrix um folgende Zeile

$$\begin{array}{c} \begin{matrix} b_0 & b_1 & b_2 & \cdots & b_{n-1} & b_n \end{matrix} \\ \begin{pmatrix} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ cq & c & c & \cdots & c & c \end{pmatrix} \end{array}$$

mit $q = \sum_{i=1}^{n} x_i$ und einer hinreichend großen Konstanten $c \in \mathbb{N}$ ergänzt werden.

Eine Verallgemeinerung der Rucksackprobleme auf k Gleichungen wurde von Hörner [H94] vorgenommen. Dieses Problem hat die folgende Form:
Gegeben seien $u_{hi} \in \mathbb{Z}, \quad A_{hi} \in \mathbb{N}, \quad a_{hi} \in [u_{hi}, u_{hi} + A_{hi} - 1] \cap \mathbb{Z}, \quad s_h \in \mathbb{Z}, \quad i = 1, \ldots, n \quad h = 1, \ldots, k$. Dabei sind alle Gewichte a_{hi} unabhängig voneinander und in ihren Intervallen gleichverteilt.
Gesucht ist $e = (e_1, \ldots, e_n) \in \{0,1\}^n \setminus 0^n$ mit

$$s_h = \sum_{i=1}^{n} e_i a_{hi} \quad 1 \le h \le k$$

Die zugehörige Dichte des $k-$fachen Rucksackproblems ist eine Funktion der Intervallängen:

Definition: Die **Dichte** d des oben beschriebenen $k-$fachen Rucksackproblems ist

$$d := \frac{n}{\log_2 \min_{1 \le i \le n} \prod_{h=1}^{k} A_{hi}}$$

□

Unter Verwendung der folgenden Basismatrix B_5

$$B_5 = \begin{array}{c} \begin{matrix} b_0 & b_1 & b_2 & \cdots & b_{n-1} & b_n & b_{n+1} \end{matrix} \\ \begin{pmatrix} \frac{1}{n} & 1 & 0 & \cdots & \cdots & 0 & 0 \\ \vdots & 0 & 1 & \ddots & & \vdots & \vdots \\ \vdots & \vdots & 0 & \ddots & 0 & \vdots & \vdots \\ \vdots & \vdots & & \ddots & 1 & 0 & \vdots \\ \frac{1}{n} & 0 & \cdots & \cdots & 0 & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 1 \\ 0 & Na_{1,1} & Na_{1,2} & \cdots & Na_{1,n-1} & Na_{1,n} & Ns_1 \\ 0 & Na_{2,1} & Na_{2,2} & \cdots & Na_{2,n-1} & Na_{2,n} & Ns_2 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & Na_{k,1} & Na_{k,2} & \cdots & Na_{k,n-1} & Na_{k,n} & Ns_k \end{pmatrix} \end{array} \in \mathbb{Q}^{(n+k+1)\times(n+2)}$$

gilt:

Satz: [H94] Das $k-$fache Rucksackproblem wird für hinreichend große n und Intervalle der Dichte $d < 1/c_0(\frac{q}{n})$ für fast alle $((a_{11},\ldots,a_{1n}),\ \ldots,\ (a_{k1},\ldots,a_{kn}))$ durch Finden eines kürzesten nichttrivialen Gittervektors des von den Spaltenvektoren der Matrix B_5 erzeugten Gitters gelöst. □
Hierbei ist $c_0(q/n)$ eine Funktion von q und n, für die gilt:

$$\left|\left\{x \in \mathbb{Z}^n + \left(\frac{1}{n},\ldots,\frac{1}{n}\right)\mathbb{Z} \quad : \quad \|x\| \le \sqrt{\frac{q}{n}(n-q)+1}\right\}\right| \le 2^{2\,c_0\left(\frac{q}{n}\right)n+o(n)}$$

Für den Fall $k = 1$ und $q = n/2$ ergibt sich die aus [CJLOSS92] bekannte Dichte von $0.9408\ldots$. (Eine Tabelle von Funktionswerten von $c_0(q/n)$ an geeigneten Stützstellen ist in [H94] gegeben.)

4. Parallelrechner

Da der Bedarf an Rechenleistung ständig ansteigt, und die Erhöhung der Taktfrequenz durch die physikalischen Gegebenheiten immer schwieriger und teurer wird, werden in den letzten Jahren in stärkerem Maße parallele Rechnerarchitekturen erforscht und entwickelt.
Die Hauptziele, die durch Parallelrechner verwirklicht werden sollen, sind:
1. Steigerung der Leistung
2. Fehlertoleranz bzw. Ausfallsicherheit
3. Programmierfreundlichkeit

Die Entwicklung von Maschinen zur Steigerung der Programmierfreundlichkeit führt meist dazu, daß die entwickelten Maschinen für spezielle Problemgebiete konzipiert sind. Auch werden auf diesen Maschinen eigene Programmiersprachen verwendet, die dem Typ der Maschine und des Problems angepaßt sind. Die Art der verwendeten Parallelität ist jedoch stark problemabhängig. Im Folgenden befassen wir uns nur noch mit Computern, die frei programmierbar und damit nicht auf ein bestimmtes Anwendungsgebiet festgelegt sind, den Universalrechnern.
Die Ausfallsicherheit eines Rechnersystems kann zum einen durch Verdopplung von Komponenten erreicht werden, jedoch liegt hier keine für den Benutzer erkennbare Parallelität vor. Die zweite Ausführung einer Komponente übernimmt erst dann Aufgaben, wenn die erste ausgefallen ist. Da hierbei Teile des Systems doppelt existieren, wird dieses Verfahren im wesentlichen nur bei verhältnismäßig billigen Komponenten angewendet (z.B. Wasserkühlung). Einzig, wenn der Ausfall eines Systems hohen Schaden an Menschen und Material bewirken könnte, werden auch gesamte Computersysteme gedoppelt.
Eine zweite Möglichkeit zur Steigerung der Fehlertoleranz besteht in der Vervielfachung von gleichartigen Komponenten, die im normalen Betrieb parallel arbeiten und sich gegenseitig überwachen. Fällt eine Komponente wegen eines Defektes aus, wird dies bemerkt und sie erhält keine Teilaufgaben mehr zugeteilt. Die Leistung sinkt in diesem Fall, das System kann aber weiterarbeiten. Ferner wird im Normalbetrieb eine Leistungssteigerung erzielt, auf deren Art weiter unten noch eingegangen wird.
Die Steigerung der Verarbeitungsleistung führte schon bei seriellen Computern zu einer speziellen Art der Parallelverarbeitung, indem der teure

Hauptprozessor von bestimmten Aufgaben durch meist billigere Hilfsprozessoren entbunden wurde, oder sogar durch eigens dafür angeschaffte Rechensysteme. Zuerst geschah dies bei der Ein-/Ausgabe, da die teilweise mechanisch arbeitende Peripherie um den Faktor 1000 langsamer als der Prozessor war und diesen bremste. Dieser Trend wurde beibehalten, so daß heute fast alle Computersysteme über eine Reihe von Spezialprozessoren verfügen, die die Kommunikation zur Peripherie übernehmen.
Um den eigentlichen Prozessor zu beschleunigen, bedient man sich einer Parallelität, die durch Pipelining, Nebenläufigkeit oder eine Mischform erreicht wird.
Beim Pipelining wird die Verarbeitung in möglichst gleichlange Teilschritte zerlegt. Jeder Teilschritt wird durch ein Schaltnetz realisiert, das seine Signale nur an nachfolgende Schaltnetze weitergibt. Zwischen den Schaltnetzen werden die Zwischenergebnisse gepuffert. Der Takt, mit dem die Pipeline betrieben wird, bestimmt sich aus der langsamsten Stufe. In jedem Taktzyklus nimmt sich jede Pipelinestufe die Ergebnisse des vorherigen Schaltnetzes, führt seine Bearbeitung durch und legt sein Ergebnis im nächsten Pufferspeicher ab. Dadurch kann die Pipeline in jedem Taktzyklus ein Operandenpaar aufnehmen und ein Ergebnis abgeben. Der Vorteil dieses Verfahrens ist, daß mit einem geringen Hardwaremehraufwand die vorhandene Hardware besser genutzt wird. Da durch die Pufferung der Zwischenergebnisse sowie eine geeignete Realisierung der Schaltnetze die gesamte Verarbeitung einer Operation länger dauert, arbeitet die Pipeline nur dann effizient, wenn sie über längere Phasen im eingeschwungenen Zustand arbeitet, d.h. alle Schaltnetze mit Teilaufgaben beschäftigt sind. Nach Störungen der Pipelineverarbeitung, die eine Leerung verursachen, erfolgt stets ein Zeitverlust durch die neue Anlaufzeit, in der keine Ergebnisse produziert werden.
Die Nebenläufigkeit erreicht man durch Vervielfachung von Funktionseinheiten, wobei dies auf einem Prozessor erfolgen kann oder vollständige Prozessoren mehrfach verwendet werden. Der Vorteil letzterer Methode ist die Modularität, die das entstandene System aufweist. Da jede einzelne Funktionseinheit verhältnismäßig klein ist, kann sie leichter entwickelt und besser optimiert werden. Da die Funktionseinheiten nicht notwendigerweise alle auf einem Chip integriert werden müssen, kann man meistens eine hohe Anzahl paralleler Elemente realisieren. Der gravierende Nachteil ist jedoch, daß die Teilaufgaben den einzelnen Funktionseinheiten zugeteilt werden müssen, d.h. es entsteht Verwaltungsaufwand. Dieser kann bei ungeeignetem Design der einzelnen Funktionseinheiten größer als der eigentliche Rechenaufwand werden; er ist jedoch in keinem Falle vernachlässigbar.

Durch die verschiedenen Möglichkeiten, parallele Systeme zu realisieren, sind in der Vergangenheit viele Computer unterschiedlicher Architekturen entstanden. Daher war es notwendig, diese zu klassifizieren und zu bewerten.

4.1 Klassifikationen

4.1.1 Die Klassifikation nach Flynn

Das wohl bekannteste Klassifikationsschema stammt von Flynn [F72] aus dem Jahre 1972. Es ist sehr einfach und abstrakt. Ein Rechner wird dabei durch zwei Eigenschaften charakterisiert:

1. Einfache oder mehrfache Befehlsströme (**S**ingle **I**nstruction Stream oder **M**ultiple **I**nstruction Stream): hier wird bestimmt, ob ein Rechner mehrere verschiedene Befehle gleichzeitig ausführen kann. Eine Vorbedingung für mehrfache Befehlsströme ist das mehrfache Vorkommen von Leitwerken.
2. Einfache oder mehrfache Datenströme (**S**ingle **D**ata Stream oder **M**ultiple **D**ata Stream): hier wird bestimmt, ob ein Befehl nur auf einem Datenwort wirkt oder mehrere Datenwörter gleichzeitig bearbeitet werden können.

Die klassische von-Neumann-Architektur erscheint somit in der Klasse `SISD`. Durch den hohen Abstraktionsgrad des Schemas gehören unterschiedliche Architekturen wie Feld- und Assoziativrechner einer Klasse (`SIMD`) an. Weiterhin ist die Einordnung von Vektorrechnern sowohl in `SISD` als auch in `SIMD` möglich, da ein Befehl zwar auf viele Datenworte wirkt, aber dies nicht gleichzeitig geschieht. Durch die Systematik ist eine leere Klasse `MISD` entstanden. Diese Gründe veranlaßten Autoren, Änderungen an dieser Klassifikation vorzunehmen oder gänzlich neue zu kreieren. So erfolgte 1973 von Higbie eine Verfeinerung der Klasse `SIMD` in die Unterklassen Feldrechner, Assoziativrechner, assoziative Feldrechner und Orthogonalrechner. Kuck ändert 1978 die Eigenschaft Data Stream in Execution Stream und führte eine dritte Komponente „scalar bzw. array“ ein, sodaß z.B. der Feldrechner ILLIAC IV der Klasse „Single Instruction scalar, single execution array“ angehört.

4.1.2 Klassifizierung nach Feng

Dieses Schema betrachtet speziell Parallelrechner mit Nebenläufigkeit. Dabei erhält der Computer eine Charakterisierung nach den Komponenten

1. Datenwortlänge und
2. Bitscheibenlänge.

Dieses Wertepaar gibt den maximal erreichbaren Wert an Parallelität an. Hierbei bezeichnet die Bitscheibenlänge die Anzahl der parallel verarbeitbaren Datenworte. Die Klassifizierung nach Feng bezieht sich jedoch nur auf homogene Prozessoranordnungen, die unter einem Steuerwerk arbeiten; Felder selbständig arbeitender Prozessoren werden nicht erfaßt.

4.1.3 Die OS-Matrix nach Giloi

Giloi [G93] teilt die Rechnerarchitekturen nach den Kriterien
1. Operationsprinzip
2. Struktur

ein. Die Menge der Operationsprinzipien unterteilt sich in
1. das von-Neumann-Operationsprinzip
2. die Operationsprinzipien der Programmparallelität
3. die Operationsprinzipien der Datenparallelität

Dabei wird die Datenparallelität auch als explizite Parallelität bezeichnet, da sie unmittelbar ersichtlich ist und nicht aus der Programmstruktur ermittelt werden muß. Bezüglich der Struktur unterscheidet er die folgenden Eigenschaften der Hardware
1. Einprozessorsysteme
2. Arrays gleichartiger, universeller Recheneinheiten
3. Pipelines aus unterschiedlichen, spezialisierten Recheneinheiten
4. systolische Arrays
5. Multiprozessor-Systeme
   Bei diesen unterscheidet man noch zwischen gemeinsamem und verteiltem Speicher, sowie der Art der Kommunikation der Prozessoren untereinander; sie kann speichergekoppelt oder nachrichtenorientiert sein.

4.1.4 Erlanger Klassifikationsschema (ECS)

Das ECS [BH83] ist das bislang umfangreichste und genaueste Schema. Es geht von der Vorstellung aus, daß ein Rechner aus k Leitwerken besteht, die d Rechenwerke steuern; diese operieren auf w-Bit-Zahlen. Dieses Zahlentupel (k, d, w) bezeichnet somit die durch Nebenläufigkeit erreichbare Paralleliät. Ferner kann auf jeder Stufe durch Pipelining eine zusätzliche Parallelität $(*k', *d', *w')$ realisiert werden, so daß ein Rechner mit

$$t_{\text{Rechner}} = (k * k', d * d', w * w')$$

spezifiziert wird. Das Produkt der drei Komponenten liefert den maximalen Parallelitätsgrad, der mit dem Produkt der Datenwortlänge und Bitscheibenlänge aus der Feng'schen Klassifikation vergleichbar ist.
Zur Vereinfachung kann bei Wegfall des Pipelinings auf einem Niveau das zugehörige $*1$ entfallen, fehlt die Nebenläufigkeit, kann die führende 1 entfallen.
Will man zusammengesetzte Architekturen beschreiben, so bietet das ECS die Operatoren '$*$' und '$+$' an, mit denen die Klassifikationen der verschiedenen parallel arbeitenden Komponenten verknüpft werden können. Dabei findet der '$*$'-Operator Anwendung, wenn die einzelnen Komponenten ihre Teile an der Verarbeitung nacheinander wie in einer Pipeline abarbeiten. Als Beispiel sei ein Feldrechner mit Steuerrechner genannt; beide arbeiten unabhängig voneinander, aber an unterschiedlichen Aufgaben.

Der Operator '+' zeigt an, daß die verknüpften Komponenten alternativ oder parallel an der gleichen Gesamtaufgabe beteiligt sein können. Dies tritt z.B. bei den Vektorrechnern auf, die über Pipelines für Vektor- und Skalaroperationen verfügen, wobei diese parallel arbeiten können.
Ein weiterer Operator '$\vee$' zeigt an, daß die Hardware verschiedene Betriebsmodi aufweisen kann, d.h. dynamisch konfiguriert werden kann. Zu einem bestimmten Zeitpunkt liegt aber genau eine Betriebsart vor. Die Anzahl der verschiedenen Betriebsarten bezeichnet man als Flexibilität.
Der Wert '0' in einer Komponente zeigt an, daß die zugehörige Komponente nicht vorhanden ist.
Beispiel:

$$
\begin{aligned}
t_{\text{CRAY-1}} = (1,0,0) \quad * \quad & [(0,*1,24*2) \quad + \quad (0,*1,24*6) \\
+ \quad & (0,*1,64*3) \quad + \quad (0,*1,64*2) \\
+ \quad & (0,*1,64*1) \quad + \quad (0,*1,64*3) \\
+ \quad & (0,*1,64*2) \quad + \quad (0,*1,64*3) \\
+ \quad & (0,*1,64*4) \quad + \quad (0,*1,64*6) \\
+ \quad & (0,*1,64*7) \quad + \quad (0,*1,64*14)]
\end{aligned}
$$

Es existiert ein Leitwerk, das 12 Funktionseinheiten steuert; die ersten zwei Funktionseinheiten sind die Adreßrechenwerke mit der Datenwortbreite 24 bit, anschließend folgen die Skalar- und Vektorpipelines mit der Datenwortbreite 64 bit, aber unterschiedlichen Pipeline-Graden. Zu beachten ist, daß dies nur die Klassifikation des Vektorprozessors der Cray-1 ist; ein vorhandener Host-Rechner mit I/O-Prozessoren besitzt zusätzlich eine eigene Klassifikation, die mit der obigen konkateniert wird.

Anhand einer gegebenen Klassifikation im ECS ist somit unmittelbar erkennbar, wie die Parallelität in der Hardware verteilt ist. Es ist das einzige Schema, das einen Verbund von unterschiedlichen Computern geeignet beschreiben kann. Die Bewertungen unterschiedlicher Rechnersysteme sind jedoch nur dann vergleichbar, wenn sie in der gleichen Technologie realisiert sind.

4.2 Vektorrechner

Wie im vorigen Abschnitt ersichtlich wurde, sind in der Vergangenheit viele verschiedene Arten Parallelrechner entwickelt worden. Bei den meisten Typen handelte es sich jedoch um Forschungsprojekte, die nie verkauft wurden. Den Computermarkt haben bislang Universalrechner in Form von Mainframes, Minicomputern, Workstations und PC's beherrscht. Lediglich zur Lösung großer numerischer Probleme finden seit etwa 20 Jahren die Vektorrechner Verwendung. Diese Computer lassen sich wiederum in verschiedene Arten unterteilen:

1. Vektorrechner
   Sie bestehen aus den schnellsten Rechenwerken, die zu dieser Zeit gebaut werden können; anschließend wird die Peripherie (Hauptspeicher, I/O-Prozessoren, u.ä.) mit der notwendigen Geschwindigkeit aufgebaut, so daß der schnelle Prozessor die nötigen Daten zur Bearbeitung erhalten kann. Da sie stets das aktuell technisch Machbare darstellen, sind sie sowohl in der Anschaffung als auch in den laufenden Kosten teuer. Zumeist sind sie in der sogenannten ECL-Technologie gefertigt (emitter-coupled-logic). Mit dieser Logik konnten bislang die schnellsten Schaltzeiten realisiert werden. Da die Schaltung stromgesteuert ist, bedingt dies einen hohen Stromverbrauch und Kühlungsbedarf.
2. Mini-Supercomputer
   Sie ähneln den obigen Vektorrechnern, sind jedoch aus preiswerteren, langsameren Komponenten gebaut. Daher besitzen sie deutlich weniger Leistung, weisen aber ein besseres Preis-/Leistungsverhältnis auf.
3. Vektor-Koprozessoren
   Hierbei werden in bestehende Universalrechnersysteme Vektorprozessoren integriert. Dies läßt sich durch zwei Methoden realisieren:
   a. der Skalarprozessor erhält ein vektorielles Rechenwerk (z.B. IBM 3090/VF);
   b. das Gesamtsystem wird um einen eigenständig arbeitenden Vektorprozessor ergänzt (z.B. Sperry ISP).

   Diese Systeme haben typischerweise keine allzu hohe Geschwindigkeit, da die Peripherie dieser Prozessoren die zu verarbeitenden Daten nicht schnell genug liefern kann.

Die Varianten Mini-Supercomputer und Vektor-Koprozessoren haben keine große Bedeutung erlangt, da sie nur für Benutzer mit geringem Rechenbedarf geeignet sind.

Vektorrechner zeichnen sich durch eine hohe Verarbeitungsgeschwindigkeit in der Größenordnung von 1 Milliarde Ergebnissen pro Sekunde aus; sie erreichen diese durch Pipelining der arithmetischen Operationen und eine sehr kurze Zykluszeit (im Bereich von 2–20 $*10^{-9}$ sec). Dabei können nur gleichartige Operationen auf Elementen ausgeführt werden.

Sie verfügen über große Haupt- und Hintergrundspeicher mit Kapazitäten im Bereich von mehreren Gigabytes. Weitere Leistungssteigerungen wurden durch die Verwendung von mehreren Rechenpipes, die parallel arbeiten können, erreicht.

In der Art der Versorgung der Rechenpipes ergibt sich die Unterteilung der Vektorrechner nach

1. Speichermaschinen (z.B. CDC Cyber 205)
2. Registermaschinen (z.B. Cray-1)

Erstere versorgen die Rechenpipes unmittelbar aus dem Hauptspeicher und legen die Ergebnisse auch wieder dort ab. Da der Datentransfer vom Hauptspeicher zu den Pipes und zurück sich immer mehr als Flaschenhals

eines Computersystems herausgestellt hat, wird diese Art seit längerer Zeit nicht mehr hergestellt.
Registermaschinen besitzen einen schnellen Zwischenspeicher (Cache), die Vektorregister. Diese haben eine bestimmte Länge, die auch die maximale Anzahl an Operanden angibt, die mit einem Vektorbefehl verarbeitet werden können. Längere Vektoren werden in Streifen zerlegt und nacheinander bearbeitet. Daten, die verarbeitet werden sollen, werden vom Hauptspeicher über Ladepipes in die Vektorregister geladen; von dort gelangen sie in die Rechenpipes, die ihre Ergebnisse wieder in die Vektorregister ablegen. Daten, die nur als Zwischenergebnisse dienen, verbleiben zur weiteren Verwendung in den Vektorregistern; erst die Endergebnisse werden wieder in den Hauptspeicher zurücktransferiert.
Eine Registermaschine verfügt somit über drei Geschwindigkeitsbereiche:

1. Skalargeschwindigkeit
   die Arbeitsgeschwindigkeit einer eventuell vorhandenen skalaren Recheneinheit oder die Geschwindigkeit der Vektoreinheit bei einzelnen Operationen;
2. Vektorgeschwindigkeit
   die Geschwindigkeit, mit der Daten aus dem Hauptspeicher von der Vektoreinheit bearbeitet werden und die Ergebnisse wieder in den Hauptspeicher geschrieben werden; es handelt sich dabei um einfache Operationen wie die Addition zweier Vektoren;
3. Supervektorgeschwindigkeit
   die Arbeitsgeschwindigkeit der Rechenpipes, wenn die Operanden bereits in den Vektorregistern vorhanden sind und die Ergebnisse wieder in die Vektorregister abgelegt werden.

Für diese Geschwindigkeiten werden stets Maximalwerte oder Ergebnisse von Benchmarks (z.B. LINPACK, Livermoore) angegeben. Eine bekannte und vor allem für Vektorpipes geeignete Charakterisierung der Geschwindigkeit stammt von Hockney (siehe [HJ81]). Hierbei genügt die Zeit t, die die Pipe mit Zykluszeit τ für die Ausführung von n gleichartigen arithmetischen Operationen benötigt, der Darstellung

$$t = [s + l + (n-1)]\,\tau = \left[n + n_{1/2}\right]\frac{1}{r_\infty}.$$

$s\tau$ ist die Zeit für Operationen, die die eigentliche Vektoroperation vorbereiten; darin sind Adreßberechnungen, Speichertransfers von Konstanten u.ä. enthalten. l bezeichnet die Anzahl der Pipelinestufen und kann für unterschiedliche Operationen verschiedene Werte annehmen. Aus der obigen Darstellung ergibt sich unmittelbar, daß die Pipe im gefüllten Zustand nach jedem Taktzyklus ein Ergebnis liefert.
Die Größe r_∞ gibt die maximal erreichbare Geschwindigkeit an; sie wird im wesentlichen von der verwendeten Technologie bestimmt, aus der sich auch die Zykluszeit ergibt. $n_{1/2}$ bestimmt die Länge des Vektors, bei dessen Verarbeitung die halbe Maximalgeschwindigkeit erreicht wird. Sie ergibt sich aus der Architektur der Pipes, insbesondere durch die Anzahl

der Pipelinestufen. Daher kann sie als Maß für die Parallelität angesehen werden.
Bei gleichzeitiger Verwendung von p gleichartigen Pipes mit Kenngrößen $n_{1/2}$ und r_∞ verhalten sich diese wie eine Pipe mit den Kenngrößen

$$n_{1/2}^{neu} = p * n_{1/2} \quad , \quad r_\infty^{neu} = p * r_\infty$$

Dient der Ergebnisstrom einer Pipe i mit den Kenngrößen $n_{1/2}^{(i)}$ und $r_\infty^{(i)}$ als Eingabestrom einer weiteren parallel arbeitenden Pipe mit $n_{1/2}^{(i+1)}$ und $r_\infty^{(i+1)}$, so besitzt diese Superpipeline die Größen

$$n_{1/2} = \sum_{i=1}^{n} n_{1/2}^{(i)} \quad , \quad r_\infty = \sum_{i=1}^{n} r_\infty^{(i)}$$

Die Ladepipes zwischen Hauptspeicher und Vektorregister können ebenfalls durch solche Größen beschrieben werden, so daß sie auch in eine solche Kette integriert werden können. Dabei liefern diese jedoch keine Ergebnisse, sondern Datenworte aus dem bzw. in den Hauptspeicher.

4.2.1 Der Vektorrechner SNI S200/10

Der in Frankfurt verfügbare Vektorrechner wird von der Firma Fujitsu Ltd. in Japan gebaut und in Deutschland von der Firma Siemens-Nixdorf-Informationssysteme (SNI) vertrieben. Er gehört zu einer Baureihe, deren theoretische Höchstleistung von 500–5000 MFlop/s reicht. Die einzelnen Modelle unterscheiden sich durch die Zykluszeit, die Anzahl der Vektor- und Skalarprozessoren, die Anzahl der Pipelines pro Vektorprozessor sowie die Ausführung der Pipelines als Einfach-, Doppel- oder Vierfachpipelines. Der hiesige Vektorrechner besitzt je zwei Maskpipelines, Load-/Store-pipelines, Multifunktionspipelines und eine Divisionspipeline, die alle mit einem Takt von 250 MHz betrieben werden. Da jede Multifunktionspipe eine Multiplikation und eine Addition parallel ausführen und beide Pipelines gleichzeitig arbeiten können, resultiert daraus eine maximale Geschwindigkeit von 1000 MFlop/s. Die Divisionspipeline führt eine echte Division aus, liefert jedoch nur in jedem dritten Taktzyklus ein Ergebnis.

Als Betriebssystem dient UXP/M, ein System-V-Unix-Derivat, das von Fujitsu auf die Besonderheiten von Vektorrechnern mit einem bzw. mehreren Prozessoren angepaßt wurde.

Die Zeichnung auf der folgenden Seite zeigt den schematischen Aufbau. Funktionseinheiten werden durch beschriftete Rechtecke und Datenpfade

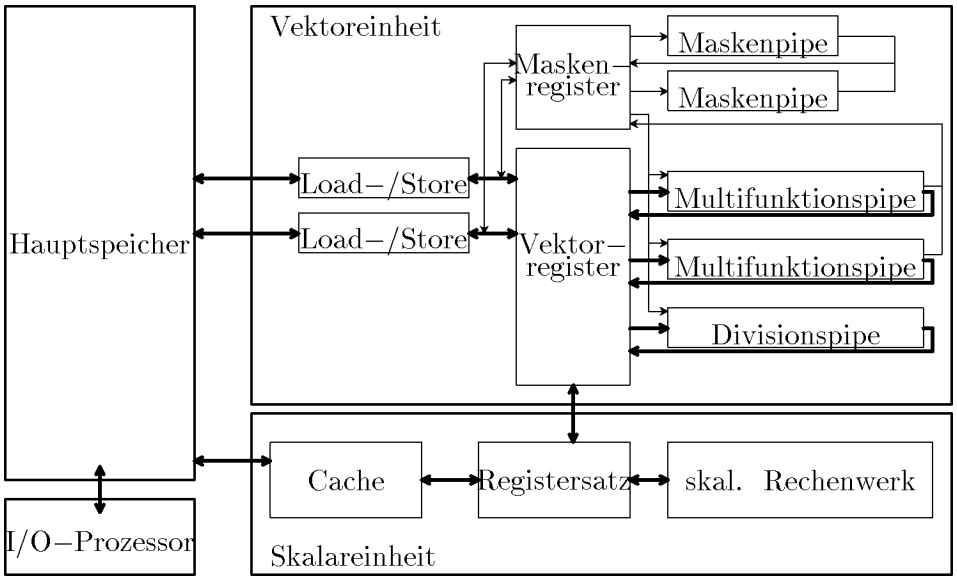

Abb. 4.1: Blockschaltbild der SNI S200

durch Pfeile dargestellt. Datenpfade mit der Breite von einem Bit sind durch dünnere Pfeile gekennzeichnet.
Die Skalareinheit ist binärkompatibel zu den Mainframes der Serie /370 der Firma IBM. Der Registersatz besteht aus 16 Festpunktregistern mit der maximalen Breite von 32 Bit und acht Fließpunktregistern der maximalen Breite von 64 Bit. Zusätzlich enthält die Skalareinheit noch weitere Register, die Maschinenzustände enthalten oder der Steuerung dienen. Die Skalareinheit selbst verarbeitet 231 verschiedene Befehlsarten, die auf Hauptspeicher, Register oder beidem operieren. Dabei verarbeitet sie die folgenden Datentypen:

Datentyp:	Breite in Bits:
Logischer Wert	8, 16, 32
binäre Ganzzahl	16, 32
Fließkommazahl	32, 64
Dezimalzahl	8 pro Ziffer

Weitere 132 Befehlsarten, die Vektorbefehle, interpretiert sie und übergibt sie an die Vektoreinheit zur Bearbeitung. Um seltener auf den Hauptspeicher zuzugreifen, verfügt sie über 128 KBytes Cache, der in Zeilen zu 8 Bytes organisiert ist. Zur Unterstützung der virtuellen Adressierung besitzt sie eine assoziative Adreßübersetzungstabelle mit 1024 Einträgen. Das skalare Rechenwerk besteht aus den Interfaces zu dem I/O-Prozessor, der Vektoreinheit, dem Hauptspeicher, der Systemkonsole und einer siebenstufigen Pipeline. Die Skalareinheit wird mit einem Taktzyklus von 8 ns betrieben und besitzt eine theoretische Verarbeitungsgeschwindigkeit von 125 MFlop/s.

Der I/O-Prozessor kontrolliert die Datentransfers zwischen dem Hauptspeicher und der Peripherie. Dazu zählen Festplatten, Netzwerkkarten, Terminals, Drucker u.ä. Hierzu verwendet er bis zu 128 Kanäle, über die er elektrische oder optische Signale sendet bzw. empfängt. Die maximale Datenübertragungsrate eines elektrischen Kanals liegt bei 75 KBytes/sec und die eines optischen Kanals bei 9 MBytes/sec. Die Übertragungsrate des gesamten Prozessors liegt bei etwa 1 GBytes/sec.

Der Hauptspeicher besteht aus statischen 1-MBit RAM-Bausteinen, deren Zugriffszeit 35 ns beträgt. Es handelt sich um einen SECDED-Speicher (single error correction, double error detection), d.h. ein-Bit-Fehler in einem 32-Bit-Wort können während des Zugriffs korrigiert werden, zwei-Bit-Fehler werden erkannt. Seine Größe betrug bei Installation des Systems 256 MBytes und wurde im Oktober 1993 auf 512 MBytes angehoben. Eine Eigenheit dieses Speichers unter UXP/M ist das sogenannte „VP-Limit“, eine Grenze, oberhalb derer nur Programme ihre Daten ablegen dürfen, die die Vektoreinheit verwenden. Daten- und Stacksegmente von nichtvektorisierten Programmen sowie alle Textsegmente werden stets im Speicherbereich unterhalb abgelegt. Für beide Bereiche gibt es ein eigenes Swap-Device, auf die nicht benötigte Speicherseiten ausgelagert werden. Dies geschieht unterhalb des VP-Limit mit Seiten der Größe 4 KBytes und oberhalb mit der Größe 4 MBytes.
Diese Grenze kann bei jedem Systemstart verändert werden.

Die Vektoreinheit besteht aus sieben Pipelines sowie dem Vektor- und Maskenregistersatz. Sie erhält ihre Instruktionen und zu verarbeitende skalare Daten von der Skalareinheit. Dabei verarbeitet sie die folgenden Datentypen:

Datentyp:	Breite in Bits:
Logischer Wert	1, 64
binäre Ganzzahl	32
Fließkommazahl	32, 64

Der Vektorregistersatz umfaßt 4K Elemente der Breite 64 Bit, und kann zur Laufzeit in 8 (16, 32, 64, 128, 256) Register mit je 512 (256, 128, 64, 32, 16) Elementen unterteilt werden. Die Vektorregister wurden durch ECL-Gatter realisiert; die Zugriffszeit beträgt 1.6 ns bei einer Gatterverzögerungszeit von 80 ps. Der Maskenregistersatz liegt physikalisch zweimal vor; jeder dieser Sätze besteht aus 4K Elementen à 1 Bit und kann auf die gleiche Art wie der Vektorregistersatz in Maskenregister unterteilt werden. Beim Beschreiben eines Maskenvektors werden beide Sätze simultan beschrieben; beim Lesen zweier Maskenvektoren stammen diese aus unterschiedlichen Sätzen.
Der Datentransfer zwischen Vektorregister und Hauptspeicher erfolgt durch zwei unterschiedliche Load-/Store-Pipes. Dabei gibt es drei verschiedene Arten, auf Daten zuzugreifen:

1. Kontinuierlicher Zugriff: die Daten liegen im Hauptspeicher unmittelbar aufeinanderfolgend vor; ab einer vorgegebenen Adresse wird eine

bestimmte Anzahl von Worten gelesen bzw. geschrieben; dazu wird die Zugriffsadresse jeweils um ein Wort erhöht.

2. Zugriff mit konstantem Versatz: die Daten liegen nicht aufeinanderfolgend vor; der Zugriff ist ähnlich, jedoch wird die Zugriffsadresse um eine Konstante erhöht.
3. Wahlfreier Zugriff (Gather/Scatter): auf die Daten wird mittels eines Vektorregisters, das als Indexvektor fungiert, zugegriffen; für jeden einzelnen Zugriff wird dabei eine Adresse generiert.

Während die erste Load-/Store-Pipe alle drei Zugriffsarten ausführen kann, ist die zweite Pipe nur für den kontinuierlichen Zugriff vorbereitet. Desweiteren kann die zweite Pipe nur arbeiten, wenn auch die erste den kontinuierlichen Zugriff ausführt.

Die Multifunktionspipes bestehen aus einem neunstufigen Multiplizierwerk und einem achtstufigen Addierwerk; durch die parallele Anordnung einiger Stufen beider Werke ergibt sich eine maximale Anzahl aufeinander folgender Stufen von 15. Sie führen damit die folgenden Operationen aus:

- Multiplikation
- Addition oder Subtraktion
- Multiplikation mit anschließender Addition bzw. Subtraktion
- Vergleiche (z.B. $a_i < b_i$)
- Logische Operationen (AND, OR, NOT, ...) auf ganzen Zahlen
- Konvertierung von Datenformaten
- Minimum- und Maximumbestimmung
- Summation ($x = \sum_i a_i$)

Die Operanden für alle Operationen stammen stets aus den Vektorregistern. Für die ersten vier Operationen müssen die Operanden vom gleichen Typ sein. Die Summation wird von einem nachgeschalteten Addierwerk vorgenommen, das nur an einer Pipe vorliegt, aber auch von der zweiten Pipe verwendet werden kann. Ergebnisvektoren werden wieder in die Vektorregister zurückgespeichert; die Ergebnisse der Minimum- und Maximumbestimmung und der Summation werden an die Skalareinheit übergeben, die sie in einem Fließkommaregister ablegt. Der Bitvektor, der sich aus einer Vergleichsoperation ergibt, wird in einem Maskenregister abgespeichert. Jede Operation kann als weitere Eingabe einen solchen Maskenvektor erhalten, der angibt, welche Vektorelemente verarbeitet werden sollen; dabei wird zwar weiterhin jedes Element bearbeitet, aber nur die gewünschten Ergebnisse werden zurückgespeichert.

Die Divisionspipeline führt in einem 18-stufigen Rechenwerk eine echte elementweise Division zweier Vektoren aus, liefert jedoch nur in jedem dritten Taktzyklus ein Ergebnis.
Während alle anderen Pipes über Datenpfade mit einer Breite von 64 Bit mit Daten versorgt werden, beträgt die Breite der Datenpfade von und zur Divisionspipe nur 24 Bit.

Von den drei arithmetischen Pipelines können je zwei gleichzeitig arbeiten. Die nötige Versorgung mit Daten und das Abspeichern der Ergebnisse in Vektorregister ist dabei sichergestellt.

