Technische Universität Chemnitz
Sonderforschungsbereich 393
*Numerische Simulation auf massiv parallelen Rechnern*

Thomas Ermer

# Mappingstrategien für Kommunikatoren

**Preprint SFB393/98-10**

**Zusammenfassung**

Zu effektiver Kommunikation in parallelen FEM–Systemen gelangt man auf mehreren Wegen. Zum einen kann man versuchen, die Kommunikationslast zu minimieren, in dem das betrachtete Gebiet geschickt auf die verfügbaren Prozessoren verteilt wird. Solche Partitionierungen lassen sich z.B. mit dem Programmsystem *chaco* erzeugen [9], [2]; jedoch ist der erforderliche Rechenaufwand enorm und der Einsatz derartiger Strategien nur im Preprocessing sinnvoll. Zum anderen kann man für eine geeignete Anordnung der sequentiellen Kommunikationsschritte in sogenannten Linkleveln sorgen, um den Datenaustausch möglichst parallel realisieren zu können. Die in [2] beschriebene MPI–basierte Koppelrandkommunikation ist in dieser Hinsicht ein effektives Verfahren.

Eine dritte Möglichkeit leitet sich aus der Eigenschaft von Kommunikationshardware ab, ein großes Datenpaket schneller zu übertragen, als eine auf mehrere Pakete verteilte äquivalente Datenmenge. Bei letzteren addieren sich die Latenzzeiten für die Kommunikation, die bei einem Datenpaket nur einmal zu Buche schlägt.

In der vorliegenden Arbeit wird ausgehend von der MPI-basierten Koppelrandkommunikation ein Splitalgorithmus vorgestellt, der versucht, die Koppelranddaten großer Kommunikatoren auf die kleinerer Sub–Kommunikatoren abzubilden und damit die Anzahl zu übertragender Datenpakete zu minimieren.

**Preprint-Reihe des Chemnitzer SFB 393**

**SFB393/98-10** **April 1998**

# Inhaltsverzeichnis



Author's addresses:

Thomas Ermer
TU Chemnitz
Fakultät für Informatik
D-09107 Chemnitz

`therminformatik.tu-chemnitz.de`

# 1 Einleitung

## 1.1 Gliederung

Ausgangspunkt nachfolgender Überlegungen ist die in [4] dargestellte Möglichkeit zur Koppelrandkommunikation. Abschnitt 2 beschreibt Möglichkeiten von Optimierungen, die zu einer Beschleunigung des *getcomm*–Algorithmus[1] führen. Abschnitt 3 erläutert eine Idee, wie die Koppelrandkommunikation durch Zerlegung von Kommunikatoren effizienter gestaltet werden kann. Anschließend werden die damit verbundenen Probleme diskutiert, die diesen Ansatz mehr oder weniger sinnvoll erscheinen lassen.

Neben Effizienzbetrachtungen in Abschnitt 3.3 wird in Abschnitt 4 kurz auf den Einfluß von Partitionierungsverfahren in Zusammenhang mit dem neuen Split–Algorithmus eingegangen.

## 1.2 MPI–Basiskonzepte

Die Kenntnis zweier Basiskonzepte des Message Passing Interfaces (MPI) sind für die folgenden Ausführungen erforderlich.

*Kommunikatoren* bieten als herausragende Eigenschaft die Möglichkeit, voneinander geschützte Kommunikationswelten zu schaffen. Innerhalb eines Kommunikators bleibt die dortige Kommunikation auf einunddenselben Kontext beschränkt; Auswirkungen auf den Datenaustausch in anderen Kommunikatoren sind nicht möglich. Initial gehören alle Prozesse während des Programmstarts zum vordefinierten Kommunikator MPI_COMM_WORLD. Von diesem ausgehend, bietet MPI dem Nutzer die Möglichkeit, hierarchisch Kommunikatoren von bereits bestehenden abzuleiten. Jeder Kommunikator besitzt einen eindeutigen Identifikator.

MPI realisiert den gesamten Datenaustausch über MPI–spezifische *Datentypen*, die mit den Datentypen üblicher Programmiersprachen korrespondieren. Darüber hinaus bietet MPI Mechanismen, um nutzereigene Datentypen zu erzeugen [4], [8].

In Bezug auf die betrachteten FEM–Systeme, werden die innerhalb einer Prozessorgruppe auszutauschenden Randketten zu einem eigenen Datentyp zusammengefaßt und über alle beteiligten Prozessoren wird ein Kommunikator gebildet.

## 1.3 Weitere Begriffe

Ausgehend von einem gegebenen Grobnetz werden die im Laufe seiner Verfeinerung entstehenden Knoten zwischen den Crosspoints zu sogenannten Ketten

[1] Der Algorithmus bewerkstelligt u.a. die Erzeugung der Datentypen und Kommunikatoren für die Koppelrandkommunikation.

zusammengefaßt. Einer solchen Kette, die durch ihren Start–, ihren Endcrospoint und ihre Länge charakterisiert ist, kann eine eindeutige Zahl zugewiesen werden, diese wird nachfolgend *KettID* genannt [4, 2].

Die auf dem Koppelrand liegenden Knoten bedingen Kommunikation. Ein Prozessor kann zu einem Zeitpunkt jedoch nur mit einem anderen Prozessor seine Daten austauschen. Sind mehrere Kommunikationsschritte erforderlich, müssen diese in einer eindeutigen Reihenfolge – dem *Linklevel* – abgearbeitet werden.

Für die globalen Kommunikationsoperationen werden nachfolgend die Cubecom–spezifischen Bezeichnungen verwendet [5], so verbirgt sich z.B. hinter dem Tree_down ein MPI_Bcast.

# 2 Erzeugen der Kommunikatoren

Die hier vorgestellten Algorithmen können uneingeschränkt für die Programmodule SPC–PM Po 2D und SPC–PM Po 3D verwendet werden, wobei der Aufbau verschiedener Listen schon in Hinblick auf den später vorgestellten *Split–Algorithmus* gewählt wurde.

Für den Aufbau der Kommunikatoren werden von jedem Prozessor folgende Informationen benötigt:

- die global eindeutige KettID der lokal vorhandenen Ketten (Routine setuptypes)
- die an den lokalen Ketten noch beteiligten Prozessoren (Routine getcomm, Abschnitt 2.2)
- eine eindeutige Linklevelzuordnung für die Kommunikatoren (Routine getorder, Abschnitt 2.2)

## 2.1 Prinzip des originalen getcomm-Algorithmus

Der Ablauf der in [4] aufgeführten Version läßt sich grob in folgende Abschnitte gliedern:

1. jeder Prozeß bestimmt die global eindeutigen KettIDs seiner lokalen Ketten (Routine getcomm)
2. mittels TREE_UP werden alle diese KettIDs zu Prozeß 0 geschickt
3. Prozeß 0 sortiert die Liste nach den KettIDs, ermittelt die zugehörigen Prozesse und bestimmt eine eindeutige Linklevelzuordnung (Routine getorder)
4. mittels TREE_DOWN erhält jeder Prozeß die komplette Liste mit den zugehörigen Linkleveln

5. jeder Prozeß ordnet seine lokalen KettIDs in die entsprechenden Linklevel ein und kreiert mittels MPI_COMM_SPLIT einen Kommunikator (Routine gettypes)

## 2.2 Prinzip des neuen getcomm-Algorithmus

Die grundlegende Änderung spiegelt sich in der weitestgehend lokalen Berechnung der Linklevelzuordung wieder. Hierzu benötigen alle Prozessoren die Kenntnis über alle KettIDs.

### 2.2.1 Wer benutzt welche Kette?

In einem Vektor, der die Länge der Anzahl der globalen Ketten besitzt, belegt jeder Prozessor diejenigen Einträge[2] mit seinem Prozessorbit (*Prozeßindex*, Abbildung 1), die seinen lokal vorhandenen Ketten entsprechen.

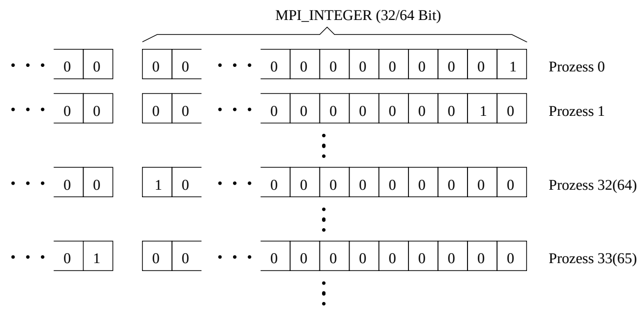



Abbildung 1: Prozeßindex bezeichnet das jedem Prozeß zugeordnete Bit.

Mittels CUBE_DOD (MPI_Allreduce) und einer bitweisen ODER–Verknüpfung (MPI_BOR) über dem Vektor erhält jeder Prozessor die Informationen über die Verteilung aller Ketten, die entstehende Liste wird im weiteren Verlauf als *globale Prozeßliste* bezeichnet. Abbildung 2 zeigt das Prinzip für 8 Prozesse, wobei vereinfachend der MPI_INTEGER mit einer Länge von einem Byte angenommen wird.

### 2.2.2 Lokale Linklevel–Sortierung

Häufig sind zwischen einer Gruppe von Prozessoren nicht nur eine einzige Kette, sondern mehrere Ketten auszutauschen. In der Subroutine getorder durchsucht jeder Prozessor daher die zu seinen lokalen Ketten korrespondierenden Einträge der *globalen Prozeßliste* und sortiert alle Ketten, die zwischen genau denselben Prozessoren auszutauschen sind, zusammen mit ihren *lokalen Linkleveln* in die *lokale Prozeßliste* ein.

[2]In Abhängigkeit von der Größe des MPI_INTEGER und der benötigten Prozessoranzahl wird für jeden Eintrag ein genügend großes Feld von INTEGER–Werten verwendet.

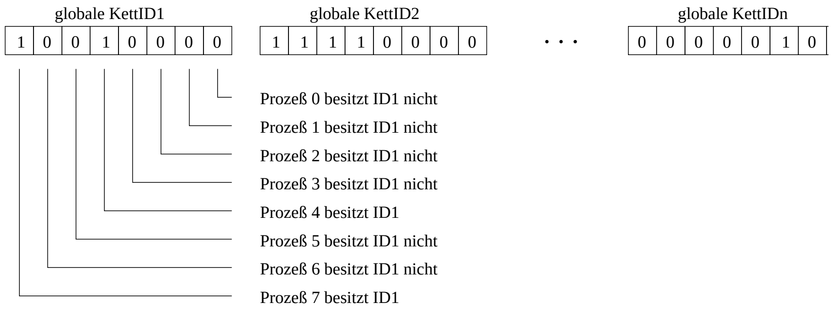



Abbildung 2: Globale Prozeßliste der KettIDs mit zugehörigen Prozeßeinträgen. Ein gesetztes Bit $i$ zeigt an, daß diese Kette von Prozeß $i$ benutzt wird.

Für alle Ketten mit identischem lokalen Linklevel (und damit auch mit identischer Prozessorzugehörigkeit) wird diejenige mit der kleinsten KettID in der *commlist* gespeichert, anhand welcher später die globale Linkleveleinordnung erfolgt.

Zusätzlich wird aus dieser *commlist* eine *comm0list* generiert. Eine Kette erscheint nur dann in dieser Liste, wenn das niederwertigste Bit aus dem globalen Prozeßlisteneintrag dem eigenen Prozessorindex entspricht.

Abbildung 3 zeigt den beispielhaften Aufbau der drei Listen.

| KettID | Prozesszugehörigkeit | commlist | comm0list |
|---|---|---|---|
| 1 | 1 1 0 1 0 1 0 0 | 1 1 0 1 0 1 0 0 | 1 1 0 1 0 1 0 0 |
| 2 | 0 1 1 0 1 1 0 0 | 0 1 1 0 1 1 0 0 | 0 1 1 0 1 1 0 0 |
| 3 | 0 0 0 1 0 1 0 1 | 0 0 0 1 0 1 0 1 | |
| 4 | 0 0 0 0 0 1 1 0 | 0 0 0 0 0 1 1 0 | |
| 5 | 1 1 0 1 0 1 0 0 | 0 1 0 1 0 1 0 1 | |
| 6 | 0 1 0 1 0 1 0 1 | | |
| 7 | 1 1 0 1 0 1 0 0 | | |
| 8 | 0 1 1 0 1 1 0 0 | | |

Abbildung 3: Prozeß 2 besitzt 8 Ketten mit den globalen IDs 1 bis 8. Die commlist enthält alle disjunkten KettIDs (also mit unterschiedlicher Prozeßbeteiligung), comm0list enhält jene Ketten, deren niederwertigstes gesetztes Bit das Bit 2 ist.

Die Listen *comm0list* dienen der späteren Generierung der globalen Linklevelreihenfolge auf Prozessor 0. Werden diese Listen von jedem Prozessor mittels TREE_UP zur 0 gesendet, so enthält die dort entstehende Liste nur die *Referenzketten*, d.h. es existiert keine KettID mit gleicher Prozessorzugehörigkeit. Gleichzeitig hat jede in den lokalen *commlist* eingetragene Kette einen bzgl. der Prozessorzugehörigkeit äquivalenten Eintrag in dieser *Referenzkettenliste*.

### 2.2.3 Globale Linklevel–Sortierung

TREE_UP über alle lokalen *comm0list* liefert auf Prozessor 0 eine Liste mit Referenzketten. Für jede mögliche Kommunikationsbeziehung ist eine Kette in dieser Liste enthalten und keine Kommunikationsbeziehung wird durch mehrere Ketten repräsentiert.

Prozessor 0 generiert anhand dieser Liste eine globale Linklevelreihenfolge nach folgendem Algorithmus:

1. Initialisiere alle Referenzketten mit Linklevel 0.

2. Linklevel = Linklevel + 1

3. Benutze erste Referenzkette mit Linklevel 0, speichere den zugehörigen Eintrag der globalen Prozeßliste in einem Hilfsvektor HV1 und setze Linklevel.

4. Für alle weiteren Einträge der Referenzkettenliste

   (a) Speichere zugehörigen Eintrag aus der globalen Prozeßliste in einem Hilfsvektor HV2.

   (b) Ist (HV1.and.HV2) der Nullvektor, dann setze HV1 = (HV1.or.HV2) und setze Linklevel.

5. Solange Ketten mit Linklevel 0 existieren, fahre mit 2. fort.

Im originalen Algorithmus wurden die Ketten zuvor in zunehmender Folge ihrer Prozeßzugehörigkeit (zuerst Ketten, die zu zwei Prozessen gehören, anschließend Ketten, an denen drei Prozesse beteiligt sind, ..., zuletzt die Ketten, mit der maximal auftretenden Anzahl an Prozessen) angeordnet. Tests zeigen aber, daß eine derartige Vorsortierung zu keinen geringeren Linklevelanzahlen führt.

