UNIVERSITÄT PASSAU

Diplomarbeit

Analyse und Optimierung des Kommunikationsaufwands in automatisch parallelisierten Programmen

Verfasser: Sven Anders

Betreuer: Dr. Martin Griebl

Passau, 22. Mai 2000

Aufgabensteller

Prof. C. Lengauer, Ph. D.

Lehrstuhl für Programmierung

Meiner Mutter

Diese Arbeit wurde möglich durch die theoretische und moralische Unterstützung mehrerer Personen.
In erster Stelle möchte ich Dr. Martin Griebl danken, welcher für meine Fragen immer ein offenes Ohr hatte, mich mit Korrekturen, Anregungen und vielen motivierenden Worten unterstützt hat und sich auch mal Zeit nahm, wenn er eigentlich keine hatte.
Prof. Christian Lengauer, welcher diese Arbeit korrekturgelesen hat und mich mit Korrekturen und Anregungen versorgt hat.
Peter Faber, Matthias Berberich, Michael Schön, Oliver Nyderle und Marcus Junker für Ihre Korrekturen, Anregungen und Hinweise.
Ganz besonders meiner Freundin Manuela und meiner Familie, welche mir das Studium erst ermöglicht hat. Ihnen danke ich für Ihre große Geduld und, daß sie mich nie unter Druck gesetzt haben.

Zusammenfassung

Parallelisierung ist eine wichtige Möglichkeit bestehende Programme zu beschleunigen. Zur Erleichterung wurden in den letzten Jahren Methoden und Programme zur automatischen Parallelisierung entwickelt. Ein solches Programm ist *LooPo*, ein Schleifenparallelisierer, welcher auf den Methoden des Polytop-Modells aufbaut.

Diese Arbeit analysiert exemplarisch anhand des durch LooPo erzeugten Codes die Probleme bei der automatischen Parallelisierung und stellt zwei Verfahren zur Reduzierung der auf Rechnerarchitekturen mit verteiltem Speicher erzeugten, extrem teuren Kommunikationen vor.

Das erste Verfahren dient dabei auschließlich zur Reduzierung der Kommunikationen und läßt den Programmablauf des parallelen Programms unverändert. Beim zweiten Verfahren hingegen führt das Zusammenfassen der Zeit nicht nur zu einer weiteren Reduzierung der Kommunikation und Vergröberung der Granularität, sondern auch zu Veränderungen im Ablauf. Diese Veränderungen und deren Auswirkungen auf den erzeugten Code werden ebenfalls näher betrachtet.

Inhaltsverzeichnis

Kapitel 1

Einführung

Parallelisierung ist eine wichtige Möglichkeit, um bestehende Programme zu beschleunigen. Bisher wurde in Fällen, in denen Parallelität benötigt wurde, dies oft in Handarbeit gemacht. Erleichterungen entstanden erst in den letzten Jahren, als Methoden und Programme zur automatischen Parallelisierung entwickelt wurden.

Eines dieser Programme ist *LooPo* [1], welches sich der Methoden des Polytop-Modells [2] bedient. Dabei werden insbesondere Schleifensätze betrachtet, da diese ein großes Potential an möglicher Parallelisierung bieten. Dieses Potential entsteht durch die wiederholte Ausführung der darin enthaltenen Anweisungen. Schleifen haben weiterhin den Vorteil, daß deren Parallelisierung häufig statisch durchgeführt werden kann.

LooPo führt dafür zunächst eine Abhängigkeitsanalyse durch, welche alle bei der Umformung zu berücksichtigenden Abhängigkeiten berechnet. Anschließend wird eine Ausführungshalbordnung der einzelnen Instanzen berechnet, der sogenannte Schedule. Die Instanzen werden dann durch die Berechnung der Allokation auf die Prozessoren des zugrundeliegenden Mehrprozessorsystems verteilt.

Das Polytop-Modell arbeitet dabei auf einer abstrakten Architektur mit gemeinsamen Speicher. In der Realität existieren solche Rechnerarchitekturen jedoch aufgrund der Speichermanagmentproblematik bei Zugriff auf den gemeinsamen Bus nur mit einer geringen Anzahl von Prozessoren. Damit sind diese Architekturen nicht beliebig skalierbar und schließen massive Parallelität aus.
Da bei Architekturen mit gemeinsamen Speicher und einer sehr großen Anzahl an Prozessoren dieser nur virtuell gemeinsam ist (d.h. dies ist über eine Softwareschicht realisiert), versucht man das Modell aus Geschwindigkeitsgründen auch auf Architekturen mit verteiltem Speicher einzusetzen.
Auf Architekturen mit verteiltem Speicher erzeugen die berechneten Abhängigkeiten Kommunikation. Diese sind jedoch im Vergleich zu normalen Speicherzugriffen und Berechnungen sehr teuer und stellen daher ein ernstzunehmendes Hindernis für die gewünschte Beschleunigung dar.

Diese Arbeit beschäftigt sich daher mit Möglichkeiten, die Kosten für diese unvermeidlichen Kommunikationen zu senken. Damit soll das Polytop-Modell auch auf Architekturen mit verteiltem Speicher besser nutzbar gemacht werden.

Wir analysieren dazu die Kosten der Kommunikationen, stellen diese in Relation zu den weiteren anfallenden Kosten und zeigen Möglichkeiten zur Senkung beider.
In zwei Schritten werden die Kosten der Kommunikation gesenkt und die Granularität vergröbert. Zuerst erweitern wir die bekannten Techniken zur Partionierung um die Bündelung von Kommunikationen. Anschließend wird die Partionierung auf die Zeit erweitert, wobei wir hier untersuchen, wie und unter welchen Voraussetzungen mehrere Zeitschritte zusammengefaßt werden können. Am Ende dieser Arbeit stellen wir einen entsprechenden Algorithmus vor und belegen die Laufzeitabschätzungen durch Laufzeitmessungen.

Kapitel 2

Grundlagen

In diesem Abschnitt sollen einige im folgenden verwendete Begriffe und Notationen eingeführt werden. Außerdem sollen die nötigen Grundlagen im Zusammenhang mit dem Polytop-Modell und LooPo gelegt werden.

2.1 Schleifen

Die im folgenden vorgestellten Verfahren lassen sich sowohl auf FOR-Schleifen als auch WHILE-Schleifen anwenden. Da selbst das in LooPo implementierte erweiterte Modell, welches auch WHILE-Schleifen behandeln kann [3], die WHILE-Schleifen auf FOR-Schleifen zurückführt, können wir uns zur Definition aller im folgenden wichtigen Begriffe auf diese beschränken. Wir nehmen Schleifen mit der folgenden Struktur an:

```
for i := l to u step w

 S1:  A1[x1, ..., xm] := ...
 :
 Sn:  An[x1, ..., xm] := ...

endfor
```

Eine *Schleife* besteht aus dem Schlüsselwort `FOR`, welches von einen *Schleifenindex (i)* sowie der *Untergrenze (l)* und der *Obergrenze (u)* gefolgt wird. Diese Grenzen definieren den *Indexraum* einer Schleife. Die Angabe einer *Schrittweite (w)* ist möglich. Ist diese nicht angegeben wird eine Schrittweite von 1 angenommen.
S_1 bis S_n sind die *Statements* der Schleife. Interessant für die Analyse der Abhängigkeiten sind dabei die Zuweisungen. Hierfür werden Zuweisungen innerhalb von *Arrays* ($A_x[...]$) angenommen. Im Polytop-Modell werden nur einfache, nicht-verschachtelte Arrays beliebiger Dimensionalität betrachtet. Die *Indexausdrücke* dürfen affin sein, wobei skalare Variablen als 0-dimensionale Arrays behandelt werden.

Ein *Schleifenprogramm* besitzt eine oder mehrere Schleifen, welche *perfekt verschachtelt* oder *imperfekt verschachtelt* sein können. Von perfekter Verschachtelung spricht man, wenn jede äußere Schleife nur eine weitere Schleife und die innerste Schleife nur Statements enthält. Andernfalls spricht man von imperfekter Verschachtelung.
Die Array-Indices und Schleifengrenzen können Konstanten enthalten, die zur Übersetzungszeit unbekannt sind. Diese werden *Strukturparameter* genannt.

2.2 Instanzen

Wird ein Statement S von y Schleifen umgeben, so bezeichnet das Tupel $\vec{i} = (i_1, ..., i_y)$ den *Indexvektor*, wobei i_1 bis i_y jeweils durch ihre im Schleifenkopf angegebenen Grenzen ($l_1...l_y$ und $u_1...u_y$) beschränkt sind.
Eine *Instanz* (oder *Operation*) bezeichnet ein Paar bestehend aus einem Statement S und einem Zeitpunkt $\vec{i}$. Sie wird $\langle S, \vec{i} \rangle$ bzw. $\langle S, (i_1, ..., i_y) \rangle$ notiert.

2.3 Indexraum

Der *Indexraum* D_S eines Statements S besteht aus der Menge aller durch die umgebenden Schleifen generierten Instanzen. Dieser definiert sich damit aus der Menge aller Indexvektoren $\vec{i}$ für die gilt:

1. $(i_1, ..., i_y) \in \mathbf{Z}^y$ und
2. $\forall j : 1 \leq j \leq y : l_j \leq i_j \leq u_j$

Damit wird die Menge D_S im Raum durch die Schleifengrenzen begrenzt. Der Indexraum wird mathematisch durch ein Ungleichungssystem beschrieben, welches sich geometrisch als Polyeder veranschaulichen läßt.

2.4 Abhängigkeiten

Abhängigkeiten spiegeln die durch das Programm zwingend vorgeschriebene Ausführungsreihenfolge von Instanzen wieder. In Programmen ist eine Instanz I_1 genau dann von einer Instanz I_2 abhängig, wenn die folgenden Bedingungen erfüllt sind:

1. I_1 wird vor I_2 ausgeführt.

   Dann wird I_1 als *Quelle* und I_2 als *Ziel* bezeichnet.

2. I_1 und I_2 referenzieren die gleiche Speicherzelle.

3. Beide Instanzen werden durch das Programm generiert.

4. Zwischen den Instanzen wird die Speicherzelle nicht mehr modifiziert.

5. Mindestens einer der Zugriffe muß schreibend sein.

Eine Abhängigkeit (d) wird dann wie folgt notiert:

$$d : \langle S_1, (i_1, ..., i_y)\rangle \xrightarrow{\mathcal{T}}_{\mathcal{S}} \langle S_2, (j_1, ..., j_z)\rangle \tag{2.1}$$

Die Statements S_x der abhängigen Instanzen können hier identisch sein.

Je nachdem, ob der Zugriff auf die Speicherzelle ($\mathcal{S}$) lesend oder schreibend ist, wird der *Typ der Abhängigkeit* ($\mathcal{T}$) unterschieden (siehe Tabelle 2.1).

Ziel \ Quelle	liest	schreibt
liest	*Input*	*True*
schreibt	*Anti*	*Output*

Tabelle 2.1: Abhängigkeitstypen

Die *True*-Abhängigkeit wird unter Berücksichtigung der Bedingung 4 auch als *Flow* oder *Fluß*-Abhängigkeit bezeichnet. Diese Bedingung kann weggelassen werden und stellt eine Optimierung dar, da hierdurch indirekte Abhängigkeiten wegfallen.

Punkt 3 der Bedingungen sichert zu, daß beide Instanzen dieser Abhängigkeit, auch vom Programm generiert werden, d.h. zum Indexraum gehören. Ist dies nicht der Fall, so existiert zwar eine theoretisch mögliche Abhängigkeit, die jedoch im Programm nie zum Tragen kommt.
Diese Bedingung wird durch die Indexräume der Quelle und des Zieles der Abhängigkeiten angegeben. Die Relationen bezüglich der umgebenden Schleifenindices stellen also die Grenzen der Gültigkeit der Abhängigkeit dar.
Bemerkung: Diese Bedingung ist schon durch den Umstand zugesichert, daß hier Abhängigkeiten nur zwischen *Instanzen* bestehen können. Zur Klarheit sei dieser Punkt aber noch einmal herausgestellt.

Der optionale Punkt 5 der Bedingungen (der sogenannte *Bernsteinzusatz*) schließt Input-Abhängigkeiten aus.

Für die unterschiedlichen Typen werden auch ihre Anfangsbuchstaben als Kennzeichnung am Pfeil verwendet. (Also z.B. $\langle S_4, (1, 0)\rangle \xrightarrow{F}_{B} \langle S_7, (0, -4)\rangle$ für eine Fluß-Abhängigkeit.)

Die Differenz $\vec{j} - \vec{i}$ ist der *Abhängigkeitsvektor*. Ist dieser konstant, so bezeichnet man die Abhängigkeit als *uniform*.

Um affine Abhängigkeiten einheitlich behandeln zu können abstrahiert man vom Wert und betrachtet nur die Richtung der Abhängigkeit, d.h. das Vorzeichen.
Notation:

$$\vec{r} = (i_1, ..., i_y) \text{ wobei } i_x \in (-1, 0, +1).$$

Man bezeichnet diesen Vektor als *Richtungsvektor*.

Mitunter verwendet man noch die folgenden Abkürzungen:

$$\begin{aligned} 0+ &= 0 \text{ oder } 1 \\ 0- &= 0 \text{ oder } -1 \\ * &= \text{beliebig} \end{aligned}$$

Nähere Informationen zu Abhängigkeiten und deren Analyse sind der entsprechenden Literatur [4, 5, 10] zu entnehmen.

2.5 Das Polytop-Modell

Das Polytop-Modell bietet einen mathematischen Ansatz, um verschachtelte Schleifen zu parallelisieren. Die Schleifengrenzen müssen dabei affine Ausdrücke aus umgebenden Schleifenindices oder Strukturparametern darstellen.
Beim Parallelisieren wird jede Instanz des Schleifenkörpers im Raum und der Zeit verteilt, d.h. auf die virtuellen Prozessoren und Zeitpunkte. Die Verteilung im Raum wird *Allokation* und die in der Zeit *Schedule* genannt. Wendet man die dadurch entstehende *Raum-/Zeit-Abbildung* auf das Programm an, so entstehen für den Raum und die Zeit jeweils eigene Schleifensätze. Der Schedule und die Allokation werden hier als Funktionen bezüglich der Schleifenindices angegeben. Durch die Transformation wird die in den Abhängigkeiten enthaltene Zeit als eigene Dimension ausgelegt. Dies erlaubt es im Idealfall, wenn die Abhängigkeiten nur durch die Zeitdimension getragen werden, alle verbleibenden Raumschleifen parallel auszuführen.

2.6 LooPo

LooPo ist ein automatischer Schleifenparallelisierer, welcher die Verfahren des Polytop-Modells implementiert.
Die Parallelisierung geschieht dabei in mehreren Phasen:

1. *Scannen und Parsen.*
   Hierbei wird aus dem Quellcode der Parsebaum und die Symboltabelle erzeugt.

2. *Abhängigkeitsanalyse.*
   In diesem Schritt werden alle bestehenden Abhängigkeiten berechnet. Dabei werden alle zuvor beschriebenen Informationen über die Abhängigkeiten gespeichert.

3. *Der Scheduler.*
   Dieser berechnet aus den Abhängigkeiten [11–13] die Ausführungszeitpunkte jeder Instanz.

4. *Der Allokator.*
   Durch diesen werden die Orte (d.h. die Prozessoren) der Ausführung berechnet.

5. *Codegenerierung.*
   Die Daten des Schedulers und Allokators werden zur Erzeugung der Schleifen benutzt, wobei zuerst abstrakter Code erzeugt wird, welcher nach Wahl der Programmiersprache und des Maschinenmodells als übersetzbarer Quellcode generiert wird.

2.7 Rechnerarchitektur

Als Parallelrechnerarchitektur wird von einem Prozessorfeld mit verteiltem Speicher ausgegangen. Für die Tests wurde zur Implementierung der Kommunikationen auf die `MPI`-Bibliothek [23] zurückgegriffen. Diese stellt auch das für den asynchronen Fall nötige *blockierende Empfangen (Waiting Receives)* bereit.
Als reale Rechnerarchitekturen wurden für die Tests ein Feld von SUN Sparc Stations unterschiedlichen Typs, sowie ein Parallelrechner, eine Parsytec [26] mit 64 Prozessoren, verwendet.

2.8 Position innerhalb von LooPo

Diese Arbeit und die vorgestellen Algorithmen arbeiten auf Basis der in den Schritten 1 bis 4 berechneten Daten. Sie setzen darüber hinaus noch auf den transformierten Abhängigkeiten auf, welche als Teil des Schrittes 5 berechnet werden.
Theoretisch wäre die Positionierung auch zwischen den Schritten 2 und 3 denkbar, was jedoch, wie wir später noch sehen werden, Nachteile mit sich brächte.

Kapitel 3

LooPo-Code-Analyse

Für die im folgenden durchgeführten Analysen wurde als Beispiel der LU-Algorithmus gewählt, der zur Zerlegung einer regulären $n \times n$ Matrix in zwei Dreiecksmatrizen dient, deren Produkt wieder die Ausgangsmatrix darstellt.
Der serielle Algorithmus, der die Komplexität $O(n^3)$ besitzt, hat folgendes Aussehen:

3.1 Der Quellcode

```
    CONSTANT n;

    for (i1=1; i1<=n; i1++)
S1:    L[i1][1] = A[i1][1];

    for (jj=1; jj<=n; jj++)
S2:    U[1][jj] = A[1][jj] / L[1][1];

    for (j=2; j<=n; j++) {
       for (i=j; i<=n; i++) {
S3:       SUM[i][j] =0;
          for (k1=1; k1<=j-1; k1++) {
S4:          SUM[i][j] = SUM[i][j] + L[i][k1]*U[k1][j];
          }
S5:       L[i][j] = A[i][j] - SUM[i][j];
       }
S6:    U[j][j] = 1;
       for (i2=j+1; i2<=n; i2++) {
S7:       SUMM[j][i2] = 0;
          for (k=1; k<=j-1; k++) {
S8:          SUMM[j][i2] = SUMM[j][i2] + L[j][k]*U[k][i2];
          }
S9:       U[j][i2] = ( A[j][i2] - SUMM[j][i2] ) / L[j][j];
       }
    }
```

3.1.1 Abhängigkeiten

Der erste Schritt in der automatischen Codeanalyse von LooPo ist die Abhängigkeitsberechnung. Für den angegebenen LU-Algorithmus bestehen die folgenden Abhängigkeiten:

$$
\begin{aligned}
&d_1 : \langle S_1, (1, jj, j, i, k_1, i_2, k)\rangle \xrightarrow{F}_L \langle S_2, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_2 : \langle S_3, (i_1, jj, j, i, k_1, i_2, k)\rangle \xrightarrow{F}_{SUM} \langle S_4, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_3 : \langle S_4, (i_1, jj, j, k, k_1 - 1, i_2, k)\rangle \xrightarrow{F}_{SUM} \langle S_4, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_4 : \langle S_1, (i, jj, j, i, k_1, i_2, k)\rangle \xrightarrow{F}_L \langle S_4, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_5 : \langle S_5, (i_1, jj, k_1, i, k_1, i_2, k)\rangle \xrightarrow{F}_L \langle S_4, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_6 : \langle S_2, (i_1, j, j, i, k_1, i_2, k)\rangle \xrightarrow{F}_U \langle S_4, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_7 : \langle S_9, (i_1, jj, k_1, i, k_1, j, k)\rangle \xrightarrow{F}_U \langle S_4, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_8 : \langle S_3, (i_1, jj, j, i, k_1, i_2, k)\rangle \xrightarrow{F}_{SUM} \langle S_5, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_9 : \langle S_4, (i_1, jj, j, i, j - 1, i_2, k)\rangle \xrightarrow{F}_{SUM} \langle S_5, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_{10} : \langle S_7, (i_1, jj, j, i, k_1, i_2, k)\rangle \xrightarrow{F}_{SUMM} \langle S_8, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_{11} : \langle S_8, (i_1, jj, j, i, k_1, i_2, k - 1)\rangle \xrightarrow{F}_{SUMM} \langle S_8, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_{12} : \langle S_1, (j, jj, j, i, k_1, i_2, k)\rangle \xrightarrow{F}_L \langle S_8, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_{13} : \langle S_5, (i_1, jj, k, j, k_1, i_2, k)\rangle \xrightarrow{F}_L \langle S_8, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_{14} : \langle S_2, (i_1, i_2, j, i, k_1, i_2, k)\rangle \xrightarrow{F}_U \langle S_8, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_{15} : \langle S_9, (i_1, jj, k, i, k_1, i_2, k)\rangle \xrightarrow{F}_U \langle S_8, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_{16} : \langle S_7, (i_1, jj, j, i, k_1, i_2, k)\rangle \xrightarrow{F}_{SUMM} \langle S_9, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_{17} : \langle S_8, (i_1, jj, j, i, k_1, i_2, j - 1)\rangle \xrightarrow{F}_{SUMM} \langle S_9, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_{18} : \langle S_5, (i_1, jj, j, j, k_1, i_2, k)\rangle \xrightarrow{F}_L \langle S_9, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_{19} : \langle S_4, (i_1, jj, j, i, k_1 - 1, i_2, k)\rangle \xrightarrow{A}_{SUM} \langle S_4, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_{20} : \langle S_8, (i_1, jj, j, i, k_1, i_2, k - 1)\rangle \xrightarrow{A}_{SUMM} \langle S_8, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_{21} : \langle S_3, (i_1, jj, j, i, k_1, i_2, k)\rangle \xrightarrow{O}_{SUM} \langle S_4, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_{22} : \langle S_4, (i_1, jj, j, i, k - 1, i_2, k)\rangle \xrightarrow{O}_{SUM} \langle S_4, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_{23} : \langle S_7, (i_1, jj, j, i, k_1, i_2, k)\rangle \xrightarrow{O}_{SUMM} \langle S_8, (i_1, jj, j, i, k_1, i_2, k)\rangle\\
&d_{24} : \langle S_8, (i_1, jj, j, i, k_1, i_2, k - 1)\rangle \xrightarrow{O}_{SUMM} \langle S_8, (i_1, jj, j, i, k_1, i_2, k)\rangle
\end{aligned}
$$

Tabelle 3.1: Berechnete Abhängigkeiten

3.1.2 Schedule

Im nächsten Schritt berechnet LooPo einen Schedule. Dafür bietet LooPo drei Verfahren an: Lamport, Feautrier und Darte/Vivien.
Die Hyperebenenmethode nach Lamport ist für den LU-Algorithmus ungeeignet, da dieser perfekte Verschachtelung und uniforme Abhängigkeiten verlangt.
Die Methode nach Feautrier besitzt diese Einschränkungen nicht und ist daher geeignet. Die Methode verfolgt einen statementbasierten Ansatz und berechnet

daher je Statement einen eigenen Schedule. Dadurch lassen sich auf der einen Seite die Statements zwar besser in der Zeit verteilen, jedoch hat dies durch den zusätzlichen Kontrollaufwand Auswirkungen auf die Laufzeit (siehe auch Abschnitt 3.2).
Der Darte/Vivien Scheduler, als weitere denkbare Möglichkeit, liefert in diesem Falle, einen Schedule, der von der Allokation linear abhängig ist. Als Konsequenz daraus versendet er n^3 Daten. Dies ist schlechter als der Schedule nach der Methode von Feautrier.
Beim LU-Beispiel wurde daher der Feautrier-Scheduler angewandt. Dabei erhält man für diesen Fall je Statment den folgenden Schedule:

$$\begin{aligned}
S_1 &: Sched(i1, jj, j, i, k1, i2, k) = 0 \\
S_2 &: Sched(i1, jj, j, i, k1, i2, k) = 1 \\
S_3 &: Sched(i1, jj, j, i, k1, i2, k) = 0 \\
S_4 &: Sched(i1, jj, j, i, k1, i2, k) = 3 \cdot k_1 - 1 \\
S_5 &: Sched(i1, jj, j, i, k1, i2, k) = 3 \cdot j - 3 \\
S_6 &: Sched(i1, jj, j, i, k1, i2, k) = 0 \\
S_7 &: Sched(i1, jj, j, i, k1, i2, k) = 0 \\
S_8 &: Sched(i1, jj, j, i, k1, i2, k) = 3 \cdot k - 1 \\
S_9 &: Sched(i1, jj, j, i, k1, i2, k) = 3 \cdot j - 2
\end{aligned}$$

Tabelle 3.2: Berechneter Schedule

Die Statements S_1, S_3, S_6 und S_7 werden hier ausschließlich zum Zeitpunkt 0 und das Statement S_2 ausschließlich zum Zeitpunkt 1 ausgeführt.
Im weiteren sind damit nur die Statements S_4, S_5, S_8 und S_9 interessant, da nur diese wiederholt ausgeführt werden und dadurch den Hauptteil der benötigten Rechenzeit ausmachen. Der Umstand, daß einige Statements nur einmalig ausgeführt werden, kann ausgenutzt werden, um Kommunikationen einzusparen. Dazu müssen diese Operationen einmal auf jedem Prozessor ausgeführt werden, was sich bei den um ein vielfaches höheren Kommunikationskosten jedoch rentiert.