Wie die arithmetischen Pipes auf Datenwortvektoren, so operieren die Maskenpipes auf Bitvektoren. Dabei führen sie logische Operationen (z.B. UND, ODER, XOR) aus, deren Ergebnisse in ein Maskenregister abgelegt werden, oder Makrofunktionen (Anzahl der gesetzten Bits oder erstes gesetztes Bit eines Maskenvektors), deren Ergebnis an die Skalareinheit übergeben wird. Diese legt es in einem Festpunktregister ab. Beide Pipes können gleichzeitig arbeiten.

Wie auch die Vektorregister sind alle Pipes in ECL-Technologie gefertigt. Auch hier beträgt die Gatterverzögerungszeit 80 ps.
Skalar- und Vektoreinheit sind als quadratisches Board mit Kantenlänge 10 Zoll realisiert, das etwa 144 Chips beherbergt; auf einem Chip wurden etwa 15000 Gatter integriert. Der Kontakt eines Boards mit dem System wird durch mehr als 8000 vergoldete Pins an der Unterseite hergestellt.

5. Reduktionsalgorithmen

Im folgenden Kapitel werden die verwendeten Algorithmen zur Gitterbasenreduktion vorgestellt. Dabei werden zuerst die seriellen Versionen behandelt und anschließend auf die vektorisierten eingegangen.

5.1 LLL-Reduktion

Von zentraler Bedeutung ist der Algorithmus zur LLL-Reduktion von ganzzahligen Gitterbasen. Nachteilig an diesem Verfahren ist, daß ein großer Teil der Laufzeit in Berechnungen auf langen Zahlen verbraucht wird. Schnorr und Euchner stellten in [SE91],[E91] eine Version der LLL-Reduktion mit iterativer Orthogonalisierung vor, die möglichst viele Berechnungen auf langen Zahlen durch solche auf Fließkommazahlen ersetzt. Lediglich die Basismatrix $(b_1, \ldots, b_m)$ wird in exakter Darstellung mitgeführt, da Rundungsfehler in dieser das Gitter verändern würden und nicht mehr korrigiert werden könnten. Daher werden Basistransformationen stets auf den exakten Daten ausgeführt. Neben einer Kopie der Basismatrix $(b'_1, \ldots, b'_m)$ liegen auch die Gram-Schmidt-Koeffizienten $\mu_{i,j}$ und Höhenquadrate $c_i = \|\hat{b}_i\|^2$ als Fließkommawerte vor. Dort, wo Rundungsfehler auftreten können, wird überprüft, ob diese Auswirkungen auf die weitere Reduktion haben. Im einzelnen sollen die folgenden vier Punkte eine Reduktion ohne größere Rundungsfehler garantieren:

1. Jede Veränderung der Basis wird exakt durchgeführt und anschließend die Basis in Fließkommadarstellung korrigiert.
2. Bei Eintritt in die Stufe k werden die Gram-Schmidt-Koeffizienten $\mu_{k,j}$ und das Höhenquadrat c_k neu berechnet. Die dabei in die Berechnung eingehenden Größen wurden entweder auf niedrigeren Stufen berechnet, so daß sie als hinreichend exakt gelten, oder stammen aus der Basismatrix in Fließkommadarstellung.
3. Gilt bei der Berechnung der $\mu_{k,j}$ und c_k, daß ein berechnetes Skalarprodukt „klein“ gegenüber dem Produkt der Normen seiner Eingangsvektoren ist, so wird diese Berechnung mit den exakten Daten wiederholt, um eine Auslöschung signifikanter Stellen zu unterbinden.
4. Tritt bei der Längenreduktion ein sehr großer Reduktionskoeffizient auf, so kann nicht ausgeschlossen werden, daß bei der Berechnung der

$\mu_{k,j}$ eine Auslöschung signifikanter Stellen auftritt. Daher wird die Berechnung anschließend auf Stufe $k-1$ weitergeführt.

Die Bezeichnungen „sehr groß“ und „klein“ sind hierbei von der verwendeten Arithmetik abhängig. Die Algorithmen verwenden typischerweise einen Faktor von $2^{r/2}$, wobei r die Anzahl der Mantissenbits der Fließkommadarstellung ist. Die Speicherung und Berechnung erfolgt mittels doppelt-genauer Zahlen, die auf Workstations über eine 53-Bit-Mantisse verfügen. Der Vektorrechner stellt in seiner Darstellung 56 Bits zur Verfügung, wovon drei führende Bits jedoch zu Null werden können.
Oben aufgeführter Punkt 4 bewirkt, daß bei großen Reduktionskoeffizienten Stufen wiederholt werden. Die dadurch entstehende Laufzeiterhöhung durch zusätzliche Iterationen wird jedoch durch den Wegfall von Berechnungen in langer Arithmetik mehr als aufgefangen. Es zeigt sich, daß die Anzahl dieser „Korrekturschritte“ mit wachsender Maschinengenauigkeit abnimmt.

Der im Algorithmus L3FP (siehe folgende Seite) ausgelassene Schritt 4 erniedrigt die Stufe, falls ein Längenreduktionskoeffizient sehr groß ist. Ist dies nicht der Fall, so wird überprüft, ob der neu entstandene Vektor ein um den Faktor δ kleineres Höhenquadrat bezüglich der Vektoren $(b_1, \ldots, b_{k-2})$ als der Vektor b_{k-1} besitzt. Falls dies zutrifft, werden die beiden Vektoren vertauscht und die Stufe dekrementiert, sonst inkrementiert. Dieses Programmstück hat die folgende Gestalt:

```
IF flag THEN
    flag := false
    k := max(k − 1, 2)
ELSE
    IF δc_{k−1} > c_k + μ²_{k,k−1} c_{k−1} THEN
        b_k ↔ b_{k−1}; b'_k ↔ b'_{k−1}
        k := max(k − 1, 2)
    ELSE
        k := k + 1
    ENDIF
ENDIF
```

Da es sich bei der Matrix der Gram-Schmidt-Koeffizienten um eine untere Dreiecksmatrix mit Einsen in der Diagonale handelt, wird in der Implementierung von Euchner ([E91]) lediglich dieser Teil der Matrix ohne die Diagonalelemente abgespeichert. Wenn sichergestellt ist, daß die Gram-Schmidt-Koeffizienten und Höhenquadrate für die Stufen $i < k$ bereits berechnet sind, kann der Algorithmus auf Stufe $k > 2$ aufgerufen werden. Außerdem kann man ihn veranlassen, nur bis zur Stufe $l < m$ zu reduzieren. Taucht während der Reduktion ein Nullvektor auf, weil ein Erzeugendensystem statt einer Basis übergeben wurde, so stoppt der Algorithmus und gibt dessen Position zurück. Eine weitere Änderung betrifft

Algorithmus L3FP

Eingabe: Gitterbasis $b_1, \ldots, b_m \in \mathbb{Z}^n$ $\delta \in [\frac{1}{2}, 1]$

/* Schritt 1: Initialisierung */
$k := 2$
$flag := false$
FOR $i = 1, \ldots, m$ DO
  $b'_i := (b_i)'$ /* Berechnung der Fließkommamatrix */
ENDFOR
WHILE $k \leq m$ DO
  /* Schritt 2: Berechnung der $\mu_{k,j} \quad j = 1, \ldots, k-1$ und c_k */
  $c_k := \|b'_k\|^2$
  IF $k = 2$ THEN $c_1 := \|b'_1\|^2$
  FOR $j = 1, \ldots, k-1$ DO
    $s := \langle b'_k, b'_j \rangle$
    IF $s \leq 2^{r/2} \|b'_k\| * \|b'_j\|$ THEN
      $s := \langle b_k, b_j \rangle'$ /* Auslöschung signifikanter Bits (siehe Punkt 3) */
    ENDIF
    $\mu_{k,j} := \left(s - \sum_{i=1}^{j-1} \mu_{j,i} \mu_{k,i} c_i \right) / c_j$
    $c_k := c_k - \mu_{k,j}^2 c_j$
  ENDFOR
  /* Schritt 3: Längenreduktion von b_k */
  FOR $j = k-1, \ldots, 1$ DO
    IF $|\mu_{k,j}| > \frac{1}{2}$ THEN
      $\mu := \mu_{k,j}$
      IF $|\mu| > 2^{r/2}$ THEN
        $flag := true$ /* Reduktionskoeffizient zu groß (siehe Punkt 4) */
      ENDIF
      FOR $i = 1, \ldots, j-1$ DO
        $\mu_{k,i} := \mu_{k,i} - \mu \mu_{j,i}$
      ENDFOR
      $\mu_{k,j} := \mu_{k,j} - \mu$
      $b_k := b_k - \mu b_j$; $b'_k := (b_k)'$
    ENDIF
  ENDFOR
  /* Schritt 4: Vertauschen von b_k und b_{k-1} und Verändern der Stufe */
ENDWHILE

Ausgabe: mit δ LLL-reduzierte Basis $b_1, \ldots, b_m$

Schritt 4, der durch folgendes Programmstück „tiefe Einfügungen“ ersetzt wurde:

```
IF flag THEN
    flag := false
    k := max(k − 1, 2)
ELSE
    c := ‖b'_k‖²
    i := 1
    WHILE i < k DO
        IF δc_i ≤ c THEN
            c := c − μ²_{k,i} c_i
            i := i + 1
        ELSE
            (b_1, ..., b_k) := (b_1, ..., b_{i−1}, b_k, b_i, ..., b_{k−1})
            (b'_1, ..., b'_k) := (b'_1, ..., b'_{i−1}, b'_k, b'_i, ..., b'_{k−1})
            verlasse die Schleife
        ENDIF
    ENDWHILE
    IF i < k THEN
        k := max(i − 1, 2)
    ELSE
        k := k + 1
    ENDIF
ENDIF
```

Hierbei wird versucht, den neuentstandenen Vektor b_k an einer möglichst kleinen Position i einzufügen, sodaß das entstehende Höhenquadrat c sich mindestens um den Faktor δ verringert. Offensichtlich wird hierdurch gewährleistet, daß stets der bis auf einen Faktor δ kürzeste Vektor an die erste Position gelangt. Schnorr und Euchner zeigten in [SE91], daß die dabei entstehenden Vektoren kürzer werden und im allgemeinen die Laufzeit sinkt. Im worst-case hingegen könnte die Laufzeit nicht mehr polynomial sein. Führt man die tiefen Einfügungen nur bis zu einer maximalen Position d durch, so ist das Verfahren auch weiterhin polynomial.

Die Vektorisierung des LLL-Algorithmus geht von der seriellen Version aus. Dabei sollte seine Flexibilität erhalten bleiben; es ist weiterhin möglich, die Reduktion auf höheren Stufen zu beginnen bzw. vorzeitig zu beenden. Nach jeder Längenreduktion kann der neu entstandene Vektor auf seine Struktur getestet werden. Vektorisiert wurden sämtliche arithmetischen Operationen auf den Basisvektoren sowie alle inneren Schleifen, die keinen Funktionsaufruf beinhalten. Ferner wurde die Reihenfolge von Berechnungen verändert und Schleifen aufgeteilt, da dies erst Vektorbefehle ermöglichte. Aus technischen Gründen wird die Gram-Schmidt-Matrix transponiert abgespeichert, und deren Diagonalelemente werden zu Beginn eingetragen. Daraus resultiert Algorithmus L3FPV.

Algorithmus L3FPV

Eingabe: Gitterbasis $b_1, \ldots, b_m \in \mathbb{Z}^n$, Stufe $k \in [2, m]$

/* 1. Schritt: Initialisierung */
$\mu_{i,i} := 1 \qquad (i = 1, \ldots, m)$
FOR $i = 1, \ldots, m$ DO
  $b'_i := (b_i)'$
ENDFOR
$flag := false$
WHILE $k < m$ DO
  IF $k = 2$ THEN $c_1 := \|b_1\|^2$
  /* 2. Schritt: Berechnung der $\mu_{k,j}$ und c_k */
  $c_k := \|b'_k\|^2$
  FOR $j = 1, \ldots, k-1$ DO
    $\mu_{k,j} := \langle b'_j, b'_k \rangle$
    IF $|\mu_{k,j}| < 2^{-r/2} * \|b'_j\| * \|b'_k\|$ THEN
      $\mu_{k,j} := \langle b_j, b_k \rangle'$
    ENDIF
  ENDFOR
  FOR $j = 1, \ldots, k-1$ DO
    $\mu_{k,j} := \mu_{k,j} - \mu_{j,i} * \mu_{k,i} \qquad (i = 1, \ldots, j-1)$
  ENDFOR
  $s := \mu_{k,j}; \quad \mu_{k,j} := s/c_j; \quad c_k := c_k - \mu_{k,j} * s \qquad (j = 1, \ldots, k-1)$
  /* 3. Schritt: Längenreduktion von b_k */
  FOR $j = k-1, \ldots, 1$ DO
    $s := \lceil \mu_{k,j} \rfloor$
    IF $|s| > \frac{1}{2}$ THEN
      $b_k := b_k - s * b_j$
      IF $|s| > 2^{r/2}$ THEN $flag := true$
      $\mu_{k,i} := \mu_{k,i} - \mu_{j,i} * s \qquad (i = 1, \ldots, j)$
    ENDIF
  ENDFOR
  $b'_k := (b_k)'$
  /* 4. Schritt: Tiefes Einfügen von b_k und Verändern der Stufe */
ENDWHILE

Ausgabe: $b_1, \ldots, b_m \in \mathbb{Z}^n$ mit δ LLL-reduzierte Gitterbasis

5.2 Block-Reduktion

Das folgende Verfahren der Blockreduktion geht ebenfalls auf Schnorr und Euchner ([SE91],[E91]) zurück.

Algorithmus Block-Reduktion

Eingabe: Gitterbasis $b_1, \ldots, b_m \in \mathbb{Z}^n$, $\delta \in [\frac{1}{4}, 1[$

$j := 0$; $z := 0$
δ-LLL-Reduktion der Basis
WHILE $z < m$ **DO**
  $j := j + 1$
  IF $j = m$ **THEN** $j := 1$
  $k := \min(j + \beta - 1, m)$
  $h := \min(k + 1, m)$
  $\{c, (u_j, \ldots, u_k)\}$:=Enum(j, k)
  IF $c < \delta c_j$ **THEN**
    Integriere $b := \sum_{i=j}^{k} u_i b_i$ in die Basis
    δ-LLL-Reduktion
    $z := 0$
  ELSE
    $z := z + 1$
    Längenreduktion von $(b_1, \ldots, b_h)$
  ENDIF
ENDWHILE

Ausgabe: Mit δ β−Blockreduzierte Basis $b_1, \ldots, b_m$

Der anfängliche Aufruf der LLL-Reduktion dient zur Bereitstellung der Höhenquadrate und der Gram-Schmidt-Koeffizienten. Während der Blockreduktion wird ein Fenster von β Vektoren zyklisch durch die Basis verschoben. Die Funktion Enum(j, k) sucht aus allen Linearkombinationen der Vektoren $b_j, \ldots, b_k$ die mit minimalem Höhenquadrat $(\bar{c}_j)$ bezüglich der Vektoren $b_1, \ldots, b_{j-1}$. Wird eine solche gefunden und ist deren zugehöriges Höhenquadrat mindestens um den Faktor δ kleiner als das bisherige, so wird der entstehende Vektor an die Position j der Basis eingefügt. Wir sprechen in diesem Fall von einer „Verbesserung“. Der Zähler z, der die Anzahl aufeinanderfolgender erfolgloser Enumaufrufe (d.h. es wurde keine Verbesserung gefunden) zählt, wird dann wieder auf Null gesetzt. Wenn die While-Schleife verlassen wird, liegt offensichtlich eine mit δ β-blockreduzierte Gitterbasis vor.
Die von mir verwendete Version der Blockreduktion ist mit der seriellen identisch. Dies hat zwei Gründe:

1. Das Verfahren besitzt fast keine implizite Parallelität. Die Berechnungen verwenden zumeist Daten, die erst kurz vorher berechnet wurden.

2. Fast die gesamte Rechenzeit wird in der LLL-Reduktion und der Aufzählungsfunktion verbraucht. Diese Funktionen sind hochgradig vektorisiert. Der Anteil der übrigen Berechnungen ist verschwindend gering.

Die Integration des neuentstandenen Vektors in die Basis kann auf verschiedene Arten erfolgen:
In [SE91] wurde dieser Vektor an die Position j eingefügt und anschließend eine LLL-Reduktion beginnend auf Stufe j durchgeführt. Da hierbei die Basis zu einem Erzeugendensystem expandiert wurde, entstand bei der Reduktion ein Nullvektor. Dieser wurde entfernt und nochmals eine LLL-Reduktion aufgerufen. Konnte die Funktion Enum keinen neuen Vektor zurückliefern, so wurden die ersten h Vektoren der Basis längenreduziert. Da hierbei kleine Höhenquadrate aufgetreten sind und aufgrund von Rundungsfehlern in sehr seltenen Fällen Programmläufe nicht terminierten, wurden einige Veränderungen vorgenommen. Die Längenreduktion wurde durch eine δ-LLL-Reduktion ersetzt und die LLL-Reduktionen auf Stufe $j-1$ statt j aufgerufen. Dadurch stieg der Rechenzeitbedarf für die LLL-Reduktion an, aber alle Programmläufe terminierten. Um diesen Anstieg der Rechenzeit wieder zu beseitigen, wurde der doppelte Aufruf der LLL-Reduktion eleminiert. Dazu werden die Vektoren $b_j^{alt}, \ldots, b_k^{alt}$ zu einer Basis $b_j^{neu}, \ldots, b_k^{neu}$ von $L(b_j^{alt}, \ldots, b_k^{alt})$ transformiert, so daß für den neuen Vektor $b_j^{neu} = \sum_{i=j}^{k} u_i b_i^{alt}$ gilt.

Algorithmus Basis (siehe [H94])

Eingabe: $(u_j, \ldots, u_k) \in \mathbb{Z}^{k+1-j} \setminus 0^{k+1-j}$, $b_j, \ldots, b_k$

$b_j^{neu} := \sum_{i=j}^{k} u_i b_i$
WHILE $u_k = 0$ DO
  $k := k - 1$
ENDWHILE
$i := k - 1$
WHILE $|u_k| > 1$ DO
  WHILE $u_i = 0$ DO
    $i := i - 1$
  ENDWHILE
  $q := \lceil u_k / u_i \rfloor$
  $u_i := u_k - q u_i$
  $u_k := u_i^{alt}$
  $b_k := q b_k + b_i$
  $b_i := b_k^{alt}$
ENDWHILE
FOR $i = k, \ldots, j+1$ DO
  $b_i := b_{i-1}$
ENDFOR
$b_j := b_j^{neu}$

Ausgabe: $b_j, \ldots, b_k$

Der Nachweis der Korrektheit findet sich in [H94]. Nachteilig an diesem Verfahren ist, daß es außer den Operationen auf Basisvektoren keinerlei Möglichkeiten der Vektorisierung bietet. In der von mir verwendeten Version der Blockreduktion geschieht die Integration des neuen Vektors auf die folgende Weise:

$b_j^{neu} := 0$
FOR $i = j, \ldots, k$ DO
  IF $u_i \neq 0$ THEN
    $b_j^{neu} := b_j^{neu} + u_i b_i$
    $l := i$
  ENDIF
ENDFOR
IF $u_l = 1$ THEN
  $(b_j, \ldots, b_k) := (b_j^{neu}, b_j, \ldots, b_{l-1}, b_{l+1}, \ldots, b_k)$
  δ-LLL-Reduktion von $(b_1, \ldots, b_h)$
ELSE
  δ-LLL-Reduktion von$(b_1, \ldots, b_{j-1}, b_j^{neu}, b_j, \ldots, b_h)$
  entferne den entstehenden Nullvektor
  nochmalige δ-LLL-Reduktion
ENDIF

Ist der höchste Koeffizient bereits eins, so wird der zugehörige Vektor aus der Basis entfernt und der neuentstandene Vektor eingefügt. Andernfalls wird das ursprüngliche Verfahren angewendet. Da bislang der höchste Koeffizient stets eins war, wurde der Programmzweig mit zweimaliger LLL-Reduktion nie aufgerufen.

5.3 Das Aufzählungsverfahren

Die folgende Funktion Enum zur vollständigen Aufzählung aller in Frage kommender Linearkombinationen mit zugehörigem Höhenquadrat stammt aus [SE91]. Zur Minimierung des Ausdrucks

$$c_j(\tilde{u}_j, \ldots, \tilde{u}_k) = \left\| \pi_j \left(\sum_{i=j}^{k} \tilde{u}_i b_i \right) \right\|^2 = \sum_{i=j}^{k} \left(\sum_{l=i}^{k} \tilde{u}_l \mu_{l,i} \right)^2 c_i$$

mit Minimalstelle $(u_j, \ldots, u_k) \in \mathbb{Z}^{k+1-j} \setminus 0^{k+1-j}$ durchläuft sie einen Berechnungsbaum, dessen Knoten auf einer Stufe t mit temporären Koeffizientenvektoren $(\tilde{u}_t, \ldots, \tilde{u}_k)$ und dem zugehörigen $c_t(\tilde{u}_t, \ldots, \tilde{u}_k)$ markiert

Algorithmus Enum(j,k)

Eingabe: j,k mit $1 \le j < k \le m$ $\quad c_j,\ldots,c_k$ $\quad \mu_{i,l}$ mit $j \le l < i \le k$

```
/* Initialisierung */
c̄_j := c_j
δ_j := 1
ũ_j := u_j := 1
Δ_j := y_j := v_j := 0
s := t := j
FOR i = j+1,...,k DO
    c̃_j := u_i := ũ_i := Δ_i := y_i := v_i := 0
    δ_i := 1
ENDFOR
WHILE t ≤ k DO
    c̃_t := c̃_{t+1} + (ũ_t + y_t)^2 c_t
    IF c̃_t < c̄_j THEN
        IF t > j THEN
            t := t - 1
            y_t := Σ_{i=t+1}^{k} ũ_i μ_{i,t}
            ũ_t := v_t := ⌈-y_t⌋
            Δ_t := 0
            IF ũ_t > -y_t THEN
                δ_t := -1
            ELSE
                δ_t := 1
            ENDIF
        ELSE
            c̄_j := c̃_j
            u_i := ũ_i   i = j,...,k
        ENDIF
    ELSE
        t := t + 1
        s := max(s,t)
        IF t < s THEN  Δ_t := -Δ_t
        IF Δ_t δ_t ≥ 0 THEN  Δ_t := Δ_t + δ_t
        ũ_t := v_t + Δ_t
    ENDIF
ENDWHILE
```

Ausgabe: $\bar{c}_j = \min c_j(\tilde{u}_j,\ldots,\tilde{u}_k)$ und $u_j,\ldots,u_k$

sind, in depth-first-search-order. Ein Knoten unterscheidet sich von seinem Vater durch den Koeffizienten $\tilde{u}_t$, und er besitzt ein Höhenquadrat, das um den Summand

$$\left(\tilde{u}_t + \underbrace{\sum_{l=t+1}^{k} \tilde{u}_l \mu_{l,t}}_{=:y_t}\right)^2 c_t$$

größer als das seines Vaters ist. Bei der Berechnung eines neuen Koeffizienten $\tilde{u}_t$ werden alle ganzen Zahlen in der Reihenfolge wachsender $\tilde{u}_t + y_t$ betrachtet. Für den nichttrivialen Koeffizienten mit maximalem Index werden nur positive Werte zugelassen, um Redundanzen in der Berechnung zu vermeiden. Da die Höhenquadrate entlang eines Weges von der Wurzel zu einem Blatt monoton wachsen, können ganze Berechnungszweige ignoriert werden, wenn deren Wurzel bereits ein zu großes Höhenquadrat besitzt.

Da die Anzahl der Knoten in dem Berechnungsbaum exponentiell mit der Blockweite $\beta := k + 1 - j$ wächst, muß der Baum bei größeren Blockweiten geschnitten werden. Ziel der verschiedenen Schnittmethoden ist es, den Quotienten aus Anzahl aufzuzählender Knoten und der Wahrscheinlichkeit, einen Vektor mit kleinster Höhe zu finden, zu minimieren. Die Wahrscheinlichkeit bezieht sich hierbei auf die Gleichverteilung der Gram-Schmidt-Koeffizienten im Intervall $[-\frac{1}{2}, \frac{1}{2}]$. Im Algorithmus Enum$(j, k)$ wird ein Schnitt durch das Ersetzen von

`IF` $\tilde{c}_t < \bar{c}_j$ `THEN`

durch

`IF` $\tilde{c}_t < \alpha(t,j)\bar{c}_j$ `THEN`

mit $\alpha(t, j) \in [0, 1]$ realisiert. Die Werte der Schnittfunktion α werden in der Initialisierung bereitgestellt. Hörner hat in [H94] eine Anzahl Schnittvarianten in Bezug auf Erfolg und Rechenzeit verglichen.

Da die serielle Version der Aufzählung einen Baum abläuft, dessen Knoten jeweils von dem Elternknoten abhängen, ist sie nicht vektorisierbar.
Die vektorisierte Aufzählungsfunktion EnumV erhält als Eingabe die Blockgrenzen j, k, die Höhenquadrate $\|\hat{b}_j\|^2 =: C_j, \ldots, \|\hat{b}_k\|^2 =: C_k$ und die Gram-Schmidt-Koeffizienten $\mu_{i,l} \quad j \leq i < l \leq k$. Sie zählt die Knoten einer Stufe parallel auf. Die Knotenmenge $\mathbb{V}$ besteht aus Elementen der Form $(\tilde{u}_j, \ldots, \tilde{u}_k, c_s(\tilde{u}_s, \ldots, \tilde{u}_k))$. Die Funktion EnumV startet mit einer leeren Knotenmenge $\mathbb{V}$. Zu Beginn eines Schleifendurchlaufs werden alle Knoten gelöscht, deren Höhenquadrate bereits zu groß sind. Anschließend wird berechnet, wieviele Kinder ein Knoten auf dieser Stufe höchstens haben kann. Danach wird jeder Knoten entsprechend vervielfacht, die Komponente $\tilde{u}_s$ mit den nächsten ganzen Zahlen zu $-y_s$ belegt und das neue Höhenquadrat berechnet. Am Ende der Schleife werden Knoten mit

Algorithmus EnumV(j,k):

Eingabe: $\mu_{i,l}\ \ j \le i < l \le k \quad C_i = \|\hat{b}_i\|^2 \quad i = j,\ldots,k$

/* Initialisierung */
$s := k+1$
Vektormenge $\mathbb{V} := \emptyset$
/* Die Schleife über die Stufen */
`WHILE` $s > j$ `DO`
$s := s-1$
Eliminiere alle Vektoren $v \in \mathbb{V}$ mit $c_{s+1}(v_{s+1},\ldots,v_k) > C_j$
$vzw := \lfloor \sqrt{\frac{C_j - \min_{v\in\mathbb{V}} c_{s+1}(v_{s+1},\ldots,v_k)}{C_s}} + 1 \rfloor$
`FOR` $v = (0^{s+1-j}, v_{s+1},\ldots,v_k, c_{s+1}(v_{s+1},\ldots,v_k)) \in \mathbb{V}$ `DO`
$y(v) := \sum_{i=s+1}^{k} v_i \mu_{i,s}$
Erstelle $2 * vzw$ Kopien von v mit $v_s := \lceil -y(v) \pm (i-0.5) \rfloor$
$i = 1,\ldots,vzw$
$c_s(v_s,\ldots,v_k) = c_{s+1}(v_{s+1},\ldots,v_k) + (v_s - y(v))^2 C_s$
`ENDFOR`
Füge Vektoren v der Form $(0^{s+1-j}, w, 0^{k-s}, w^2 C_s)$
mit $w \in \mathbb{N} \cap [1, \sqrt{C_j/C_s}]$ in $\mathbb{V}$ ein
`ENDWHILE`
Bestimme $\min_{v\in\mathbb{V}} c_j(v_j,\ldots,v_k)$ und zugehörigen
Vektor $v = (v_j,\ldots,v_k, c(v_j,\ldots,v_k))$

Ausgabe: $(v_j,\ldots,v_k, c_j(v_j,\ldots,v_k)) \in (\mathbb{Z}^{k+1-j} \times \mathbb{R}) \setminus 0$ mit minimalem $c_j(v_j,\ldots,v_k)$

$\tilde{u}_{s+1} = \ldots = \tilde{u}_k = 0$ in $\mathbb{V}$ eingefügt. Nach Beendigung der While-Schleife wird der Knoten mit dem kleinsten Höhenquadrat zurückgegeben.
Zur Korrektheit des Algorithmus ist folgendes zu zeigen:

i. Alle Linearkombinationen mit hinreichend kleinem Höhenquadrat werden aufgezählt.
ii. Es wird niemals der Nullvektor zurückgegeben.
iii. Der Algorithmus terminiert.

zu i. Ein beliebiger Vektor v wird nur dann nicht in die Vektormenge aufgenommen, wenn er eine der beiden folgenden Bedingungen erfüllt:
- Er besitzt die Form $v = (0^{s-j}, v_s, 0^{k-s}, c_s(v_s,\ldots,v_k))$, wobei $v_s \in \mathbb{N} \cap (\sqrt{C_j/C_s}, \infty)$. Dieser Vektor besitzt das Höhenquadrat

$$c_s(v_s,\ldots,v_k) = C_s * v_s^2 > C_s \frac{C_j}{C_s} = C_j$$

und kann daher niemals zu einer Verringerung des Höhenquadrates beitragen.

- Er hat die Form

$$\begin{aligned} v &= (0^{s-j}, v_s, v_{s+1}, \ldots, v_k, x) \\ \text{mit} \quad (v_{s+1}, \ldots, v_k) &\neq 0^{k-s}, \\ x &= c_{s+1}(v_{s+1}, \ldots, v_k) \\ \text{und } |v_s - y(0^{s-j+1}, v_{s+1}, \ldots, v_k)| &> vzw. \end{aligned}$$

Das neue Höhenquadrat dieses Vektors berechnet sich zu

$$\begin{aligned} c_s(v_s, \ldots, v_k) &= x + (y(0^{s-j+1}, v_{s+1}, \ldots, v_k, x) + v_s)^2 C_s \\ &> x + vzw^2 C_s \\ &= x + \left\lfloor \sqrt{\frac{C_j - \min_{v \in \mathbb{V}} c_{s+1}(v_{s+1}, \ldots, v_k)}{C_s}} + 1 \right\rfloor^2 C_s \\ &> x + \left(\sqrt{\frac{C_j - \min_{v \in \mathbb{V}} c_{s+1}(v_{s+1}, \ldots, v_k)}{C_s}} \right)^2 C_s \\ &= x + \frac{C_j - \min_{v \in \mathbb{V}} c_{s+1}(v_{s+1}, \ldots, v_k)}{C_s} C_s \\ &= x + C_j - \min_{v \in \mathbb{V}} c_{s+1}(v_{s+1}, \ldots, v_k) \\ &\geq C_j \end{aligned}$$

und muß daher nicht aufgezählt werden.

Ein Vektor wird nur dann aus der Menge $\mathbb{V}$ entfernt, wenn sein bisheriges Höhenquadrat schon zu groß ist. Da die Höhenquadrate auf jedem Pfad von der Wurzel zu einem Blatt monoton wachsen, können hierdurch keinerlei Vektoren verloren gehen, die zu einem Minimum führen. Somit liegen nach der While-Schleife alle Vektoren v mit ihrem Höhenquadrat $c_j(v_j, \ldots, v_k) \leq C_j$ vor.

zu ii. Es werden nur Vektoren mit mindestens einem von Null verschiedenen Koeffizienten in die Vektormenge aufgenommen, daher wird der Nullvektor nie aufgezählt.

zu iii. Da innerhalb der While-Schleife die Stufe s stets erniedrigt wird und die Mächtigkeit der Vektormenge endlich ist, terminiert der Algorithmus auch.

Die Implementierung unterscheidet sich leicht von dem obigen Algorithmus. DieVektormenge wird als Matrix mit β Spalten und einer hohen Anzahl (=: l) von Zeilen realisiert. Zusätzlich existieren zwei Vektoren der Länge l, in denen die zugehörigen Höhenquadrate und Werte für y gespeichert werden. Nachdem die entsprechenden Kopien der Koeffizienten angelegt worden sind, werden mit $\lceil -y(v) \pm (i - 0.5) \rfloor \quad i = 1, \ldots, vzw$ die nächsten $2 \times vzw$ ganzen Zahlen zu $-y(v)$ erzeugt.

Da die Anzahl der Vektoren bei größeren Blockweiten jeden bisher verfügbaren Hauptspeicher sprengt, muß man Teile dieser Matrix auf Massenspeicher auslagern, um später wieder an diesem Punkt aufzusetzen. Falls man inzwischen ein neues Minimum gefunden hat und noch ausgelagerte Daten verarbeiten muß, kann man im obigen Algorithmus alle Werte von

C_j durch dieses neue Minimum ersetzen. Dadurch wird die Anzahl der noch aufzuzählenden Vektoren verringert.
Wenn $\sqrt{\frac{C_j - \min_{v \in \mathbb{V}} c_{s+1}(v_{s+1}, \ldots, v_k)}{C_s}} < 1/2$ gilt, müssen auch keine mehrfachen Kopien der Vektoren angelegt werden. In diesem Fall ergibt sich $v_s = \lceil -y(v) \rfloor$, womit umfangreiche Hauptspeicherbewegungen verhindert werden.

Da bei diesem Verfahren ebenfalls alle möglichen Linearkombinationen aufgezählt werden, ist es für größere Blockweiten nicht mehr effizient anwendbar. Die maximale Blockgröße hängt hierbei von dem zur Verfügung stehenden Hauptspeicher und der Geschwindigkeit des verwendeten Massenspeichers ab. Bei meinen Testläufen, deren Ergebnisse im nächsten Kapitel vorgestellt werden, wurde mit Blockweiten $\beta \leq 30$ reduziert. Der Hauptspeicheraufwand betrug fast 200 MBytes, und es wurden nur selten Teilmatrizen auf Massenspeicher ausgelagert.

Der Algorithmus SchnittEnum realisiert eine geschnittene Aufzählung über einen Block von Basisvektoren $b_j, \ldots, b_k$. Als Eingabe erhält er die Blockgrenzen j, k, die Höhenquadrate C_i und die benötigten Gram-Schmidt-Koeffizienten. Der Eingabeparameter *mem* legt fest, wieviele Knoten gleichzeitig auf einer Stufe maximal betrachtet werden. Die Funktion SchnittEnum ruft die Funktion Enump, die die eigentliche Aufzählung durchführt, mit Blockweiten von 3 bis $k + 1 - j$ auf. Aus den Rückgabewerten von Enump bestimmt sie das Minimum.
Die Funktion Enump mit Eingabeparameter j, k zählt nur Koeffizientenvektoren mit $\tilde{u}_k > 0$ auf. In der Initialisierung werden diese Elemente bereits in die Knotenmenge eingefügt. Innerhalb die Schleife über die Stufen s werden die Knoten mit zu großem Höhenquadrat aus $\mathbb{V}$ entfernt. Hierbei können auch Schnittmethoden, wie sie in sequentiellen Algorithmen verwendet werden, angewandt werden. Danach wird berechnet, wieviele Kinder für jeden Knoten betrachtet werden sollen. Die Knotenmenge wird sortiert, und die $mem/(2 \times vzw)$ Elemente mit den kleinsten Höhenquadraten werden weiterhin betrachtet; alle übrigen Knoten werden verworfen.
Anschließend werden die Knoten vervielfacht, die Koeffizienten $\tilde{u}_s$ eingetragen und die neuen Höhenquadrate berechnet.
Nach dem letzten Schleifendurchlauf wird der Knoten mit dem kleinsten Höhenquadrat zurückgegeben.
In meinen Testläufen war $vzw \leq 2$, wenn geschnitten werden mußte. Es ist jedoch auch möglich, vzw durch eine Konstante zu beschränken.

Algorithmus SchnittEnum(j,k,mem) mit Schnitt:

FOR $i = j+2,\ldots,k$ DO
  CALL Enump(j,i,mem)
ENDFOR
Bestimme das Minimum aller Rückgaben

Algorithmus Enump(j,k,mem)

Eingabe: $\mu_{i,l}\ \ j \le i < l \le k$, $C_i = \|\hat{b}_i\|^2 i = j,\ldots,k$

/* Initialisierung */
$s := k$
Vektormenge $\mathbb{V} := \emptyset$
Füge Vektoren $v^{(i)}$ der Form $(0^{k-j}, i, i^2 C_k) \quad i \in \mathbb{N} \cup [1, \sqrt{C_k/C_j}]$ in $\mathbb{V}$ ein
/* Die Schleife über die Stufen */
WHILE $s > j$ DO
  $s := s-1$
  Wende Schnittstrategien auf $\mathbb{V}$ an
  Eliminiere alle Vektoren $v \in \mathbb{V}$ mit $c_{s+1}(v_{s+1},\ldots,v_k) > C_j$
  $vzw := \lfloor \sqrt{\frac{C_j - \min_{v\in\mathbb{V}} c_{s+1}(v_{s+1},\ldots,v_k)}{C_s}} + 1 \rfloor$
  Sortiere $\mathbb{V}$ nach $c_{s+1}(v_{s+1},\ldots,v_k)$ in aufsteigender Reihenfolge
  Lösche die Vektoren $v^{(l)}$ mit $l > mem/(2 * vzw)$
  FOR $v = (0^{s+1-j}, v_{s+1},\ldots,v_k, c_{s+1}(v_{s+1},\ldots,v_k)) \in \mathbb{V}$ DO
    $y(v) := \sum_{i=s+1}^{k} v_i \mu_{i,s}$
    Erstelle $2 * vzw$ Kopien von v mit
      $v_s := \lceil -y(v) \pm (i-0.5) \rfloor \quad i = 1,\ldots,vzw$
    Berechne $c_s(v_s,\ldots,v_k) := c_{s+1}(v_{s+1},\ldots,v_k) + (v_s - y(v))^2 C_s$
    Aktualisiere Schnittinformation
  ENDFOR
ENDWHILE
Bestimme $\min_{v\in V} c_j(v_j,\ldots,v_k)$ und zugehörigen Vektor $v = (v_j,\ldots,v_k, c_j(v_j,\ldots,v_k))$

Ausgabe: $v \in (\mathbb{Z}^{k+1-j} \times \mathbb{R}) \setminus 0$

Anmerkung: Die Knoten v sind numeriert; $v^{(l)}$ ist der bezüglich dieser Numerierung $l-$te Vektor. Untere Indizes (v_s) bezeichnen die entsprechenden Komponenten des Koeffizientenvektors.