Im Hinblick auf den weiter unten vorgestellten Split–Algorithmus erweist es sich als günstig, Hypercubeketten (Ketten, die von $2^d, d \geq 2$ Prozessen benutzt werden), in abnehmender Folge ihrer Prozeßzugehörigkeit am Anfang der Liste zu speichern; alle anderen Ketten werden anschließend unsortiert gespeichert.

Die Liste der Referenzketten mit den zugehörigen Linkleveleinordnungen wird mittels Tree_down an alle Prozessoren verteilt.

### 2.2.4 Lokale Konstruktion der Kommunikatoren

Jeder Prozeß besitzt nun eine Liste aller globalen Referenzketten mit den zugehörigen Linkleveln. Alle in der *commlist* eingetragenen KettIDs erscheinen auch in dieser Liste, so daß in der Routine *sortlevel* die Zuordnung der globalen Linklevel zu den lokalen KettIDs erfolgen kann.

Alle Ketten mit demselben Linklevel werden anschließend zu einem neuen MPI–Datentyp zusammengefaßt und über der Prozeßgruppe ein Kommunikator (mpi_comm_split) erzeugt. Als eindeutiger Kommunikatorindex dient dabei die jeweils größte globale KettID eines jeden Linklevels.

## 2.3 Zur Effizienz der optimierten Implementierung

Der (zeitlich) dominierende Anteil der neuen getcomm–Implementierung spiegelt sich im MPI_ALLREDUCE wieder, abhängig von der Leistungsfähigkeit der verwendeten Architektur trägt dieser Teil zu 60 % (Pentium 100 MHz) bis zu 95 % (PowerGC) zur Gesamtlaufzeit bei. Vergleicht man den MPI_ALLREDUCE allein mit dem TREE_UP aus Punkt 2 der originalen Version, so liegt dieser unabhängig vom Prozessortyp nur ca. 15 % darüber.

Eine Abschätzung des Kommunikationaufkommens liefert für den Allreduce $2 \cdot (n-1)$ Kommunikationsschritte (bei $n$ Prozessen) mit jeweils $nobjs$ Ketten. Der TREE_UP benötigt ebenso viele Schritte zum Datenaustausch (je $(n-1)$ für die Längeninformation und für die KettIDs), das Datenaufkommen bei realistischen Gebietsverteilungen liegt bei durchschnittlich mindestens (!) $nobjs/2$ KettIDs pro Schritt (gemessene Werte).

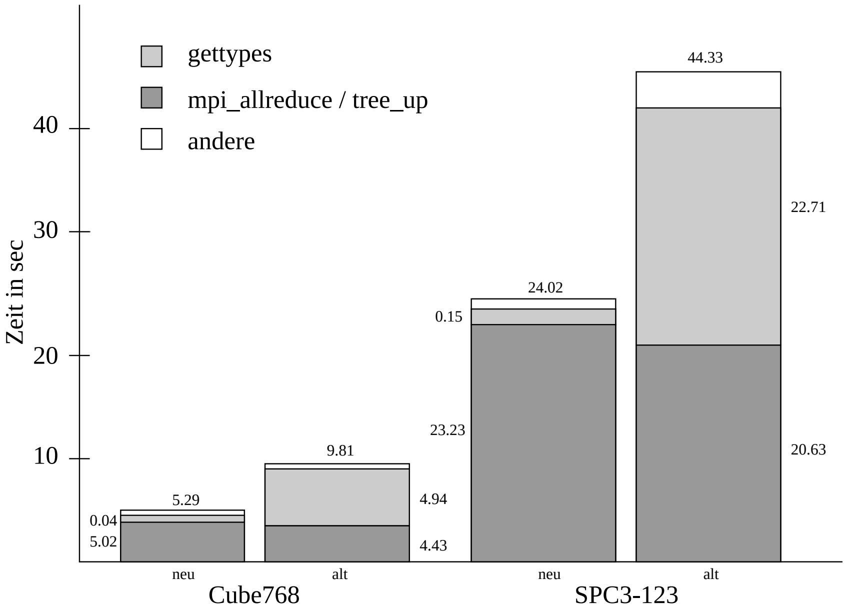



Abbildung 4: Vergleich des originalen mit dem neuen *getcomm*– Algorithmus auf PowerGC (16 Prozessoren, Level 3)

Die im originalen Algorithmus verwendeten Sortierroutinen finden in der neuen Variante keine Verwendung, eventuelle Tests wurden auf einfache Bitoperationen abgebildet und benötigen "nahezu keine Zeit".

Der zeitkritische MPI_COMM_SPLIT konnte durch Reduzierung des Aufwandes bei der Typenbildung und der damit verbundenen Verminderung eventueller Synchronisations–Wartezeiten (der eigentliche mpi_comm_split) um ca. 90 % seiner originalen Laufzeit verringert werden.

# 3 Mapping von Kommunikatoren

## 3.1 Grundprinzip: Ein einfaches Beispiel

Das nachfolgende Beispiel betrachtet ein auf vier Prozessoren verteiltes 2–dimensionales Gebiet: Jeweils 2 Prozessoren besitzen eine (2D–) Kette und einen Crospoint gemeinsam, alle 4 Prozessoren benutzen einen Crosspoint gemeinsam[3] (Abbildung 5).

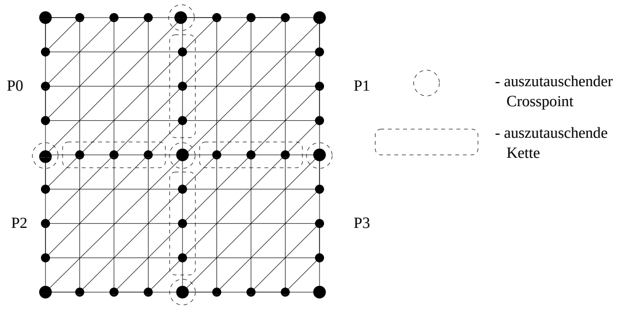



Abbildung 5: Beispielgebiet für 4 Prozessoren

Die Crosspoints können als Spezialfall der 2D–Ketten mit Länge 1 angesehen werden und als diese in die Kommunikationsstruktur eingebunden werden[4].

Als eine mögliche Anordnung der Kommunikationsreihenfolge ergibt sich hier:

1. Prozessor 0 ($P_0$) $\Longleftrightarrow$ Prozessor 1 ($P_1$)
   Prozessor 2 ($P_2$) $\Longleftrightarrow$ Prozessor 3 ($P_3$)

2. Prozessor 0 ($P_0$) $\Longleftrightarrow$ Prozessor 2 ($P_2$)
   Prozessor 1 ($P_1$) $\Longleftrightarrow$ Prozessor 3 ($P_3$)

3. globale Aufsummierung des Crosspoints über alle Prozessoren

D.h.: letztendlich sind drei Kommunikationsschritte notwendig, um alle Randketten und Crosspoints zwischen den Prozessoren auszutauschen. In Abbildung 6 sind die Kommunikationsschritte mit den darin aufsummierten Knoten dargestellt.

Betrachtet man nun den "zentralen" Crosspoint, den sich alle vier Prozessoren teilen, erkennt man, daß dieser allein ein einziges Kommunikationslevel erfordert und daß dieser mit den beiden anderen Leveln bereits vollständig berechenbar ist. Hieraus resultiert die Idee der Reduzierung dieses Levels durch **Splittung** der Berechnung nach folgendem Schema:

[3] Solche Gebiete werden z.B. im Programmsystem SPC–PM Po 2D verwendet.

[4] Im Programmmodul SPC–PM Po 2D ist diese Vorgehensweise nur dann sinnvoll, wenn auf Prozessor 0 keine Lösung des Grobgittersystems vorgesehen ist.

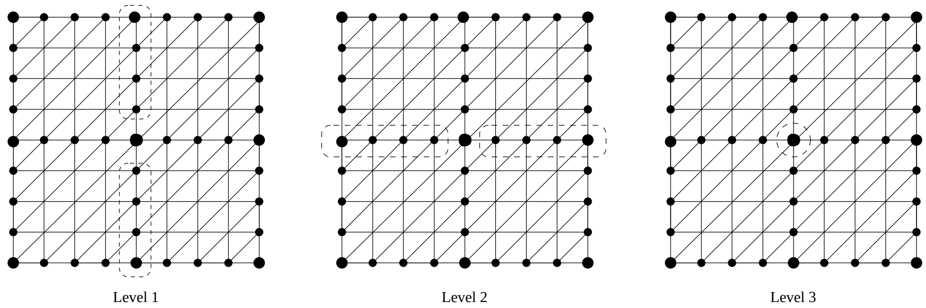



Abbildung 6: Kommunikationsreihenfolge und auszutauschende Knoten mit separatem Crosspoint.

- Erweitere die im Schritt 1 zu berechnenden Ketten und Crosspoints um diesen "zentralen" Crosspoint
- Erweitere die im Schritt 2 zu berechnenden Ketten und Crosspoints um diesen "zentralen" Crosspoint
- Streiche Schritt 3 der Kommunikation

Nach zwei Kommunikationsleveln steht der "zentrale" Crosspoint auf allen Prozessoren bereits zur Verfügung. Diese Vorgehensweise ist in Abbildung 7 schematisch dargestellt.

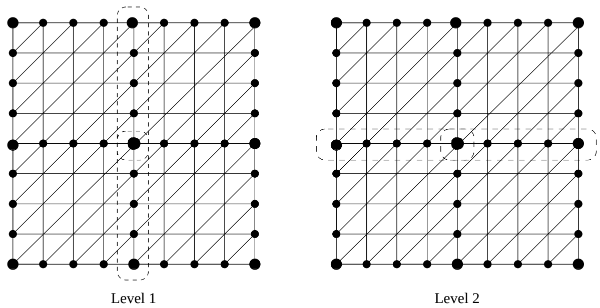



Abbildung 7: Kommunikationsreihenfolge und auszutauschende Knoten mit gesplittetem Crosspoint.

Wie im weiteren Verlauf dieser Arbeit noch zu erkennen sein wird, handelt es sich im obigen Beispiel um ein ideales Beispiel, das nur bedingt auf den dreidimensionalen Fall übertragbar ist. Zum einen ist die Aufsplittung eines Kommunikationsmusters nur selten mit der unmittelbaren Reduzierung eines Linklevels verbunden und zum anderen muß eine solche Kommunikationsbeziehung nicht immer durch andere Muster darstellbar sein. Auch spielen bei komplexeren Kommunikationsstrukturen andere Faktoren eine Rolle, wie weiter unten noch diskutiert wird.

## 3.2 Welche Ketten können überhaupt gesplittet werden?

### 3.2.1 Hypercube–Ketten

Unter *Hypercube–Ketten* versteht man alle Ketten, die zu $n$ Prozessen gehören, welche einen Hypercube der Dimension $\geq 2$ aufspannen, also die Ketten, die von $4, 8, 16, ..., 2^m$ Prozessen verwendet werden. Die zu einer *Hypercube–Kette* der Dimension $m$ zugehörige *Subhypercube–Kette* beschreibt einen Hypercube der Dimension $m-1$ und gehört zu einer Teilmenge der Prozesse, die die Hypercube–Kette benutzen. Für Hypercube–Ketten der Dimensionen 2 und 3 sind die möglichen Subhypercube–Ketten in Abbildung 8 dargestellt.

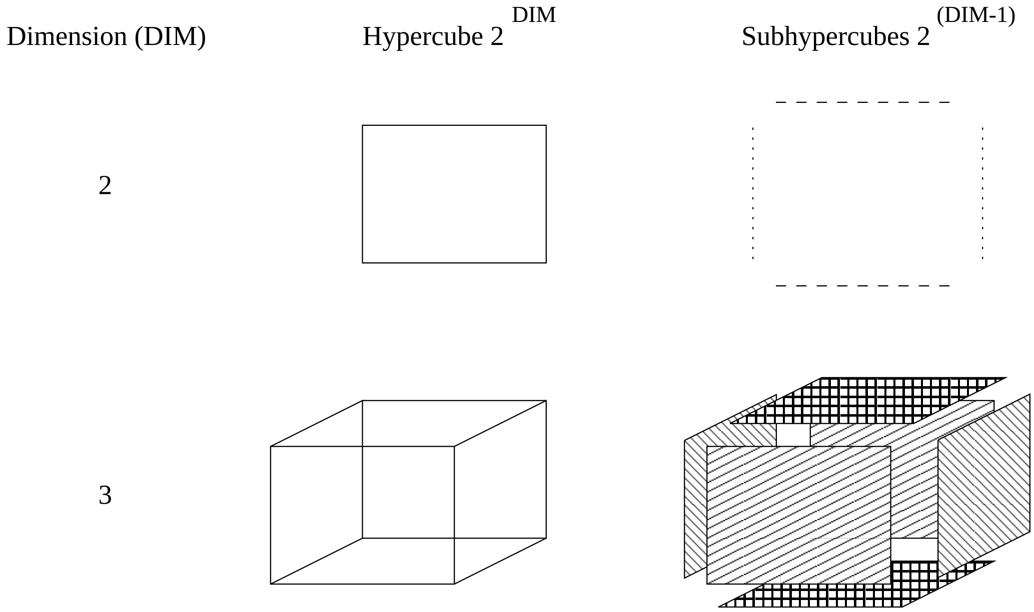



Abbildung 8: Hypercubes der Dimensionen 2 und 3 mit den zugehörigen Subhypercubes (Auswahl).

Um eine *Hypercube–Kette* splitten zu können, müssen folgende zwei Voraussetzungen erfüllt sein:

- *Jede Hypercube–Kette der Dimension $i$ muß durch $2 \cdot 2$ Subhypercube–Ketten dargestellt werden können:*

  Sei $M$ die Menge der Prozesse, die zu einer Hypercube–Kette gehören und die *unterschiedlichen* Mengen $O, P, Q, R$ enthalten die Prozesse korrespondierender Subhypercube–Ketten.

  Gesucht wird folgende Anordnung:

  1. $(O \cup P = M) \wedge (O \cap P = \emptyset)$

2. $(Q \cup R = M) \wedge (Q \cap R = \emptyset)$

Das heißt: ein zu splittender Hypercube muß durch mindestens zwei unterschiedliche Subhypercube–Paare darstellbar sein.

