# Implementierung von parallelem Divide-and-Conquer auf Gittertopologien

*Diplomarbeit*

*von*

**Marian Musiol**

*eingereicht an der*

**Fakultät für Mathematik und Informatik**
**Universität Passau**

*im*
**Mai 1996**

*Aufgabenstellung:*
**Prof. Christian Lengauer, Ph.D.**
**Lehrstuhl für Programmierung**

*Betreuer:*
**Christoph Herrmann**

# Eidesstattliche Erklärung

Hiermit erkläre ich, Marian Musiol, eidesstattlich, diese Arbeit ohne Verwendung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und alle Ausführungen, die wörtlich oder sinngemäß übernommen sind, als solche gekennzeichnet zu haben. Weiter erkläre ich, daß diese Diplomarbeit in gleicher oder ähnlicher Form noch keiner anderer Prüfungsbehörde vorgelegt wurde.

Passau, den 30.05.96

Marian Musiol

# Danksagung

Ich möchte allen herzlich danken, die mir direkt oder indirekt bei der Erstellung dieser Arbeit beigestanden haben. Ganz besonders danke ich Herrn Prof. Christian Lengauer, Ph.D. für eine gute methodische Vorbereitung und Inspirationen auf dem Gebiet der Parallelprogrammierung, Christoph Herrmann für seine fachliche Betreuung, tatkräftige Unterstützung und unermüdliche Diskussionen, Dr. Sergei Gorlatch für die hilfreichen Anregungen und dem Paderborn Center for Parallel Computing für die Möglichkeit der Experimentdurchführung auf der Parsytec GCel1024.

# Zusammenfassung

Das Ziel dieser Arbeit ist die imperative Implementierung eines funktionalen Skeletts für eine Klasse von parallelen Divide-and-Conquer-Algorithmen auf Gittertopologien. Das imperative Skelett verwaltet die parallele Architektur, verteilt das Eingabeproblem auf die Prozessoren und steuert die Kommunikation, d.h. das Skelett ist ein Programm, das eine Schnittstelle zu speziellen problemspezifischen Funktionen bietet. Die problemspezifischen Funktionen implementieren einen bestimmten Algorithmus, der in der Skelettklasse enthalten ist. Das Skelett soll den Programmierer von fehleranfälligen Entscheidungen bzgl. der parallelen Architektur befreien und seine Sicht auf die problemspezifischen Funktionen einschränken.

Als Anwendungsbeispiel des Skeletts auf eine bestimmte Probleminstanz aus einer Problemklasse wird das Polynomprodukt implementiert. Das Polynomprodukt mit unterschiedlichen Problemteilungsgraden ist ein Repräsentant der Algorithmenklasse, die das Skelett mit einem bestimmten Problemteilungsgrad instanziiert.

Die imperative Implementierung des Skeletts basiert auf einem geometrischen Modell, das in der Arbeit vorgestellt wird. Ob die erbrachte Leistung die an eine parallele Architektur gestellte Anforderungen erfüllt, hängt von vielen Faktoren ab, wie dem Teilungsgrad des Problems, der Art der Ein-/Ausgabe (zentralisiert/verteilt) oder dem Plazierungsproblem.

Um testen zu können, welche Faktoren welche Auswirkungen auf die Leistung haben, wurde eine parametrisierte Experimentierumgebung implementiert, die vor allem die Möglichkeit bietet, einen ausgewählten Algorithmus mit dem Skelett auszuführen, die Kommunikationszeiten für verschiedene Plazierungsfunktionen und Problemteilungsgrade zu messen und die Ergebnisse zu analysieren. Eine ganze Reihe von Experimenten, die auf der Parsytec GCel64 in Passau und der GCel1024 des Paderborn Center for Parallel Computing durchgeführt wurden, schließen die Arbeit ab.

# Inhaltsverzeichnis

# Kapitel 1

# Einführung

## 1.1 Das Divide-and-Conquer-Prinzip

Divide-and-Conquer(dt. teile und herrsche) ist eine der ältesten Strategien, die nicht nur im Bereich der Mathematik, sondern auch dem des Militärs, der Politik und in anderen Bereichen der Gesellschaft erfolgreich angewendet wurde. In der Zeit des algorithmischen Denkens hat Divide-and-Conquer (DC) sehr an Bedeutung gewonnen.

DC bezeichnet ein allgemeines, algorithmisches Lösungsverfahren, das sich wesentlich in drei Schritte gliedert:

- Im ersten Schritt wird ein Eingabeproblem der Größe $n$ in $k$ $(1 < k \leq n)$ Teilprobleme aufgeteilt mit möglicher Vorbehandlung der Teilprobleme.

- Im zweiten Schritt werden die Teilprobleme getrennt von einander gelöst.

- Im dritten Schritt werden die Teillösungen zu einer Gesamtlösung zusammengesetzt, und es wird evtl. eine Nachbehandlung durchgeführt.

Diese Schritte wendet man solange rekursiv auf die immer wieder entstehenden Teilprobleme an, bis diese einfach gelöst werden können.

Trotz der Allgemeinheit dieses Ansatzes benötigt man sehr spezielle Algorithmen zur Lösung der verschiedenen Probleme. Es gibt eine Vielzahl von Algorithmen, die nach der Divide-and-Conquer Strategie arbeiten und ein breites Problemspektrum decken. Als Repräsentanten der Klasse der Divide-and-Conquer (DC) Algorithmen können Polynomprodukt, Multiplikation großer Zahlen, Sortier-, Such-, und Matrixalgoritmen, und andere Beispiele angesehen werden [AHU74, HS78, OW93].

## 1.2 Motivation

Sehr viele Anwendungen erfordern Rechner, die um einige Größenordnungen schneller als die schnellsten heutigen Rechner sind. Die Parallelrechner scheinen einen Weg zu bieten, die unglaublich hohe Leistung zu erreichen, die erwartet wird. Die Programmierung solcher Anlagen ist äußerst schwierig und erfordert neue Methoden und Werkzeuge, um sie effizienter und einfacher zu machen.

### 1.2.1 Divide-and-Conquer und Parallelität

DC-Algorithmen wurden für die sequentielle Bearbeitung entworfen, sie enthalten jedoch implizite Parallelität, die aus der Algorithmenstruktur ersichtlich ist. Mit Entwicklung der Parallelrechner, die mehr Rechenleistung anbieten, sind neue Probleme aufgetreten, da wir neue Algorithmen und Entwurfsmethoden brauchen, um die Parallelität möglichst effizient ausnutzen zu können. Auf den ersten Blick scheinen gerade die DC-Algorithmen zur Parallelisierung besonders geeignet zu sein, da die Parallelität in ihrer Natur liegt. Man stellt sich die Frage, ob sich alle DC-Algorithmen parallelisieren lassen. Unterschiedliche Ansätze, die man in der Literatur findet, zeigen, daß sich nicht alle parallelen DC-Algorithmen auf gleiche Weise parallelisieren lassen. Es gibt unterschiedliche Unterklassen von parallelen DC-Algorithmen, deren Parallelisierbarkeit von verschiedenen Faktoren abhängt, wie vom Teilungsgrad des Problems, von der Zielarchitektur oder von der Anzahl der Prozessoren.

### 1.2.2 Ansätze zum Entwurf paralleler DC-Algorithmen

Die erste formale Definition von DC, die auf einem algebraischen Modell basiert, findet man in [Mou89]. Man unterscheidet allgemein zwei Formen von DC-Algorithmen:

- Sequentielles DC (SDC):
  Die Ausführung der Funktion angewendet auf die Teilprobleme erfolgt sequentiell, weil die rekursive Auswertung des Teilproblem von der Lösung des vorherigen Teilproblems abhängt.
  z.B.: Lösung eines Gleichungssystems in der Form einer Dreiecksmatrix:

  $$Ax = b$$

  wobei: $A$ eine $n \times n$-Dreiecksmatrix und $x, b$ $n$-Vektoren

  ```
  procedure GS(n)
  integer n;
  global A[1..n,1..n ], b[1..n], x[1..n];
  ```

```
if n = 1 then return x[1] = b[1]/A[1,1];
          else
          GS(n-1);
          x[n] = (b[n] - A[n,1 .. n-1] * x[1 .. (n-1)])/A[n,n];
endif
end GS
```

- Paralleles DC (PDC):
  Die Anwendung der Funktion auf die Teilprobleme erfolgt nebenläufig.
  z.B.: *scan*$\oplus$-Funktion, die auf einen Vektor angewendet wird und einen Vektor zurückgibt, dessen Element $i$ das Ergebnis der Anwendung des Operators $\oplus$ an alle Elemente $j$ mit $j \leq i$ enthält.

```
procedure SCAN ⊕ (p, q)
integer p,q,n;
global A[1..n], B[1..n], q = n;
if p = q then  A[p]; B[p];
          else
          SCAN ⊕ (1,q/2); SCAN ⊕ (q/2+1,q);
          A[q/2+1 .. q] = A[q/2+1 .. q] ⊕ B[p .. q/2];
          B[p .. q/2] = A[q/2+1 .. q];
          B[q/2+1 .. q] = B[q/2+1 .. q] ⊕ B[p .. q/2];
endif
end SCAN
```

Das algebraische Modell von Mou wird noch in Abschnitt 3.1 näher erläutert.

In [Smi87] wird eine Vorgehensweise zum Entwurf von DC-Algorithmen auf formaler Basis vorgestellt. Wir betrachten eine Menge abstrakter Objekte und klassifizieren sie bzgl. der Charakteristiken, die sie aufweisen. Auf der abstrakten Ebene gibt es eine Klasse von DC-Algorithmen. Diese Klasse wird durch gemeinsame strukturelle und funktionale Eigenschaften gebildet.

Wir untersuchen diese Objekte nicht nur um ein klares und tieferes Verständnis der Algorithmen zu gewinnen, sondern auch um einen Formalismus für den Algorithmenentwurf zu entwickeln. Die Untersuchung von gemeinsamen strukturellen und funktionalen Eigenschaften, welche auf abstrakter Ebene die Klasse der DC-Algorithmen bilden, erlaubt die Entwicklung von Designmethoden für Algorithmen dieser Klasse, die prinzipiell nach folgendem Schema arbeiten. Wenn ein Problem bei gegebenem Input ausreichend einfach ist, wird es direkt gelöst, andernfalls wird es in unabhängige Teilprobleme des gleichen Typs aufgeteilt. Die Teilprobleme werden gelöst, die Ergebnisse der Teilprobleme werden zusammengesetzt und aufgrund der Unabhängigkeit der Teilprobleme besteht die Möglichkeit diese parallel zu lösen. Wesentlich für den Entwurf eines PDC-Algorithmus ist der Zusammenhang der beabsichtigten Funktionalität mit seiner Struktur und der Funktionalität seiner Bestandteile. Die Funktionalität wird durch eine formale Spezifikation dargestellt. Das Ziel ist die Formalisierung des Entwurfsprozesses, um ihn automatisch zu unterstüzen. Wir brauchen ein Programmschema, das die Struktur der Unterklasse der PDC-Algorithmen repräsentiert und dessen Korrektheit anhand der Spezifikation nachgewiesen werden kann. Das Schema kann, bei Erfüllung der gegebenen Spezifikation, mit einem Problem aus der Klasse instanziiert werden. In der Klasse der PDC-Algorithmen gibt es leider nur eine Unterklasse, die man mit dem Formalismus beschreiben kann und die sich parallelisieren läßt.

Ein auf dem SPMD (Single Programm Multiple Data)-Modell implementierbares Programmschema für das DC-Verfahren wird in [GL95] vorgeschlagen als formale Verfeinerung der Spezifikation, die in Form von rekursiven funktionalen Definitionen (sog. *Skelett*) erscheint. Das Programmschema besteht aus einem Berechnungsgraphen (*call tree*) und einem funktionalen Programm, das gemeinsam von jedem Knoten dieses Baumes aufgerufen wird. Die Korrektheit des Ergebnisses in der Phase der funktionalen Implementation wird durch eine Transformation in den Bird-Meertens Formalismus für Funktionen höherer Ordnung garantiert [Ski90]. In der zweiten Phase wird das funktionale Programm in eine imperative verteilte SPMD-Implementation umgesetzt.

### 1.2.3 Zielsetzung

Das Ziel dieser Arbeit ist die imperative SPMD-Implementation eines Skeletts mit zentralisierten und verteilten Ein/Ausgabedaten, d.h. eines Programmschemas, das gemeinsam mit einem Algorithmus aus der Unterklasse von PDC durch Einbinden einer Menge von Schnittstellenfunktionen ausgeführt werden kann. Die Schnittstellenfunktionen (*problemspezifischen Funktionen*) müssen für einen bestimmten DC-Algorithmus implementiert werden. Als Beispiel wird das Polynomprodukt mit Problemteilungsgrad $k = 3$ und $k = 4$ implementiert. Das Skelett soll auf einer virtuellen $n$-dimensionalen Gitter-Topologie implementiert werden. Die virtuelle Topologie wird später auf eine reale 2-dimensionale Gitter-Topologie abgebildet. Diese Abbildung wird mit sog. *Plazierungsfunktionen* spezifiziert. Es werden die Kommunikationszeiten für verschiedene Plazierungsfunktionen und für unterschiedliche Problemteilungsgrade gemessen. Die abstrakte Grundlage der Implementation bildet [GL95] und [HL95].

## 1.3 Rolle der SPMD-Implementierung

Die Abb. 1.1 auf S. 7 zeigt die Entwurfsphasen und die Rolle der SPMD-Implementierung. Am Anfang gibt es eine Spezifikation mittels eines funktionalen rekursiven Programms. Sie besteht aus zwei Teilen:

- aus einem Skelett, einem allgemeinen Programmschema für DC-Algorithmen (Abb. 1.1 oben links) und
- aus einer Menge von speziellen problemspezifischen Funktionen, die das Skelett mit einem festen Problemteilungsgrad instanziieren (Abb. 1.1 oben rechts).

Daraus ergibt sich ein spezielles funktionales DC-Skelett. Die funktionale Spezifikation mit Rekursionen soll eine Grundlage für die imperative Skelett-Implementierung schaffen. Da die Skelett-Implementierung nicht ein funktionales sondern ein imperatives Programm ist, entwickeln wir ein Schleifenprogramm mit sequentiellen und parallelen Schleifen. Die parallelen Schleifen lösen die Rekursion im funktionalen Programm auf und werden zusammen mit den sequentiellen Schleifen auf die Prozessoren abgebildet. Zur Zeit werden alle Übergänge zwischen den Entwurfsphasen manuell vorgenommen. In Folgearbeiten wird es versucht sie zu automatisieren. So sollen auch die problemspezifischen Funktionen nicht vom Programmierer selbst sondern automatisch erzeugt werden. Das Umfeld meiner Diplomarbeit ist auf Abb. 1.1 eingerahmt.
Die imperative Implementierung besteht wiederum aus zwei Teilen:

- aus der SPMD-Implementierung des Skeletts auf eine Gittertopologie, d.h. aus einem C-Programm mit synchronen *send/receive*-Operationen für die Verteilung der parallelen Schleifen im Schleifenprogramm auf die Prozessoren mit Berücksichtigung der Datenabhängigkeiten, und
- aus der C-Implementierung der problemspezifischen Schnittstellenfunktionen, die vom Skelett aufgerufen werden.

Dazu kommt eine Experimentierumgebung, d.h. ein C-Programm, das auf einem Hostrechner läuft und interaktiv mit dem Benutzer agiert, um die Programmparameter zu bestimmen. Die gemeinsame Übersetzung und Ausführung der getrennten Teile (Skelett + problemspezifische Funktionen) auf dem Transputernetz wird ebenso von dem Hostprogramm vorgenommen.

## 1.4 Übersicht

- In Kapitel 2 werden zwei Verfahren zur Berechnung des Polynomprodukts vorgestellt, die dem DC-Prinzip folgen. Die Algorithmen für das Polynomprodukt mit verschiedenen Problemteilungsgraden (Dreiteilung und Vierteilung) sind als Repräsentanten der Klasse der DC-Algorithmen gut geeignet. Für die DC-Algorithmen wird ein Skelett entwickelt. Es wird auch gezeigt, wie man das Verfahren nach einer Modifikation auf die Multiplikation großer Zahlen anwenden kann. Weiter wird die Komplexität der DC-Algorithmen betrachtet.

- In Kapitel 3 werden zwei Modelle zur Spezifikation einer Klasse von DC-Algorithmen erläutert. Das erste Modell ist algebraisch, in ihm werden DC-Funktionen als Morphismen betrachtet. Das zweite Modell besteht aus einem funktionalen rekursiven Programmschema mit Funktionen höherer Ordnung, die die Klasse problemspezifisch instanziieren und einem Berechnungsgraphen zur Auswertung des funktionalen Programms. Die Struktur des Algorithmus und des Berechnungsgraphen deuten auf die Parallelität hin. Das allgemeine Schema wird dann auf eine Unterklasse von DC-Algorithmen eingeschränkt, die für uns weiter von Interesse sind.

- In Kapitel 4 wird ein geometrisches Modell für das imperative DC-Skelett entwickelt. In das geometrische Modell in der Form eines $n$-dimensionalen Gitters läßt sich der Berechnungsgraph der funktionalen Spezifikation einbetten. Das Modell berücksichtigt alle Datenabhängigkeiten und bildet eine Grundlage der virtuellen Topologie. Die virtuelle Topologie kann man mehr oder weniger effizient auf die reale 2-dimensionale Gittertopologie abbilden. Knoten der realen Topologie sind Prozessoren, die nach dem SPMD-Prinzip arbeiten. Wenn weniger Prozessoren zur Verfügung stehen, als man braucht, was fast immer der Fall ist, werden zusätzliche Techniken, wie Simulation oder Compress-and-Conquer, benötigt.

- In Kapitel 5 wird die C-Implementierung des *balancierten fixed degree* DC-Skeletts auf einer SPMD-Gittertopologie dokumentiert. In Bezug auf die vorherigen Kapitel werden entsprechende Datenstrukturen und C-Funktionen implementiert und der Zusammenhang der einzelnen Dateien erläutert. Um die Arbeit mit der parallelen Architektur zu erleichtern, wird eine Experimentierumgebung entwickelt, das die Übersetzung, die Aufnahme der Programmparameter und die Ausführung, und auch die Zeitmessungen ermöglicht.

- In Kapitel 6 werden die Ergebnisse der in Paderborn und in Passau durchgeführten Experimente präsentiert.

- Der Anhang enthält die Experimentergebnisse und ausgewählte Teile der C-Implementierung.

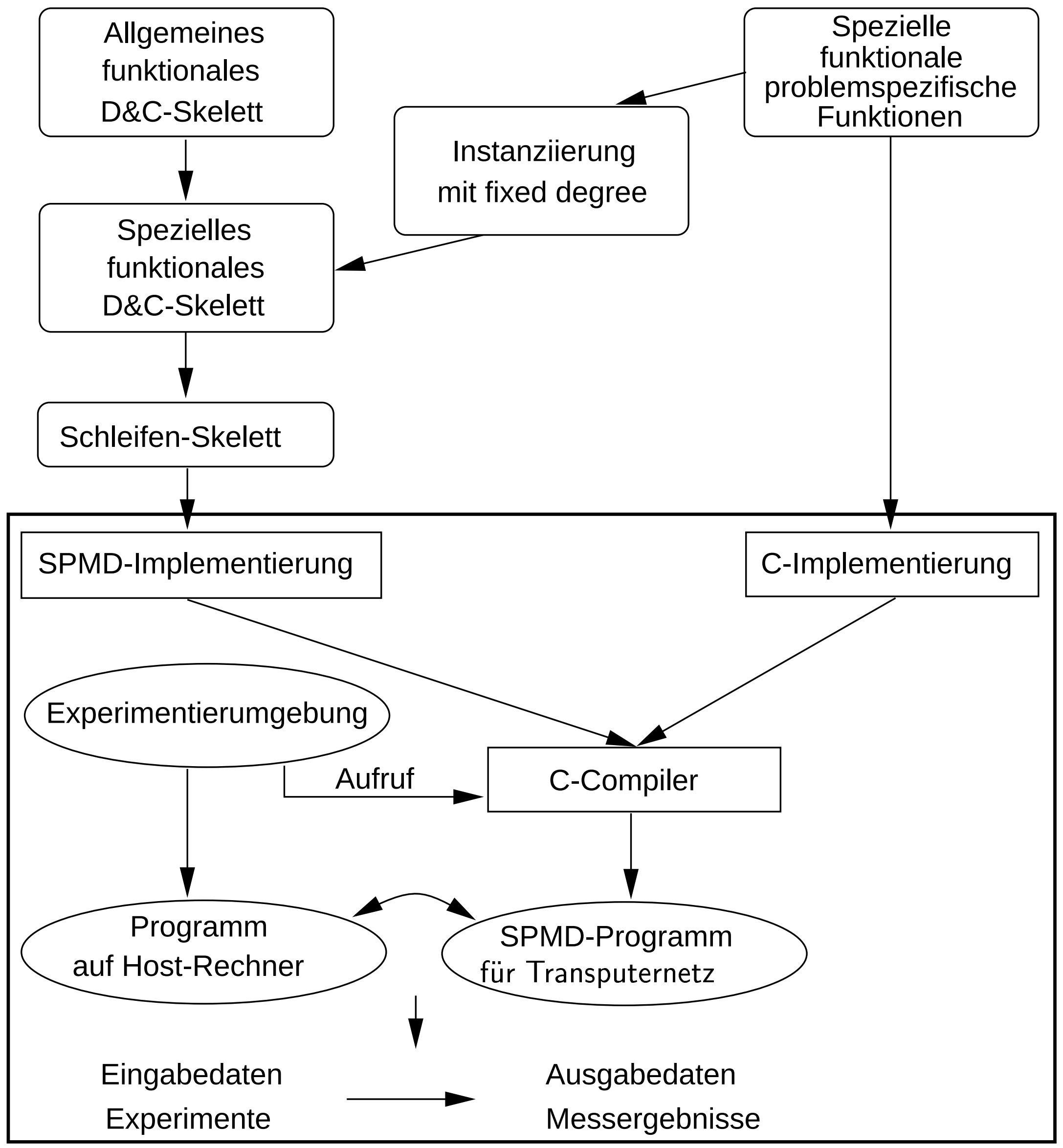



Abbildung 1.1: Entwurfsphasen

# Kapitel 2

# Das Polynomprodukt als Beispiel

Ein Polynom $A$ vom Grade $2N - 1$ kann durch $2 * N$ Koeffizienten $a_0, ..., a_{2*N-1}$ repräsentiert werden. Es hat die Form:

$$A = a_0 * x^0 + a_1 * x^1 + \cdots + a_{2*N-1} * x^{2*N-1}$$

Wir abstrahieren zunächst von der programmtechnischen Realisierung. Seien nun zwei Polynome $A$ und $B$ vom Grade $2*N-1$ gegeben. Sei $A$ wie oben angegeben und

$$B = b_0 * x^0 + b_1 * x^1 + \cdots + b_{2*N-1} * x^{2*N-1}$$

Wie kann man das Produkt der beiden Polynome $P = A * B$ berechnen?

Das Produktpolynom ist das Polynom vom Grade $4 * N - 2$, das man erhält, wenn man jeden Term $a_i * x^i$ des Polynoms $A$ mit jedem Term $b_j * x^j$ des Polynoms $B$ multipliziert und dann die Terme mit gleichen Exponenten zusammenfaßt. Man kann leicht eine Implementation dieses sog. naiven Verfahrens mit zwei for-Schleifen angeben, unter der Voraussetzung, daß die Polynome durch Arrays realisiert werden, die die Koeffizienten enthalten:

```
for i = 0, ..., 2 * N − 1
    for j = 0, ..., 2 * N − 1
        P[i + j] := P[i + j] + A[i] * B[j]
```

Diese Darstellung zeigt unmittelbar, daß die Berechnung der Koeffizienten des Produktpolynoms die Komplexität $\mathcal{O}(N^2)$ hat. Man will ein anderes Verfahren zur Berechnung des Produktpolynoms, das mit weniger als $N^2$ Koeffizientenberechnungen auskommt. Das Verfahren folgt der DC-Strategie [AHU74, KO62].