6. Ergebnisse und Bewertung

6.1 Allgemeine Rucksackprobleme

Mit Hilfe der im letzten Kapitel vorgestellten Algorithmen wurde versucht, allgemeine Rucksackprobleme zu lösen. Dabei wurde die folgende Gitterbasis B mit $c = n^2$ verwendet:

$$B = \begin{array}{c} \begin{matrix} b_0 & b_1 & b_2 & \cdots & b_{n-1} & b_n \end{matrix} \\ \begin{pmatrix} q & 0 & 0 & \cdots & 0 & 0 \\ q & n & 0 & \cdots & \cdots & 0 \\ q & 0 & n & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ q & \vdots & & \ddots & n & 0 \\ q & 0 & \cdots & \cdots & 0 & n \\ cs & ca_1 & ca_2 & \cdots & ca_{n-1} & ca_n \\ cq & c & c & \cdots & c & c \end{pmatrix} \end{array} \in \mathbb{Z}^{(n+1)\times(n+3)}$$

Hierbei war n die Dimension des Rucksackproblems und q die Anzahl der Gewichte, die in die Summe eingingen.
Darauf wurde der folgende Algorithmus angewendet:

Algorithmus zur Lösung von Rucksackproblemen

Eingabe: Gitterbasis $b_0, \ldots, b_n \in \mathbb{Z}^{n+3}$

1. LLL-Reduktion

Entferne die letzten zwei Vektoren

und die letzten zwei Komponenten jedes Vektors

`FOR` Runde $= 1, \ldots, 16$ `DO`

2. Permutiere die Basis zufällig
3. Blockreduktion mit $\beta = 10(20, 30)$
4. SchnittEnum$(0, n-2, 2048 * n)$

`ENDFOR`

Ausgabe: Gesamtzeit, LLL-Zeit, Zeiten der Aufzählungsverfahren

Die LLL-Reduktion wurde sowohl zu Beginn des Algorithmus als auch während der Blockreduktion stets mit tiefen Einfügungen bis zur Position 5 durchgeführt. Desweiteren wurde nach jeder Längenreduktion getestet, ob der entstandene Vektor eine Lösung des Rucksackproblems darstellt. In diesem Fall stoppte der Algorithmus sofort. Nach der anfänglichen LLL-Reduktion sind im allgemeinen nur von den letzten zwei Vektoren die letzten zwei Komponenten von Null verschieden. Besitzt diese 2×2-Untermatrix eine nichtverschwindende Determinante, können diese Vektoren nicht in die Lösung eingehen, da keine Linearkombination mit diesen Vektoren in den letzten zwei Komponenten verschwindet. Daher können diese zwei Vektoren gelöscht werden. Andernfalls wäre eine Fehlermeldung ausgegeben worden; dies ist jedoch niemals aufgetreten.
Die Permutation wurde so durchgeführt, daß galt:

$$\forall i, j \quad b_{i,0} = 0 \wedge b_{j,0} \neq 0 \Rightarrow j < i$$

d.h. die Vektoren mit Anteilen des ursprünglichen Vektors b_0 standen am Anfang der Basis.
Die Blockreduktion benutzte das ungeschnittene Aufzählungsverfahren.
Da die Lösung des Rucksackproblems nicht immer durch den kürzesten Gittervektor repräsentiert wird, wurde in den Funktionen SchnittEnum und Enump der Schnittenumeration die Minimumbestimmung durch eine minimale Abstandsbestimmung zur Norm des Lösungsvektors ersetzt. Hierdurch wurden bei Dichten > 1 deutlich mehr Lösungen gefunden.
Verglichen werden meine Ergebnisse mit denen von Hörner aus [H94]; diese sind eine Verbesserung der Ergebnisse aus [SE91]. Jede Zeile gibt den Durchschnitt von 20 zufällig erzeugten lösbaren Problemen an. Da sich sowohl die Art des verwendeten Rechners als auch der Algorithmus veränderten, wurden die vektorisierbaren Programme auch auf einer skalaren Maschine gestartet. Es handelte sich hierbei um einen von der Firma IBM hergestellten Server vom Typ 590H. Ein Vergleich von Programmlaufzeiten ergab, daß dieser Computer etwa um den Faktor 2.3 schneller ist als der von Hörner in [H94] verwendete HP 9000/710. Entsprechend sind diese Laufzeitangaben um diesen Faktor verringert worden. Die Zeiten der IBM 590H und der S200 wurden nicht umgerechnet.
Die obere Tabelle der jeweils folgenden Seiten zeigt somit die Ergebnisse und Laufzeiten aus [H94], wenn die Programme auf einer IBM 590H gelaufen wären. In den mittleren Tabellen finden sich die Ergebnisse und Laufzeiten der vektorisierbaren Programme auf dieser skalaren Maschine. Ein Vergleich dieser Tabellen zeigt den Einfluß der neuen Algorithmen auf Lösungswahrscheinlichkeit und Laufzeitverhalten.
Die untere Tabelle auf jeder Seite zeigt die Ergebnisse der Programmläufe auf dem Vektorrechner SNI S200. Ein Vergleich dieser Tabelle mit der mittleren auf der gleichen Seite zeigt den Geschwindigkeitsgewinn durch die Verwendung der Vektorpipelines.

Die einzelnen Spalten der Tabellen bedeuten:

n :	Dimension des Rucksackproblems
$bitl$:	maximale Bitlänge der Rucksackgewichte
E :	Anzahl der gelösten Probleme
E_1 :	Anzahl der in der ersten Runde gelösten Probleme
Runden:	Gesamtrundenzahl für alle Probleme ($\in [20, 320]$)
LLL:	Anzahl der im LLL gefundenen Lösungen
BKZR:	Anzahl der in der Blockreduktion gefundenen Lösungen
SE:	Anzahl der vom SchnittEnum gefundenen Lösungen
Zeit:	über alle 20 Probleme gemittelte Zeit
Zeit o. SE:	dito. aber ohne die SchnittEnums
Zeit / SE:	durchschnittliche Laufzeit eines SchnittEnums

Ergebnisse für Dimension 66 und Blockweite 10

Hörner (HP 9000/710 Zeiten transformiert)

n	$bitl$	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
66	18	20	20	20	20	0	0	00:00:01	00:00:01	00:00:00
66	26	20	20	20	20	0	0	00:00:02	00:00:02	00:00:00
66	34	20	16	27	6	14	0	00:00:10	00:00:10	00:00:00
66	42	20	5	81	0	20	0	00:00:33	00:00:33	00:00:00
66	50	4	0	300	0	4	0	00:01:47	00:01:47	00:00:00
66	58	3	0	303	0	0	3	00:18:05	00:01:52	00:01:04
66	66	19	12	48	0	0	19	00:15:58	00:00:38	00:06:23
66	72	20	17	23	0	0	20	00:03:18	00:00:35	00:02:31
66	80	20	18	25	0	5	15	00:00:53	00:00:42	00:00:11
66	88	20	18	23	0	12	8	00:00:40	00:00:39	00:00:01
66	96	20	19	21	1	16	3	00:00:34	00:00:34	00:00:00
66	104	20	18	22	1	18	1	00:00:34	00:00:34	00:00:00
66	112	20	20	20	6	14	0	00:00:31	00:00:31	00:00:00

neu (IBM)

n	$bitl$	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
66	18	20	20	20	20	0	0	00:00:01	00:00:01	00:00:00
66	26	20	20	20	20	0	0	00:00:01	00:00:01	00:00:00
66	34	20	20	20	6	10	4	00:22:22	00:00:04	01:51:33
66	42	20	20	20	1	5	14	00:58:24	00:00:05	01:23:18
66	50	20	19	21	0	0	20	01:05:53	00:00:07	01:02:38
66	58	20	14	29	0	0	20	01:26:18	00:00:09	00:59:25
66	66	20	15	47	0	0	20	02:10:29	00:00:15	00:56:37
66	72	20	19	21	0	0	20	00:31:49	00:00:13	00:30:06
66	80	20	20	20	0	4	16	00:11:38	00:00:12	00:14:15
66	88	20	19	21	0	11	9	00:04:34	00:00:13	00:08:43
66	96	20	20	20	2	9	9	00:02:47	00:00:11	00:05:43
66	104	20	20	20	6	12	2	00:00:15	00:00:11	00:00:44
66	112	20	20	20	4	15	1	00:00:14	00:00:01	00:00:43

neu (SNI S200)

n	$bitl$	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
66	18	20	20	20	20	0	0	00:00:00	00:00:00	00:00:00
66	26	20	20	20	20	0	0	00:00:01	00:00:01	00:00:00
66	34	20	20	20	6	12	2	00:01:47	00:00:02	00:17:35
66	42	20	20	20	1	4	15	00:09:48	00:00:03	00:12:59
66	50	20	20	20	0	0	20	00:09:37	00:00:04	00:09:32
66	58	20	15	26	0	0	20	00:11:33	00:00:05	00:08:50
66	66	20	14	33	0	1	19	00:13:56	00:00:07	00:08:38
66	72	20	19	25	0	1	19	00:05:34	00:00:07	00:04:32
66	80	20	20	20	0	4	16	00:01:46	00:00:07	00:02:03
66	88	20	18	22	0	12	8	00:00:48	00:00:06	00:01:23
66	96	20	20	20	2	11	7	00:00:25	00:00:06	00:00:54
66	104	20	20	20	6	13	1	00:00:06	00:00:05	00:00:14
66	112	20	20	20	4	15	1	00:00:06	00:00:06	00:00:07

Ergebnisse für Dimension 66 und Blockweite 20

Hörner (HP 9000/710 Zeiten transformiert)

n	*bitl*	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
66	18	20	20	20	20	0	0	00:00:01	00:00:01	00:00:00
66	26	20	20	20	20	0	0	00:00:02	00:00:02	00:00:00
66	34	20	16	21	6	14	0	00:00:09	00:00:09	00:00:00
66	42	20	5	63	0	20	0	00:00:52	00:00:52	00:00:00
66	50	10	0	213	0	10	0	00:02:48	00:02:48	00:00:00
66	58	9	0	233	0	8	1	00:07:26	00:04:02	00:00:18
66	66	20	12	24	0	4	16	00:05:41	00:01:30	00:04:11
66	72	20	17	26	0	3	17	00:02:58	00:01:45	00:01:04
66	80	20	18	26	0	14	6	00:01:10	00:01:05	00:00:08
66	88	20	18	20	0	18	2	00:00:52	00:00:52	00:00:00
66	96	20	19	20	1	19	0	00:00:32	00:00:32	00:00:00
66	104	20	18	20	1	19	0	00:00:32	00:00:32	00:00:00
66	112	20	20	20	6	14	0	00:00:30	00:00:30	00:00:00

neu (IBM)

n	*bitl*	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
66	18	20	20	20	20	0	0	00:00:01	00:00:01	00:00:00
66	26	20	20	20	20	0	0	00:00:01	00:00:01	00:00:00
66	34	20	20	20	6	13	1	00:05:22	00:00:04	01:55:57
66	42	20	20	20	1	8	11	00:46:20	00:00:09	01:23:58
66	50	20	20	20	0	3	17	00:51:47	00:00:20	01:00:32
66	58	20	17	25	0	0	20	01:03:29	00:00:24	00:50:28
66	66	20	9	54	0	2	18	02:24:46	00:00:46	00:55:23
66	72	20	14	37	0	2	18	00:57:12	00:00:42	00:32:17
66	80	20	20	20	0	12	8	00:04:30	00:00:22	00:10:19
66	88	20	20	20	0	19	1	00:00:43	00:00:16	00:08:49
66	96	20	20	20	2	18	0	00:00:13	00:00:13	00:00:00
66	104	20	20	20	6	13	1	00:00:13	00:00:12	00:19:20
66	112	20	20	20	4	16	0	00:00:12	00:00:12	00:00:00

neu (SNI S200)

n	*bitl*	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
66	18	20	20	20	20	0	0	00:00:00	00:00:00	00:00:00
66	26	20	20	20	20	0	0	00:00:01	00:00:01	00:00:00
66	34	20	20	20	6	13	1	00:00:51	00:00:02	00:16:20
66	42	20	20	20	1	7	12	00:07:55	00:00:05	00:13:02
66	50	20	20	20	0	2	18	00:08:33	00:00:09	00:09:19
66	58	20	17	25	0	0	20	00:10:01	00:00:12	00:07:51
66	66	20	11	43	0	2	18	00:16:58	00:00:19	00:08:07
66	72	20	14	34	0	3	17	00:07:55	00:00:19	00:04:55
66	80	20	19	23	0	14	6	00:01:00	00:00:11	00:01:50
66	88	20	20	20	0	19	1	00:00:11	00:00:07	00:01:20
66	96	20	20	20	2	18	0	00:00:07	00:00:07	00:00:00
66	104	20	20	20	6	13	1	00:00:06	00:00:06	00:00:03
66	112	20	20	20	4	16	0	00:00:06	00:00:06	00:00:00

Ergebnisse für Dimension 66 und Blockweite 30

Hörner (HP 9000/710 Zeiten transformiert)

n	$bitl$	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
66	18	20	20	20	20	0	0	00:00:01	00:00:01	00:00:00
66	26	20	20	20	20	0	0	00:00:02	00:00:02	00:00:00
66	34	20	20	20	6	14	0	00:00:10	00:00:10	00:00:00
66	42	20	20	20	0	20	0	00:00:52	00:00:52	00:00:00
66	50	18	10	76	0	18	0	00:04:28	00:04:28	00:00:00
66	58	19	7	83	0	19	0	00:11:41	00:11:41	00:00:00
66	66	20	16	26	0	12	8	00:09:45	00:07:10	00:03:42
66	72	20	18	22	0	17	3	00:04:25	00:04:06	00:01:14
66	80	20	20	20	0	19	1	00:02:04	00:02:04	00:00:05
66	88	20	20	20	0	20	0	00:00:58	00:00:58	00:00:00
66	96	20	20	20	1	19	0	00:00:42	00:00:42	00:00:00
66	104	20	20	20	1	19	0	00:00:37	00:00:37	00:00:00
66	112	20	20	20	6	14	0	00:00:35	00:00:35	00:00:00

neu (IBM)

n	$bitl$	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
66	18	20	20	20	20	0	0	00:00:01	00:00:01	00:00:00
66	26	20	20	20	20	0	0	00:00:01	00:00:01	00:00:00
66	34	20	20	20	6	14	0	00:00:13	00:00:13	00:00:00
66	42	20	20	20	1	17	2	00:10:23	00:01:39	01:27:21
66	50	20	20	20	0	13	7	00:24:23	00:05:27	00:54:05
66	58	20	18	23	0	12	8	00:37:12	00:13:50	00:42:28
66	66	20	16	33	0	11	9	01:08:16	00:16:48	00:46:47
66	72	20	20	20	0	18	2	00:12:44	00:09:47	00:29:31
66	80	20	20	20	0	20	0	00:03:34	00:03:34	00:00:00
66	88	20	20	20	0	20	0	00:01:38	00:01:38	00:00:00
66	96	20	20	20	2	18	0	00:01:40	00:01:40	00:00:00
66	104	20	20	20	6	14	0	00:00:36	00:00:36	00:00:00
66	112	20	20	20	4	16	0	00:00:44	00:00:44	00:00:00

neu (SNI S200)

n	$bitl$	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
66	18	20	20	20	20	0	0	00:00:00	00:00:00	00:00:00
66	26	20	20	20	20	0	0	00:00:01	00:00:01	00:00:00
66	34	20	20	20	6	14	0	00:00:03	00:00:03	00:00:00
66	42	20	20	20	1	18	1	00:01:01	00:00:16	00:14:53
66	50	20	20	20	0	11	9	00:05:12	00:01:19	00:08:38
66	58	20	18	23	0	13	7	00:06:04	00:02:41	00:06:45
66	66	20	15	39	0	8	12	00:14:56	00:03:42	00:07:15
66	72	20	20	20	0	19	1	00:02:04	00:01:49	00:04:49
66	80	20	20	20	0	20	0	00:00:52	00:00:52	00:00:00
66	88	20	20	20	0	20	0	00:00:20	00:00:20	00:00:00
66	96	20	20	20	2	18	0	00:00:22	00:00:22	00:00:00
66	104	20	20	20	6	14	0	00:00:10	00:00:10	00:00:00
66	112	20	20	20	4	16	0	00:00:11	00:00:11	00:00:00

Ergebnisse für Dimension 74 und Blockweite 10

Hörner (HP 9000/710 Zeiten transformiert)

n	*bitl*	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
74	26	20	20	20	19	1	0	00:00:03	00:00:03	00:00:00
74	34	20	17	23	10	10	0	00:00:09	00:00:09	00:00:00
74	42	20	4	85	0	20	0	00:00:48	00:00:48	00:00:00
74	50	7	0	279	0	7	0	00:02:26	00:02:26	00:00:00
74	58	1	0	316	0	1	0	00:03:44	00:02:54	00:00:03
74	66	6	0	275	0	0	6	17:51:02	00:02:46	01:17:42
74	74	20	9	44	0	0	20	10:40:58	00:00:58	04:50:28
74	82	20	14	37	0	0	20	04:28:54	00:01:06	01:02:36
74	90	20	18	23	0	1	19	00:11:51	00:01:03	00:09:50
74	98	20	18	23	0	5	15	00:01:53	00:01:12	00:00:46
74	106	20	17	23	0	6	14	00:01:22	00:01:18	00:00:05
74	114	20	18	24	0	9	11	00:01:21	00:01:20	00:00:01
74	122	20	19	21	0	19	1	00:01:11	00:01:11	00:00:00

neu (IBM)

n	*bitl*	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
74	26	20	20	20	19	1	0	00:00:01	00:00:01	00:00:00
74	34	20	20	20	5	13	2	00:20:36	00:00:04	03:25:22
74	42	20	20	20	0	2	18	02:29:43	00:00:07	02:46:12
74	50	20	20	20	0	0	20	02:14:17	00:00:09	02:14:08
74	58	20	18	22	0	0	20	01:56:30	00:00:11	01:45:44
74	66	15	3	172	0	0	15	16:18:12	00:00:39	01:54:20
74	74	12	4	190	0	0	12	19:41:16	00:00:44	02:04:16
74	82	20	6	75	0	0	20	04:09:28	00:00:28	01:06:24
74	90	20	10	44	0	0	20	01:29:11	00:00:26	00:40:20
74	98	20	16	28	0	3	17	00:35:33	00:00:24	00:28:07
74	106	20	19	21	0	8	12	00:13:03	00:00:22	00:19:31
74	114	20	20	20	0	11	9	00:07:15	00:00:23	00:15:14
74	122	20	20	20	0	15	5	00:01:56	00:00:23	00:06:11

neu (SNI S200)

n	*bitl*	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
74	26	20	20	20	19	1	0	00:00:01	00:00:01	00:00:00
74	34	20	20	20	5	13	2	00:03:15	00:00:02	00:32:02
74	42	20	20	20	1	4	15	00:19:14	00:00:04	00:25:33
74	50	20	20	20	0	0	20	00:20:29	00:00:05	00:20:24
74	58	20	19	21	0	0	20	00:17:11	00:00:06	00:16:16
74	66	15	3	177	0	0	15	02:37:00	00:00:22	00:17:42
74	74	13	4	206	0	0	13	03:16:08	00:00:25	00:19:00
74	82	18	6	105	0	0	18	00:52:59	00:00:18	00:10:02
74	90	20	11	35	0	0	20	00:10:44	00:00:12	00:06:01
74	98	20	17	23	0	3	17	00:04:24	00:00:11	00:04:13
74	106	20	20	20	0	8	12	00:01:55	00:00:10	00:02:55
74	114	20	20	20	0	11	9	00:01:09	00:00:11	00:02:10
74	122	20	20	20	0	15	5	00:00:23	00:00:10	00:00:52

Ergebnisse für Dimension 74 und Blockweite 20

Hörner (HP 9000/710 Zeiten transformiert)

n	*bitl*	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
74	26	20	20	20	19	1	0	00:00:03	00:00:03	00:00:00
74	34	20	18	22	10	10	0	00:00:09	00:00:09	00:00:00
74	42	20	8	63	0	20	0	00:01:03	00:01:03	00:00:00
74	50	10	3	218	0	10	0	00:03:55	00:03:55	00:00:00
74	58	4	1	271	0	4	0	00:05:57	00:05:57	00:00:00
74	66	3	0	283	0	0	3	07:14:18	00:07:37	00:30:09
74	74	20	12	39	0	0	20	06:02:01	00:03:05	03:04:04
74	82	20	18	24	0	3	17	00:25:07	00:03:39	00:20:27
74	90	20	18	22	0	7	13	00:05:01	00:03:14	00:02:23
74	98	20	20	20	0	12	8	00:02:42	00:02:37	00:00:12
74	106	20	20	20	0	19	1	00:01:50	00:01:50	00:00:00
74	114	20	20	20	0	20	0	00:01:18	00:01:18	00:00:00
74	122	20	20	20	0	20	0	00:01:05	00:01:05	00:00:00

neu (IBM)

n	*bitl*	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
74	26	20	20	20	19	1	0	00:00:01	00:00:01	00:00:00
74	34	20	20	20	5	14	1	00:10:20	00:00:05	03:24:59
74	42	20	20	20	0	9	11	01:30:58	00:00:12	02:45:01
74	50	20	20	20	0	3	17	01:53:39	00:00:25	02:13:13
74	58	20	20	20	0	0	20	01:44:04	00:00:31	01:43:33
74	66	18	7	95	0	0	18	08:31:54	00:01:21	01:47:29
74	74	19	3	126	0	0	19	12:16:18	00:02:09	01:56:32
74	82	20	11	38	0	3	17	01:44:03	00:01:20	00:58:42
74	90	20	18	22	0	8	12	00:25:38	00:01:04	00:35:05
74	98	20	19	21	0	14	6	00:10:14	00:00:46	00:27:03
74	106	20	20	20	0	19	1	00:01:14	00:00:37	00:12:31
74	114	20	20	20	0	20	0	00:00:28	00:00:28	00:00:00
74	122	20	20	20	0	20	0	00:00:24	00:00:24	00:00:00

neu (SNI S200)

n	*bitl*	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
74	26	20	20	20	19	1	0	00:00:01	00:00:01	00:00:00
74	34	20	20	20	5	15	0	00:00:02	00:00:02	00:00:00
74	42	20	20	20	1	9	10	00:12:52	00:00:05	00:25:34
74	50	20	20	20	0	2	18	00:18:29	00:00:10	00:20:22
74	58	20	20	20	0	0	20	00:16:03	00:00:13	00:15:50
74	66	19	3	107	0	0	19	01:29:20	00:00:42	00:16:34
74	74	18	4	119	0	0	18	01:46:18	00:00:55	00:17:43
74	82	20	12	38	0	1	19	00:15:39	00:00:35	00:08:09
74	90	20	18	22	0	8	12	00:04:13	00:00:26	00:05:24
74	98	20	19	21	0	12	8	00:02:14	00:00:20	00:03:48
74	106	20	20	20	0	18	2	00:00:29	00:00:16	00:02:12
74	114	20	20	20	0	20	0	00:00:11	00:00:11	00:00:00
74	122	20	20	20	0	20	0	00:00:10	00:00:10	00:00:00

Ergebnisse für Dimension 74 und Blockweite 30

Hörner (HP 9000/710 Zeiten transformiert)

n	*bitl*	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
74	26	20	20	20	19	1	0	00:00:03	00:00:03	00:00:00
74	34	20	20	20	10	10	0	00:00:15	00:00:15	00:00:00
74	42	20	18	27	0	20	0	00:01:31	00:01:31	00:00:00
74	50	18	7	92	0	18	0	00:07:17	00:07:17	00:00:00
74	58	7	3	245	0	7	0	00:25:29	00:25:29	00:00:00
74	66	11	4	189	0	9	2	03:58:11	00:48:53	00:21:02
74	74	19	11	54	0	4	15	05:33:17	00:28:58	02:01:44
74	82	20	19	21	0	18	2	00:18:20	00:16:40	00:11:08
74	90	20	20	20	0	20	0	00:10:21	00:10:21	00:00:00
74	98	20	20	20	0	20	0	00:03:22	00:03:22	00:00:00
74	106	20	20	20	0	20	0	00:02:14	00:02:14	00:00:00
74	114	20	20	20	0	20	0	00:01:27	00:01:27	00:00:00
74	122	20	20	20	0	20	0	00:01:10	00:01:10	00:00:00

neu (IBM)

n	*bitl*	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
74	26	20	20	20	19	1	0	00:00:01	00:00:01	00:00:00
74	34	20	20	20	5	15	0	00:00:16	00:00:16	00:00:00
74	42	20	20	20	0	17	3	00:27:48	00:01:51	02:53:03
74	50	20	20	20	0	6	14	01:39:05	00:08:16	02:09:44
74	58	20	20	20	0	4	16	01:36:33	00:16:02	01:40:38
74	66	17	12	80	0	8	9	07:00:14	01:11:47	01:36:48
74	74	20	8	74	0	8	12	07:19:48	01:20:19	01:55:54
74	82	20	14	38	0	16	4	01:44:27	00:39:35	00:58:58
74	90	20	19	22	0	18	2	00:25:54	00:19:27	00:32:17
74	98	20	20	20	0	20	0	00:07:41	00:07:41	00:00:00
74	106	20	20	20	0	20	0	00:04:08	00:04:08	00:00:00
74	114	20	20	20	0	20	0	00:02:36	00:02:36	00:00:00
74	122	20	20	20	0	20	0	00:02:08	00:02:08	00:00:00

neu (SNI S200)

n	*bitl*	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
74	26	20	20	20	19	1	0	00:00:01	00:00:01	00:00:00
74	34	20	20	20	5	15	0	00:00:05	00:00:05	00:00:00
74	42	20	20	20	1	15	4	00:05:42	00:00:27	00:26:15
74	50	20	20	20	0	8	12	00:13:36	00:01:26	00:20:16
74	58	20	20	20	0	2	18	00:17:17	00:03:33	00:15:15
74	66	18	11	93	0	6	12	01:19:30	00:12:06	00:15:30
74	74	20	11	46	0	5	15	00:46:45	00:13:32	00:16:12
74	82	20	16	26	0	13	7	00:13:10	00:08:09	00:07:43
74	90	20	19	22	0	18	2	00:04:45	00:03:46	00:04:52
74	98	20	20	20	0	20	0	00:02:03	00:02:03	00:00:00
74	106	20	20	20	0	20	0	00:00:55	00:00:55	00:00:00
74	114	20	20	20	0	20	0	00:00:35	00:00:35	00:00:00
74	122	20	20	20	0	20	0	00:00:27	00:00:27	00:00:00

Ergebnisse für Dimension 82 und Blockweite 30

Hörner (HP 9000/710 Zeiten transformiert)

n	$bitl$	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
82	34	20	20	20	9	11	0	00:00:12	00:00:12	00:00:00
82	42	20	12	43	0	20	0	00:02:30	00:02:30	00:00:00
82	50	17	3	128	0	17	0	00:11:14	00:11:14	00:00:00
82	58	4	1	278	0	4	0	00:33:14	00:33:14	00:00:00
82	66	2	0	309	0	2	0	01:03:33	01:03:33	00:00:00
82	74	7	3	230	0	0	7	78:19:31	02:02:12	06:38:02
82	82	20	9	52	0	2	18	70:27:46	01:37:06	27:32:16
82	90	20	15	29	0	9	11	09:23:11	01:07:15	08:11:35
82	98	20	19	21	0	17	3	01:07:28	00:43:37	01:59:14
82	106	20	20	20	0	18	2	00:15:37	00:14:59	00:06:21
82	114	20	20	20	0	19	1	00:08:04	00:08:00	00:01:16
82	122	20	20	20	0	20	0	00:05:57	00:05:57	00:00:00
82	130	20	20	20	0	20	0	00:03:04	00:03:04	00:00:00

neu (IBM)

n	$bitl$	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
82	34	20	19	24	9	11	0	01:16:51	00:00:10	06:23:25
82	42	20	20	20	1	13	6	01:29:23	00:01:35	04:52:41
82	50	20	20	20	0	7	13	02:45:11	00:08:43	04:00:44
82	58	20	20	20	0	2	18	03:13:44	00:19:29	03:13:37
82	66	20	13	33	0	1	19	05:21:19	00:45:43	02:52:15
82	74	14	3	169	0	0	14	27:58:35	03:25:20	02:54:21
82	82	9	2	213	0	3	6	38:22:23	05:45:49	03:08:08
82	90	20	6	84	0	11	9	10:18:32	03:31:09	01:48:38
82	98	20	14	30	0	17	3	02:31:16	01:47:35	01:07:13
82	106	20	19	21	0	20	0	00:39:14	00:35:59	01:05:07
82	114	20	20	20	0	20	0	00:15:48	00:15:48	00:00:00
82	122	20	20	20	0	20	0	00:10:49	00:10:49	00:00:00
82	130	20	20	20	0	20	0	00:07:33	00:07:33	00:00:00

neu (SNI S200)

n	$bitl$	E	E_1	Runden	LLL	BKZR	SE	Zeit	Zeit o. SE	Zeit/SE
82	34	20	19	24	9	11	0	00:12:06	00:00:04	01:00:12
82	42	20	20	20	1	12	7	00:16:16	00:00:28	00:45:07
82	50	20	20	20	0	8	12	00:24:17	00:01:54	00:37:17
82	58	20	20	20	0	1	19	00:32:16	00:03:53	00:29:52
82	66	20	13	38	0	1	19	00:56:11	00:08:15	00:25:55
82	74	15	4	152	0	1	14	03:35:17	00:39:34	00:23:16
82	82	13	4	182	0	5	8	04:40:24	00:59:01	00:25:01
82	90	20	7	79	0	10	10	01:37:16	00:43:52	00:15:29
82	98	20	14	33	0	17	3	00:28:27	00:20:28	00:09:59
82	106	20	19	21	0	20	0	00:09:06	00:08:37	00:09:48
82	114	20	20	20	0	20	0	00:02:54	00:02:54	00:00:00
82	122	20	20	20	0	20	0	00:02:14	00:02:14	00:00:00
82	130	20	20	20	0	20	0	00:01:24	00:01:24	00:00:00

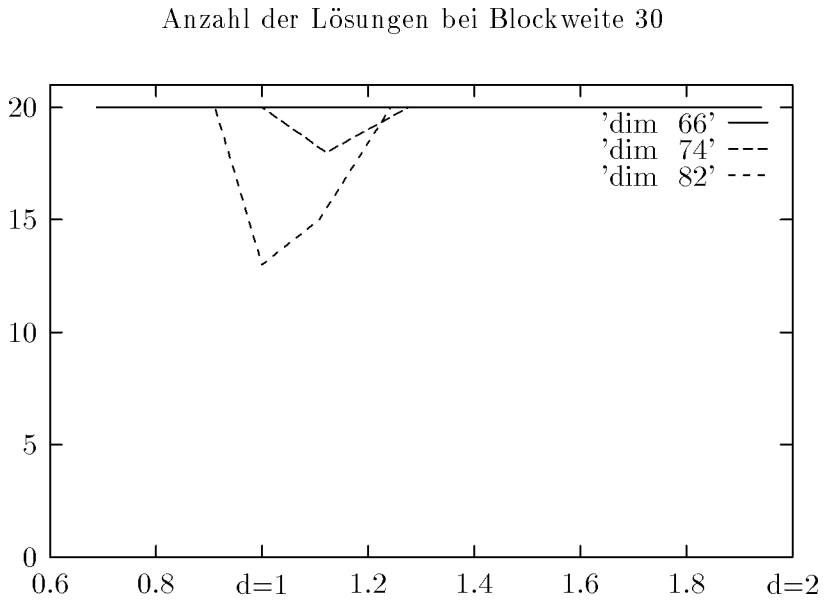

Abb. 6.1: parallele Version (SNI S200)

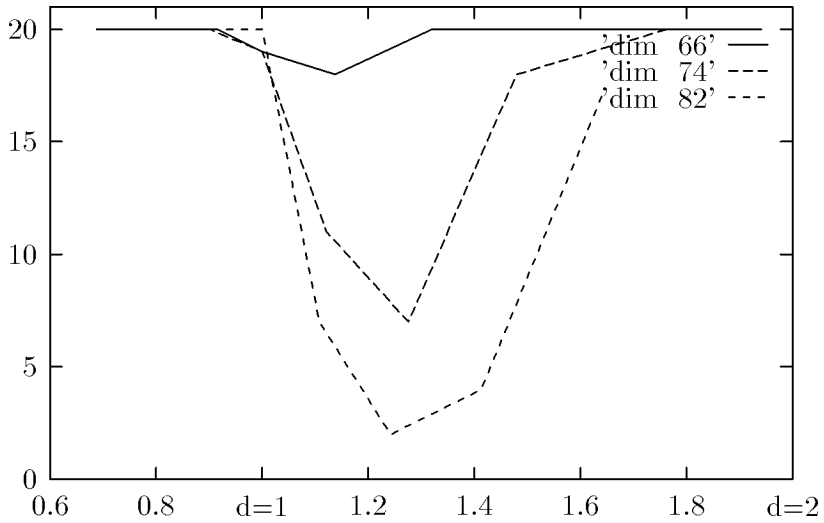

Abb. 6.2: serielle Version

Die vorangegangenen Statistiken zeigen, daß mit der parallelen Version der Algorithmen hohe Erfolgsquoten bei der Lösung allgemeiner Rucksackprobleme erreicht werden können. So werden in den Dimensionen 66 und 74 fast alle Probleme gelöst, lediglich in der Dimension 82 gehen einige Lösungen verloren. Ein Vergleich der Abbildungen 6.1 und 6.2 zeigt, daß dabei immer noch deutlich mehr Lösungen gefunden werden als im seriellen Fall. Wie aus den beiden obigen Abbildungen hervorgeht, ist der Bereich der Dichte, in dem der Algorithmus wenige Lösungen findet, in

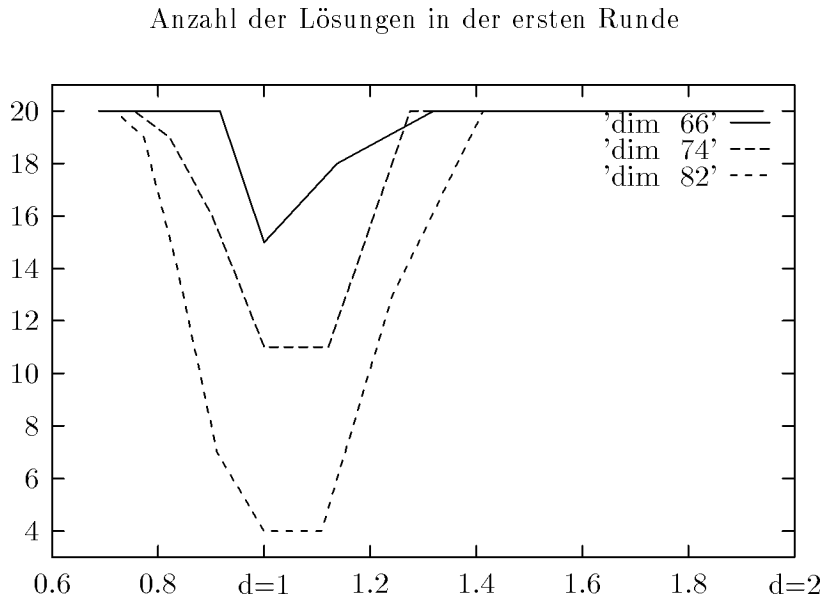

Abb. 6.3: parallele Version (SNI S200)

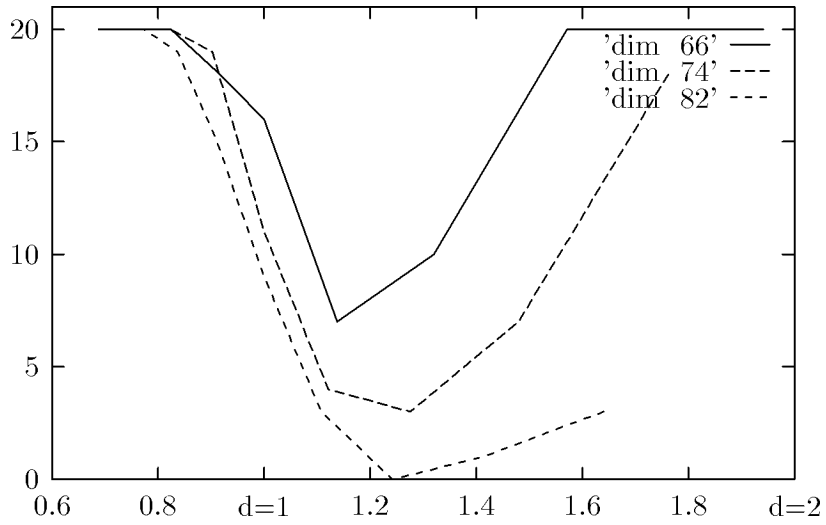

Abb. 6.4: serielle Version

der parallelen Version sichtbar kleiner. Insbesondere erkennt man, daß beide Algorithmen in höheren Dimensionen weniger Lösungen finden. Die Abbildungen lassen schließen, daß der parallele Algorithmus auch noch in höheren Dimensionen angewendet werden kann.
Betrachtet man die Anzahl der Lösungen, die in der ersten Runde gefunden werden (Abbildungen 6.3 und 6.4), so stellt man fest, daß der parallele Algorithmus in der ersten Runde etwa so viele Lösungen findet, wie der serielle insgesamt (Abbildung 6.2).