- *Die Kommunikationslevel des einen Subhypercube–Ketten–Paares müssen beide vor denen des zweiten Paares liegen:*

**Beispiel: CYL24 8 Prozessoren, Level 1**

Die folgende Tabelle enthält nur die für das Beispiel relevanten KettIDs mit der Bitdarstellung (Prozeßzugehörigkeit) und den Levelzuordnungen, so wie sie von Prozeß 0 erzeugt wurden:

| KettID | 1 | 30 | 3 | 8 | 14 |
|---|---|---|---|---|---|
| Bit | 11000011 | 00111100 | 00000011 | 01000001 | 10000010 |
| in Level | 1 | 1 | 5 | 7 | 7 |

| KettID | 25 | 36 | 45 | 50 | 73 |
|---|---|---|---|---|---|
| Bit | 00001100 | 00100100 | 00011000 | 00110000 | 11000000 |
| in Level | 5 | 7 | 7 | 8 | 8 |

Die Aufspaltung ergibt (Abbildung 9):

- für KettID 1
  - KettID 3 und KettID 73 (Level 5 und Level 8)
  - KettID 8 und KettID 14 (Level 7 und Level 7)
- für KettID 30
  - KettID 25 und KettID 50 (Level 5 und Level 8)
  - KettID 36 und KettID 45 (Level 7 und Level 7)

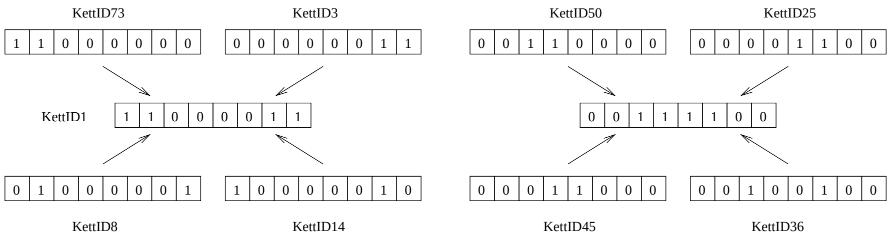



Abbildung 9: Die KettIDs 1 und 30 lassen sich jeweils durch zwei unterschiedliche Aufspaltungen darstellen.

Angenommen die Ketten hätten alle die Länge 1 und auf jedem Prozeß den Wert 1. Eine Aufsummierung für die KettIDs 1 und 30 sollte zum Ergebnis = 4 auf allen, die jeweilige Id besitzenden Prozessen führen, aber (Beispiel KettID 30):

| Prozeß | 2 | 3 | 4 | 5 | Kommunikation |
|---|---|---|---|---|---|
| Wert vorher | 1 | 1 | 1 | 1 | |
| Wert nach Level 5 | 2 | 2 | 1 | 1 | 2 ⇔ 3 |
| Wert nach Level 5 | 2 | 2 | 1 | 1 | 2 ⇔ 3 |
| Wert nach Level 7 | 3 | 3 | 3 | 3 | 2 ⇔ 5, 3⇔4 |
| Wert nach Level 8 | 3 | 3 | 6 | 6 | 4⇔5 |

D.h. die Reihenfolge der Level ist entscheidend. Bildlich gesprochen, müssen alle Kommunikationen einer Hypercubeebene abgeschlossen sein, bevor die Kommunikation für eine andere Ebene startet. Ein Tausch der Level 7 und 8 im obigen Beispiel bringt das gewünschte Ergebnis:

| Prozeß | 2 | 3 | 4 | 5 | Kommunikation |
|---|---|---|---|---|---|
| Wert vorher | 1 | 1 | 1 | 1 | |
| Wert nach Level 5 | 2 | 2 | 1 | 1 | 2 ⇔ 3 |
| Wert nach Level 8 (7) | 2 | 2 | 2 | 2 | 4⇔5 |
| Wert nach Level 7 (8) | 4 | 4 | 4 | 4 | 2 ⇔ 5, 3⇔4 |

In diesem, bewußt einfach gewählten Beispiel läßt sich das Problem der Levelreihenfolge recht einfach durch simple Umnumerierung erreichen. In anderen Beispielen aber führt dieser Ansatz schnell zu einer Sackgasse: Durch rekursive Umsortierung der Level tritt nicht selten der Fall auf, daß eine einmal gewählte Reihenfolge wieder umgekehrt wird. Es ist also erforderlich, nach einer für alle Hypercube–Aufspaltungen eindeutigen und korrekten Reihenfolge zu suchen (Siehe 3.2.4).

### 3.2.2 Nicht–Hypercube–Ketten

Zur Darlegung des Sachverhaltes genügt es, den 2D–Fall zu betrachten. In Abbildung 10 teilen sich 3 Prozesse einen gemeinsamen Crosspoint, Prozeß $P_0$ teilt sich eine Kette mit Prozeß $P_1$ und Prozeß $P_1$ besitzt eine Kette mit Prozeß $P_2$ gemeinsam. Eine Aufteilung des gemeinsamen Crosspoints analog zum einführenden Beispiel (3.1) würde nur auf Prozeß $P_1$ zum korrekten Ergebnis führen, auch jede beliebige andere Anordnung führt zu verfälschten Resultaten: *Eine simple Aufsplittung ist nicht möglich.* In diesem Beispiel brächte ein anschließender Broadcast von Prozeß $P_1$ zu den anderen beiden Prozessen die Lösung, *ohne zusätzliche Kommunikation kommt man nicht aus.*

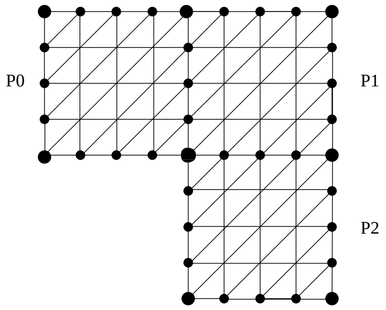



Abbildung 10: Beispielgebiet für 3 Prozessoren, die sich einen Crosspoint teilen.

*Was bedeutet dies für die Effizienz?* Im Beispiel aus Abbildung 10 wären ohne Splittung drei Kommunikationslevel notwendig:

1. $P_0$ kommuniziert mit $P_1$ (Kette)

2. $P_1$ kommuniziert mit $P_2$ (Kette)

3. $P_0, P_1, P_2$ kommunizieren miteinander (Crosspoint)

Mit Splittung entfällt wiederum das Level für die Aufsummierung des Crospoints, allerdings tritt an dessen Stelle ein Broadcast, der zwar schneller vonstatten geht als die entsprechende Reduce–Operation, jedoch benötigen die beiden ersten Level nun mehr Zeit (größere Datenmenge) und der Aufwand für den Split–Algorithmus sorgt für zusätzliche Laufzeit. Im Endeffekt ist bestenfalls eine minimale Effizienzsteigerung zu erwarten.

Für eine Teilmenge der *Nicht–Hypercube–Ketten* zeichnet sich eine andere Lösung ab, und zwar für Ketten mit $n$ Prozessen, wobei $n+1$ Prozesse einen Hypercube aufspannen. Daß auf diese Möglichkeit dennoch verzichtet wurde, ist in der durch die zwingende Linklevelreihenfolge beschränkten Rekursionstiefe des Split–Algorithmus begründet, wie später noch ausgeführt wird. Dennoch sei auf diese Möglichkeit kurz verwiesen:

Man erweitert die $n$ Prozesse um einen weiteren Prozeß und verfährt mit dieser Gruppe wie im vorhergehenden Abschnitt beschrieben. Dieser zusätzliche Prozeß fungiert nur als Dummy–Prozeß und ignoriert am Ende die Resultate der Kommunikation (Abbildung 11 zeigt die Idee für 7 Prozesse).

Mehrere Aspekte bedürfen weiterer Untersuchungen:

- *Welcher Prozeß wird zum Dummy–Prozeß?* Zum einen muß dieser Prozeß die Gruppe derart ergänzen, daß der resultierende Hypercube durch Subhypercubes darstellbar wird und zum anderen sollte der Prozeß möglichst nicht andersweitig beschäftigt sein (Problem der Lastbalancierung).

- *Wohin speichert der Dummy–Prozeß die Zwischenergebnisse?* Vorab müßte ein hinreichend langes Feld für derartige Zwecke reserviert werden (Wie

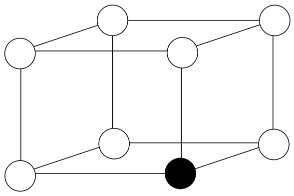



Abbildung 11: Eine Gruppe von Prozessen wird um einen zusätzlichen Prozeß erweitert, um wieder einen Hypercube aufzuspannen.

lang? Der Prozeß kann seine Beteiligung an einer Kommunikation zwar anhand der Referenzketten–Liste herausfinden, die Anzahl der auszutauschenden Ketten kann er jedoch nur nach nochmaligem Durchsuchen der globalen Prozeßliste feststellen $\Longrightarrow$ Zeitaufwand!)

- *Effizienz?* Nur bei *großen $n$* wird der eingesparte Kommunikationsschritt die *unnütze* Kommunikation mit dem Dummy–Prozeß aufwiegen.

Andere Nicht–Hypercube–Ketten bedingen immer zusätzliche Kommunikation in Form nachfolgender Broadcasts und die Sinnfälligkeit deren Aufsplittung erscheint äußerst fraglich.

Wie bereits angedeutet, wurden im nachfolgend beschriebenen Split–Algorithmus einzig die Hypercube–Ketten berücksichtigt und im folgenden Abschnitt wird sich zeigen, daß auch diese nicht immer in kleinere Gruppen zerfallen, selbst wenn diese durch Subhypercube–Ketten darstellbar sind.

### 3.2.3 Weitere Einschränkungen

Eine Problematik wurde im Beispiel des Abschnitts 3.2.1 bereits angesprochen: die der zwingenden Linklevelreihenfolge. Erkennbar wurde auch, daß eine Änderung der Anordnung der Level in diesem Beispiel die Lösung bringt. Schwieriger wird es, wenn der Split–Algorithmus auf weitere Ketten angewendet wird, es muß nun sichergestellt sein, daß eine einmal festgelegte Linklevelreihenfolge nicht wieder umgekehrt wird.

**Erster Versuch:** Ein vertauschtes Level erhält eine Markierung (Abbildung 12). Zwei Level dürfen nun nur ausgewechselt werden, wenn entweder beide Level noch nicht markiert sind oder das markierte Level das erste bzw. letzte markierte der Liste ist und das andere Level noch keine Markierung besitzt. In Abbildung 12 darf demnach Level 7 nicht den Platz mit Level 6 (originales Level) wechseln, da dann die einmal getroffene Reihenfolge zwischen den Leveln 5 und 6 verändert werden würde. Aber: Level 7 könnte mit Level 3 ausgetauscht werden.

**Zweiter Versuch:** Zusätzlich zu Versuch Eins kann ein noch nicht markiertes Level genutzt werden, um die geforderte Reihenfolge herzustellen. Abbildung 13 zeigt den Tausch und verweist bereits auf ein weiteres Problem: Die Level 3 und 6


Abbildung 12: Unmarkierte Level dürfen getauscht werden, markierte Level dürfen mit einem unmarkierten den Platz wechseln, wenn erstere am Listenanfang bzw. –ende stehen.

sind beide markiert, können demnach nicht die geforderte Anordung einnehmen, obwohl ein Tausch z.B. in der Reihenfolge $3 \leftrightarrow 4$ und anschließend $3 \leftrightarrow 6$ keine bisherige Forderung verletzen würde.

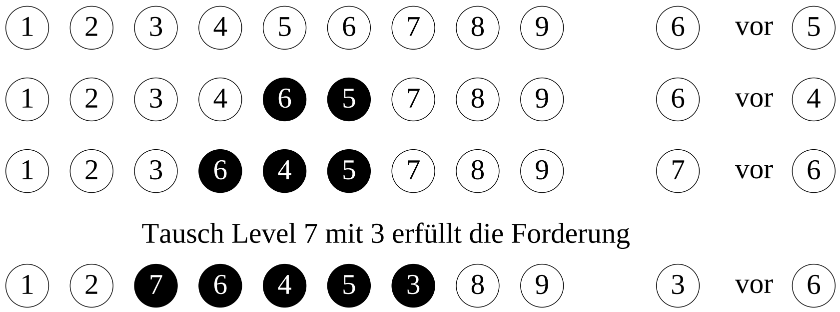



Abbildung 13: Zusätzlich kann ein unmarkiertes Level mit einem beliebigen, noch nicht markierten Level den Platz tauschen.

**Weitere Versuche:** Die Einführung weiterer Bedingungen erweitert den Suchraum nach einer möglichen Anordnung der Level erheblich. Die Ermittlung des Optimums bedarf einer $n^2$–komplexen Suche für jede zu splittende Kette, wobei $n$ die Anzahl der Linklevel ist. Bei möglichen Linklevelanzahlen $n > 100$ und bis zu 1000 zu untersuchende Ketten steigen Zeit– und Speicheraufwand in nicht mehr zu realisierende Bereiche. Es wird also nach einem Algorithmus gesucht, der eine "fast optimale" Lösung in "vertretbarer" Zeit ermittelt.

### 3.2.4 Der Split–Algorithmus

**Eine Lösung:** Anhand der Beispielvernetzung CUBE768 (Abbildung 14) soll der Mechanismus des gewählten Algorithmus demonstriert werden. Gerechnet wurde auf 16 Prozessoren mit einem Verfeinerungsschritt[5] und linearer Verteilung der Elemente auf die Prozessoren. Da die KettIDs hier keine Rolle spielen, sollen nur die bei der Splittung ermittelten Level dieser KettIDs berücksichtigt werden.

[5] Die Anzahl der Verfeinerungsschritte spielt für den Split–Algorithmus keine Rolle, da sich die Liste der Referenzketten nicht unterscheidet (im Unterschied zur Kettenanzahl insgesamt).

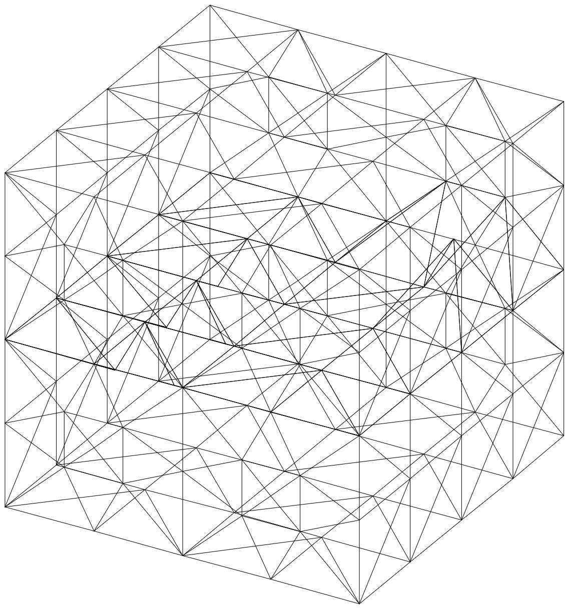


Abbildung 14: Beispielvernetzung Cube768

Zu Beginn des Algorithmus werden zwei Hilfsfelder der Länge $maxlevel$[6] zu 0 initialisiert. Das erste Feld enthält nach Beendigung des Algorithmus die sogenannte *Linkebene*[7] des ursprünglichen Levels, das zweite Feld beschreibt die *relative Verschiebbarkeit*[8].

Die Splittung der ersten Hypercube–Kette ergibt zwei Paare mit den originalen Linkleveln 6, 6 und 4, 8. Alle ermittelten Level sind noch nicht in den Hilfsfeldern erfaßt, Level 6 wird demnach in die neue Linkebene 1 eingeordnet, Level 4 und 8 erhalten die Linkebene 2. Zusätzlich wird der Eintrag für Level 6 in der *Verschiebungsliste* auf 1 gesetzt, was besagt, daß dieses Linklevel niemals die Position relativ zu einem Linklevel einer höheren Linkebene ändern darf (der Nutzen wird weiter unten erklärt). Abbildung 15 verdeutlicht den ersten Schritt.