3.1.3 Allokationen

Die Allokation berechnet für jede Operation eine Zuordnung auf einen virtuellen Prozessor. Im folgenden werden zwei Allokationen (nach dem Verfahren von Feautrier) berechnet: eine eindimensionale und eine zweidimensionale Allokation. Eine zweidimensionale Allokation kann den Vorteil von mehr Paralleliät haben, bringt aber in der Regel auch mehr Kommunikation mit sich.

1-dimensionale Allokation

$$\begin{aligned}
S_1 &: Allok_1(i1, jj, j, i, k1, i2, k) = i1 \\
S_2 &: Allok_1(i1, jj, j, i, k1, i2, k) = jj \\
S_3 &: Allok_1(i1, jj, j, i, k1, i2, k) = i \\
S_4 &: Allok_1(i1, jj, j, i, k1, i2, k) = i
\end{aligned}$$

$$
\begin{aligned}
S_5 &: Allok_1(i1,jj,j,i,k1,i2,k) = i \\
S_6 &: Allok_1(i1,jj,j,i,k1,i2,k) = j \\
S_7 &: Allok_1(i1,jj,j,i,k1,i2,k) = j \\
S_8 &: Allok_1(i1,jj,j,i,k1,i2,k) = j \\
S_9 &: Allok_1(i1,jj,j,i,k1,i2,k) = j
\end{aligned}
$$

Tabelle 3.3: Berechnete Allokation (1-dimensional)

2-dimensionale Allokation

$$
\begin{aligned}
S_1 &: Allok_2(i1,jj,j,i,k1,i2,k) = (i1,0) \\
S_2 &: Allok_2(i1,jj,j,i,k1,i2,k) = (jj,0) \\
S_3 &: Allok_2(i1,jj,j,i,k1,i2,k) = (j,i) \\
S_4 &: Allok_2(i1,jj,j,i,k1,i2,k) = (j,i) \\
S_5 &: Allok_2(i1,jj,j,i,k1,i2,k) = (j,i) \\
S_6 &: Allok_2(i1,jj,j,i,k1,i2,k) = (j,0) \\
S_7 &: Allok_2(i1,jj,j,i,k1,i2,k) = (j,i2) \\
S_8 &: Allok_2(i1,jj,j,i,k1,i2,k) = (j,i2) \\
S_9 &: Allok_2(i1,jj,j,i,k1,i2,k) = (j,i2)
\end{aligned}
$$

Tabelle 3.4: Berechnete Allokation (2-dimensional)

3.1.4 Die transformierten Abhängigkeiten

Für die Codegenerierung werden nun die berechneten Abhängigkeiten gemäß des Schedules und der Allokation transformiert. Dabei erhält man dann im Falle der zweidimensionalen Allokation die folgenden Abhängigkeiten:

$$
\begin{aligned}
&d_1 : \langle S_1,(0,1,0)\rangle \xrightarrow{F}_L \langle S_2,(t,p_1,p_2)\rangle^{\dagger} \\
&d_2 : \langle S_3,(0,p_1,p_2)\rangle \xrightarrow{F}_{SUM} \langle S_4,(t,p_1,p_2)\rangle^{\dagger\ddagger} \\
&d_3 : \langle S_4,(t-3,p_1,p_2)\rangle \xrightarrow{F}_{SUM} \langle S_4,(t,p_1,p_2)\rangle^{\ddagger} \\
&d_4 : \langle S_1,(0,p_2,0)\rangle \xrightarrow{F}_L \langle S_4,(t,p_1,p_2)\rangle^{\dagger} \\
&d_5 : \langle S_5,(t-2,\tfrac{t+1}{3},p_2)\rangle \xrightarrow{F}_L \langle S_4,(t,p_1,p_2)\rangle \\
&d_6 : \langle S_2,(1,p_1,0)\rangle \xrightarrow{F}_U \langle S_4,(t,p_1,p_2)\rangle^{\dagger} \\
&d_7 : \langle S_9,(t-1,\tfrac{t+1}{3},p_1)\rangle \xrightarrow{F}_U \langle S_4,(t,p_1,p_2)\rangle \\
&d_8 : \langle S_3,(0,\tfrac{t}{3}+1,p_2)\rangle \xrightarrow{F}_{SUM} \langle S_5,(t,p_1,p_2)\rangle^{\dagger} \\
&d_9 : \langle S_4,(t-1,p_1,p_2)\rangle \xrightarrow{F}_{SUM} \langle S_5,(t,p_1,p_2)\rangle^{\ddagger} \\
&d_{10} : \langle S_7,(0,p_1,p_2)\rangle \xrightarrow{F}_{SUMM} \langle S_8,(t,p_1,p_2)\rangle^{\dagger\ddagger} \\
&d_{11} : \langle S_8,(t-3,p_1,p_2)\rangle \xrightarrow{F}_{SUMM} \langle S_8,(t,p_1,p_2)\rangle^{\ddagger} \\
&d_{12} : \langle S_1,(0,p_1,0)\rangle \xrightarrow{F}_L \langle S_8,(t,p_1,p_2)\rangle^{\dagger} \\
&d_{13} : \langle S_5,(t-2,\tfrac{t+1}{3},p_1)\rangle \xrightarrow{F}_L \langle S_8,(t,p_1,p_2)\rangle \\
&d_{14} : \langle S_2,(1,p_2,0)\rangle \xrightarrow{F}_U \langle S_8,(t,p_1,p_2)\rangle^{\dagger}
\end{aligned}
$$

$$d_{15} : \langle S_9, (t-1, \tfrac{t+1}{3}, p_2)\rangle \xrightarrow{F}_{U} \langle S_8, (t, p_1, p_2)\rangle$$
$$d_{16} : \langle S_7, (0, \tfrac{t+2}{3}, p_2)\rangle \xrightarrow{F}_{SUMM} \langle S_9, (t, p_1, p_2)\rangle^{\dagger}$$
$$d_{17} : \langle S_8, (t-2, p_1, p_2)\rangle \xrightarrow{F}_{SUMM} \langle S_9, (t, p_1, p_2)\rangle^{\ddagger}$$
$$d_{18} : \langle S_5, (t-1, p_1, p_1)\rangle \xrightarrow{F}_{L} \langle S_9, (t, p_1, p_2)\rangle$$
$$d_{19} : \langle S_4, (t-3, p_1, p_2)\rangle \xrightarrow{A}_{SUM} \langle S_4, (t, p_1, p_2)\rangle^{\ddagger}$$
$$d_{20} : \langle S_8, (t-3, p_1, p_2)\rangle \xrightarrow{A}_{SUMM} \langle S_8, (t, p_1, p_2)\rangle^{\ddagger}$$
$$d_{21} : \langle S_3, (0, p_1, p_2)\rangle \xrightarrow{O}_{SUM} \langle S_4, (t, p_1, p_2)\rangle^{\dagger\ddagger}$$
$$d_{22} : \langle S_4, (t-3, p_1, p_2)\rangle \xrightarrow{O}_{SUM} \langle S_4, (t, p_1, p_2)\rangle^{\ddagger}$$
$$d_{23} : \langle S_7, (0, p_1, p_2)\rangle \xrightarrow{O}_{SUMM} \langle S_8, (t, p_1, p_2)\rangle^{\dagger\ddagger}$$
$$d_{24} : \langle S_8, (t-3, p_1, p_2)\rangle \xrightarrow{O}_{SUMM} \langle S_8, (t, p_1, p_2)\rangle^{\ddagger}$$

Tabelle 3.5: Transformierte Abhängigkeiten, 2-dimensionale Allokation

Von den berechneten Abhängigkeiten (hier 24) ist nur ein Teil interessant. Dies sind die Abhängigkeiten, welche nicht nur einmalig (zu einem festen Zeitpunkt, gekennzeichnet mit †) und auch nicht auf demselben Prozessor (‡) ausgeführt werden.
Die Abhängigkeiten, welche auf denselben Prozessor verweisen, werden durch den Allokator eliminiert. Die Punkte, welche nur zu einem festen Zeitpunkt ausgeführt werden, dienen zur Initialisierung und können daher bei der Codegenerierung aus der Schleife herausgezogen werden.
Damit schränkt sich die Zahl der später zu beachtenden Abhängigkeiten auf die folgenden ein:

$$d_5,\ d_7,\ d_{13},\ d_{15} \text{ und } d_{18}.$$

Im Falle der 1-dimensionalen Allokation ergeben sich die folgenden Abhängigkeiten:

$$d_1 : \langle S_1, (0, 1, 0)\rangle \xrightarrow{F}_{L} \langle S_2, (t_1, p, t_2)\rangle^{\dagger}$$
$$d_2 : \langle S_3, (0, p, t_2)\rangle \xrightarrow{F}_{SUM} \langle S_4, (t_1, p, t_2)\rangle^{\dagger\ddagger}$$
$$d_3 : \langle S_4, (t_1-3, p, t_2)\rangle \xrightarrow{F}_{SUM} \langle S_4, (t_1, p, t_2)\rangle^{\ddagger}$$
$$d_4 : \langle S_1, (0, p, 0)\rangle \xrightarrow{F}_{L} \langle S_4, (t_1, p, t_2)\rangle^{\dagger\ddagger}$$
$$d_5 : \langle S_5, (t_1-2, p, 0)\rangle \xrightarrow{F}_{L} \langle S_4, (t_1, p, t_2)\rangle^{\ddagger}$$
$$d_6 : \langle S_2, (1, t_2, 0)\rangle \xrightarrow{F}_{U} \langle S_4, (t_1, p, t_2)\rangle^{\dagger}$$
$$d_7 : \langle S_9, (t_1-1, \tfrac{t_1+1}{3}, t_2)\rangle \xrightarrow{F}_{U} \langle S_4, (t_1, p, t_2)\rangle$$
$$d_8 : \langle S_3, (0, p, \tfrac{t_1+1}{3})\rangle \xrightarrow{F}_{SUM} \langle S_5, (t_1, p, t_2)\rangle^{\dagger\ddagger}$$
$$d_9 : \langle S_4, (t_1-1, p, \tfrac{t_1}{3}+1)\rangle \xrightarrow{F}_{SUM} \langle S_5, (t_1, p, t_2)\rangle^{\ddagger}$$
$$d_{10} : \langle S_7, (0, p, t_2)\rangle \xrightarrow{F}_{SUMM} \langle S_8, (t_1, p, t_2)\rangle^{\dagger\ddagger}$$
$$d_{11} : \langle S_8, (t_1-3, p, t_2)\rangle \xrightarrow{F}_{SUMM} \langle S_8, (t_1, p, t_2)\rangle^{\ddagger}$$
$$d_{12} : \langle S_1, (0, p, 0)\rangle \xrightarrow{F}_{L} \langle S_8, (t_1, p, t_2)\rangle^{\dagger\ddagger}$$
$$d_{13} : \langle S_5, (t_1-2, p, 0)\rangle \xrightarrow{F}_{L} \langle S_8, (t_1, p, t_2)\rangle^{\ddagger}$$

$$
\begin{aligned}
&d_{14} : \langle S_2, (1, t_2, 0)\rangle \xrightarrow{F}_{U} \langle S_8, (t_1, p, t_2)\rangle^{\dagger}\\
&d_{15} : \langle S_9, (t_1 - 1, \tfrac{t_1+1}{3}, t_2)\rangle \xrightarrow{F}_{U} \langle S_8, (t_1, p, t_2)\rangle\\
&d_{16} : \langle S_7, (0, \tfrac{t_1+2}{3}, t_2)\rangle \xrightarrow{F}_{SUMM} \langle S_9, (t_1, p, t_2)\rangle^{\dagger}\\
&d_{17} : \langle S_8, (t_1 - 2, \tfrac{t_1+2}{3}, t_2)\rangle \xrightarrow{F}_{SUMM} \langle S_9, (t_1, p, t_2)\rangle\\
&d_{18} : \langle S_5, (t_1 - 1, \tfrac{t_1+2}{3}, 0)\rangle \xrightarrow{F}_{L} \langle S_9, (t_1, p, t_2)\rangle\\
&d_{19} : \langle S_4, (t_1 - 3, p, t_2)\rangle \xrightarrow{A}_{SUM} \langle S_4, (t_1, p, t_2)\rangle^{\ddagger}\\
&d_{20} : \langle S_8, (t_1 - 3, p, t_2)\rangle \xrightarrow{A}_{SUMM} \langle S_8, (t_1, p, t_2)\rangle^{\ddagger}\\
&d_{21} : \langle S_3, (0, p, t_2)\rangle \xrightarrow{O}_{SUM} \langle S_4, (t_1, p, t_2)\rangle^{\dagger\ddagger}\\
&d_{22} : \langle S_4, (t_1 - 3, p, t_2)\rangle \xrightarrow{O}_{SUM} \langle S_4, (t_1, p, t_2)\rangle^{\ddagger}\\
&d_{23} : \langle S_7, (0, p, t_2)\rangle \xrightarrow{O}_{SUMM} \langle S_8, (t_1, p, t_2)\rangle^{\dagger\ddagger}\\
&d_{24} : \langle S_8, (t_1 - 3, p, t_2)\rangle \xrightarrow{O}_{SUMM} \langle S_8, (t_1, p, t_2)\rangle^{\ddagger}
\end{aligned}
$$

Tabelle 3.6: Transformierte Abhängigkeiten, 1-dimensionale Allokation

Ohne Berücksichtigung der Grenzen erhalten wir somit die folgenden Abhängigkeiten:

$$d_7,\ d_{15},\ d_{17} \text{ und } d_{18}.$$

LooPo berechnet bei den Grenzen der Abhängigkeiten d_{17} und d_{18} jeweils die leere Menge. Bei der Abhängigkeit d_{17} kommen innerhalb der Grenzen die Ungleichung $-1 \geq 0$ und innerhalb der Grenzen der Abhängigkeit d_{18} die sich widersprechenden Ungleichungen $t \geq 4$ und $t \leq -2$ vor. Daher ist die Existenzbedingung für Abhängigkeiten nicht erfüllt und die Abhängigkeiten fallen weg.

3.1.5 Der LooPo Code

Im nächsten Schritt wurde nun von LooPo der parallele Code generiert.
In dem mit Kommunikation generierten Code wird bei Benutzung der LooPo-internen Kommunikationsroutinen von indirekter Kommunikation ausgegangen. In der momentanen Implementierung besitzt jedes Datenelement einen ihm fest zugewiesenen Prozessor. Dies hat zur Folge, daß das entsprechende Datum immer nur von diesem Prozessor abgerufen werden kann und somit, wenn dieses Datum auf einem anderen Prozessor als dem „Heimatprozessor“ berechnet wurde, den Umweg über diesen nehmen muß. Diese Indirektion entspricht der gleichen Vorgehensweise, derer sich HPF bedient [7].
Dadurch entsteht zusätzliche Kommunikation, die durch eine direkte Kommunikation (wie sie momentan in LooPo implementiert wird) vermieden werden kann. Die indirekte Kommunikation ist für den allgemeinen Fall, in dem auch WHILE Schleifen zugelassen sind, notwendig, jedoch nicht für die einfachen Fälle der FOR-Schleifen.
Der Code mit umgewandelten Send/Receives besteht aus über 4400 Codezeilen und ist aufgrund der Größe und Laufzeit nicht als Grundlage zu gebrauchen.

Der als Grundlage für die Optimierung benutzte Code (siehe Anhang A.1), wurde deshalb ohne Kommunikation generiert. Die Kommunikationsstatements wurden von Hand eingefügt.

Der aufgearbeitete LooPo Code

Der mit den Kommunikationsstatements versehene und für die weiteren Experimente verwendete Code ist im Anhang A.2 abgedruckt.
Der Code orientiert sich stark an der Struktur die auch LooPo generiert. Dadurch soll die Vergleichbarkeit mit dem bisherigen LooPo-Code gewährleistet werden.
Im Anschluß an die Berechnung eines Wertes folgt direkt die Kommunikation. Hierzu werden die Kommunikationspartner, die das soeben berechnete Datum benötigen, durch Aufzählen aller Prozessoren und Abtesten der Abhängigkeit ermittelt. Dies unterscheidet sich von dem Verhalten von LooPo (siehe [9]) insofern, daß sich LooPo der umgekehrten Transformationsmatrix bedient und daraus die (eventuell zu vielen) Kommunikationspartner berechnet. Dies geschieht in vielen verschachtelten Schleifen. Außerdem wird bei LooPo ein Datum immer nur von dem besitzenden Prozessor versandt, was zur Folge hat, daß dieses eventuell zuerst zu diesem gesandt werden muß, also die Kommunikation indirekt stattfindet.

Hier stehen sich also die folgenden Kosten gegenüber:

1. Die Kosten durch die Aufzählung in der FOR-Schleife, der darin enthaltenen IF-Statements und der direkten Kommunikation.

2. Die Kosten durch die Berechnung der Kommunikationspartner und der indirekten Kommunikation.

Da die Kommunikation im Falle von LooPo doppelt stattfindet und dadurch einen hohen Kostenfaktor darstellt, unterscheiden sich die Kosten von dem hier angewendeten Verfahren nur gering, welche nach einigen Optimierungen an den Schleifenköpfen und der Art der Berechnung sogar unter den LooPo-Kosten liegen.

3.2 Die Analyse

Im folgenden soll nun der entstandene LooPo-Code analysiert werden, um Möglichkeiten zur Optimierung festzustellen.

Zwei Punkte fallen sofort auf:

1. Jedes Statement wird durch eine sehr lange IF-Anweisung eingeschlossen (siehe auch Anhang A.1). Dies kommt durch den statementbasierten Ansatz des Feautrier-Schedulers zustande. Hierbei wird jedes Statement durch eine IF-Anweisung eingeschlossen, welche den Ausführungszeitpunkt bestimmt. Dazu generiert LooPo je umgebender Raum- und

Zeitschleife zwei logische Ausdrücke, welche die Ober- und Untergrenzen angeben. Es entstehen hierdurch also $2(n_p + n_t)$ logische Ausdrücke pro Statement (n_p = Anzahl Raumschleifen und n_t = Anzahl Zeitschleifen).

Die Berechnung dieser logischen Ausdrücke stellt eine nicht zu vernachlässigende zusätzliche Rechenzeit dar, welche bei entsprechender Anzahl umgebender Schleifen sogar länger sein kann, als die des dadurch eingeschlossen Statements.

2. Die Kommunikation ist an das Statement gebunden. Sie findet sofort nach der Ausführung einer Operation und zu jedem Zeitschritt statt.

   Dies bedeutet, falls eine Abhängigkeit existiert, daß direkt im Anschluß an eine Berechnung auch eine Kommunikation folgt. Im schlechtesten Fall beträgt das Verhältnis von Berechnungen zu Kommunikationen $1 : (p-1)$, wobei p die Anzahl der Prozessoren darstellt.

   Da die Kosten für die Kommunikationen um ein Vielfaches höher liegen als die der Berechnungen bzw. der Auswertung der IF-Statements (siehe auch Abschnitt 4.1), steckt hier das größte Potential für Optimierungen.

Bei näherer Betrachtung der IF-Anweisungen macht man in unserem LU-Beispiel noch eine weitere Beobachtung:

In jeder IF-Anweisung, die ein Statement mit einer Fluß-Abhängigkeit umgibt, wird der Zeitpunkt durch eine Modulo-Berechnung kodiert. Diese vermeidbare Linearisierung eines Zeitschrittes resultiert in einer Vervielfachung der Zeitschritte (Faktor ist hier der Modulowert), wodurch die IF-Anweisungen mehrfach ausgewertet werden müssen. Dies trägt ebenfalls zur Verlängerung der Laufzeit bei.

Kapitel 4

Optimierung

In der Analyse (Abschnitt 3.2) stellten wir drei mögliche zu optimierende Punkte fest. Diese waren die hohe Anzahl der IF-Anweisungen, die Vervielfachung der Berechnungsschritte durch die Modulo-Berechnung sowie die Bindung der Kommunikationen an die Statements.
Hierbei stellt die Optimierung der Kommunikation das größte Potential für die Reduzierung der Kosten bereit. Deshalb beschäftigen wir uns, als zentralen Punkt dieser Arbeit, mit der Reduzierung der Kommunikationskosten.
Die beiden anderen Punkte sollen jedoch auch kurz angesprochen werden und wir werden diese in den Experimenten ebenfalls untersuchen.

Zu Beginn vergleichen wir die Kosten einiger realer Maschinen und gehen kurz auf Optimierungsmöglichkeiten bei der Codeerzeugung ein. Dann sehen wir uns zwei bekannte Verfahren zur Reduzierung der Kommunikation und deren Erweiterung an (Abschnitte 4.3.1 und 4.3.2). Im Anschluß daran stellen wir ein neues Verfahren vor, welches eine weitere Reduzierung der Kosten ermöglicht (Abschnitt 4.3.3). Die abschließenden Messungen in Abschnitt 4.4.2 überprüfen die Annahmen in der Realität.

4.1 Reale Kosten und Verhältnisse

Zuerst wollen wir die Kosten zweier realer Maschinen betrachten. Hierbei ist jeweils das Verhältniss von Berechnung zu Kommunikation besonders interessant. Diese Kosten fließen später u.a. in die Kostenabschätzungen der neuen Verfahren ein.

SUN Sparcstations:
Dieser Test (Tabellen 4.1 und 4.2) wurde auf zwei SUN Ultra 2 Sparcstations durchgeführt. Da sich hier viele interne (Cacheeffekte und die aktuelle Rechnerauslastung) sowie externe Einflüsse (z.B. die Netzauslastung) auswirkten, sind diese Werte ungenau und bedürfen bei Verwendung in der Implementierung einer Überprüfung. Besonders die Einflüsse durch die Rechner- und Netzauslastungen stellen auch den Grund der Nichtreproduzierbarkeit vieler Messungen auf dieser Architektur dar.

Ausführungszeiten		
Aktion	10000 ×	1 ×
Multiplizieren	0.001 s	0.1193 μs
LU-Berechnung	0.009 s	0.9001 μs
Umkopieren	0.008 s	0.8206 μs
LU IF-Statement	0.012 s	1.1876 μs
Aufbau und versenden	13.145 s	1314.4950 μs
- Kommunikationsaufbau - Double versenden		1313.6689 μs 0.8261 μs
Aufbau und Datenpaket versenden (Datenpaket = 10000 Double)	529.584 s	5.2958 μs

Tabelle 4.1: Messergebnisse auf SUN Ultra 2 Sparcstations (gemittelt)

Verhältnisse	
Senden : Aufbau	1 : 1590.26
Multiplizieren : Aufbau + Senden	1 : 11018.4
Multiplizieren : Aufbau	1 : 11011.475
Multiplizieren : Double senden	1 : 6.924
Multiplizieren : Umkopieren	1 : 6.878
Kopieren : Aufbau + Senden	1 : 11667.546
Double senden : Datenpaket senden	1 : 5.531
Double senden : 2 · Umkopieren + Paket senden	1 : 6.253
LU Berechnung : LU IF-Statement	1 : 1.319

Tabelle 4.2: Verhältnisse auf SUN Ultra 2 Sparcstations

Parsytec Parallelrechner:
Auf der Parsytec (Tabellen 4.3 und 4.4) erhält man sehr gut reproduzierbare Werte, was für die weiteren Tests ausgenutzt wurde. Allerdings gibt es Schwankungen aufgrund der Topologie, weshalb hier auch zwei Messungen mit unterschiedlicher Prozessorentfernung durchgeführt wurden.
Dies hängt damit zusammen, daß bei dem verwendeten Hauptprozessor (einem T800) Kommunikationen zu einem nicht-benachbarten Prozessor über alle Prozessoren auf dem Pfad gehen. Damit steigt die benötigte Zeit mit der Anzahl der dazwischenliegenden Prozessoren (siehe auch [6]) an.
Ein Problem stellte allerdings der sehr geringe Speicher der Parsytec (je Prozessor nur 4 MB) dar, was Tests mit größeren Problemgrößen unmöglich machte.

4.2 Optimierung des Codes

In der Analyse (Abschnitt 3.2) stellten wir zwei mögliche Optimierungen des generierten Codes fest. Zum einen betraf dies die hohe Anzahl der um jedes Statement herum generierten IF-Anweisungen und zum anderen die Vervielfachung der Berechnungschritte durch die Linearisierung.