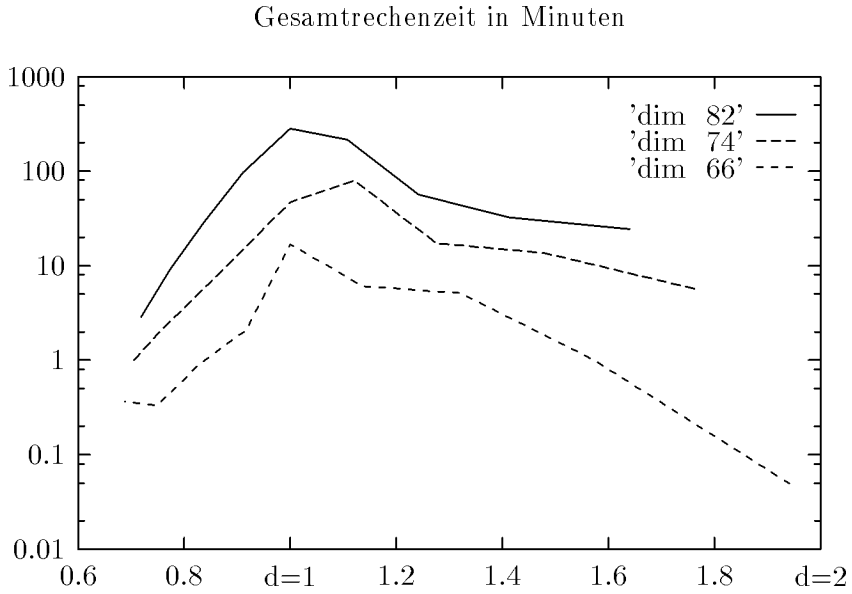

Abb. 6.5: parallele Version (SNI S200)

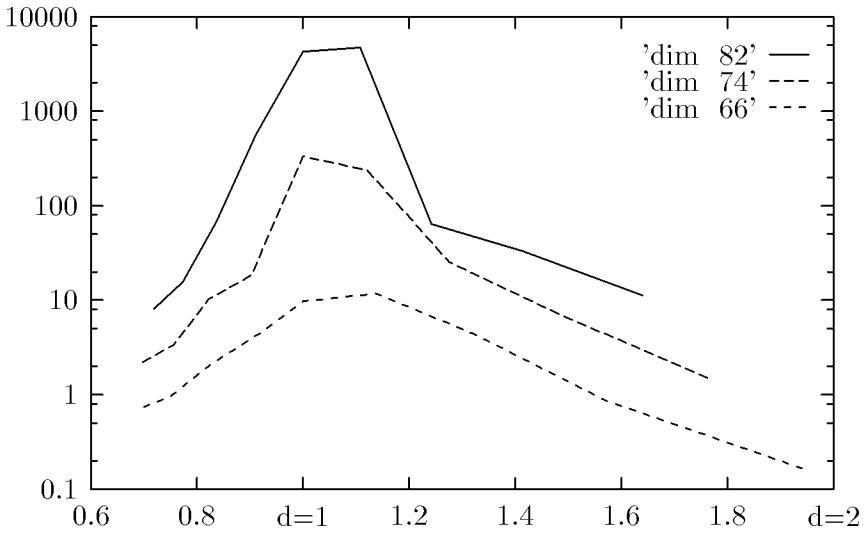

Abb. 6.6: serielle Version

Beide Algorithmen zeigen jedoch ein unterschiedliches Zeitverhalten. In der Implementierung von Hörner bricht die serielle Schlußaufzählung ab, sobald ein Vektor gefunden wurde, der höchstens so lang wie der Lösungsvektor ist. Bei Dichten größer als 1 handelt es sich hierbei meistens um einen bereits von der Blockreduktion berechneten Vektor, der zu kurz ist. So läßt sich bei Dichten größer als 1 kein vernünftiger Zeitvergleich durchführen, da die serielle Schlußaufzählung in diesem Bereich keine nennenswerte Arbeit verrichtet. Die Skala der Dichte reicht deshalb in Abbildung 6.8 nur von 0.7 bis 1.15.

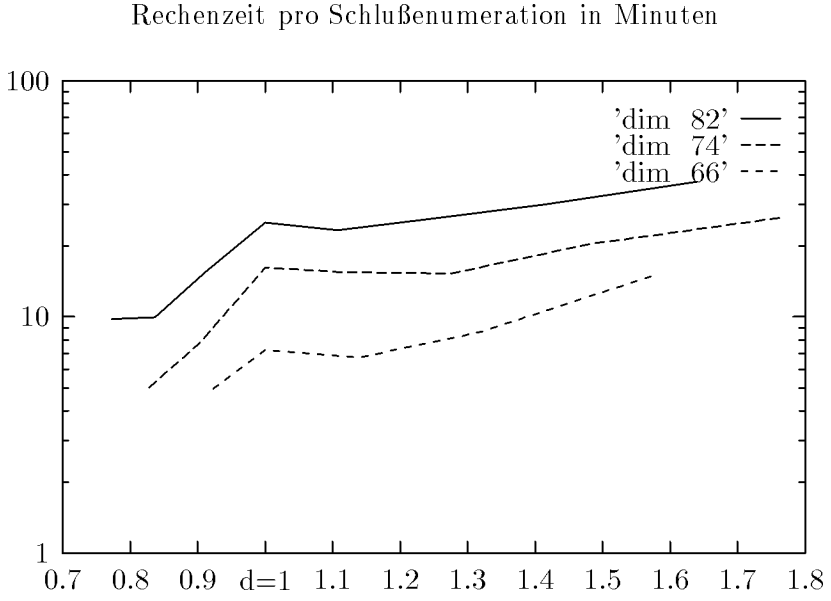

Abb. 6.7: parallele Version (SNI S200)

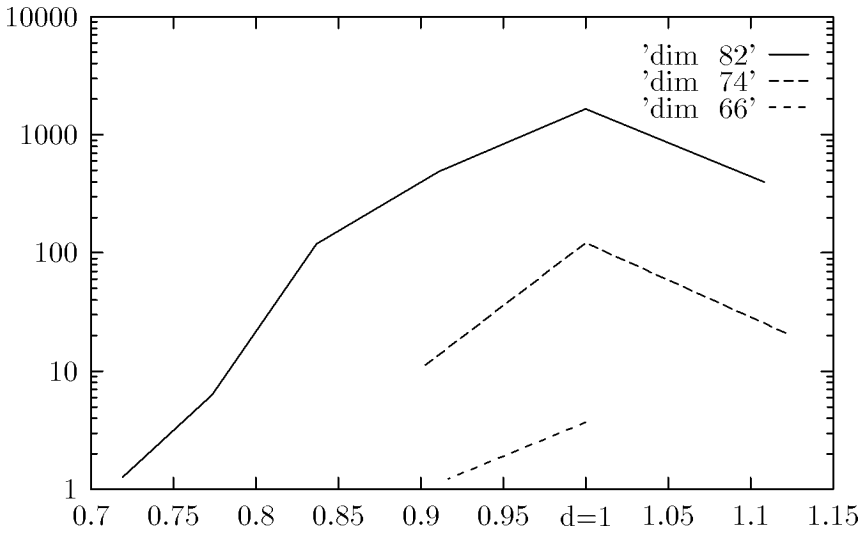

Abb. 6.8: serielle Version

Anhand der Statistiken erkennt man, daß in der Dimension 66 der parallele Algorithmus deutlich mehr arbeitet, so daß der durch den schnelleren Computer bedingte Geschwindigkeitsgewinn wieder wettgemacht wird. In Dimension 74 verringert sich diese Mehrarbeit, und in Dimension 82 ist die parallele Version bereits auf der skalaren Maschine schneller als die serielle (siehe auch Abbildungen 6.5 und 6.6).

Bei der Abhängigkeit der Rechenzeit von der Blockweite der Vorreduktion verhalten sich beide Algorithmen ähnlich. Obwohl größere Blockweiten

erheblich mehr Rechenzeit benötigen, vermindert sich im allgemeinen die Gesamtrechenzeit, da Lösungen früher gefunden werden. Dadurch verringert sich die Anzahl der Runden und der Aufrufe der Schlußenumeration.

Betrachtet man die Laufzeiten der einzelnen Schlußenumerationen, so stellt man fest, daß die Laufzeiten der parallelen Version eine leicht steigende Funktion in der Dichte darstellen (siehe Abbildung 6.7). Die lokale Maximum bei Dichte 1 begründet sich in der Tatsache, daß in diesem Bereich zur Verbesserung der Lösungswahrscheinlichkeit bis zu 45% mehr Hauptspeicher zur Verfügung gestellt wurde. Die serielle Version der Schlußenumeration benötigt bei Dichte 1 die maximale Rechenzeit, bei höheren und niedrigeren Dichten sinkt die Rechenzeit deutlich. Während die Laufzeiten der Schlußenumeration im seriellen Fall mit steigender Blockweite der Blockreduktion stark fallen, läßt sich dies bei der parallelen Version nicht feststellen.

Beachtenswert ist außerdem, daß durch die Erhöhung der Dimension um 8 die parallele Version der Schlußenumeration eine etwa verdoppelte Laufzeit benötigt, während sich die der seriellen Version mehr als verzehnfacht.

Ein Vergleich der Laufzeiten auf der IBM und der SNI S200 zeigt, daß der parallele Algorithmus die Rechenkapazität der Vektorpipelines nutzt. Es fällt auf, daß die Schlußenumeration auf dem Vektorrechner ungefähr um den Faktor 7 schneller arbeitet. Dies entspricht etwa dem Verhältnis der größten gemessenen Rechengeschwindigkeiten der beiden Computer (860 MFlop/s zu 130 MFlop/s $\approx 6.62 : 1$). Die restlichen Programmteile erfahren eine Beschleunigung um den Faktor 5. Dies ist durch die kurzen Vektoroperationen, die in der LLL-Reduktion auftreten, bedingt. Diese haben eine Länge von maximal 85 Operanden, während die optimale Vektorlänge für den Vektorrechner mindestens 512 Operanden beträgt. Da der Anteil der LLL-Reduktion an der Gesamtrechenzeit jedoch sehr klein ist, ergibt sich in der Summe ein Beschleunigungsfaktor von mehr als 6.

Nachteilig an der parallelen Version der Schlußenumeration ist zum einen, daß auf großen Datenfeldern gearbeitet wird, um die Vektorpipelines möglichst lange mit voller Geschwindigkeit arbeiten zu lassen. Hierbei stößt man bei Verwendung von Vektorrechnern an Universitäten sehr schnell an die Grenzen der Hauptspeicherkontingentierung. In Kreisen des „richtigen“ Supercomputings sollte diese Beschränkung jedoch keine Rolle spielen, da heutige Vektorrechner über mehrere Gigabytes Hauptspeicher und schnellen Hintergrundspeicher verfügen. Weiterhin werden diese Daten häufig sortiert und umgespeichert, d.h. der eingentliche Rechenaufwand beträgt weniger als 50%. Setzt man jedoch größere Hauptspeicher und geeignete Schnittmethoden bei der Berechnung ein, so könnte der Zeitanteil der Sortierfunktion deutlich verringert werden.

Ein weiterer spezieller Nachteil des Vektorrechners S200 ist das Vorhandensein der Multifunktionspipelines, die bei unseren Algorithmen nicht adäquat genutzt werden können. Diese Pipelines setzen eine Multiplikation gefolgt von einer Addition voraus. So wird zum Beispiel die Berech-

nung des neuen Höhenquadrates

$$c_i^{neu} = c_i^{alt} + (\tilde{u}_i^{(t)} + y_i)^2 C_t \qquad (i = 1, \ldots, vlen)$$

durch die folgenden drei Vektorbefehle realisiert:

`VADD`	$\tilde{u}, y$	$\rightarrow VR_1$	durch Pipe 1
`VMULT`	VR_1, VR_1	$\rightarrow VR_2$	durch Pipe 2
`VMULTADD`	VR_2, C_t, c^{alt}	$\rightarrow c^{neu}$	durch Pipe 1

Auf einem Vektorrechner der Firma CRAY mit vier Funktionseinheiten, die entweder eine Addition oder eine Multiplikation ausführen können, würde die folgende Befehlssequenz verwendet:

`VADD`	$\tilde{u}, y$	$\rightarrow VR_1$	durch Pipe 1
`VMULT`	VR_1, VR_1	$\rightarrow VR_2$	durch Pipe 2
`VMULT`	VR_2, C_t	$\rightarrow VR_3$	durch Pipe 3
`VADD`	VR_3, c^{alt}	$\rightarrow c^{neu}$	durch Pipe 4

Da beide Rechner über die Möglichkeit des Chainings verfügen (Ergebnisse einer Vektoroperation können ohne wesentliche Zeitverzögerung als Operanden für eine andere Vektoroperation verwendet werden), kann im zweiten Fall die gesamte Berechnung in einer „Superpipeline“ abgearbeitet werden, während im ersten Fall zwischen dem zweiten und dritten Vektorbefehl gewartet werden muß, bis der erste Befehl vollständig abgearbeitet ist.

6.2 Das Chor-Rivest-Kryptosystem

Das Kryptosystem, das von Chor und Rivest in [CR84] vorgeschlagen wurde, basiert auf dem Rucksackproblem. Da Rucksackprobleme mit kleinen Dichten verhältnismäßig einfach zu lösen sind (siehe [CJLOSS92] und auch Kapitel 3), beruht dieses System auf einem Rucksackproblem mit Dichten > 1. Weiterhin sind die Gewichte so gewählt, daß sich für jeden $(0,1)-$Vektor eine eindeutige Summe ergibt. Verfügt man über geheime Informationen, die sich bei der Generierung des Systems ergeben, kann die Entschlüsselung schnell erfolgen. Vorgeschlagen wurden in [CR84] als Parameterpaare für die Dimension des Rucksackproblems und der Anzahl der Gewichte die Tupel $(103, 12)$, $(197, 24)$ und $(211, 24)$, wobei das System mit der Dimension 103 für Tests von Angriffen gedacht war. Die Gewichte a_i dieses Schemas wurden unserer Arbeitsgruppe von Chor und Rivest zur Verfügung gestellt.
Zum Test wurden 50 zufällige $(0,1)-$Vektoren $(x_1, \ldots, x_{103})$ mit $\sum_{i=1}^{103} x_i = 12$ gewählt und die zugehörige Summe $s = \sum_{i=1}^{103} x_i a_i$ berechnet. Daraus wurde jeweils eine Basismatrix B (wie auf Seite 45) mit $n = 103$, $q = 12$ und $c = n^2$ generiert. Auf diese Basen wurde der folgende Algorithmus angewandt:

Algorithmus Chor-Rivest

Eingabe: Gitterbasis $b_0, \ldots, b_{103} \in \mathbb{Z}^{106}$

`FOR` Offset $= 0, 10, \ldots, 40$ `DO`

1. Rechtsrotation der Vektoren $b_1, \ldots, b_{103}$ der Ausgangsbasis mit Offset
2. LLL-Reduktion
3. Entferne die letzten zwei Vektoren
   und die letzten zwei Komponenten jedes Vektors
4. Blockreduktion mit $\beta = 28$, maximal 3 Minuten

`ENDFOR`

Ausgabe: Gesamtzeit, LLL-Zeit, Zeiten der Aufzählungsverfahren,

Wie bei dem Algorithmus zur Lösung von Rucksackproblemen wird die LLL-Reduktion stets mit tiefen Einfügungen bis zur Position 5 durchgeführt. Nach jeder Längenreduktion wird getestet, ob ein Lösungsvektor erzeugt wurde. Der Algorithmus bricht dann sofort ab. Die Blockreduktion benutzt das ungeschnittene Aufzählungsverfahren und wird nach drei Minuten abgebrochen. Zwischen den einzelnen Schritten wird keine Permutation durchgeführt.
Hierbei ergaben sich folgende Ergebnisse:

		Lösungen		Zeit		max. Zeit
Runde	Offset	pro Stufe	insg.	dieser Stufe	insgesamt	
1	00	7	7	2:32.849	2:32.849	3:05.690
2	10	5	12	2:12.836	4:45.685	6:12.181
3	20	2	14	2:04.615	6:50.300	9:15.915
4	30	3	17	1:52.501	8:42.801	12:20.682
5	40	5	22	1:46.461	10:29.262	15:21.275

Die Zeiten sind über 50 Probleme gemittelt und in Minuten:Sekunden.Millisekunden angegeben.

7. Bilanz und Ausblick

Die vorigen Ergebnisse haben gezeigt, daß die bisherigen sequentiellen Algorithmen vektorisiert werden können, um den hohen Geschwindigkeitsvorteil eines Vektorrechners zu nutzen. Die Programme und Funktionen wurden in Fortran gemäß dem Standard ANSI X3.9-1978 mit Erweiterungen des MIL-STD-1753 (Military Standard Fortran DOD Supplement to ANSI X3.9-1978) programmiert, so daß gängige Fortran-77-Compiler die Quelltexte übersetzen können.
Erwartungsgemäß benötigen die Programme viel Arbeitsspeicher, was jedoch bereits heute keinerlei Probleme bereiten sollte.
Das Ergebnis dieser Arbeit ist insofern höher zu bewerten, da der verwendete Vektorrechner einer Baureihe angehört, die bereits seit fünf Jahren gebaut wird. Sowohl die seriellen Computer als auch die Vektorrechner haben in dieser Zeit Steigerungen der Rechenleistung erfahren (Workstations etwa Faktor 6 zur HP 710/50; Vektorrechner Faktor >200 zur SNI S200).
Unmittelbare Verbesserungen des Resultats ergäben sich durch einen Wechsel zu einer Maschine mit kleinerem $n_{1/2}$ und ohne Multifunktionspipeline, die ohnehin nicht ausreichend genutzt werden kann. Dies würde zu einem besseren Laufzeitverhalten der LLL-Reduktion führen.
Offenbar profitiert die parallele Aufzählungsfunktion von Schnittmethoden, die auch in sequentiellen Algorithmen verwendet werden. Bei der Entwicklung neuer Schnittvarianten sollte das Augenmerk auf solche gelegt werden, die deterministisch arbeiten, d.h. nicht auf Heuristiken beruhen.
Ein großer Vorteil der parallelen Enumeration ergibt sich im Zusammenhang mit der Blockreduktion. Hier wäre es möglich, mehrere geeignete Verbesserungen zurückzugeben. Außerdem ist es vorstellbar, Blockreduktionen mit „variabler“ Blockweite durchzuführen (z.B. Suche Verbesserungen in einem Fenster mit mindestens 20 Vektoren).
Schließlich ist auch ein Wechsel der Parallelität möglich, indem man die Algorithmen für ein Workstation-Cluster oder einen Vektorparallelrechner mit verteiltem Hauptspeicher modifiziert. Hierbei ist jedoch zu beachten, daß die entsprechende Maschine über eine hinreichend große Kommunikationsleistung verfügt.

8. Unterprogramme

Zum Verständnis der nachfolgenden Unterprogramme und Funktionen, in denen die Arithmetik auf langen Zahlen erfolgt, seien nun einige Konventionen erläutert. Ein Vektor von ganzen Zahlen mit einstellbaren Wertebereich wird durch die folgende Zeile deklariert:

```
DOUBLE PRECISION OP(1:LDOP, -2:IVAZBL)
```

Hierbei gibt `LDOP` die Anzahl der Elemente des Vektors an. Jedes Element besteht aus dem Vorzeichen (Spalte =-2), dem höchsten gültigen Block (Spalte =-1) und den Ziffernblöcken (Spalte 0–IVAZBL). Jeder Ziffernblock beinhaltet eine Zahl im Bereich von 0 bis $2^{25} - 1$.

8.1 Grundrechenarten

8.1.1 add.f

```
C
C ADDITION ZWEIER VEKTOREN OP1 = OP1 + OP2
C
      SUBROUTINE VIADD1(IV, IVAZBL, LDOP1, OP1, LDOP2, OP2)
C
      IMPLICIT NONE
      INTEGER IV, LDOP1, LDOP2, IVAZBL
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      DOUBLE PRECISION OP1(1:LDOP1, -2:IVAZBL)
      DOUBLE PRECISION OP2(1:LDOP2, -2:IVAZBL)
C
      INTEGER I, J
      DOUBLE PRECISION MAXLD, MINLD, TMP
C
C     DIE TRIVIALEN FAELLE
C
      IF (OP2(1,ILEAD) .LT. 0.0) RETURN
      IF (OP1(1,ILEAD) .LT. 0.0) THEN
```

```
        CALL VICOPY(IV, IVAZBL, LDOP2, OP2, LDOP1, OP1)
        RETURN
      ENDIF
C
C BIS MINLD WIRD ADDIERT, DANN NUR NOCH KOPIERT
C
      MINLD = MIN(OP1(1,ILEAD), OP2(1,ILEAD))
      MAXLD = MAX(OP1(1,ILEAD), OP2(1,ILEAD))
C
      DO 100 J=1, IV, 1
 100    OP1(J,ILEAD) = OP2(J,ISIGN) * OP1(J,ISIGN)
C
      DO 200 I=0, MINLD, 1
        DO 200 J=1, IV, 1
 200      OP1(J,I) = OP1(J,I) + OP2(J,I) * OP1(J,ILEAD)
C
      IF(OP2(1,ILEAD) .GT. MINLD) THEN
        DO 210 I=MINLD+1, MAXLD, 1
          DO 210 J=1, IV, 1
            OP1(J,I) = OP2(J,I) * OP1(J,ILEAD)
 210    CONTINUE
      ENDIF
C
C FALLS DIE ZAHL GROESSER WIRD
C
      DO 220 J=1, IV, 1
 220    OP1(J,MAXLD+1) = 0.0
C
C VERARBEITUNG DER UEBERTRAEGE
C
      DO 250 I=0, MAXLD, 1
        DO 250 J=1, IV, 1
          TMP = 0
          IF (OP1(J,I) .LT. 0.0) THEN
            TMP = -1
          ELSE IF (OP1(J,I) .GT. IVLOWB) THEN
            TMP = 1
          ENDIF
          OP1(J,I) = OP1(J,I) - TMP * IVBASI
          OP1(J,I+1) = OP1(J,I+1) + TMP
 250  CONTINUE
C
C FALLS SICH DAS VORZEICHEN AENDERT
C
      DO 270 I=0, MAXLD, 1
        DO 270 J=1, IV, 1
          IF(OP1(J,MAXLD+1) .LT. 0.0) THEN
```

```
            OP1(J,I) = IVLOWB - OP1(J,I)
          ENDIF
 270  CONTINUE
      DO 290 J=1, IV, 1
        IF(OP1(J,MAXLD+1) .LT. 0.0) THEN
          OP1(J,0) = OP1(J,0) + 1.0
          OP1(J,ISIGN) = -OP1(J,ISIGN)
          OP1(J,MAXLD+1) = 0.0
        ENDIF
 290  CONTINUE
C
C FESTSTELLEN DES NEUEN LEADS
C
      I = MAXLD+2
 305    TMP = 0.0
        I = I - 1
        DO 300 J=1, IV, 1
          TMP = TMP + OP1(J,I)
 300    CONTINUE
      IF ((TMP .EQ. 0.0) .AND. (I .GT. 0)) GOTO 305
      IF ((I .EQ. 0) .AND. (TMP .EQ. 0.0)) I = -1
C
      DO 310 J=1, IV, 1
        OP1(J,ILEAD) = I
 310  CONTINUE
C
      IF(I .GE. IVAZBL) THEN
        CALL OVERFL('IADD      ')
      ENDIF
C
      RETURN
C
      END
```

8.1.2 sub.f

```
C
C SUBTRAKTION ZWEIER VEKTOREN OP1 = OP1 - OP2
C
      SUBROUTINE VISUB1(IV, IVAZBL, LDOP1, OP1, LDOP2, OP2)
C
      IMPLICIT NONE
      INTEGER IV, LDOP1, LDOP2, IVAZBL
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      DOUBLE PRECISION OP1(1:LDOP1, -2:IVAZBL)
```

```
      DOUBLE PRECISION OP2(1:LDOP2, -2:IVAZBL)
C
      INTEGER I, J
      DOUBLE PRECISION MAXLD, MINLD, TMP
C
C     DIE TRIVIALEN FAELLE
C
      IF (OP2(1,ILEAD) .LT. 0.0) RETURN
      IF (OP1(1,ILEAD) .LT. 0.0) THEN
        CALL VICOPY(IV, IVAZBL, LDOP2, OP2, LDOP1, OP1)
        DO 90 J=1, IV, 1
 90       OP1(J,ISIGN) = -OP1(J,ISIGN)
        RETURN
      ENDIF
C
C BIS MINLD WIRD SUBTRAHIERT, DANN NUR NOCH KOPIERT
C
      MINLD = MIN(OP1(1,ILEAD), OP2(1,ILEAD))
      MAXLD = MAX(OP1(1,ILEAD), OP2(1,ILEAD))
C
      DO 100 J=1, IV, 1
 100    OP1(J,ILEAD) = 0 - OP2(J,ISIGN) * OP1(J,ISIGN)
C
      DO 200 I=0, MINLD, 1
        DO 200 J=1, IV, 1
 200      OP1(J,I) = OP1(J,I) + OP2(J,I) * OP1(J,ILEAD)
C
      IF(OP2(1,ILEAD) .GT. MINLD) THEN
        DO 210 I=MINLD+1, MAXLD, 1
          DO 210 J=1, IV, 1
            OP1(J,I) = OP2(J,I) * OP1(J,ILEAD)
 210    CONTINUE
      ENDIF
C
C FALLS DIE ZAHL GROESSER WIRD
C
      DO 220 J=1, IV, 1
 220    OP1(J,MAXLD+1) = 0.0
C
C VERARBEITUNG DER UEBERTRAEGE
C
      DO 250 I=0, MAXLD, 1
        DO 250 J=1, IV, 1
          TMP = 0
          IF (OP1(J,I) .LT. 0.0) THEN
            TMP = -1
          ELSE IF (OP1(J,I) .GT. IVLOWB) THEN
```

```
            TMP = 1
          ENDIF
          OP1(J,I) = OP1(J,I) - TMP * IVBASI
          OP1(J,I+1) = OP1(J,I+1) + TMP
 250  CONTINUE
C
C FALLS SICH DAS VORZEICHEN AENDERT
C
      DO 270 I=0, MAXLD, 1
        DO 270 J=1, IV, 1
          IF(OP1(J,MAXLD+1) .LT. 0.0) THEN
            OP1(J,I) = IVLOWB - OP1(J,I)
          ENDIF
 270  CONTINUE
      DO 280 J=1, IV, 1
        IF(OP1(J,MAXLD+1) .LT. 0.0) THEN
          OP1(J,0) = OP1(J,0) + 1.0
          OP1(J,ISIGN) = -OP1(J,ISIGN)
          OP1(J,MAXLD+1) = 0.0
        ENDIF
 280  CONTINUE
C
C FESTSTELLEN DES NEUEN LEADS
C
      I = MAXLD+2
 305    TMP = 0.0
        I = I - 1
        DO 300 J=1, IV, 1
          TMP = TMP + OP1(J,I)
 300    CONTINUE
      IF ((TMP .EQ. 0.0) .AND. (I .GT. 0)) GOTO 305
      IF ((I .EQ. 0) .AND. (TMP .EQ. 0.0)) I = -1
C
      DO 310 J=1, IV, 1
        OP1(J,ILEAD) = I
 310  CONTINUE
C
      IF(I .GE. IVAZBL) THEN
        CALL OVERFL('ISUB      ')
      ENDIF
C
      RETURN
C
      END
```

8.1.3 mult.f

```
C
C MULTIPLIKATION EINES VEKTORS MIT EINEM SKALAR
C
      SUBROUTINE MULTI1(IV, OP, IVAZBL, LDOP1, OP1)
C
      IMPLICIT NONE
      INTEGER IV, LDOP1, OP, IVAZBL
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      DOUBLE PRECISION OP1(1:LDOP1, -2:IVAZBL)
C
      INTEGER I, J
      DOUBLE PRECISION LEAD, FAKTOR, TMP
C
C     DIE TRIVIALEN FAELLE
C
      IF ((OP .EQ. 1) .OR. (OP1(1,ILEAD) .LT. 0.0)) RETURN
      IF (OP .EQ. 0) THEN
        CALL VIZERO(IV, IVAZBL, LDOP1, OP1)
        RETURN
      ENDIF
      IF (OP .EQ. -1) THEN
        DO 100 J=1, IV, 1
          OP1(J,ISIGN) = -OP1(J,ISIGN)
 100    CONTINUE
        RETURN
      ENDIF
C
      LEAD = OP1(1,ILEAD)
      IF (OP .LT. 0) THEN
        FAKTOR = -OP
        DO 110 J=1, IV, 1
          OP1(J,ISIGN) = -OP1(J,ISIGN)
 110    CONTINUE
      ELSE
        FAKTOR = OP
      ENDIF
C
C JETZT WIRD GEARBEITET
C
      DO 200 I=0, OP1(1,ILEAD), 1
        DO 200 J=1, IV, 1
 200      OP1(J,I) = OP1(J,I) * FAKTOR
C
```

```
C FALLS DIE ZAHL GROESSER WIRD
C
      DO 220 J=1, IV, 1
 220    OP1(J,LEAD+1) = 0.0
C
C VERARBEITUNG DER UEBERTRAEGE
C
      DO 250 I=0, LEAD, 1
        DO 250 J=1, IV, 1
          TMP = DINT(OP1(J,I) * IVBHM1)
          OP1(J,I) = OP1(J,I) - IVBASI * TMP
          OP1(J,I+1) = OP1(J,I+1) + TMP
 250  CONTINUE
C
C FESTSTELLEN DES NEUEN LEADS
C
      TMP = LEAD
      DO 300 J=1, IV, 1
        IF (OP1(J,LEAD+1) .GT. 0.0) TMP = LEAD+1
 300  CONTINUE
C
      DO 310 J=1, IV, 1
        OP1(J,ILEAD) = TMP
 310  CONTINUE
C
      IF(TMP .GE. IVAZBL) THEN
        CALL OVERFL('IVMULTI1  ')
      ENDIF
C
      RETURN
C
      END
```

8.1.4 multbi.f

```
C
C MULTIPLIZIEREN EINES VEKTORS MIT EINEM GROSSEN VINT
C
      SUBROUTINE MULTBI(IV, IVAZBL, LDOP, OP, LDVIV, VIV, LDERG, ERG)
C
      IMPLICIT NONE
      INTEGER IV, IVAZBL, LDVIV, LDOP, LDERG
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      DOUBLE PRECISION OP(1:LDOP,-2:IVAZBL)
```

```
      DOUBLE PRECISION VIV(1:LDVIV,-2:IVAZBL)
      DOUBLE PRECISION ERG(1:LDERG,-2:IVAZBL)

C
C DIE LOKALEN VARIABLEN
C
      INTEGER I, J, K, L, COUNT
      DOUBLE PRECISION TMP
      COUNT = 0
C
C DIE MULTIPLIKATION MIT NULL
C
      IF ((OP(1,ILEAD) .LT. 0.0) .OR. (VIV(1,ILEAD) .LT. 0.0)) THEN
        DO 90 J=1, IV, 1
          ERG(J,ISIGN) = 1.0
          ERG(J,ILEAD) = -1.0
 90     CONTINUE
        RETURN
      ENDIF
C
C BESTIMMUNG DES NEUEN LEADS
C
      ERG(1, ILEAD) = VIV(1, ILEAD) + OP(1, ILEAD)
      IF(ERG(1, ILEAD) .GE. IVAZBL) THEN
        CALL OVERFL('MULTBI    ')
      ENDIF
C
C ALLES NOETIGE LOESCHEN
C
      DO 100 K=0, ERG(1, ILEAD)+1, 1
        DO 100 J=1, IV, 1
          ERG(J, K) = 0.0
 100  CONTINUE
C
C DIE AEUSSERSTE STELLE LAEUFT UEBER DIE ERGEBNISSTELLEN
C
      DO 200 K=0, ERG(1, ILEAD), 1
C DIE INNERE UEBER DIE OPERANDENSTELLEN
        DO 205 I=0, VIV(1, ILEAD), 1
          L = K - I
          IF((L .LE. OP(1, ILEAD)) .AND. (L .GE. 0.0)) THEN
            DO 210 J=1, IV, 1
              ERG(J,K) = ERG(J,K) + OP(1,L) * VIV(J,I)
 210        CONTINUE
            COUNT = COUNT + 1
C NACH 32 ADDITIONEN KOENNTEN UNS BITS VERLOREN GEHEN
            IF(COUNT .GE. 8) THEN
```

```
              DO 220 J=1, IV, 1
                TMP = DINT(ERG(J,K) * IVBHM1)
                ERG(J,K) = ERG(J,K) - IVBASI * TMP
                ERG(J,K+1) = ERG(J,K+1) + TMP
 220            CONTINUE
                COUNT = 0
              ENDIF
            ENDIF
 205      CONTINUE
C SCHLUSSBETRACHTUNG
         IF (COUNT .GT. 0) THEN
           DO 230 J=1, IV, 1
             TMP = DINT(ERG(J,K) * IVBHM1)
             ERG(J,K) = ERG(J,K) - IVBASI * TMP
             ERG(J,K+1) = ERG(J,K+1) + TMP
 230         CONTINUE
           COUNT = 0
         ENDIF
 200   CONTINUE
C
C DAS RICHTIGE VORZEICHEN ERMITTELN
C
       DO 250 J=1, IV, 1
         ERG(J, ISIGN) = OP(1, ISIGN) * VIV(J, ISIGN)
 250   CONTINUE
C
C FALLS DIE ZAHLEN GROESSER GEWORDEN SIND
C
       L = -1
       K = ERG(1, ILEAD) + 1
       DO 300 J=1, IV, 1
         IF(ERG(J,K) .GT. 0.0) L = J
 300   CONTINUE
       IF(L .LT. 0) K = K - 1
C
       IF(K .GE. IVAZBL) THEN
         CALL OVERFL('MULTBI    ')
       ENDIF
C
C DEN NEUEN LEAD EINTRAGEN
C
       DO 320 J=1, IV, 1
         ERG(J,ILEAD) = K
 320   CONTINUE
C
       RETURN
C
```

```
      END
```