## 2.1 Schulmethode: Vierteilung

Um das DC-Prinzip auf das Problem, das Produkt zweier Polynome zu berechnen, einfach anwenden zu können, nehmen wir an, daß die Anzahl der Koeffizienten der Polynome $A$ und $B$ eine Potenz von 2 ist ($2 * N = 2^n, n \in \mathbb{N}$, d.h. $N = 2^{n-1}$). Man kann schreiben:

$$A = A_l + x^N A_h \qquad mit \qquad \begin{aligned} A_l &= a_0 * x^0 + a_1 * x^1 + \cdots + a_{N-1} * x^{N-1} \\ A_h &= a_N * x^0 + a_{N+1} * x^1 + \cdots + a_{2*N-1} * x^{N-1} \end{aligned}$$

Ebenso kann man schreiben:

$$B = B_l + x^N B_h$$

mit zwei analog definierten Polynomen $B_l$ und $B_h$ vom Grade $N-1$. Dann ist

$$\begin{aligned} P &= A * B \\ &= (A_l + x^N * A_h) * (B_l + x^N * B_h) \\ &= A_l * B_l + x^N * A_l * B_h + x^N * A_h * B_l + x^{2*N} * A_h * B_h \\ &= A_l * B_l + x^N * (A_l * B_h + A_h * B_l) + x^{2*N} * A_h * B_h. \end{aligned}$$

Wir haben das Problem, das Produkt zweier Polynome vom Grade $2N-1$ mit jeweils $2N$ Koeffizienten zu berechnen, in vier Teilprobleme zerlegt. Vier Produkte von Polynomen vom Grade $N-1$ mit jeweils $N$ Koeffizienten zu berechnen. Das sind die Produkte $A_l * B_l,\ A_l * B_h,\ A_h * B_l,\ A_h * B_h$.
Aus der oben genannten Gleichung für $P$ kann man ablesen, daß man die Koeffizienten des Produktpolynoms $P$ mit Shifts und Additionen erhält, ohne daß weitere Koeffizientprodukte berechnet werden müssen.

## 2.2 Karatsuba-Verfahren: Dreiteilung

Ein auf der oben angegebenen Zerlegung begründetes DC-Verfahren liefert keine Verbesserung (sequentiell) gegenüber dem naiven Verfahren (siehe Abschnitt 2.4). Es ist aber nicht schwer eine andere Zerlegung des Problems anzugeben. Dadurch kommt ein auf dieser Zerlegung begründetes DC-Verfahren mit weniger Koeffizientproduktberechnungen aus.

$$\begin{aligned} P &= A_l * B_l + x^N * A_l * B_h + x^N * A_h * B_l + x^{2*N} * A_h * B_h \\ & \quad wegen\{(A_l + A_h) * (B_l + B_h) = A_l * B_l + A_l * B_h + A_h * B_l + A_h * B_h\} \\ &= A_l * B_l + x^N * ((A_l + A_h) * (B_l + B_h) - A_l * B_l - A_h * B_h) + x^{2*N} * A_h * B_h \\ &= A_l * B_l + x^N * A_m * B_m + x^{2*N} * A_h * B_h \end{aligned}$$

*wobei* $A_m * B_m = (A_l + A_h) * (B_l + B_h) - A_l * B_l - A_h * B_h$

Wir haben das Polynomproduktproblem zweier Polynome vom Grade $2*N-1$ in drei unabhängige Teilpolynomproduktprobleme zweier Polynome vom Grade $N-1$ zerlegt. Das Teilproblem $A_m * B_m$ des Polynomprodukts $P$ kann mit Hilfe von Ergebnissen der Teilprobleme $A_l * B_l$, $A_h * B_h$ und Shifts, Additionen und Subtraktionen gelöst werden. Ein auf dieser Zerlegung gegründetes DC-Verfahren zur Berechnung des Produkts zweier Polynome mit $2N$ Koeffizienten kann wie folgt formuliert werden.

*Input:*
$A, B$ zwei Polynome vom Grade $2^n - 1$ mit $2^n$ Koeffizienten.

*Output:*
$P = A * B$ ein Polynom vom Grade $2^{n+1} - 2$ mit $2^{n+1}$ Koeffizienten.

*Methode:*

1. Basic:

   Falls $n = 0$, berechne das Produkt der beiden Koeffizienten $a_i$ und $b_j$; sonst:

2. Divide:

   (a) Zerlege die Polynome $A$ und $B$ in die Form:
   $A = A_l + x^N * A_h,\ B = B_l + x^N * B_h,$ setze $A_m = A_l + A_h,\ B_m = B_l + B_h$

   (b) Wende das Verfahren rekursiv an, um die folgenden Polynomprodukte zu berechnen:
   $C_l = A_l * B_l,\ C_h = A_h * B_h,\ C_m = A_m * B_m$

3. Combine:

   Setze $A * B = C_l + x^N * (C_m - C_l - C_h) + x^{2*N} * C_h$

## 2.3 Anwendung auf die Multiplikation großer Zahlen

Beide gerade vorgestellte Algorithmen können auch auf die Multiplikation großer Zahlen angewendet werden, wenn $A$ und $B$ zwei $2 * N$-Bit Zahlen repräsentieren. Im Fall der Dreiteilung müssen wir von der Annahme ausgehen, daß die Multiplikationen immer nur mit $N$-Bit Zahlen erfolgen, sodaß es keine Überträge gibt, d.h. $A_l + A_h$ oder $B_l + B_h$ zu $(N + 1)$-Bit Zahlen werden.
Um den Algorithmus für die Multiplikation großer Zahlen vollständig zu machen, müssen wir noch den Fall betrachten, daß $A_l + A_h$ und $B_l + B_h$ $(N+1)$-Bit Zahlen

sind. So kann nämlich $(A_l + A_h) * (B_l + B_h)$ nicht direkt durch die rekursive Anwendung des Algorithmus auf ein Problem der Größe $N$ berechnet werden. Eine Lösung dieses Problems ergibt sich durch folgende Modifikationen:

- $A_l + A_h$ kann als $a * 2^N + b$ geschrieben werden, wobei $a$ das führende Bit der Summe $A_l + A_h$ ist und $b$ die verbleibenden Bits darstellt,
- $B_l + B_h$ analog als $c * 2^N + d$.

Damit läßt sich $(A_l + A_h) * (B_l + B_h)$ durch $a * c * 2^{2*N} + (a * d + b * c) * 2^N + b * d$ ausdrücken. In dieser Form kann der Term $b * d$ durch rekursive Anwendung des Algorithmus auf ein Problem der Größe $N$ berechnet werden. Die anderen Multiplikationen sind mit linearer Komplexität ausführbar, da sie entweder einfache Bits ($a$, $c$) oder eine Potenz von zwei als Argument haben.
Der vollständige Algorithmus sieht wie folgt aus:

$$\begin{aligned}
a &= (A_l + A_h)\ div\ 2^N \\
b &= (A_l + A_h)\ mod\ 2^N \\
c &= (B_l + B_h)\ div\ 2^N \\
d &= (B_l + B_h)\ mod\ 2^N \\
x &= a * c * 2^{2*N} + (a * d + b * c) * 2^N + b * d \\
y &= A_l * B_l \\
z &= A_h * B_h \\
A * B &= y * 2^{2*N} + (x - y - z) * 2^N + z
\end{aligned}$$

Dieser Algorithmus paßt leider nicht in unser Schema, d.h. er liegt nicht in der DC-Unterklasse, für die das Skelett implementiert ist. Man kann den vorgestellten Algorithmus für das Polynomprodukt auch auf die Multiplikation großer Zahlen anwenden und das Ergebnis entsprechend interpretieren und in der Ausgabefunktion nachbehandeln. Alle Überträge können in linearer Zeit mittels Shifts korrigiert werden, sodaß das Ergebnis das Produkt der multiplizierten Zahlen in der Binärdarstellung repräsentiert.

## 2.4 Bedeutung des Teilungsgrades für die Komplexität

Wir betrachten den allgemeinen Fall einer DC-Anwendung. Hierbei wird das Ausgangsproblem der Größe $N = b^d$ in $k$ Teilprobleme aufgeteilt. Unter diesen Voraussetzungen läßt sich die sequentielle Zeitkomplexität des Problems durch folgende Funktion beschreiben:

$$T(N) = \begin{cases} c & N = 1, \\ kT(\frac{N}{b}) + t(N) & N > 1 \end{cases} \tag{2.1}$$

*wobei*: $b$ der Datenteilungsgrad ist (im folgenden immer $b = 2$), $c$ Kosten zum Lösen des Basisfalls sind, $t(N)$ die benötigte Zeit zum Spalten des Problems und zur Verarbeitung der Teillösungen beschreibt.

Löst man die Rekursivität der Gleichung 2.1 auf, so gilt:

$$\begin{aligned}
T(N) &= kT(\frac{N}{b}) + t(N) && \textit{Rekursionsgleichung} \\
&= k(kT(\frac{N}{b^2}) + t(\frac{N}{b})) + t(N) && \textit{Einsetzen} \\
&= k^2T(\frac{N}{b^2}) + kt(\frac{N}{b}) + t(N) && \textit{Ausmultiplizieren} \\
&= k^3T(\frac{N}{b^3}) + k^2t(\frac{N}{b^2}) + kt(\frac{N}{b}) && \textit{erneutes Einsetzen} \\
& && \textit{und Ausmultiplizieren} \\
&\vdots \\
&= k^dT(\frac{N}{b^d}) + \sum_{i=0}^{d-1} k^i t(\frac{N}{b^i}) && \textit{Rekursion aufgelöst} \\
&= ck^d + \sum_{i=0}^{d-1} k^i t(b^{d-i}) && \textit{Einsetzen der Konstanten } c,\ N
\end{aligned}$$

wegen $N = b^d$ folgt $ck^d = ck^{\log_b N} = cN^{\log_b k} \in \mathcal{O}(N^{\log_b k})$.

Das heißt die Lösung aller $k^d$ kleinsten Probleme (insgesamt) erfolgt in polynomialer Zeit. (Im parallelen Fall werden die Teilprobleme gleichzeitig behandelt.) Der zweite Term enthält die Zeit zum Aufspalten der Probleme und zum Zusammensetzen der Teillösungen. Zur Vereinfachung wird angenommen, daß $t(N)$ polynomial ist, d.h. $t(N) \le c_1 N^e$. Es gilt:

$$\sum_{i=0}^{d-1} k^i t(b^{d-i}) \le c_1 \sum_{i=0}^{d-1} k^i t(b^{e(d-i)}) = c_1 b^{ed} \sum_{i=0}^{d-1} (\frac{k}{b^e})^i \qquad (*)$$

Nach [LD91] unterscheidet man drei Fälle:

- $k < b^e \Longrightarrow (*) \in \ \mathcal{O}(b^{ed}) = \mathcal{O}(N^e)$
- $k > b^e \Longrightarrow (*) \in \ \mathcal{O}(k^d) = \mathcal{O}(N^{\log_b k})$
- $k = b^e \Longrightarrow (*) \in \ \mathcal{O}(N^e \ \log_b N)$

Um den Einfluß von $k$ zu vergleichen, betrachten wir folgende Spezialfälle (mit $e = 1$ und Datenteilungsgrad $b = 2$):

- Für Algorithmen mit Problemteilungsgrad $k = 2$ (z.B.: Quicksort):

$$T(N) \in \mathcal{O}(N \log N)$$

  - Spezialfälle (z.B.: *scan* $\oplus$):

$$T(N) \in \mathcal{O}(N)$$

- Für Algorithmen mit Problemteilungsgrad $k = 3$:

$$T(N) \in \mathcal{O}(N^{\log_2 3})$$

- Für Algorithmen mit Problemteilungsgrad $k = 4$:

$$T(N) \in \mathcal{O}(N^{\log_2 4}) = \mathcal{O}(N^2)$$

Für das Polynomprodukt mit $k = 4$ gibt es keine Vorteile gegenüber dem sequentiellen Algorithmus, beide liegen in $\mathcal{O}(N^2)$. Mit dem modifizierten Algorithmus mit $k = 3$ verbessert sich die Zeitkomplexität.

# Kapitel 3

# Das funktionale Skelett

Die Abbildung eines parallelen Programms auf eine Multiprozessor-Architektur ist ein komplexer Prozeß, der Entscheidungen über die Prozessverteilung auf die Prozessoren, die Synchronisation zwischen den Prozessen und Kommunikationsmuster zwischen den Prozessoren erfordert. Der Programmierer muß alle Entscheidungen selbst treffen und ihre Kontrolle im Programm implementieren. Das Programm ist, wegen der starken Programm-Zielmaschine-Abhängigkeit, nicht auf jede parallele Architektur portierbar. Um den Programmierer von fehleranfälligen Entscheidungen zu befreien und maschinenunabhängig zu bleiben, werden Skelette implementiert. Skelette stellen eine effiziente Implementierung für eine Klasse von Programmen dar und ermöglichen eine Anpassung an unterschiedliche Architekturen. Ein Skelett repräsentiert eine algorithmische Form, die auf die ganze Klasse von Algorithmen angewendet werden kann, die gleiche Struktur und Funktionalität aufweisen. Komponenten des Skeletts sind problemspezifische Funktionen, die problemabhängig als sequentielle Algorithmen implementiert und zusammen mit dem Skelett ausgeführt werden können. Skelette werden zuerst mit funktionaler Programmierung als Funktionen höherer Ordnung (*higher-order-functions*) entwickelt und dann in eine imperative Sprache transformiert. Das transformierte Skelett muß das spezifische Verhalten der Zielmaschine aufweisen.

In diesem Kapitel werden zwei Spezifikationsmethoden vorgestellt. Eine basiert auf einem algebraischen Morphismus-Modell, die andere auf Funktionen höherer Ordnung.

## 3.1 Das Morphismus-Modell

Dieses Kapitel befaßt sich mit dem Pseudomorphismus-Modell von Mou [Mou89], der als erster einen mathematischen Formalismus für die DC-Algorithmen entwickelt hat.

Formal ist ein *Morphismus* eine Funktion von einer *Algebra* $(X, op_x)$ in eine *Algebra* $(Y, op_y)$.

$$(X, op_x) \rightarrow (Y, op_y)$$

Eine *Algebra* ist ein Paar $A = (\mathcal{S}, op_s)$, wobei: $\mathcal{S}$ eine Menge und $op_s$ eine unter $\mathcal{S}$ abgeschlossene Operation (allg. eine Menge von Operationen) ist. Wenn $op_s$ eine $k$-stellige Operation ist und $S_0, .., S_{k-1}$ $k$ Elemente aus $\mathcal{S}$ sind ($S_0, .., S_{k-1} \in \mathcal{S}$) dann gilt: $S = op_s(S_0, .., S_{k-1}) \in \mathcal{S}$.

Abstrakte Datenstrukturen und Datenstrukturen in Programmiersprachen können in dem algebraischen Modell als *Räume* dargestellt werden.
Ein Raum ist ein Paar $S = (U, R)$, wobei $U$ eine Menge, genannt *Universum*, und $R$ eine Menge von Relationen ist. Ein Graph $G = (V, E)$ ist ein Raum, wobei $V$ eine Menge von Knoten (Universum) und $E$ eine Relation auf $V$ ist. Listen und Bäume können als spezielle Graphen betrachtet werden, Vektoren und Matrizen können als Räume modelliert werden.
Eine *Raumdomäne* $\mathcal{S}$ ist eine Menge von endlichen oder unendlichen Räumen mit gewissen gemeinsamen Eigenschaften. Beispiele für Raumdomänen sind: Listen ($\mathcal{L}$), Bäume ($\mathcal{B}$), Vektoren ($\mathcal{V}$), usw. Operationen auf Raumdomänen sind also Algebren. So gilt auch:

1. Die *combine*-Operation $com$ auf einer Raumdomäne $\mathcal{S}$ eine Algebra $(\mathcal{S}, com)$,

2. Ist $div$ die *divide*-Operation auf einer Raumdomäne $\mathcal{S}$, dann ist $(\mathcal{S}, div^{-1})$ eine Algebra.

In diesem Abschnitt gilt folgende Notation (siehe auch in [Mou89]):

- Sei $f : X \rightarrow Y$ eine Funktion. $!f$ ist eine Funktion von Vektor vom Typ $X$ nach Vektor vom Typ $Y$ und heißt *Verteilung der Funktion f*.

- Seien $f,\ g$ zwei Funktionen. $f : g$ ist die Funktionskomposition.

- Seien $k$-Funktionen $f_0, ..., f_{k-1}$. $(f_0, ..., f_{k-1})$ bezeichnet eine Funktion auf $k$-Tupel definiert als $(f_0, ..., f_{k-1})\ (X_0, ..., X_{k-1}) = (f_0X_0, ..., f_{k-1}X_{k-1})$. Falls alle $k$ Funktionen identisch sind kann man schreiben $(f, ..., f) =!f$.
  $!f$ kann als *Verteilung auf Tupels* gedacht werden.

### 3.1.1 Divide Operation

Sei $\mathcal{S}$ eine Menge. Eine Abbildung $divide : \mathcal{S} \rightarrow \mathcal{S}^k$ ist eine Funktion auf $\mathcal{S}$, sodaß $divide\ (S) = S_0, ..., S_{k-1}$ wobei $S, S_0, ..., S_{k-1} \in \mathcal{S}$ und $S$ teilbar, d.h. $\neg\exists S_i = \{\}$.

### 3.1.2 Combine Operation

Sei $\mathcal{S}$ eine Menge. Eine Abbildung $combine : \mathcal{S}^k \rightarrow \mathcal{S}$ ist eine *k-stellige* Funktion auf $\mathcal{S}$, sodaß $combine\ (S_0, ..., S_{k-1}) = S$ wobei $S, S_0, ..., S_{k-1} \in \mathcal{S}$ und $S_i \neq \{\}$.

**Beispiel 3.1.1**

Ein einfaches Beispiel für einen Morphismus ist die *reduce* $\oplus$-Funktion, die auf einen *integer* Vektor angewendet wird.

$$reduce \oplus \circ\, cat(a,\ b) = \oplus\ (reduce \oplus (a), reduce \oplus (b))$$
$$\text{wobei: } com = \oplus,\ div^{-1} = cat \text{ und } cat(a,\ b) = ab.$$

### 3.1.3 Anpassungsfunktionen

Mit einem Morphismus wird vorausgesetzt, daß die Anwendung der Funktion auf eine Teilstruktur unabhängig von anderen Teilstrukturen ist. Dies trifft auf viele DC-Algorithmen aber nicht zu. In fast allen nicht trivialen DC-Algorithmen ist es notwendig, daß die Teilstrukturen mit anderen Teilstrukturen kommunizieren. Im Pseudomorphismus-Modell werden *Anpassungsfunktionen* eingeführt um die wechselseitige Abhängigkeit zwischen Teilstrukturen darzustellen. Es gibt zwei Arten von Anpassungsfunktionen:

*preadjust* $g : X^k \to X^k$
*postadjust* $h : Y^k \to Y^k$

Die *adjust* Funktionen sind Abbildungen über mehrere Strukturen, die wechselseitig voneinander abhängen und Inter-Struktur Kommunikation erfordern. Eine *adjust* Funktion kann deshalb immer in zwei Teile zerlegt werden:

- in eine Kommunikationsfunktion (*comm*), die alle Operationen ausführt, die Kommunikationen zwischen den Strukturen benötigen und
- in eine lokale Funktion (*loc*), die aus den lokalen Operationen auf einer Struktur besteht.

$$adjust = loc : comm$$

Die verallgemeinerte DC-Form mit Anpassungsfunktionen ist dann:

$$DC\ div\ com\ g\ h\ p\ b\ x = f\ x$$
$$\text{wobei: } f\ x = p\ x \to b\ x\ ;\ (\ com : h : (\ !\ f\ ) : g : div\ )\ x$$

Die Anpassungsfunktion $g$ wird vor , die Anpassungsfunktion $h$ nach der Rekursionsfunktion $f$ angewendet. Eine Funktion, die durch $DC$ definiert ist, wird *Pseudomorphismus* genannt (Abb. 3.1).

### 3.1.4 Postmorphismus

Wenn die *preadjust*-Funktion die Identität ist (*scan* $\oplus$), heißt die $DC$-Funktion *Postmorphismus*.

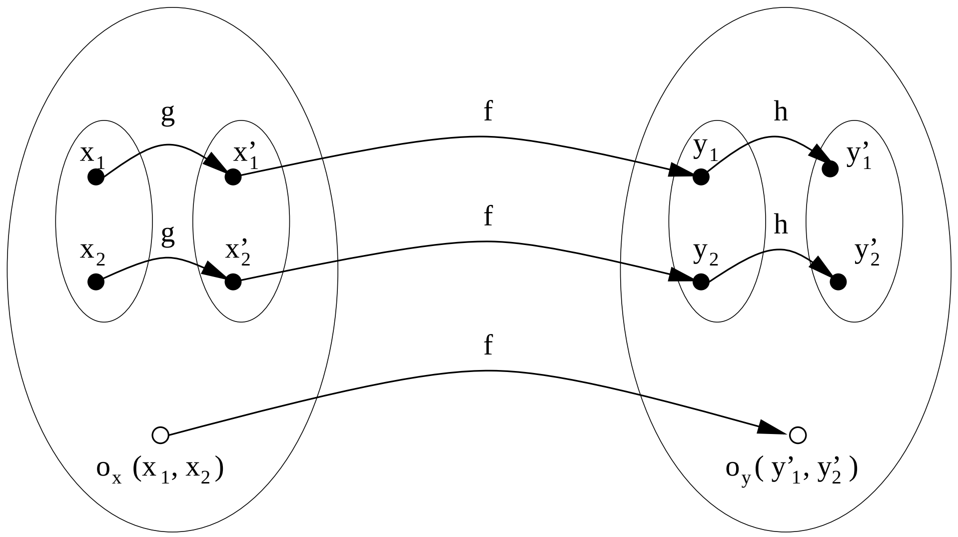



Abbildung 3.1: Pseudomorphismus

### Scan

Die *scan* $\oplus$-Funktion wird auf einen Vektor angewendet und gibt einen Vektor zurück, dessen Element $i$ das Ergebnis der Anwendung des Operators $\oplus$ an alle Elemente $j$ mit $j \leq i$ enthält.

$$scan \oplus v = r \text{ mit } r[i] = \oplus_{j=0}^{i} v[j]$$

Der folgende DC-Algorithmus für *scan* $\oplus$ nutzt die *links/rechts* Teilung. Für die *postadjust*-Funktion $h$ bieten sich zwei Möglichkeiten an, abhängig von der Art der Kommunikation zwischen den Teilstrukturen:

$$\begin{aligned} scan\oplus = {} & DC\ (div_{lr},\ com_{lr},\ id,\ h,\ p,\ b) \\ & \text{mit } h = loc : comm \end{aligned}$$

wobei die *postadjust*-Funktion $h$:

- das letzte Element des linken Teilvektors zu jedem Element des rechten Teilvektors addiert:

$$\begin{aligned} comm &= broadcast, \\ loc &= (id, (!\oplus)), \\ b &= id, \end{aligned}$$

- die Elemente des linken Teilvektors zu den entsprechenden Elementen des rechten Teilvektors addiert:

$$
\begin{aligned}
comm &= correspondent, \\
loc &= (!\ loc_1,\ !\ loc_2), \\
loc_1 &= ((x_1, sum_1),\ (x_2, sum_2)) = (x_1, sum_1 \oplus sum_2), \\
loc_2 &= ((x_1, sum_1), (x_2, sum_2)) = (x_1 \oplus x_2, sum_1 \oplus sum_2), \\
bx &= (x, x).
\end{aligned}
$$

*Broadcast* ist eine relativ teure Kommunikationsart; man kann sie durch *correspondent* Kommunikation ersetzen. Die Transformation ist dann und nur dann möglich, wenn jedes Vektorelement nicht nur seinen eigenen Wert sondern auch den des letzten Elements im Teilvektor enthält.

$$
\begin{aligned}
scan \oplus :\ cat\ ([1\ 2], [3\ 4]) &= cat : h : (!\ scan \oplus)\ ([1\ 2], [3\ 4]) \\
&= cat : h\ ([1\ 3], [3\ 7]) \\
&= cat : (id,\ (!\ \oplus)) : broadcast\ ([1\ 3],\ [3\ 7]) \\
&= cat : (id,\ (!\ \oplus))\ ([1\ 3],\ [(3,3)\ (7,3)]) \\
&= cat\ (id\ [1\ 3],\ (!\ \oplus)\ [(3,3)\ (7,3)]) \\
&= cat\ ([1\ 3],\ [6\ 10]) \\
&= [1\ 3\ 6\ 10]
\end{aligned}
$$

### 3.1.5 Premorphismus

Wenn die *postadjust*-Funktion die Identität ist (*reverse*), heißt die *DC*-Funktion *Premorphismus*.