Ausführungszeiten				
	Proz. 1 $\rightarrow$ Proz. 2		Proz. 1 $\rightarrow$ Proz. 64	
Aktion	10000 $\times$	1 $\times$	10000 $\times$	1 $\times$
Multiplizieren	0.1 s	10.3168 μs	0.1 s	10.3104 μs
LU-Berechnung	0.3 s	30.7392 μs	0.3 s	30.7392 μs
Umkopieren	0.4 s	44.6208 μs	0.4 s	44.6272 μs
LU IF-Statement	0.5 s	49.4272 μs	0.5 s	49.4272 μs
Aufbau und versenden	4.0 s	404.3776 μs	16.9 s	1693.2480 μs
- Kommunikationsaufbau - Double versenden		397.1599 μs 7.2177 μs		1679.0583 μs 14.1897 μs
Aufbau und Paket (1000 Double) senden	184.0 s	18.4002 μs	362.5 s	36.2451 μs

Tabelle 4.3: Messergebnisse auf Parsytec

Verhältnisse	$1 \rightarrow 2$	$1 \rightarrow 64$
Senden : Aufbau	1 : 55.0256	1 : 118.329
Multiplizieren : Aufbau + Senden	1 : 39.196	1 : 164.227
Multiplizieren : Aufbau	1 : 38.496	1 : 162.851
Multiplizieren : Double senden	1 : 0.7	1 : 1.376
Multiplizieren : Umkopieren	1 : 4.325	1 : 4.328
Kopieren : Aufbau + Senden	1 : 170.658	1 : 355.585
Double senden : Datenpaket senden	1 : 2.416	1 : 2.284
Double senden : 2 $\cdot$ Umkopieren + Paket senden	1 : 2.508	1 : 2.444
LU Berechnung : LU IF-Statement	1 : 1.608	1 : 1.608

Tabelle 4.4: Verhältnisse auf Parsytec

Die Messungen am LU-Beispiel, wie anhand der Tabellen 4.2 und 4.4 nachzuprüfen ist, bestätigen dieses Bild:

1. Die Zeit für die Bearbeitung der umschließende IF-Anweisung des Statements 9 ist 1.6 mal so hoch, wie die des Statements selbst. Dabei wurde die jeweilige IF-Anweisung komplett ausgewertet.

2. Durch die Linearisierung wird je ausgeführtem Statement die IF-Anweisung dreifach ausgeführt, was im schlechtesten Fall einen Faktor von 4.8 zur Folge hat (dieser wird besser, wenn die IF-Anweisung zuvor abgebrochen werden kann).

Als Konsequenz wäre hier eine Optimierung der IF-Anweisungen denkbar. So könnten Moduloberechnungen an den Anfang der IF-Anweisung gesetzt werden und die übrigen Bedingungen nach deren Abhängigkeit von den umgebenden Schleifen sortiert werden. Dies würde zu einem früheren Abbruch der Bedingung und damit zu einer schnelleren Ausführung führen.

Für beide Probleme können aber auch umfassendere Optimierungen angewandt werden:

1. Die Generierung dieser IF-Anweisungen kann vermieden werden, indem die Schleifen in mehrere Schleifen aufgeteilt werden, welche jeweils nur die zu berechnenden Punkte aufzählen. Diese Aufteilung kann durch den Algorithmus von Querillé [24] berechnet werden, dessen Anwendbarkeit für LooPo zur Zeit analysiert wird [25].

2. Die Generierung der Modulo-Berechnungen läßt sich algorithmisch durch die Verwendung eines multidimensionalen Schedules vermeiden. Dieser ist jedoch zur Zeit noch nicht in LooPo implementiert.

Beide Punkte betreffen Optimierungen, welche eine Verbesserung der Codeerzeugung *vor* der Anwendung der im folgenden vorgestellten Verfahren darstellen, wodurch der Code als Grundlage besser geeignet ist.

4.3 Optimierung der Kommunikation

Im Gegensatz zu der Optimierung des Codes bietet die Optimierung der Kommunikation das größte Potential für die Reduzierung der Kosten. Die sich hier auswirkenden Faktoren liegen um einige Größenordnungen höher (siehe Tabellen 4.2 und 4.4).

Zuerst betrachten wir zwei bekannte Verfahren zur Reduzierung der Kommunikation (Abschnitt 4.3.1) und stellen zwei Erweiterungen vor (Abschnitte 4.3.2 und 4.3.3). Die Aufstellung von Kostenmodellen und eines Algorithmus für die Verfahren beschließen diesen Abschnitt.

4.3.1 Partitionierung des Raums

Betrachtet man das Verhältnis von Kommunikation und Berechnung, so stellt man die hohen Kosten der Kommunikation fest. Als erstes sollte also das Verhältnis der Berechnungen zu den Kommunikationen verbessert werden. Dazu faßt man mehrere Berechnungsschritte auf einem Prozessor zusammen.

Diese Technik wird *Partitionierung* genannt [8] und hat generell folgende Ziele:

1. *Prozessoren sparen.* Das Modell geht bei der Allokation von einer beliebigen Anzahl Prozessoren aus. Da in der Realität die Anzahl der realen Prozessoren beschränkt ist, müssen die virtuellen Prozessoren auf die realen Prozessoren verteilt werden. Dies kann zyklisch oder durch Zusammenfassen von virtuellen Prozessoren zu Blöcken geschehen.

2. *Kommunikation reduzieren.* Kommunikation zwischen virtuellen Prozessoren erzeugt nur dann wahre Kommunikation, wenn die erzeugenden Abhängigkeiten über die Kachelränder hinweg gehen. Es entsteht also nur

Kommunikation zwischen den realen Prozessoren, was es weiterhin erlaubt, durch die freie Wahl der Größe und (weitgehend) der Form der Blöcke, die Zahl der Kommunikationen zu reduzieren. Die Verfahren zum Finden der optimalen Kachelgröße und -form sind in [14–20] beschrieben.

3. *Das Verhältnis von Rechenzeit zu Kommunikation verbessern.* Ziel ist es ein ausgewogenes Verhältnis zwischen den Kosten der Berechnung und denen der Kommunikation zu erhalten. Wird auf einem Prozessor zuwenig gerechnet, so fressen die Kosten der Kommunikation die Parallelität auf. Durch die Vermehrung der Berechnungen (und eventuell gleichzeitigem Einsparen von Kommunikationen) wird dieses Verhältnis verbessert.

   Die Kosten einer Berechnung hängt dabei auch von Hardwareeinflüssen (wie z.B. der Cachegröße) ab. Das Verhältnis einer Multiplikation zum Versenden des Ergebnisses beträgt auf der Parsytec z.B. bis zu $1 : 160$ (siehe Tabelle 4.4).

Der LooPo Code vor der Partitionierung hat grob die folgende Struktur:
(`t` = Zeit, sequentiell und `p` = Raum, parallel)

```
for t=ta to te
  for p=pa to pe
    [Statement]
    [Kommunikation]
  done
done
```

Die beiden existierenden und allgemein bekannten Verfahren hierfür, welche beide bereits in LooPo implementiert sind [8], sind das LPGS und LSGP Verfahren.

4.3.1.1 Die Verfahren

Das *LPGS Verfahren* (Lokal Parallel, Global Sequentiell) geht folgendermaßen vor: Der Raum wird in n *Kacheln* aufgeteilt, deren Größe der Anzahl der vorhandenen realen Prozessoren p entspricht. Diese Kacheln werden sequentiell aufgezählt. Lokal zur Kachel werden alle Berechnungspunkte parallel ausgeführt.

Das *LSGP Verfahren* (Lokal Sequentiell, Global Parallel) dagegen funktioniert folgendermaßen: Der gesamte Raum wird in p_r Kacheln aufgeteilt, wobei p_r der Anzahl der real existierenden Prozessoren entspricht. Diese Kacheln werden parallel ausgeführt. Lokal zur Kachel werden die Berechnungspunkte sequentiell ausgeführt.

Da beim LPGS Verfahren nicht die Aufteilung des gesamten Raumes von der Anzahl der Prozessoren abhängig ist, sondern dieser mit Kacheln der Größe p_r sukzessive aufgezählt werden kann, ist dieses Verfahren auch für WHILE Schleifen geeignet [8, Seite 16ff].

Anmerkung 1 *Bei diesen Verfahren sind beliebige Formen und Größen der Kacheln gültig [8].*

Durch Anwenden der obigen Techniken erhalten wir (hier im synchronen Fall) grob die folgenden Programmstruktur:

```
for t=ta to te
  for pb=pa to pe step ps
    for p=pb to pb+ps-1
      [Statement]
      [Kommunikation]
    done
  done
done
```

4.3.1.2 Partitionierungszeitpunkt

Die Partitionierung kann theoretisch zu zwei Zeitpunkten geschehen:

1. Vor der Transformation, direkt nach der Abhängigkeitsberechnung.

2. Nach den Transformation und vor der Codegenerierung.

Da LooPo auch mit imperfekten Verschachtelungen und nicht-uniformen Abhängigkeiten umgehen kann, hat die Partitionierung nach der Transformation den Vorteil, daß wir den Quellprogrammen weniger Einschränkungen auferlegen müssen.
Desweiteren haben nach der Transformation alle Instanzen der parallelen Schleifen ein gemeinsames Koordinatensystem und wir kennen zu diesem Zeitpunkt die maximale Anzahl der benötigten virtuellen Prozessoren. Dieser letzte Punkt ist besonders von Vorteil, da wir erst hierdurch eine realistische Abschätzung der Kosten machen können.

Ingesamt betrachtet erreichen wird durch die Partitionierung des Raumes wir drei Ziele:

1. Einsparung von Prozessoren.

2. Reduktion der Kommunikation.

3. Verbesserung des Verhältnisses zwischen Kommunikation und Berechnungen.

Dafür haben wir zwei mögliche Verfahren, das LPGS und LSGP-Verfahren, vorgestellt, wobei das LSGP auch für WHILE Schleifen anwendbar ist. Der optimale Zeitpunkt für die Partitionierung ist nach der Transformation und vor der Codegenerierung.

4.3.2 Zusammenfassen von Kommunikationen

Die zuvor vorgestellte Raumpartitionierung ist in Hinblick auf die Laufzeit und speziell den Kommunikationen nur sinnvoll, wenn durch sie viel Kommunikation zwischen den realen Prozessoren eingespart werden kann. Existiert jedoch auch nach der Raumpartitionierung noch viel Kommunikation, so reicht die Partitionierung allein nicht aus. Die Kommunikation muß also noch weiter reduziert werden, was durch ein Zusammenfassen von Kommunikationen erreicht werden soll.

In LooPo wird bei Bedarf nach jedem Statement eine Kommunikation durchgeführt. In dem Fall, wo aufgrund vieler einzelner Statements viele einzelne Kommunikationen zwischen den Prozessoren aufgebaut werden, ist dies ein sehr hoher Kostenfaktor. Selbst auf hinsichtlich der Kommunikation optimierten Parallelrechnern liegt der Aufbau einer Kommunikationsverbindung um bis zu einem Faktor von 100 höher als die Kommunikation selbst. Auf nicht optimierten Systemen ist dieser Faktor noch höher (siehe Tabellen 4.2 und 4.4). Stattdessen sollten die Kommunikationen gemeinsam jeweils am Ende eines Blocks geschehen.

Dies hat zur Folge, daß es bei mit dem Polytop-Modell parallelisierten Programmen (speziell LooPo-Programmen) zu einer linearen Verlangsamung bei Zunahme der Prozessorenanzahl kommt (siehe Abbildung 4.1):

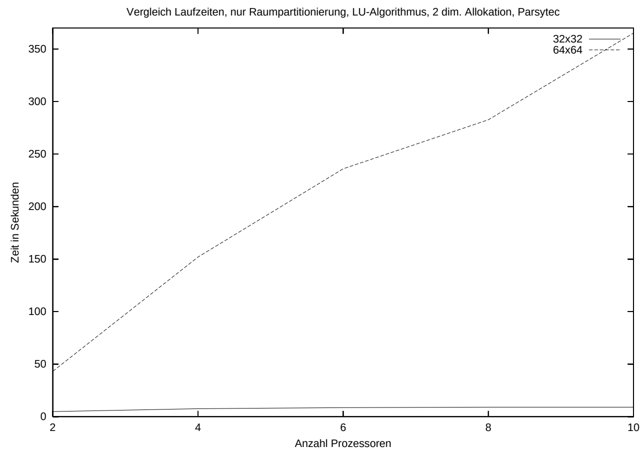

Abbildung 4.1: Verlangsamung bei Zunahme der Prozessoren infolge der gestiegenen Kommunikation

Möchte man nun die Kommunikationen zusammenfassen, so sind hierbei die folgende Punkte zu berücksichtigen:
Es können mehrere Statements pro Berechnungsschritt des Modells berechnet werden. Jedes dieser Statements kann mehrere Abhängigkeiten besitzen, was im Fall von affinen Abhängigkeiten bedeuten kann, daß zu jedem Zeitschritt Kommunikationen zwischen den Prozessoren erzeugt werden. Nimmt man nun

an, ohne den allgemeinen Fall zu beschränken, daß – wie für affine bzw. nicht-uniforme Abhängigkeiten typisch – nicht jeder Berechnungsschritt dieser Kachel auch zu einer Kommunikation führt, so müßte nur ein Teil der berechneten Daten jedes Statements an jeden anderen Prozessor versandt werden. Man hat somit eine irreguläre Verteilung der Abhängigkeiten, die über Blockgrenzen hinweg gehen.

Aufgrund dieser Verteilung der zu versendenden Daten ist es sehr schwierig, eine automatische Bündelung zu erhalten. Man kann sich dies vorstellen, als existierten zwischen den zu versendenen Daten Löcher, die nicht zum Versand bestimmt wären. Das Problem ergibt sich aus der Struktur der Arrays im Speicher. Zwei einfache Lösungsvorschläge sollen hier vorgestellt werden. Diese könnten durchaus noch weiter optimiert werden, was jedoch nicht Gegenstand dieser Arbeit sein soll.

4.3.2.1 Statisches Zusammenfassen

In diesem Fall umgehen wir das Problem der Löcher innerhalb der zu versendenden Datenpaketen, indem wir ein zusätzliches Kommunikationsvolumen in Kauf nehmen. Dies rechnet sich aufgrund des Verhältnisses von Aufbau- zu Übertragungskosten.
Haben wir nur uniforme Abhängigkeiten, so verbessert sich das Verhältnis noch weiter und ist genau dann optimal, wenn zwischen den uniformen Abhängigkeiten keine Löcher existieren.
Durch Berechnung wird das minimale und maximale zu versendende Datum bezüglich **aller** Abhängigkeiten dieser Speicherzelle für jeden einzelnen Empfangsprozessor ermittelt. Diese bestimmen dann die Grenzen des zu versendenden Paketes.
Man erhält so je Prozessor und je zu versendender Speicherzelle (Array) ein Datenpaket.
Im statischen Fall stehen sich dann die folgenden Kosten gegenüber: Die einmaligen Aufbaukosten (je Statement je empfangendem Prozessor) den wiederholten Aufbaukosten für jede Kommunikation. Es lassen sich so im schlechtesten Fall, in dem von jeder Kachel nur eine Kommunikation ausgeht, keine Kosten sparen (bzw. es kommt zu einer leichten Verschlechterung durch den zusätzlichen Berechnungsaufwand). Je mehr Kommunikationen von einem Statement (einer Abhängigkeit) zu einem Prozessor fließen, desto höher ist die Ersparnis.

Im LU-Beispiel käme es hier zu folgender Ersparnis (siehe Tabelle 4.5).

Partitionierung	normal	zusammengefaßt	Erparnis
Y Dimension	$22908 \cdot (C_A + C_S)$	$3136 \cdot C_A + 22908 \cdot C_S$	$19772 \cdot C_A$
X Dimension	$8190 \cdot (C_A + C_S)$	$168 \cdot C_A + 8190 \cdot C_S$	$8022 \cdot C_A$
X und Y Dim.	$4992 \cdot (C_A + C_S)$	$192 \cdot C_A + 4992 \cdot C_S$	$4800 \cdot C_A$
Kosten: C_A = Kommunikationsaufbau, C_S = Wert versenden			

Tabelle 4.5: Ersparnis durch Zusammenfassen von Kommunkationen

Hierbei wurden für den LU-Algorithmus (100×100, 16 Prozessoren, 2-dimensionale Allokation) die Kommunikationsvolumen der Prozessoren gemessen und das jeweils größte Volumen für die Auswertung herangezogen.

4.3.2.2 Dynamisches Zusammenfassen

Im dynamischen Fall geht man wie folgt vor: Im Anschluß an jede Berechnung wird (wie bisher auch) geprüft, ob dieses Datum von einem anderen Prozessor benötigt wird. Ist dies der Fall, so wird das Datum – anstelle der Kommunikation – in einen Puffer kopiert, der dem empfangenden Prozessor zugeordnet ist. Am Ende der Raumschleifen werden dann diese Puffer an die entsprechenden Prozessoren versandt. Aus den Datenpaketen werden (nach dem Empfang), die einzelnen Daten an den auf dem jeweiligen Prozessor zugewiesenen Speicherplatz zurückkopieren. Damit ergibt sich grob folgende Programmstruktur:

```
for t=ta to te
  [Paket empfangen, Umkopieren]
  for pb=pa to pe step ps
    for p=pb to pb+ps-1
      [Statement]
      [Wert in Puffer kopieren]
    done
  done
  [Kommunikation: Puffer versenden]
done
```

Dieses Verfahren hat gegenüber dem vorgenannten den Nachteil, daß ein zusätzlicher Aufwand für das Ein- und Auspacken nötig wird. Allerdings werden beim dynamischen Zusammenfassen, im Gegensatz dazu nur die wirklich benötigten Daten übertragen.

Trotz des zusätzlichen Aufwandes durch das Umkopieren beim dynamischen Zusammenfassen, wird so eine Geschwindigkeitssteigerung um einen Faktor von 3 bis 12 erreicht (siehe Tabelle 4.6), allerdings zu Lasten des Speicherverbrauchs, der durch den zusätzlichen Puffer ansteigt. Auch das Datenvolumen, welches versandt werden muß, steigt durch zusätzliche Verwaltungsdaten an. Diese sind hier mit berücksichtigt.

1000 × Double versenden	0.404 s	-	1.693 s
1 × Package von 1000 versenden	0.018 s	-	0.036 s
2000 × Daten kopieren	0.089 s		
2000 × Berechnung	0.021 s		
Gesamt	0.128 sec	-	0.146 sec
Faktor	3.152		11.588

Tabelle 4.6: Meßergebnisse Umkopieren

Beide hier diskutierten Ansätze zum Zusammenfassen von Kommunikationen ermöglichen eine deutliche Reduktion der benötigten Kommunikationskosten. Dies wird hauptsächlich durch eine Reduzierung der Kommunikationsaufbauten erreicht, welche den teuersten Teil der Kommunikation darstellen. Selbst beim dynamischen Zusammenfassen, bei dem zusätzliche Kosten durch das Umkopieren entstehen, ist eine Beschleunigung um den Faktor 3 bis 12 zu realisieren. Allerdings reicht die hier erreichte Beschleunigung noch nicht aus, um auch eine Beschleunigung des parallelisierten Programms im Vergleich zum sequentiellen zu erreichen. Daher müssen wir uns nach weiteren Möglichkeiten umsehen, ohne die hier vorgestellten Techniken zur Reduzierung der Kommunikationskosten zu vernachlässigen.

4.3.3 Partitionierung der Zeit

Nachdem wir nun den größten Kostenfaktor durch die obigen Verfahren reduziert haben, stellt sich die Frage, ob noch eine weitere Reduzierung möglich und nötig ist. Oft reicht die Granularität noch nicht aus, weshalb noch weitere Einsparungen wünschenswert wären. Dies wird jedoch – wie wir noch sehen werden – durch die in der Zeit ausgelegten Abhängigkeiten behindert.

4.3.3.1 Voraussetzungen

Im folgenden gehen wir von der direkten Kommunikation aus. Wir nehmen als Vorzugsrichtung der Abhängigkeiten, ohne die Allgemeinheit zu beschränken, die Richtung in aufsteigender Prozessornumerierung an. Diese stellt in den späteren Beispielen auch die *Verschieberichtung* dar. Die *Rückrichtung* verläuft dann in entgegengesetzter Richtung, wobei hier überall die Zeit herauszuprojizieren ist.

Als Beispiel soll hier zur einfacheren Anschauung ein Programm mit einer uniformen Abhängigkeit dienen (Abbildung 4.2):

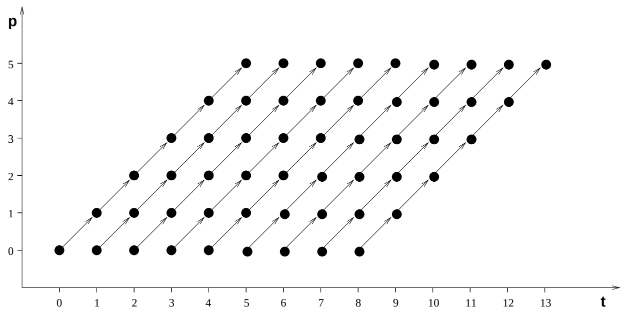

Abbildung 4.2: Einfaches Beispiel mit Abhängigkeiten

Partitioniert man dieses Programm mit den bereits bekannten Techniken (LPGS), so erhält man folgendes (Abbildung 4.3).

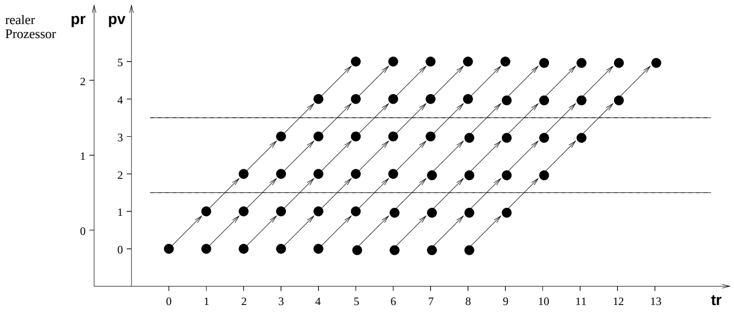

Abbildung 4.3: Normale Partitionierung

Hier ist gut zu sehen, daß im folgenden nur die Abhängigkeiten, welche Kommunikationen zwischen den Prozessoren erzeugen, interessant sind.

4.3.3.2 Die Idee

Eine Möglichkeit, diese Kosten zu senken, wäre auch Kommunikationen über mehrere Zeitschritte hinweg zusammenzufassen. Da die Kommunikationen nur entlang der Zeitschritte stattfinden [8, Seite 33ff], erweitert dies die bekannten Techniken der Partitionierung des Raumes auf Grundlage des Zielprogrammes auch auf die der Zeit.

Die Grundidee liegt darin, mehrere *reale Zeitschritte* (t_r) zu einem Zeitblock zusammenzufassen und die Kommunikationen der bisher einzelnen Zeitschritte gebündelt am Ende des Zeitblocks durchzuführen. Dadurch spart man die Kommunikationenaufbaukosten um den Faktor der Zeitblockbreite ein. Durch das Zusammenfassen erhält jeder Zeitblock einen eigenen *logischen Ausführungszeitpunkt* (t_l).

In unserem einfachen Beispiel erhalten wir nach Anwendung des beschriebenen Verfahren die folgende Struktur (siehe Abbildung 4.4).

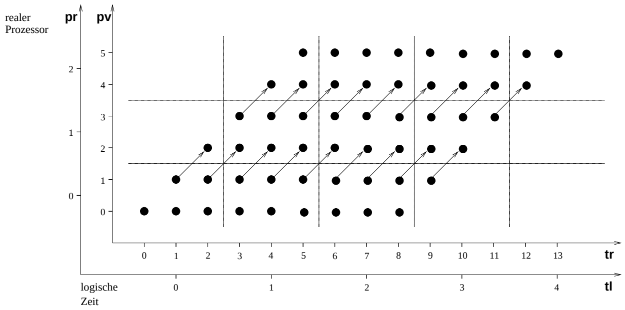

Abbildung 4.4: Naïve Partitionierung der Zeit

Das Bündeln der Kommunikationen am Ende eines Zeitblocks hat jedoch Folgen. Um die Abhängigkeiten zu wahren müssen alle von diesen verzögerten Kommunikationen abhängigen Zeitblöcke ebenfalls verzögert werden.

> Dies wird auch am obigen Beispiel (Abbildung 4.4) deutlich, da hier Kommunikationen zwischen zwei zeitgleichen Blöcken in einen Zeitblock fallen. Da das jedoch nicht geschehen darf, ist eine Verschiebung (Abbildung 4.5) nötig.

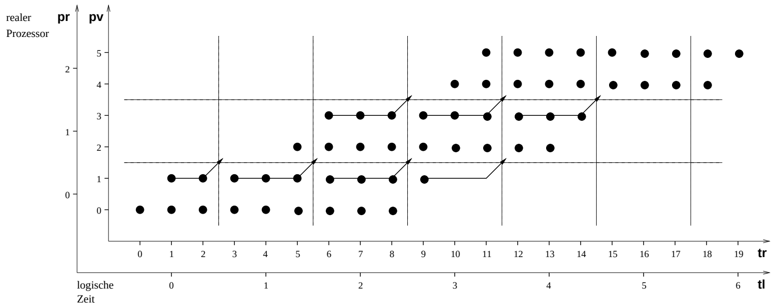

Abbildung 4.5: Nötige Verschiebung

Die Anzahl der zur Partitionierung verwendeten Schleifen läßt sich unter Berücksichtigung der Gültigkeit des Verfahrens frei bestimmen. Durch die Anzahl und die Länge der gewählten Schleifen läßt sich die Granularität und damit das Verhältnis von Rechenzeit zu Kommunikation festlegen.