8.2 Vektoroperationen

8.2.1 isubmult.f

```
C
C SUBTRAHIEREN DES VIELFACHEN EINES VEKTORS VON EINEM ANDEREN
C
      SUBROUTINE ISUBML(IV, IVAZBL, LDERG, ERG, LDOP, OP, LDV, V)
C
      IMPLICIT NONE
      INTEGER IV, IVAZBL, LDOP, LDERG, LDV
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      DOUBLE PRECISION ERG(1:LDERG,-2:IVAZBL)
      DOUBLE PRECISION OP(1:LDOP,-2:IVAZBL)
      DOUBLE PRECISION V(1:LDV,-2:IVAZBL)
C
C DIE LOKALEN VARIABLEN
C
      DOUBLE PRECISION LEAD, TMP
      INTEGER I, J, K, L, ICOUNT
C
C DIE EINFACHEN FAELLE
C
      IF ((OP(1, ILEAD) .LT. 0.0) .OR. (V(1, ILEAD) .LT. 0.0)) RETURN
C
      LEAD = OP(1, ILEAD) + V(1, ILEAD)
      IF (LEAD .GE. IVAZBL) CALL OVERFL('ISUBMUL   ')
C
C NOETIGES LOESCHEN
C
      DO 100 I=ERG(1,ILEAD)+1, LEAD+1, 1
        DO 100 J=1, IV, 1
 100      ERG(J,I) = 0.0
C
      ICOUNT = 1
      IF(ERG(1,ILEAD) .GT. LEAD) LEAD = ERG(1,ILEAD)
C
C
C DIE AEUSSERSTE SCHLEIFE LAEUFT UEBER DIE ERGEBNISSTELLEN
C
      DO 200 I=0, LEAD, 1
C DIE INNERE UEBER DIE OPERANDENSTELLEN
```

```
        DO 205 K=0, V(1, ILEAD), 1
          L = I - K
          IF((L .LE. OP(1, ILEAD)) .AND. (L .GE. 0)
     a                    .AND. (OP(1,L) .NE. 0.0D0)) THEN
            DO 210 J=1, IV, 1
              ERG(J,I)=ERG(J,I)-OP(1,L)*V(J,K)
     a                *V(J,ISIGN)*OP(1,ISIGN)*ERG(J,ISIGN)
 210        CONTINUE
            ICOUNT = ICOUNT + 1
C NACH 32 ADDITIONEN KOENNTEN UNS BITS VERLOREN GEHEN
            IF(ICOUNT .GE. 8) THEN
              DO 220 J=1, IV, 1
                TMP = -DINT((-ERG(J,I) + IVLOWB) * IVBHM1)
                IF (ERG(J,I) .GE. 0) THEN
                  TMP = DINT(ERG(J,I) * IVBHM1)
                ENDIF
                ERG(J,I) = ERG(J,I) - IVBASI * TMP
                ERG(J,I+1) = ERG(J,I+1) + TMP
 220          CONTINUE
              ICOUNT = 1
            ENDIF
          ENDIF
 205    CONTINUE
C SCHLUSSBETRACHTUNG
        DO 230 J=1, IV, 1
          TMP = -DINT((-ERG(J,I) + IVLOWB) * IVBHM1)
          IF (ERG(J,I) .GE. 0) THEN
            TMP = DINT(ERG(J,I) * IVBHM1)
          ENDIF
          ERG(J,I) = ERG(J,I) - IVBASI * TMP
          ERG(J,I+1) = ERG(J,I+1) + TMP
 230    CONTINUE
        ICOUNT = 1
 200  CONTINUE
C
C FALLS SICH DAS VORZEICHEN AENDERT
C
      DO 270 I=0, LEAD, 1
        DO 270 J=1, IV, 1
          IF(ERG(J,LEAD+1) .LT. 0.0) THEN
            ERG(J,I) = IVLOWB - ERG(J,I)
          ENDIF
 270  CONTINUE
      DO 290 J=1, IV, 1
        IF(ERG(J,LEAD+1) .LT. 0.0) THEN
          ERG(J,0) = ERG(J,0) + 1.0
          ERG(J,ISIGN) = -ERG(J,ISIGN)
```

```
          ERG(J,LEAD+1) = -1.0 - ERG(J, LEAD+1)
        ENDIF
 290  CONTINUE
C
C FESTSTELLEN DES NEUEN LEADS
C
      I = LEAD+2
 305    TMP = 0.0
        I = I - 1
        DO 300 J=1, IV, 1
          TMP = TMP + ERG(J,I)
 300    CONTINUE
      IF ((TMP .EQ. 0.0) .AND. (I .GT. 0)) GOTO 305
      IF ((I .EQ. 0) .AND. (TMP .EQ. 0.0)) I = -1
C
      DO 310 J=1, IV, 1
        ERG(J,ILEAD) = I
 310  CONTINUE
C
      IF(I .GE. IVAZBL) THEN
        CALL OVERFL('ISUBMUL   ')
      ENDIF
C
      RETURN
C
      END
```

8.2.2 isubmultbi.f

```
C
C SUBTRAHIEREN DES VIELFACHEN EINES VEKTORS VON EINEM ANDEREN
C
      SUBROUTINE SUBMUL(IV, IVAZBL, LDOP1, OP1, LAMBDA, LDOP2, OP2)
C
      IMPLICIT NONE
      INTEGER IV, IVAZBL, LDOP1, LDOP2, LAMBDA
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      DOUBLE PRECISION OP1(1:LDOP1,-2:IVAZBL)
      DOUBLE PRECISION OP2(1:LDOP2,-2:IVAZBL)
C
C DIE LOKALEN VARIABLEN
C
      DOUBLE PRECISION FAKTR1, FAKTR2, MINLD, MAXLD, TMP
      INTEGER I, J
```

```
C
C DIE EINFACHEN FAELLE
C
      IF ((LAMBDA .EQ. 0) .OR. (OP2(1, ILEAD) .LT. 0.0)) RETURN
C
      IF (LAMBDA .GT. 0) THEN
        FAKTR1 = INT(LAMBDA / IVBASI)
        FAKTR2 = MOD(LAMBDA , IVBASI)
      ELSE
        FAKTR1 = -INT(-LAMBDA / IVBASI)
        FAKTR2 = -MOD(-LAMBDA , IVBASI)
      ENDIF
C
      MINLD = MIN(OP1(1, ILEAD), OP2(1, ILEAD))
      MAXLD = MAX(OP1(1, ILEAD), OP2(1, ILEAD))
C
C BENOETIGTEN TEIL LOESCHEN
C
      DO 100 I=OP1(1, ILEAD)+1, MIN(NINT(MAXLD+2),IVAZBL), 1
        DO 100 J=1, IV, 1
 100      OP1(J,I) = 0.0
C
C DER KLEINERE TEIL
C
      IF (FAKTR2 .NE. 0) THEN
        DO 200 J=1, IV, 1
 200      OP2(J,IVAZBL) = OP2(J, ISIGN) * OP1(J, ISIGN) * FAKTR2
C
        DO 220 I=0, MINLD, 1
          DO 220 J=1, IV, 1
 220        OP1(J,I) = OP1(J,I) - OP2(J,I) * OP2(J,IVAZBL)
C
        IF(OP1(1, ILEAD) .LT. OP2(1, ILEAD)) THEN
          DO 240 I=MINLD+1, MAXLD, 1
            DO 240 J=1, IV, 1
 240          OP1(J,I) = 0.0 -  OP2(J,I) * OP2(J,IVAZBL)
        ENDIF
C
      ENDIF
C
C
C DER GROESSERE TEIL
C
      IF (FAKTR1 .NE. 0) THEN
C
C VORHERSEHBAREN UEBERLAUF VERHINDERN
C
```

```
        IF ((OP2(1, ILEAD) + 1) .EQ. IVAZBL) THEN
          CALL OVERFL('SUBMUL    ')
        ENDIF
C
        DO 260 J=1, IV, 1
 260      OP2(J,IVAZBL) = OP2(J, ISIGN) * OP1(J, ISIGN) * FAKTR1
C
        DO 280 I=0, OP2(1, ILEAD), 1
          DO 280 J=1, IV, 1
 280        OP1(J,I+1) = OP1(J,I+1) - OP2(J,I) * OP2(J,IVAZBL)
C
        MAXLD = MAX( MAXLD, OP2(1,ILEAD) + 1.0)
C
      ENDIF
C
      DO 295 J=1, IV, 1
 295    OP2(J,IVAZBL) = 0.0
C
      DO 300 I=0, MAXLD, 1
        DO 300 J=1, IV, 1
          TMP = -DINT( (-OP1(J,I)+IVLOWB) * IVBHM1)
          IF (OP1(J,I) .GE. 0.0) THEN
            TMP = DINT(OP1(J,I) * IVBHM1)
          ENDIF
          OP1(J,I) = OP1(J,I) - TMP * IVBASI
          OP1(J,I+1) = OP1(J,I+1) + TMP
 300  CONTINUE
C
C FALLS SICH DAS VORZEICHEN VERAENDERT
C
      DO 320 I=0, MAXLD, 1
        DO 320 J=1, IV, 1
          IF(OP1(J,MAXLD+1) .LT. 0.0) THEN
            OP1(J,I) = IVLOWB - OP1(J,I)
          ENDIF
 320  CONTINUE
      DO 340 J=1, IV, 1
        IF(OP1(J,MAXLD+1) .LT. 0.0) THEN
          OP1(J, MAXLD+1) = -1.0 - OP1(J, MAXLD+1)
          OP1(J,ISIGN) = -OP1(J,ISIGN)
          OP1(J,0) = OP1(J,0) + 1.0
        ENDIF
 340  CONTINUE
C
C LEAD ERMITTELN UND UEBERTRAGEN
C
      I = MAXLD+2
```

```
 405    TMP = 0.0
        I = I - 1
        DO 400 J=1, IV, 1
          TMP = TMP + OP1(J,I)
 400    CONTINUE
      IF ((TMP .EQ. 0.0) .AND. (I .GT. 0)) GOTO 405
      IF ((I .EQ. 0) .AND. (TMP .EQ. 0.0)) I = -1
C
      DO 410 J=1, IV, 1
        OP1(J,ILEAD) = I
 410  CONTINUE
C
      IF (I .GE. IVAZBL) CALL OVERFL('ISUBMUL   ')
C
      RETURN
C
      END
```

8.2.3 skalprod.f

```
C
C SKALARPRODUKTBERECHNUNG, DAS ERGEBNIS WIRD IN ERG ZURUECKGELIEFERT
C ERG DIENT DABEI ALS HILFSVEKTOR FUER ZWISCHENERGEBNISSE, ER MUSS
C DAHER MINDESTENS DIE LAENGE IV HABEN
C
      SUBROUTINE VISKPR(IV, IVAZBL, LDOP1, OP1, LDOP2, OP2, LDERG, ERG)
C
      IMPLICIT NONE
      INTEGER IV, IVAZBL, LDOP1, LDOP2, LDERG
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      DOUBLE PRECISION OP1(1:LDOP1,-2:IVAZBL)
      DOUBLE PRECISION OP2(1:LDOP2,-2:IVAZBL)
      DOUBLE PRECISION ERG(1:LDERG,-2:IVAZBL)
C
C DIE LOKALEN VARIABLEN
C
      INTEGER I, J, K, L, ICOUNT
      DOUBLE PRECISION LEAD, TMP
C
C DIE EINFACHEN FAELLE
C
      IF ((OP1(1,ILEAD) .LT. 0.0) .OR. (OP2(1,ILEAD) .LT. 0.0)) THEN
        ERG(1,ILEAD) = -1.0
        ERG(1,ISIGN) = 1.0
```

```
        RETURN
      ENDIF
C
      LEAD = OP1(1,ILEAD) + OP2(1,ILEAD)
C
C FALLS DIE ZAHLEN ZU GROSS WERDEN KOENNTEN, WIRD EINE BESONDERE
C BETRACHTUNG NOTWENDIG
C
      IF (LEAD .GE. IVAZBL) THEN
        DO 100 J=1, IV, 1
          ERG(J,0) = 0.0
          ERG(J,1) = 0.0
 100    CONTINUE
        DO 110 I=0, OP1(1,ILEAD), 1
          DO 110 J=1, IV, 1
            IF (OP1(J,I) .GT. 0) ERG(J,0) = I
 110    CONTINUE
        DO 120 I=0, OP2(1,ILEAD), 1
          DO 120 J=1, IV, 1
            IF (OP2(J,I) .GT. 0) ERG(J,1) = I
 120    CONTINUE
C
        LEAD = -1.0
        DO 130 J=1, IV, 1
          ERG(J,1) = ERG(J,1) + ERG(J,0)
          IF(ERG(J,1) .GT. LEAD) LEAD = ERG(J,1)
 130    CONTINUE
C
      ENDIF
C
C DER LEAD IST NUN EVENTUELL GROSSZUEGIG ABGESCHAETZT
C
      IF(LEAD .GE. IVAZBL) THEN
        CALL OVERFL('VISKALPROD')
      ENDIF
C
C DIE VORZEICHEN BESTIMMEN UND BENOETIGTEN TEIL LOESCHEN
C
      DO 200 J=1, IV, 1
 200    ERG(J,ISIGN) = OP1(J, ISIGN) * OP2(J,ISIGN)
C
      DO 220 K=0, LEAD+1, 1
        DO 220 J=1, IV, 1
 220      ERG(J,K) = 0.0
      DO 222 I=LEAD+1, IVAZBL, 1
 222    ERG(1,I) = 0.0
C
```

```
C DIE BERECHNUNG DER TEILPRODUKTE
C
      ICOUNT = 0
      DO 300 K=0, LEAD, 1
        DO 310 I=0, OP1(1,ILEAD), 1
          L = K - I
          IF ((L .LE. OP2(1,ILEAD)) .AND. (L .GE. 0)) THEN
            DO 311 J=1, IV, 1
              ERG(J,K) = ERG(J,K) + OP1(J,I) * OP2(J,L)
 311        CONTINUE
            ICOUNT = ICOUNT + 1
            IF ( ICOUNT .GT. 8) THEN
              DO 312 J=1, IV, 1
                TMP = DINT( ERG(J,K) * IVBHM1 )
                ERG(J,K) = ERG(J,K) - IVBASI * TMP
                ERG(J,K+1) = ERG(J,K+1) + TMP
 312          CONTINUE
              ICOUNT = 0
            ENDIF
          ENDIF
 310    CONTINUE
C LETZTE UEBERTRAGBETRACHTUNG
        IF ( ICOUNT .GT. 0) THEN
          DO 313 J=1, IV, 1
            TMP = DINT( ERG(J,K) * IVBHM1 )
            ERG(J,K) = ERG(J,K) - IVBASI * TMP
            ERG(J,K+1) = ERG(J,K+1) + TMP
 313      CONTINUE
          ICOUNT = 0
        ENDIF
C
 300  CONTINUE
C
C JETZT WIRD AUFADDIERT
C
      DO 400 I=0, LEAD+1, 1
        TMP = 0.0
        DO 410 J=1, IV, 1
          TMP = TMP + ERG(J,I) * ERG(J,ISIGN)
 410    CONTINUE
        ERG(1,I) = TMP
 400  CONTINUE
C
C UEBERTRAEGE VERARBEITEN
C
      DO 420 I=0, IVAZBL-1, 1
        IF ( ERG(1,I) .GE. 0 ) THEN
```

```
          TMP = DINT( ERG(1,I) * IVBHM1)
        ELSE
          TMP = - DINT( (-ERG(1,I) + IVLOWB) * IVBHM1)
        ENDIF
        ERG(1,I) = ERG(1,I) - TMP * IVBASI
        ERG(1,I+1) = ERG(1,I+1) + TMP
 420  CONTINUE
C
C FALLS DAS ERGEBNIS NEGATIV IST, ERFOLGT EINE ENDBEHANDLUNG
C
      IF (ERG(1,IVAZBL) .LT. 0.0) THEN
        DO 430 I=0, IVAZBL-1, 1
          ERG(1,I) = IVLOWB - ERG(1,I)
 430    CONTINUE
        ERG(1,IVAZBL) = -1.0 - ERG(1,IVAZBL)
        ERG(1,ISIGN) = -1.0
        ERG(1,0) = ERG(1,0) + 1.0
      ELSE
        ERG(1,ISIGN) = 1.0
      ENDIF
C
C NUN WIRD DER LEAD BESTIMMT
C
      K = -1
      DO 450 I=0, IVAZBL, 1
        IF (ERG(1,I) .GT. 0.0) K = I
 450  CONTINUE
      ERG(1,ILEAD) = K
C
      IF (K .GE. IVAZBL) THEN
        CALL OVERFL('VISKALPROD')
      ENDIF
C
      RETURN
C
      END
```

8.2.4 norm.f

```
C
C NORMBERECHNUNG, DAS ERGEBNIS WIRD IN ERG ZURUECKGELIEFERT
C ERG DIENT DABEI ALS HILFSVEKTOR FUER ZWISCHENERGEBNISSE, ER MUSS
C DAHER MINDESTENS DIE LAENGE IV HABEN
C
      SUBROUTINE VINORM(IV, IVAZBL, LDOP, OP, LDERG, ERG)
C
```

```
      IMPLICIT NONE
      INTEGER IV, IVAZBL, LDOP, LDERG
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      DOUBLE PRECISION OP(1:LDOP,-2:IVAZBL)
      DOUBLE PRECISION ERG(1:LDERG,-2:IVAZBL)
C
C DIE LOKALEN VARIABLEN
C
      INTEGER I, J, K, L, ICOUNT
      DOUBLE PRECISION LEAD, TMP
C
C DER EINFACHE FALL
C
      IF (OP(1,ILEAD) .LT. 0.0) THEN
        ERG(1,ILEAD) = -1.0
        ERG(1,ISIGN) = 1.0
        RETURN
      ENDIF
C
      LEAD = OP(1,ILEAD) * 2.0 + 1.0
      IF (LEAD .GT. IVAZBL) THEN
        CALL OVERFL('VINORM    ')
      ENDIF
C
C ALLES NOETIGE LOESCHEN
C
      DO 100 I=0, LEAD, 1
        DO 100 J=1, IV, 1
 100      ERG(J,I) = 0.0
      DO 102 I=LEAD+1, IVAZBL, 1
 102    ERG(1,I) = 0.0
C
C DIE BERECHNUNG DER TERME, DIE ZWEIMAL IN DIE ADDITION EINGEHEN
C
      ICOUNT = 0
      DO 200 I=1, OP(1,ILEAD), 1
        DO 200 K=0, I-1, 1
          DO 210 J=1, IV, 1
            ERG(J,K+I) = ERG(J,K+I) + OP(J,K) * OP(J,I) * 2.0
 210      CONTINUE
          ICOUNT = ICOUNT + 1
          IF (ICOUNT .GT. 8) THEN
            DO 212 L=1, I, 1
              DO 212 J=1, IV, 1
                TMP = DINT( ERG(J,L) * IVBHM1 )
```

```
                ERG(J,L) = ERG(J,L) - IVBASI * TMP
                ERG(J,L+1) = ERG(J,L+1) + TMP
 212          CONTINUE
              ICOUNT = 0
            ENDIF
 200    CONTINUE
C
C DIE SYMMETRISCHEN TERME
C
       DO 250 I=0, OP(1,ILEAD), 1
         DO 250 J=1, IV, 1
 250       ERG(J,I+I) = ERG(J,I+I) + OP(J,I) * OP(J,I)
C
C LETZTE UEBERTRAGBETRACHTUNG
C
       DO 270 I=0, LEAD-1, 1
         DO 270 J=1, IV, 1
           TMP = DINT( ERG(J,I) * IVBHM1 )
           ERG(J,I) = ERG(J,I) - IVBASI * TMP
           ERG(J,I+1) = ERG(J,I+1) + TMP
 270    CONTINUE
C
C JETZT WIRD AUFADDIERT
C
       DO 300 I=0, LEAD, 1
         TMP = 0.0
         DO 310 J=1, IV, 1
           TMP = TMP + ERG(J,I)
 310     CONTINUE
         ERG(1,I) = TMP
 300   CONTINUE
C
C UEBERTRAEGE VERARBEITEN
C
       DO 320 I=0, IVAZBL-1, 1
         TMP = DINT( ERG(1,I) * IVBHM1)
         ERG(1,I) = ERG(1,I) - TMP * IVBASI
         ERG(1,I+1) = ERG(1,I+1) + TMP
 320   CONTINUE
C
C NUN WIRD DER LEAD BESTIMMT
C
       K = -1
       DO 350 I=0, IVAZBL, 1
         IF (ERG(1,I) .GT. 0.0) K = I
 350   CONTINUE
       ERG(1,ILEAD) = K
```

```
      ERG(1,ISIGN) = +1.0
C
      IF (K .GE. IVAZBL) THEN
        CALL OVERFL('VINORM    ')
      ENDIF
C
      RETURN
C
      END
```

8.2.5 shift.f

```
C
C SHIFTEN EINES VEKTORINT-VEKTORS UM N BITS
C
      SUBROUTINE SHIFT1(IV, IVAZBL, LDOP, OP, N)
C
      IMPLICIT NONE
      INTEGER IV, IVAZBL, LDOP, N
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      DOUBLE PRECISION OP(1:LDOP,-2:IVAZBL)
C
C DIE LOKALEN VARIABLEN
C
      INTEGER I, J, BLOCKS, RESTBI
      DOUBLE PRECISION TMP, FAKTOR, LEAD
C
C DER EINFACHE FALL
C
      IF ((N .EQ. 0) .OR. (OP(1, ILEAD) .LT. 0.0)) RETURN
C
C ES WIRD ERST MULTIPLIZIERT UND DANN DIE BLOECKE KOPIERT
C   BZW. VERSCHOBEN
C
      IF(N .GT. 0) THEN
        BLOCKS = INT(N / 25)
      ELSE
        BLOCKS = -(INT(-N / 25) + 1)
      ENDIF
      RESTBI = N - BLOCKS * 25
      FAKTOR = 2**RESTBI
      LEAD = OP(1,ILEAD)
C
      IF(RESTBI .GT. 0) THEN
```

```
        DO 100 J=1, IV, 1
          OP(J,LEAD+1) = 0.0
 100    CONTINUE
        DO 120 I=0, LEAD, 1
          DO 120 J=1, IV, 1
 120        OP(J,I) = OP(J,I) * FAKTOR
        DO 140 I=0, LEAD, 1
          DO 140 J=1, IV, 1
            TMP = DINT(OP(J,I) * IVBHM1)
            OP(J,I) = OP(J,I) - TMP * IVBASI
            OP(J,I+1) = OP(J,I+1) + TMP
 140    CONTINUE
C
        I = 0
        DO 160 J=1, IV, 1
          IF (OP(J,LEAD+1) .GT. 0.0) I=J
 160    CONTINUE
C
C DIE GROESSENORDNUNG HAT ZUGENOMMEN
C
        IF (I .GT. 0) LEAD = LEAD + 1.0
      ENDIF
C
      TMP = LEAD + BLOCKS
      IF (TMP .GE. IVAZBL) THEN
        CALL OVERFL('SHIFT1    ')
      ENDIF
C
C FALLS ALLE BITS NACH RECHTS HERAUSGESCHOBEN WERDEN
C
      IF(TMP .LT. 0) THEN
        DO 200 J=1, IV, 1
          OP(J,ISIGN) = 1.0
          OP(J,ILEAD) = -1.0
 200    CONTINUE
        RETURN
      ENDIF
C
C NACH RECHTS SHIFTEN
C
      IF (BLOCKS .LT. 0) THEN
        DO 300 I=-BLOCKS, LEAD, 1
          DO 300 J=1, IV, 1
 300        OP(J,I+BLOCKS) = OP(J,I)
      ENDIF
C
C NACH LINKS SHIFTEN
```

```
C
      IF (BLOCKS .GT. 0) THEN
        DO 320 I=TMP, BLOCKS, -1
          DO 320 J=1, IV, 1
 320        OP(J,I) = OP(J,I-BLOCKS)
        DO 330 I=0, BLOCKS-1, 1
          DO 330 J=1, IV, 1
 330        OP(J,I) = 0.0
      ENDIF
C
C LEAD EINTRAGEN
C
      DO 370 J=1, IV, 1
 370    OP(J,ILEAD) = TMP
C
      RETURN
C
      END
```

8.2.6 copy.f

```
C
C KOPIEREN EINES VEKTORINT-VEKTORS
C
      SUBROUTINE VICOPY(IV, IVAZBL, LDFROM, FROM, LDTO, TO)
C
      IMPLICIT NONE
      INTEGER IV, IVAZBL, LDFROM, LDTO
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      DOUBLE PRECISION FROM(1:LDFROM,-2:IVAZBL)
      DOUBLE PRECISION TO(1:LDTO,-2:IVAZBL)
C
C DIE LOKALEN VARIABLEN
C
      INTEGER I, J
C
C STIMMT DIE LEADING DIMENSION MIT DER VEKTORLAENGE UEBEREIN ?
C
      IF( (IV .EQ. LDTO) .AND. (IV .EQ. LDFROM) ) THEN
        DO 100 I=1, IV*(IVAZBL+3), 1
          TO(I,ISIGN) = FROM(I,ISIGN)
 100    CONTINUE
      ELSE
        DO 110 I=ISIGN, FROM(1,ILEAD), 1
```

```
          DO 110 J=1, IV, 1
            TO(J,I) = FROM(J,I)
 110    CONTINUE
      ENDIF
C
      RETURN
C
      END
```

8.2.7 zero.f

```
C
C LOESCHEN EINES VEKTORINT-VEKTORS
C
      SUBROUTINE VIZERO(IV, IVAZBL, LDOP, OP)
C
      IMPLICIT NONE
      INTEGER IV, IVAZBL, LDOP
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      DOUBLE PRECISION OP(1:LDOP,-2:IVAZBL)
C
C DIE LOKALEN VARIABLEN
C
      INTEGER J
C
C ES WIRD NUR LOGISCH GELOESCHT, D.H. DAS VORZEICHEN WIRD POSITIV
C UND DER LEAD AUF -1 GESETZT
C
      DO 100 J=1, IV, 1
        OP(J,ISIGN) = 1.0
        OP(J,ILEAD) = -1.0
 100  CONTINUE
C
      RETURN
C
      END
```

8.3 Funktionen auf Floatingpointvektoren

8.3.1 norm.f

```
      DOUBLE PRECISION FUNCTION FNORMQ(N, F)
C
      IMPLICIT NONE
      INTEGER N
      DOUBLE PRECISION F(1:N)
C
      INTEGER I
C
      FNORMQ = 0.0D0
      DO 100 I=1, N, 1
 100    FNORMQ = FNORMQ + F(I) * F(I)
C
      RETURN
C
      END
```

8.3.2 skalprod.f

```
      DOUBLE PRECISION FUNCTION FSKPR(N, F1, F2)
C
      IMPLICIT NONE
      INTEGER N
      DOUBLE PRECISION F1(1:N), F2(1:N)
C
      INTEGER I
C
      FSKPR = 0.0D0
      DO 100 I=1, N, 1
 100    FSKPR = FSKPR + F1(I) * F2(I)
C
      RETURN
C
      END
```

Da der Bereich der Floatingpointdarstellung des verwendeten Vektorrechners nur bis 16^{64} reicht, folgen nun einige Funktionen, die den Exponenten separat verwalten. Diese Funktionen werden bei einer Variante der LLL-Reduktion verwendet, die später noch ausführlicher vorgestellt wird.

8.3.3 add.f

```
      SUBROUTINE EFADD(N, LDOP1, OP1, LDOP2, OP2, LDERG, ERG)
C
      IMPLICIT NONE
      INTEGER N, LDOP1, LDOP2, LDERG
      DOUBLE PRECISION OP1(1:LDOP1, 1:2)
      DOUBLE PRECISION OP2(1:LDOP2, 1:2)
      DOUBLE PRECISION ERG(1:LDERG, 1:2)
C
      INTEGER J
      DOUBLE PRECISION TMP, DLOG2
C
      DO 100 J=1, N, 1
        TMP = MAX(OP1(J,2), OP2(J,2))
        ERG(J,1) = OP1(J,1) * 2**(OP1(J,2) - TMP) +
     A             OP2(J,1) * 2**(OP2(J,2) - TMP)
        ERG(J,2) = TMP
 100  CONTINUE
C
      DO 120 J=1, N, 1
        TMP = -ERG(J,2)
*VOCL STMT,IF(90)
        IF(ERG(J,1) .NE. 0.0) THEN
          TMP = DINT(DLOG2(DABS(ERG(J,1))))
        ENDIF
        ERG(J,1) = ERG(J,1) * 2**(-TMP)
        ERG(J,2) = ERG(J,2) + TMP
 120  CONTINUE
C
      RETURN
C
      END
```

8.3.4 mult.f

```
      SUBROUTINE EFMUL1(N, LDERG, ERG, LDOP1, OP1)
C
      IMPLICIT NONE
      INTEGER N, LDOP1, LDERG
      DOUBLE PRECISION OP1(1:LDOP1, 1:2)
      DOUBLE PRECISION ERG(1:LDERG, 1:2)
C
      INTEGER J
      DOUBLE PRECISION TMP
```

```
C
      DO 100 J=1, N, 1
        ERG(J,2) = ERG(J,2) + OP1(1,2)
        ERG(J,1) = ERG(J,1) * OP1(1,1)
 100  CONTINUE
C
      DO 120 J=1, N, 1
        TMP = 0
*VOCL STMT,IF(90)
        IF(ERG(J,1) .NE. 0.0) THEN
          TMP = DINT(DLOG(DABS(ERG(J,1))) / DLOG(2.0D0))
        ENDIF
        ERG(J,1) = ERG(J,1) * 2**(-TMP)
        ERG(J,2) = ERG(J,2) + TMP
 120  CONTINUE
C
      RETURN
C
      END
```

8.3.5 skalprod.f

```
      SUBROUTINE EFSKPR(N, LDOP1, OP1, LDOP2, OP2, LDERG, ERG)
C
      IMPLICIT NONE
      INTEGER N, LDOP1, LDOP2, LDERG
      DOUBLE PRECISION OP1(1:LDOP1, 1:2)
      DOUBLE PRECISION OP2(1:LDOP2, 1:2)
      DOUBLE PRECISION ERG(1:LDERG, 1:2)
C
      INTEGER J
      DOUBLE PRECISION MAXEXP, SUMME, TMP, DLOG2
C
      DO 100 J=1, N, 1
        ERG(J,1) = OP1(J,1) * OP2(J,1)
        ERG(J,2) = OP1(J,2) + OP2(J,2)
 100  CONTINUE
C
      IF (N .GT. 0) THEN
        MAXEXP = ERG(1,2)
        DO 120 J=2, N, 1
          IF(ERG(J,2) .GT. MAXEXP) MAXEXP = ERG(J,2)
 120    CONTINUE
C
        SUMME = 0.0
```

```
        DO 140 J=1, N, 1
          SUMME = SUMME + ERG(J,1) * 2**(ERG(J,2) - MAXEXP)
 140    CONTINUE
C
        IF (SUMME .NE. 0.0) THEN
          TMP = DINT(DLOG2(DABS(SUMME)))
          ERG(1,1) = SUMME * 2**(-TMP)
          ERG(1,2) = MAXEXP + TMP
        ELSE
          ERG(1,1) = 0.0
          ERG(1,2) = 0.0
        ENDIF
      ELSE
        ERG(1,1) = 0.0
        ERG(1,2) = 0.0
      ENDIF
C
      RETURN
C
      END
```

8.3.6 norm.f

```
      SUBROUTINE EFNORM(N, LDOP1, OP1, LDERG, ERG)
C
      IMPLICIT NONE
      INTEGER N, LDOP1, LDERG
      DOUBLE PRECISION OP1(1:LDOP1, 1:2)
      DOUBLE PRECISION ERG(1:LDERG, 1:2)
C
      INTEGER J
      DOUBLE PRECISION MAXEXP, SUMME, TMP, DLOG2
C
      DO 100 J=1, N, 1
        ERG(J,1) = OP1(J,1) * OP1(J,1)
        ERG(J,2) = OP1(J,2) * 2
 100  CONTINUE
C
      IF (N .GT. 0) THEN
        MAXEXP = ERG(1,2)
        DO 120 J=2, N, 1
          IF(ERG(J,2) .GT. MAXEXP) MAXEXP = ERG(J,2)
 120    CONTINUE
C
        SUMME = 0.0
```

```
        DO 140 J=1, N, 1
          SUMME = SUMME + ERG(J,1) * 2**(ERG(J,2) - MAXEXP)
 140    CONTINUE
C
        IF (SUMME .NE. 0.0) THEN
          TMP = DINT(DLOG2(SUMME))
          ERG(1,1) = SUMME * 2**(-TMP)
          ERG(1,2) = MAXEXP + TMP
        ELSE
          ERG(1,1) = 0.0
          ERG(1,2) = 0.0
        ENDIF
      ELSE
        ERG(1,1) = 0.0
        ERG(1,2) = 0.0
      ENDIF
C
      RETURN
C
      END
```

8.4 Datentypkonvertierung

8.4.1 eftof.f

```
      SUBROUTINE EFTOF(N, LDEF, EF, F)
C
      IMPLICIT NONE
      INTEGER LDEF, N
      DOUBLE PRECISION F(1:N)
      DOUBLE PRECISION EF(1:LDEF, 1:2)
C
      INTEGER J
C
      DO 100 J=1, N, 1
        F(J) = EF(J,1) * 2**(EF(J,2))
 100  CONTINUE
C
      RETURN
C
      END
```

8.4.2 ftoef.f

```
      SUBROUTINE FTOEF(N, F, LDEF, EF)
C
      IMPLICIT NONE
      INTEGER LDEF, N
      DOUBLE PRECISION F(1:N)
      DOUBLE PRECISION EF(1:LDEF, 1:2)
C
      INTEGER J
      DOUBLE PRECISION DLOG2
C
      DO 100 J=1, N, 1
        EF(J,2) = 0
*VOCL STMT,IF(95)
        IF (F(J) .NE. 0.0D0) THEN
          EF(J,2) = DINT(DLOG2(DABS(F(J))))
        ENDIF
        EF(J,1) = F(J) * 2**(-EF(J,2))
 100  CONTINUE
C
      RETURN
C
      END
```

8.4.3 eftovi.f

```
*VOCL TOTAL,SCALAR(8)
*VOCL TOTAL,VDOPT
      SUBROUTINE EFTOVI(N, LDEF, EF, IVAZBL, LDVIV, VIV)
C
      IMPLICIT NONE
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      INTEGER N, LDEF, LDVIV, IVAZBL
      DOUBLE PRECISION VIV(1:LDVIV, -2:IVAZBL)
      DOUBLE PRECISION EF(1:LDEF, 1:2)
C
      INTEGER I,J
      DOUBLE PRECISION MAXLD, TMP
C
      DO 100 J=1, N, 1
        VIV(J,ISIGN) = 1.0
```

```
        IF(EF(J,1) .LT. 0.0) VIV(J,ISIGN) = -1.0
 100  CONTINUE
C
      MAXLD = 0
      DO 120 J=1, N, 1
        IF(EF(J,2) .GT. MAXLD) MAXLD = EF(J,2)
 120  CONTINUE
C
      IF(MAXLD .GT. 823) CALL OVERFL('EFTOVIV   ')
      K = MAXLD / 25 + 2
C
      DO 140 I=0, K, 1
        DO 140 J=1, N, 1
          VIV(J,I) = 0.0
 140  CONTINUE
C
      DO 200 J=1, N, 1
        I = INT(EF(J,2) / 25 - 3)
        IF (I .LT. 0) I = 0
        VIV(J,I) = VIV(J,ISIGN) * EF(J,1) * 2**(EF(J,2) - 25*I)
        VIV(J,0) = DINT(VIV(J,0))
 200  CONTINUE
C
      DO 300 I=0, K-1, 1
        DO 300 J=1, N, 1
          TMP = DINT(VIV(J,I) * IVBHM1)
          VIV(J,I) = VIV(J,I) - TMP * IVBASI
          VIV(J,I+1) = VIV(J,I+1) + TMP
          IF (VIV(J,I) .GT. IVBASI) VIV(J,I) = 0.0D0
 300  CONTINUE
C
      I=K+1
 400  TMP = 0.0
      I=I-1
      DO 410 J=1, N, 1
        TMP = TMP + VIV(J,I)
 410  CONTINUE
      IF((TMP .EQ. 0.0) .AND. (I .GT. 0)) GOTO 400
      IF((TMP. EQ. 0.0) .AND. (I. EQ. 0)) I = -1
C
      DO 450 J=1, N, 1
        VIV(J, ILEAD) = I
 450  CONTINUE
C
      RETURN
C
      END
```

8.4.4 vitoef.f

```
*VOCL TOTAL,SCALAR(8)
*VOCL TOTAL,VDOPT
      SUBROUTINE VITOEF(N, IVAZBL, LDVIV, VIV, LDEF, EF, LDH, H)
C
      IMPLICIT NONE
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      INTEGER N, LDVIV, LDEF, LDH, IVAZBL
      DOUBLE PRECISION VIV(1:LDVIV, -2:IVAZBL)
      DOUBLE PRECISION EF(1:LDEF, 1:2)
      DOUBLE PRECISION H(1:LDH, 1:2)
C
C DIE LOKALEN VARIABLEN
C
      INTEGER I,J,LD
C
      LD = NINT(VIV(1, ILEAD))
C
      DO 50 I=ISIGN, LD, 1
        DO 50 J=1, N, 1
          VIV(J,I) = DNINT(VIV(J,I))
50    CONTINUE

      DO 100 J=1, N, 1
        EF(J,1) = 0
        EF(J,2) = 0
 100  CONTINUE
C
      DO 200 I=LD , 0 , -1
        DO 210 J=1, N, 1
          H(J,1) = VIV(J,I)
          H(J,2) = 25.0 * I
 210    CONTINUE
        CALL EFADD(N, LDEF, EF(1,1), LDH, H(1,1), LDEF, EF(1,1))
 200  CONTINUE
C
      DO 300 J=1, N, 1
        EF(J,1) = EF(J,1) * VIV(J,ISIGN)
 300  CONTINUE
C
      RETURN
```

```
C
      END
```