#### Reverse

Die *reverse*-Funktion kehrt die Ordnung der Vektorelemente um:

$$reverse\ v = r, \text{ mit } r[i] = v[(N-1) - i]\ ;\ i = 0, ..., N-1$$

Nach der *links/rechts* Teilung des Vektors tauscht die *preadjust*-Funktion $g$ die entsprechenden *links/rechts* Teile, und dann wird die Rekursionsfunktion angewendet:

$$reverse = DC\ (div_{lr},\ com_{lr},\ g,\ id,\ p,\ id) \text{ mit } g = loc : correspondent$$

wobei: $loc = change(l,\ r) = (r,\ l)$

$$
\begin{aligned}
reverse : cat\ ([1\ 2], [3\ 4]) &= cat : id : (!\ reverse) : g\ ([1\ 2], [3\ 4]) \\
&= cat : id : (!\ reverse)\ ([3\ 4],\ [1\ 2]) \\
&= cat\ ([4\ 3],\ [2\ 1]) \\
&= [4\ 3\ 2\ 1]
\end{aligned}
$$

### 3.1.6 Pseudomorphismus

#### Polynomprodukt

Die im Kapitel 2 definierte Funktion *poly* berechnet das Polynomprodukt. Keine von den beiden Anpassungsfunktionen darf die Identitätsfunktion sein. Die *poly*-Funktion ist ein Pseudomorphismus.

$$poly\ v = r, \text{ mit } v = zip(x,\ y),\ \ r = x\ *\ y$$

$$\begin{gathered} poly = DC\ (div_{lr},\ com_{lr},\ g,\ h,\ p,\ b) \\ \text{mit } g = loc_g : correspondent,\ h = loc_h : correspondent \end{gathered}$$

*wobei* :

- für Dreiteilung :

$$\begin{aligned} loc_g\ ([x_l\ \ y_l],\ [x_r\ \ y_r]) =&([x_l\ \ y_l],\ [x_l + x_r\ \ y_l + y_r],\ \ [x_r\ \ y_r]) \\ loc_h\ ([h_h\ \ h_l],\ \ [m_h\ \ m_l],\ \ [l_h\ \ l_l]) =& \\ &([h_h\ \ l_h{+}m_l - h_l - l_l],\ \ [h_l + m_h - h_h - l_h\ \ l_l]) \\ b\ ([x\ \ y]) =&([0\ \ x * y]) \end{aligned}$$

$$\begin{aligned} poly:\ \ cat\ ([1\ 1],[1\ 1]) &= cat : h : (!\ poly) : g\ ([1\ 1],\ [1\ 1]) \\ &= cat : h : (!\ poly)\ ([1\ 1],\ [2\ 2],\ [11]) \\ &= cat : h\ ([0\ 1],\ [0\ 4],\ [0\ 1]) \\ &= cat\ ([0\ 1],\ [2\ 1]) \\ &= [0\ 1\ 2\ 1] \end{aligned}$$

- für Vierteilung :

$$\begin{aligned} loc_g\ ([x_l\ \ y_l],\ \ [x_r\ \ y_r]) =&([x_l\ \ y_l],\ [x_l\ \ y_r],\ [x_r\ \ y_l],\ \ [x_r\ \ y_r]) \\ loc_h\ ([h_h\ \ h_l],\ [m_{hh}\ \ m_{hl}],\ \ [m_{lh}\ \ m_{ll}],\ \ [l_h\ l_l]) =& \\ &([h_h\ \ h_l{+}m_{hh} + m_{lh}],\ \ [l_h + m_{hl} + m_{ll}\ \ l_l]) \\ b\ ([x\ \ y]) =&([0\ \ x * y]) \end{aligned}$$

## 3.2 Funktionen höherer Ordnung

Wir können DC-Algorithmen durch ein funktionales Programmschema mit Hilfe der *Funktionen höherer Ordnung* (d.h. Funktionen, die Funktionen als Argumente enthalten) ausdrücken [Col89]. In der Algorithmenstruktur sind eindeutig drei Phasen erkennbar, die jeweils durch eine Funktion realisiert werden können. Diese Funktionen, die wir problemspezifische Funktionen nennen, sind vom speziellen Algorithmus abhängig.

Die Begriffe in den in diesem Kapitel vorgestellten Modellen weichen von einander ab. Es folgt eine Zuordnungstabelle, um die Differenzen zu beseitigen und eine bessere Übersicht zu geben:

| Abschnitt 3.1 | Abschnitt 3.2 |
|---|---|
| $g \circ div$ | $d$ |
| $com \circ h$ | $c$ |
| *correspondent*-Kommunikation | *elementweise* Listenoperationen |
| $!f$ | $map\ f$ |
| $:$ | $\circ$ |

### 3.2.1 Allgemeines DC-Skelett

Die *DC*-Funktion löst ein Problem, das durch den Parameter $x$ beschrieben wird, anhand der DC-Methode. Das Eingabeproblem steht in einer Liste und ist von beliebigem Typ:

$$DC\ d\ c\ p\ b\ x = f\ x$$
$$\text{wobei: } f\ x = p\ x \rightarrow b\ x\ ;\ (c \circ (map\ f) \circ d)\ x$$

Die *DC*-Funktion ist vom Typ:

$$DC\text{: } (\alpha \rightarrow [\alpha]) \rightarrow ([\beta] \rightarrow \beta) \rightarrow (\alpha \rightarrow bool) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$

und hat fünf Eingabeparameter:

- $d$ zerlegt ein Problem in eine Liste von Problemen:
  $d$: $\alpha \rightarrow [\alpha]$ (*divide*)
- $c$ faßt Teillösungen zu einer Gesamtlösung zusammen:
  $c$: $[\beta] \rightarrow \beta$ (*combine*)
- Das Prädikat $p$ sagt aus, ob die Probleminstanz einfach ist und bestimmt, wann die Rekursion abbricht:
  $p$: $\alpha \rightarrow bool$ (*predicate*)
- $b$ löst das einfache Problem nach dem Abbruchsfall:
  $b$: $\alpha \rightarrow \beta$ (*basic*)

- $x$ ist das ursprüngliche Problem:
  $x$: $\alpha$ (*Eingabedaten*)

Die Funktionen $p$, $b$, $d$ und $c$ sind von dem speziellen DC-Algorithmen abhängig. Die Funktion $p$ nimmt das Problem als Argument und liefert einen boolschen Wert zurück. Sie bestimmt, ob das Problem trivial lösbar ist oder der mögliche Grad an Parallelität erreicht wurde.
Gibt $p$ `TRUE` zurück, so wird das Problem direkt mittels $b$ gelöst, andernfalls wird das Problem mittels $d$ in eine Liste von Teilproblemen aufgeteilt. Diese Teilprobleme werden autonom mittels *map f* gelöst. Die *map*-Funktion ist eine Standardfunktion, die selbst als ein Skelett betrachtet werden kann. Sie wendet eine Funktion (erster Parameter) auf jedes Element einer Eingabeliste (zweiter Parameter) an.
Die $c$-Funktion nutzt die Liste der Teillösungen, um das ursprüngliche Problem zu lösen.
Die Anwesenheit der Funktion höherer Ordnung *map* reflektiert die Parallelität in der Spezifikation, da alle Elemente der entsprechenden Listen simultan bearbeitet werden können. Die Listenelemente sind wieder rekursive Funktionen.
Allgemein betrachtet sind $d$ und $c$ Operationen, die auf eine Datenmenge angewendet werden, um sie zu zerlegen oder zusammenzusetzen.

### 3.2.2 Berechnungsgraph

Der mit der *DC*-Funktion verbundene Berechnungsgraph (*call graph*) ist ein Baum mit dem Verzweigungsgrad $k$, der wie folgt entsteht:

- Das zu lösende Problem liegt in der Wurzel des Baumes.
- Das Problem in der Wurzel wird in $k$ Teile aufgespalten und an die Kinder zur Lösung weitergegeben.
- Das Aufspalten wird rekursiv wiederholt, bis das Problem einfach zu lösen ist (d.h. die Blätter erreicht sind).
- Die Blätter des Baumes lösen ihre Teilprobleme und geben die Ergebnisse an die Eltern zurück.
- Die Eltern (die inneren Knoten) fassen die Teilergebnisse, die sie von den Kindern erhalten haben zusammen und geben das Resultat an ihre Eltern weiter,
- Das Zusammensetzen wird rekursiv wiederholt, bis die Wurzel erreicht wird.

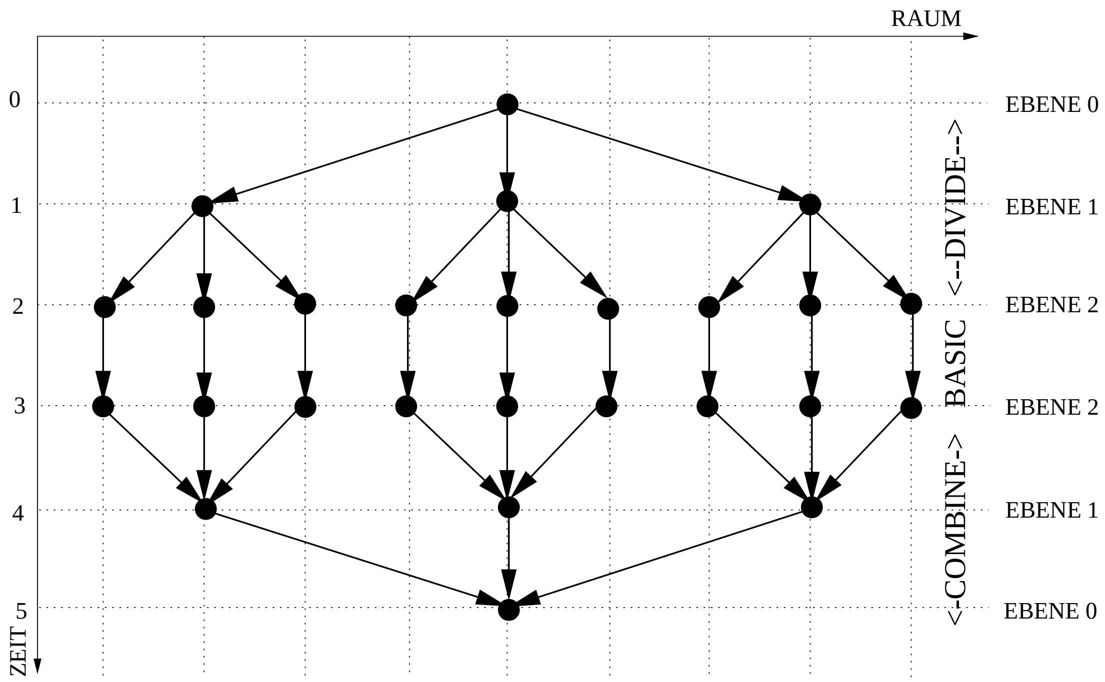



Abbildung 3.2: DC-Berechnungsgraph ($d = 2$, $k = 3$)

Abb. 3.2 zeigt ein Beispiel für einen Berechnungsgraphen. Die Knoten, die auf der gleichen Ebene liegen, können parallel berechnet werden. Ein DC-Skelett hat eine offensichtliche parallele Implementierung, bei der ein Baum $k$-ten Grades von Prozessen ($Prozeßbaum$) auf einen Baum von Prozessoren ($Prozessorbaum$) abgebildet wird [GL95].

### 3.2.3 Balanciertes fixed degree DC-Skelett

Wir schränken die Klasse der DC-Algorithmen auf eine Unterklasse ein. Die Unterklasse heißt *balancierter fixed degree* DC (FDDC). Diese Unterklasse wird von DC-Algorithmen gebildet, deren Berechnungsgraph balanciert ist und deren Verzweigungsgrad $k$ für eine spezielle Instanz konstant ist. Die $d$-Funktion ist $balanciert$, wenn für ein beliebiges teilbares Element $S$ aus der Menge $\mathcal{S}$ eine Konstante $m > 1$ existiert, sodaß $dS = (S_0, ..., S_{k-1}) \rightarrow |S_i| \leq |S|/m$, für $i = 0, ..., k - 1$.

Ein Berechnungsgraph heißt $balanciert$, wenn seine Höhe (die Entfernung von der Wurzel zu den Blättern) $H = log_m|S|$ ist, wobei $|S| = N$. Das ist der Fall, wenn die $d$-Funktion balanciert ist. Balancierte Berechnungsgraphen erreichen wir, indem wir $N$ auf eine Zweierpotenz und den Datenteilungsgrad auf 2 einschränken. Eine gute Grundlage einer Datenstruktur für die Elemente aus $\mathcal{S}$ bieten deswegen $Powerlisten$ [Mis94]. Eine $Powerliste$ ist eine geordnete Liste der Länge $2^n$ ($n \geq 0$), wobei die Elemente der Liste vom selben Typ sind: skalar, Powerliste, Powerliste von Powerliste, etc. Zusammen mit einigen grundlegenden Operatoren, die auf der Powerliste definiert sind, ist sie gut geeignet, parallele, rekursive Algorithmen zu beschreiben und Korrektheitsbeweise durchzuführen. Wir unterscheiden weiter zwischen Datenteilung und Problemteilung:

- Datenteilung: Ein Element der Größe $2^n$ wird in der Mitte in zwei Hälften geteilt: in den linken Teil *left* und in den rechten Teil *right*. Es ist *teilbar* für $n > 0$.

- Problemteilung: Ein Problem der Größe $2^n$ wird in jeder Ebene in $k$-Teile der Größe $2^{n-1}$ zerlegt. Alle Teile sind voneinander unabhängig und können parallel berechnet werden. Nach $n$ Teilungen eines Problems mit dem Teilungsgrad $k$ entstehen also $k^n$ Berechnungspunkte.

Die Knoten des Prozessorbaumes sind Prozessoren. Es werden also $\sum_{i=0}^{n-1} k^i$ Prozessoren benötigt, und es gibt $k^n$ Berechnungspunkte für die Berechnung des Basisfalles. Für eine effizientere Implementierung wäre eine parallele Architektur mit $k^n$ Prozessoren besser geeignet. Eine solche Architektur wird im Abschnitt 4.1 vorgeschlagen.

**Beispiel 3.2.1**

Ein Beispiel eines funktionalen balancierten FDDC-Skeletts auf Powerlisten in der Notation von Haskell [HJW92] bietet [HL95].

```
divcon :: Int→(a→b)→(a→a→[a])→([b]→(b, b))→[a]→[b]

divcon k basic divide combine = solve
where solve indata =
        if length indata == 1
          then map basic indata
          else let x = zipWith divide (left indata) (right indata)
                   y = transpose (kmap k solve (transpose x))
               in  map ((fst . combine) y) ++ (map (snd . combine) y )
```

| | | | |
|---|---|---|---|
| *length* | | $[a_0,...,a_{u-1}]$ | $= u$ |
| *map* | $f$ | $[a_0,...,a_{u-1}]$ | $= [f\ a_0,...,f\ a_{u-1}]$ |
| *kmap* | $k\ f$ | $[a_0,...,a_{k-1}]$ | $= [f\ a_0,...,f\ a_{k-1}]$ |
| *left* | | $[a_0,...,a_{2*u-1}]$ | $= [a_0,...,a_{u-1}]$ |
| *right* | | $[a_0,...,a_{2*u-1}]$ | $= [a_u,...,a_{2*u-1}]$ |
| *zipWith* | $f$ | $[a_0,...,a_{u-1}]\ [b_0,...,b_{u-1}]$ | $= [f\ a_0\ b_0,...,f\ a_{u-1}\ b_{u-1}]$ |
| *transpose* | | $[[a_{(0,0)},...,a_{(0,u-1)}],...,[a_{(v-1,0)},...,a_{(v-1,u-1)}]]$ | $= [[a_{(0,0)},...,a_{(v-1,0)}],...,[a_{(0,u-1)},...,a_{(v-1,u-1)}]]$ |
| *fst* | | $(a,b)$ | $= a$ |
| *snd* | | $(a,b)$ | $= b$ |
| *“append”* | | $[a_0,...,a_{u-1}] +\!\!+ [a_u,...,a_{2*u-1}]$ | $= [a_0,...,a_{2*u-1}]$ |

Wir lassen nur komponentenweise Listenoperationen und *left/right* Listenpartitionierung zu. Die problemspezifischen Funktionen: $k$, *basic*, *divide*, *combine* operieren auf einer Liste der Länge $2^n$ und liefern eine Liste der gleichen Länge aber möglicherweise anderen Typs zurück. Es kann vorkommen, daß die Funktionen mehr als ein Argument verlangen, z.B. im Fall des Polynomprodukts. Dann müssen die Argumente verzahnt werden; statt eines Tupels von Powerlisten nehmen wir Powerlisten von Tupeln). Die Polynome werden durch eine Liste ihrer Koeffizienten repräsentiert. Die *divide*-Funktion wird auf alle Elemente der verzahnten linken und rechten Teile der Liste solange rekursiv angewendet, bis die Liste nur ein Element erhält. Die *basic*-Funktion bekommt eine Powerliste von einzelnen Elementen und wird auf jedes Element angewendet. Sie liefert ein Element zurück, das einen anderen Typ als die Eingabe haben kann. Die *divide*-Funktion wird elementweise auf eine Powerliste angewendet und liefert eine Powerliste mit $k$-elementigen Listen von Powerlisten der halben Länge zurück. Die Liste von $k$-elementigen Listen von Powerlisten muß transponiert werden, um $k$ Powerlisten zu bekommen. *kmap k* wendet die Funktion *solve* auf die $k$-Teilprobleme ($k$ Powerlisten), und die Rekursion geht weiter bis wir eine $k$-elementige Liste von unteilbaren Powerlisten erreichen. Die Ergebnisliste muß wieder transponiert werden, um die vorherige Transposition rückgängig zu machen. Die *combine*-Funktion hat zwei Komponenten. Die erste Komponente wird auf den linken Teil der Ergebnisliste angewendet und die zweite Komponente auf den rechten Teil. Die Ergebnisliste ist eine Powerliste.

**Beispiel 3.2.2**

Das Skelett aus Bsp. 3.2.1 bekommt folgende problemspezifische Funktionen für das Karatsuba-Verfahren zum Polynomprodukt (Dreiteilung):

$$
\begin{array}{lll}
\multicolumn{3}{l}{karat\ a\ b = (map\ fst\ y) +\!\!+ (map\ snd\ y)} \\
\multicolumn{3}{l}{\textsf{where}\ y = divcon\ 3\ basic\ divide\ combine\ (zip\ a\ b)} \\
basic & (x, y) & = (0, x * y) \\
divide & (xh, yh)\ (xl, yl) & = [(xh, yh), (xh + xl, yh + yl), (xl, yl)] \\
combine & [(hh, hl), (mh, ml), (lh, ll)] & = ((hh, lh + ml - hl - ll), \\
 & & \qquad (hl + mh - hh - lh, ll))
\end{array}
$$

**Beispiel 3.2.3**

Durch die Anwendung der *karat*-Funktion ($karat\ [1, 2, 1, 0]\ [0, 1, 1, 1]$) auf zwei Polynome:

$$A = x^3 + 2x^2 + x,\ B = x^2 + x + 1$$

entsteht der Berechnungsgraph in Abb. 3.3.

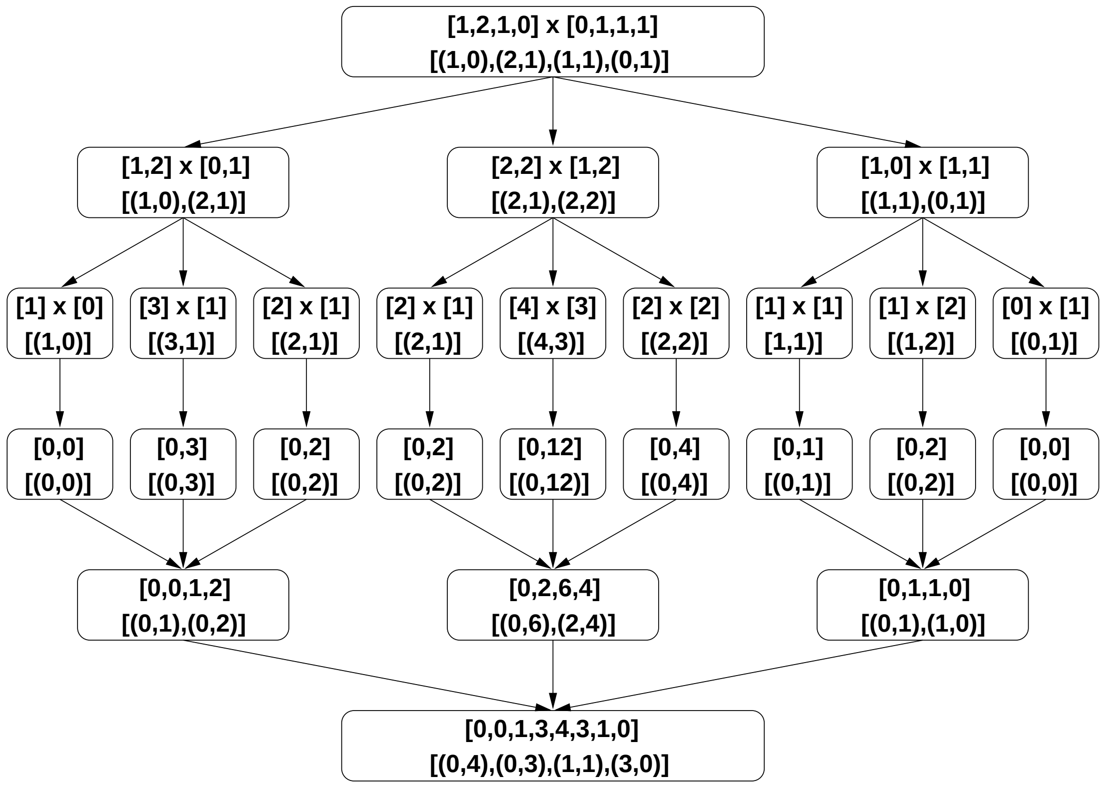



Abbildung 3.3: Berechnungsgraph für das *karat*-Polynomprodukt

# Kapitel 4

# Das Modell für das imperative Skelett

## 4.1 Virtuelle Topologie

Der Berechnungsgraph für den balancierten FDDC-Algorithmus wird auf ein $n$-dimensionales Gitter abgebildet. Die Anzahl $n$ der Dimensionen ist gleich der Höhe $h$ des Berechnungsgraphen, unter der Annahme, daß wir genug Prozessoren zur Verfügung haben, und die Ausdehnung in jeder Dimension ist gleich dem Teilungsgrad $k$ des Problems.

**Definition 4.1.1 ($n$-dimensionales $k$-Gitter)**

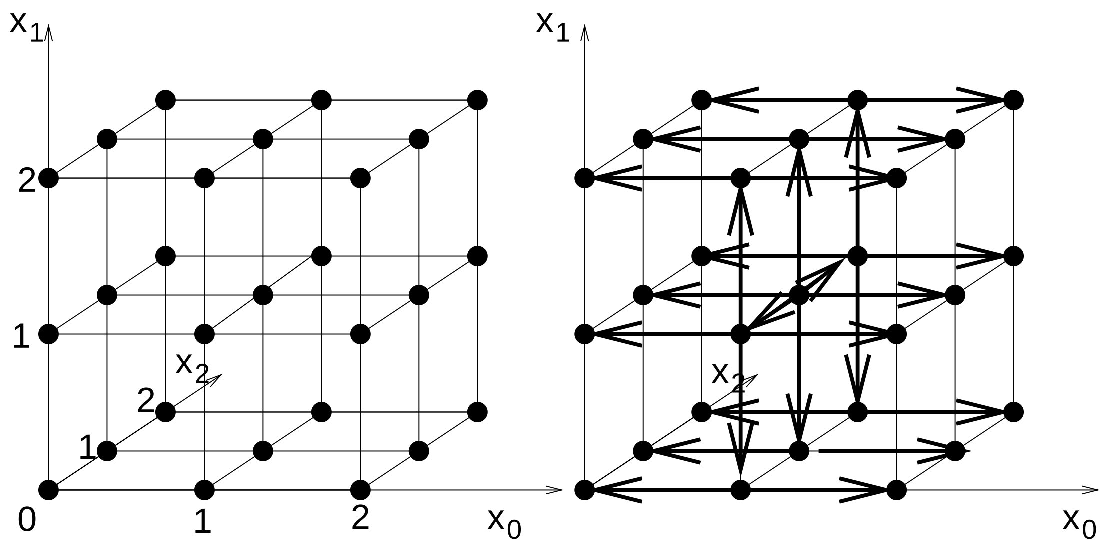
Ein $n$-dimensionales $k$-Gitter ist ein ungerichteter Graph $M_v = (V_v, E_v)$ mit $|V_v| = k^n$, wobei $k$ die Ausdehnung des Gitters in jeder Dimension ist. Jeder Knoten wird eindeutig durch ein $n$-Tupel $(x_0, ..., x_{n-1})$ identifiziert, wobei $x_i$ die Koordinate des Gitters in der Dimension $i \in \{0, ..., n-1\}$ ist. Zwei Knoten im Gitter sind miteinander verbunden, wenn sie sich in genau einer der Koordinaten um 1 unterscheiden. Eigenschaften eines $n$-dimensionales $k$-Gitters:

- Jeder innere Knoten hat genau $2 * n$ Nachbarn, jeder äußere Knoten hat $2 * (n-1)$, $2 * n - 1$ oder $n$ Nachbarn.
- Durchmesser des Gitters (= der maximale Abstand zwischen zwei Knoten) beträgt $n * (\sqrt[n]{p} - 1)$, wobei $p = k^n$ die Anzahl der Prozessoren ist.