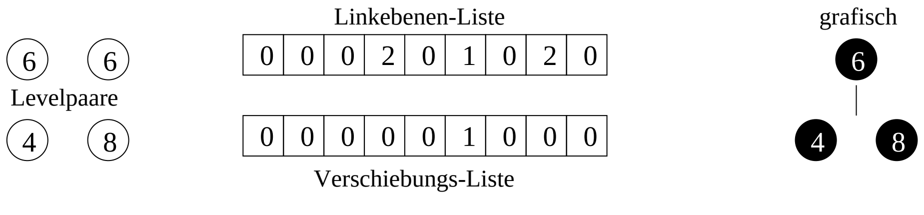



Abbildung 15: Schritt 1 des Algorithmus

Die zweite Hypercube–Kette zerfällt in Kettenpaare mit den Linkleveln 6, 6

[6] Von Prozeß 0 ermittelt, im Beispiel = 9.

[7] Zwei Linklevel besitzen dieselbe Linkebene, wenn ihre relative Reihenfolge zueinander nicht vorgeschrieben ist.

[8] wird im Kontext erläutert

bzw. 5 und 9. Level 5 und 9 werden demnach eine Linkebene höher als Level 6 eingeordnet (Abbildung 16).

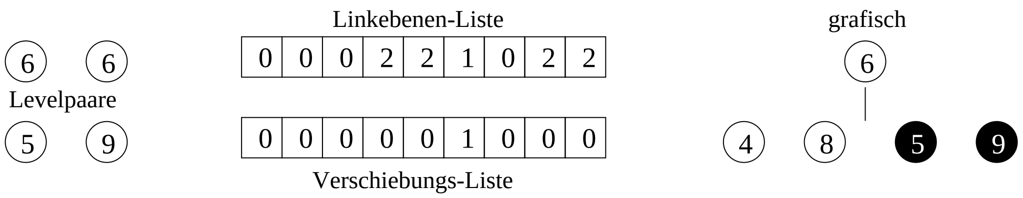



Abbildung 16: Schritt 2 des Algorithmus

Die dritte Aufspaltung liefert für das erste Kettenpaar das Level 4 und für das zweite Level 5. Beide Level wurden bereits eingeordnet, jedoch in derselben Linkebene. Die gewünschte Reihenfolge erhält man nun, indem man *alle* höheren Linkebenen um *eine* Linkebene erhöht (in diesem Beispiel existiert noch keine solche Ebene) und Level 5 in die nächste Ebene verschiebt (Abbildung 17). Level 4 erhält in der Verschiebungsliste eine Markierung.


Abbildung 17: In Schritt 3 des Algorithmus wird eine Linkebene geteilt, um die geforderte Reihenfolge herzustellen.

Zwei weitere Schritte des Algorithmus erzeugen die in Abbildung 18 gezeigten Listen.


Abbildung 18: Schritt 4 und 5 des Algorithmus.

Die sechste Hypercube–Kette zerfällt in Levelpaare 8, 8 bzw. 9, 9. Beide Level liegen wiederum in ein und derselben Linkebene. Level 9 wandert also eine

Ebene nach oben, wie auch alle anderen höheren Linkebenen (im Beispiel nur die Ebene 3). In weiteren Durchläufen des Split–Algorithmus zerfallen insgesamt 16 Hypercube–Ketten der Dimension 2, wobei keine weiteren Änderungen der Listen notwendig werden. Abbildung 19 zeigt also die endgültigen Listen.

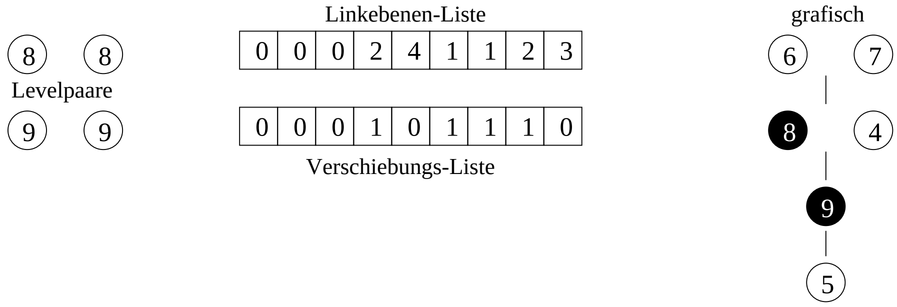



Abbildung 19: Schritt 6 erzeugt die letzte Änderung in den Listen, insgesamt zerfallen 16 Hypercube–Ketten.

Noch eine Anmerkung zur Verschiebungsliste, deren Einträge im obigen Beispiel keine Verwendung fanden. Ausgangspunkt sei die in Abbildung 19 gezeigte Situation. Angenommen, eine Kette zerfalle z.B. in die Levelkombinationen 4, 5 und 9, 9. Mit dem oben demonstrierten Mechanismen allein wäre keine Splittung der Kette möglich, da alle Level bereits in eine Ebene eingeordnet wurden und die Linkebene von Level 9 zwischen den Ebenen der anderen beiden Level liegt. Eine genaue Betrachtung des Sachverhaltes zeigt aber, daß Level 9 völlig unabhängig von Level 5 ist, demzufolge "hinter" Level 5 verschoben werden könnte. Genau diese Möglichkeit wird durch die Verschiebungsliste repräsentiert. Ein "0"–Eintrag besagt, daß dieses Level aus seiner bisherigen Linkebene beliebig nach hinten verschoben werden kann, also von allen Leveln, die einer höheren Ebene zugeordnet wurden, unabhängig ist.

Die Verschiebungsliste ist als eine Art Kompromiß zwischen einer möglichst aufwandsarmen Suche nach einer Levelzuordnung einerseits und einer möglichst optimalen Einordnung andererseits zu sehen.

Bei der Berechnung auf höheren Prozessorzahlen treten oft auch Hypercube–Ketten größerer Dimensionen (3 und 4) auf. Beim Splitten einer solchen Kette werden nur die korrespondierenden Subhypercube–Ketten betrachtet, die wiederum in einem späteren Schritt des Split–Algorithmus auf ihre Zerlegbarkeit hin untersucht werden könnten. Je Schritt versucht man also maximal vier Level in eine bestimmte Reihenfolge zu bringen, eine erste Rekursion müßte demzufolge maximal $4 \cdot 4 = 16$ Level berücksichtigen und eine dritte Verfeinerung bereits bis zu 64.

Ist die Wahrscheinlichkeit zum Auffinden einer korrekten Anordnung von 16 Linkleveln eher gering, so scheint sie für 64 Level mehr als fraglich (derartige Größenordnungen von Leveln treten ohnehin nur bei ungewöhnlichen Gebietsverteilungen[9] auf). Aber selbst ein einzelner Rekursionsschritt erhöht den Suchaufwand erheblich, so daß in obigem Algorithmus eine einmal verwendete Subhypercube–Kette nicht selbst zerlegt werden darf[10]. Abschließend sei darauf verwiesen, daß die Anzahl der Linkebenen die Anzahl von Prozeß 0 berechnender Linklevel nicht übersteigen kann.

Nachdem nun alle Hypercube–Ketten auf ihre Teilbarkeit hin untersucht wurden, erfolgt eine Neueinordung der KettIDs in die Linklevel:

Bemerkt sei nochmals, daß alle Prozesse die gleiche Sicht auf alle Referenzketten haben, die folgende lokal vorgenommene Linklevel–Neueinordnung also zu identischen Ergebnissen auf allen Prozessen führt.

Warum überhaupt eine erneute Generierung der Linklevel Sinn macht, begründet sich aus der Tatsache, daß durch den Wegfall von zahlreichen Ketten eine Reduzierung der Level zu erwarten ist, da anstelle dieser Kommunkationen andere unabhängige Kommunikationsmuster in niedrigere Linklevel eingeordnet werden können.

Zur Erläuterung der Vorgehensweise soll nochmals die Abbildung 19 (Seite 17) dienen. Von Interesse sind die Linkebenen–Liste und die reduzierte Liste der Referenzketten, also ohne die gesplitteten Hypercube–Ketten.

Der Index der Linkebenen–Liste[11] entspricht dem originalen Linklevel, so wie es von Prozeß 0 berechnet wurde, ein 0–Eintrag bedeutet die völlige Unabhängigkeit des Levels von allen anderen und ein Eintrag $> 0$ besagt, daß alle Ketten der Linkebene $i$ in einem niedrigeren Linklevel erscheinen müssen, als die erste Kette aus der Linkebene $i + 1$.

Der Algorithmus startet nun mit Linklevel 1 und ordnet dieses den disjunkten Ketten[12] aus der ersten Linkebene zu. Läßt sich keine disjunkte Kette mehr finden, wird versucht, das Linklevel durch eine unabhängige Kette (Linkebene 0) zu ergänzen. Ist auch diese Möglichkeit erschöpft, wird das Linklevel inkrementiert und mit Ketten der ersten Linkebene fortgefahren, solange, bis allen Ketten dieser Ebene ein Linklevel zugeordnet wurde.

Nach erneuter Erhöhung des Linklevels wird mit den Ketten der zweiten Linkebene nach gleichem Schema verfahren, usw...

Das weitere Verfahren ist analog zum unter Abschnitt 2 beschriebenem getcomm–Algorithmus.

[9] z.B. zufällige Verteilung

[10] Betrachtungen verschiedener Tetraeder–Vernetzungen zeigen, daß der Algorithmus auch ohne solche Rekursionen eine fast optimale Levelzuordung liefert (Der Grad der Belegung der Linkebenen– und Verschiebungs–Liste kann als Maß dafür genommen werden.).

[11] beginnend mit 1

[12] Ketten mit disjunkter Prozeßzugehörigkeit

## 3.3 Effizienz des Verfahrens

*Entscheidender Faktor für die Effizienz des Verfahrens ist die Güte des Split–Algorithmus*, d.h. das Verhältnis von splittbaren zu tatsächlich gesplitteten Ketten. Zwei charakteristische Tetraedernetze sollen stellvertretend betrachtet werden. Zum einen *cube768* (Abbildung 14), bestehend aus 768 Tetraederelementen, deren regelmäßige Anordnung das Vorhandensein von Hypercube–Ketten bei der Verteilung auf die Prozessoren begünstigt und – als das andere Extrem – *spc3-123* (Abbildung 20), bestehend aus 1398 Tetraederelementen, als ein Beispiel für eine stark diskontinuierliche Anordnung. Die nachfolgenden Resultate beschränken sich auf die Verwendung der linearen Verteilung der Tetraederelemente auf die Prozessoren, prinzipiell lassen sich die Aussagen aber auf andere Verteilungen übertragen.

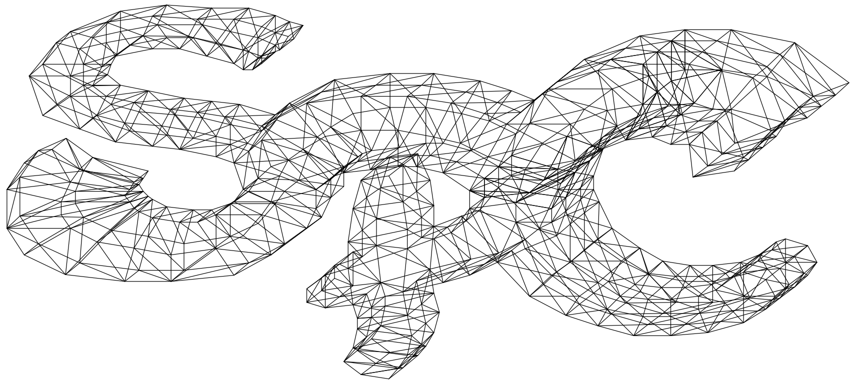


Abbildung 20: Das Netz spc3-123 besteht aus 1398 Tetraederelementen.

*cube768*: Nachfolgende Tabelle faßt das Verhältnis der Anzahl Hypercube–Ketten zur Anzahl gesplitteter Ketten[13] sowie das Verhältnis von Linklevelanzahlen ohne bzw. mit Anwendung des Split–Algorithmus für 8 bis 64 Prozessoren zusammen.

| **Anzahl Prozessoren** | **untersuchte Ketten** | **davon gesplittet** | **Level ohne / mit Splittung** |
|---|---|---|---|
| 8 | 6 | 6 | 6 / 3 |
| 16 | 16 | 16 | 9 / 5 |
| 32 | 40 | 31 | 20 / 16 |
| 64 | 120 | 58 | 23 / 23 |

Für 8 bzw. 16 Prozessoren zerfallen alle Hypercube–Ketten der Dimension 2. Wie bereits erwähnt, belegten diese Ketten vor der Splittung vollständig eigene Linklevel (3 Level für 6 Hypercube–Ketten bei 8 Prozessoren bzw. 4 Level für

[13] Die angegebenen Werte beziehen sich auf die Anzahl der Referenzketten.

16 Hypercube–Ketten bei 16 Prozessoren), woraus auch die Reduzierung dieser Linklevel resultiert. Mit zunehmender Prozessorzahl ändert sich das Verhältnis von gesplitteten zu nicht gesplitteten Ketten und damit minimiert sich die Verringerung der Linklevel, für 64 Prozessoren konnte die Anzahl der Level nicht gesenkt werden.

Untersuchungen der Hypercube–Ketten bei höheren Prozessoranzahlen ($\geq 32$) weisen als dominierende Ursache für den fehlgeschlagenen Split–Versuch fehlende Subhypercube–Ketten aus, d.h. die Ketten lassen sich nicht durch entsprechende Kommunikationsmuster reproduzieren. Der Fall der inkonsistenten Linklevelreihenfolge tritt verhältnismäßig selten auf (0...20%), wobei der Anteil nahezu linear zur Prozessoranzahl steigt.

*spc3-123*: Ein analoges Verhalten zum Fall der regelmäßigen Netzanordnung von cube768 zeigt auch die *spc3-123*–Vernetzung:

| **Anzahl Prozessoren** | **untersuchte Ketten** | **davon gesplittet** | **Level ohne / mit Splittung** |
|---|---|---|---|
| 8 | 3 | 3 | 15 / 12 |
| 16 | 13 | 11 | 27 / 20 |
| 32 | 50 | 42 | 33 / 33 |
| 64 | 65 | 38 | 33 / 31 |

Mit steigender Prozessorzahl sinkt die Wahrscheinlichkeit der Splittung aller vorhandener Hypercube–Ketten, dennoch ist auch bei 64 Prozessoren eine Verringerung der Linklevel zu verzeichnen. Diese Eigenschaft erklärt sich aus der komplexen Kommunikationsstruktur und der daraus folgenden (im Verhältnis zum cube768) geringen Auslastung der Linklevel[14]. Der Wegfall einer Kommunikationsstruktur aus einem Level ermöglicht häufig die Hinzunahme eines neuen (oder mehrerer neuer) Musters; Muster aus höheren Leveln "wandern" in "frühere" Level und höhere Level entfallen u.U. gänzlich.