8.4.5 fvtoviv.f

```
C
C UMWANDLUNG VON FLOAT IN VEKTORINT
C
*VOCL TOTAL,SCALAR(8)
*VOCL TOTAL,VDOPT
      SUBROUTINE FVTOVI(IV, FVOP, IVAZBL, LDOP, OP)
C
      IMPLICIT NONE
      INTEGER IV, IVAZBL, LDOP
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      DOUBLE PRECISION OP(1:LDOP,-2:IVAZBL)
      DOUBLE PRECISION FVOP(1:IV)
C
C DIE LOKALEN VARIABLEN
C
      INTEGER I, J
      DOUBLE PRECISION TMP
C
C BELEGUNG DES ERSTEN ELEMENTES MIT DEM FLOATWERT
C
      DO 100 J=1, IV, 1
        OP(J,ISIGN) = 1.0
        IF (FVOP(J) .LT. 0.0) THEN
          OP(J,ISIGN) = -1.0
        ENDIF
        OP(J,0) = DNINT(FVOP(J) * OP(J,ISIGN))
 100  CONTINUE
C
C DIE ZAHL AUFTEILEN
C
      DO 200 I=0, IVAZBL-1, 1
*VOCL LOOP,TEMP(TMP)
        DO 200 J=1, IV, 1
          TMP = DINT(OP(J,I) * IVBHM1)
          OP(J,I) = OP(J,I) - TMP * IVBASI
          OP(J,I+1) = TMP
          IF (OP(J,I) .GE. IVBASI) OP(J,I) = 0.0D0
 200  CONTINUE
C
```

```
      I = IVAZBL + 1
 305    TMP = 0.0
        I = I - 1
        DO 300 J=1, IV, 1
          TMP = TMP + OP(J,I)
 300    CONTINUE
      IF ((TMP .EQ. 0.0) .AND. (I .GT. 0)) GOTO 305
      IF ((I .EQ. 0) .AND. (TMP .EQ. 0.0)) I = -1
C
      DO 310 J=1, IV, 1
        OP(J,ILEAD) = I
 310  CONTINUE
C
      IF (I .GE. IVAZBL) THEN
        CALL OVERFL('FVTOVIV   ')
      ENDIF
C
      RETURN
C
      END
```

8.4.6 vivtofv.f

```
C
C UMWANDLUNG VON VEKTORINT IN FLOAT
C
*VOCL TOTAL,SCALAR(8)
*VOCL TOTAL,VDOPT
      SUBROUTINE VITOFV(IV, IVAZBL, LDOP, OP, FVOP)
C
      IMPLICIT NONE
      INTEGER IV, IVAZBL, LDOP
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      DOUBLE PRECISION OP(1:LDOP,-2:IVAZBL)
      DOUBLE PRECISION FVOP(1:IV)
C
C DIE LOKALEN VARIABLEN
C
      INTEGER I, J
      DOUBLE PRECISION FAKTOR
C
C INITIALISIERUNG
C
      FAKTOR = IVBASI
```

```
      DO 100 J=1, IV, 1
        FVOP(J) = 0.0
 100  CONTINUE
C
C KONVERTIERUNG
C
      DO 200 I=OP(1,ILEAD), 0, -1
        DO 200 J=1, IV, 1
          FVOP(J) = FVOP(J) * FAKTOR + OP(J, I)
 200  CONTINUE
C
C VORZEICHEN UEBERTRAGEN
C
      DO 300 J=1, IV, 1
        FVOP(J) = FVOP(J) * OP(J,ISIGN)
 300  CONTINUE
C
      RETURN
C
      END
```

8.4.7 fvstoviv.f

```
C
C UMWANDLUNG VON SKALEDFLOAT IN VEKTORINT
C
*VOCL TOTAL,SCALAR(8)
*VOCL TOTAL,VDOPT
      SUBROUTINE FSTOVI(IV, FVOP, IVAZBL, LDOP, OP, POS)
C
      IMPLICIT NONE
      INTEGER IV, IVAZBL, LDOP, POS
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      DOUBLE PRECISION OP(1:LDOP,-2:IVAZBL)
      DOUBLE PRECISION FVOP(1:IV)
C
C DIE LOKALEN VARIABLEN
C
      INTEGER I, J
      DOUBLE PRECISION TMP
C LOESCHEN DER ERSTEN POS-1 STELLEN
      DO 90 I=0, POS-1, 1
        DO 90 J=1, IV, 1
          OP(J,I) = 0.0D0
```

```
 90   CONTINUE
C
C BELEGUNG DES POS.TEN ELEMENTES MIT DEM FLOATWERT
C
      DO 100 J=1, IV, 1
        OP(J,ISIGN) = 1.0
        IF (FVOP(J) .LT. 0.0) THEN
          OP(J,ISIGN) = -1.0
        ENDIF
        OP(J,POS) = DNINT(FVOP(J) * OP(J,ISIGN))
 100  CONTINUE
C
C DIE ZAHL AUFTEILEN
C
      DO 200 I=POS, IVAZBL-1, 1
        DO 200 J=1, IV, 1
          TMP = DINT(OP(J,I) * IVBHM1)
          OP(J,I) = OP(J,I) - TMP * IVBASI
          OP(J,I+1) = TMP
          IF (OP(J,I) .GT. IVLOWB) OP(J,I) = 0.0D0
 200  CONTINUE
C
      DO 250 J=1, IV, 1
        OP(J,0) = NINT(OP(J,0))
 250  CONTINUE
C
      I = IVAZBL + 1
 305    TMP = 0.0
        I = I - 1
        DO 300 J=1, IV, 1
          TMP = TMP + OP(J,I)
 300    CONTINUE
      IF ((TMP .EQ. 0.0) .AND. (I .GT. 0)) GOTO 305
      IF ((I .EQ. 0) .AND. (TMP .EQ. 0.0)) I = -1
C
      DO 310 J=1, IV, 1
        OP(J,ILEAD) = I
 310  CONTINUE
C
      IF (I .GE. IVAZBL) THEN
        CALL OVERFL('FVSTOVIV  ')
      ENDIF
C
      RETURN
C
      END
```

8.4.8 vivtofvs.f

```
C
C UMWANDLUNG VON VEKTORINT IN SKALIERTEN FLOAT
C
*VOCL TOTAL,SCALAR(8)
*VOCL TOTAL,VDOPT
      SUBROUTINE VITOFS(IV, IVAZBL, LDOP, OP, FVOP, POS)
C
      IMPLICIT NONE
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      INTEGER IV, IVAZBL, LDOP, POS
      DOUBLE PRECISION OP(1:LDOP,-2:IVAZBL)
      DOUBLE PRECISION FVOP(1:IV)
C
C DIE LOKALEN VARIABLEN
C
      INTEGER I, J
      DOUBLE PRECISION FAKTOR
C
C INITIALISIERUNG
C
      FAKTOR = IVBASI
      DO 100 J=1, IV, 1
        FVOP(J) = 0.0
 100  CONTINUE
C
C KONVERTIERUNG
C
      DO 200 I=OP(1,ILEAD), POS, -1
        DO 200 J=1, IV, 1
          FVOP(J) = FVOP(J) * FAKTOR + OP(J, I)
 200  CONTINUE
C
C VORZEICHEN UEBERTRAGEN
C
      DO 300 J=1, IV, 1
        FVOP(J) = FVOP(J) * OP(J,ISIGN)
 300  CONTINUE
C
      RETURN
C
      END
```

8.5 LLL-Reduktion

8.5.1 l3fmlq.f

```
      SUBROUTINE L3FMLQ(M, N, KI, KOUT, INDEXV, DELTA, MAXMUE, LDB,
     +               IVAZBL, B, BFP, NORM, HOEHEN, LDMUE, MUE, L3INFO)
C
      IMPLICIT NONE
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      INTEGER M, N, KI, LDB, KOUT, LDMUE
      INTEGER IVAZBL
      INTEGER INDEXV(1:M+1)
      INTEGER L3INFO(1:17)
      DOUBLE PRECISION DELTA, MAXMUE, HOEHEN(1:M), NORM(1:M)
      DOUBLE PRECISION MUE(1:LDMUE, 1:M), BFP(1:LDB, 1:M+1)
      DOUBLE PRECISION B(1:LDB, -2:IVAZBL, M+1)
C
      DOUBLE PRECISION FNORMQ, FSKPR
      INTEGER J, I, K1,  KMAX, EXCH, REDS, SKORR, OLDRED, TIME
      INTEGER K, TI(1:4), VTIME, IM1, IK, LIMIT
      DOUBLE PRECISION MAXMI, TMP, S, T, ALOESN
      LOGICAL KONTR, KINCED, OKM
C
C INITIALISIERUNG DER VARIABLEN
C
      IM1 = INDEXV(M+1)
      ALOESN = L3INFO(16)**2 * L3INFO(15) + 1.0 +
     +   L3INFO(17)**2 * (L3INFO(14)-L3INFO(15))
      ALOESN = SQRT(ALOESN) * 1.1
      MAXMI = MAXMUE
      OKM = .TRUE.
      LIMIT = MIN(M, KOUT)
      K = MAX(KI,2)
      IF(K .GT. M) K=2
      EXCH = 0
      REDS = 0
      SKORR = 0
      L3INFO(7) = M
      L3INFO(8) = M
      KONTR =  (L3INFO(9) .NE. 0)
      KINCED = (L3INFO(10) .NE. 0)
C
C ZEITMESSUNG STARTEN
C
```

```
      CALL CLOCKV(TI(1),TI(2),1,0)
C
C INIT-STEP
C
      IF(K .EQ. 2) THEN
        HOEHEN(1) = FNORMQ(N, BFP(1, INDEXV(1)))
      ENDIF
      NORM(1) = SQRT(HOEHEN(1))
      DO 90 I=2, LIMIT, 1
        NORM(I) = DSQRT(FNORMQ(N, BFP(1, INDEXV(I))))
 90   CONTINUE
      DO 92 I=1, LIMIT, 1
        MUE(I,I) = 1.0D0
        IF(NORM(I) .LE. 0.5D0) THEN
          L3INFO(8) = I
          GOTO 800
        ENDIF
        IF (NORM(I) .LT. ALOESN) THEN
          IF(ISLOES(LDB, BFP(1,INDEXV(I)), L3INFO)) THEN
            L3INFO(8) = I
            L3INFO(11) = I
            GOTO 800
          ENDIF
        ENDIF
 92   CONTINUE
C
C ENDE INIT-STEP
C
      KMAX = K
      L3INFO(5) = 0
C
C JETZT DIE WHILE-SCHLEIFE UEBER DIE ITERATIONEN
C
 100  IF (K .GT. LIMIT) GOTO 800
        T = NORM(K) ** 2
        IK = INDEXV(K)
C
C BERECHNUNG DER GRAM-SCHMIDT-KOEFFIZIENTEN UND HOEHEN
C
*VOCL LOOP,TEMP(S)
        DO 120 J=1, K-1, 1
          S = FSKPR(N, BFP(1, INDEXV(J)), BFP(1, IK))
          IF (ABS(S) .LT. (NORM(K) * NORM(J) * 1D-7)) THEN
            CALL VISKPR(N,IVAZBL,LDB,B(1,ISIGN,INDEXV(J)),
     +             LDB,B(1,ISIGN,IK), LDB, B(1,ISIGN,IM1))
            CALL VITOFV(1, IVAZBL, LDB, B(1,ISIGN,IM1), S)
            SKORR = SKORR + 1
```

```
          ENDIF
          MUE(J,K) = S
 120    CONTINUE
        DO 140 J=1, K-1, 1
*VOCL LOOP,SCALAR(8)
*VOCL LOOP,VDOPT
          DO 140 I=1, J-1 ,1
            MUE(J,K) = MUE(J,K) - MUE(I,J) * MUE(I,K)
 140    CONTINUE
*VOCL LOOP,TEMP(S)
        DO 160 J=1, K-1, 1
          S = MUE(J,K)
          MUE(J,K) = S / HOEHEN(J)
          T = T - S * MUE(J,K)
 160    CONTINUE
        HOEHEN(K) = T
C
        K1 = K - 1
        OLDRED = REDS
C
C JETZT DIE LAENGENREDUKTION
C
        DO 200 J=K1, 1, -1
          S = DNINT(MUE(J,K))
          IF(S .NE. 0.0D0) THEN
            IF (ABS(S) .LT. IVBASI) THEN
              I = INT(S)
              IF ((B(1,ILEAD,IK) .LT. 0.5D0) .AND.
     +              (B(1,ILEAD,INDEXV(J)) .LT. 0.5D0)) THEN
                CALL L3SMS(N, IVAZBL, LDB, B(1,ISIGN,IK), I,
     +                          LDB, B(1, ISIGN, INDEXV(J)))
              ELSE
                CALL SUBMUL(N, IVAZBL, LDB, B(1,ISIGN,IK), I,
     +                          LDB, B(1, ISIGN, INDEXV(J)))
              ENDIF
            ELSE
              CALL FVTOVI(1, S, IVAZBL, LDB, B(1,ISIGN,IM1))
              CALL ISUBML(N, IVAZBL, LDB, B(1, ISIGN, IK), LDB,
     +         B(1, ISIGN, IM1), LDB, B(1, ISIGN, INDEXV(J)))
            ENDIF
            REDS = REDS + 1
            IF(ABS(S) .GT. MAXMI) KONTR = .TRUE.
*VOCL LOOP,SCALAR(8)
*VOCL LOOP,VDOPT
            DO 220 I=1, J, 1
              MUE(I,K) = MUE(I,K) - MUE(I,J) * S
 220        CONTINUE
```

```
          ENDIF
 200    CONTINUE
C
        IF(OLDRED .NE. REDS) THEN
          IF (B(1,ILEAD,IK) .EQ. 0.0D0) THEN
            DO 300 J=1,N,1
              BFP(J,IK) = B(J,0,IK)*B(J,ISIGN,IK)
 300        CONTINUE
          ELSE
            CALL VITOFV(N, IVAZBL, LDB, B(1, ISIGN, IK), BFP(1, IK))
          ENDIF
          NORM(K) = DSQRT(FNORMQ(N, BFP(1, IK)))
          IF(NORM(K) .LT. 1.0D0) THEN
            L3INFO(8) = K
            GOTO 800
          ENDIF
          IF (NORM(I) .LT. ALOESN) THEN
            IF(ISLOES(LDB, BFP(1,IK), L3INFO)) THEN
              L3INFO(8) = K
              L3INFO(11) = K
              GOTO 800
            ENDIF
          ENDIF
        ENDIF
C
        IF (KONTR) THEN
          SKORR = SKORR + 1
          KONTR = .FALSE.
          IF(KINCED) THEN
            KINCED = .FALSE.
            K = K - 1
            IF (K .LT. 2) K = 2
          ENDIF
        ELSE
          S = NORM(K)*NORM(K)
          DO 240 I=1, MIN(K, L3INFO(6))-1, 1
            IF(DELTA * HOEHEN(I) .GT. S) THEN
              OKM = .FALSE.
              GOTO 260
            ELSE
              S = S - MUE(I,K) * MUE(I,K) * HOEHEN(I)
            ENDIF
 240      CONTINUE
 260      CONTINUE
C
          IF (OKM) THEN
            IF(HOEHEN(K1) * (DELTA - MUE(K1,K)**2) .GT. HOEHEN(K)) THEN
```

```
                I = K1
                OKM = .FALSE.
              ENDIF
            ENDIF
C
            IF( .NOT. OKM) THEN
              IF ((I+1) .NE. K) L3INFO(5) = L3INFO(5) + 1
              TMP = NORM(K)
              K1 = INDEXV(K)
              OKM = .TRUE.
              DO 280 J=K, I+1, -1
                NORM(J) = NORM(J-1)
                INDEXV(J) = INDEXV(J-1)
 280          CONTINUE
              INDEXV(I) = K1
              NORM(I) = TMP
              EXCH = EXCH + 1
              KINCED = .FALSE.
              L3INFO(7) = MIN(L3INFO(7), I)
              K = MAX(I,2)
              IF(K .EQ. 2) THEN
                HOEHEN(1) = FNORMQ(N, BFP(1, INDEXV(1)))
              ENDIF
            ELSE
              K = K + 1
              L3INFO(8) = K
              KINCED = .TRUE.
              KMAX = MAX(K, KMAX)
            ENDIF
          ENDIF
C
          GOTO 100
C
C UND NUN DAS ENDE
C
 800  CONTINUE
      CALL CLOCKV(TI(3),TI(4),1,0)
      TIME = TI(4)-TI(2)
      VTIME = TI(3)-TI(1)
      L3INFO(3) = L3INFO(3) + REDS
      L3INFO(2) = L3INFO(2) + EXCH
      L3INFO(4) = L3INFO(4) + SKORR
      L3INFO(1) = L3INFO(1) + TIME
      L3INFO(12) = L3INFO(12) + VTIME
C
      RETURN
C
```

```
      END
```

8.5.2 l3sms.f

```
      SUBROUTINE L3SMS(VLEN, IVAZBL, LDOP1, OP1, L, LDOP2, OP2)
C
      IMPLICIT NONE
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      INTEGER VLEN, LDOP1, LDOP2, L, IVAZBL
      DOUBLE PRECISION OP1(1:LDOP1, -2:IVAZBL)
      DOUBLE PRECISION OP2(1:LDOP2, -2:IVAZBL)
C
      INTEGER J
      DOUBLE PRECISION TMP
C
      DO 100 J=1,VLEN,1
        OP1(J,0) = OP1(J,0)*OP1(J,ISIGN)-L*OP2(J,0)*OP2(J,ISIGN)
        OP1(J,ISIGN) = 1.0
        IF (OP1(J,0) .LT. 0.0D0) OP1(J,ISIGN) = -1.0D0
        OP1(J,0) = OP1(J,0) * OP1(J,ISIGN)
        OP1(J,1) = DINT(OP1(J,0) * IVBHM1)
 100  CONTINUE
C
      TMP = 0.0D0
      DO 110 J=1,VLEN,1
        TMP = TMP + OP1(J,1)
 110  CONTINUE
C
      IF (TMP .GT. 0.0D0) THEN
        DO 120 J=1,VLEN,1
          OP1(J,ILEAD) = 1.0D0
          OP1(J,0) = OP1(J,0) - IVBASI * OP1(J,1)
 120    CONTINUE
      ENDIF
C
      RETURN
C
      END
```

8.5.3 l3efml.f

Die folgende Variante der LLL-Reduktion geht von den folgenden Annahmen aus, die sich in den durchgeführten Testläufen bestätigt haben:

- Die Bitlänge der Zahlen, die bereits in die Berechnung eingegangen sind, sinkt sehr schnell. Wenn ein Fünftel der Basis bearbeitet ist, tauchen in diesem Teil keine Zahlen mit Bitlänge größer als 70 auf.
- Die Elemente der Vektoren, die zum erstenmal längenreduziert werden, werden in jedem Längenreduktionsschritt um etwa 50 Bits kleiner.

Dies führte zu folgender Veränderung der LLL-Reduktion gegenüber der Funktion `l3fmlq.f`: am Anfang, wenn die Zahlen noch zu groß sind, wird eine Vorreduktion (`l3efvr.f`) durchgeführt, die mit Floatingpointzahlen mit einem größeren Wertebereich arbeitet. Nachdem etwa ein Fünftel der Basis bearbeitet ist, sind die Zahlen in diesem Bereich hinreichend klein, um mit der normalen Floatingpointdarstellung weiter zu arbeiten. Beim Erhöhen der Stufe wird kontrolliert, ob der neu hinzugenommene Vektor zu groß ist. In diesem Fall wird eine spezielle Version der Längenreduktion (`l3efsr.f`) aufgerufen, die nur auf den führenden Bits arbeitet.

```
      SUBROUTINE L3EFML(M, N, INDEXV, DELTA, MAXMUE, LDB, IVAZBL, B,
     +                               BFP, NORM, HOEHEN, MUE, L3INFO)
C
      IMPLICIT NONE
      INTEGER M, N, LDB, IVAZBL
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      INTEGER INDEXV(1:M+1)
      INTEGER L3INFO(1:17)
      DOUBLE PRECISION DELTA, MAXMUE, HOEHEN(1:M), NORM(1:M)
      DOUBLE PRECISION MUE(1:M, 1:M), BFP(1:LDB, 1:M+1)
      DOUBLE PRECISION B(1:LDB, -2:IVAZBL, M+1)
C
      DOUBLE PRECISION FNORMQ, FSKPR
      INTEGER J, I, K1,  KMAX, EXCH, REDS, SKORR, OLDRED, TIME
      INTEGER K, TI(1:4), IM1, IK, VTIME
      DOUBLE PRECISION MAXMI, TMP, S, T
      LOGICAL KONTR, KINCED, OKM
C
C INITIALISIERUNG DER VARIABLEN
C
      IM1 = INDEXV(M+1)
      MAXMI = MAXMUE
      OKM = .TRUE.
      K=2
```

```
      EXCH = 0
      REDS = 0
      SKORR = 0
      L3INFO(7) = M
      L3INFO(8) = M
      KONTR =  (L3INFO(9) .NE. 0)
      KINCED = (L3INFO(10) .NE. 0)
C
C ZEITMESSUNG STARTEN
C
      CALL CLOCKV(TI(1),TI(2),1,0)
C
C INIT-STEP
C
      CALL L3EFVR(M,N,INDEXV(1),DELTA,MAXMUE,LDB,IVAZBL,
     +     B(1,ISIGN,1),BFP(1,1),NORM(1),HOEHEN(1),MUE(1,1),L3INFO)
      DO 10 I=1, M, 1
        IF (B(1,ILEAD,INDEXV(I)) .LT. 3.1D0) THEN
          CALL VITOFV(N,IVAZBL,LDB,B(1,ISIGN,INDEXV(I)),
     +                                BFP(1,INDEXV(I)))
          NORM(I) = DSQRT(FNORMQ(N,BFP(1,INDEXV(I))))
        ENDIF
        MUE(I,I) = 1.0D0
 10   CONTINUE
      HOEHEN(1) = FNORMQ(N,BFP(1,INDEXV(1)))
      IF(NORM(1) .LT. 0.5D0) THEN
        L3INFO(8) = 1
        GOTO 800
      ENDIF
      IF(NORM(2) .LT. 0.5D0) THEN
        L3INFO(8) = 2
        GOTO 800
      ENDIF
      IF(ISLOES(LDB,BFP(1,INDEXV(1)),L3INFO(1))) THEN
        L3INFO(8) = 1
        L3INFO(11) = 1
        GOTO 800
      ENDIF
      IF(ISLOES(LDB,BFP(1,INDEXV(2)),L3INFO(1))) THEN
        L3INFO(8) = 2
        L3INFO(11) = 2
        GOTO 800
      ENDIF
C
C ENDE INIT-STEP
C
      KMAX = 2
```

```
      L3INFO(5) = 0
C
C JETZT DIE WHILE-SCHLEIFE UEBER DIE ITERATIONEN
C
 100  IF (K .GT. M) GOTO 800
        T = NORM(K) ** 2
        IK = INDEXV(K)
C
C BERECHNUNG DER GRAM-SCHMIDT-KOEFFIZIENTEN UND HOEHEN
C
*VOCL LOOP,TEMP(S)
        DO 120 J=1, K-1, 1
          S = FSKPR(N, BFP(1, INDEXV(J)), BFP(1, IK))
          IF (ABS(S) .LT. (NORM(K) * (NORM(J) * 1D-7))) THEN
            CALL VISKPR(N,IVAZBL,LDB,B(1,ISIGN,INDEXV(J)),
     +             LDB,B(1,ISIGN,IK), LDB, B(1,ISIGN,IM1))
            CALL VITOFV(1, IVAZBL, LDB, B(1,ISIGN,IM1), S)
            SKORR = SKORR + 1
          ENDIF
          MUE(J,K) = S
 120    CONTINUE
        DO 140 J=1, K-1, 1
*VOCL LOOP,SCALAR(8)
*VOCL LOOP,VDOPT
          DO 140 I=1, J-1,1
            MUE(J,K) = MUE(J,K) - MUE(I,J) * MUE(I,K)
 140    CONTINUE
*VOCL LOOP,TEMP(S)
        DO 160 J=1, K-1, 1
          S = MUE(J,K)
          MUE(J,K) = S / HOEHEN(J)
          T = T - S * MUE(J,K)
 160    CONTINUE
        HOEHEN(K) = T
C
        K1 = K - 1
        OLDRED = REDS
C
C JETZT DIE LAENGENREDUKTION
C
        DO 200 J=K1, 1, -1
          S = DNINT(MUE(J,K))
          IF(S .NE. 0.0D0) THEN
            IF (ABS(S) .LT. IVBASI) THEN
              I = INT(S)
              IF ((B(1,ILEAD,IK) .LT. 0.5D0) .AND.
     +           (B(1,ILEAD,INDEXV(J)) .LT. 0.5D0)) THEN
```

```
                CALL L3SMS(N, IVAZBL, LDB, B(1,ISIGN,IK), I,
     +                          LDB, B(1,ISIGN,INDEXV(J)))
              ELSE
                CALL SUBMUL(N, IVAZBL, LDB, B(1,ISIGN,IK), I,
     +                          LDB, B(1, ISIGN, INDEXV(J)))
              ENDIF
            ELSE
              CALL FVTOVI(1, S, IVAZBL, LDB, B(1,ISIGN,IM1))
              CALL ISUBML(N, IVAZBL, LDB, B(1, ISIGN, IK), LDB,
     +         B(1, ISIGN, IM1), LDB, B(1, ISIGN, INDEXV(J)))
            ENDIF
            REDS = REDS + 1
            IF(ABS(S) .GT. MAXMI) KONTR = .TRUE.
*VOCL LOOP,SCALAR(8)
*VOCL LOOP,VDOPT
            DO 220 I=1, J, 1
              MUE(I,K) = MUE(I,K) - MUE(I,J) * S
 220        CONTINUE
          ENDIF
 200    CONTINUE
C
        IF(OLDRED .NE. REDS) THEN
          IF (B(1,ILEAD,IK) .EQ. 0.0D0) THEN
            DO 300 J=1,N,1
              BFP(J,IK) = B(J,0,IK)*B(J,ISIGN,IK)
 300        CONTINUE
          ELSE
            CALL VITOFV(N, IVAZBL, LDB, B(1, ISIGN, IK), BFP(1, IK))
          ENDIF
          NORM(K) = DSQRT(FNORMQ(N, BFP(1, IK)))
          IF(NORM(K) .LT. 1.0D0) THEN
            L3INFO(8) = K
            GOTO 800
          ENDIF
          IF(ISLOES(LDB, BFP(1,IK), L3INFO)) THEN
            L3INFO(8) = K
            L3INFO(11) = K
            GOTO 800
          ENDIF
        ENDIF
C
        IF (KONTR) THEN
          SKORR = SKORR + 1
          KONTR = .FALSE.
          IF(KINCED) THEN
            KINCED = .FALSE.
            K = K - 1
```

```
            IF (K .LT. 2) K = 2
          ENDIF
        ELSE
          S = NORM(K)*NORM(K)
          DO 240 I=1, MIN(K, L3INFO(6))-1, 1
            IF(DELTA * HOEHEN(I) .GT. S) THEN
              OKM = .FALSE.
              GOTO 260
            ELSE
              S = S - MUE(I,K) * MUE(I,K) * HOEHEN(I)
            ENDIF
 240      CONTINUE
 260      CONTINUE
C
          IF (OKM) THEN
            IF(HOEHEN(K1) * (DELTA - MUE(K1,K)**2) .GT. HOEHEN(K)) THEN
              I = K1
              OKM = .FALSE.
            ENDIF
          ENDIF
C
          IF( .NOT. OKM) THEN
            IF ((I+1) .NE. K) L3INFO(5) = L3INFO(5) + 1
            TMP = NORM(K)
            K1 = INDEXV(K)
            OKM = .TRUE.
            DO 280 J=K, I+1, -1
              NORM(J) = NORM(J-1)
              INDEXV(J) = INDEXV(J-1)
 280        CONTINUE
            INDEXV(I) = K1
            NORM(I) = TMP
            EXCH = EXCH + 1
            KINCED = .FALSE.
            L3INFO(7) = MIN(L3INFO(7), I)
            K = MAX(I,2)
            IF(K .EQ. 2) THEN
              HOEHEN(1) = FNORMQ(N, BFP(1, INDEXV(1)))
            ENDIF
          ELSE
            K = K + 1
            L3INFO(8) = K
            KINCED = .TRUE.
            KMAX = MAX(KMAX,K)
            IF (K .LE. M) THEN
              IF (B(1,ILEAD,INDEXV(K)) .GT. 3.1D0) THEN
                CALL L3EFSR(M,N,K,INDEXV(1),DELTA,MAXMUE,LDB,IVAZBL,
```

```
     +                            B(1,ISIGN,1),BFP(1,1),NORM(1),
     +                            HOEHEN(1),MUE(1,1),L3INFO(1))
               IF (B(1,ILEAD,INDEXV(K)) .LT. 3.1D0) THEN
                 CALL VITOFV(N,IVAZBL,LDB,B(1,ISIGN,INDEXV(K)),
     +                                       BFP(1,INDEXV(K)))
                 NORM(K) = DSQRT(FNORMQ(N,BFP(1,INDEXV(K))))
                 MUE(K,K) = 1.0D0
               ELSE
                 KINCED = .FALSE.
                 K = K - 1
               ENDIF
             ENDIF
           ENDIF
         ENDIF
       ENDIF
C
       GOTO 100
C
C UND NUN DAS ENDE
C
 800  CONTINUE
      CALL CLOCKV(TI(3),TI(4),1,0)
      TIME = TI(4)-TI(2)
      VTIME = TI(3)-TI(1)
      L3INFO(3) = L3INFO(3) + REDS
      L3INFO(2) = L3INFO(2) + EXCH
      L3INFO(4) = L3INFO(4) + SKORR
      L3INFO(1) = L3INFO(1) + TIME
      L3INFO(12) = L3INFO(12) + VTIME
C
      RETURN
C
      END
```

8.5.4 l3efvr.f

```
*VOCL TOTAL,SCALAR(8)
      SUBROUTINE L3EFVR(M, N, INDEXV, DELTA, MAXMUE, LDB, IVAZBL, B,
     +                            BFP, NORM, HOEHEN, MUE, L3INFO)
C
      IMPLICIT NONE
      INTEGER M, N, LDB, IVAZBL
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      INTEGER INDEXV(1:M+1)
```

```
      INTEGER L3INFO(1:17)
      DOUBLE PRECISION HOEHEN(1:M,1:2), NORM(1:M,1:2)
      DOUBLE PRECISION MUE(1:M, 1:M+1), BFP(1:LDB, 1:2, 1:M+1)
      DOUBLE PRECISION DELTA, MAXMUE, B(1:LDB, -2:IVAZBL, 1:M+1)
C
      INTEGER J, I, K1,  KMAX, EXCH, REDS, SKORR, OLDRED, K
      INTEGER IKM
      DOUBLE PRECISION MAXMI(1:2), TMP(1:2), S(1:2), T(1:2)
      DOUBLE PRECISION DMP(1:2)
      DOUBLE PRECISION DLOG2
      LOGICAL KONTR, KINCED, OKM, VILOES
C
C INITIALISIERUNG DER VARIABLEN
C
      MAXMI(2) = DNINT(DLOG2(MAXMUE))
      MAXMI(1) = MAXMUE / 2**MAXMI(2)
      OKM = .TRUE.
      K=2
      EXCH = 0
      REDS = 0
      SKORR = 0
      L3INFO(7) = M
      L3INFO(8) = M
      KONTR =  (L3INFO(9) .NE. 0)
      KINCED = (L3INFO(10) .NE. 0)
C
C INIT-STEP
C
      CALL VITOEF(N,IVAZBL,LDB,B(1,ISIGN,INDEXV(1)),LDB,
     +             BFP(1,1,INDEXV(1)), LDB, BFP(1,1,INDEXV(3)))
      CALL EFNORM(N, LDB, BFP(1,1,INDEXV(1)), LDB,
     +                                     BFP(1,1,INDEXV(2)))
      HOEHEN(1,1) = BFP(1,1,INDEXV(2))
      HOEHEN(1,2) = BFP(1,2,INDEXV(2))
      NORM(1,2) = DINT(HOEHEN(1,2) / 2.0D0)
      NORM(1,1) = DSQRT(HOEHEN(1,1) *
     +              2**(HOEHEN(1,2) - 2 * NORM(1,2)))
      CALL VITOEF(N,IVAZBL,LDB,B(1,ISIGN,INDEXV(2)),LDB,
     +             BFP(1,1,INDEXV(2)), LDB, BFP(1,1,INDEXV(3)))
      CALL EFNORM(N, LDB, BFP(1,1,INDEXV(2)), LDB,
     +                                 BFP(1,1,INDEXV(3)))
      NORM(2,2) = DINT(BFP(1,2,INDEXV(3)) / 2.0D0)
      NORM(2,1) = DSQRT(BFP(1,1,INDEXV(3)) *
     +                2**(BFP(1,2,INDEXV(3)) - 2 * NORM(2,2)))

      DO 92 I=1, 2, 1
        IF(NORM(I,2) .LT. 1.0D0) THEN
```

```
            IF((NORM(I,1) * 2**NORM(I,2)) .LT. 0.5D0) THEN
              L3INFO(8) = I
              GOTO 800
            ENDIF
          ENDIF
          IF(VILOES(LDB, IVAZBL, LDB, B(1,ISIGN,INDEXV(I)), L3INFO)) THEN
            L3INFO(8) = I
            L3INFO(11) = I
            GOTO 800
          ENDIF
          MUE(I,I)= 1.0D0
 92     CONTINUE
C
C ENDE INIT-STEP
C
       KMAX = K
       L3INFO(5) = 0
C
C JETZT DIE WHILE-SCHLEIFE UEBER DIE ITERATIONEN
C
 100   IF (K .GT. M) GOTO 800
         T(1) = NORM(K,1) ** 2
         T(2) = NORM(K,2) * 2
         MUE(K,K+1) = 0.0D0
         IKM = INDEXV(KMAX+1)
C
C BERECHNUNG DER GRAM-SCHMIDT-KOEFFIZIENTEN UND HOEHEN
C
         DO 120 J=1, K-1, 1
           CALL EFSKPR(N, LDB, BFP(1,1, INDEXV(J)), LDB,
     +                 BFP(1, 1, INDEXV(K)), LDB, BFP(1, 1, IKM))
           IF(BFP(1,2,IKM) .LT. (NORM(K,2)+NORM(J,2)-20)) THEN
             CALL VISKPR(N,IVAZBL,LDB,B(1,ISIGN,INDEXV(J)),LDB,
     +            B(1,ISIGN,INDEXV(K)), LDB, B(1,ISIGN,INDEXV(M+1)))
             CALL VITOEF(1, IVAZBL, LDB, B(1,ISIGN,INDEXV(M+1)), 1,
     +                                       S(1), LDB,BFP(1,1,IKM))
             SKORR = SKORR + 1
           ELSE
             S(1) = BFP(1,1,IKM)
             S(2) = BFP(1,2,IKM)
           ENDIF
           IF (J .EQ. 1) THEN
             TMP(1) = S(1)
             TMP(2) = S(2)
           ELSE
             IF (J .EQ. 2) THEN
               DMP(1) = MUE(1,J) * MUE(1,K) * HOEHEN(1,1)
```

```
              DMP(2) = MUE(1,K+1) + HOEHEN(1,2)
            ELSE
*VOCL LOOP,VDOPT
              DO 130 I=1, J-1, 1
                BFP(I,1,IKM) = MUE(I,J)*MUE(I,K)*HOEHEN(I,1)
                BFP(I,2,IKM) = MUE(I,K+1)+HOEHEN(I,2)
130           CONTINUE
              DMP(2) = 0.0D0
*VOCL LOOP,VDOPT,NOVREC
              DO 140 I=1, J-1, 1
                IF(BFP(I,2,IKM) .GT. DMP(2)) THEN
                  DMP(2) = BFP(I,2,IKM)
                ENDIF
140           CONTINUE
              DMP(1)=0.0D0
*VOCL LOOP,VDOPT
              DO 150 I=1, J-1, 1
                DMP(1) = DMP(1) + BFP(I,1,IKM) * 2**
     +                                (BFP(I,2,IKM) - DMP(2))
150           CONTINUE
            ENDIF
            TMP(2) = MAX(S(2), DMP(2))
            TMP(1) = S(1) * 2**(S(2)-TMP(2)) -
     +        DMP(1) * 2**(DMP(2) - TMP(2))
          ENDIF
C JETZT GILT: TMP = S - MUE(I,J) * MUE(I,K) * HOEHEN(I)
          MUE(J,K) = TMP(1) / HOEHEN(J,1)
          MUE(J,K+1) = TMP(2) - HOEHEN(J,2)
C JETZT GILT: MUE(J,K) = TMP / HOEHEN(J)
C JETZT KOMMT: T <- T - S * TMP * MUE(J,K)
          IF(T(2) .GE. TMP(2) + MUE(J,K+1)) THEN
            T(1) = T(1) - TMP(1) * MUE(J,K) *
     +        2**(MUE(J,K+1) + TMP(2) - T(2))
          ELSE
            T(1) = T(1) * 2**(T(2) - MUE(J,K+1) - TMP(2)) -
     +        MUE(J,K) * TMP(1)
            T(2) = MUE(J,K+1) + TMP(2)
          ENDIF
          IF(T(1) .NE. 0.0D0) THEN
            S(1) = DINT(DLOG2(DABS(T(1))))
            T(2) = T(2) + S(1)
            T(1) = T(1) * 2**(-S(1))
          ENDIF
 120    CONTINUE
        HOEHEN(K,1) = T(1)
        HOEHEN(K,2) = T(2)
        IF ( DABS(MUE(K-1,K)) .GT. 2) THEN
```

```
          S(2) = DINT(DLOG2(DABS(MUE(K-1,K))))
          MUE(K-1,K) = MUE(K-1,K) * 2**(-S(2))
          MUE(K-1,K+1) = MUE(K-1,K+1) + S(2)
        ENDIF
C
        K1 = K - 1
        OLDRED = REDS
C
C JETZT DIE LAENGENREDUKTION
C
        DO 200 J=K1, 1, -1
          S(1) = MUE(J,K)
          S(2) = MUE(J,K+1)
          IF(S(2) .LT. 58) THEN
            S(1) = DNINT(S(1) * 2**S(2)) * 2 ** (-S(2))
          ENDIF
          IF(S(1) .NE. 0.0D0) THEN
            IF (S(2) .LT. 24) THEN
              I = INT(S(1) * 2 ** (S(2)))
              CALL SUBMUL(N, IVAZBL, LDB, B(1,ISIGN,INDEXV(K)), I,
     +                        LDB, B(1, ISIGN, INDEXV(J)))
            ELSE
              CALL EFTOVI(1, 1, S(1), IVAZBL, LDB,
     +                                    B(1,ISIGN,INDEXV(M+1)))
              CALL ISUBML(N, IVAZBL, LDB, B(1, ISIGN, INDEXV(K)), LDB,
     +         B(1, ISIGN, INDEXV(M+1)), LDB, B(1, ISIGN, INDEXV(J)))
            ENDIF
            REDS = REDS + 1
            IF(S(2) .GT. MAXMI(2)) KONTR = .TRUE.
*VOCL LOOP,VDOPT
            DO 210 I=1, J, 1
              BFP(I,1,IKM) = MUE(I,J) * S(1)
              BFP(I,2,IKM) = S(2)
 210        CONTINUE
            CALL EFSUB(J, M, MUE(1,K), LDB, BFP(1,1,IKM), M, MUE(1,K))
          ENDIF
C NOCH SCHNELL DEN GRAM-SCHMIDT-KOEFF. IN ORDNUNG BRINGEN
          MUE(J,K) = MUE(J,K) * 2**MUE(J,K+1)
 200    CONTINUE
C
        IF(OLDRED .NE. REDS) THEN
          CALL VITOEF(N, IVAZBL, LDB, B(1, ISIGN, INDEXV(K)),
     +       LDB, BFP(1, 1,INDEXV(K)), LDB, BFP(1, 1,IKM))
          CALL EFNORM(N, LDB, BFP(1,1,INDEXV(K)), LDB,
     +                                  BFP(1,1,IKM))
          NORM(K,2) = DINT(BFP(1,2,IKM) / 2.0D0)
          NORM(K,1) = DSQRT(BFP(1,1,IKM)
```

```
     +          * 2**(BFP(1,2,IKM) - 2 * NORM(K,2)))
          IF(NORM(K,2) .LT. 1.0D0) THEN
            IF((NORM(K,1) * 2**NORM(K,2)) .LT. 0.5D0) THEN
              L3INFO(8) = K
              GOTO 800
            ENDIF
          ENDIF
          IF(VILOES(LDB, IVAZBL, LDB, B(1,ISIGN,INDEXV(K)), L3INFO))
     +                                                        THEN
            L3INFO(8) = K
            L3INFO(11) = K
            GOTO 800
          ENDIF
        ENDIF
C
        IF (KONTR) THEN
          SKORR = SKORR + 1
          KONTR = .FALSE.
          IF(KINCED) THEN
            KINCED = .FALSE.
            K = K - 1
            IF (K .LT. 2) K = 2
          ENDIF
        ELSE
          S(1) = NORM(K,1)*NORM(K,1)
          S(2) = NORM(K,2) * 2
          DO 240 I=1, MIN(K, L3INFO(6))-1, 1
            IF(DELTA*HOEHEN(I,1)*2**(HOEHEN(I,2)-S(2)) .GT. S(1)) THEN
              OKM = .FALSE.
              GOTO 260
            ELSE
              IF(HOEHEN(I,2) .LE. S(2)) THEN
                S(1) = S(1) - MUE(I,K)**2 * HOEHEN(I,1) * 2**
     +            (HOEHEN(I,2)-S(2))
              ELSE
                S(1) = S(1) * 2**(S(2)-HOEHEN(I,2)) -
     +              MUE(I,K)**2 * HOEHEN(I,1)
                S(2) = HOEHEN(I,2)
              ENDIF
            ENDIF
 240      CONTINUE
 260      CONTINUE
C
          IF (OKM) THEN
            TMP(1) = DELTA - MUE(K1, K)**2
            TMP(2) = HOEHEN(K1,2) - HOEHEN(K,2)
            IF((HOEHEN(K1,1)*TMP(1)*2**TMP(2)) .GT. HOEHEN(K,1))THEN
```

```
                I = K1
                OKM = .FALSE.
              ENDIF
            ENDIF
C
            IF( .NOT. OKM) THEN
              IF ((I+1) .NE. K) L3INFO(5) = L3INFO(5) + 1
              TMP(1) = NORM(K,1)
              TMP(2) = NORM(K,2)
              K1 = INDEXV(K)
              OKM = .TRUE.
              DO 280 J=K, I+1, -1
                NORM(J,1) = NORM(J-1,1)
                NORM(J,2) = NORM(J-1,2)
                INDEXV(J) = INDEXV(J-1)
 280          CONTINUE
              INDEXV(I) = K1
              NORM(I,1) = TMP(1)
              NORM(I,2) = TMP(2)
              EXCH = EXCH + 1
              KINCED = .FALSE.
              L3INFO(7) = MIN(L3INFO(7), I)
              K = MAX(I,2)
              IF(K .EQ. 2) THEN
                CALL EFNORM(N, LDB, BFP(1,1,INDEXV(1)),
     +                                    LDB, BFP(1,1,IKM))
                HOEHEN(1,1) = BFP(1,1,IKM)
                HOEHEN(1,2) = BFP(1,2,IKM)
              ENDIF
            ELSE
              OKM = .TRUE.
              DO 400 I=1,KMAX,1
                OKM = OKM .AND. (B(1,ILEAD,INDEXV(I)) .LE. 3.1D0)
 400          CONTINUE
              IF (OKM) GOTO 800
              OKM = .TRUE.
              K = K + 1
              L3INFO(8) = K
              KINCED = .TRUE.
              IF (KMAX .LT. K) THEN
                KMAX = K
                CALL VITOEF(N,IVAZBL,LDB,B(1,ISIGN,INDEXV(K)),LDB,
     +            BFP(1,1,INDEXV(K)), LDB, BFP(1,1,INDEXV(KMAX+1)))
                CALL EFNORM(N, LDB, BFP(1,1,INDEXV(K)), LDB,
     +                                      BFP(1,1,INDEXV(KMAX+1)))
                NORM(K,2) = DINT(BFP(1,2,INDEXV(KMAX+1))/2)
                NORM(K,1) = DSQRT(BFP(1,1,INDEXV(KMAX+1)) * 2**
```

```
     +                     (BFP(1,2,INDEXV(KMAX+1))-2*NORM(KMAX,2)))
               MUE(K,K) = 1.0D0
             ENDIF
           ENDIF
         ENDIF
C
         GOTO 100
C
C UND NUN DAS ENDE
C
 800   CONTINUE

       L3INFO(3) = L3INFO(3) + REDS
       L3INFO(2) = L3INFO(2) + EXCH
       L3INFO(4) = L3INFO(4) + SKORR
C
       RETURN
C
       END
```