**Beispiel 4.1.1**

Abb. 4.1 zeigt ein Beispiel für ein Gitter und seinen aufspannenden Berechnungsgraphen.

Der Graph aus Def. 4.1.1 ist ein Modell für eine parallele Berechnung, da ein paralleler Algorithmus ein Netzwerk kommunizierender sequentieller Prozesse ist




(a) 3-dim 3-Gitter (b) aufspannender Berechnungsgraph

Abbildung 4.1: 3-dimensionales 3-Gitter und aufspannender Berechnungsgraph

[LRG+90]. Die Prozesse werden während der Berechnung auf einem Prozessor des Rechnernetzes ausgeführt. Jeder Knoten im Gitter repräsentiert also einen Prozessor, und jede Kante zwischen zwei Knoten repräsentiert eine (möglicherweise bidirektionale) Kommunikationsverbindung (*Link*). Mit Hilfe der Knotenkoordinaten kann die Adresse des entsprechenden Prozessors bestimmt werden. Das ist aber kein Prozessoridentifikator (*PID*) im Transputernetz, sondern eine virtuelle Adresse in unserer virtuellen Gittertopologie (*VPID*). Diese virtuelle Prozessoradresse wird durch folgende Funktion angegeben:

$$vpid : (x_0, ..., x_{n-1}) \rightarrow \sum_{i=0}^{n-1} x_i * k^i$$

Die Ein/Ausgabedaten befinden sich in den Knoten des Gitters. Es gibt zwei Möglichkeiten der Datenverteilung auf die $k^n$ Prozessoren:

- Zentralisiert: Die Wurzel des Berechnungsgraphen wird auf den Knoten $(x_0, ..., x_{n-1})$ abgebildet, wobei $x_i = rightpos$ für $i = 0, ..., n-1$ und $rightpos = \lfloor k/2 \rfloor$.

- Verteilt: Die Eingabedaten sind am Anfang in $2^n$ Partitionen zerlegt. Die Partitionen entstehen durch rekursive *links/rechts* Teilung der Daten. Die Wurzel des Berechnungsgraphen wird auf $2^n$ Knoten $(x_0, ..., x_{n-1})$ abgebildet, wobei $x_i = rightpos \vee x_i =$*leftpos* für $i = 0, ..., n-1$ und $rightpos = \lfloor k/2 \rfloor$, *leftpos*$= rightpos - 1$. Die Zuordnung der Partitionen zu den Knoten ist wie folgt angegeben:

$$part : vpid \rightarrow \{-1, 0, ..., 2^d - 1\}$$

**Beispiel 4.1.2**

Abb. 4.2 zeigt die Knoten des Gitters, die die Ein/Ausgabe enthalten ($k = 4$, $d = 2$). Die Verteilung der Partitionen auf die Knoten berechnet die *part*-Funktion anhand der *vpid*-Nummern. Im verteilten Fall (*leftpos*$= 1$, *rightpos*$= 2$) gibt es $2^d = 4$ Partitionen:
$part(5) = 0,\ part(6) = 1,$
$part(9) = 2,\ part(10) = 3.$
Auf die anderen Knoten wird die Wurzel nicht abgebildet ($part = -1$).

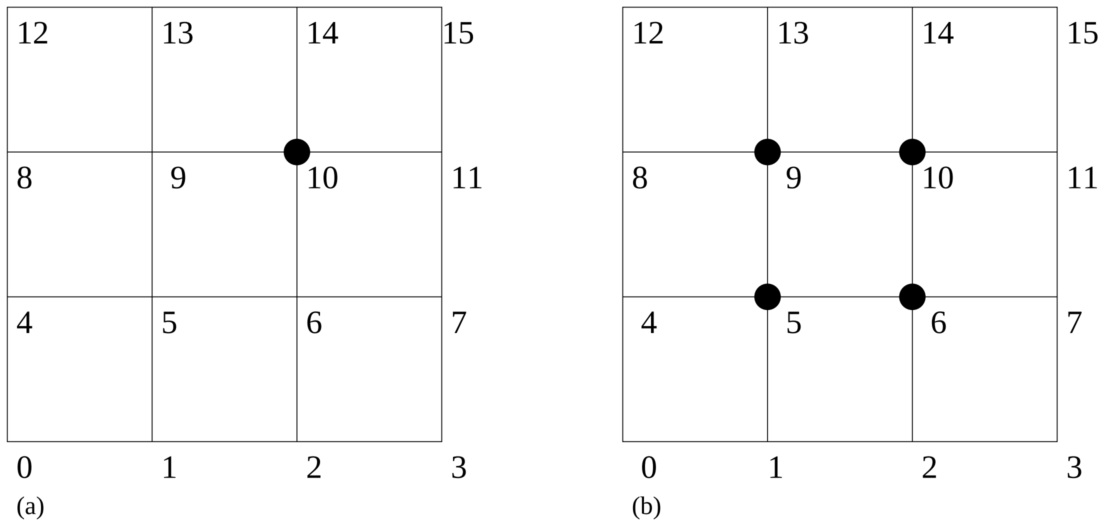



Abbildung 4.2: (a) Zentralisierte Ein/Ausgabe, (b) Verteilte Ein/Ausgabe

Um das zeitliche Verhalten des Modells darstellen zu können, führen wir eine zusätzliche Dimension ein, die sog. *Tiefendimension* (*depth dimension*). Die Tiefendimension wird mit der Zeit assoziert, die übrigen $n$ Dimensionen spannen das $n$-dimensionale $k$-Gitter auf und werden mit dem Raum oder mit der Zeit assoziert, abhängig davon, wieviele Prozessoren zur Verfügung stehen. Dieses Modell (siehe Abb. 4.4) wird in [HL95] *Berechnungsdomäne* (*computation domain*) genannt.

## 4.2 Das Schleifenprogramm

Zu dem funktionalen Programschema gibt es ein entsprechendes imperatives Schleifenprogramm mit sequentiellen und parallelen Schleifen. Das Programm enthält drei zweifach vernestete Schleifen; sie entsprechen der Teilungsphase, der Basisphase und der Zusammensetzungsphase. Die äußere Schleife für $d$ wird mit der Tiefendimension identifiziert und die innere Schleife für $q$ wird mit $n$ anderen Dimensionen der Berechnungsdomäne identifiziert. Die $n$ Dimensionen werden in die Zeit und den Raum verteilt. Die innere Schleife hat eine sequentielle (für die Zeit) und eine parallele (für den Raum) Komponente. $q$ wird als ein Vektor $(q[0], ..., q[n-1])$ repräsentiert. Die Elemente $q[i]$ des Vektors stehen für die Dimensionen der Berechnungsdomäne ohne die Tiefendimension. Die Notation für die Substitution von $q$ stammt aus [Gri91] und bedeutet: $(q; i : a)$ ist gleich $q$ mit Ausnahme der Position $i$, die $a$ enthält.

$$
\begin{array}{l}
\textsf{for}\ \ d = 1, ..., n \\
\quad \textsf{for}\ \ q = 0, ..., k^n - 1 \\
\quad\quad A[d,q] = \ divide[q[n-d]]\ \ A[d-1, (q; n-d : leftpos)] \\
\quad\quad\phantom{A[d,q] = \ divide[q[n-d]]\ \ } A[d-1, (q; n-d : rightpos)] \\
\textsf{for}\ \ d = n+1 \\
\quad \textsf{for}\ \ q = 0, ..., k^n - 1 \\
\quad\quad B[d,q] = basic\ A[d-1, q] \\
\textsf{for}\ \ d = n+2, ..., 2*n+1 \\
\quad \textsf{for}\ \ q = 0, ..., k^n - 1 \\
\quad\quad C[d,q] = (B[d-1, (q; d-(n+2) : 0)], \ldots, \\
\quad\quad\phantom{C[d,q] = (} B[d-1, (q; d-(n+2) : k-1)]) \\
\quad\quad B[d, (q; d-(n+2) : leftpos)] \ = (fst\ \ .\ combine)\ C[d,q] \\
\quad\quad B[d, (q; d-(n+2) : rightpos)] = (snd\ .\ combine)\ C[d,q]
\end{array}
$$

$A$ und $B$ sind Vektoren indiziert mit Punkten der Berechnungsdomäne. Die Elemente von $A$ enthalten die Eingabedaten, die Elemente von $B$ die Ausgabedaten. Die Elemente von $C$ sind $k$-Tupel der Elemente aus $B$, und sie bilden die Eingabe der *combine*-Funktion. Die Kommunikation in der Teilungsphase findet immer entlang der Dimension $q[n-d]$ für die Tiefendimension $d$ statt. Deswegen selektieren wir die Komponenten des Argumentvektors, auf die *divide* angewendet wird. Wir gehen ähnlich in der Zusammensetzungshase vor, die Kommunikation findet jetzt entlang der Dimension $q[d]$ statt.

## 4.3 Kommunikation

Eine Kommunikation zwischen den Prozessoren findet in zwei Berechnungsphasen statt: in der Teilungsphase und in der Zusammensetzungsphase. Wir brauchen also zwei Kommunikationsmuster [MCH92], und jeder enthält $log(N)$ Schritte:

- für die absteigende Kommunikation (*descend*) und
- für die aufteigende Kommunikation (*ascend*).

**Definition 4.3.1 (Kommunikationsmuster)**

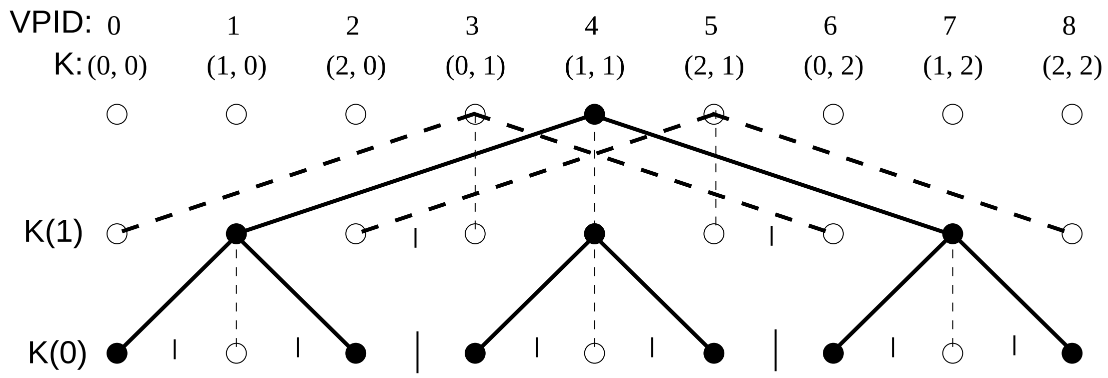
Das Kommunikationsmuster ist auf einem Vektor $X$ der Länge $k^n$ definiert, dessen Elemente von 0 bis $k^n - 1$ indiziert sind: $X = (x_0, ..., x_{k^n-1})$. Der Index repräsentiert die Adresse eines Knotes im Gitter in der Koordinatendarstellung. Der Vektor ist polymorph, d.h. der Kommunikationspartner wird aus seinem Vektorindex bestimmt, unabhängig von seinem Wert. Der $d$-te Kommunikationsschritt, $d = 0, ..., n-1$, auf dem Vektor $X$ selektiert die Kommunikationspartner, deren Adressen sich nur in der $(n-d)$-ten Koordinate unterscheiden (d.h. es gibt $k-1$ Kommunikationsmöglichkeiten).

**Beispiel 4.3.1**

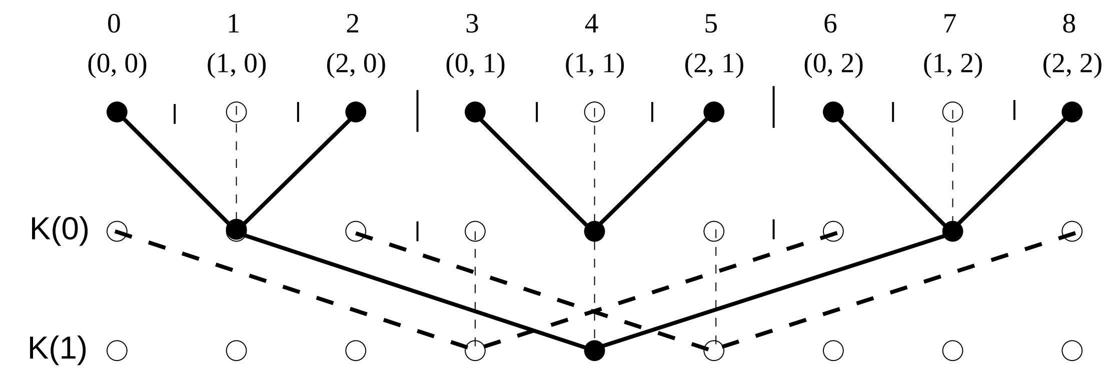
Abb. 4.3 zeigt die Kommunikationsmuster für die absteigende und aufsteigende Kommunikation. Die gestrichelte Linien bedeuten Kommunikation, die keinen Einfluß auf die Berechnung hat, da wir die Daten im Baum verschicken.

Der Entwurfsschritt muß noch verfeinert werden, da die wirkliche Kommunikation nur zwischen den Nachbarknoten (d.h. Knoten, die sich in $(n-d)$-ter Koordinate um 1 unterscheiden) stattfindet [HL95]. Der Übergang vom $d$-ten zu $(d+1)$-ten Kommunikationsschritt muß in kleinere Kommunikationsschritte zwischen den Nachbarknoten aufgeteilt werden. Wir verfeinern eine Koordinate in $\lceil k/2 \rceil + 2$ Koordinaten (im verteilten Fall), in $\lceil k/2 \rceil + 1$ Koordinaten (im zentralisierten Fall). Die neue Koordinate ist ein Paar: das erste Element ist die Koordinate vor der Verfeinerung, das zweite entspricht einem Zwischenschritt entlang der Tiefendimension. Abb. 4.5 zeigt die verfeinerten absteigenden Kommunikationsmuster entlang einer Dimension $d$ für zentralisierte Eingabedaten, und Abb. 4.7 für die verteilten Eingabedaten.

Die aufsteigende Kommunikation kann ähnlich definiert werden, mit dem Unterschied, daß die Koordinaten in die andere Richtung durchlaufen werden. Die Abb. 4.6 und 4.8 zeigen die verfeinerten aufsteigenden Kommunikationsmuster entlang einer Dimension $d$. Während der absteigenden Kommunikation wird der Vektor rekursiv in $k$ Teile geteilt. Analog wird der Vektor während der aufsteigenden Kommunikation aus $k$ Teilen zusammengesetzt.
Den parallelen DC-Algorithmus kann man also wie folgt formulieren:




(a) descend-Kommunikation




(b) ascend-Kommunikation

Abbildung 4.3: Kommunikationsmuster ($n$=2, $k$=3)

for $i = 0...log(N)\{$
*absteigende Kommunikation* (*Inter-Prozessor Kommunikation*)
*lokale Berechnung* (*Inter-Vektor Kommunikation*)}

Lösung des Basisfalles

for $i = 0...log_k(N)\{$
*aufsteigende Kommunikation* (*Inter-Prozessor Kommunikation*)
*lokale Berechnung* (*Inter-Vektor Kommunikation*)}

## 4.4 Das SPMD-Modell

Nach Flynns Taxonomie [Brä93] gibt es: SISD, SIMD, MISD, MIMD Rechnerarchitekturen. SIMD und MIMD charakterisieren zwei parallele Berechnungsmodelle, die sich in der Anzahl der Kontrolleinheiten unterscheiden, die die Berechnung auf einer Menge von Daten durchführen. SIMD Rechner sind einfacher aufgebaut

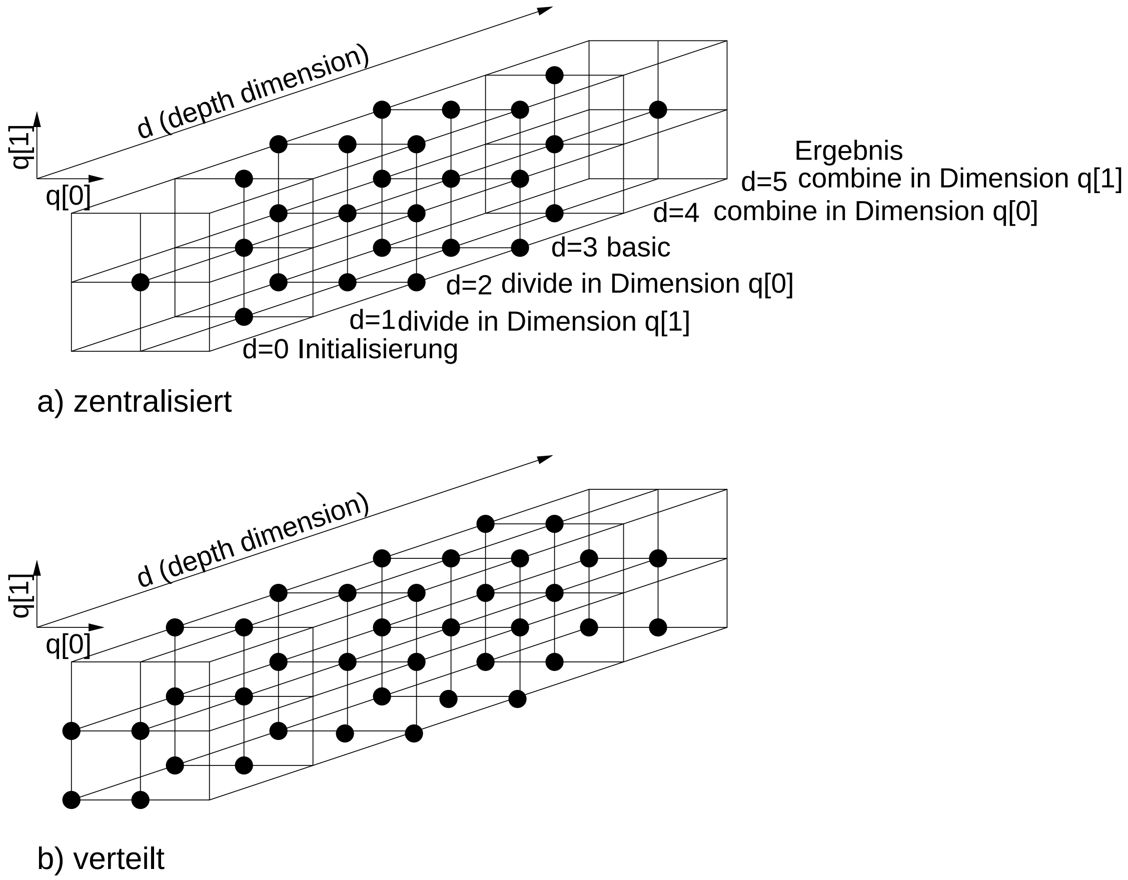



Abbildung 4.4: Berechnungsdomäne ($n$=2, $k$=3) (a)verteilte Ein/Ausgabedaten, (b)zentralisierte Ein/Ausgabedaten

als MIMD Rechner. Die Hardware für den Befehlszyklus ist nur einmal im zentralen Hostrechner vorhanden. Die Prozessoren bestehen nur aus einer arithmetisch logischen Einheit, lokalem Speicher und einer Kommunikationseinrichtung. Da nur ein Befehlsgeber vorhanden ist, ist die Ausführung eines SIMD Programms immer synchron. Es gibt nur einen einzigen Kontrolfluß; jeder Prozessor führt entweder die gleiche Instruktion aus oder ist inaktiv. MIMD Rechner besitzen im Vergleich zu SIMD die allgemeinere Struktur und arbeiten stets asynchron. Jeder Prozessor führt ein eigenständiges Programm aus und besitzt einen eigenen Kontrollfluß. Im MIMD System existieren also mehrere Kontrollflüße. Man unterscheidet eng gekoppelte MIMD Rechner (mit gemeinsamen Speicher) und lose gekoppelte MIMD Rechner (ohne gemeinsamen Speicher). Bei MIMD Rechnern läßt sich häufig das Phänomen beobachten, daß diese im SIMD Modus programmiert werden [Brä93]. Dies trifft recht oft bei der Programmzerlegung auf sehr viele Prozessoren, also bei massiver Parallelität, auf. Die meisten Probleme, die sich auf viele Prozessoren zerlegen lassen, weisen ein einheitliches Verarbeitungs-

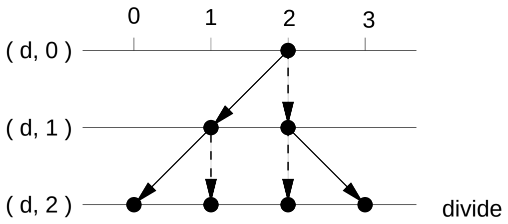



Abbildung 4.5: Schema der zentralisierten Datenverteilung entlang der Dimension $d$, $(k = 4)$

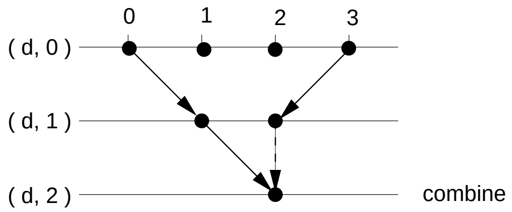



Abbildung 4.6: Schema der zentralisierten Datenzusammensetzung entlang der Dimension $d$, $(k = 4)$

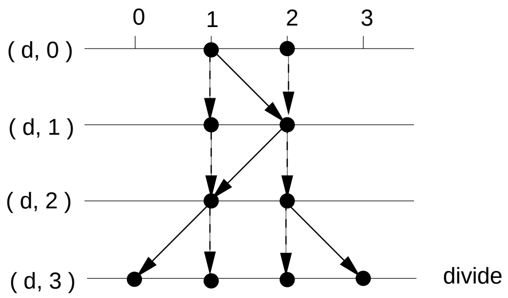



Abbildung 4.7: Schema der verteilten Datenverteilung entlang der Dimension $d$, $(k{=}4)$

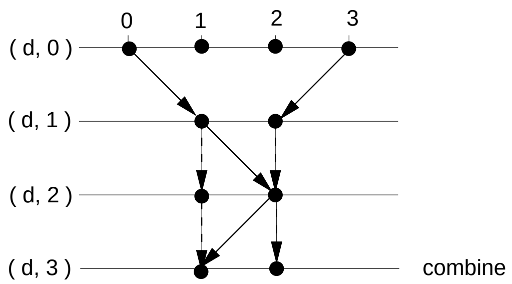



Abbildung 4.8: Schema der verteilten Datenzusammensetzung entlang der Dimension $d$, $(k{=}4)$

muster auf, zu deren Lösung ein einfacherer SIMD Rechner genügen würde. Dies ist ein bedeutendes Argument für Parallelrechner nach dem SPMD Modell. Das SPMD Modell ist eine Mischung, die versucht die Leistungsvorteile von SIMD zu nutzen und die Programm-Komplexität von MIMD zu vermeiden [Lew91]. Es gibt nur ein Programm, das in den Speicherbereich jedes Prozessors kopiert wird. Jeder Prozessor verfügt über einen eigenen Kontrollfluß. Verschiedene Prozessoren durchlaufen verschiedene Programmpfade. Die Berechnung kann synchron oder asynchron erfolgen. Jede Datenaustauschoperation ist eine Synchronisationsoperation.