Im folgenden wollen wir nun die einzelnen Probleme näher betrachten. Zuerst legen wir fest, wann das Verfahren einsetzbar ist. Anschließend betrachten wir die Restriktionen bezüglich der Form und Größe der Blöcke. Dann beschäftigen wir uns mit der Größe des nötigen Versatzes und den Auswirkungen auf den generierten Code. Einzelne Probleme, bei denen die Einschränkungen zu streng waren, werden wir dann noch versuchen zu verallgemeinern.

4.3.3.3 Gültigkeit

Da das Zusammenfassen von Zeitschritten Einfuß auf die Abhängigkeiten nimmt, können diese auch hierdurch zerstört werden. In diesem Fall kann das Verfahren nicht angewandt werden. Wann das Zusammenfassen gültig ist, soll der nächste Abschnitt klären.

In unserem obigen Beispiel (Abbildung 4.5) stellt die Verschiebung kein Problem dar. Hier verlaufen die Abhängigkeiten ausschließlich in aufsteigender Richtung und werden alle durch die Verzögerung der Blöcke erhalten.
Ein Problem stellen Abhängigkeiten dar, wenn sie sowohl in aufsteigender als auch in absteigender Richtung existieren (siehe Abbildung 4.6). Hier können aus der Verzögerung Abhängigkeiten resultieren, die innerhalb eines Zeitblocks liegen. Dann müßte innerhalb dieses Zeitblocks eine Kommunikation in Rückrichtung stattfinden, was jedoch aufgrund der Bündelung der Kommunikation am Ende nicht möglich ist.

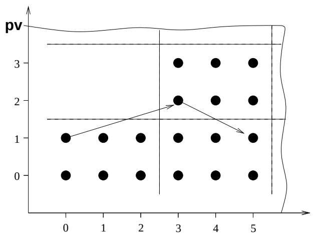

Abbildung 4.6: Zyklische Abhängigkeit

Projiziert man aus den Abhängigkeiten die Zeit heraus, so erhält man zyklische Abhängigkeiten zwischen den einzelnen Prozessoren. Diese stellen die Ursache des Problems dar.

Somit ist das Partitionieren der Zeit nur gültig, wenn nach der Partitionierung zwischen den Zeitblöcken gilt:

Es existieren keine zyklischen Abhängigkeiten.

Für die Zeit können diese vom Modell her nicht erzeugt werden, da sonst negative Zeitschritte erzeugt würden. Daher gilt für alle Abhängigkeitsrichtungsvektoren der Zeit, sie haben immer die Form: $(0+, ..., 0+)$.

Für die Raumdimensionen, die zusammengefaßt werden sollen, muß allerdings jeweils eine zusätzliche Bedingung erfüllt sein:

Die Richtungsvektoren aller Abhängigkeiten **einer**
Raumdimension müssen denselben Wert haben.

Durch diese hinreichende Bedingung ist jeder Zyklus ausgeschlossen und damit auch zyklische Abhängigkeiten, die über mehrere Dimensionen hinweg entstehen könnten. Ist diese Bedingung für eine beliebige Menge an Raumdimensionen erfüllt, so können diese und eine beliebige Anzahl an Zeitschleifen partitioniert werden.
Ist die Bedingung nicht erfüllt, so kann die Methode zur Partitionierung der Zeit nicht angewendet werden.
Wenn die Partitionierung der Zeit nicht durchgeführt werden kann, so bleibt allerdings immer noch die Möglichkeit die Kommunikationen der Raum-Partitionierung zusammenzufassen (siehe Abschnitt 4.3.2) und dadurch Zeit zu sparen.

Wir wissen nun, daß das Zusammenfassen der Blöcke erlaubt ist, wenn es gelingt Zyklen zu vermeiden. Die hier angegebene Bedingung ist allerdings etwas zu streng und wird später in Abschnitt 4.3.3.7 noch einmal näher betrachtet. Für die weiteren Betrachtungen gehen wir von der hier angegeben Bedingung aus.
Als nächstes wollen wir die Form der Zeitblöcke betrachten.

4.3.3.4 Form der Blöcke

Im einfachsten Fall existiert nur eine Zeitschleife, so daß hier die Form der Blöcke nicht von Bedeutung ist. In den Fällen, wo mehr als eine Zeitschleife existiert, wird die Form durch den Umstand bestimmt, daß alle Zeitdimensionen Abhängigkeiten tragen. Die Form kann also nicht frei gewählt werden.

Wir wissen (siehe Anmerkung 1, Abschnitt 4.3.1.1), daß beliebige Formen und Größen der Kacheln in den Raumdimensionen gültig sind, solange die Zeitdimensionen jeweils eine Länge von 1 besitzen

Für die Zeitdimensionen dagegen ergibt sich nun folgendes:

Es existieren n_t ineinander verschachtelte Zeitschleifen. Es seien in einer Zeitdimension t_m (mit $1 < m < n_t$) I' und I'' zwei (zeitlich) aufeinanderfolgende Instanzen. Werden I' und I'' nun in einem Zeitblock B zusammengefaßt, so müssen auch alle inneren Zeitdimensionen $t_{m+1}...t_{n_t}$ und damit alle Instanzen dieser Zeitdimensionen vollständig in B enthalten sein.

Formal:

Für beliebige $i_x : (I' := (t_1, ..., t_m, i_1, ..., i_{d-m}) \in B \wedge$
$I'' := (t_1, ..., t_m + 1, i_1, ..., i_{d-m}) \in B) \Rightarrow$
$\forall i_x : (I := (t_1, ..., t_m, i_1, ..., i_{d-m}) \in B$
mit $\forall 1 \leq x \leq d - m : i_{x+m} \in T_{x+m}$ (T_i ist Indexraum von Schleife i).

Begründung:

Da $I' \in B \wedge I'' \in B$ gilt, darf zwischen diesen Instanzen und keiner Instanz *dazwischen*, welche durch die untergeordneten Schleifen aufgezählt werden, eine Kommunikation stattfinden. Da jedoch die Kommunikationen durch die Abhängigkeiten erzeugt werden, welche wiederum aufgrund des Modells gerade durch die Zeitschleifen aufgezählt werden, muß es hier Kommunikationen geben. Also müssen alle Instanzen dazwischen und damit alle untergeordneten Schleifen zum Zeitblock gehören.
Daraus folgt weiter, daß nur eine einzige Zeitschleife eine Länge ungleich 1 oder der maximalen Ausdehnung der Dimension haben kann.
Zu bemerken bleibt, daß die rechteckige Form der Zeitkacheln durch den Zwang die inneren Zeitschleifen komplett in den Zeitblock mit aufzunehmen (was sich wiederum durch die in der Zeit ausgelegten Abhängigkeiten bedingt), erzwungen wird.

Allerdings ist die rechteckige Form nicht immer das Optimum. Im Falle von mehrdimensionaler Zeit kann sich die obige Bedingung negativ auswirken, da hierdurch die maximal mögliche Anzahl der zusammenfaßbaren Zeitschritte reduziert wird.

Das folgenden Beispiel (Abbildung 4.7) macht dies deutlich:

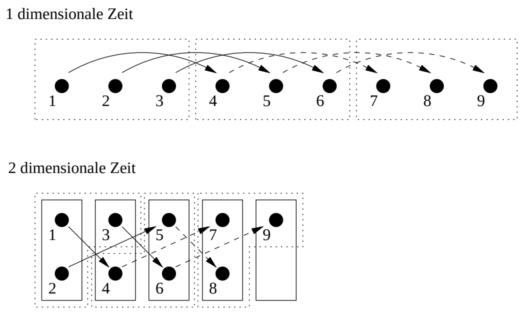

Abbildung 4.7: Ausgeschlossene Blockform

> Liegt eine eindimensionale Zeit vor, so ist aufgrund der Abhängigkeiten eine Blockgröße von 3 möglich. Im Falle einer zweidimensionalen Zeit läßt dies jedoch die erzwungene rechteckige Form nicht zu. Dadurch beträgt die maximale Blockgröße im zweidimensionalen Fall nur 2, was unter dem Optimum liegt. Die in der zweidimensionalen Zeit entstehende Form ist in Abbildung 4.7 angedeutet.

Als Lösung bietet sich hier an die mehrdimensionale Zeit auf eine Dimension „auszubreiten“. Hierdurch wird es dann ermöglicht, eine beliebige Anzahl an Zeitschritten zusammenzufassen bzw. (nach Rückprojektion) nichtrechteckige Formen zu erzeugen.
Diese Lösung ist jedoch nur für Zeitdimensionen mit konstanten Grenzen durchführbar, da dies nicht zur Laufzeit sondern zur Übersetzungszeit zu geschehen hat.

Durch die Zeitschleifen wird also eine rechteckige Form der Kacheln erzwungen, obwohl dies nicht immer die optimale Form darstellt. Da, wenn nur eine Zeitschleife existiert, die Anzahl der zusammenzufassenden Zeitschritte fast frei gewählt werden kann, kann so weit wie möglich versucht werden, mehrere Zeitschleifen zu einer auszubreiten.
Daß die Anzahl jedoch nicht völlig frei gewählt werden kann zeigt der nächste Abschnitt.

4.3.3.5 Größe der Blöcke

Die Größe der Blöcke kann nicht völlig frei gewählt werden, sondern unterliegt einigen Restriktionen, welche aus mehreren Problemen entstehen. Unter den in Abschnitt 4.3.3.3 getroffenen Einschränkungen bezüglich der Gültigkeit kann es hier zu zwei Problemen kommen:

1. Falls sich in einem Datenpaket die Daten zweier, zeitlich aufeinanderfolgender Kommunikationen zu einer Speicherzelle befinden, so geht durch das Zusammenfassen der Kommunikationen der vorherige Wert verloren. Zwischen diesen beiden Instanzen besteht dann eine Output-Abhängigkeit.

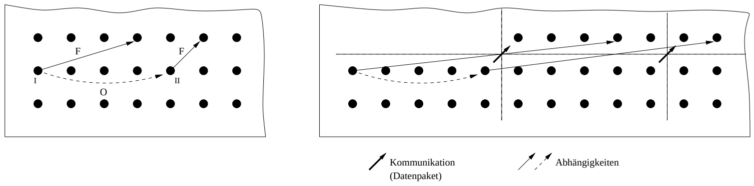

Abbildung 4.8: Überschneidung durch Output-Abhängigkeiten

> In diesem Beispiel kommt es bei einer Zeitblockgröße von 5 zu einem Konflikt in den Kommunikationen, da in einem Datenpaket beide, aufgrund der Fluß-Abhängigkeiten, zu übertragenden Werte (I und II), enthalten sind.

Da der Scheduler **alle** Abhängigkeiten berücksichtigt, sind abhängige Instanzen mindestens einen Zeitschritt voneinandern entfernt. Somit kann dieses Problem bei einer Zeitblockgröße von 1 nie auftreten.

Beim statischen Zusammenfassen der zu übertragenden Werte (siehe dazu 4.3.2.1) tritt dieses Problem unabhängig vom Zielprozessor auf. Beim dynamischen Zusammenfassen kommt es hingegen (wenn das Umkopieren direkt am Statement stattfindet) nur beim gleichen Zielprozessor zu Problemen.

2. Ein weiteres Problem tritt auf, wenn ein bereits übertragener Wert durch die Übertragung eines neuen Datenpaketes vor dessen Verwendung überschrieben wird. Dann besteht zwischen der Empfängerinstanz und der zweiten Senderinstanz eine induzierte Anti-Abhängigkeit.

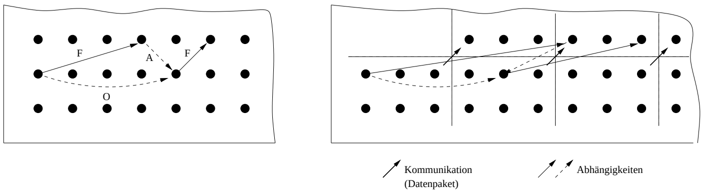

Abbildung 4.9: Überschneidung durch Anti-Abhängigkeiten

> In diesem Beispiel (Abbildung 4.9) wird der Wert des Datums, welcher aufgrund der linken Fluß-Abhängigkeit in der Abbildung im ersten Datenpaket übertragen wird, durch den für die zweite Abhängigkeit übertragenen Wert im zweiten Datenpaket überschrieben.

Für beide Probleme gibt es nun mehrere Lösungsmöglichkeiten:

- Ohne Anpassungen an der Codeerzeugung durchführen zu müssen, bleibt nur der Weg, die Größe der Blöcke zu beschränken. Hierzu sind die bereits erwähnten Output- und Anti-Abhängigkeiten zu betrachten. Zu berücksichtigen sind nur die Abhängigkeiten, die in den zusammenzufassenden Dimensionen liegen.

Hiervon sind nur die Output-Abhängigkeiten interessant, die nach der Transformation immer noch auf demselben Prozessor liegen. Mindestens eine der von beiden Enden der Output-Abhängigkeit abgehenden Fluß-Abhängigkeiten muß hierbei auf einen anderen Prozessor verweisen. Beim dynamischen Zusammenfassen müssen die von beiden Punkten abgehenden Fluß-Abhängigkeiten zusätzlich noch auf denselben Prozessor zeigen.

Von den Anti-Abhängigkeiten sind nur solche interessant, deren Quelle und Ziel auf unterschiedlichen Prozessoren liegen. Durch das Zusammenfassen der Zeitschritte dürfen keine Anti-Abhängigkeiten zwischen den Zeitblöcken entstehen, die eine negative Richtung in einer der Zeitkomponenten besitzen. Innerhalb eines Zeitblocks dürfen allerdings solche Anti-Abhängigkeiten existieren.

Damit ergibt sich die maximale Größe in der betrachteten Zeitdimension durch das Minimum der Länge in der Zeit aller beim Zusammenfassen zu berücksichtigenden Output- und Anti-Abhängigkeiten.

In dem obigen Fall (Abbildung 4.8) betrüge die maximale Zeitblockbreite bedingt durch die Output-Abhängigkeiten 4. Dadurch ist allerdings noch nicht das zweite Problem (Abbildung 4.9) ausgeschlossen.
Dies wird durch Berücksichtigung der Anti-Abhängigkeiten erreicht. Das Minimum beider Längenbegrenzungen beträgt somit 1, was dann die Obergrenze der Zeitblockbreite darstellt. In diesem Fall wir die Breite soweit eingeschränkt, daß kein Zusammenfassen der Zeit möglich ist.

Diese Bedingung läßt sich noch weiter verschärfen indem man überhaupt keine Anti-Abhängigkeiten zuläßt, deren Richtung den der Fluß-Abhängigkeiten widerspricht. Dadurch kann es nicht zu dem Überschneidungsproblem durch Anti-Abhängigkeiten kommen und die Anti-Abhängigkeiten bleiben bezüglich der Blockgröße unberücksichtigt. Dies wurde durch das Einbeziehen der Anti-Abhängigkeiten in Abschnitt 4.3.3.3 bereits getan.

- Soll die Größe der Blöcke nicht beschränkt werden, so muß der Code zur Kommunikation angepaßt werden. Hier wird nach dem verwendeten Verfahren des Zusammenfassens unterschieden:

  Wird **statisch** zusammengefaßt, so müssen die Daten in separaten Datenpaketen versandt werden und auf der anderen Seite später (d.h. der entsprechenden Zeitschleife untergeordnet) empfangen werden.

  Damit werden für die problematischen Abhängigkeiten kleinere Blöcke erzeugt, was im schlechtesten Fall, wenn dies alle Abhängigkeiten betrifft, der Partitionierung mit Kommunikationsbündelung (Abschnitt 4.3.2) entspricht.

Im **dynamischen** Fall muß das Datenpaket stückchenweise ausgepackt werden, was abhängig von der virtuellen Zeit des Zeitblocks stattzufinden hat. Dazu muß in das Datenpaket der Zeitpunkt der Gültigkeit mit einkodiert und das Umkopieren nicht direkt anschließend an das Empfangen, sondern am Anfang der Zeitschleife, welche die einzelnen Zeitpunkte des Zeitblocks aufzählt, ausgeführt werden.

Damit ergibt sich grob folgende Programmstruktur:

```
for tb=ta to te step ts
  [Paket empfangen]
  for t=tb to tb+ts-1
   [Teilweise umkopieren]
   for pb=pa to pe step ps
     for p=pb to pb+ps-1
       [Statement]
       [Wert in Sende-Puffer kopieren]
     done
   done
  done
  [Kommunikation: Puffer versenden]
done
```

Hierbei gibt die Länge der Output-Abhängigkeit in der Zeit die maximale Anzahl der Zeitschritte an, welche zu einem Zeitpunkt umkopiert werden dürfen.

Ein Problem stellen in diesem Zusammenhang Werte dar, die zu einem sehr frühen Zeitpunkt in einem Datenpaket übertragen wurden, jedoch erst viel später (d.h. mehrere Zeitblöcke später) benötigt werden. In diesem Fall sind beim sukzessiven Entpacken der Datenpakete diese solange zwischenzuspeichern. Der zusätzliche Pufferbedarf steigt hierbei um den Faktor:

$$\left\lceil \frac{\text{Länge der längsten Output-Abhängigkeit}}{\text{Zeitblockgröße}} \right\rceil \tag{4.1}$$

Als Alternative dazu könnte auch die Kommunikation der Werte (bzw. das Umkopieren der Daten) um die *Länge der jeweiligen Output-Abhängigkeit minus 1* Schritte verzögert werden.

- Als letzte Möglichkeit bleibt natürlich auch noch die *Single Assignment* Methode [21]. Da hier jeder Instanz eine eigene Speicherstelle zugewiesen wird, können Überschneidungen nicht auftreten.

Zusammenfassend betrachtet kann es also bei zu großer Wahl eines Zeitblocks zu einem durch Output- oder Anti-Abhängigkeiten induzierten Überschneidungsproblem kommen. Um dieses Problem zu lösen hat man drei Möglichkeiten. Erstens läßt sich die Blockgröße unter Beachtung der Längen der Output- und Anti-Abhängigkeiten in der Zeit beschränken. In diesem Fall müssen keine Anpassungen bezüglich der Codegenerierung gemacht werden. Verzichtet man in der zweiten Möglichkeit auf die Beschränkung der Größe, so muß der generierte Code diesem Problem durch zusätzlichen Puffer Rechnung tragen, wobei hier nach statischem und dynamischen Zusammenfassen unterschieden wird. Als letzte Möglichkeit bleibt allerdings auch noch die Single Assignment Methode.
Nachdem nun die Gültigkeit, Form und Größe der Zeitblöcke bestimmt sind, wollen wir nun die Auswirkungen auf den zeitlichen Ablauf, d.h. der bereits angesprochenen nötigen Verzögerung, und die verschiedenen Möglichkeiten zur Kommunikationsbündelung betrachten.

4.3.3.6 Auswirkungen auf Ablauf und Kommunikation

Wenn wir die Kommunikationen mehrerer Zeitschritte zusammenfassen, so hat dies Auswirkungen auf den Ablauf des Codes. Da die Kommunikationen, durch die in der Zeit ausgelegten Abhängigkeiten erzeugt werden, folgt daraus, daß eventuell abhängige Kacheln anderer Prozessoren verzögert werden müssen.
Außerdem bestehen mehrere Möglichkeiten der Kommunikationsbündelung, deren Vor- und Nachteile hier ebenfalls betrachtet werden sollen.

Die Art und Weise wie die Verzögerung und Kommunikationsbündelung zu implementieren sind, ist abhängig davon, ob es sich um ein synchrones oder asynchrones Programm handelt.

Synchrone Programme:

Ein Problem ergibt sich, wenn Kacheln, die zum Zeitpunkt t_l ausgeführt werden, aufgrund der Abhängigkeiten eigentlich erst zum Zeitpunkt $t_l + x$ ausgeführt werden dürften. Um die Berechnungen innerhalb der Kacheln zum richtigen Zeitpunkt beginnen zu lassen, muß die lokale Prozessorzeit diese Verzögerung berücksichtigen.
Dazu wird zur lokalen Prozessorzeit das Produkt aus Kachelgröße und Prozessornummer hinzuaddiert: $t_{l_{neu}} = t_{l_{alt}} + (s \cdot p_r)$. Hierbei gibt s die Breite des Zeitblocks an[1]. Dies hat allerdings zur Folge, daß alle von LooPo generierten Ausdrücke, welche die Zeit berücksichtigen, ebenfalls umgerechnet werden müssen: $T_{intern}(t_r) = t_r - (s \cdot p_r)$. Dadurch wird die jeweils erste Kachel um $s \cdot p_r$ vergrößert und damit der Beginn der ersten Berechnung entsprechend verzögert.

Um dies zu vermeiden, kann man die Numerierung der Kacheln, d.h. die logische Zeit des Prozessors p, mit $-p_r$ beginnen lassen.

[1]Wir gehen davon aus, daß die Numerierung der Prozessoren mit 0 beginnt

Dies hat zur Folge, daß hier zusätzliche *Dummy-Blöcke* generiert werden, in denen keine Berechnung stattfindet:

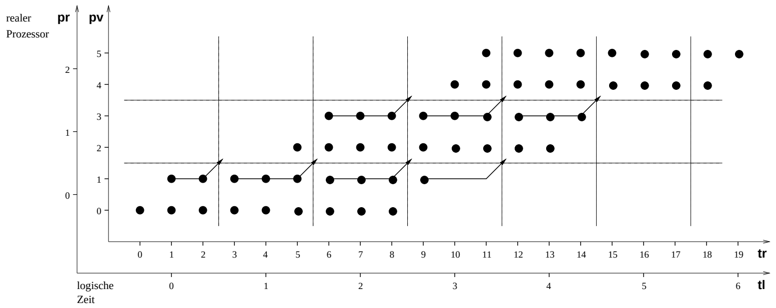

Abbildung 4.10: synchroner Fall, senderbezogen

Der Versatz (V) ist abhängig von der Kachelgröße und der Länge der Abhängigkeiten über alle in Betracht gezogenen Dimensionen (d), d.h. alle zusammenzufassenden Raum- und Zeitschleifen.

Er berechnet sich (für Prozessor p) durch:

$$V^d(p) = \min_d \left\lfloor p^d \frac{\text{Kachelgröße}}{min(\text{Länge der Abhängigkeiten in } p^d \text{ Richtung})} \right\rfloor \qquad (4.2)$$

Daraus ergibt sich auch, daß der Versatz nicht direkt von der Prozessornummer abhängt. Beträgt das Minimum der Abhängigkeiten mindestens das doppelte der Kachelbreite, so hat dies zur Folge, daß der Versatz nicht regelmäßig durchzuführen ist.

> Im folgenden Beispiel (Abbildung 4.11) haben so z.B. jeweils zwei Prozessoren den gleichen Versatz.

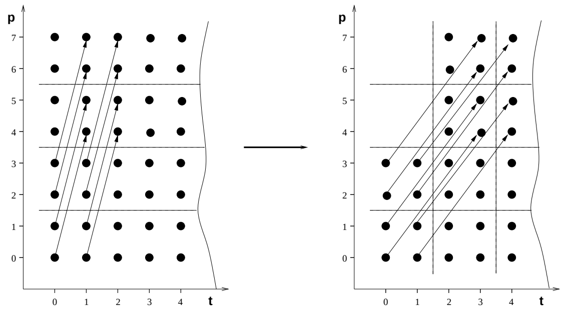

Abbildung 4.11: Verzögerung

Zu bemerken ist noch, daß, falls die Abhängigkeiten entgegen der Prozessornumerierung laufen, auch der Versatz entgegen der Prozessornumerierung geschieht.

Betrachtet man nun die Abhängigkeiten näher, so stellt man fest, daß mehrere Möglichkeiten existieren, die Datenpakete zu packen. Dies kann sender- oder empfängerbezogen geschehen.

- Im zuvor betrachteten synchronen Programm (Abbildung 4.10) wurden die Daten senderbezogen gepackt und damit alle Daten, die der Empfänger später einmal benötigen wird, am Ende aller Berechnungsschritte des Zeitblocks versandt.

  Dies hat gegenüber der folgenden Möglichkeit den Vorteil, daß das Versenden konsequent am Ende der Berechnung stattfinden kann. Es kann daher wie unter Abschnitt 4.3.2 beschrieben implementiert werden, was die Codeerzeugung vereinfacht.

- Wenn man die Daten empfängerbezogen zusammenpackt, so werden nur die Daten versandt, die der Empfänger im nächsten logischen Zeitschritt auch benötigt.

  Damit ergibt sich für unser Beispiel die folgende Struktur (Abbildung 4.12).