Allgemein läßt sich feststellen, daß der Laufzeitgewinn weniger aus einer verringerten Levelanzahl resultiert, sondern vielmehr die Anzahl gesplitteter Kommunikationsmuster die Effizienz positiv beeinflußt. Dieser Effekt ist auch logisch, bestimmt doch die größte Kommunikatorgruppe die Dauer der Abarbeitung eines Linklevels und eben diese großen Gruppen werden durch den Split–Algorithmus weitgehend eliminiert.

Abbildung 21 zeigt die gemessenen Zeiten für zwei beispielhafte Tetraedervernetzungen. Für *cube768* liefert der Split–Algorithmus die besseren Ergebnisse, dies leitet sich aus der regelmäßigen Struktur der Vernetzung und der damit verbundenen Existenz zahlreicher Hypercube–Ketten zusammen mit ihren

[14] Der Parallelitätsgrad der Kommunikation ist relativ gering, in manchen (hohen) Leveln wird nur ein einziges Muster bearbeitet.

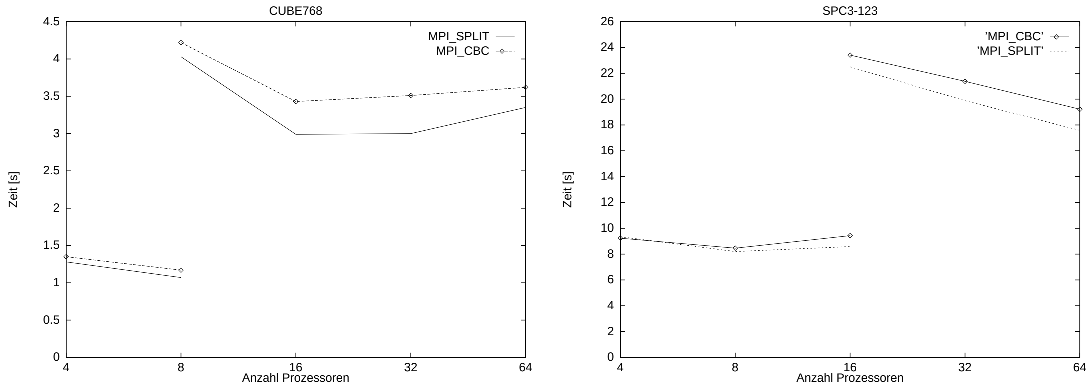



Abbildung 21: Zeiten für die Tetraedervernetzungen *cube768* und *spc3-123* der originalen Variante (MPI_CBC) und der neuen Version (MPI_SPLIT). Gerechnet wurde für 4–16 Prozessoren mit 2 und für 16–64 Prozessoren mit 3 Verfeinerungsschritten.

Subhypercube–Ketten ab, aber auch für *spc3-123* sind signifikante Laufzeitgewinne zu verzeichnen.

Abbildung 22 bestätigt die vermutete Effizienzsteigerung bei Quadernetzen, deren Elementestruktur das Entstehen großer Kommunikatorgruppen begünstigt.

# 4 Einfluß Topologie–optimierter Partitionierung im 3D–Fall

Als Weiterführung der Arbeiten in [9, 2] wurde der Einfluß von Topologie–optimierten Partitionierungen in Zusammenhang mit dem Mapping von Kommunikatoren untersucht. In Anlehnung an obige Arbeiten wurden die beiden Vernetzungen *cube768* und *spc3–123* betrachtet. Für die Vernetzung *cube768* wurde bei Verwendung von 2 und 8 Prozessoren mit 2 Verfeinerungsschritten gerechnet und von 8 – 64 Prozessoren mit 3 Schritten. Für *spc3–123* wurde bei 8 – 16 Prozessoren mit 2, bei 16 – 64 Prozessoren mit 3 Verfeinerungsschritten gearbeitet.

An dieser Stelle wird auf eine Beschreibung der einzelnen Verfahren verzichtet, die theoretischen Grundlagen findet der interessierte Leser in [9, 6, 1, 10].

## 4.1 Partitionierung mit chaco

Die Ergebnisse der bisherigen Untersuchengen sollen noch einmal kurz zusammengefaßt werden.

Von den spektralen Verfahren wurde in [9] nur die Bisektion als zuverlässiger effizienter Algorithmus bewertet, für Quadrisektion und Oktasektion gelangte der

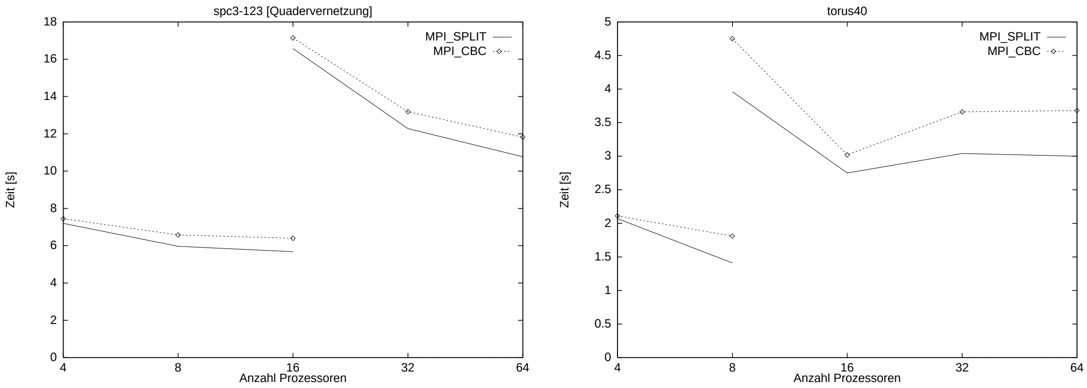


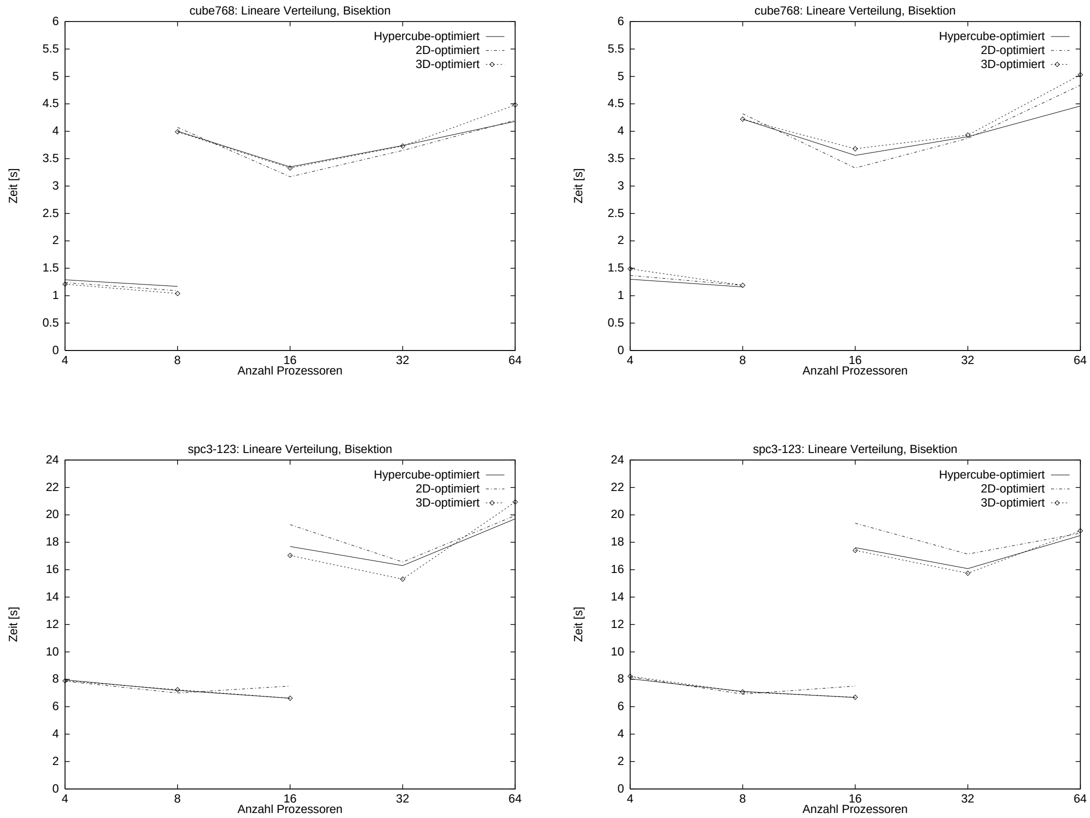

Abbildung 22: Zeiten für die Quadervernetzung *spc3-123* der originalen Variante (MPI_CBC) und der neuen Version (MPI_SPLIT). Gerechnet wurde für 4–16 (Torus40 = 8) Prozessoren mit 2 und für 16 (Torus40 = 8) –64 Prozessoren mit 3 Verfeinerungsschritten.

Autor zu stark netzspezifischen und von der Prozessoranzahl abhängigen Ergebnissen. Effizienzbetrachtungen in [2] bestätigten dieses Verhalten für die Quadrisektion, wobei die Oktasektion in Verbindung mit dem in [4] beschriebenem Verfahren zu durchaus befriedigenden Ergebnissen führte. Letztere soll deshalb neben der Bisektion nachfolgend betrachtet werden.

Sowohl [9], als auch [2] stellten die Sinnfälligkeit der *Erhöhung innerer Ecken* in Frage, weshalb diese chaco–Option für folgende Messungen abgestellt wurde (INTERNAL_VERTICES = False). Der enorme Nutzen einer globalen KL–Nachbesserung (Kernighan–Lin) im Postprocessing, der ebenso in den beiden Arbeiten nachgewiesen wurde, führt zur Anwendung dieser, wobei mit 10 Iterationsschritten (REFINE_PARTITION = 10) gearbeitet wurde. Ebenso war die Option der Architekturoptimierung aktiv (REFINE_MAP = True), wobei die drei Anpassungen an eine Hypercube–Topologie, an ein 2D– und an ein 3D–Netz getestet wurden. Neben dem Vergleich der verschiedenen Optimierungen untereinander sollen nachfolgend die beiden auf dem Kommunikatorprinzip beruhenden Algorithmen gegenübergestellt werden. MPI_SPLIT bezeichnet dabei die Variante mit Aufspaltung von Hypercube–Ketten, MPI_CBC steht für die originale Version. Die originale Programmversion (ohne Kommunikatorprinzip) wird nicht betrachtet, da der dort verwendete Kommunikationsalgorithmus auf dem Hypercube basiert und eine diesbezügliche Optimierung bevorteilen würde.

### 4.1.1 Lineare Verteilung

In den Abbildungen 23 und 24 sind die lineare Verteilung in Verbindung mit der Bisektion bzw. Oktasektion als Partitionierungsmethode gegenübergestellt. In ersterem Fall wird die Elementemenge solange halbiert, bis deren Anzahl der




MPI_SPLIT MPI_CBC

Abbildung 23: Rechenzeiten für lineare Verteilung in Verbindung mit Bisektion.

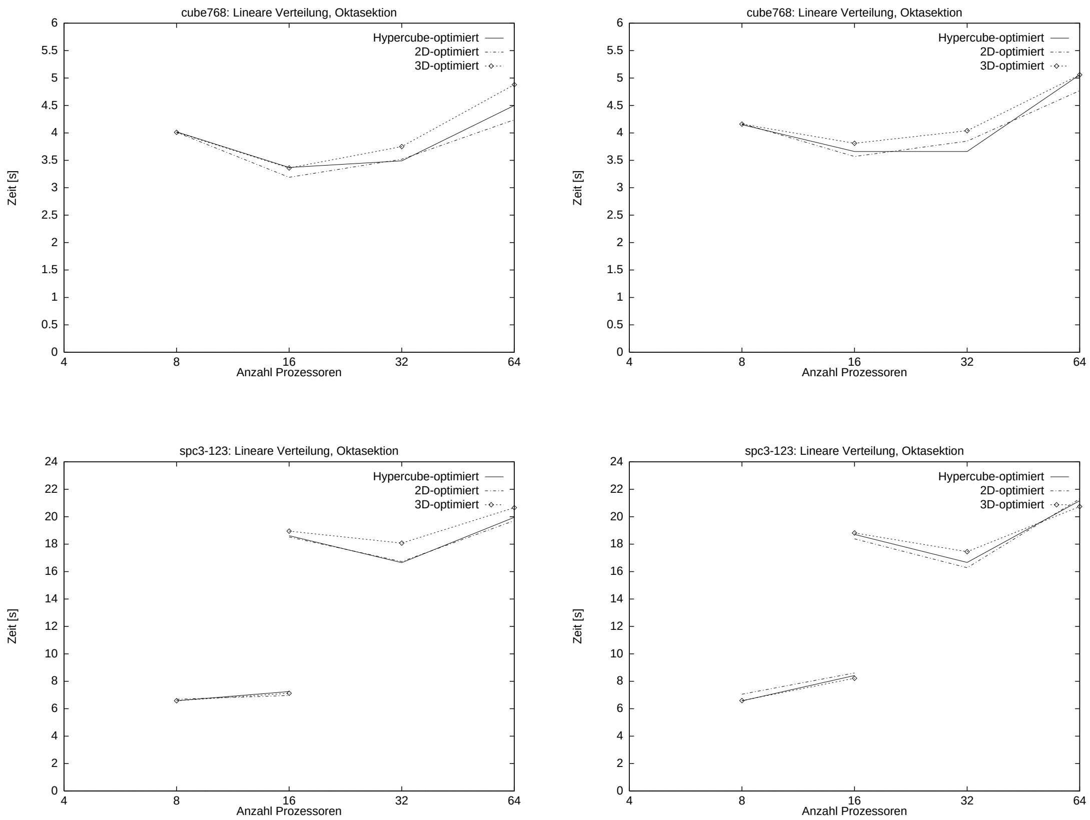
Anzahl der Prozesse entspricht, im anderen Fall wird die Menge in jedem Schritt gleichzeitig in 8 Teilmengen[15] zerlegt. Das Ergebnis sollte für beide Fälle identisch sein, die dennoch erkennbaren Laufzeitunterschiede begründen sich durch Meßungenauigkeiten und (vermuteter) partitionsabhängiger Kommunikationsleistung auf dem PowerGC.

Auch die Anwendung von Terminal Propagation (Abbildung 25) bringt keine wesentlichen Laufzeitverbesserungen, einzig die Hypercube–Anpassung wird hier insbesondere unterstützt. Alleinige Betrachtung der linearen Verteilung läßt in nahezu allen Fällen die 3D–Optimierung als ineffizient erscheinen, im Mittel liefert die Hypercube-Anpassung die besten Verteilungen.