Als parallele Architektur betrachten wir also ein Transputernetz, das nach dem Message-Passing-Prinzip arbeitet. Ein Transputernetz ist ein Netzwerk von Prozessoren mit lokalem Speicher und einer Ein/Ausgabe-Einrichtung zum Senden und Empfangen von Nachrichten anderer Prozessoren des Netzwerks. Jeder Prozessor hat nur Zugang zu seinem lokalen Speicher, einen globalen Speicher gibt es nicht. Ähnlich kann man ein Transputernetz als Graph $M_r = (V_r, E_r)$ modellieren. Da unsere reale Architektur ein 2-dimensionales Gitter ist, wird jeder Knoten eindeutig durch ein Paar $(y_0, y_1)$ identifiziert.

## 4.5 Das Plazierungsproblem

Das *Plazierungsproblem* beinhaltet zwei Entscheidungen:

- Wie werden die Knoten der virtuellen Topologie auf die Knoten der realen Topologie abgebildet?
- Wie werden die Kommunikationskanten auf die Prozessorverbindungen übertragen?

Wir gehen im folgenden davon aus, daß die Anzahl der verfügbaren Prozessoren größer oder gleich der Anzahl der Prozessoren der virtuellen Topologie ist ($|V_v| \leq |V_r|$), sodaß jedem Prozeß der parallelen Berechnung ein eigener Prozessor zugewiesen werden kann. Formal wird das Plazierungsproblem durch zwei Funktionen spezifiziert, die wie folgt dargestellt werden können:

$$mapping_V : V_v \rightarrow V_r$$
$$mapping_E : E_v \rightarrow Path_r$$

wobei:
$mapping_E((a_v, b_v)|(a_v, b_v) \in E_v)$ der Pfad von $mapping_V(a_v)$ nach $mapping_V(b_v)$ in $M_r$ ist und $Path_r$ die Menge aller möglichen Pfade in $M_r$.

**Beispiel 4.5.1**

Abb. 4.9 zeigt ein Beispiel für eine Plazierung mit $mapping_V = id$. Für die $Path_r$-Verbindung in $M_r$ ist das Routing verantwortlich.

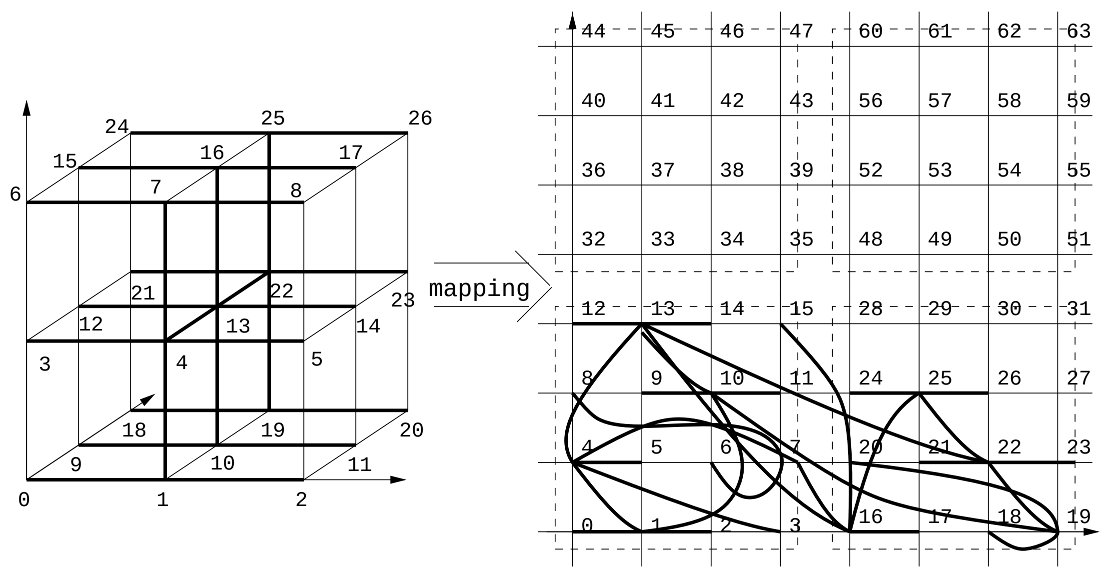



Abbildung 4.9: Plazierung der virtuellen Topologie auf die reale Topologie

## 4.6 Virtuelle Prozessoren

In der Realität ist die Anzahl der Prozessoren beschränkt. Wir sind jedoch daran interessiert, daß auch Probleme gelöst werden, die mehr Prozessoren benötigen, als vorhanden sind. In diesem Fall ist es nötig, entweder das Problem umzuformulieren, es sequentiell rekursiv zu lösen, oder nicht vorhandene Prozessoren zu simulieren. Wir haben $p$ Prozessoren zur Verfügung. Für die Problemgröße $n$ und dem Problemteilungsgrad $k$ brauchen wir $k^n$ Prozessoren. Übersteigt die Größe $k^n$ die Anzahl der Prozessoren $p$ um einen Faktor $M$, so muß jeder physische Prozessor $M$ virtuelle Prozessoren simulieren ($M = k^n/k^d = k^{n-d} = k^m$). Diese Simulation wird von einem Hilfsprogramm ausgeführt.

## 4.7 Compress-and-Conquer

Eine andere Möglichkeit, ein Problem, das mehr Prozessoren benötigt als vorhanden sind, zu lösen, besteht darin, das Originalproblem so umzuformulieren, daß das neue Problem durch die vorhandene Architektur gelöst werden kann. Diese Technik, die man als Compress-and-Conquer bezeichnet, kann jedoch nicht bei jedem Algorithmus angewendet werden, da nicht immer eine solche Umformung gefunden werden kann (z.B. Bitonic Sort, Fast Fourier Transformation, Polynomprodukt, ...). Ist die Technik anwendbar, so werden sowohl die Berechnungs-, als auch die Kommunikationskosten gesenkt. Wenn man wieder davon ausgeht, daß die Problemgröße $N$ die Prozessorenanzahl $p$ um den Faktor $M$ übersteigt, so hat die Compress-and-Conquer Technik folgende Struktur:

- Auf allen Prozessoren wird parallel eine Vorbehandlungsfunktion angewendet, die die sequentiellen Vektorkomponenten der Länge $M$ auf jedem Prozessor auf eine Komponente der Länge 1 reduziert.
- Das reduzierte Problem wird durch DC gelöst.
- Auf allen Prozessoren wird parallel eine Nachbehandlungsfunktion angewendet, um die Lösung in $M$ Originallösungen zu transformieren.

Die Kommunikationszeit fällt im Vergleich zu virtuellen Prozessoren um den Faktor $M$, da nur noch die Lösungen der reduzierten Probleme übermittelt werden und keine Kommunikation für die Berechnung der reduzierten Probleme erforderlich ist. Die lokale Berechnungszeit kann mit den virtuellen Prozessoren nicht verglichen werden, da sowohl eine Vor- als auch eine Nachbehandlung erforderlich ist. In manchen Fällen ist durch die Reduktion der $M$ Problemkomponenten eine effizientere und damit schnellere Lösung möglich.

**Beispiel 4.7.1 (*scan* $\oplus$ auf einem Vektor)**

$$A = (a_0, ..., a_{M-1}, a_M, ..., a_{2M-1}, ..., a_{(p-1)M}, ..., a_{pM-1})$$

Der Vektor $A$ hat $p$ parallele Komponenten und $M$ sequentielle Komponenten und kann umformuliert werden zu:

$$(a_0^{'}, ..., a_{p-1}^{'})$$

$$\begin{aligned} mit\ a_0^{'} &= (a_0, ..., a_{M-1}) \\ a_1^{'} &= (a_M, ..., a_{2*M-1}) \\ &\vdots \\ a_{p-1}^{'} &= (a_{(p-1)*M}, ..., a_{p*M-1}) \end{aligned}$$

Daraus folgt, daß man das Originalproblem um den Faktor $M$ reduzieren kann, wenn man schreibt:

$$scan \oplus (a_0^{'}, a_1^{'}, ..., a_{p-1}^{'})$$

- Die Daten werden auf jedem Prozessor eingelesen.
- Jeder Prozessor berechnet einen *scan* $\oplus$ seiner Werte.
- Nur auf dem letzten Wert jedes Prozessors wird *divide-basic-combine* ausgeführt.
- Jeder Prozessor $p$ schickt den *scan* $\oplus$-Wert zum Prozesor $p + 1$.
- Jeder Prozessor addiert den *scan* $\oplus$-Wert zu allen Elementen. Da wir uns nur mit dem Skelett befassen, sind Experimente mit der Datengröße 1 in jedem Prozessor durchaus sinnvoll. Die sequentiellen Fälle sind immer auch die Basisfälle.

## 4.8 Komplexität

Für die Zeitkomplexität müssen drei Zeiten betrachtet werden: $T_1$, $T_2$, $T_3$ [HL95]. Wenn die Problemgröße $N = 2^n$ ist, ist $n = \log_2 N$ die Rekursionstiefe. $d = \lfloor \log_k p \rfloor$ die Anzahl der parallelen Datenteilungsschritte und $Q = 2^{n-d}$ die Größe der Daten, die auf jedem Prozessor bleiben. Dann liegt

- die Zeit $T_1$ für die Berechnung des sequentiellen Falls in:

  $\mathcal{O}(Q * \log_k Q)$ für $k = 2$,
  $\mathcal{O}(Q^{\log_k Q})$ für $k > 2$,

- die Zeit $T_2$ für die parallele Teilung und Zusammensetzung in:
  $\mathcal{O}(d * Q)$ weil wir $d$ Teilungsschritte haben.
- die Zeit $T_3$ für den Kommunikationsaufwand bei der Plazierung auf ein niedrig-dimensionales Gitter der Dimension $s$ liegt in:
  $\mathcal{O}(Q * s * p^{\frac{1}{s}})$

Die Zeitkomplexität ist also $T_1 + T_2 + T_3$.

# Kapitel 5

# Implementierung

## 5.1 Aufbau

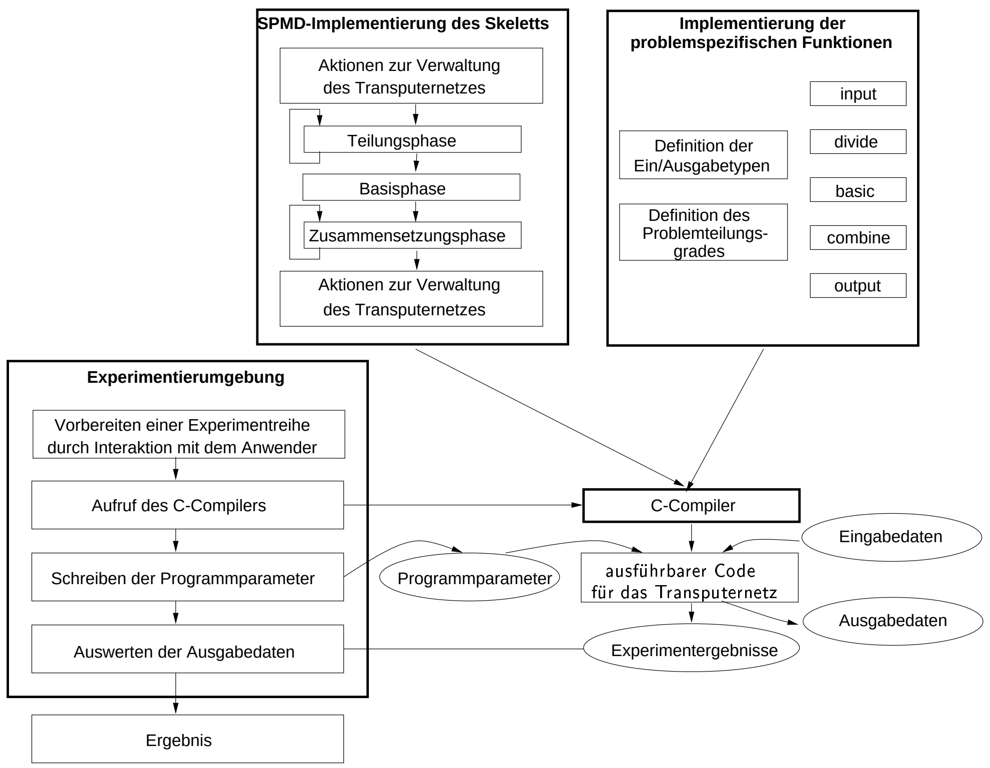



Abbildung 5.1: Verfeinerung der Entwurfsphasen

Wie bereits in Abb. 1.1 auf S. 7 im eingerahmten Teil gesehen, ergibt sich die DC-Implementierung durch Verbinden der Skelett-Implementierung und der Implementierung der problemspezifischen Funktionen. In Abb. 5.1 nach Verfeinerung der Enwurfsphasen ist es deutlicher zu sehen.

Das funktionale Schema, das als eine Funktion höherer Ordnung dargestellt wurde muß in die Form eines imperativen Programms (in C-Notation) umgesetzt werden. Das C-Programm ist also eine Schablone, die die Daten mittels Datenaustauschoperationen (send/receive) nach einem festen Kommunikationsschema (siehe Abb. 4.5, 4.6, 4.7 und 4.8) auf die Prozessoren im SPMD-Modell (Abschnitt 4.4) verteilt und wiederholt die problemspezifischen Funktionen aufruft, um die Problemistanzen zu teilen, die Basisfälle zu lösen, und die Ergebnisse zusammenzusetzen. Die problemspezifischen Funktionen in der funktionalen Form werden an die Funktion höherer Ordnung als Parameter übergeben werden. Im C-Programm sind es Funktionen, deren Körper abhängig von der Probleminstanz unterschiedlich implementiert werden kann. Die Aufrufstelle der problemspezifischen Funktionen ändert sich nicht. Die Unabhängigkeit der Implementierung der problemspezifischen Funktionen von der Skelett-Implementierung wird durch die Definition einer Schnittstelle zwischen Skelett-Implementierung und der Implementierung der problemspezifischen Funktionen erreicht.

Es können verschieden Algorithmen aus der Skelettklasse implementiert werden. Durch die Auswahl nur eines bestimmten Algorithmus wird das imperative Skelett instanziiert. Mit der Auswahl eines bestimmten Algorithmus wird auch ein bestimmter Problemteilungsgrad und eine bestimmte Probleminstanz festgelegt. Der Anwender muß selbst nur den Problemteilungsgrad des Algorithmus bestimmen, die Datentypen der Ein/Ausgabedaten definieren und die problemspezifischen Funktionen implementieren, darunter versteht man auch die Funktionen, die die Eingabedaten lesen und das Ergebnis ausgeben.

Es gibt zwei Implementierungsmöglichkeiten der Funktion zur Lösung des Basisfalles, entweder die Implementierung der Basisfunktion aus der funktionalen Spezifikation oder die Implementierung des besten sequentiellen Algorithmus. Für das Zusammenspiel der beiden Implementierungen gelten folgende Konventionen:

- Die Skelett-Implementierung heißt `sk_mesh.c` (unwichtig für den Anwender).

- Die Implementierung der problemspezifischen Funktionen heißt `a'`*k*`'_'`*name*`'.c`, wobei das '*k*' für den Problemteilungsgrad des Algorithmus steht.

  **Beispiel 5.1.1**

  Die Implementierung der problemspezifischen Funktionen für das Polynomprodukt heißen `a3_poly.c` und `a4_poly.c`.

- Die Datei `sk.h` bildet die Schnittstellendefinition zwischen Skelett-Implementierung und der Implementierung der problemspezifischen Funktionen. Sie wird von den Dateien `sk_mesh.c` und `a'`*k*`'_'`*name*`'.c` eingebunden.

- Das Verbinden der Skelett- und der Implementierung der problemspezifischen Funktionen, d.h. nach der Auswahl eines bestimmten Algorithmus durch den Anwender erfolgt die Übersetzung, die Übergabe der Programmparameter und die Ausführung auf dem Transputernetz automatisch und wird in der Experimentierumgebung (`host.c`) implementiert.

  **Beispiel 5.1.2**

  Mit `host` kann man auf einem Hostrechner im Verzeichnis ~/`bin` die Datei `host.c` compilieren. Die Compilation erzeugt eine ausführbare Datei `pdc`. Mit dem Aufruf von `pdc` beginnt die Arbeit mit der Experimentierumgebung.

Die Implementierung besteht also generell aus drei Teilen, aus der Implementierung des Skeletts, der problemspezifischen Funktionen und der Experimentierumgebung. Die Abhängigkeit der einzelnen Dateien wird in 5.2 deutlich gemacht. Es gibt zwei Betriebsarten:

- Normalbetrieb: Die Eingabedaten werden von einer Datei gelesen und auf eine Datei ausgegeben. Da das Betriebssystem des Transputernetzes gleichzeitig nur eine beschränkte Anzahl von Dateien geöffnet halten kann und die Ausgabe von allen Prozessoren auf eine Datei nicht gepuffert wird (d.h. die Ausgabe wird verzahnt), erfolgt die Ausgabe auf die Standardausgabedatei. Es besteht die Möglichkeit die Ausführungszeiten zu messen.

  **Beispiel 5.1.3 (Syntax der Eingabedatei)**

  Die erste Zeile enthält die Zweierpotenz der Länge der Eingabedaten $n$. Danach folgen $2^n$ Daten vom Typ des Eingabeproblems.
  $n$:
  $\{_1\{\textit{Zeichenkette} \sqcup \}^{2^n}\}^+$
  wobei: $2^n$ die Eingabelänge ist.

- Experimentbetrieb: Die Eingabedaten werden von dem Anwender erzeugt. Es werden Zeitmessungen für die einzelnen Berechnungsphasen vorgenommen und auf die Zeitdatei (`result`) ausgegeben und von der Experimentierumgebung ausgewertet. Es erscheint kein Ergebnis der Berechnung.

Die Plazierungsfunktion wird auch von einer Datei gelesen. Damit können durch Eingabe der realen Prozessornummern in dieser Datei unterschiedliche Plazierungsfunktionen ausprobiert werden. Die Kommunikation zwischen der Experimentierumgebung und dem SPMD Programm findet über die `control`-Datei und die Ausgabedateien statt.

Die Abb. 5.3 stellt den Zusammenhang der Programmteile dar.

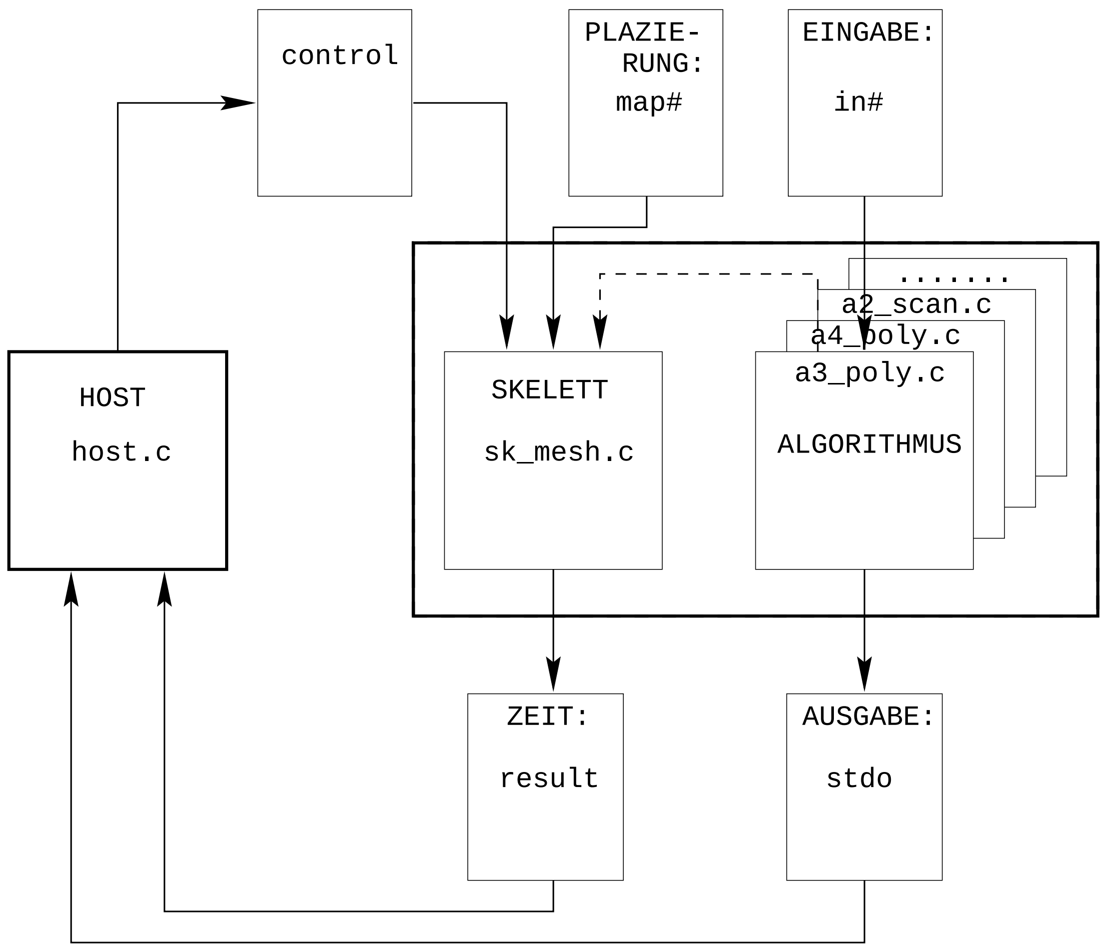



Abbildung 5.2: Die Abhängigkeit der Dateien

## 5.2 Sicht des Anwenders

Die Sicht des Anwenders (Abb. 5.1 oben rechts) beschränkt sich auf

- die Bestimmung des Problemteilungsgrades,
- die Bestimmung der Typen der Ein/Ausgabedaten und
- die Implementierung der problemspezifischen Funktionen.

### 5.2.1 Die Typen der Ein/Ausgabedaten und der Problemteilungsgrad

In der Datei `a'`*k*`'_'`*name*`'.h` stehen die Definitionen der problemabhängigen benutzerdefinierten Typen (`INTYPE` , `OUTTYPE`) und des Problemteilungsgrades. Das

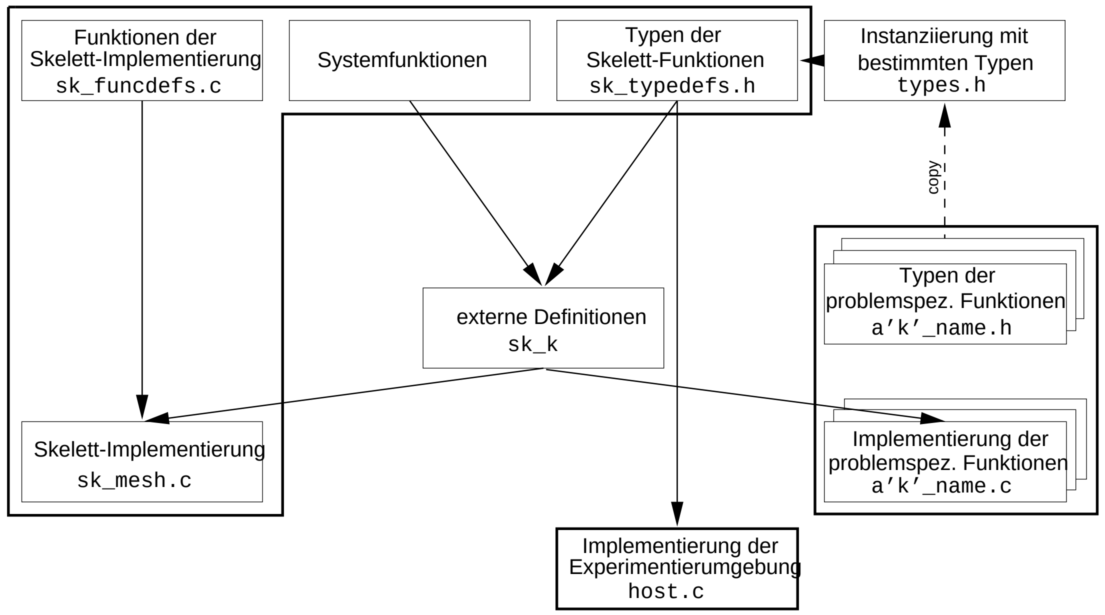



Abbildung 5.3: Zusammenhang der Dateien

Skelett sieht nur eine aktuelle Kopie einer Datei `a'`*k*`'_'`*name*`'.h`. Beim Polynomprodukt besteht der Eingabevektor aus Paaren, von denen jeweils das erste Element zum einen Operanden und das andere zum anderen Operanden gehört. Der Ergebnisvektor besteht auch aus Paaren, ein Element für den linken Teil und ein Element für den rechten Teil des ursprünglichen Ergebnisvektors.