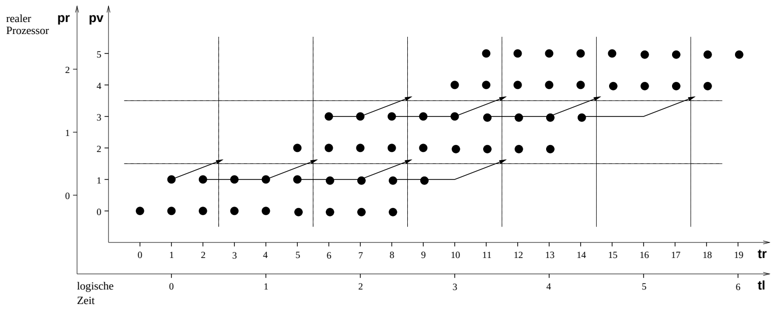

Abbildung 4.12: synchroner Fall, empfängerbezogen

Das Problem beim empfängerbezogenen Zusammenfassen ist, daß die Kommunikation nicht konsequent am Ende des Zeitblocks stattfinden kann. Je nach Struktur der Abhängigkeiten kann dies im Programm zu unterschiedlichen Zeitpunkten geschehen, was die Codeerzeugung verkompliziert.

Das empfängerbezogene Zusammenfassen kann dagegen das durch die Output-Abhängigkeiten induzierte Problem (siehe Abschnitt 4.3.3.5) beseitigen.

Asynchrone Programme:

In diesem Fall erhält man einige Lösungen frei Haus. Da hier die Prozessoren durch blockierendes Empfangen selbständig solange warten, bis sie alle zum Begin des Rechnens nötigen Daten empfangen haben, ist kein zusätzlicher Aufwand seites der Codeerzeugung nötig. Außerdem entfallen dadurch die Dummy-Blöcke und das Feld der Berechnungspunkte wird dichter (und damit auch schneller).

Im hier dargestellten Fall (Abbildung 4.13) wird empfängerbezogen zusammengefaßt, was den Vorteil bietet, daß die Berechnungen an den empfangenden Prozessoren früher beginnen können.

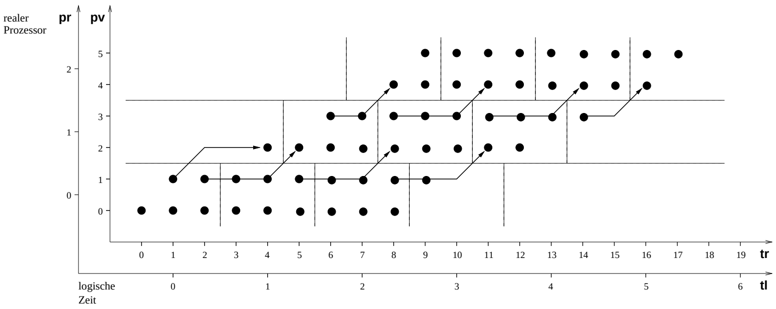

Abbildung 4.13: asynchroner Fall, empfängerbezogen

In der Realität kommt es jedoch auch im asynchronen Fall, bei der bevorzugten senderbezogenen Zusammfassung, zu einer Synchronisation. Da das Senden der Datenblöcke bis zum Berechnungsende der Kacheln verzögert wird, warten die empfangenden Prozessoren mindestens bis zum Versand des Datenpaketes. Die einzige verbleibende Asynchronität ist die, daß ein Prozessor weiterrechnen kann, sobald er alle seine Datenpakete erhalten hat.

Im Falle von synchronen Programmen werden also zusätzliche Blöcke eingefügt, in denen keine Berechnung stattfindet. Die Anzahl der zusätzlichen Verzögerungsblöcke je Prozessor muß dabei nicht konstant sein. Bei asynchronen Programmen findet die Verzögerung durch blockierende Kommunikationen statt.
In beiden Fällen besteht jeweils die Möglichkeit die zu versendenen Daten sender- oder empfängerbezogen zusammenzufassen. Das empfängerbezogene Zusammenfassen hat dabei einen Geschwindigkeitsvorteil in asynchronen Programmen und vermeidet das in Abschnitt 4.3.3.5 besprochendes Überschneidungsproblem durch Output-Abhängigkeiten. Allerdings kann dann das Versenden der Datenpakete nicht mehr konsequent am Ende der Berechnungen stattfinden, wodurch der zu generierende Code komplizierter wird.

Damit haben wir alle für das Verfahren nötigen Punkte betrachtet. Allerdings haben wir bezüglich der Gültigkeit einige zu starke Einschränkungen gemacht, die wir im nächsten Punkt näher betrachten wollen.

4.3.3.7 Verallgemeinerungen

Die oben angeführten Bedingungen zur Gültigkeit (siehe Abschnitt 4.3.3.3) sind hinreichend aber nicht notwendig. Sie schließen Fälle aus, in denen trotz zyklischer Abhängigkeiten ein Zusammenfassen der Zeit möglich wäre. Einige Fälle sollen hier näher betrachtet werden:

1. Häufig werden die Bedingungen nur an den Schleifenrändern verletzt. Dies kann beispielsweise während der Initialisierungsphase eines Algorithmus

der Fall sein. In diesem Fall können die Iterationen der Schleifenränder herausgezogen werden und das Verfahren für die übrigen Iterationen angewandt werden.

2. Es können *immer* soviel Berechnungspunkte zusammengefaßt werden, wie das Minimum aller Längen der Abhängigkeiten in der Zeitdimension groß ist. Bei guten Schedulern sollte allerdings die Länge mindestens einer Abhängigkeit in jeder Zeitdimension 1 betragen, da dieser die Punkte in der Zeit auslegt und versuchen sollten die Punkte möglichst dicht zu packen. Dadurch beträgt das Minimum 1, womit diese Verallgemeinerung dann nicht anwendbar ist.

3. Die Möglichkeit der Anwendung des Zeitblockens ist stark abhängig von der Richtung und der Länge der Abhängigkeiten. Ein Beispiel für einen durch die zu harten Einschränkungen ausgeschlossenen Fall ist in der folgenden Abbildung 4.14 dargestellt.

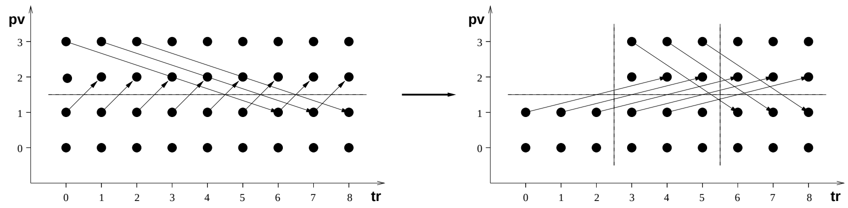

Abbildung 4.14: Ausgeschlossener Fall

Betrachtet man diesen Fall nun näher, so stellt man fest, daß das Zusammenfassen der Zeitschritte aufgrund der sehr langen Abhängigkeit in Rückrichtung ermöglicht wird.

Wir wollen hier nun versuchen die Grenzen der Gültigkeit zu erweitern, indem wir auch Abhängigkeiten in Rückrichtung zulassen. Dazu betrachten wir den Ablauf der Zeitpartitionierung näher und dessen Einfluß auf die Abhängigkeiten.

Das Verfahren der Zeitpartitionierung gliedert sich in zwei Phasen. Betrachtet wir nun das Verfahren nachdem die Größe der Zeitblöcke bereits feststeht, so erhält man die folgenden Phasen:

i. *Das Verschieben.*
Hierbei werden die Zeitblöcke soweit verschoben, daß alle Abhängigkeiten auch durch die gebündelten Kommunikationen erhalten bleiben.

ii. *Das Zusammenfassen von Berechnungschritten in der Zeit.*
Anschließend werden die Zeitschritte entsprechend der zuvor berechneten Anzahl zusammengefaßt.

Steht die Anzahl der zusammenzufassenden Zeitschritte noch nicht fest, so ergibt sich das Problem, daß die erste Phase des Verschiebens Einfluß auf die Anzahl der zusammenfaßbaren Zeitschritte nimmt. Die Phasen können damit nicht völlig getrennt voneinander betrachtet werden.

Nimmt man ganz grob die folgende Struktur der Berechnungsschritte und Abhängigkeiten an (Abbildung 4.15), wobei H die Abhängigkeiten in Richtung der Verschiebungen und R die Abhängigkeiten in Rückrichtung bezeichnen, dann sieht man wie das Verschieben Einfluß auf die Länge der Abhängigkeiten nimmt.

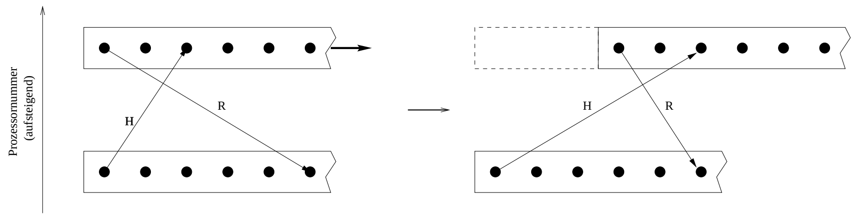

Abbildung 4.15: Verallgemeinerung: Auswirkung des Verschiebens

Durch das Verschieben werden die Abhängigkeiten in Verschieberichtung länger, während Abhängigkeiten in Rückrichtung kürzer werden.

Die unter Abschnitt 4.3.3.3 festgelegte Einschränkung besagt, daß Abhängigkeiten innerhalb einer Raumdimension nur eine Richtung haben dürfen. Diese Richtung entspricht der Verschieberichtung des Blocks, wodurch die Abhängigkeiten nur verlängert werden. Dies vermeidet, daß nach den beiden Phasen Abhängigkeiten entstehen können deren Zeitkomponente durch das Verkürzen (während der Phase des Verschiebens) Null oder negativ ist. Dies würde dann nämlich bedeuten, daß eine Kommunikation innerhalb eines logischen Zeitschrittes (oder sogar rückwärts in der Zeit) durchgeführt werden müßte, was durch die gebündelte Kommunikation nicht möglich ist.

Allgemein betrachtet: Wenn man Abhängigkeiten auch in Rückrichtung zuläßt, bedeutet dies, daß beim Anwenden des Verfahrens nach den zwei Phasen keine Abhängigkeiten entstehen dürfen, deren Zeitkomponente Null (oder kleiner) ist.

Existieren Abhängigkeiten nur in Verschieberichtung, so werden sie durch die Phase des Verschiebens nur verlängert und stellen daher kein Problem dar. In diesem Fall kann die Anzahl der zusammenzufassenden Zeitschritte frei gewählt werden.

In Abschnitt 4.3.3.5 gingen wir bereits auf die Anti-Abhängigkeiten in Rückrichtung ein. Diese hatte wir zwar in Abschnitt 4.3.3.3 ausgeschlossen, doch können wir diese unter Beachtung der Bedingungen des Abschnittes 4.3.3.5 zulassen. Tun wir dies, so wird die Blockgröße zusätzlich durch diese Bedingungen begrenzt.

Wir erweitern hier aber nun die Gültigkeit nicht nur um diese induzierten Abhängigkeiten, sondern generell. Damit beschränken wir die Blockgröße weiter.

Im generellen Fall beschränken dann alle Abhängigkeiten in Rückrichtung die maximale Blockbreite.

Das Minimum aller Abhängigkeiten in Rückrichtung (R_x) minus 1 ist das Maximum der möglichen Verschiebung:

$$V_{max} = min(|R_1|...|R_r|) - 1.$$

Dies resultiert aus dem Umstand, daß sich mit jedem verschobenen Berechnungsschritt die Länge der Abhängigkeit in der Zeit ebenfalls um 1 verringert, diese jedoch nicht 0 werden darf.

Wird maximal verschoben, so kann die Blockgröße anschließend nur noch 1 betragen, da sonst nach dem Zusammenfassen eine Abhängigkeit mit einer Zeitkomponente von 0 (oder negativ) entstünde.

Das folgende Beispiel (Abbildung 4.16) macht dies deutlich:

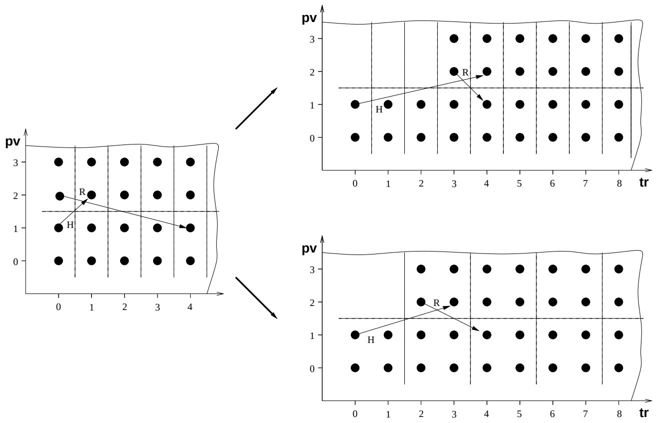

Abbildung 4.16: Beispiel für Auswirkungen auf Abhängigkeiten

Durch das Verschieben wird die Abhängigkeit in Richtung der Verschiebung (H) verlängert wohingegen die Abhängigkeit in Rückrichtung (R) verkürzt wird. Die Länge der Abhängigkeit in Richtung der Verschiebung beträgt 1 und die der Abhängigkeit in Rückrichtung 4. Wird nun in der ersten Phase um drei Zeitschritte verschoben (oberer Fall), so wird die Abhängigkeit in Rückrichtung soweit verkürzt, daß, um die Abhängigkeit zu erhalten, maximal ein Zeitschritt (also nicht) zusammengefaßt werden kann.

Da nun das Verfahren nicht nur aus dem Verschieben, sondern auch aus dem Zusammenfassen besteht, müssen in der Länge der Rückabhängigkeit in der Zeit beide Phasen (also das Verschieben **und** das Zusammenfassen) Platz finden.

Damit beträgt die maximale Blockgröße (aufgrund der Rückabhängigkeiten):

$$B_{max} = \left\lceil \frac{min(|R_1|...|R_r|)-1}{2} \right\rceil.$$

Für das obige Beispiel (Abbildung 4.16) bedeutet dies, daß die maximale Blockgröße 2 beträgt. Dann wird um zwei Zeitschritte verschoben und alle Abhängigkeiten bleiben erhalten (siehe unterer Fall).

Die bereits bestehenden Maximalgrenzen aus Abschnitt 4.3.3.5 (durch die Anti- und Output-Abhängigkeiten) bleiben hierdurch unberührt. Die endgültige Grenze stellt das Minimum aller bekannten Grenzen dar.

Es bleibt anzumerken, daß die eventuell durch die Rückabhängigkeiten induzierten Anti- und Output-Abhängigkeiten ebenfalls zu berücksichtigen sind.

Die erste der hier gemachten Verallgemeinerungen greift in die Codeerzeugung vor der Anwendung des Verfahrens ein und beseitigt das Verfahren behindernde Sonderfälle. Die zweite Verallgemeinerung betrifft Fälle, in denen der Scheduler schlechte Verteilungen liefert. In der dritten Verallgemeinerung wird das Problem der Zyklen näher betrachtet, welche durch Rückabhängigkeiten entstehen können. Die Konsequenzen hieraus sind eine weitere Einschränkung der möglichen Zeitblockgröße beim Zulassen beliebiger Abhängigkeiten.

4.3.4 Laufzeitabschätzungen

Betrachtet man nun das Verfahren insgesamt, so wird klar, daß wir das Partitionieren der Zeit nicht völlig umsonst bekommen. Den eingesparten Kommunikationsaufbaukosten stehen die verzögerten Startpunkte, eine größere Zahl logischer Zeitschritte und dadurch eine schlechtere Prozessornutzung gegenüber.

In diesem Abschnitt sollen daher die Laufzeiten der oben vorgestellen Verfahren abgeschätzt und miteinander verglichen werden.

4.3.4.1 Werte für das Kostenmodell

Wir nehmen die folgenden Kosten in das Kostenmodell auf:

- C_B Kosten für eine Berechnung (Multiplikation).
- C_U Kosten für das Umkopieren eines Double-Wertes in das zu versendende Datenpaket.
- C_A Aufbaukosten für eine Kommunikation.
- C_S Übertragungskosten eines einzelnen Double-Wertes.
- C_P Übertragungskosten eines Double-Wertes innerhalb eines Datenpakets.

Hierbei stellt $C_A + C_S$ die gesamten Kommunikationskosten für das Versenden eines einzelnen Double-Wertes dar.

Desweiteren haben wir noch folgende Werte, welche im Kostenmodell eine Rolle spielen:

- s Kachelbreite ($1 < s < min(...)$, siehe Abschnitt 4.3.3.5).
- t Anzahl reale Zeitschritte.
- p Anzahl virtuelle Prozessoren.
- p_r Anzahl reale Prozessoren.
- n Anzahl Berechnungschritte insgesamt (abhängig von der Problemgröße) und es gilt $n = t \cdot p$.
- a Anzahl der Statements welche Kommunikationen verursachen.
- k Volumen der abgehenden Kommunikationen je realem Zeitschritt. Dieses hängt von der Struktur der Abhängigkeiten ab und liegt im Bereich von $0 \leq k \leq \frac{p \cdot a}{p_r}$.
- f Anzahl der Zielprozessoren bezüglich der abgehenden Kommunikationen je realem Zeitschritt. Hängt wiederum von der Struktur der Abhängigkeiten ab und bewegt sich zwischen $0 \leq f \leq p_r - 1$.

Die Kachelbreite ist hier durch die in den Abschnitten 4.3.3.5 und 4.3.3.7 besprochenen Einschränkungen nach oben begrenzt.

Die Werte k, f und a sind stark vom Algorithmus, dessen Abhängigkeiten und der Dimensionalität der Allokation abhängig. Daher müssen diese abgeschätzt werden, wofür wir jetzt einige Verfahren aufzeigen wollen.

4.3.4.2 Abschätzung von k, f und a

Von der Genauigkeit der Abschätzung dieser Werte hängt die Genauigkeit der dann folgenden Kostenabschätzungen und die der Berechnung der optimalen Zeitblockgröße ab. Diese Werte beziehen sich auf einen Prozessor.

Abschätzung: k

Der Wert von k gibt das Volumen aller abgehenden Kommunikationen je realem Zeitschritt an. Dieser ist abhängig von der Struktur der Abhängigkeiten. Jede Abhängigkeit wird hier gesondert betrachtet und die einzelnen Volumina zu einem Gesamtvolumen aufaddiert:

$$k = \sum_{x=1}^{\#Abh.} k_x \tag{4.3}$$

Uniforme Abhängigkeiten besitzen eine feste Länge, weshalb auch viele der Abhängigkeiten komplett in einer Kachel liegen. Dadurch erzeugen nur die Abhängigkeiten an den Kachelrändern Kommunikationen. Auf der anderen Seite spielt dann natürlich auch – durch die Regelmäßigkeit der Abhängigkeiten entlang des Kachelrandes – die Länge des Randes eine Rolle. Desweiteren können längere Abhängigkeiten zu mehreren Kommunikationen über den Kachelrand führen, weshalb die Länge als Faktor in die Abschätzung einfließen muß.
Im uniformen Fall schätzen wir daher das Volumen (je Abhängigkeit) durch die Summe der Multiplikation jeder Oberfläche der Raumdimensionen, durch welche die Abhängigkeit geht, mit der Länge der Abhängigkeit in dieser Raumdimension ab.
Im nicht-uniformen Fall müssen wir davon ausgehen, daß Kommunikation von jedem Berechnungspunkt innerhalb der Kachel zu jedem Prozessor ausgehen kann. Um etwas genauer zu sein nehmen wir hierzu den abgeschätzten Quellraum (q) der jeweiligen Abhängigkeit, welcher für einen Prozessor im schlechtesten Fall $q = p_r \cdot \frac{p}{p_r}$ und im Fall ohne Rückabhängigkeiten $q = p_r \cdot \frac{p}{p_r \cdot 2}$ beträgt. Die Daten werden verteilt über die Zeit t übertragen, was dann in einer Abschätzung von $k_x = \frac{p}{2 \cdot t}$ je nicht-uniformer Abhängigkeit resultiert.

Abschätzung: f

Der Wert von f ist ebenfalls abhängig von der Struktur der Abhängigkeiten. Er gibt an, zu wieviel Prozessoren je realem Zeitschritt Kommunikationen durchschnittlich erfolgen. Wir unterscheiden auch hier uniforme und nicht-uniforme Abhängigkeiten, die aufgrund ihrer Eigenschaften jeweils einer anderen Abschätzung bedürfen.

Da wir die Anzahl der Prozessoren ermitteln wollen, die mögliche Ziele darstellen, betrachten wir im uniformen Fall nur die Abhängigkeiten, welche sich in den Raumkomponenten unterscheiden. Der Wert u gibt dann die Anzahl der Abhängigkeiten mit unterschiedlichen Raumkomponenten an. Da lange Abhängigkeiten bei kleinen Blockgrößen mehrere Kommunikationen initiieren können, wird diese Anzahl noch mit diesem möglichen Faktor multipliziert. Dieser berechnet sich aus $l_{\max} = \max(\text{Länge aller Abhängigkeiten})$ und der Blockgröße zu $\left\lceil \frac{l_{\max}}{\frac{p}{p_r}} \right\rceil$. Damit erhalten wir eine Abschätzung von $f = u \cdot \left\lceil \frac{l_{\max} \cdot p_r}{p} \right\rceil$.

Im nicht-uniformen Fall gehen wir davon aus, daß im schlechtesten Fall von jedem Prozessor zu allen anderen Prozessoren eine Kommunikation stattzufinden hat. Kommen keine Rückabhängigkeiten vor oder sind diese nicht zugelassen, so reduziert sich dieser Wert abhängig von der Dimensionalität der Allokation. Wir erhalten damit die folgenden Abhschätzungen: $f = \frac{p_r}{2^{dim(Allokation)}}$ bzw. beim Zulassen und Vorkommen von Rückabhängigkeiten $f = p_r - 1$.

In jedem Fall liegt die Anzahl der Kommunikationen zwischen 0 und $p_r - 1$, wobei letzteres bedeuten würde, daß in jedem Zeitschritt zwischen jedem Prozessor eine Kommunikation stattfände. Betrachtet man nun mehrere Abhängigkeiten (unabhängig davon, ob uniform oder nicht-uniform), so ist die geschätzte Obergrenze gegeben durch das Maximum.

Abschätzung: a

Der Wert von a kann relativ einfach abgeschätzt werden. Hier ist die Anzahl der Statements, welche Abhängigkeiten bezüglich unterschiedlicher Variablen besitzen, festzustellen.

4.3.4.3 Kostenmodelle

Die nachfolgend angegebenen Kosten beziehen sich jeweils auf einen Prozessor. Sie beziehen sich darüber hinaus relativ zu den Kosten des Polytop-Modells, so wie es idealerweise sein sollte.

Ohne Partitionierung

Zuerst betrachten wir die Kosten, wie sie durch das Modell ohne jede Partitionierung entstünden.
Da wir über eine theoretisch unendliche Anzahl von Prozessoren verfügen ($p = p_r$), können wir alle möglichen parallelen Berechnungspunkte auch wirklich parallel ausführen. Jeder Prozessor hat daher zu jedem der t Zeitpunkte nur eine Berechnung durchzuführen. Das Kommunikationsvolumen und damit die Anzahl der Aufbauvorgänge für die Kommunikationen hängen von den Abhängigkeiten ab. Die Anzahl der Kommunikationen kann zwischen 1 und $p - 1$ schwanken, je nach Abhängigkeit.

Zeitschritte (reale):	$T_o = t$
Anzahl benötigter Prozessoren:	$P_o = p$
Berechnungen insgesamt:	$B_o = t$
Berechnungen je Zeitschritt:	$E_o = 1$
Kommunikationsvolumen je Zeitschritt:	$K_o = f \cdot a$
Kommunikationsaufbauten je Zeitschritt:	$A_o = f \cdot a$

Da hier jeder Berechnungspunkt auf einem eigenen Prozessor liegt, führt zwangsläufig auch jede Abhängigkeit zu einer Kommunikation. Die Anzahl der abgehenden Kommunikationen hatten wir zuvor durch f abgeschätzt, wobei allerdings die Anzahl der Abhängigkeiten unberücksichtigt blieb. Deshalb nehmen wir hier a als Faktor hinzu und schätzen das Kommunikationvolumen mit $f \cdot a$ ab. Die Anzahl der Kommunikationsaufbauten entspricht in diesem Fall der Anzahl der versandten Daten. Dies ist der Grund, weshalb die Kommunikationskosten in diesem Fall so hoch sind.