8.5.5 l3efsr.f

```
*VOCL TOTAL,SCALAR(8)
      SUBROUTINE L3EFSR(M, N, KI, INDEXV, DELTA, MAXMUE, LDB, IVAZBL,
     +                             B, BFP, NORM, HOEHEN, MUE, L3INFO)
C
      IMPLICIT NONE
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      INTEGER M, N, KI, LDB, IVAZBL
      INTEGER INDEXV(1:M+1)
      INTEGER L3INFO(1:17)
      DOUBLE PRECISION HOEHEN(1:M), NORM(1:M)
      DOUBLE PRECISION MUE(1:M, 1:M+1), BFP(1:LDB, 1:M+1)
      DOUBLE PRECISION DELTA, MAXMUE, B(1:LDB, -2:IVAZBL, 1:M+1)
C
      INTEGER J, I, K1, REDS, SKORR, OLDRED, K, IM1, SF, L
      INTEGER LD1, LD2
      DOUBLE PRECISION FSKPR, DLOG2, S
      LOGICAL OKM
C
C INITIALISIERUNG DER VARIABLEN
C
```

```
      OKM = .TRUE.
      K = INDEXV(KI)
      IM1 = INDEXV(M+1)
      IF (B(1,ILEAD,K) .LT. 3.1D0) RETURN
      REDS = 0
      SKORR = 0
C
C INIT-STEP
C
 501  SF = MAX( NINT( B(1,ILEAD,K)+B(1,ILEAD,INDEXV(KI-1)) ) - 7 , 0 )
      CALL VITOFS(N, VIAZBL, LDB, B(1, ISIGN, K), BFP(1, K), SF)
      NORM( KI ) = 0
      DO 400 J=1, N, 1
        IF (NORM(KI) .LT. DABS(BFP(J,K))) NORM(KI) = DABS(BFP(J,KI))
 400  CONTINUE
C
C JETZT DIE REDUKTION MIT NORMALEN ZAHLEN
C
 500  CONTINUE
       LD1 = NINT(B(1,ILEAD,K))
C
C BERECHNUNG DER GRAM-SCHMIDT-KOEFFIZIENTEN UND HOEHEN
C
*VOCL LOOP,TEMP(S)
        DO 520 J=1, KI-1, 1
          S = FSKPR(N, BFP(1, INDEXV(J)), BFP(1, K))
          IF (ABS(S) .LT. (NORM(KI) * NORM(J) * 1D-7)) THEN
            CALL VISKPR(N,IVAZBL,LDB,B(1,ISIGN,INDEXV(J)),LDB,
     +                      B(1,ISIGN,K),LDB, B(1,ISIGN,IM1))
            CALL VITOFS(1, VIAZBL, LDB, B(1,ISIGN,IM1), S, SF)
            SKORR = SKORR + 1
          ENDIF
          MUE(J,KI) = S
 520    CONTINUE
*VOCL LOOP,VDOPT
        DO 540 J=1, KI-1, 1
          DO 540 I=1, J-1,1
            MUE(J,KI) = MUE(J,KI) - MUE(I,J) * MUE(I,KI)
 540    CONTINUE
C
        DO 550 J=1, KI-1, 1
          MUE(J,KI) = MUE(J,KI) / HOEHEN(J)
 550    CONTINUE
C
        IF (SF .GT. 0) THEN
          S = 0
          DO 560 J=1, KI-1, 1
```

```
            IF (ABS(MUE(J,KI)) .GT. S) S = ABS(MUE(J,KI))
 560      CONTINUE
          L = MAX(0,INT(DLOG2(DABS(S))))
          L = (175 - L) / 25
          L = MIN(L, SF)
          DO 570 J=1, KI-1, 1
            MUE(J,KI) = MUE(J,KI) * DBLE(IVBASI)**L
 570      CONTINUE
          SF = SF - L
        ENDIF
C
        K1 = KI - 1
        OLDRED = REDS
C
C JETZT DIE LAENGENREDUKTION
C
        DO 600 J=K1, 1, -1
          S = DNINT(MUE(J,KI))
          CALL FSTOVI(1, S, IVAZBL, LDB, B(1,ISIGN,IM1), SF)
          CALL ISUBML(N, IVAZBL, LDB, B(1, ISIGN, K), LDB,
     +                B(1, ISIGN, IM1), LDB, B(1, ISIGN, INDEXV(J)))
          REDS = REDS + 1
*VOCL LOOP,VDOPT
          DO 620 I=1, J, 1
            MUE(I,KI) = MUE(I,KI) - MUE(I,J) * S
 620      CONTINUE
 600    CONTINUE
C
        IF(OLDRED .NE. REDS) THEN
          IF(B(1,ILEAD,K) .LT. -0.5D0) THEN
            L3INFO(8) = KI
            GOTO 800
          ENDIF
          LD2 = NINT(B(1,ILEAD,K))
          IF ((LD2 .LT. 3.1D0) .OR. (LD2 .GE. LD1)) GOTO 800
          SF = MAX( NINT( B(1,ILEAD,K)+B(1,ILEAD,INDEXV(KI-1)) ) - 7
          CALL VITOFS(N, IVAZBL, LDB, B(1, ISIGN, K), BFP(1, K), SF)
          NORM( KI ) = 0
          DO 700 J=1, N, 1
            IF (NORM(KI) .LT. DABS(BFP(J,K))) NORM(KI)=DABS(BFP(J,KI))
 700      CONTINUE
          NORM(KI) = NORM(KI) * N / 2
          GOTO 500
        ENDIF
C
C UND NUN DAS ENDE
C
```

```
 800  CONTINUE
      L3INFO(3) = L3INFO(3) + REDS
      L3INFO(4) = L3INFO(4) + SKORR
C
      RETURN
C
      END
```

8.5.6 setl3inf.f

```
      SUBROUTINE SETL3I(INFO)
C
      IMPLICIT NONE
      INTEGER INFO(1:17)
C
      INTEGER I
C
C DABEI BEDEUTEN DIE EINZELNEN ELEMENTE:
C  1 CPU-ZEIT
C  2 AUSTAUSCHE
C  3 REDUKTIONSSCHRITTE
C  4 SELBSTKORREKTUREN
C  5 AUSTAUSCHE UEBER GROESSERE DISTANZ
C  6 FIRSTVECS
C  7 KLEINSTE AUSTAUSCHPOSITION
C  8 LAST-K-POS (LETZTE STUFE VON K)
C  9 KONTR (FALLS DER REDUKTIONSKOEFFIZIENT ZU GROSS WAR)
C 10 KINCED (OB DIE STUFE ERHOEHT WURDE)
C 11 LOESUNGSINDEX
C 12 VU-TIME
      DO 100 I=1,12,1
 100    INFO(I) = 0
C
C FIRSTVECS
      INFO(6) = 2
C LOESINDEX
      INFO(11) = -1
C
      RETURN
C
      END
```

8.5.7 setloesinf.f

```
      SUBROUTINE SETLOE(INFO,DIM,KDIM,ANZ,W1,W2)
C
      IMPLICIT NONE
      INTEGER INFO(1:17)
      INTEGER DIM, KDIM, ANZ, W1, W2
C
C
C DIMENSION DES VEKTORS
      INFO(13) = DIM
C ...DES KNAPPSACKPROBLEMS
      INFO(14) = KDIM
C ANZAHL DES AUFTRETENS VON W1
      INFO(15) = ANZ
C WERT W1
      INFO(16) = W1
C WERT W2
      INFO(17) = W2
C
      RETURN
C
      END
```

8.5.8 loestest.f

```
      LOGICAL FUNCTION ISLOES(N, BFP, INFO)
C
      IMPLICIT NONE
      INTEGER N
      DOUBLE PRECISION BFP(0:N-1)
      INTEGER INFO(1:17)
C
      INTEGER I,QMAL,NQMAL
      DOUBLE PRECISION W1,W2
C
      ISLOES = .TRUE.
C
      IF(BFP(0) .EQ. 0.0) ISLOES = .FALSE.
      DO 100 I=INFO(14)+1, INFO(13)-1, 1
        IF(BFP(I) .NE. 0.0D0) ISLOES = .FALSE.
 100  CONTINUE
C
```

```
      IF (ISLOES) THEN
        W1 = INFO(16)* BFP(0)
        W2 = INFO(17)* BFP(0)
        QMAL = 0
        NQMAL = 0
        DO 200 I=1, INFO(14), 1
          IF(BFP(I) .EQ. W1) QMAL = QMAL + 1
          IF(BFP(I) .EQ. W2) NQMAL = NQMAL + 1
 200    CONTINUE
        IF(QMAL .NE. INFO(15)) ISLOES = .FALSE.
        IF(NQMAL .NE. INFO(14)-INFO(15)) ISLOES = .FALSE.
      ENDIF
C
      RETURN
C
      END
```

8.5.9 viloestest.f

```
*VOCL TOTAL,SCALAR(8)
*VOCL TOTAL,VDOPT
      LOGICAL FUNCTION VILOES(N, IVAZBL, LDB, B, INFO)
C
      IMPLICIT NONE
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      INTEGER N, IVAZBL
      DOUBLE PRECISION B(0:LDB-1, -2:IVAZBL)
      INTEGER INFO(1:17)
C
      INTEGER I,QMAL,NQMAL
      DOUBLE PRECISION W1,W2
C
      VILOES = .TRUE.
C
      IF(B(0,ILEAD) .NE. 0.0D0) VILOES= .FALSE.
      IF(B(0,0) .EQ. 0.0) VILOES = .FALSE.
      DO 100 I=INFO(14)+1, INFO(13)-1, 1
        IF(B(I,0) .NE. 0.0D0) VILOES = .FALSE.
 100  CONTINUE
C
      IF (VILOES) THEN
        W1 = INFO(16) * B(0,0) * B(0,ISIGN)
        W2 = INFO(17) * B(0,0) * B(0,ISIGN)
```

```
        QMAL = 0
        NQMAL = 0
        DO 200 I=1, INFO(14), 1
          IF(B(I,0)*B(I,ISIGN) .EQ. W1) QMAL = QMAL + 1
          IF(B(I,0)*B(I,ISIGN) .EQ. W2) NQMAL = NQMAL + 1
 200    CONTINUE
        IF(QMAL .NE. INFO(15)) VILOES = .FALSE.
        IF(NQMAL .NE. INFO(14)-INFO(15)) VILOES = .FALSE.
      ENDIF
C
      RETURN
C
      END
```

8.6 Blockreduktion

8.6.1 bkzr.f

```
      SUBROUTINE BKZRML(M, N, BETA, DELTA, LDB, IVAZBL, B, BFP, C,
     +     MUE, NORM, INDEXV, GRENZE, MEMORY, ENMINF, L3INFO, ENUMTF)
C
      IMPLICIT NONE
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      INTEGER M, N, L3INFO(1:17), ENMINF(1:3), LDB, BETA
      INTEGER IVAZBL
      CHARACTER ENUMTF*(*)
      INTEGER INDEXV(1:M+2), GRENZE
      DOUBLE PRECISION DELTA, MUE(1:M,1:M), C(1:M), NORM(1:M)
      DOUBLE PRECISION BFP(1:LDB, 1:M+2), B(1:LDB, -2:IVAZBL, 1:M+2)
      INTEGER MEMORY(1:GRENZE)
C
      INTEGER H, I, J, K, L, Z, IVM1
      INTEGER MEMTOP, OFFSET, IDX, HIDX
      DOUBLE PRECISION ENUMOS, ERFOLG
C
      J = 0
      Z = 0
      MEMTOP = GRENZE / (BETA + 4)
      OFFSET = 4 * MEMTOP + 1
      CALL L3FMLQ(M+1, N, 2, M, INDEXV, DELTA, 1.0D7, LDB, IVAZBL, B,
     +                               BFP, NORM, C, M, MUE, L3INFO)
      IF (L3INFO(11) .GT. 0) RETURN
C
C die WHILE-Schleife ueber Z
```

```
C
 100  CONTINUE
        IF (Z .GT. M) RETURN
C
        J = J+1
        IF (J .EQ. M) J = 1
        K = MIN(J + BETA - 1, M)
        H = MIN(K + 1, M)
        ERFOLG = ENUMOS(J, K-J+1, C, M, MUE, MEMTOP, MEMORY(OFFSET),
     +                                  MEMORY(1), ENMINF, ENUMTF)
        PRINT '(3I4,3E15.6)',J,K,Z,ERFOLG,C(J),C(J)/ERFOLG
        IF (ERFOLG .LT. DELTA*C(J)) THEN
          IVM1 = INDEXV(M+1)
          IDX = 0
          CALL VIZERO(N, IVAZBL, LDB, B(1,ISIGN,IVM1))
          DO 200 I=J, K, 1
            IF (MEMORY(I-J+OFFSET) .NE. 0) IDX = I
            CALL ADDMUL(N, IVAZBL, LDB, B(1,ISIGN,IVM1),
     +                  MEMORY(I-J+OFFSET), LDB, B(1,ISIGN,INDEXV(I)))
 200      CONTINUE
          CALL VITOFV(N, IVAZBL, LDB, B(1,ISIGN,IVM1), BFP(1,IVM1))
          IF (MEMORY(IDX+OFFSET-J) .GT. 1) THEN
            DO 220 L=M, J, -1
              INDEXV(L+1) = INDEXV(L)
 220        CONTINUE
            INDEXV(J) = IVM1
            L3INFO(9) = 1
            L3INFO(10) = 1
            CALL L3FMLQ(M+1, N, J, H, INDEXV, DELTA, 1.0D7, LDB, IVAZBL,
     +                                B, BFP, NORM, C, M, MUE, L3INFO)
            IF (L3INFO(11) .GT. 0) RETURN
            L = L3INFO(8)
            IF (NORM(L) .LT. 0.5D0) THEN
              IVM1 = INDEXV(L)
              DO 240 I=L, M, 1
                INDEXV(I) = INDEXV(I+1)
 240          CONTINUE
              INDEXV(M+1) = IVM1
            ENDIF
            CALL L3FMLQ(M+1, N, L, H, INDEXV, DELTA, 1.0D7, LDB, IVAZBL,
     +                                B, BFP, NORM, C, M, MUE, L3INFO)
            IF(L3INFO(11) .GT. 0) RETURN
            L3INFO(9) = 0
            L3INFO(10) = 0
            Z = 0
          ELSE
            HIDX = INDEXV(IDX)
```

```
            DO 260 L = IDX, J+1, -1
              INDEXV(L) = INDEXV(L-1)
260         CONTINUE
            INDEXV(J) = INDEXV(M+1)
            INDEXV(M+1) = HIDX
            L3INFO(9) = 1
            L3INFO(10) = 1
            CALL L3FMLQ(M+1, N, J, H, INDEXV, DELTA, 1.0D7, LDB, IVAZBL,
     +                                 B, BFP, NORM, C, M, MUE, L3INFO)
            IF (L3INFO(11) .GT. 0) RETURN
            L = L3INFO(8)
            L3INFO(9) = 0
            L3INFO(10) = 0
            Z = 0
          ENDIF
        ELSE
          Z = Z + 1
          CALL L3FMLQ(M+1, N, H-1, H, INDEXV, DELTA, 1.0D7, LDB, IVAZBL,
     +                              B, BFP, NORM, C, M, MUE, L3INFO)
          IF (L3INFO(11) .GT. 0) RETURN
        ENDIF
C
      GOTO 100
C
      END
C
```

8.6.2 enum.f

```
      DOUBLE PRECISION FUNCTION ENUMOS(J, BETA, C, M, MUE, MEMTOP,
     +                              MEM, CS, ENMINF, ENUMTF)
C
      IMPLICIT NONE
      INTEGER J, MEMTOP, BETA, M
      INTEGER MEM(1:MEMTOP,1:BETA), ENMINF(1:3)
      CHARACTER*(*) ENUMTF
      DOUBLE PRECISION C(1:M),MUE(1:M,1:M)
      DOUBLE PRECISION CS(1:MEMTOP, 1:3)
C
      INTEGER IV, LAST, STUFE, I, L, COFF, T(4), IEND, IANF
      INTEGER AKTVZW, FILENO, IANZ, ANZFIL, ANZVEC
      LOGICAL GELESEN
      DOUBLE PRECISION AW, LIMIT
C
      CALL CLOCKV(T(1),T(2),1,0)
```

```
C
C INITIALISIERUNG
C
      GELESEN = .FALSE.
      COFF = J-1
      CS(1,1) = C(J)
      DO 80 L=2, BETA, 1
        MEM(1,L) = 0
  80  CONTINUE
      MEM(1,1) = 1
      STUFE = BETA
      LAST = 1
      FILENO = 0
      GOTO 399
C
C DIE SCHLEIFE UEBER DIE STUFEN
C
 100  STUFE = STUFE - 1
      IF (STUFE .LE. 0) GOTO 600
C
C KOMPRIMIERUNG DER DATEN
C
 101  LIMIT = CS(1,1)
*VOCL LOOP,VI(IV),NOVREC(MEM)
      DO 150 I=STUFE+1, BETA, 1
        L = 1
        DO 150 IV=2, LAST, 1
          IF (CS(IV,1) .LE. LIMIT) THEN
            L = L + 1
            MEM(L,I) = MEM(IV,I)
          ENDIF
 150  CONTINUE
      L = 1
*VOCL LOOP,VI(IV),NOVREC(CS)
      DO 160 IV=2, LAST, 1
        IF (CS(IV,1) .LE. LIMIT) THEN
          L = L + 1
          CS(L,1) = CS(IV,1)
        ENDIF
 160  CONTINUE
      LAST = L
      IF (LAST .LT. 2) THEN
        IF (GELESEN) GOTO 800
        GOTO 399
      ENDIF
C
C BERECHNUNG DER AKTUELLEN VERZWEIGUNGSORDNUNG
```

```
C
      IF (LAST .GE. 2) THEN
        AW = CS(2,1)
        DO 180 IV=3, LAST, 1
          AW = MIN(AW, CS(IV,1))
 180    CONTINUE
      ELSE
        AW = 0.0D0
      ENDIF
      AW = SQRT((CS(1,1)-AW)/C(COFF+STUFE))
      AKTVZW = INT(AW) + 1
C
C BERECHNUNG VON Y_T
C
*VOCL LOOP,VI(IV),NOVREC(CS)
        DO 200 IV=2, LAST, 1
          CS(IV,2) = 0.0D0
          DO 220 I=STUFE+1, BETA, 1
            CS(IV,2) = CS(IV,2) + MEM(IV,I) * MUE(STUFE+COFF, I+COFF)
 220      CONTINUE
 200    CONTINUE
C
C FALLS WIR GERADE DAS OPTIMUM NEHMEN
C
        IF (AKTVZW .EQ. 0) THEN
          DO 260 IV=2, LAST, 1
            MEM(IV,STUFE) = NINT(-CS(IV,2))
            CS(IV,1) = CS(IV,1) + (MEM(IV,STUFE)+CS(IV,2))**2 *
     +                                C(STUFE+COFF)
 260      CONTINUE
        ELSE
C
C WENN ALLE VEKTOREN IN DEN HAUPTSPEICHER PASSEN
C
          IF (2*AKTVZW*LAST+1 .LT. MEMTOP) THEN
            DO 300 I=2, 2*AKTVZW, 1
*VOCL LOOP,VI(IV),NOVREC(CS)
              DO 310 IV=2, LAST, 1
C C KOPIEREN
                CS((LAST-1)*(I-1)+IV,1) = CS(IV,1)
C Y_T KOPIEREN
                CS((LAST-1)*(I-1)+IV,2) = CS(IV,2)
 310          CONTINUE
              DO 300 L=STUFE+1, BETA, 1
C KOPIEREN DER KOEFFIZIENTEN
*VOCL LOOP,VI(IV),NOVREC(MEM)
                DO 300 IV=2, LAST, 1
```

```
                  MEM((LAST-1)*(I-1)+IV,L) = MEM(IV,L)
 300         CONTINUE
             DO 320 I=1, 2*AKTVZW, 1
               IF ((I/2)*2 .EQ. I) THEN
                 AW = (I-1) * 0.5D0
               ELSE
                 AW = I * (-0.5D0)
               ENDIF
C BERECHNUNG DER NEUEN U_T
*VOCL LOOP,VI(IV),NOVREC(CS)
               DO 320 IV=2, LAST, 1
                 MEM((LAST-1)*(I-1)+IV,STUFE) =
     +                 NINT( -CS((LAST-1)*(I-1)+IV,2) - AW)
 320         CONTINUE
             LAST = (LAST-1)*2*AKTVZW+1
             DO 340 IV=2, LAST, 1
C BERECHNUNG DER NEUEN C
               CS(IV,1) = CS(IV,1) + (MEM(IV,STUFE)+CS(IV,2))**2
     +                                     * C(COFF+STUFE)
 340         CONTINUE
           ELSE
C
C DER HAUPTSPEICHER REICHT NICHT AUS, DAHER SCHREIBEN WIR AUF PLATTE
C
             ANZVEC = (LAST-1) * AKTVZW * 2
             ANZFIL = ANZVEC / (MEMTOP - 1)
             IF (ANZFIL * (MEMTOP-1) .EQ. ANZVEC) ANZFIL = ANZFIL - 1
             IEND = 2*AKTVZW
             IANZ = (MEMTOP-1)/(LAST-1)
             IANF = (IEND - IANZ) + 1
 370         CONTINUE
             IF (ANZFIL .GE. 1) THEN
               FILENO = FILENO + 1
               CALL ENUMWR(MEM, MEMTOP, BETA, STUFE, CS, FILENO, C,
     +                                 J, LAST, IANF, IEND, ENUMTF)
               IEND = IANF - 1
               IANZ = IEND / ANZFIL
               IF (IANZ * ANZFIL .LT. IEND) IANZ = IANZ + 1
               IANF = (IEND - IANZ) + 1
               ANZFIL = ANZFIL - 1
               GOTO 370
             ELSE
C KOPIEREN DER KOEFFIZIENTEN
*VOCL LOOP,VI(IV),NOVREC(MEM)
               DO 380 I=IANF+1, IEND, 1
                 DO 380 L=STUFE+1, BETA, 1
                   DO 380 IV=2, LAST, 1
```

```
                    MEM((LAST-1)*(I-1)+IV,L)=MEM(IV,L)
 380          CONTINUE
C C UND Y_T KOPIEREN
*VOCL LOOP,VI(IV),NOVREC(CS)
              DO 385 I=IANF+1, IEND, 1
                DO 385 IV=2, LAST, 1
                  CS((LAST-1)*(I-1)+IV,1) = CS(IV,1)
                  CS((LAST-1)*(I-1)+IV,2) = CS(IV,2)
 385          CONTINUE
              DO 390 I=IANF, IEND, 1
                IF ( ((I/2)*2) .EQ. I) THEN
                  AW = (I-1) * 0.5D0
                ELSE
                  AW = I * (-0.5D0)
                ENDIF
C NEUEN KOEFFIZIENTEN BERECHNEN
                DO 390 IV=2, LAST, 2
                  MEM((LAST-1)*(I-1)+IV,STUFE) =
     +               NINT( -AW -CS((LAST-1)*(I-1)+IV,2) )
 390          CONTINUE
              LAST = (LAST-1)*(IEND+1-IANF)+1
C NEUE HOEHE BERECHNEN
              DO 395 IV=2, LAST, 1
                CS(IV,1) = CS(IV,1) + (CS(IV,2)+MEM(IV,STUFE))**2 *
     +                                                C(COFF+STUFE)
 395          CONTINUE
            ENDIF
          ENDIF
        ENDIF
C
C NEUE VEKTOREN DAZUADDIEREN
C
 399    CONTINUE
        AW = SQRT(CS(1,1)/C(COFF+STUFE))
        AKTVZW = INT(AW) + 1
        IF (.NOT. GELESEN) THEN
*VOCL LOOP,SCALAR
          DO 400 IV=LAST+1, LAST+AKTVZW, 1
            MEM(IV,STUFE) = IV - LAST
            CS(IV,1) = MEM(IV,STUFE) * MEM(IV,STUFE) * C(STUFE+COFF)
            DO 400 L=STUFE+1, BETA, 1
              MEM(IV,L) = 0.0
 400      CONTINUE
          LAST = LAST + AKTVZW
        ENDIF
C
        GOTO 100
```

```
C
 600  CONTINUE
C
C JETZT WIRD DAS AKTUELLE MINIMUM GESUCHT
C UND AN DIE ERSTE STELLE GESCHRIEBEN
C
      L = 1
      LIMIT = CS(1,1)
      DO 700 IV=2, LAST, 1
        IF (CS(IV,1) .LT. LIMIT) THEN
          LIMIT = CS(IV,1)
          L = IV
        ENDIF
 700  CONTINUE
      IF (L .GT. 1) THEN
        CS(1,1) = CS(L,1)
        DO 710 I=1, BETA, 1
          MEM(1,I) = MEM(L,I)
 710    CONTINUE
      ENDIF
C
C JETZT UEBERPRUEFEN WIR, OB NOCH DATEIEN AUSSTEHEN
C
 800  IF (FILENO .GT. 0) THEN
        CALL ENUMRD(MEM,MEMTOP,BETA,STUFE,CS,FILENO,C,J,LAST,ENUMTF)
        FILENO = FILENO - 1
        GELESEN = .TRUE.
        GOTO 100
      ENDIF
C
C JETZT NOCH DIE KOEFFIZIENTEN BEREITSTELLEN
C
      DO 810 I=2, BETA, 1
        MEM(I,1) = MEM(1,I)
 810  CONTINUE
C
      CALL CLOCKV(T(3),T(4),1,0)
C
      ENMINF(1) = ENMINF(1) + (T(4)-T(2))
      ENMINF(2) = ENMINF(2) + (T(3)-T(1))
      ENMINF(3) = ENMINF(3) + 1
C
      ENUMOS = CS(1,1)
C
      RETURN
      END
C
```

```
C
      SUBROUTINE ENUMWR(MEM,MEMTOP,BETA,STUFE,CS,FILENO,C,J,
     +                                    LAST,IANF,IEND,ETF)
C
      IMPLICIT NONE
      INTEGER MEMTOP, BETA, J, STUFE, FILENO, LAST, IEND, IANF
      INTEGER MEM(1:MEMTOP, 1:BETA), IANZ
      CHARACTER*(*) ETF
      DOUBLE PRECISION CS(1:MEMTOP,1:3), C(1:J+BETA-1)
C
      INTEGER COFF, I, I1, L, IV
      DOUBLE PRECISION TMP
      CHARACTER*1 HZE(1:3)
      CHARACTER*60 FILENM
      CHARACTER*10 ZIFFER
C
      IANZ = IEND + 1 - IANF
      PRINT *,'ENUMWR: STUFE= ',STUFE,' LAST= ',LAST,' IANZ= ',IANZ
      COFF = J-1
      ZIFFER = '0123456789'
      I1 = FILENO
      DO 10 I = 1, 3, 1
        L = I1 - (I1 / 10) * 10 + 1
        HZE(I) = ZIFFER(L:L)
        I1 = I1 / 10
  10  CONTINUE
      FILENM = ETF // HZE(3) // HZE(2) // HZE(1)
C
      OPEN(UNIT = 2, FILE=FILENM, FORM='UNFORMATTED')
C
      WRITE (2) STUFE,LAST,IANZ
      DO 100 I = STUFE+1, BETA, 1
        WRITE (2)(MEM(L,I),L=2,LAST,1)
 100  CONTINUE
C
      DO 200 I=IANF, IEND, 1
        IF ( ((I/2)*2) .EQ. I) THEN
          AW = (I-1) * 0.5D0
        ELSE
          AW = I * (-0.5D0)
        ENDIF
        DO 220 IV = 2, LAST, 1
          MEM(IV, STUFE) = NINT(- AW - CS(IV,2) )
 220    CONTINUE
        WRITE (2)(MEM(IV,STUFE),IV=2,LAST,1)
        DO 210 L=2, LAST, 1
          CS(L,3) = CS(L,1)+(CS(L,2)+MEM(L,STUFE))**2*C(COFF+STUFE)
```

```
 210    CONTINUE
        WRITE (2)(CS(IV,3),IV=2,LAST,1)
 200  CONTINUE
C
      CLOSE(2)
C
      RETURN
      END
C
      SUBROUTINE ENUMRD(MEM,MEMTOP,BETA,STUFE,CS,FILENO,C,J,LAST,ETF)
C
      IMPLICIT NONE
      INTEGER MEMTOP, BETA, J, STUFE, FILENO, LAST
      INTEGER MEM(1:MEMTOP, 1:BETA), IANZ
      CHARACTER*(*) ETF
      DOUBLE PRECISION CS(1:MEMTOP,1:2), C(1:J+BETA-1)
C
      INTEGER COFF, I, L, I1
      CHARACTER*1 HZE(1:3)
      CHARACTER*60 FILENM
      CHARACTER*10 ZIFFER
C
      COFF = J-1
      ZIFFER = '0123456789'
      I1=FILENO
      DO 10 I = 1, 3, 1
        L = I1 - (I1 / 10) * 10 + 1
        HZE(I) = ZIFFER(L:L)
        I1 = I1 / 10
  10  CONTINUE
      FILENM = ETF // HZE(3) // HZE(2) // HZE(1)
C
      OPEN(UNIT = 2, FILE=FILENM, FORM='UNFORMATTED')
C
      READ (2) STUFE,LAST,IANZ
      DO 100 I = STUFE+1, BETA, 1
        READ (2)(MEM(L,I),L=2,LAST,1)
 100  CONTINUE
C
C KOPIEREN DER KOEFFIZIENTEN
      DO 150 I=2, IANZ, 1
        DO 150 L=STUFE+1, BETA, 1
          DO 150 IV =2, LAST, 1
            MEM((I-1)*(LAST-1)+IV,L) = MEM(IV,L)
 150  CONTINUE
C
C LESEN DER U_T UND CS
```

```
C
      DO 200 I=1, IANZ, 1
        I1 = (I-1)*(LAST-1)
        READ (2)(MEM(I1+L,STUFE),L=2, LAST, 1)
        READ (2)(CS(I1+L,1),L=2, LAST, 1)
 200  CONTINUE
C
      CLOSE(2,STATUS='DELETE')
C
      LAST = (LAST - 1) * IANZ + 1
C
      RETURN
      END
```