**Beispiel 5.2.1 (Die `a3_poly.h`-Datei für das Polynomprodukt)**

```
#define degree 3                           /* Problemteilungsgrad */

typedef struct {
           long a;       /* Der erste Operand des Polynomprodukts  */
           long b;       /* Der zweite Operand des Polynomprodukts */
} INTYPE;

typedef struct {
           long l;          /* Der untere Teil des Polynomprodukts */
           long h;          /* Der obere Teil des Polynomprodukts  */
} OUTTYPE;
```

### 5.2.2 Problemspezifische Funktionen

Unter den problemspezifischen Funktionen verstehen wir fünf Funktionen, die von dem Skelett aufgerufen werden. Sie ermöglichen die Implementierung der einzelnen DC-Algorithmen, die die funktionale Spezifikation erfüllen. Um die Syntax des C-Programms der funktionalen Spezifikation anzunähern werden speziellen Makros definiert:

*makroname(parameterliste)*

wobei:

| | | |
|---|---|---|
| *makroname* | ::= | `INPUT` \| `OUTPUT` \| `DIVIDE` \| `COMBINE` \| `BASIC` \| `SEQUENTIAL` |
| *parameterliste* | ::= | *parameter* \| *parameter* ; *parameterliste* |
| *parameter* | ::= | C-Anweisung |

Überall dort, wo ein Makroname steht, wird eine Funktionsdefinition hineinkopiert, deren Körper die Makroparametern bilden. Es besteht auch die Möglichkeit, die Funktionen aus `design.c` zu verwenden, die in `sk.h` als `extern` deklariert sind.

### input()

*Funktionalität*: input: $\alpha$
*C-Funktion* : `void input(FILE *fin_file, INFO fparam, INVECTOR *fin_vec)`
*Makro* : INPUT(*aktionen*)

*input*() initialisiert den Eingabevektor (`IN_V`). Es gibt zwei Möglichkeiten abhängig davon in welchem Modus gearbeitet wird: für Experimente (`case 0:`) wird der Eingabevektor direkt geschrieben, sonst (`case 1:`) wird von einer Datei (`SCAN_IN` weiß welcher) gelesen. Makros innerhalb von `INPUT()`:

- `IN_ITER(`*body*`)` iteriert den Eingabevektor `IN_V`, seine Länge ist `INITERLEN`. Der `body` darf sinnvolle C-Anweisungen enthalten und Makros: `IN_V`, `SCAN_IN()`. `SCAN_IN()` ist *fscanf*() C-Funktion, die zusätzlich den richtigen Dateideskriptor bekommt und sich auf die Elemente von `IN_V` bezieht.

- `OVERREAD(n)` überliest `n` Elemente von der Eingabedatei.

- `IN_DIST, IN_CENT` sind Prädikaten, die über die Art der Eingabe bestimmen.

### output()

*Funktionalität*: output: $\beta$
*C-Funktion* : `void output(FILE *fout_file, INFO fparam, OUTVECTOR *fout_vec)`
*Makro* : OUTPUT(*aktionen*)

*output*() schreibt den Ausgabevektor `OUT_V` auf die Datei `OUT_FILE`. Makros innerhalb von `OUTPUT()`:

- `OUT_ITER(body)` iteriert den Ausgabevektor `OUT_V`, der `body` darf `PRINT_OUT()` enthalten. `PRINT_OUT()` ist *fprintf*() C-Funktion, die zusätzlich den richtigen Dateideskriptor bekommt und sich auf die Elemente von `OUT_V` bezieht.

### basic()

*Funktionalität*: basic: $\alpha \longrightarrow \beta$
*C-Funktion* : `void basic( INTYPE fin_elem, OUTTYPE *fout_elem )`
*Makro* : BASIC(*aktionen*)

*basic*() bildet ein Element einer Probleminstanz auf ein Element einer Lösungsinstanz ab. Auf den Parameter *fin_elem* kann man mit dem Makro `BDIV` (eine Variable vom Typ `INTYPE`) zugreifen, der andere Parameter der Funktion ist ein Zeiger auf ein *dummy* -Element, in dem das Ergebnis abgespeichert wird. Auf das *dummy* -Element kann mit `BCOM` (eine Variable vom Typ `OUTTYPE`) zugegriffen werden.
Beispiel der Anwendung: `BASIC( BCOM = BDIV; )`; andere Beispiele im Anhang.

### sequential()

*Funktionalität*: sequential: $\alpha \longrightarrow \beta$
*C-Funktion* : `void sequential( INTYPE *fin_elem, OUTTYPE *fout_elem )`
*Makro* : SEQUENTIAL(*aktionen*)

Die Implementierung der Funktion *sequential*() wird dem Programmierer überlassen.

### divide()

*Funktionalität*: divide:$(\mathbb{N},\ \alpha) \rightarrow \alpha^k$
*C-Funktion* : `void divide( long branch, INVECTOR *fin_vec )`
*Makro* : DIVIDE(case *branch: aktionen*; break;)

*divide* bildet einen Vektor einer Probleminstanz auf einen Vektor einer Teilprobleminstanz ab. Die Funktion liefert $k$ verschiedene Ergebnisse, in Abhängigkeit vom Parameter *branch*, zurück. Der Parameter *branch* bezieht sich auf eine von $0..k-1$ Positionen im Gitter, die Implementierung muß alle Fälle berücksichtigen. Die Parameter des `DIVIDE`-Makros werden in die *divide*()-Funktion als Körper von switch(*branch*) hineinkopiert, sie dürfen also das alles enthalten was in der switch()-Anweisung vorkommen darf.
Makros innerhalb von `DIVIDE()`:

- `DIV_ITER(body)` iteriert einen *dummy*-Vektor, der für das Ergebnis der *divide()*-Funktion vorgesehen ist. Es werden automatisch zwei Variablen deklariert: `I_VAR` und `I_LEN`, deren Namen, um Namenskonflikte zu vermeiden, in den problemspezifischen Funktionen nicht neu deklariert werden dürfen. `I_LEN` wird mit der Länge des iterierten Vektors initialisiert, `I_VAR` ist eine Iterationsvariable einer for-Schleife. *body* darf durch Kommata getrennte C-Anweisungen enthalten, insbesondere Zuweisungen mit `DIV(n)` und `UNDIV(n)`:
  - `DIV(n)` greift auf das Element des $n$-ten Teils des Ergebnisvektors mit dem Index (`I_VAR`) zu,
  - `UNDIV(n)` greift auf das Element des $n$-ten Teils des Argumentvektors mit dem Index (`I_VAR`) zu,
- `COPY_L` kopiert den linken Teil des Argumentvektors in den Ergebnisvektor,
- `COPY_R` kopiert den rechten Teil des Argumentvektors in den Ergebnisvektor.

**Beispiel 5.2.2 (*divide* für das Polynomprodukt mit 3-Teilung)**

Die funktionale Spezifikation:

$$divide\ (xh, yh)\ (xl, yl) = [(xh, yh), (xh + xl, yh + yl), (xl, yl)]$$

Die Implementierung:

```
DIVIDE( case 0: COPY_L;                                       break;
        case 1: DIV_ITER( (DIV(0).a = UNDIV(0).a + UNDIV(1).a,
                           DIV(0).b = UNDIV(0).b + UNDIV(1).b) ); break;
        case 2: COPY_R;                                       break; )
```

### combine()

*Funktionalität*: combine: $(\mathbb{N},\ \beta^k) \rightarrow \beta$
*C-Funktion* : `void combine(long partition, OUTVECTOR *fout_vec)`
*Makro* : COMBINE(case *partition: aktionen*; break;)

*combine*() bildet einen Vektor der Teillösungsinstanzen auf einen Vektor einer Lösungsinstanz ab. Die Funktion liefert 3 verschiedene Ergebnisse, in Abhängigkeit von der Art der Ausgabe (zentralisiert/verteilt) zurück. Für die zentralisierte Ausgabe wird ein Ergebnisvektor zurückgeliefert, der aus $k$-Teilvektoren zusammengesetzt wird. Für die verteilte Ausgabe wird der linke Teil (`LPOS`) und der rechte Teil (`RPOS`) des Ergebnisses zurückgeliefert. Die Parameter des `COMBINE`-Makros werden in die *combine*()-Funktion als Körper von switch(*partition*) hineinkopiert, sie dürfen also das alles enthalten was in der switch()-Anweisung vorkommen darf.
Makros innerhalb von `COMBINE()`:

- `COM_CENT_ITER(body)` iteriert einen *dummy*-Vektor, der für das Ergebnis der *combine()*-Funktion vorgesehen ist. Es werden automatisch die Variablen `I_VAR` und `I_LEN` deklariert. `I_LEN` wird mit der Hälfte der Länge des iterierten Vektors initialisiert, `I_VAR` ist eine Iterationsvariable einer for-Schleife. *body* darf durch Komata getrennte C-Anweisungen enthalten, insbesondere Zuweisungen mit `COM(n)` und `UNCOM(n)`:
  - `COM(n)` greift auf das Element des $n$-ten Teils des Ergebnisvektors mit dem Index (`I_VAR`) zu,
  - `UNCOM(n)` greift auf das Element des $n$-ten Teils des Argumentvektors mit dem Index (`I_VAR`) zu,
- `COM_DIST_ITER(body)` analog zu `COM_CENT_ITER(body)` mit dem Unterschied, daß `I_LEN` mit der Länge des iterierten Vektors initialisiert wird,
- Konstanten: `CENT, LPOS, RPOS` für den Parameter *partition*

## 5.3 Sicht der Parallelisierung

Das Skelett (Abb. 5.1 oben links) nimmt die Verwaltung des Transputernetzes vor, d.h. das Skelett muß die virtuelle Topologie aus Abschnitt 4.1 aufbauen, die problemspezifische Funktionen aufrufen und die parallelen Komponenten der Eingabedaten auf die Prozessoren verteilen.

### 5.3.1 Datenstruktur

Statt Powerlisten aus Abschnitt 3.2.3 definieren wir *pseudopolymorphe* Vektoren, deren Länge eine Zweierpotenz ist. Sie sind pseudopolymorph, weil die *Polymorphie* dadurch erreicht wird, daß die Elemente der Vektoren einen Zeiger auf einen Elementtyp enthalten. Der Typ der Vektoren ist problemunabhängig. Der Elementtyp kann vom Anwender problemabhängig definiert werden.

**Beispiel 5.3.1 (Typen für das Skelett)**

- `INVECTOR` ist eine Struktur mit vier Komponenten:
  - *undivided* repräsentiert den ungeteilten Vektor (Argument der *divide*-Funktion),
  - *undivsize* seine Länge und
  - *divided* repräsentiert den geteilten Vektor (Ergebnis der *divide*- Funktion),
  - *divsize* seine Länge.

- `OUTVECTOR` ist eine Struktur mit vier Komponenten:
  - *uncombined* repräsentiert den noch nicht zusammengefügten Vektor (Argument der *combine*-Funktion),
  - *uncomsize* seine Länge und
  - *combined* repräsentiert den zusammengefügten Vektor (Ergebnis der *combine*-Funktion),
  - *comsize* seine Länge.

Ausführliche Datentyp- und Makrodefinitionen befinden sich in der Datei `sk_typedefs.h` (siehe Anhang).

### 5.3.2 Aufbau der Skelettdatei

Die Steuerung des Skeletts erfolgt über die `control`-Datei, die automatisch nach einer Interaktion mit dem Benutzer erzeugt wird. Es hat erhebliche Probleme bei der Ausgabe der Kommunikations- und Berechnungszeiten gegeben. Jeder Prozessor sollte die Zeiten auf seine eigene Ausgabedatei schreiben. Das Betriebssystem kann aber auf einmal nicht so viele Dateien verwalten. Schreiben der Zeiten auf eine gemeinsame Datei oder Umleiten der `stdout`-Datei kam auch nicht in Frage, da die Ausgabe nicht gepuffert wird. Wir haben die Ausgabe von allen Prozessoren in verzahnter Form bekommen, aus der man keine Information gewinnen konnte. Wir haben das Ausgabeproblem gelöst, indem jeder Prozessor die Daten in einen String abspeichert und dann alle Prozessoren nacheinander die Daten an die `result`-Datei anhängen.

Die Implementierung des Skeletts befindet sich in der Datei `sk_mesh.c` (siehe Anhang), die folgenden Aufgaben übernimmt:

- Lesen der Programmparameter von der `control`-Datei.
- Abfrage des Problemteilungsgrades; `degree` ist ein in der problemabhängigen Typdatei definierter Makro.
- Berechnung der Anzahl der Prozessoren, die an der Berechnung beteiligt sind $k^d$.
- Zum Aufbau der Verbindungsstruktur zwischen den Prozessoren des Transputernetzes wird eine Prozessortabelle gehalten, wo die Zuordnung `VPID` zu `PID` gespeichert wird und eine Linktabelle, wo die Verbindungen (*Links*) in jeder Dimension für den linken und rechten Nachbarn gespeichert werden. Zusätzlich gibt es eine Dimension für die Speicherung der Links einer Kette.

- Plazierung: Aufruf der `mapping()` -Funktion mit der Nummer der `map#`-Datei als Parameter. Die `map#`-Datei muß bereits existieren und eine Reihenfolge von `PID`s enthalten. Die `mapping()`- Funktion initialisiert die Prozessortabelle `p[0,...,nProcs-1]` gemäß der `map#`-Datei.
- Setzen des `VPID`s: jeder Prozessor bekommt sein `VPID` durch den Aufruf von `find_vpid()`
- Aufbau der Gitter-Struktur gemäß Kap. 4.1: Aufruf von `make_mesh()`
- Aufbau der Ketten-Struktur: Aufruf von `make_chain()`
- Selektion der Prozessoren, die an der Berechnung beteiligt sind, d.h. die mit `VPID` $\in \{0, ..., k^d - 1\}$
  - Öffnen der entsprechenden `input`-Datei (`in#`) aus der später im Normalbetrieb die Eingabedaten gelesen werden.
  - Partitionierung der Gitter-Struktur für die verteilten Eingabedaten. Jede Partition liest nur den entsprechenden Teil des Eingabevektors: Aufruf von `partition()` (siehe auch S. 29).
  - Speicherplatzreservierung für den Ausgabestring. Wir brauchen Einträge für die Zeiten der `input(), basic()` und `output()` Funktionen und für die Kommunikationszeiten der *divide*, *combine* Phasen als auch die Zeiten der `divide(), combine()` Funktionen (insgesamt $4*d+3$ Zeiten). `STRLEN` ist ein Makro für die Anzahl der Zeichen einer Ausgabe.
  - Öffnen einer Ausgabedatei für das Berechnungsergebnis.
  - Eine `do-while`-Schleife mit `sk_input(), sk_divide(), sk_basic(), sk_combine(), sk_output()`, die so oft ausgeführt wird, wieviele Experimente durchzuführen sind. Das hat den Vorteil, das die reservierte Partition erst nach einer ganzen Experimentreihe freigegeben wird.
  - Synchrones Schreiben auf eine Ausgabedatei: Aufruf von `write_sync()`.
  - Erhöhen des Experimentzählers.
- Schliessen der Ein- und Ausgabedatei.
- Abbau der Gitter-Struktur: Aufruf von `free_mesh()`.
- Abbau der Ketten-Struktur: Aufruf von `free_chain()`.
- Speicherplatzfreigabe.

Alle von `sk_mesh()` benutzte Funktionen sind in `sk_funcdefs.c` und `design.c` definiert.

### Inhalt der sk_funcdefs.c -Datei:

Alle Funktionen, die die Arbeit des Skellets steuern scheinen nicht so wichtig zu sein (siehe Anhang). Es folgen die wichtigsten Funktionen, die die DC-Phasen Abbilden:

#### sk_divide():

In `sk_divide()` wird das absteigende Kommunikationsmuster implementiert.
`sk_divide()` ist eine Funktion, die die parallelen Komponenten des Eingabevektors auf die Prozessoren verteilt und die `divide()`-Funktion aufruft. Diese Funktion entspricht der ersten vernesteten for-Schleife im Schleifenprogramm (Abschnitt 4.2) und simuliert die Rekursion im funktionalen Programm. Als Parameter bekommt `sk_divide()` einen Zeiger auf eine Struktur `INVECTOR`, die den Ein- und Ausgabevektor der Funktion `divide()` repräsentiert und eine Variable `param` vom Typ `INFO`, die die Art der Eingabe, Anzahl der Teilungsschritte, Teilungsgrad des Problems, virtuelle Prozessornummer beinhaltet. Der an die Funktion übergebene Vektor ist in der *divided*-Komponente von `INVECTOR` abgelegt (Makro `DIV_V`). Es muß noch Speicherplatz für die *undivided*-Komponente von `INVECTOR` (Makro `UNDIV_V`) reserviert und seine Länge in Abhängigkeit von der Art der Eingabe bestimmt werden. Der Grund dafür, warum wir zwei Vektoren brauchen, ist die `divide()`-Funktion, die einen Argumentvektor aufnimmt und das Ergebnis in einen Ergebnisvektor schreibt. Der Ergebnisvektor wird im nächsten Teilungsschritt als Argumentvektor an `divide()` übergeben und umgekehrt steht der Argumentvektor frei für das Ergebnis. Mit den Makros `LEFTPOS(param.k)`, `RIGHTPOS(param.k)` wird die Position bestimmt, an der sich die gültigen Daten befinden und in der Variablen *leftpos, rightpos* gespeichert. Die variable *offset* enthält den Abstand zwischen zwei Nachbarprozessoren entlang der Dimension $d$ ausgedrückt in *vpid*. Der Prozessorabstand ist wichtig um die aktuelle Position des Prozessors im Gitter entlang der Dimension $d$ zu bestimmen (Makro `KOORDINATE(param.vpid, offset, param.k)`). Der wichtigste Teil der `sk_divide()`-Funktion ist die for-Schleife, deren Iterationsvariable $d$ die aktuell behandelnde Dimension repräsentiert. Der Körper der for-Schleife implementiert die Verfeinerungsschritte der Kommunikation, da der Übergang zwischen zwei Dimensionen nicht mit einem Kommunikationsschritt erreicht werden kann. Im Fall der zentralisierten Eingabe wird der zu verteilende Vektor in jedem Teilungsschritt halbiert, deshalb muß seine Länge korrigiert werden. Der Speicherplatz wird aus Effizienzgründen erst zum Schluß freigegeben, es hat keine Auswirkungen auf den Speicherbedarf, da wir nur so viel Speicherplatz reserviert haben, wie nötig war. Die Variablen *time, timecom, timediv* dienen zur Speicherung der Zeitmessungen. In Abhängigkeit von der Art der Eingabe und der Position des Prozessors im Gitter sind unterschiedliche Aktionen vorzunehmen.
a) Zentralisierte Eingabedaten (Abb. 5.4):

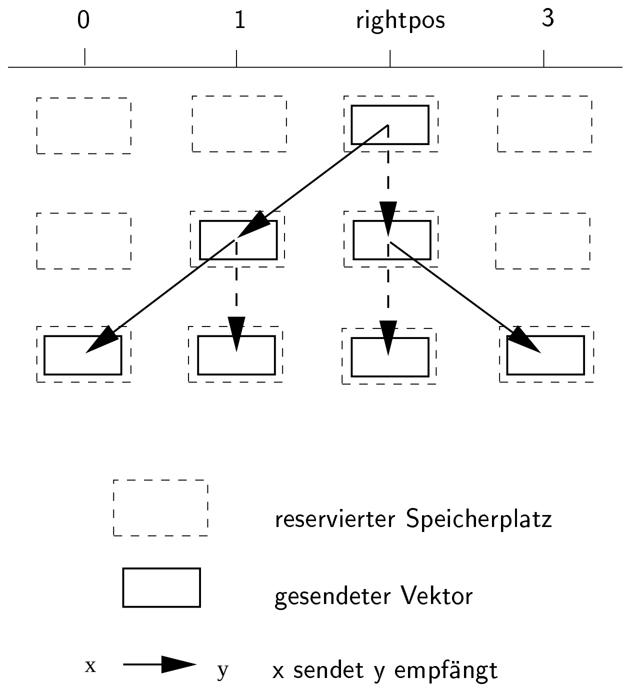



Abbildung 5.4: Zentralisierte Datenverteilung entlang der Dimension $d$ ($k$=4)

Am Anfang wird `DIV_V` (geteilter Vektor) mit `UNDIV_V` (ungeteilter Vektor) vertauscht und dann:

- rightpos: Der *undivided*-Vektor wird zum linken Nachbarn und falls sich der Prozessor nicht am Rand des Gitters befindet (d.h. $x_i \neq k-1$) zum rechten Nachbarn gesendet,

- *links von* rightpos : Es wird von rechts in den *undivided*-Vektor empfangen und falls die Position nicht am linken Rand ($x_i \neq 0$) ist wird der *undivided*-Vektor zum linken Nachbarn gesendet,

- *rechts von* rightpos: Es wird von links in den *undivided*-Vektor empfangen und falls die Position nicht am rechten Rand ($x_i \neq k-1$) ist wird der *undivided*-Vektor zum rechten Nachbarn gesendet.

b) Verteilte Eingabedaten (Abb. 5.5):

- leftpos: Sende den *divided*-Vektor nach rechts kopiere *divided*-Vektor in den linken Teil vom *undivided*-Vektor, empfange von rechts in den rechten Teil vom *undivided*-Vektor, sende den *undivided*-Vektor nach links falls noch nicht am linken Rand,

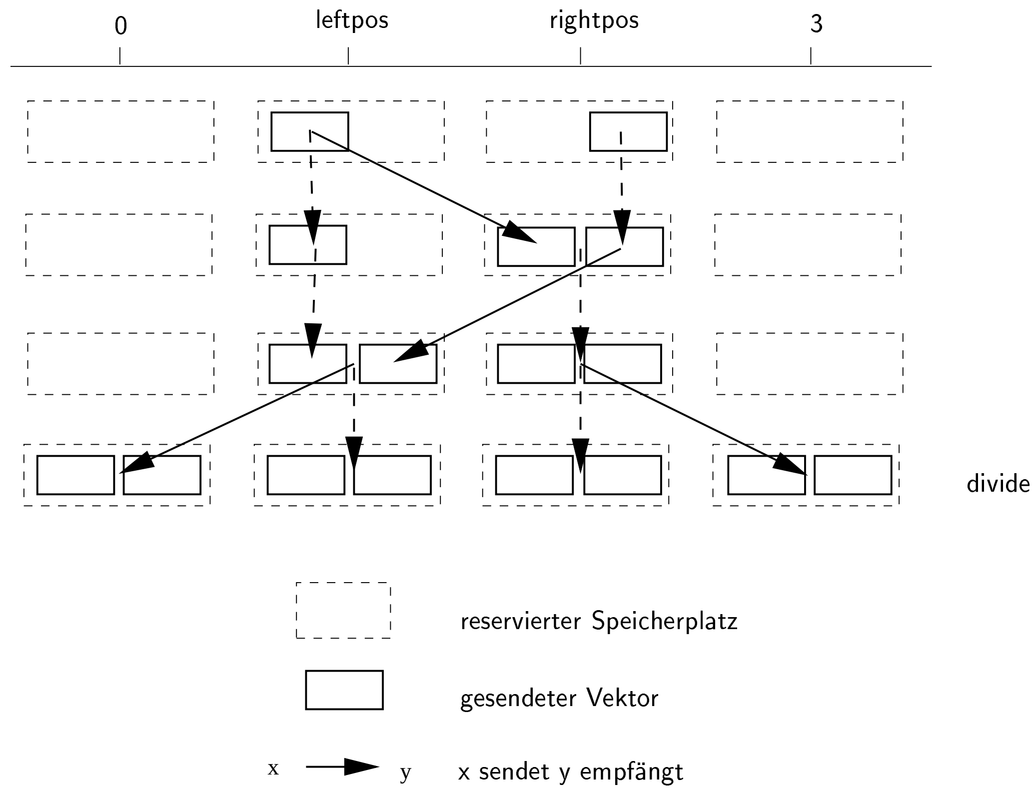



Abbildung 5.5: Verteilte Datenverteilung entlang der Dimension $d$ ($k = 4$)

- rightpos: Empfange von links in den linken Teil vom *undivided*-Vektor, kopiere *divided*-Vektor in rechten Teil vom *undivided*-Vektor, sende *divided*-Vektor nach links, sende den *undivided*-Vektor nach rechts falls nicht am rechten Rand,
- *links von* leftpos : Es wird von rechts in den *undivided*-Vektor empfangen und falls die Position nicht am linken Rand ( $x_i \neq 0$ ) *undivided*-Vektor nach links gesendet,
- *rechts von* rightpos: Es wird von links in den *undivided*-Vektor empfangen und falls die Position nicht am rechten Rand ( $x_i \neq k-1$ ) *undivided*-Vektor nach rechts gesendet.