Nur Raumpartitionierung

Bei Verwendung der Raumpartitionierung ergeben sich die folgenden Kosten:
Da wir in der Realität nur über eine endliche Anzahl an Prozessoren verfügen, ist dieses Modell das am häufigsten verwendete. Hierbei führt jeder der p_r Prozessoren $\frac{n}{t \cdot p_r} = \frac{p}{p_r}$ Berechnungen in jedem der t Zeitschritte des Modells aus, was ingesamt $\frac{n}{p_r}$ Berechnungen für jeden Prozessor ausmacht.
Da alle Kommunikationen einzeln ausgeführt werden, ist hier die Anzahl der zu übertragenen Daten ebenfalls mit der Anzahl der dafür nötigen Aufbauten identisch.

Zeitschritte (reale):	$T_r = t$
Anzahl benötigter Prozessoren:	$P_r = p_r$
Berechnungen insgesamt:	$B_r = \frac{n}{p_r}$
Berechnungen je Zeitschritt:	$E_r = \frac{p}{p_r}$
Kommunikationsvolumen je Zeitschritt:	$K_r = k$
Kommunikationsaufbauten je Zeitschritt:	$A_r = k$

Hierdurch kann eine Verbesserung des Berechnungs/Kommunikations-Verhältnisses von $1 : \frac{p}{p_r}$ erreicht werden.
In günstigen Fällen können Kommunikationen zwischen virtuellen Prozessoren, die nun auf denselben Prozessor fallen, eingespart werden, wodurch es zu einer Reduzierung des Kommunikationsvolumens kommt. Dies floß in die Abschätzung von k bereits ein. Allerdings erreicht man nur eine unwesentliche Reduzierung der Kosten, da je zu übertragendem Wert immer noch ein Kommunikationsaufbau stattfinden muß, wodurch dieses Verfahren hauptsächlich zur Vergröberung der Granularität dient.

Raumpartitionierung mit Kommunikationsbündelung

Wendet man nun die Raumpartitionierung zusammen mit der Kommunikationsbündelung an, so gelangt man zu der folgenden Kostenabschätzung, wobei hier der statische und dynamische Fall der Kommunikationsbündelung zu berücksichtigen sind.
Die Anzahl der Berechnungschritte, Zeitschritte, benötigten Prozessoren und des Kommunikationsvolumens sind die gleichen wie bei der alleinigen Raumpar-

titionierung. Unterschiede gibt es hier nur in der Anzahl der benötigen Aufbauvorgänge.
Die Zahl der Aufbauvorgänge haben wir zuvor durch f abgeschätzt, welches allerdings nur die Anzahl bezüglich der Prozessoren abschätzt. Beim statischen Zusammenfassen liegt die Anzahl höher, da hier für jedes Array eine separate Kommunikation nötig ist. Diesen Faktor haben wir hier durch a abgeschätzt, womit sich eine Abschätzung der Aufbauvorgänge von $f \cdot a$ ergibt.

Für den dynamischen Fall kann der Wert von f direkt übernommen werden, allerdings ist zusätzlich der Aufwand durch das Umkopieren der Daten einzuberechnen, welcher proportional zum Kommunikationsvolumen ist.

Zeitschritte (reale)		$T_b = t$
Anzahl benötigter Prozessoren:		$P_b = p_r$
Berechnungen insgesamt:		$B_b = \frac{n}{p_r}$
Berechnungen je Zeitschritt:		$E_b = \frac{p}{p_r}$
Kommunikationsvolumen je Zeitschritt:		$K_b = k$
Kommunikationsaufbauten je Zeitschritt:	statisch	$A_{b_s} = f \cdot a$
	dynamisch	$A_{b_d} = f$
Umkopiervorgänge je Zeitschritt:	statisch	$U_{b_s} = 0$
	dynamisch	$U_{b_d} = K_b$

Man erreicht hierdurch eine teilweise drastische Reduzierung der Kommunikationskosten. Dies wird durch die Einsparung von Aufbauvorgängen erreicht, deren Kosten, wie zuvor bereits belegt (siehe Tabellen 4.2 und 4.4), um ein Vielfaches höher liegen als die der Kommunikation.

Raum-/Zeitpartitionierung mit Kommunikationsbündelung

Verwendet man nun die Raum- und Zeitpartitionierung (zusammen mit der Kommunikationsbündelung), so gelangt man zu folgenden Kosten. Hierbei sind wieder das statische und dynamische Bündeln zu unterscheiden.
Das statische Bündeln unterscheidet sich vom dynamische hier wiederum nur durch den zusätzlichen Faktor bei den Kommunikationsaufbauten und durch den Wegfall der Umkopierkosten.

Zeitschritte (logische):		$T_z = \frac{t}{s}$
Anzahl benötigter Prozessoren:		$P_z = p_r$
Berechnungen insgesamt:		$B_z = \frac{n}{p_r}$
Berechnungen je Zeitschritt:		$E_z = s \cdot \frac{p}{p_r}$
Kommunikationsvolumen je Zeitschritt:		$K_z = s \cdot k$
Kommunikationsaufbauten je Zeitschritt:	statisch	$A_{z_s} = f \cdot a$
	dynamisch	$A_{z_d} = f$
Umkopiervorgänge je Zeitschritt:	statisch	$U_{z_s} = 0$
	dynamisch	$U_{z_d} = K_z$
Verzögerung (siehe Abschnitt 4.3.3.6):		$V_z = V(p_r)$

In beiden Fällen haben wir nun durch das Zusammenfassen von jeweils s Zeitschritten zu einer Kachel nur noch $\frac{t}{s}$ logische Zeitschritte. Damit erhöht sich jedoch ungekehrt die Anzahl der Berechnungsschitte je logischem Zeitschritt (Kachel) um s auf $s \cdot \frac{p}{p_r}$ und damit auch die Granularität. Das Kommunikationsvolumen bleibt mit k gleich, wobei hier eine eventuelle Ersparnis durch komplett in den Zeitschritt fallende Abhängigkeiten unberücksichtigt bleibt.
Eine Reduzierung ist bei den Kommunikationsaufbauten zu verzeichnen, welche abhängig von der Anzahl der zusammengefaßten Zeitschritte ist und nun insgesamt mit $T_z \cdot A_z = \frac{t}{s} \cdot f$ abgeschätzt ist.

Durch das Zusammenfassen der Zeitschritte kommt es, wie u.a. in Abschnitt 4.3.3.6 beschrieben, zu einer Verzögerung. Diese Verzögerung V_z wird gemäß Formel 4.2 abgeschätzt. Hierbei ist zu beachten, daß p_r die maximale Prozessornummer in Versatzrichtung und damit den maximal nötigen Versatz darstellt. Für viele Fälle genügt jedoch eine grobe Abschätzung mit $V_z = p_r$.
Für jeden dieser zusätzlichen Zeitschritte sind die kompletten Kosten für eine Kachel zu berechnen. Damit erhöht sich die folgende Gesamtkostenrechnung in diesem Kostenmodell um diese Kosten.

Gesamtkosten

Die einzelnen Kostenparameter fließen nun in die folgende Gesamtkostenrechnung ein. Hierbei sind alle in den vorigen Kostenmodellen nicht angegebenen Werte gleich 0 zu setzen.
Je logischem Zeitschritt (Kachel), welcher außer beim Zusammenfassen der Zeit, mit einem realen Zeitschritt identisch ist, erhalten wir so die folgenden Kosten:

$$G_{Kachel} = E \cdot C_B + K \cdot C_{S/P} + U \cdot C_U + A \cdot C_A \tag{4.4}$$

Die Gesamtkosten ergeben sich damit zu:

$$G = T \cdot G_{Kachel} + V \cdot G_{Kachel} \tag{4.5}$$

$C_{S/P}$ gibt hierbei die Kosten für das Versenden eines einzelnen Wertes bzw. beim dynamischen Bündeln das Versenden eines Wertes im Datenpaket an. Aufgrund von Messungen (siehe Abschnitt 4.1) kann man hier ein Faktor < 10 annehmen.

Die aufgrund der Struktur des parallelisierten Programms eventuell bestehende ungleichmäßige Verteilung der Berechnungspunkte auf die Prozessoren bleibt hierbei unberücksichtigt. Es wird lediglich der für das Programm schlechteste Fall berücksichtigt, wobei auch nur dieser für die Gesamtlaufzeit entscheidend ist.

4.3.4.4 Kostenvergleiche

Vergleicht man nun die Kosten vom reinen Modell zum realistischeren Ansatz mit Raumpartitionierung, so kommt es erwartungsgemäß im schlechtesten Fall, wo keine Kommunikationen eingespart werden können, zu einem Verhältnis von 1 : 1. Es kommt hier nur zu einer besseren Ausnutzung der Prozessoren, wie man beim Vergleich von E mit $1 : \frac{p}{p_r}$ sieht. Dieser Vergleich drückt auch die Vergröberung der Granularität aus.

Beim Vergleich der Raumpartitionierung mit der Raumpartitionierung mit zusätzlicher Kommunikationsbündelung stellt man nur eine Reduzierung der Aufbaukosten fest. Doch genau diese sollten ja auch wegen der hohen Kosten gesenkt werden. Wir haben hier ein Verhältnis von $k : f$ (bzw. $k : (f \cdot a)$ beim statischen Bündeln), was eine drastische Reduzierung und damit Beschleunigung darstellen kann (siehe auch die Tabellen 4.2 und 4.4).
Beim dynamischen Bündeln haben wir auf der einen Seite noch weniger Aufbauten, doch wird diese Reduzierung durch die zusätzlichen Kosten für das Umkopieren der Daten geschmälert.
Hier wurden beim statischen Bündeln die Kosten für die Berechnung der Blockgrenzen der zu versendenen Datenblöcke unterschlagen. Diese Kosten bewegen sich ungefähr im gleichen Rahmen, wie die Kosten für eine normale Berechnung (B). Im schlechtesten Fall kommt es zu einer geringen Verlangsamung. Durch den Umstand, daß die Aufbaukosten um bis zu einem Faktor 1000 höher liegen, hat man jedoch diese Kosten durch wenige eingesparte Aufbauvorgänge wieder kompensiert.

Wenn wir nun die zuvor entstandenen Kosten mit denen des um die Zeitpartitionierung erweiterten Modells vergleichen, so stellt man beim Betrachten der Werte E und K die weitere Vergröberung der Granularität um den Faktor s und beim Betrachten der Aufbauten (A) eine weitere Senkung fest.

In allen Fällen hängen die Kostenmodelle stark von einer guten Abschätzung der Wert für k, f und a ab. Sind diese ungenau abgeschätzt, so kommt es zu großen Verschiebungen innerhalb des Kostenmodells.
Ein Vergleich des Kostenmodells für die Raum-/Zeit-Partitionierung mit der Realität zeigte dann auch, daß das Modell bei guten Werten realistische Berechnungszeiten liefert (siehe Tabelle 4.7).

	100 × 100		75 × 75		50 × 50	
Proz.	real	Schätzung	real	Schätzung	real	Schätzung
3	126.0 s	109.9 s	62.9	42.2	21.0 s	21.9 s
6	82.5 s	81.2 s	42.6	41.1	16.6 s	19.2 s
12	70.5 s	71.9 s	39.2	40.3	18.2 s	21.1 s
16	69.0 s	69.9 s	41.0	41.0	19.6 s	22.1 s
32	81.4 s	74.6 s	54.3	47.7	32.6 s	27.8 s

Tabelle 4.7: Vergleich Kostenmodell - Messungen

Die hierbei einfließenden Werte für den LU-Algorithmus (welche gemessen wurden) betragen:

100×100: $k = 77$, $f = 3$ und $a = 2$.
75×75: $k = 54$, $f = 3$ und $a = 2$.
50×50: $k = 39$, $f = 3$ und $a = 2$.

4.3.4.5 Die optimale Zeitblockgröße

Man muß das folgende gegeneinander abwägen: Je mehr Zeitschritte man zusammenfaßt, desto mehr Aufbaukosten spart man. Auf der anderen Seite verlängert sich dadurch aber auch die Gesamtverzögerung.
Der entscheidende Faktor ist also die Größe des Zeitblocks: s. Die Verzögerung beträgt $V(p_r) \cdot G_{Kachel}$ und die eingesparten Aufbaukosten je Zeitblock $(s-1) \cdot C_A$.

Betrachtet man nun die Gesamtkostenrechnung (beim dynamischen Bündeln) näher, so erhält man die folgende Kostenabschätzung für die Gesamtlaufzeit:

$$\begin{aligned} G &= \frac{p}{pr} \cdot t \cdot C_B + t \cdot k \cdot (C_P + C_U) + \frac{t}{s} \cdot f \cdot C_A + \\ & \quad V(p_r) \cdot \left(s \cdot \left(\frac{p}{p_r} \cdot C_B + k \cdot (C_P + C_U) \right) + f \cdot C_A \right) \end{aligned} \tag{4.6}$$

Die Breite s des Zeitblocks, welche minimale Kosten verursacht, berechnet sich durch Ableitung der obigen Gleichung für die Gesamtkosten. Man erhält damit, im dynamischen Fall:

$$s_0 = \sqrt{\frac{t \cdot f \cdot C_A}{V(p_r) \cdot \left(\frac{p}{p_r} \cdot C_B + k \cdot (C_P + C_U) \right)}} \tag{4.7}$$

Und im statischen Fall:

$$s_0 = \sqrt{\frac{t \cdot f \cdot C_A}{V(p_r) \cdot \left(\frac{p}{p_r} \cdot C_B + k \cdot C_S \right)}} \tag{4.8}$$

Anstelle der realen Zeiten für C_x kann hier auch mit den Verhältnissen der Kosten (siehe Tabellen 4.2 und 4.4) gerechnet werden.

Die in diesem Abschnitt aufgestellten Kostenmodelle für die verschiedenen zuvor angesprochenen Partionierungsmethoden und der anschließende Vergleich führten uns zu einer Formel zur Ermittlung der optimalen Zeitblockgröße. Die Genauigkeit ist jedoch abhängig von der möglichst genauen Abschätzung der

Werte k, f und a. Für diese hatten wir Abschätzungen aufgeführt, welche es jedoch noch zu verbessern gilt und wofür eventuell noch sehr viel Detailarbeit nötig ist.
Die Berechnung der optimalen Zeitblockgröße ermöglicht es uns nun für das Verfahren einen Algorithmus anzugeben.

4.3.5 Der Algorithmus

Die vorherigen Abschnitte führen nun zu dem folgenden Algorithmus.

Die Eingabe besteht aus den einzelnen zu berücksichtigenden Kosten (siehe Abschnitt 4.3.4), allen transformierten Abhängigkeiten und der Prozessorenanzahl. Als Ausgabe liefert der Algorithmus entweder die Entscheidung, ob und wieviel in der Zeit zusammengefaßt werden kann oder den komplett transformierten Code.

Hauptalgorithmus

1. Fasse alle aufeinanderfolgenden Zeitschleifen mit konstanten Längen zusammen (siehe auch Abschnitt 4.3.3.4).

2. Sei $x = 1$ die innerste Zeitschleife und $v_0 = 0$.

3. Berechne die optimale Zeitblockbreite o. (Unteralgorithmus A)

4. Überprüfe, ob durch das Blocken der x innersten Zeitschleifen ein Zyklus entstehen würde. (Unteralgorithmus B)

   Ist dies der Fall, so setze $x = x - 1$ und gehe zu Punkt 7.

5. Berechne die aufgrund der Abhängigkeiten maximal mögliche Zeitblockbreite m der x-ten innersten Zeitschleife mit der Länge l. (Unteralgorithmus D)

   Ist die maximale Länge m kleiner als die Länge der Zeitschleife, dann gehe zu Punkt 7.

6. Berechne die maximale Zeitblockbreite (bzw. das Volumen) $v_x = v_{x-1} \cdot l$ aller x innersten Zeitschleifen unter Berücksichtigung der Abhängigkeiten. (Unteralgorithmus D)

   Wenn die optimale Zeitblockbreite o größer als die durch diese Dimension erreichbare Breite v_x ist, dann betrachte die $x+1$ innerste Zeitschleife und gehe zu Punkt 4.

   Wenn nicht, dann berechne die optimale Zeitblocklänge in der x. Dimension mit: $m = \left\lfloor \frac{o}{v_{x-1}} \right\rfloor$.

Hauptalgorithmus (Fortsetzung)

7. Falls $x > 0$, so wurde die maximale Zeitblockbreite oder das Optimum gefunden:

   > Fasse die x innersten Zeitschleifen zusammen und innerhalb der x-ten Zeitschleife m Zeitschritte.
   >
   > Falls die maximale Zeitblockbreite dem Optimum entspricht überspringe die folgende Kostenabschätzung und gehe zu Punkt 9.

   Falls nicht, so kann das Zeitblocken nicht durchgeführt werden. Man beschränkt sich dann im Punkt 9 auf das Zusammenfassen der Kommunikationen.

8. Überprüfe nun, ob die Laufzeit mit dieser Zeitblockgröße besser ist, als ohne. (Unteralgorithmus C)

   Wenn ja, dann führe Zeitblocken durch.

   Wenn nicht, dann beschränkt man sich auf das Zusammenfassen der Kommunikationen.

9. Erzeuge Code.

   (Je nach vorigem Ergebnis mit oder ohne dem Zusammenfassen der Zeit).

4.3.5.1 Unteralgorithmen

Der obige Algorithmus benötigt einige Hilfsfunktionen. Diese sollen hier separat aufgeführt werden, da einige wiederholt benötigt werden oder für Verallgemeinerungen (ohne den Grundalgorithmus zu berühren) austauschbar sind.

A) Optimale Zeitblockbreite

Eingabe:

- Anzahl Prozessoren
- Werte der benötigten Kosten: C_X

Ausgabe:

- Optimale Zeitblockbreite

Struktur:

1. Berechnen der optimalen Breite unter Verwendung der Formel 4.7.

B) Gültigkeit feststellen

Eingabe:

- Alle transformierten Fluß-Abhängigkeiten
- Liste aller zu berücksichtigenden Dimensionen

Ausgabe:

- Wahrheitswert (Boolean)

Struktur:

1. Berechne für alle Abhängigkeiten die Richtungsvektoren.
2. Für alle Raumdimensionen:
   (a) Speichere Wert der Richtung des ersten Richtungsvektors dieser Dimension.
   (b) Für alle weiteren Richtungsvektoren teste, ob sich Wert von dem gespeicherten Wert unterscheidet.
   Wenn ja, dann beende Algorithmus und gib *Unwahr* zurück.
3. Gib *Wahr* zurück.

C) Kostenabschätzung

Eingabe:

- Kachelbreite
- Anzahl Prozessoren
- Werte der benötigten Kosten: C_X

Ausgabe:

- Wert der geschätzten Kosten

Struktur:

1. Wenn Kachelbreite = 1, dann setze $V(p_r) = 0$ und gehe zu Punkt 3.
2. Berechne $V(p_r)$ nach Formel 4.2.
3. Berechnen der Kosten unter Verwendung der Formel 4.6.

D) Maximale Zeitblockgröße

Eingabe:

- Alle transformierten Output- und Anti-Abhängigkeiten
- Liste aller zu berücksichtigenden Zeitdimensionen

Ausgabe:

- Maximale Breite

Struktur:

1. Lösche alle Abhängigkeiten, die nicht nur in den übergebenen Zeitdimensionen liegen.
2. Lösche alle Output-Abhängigkeiten, deren Quelle und Ziel nicht auf demselben Prozessor liegen.
3. Kontrolliere für alle von Output-Abhängigkeiten abgehenden Fluß-Abhängigkeiten, ob deren Quelle und Ziel auf denselben Prozessor verweisen. Lösche alle Output-Abhängigkeiten, bei denen dies der Fall ist.
4. Lösche alle Anti-Abhängigkeiten, deren Quelle und Ziel auf demselben Prozessor liegen.
5. (Im dynamischen Fall: Lösche alle Output-Abhängigkeiten, deren abgehende Fluß-Abhängigkeiten auf unterschiedliche Prozessoren zeigen.)
6. Wenn keine Abhängigkeiten übrig, dann gib ∞ zurück.
7. Berechne für alle übrigen Abhängigkeiten deren Länge und suche das Minimum.
8. Gib gefundenes Minimum zurück.

Hier muß unterschieden werden, ob mögliche Überlappungen durch Anti- und Output-Abhängigkeiten (siehe Abschnitt 4.3.3.5) ausgeschlossen werden sollen oder nicht. Falls nicht, so ist die Code-Generierung entsprechend anzupassen.

4.3.5.2 Codegenerierung

Die Codegenerierung erzeugt ein Programm der folgenden Struktur:

```
for tb=ta to te step ts
  [Paket empfangen]
  for t=tb to tb+ts-1
    for pb=pa to pe step ps
      for p=pb to pb+ps-1
        [Statement]
        [Wert in Puffer kopieren]
      done
    done
  done
  [Kommunikation: Puffer versenden]
done
```

Hierbei sind (im dynamischen Fall) die bisher generierten Kommunikationsanweisungen durch Funktionsaufrufe auf die Umkopier-Funktionen zu ersetzen.
Im statischen Fall entfallen die zusätzlich generierten Anweisungen im Anschluß an die Statements völlig und es werden vor den Anweisungen zum Versenden der Datenpakete Anweisungen zum Berechnen der Grenzen generiert.
Anschließend sind am Beginn und Ende der äußersten zusammengefaßten Zeitschleife Kommunikationsanweisungen für das Versenden der Datenpakete zu generieren.

Der hier vorgestellte Algorithmus schließt Abhängigkeiten entgegen der Versatzrichtung völlig aus. Um den Algorithmus in dieser Hinsicht zu erweitern muß der Unteralgorithmus zum Feststellen der Gültigkeit entfernt und der Unteralgorithmus zur Berechnung der Zeitblockgröße um die in der Verallgemeinerung (Abschnitt 4.3.3.7) angegebenen Beschränkungen erweitert werden.

4.4 Beispielanwendung

In diesem Abschnitt wollen wir das Verfahren exemplarisch auf den LU-Algorithmus anwenden und durch Messungen an dem dabei entstehenden Code belegen, daß eine Beschleunigung zu erreichen ist.

4.4.1 Die Anwendung

Der transformierte Algorithmus enthält nur eine Zeitschleife, wodurch der gesamte Algorithmus nur einmal durchlaufen werden muß. Daher wollen wir zunächst die Gültigkeit des Zeitblockens feststellen.
Die Initialisierung des Algorithmus, welche durch die Statements S_1, S_2, S_3, S_6 und S_7 geschieht und nur zu zwei festen Zeitpunkten stattfindet, haben wir herausgezogen. Dadurch vermeiden wir Kommunikationen in diesem Randbereich, was Rechenzeit einspart.

Um nun die Gültigkeit festzustellen, haben wir die Richtungsvektoren der verbleibenden Abhängigkeiten zu betrachten. Dies sind im Falle der 2-dimensionalen Allokation:

$$\begin{aligned}\vec{r}_5 &= (+1,+1,0)\\ \vec{r}_7 &= (+1,+1,+1)\\ \vec{r}_{13} &= (+1,+1,+1)\\ \vec{r}_{15} &= (+1,+1,0)\\ \vec{r}_{18} &= (+1,0,+1)\end{aligned}$$

Und im Falle der 1-dimensionalen Allokation:

$$\begin{aligned}\vec{r}_7 &= (+1,+1)\\ \vec{r}_{15} &= (+1,+1)\end{aligned}$$

Der Test (B) der einzelnen Dimensionen der Richtungsvektoren verläuft positiv, d.h. die Vorzeichen je Dimension sind gleich und damit das Verfahren anwendbar.

Nun berechnen wir mit der Formel 4.7 die optimale Zeitblockbreite (A), welche 4 beträgt.

Für das Berechnen der maximalen Zeitblockbreite (D) benötigen wir alle Anti- und Output-Abhängigkeiten. Dies sind die Abhängigkeiten d_{19-24}.
Die Quellen und Ziele der Output-Abhängigkeiten liegen alle auf denselben Prozessoren. Durch diesen Test können also keine entfernt werden. Allerdings liegen alle möglichen abgehenden Fluß-Abhängigkeiten (dies sind für d_{22}: d_3 und d_9 sowie für d_{24}: d_{11} und d_{17}) auf demselben Prozessor, weshalb diese entfernt werden können.
Die beiden Anti-Abhängigkeiten können entfernt werden, da deren Quelle und Ziel auf demselben Prozessor liegen.
Damit bleiben keine zu berücksichtigenden Abhängigkeiten über, weshalb der Algorithmus ∞ zurückgibt und die Breite frei gewählt werden kann.
Die maximale Zeitblockbreite ist damit größer als die optimale Beite und wir können die optimale Kachelbreite verwenden.

Da wir die Blockbreite frei wählen können und sie damit optimal ist, brauchen wir die Kostenabschätzung (C) nicht durchzuführen. Es wird sofort der Code generiert.

4.4.2 Laufzeitergebnisse

Unter Anwendung dieses Verfahrens erhalten wir die im folgenden dargestellten Laufzeitergebnisse. Dazu vergleichen wir den LU-Algorithmus in mehreren Varianten. Wir wollen in diesem Abschnitt auch zeigen, daß die obigen Annahmen über die Laufzeit und die Beschleunigung zutreffen.
Fehlende Daten resultieren aus den schon angedeuteten Speicherproblemen der Parsytec.