8.6.3 enuminfo.f

```
      SUBROUTINE SETENU(INFO)
C
      IMPLICIT NONE
      INTEGER INFO(1:3)
C     INFO(1) = TICKS
C     INFO(2) = V-TICKS
C     INFO(3) = ANZAHL AUFRUFE
C
      INFO(1) = 0
      INFO(2) = 0
      INFO(3) = 0
C
      RETURN
      END
```

8.7 Schnittenumeration

8.7.1 schenum.f

```
      DOUBLE PRECISION FUNCTION ENUMO7(BETA, C, M, MUE, MEMTOP, MEM,
     +                     VZW, GA, GRENZE, LSG, VON, ZEIT, ENMINF)
C
      IMPLICIT LOGICAL (A-Z)
      INTEGER MEMTOP, BETA, M, VON, ZEIT
      INTEGER MEM(1:MEMTOP), ENMINF(1:3)
      DOUBLE PRECISION C(1:M),MUE(1:M,1:M), GRENZE
      INTEGER VZW(1:BETA), GA(1:BETA)
      INTEGER LSG(1:BETA)
```

```
C
      DOUBLE PRECISION EN1MO7
C
      DOUBLE PRECISION ERG
      INTEGER GRZEIL, GRSPAL
      INTEGER CSOFF, INDOFF, SWIOFF
      INTEGER AVOFF, MEMOFF, ZEIL
      INTEGER I, J, T(1:4)
C
      CALL CLOCKV(T(1),T(2),1,0)
C
      ZEIL = (BETA + 3) / 4
      GRSPAL = ZEIL + 6
      GRZEIL = MEMTOP / GRSPAL
      CSOFF = 1
      AVOFF = 4 * GRZEIL + CSOFF
      INDOFF = AVOFF + GRZEIL
      MEMOFF = INDOFF + GRZEIL
      SWIOFF = MEMOFF
      GRZEIL = GRZEIL - 1
C
      J = MAX(VON-1,1)
 100    J = J + 1
        I = BETA + 1 - J
        ERG = EN1MO7(J, C, M, MUE, GRZEIL, ZEIL, MEM(MEMOFF), VZW(I),
     +       MEM, MEM(AVOFF), GA, MEM(INDOFF), MEM(SWIOFF), GRENZE)
        CALL CLOCKV(T(3),T(4),1,0)
      IF ((J .LT. BETA) .AND. (DABS(ERG-GRENZE) .GT. 0.5) .AND.
     +                    ((T(4)-T(2)) .LT. ZEIT)) GOTO 100
C
      VON = J + 1
      ENUMO7 = ERG
      DO 200 I=1, J, 1
        LSG(I) = MEM(MEMOFF-1+I)
 200  CONTINUE
      DO 210 I=J+1, BETA, 1
        LSG(I) = 0
 210  CONTINUE
C
      CALL CLOCKV(T(3),T(4),1,0)
      ENMINF(1) = ENMINF(1) + (T(4)-T(2))
      ENMINF(2) = ENMINF(2) + (T(3)-T(1))
      ENMINF(3) = ENMINF(3) + 1
      RETURN
C
      END
C
```

```
      DOUBLE PRECISION FUNCTION EN1M07(BETA, C, LDMUE, MUE, LDMEM, SP,
     +                      MEM, VZW, CS, AV, GA, IND, SWAPI, GRENZE)
C
      IMPLICIT LOGICAL (A-Z)
      INTEGER BETA, LDMEM, LDMUE, SP
      DOUBLE PRECISION GRENZE
      DOUBLE PRECISION C(1:BETA)
      DOUBLE PRECISION MUE(1:LDMUE,1:LDMUE)
      DOUBLE PRECISION CS(0:LDMEM, 1:2)
      INTEGER MEM(0:LDMEM,1:SP)
      INTEGER GA(1:BETA)
      INTEGER SWAPI(1:LDMEM)
      INTEGER IND(1:LDMEM)
      INTEGER VZW(1:BETA)
      INTEGER AV(0:LDMEM)
C
      INTEGER IV, LAST, STUFE, I, L, SOLL, INBLO1
      INTEGER AKTVZW, ANZ, I1, BLOCK, INBLO, BLOCK1
      INTEGER ZEILEN, SPALTE, SMOD(1:8)
      INTEGER M1, M2, M3, M4, S0, IOFF
      DOUBLE PRECISION AW, LIMIT, H, CSMAX, CSMIN
      DOUBLE PRECISION OFFSET(1:20)
C
      SAVE SMOD, IOFF, OFFSET
      DATA IOFF /127/
      DATA OFFSET /0 , 1, -1, 2, -2, 3, -3, 4, -4, 5, -5, 6, -6,
     +                                7, -7, 8, -8, 9, -9, 10 /
      DATA SMOD /1, 7, 47, 139, 419, 1259, 3571, 9181/
C
C INITIALISIERUNG
C
      ANZ = LDMEM
      CS(0,1) = GRENZE+2.0
      STUFE = BETA
      LAST = 0
      GOTO 399
C
C DIE SCHLEIFE UEBER DIE STUFEN
C
 100  STUFE = STUFE - 1
      IF (STUFE .LE. 0) GOTO 600
C
      BLOCK = SP - (BETA - STUFE) / 4
      INBLO = MOD(BETA - STUFE , 4)
      INBLO1 = MOD(BETA - 1 - STUFE , 4)
      BLOCK1 = SP - (BETA - 1 - STUFE) / 4
C
```

```
C AUSWERTEN DER VERZWEIGUNGEN
C
      LIMIT = CS(0,1)
      DO 120 IV=1, LAST, 1
        IF (AV(IV) .GT. GA(STUFE)) THEN
          CS(IV,1) = 2*LIMIT
        ENDIF
 120  CONTINUE
C
C KOMPRIMIERUNG DER DATEN
C
*VOCL LOOP,VI(IV),NOVREC(MEM)
      DO 130 I=BLOCK1, SP, 1
        L = 0
        DO 130 IV=1, LAST, 1
          IF (CS(IV,1) .LE. LIMIT) THEN
            L = L + 1
            MEM(L,I) = MEM(IV,I)
          ENDIF
 130  CONTINUE
      L = 0
      CSMIN = LIMIT
*VOCL LOOP,VI(IV),NOVREC(CS,AV)
      DO 140 IV=1, LAST, 1
        IF (CS(IV,1) .LE. LIMIT) THEN
          L = L + 1
          CS(L,1) = CS(IV,1)
          AV(L) = AV(IV)
          CSMIN = MIN(CSMIN, CS(L,1))
        ENDIF
 140  CONTINUE
      LAST = L
      IF (LAST .LT. 1) GOTO 600
C
C BERECHNUNG DER AKTUELLEN VERZWEIGUNGSORDNUNG
C
      AW = SQRT((CS(0,1)-CSMIN)/C(STUFE))
      AKTVZW = INT(AW) + 1
      IF ((AW .LT. 0.5D0) .OR. (STUFE .EQ. 1)) AKTVZW=0
      AKTVZW = MIN(AKTVZW, VZW(STUFE))
C
      IF (2 * AKTVZW * LAST .GE. ANZ) THEN
        IF(AKTVZW .GT. 1) THEN
          CSMAX = CS(1,1)
          SOLL = MIN( ANZ / (2 * AKTVZW), LAST)
          DO 141 IV=1, SOLL, 1
            CSMAX = MAX(CSMAX, CS(IV,1))
```

```
 141      CONTINUE
          LIMIT = CSMAX
C
C 2.  KOMPRIMIERUNG DER DATEN, DAMIT WENIGER ZU SORTIEREN IST
C
*VOCL LOOP,VI(IV),NOVREC(MEM)
          DO 143 I=BLOCK1, SP, 1
            L = 0
            DO 143 IV=1, LAST, 1
              IF (CS(IV,1) .LE. LIMIT) THEN
                L = L + 1
                MEM(L,I) = MEM(IV,I)
              ENDIF
 143      CONTINUE
          L = 0
*VOCL LOOP,VI(IV),NOVREC(CS,AV)
          DO 145 IV=1, LAST, 1
            IF (CS(IV,1) .LE. LIMIT) THEN
              L = L + 1
              CS(L,1) = CS(IV,1)
              AV(L) = AV(IV)
            ENDIF
 145      CONTINUE
          LAST = L
        ENDIF
C
C SORTIEREN DER DATEN
C
        DO 150 IV=1, LAST, 1
          IND(IV) = IV
 150    CONTINUE
        DO 160 I=8, 1, -1
          ZEILEN = SMOD(I)
          SPALTE = LAST / ZEILEN
          IF ((((I/2)*2) .NE. I) .AND. (I .GT. 4)) THEN
            L = LAST + 1 - ZEILEN * SPALTE
            CALL PSORT1(CS(L,1), IND(L), SPALTE, ZEILEN)
          ELSE
            CALL PSORT1(CS(1,1), IND(1), SPALTE, ZEILEN)
          ENDIF
 160    CONTINUE
C
C ABSCHNEIDEN DER GROESSEREN WERTE
C
        LAST = MIN( ANZ / (2 * AKTVZW), LAST)
C
        DO 180 I=BLOCK1, SP, 1
```

```
          DO 182 IV=1, LAST, 1
            SWAPI(IV) = MEM(IND(IV),I)
 182      CONTINUE
          DO 184 IV=1, LAST, 1
            MEM(IV,I) = SWAPI(IV)
 184      CONTINUE
 180    CONTINUE
        DO 194 IV=1, LAST, 1
          SWAPI(IV) = AV(IND(IV))
 194    CONTINUE
        DO 196 IV=1, LAST, 1
          AV(IV) = SWAPI(IV)
 196    CONTINUE
      ENDIF
C
C BERECHNUNG VON Y_T
C
        SO = BETA - (SP-BLOCK1) * 4
*VOCL LOOP,TEMP(M1,M2,M3,M4)
        DO 240 IV=1, LAST, 1
          M1 = IAND(MEM(IV,BLOCK1),255) - IOFF
          M2 = ISHFT(MEM(IV,BLOCK1),-8)
          M3 = ISHFT(M2,-8)
          M4 = ISHFT(M3,-8)
          CS(IV,2) = M1 * MUE(STUFE,SO)
          IF (INBLO1 .GT. 0) THEN
            M2 = IAND(M2,255) - IOFF
            CS(IV,2)=CS(IV,2) + M2 * MUE(STUFE,SO-1)
          ENDIF
          IF (INBLO1 .GT. 1) THEN
            M3 = IAND(M3,255) - IOFF
            CS(IV,2)=CS(IV,2) + M3 * MUE(STUFE,SO-2)
          ENDIF
          IF (INBLO1 .GT. 2) THEN
            M4 = IAND(M4,255) - IOFF
            CS(IV,2)=CS(IV,2) + M4 * MUE(STUFE,SO-3)
          ENDIF
 240    CONTINUE
        SO = BETA
        DO 250 I=SP, BLOCK1+1, -1
*VOCL LOOP,TEMP(M1,M2,M3,M4)
          DO 255 IV=1, LAST, 1
            M1 = IAND(MEM(IV,I),255) - IOFF
            M2 = ISHFT(MEM(IV,I),-8)
            M3 = ISHFT(M2,-8)
            M4 = ISHFT(M3,-8)
            M2 = IAND(M2,255) - IOFF
```

```
            M3 = IAND(M3,255) - IOFF
            M4 = IAND(M4,255) - IOFF
            CS(IV,2) = CS(IV,2) + M1 * MUE(STUFE, SO)
     +                          + M2 * MUE(STUFE, SO-1)
     +                          + M3 * MUE(STUFE, SO-2)
     +                          + M4 * MUE(STUFE, SO-3)
 255      CONTINUE
          SO = SO - 4
 250    CONTINUE
C
C FALLS WIR GERADE DAS OPTIMUM NEHMEN
C
        IF (AKTVZW .EQ. 0) THEN
          IF(INBLO .EQ. 0) THEN
*VOCL LOOP,TEMP(M1)
            DO 260 IV=1, LAST, 1
              M1 = NINT(-CS(IV,2))
              CS(IV,1) = CS(IV,1) + (M1+CS(IV,2))**2 * C(STUFE)
              MEM(IV,BLOCK) = IOFF + M1
 260        CONTINUE
          ELSE
*VOCL LOOP,TEMP(M1,M2)
            DO 261 IV=1, LAST, 1
              M1 = NINT(-CS(IV,2))
              CS(IV,1) = CS(IV,1) + (M1+CS(IV,2))**2 * C(STUFE)
              M1 = ISHFT(M1+IOFF,8*INBLO)
              MEM(IV,BLOCK) = MEM(IV,BLOCK) + M1
 261        CONTINUE
          ENDIF
        ELSE
C
C ALLE VEKTOREN PASSEN IN DEN HAUPTSPEICHER
C
          DO 300 I=2, 2*AKTVZW, 1
            I1 = LAST * (I - 1)
            DO 300 L=BLOCK+1, SP, 1
C KOPIEREN DER KOEFFIZIENTEN
*VOCL LOOP,VI(IV),NOVREC(MEM)
              DO 300 IV=1, LAST, 1
                MEM(I1+IV,L) = MEM(IV,L)
 300      CONTINUE
          DO 325 I=2*AKTVZW, 1, -1
            I1 = LAST * (I - 1)
            AW = OFFSET(I) - 0.5D0
            IF (INBLO .EQ. 0) THEN
C BERECHNUNG DER NEUEN U_T
*VOCL LOOP,VI(IV),NOVREC(AV,CS,MEM),TEMP(H,M1)
```

```
              DO 320 IV=1, LAST, 1
                M1 = NINT(-CS(IV,2)-AW)
C BERECHNUNG DER GROESSEN, DIE DEN BAUM STUTZEN
                H = DABS(M1 + CS(IV,2))
                MEM(IV+I1,BLOCK) = M1 + IOFF
*VOCL STMT,IF(90)
                IF ( H .GT. 0.5D0 ) THEN
                  AV(I1+IV) = AV(IV) + 1
                ELSE
                  AV(I1+IV) = AV(IV)
                ENDIF
C BERECHNUNG DER NEUEN C
                CS(I1+IV,1) = CS(IV,1) + H**2 * C(STUFE)
 320          CONTINUE
            ELSE
*VOCL LOOP,VI(IV),NOVREC(AV,CS,MEM),TEMP(H,M1)
              DO 321 IV=1, LAST, 1
                M1 = NINT(-CS(IV,2)-AW)
                H = DABS(M1 + CS(IV,2))
                MEM(IV+I1,BLOCK) = MEM(IV,BLOCK)
     +                      + ISHFT( M1+IOFF , 8*INBLO )
*VOCL STMT,IF(90)
                IF ( H .GT. 0.5D0 ) THEN
                  AV(I1+IV) = AV(IV) + 1
                ELSE
                  AV(I1+IV) = AV(IV)
                ENDIF
                CS(I1+IV,1) = CS(IV,1) + H**2 * C(STUFE)
 321          CONTINUE
            ENDIF
 325      CONTINUE
          LAST = LAST * 2 * AKTVZW
        ENDIF
C
C NEUE VEKTOREN DAZUADDIEREN
C
 399    CONTINUE
        IF (BETA .EQ. STUFE) THEN
          AKTVZW = VZW(STUFE)
          DO 400 IV=1, AKTVZW, 1
            AV(IV) = 0
            MEM(IV,SP) = IV + IOFF
            CS(IV,1) = IV * IV * C(STUFE)
 400      CONTINUE
          LAST = AKTVZW
        ENDIF
C
```

```
        GOTO 100
C
 600  CONTINUE
C
C JETZT WIRD DER BESTE WERT GESUCHT
C UND AN DIE ERSTE STELLE GESCHRIEBEN
C
      L = 0
      LIMIT = DABS(CS(0,1)-GRENZE)
      DO 700 IV=1, LAST, 1
        H = DABS(CS(IV,1)-GRENZE)
        IF (H .LT. LIMIT) THEN
          LIMIT = H
          L = IV
        ENDIF
 700  CONTINUE
      IF (L .GT. 0) THEN
        CS(0,1) = CS(L,1)
        DO 710 I=1, SP, 1
          MEM(0,I) = MEM(L,I)
 710    CONTINUE
      ENDIF
C
C JETZT NOCH DIE KOEFFIZIENTEN BEREITSTELLEN
C
*VOCL LOOP,NOVREC(MEM)
      DO 810 I=1, SP, 1
        MEM(I,2) = MEM(0,I)
 810  CONTINUE
      L = BETA
      DO 820 I=SP, 1, -1
        M1 = MEM(I,2)
        DO 830 IV=1, 4, 1
          M2 = IAND(M1, 255)
          M1 = ISHFT(M1, -8)
          IF (L .GT. 0) MEM(L-1,1) = M2 - IOFF
          L = L - 1
 830    CONTINUE
 820  CONTINUE
C
      EN1M07 = CS(0,1)
C
      RETURN
      END
C
C
```

8.7.2 sort.f

```
      SUBROUTINE PSORT1(KEY, IND, SPALTE, ZEILEN)
C
      INTEGER SPALTE, ZEILEN
      DOUBLE PRECISION KEY(1:ZEILEN, 1:SPALTE)
      INTEGER IND(1:ZEILEN, 1:SPALTE)
C
      INTEGER I, J, K, S1, S2
      DOUBLE PRECISION Q
      LOGICAL TAUSCH
C
      IF(ZEILEN .GT. 255) THEN
        S1 = 1
        S2 = SPALTE - 1
 220    CONTINUE
        DO 200 I=S1, S2, 1
*VOCL LOOP,NOVREC(KEY,IND),TEMP(Q,K)
          DO 200 J=1, ZEILEN, 1
            IF(KEY(J,I) .GT. KEY(J,I+1)) THEN
              K = IND(J,I)
              IND(J,I) = IND(J,I+1)
              IND(J,I+1) = K
              Q = KEY(J,I)
              KEY(J,I) = KEY(J,I+1)
              KEY(J,I+1) = Q
            ENDIF
 200    CONTINUE
C
        DO 210 I=S2-1, S1, -1
*VOCL LOOP,NOVREC(KEY,IND),TEMP(Q,K)
          DO 210 J=1, ZEILEN, 1
            IF(KEY(J,I) .GT. KEY(J,I+1)) THEN
              K = IND(J,I)
              IND(J,I) = IND(J,I+1)
              IND(J,I+1) = K
              Q = KEY(J,I)
              KEY(J,I) = KEY(J,I+1)
              KEY(J,I+1) = Q
            ENDIF
 210    CONTINUE
        S1 = S1 + 1
        S2 = S2 - 1
        IF (S1 .LT. S2) GOTO 220
      ELSE
C
 100    TAUSCH = .FALSE.
```

```
C
        DO 300 J=1, ZEILEN, 1
*VOCL LOOP,NOVREC(KEY,IND),TEMP(Q,K),VI(I)
          DO 300 I=1, SPALTE-1, 2
            IF(KEY(J,I) .GT. KEY(J,I+1)) THEN
              K = IND(J,I)
              IND(J,I) = IND(J,I+1)
              IND(J,I+1) = K
              Q = KEY(J,I)
              KEY(J,I) = KEY(J,I+1)
              KEY(J,I+1) = Q
              TAUSCH = .TRUE.
            ENDIF
 300    CONTINUE
C
        DO 310 J=1, ZEILEN, 1
*VOCL LOOP,NOVREC(KEY,IND),TEMP(Q,K),VI(I)
          DO 310 I=2, SPALTE-1, 2
            IF(KEY(J,I) .GT. KEY(J,I+1)) THEN
              K = IND(J,I)
              IND(J,I) = IND(J,I+1)
              IND(J,I+1) = K
              Q = KEY(J,I)
              KEY(J,I) = KEY(J,I+1)
              KEY(J,I+1) = Q
              TAUSCH = .TRUE.
            ENDIF
 310    CONTINUE
C
        IF (SPALTE .LE. 2) RETURN
        IF (TAUSCH) GOTO 100
C
      ENDIF
C
      RETURN
      END
C
```

8.8 Verschiedenes

8.8.1 vioverflow.f

```
C
C MELDUNG EINES OVERFLOWS IN VEKTORINT-ARITHMETIK
C
      SUBROUTINE OVERFL(A)
C
      IMPLICIT NONE
      CHARACTER A*10, B*12
C
      B(1:12) = 'OVERFLOW IN '
      PRINT *,B,A
C
      STOP
C
      END
```

8.8.2 fileio.f

```
C LESEN EINER VEKTORINT-MATRIX DER DIMENSION ANZV * VLEN
C AUS DER DATEI   FILENM    MIT KANALNUMMER   CHAN
C UEBER DEN INDEXVEKTOR  INDEXV
C
      SUBROUTINE RVIMTX(B, IVAZBL, LDB, AS, INDEXV, ANZV, VLEN,
     +                                            FILENM, CHAN)
C
      IMPLICIT NONE
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      INTEGER ANZV, VLEN, CHAN, LDB, AS, INDEXV(1:ANZV)
      INTEGER IVAZBL
      DOUBLE PRECISION B(1:LDB, -2:IVAZBL, 1:AS)
      CHARACTER FILENM*(*)
C
      INTEGER I,J,K,IERR
C
      OPEN(UNIT=CHAN, FILE=FILENM, IOSTAT=IERR)
      IF (IERR .NE. 0) CALL FILERR(FILENM, 'OPEN', IERR)
C
      DO 100 I=1, ANZV, 1
        READ(2,*,IOSTAT=IERR) (B(J,ISIGN,I),J=1,VLEN,1)
```

```
        READ(2,*,IOSTAT=IERR) (B(J,ILEAD,I),J=1,VLEN,1)
        DO 100 K=0, NINT(B(1,ILEAD,I)), 1
          READ(2,*,IOSTAT=IERR) (B(J,K,I),J=1,VLEN,1)
          IF (IERR .NE. 0) CALL FILERR(FILENM, 'READ', IERR)
 100  CONTINUE
C
      CLOSE(CHAN, IOSTAT=IERR)
      IF (IERR .NE. 0) CALL FILERR(FILENM, 'CLOSE', IERR)
C
      DO 200 I=1, ANZV, 1
        INDEXV(I) = I
 200  CONTINUE
C
      RETURN
      END
C
C
C SCHREIBEN EINER VEKTORINT-MATRIX DER DIMENSION ANZV * VLEN
C IN DIE DATEI   FILENM    MIT KANALNUMMER   CHAN
C UEBER DEN INDEXVEKTOR  INDEXV
C
      SUBROUTINE WVIMTX(B, IVAZBL, LDB, AS, INDEXV, ANZV, VLEN,
     +                                          FILENM, CHAN)
C
      IMPLICIT NONE
      INTEGER IVBASI, IVLOWB, ISIGN, ILEAD
      DOUBLE PRECISION IVBHM1
      COMMON /CONST/ IVBASI, IVLOWB, ISIGN, ILEAD, IVBHM1
      INTEGER ANZV, VLEN, CHAN, LDB, AS, INDEXV(1:ANZV)
      INTEGER IVAZBL
      DOUBLE PRECISION B(1:LDB, -2:IVAZBL, 1:AS)
      CHARACTER FILENM*(*)
C
      INTEGER I,J,K,IERR
C
      OPEN(UNIT=CHAN, FILE=FILENM, IOSTAT=IERR)
      IF (IERR .NE. 0) CALL FILERR(FILENM, 'OPEN', IERR)
C
      DO 100 I=1, ANZV, 1
        DO 100 K=ISIGN, NINT(B(1,ILEAD,INDEXV(I))), 1
          WRITE(2,*,IOSTAT=IERR) (NINT(B(J,K,INDEXV(I))),J=1,VLEN,1)
          IF (IERR .NE. 0) CALL FILERR(FILENM, 'WRITE', IERR)
 100  CONTINUE
C
      CLOSE(CHAN, IOSTAT=IERR)
      IF (IERR .NE. 0) CALL FILERR(FILENM, 'CLOSE', IERR)
C
```

```
      RETURN
      END
C
C
C SCHREIBEN EINER FLOAT-MATRIX DER DIMENSION ANZV * VLEN
C IN DIE DATEI   FILENM    MIT KANALNUMMER   CHAN
C UEBER DEN INDEXVEKTOR  INDEXV
C
      SUBROUTINE WFIMTX(BFP, LDB, AS, INDEXV, ANZV, VLEN, FILENM, CHAN)
C
      IMPLICIT NONE
      INTEGER ANZV, LDB, VLEN, CHAN, AS, INDEXV(1:ANZV)
      DOUBLE PRECISION BFP(1:LDB, 1:AS)
      CHARACTER FILENM*(*)
C
      INTEGER I,J,IERR
C
      OPEN(UNIT=CHAN, FILE=FILENM, FORM='UNFORMATTED', IOSTAT=IERR)
      IF (IERR .NE. 0) CALL FILERR(FILENM, 'OPEN', IERR)
C
      DO 100 I=1, ANZV, 1
        WRITE(2,IOSTAT=IERR) (BFP(J,INDEXV(I)),J=1,VLEN,1)
        IF (IERR .NE. 0) CALL FILERR(FILENM, 'WRITE', IERR)
 100  CONTINUE
C
      CLOSE(CHAN, IOSTAT=IERR)
      IF (IERR .NE. 0) CALL FILERR(FILENM, 'CLOSE', IERR)
C
      RETURN
      END
C
C
C SCHREIBEN EINER FLOAT-MATRIX DER DIMENSION ANZV * VLEN
C IN DIE DATEI   FILENM    MIT KANALNUMMER   CHAN
C OHNE INDEXVEKTOR
C
      SUBROUTINE WFLMTX(BFP, LDB, AS, ANZV, VLEN, FILENM, CHAN)
C
      IMPLICIT NONE
      INTEGER ANZV, VLEN, CHAN, LDBFP, AS
      DOUBLE PRECISION BFP(1:LDB, 1:AS)
      CHARACTER FILENM*(*)
C
      INTEGER I,J,IERR
C
      OPEN(UNIT=CHAN, FILE=FILENM, FORM='UNFORMATTED',IOSTAT=IERR)
      IF (IERR .NE. 0) CALL FILERR(FILENM, 'OPEN', IERR)
```

```
C
      DO 100 I=1, ANZV, 1
        WRITE(2,IOSTAT=IERR) (BFP(J,I),J=1,VLEN,1)
        IF (IERR .NE. 0) CALL FILERR(FILENM, 'WRITE', IERR)
 100  CONTINUE
C
      CLOSE(CHAN,IOSTAT=IERR)
      IF (IERR .NE. 0) CALL FILERR(FILENM, 'CLOSE', IERR)
C
      RETURN
      END
C
C
C LESEN EINER FLOAT-MATRIX DER DIMENSION ANZV * VLEN
C AUS DER DATEI   FILENM    MIT KANALNUMMER   CHAN
C OHNE INDEXVEKTOR
C
      SUBROUTINE RFLMTX(BFP, LDB, AS, ANZV, VLEN, FILENM, CHAN)
C
      IMPLICIT NONE
      INTEGER ANZV, VLEN, CHAN, LDBFP, AS
      DOUBLE PRECISION BFP(1:LDB, 1:AS)
      CHARACTER FILENM*(*)
C
      INTEGER I,J,IERR
C
      OPEN(UNIT=CHAN, FILE=FILENM, FORM='UNFORMATTED',IOSTAT=IERR)
      IF (IERR .NE. 0) CALL FILERR(FILENM, 'OPEN', IERR)
C
      DO 100 I=1, ANZV, 1
        READ(2,IOSTAT=IERR) (BFP(J,I),J=1,VLEN,1)
        IF (IERR .NE. 0) CALL FILERR(FILENM, 'READ', IERR)
 100  CONTINUE
C
      CLOSE(CHAN,IOSTAT=IERR)
      IF (IERR .NE. 0) CALL FILERR(FILENM, 'CLOSE', IERR)
C
      RETURN
      END
C
C
C LESEN EINER INT-MATRIX DER DIMENSION ANZV * VLEN
C AUS DER DATEI   FILENM    MIT KANALNUMMER   CHAN
C OHNE INDEXVEKTOR
C
      SUBROUTINE RINTMX(V, LDV, AS, ANZV, VLEN, FILENM, CHAN)
C
```

```
      IMPLICIT NONE
      INTEGER ANZV, VLEN, CHAN, LDV, AS
      INTEGER V(1:LDV, 1:AS)
      CHARACTER FILENM*(*)
C
      INTEGER I,J,IERR
C
      OPEN(UNIT=CHAN, FILE=FILENM, FORM='UNFORMATTED',IOSTAT=IERR)
      IF (IERR .NE. 0) CALL FILERR(FILENM, 'OPEN', IERR)
C
      DO 100 I=1, ANZV, 1
        READ(2,IOSTAT=IERR) (V(J,I),J=1,VLEN,1)
        IF (IERR .NE. 0) CALL FILERR(FILENM, 'READ', IERR)
 100  CONTINUE
C
      CLOSE(CHAN,IOSTAT=IERR)
      IF (IERR .NE. 0) CALL FILERR(FILENM, 'CLOSE', IERR)
C
      RETURN
      END
C
C
C SCHREIBEN EINER INT-MATRIX DER DIMENSION ANZV * VLEN
C IN DIE DATEI   FILENM    MIT KANALNUMMER   CHAN
C OHNE INDEXVEKTOR
C
      SUBROUTINE WINTMX(V, LDV, AS, ANZV, VLEN, FILENM, CHAN)
C
      IMPLICIT NONE
      INTEGER ANZV, VLEN, CHAN, LDV, AS
      INTEGER V(1:LDV, 1:AS)
      CHARACTER FILENM*(*)
C
      INTEGER I,J,IERR
C
      OPEN(UNIT=CHAN, FILE=FILENM, FORM='UNFORMATTED',IOSTAT=IERR)
      IF (IERR .NE. 0) CALL FILERR(FILENM, 'OPEN', IERR)
C
      DO 100 I=1, ANZV, 1
        WRITE(2,IOSTAT=IERR) (V(J,I),J=1,VLEN,1)
        IF (IERR .NE. 0) CALL FILERR(FILENM, 'WRITE', IERR)
 100  CONTINUE
C
      CLOSE(CHAN,IOSTAT=IERR)
      IF (IERR .NE. 0) CALL FILERR(FILENM, 'CLOSE', IERR)
C
      RETURN
```

```
      END
C
      SUBROUTINE FILERR(FN, OP, IERR)
C
      IMPLICIT NONE
      CHARACTER FN*(*),OP*(*)
      INTEGER IERR
C
      PRINT *,'Fehler bei Operation ',OP,' auf Datei ',FN
      PRINT *,'Fehlercode= ',IERR
      PRINT *,'Programmabbruch '
C
      STOP
      END
```

8.8.3 tickstr.f

```
      SUBROUTINE TCKSTR(TICKS, STRING)
C
      IMPLICIT NONE
      INTEGER TICKS
      CHARACTER STRING*20
C
      INTEGER T(1:4),HLP,I
      CHARACTER*10 ZIFFER
      CHARACTER*1  STELLE
C
      ZIFFER = '0123456789'
      T(1) = MOD(TICKS, 1000)
      T(2) = TICKS / 1000
      T(3) = T(2) / 60
      T(2) = MOD(T(2), 60)
      T(4) = T(3) / 60
      T(3) = MOD(T(3),60)
C
      DO 100 J=1, 4, 1
        DO 120 I=1, 3, 1
          IF ((J .EQ. 1) .OR. (J .EQ. 4) .OR. (I .LT. 3)) THEN
            HLP = MOD(T(J),10)+1
            STELLE = ZIFFER(HLP:HLP)
            STRING = STELLE // STRING
            T(J) = T(J) / 10
          ENDIF
 120    CONTINUE
```

```
          IF (J .EQ. 1) THEN
            STRING = ',' // STRING
          ELSEIF (J .NE. 4) THEN
            STRING = ':'  // STRING
          ENDIF
  100   CONTINUE
C
        RETURN
        END
```

9. Danksagungen

Für die Betreuung meiner Diplomarbeit möchte ich mich bei Herrn Prof. Dr. C. P. Schnorr bedanken. Ein weiterer Dank gilt Harald Ritter für das Korrekturlesen dieser Arbeit und die zahlreichen Diskussionen und Anregungen. Ferner bedanke ich mich bei dem Hochschulrechenzentrum der Johann-Wolfgang-Goethe-Universität Frankfurt für die Benutzung des dort installierten Vektorrechners und der zahlreichen IBM-Workstations. Insbesondere Frau Shurawel, Herrn Sternecker und Herrn Harth verdanke ich, daß ich die für diese Arbeit nötigen Programmläufe in dieser Art durchführen konnte. Schließlich möchte ich mich noch bei Herrn Seybold von der Firma Siemens-Nixdorf-Informationssysteme AG bedanken, der als „letzte Instanz“ technische Probleme beseitigte.

A. Literaturverzeichnis

[B83] E.F. Brickell, Solving low density knapsacks, Advances in Cryptology, Proceedings of Crypto '83 (1984) pp 25–37

[BH83] A. Bode, W. Händler, Rechnerarchitektur II — Strukturen, Springer (1983)

[C71] J.W.S. Cassels, An introduction to the geometry of numbers, Springer-Verlag (1971)

[CJLOSS92] M.J.Costner, A.Joux, B.A. LaMacchia, A.M. Odlyzko, C.P. Schnorr, J.Stern, An improved low-density subset sum algorithm, computational complexity 2 (1992) pp. 111–128

[CR84] B. Chor, R.L. Rivest, A knapsack-type public key cryptosystem based on arithmetic in finite fields, Advances in Cryptology, Proc. Crypto 84, Springer-Verlag (1985) pp. 54–65

[CS88] H.H. Conway & H.J.A. Sloane, Sphere Packings, Lattices and Groups, Springer-Verlag (1988)

[E91] M. Euchner, Praktische Algorithmen zur Gitterbasenreduktion und Faktorisierung, Diplomarbeit Universität Frankfurt (1991)

[F] Fujitsu Ltd., VP 2000 — Theory of Operation, Interne Referenznummer 22FH3024E-01

[F72] M.J. Flynn, Some Computer Organisations and Their Effectiveness, IEEE Transactions on Computers (1972) pp. 948–960

[F86] A.M. Frieze, On the Lagarias-Odlyzko algorithm for the subset sum problem, SIAM J. Comput. 15 (1986) pp. 536–539

[FK89] M.L. Furst, R. Kannan, Succinct certificates for allmost all subset sum problems, SIAM J. Comput. 18 (1989) pp. 550–558

[G93] W.K. Giloi, Rechnerarchitektur 2. Auflage, Springer (1993)

[H94] H.H. Hörner, Verbesserte Gitterbasenreduktion; getestet am Chor-Rivest Kryptosystem und allgemeinen Rucksack-Problemen, Diplomarbeit Universität Frankfurt (1994)

[HB85] K. Hwang, F.A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill (1985)

[HJ81] R.W. Hockney, C.R. Jesshope, Parallel Computers — Architecture, Programming and Algorithms, Adam Hilger (1981)

[JS91] A. Joux, J. Stern, Improving the critical density of the Lagarias-Odlyzko attack against subset sum problems, Proceedings of Fundamentals of Computation Theory '91 Springer LNCS 529 (1991) pp 258–264

[K83] R. Kannan, Improved algorithms for integer programming and related lattice problems, Proceedings 15^{th} Symp. Theory of Comp. (1983) pp. 193–206

[LLL82] A.K. Lenstra, H.W. Lenstra, Jr., L. Lovász, Factoring polynomials with rational coefficients, Math. Ann. 261 (1982) pp 515–534

[LLS90] J.C. Lagarias, H.W. Lenstra, Jr., C.P. Schnorr, Korkin-Zolotarev bases and successive minima of a lattice and its reciprocal lattice, Combinatorica 10 (1990) pp. 333–348

[LO85] J.C. Lagarias, A.M. Odlyzko, Solving low-denity subset sum problems, J. Assoc. Comp. Mach. 32 (1985) pp 229–246

[R87] G. Regensprung, Hochleistungsrechner-Architekturprinzipien, McGraw-Hill (1987)

[RK88] S. Radziszowski, D. Kreher, Solving subset sum problems with the l^3 algorithm, J. Combin. Math. Combin. Comput. 3 (1988) pp. 49–63

[S87] C.P. Schnorr, A hierarchy of polynomial time lattice basis reduction algorithms, Theoret. Comput. Sci. 53 (1987) pp 201–224

[S88] C.P. Schnorr, A more efficient algorithm for lattice basis reduction, J. Algorithms 9 (1988) pp 47–62

[S91] C.P. Schnorr, Gittertheorie und ganzzahlige Optimierung, Skript zur Vorlesung im Sommersemester 1991 an der Universität Frankfurt

[S92] C.P. Schnorr, Gitter, Packungen und Kodes, Mitschrift zur Vorlesung im Wintersemester 1991/92 an der Universität Frankfurt

[S93] C.P. Schnorr, Factoring Integers and Computing Discrete Logarithms via Diophantine Approximation, AMS DIMACS Series in Discr. Math. and Theoret. Comput. Sci. 13 (1993) pp 171-182

[SE91] C.P. Schnorr, M. Euchner, Lattice basis reduction: improved algorithms and solving subset sum problems, Proceedings of Fundamentals of Computation Theory '91 Springer LNCS 529 (1991) pp 68–85

[SH95] C.P. Schnorr, H.H. Hörner, Attacking the Chor-Rivest Cryptosystem by Improved Lattice Reduction, erscheint in Springer LNCS

[W91] K. Waldschmidt, Technische Informatik — Parallele Rechnerarchitekturen, Skript zur Vorlesung im Sommersemester 1991 an der Universität Frankfurt