Nach diesen Schritten liegt der zu teilende Vektor auf den Prozessoren, die sich in der $d$-ten Koordinate unterscheiden und `divide()` wird aufgerufen.

**sk_combine():**

In `sk_combine()` wird das aufsteigende Kommunikationsmuster implementiert. `sk_combine()` ist eine Funktion, die die parallelen Komponenten des Ausgabe-

vektors zu einem Ergebnisvektor zusammensetzt und die `combine()`-Funktion aufruft. Diese Funktion entspricht der dritten vernesteten for-Schleife im Schleifenprogramm 4.2 und simuliert die Rückwärtsrekursion im funktionalen Programm. Als Parameter bekommt `sk_combine()` einen Zeiger auf eine Struktur `OUTVECTOR`, die den Ein- und Ausgabevektor der Funktion `combine()` repräsentiert und eine Variable param vom Typ `INFO`, die die Art der Ausgabe, Anzahl der Teilungsschritte, Teilungsgrad des Problems, virtuelle Prozessornummer beinhaltet. Der an die Funktion übergebene Vektor ist in der *combined*-Komponente von `OUTVECTOR` abgelegt (Makro `COM_V`). Es muß noch Speicherplatz für die *uncombined*-Komponente von `OUTVECTOR` (Makro `UNCOM_V`) reserviert werden. Seine Länge ist gleich der Länge des *combined*-Vektors. Der Grund dafür, warum wir zwei Vektoren brauchen ist die `combine()`-Funktion, die analog der `divide()`- Funktion einen Argument-Vektor aufnimmt und das Ergebnis in einen Ergebnis-Vektor schreibt. Der Ergebnis-Vektor wird im nächsten Zusammensetzungsschritt als Argument-Vektor an `combine()` übergeben und umgekehrt steht der Argument-Vektor frei für das Ergebnis. Mit den Makros `LEFTPOS(param.k)`, `RIGHTPOS(param.k)` wird die Position bestimmt, an der sich die gültigen Daten befinden und in die Variablen *leftpos, rightpos* gespeichert. Die variable *offset* enthält den Abstand zwischen zwei Nachbarprozessoren entlang der Dimension $d$ ausgedrückt in *vpid*. Der Prozessorabstand ist wichtig um die aktuelle Position des Prozessors im Gitter entlang der Dimension $d$ zu bestimmen (Makro `COORDINATE(param.vpid, offset, param.k)`). Der wichtigste Teil der `sk_combine()`-Funktion ist die for-Schleife, deren Iterationsvariable $d$ die aktuell behandelnde Dimension repräsentiert. Der Körper der for-Schleife implementiert die Verfeinerungsschritte der Kommunikation, da der Übergang zwischen zwei Dimensionen nicht mit einem Kommunikationsschritt erreicht werden kann. Im Fall der zentralisierten Ausgabe wird der zusammenzusetzende Vektor in jedem Teilungsschritt verdoppelt, deshalb muß seine Länge korrigiert werden. Die Variablen *time, timecom, timediv* dienen zur Speicherung der Zeitmessungen. Nach der Koordinatenbestimmung wird der *combined*-Vektor in den Teil des *uncombined*-Vektors kopiert, der der Gitter-Koordinate entspricht. In Abhängigkeit von der Art der Ausgabe und der Position des Prozessors im Gitter sind unterschiedliche Aktionen vorzunehmen.
a) Zentralisierte Ausgabedaten (Abb. 5.6):

- *links von* rightpos: Falls am linken Rand sende *combined*-Vektor nach rechts, sonst empfange von links in den *uncombined*-Vektor (darunter alle *combined*-Vektoren von Prozessoren die links stehen), sende den *uncombined*-Vektor nach rechts,

- *rechts von* rightpos: Falls am rechten Rand sende den *combined*-Vektor nach links, sonst empfange von rechts in den *uncombined*-Vektor (darunter alle *combined*-Vektoren von Prozessoren die rechts stehen), sende den *uncombined*-Vektor nach links,

- rightpos: Empfange von rechts in den *uncombined*-Vektor, empfange von links in den *uncombined*-Vektor.

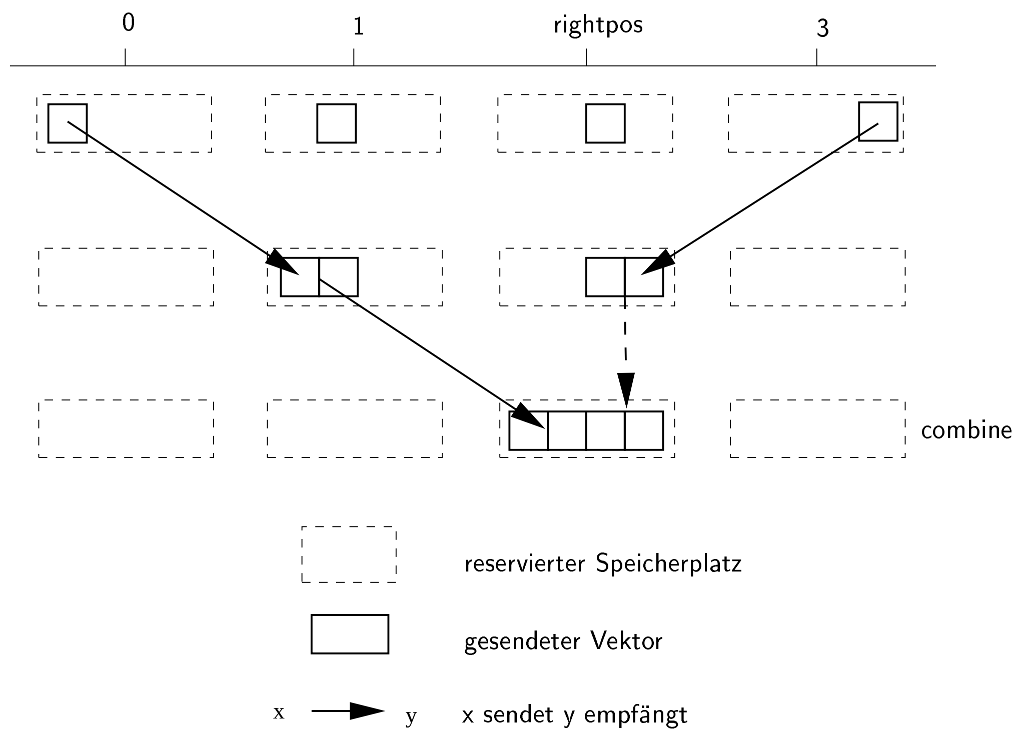



Abbildung 5.6: Zentralisierte Datenzusammensetzung entlang der Dimension $d$ ($k$=4)

b) Verteilte Ausgabedaten (Abb. 5.7):

- *links von* leftpos: Falls am linken Rand sende den entsprechenden Teil des *uncombined*-Vektors nach rechts, sonst empfange von links in den linken Teil des *uncombined*-Vektors, sende den entsprechenden Teil des *uncombined*-Vektors nach rechts,
- *rechts von* rightpos: Falls am rechten Rand sende den entsprechenden Teil des *uncombined*-Vektors nach links, sonst empfange von rechts in den rechten Teil des *uncombined*-Vektors, sende den entsprechenden Teil des *uncombined*-Vektor nach links,
- leftpos: Falls nicht am linken Rand empfange von links in den *uncombined*-Vektor, sende den *uncombined*-Vektor nach rechts, empfange von rechts in den rechten Teil des *uncombined*-Vektors,

- rightpos: Empfange von links in den linken Teil des *uncombined*-Vektors, sende den rechten Teil des *uncombined*-Vektors nach links.

Nach diesen Schritten liegen $k$-Teilvektoren auf den Prozessoren, die in der $d$-ten Koordinate rightpos (im zentralisierten Fall) oder lefttpos/rightpos (im verteilten Fall) enthalten. Die Teilvektoren werden durch den Aufruf von `combine()` zu einem Vektor zusammengesetzt.

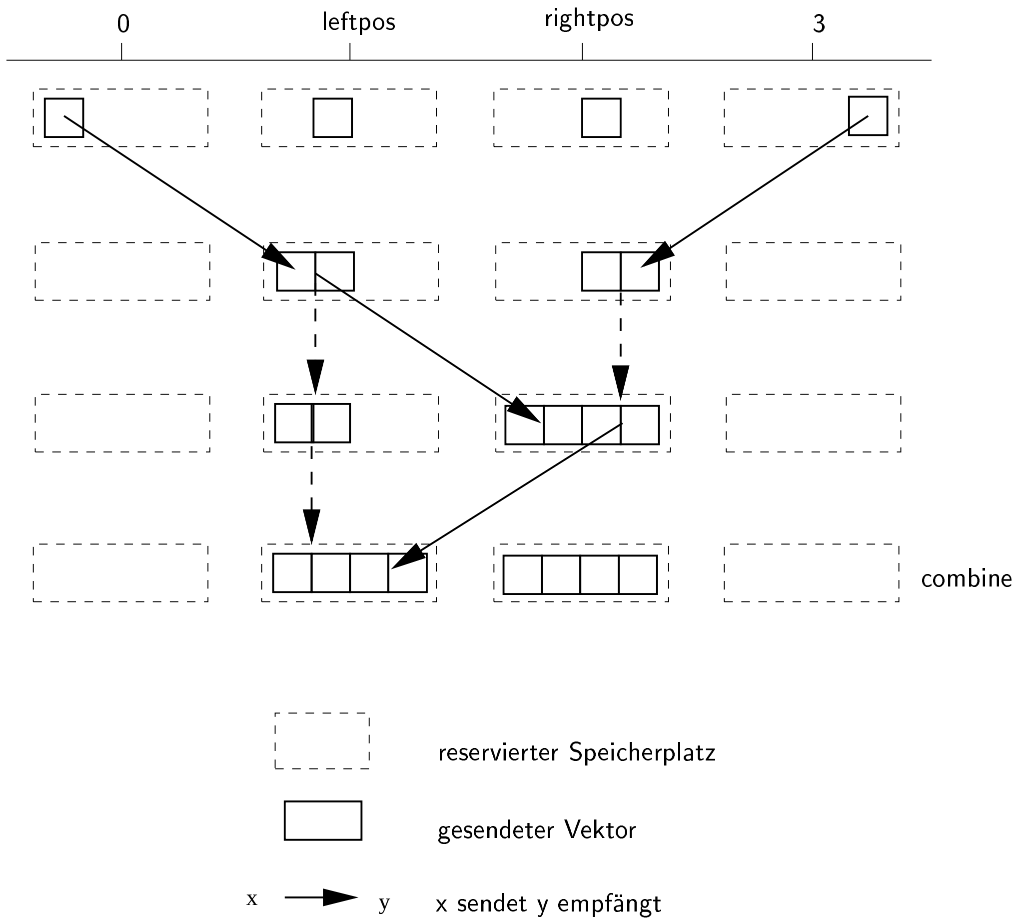



Abbildung 5.7: Verteilte Datenzusammensetzung entlang der Dimension $d$ ($k$=4)

### sk_basic():

In `sk_basic()` wird die Idee mit virtuellen Prozessoren implementiert (Abb. 5.8), sie entspricht der zweiten vernesteten for-Schleife im Schleifenprogram 4.2. Die `sk_basic()`-Funktion besteht aus drei Teilen für drei Bearbeitungsphasen: *divide*, *basic* und *combine*. Die Bearbeitung der drei Phasen unterscheidet sich von den

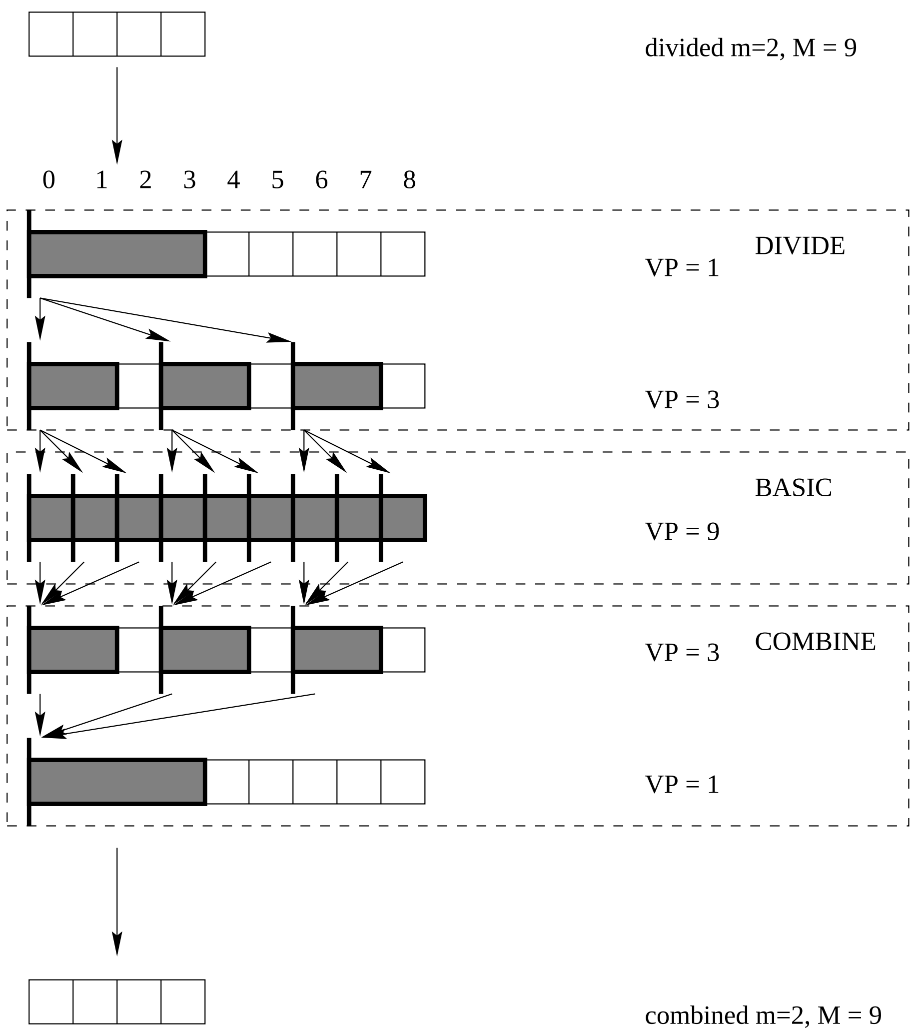



Abbildung 5.8: Virtuelle Prozessoren

entsprechenden parallelen Phasen damit, daß die Probleme nicht auf die Prozessoren verteilt werden, sondern virtuelle Prozessoren als Array-Elemente simuliert werden. Die Komunikation zwischen den virtuellen Prozessoren heißt lokal, da sie nur auf einem realen Prozessor als Speicherzugriffe (Zugriffe auf die Array-Elemente) stattfindet.

In der sequentiellen *divide*-Phase brauchen wir also zwei Arrays, deren Länge gleich der Anzahl der virtuellen Prozessoren ist und drei `for`-Schleifen. Die erste läuft über die Höhe des Berechnungsgraphen, die zweite über die Anzahl der Knoten auf einer bestimmten Ebene des Berechnungsgraphen, die dritte über die Anzahl der Söhne (Problemteilungsgrad). Auf jedem Prozessor bleibt ein Vektor gleicher Länge, der wegen des Prozessormangels nicht mehr geteilt werden kann und sequentiell bearbeitet werden muß. Wir wollen jedoch weiter die problemspezifischen Schnittstellenfunktionen in unveränderter Form anwenden. Aus diesem Grund betrachten wir die Speicherbereiche des Arrays abgegrenzt durch die Länge der Vektoren (im Code *Blöcke*) als virtuelle Prozessoren und führen die Teilung der Speicherbereiche durch den Aufruf der Funktion `divide()` durch, bis alle Vektoren die Länge 1 haben, d.h. bis jedes Array-Element einen virtuellen Prozessor repräsentiert. Ein Array steht immer für die Argumente der problemsspezifischen Funktionen und das andere für die Ergebnisse, nach der Ausführung von `divide()` werden die Vektoren vertauscht und die Berechnung fortgesetzt. Der Code unterscheidet sich von der Abb. 5.8 darin, daß es keine Löcher zwischen den Speicherbereichen gibt. Die Vektoren hängen aneinander. In der sequentiellen *basic*-Phase wird sequentiell für jeden virtuellen Prozessor die Funktion `basic()` ausgeführt. In der sequentiellen *combine*-Phase werden alle Schritte der *divide*-Phase rückwärts auf den Arrays vom anderen Typ ausgeführt. Wir brauchen eine for-Schleife und zwar die, die über die Söhne durchläft, weniger, da das Ergebnis in hintereinanderstehenden Blöcken ohne lokale Kommunikation zur Verfügung steht. Die Teilergebnisse der virtuellen Prozessoren werden zu einem Ergebnis durch Ausführen von `combine()` zusammengesetzt.

**sk_sequential():**

`sk_sequential()` bietet die Möglichkeit an, die sequentielle Komponente der Problem-Vektoren nicht mit virtuellen Prozessoren, sondern mit dem besten sequentiellen Algorithmus zu lösen. Der beste sequentielle Alorithmus wird in *sequential*() implementiert. Die Funktion `sk_sequential()` nimmt einen Vektor als Parameter auf, reserviert den Speicherplatz für den Ergebnisvektor von *sequential*(), ruft *sequential*() auf und nimmt die Zeitmessungen vor. Es gibt Algorithmen (wie z.B. scan), die schneller sequentiell ausgeführt werden können, als wenn man die Berechnung mit virtuellen Prozessoren simuliert. Das hat Auswirkungen auf die Effizienz, die besser ausfällt als theoretisch erwartet, deswegen, weil die Simulation langsamer ist.

# Kapitel 6

# Experimente

## 6.1 Übersicht

Die Leistungsverluste der parallelen Architekturen haben unter anderem folgende Quellen:

- Kommunikationsverluste entstehen durch die Überwindung von Distanzen beim Datenaustausch zwischen Systemkomponenten. Sie ergeben sich aus dem Aufwand, der notwendig ist, um Daten von einer Systemkomponente zu einer anderen zu transportieren, und der dadurch entstehenden Programmverzögerung.
- Zusatzaufwand, der bei der parallelen Abarbeitung einer Anwendung im Vergleich zur sequentiellen Abarbeitung entsteht. Das ist zum einen der Organisationsaufwand, wie das Initialisieren und Freigeben von Systemkomponenten. Zum anderen sind es zusätzliche Operationen, die im parallelen Fall für das Teilen von Eingabedaten, Zusamenfügen von Ergebnissen und Austauschen von Zwischenergebnissen entsteht.

Die Leistungsverluste hängen von vielen Faktoren ab. Mit der Durchführung verschiedener Experimente wollen wir herausfinden, welche Faktoren, wie starke Auswirkungen auf die Leistung haben.

### 6.1.1 Variationsmöglichkeiten für Durchführung der Experimente

Für die Durchführung der Experimente ergeben sich folgende Variationsmöglichkeiten:

- Problemteilungsgrad des Algorithmus (z.B. Scan: $k = 2$, Polynomprodukt: $k = 3$, $k = 4$),

- Problemgröße $N = 2^n$,
- Anzahl der Teilungsschritte $d$ (eingeschränkt durch die Anzahl der Prozesoren $p = k^d$),
- Art der Ein/Ausgabe:
  - zentralisiert,
  - verteilt,
- Berechnung des Basisfalles:
  - mit dem besten sequentiellen Algorithmus,
  - Simulation der Restberechnung mit virtuellen Prozessoren,
- Plazierungsfunktionen,
- Optimierungen durch Compress-and-Conquer.

### 6.1.2 Beobachtete Zeiten

Auf jedem Prozessor werden folgende Größen gemessen:

- Zeiten für die Initialisierung mit den Eingabedaten (Ausführungszeiten von *input*),
- Zeiten der Teilungsphase in jedem Teilungsschritt:
  - Kommunikation,
  - Teilung der Daten (Ausführungszeiten von *divide*),
- Zeiten der Basisphase (Ausführungszeiten von *basic*/*sequential*),
- Zeiten der Zusammensetzungsphase in jedem Zusammensetzungsschritt:
  - Kommunikation,
  - Zusammensetzung der Daten (Ausführungszeiten von *combine*),
- Zeiten für die Ausgabe der Ergebnisse (Ausführungszeiten von *output*),

### 6.1.3 Auswertung

Wir betrachten die Summe der Zeiten aus Abschnitt 6.1.2 ohne die Zeiten für die Initialisierung mit den Eingabedaten und die Ausgabe der Ergebnisse auf dem Prozessor, auf dem die Berechnung aufhört. Daraus ergibt sich:

- die parallelen Ausführungszeiten als Grundlage für die Effizienz und
- der Anteil der Kommunikation an der Gesamtberechnung.

## 6.2 Grundlegende Definitionen

Zwei wichtige Meßgrößen der Qualität von auf parallelen Architekturen implementierten parallelen Algorithmen sind *Beschleunigung* (*Speedup*) und *Effizienz* [Qui88].

**Definition 6.2.1 (Beschleunigung)**

Die von einem parallelen Algorithmus, der auf $p$ Prozessoren läuft, erzielte *Beschleunigung* $S_p$ ist das Verhältnis der Zeit $t_s$, die von einem Parallelrechner zur Ausführung des schnellsten sequentiellen Algorithmus benötigt wird, zu der Zeit $t_p$, die der gleiche Parallelrechner zur Ausführung des parallelen Algorithmus unter der Verwendung von $p$ Prozessoren benötigt:

$$S_p = \frac{t_s}{t_p} \tag{6.1}$$

**Definition 6.2.2 (Effizienz)**

Die *Effizienz* $E_p$ eines auf $p$ Prozessoren laufenden Algorithmus ist die durch $p$ dividierte Beschleunigung:

$$E_p = \frac{S_p}{p} \tag{6.2}$$

Wir betrachten die Beschleunigung auch als die Zeit, die der parallele Algorithmus auf einem Prozessor benötigt (virtuelle Parallelität), dividiert durch die Zeit, die der parallele Algorithmus unter der Verwendung von $p$ Prozessoren benötigt. Diese Definition bewertet die Beschleunigung über, da der Verwaltungsaufwand der parallelen Algorithmus nicht berücksichtigt wird.

## 6.3 Meßergebnisse

### 6.3.1 Der beste sequentielle Algorithmus vs. Simulation

Wie schon in Abschnitt 5.1 erwähnt gibt es zwei Möglichkeiten zur Berechnung des Basisfalles: mit dem besten sequentiellen Algorithmus und durch die Simulation mit virtuellen Prozessoren. Die Auswahl des Algorithmus zur Berechnung der Basisfalles hat einen Einfluß auf die parallelen Ausführungszeiten, wenn die Problemgröße für den Basisfall hinreichend groß ist. Die parallele Zeit $t_p$ ist gleich der Summe der Zeiten für die Lösung des Basisfalles $t_b$ und den anderen Zeiten $t_a$, die unabhängig von der Lösung des Basisfalles sind ($t_p = t_a + t_b$). Die Zeit des besten sequentiellen Algorithmus ist besser als die der Simulation daraus folgt, daß die parallele Ausführungszeit besser mit dem besten sequentiellen Basisfall, als mit der Simulation ist. Wie stehen die Effizienzen zueinander?

$$
\begin{aligned}
E_p^v &\stackrel{?}{=} E_p^b \\
\frac{t_s^v}{p*(t_a+t_b^v)} &= \frac{t_s^b}{p*(t_a+t_b^b)} \\
t_s^v*(t_a+t_b^b) &= t_s^b*(t_a+t_b^v) \\
wegen\ t_b^b = \frac{t_s^b}{\alpha},\ t_b^v &= \frac{t_s^v}{\alpha},\ \alpha \in \mathbb{N} \qquad (*) \\
t_s^v*t_a + t_s^v*\frac{t_s^b}{\alpha} &= t_s^b*t_a + t_s^b*\frac{t_s^v}{\alpha} \\
t_s^v*t_a &= t_s^b*t_a
\end{aligned}
$$

wobei das $v$ für die virtuellen Prozessoren und $b$ für den besten sequentiellen Algorithmus steht. $(*)$ gilt unter der Annahme, daß die beiden Algorithmen die gleiche Komplexität haben und die gleiche Anzahl von Teilungsschritte durchgeführt wurde. Aus $t_s^v > t_s^b$ folgt $E_p^v > E_p^b$, obwohl $t_p^v > t_p^b$ ist. Je schlechter die sequentielle Zeit für den Basisfall ist, desto bessere Effizienzen erreicht man.