4.4.2.1 2-dimensionale Allokation

Zuerst betrachten wir die Laufzeit des Verfahres im Vergleich mit den bereits bekannten Verfahren (Abbildung 4.17):

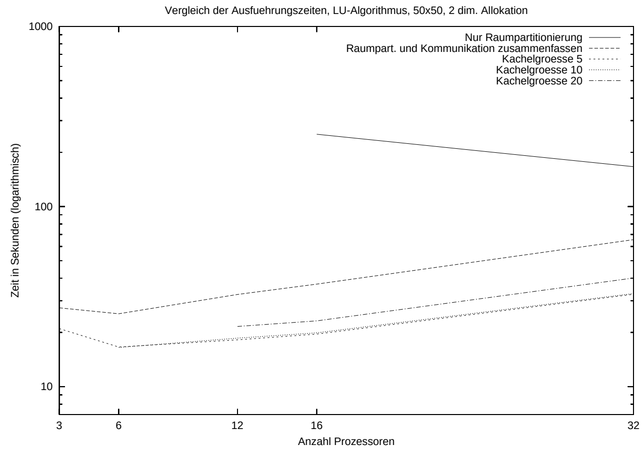

Abbildung 4.17: Vergleich der Verfahren mit der alleinigen Partitionierung (2-dimensionale Allokation)

Hier ist zu sehen, daß das Zusammenfassen der Kommunikationen (was hier dynamisch geschah) den größten Zeitgewinn brachte. Das Zusammenfassen mehrerer Zeitschritte brachte hier eine weitere Beschleunigung, wobei eine Zeitblockgröße von 5 das Optimum darstellt. Die optimale Anzahl der nötigten Prozessoren liegt bei dieser relativ kleinen Problemgröße von 50×50 bei 6 Prozessoren. Zu beachten ist hier der logarithmische Zeitmaßstab.

Interessant ist nun die Auswirkung der Zeitblockgröße und der Prozessorenanzahl auf die Laufzeit. Wie bei der kleineren Problemgröße der vorigen Messung ist hier eine optimale Prozessorenanzahl und eine optimale Zeitblockgröße bestimmbar.

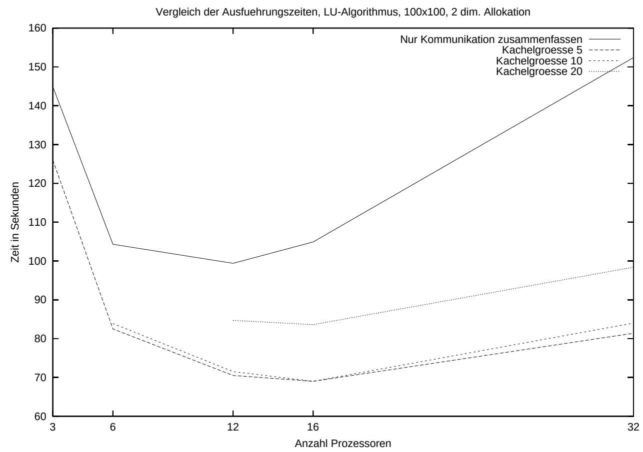

Abbildung 4.18: Optimale Prozessorenanzahl (2-dimensionale Allokation)

Allerdings liegt hier das Optimum aufgrund der anderen Problemgröße bei 16 Prozessoren (siehe Abbildung 4.18). Bei dieser Anzahl ist das Verhältnis von Rechenoperationen zu Kommunikationen optimal.
Wie die folgende Grafik (Abbildung 4.19) zeigt, liegt das Optimum der Blockgröße, unabhängig von der verwendeten Prozessorenanzahl, bei 5.

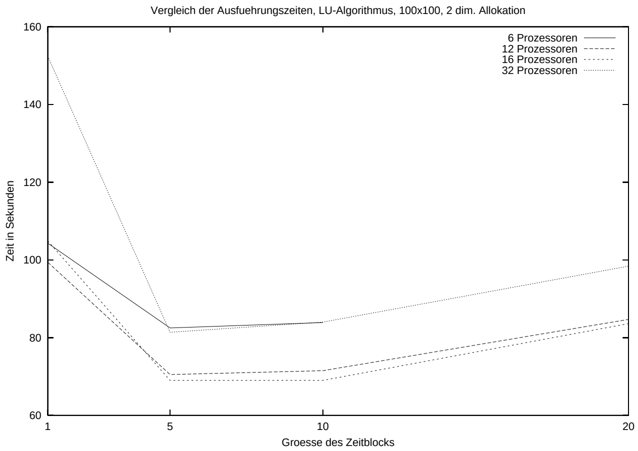

Abbildung 4.19: Optimale Blockgröße (2-dimensionale Allokation)

Der Anstieg der Rechenzeit bei größeren Blockgrößen resultiert aus der längeren nötigen Verzögerung.

Das die Verzögerung einen nicht unerheblichen Anteil an der Rechenzeit hat, macht die folgende Grafik (Abbildung 4.20) deutlich:

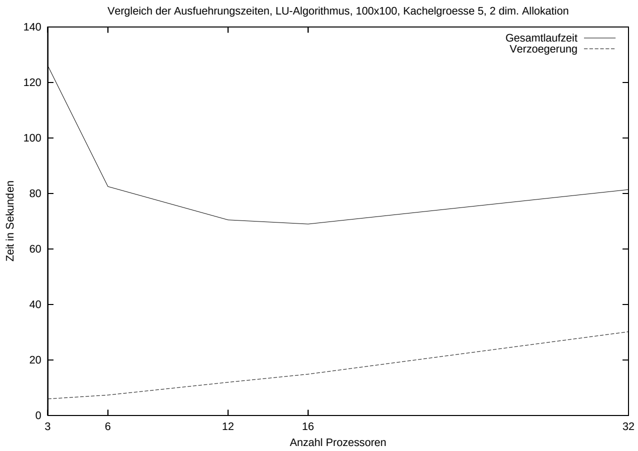

Abbildung 4.20: Anteil der Verzögerung an der Laufzeit (2-dimensionale Allokation)

Bei 32 Prozessoren, welches allerdings nicht der optimalen Prozessorenanzahl entspricht, mit einer Laufzeit von 81.4 s und einer Verzögerung von 30.2 s macht dies immerhin 37 % der gesamten Laufzeit aus.

4.4.2.2 1-dimensionale Allokation

Der 1-dimensionale Fall verhält sich sehr ähnlich. Auch hier ist bezüglich der Blockgröße und der Prozessorenanzahl ein Minimum bei 5 Zeitschritten bzw. 16 Prozessoren zu beobachten (siehe Abbildung 4.21).

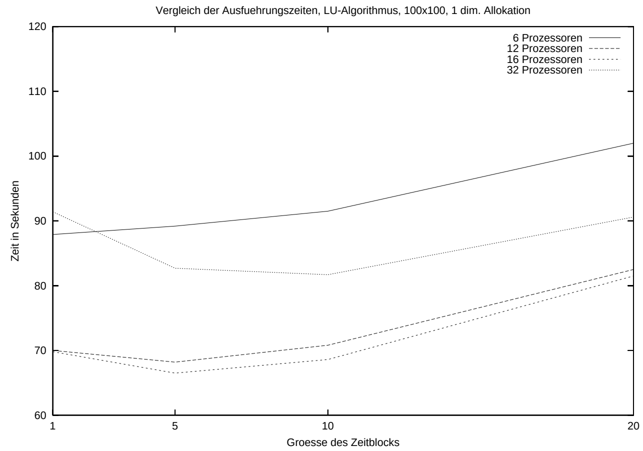

Abbildung 4.21: Optimale Blockgröße (1-dimensionale Allokation)

Bei sinkender Prozessorenanzahl macht sich das erhöhte Kommunikationsvolumen in einer steil ansteigenden Laufzeit bemerkbar (siehe Abbildung 4.22). Dies resultiert aus dem Mißverhältnis zwischen wenigen Prozessoren und viel Kommunikation.

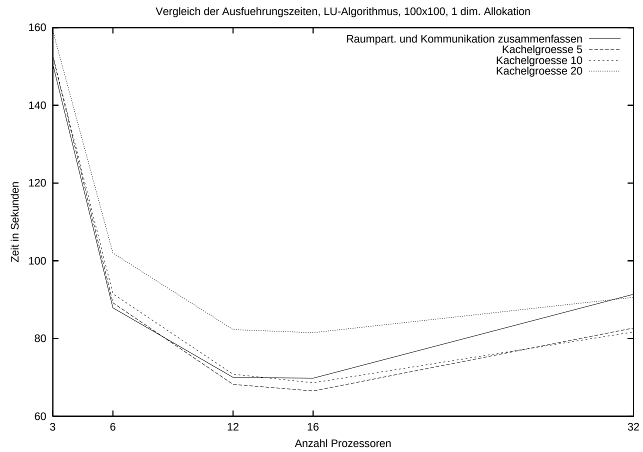

Abbildung 4.22: Optimale Prozessorenanzahl (1-dimensionale Allokation)

4.4.2.3 Vergleich

Vergleicht man nun die beiden Allokationen miteinander (Abbildung 4.23), so stellt man fest, daß sich die Laufzeiten nur gering voneinander unterscheiden.

Die Laufzeiten der 1-dimensionalen Allokation sind geringfügig besser, da hier das Kommunikationsvolumen geringer ist.

Der geringe Unterschied ist durch das Zusammenfassen der Kommunikationen zu erklären, wodurch die Anzahl der nötigen Kommunikationen (unabhängig von der Dimensionalität der Allokationen) in beiden Fällen gleich ist.

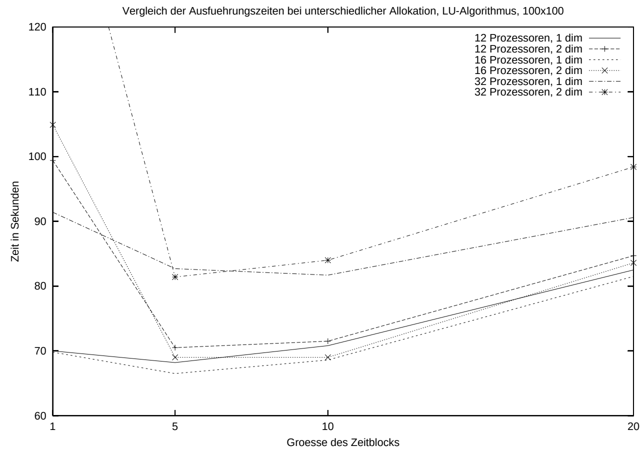

Abbildung 4.23: Vergleich der Allokationen

Dies bedeutet auch, daß beim Zusammenfassen von Kommunikationen die Kosten für die Kommunikationsaufbauten von der Zahl der Abhängigkeiten und der sonst nötigen Aufbauten (ein Aufbau je Abhängigkeit) entkoppelt wird.

4.4.2.4 Weitere Optimierungen

Die bisherigen Messungen belegen, daß durch die Verfahren eine Beschleunigung relativ zum bisherigen LooPo-Code möglich ist. Allerdings reichen diese alleine noch nicht aus, um eine tatsächliche Beschleunigung zu erreichen.
Am Code für die obigen Laufzeitmessungen, wurde bewußt auf Optimierungen per Hand verzichtet, um eine automatische Erzeugung zu simulieren und das Ergebnis mit dem bisherigen LooPo-Code vergleichbar zu machen.

Bei den folgenden Laufzeittest, welche exemplarisch im eindimensionalen Fall durchgeführt wurden, sind dagegen noch die folgenden Optimierungen zusätzlich von Hand durchgeführt worden. Diese Optimierungen orientieren sich an den in Abschnitt 4.2 besprochenen Möglichkeiten.

- *Entfernung aller Modulo-Berechnungen.*

  Durch die Modulo-Berechnungen kommt es zu einer Vervielfachnug der einzelnen Berechnungschritte. In dem Beispiel des LU-Algorithmus haben wir einen Modulowert von 3, was in einer dreimal höheren Gesamtanzahl an Zeitschritten resultiert.

- *Reduzierung der IF-Anweisungen.*

  Ein Großteil der Berechnungszeit geht, wie bereits in Abschnitt 3.2 besprochen, durch die Auswertung der IF-Anweisungen vor jedem Statement verloren. Die Anzahl der IF-Anweisungen wurde reduziert, indem die Schleifen in mehrere Schleifen aufgeteilt wurden, die jeweils nur die zu berechnenden Punkte aufzählen.

Beide Optimierungen stellen eine Verbesserung des Codes *vor* der Anwendung der Zeit-Partitionierung dar.
Die Implementation der Funktionen, welche zu den hier vogestellten Verfahren gehören wurden nicht weiter optimiert, so daß sie mit automatisch erzeugten Code vergleichbar bleiben. Dies bedeutet auf der anderen Seite aber auch, daß diese Funktionen noch optimiert werden könnten. Dazu sind jedoch intensive Tests (eventuell mit einem Profiler) und sehr viele Detailoptimierungen durchzuführen.
Außerdem hätte hier durch das statische Zusammenfassen eine weitere Beschleunigung erreicht werden können, aber der dynamische Fall wird in der Praxis wohl aufgrund seiner allgemeineren Stuktur (nur eine Kommunikation unabhängig von den Abhängigkeiten) und seiner unkomplizierteren Implementierung eher benutzt werden.

Betrachet man nun die Laufzeitmessungen dieser optimierten Versionen, so erhält man erstmals eine wahre Beschleunigung (siehe Abbildung 4.24).

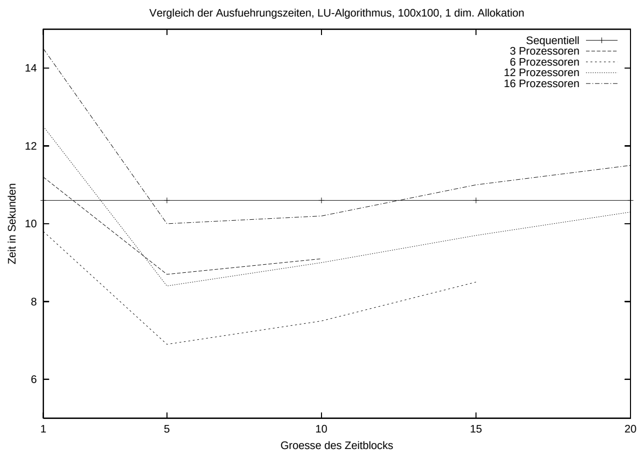

Abbildung 4.24: Optimale Blockgröße bei weiterer Optimierung (1-dimensionale Allokation)

Hier ist wieder die charakteristische Form der Kurve mit seinem Minimum bei einer Zeitblockgröße von 5 zu beobachten.

Betrachtet man nun die Laufzeiten auf unterschiedlichen Zahlen von Prozessoren, so ist zu sehen, daß das Optimum in diesem Fall bei 6 Prozessoren liegt (Abbildung 4.25).

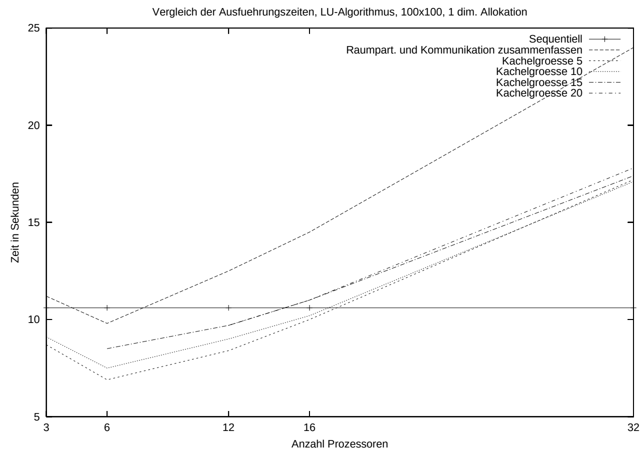

Abbildung 4.25: Optimale Prozessorenanzahl bei weiterer Optimierung (1-dimensionale Allokation)

Anhand dieser Grafik ist auch deutlich zu sehen, daß bereits das Zusammenfassen der Kommunikationen eine Beschleunigung bewirkt. Durch das Zusammenfassen der Zeitschritte und damit der Erhöhung der Granularität ist diese noch steigerbar.

Die folgenden Experimenten wurden auf einem Verbund von SUN Ultra 2 Sparcstations durchgeführt. Hier stand entsprechend mehr Speicher zur Verfügung, so daß eine Matrixgröße von 1000 berechnet werden konnte. Den Messungenauigkeiten wurden versucht durch Mittelung mehrerer Messungen entgegenzuwirken. In der den folgenden Grafik (Abbildung 4.26) wird deutlich, daß bei größeren Problemen ein höhere Beschleunigung (siehe auch Abbildung 4.27) zu erreichen ist.

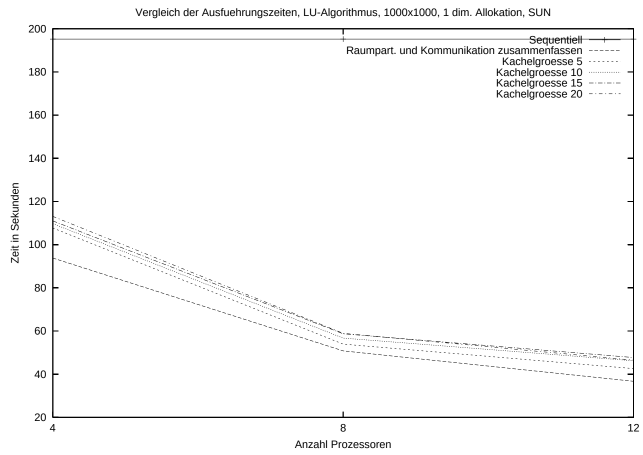

Abbildung 4.26: Laufzeiten auf SUNs bei größerem Problem (1000×1000)

Die bessere Laufzeit bei kleinerer Kachelgröße ist dadurch zu erklären, daß hier durch das Zusammenfassen von Zeitschritten das Verhältnis von Kommunika-

tionen zu Rechenschritten nicht mehr verbessert werden kann. Hier wird also die Rechenzeit nur durch das Bündeln der Kommunikationen verbessert.
Bei weiter steigender Prozessorenanzahl könnte jedoch durch das Zusammenfassen der Zeitschritte wieder Zeit eingespart werden.

Der Beschleunigungsfaktor liegt hier (Abbildung 4.27) je nach verwendeter Prozessoranzahl zwischen 1.7 und 5.3.

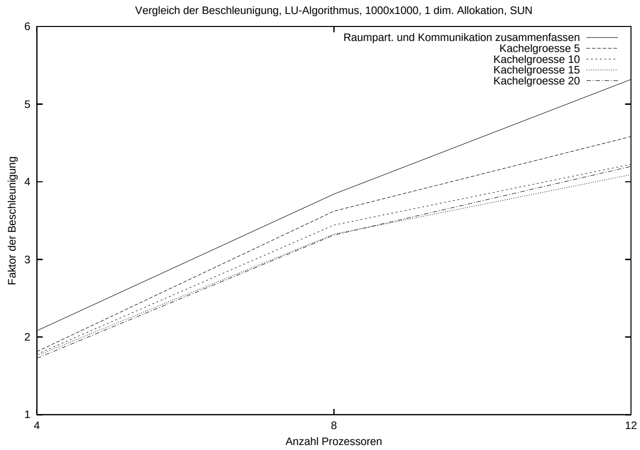

Abbildung 4.27: Mögliche Beschleunigung auf SUNs

Hier wird deutlich, daß viel unnötige Rechenzeit durch unoptimierten, automatisch erzeugten Code verloren geht und erst durch die Kombination der Verfahren auch eine tatsächliche Beschleunigung im Vergleich zum sequentiellen Fall zu erreichen ist.

4.4.3 Details zu den Experimenten

Ein gewaltiges Problem stellte der geringe Speicher der Parsytec dar. Da durch das dynamische Zusammenfassen zusätzlicher Speicher für die Puffer benötigt wurde, war es damit unmöglich die Tests mit größeren Problemgrößen als 200 $\times$ 200 durchzuführen. Häufig stellte sogar schon eine Problemgröße von 100 $\times$ 100 ein Problem dar.
Im Falle der Partitionierung ohne Zusammenfassen äußerte sich dieses Problem während der Berechnung, als sich im Verlauf noch nicht empfangene Kommunikationen anhäuften.

Im Laufe der Implementation war ein optimaler Zeitpunkt für das Zusammenfassen der zu versendenden Daten zu finden. Hier gab es zwei Möglichkeiten:

1. Erneutes Aufzählen in separate(r/n) Schleife(n) und langsames Umkopieren in den zu versendenden Block. Nachdem ein Datenblock für einen Prozessor gefüllt ist wird dieser versandt und der Speicher wird wieder

freigegeben. Dabei werden in dieser Schleife erneut die Abhängigkeiten für den aktuellen Zielprozessor überprüft.

Nachteil: Zusätzliche Schleifen und unnötige erneute Überprüfung der Abhängigkeiten.

Vorteil: Klar abgegrenzter Block für die Kommunikation. Dadurch ergibt sich ein geringerer Pufferbedarf.

Das Problem bei dieser Lösung in Verbindung mit der `MPI`-Bibliothek ist, daß der Speicher, wenn mit nicht blockierender Kommunikation gearbeitet wird, erst dann wiederverwendet werden kann, wenn die Daten komplett empfangen wurden. Wird dies nicht beachtet, so kommt es infolge von Überschneidungen zu Datenmüll.

2. Einbetten des Kopierens in das Programm. Dabei wird hinter jedes berechnete Statement (bzw. jeden möglichen Kommuikationskandidaten, der sich aus den Abhängigkeiten ergibt) eine Funktion zum Kopieren des berechnetet Datums in einen Kommuikationspuffer für den jeweiligen Zielprozessor eingefügt. Anschließend wird noch ein Statement für das Erhöhen des Kommuikationspufferzählers eingefügt.

   Nachteil: Verstreute Teile der Kommunikation zwischen den einzelnen Statements des Programms. Dadurch wird ein erhöhter Pufferbedarf nötig.

   Vorteil: Einsparung der doppelten Prüfung der Abhängigkeiten und zusätzlicher Schleifen. Das Programm wird dadurch schneller.

   Als weiterer Vorteil stellte sich im Laufe der Arbeit heraus, daß hierdurch das Überschneidungsproblem durch Output-Abhängigkeiten (siehe Abschnitt 4.3.3.5) vermieden werden kann.

Die Größe des Puffers ist abhängig von der Problemgröße und den Abhängigkeiten. Im zweiten Fall ist die Prozessorenanzahl ein Multiplikator.
Für die Tests wurde die zweite Möglichkeit implementiert, was zu komplizierterem, aber schnellerem Code führte.

Die nächste Designentscheidung betraf das Empfangen der Daten. Hier bestand die Möglichkeit, das Empfangen vor den einzelnen Statements oder in einem separaten Block vor allen Berechnungen durchzuführen. Das Problem ist allerdings, daß die Versendereihenfolge nicht unbedingt mit der Empfangsreihenfolge übereinstimmt, d.h. die Reihenfolgen der Daten im Puffer nicht gleich sind.

Dasselbe trifft auch auf den Fall zu, wo nur ein Datenpaket je Prozessor versandt wird. Hier wäre das Unterteilen des Puffer in unterschiedliche Bereiche für die Abhängigkeiten eine Idee. Dies allein genügt allerdings auch nicht, da es auch hierbei zu Reihenfolgeunterschieden kommen kann.
Damit blieb nur der Ausweg, den Empfang komplett in einen Block zusammenzufassen und das genaue Ziel (im Array) eines jeden Datums mitzuübertragen.

Die dabei versandten Datenpakete bestanden aus den folgenden Elementen:

1. Nummer der Abhängigkeit
2. Ziel der Abhängigkeit (Arrayposition)
3. Wert

Falls es zu Überschneidungen durch Anti-Abhängigkeiten kommt, so wäre noch ein zusätzliches Element für den Zeitpunkt nötig. Dies war jedoch hier nicht der Fall.

Das letzte Problem war die Übertragung der Blockgrößen. Um das Kommunikationsvolumen möglichst gering zu halten, sollten nur die tatsächlich gefüllten Teile der Puffer übertragen werden. Dazu muß der empfangende Prozessor jedoch die Größe des versendeten Datenblocks wissen. Da in LooPo bisher die dazu nötigen Berechnungen fehlten, wurden zwischen allen Prozessoren (vor dem eigentlichen Versenden der Datenblöcke) die Größen ausgetauscht. Dies geschah mit Hilfe der `MPI`-Funktion `MPI_Allgather` und stellt ein Broadcast zwischen allen Prozessoren dar. Das dabei entstehende Kommunikationsvolumen ist $p_r^2 \cdot sizeof(integer)$.
Das Problem, welches daraus entsteht, ist eine Synchronisation aller Prozessoren an diesem Programmschritt. Dies führt zu einer geringfügigen Verlangsamung wie die folgende Grafik (Abbildung 4.28) zeigt.