MPI_SPLIT erzielt verglichen mit MPI_CBC geringfügig bessere Laufzeiten, wobei bei der *spc3-123*–Vernetzung kaum noch Unterschiede erkennbar werden. Im Unterschied zu einer rein linearen Verteilung wurde in allen hier dargestell-

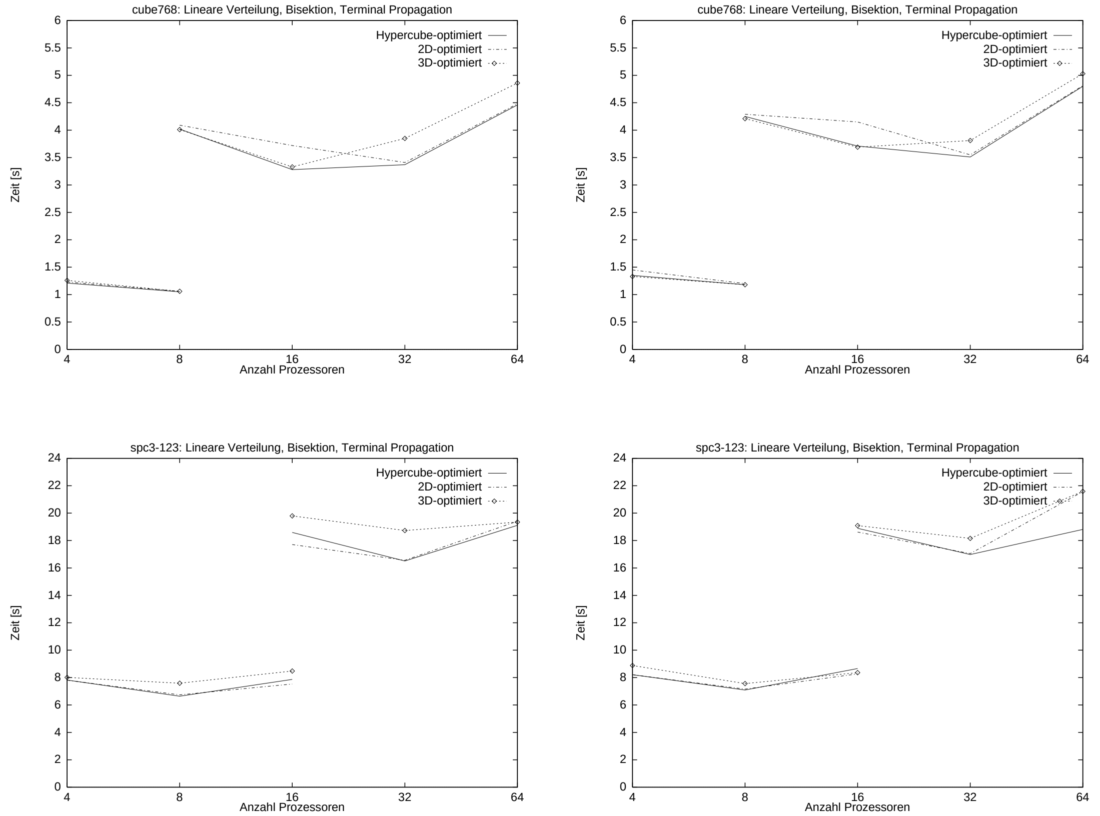
[15] Die Oktasektion ist also erst ab einer Hypercubedimension 3 möglich; wird diese Partitionierungsmethode auf eine kleinere Dimension angewandt, wechselt **chaco** selbstätig zur Quadri– bzw. Bisektion.

ten Partitionierungen in jedem Rekursionsschritt lokal mittels Kernighan Lin nachgebessert. Eine Eigenschaft des Algorithmus ist die Verringerung der Kommunikationsbeziehungen, so daß zwangsläufig die Anzahl der Ketten sinkt, an denen relativ viele Prozesse beteiligt sind (also vermindert sich auch die Anzahl der Hypercube–Ketten). Der Nutzen des Split–Algorithmus' schwindet somit.




MPI_SPLIT MPI_CBC

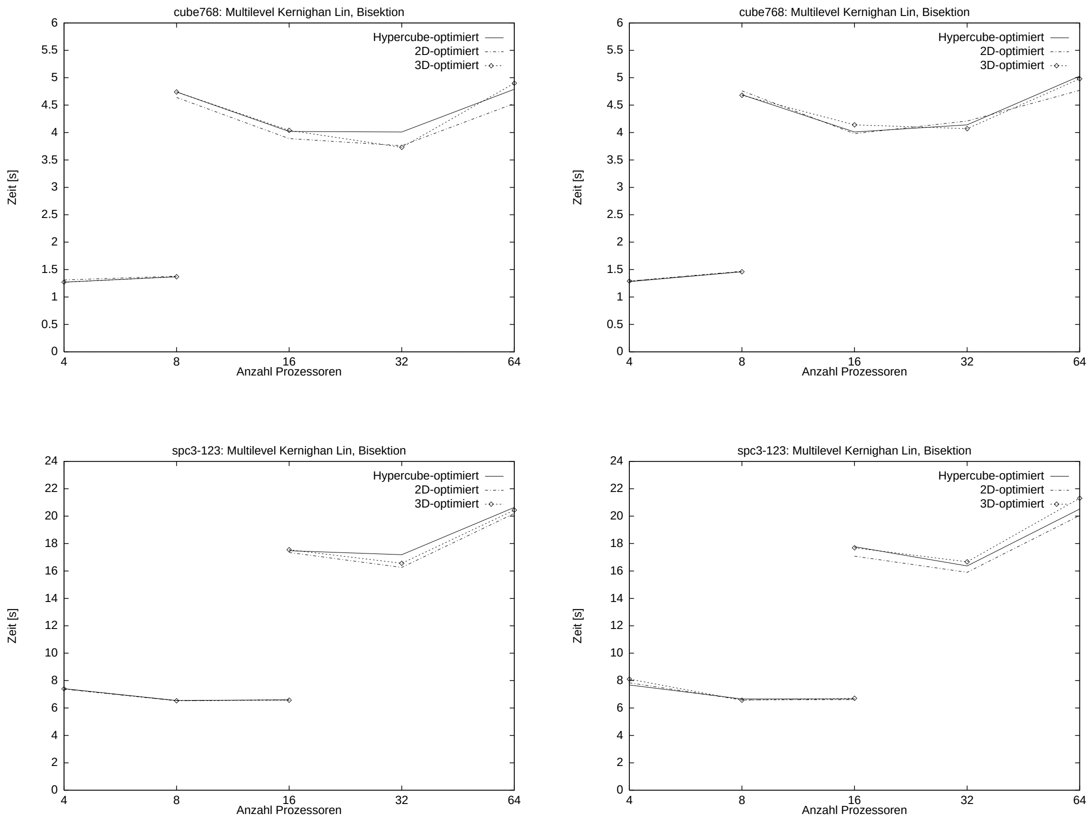
Abbildung 24: Rechenzeiten für lineare Verteilung in Verbindung mit Oktasektion.




MPI_SPLIT MPI_CBC

Abbildung 25: Rechenzeiten für lineare Verteilung mit Bisektion und Terminal Propagation.

### 4.1.2 Multilevel–Kernighan Lin




MPI_SPLIT MPI_CBC

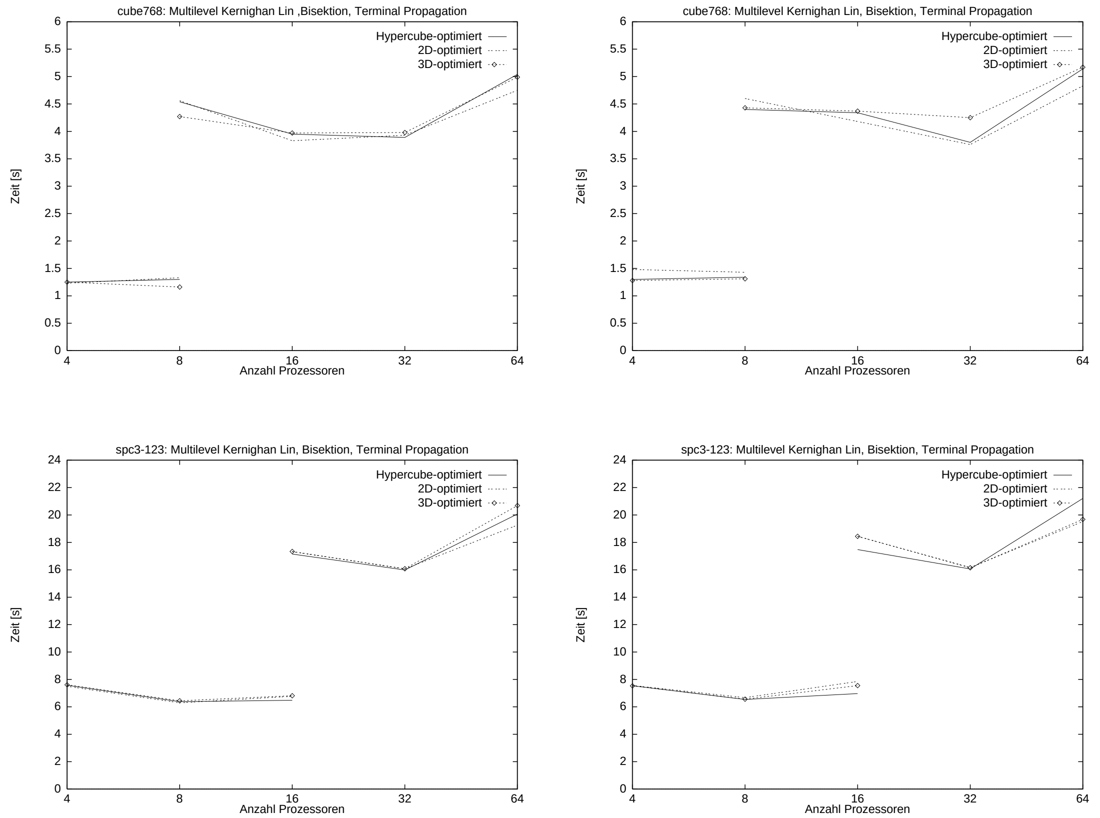
Abbildung 26: Rechenzeiten für Multilevel Kernighan Lin in Verbindung mit Bisektion.

Die Stärke dieser Partitionierungsmethode liegt eindeutig in der kurzen Laufzeit des Partitionierungsvorganges selbst. Besonders für großdimensionierte Graphen (> 10000 Knoten) ist die Zeitersparnis gegenüber den anderen hier vorgestellten Methoden signifikant. Da die Gebietsaufteilungen aber im Preprocessing stattfinden, spielt dieser Faktor für die Gesamteffizienz keine Rolle. Multilevel KL liefert sowohl für das regelmäßige Netz *cube768* als auch für die unregelmäßige *spc3-123*–Vernetzung schlechtere Resultate als die spektralen Methoden, liegt für den letzteren Fall allerdings noch vor den linearen Verteilungen.

Bezüglich der Architekturoptimierung verhält sich Multilevel KL eher neutral. Bei Partitionierung mittels Bisektion mit und ohne Terminal Propagation liegen die Zeiten für Hypercube–, 2D– und 3D–Anpassung relativ dicht beieinander, die Verwendung von Oktasektion läßt zumindest für *spc3-123* wiederum die 3D–Optimierung negativ erscheinen (Abbildungen 26, 27, 28).

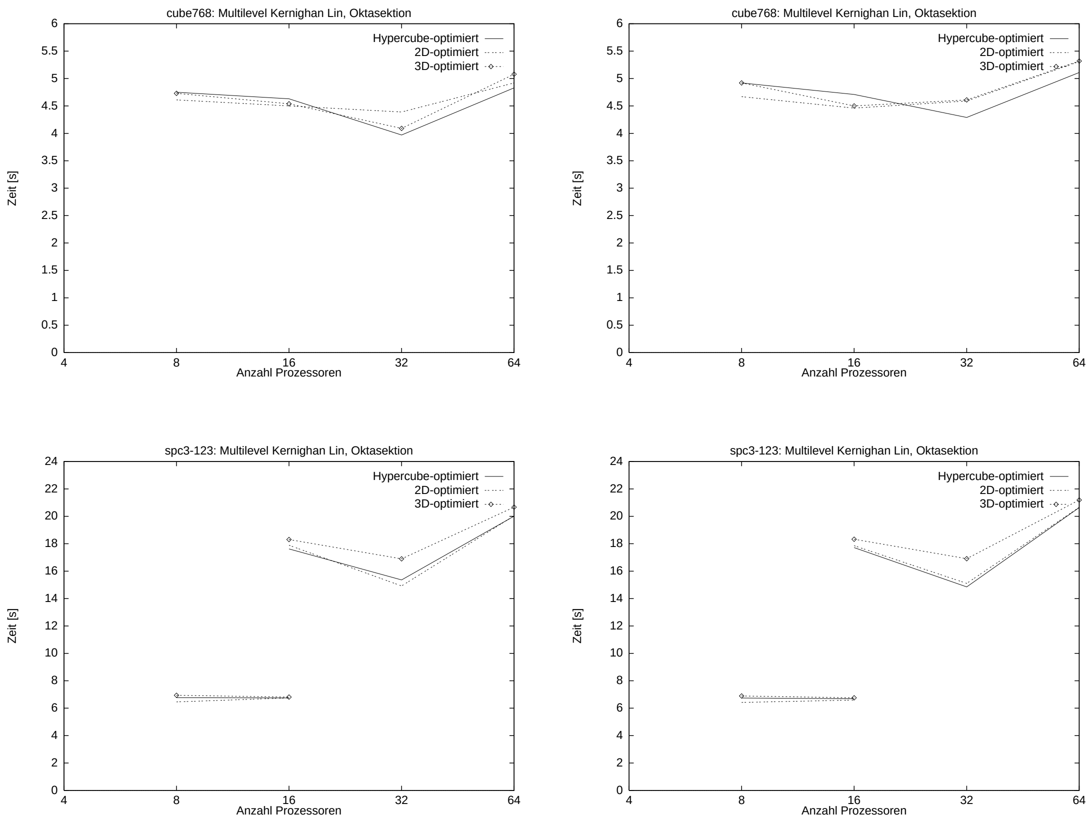
MPI_SPLIT und MPI_CBC zeigen analoges Laufzeitverhalten, wobei bei *cu-*

*be768* MPI_SPLIT leichte Vorteile erzielt.




MPI_SPLIT MPI_CBC

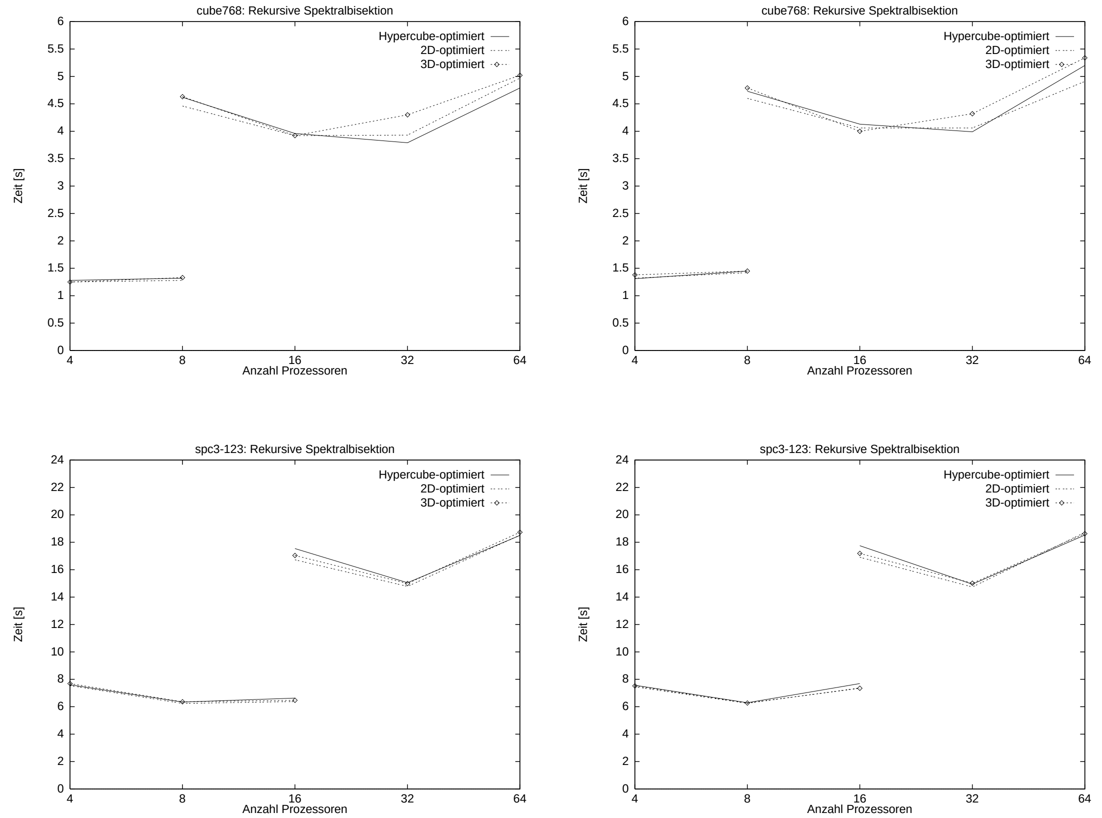
Abbildung 27: Rechenzeiten für Multilevel Kernighan Lin in Verbindung mit Bisektion und Terminal Propagation.




MPI_SPLIT MPI_CBC

Abbildung 28: Rechenzeiten für Multilevel Kernighan Lin in Verbindung mit Oktasektion.

### 4.1.3 Spektrale Methoden




MPI_SPLIT MPI_CBC

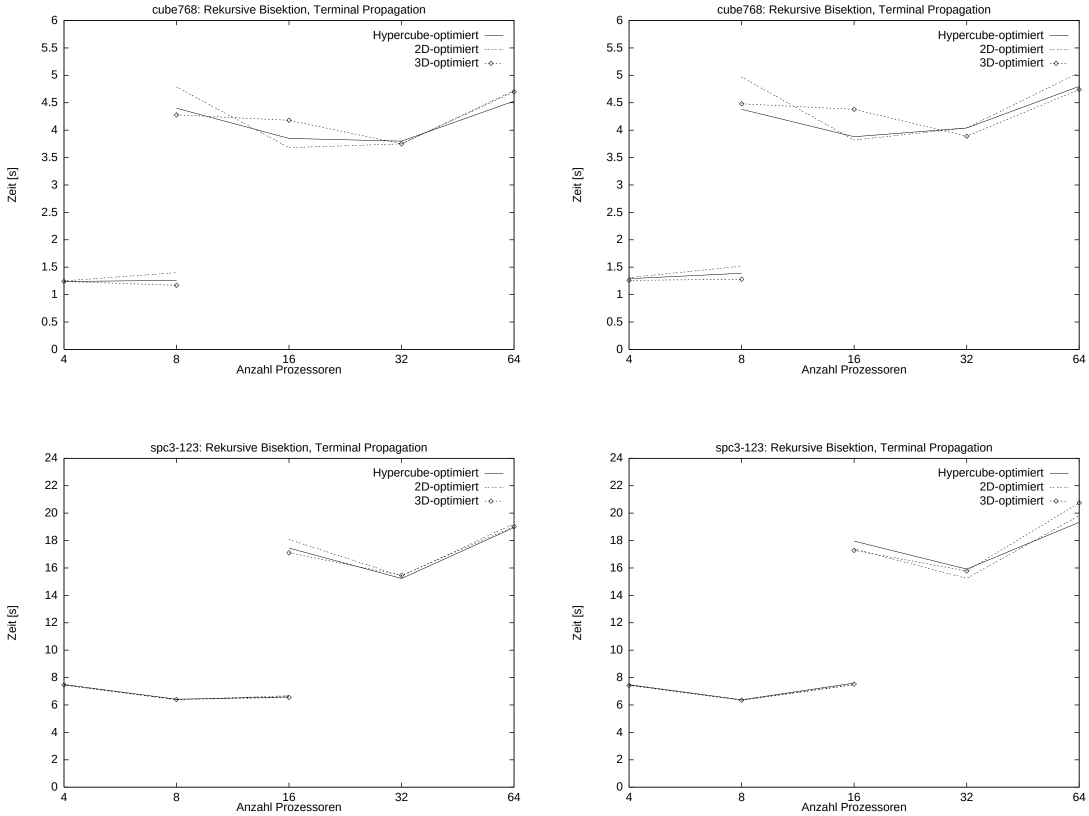
Abbildung 29: Rechenzeiten für rekursive Spektralbisektion.

Die gemessenen Resultate verdeutlichen den Unterschied zwischen einer relativ regelmäßigen und einer extrem unregelmäßigen Vernetzung. Während für letzteren Fall Rekursive Bisektion ohne und mit Terminal Propagation sowie die Rekursive Oktasektion ein sehr homogenes Laufzeitverhalten aufweisen, sind die Kurven für den regelmäßigen Fall teils beträchtlichen Schwankungen unterworfen.

Eine Analyse der Verteilungen für das *spc3-123*–Netz zeigt für alle drei Varianten die Existenz keiner oder nur weniger (höhere Prozeßanzahl) großer Kommunikationsgruppen[16], so daß ein hoher Grad an Parallelität bei der Koppelrandkommunikation erreicht wird. Gleichzeitig schwindet der Nutzen des Split–Algorithmus', die dennoch erkennbaren Laufzeitunterschiede sind in der abgewandelten Kommunikationsreihenfolge und der damit eventuell veränderten Linklevelanzahl zu vermuten.

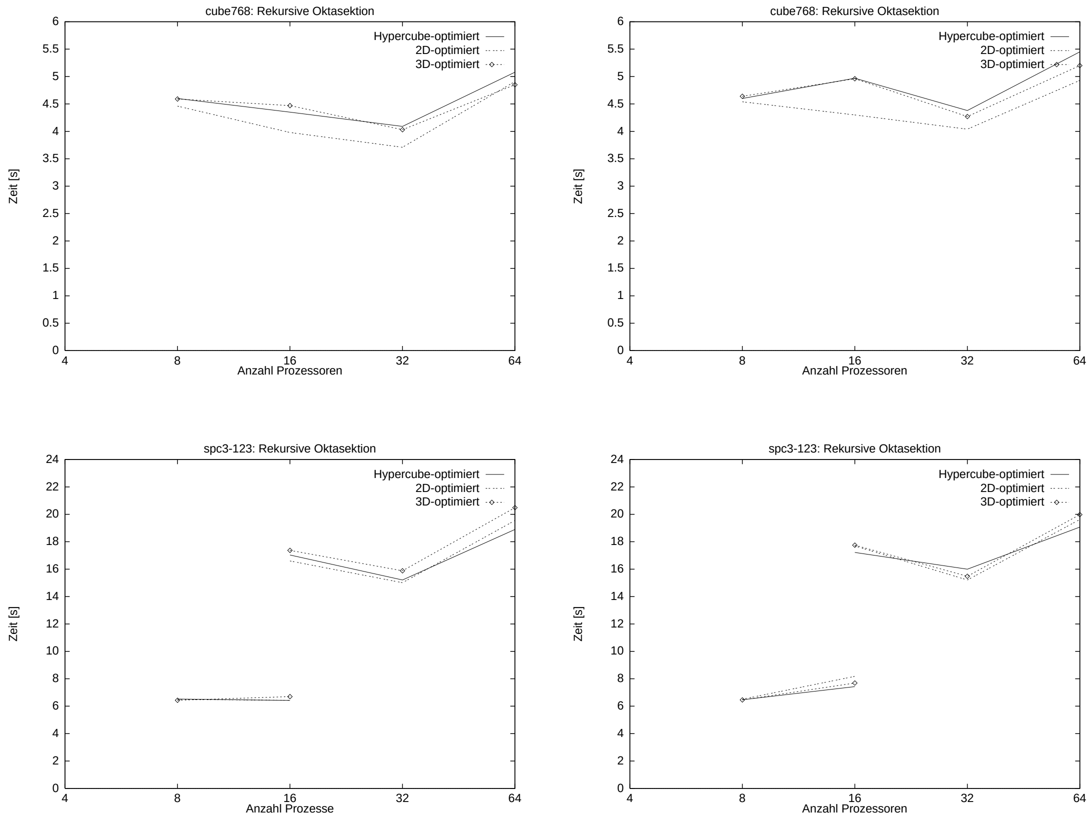
Für *cube768* zeigen 2D– und 3D–Optimierung ein offensichtlich von der verwendeten Prozessoranzahl beeinflußtes Laufzeitverhalten.

[16]Bei Verwendung von weniger als 16 Prozessoren teilen sich maximal 3 Prozesse eine Kette.




MPI_SPLIT MPI_CBC

Abbildung 30: Rechenzeiten für rekursive Spektralbisektion in Verbindung mit Terminal Propagation.




MPI_SPLIT MPI_CBC

Abbildung 31: Rechenzeiten für rekursive Oktasektion.

### 4.1.4 Die Verfahren im direkten Vergleich

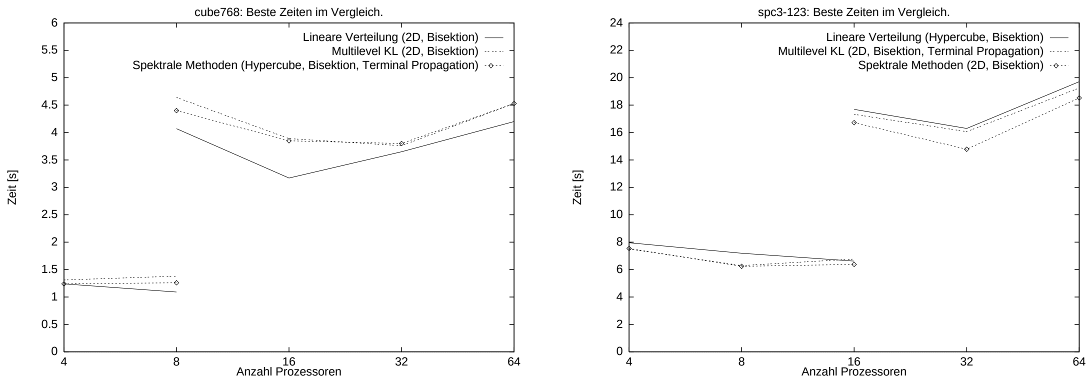



Abbildung 32: MPI_SPLIT: Die jeweils schnellsten Zeiten für Lineare Verteilung, Multilevel KL und Rekursive Methoden im Vergleich (gemittelt über die Summe der Einzelzeiten).

Abbildung 32 zeigt die jeweils schnellsten Zeiten der drei Verfahren Lineare Verteilung, Multilevel KL und Rekursive Methode für MPI_SPLIT.

Noch einmal werden die Ergebnisse der Arbeiten [9, 2] bestätigt: Lineare Verteilung liefert die besten Resultate für regelmäßige Vernetzungen, während Rekursive Spektralbisektion die Zeiten eines unregelmäßigen Netzes optimal reduziert. In beiden Fällen erscheint Multilevel Kernighan Lin als ein wenig geeignetes Partitionierungsverfahren, die Partitionierungen erreichen nicht annähernd die Qualität der jeweils besten Verteilungen.

Die schon bisher vermutete schlechte Eignung einer 3D–Optimierung wird in dieser Gegenüberstellung erst richtig deutlich: In keinem Fall erreicht eine solche Verteilung die höchste Effizienz.

Andererseits führt eine 2D–Anpassung bei den getesteten Tetraedervernetzungen zu den absolut besten Laufzeiten, Hyperube–optimierte Partitionierungen erreichen ebenso gute Ergebnisse.

MPI_CBC (Abbildung 33) weist eine ähnliche Charakteristik auf, auch hier ist die 2D–Optimierung die erfolgversprechendste Architekturanpassung (zumindest für den PowerGC).

Für beide Programmversionen liefert die Bisektion unabhängig von der gewählten Verteilungsmethode bessere Partitionierungen als die Oktasektion, in einigen Fällen kann Terminal Propagation die Resultate noch verbessern.

### 4.1.5 MPI_SPLIT und MPI_CBC im direkten Vergleich

Abbildung 34 zeigt neben den jeweils besten *chaco*–berechneten Partitionierungen die nichtoptimierte lineare Verteilung, so wie sie vom Programmodul *SPC–PM Po*

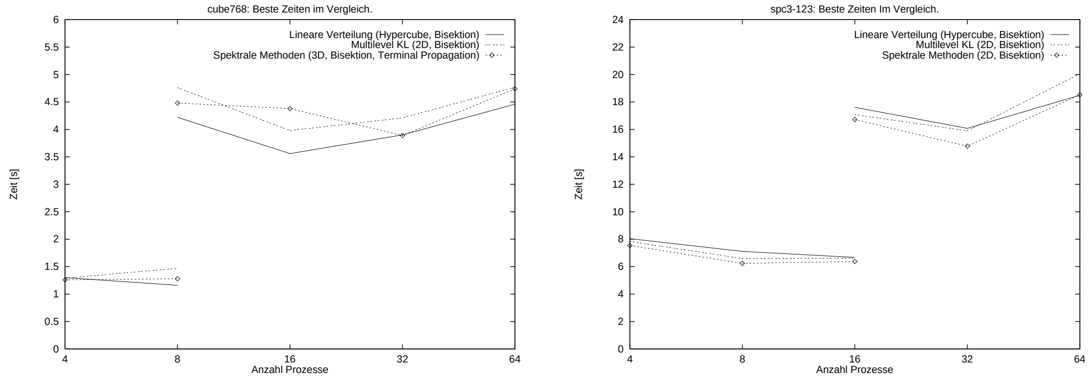



Abbildung 33: MPI_CBC: Die jeweils schnellsten Zeiten für Lineare Verteilung, Multilevel KL und Rekursive Methoden im Vergleich (gemittelt über die Summe der Einzelzeiten).