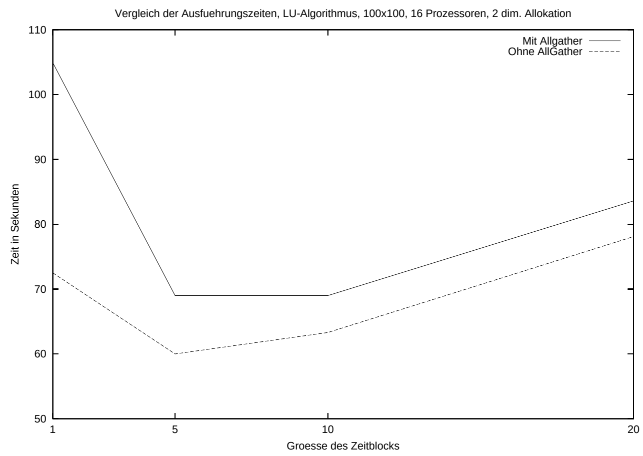

Abbildung 4.28: Auswirkungen von `MPI_Allgather` auf die Laufzeit

Auf die Verwendung von `MPI_Allgather` wurde in den mit der Hand optimierten Versionen verzichtet.

Kapitel 5

Zusammenfassung

Aufbauend auf dem durch den Schleifenparallelisierer LooPo generierten Code, wurden zwei Verfahren zur Reduzierung der Kommunikationkosten von automatisch generierten Programmen entwickelt.

Bei der Analyse des von LooPo generierten Codes für den exemplarisch verwendeten LU-Algorithmus stellten wir fest, daß neben der noch optimierbaren Codegenerierung – es wurden viele sehr lange IF-Statements erzeugt und der Code linearisiert – die Kommunikationskosten den größten Anteil an der Laufzeit darstellte. Dies bestätigten Messungen auf realen Maschinen.
Nach dem Betrachten der bereits bestehenden, in LooPo implementierten Verfahren der Raumpartitionierung stellten wir hier nur eine Vergröberung der Granularität fest, ohne eine nennenswerte Reduzierung der Kommunikation zu erreichen. Dies führte bei der alleinigen Raumpartitonierung zu einer linearen Verlangsamung bei Erhöhung der Prozessoranzahl.

Um nun die Kommunikationen zu reduzieren stellten wir ein Verfahren vor, welches auf Grundlage des transformierten Programms einzelne Kommunikationen zu großen Datenpaketen zusammenfaßt. Dabei wird zwischen dem Verfahren zum statischen und dynamischen Bündeln unterschieden. Hierbei hatte das dynamische Bündeln die niedrigeren Kommunikationskosten, wobei ein zusätzlicher Aufwand durch das Umkopieren von Daten nötig ist. Die hierdurch erreichte Einsparung der Kommunikation reicht aus, um die parallelisierten Programme deutlich zu beschleunigen.

Im zweiten entwickelten Verfahren, welches auf dem ersten aufbaut, werden zusätzlich mehrere Zeitschritte zusammengefaßt. Dadurch erreicht man eine weitere Vergröberung der Granularität und eine nochmalige Reduzierung der Kommunikationen. Allerdings ist das Zusammenfassen der Zeitschritte mit Restriktionen versehen. So muß grundsätzlich vermieden werden, innerhalb eines Zeitschrittes zwischen den rechteckigen Kacheln zyklische Abhängigkeiten zu erzeugen, was durch eine Einschränkung der zugelassenen Abhängigkeiten erreicht wird. Hierbei werden Rückabhängigkeiten verboten, wodurch ein Zyklus niemals entstehen kann.

Die Überschneidungsprobleme, welche durch das gebündelte Versenden und das Zusammenfassen der Zeitschritte entstehen können, wurden gelöst. Die Lösungen bestehen in einer veränderten Codegenerierung oder einer Beschränkung der Zeitblockgröße. Danach wurden weitere Auswirkungen auf das Programm betrachtet. Eine Auswirkung auf den zeitlichen Ablauf ist die Verzögerung von Zeitschritten, welche für die Erhaltung der Abhängigkeiten nötig ist. Eine weitere Auswirkung ist die Möglichkeit die Daten abhängig vom Sender oder Empfänger zu bündeln. Dies wurde ebenfalls näher betrachtet.

Später wurde versucht die sehr harten Restriktionen bezüglich der zugelassenen Abhängigkeiten zu lockern und Rückabhängigkeiten wieder zuzulassen. Dies führte zu einem Modell in dem alle Abhängigkeiten zugelassen sind und ausschließlich die Anzahl der zusammenfaßbaren Zeitschritte begrenzt ist.
Anschließend wurden für die Verfahren Kostenmodelle aufgestellt und verglichen. Der vorgestelle Algorithmus, welcher vom restriktiveren Modell zur Vermeidung von Zyklen ausging, ermöglichte die Überprüfung der Verfahren durch Experimente. Durch diese konnte bewiesen werden, daß durch die Verfahren eine drastische Senkung der Kommunikationskosten und damit der Laufzeit erreicht werden kann. Die Verbesserung der Laufzeiten durch die Verfahren allein reichten jedoch nicht aus, um eine Beschleunigung zum sequentiellen Programm zu erreichen. Es wurde jedoch gezeigt, daß die Verfahren in Kombination mit den anderen bezüglich der Codeoptimierung angesprochenen Verfahren eine Beschleunigung zum sequentiellen Programm erreichen können.

Allerdings stellen die Verfahren große Anforderungen an eine effizente Implementation, da sonst der Gewinn durch eine schlechte Codegenerierung wieder zunichte gemacht wird. Außerdem müssen die in die Kostenmodelle einfließenden Parameter k, f und a möglichst genau abgeschätzt werden, um eine qualitativ gute Kostenabschätzung geben zu können. Diese sind auch bezüglich der optimalen Zeitblockbreite wichtig und stellen einen Punkt dar, der noch mit viel Detailarbeit verbessert werden kann. Ebenso kann der Algorithmus noch um die Verallgemeinerungen der Rückabhängigkeiten erweitert werden.

Literaturverzeichnis

[1] Martin Griebl und andere: *LooPo*, ein Project des Lehrstuhl für Programmierung an der Fakultät für Mathematik und Informatik der Universität Passau, `http://www.infosun.fmi.uni-passau.de/cl/loopo/`, seit Juni 1994.

[2] Christian Lengauer: *Loop parallelization in the polytope model*, In E. Best, Hrsg., CONCUR'93, LNCS 715, Seite 398ff, Springer-Verlag, 1993.

[3] Martin Griebl: *The Mechanical Parallelization of Loop Nests Containing* `while` *Loops*, Dissertion, MIP-9701, Universität Passau, Oktober 1996.

[4] Utpal Banerjee: *Dependence Analysis*, Kluwer Academic Publishers, Boston/Dordrecht/London, 1996.

[5] Paul Feautrier: *Dataflow analysis of array and scalar references*, In International Journal of Parallel Programming, Band 20, Seite 23ff, 1991.

[6] Michael J. Quinn: *Parallel Computing - Theory and Practice*, 2. Auflage, McGRAW-HILL Inc., Seite 73, 1994.

[7] High Performance Fortran Forum: *High Performance Fortran Language Specification*, Rice University, Houston, TX, 1997.

[8] Martina Wieninger: *Partitionierung von parallelen Schleifenprogrammen*, Diplomarbeit, Universität Passau, September 1997.

[9] Peter Faber: *Transformation von Shared-Memory-Programmen zu Distributed-Memory-Programmen*, Diplomarbeit, Universität Passau, November 1997.

[10] Harald Keimer: *Datenabhängigkeitsanalyse zur Schleifenparallelisierung: Vergleich und Erweiterung zweier Ansätze*, Diplomarbeit, Universität Passau, Januar 1997.

[11] Wolfgang Meisl: *Practical Methods for Scheduling and Allocation in the Polytope Model*, Diplomarbeit, Universität Passau, September 1996.

[12] Paul Feautrier: *Some efficient solutions to the affine scheduling problem, Part I, One-Dimensional Time*, International Journal of Parallel Programming, Band 21, Nummer 5, Seite 313ff, 1992.

[13] PAUL FEAUTRIER: *Some efficient solutions to the affine scheduling problem, Part II, Multidimensional Time*, International Journal of Parallel Programming, Band 21, Nummer 6, Seite 389ff, 1992.

[14] SANJAY RAJOPADHYE UND R. ANDONOV: *Optimal orthogonal tiling of 2-d iterations*, J. Parallel and Distributed Computing, Seite 159ff, 1997.

[15] R. ANDONOV UND SANJAY RAJOPADHYE: *Optimal orthogonal tiling*, Euro-Par'98: Parallel Processing, LNCS 1470, Seite 480ff, Springer-Verlag, 1998.

[16] P. BOULET, A.DARTE, T.RISSET AND Y.ROBERT: *(Pen-)ultimate tiling?*, INTEGRATION, 17:33-51, 1994.

[17] E. HODŽIĆ AND W. SHANG.: *On time optimal supernode shape*, Eighth Int. Workshop on Compilers for Parallel Computers (CPC2000), Seiten 367-379, 2000.

[18] K. HÖGSTEDT, L.CARTER AND J.FERRANTE: *Selecting tile shape for minimal execution time*, Proc. 11th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '99), ACM Press, Juni 1999.

[19] H. OTHA, Y.SAITO, M.KAINAGA AND H.ONO: *Optimal tile size adjustment in compiling general DOACROSS loop nests*, Proc. 1995 Int. Conference on Supercomputing, ACM Press, Juli 1995.

[20] J.XUE: *Comunication-minimal tiling of uniform dependence loops*, J. Parallel and Distributed Computing, 42(1):42-59, April 1997.

[21] RADKO KUBIAS: *Array Datenflußanalyse und Single-Assignment-Konversion*, Diplomarbeit, Universität Passau, September 1996.

[22] MARTIN GRIEBL: *On the Mechanical Tiling of Space-Time Mapped or Imperfect Loop Nests*, Paper von 1999/2000, noch nicht erschienen.

[23] PETER S. PACHECO: *Parallel Programming with MPI*, Morgan Kaufmann Publishers, Inc., San Francisco, 1997.

[24] FABIEN QUERILLÉ, SANJAY RAJOPADHYE AND DORAN WILDE: *Generation of efficient nested loops from Polyhedra*, Kluwer Academic Press, Boston/Dordrecht/London (bisher nicht erschienen), Paper von 1999.

[25] OLIVER NYDERLE: *Verfahren zur Aufzählung von Polyedern in der Schleifenparallelisierung*, Diplomarbeit, Universität Passau, 2000.

[26] PARSYTEC COMPUTER GMBH: Parsytec Dokumentation, März 1993, Aachen, 3. Auflage.

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang A

Quellcodes

A.1 Original LooPo Code

```
/* LooPo program
**      C
*/

int n;
double ****L=NULL;
double ****A=NULL;
double ****U=NULL;
double ****SUM=NULL;
double ****SUMM=NULL;
void LooPo_PRG0(void)
 {
  int t1;
  int p1;
  int p2;
  LooPo_beginSection(2,1+((int)floor(LooPo_doubleMax(2,((double)(n)),
                        ((double)(3+t1))/3))),1+LooPo_intMax(2,0,n));
  if(LOOPO_SECTION_OK)
   {
    LooPo_initializeRTS(LOOPO_TRUE,LOOPO_FALSE);
    for(t1=0;t1<=LooPo_intMax(2,1,3*n-3);t1++)
     {
      p1=LooPo_parBegin(((int)ceil(LooPo_doubleMin(2,((double)(1)),
                       ((double)((2+t1)))/3))),((int)floor(LooPo_doubleMax(2,
                        ((double)(n)),((double)((3+t1)))/3))));

      if(LOOPO_PARFOR_OK)
       {
        p2=LooPo_parBegin(((int)ceil(LooPo_doubleMin(4,((double)(0)),((double)(t1)),
                         ((double)(p1)),((double)((3+t1)))/3))),LooPo_intMax(2,0,n));
        if(LOOPO_PARFOR_OK)
         {
          if(0 <= t1 && t1 <= 0 && 1 <= p1 && p1 <= n && 0 <= p2 && p2 <= 0)
           {
            L[p1][1]=A[p1][1];
           }
          if(1 <= t1 && t1 <= 1 && 1 <= p1 && p1 <= n && 0 <= p2 && p2 <= 0)
           {
```

```
          U[1][p1]=A[1][p1]/L[1][1];
         }
        if(0 <= t1 && t1 <= 0 && 2 <= p1 && p1 <= n && t1 <= p2 && p2 <= n)
         {
          SUM[p2][p1]=0;
         }
        if(2 <= t1 && t1 <= 3*n-4 && LooPo_mod((t1-2),3) == 0 &&
           ((double)((4+t1)))/3 <= p1 && p1 <= n && p1 <= p2 && p2 <= n)
         {
          SUM[p2][p1] = SUM[p2][p1] + L[p2][((int)floor(((double)((1+t1)))/3))] *
                        U[((int)floor(((double)((1+t1)))/3))][p1];
         }
        if(3 <= t1 && t1 <= 3*n-3 && LooPo_mod((t1-3),3) == 0 &&
           ((double)((3+t1)))/3 <= p1 && p1 <= ((double)((3+t1)))/3 &&
            ((double)((3+t1)))/3 <= p2 && p2 <= n)
         {
          L[p2][((int)floor(((double)((3+t1)))/3))] =
            A[p2][((int)floor(((double)((3+t1)))/3))] -
             SUM[p2][((int)floor(((double)((3+t1)))/3))];
         }
        if(0 <= t1 && t1 <= 0 && 2 <= p1 && p1 <= n && 0 <= p2 && p2 <= 0)
         {
          U[p1][p1]=1;
         }
        if(0 <= t1 && t1 <= 0 && 2 <= p1 && p1 <= n-1 && 1+t1 <= p2 && p2 <= n)
         {
          SUMM[p1][p2]=0;
         }
        if(2 <= t1 && t1 <= 3*n-7 && LooPo_mod((t1-2),3) == 0 &&
           ((double)((4+t1)))/3 <= p1 && p1 <= n-1 && 1+p1 <= p2 && p2 <= n)
         {
          SUMM[p1][p2] = SUMM[p1][p2] + L[p1][((int)floor(((double)((1+t1)))/3))] *
                         U[((int)floor(((double)((1+t1)))/3))][p2];
         }
        if(4 <= t1 && t1 <= 3*n-5 && LooPo_mod((t1-4),3) == 0 &&
           ((double)((2+t1)))/3 <= p1 && p1 <= ((double)((2+t1)))/3 &&
            ((double)((5+t1)))/3 <= p2 && p2 <= n)
         {
          U[((int)floor(((double)((2+t1)))/3))][p2] =
            ( A[((int)floor(((double)((2+t1)))/3))][p2] -
              SUMM[((int)floor(((double)((2+t1)))/3))][p2] ) /
              L[((int)floor(((double)((2+t1)))/3))][((int)floor(((double)((2+t1)))/3))];
         }
       }
      LooPo_parEnd();
     }
    LooPo_parEnd();
   }
  LooPo_finalizeRTS();
 }
 LooPo_endSection();
}

/*    end of
** LooPo program
*/
```

A.2 Überarbeiteter LooPo Code

```
int lu_decomp(int n)
{
  int t,p1,p2,lp1,lp2,x;

  for(t=0; t<=dmax(1,3*(double)n-3); t++)
  {
    for (p1 = iceil(dmin(1,(2+(double)t)/3));
         p1 <= ifloor(dmax((double)n,(3+(double)t)/3)); p1++ )
    {
      for (p2 = iceil(dmin(dmin(0,(double)t),dmin(p1,(3+(double)t)/3)));
           p2 <= dmax(0,(double)n); p2++)
      {
        /*------------------------------------------------------------------*/
        /* Auf allen Prozessoren ausführen */

        if(0 <= t && t <= 0 && 1 <= p1 && p1 <= n && 0 <= p2 && p2 <= 0)
          L[p1][1] = A[p1][1];

        if(1 <= t && t <= 1 && 1 <= p1 && p1 <= n && 0 <= p2 && p2 <= 0)
          U[1][p1] = A[1][p1]/L[1][1];

        if(0 <= t && t <= 0 && 2 <= p1 && p1 <= n && t <= p2 && p2 <= n)
          SUM[p2][p1] = 0;

        /*------------------------------------------------------------------*/
        /* Nur auf dem zugewiesenen Prozessor ausführen */

        if (ProX*(p1-1)+p2 == ProSelf)
        {
          if(2 <= t && t <= 3*n-4 && (t-2)%3 == 0 &&
             (4+(double)t)/3 <= p1 && p1 <= n && p1 <= p2 && p2 <= n)
          {
            x = ifloor((1+(double)t)/3);
            if (t>2)
            {
              MPI_Recv(&L[p2][x], 1, MY_MPI_DATATYPE, ProX*(x-1)+p2,
                       MPI_TAG_5_4, MPI_COMM_WORLD, &status); /* 5->4 */
              MPI_Recv(&U[x][p1], 1, MY_MPI_DATATYPE, ProX*(x-1)+p1,
                       MPI_TAG_9_4, MPI_COMM_WORLD, &status); /* 9->4 */
            }

            SUM[p2][p1] = SUM[p2][p1] + L[p2][x] * U[x][p1];
          } /*if*/

          /*------------------------------------------------------------------*/

          if(3 <= t && t <= 3*n-3 && (t-3)%3 == 0 &&
             (3+(double)t)/3 <= p1 && p1 <= (3+(double)t)/3 &&
             (3+(double)t)/3 <= p2 && p2 <= n)
          {
            x = ifloor((3+(double)t)/3);
            L[p2][x] = A[p2][x] - SUM[p2][x];

            for (lp1 = iceil(dmin(1,(2+(double)(t+2))/3));
```

```
         lp1 <= ifloor(dmax((double)n,(3+(double)(t+2))/3)); lp1++ ) {
      for (lp2 = iceil(dmin(dmin(0,(double)(t+2)),
                       dmin(lp1,(3+(double)(t+2))/3)));
          lp2 <= dmax(0,(double)n); lp2++) {

        if (2 <= (t+2) && (t+2) <= 3*n-4 && ((t+2)-2)%3 == 0 &&
            (4+(double)(t+2))/3 <= lp1 && lp1 <= n &&
            lp1 <= lp2 && lp2 <= n && (((t+2)+1)/3) == p1 && lp2 == p2)
              MPI_Send(&L[p2][x], 1, MY_MPI_DATATYPE, ProX*(lp1-1)+lp2,
                       MPI_TAG_5_4, MPI_COMM_WORLD); /* 5->4 */

        if (2 <= (t+2) && (t+2) <= 3*n-7 && ((t+2)-2)%3 == 0 &&
            (4+(double)(t+2))/3 <= lp1 && lp1 <= n-1 &&
            1+lp1 <= lp2 && lp2 <= n && (((t+2)+1)/3) == p1 && lp1 == p2)
            MPI_Send(&L[p2][x], 1, MY_MPI_DATATYPE, ProX*(lp1-1)+lp2,
                     MPI_TAG_5_8, MPI_COMM_WORLD); /* 5->8 */

      } /*for*/
    } /*for*/

    for (lp1 = iceil(dmin(1,(2+(double)(t+1))/3));
         lp1 <= ifloor(dmax((double)n,(3+(double)(t+1))/3)); lp1++ ) {
      for (lp2 = iceil(dmin(dmin(0,(double)(t+1)),
                       dmin(lp1,(3+(double)(t+1))/3)));
          lp2 <= dmax(0,(double)n); lp2++) {

        if (4 <= (t+1) && t <= 3*n-5 && ((t+1)-4)%3 == 0 &&
            (2+(double)(t+1))/3 <= lp1 && lp1 <= (2+(double)(t+1))/3 &&
            (5+(double)(t+1))/3 <= lp2 && lp2 <= n &&
            (lp1 == p1) && (lp1 == p2))
            MPI_Send(&L[p2][x], 1, MY_MPI_DATATYPE,  ProX*(lp1-1)+lp2,
                     MPI_TAG_5_9, MPI_COMM_WORLD); /* 5->9 */
      } /*for*/
    } /*for*/

    MPI_Send(&L[p2][x], 1, MY_MPI_DATATYPE, 0, MPI_TAG_RESULT,
             MPI_COMM_WORLD);
  } /*if*/
} /*if PRO...*/

/*-----------------------------------------------------------------------*/
/* Auf allen Prozessoren ausführen */

if(0 <= t && t <= 0 && 2 <= p1 && p1 <= n && 0 <= p2 && p2 <= 0)
  U[p1][p1] = 1;

if(0 <= t && t <= 0 && 2 <= p1 && p1 <= n-1 && 1+t <= p2 && p2 <= n)
  SUMM[p1][p2] = 0;

/*-----------------------------------------------------------------------*/
/* Nur auf dem zugewiesenen Prozessor ausführen */

if (ProX*(p1-1)+p2 == ProSelf)
{
  if(2 <= t && t <= 3*n-7 && (t-2)%3 == 0 &&
     (4+(double)t)/3 <= p1 && p1 <= n-1 && 1+p1 <= p2 && p2 <= n)
  {
```

```
          x = ifloor((1+(double)t)/3);
          if (t>2)
          {
            MPI_Recv(&L[p1][x], 1, MY_MPI_DATATYPE, ProX*(x-1)+p1,
                     MPI_TAG_5_8, MPI_COMM_WORLD, &status); /* 5->8 */
            MPI_Recv(&U[x][p2], 1, MY_MPI_DATATYPE, ProX*(x-1)+p2,
                     MPI_TAG_9_8, MPI_COMM_WORLD, &status); /* 9->8 */
          }
          SUMM[p1][p2] = SUMM[p1][p2] + L[p1][x] * U[x][p2];
        }

        /*--------------------------------------------------------------------*/

        if(4 <= t && t <= 3*n-5 && (t-4)%3 == 0 && (2+(double)t)/3 <= p1 &&
           p1 <= (2+(double)t)/3 && (5+(double)t)/3 <= p2 && p2 <= n)
        {
          x = ifloor((2+(double)t)/3);
          if (t>2)
            MPI_Recv(&L[x][x], 1, MY_MPI_DATATYPE, ProX*(p1-1)+p1,
                     MPI_TAG_5_9, MPI_COMM_WORLD, &status); /* 5->9 */

          U[x][p2] = (A[x][p2] - SUMM[x][p2]) / L[x][x];

          for (lp1 = iceil(dmin(1,(2+(double)(t+1))/3));
               lp1 <= ifloor(dmax((double)n,(3+(double)(t+1))/3)); lp1++ ) {
            for (lp2 = iceil(dmin(dmin(0,(double)(t+1)),
                             dmin(lp1,(3+(double)(t+1))/3)));
                 lp2 <= dmax(0,(double)n); lp2++) {

              if (2 <= (t+1) && (t+1) <= 3*n-4 && ((t+1)-2)%3 == 0 &&
                  (4+(double)(t+1))/3 <= lp1 && lp1 <= n &&
                  lp1 <= lp2 && lp2 <= n && (((t+1)+1)/3) == p1 && lp1 == p2)
                MPI_Send(&U[x][p2], 1, MY_MPI_DATATYPE,  ProX*(lp1-1)+lp2,
                         MPI_TAG_9_4, MPI_COMM_WORLD); /* 9->4 */

              if (2 <= (t+1) && (t+1) <= 3*n-7 && ((t+1)-2)%3 == 0 &&
                  (4+(double)(t+1))/3 <= lp1 && lp1 <= n-1 &&
                  1+lp1 <= lp2 && lp2 <= n && (((t+1)+1)/3) == p1 && lp2 == p2)
                MPI_Send(&U[x][p2], 1, MY_MPI_DATATYPE,  ProX*(lp1-1)+lp2,
                         MPI_TAG_9_8, MPI_COMM_WORLD); /* 9->8 */

            } /*for*/
          } /*for*/
          MPI_Send(&U[x][p2], 1, MY_MPI_DATATYPE, 0, MPI_TAG_RESULT,
                   MPI_COMM_WORLD);
        } /*if*/
      } /*if PRO...*/
    }
  }
 }
}
```

Folgende Modifikationen wurden durchgeführt:

1. `LooPo_parBegin/LooPo_parEnd` in FOR-Schleife umgewandelt und ein zusätzliches IF-Statement eingefügt, welches sicherstellt, daß die entsprechenden Statements nur auf dem zugewiesenen Prozessor ausgeführt werden.

2. LooPo spezifischen Code entfernt:

   d.h. `LooPo_beginSection`, `LooPo_initializeRTS`, `LOOPO_SECTION_OK`, `LOOPO_PARFOR_OK`, `LooPo_finalizeRTS` und `LooPo_endSection`

3. Kommunikationsstatements für die `MPI`-Bibliothek eingefügt.

Passau, 22. Mai 2000

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, daß ich die vorliegende Diplomarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet habe, sowie daß diese Diplomarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

(Sven Anders)