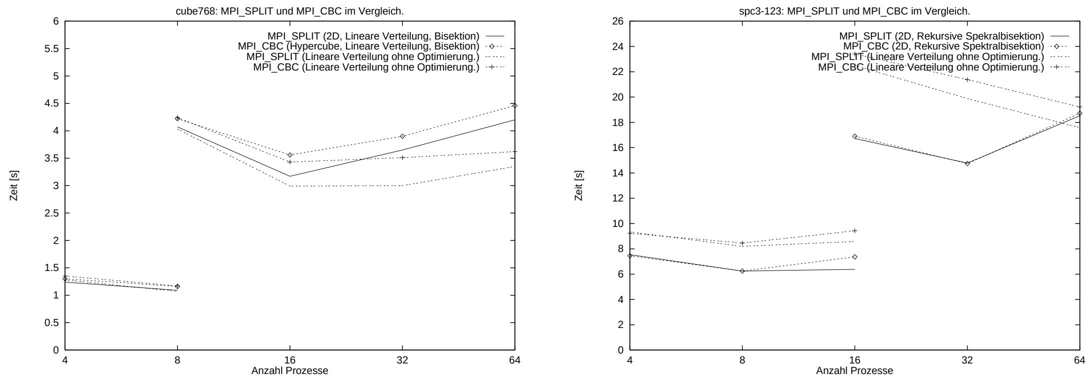



Abbildung 34: Die besten Optimierungen für MPI_SPLIT und MPI_CBC im Vergleich: Für die *cube768*–Vernetzung ist die nichtoptimierte lineare Verteilung das Optimum, hingegen reduziert die Rekursive Spektralbisektion die Zeiten für *spc3-123* erheblich.

*3D* erzeugt wird. Der abweichende Verlauf der Kurven des *spc3-123*–Netzes bei 64 Prozessoren erklärt sich durch die unterschiedlichen verwendeten Partitionen des PowerGC's während der Messungen.

*cube768* mit den 768 Tetraederelementen ist durch jede der verwendeten Prozessoranzahlen teilbar, d.h. jedem Prozeß wird bei der rein linearen Verteilung der Elemente der gleiche Anteil an Rechenaufwand zuteil. Die günstige Numerierung der Tetraeder und die regelmäßige Struktur der Vernetzung sind maßgebliche Faktoren, die diese Verteilung *für dieses Netz* stark begünstigen.

Im Gegensatz dazu benutzt die Rekursive Spektralbisektion keine Informationen über Numerierung und Aufbau der Vernetzung, so daß für den Fall einer stark unregelmäßigen Netzstruktur diese der linearen Partitionierung weit überlegen ist.

Noch eines wird im Diagramm des Netzes *spc3-123* deutlich: MPI_SPLIT reduziert nicht in jedem Fall die Laufzeit des Verfahrens. Wie schon erwähnt ist diese Aussage sehr netzspezifisch und die *spc3-123*–Vernetzung stellt mit ihren zahlreichen kleinen Kommunikationsbeziehungen (i.A. benutzen nur wenige Prozesse die gleiche Kette) eher eine Ausnahme dar.

# 5 Ausblick

Die oben dargestellte Mappingstrategie beweist ihre Wirksamkeit vor allem für nicht–optimale Verteilungen, so wie sie z.B. bei Einsatz adaptiver Netzverfeinerungen zur Laufzeit zu erwarten sind. Dennoch besitzt das Kommunikatorprinzip einen entscheidenden Nachteil: Der Aufwand zur Bildung dieser ist verhältnismäßig groß. Schaut man noch einmal auf Abbildung 2.3, Seite 6, so wird deutlich, daß einzig die Kommunikation zur Ermittlung der an einer Kette beteiligten Prozessoren für diesen Nachteil verantwortlich ist. Genau diese Kommunikation läßt sich vermeiden, indem die notwendigen Informationen zum Zeitpunkt der Netzgenerierung, also bei der Verteilung des Gebietes im Netzgenerator, erzeugt werden. Gegenwärtig wird diese Methodik exemplarisch im 2D–Fall eingesetzt [7].

Ein anderer Ansatzpunkt zur Optimierung wäre die Einbindung der Crosspointkommunikation in die Kommunikatorenstruktur. Analog zum in [3] beschriebenem Vorgehen, wäre hierfür eine lokale Berechnung des Grobgittersystems notwendig, d.h. im Unterschied zum bisherigen Verfahren müßte auf allen Prozessoren die Grobgittermatrix generiert werden. Da an einem Crosspoint meist mehr Prozessoren beteiligt sind als an Ketten, wären diese besonders geeignete Kandidaten für den Split–Algorithmus.

# Literatur


[1] T. Bui and C. Jones. A heuristic for reducing fill in sparce matrix factorization. In *Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing*, pages 445–452. SIAM, 1993.

[2] L. Grabowsky. MPI-basierte Koppelrandkommunikation und Einfluß der Partitionierung im 3D-Fall. Preprint SFB393/97-17, TU Chemnitz, August 1997.

[3] L. Grabowsky and T. Ermer. Objektorientierte Implementation eines PPCG-Verfahrens. Preprint SPC 98_13, TU Chemnitz-Zwickau, April 1998.

[4] L. Grabowsky, T. Ermer, and J. Werner. Nutzung von MPI für parallele FEM-Systeme. Preprint SFB393/97-08, TU Chemnitz-Zwickau, März 1997.

[5] G. Haase, T. Hommel, A. Meyer, and M. Pester. Bibliotheken zur Entwicklung paralleler Algorithmen. Preprint SPC 93_1, TU Chemnitz-Zwickau, Mai 1993.

[6] B. Hendrickson, R. Leland, and R. V. Driessche. Enhancing data locality by using terminal propagation. In *Proc. 29th Hawaii Conf. System Siences*, Jan. 1996.

[7] M.Meyer. Der hierarchische Netzgenerator Netgen69-C++. Preprint SPC 98_9, TU Chemnitz-Zwickau, April 1998.

[8] *MPI: A Message–Passing Interface Standard*, June 1995. available from http://www.mcs.anl.gov/Projects/mpi/index.html.

[9] U. Reichel. Partitionierung von Finite-Elemente-Netzen. Preprint SFB393/96-18, TU Chemnitz-Zwickau, November 1996.

[10] H. D. Simon. Partitioning of unstructured problems for parallel processing. In *Proc. Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications*. Pergammon Press, 1991.

Other titles in the SFB393 series:

96-01 V. Mehrmann, H. Xu. Chosing poles so that the single-input pole placement problem is well-conditioned. Januar 1996.

96-02 T. Penzl. Numerical solution of generalized Lyapunov equations. January 1996.

96-03 M. Scherzer, A. Meyer. Zur Berechnung von Spannungs- und Deformationsfeldern an Interface-Ecken im nichtlinearen Deformationsbereich auf Parallelrechnern. March 1996.

96-04 Th. Frank, E. Wassen. Parallel solution algorithms for Lagrangian simulation of disperse multiphase flows. Proc. of 2nd Int. Symposium on Numerical Methods for Multiphase Flows, ASME Fluids Engineering Division Summer Meeting, July 7-11, 1996, San Diego, CA, USA. June 1996.

96-05 P. Benner, V. Mehrmann, H. Xu. A numerically stable, structure preserving method for computing the eigenvalues of real Hamiltonian or symplectic pencils. April 1996.

96-06 P. Benner, R. Byers, E. Barth. HAMEV and SQRED: Fortran 77 Subroutines for Computing the Eigenvalues of Hamiltonian Matrices Using Van Loans's Square Reduced Method. May 1996.

96-07 W. Rehm (Ed.). Portierbare numerische Simulation auf parallelen Architekturen. April 1996.

96-08 J. Weickert. Navier-Stokes equations as a differential-algebraic system. August 1996.

96-09 R. Byers, C. He, V. Mehrmann. Where is the nearest non-regular pencil? August 1996.

96-10 Th. Apel. A note on anisotropic interpolation error estimates for isoparametric quadrilateral finite elements. November 1996.

96-11 Th. Apel, G. Lube. Anisotropic mesh refinement for singularly perturbed reaction diffusion problems. November 1996.

96-12 B. Heise, M. Jung. Scalability, efficiency, and robustness of parallel multilevel solvers for nonlinear equations. September 1996.

96-13 F. Milde, R. A. Römer, M. Schreiber. Multifractal analysis of the metal-insulator transition in anisotropic systems. October 1996.

96-14 R. Schneider, P. L. Levin, M. Spasojević. Multiscale compression of BEM equations for electrostatic systems. October 1996.

96-15 M. Spasojević, R. Schneider, P. L. Levin. On the creation of sparse Boundary Element matrices for two dimensional electrostatics problems using the orthogonal Haar wavelet. October 1996.

96-16 S. Dahlke, W. Dahmen, R. Hochmuth, R. Schneider. Stable multiscale bases and local error estimation for elliptic problems. October 1996.

96-17 B. H. Kleemann, A. Rathsfeld, R. Schneider. Multiscale methods for Boundary Integral Equations and their application to boundary value problems in scattering theory and geodesy. October 1996.

96-18 U. Reichel. Partitionierung von Finite-Elemente-Netzen. November 1996.

96-19 W. Dahmen, R. Schneider. Composite wavelet bases for operator equations. November 1996.

96-20 R. A. Römer, M. Schreiber. No enhancement of the localization length for two interacting particles in a random potential. December 1996. to appear in: Phys. Rev. Lett., March 1997

96-21 G. Windisch. Two-point boundary value problems with piecewise constant coefficients: weak solution and exact discretization. December 1996.

96-22 M. Jung, S. V. Nepomnyaschikh. Variable preconditioning procedures for elliptic problems. December 1996.

97-01 P. Benner, V. Mehrmann, H. Xu. A new method for computing the stable invariant subspace of a real Hamiltonian matrix or Breaking Van Loan's curse? January 1997.

97-02 B. Benhammouda. Rank-revealing 'top-down' ULV factorizations. January 1997.

97-03 U. Schrader. Convergence of Asynchronous Jacobi-Newton-Iterations. January 1997.

97-04 U.-J. Görke, R. Kreißig. Einflußfaktoren bei der Identifikation von Materialparametern elastisch-plastischer Deformationsgesetze aus inhomogenen Verschiebungsfeldern. March 1997.

97-05 U. Groh. FEM auf irregulären hierarchischen Dreiecksnetzen. March 1997.

97-06 Th. Apel. Interpolation of non-smooth functions on anisotropic finite element meshes. March 1997

97-07 Th. Apel, S. Nicaise. The finite element method with anisotropic mesh grading for elliptic problems in domains with corners and edges.

97-08 L. Grabowsky, Th. Ermer, J. Werner. Nutzung von MPI für parallele FEM-Systeme. March 1997.

97-09 T. Wappler, Th. Vojta, M. Schreiber. Monte-Carlo simulations of the dynamical behavior of the Coulomb glass. March 1997.

97-10 M. Pester. Behandlung gekrümmter Oberflächen in einem 3D-FEM-Programm für Parallelrechner. April 1997.

97-11 G. Globisch, S. V. Nepomnyaschikh. The hierarchical preconditioning having unstructured grids. April 1997.

97-12 R. V. Pai, A. Punnoose, R. A. Römer. The Mott-Anderson transition in the disordered one-dimensional Hubbard model. April 1997.

97-13 M. Thess. Parallel Multilevel Preconditioners for Problems of Thin Smooth Shells. May 1997.

97-14 A. Eilmes, R. A. Römer, M. Schreiber. The two-dimensional Anderson model of localization with random hopping. June 1997.

97-15 M. Jung, J. F. Maitre. Some remarks on the constant in the strengthened C.B.S. inequality: Application to $h$- and $p$-hierarchical finite element discretizations of elasticity problems. July 1997.

97-16 G. Kunert. Error estimation for anisotropic tetrahedral and triangular finite element meshes. August 1997.

97-17 L. Grabowsky. MPI-basierte Koppelrandkommunikation und Einfluß der Partitionierung im 3D-Fall. August 1997.

97-18 R. A. Römer, M. Schreiber. Weak delocalization dueto long-range interaction for two electrons in a random potential chain. August 1997.

97-19 A. Eilmes, R. A. Römer, M. Schreiber. Critical behavior in the two-dimensional Anderson model of localization with random hopping. August 1997.

97-20 M. Meisel, A. Meyer. Hierarchically preconditioned parallel CG-solvers with and without coarse-matrix-solvers inside FEAP. September 1997.

97-21 J. X. Zhong, U. Grimm, R. A. Römer, M. Schreiber. Level-Spacing Distributions of Planar Quasiperiodic Tight-Binding Models. October 1997.

97-22 W. Rehm (Ed.). Ausgewählte Beiträge zum 1. Workshop Cluster-Computing. TU Chemnitz, 6./7. November 1997.

97-23 P. Benner, Enrique S. Quintana-Ortí. Solving stable generalized Lyapunov equations with the matrix sign function. October 1997

97-24 T. Penzl. A Multi-Grid Method for Generalized Lyapunow Equations. October 1997

97-25 G. Globisch. The hierarchical preconditioning having unstructured threedimensional grids. December 1997

97-26 G. Ammar, C. Mehl, V. Mehrmann. Schur-like forms for matrix Lie groups, Lie algebras and Jordan algebras. November 1997

97-27 U. Elsner. Graph partitioning - a survey. December 1997.

97-28 W. Dahmen, R. Schneider. Composite Wavelet Bases for Operator Equations. December 1997.

97-29 P. L. Levin, M. Spasojević, R. Schneider. Creation of Sparse Boundary Element Matrices for 2-D and Axi-symmetric Electrostatics Problems Using the Bi-orthogonal Haar Wavelet. December 1997.

97-30 W. Dahmen, R. Schneider. Wavelets on Manifolds I: Construction and Domain Decomposition. December 1997.

97-31 U. Elsner, V. Mehrmann, F. Milde, R. A. Römer, M. Schreiber. The Anderson Model of Localization: A Challenge for Modern Eigenvalue Methods. December 1997.

98-01 B. Heinrich, S. Nicaise, B. Weber. Elliptic interface problems in axisymmetric domains. Part II: The Fourier-finite-element approximation of non-tensorial singularities. January 1998.

98-02 T. Vojta, R. A. Römer, M. Schreiber. Two interfacing particles in a random potential: The random model revisited. February 1998.

98-03 B. Mehlig, K. Müller. Non-universal properties of a complex quantum spectrum. February 1998.

98-04 B. Mehlig, K. Müller, B. Eckhardt. Phase-space localization and matrix element distributions in systems with mixed classical phase space. February 1998.

98-05 M. Bollhöfer, V. Mehrmann. Nested divide and conquer concepts for the solution of large sparse linear systems. to app.: April 1998.

98-06 T. Penzl. A cyclic low rank Smith method for large, sparse Lyapunov equations with applications in model reduction and optimal control. March 1998.

98-07 V. Mehrmann, H. Xu. Canonical forms for Hamiltonian and symplectic matrices and pencils. March 1998.

98-08 C. Mehl. Condensed forms for skew-Hamiltonian/Hamiltonian pencils. March 1998.

The complete list of current and former preprints is available via
`http://www.tu-chemnitz.de/sfb393/preprints.html`.