### 6.3.2 Effizienzvergleich bei verschiedenen Problemteilungsgraden

Um die Auswirkungen des Problemteilungsgrades auf die Effizienz beobachten zu können, wurden Experimente für drei Algorithmen: Scan ($k = 2$) und Polynomprodukt ($k = 3$, $k = 4$) mit zentralisierten und verteilten Ein/Ausgabedaten durchgeführt. In Abb. 6.1 und 6.2 sind die Ergebnisse für die sequentiellen Basisfälle mit dem besten sequentiellen Algorithmus zusammengefaßt. Die Ergebnisse für die sequentiellen Basisfälle mit den virtuellen Prozessoren, d.h. die parallelen Ausführungszeiten mit mehreren Prozessoren beziehen sich auf die Zeiten des parallelen Algorithmus mit einem Prozessor, der die parallele Abarbeitung simuliert, befinden sich in Anhang A.1, A.2 und A.3. Die Ausführungszeit schließt die *input* und *output* Zeiten nicht ein. Für eine feste Anzahl von Prozessoren steigt die Effizienz mit wachsender Problemgröße in allen Fällen. Für Problemteilungsgrade $k > 2$ sind die parallelen Ausführungszeiten zufriedenstellend. Für $k = 2$ gibt es im verteilten Fall unnötige Kommunikationsschritte. Die Daten werden am Anfang in $2^d$ Partitionen aufgeteilt und nach dem Kommunikationsschema wird die Information zwischen den Partitionen ausgetauscht. Es gibt aber Algorithmen, die die vollständige Information über die Eingabedaten nicht brauchen (wie z.B. Scan). Das ist der Nachteil der Skelett-Implementierung, die mit einem Problem aus einer Algorithmenklasse instanziiert wird und nicht auf bestimmte Algorithmen optimiert ist. Die absoluten Ausführungszeiten verhalten sich im Gegenteil zu Effizienzen nicht monoton (siehe Tabellen in Abb. 6.1 und 6.2), abhängig von der Anzahl der Prozessoren sinkt zuerst die Ausführungszeit und dann steigt sie wieder.

Abb. 6.3 zeigt den Einfluß des Problemteilungsgrades auf die sequentiellen Ausführungszeiten für das Polynomprodukt mit Problemteilungsgraden $k = 3$ und $k = 4$.

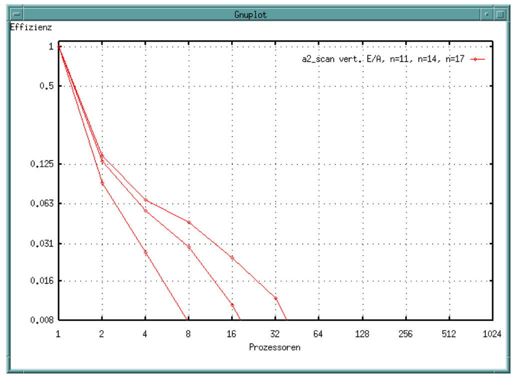
### 6.3.3 Verteilte vs. zentralisierte Ein/Ausgabedaten

Die Effizienz für verteilte und zentralisierte Ein/Ausgabedaten verhält sich unterschiedlich für unterschiedliche Problemteilungsgrade. Für alle Problemteilungsgrade ist die verteilte Ein/Ausgabe ab einer bestimmten Problemgröße besser als die zentralisierte ( A.1 und A.6). Der Grund dafür ist das Kommunikationsmuster. Wenn man die zwei Kommunikationsmuster (Abb. 4.5 und 4.7) betrachtet, sieht man, daß es im verteilten Fall eine Kommunikation mehr gibt. Außerdem gibt es einen Schritt in der verfeinerten Tiefendimension mehr. D.h. im zentralisierten Fall werden die Daten in jedem Schritt $(d, i)$ parallel versendet. Im verteilten Fall gibt es zuerst zwei sequentielle Schritte $(d, 0)$, $(d, 1)$ und ab $(d, 2)$ erst parallele Schritte. Anderseits gesehen ist die zu versendende Datenmenge im verteilten Fall nicht so groß, wie im zentralisierten. Um die mit der Anzahl der Kommunikationsschritte verbundenen Nachteile mit der Datengröße auszugleichen, müssen die Probleme sehr groß sein. Für Scan ist das die Problemgröße $2^{17}$, ab der die verteilte Ein/Ausgabe für jede Anzahl von Prozessoren vorteilhafter ist, als die zentralisierte. Wegen dem Speicherplatz können verteilten Ein/Ausgabedaten $2^d$ mal größere Probleme als mit zentralisierten gelöst werden, weil die Eingabedaten am Anfang auf $2^d$ Partitionen aufgeteilt sind ($d$ ist die Anzahl der Teilungsschritte).

### 6.3.4 Der Kommunikationsanteil

Der Anteil der Kommunikationszeiten an der Gesamtberechnung ist sehr groß und sinkt mit steigender Problemgröße. Die Kommunikationsanteilkurven verhalten sich mit der verteilten Ein/Ausgabe ähnlich für verschiedene Problemteilungsgrade (siehe Abb. A.1, A.2 und A.3). Am Anfang ist der Kommunikationsanteil sehr groß und sinkt mit wachsender Problemgröße. Je mehr Prozessoren an der Berechnung beteiligt sind, desto später (für größeres $n$) fällt die Kurve ab. Mit wachsendem Problemteilungsgrad sinkt der Kommunikationsanteil an der Gesamtberechnung für eine bestimmte Anzahl von Prozessoren.

Über den Verlauf der Kurve der zentralisierten Ein/Ausgabe kann man nicht viel sagen (Abb. A.1, A.2 und A.3). Für den Problemteilungsgrad $k > 2$ scheint er ähnlich dem Verlauf für die verteilte Ein/Ausgabe zu sein, für $k = 2$ stagniert die Kurve bei einem konstanten Wert.

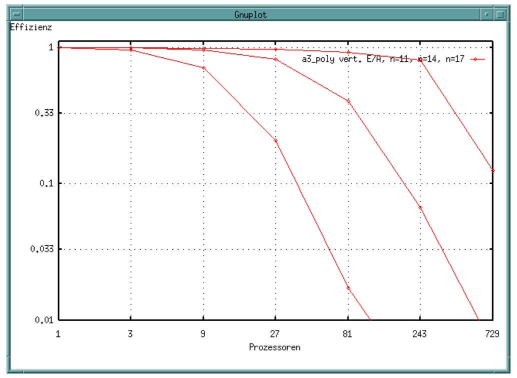



(a) Effizienz

| $p$ | Zeit in s | Beschl. | Effizienz |
|---|---|---|---|
| $n = 11$ | | | |
| 1 | 0.0209 | - | - |
| 2 | 0.1153 | 0.1812 | 0.0906 |
| 4 | 0.1973 | 0.1016 | 0.0265 |
| 8 | 0.3423 | 0.0608 | 0.0076 |
| 16 | 0.8143 | 0.0256 | 0.0016 |
| 32 | 1.8396 | 0.0114 | 0.0004 |
| 64 | 4.4985 | 0.0046 | 0.0001 |
| 128 | 8.2760 | 0.0025 | 0.0000 |
| 256 | 16.4584 | 0.0013 | 0.0000 |
| 512 | 39.6850 | 0.0005 | 0.0000 |
| 1024 | 91.4774 | 0.0002 | 0.0000 |
| $n = 14$ | | | |
| 1 | 0.1672 | - | - |
| 2 | 0.6297 | 0.2655 | 0.1328 |
| 4 | 0.7535 | 0.2219 | 0.0555 |
| 8 | 0.7209 | 0.2319 | 0.0290 |
| 16 | 0.9915 | 0.1686 | 0.0105 |
| 32 | 2.1579 | 0.0768 | 0.0024 |
| 64 | 4.7975 | 0.0320 | 0.0005 |
| 128 | 7.9562 | 0.0256 | 0.0002 |
| 256 | 17.4939 | 0.0096 | 0.0000 |
| 512 | 44.8268 | 0.0037 | 0.0000 |
| 1024 | 92.9411 | 0.0018 | 0.0000 |
| $n = 17$ | | | |
| 1 | 1.3377 | - | - |
| 2 | 4.5654 | 0.2930 | 0.1465 |
| 4 | 4.9671 | 0.2692 | 0.0673 |
| 8 | 3.6845 | 0.3632 | 0.0454 |
| 16 | 3.4716 | 0.3856 | 0.0241 |
| 32 | 3.5127 | 0.3808 | 0.0119 |
| 64 | 5.6032 | 0.2387 | 0.0037 |
| 128 | 8.9002 | 0.1503 | 0.0012 |
| 256 | 18.8216 | 0.0711 | 0.0003 |
| 512 | 38.9016 | 0.0384 | 0.0000 |
| 1024 | 110.6692 | 0.0121 | 0.0000 |

(b) Absolute Meßgrößen

Abbildung 6.1: Scan ($k$=2), verteilte Ein/Ausgabe, der beste seq. Algorithmus




(a) Effizienz

| $p$ | Zeit in s | Beschl. | Effizienz |
|---|---|---|---|
| $n = 11$ | | | |
| 1 | 10.2922 | - | - |
| 3 | 3.5634 | 2.8884 | 0.9628 |
| 9 | 1.6109 | 6.3891 | 0.7099 |
| 27 | 1.8369 | 5.6025 | 0.2075 |
| 81 | 7.3271 | 1.4013 | 0.0173 |
| 243 | 14.7230 | 0.7047 | 0.0029 |
| 729 | 52.9062 | 0.2187 | 0.0003 |
| $n = 14$ | | | |
| 1 | 282.0924 | - | - |
| 3 | 94.6603 | 2.9799 | 0.9933 |
| 9 | 32.5522 | 8.6661 | 0.9629 |
| 27 | 12.8002 | 22.0374 | 0.8162 |
| 81 | 8.6161 | 32.7402 | 0.4042 |
| 243 | 17.3260 | 5.4270 | 0.0670 |
| 729 | 60.6301 | 4.6656 | 0.0064 |
| $n = 17$ | | | |
| 1 | 7634.3229 | - | - |
| 3 | 2553.4561 | 2.9898 | 0.9966 |
| 9 | 858.7543 | 8.8902 | 0.9878 |
| 27 | 292.0528 | 26.1414 | 0.9682 |
| 81 | 102.3799 | 74.5686 | 0.9206 |
| 243 | 39.0418 | 195.5421 | 0.8047 |
| 729 | 84.2041 | 90.6876 | 0.1244 |

(b) Absolute Meßgrößen

Abbildung 6.2: Polynomprodukt ($k$=3), verteilte Ein/Ausgabe, der beste seq. Algorithmus

| Problemgröße | $n = 8$ | $n = 9$ | $n = 10$ | $n = 11$ | $n = 12$ | $n = 13$ | $n = 14$ |
|---|---|---|---|---|---|---|---|
| $k = 3$ in s | 0.363 | 1.114 | 3.396 | 10.292 | 31.092 | 93.721 | 282.092 |
| $k = 4$ in s | 10.766 | 43.064 | 172.256 | 689.024 | 2756.096 | 11024.384 | 44097.536 |

Abbildung 6.3: Gegenüberstellung der sequentiellen Ausführungszeiten mit dem besten Algorithmus für das Polynomprodukt mit $k = 3$ und $k = 4$

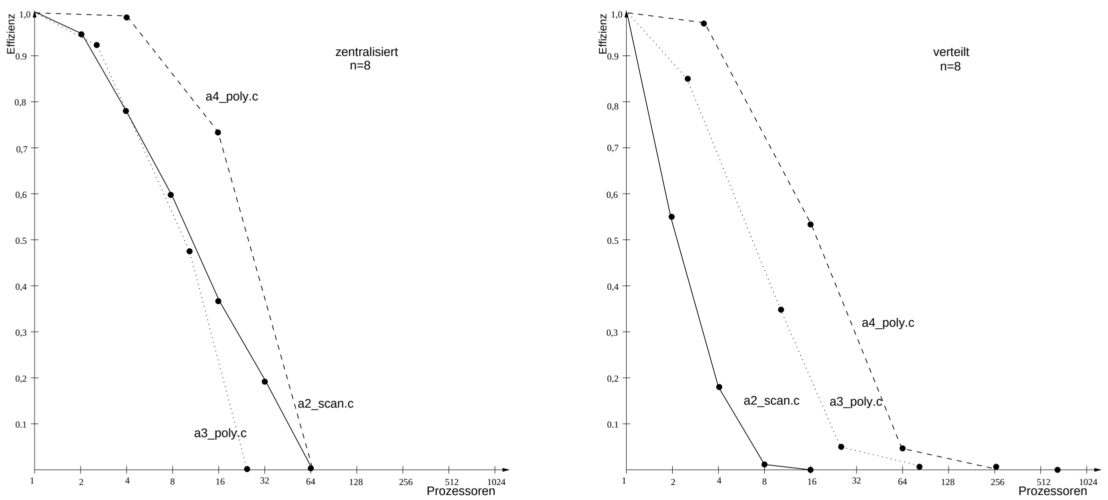



Abbildung 6.4: Effizienzvergleich (virtuelle Prozessoren)

### 6.3.5 Die Plazierung

Die Auswahl der Plazierungsfunktion hat keinen bedeutsamen Einfluß auf die Effizienz. Experimente für `a3_poly` mit verteilten Ein/Ausgabe mit 243 Prozessoren und drei Plazierungsfunktionen:
$map_0(vpid) = vpid$,
$map_1(vpid) = 13 * vpid \mathsf{\ mod\ } vpid_{max}$,
$map_2(vpid) = 32 * y_1 + y_0$ (wobei $y_0 = vpid \mathsf{\ mod\ } 27, y_1 = vpid \mathsf{\ div\ } 27$ die Koordinaten der realen Topologie sind),
zeigen Effizienzschwankungen um ein Prozent.

### 6.3.6 Die Baumkommunikation

Wir haben auch mit einem Kommunikationsschema experimentiert, das nur die Kommunikation in der Baumtopologie zuläßt, d.h. Prozessoren, die zum Berechnungszeitpunkt nicht die Knoten des Berechnungsgraphen repräsentieren, werden von der Kommunikation ausgeschlossen. Experimente für das Polynomprodukt ($k = 3$) mit verteilter Ein/Ausgabe mit 243 Prozessoren haben bestätigt, daß diese Einschränkung keinen positiven Einfluß auf die Effizienz hat. Die Effizienzdifferenzen liegen bei einem Prozent. Aus Lesbarkeitsgründen haben wir uns für das Schema mit der Gitterkommunikation entschieden, da sie weniger Bedingungen und Verwaltungsaufwand enthält.

### 6.3.7 Compress and Conquer

Für Compress-and-Conquer gelten folgende Berechnungen (siehe auch Kap. 4.7): Wir betrachten Scan auf Daten der Größe $N = 2^{d+m}$. Auf jedem von $2^d$ Prozessoren liegen Daten der Gröse $2^m$. Es werden drei Phasen ausgeführt:

- Auf jedem Prozessor sequentielle Reduktion $2^m$ Elemente: Zeit $c_0 * 2^m$
- Paralleles Scan: Zeit $c_1 * d$ (unabhängig von $m$)
- Auf jedem Prozessor sequentielles Scan: Zeit $c_2 * 2^m$.

Daraus folgt:

$$\begin{aligned}
\text{die sequentielle Zeit } t_s &= c_3 * 2^{(d+m)} \\
\text{die parallele Zeit } t_p &= (c_0 + c_2) * 2^m + c_1 * d \\
S_p &= \frac{c_3 * 2^{(d+m)}}{(c_0 + c_2) * 2^m + c_1 * d} = \frac{c_3 * 2^d}{(c_0 + c_2) + \frac{c_1 * n}{2^m}} \\
\lim_{m \to \infty} S_p &= \frac{c_3 * 2^d}{c_0 + c_2} \\
E_p &= \frac{S_p}{2^d} = \frac{c_3}{c_0 + c_2} \qquad \text{für } m \to \infty
\end{aligned}$$

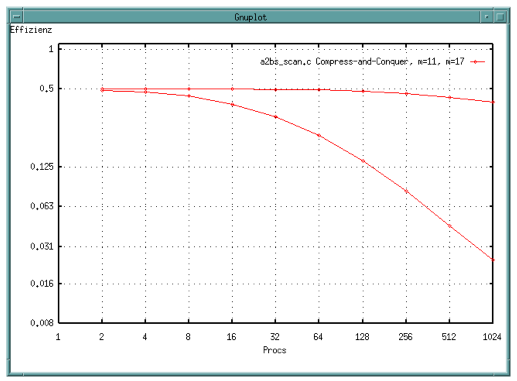



Abbildung 6.5: Scan ($k = 2$) mit Compress-and-Conquer

Für die Effizienzberechnung betrachten wir folgende Zeiten:

- Die sequentielle Zeit für Scan mit der Problemgröße $N = 2^{d+m}$

- Die parallele Zeit für Scan mit der Problemgröße $N = 2^d$ und zwei mal die sequentielle Zeit für Scan mit der Problemgröße $N = 2^m$

Die so berechnete Effizienz aus den praktischen experimentellen Werten mit dem besten sequentiellen Algorithmus für Scan bestätigt die theoretischen Überlegungen. Mit wachsendem $m$ steigt die Effizienz bis 0.5.

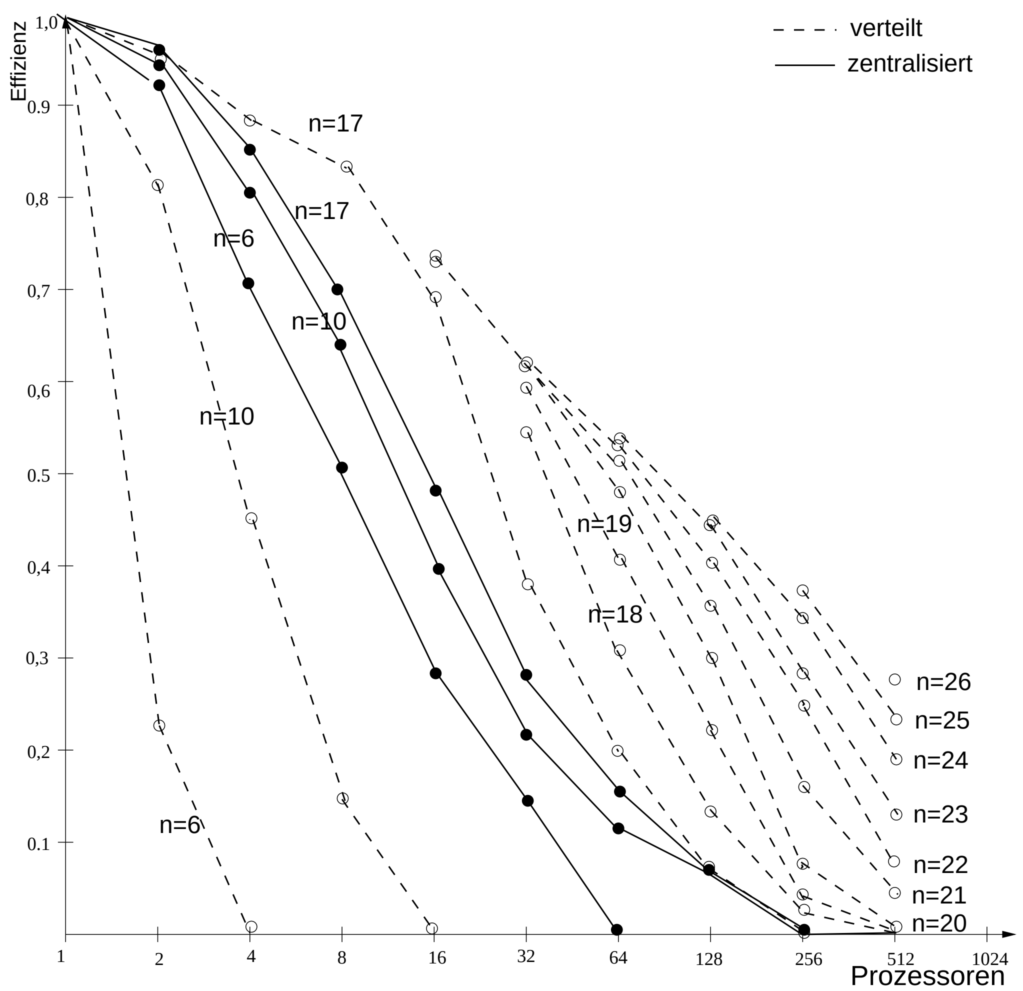
# Kapitel 7

# Abschließende Bemerkungen

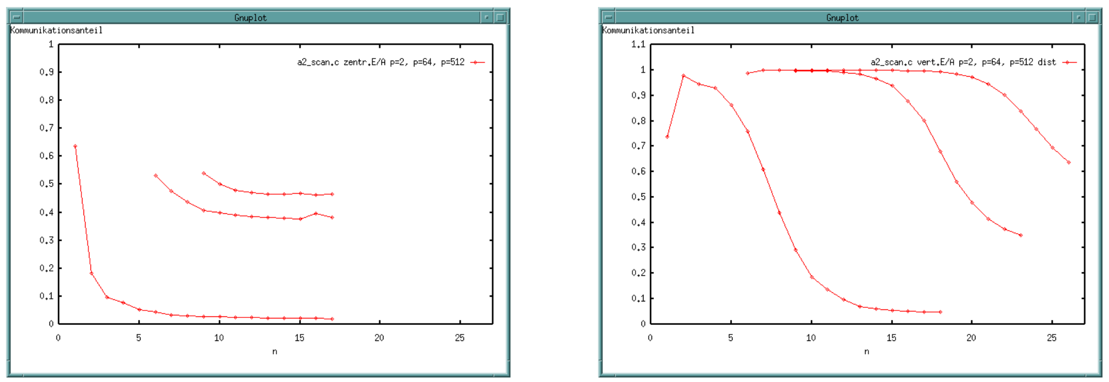
Ein imperatives DC-Skelett ist mit einer Klasse von parallelen DC-Algorithmen ausführbar. Es stellt eine effiziente Algorithmenimplementierung für diese Klasse, aber keine effiziente Implementierung einzelner Algorithmen, dar. Mit dem Skelett kann man schnell und einfach programmieren, ohne sich über die Verwaltung der parallelen Architektur Gedanken zu machen. Das Skelett arbeitet aber nach einem festen Schema und kann schlecht auf bestimmte Algorithmen optimiert werden. Daraus folgt, daß die Effizienz für manche speziellen DC-Algorithmen viel schlechter ist, als wenn man sie mit anderen Techniken implementiert (z.B. Compress-and-Conquer). Das Skelett für die verteilten Ein/Ausgabedaten enthält, um universell zu bleiben, unnötige Kommunikationsschritte, die in vielen Fällen vermieden werden könnten. Wir haben auf die Systembibliothekfunktionen für die virtuellen Topologien verzichtet, um unabhängig zu bleiben und Zeitmessungen für verschiedene Plazierungsfunktionen vornehmen zu können. Das hat den Nachteil, daß nur die synchronen send/receive Operationen ausgeführt werden können, da die asynchronen eine virtuelle Topologie aus der Systembibliothek verlangen. Die Plazierung hat nur einen geringen Einfluß auf die Kommunikationszeiten, da letztendlich die Verbindungspfade zwischen den plazierten Prozessoren durch das Routing bestimmt werden. Die verteilten Ein/Ausgabedaten bringen wegen der zusätzlichen Kommunikationsschritte nicht die erwartete Leistungssteigerung. Die verteilten Ein/Ausgabedaten sind erst für große Probleme vorteilhafter als die zentralisierten und man kann größere Probleme, als mit den zentralisierten lösen. DC-Skeletts stellen eine gute Methode zum Entwurf korrekter paralleler Programme, besonders für Problemteilungsgrade $k > 2$, dar. Mit der Effizienz für den Problemteilungsgrad $k = 2$ kann man nicht zufrieden sein. Das imperative Skelett könnte noch für bestimmte Algorithmen optimiert werden, um befriedigende Effizienzen zu erreichen.

# Anhang A

# Experimentergebnisse

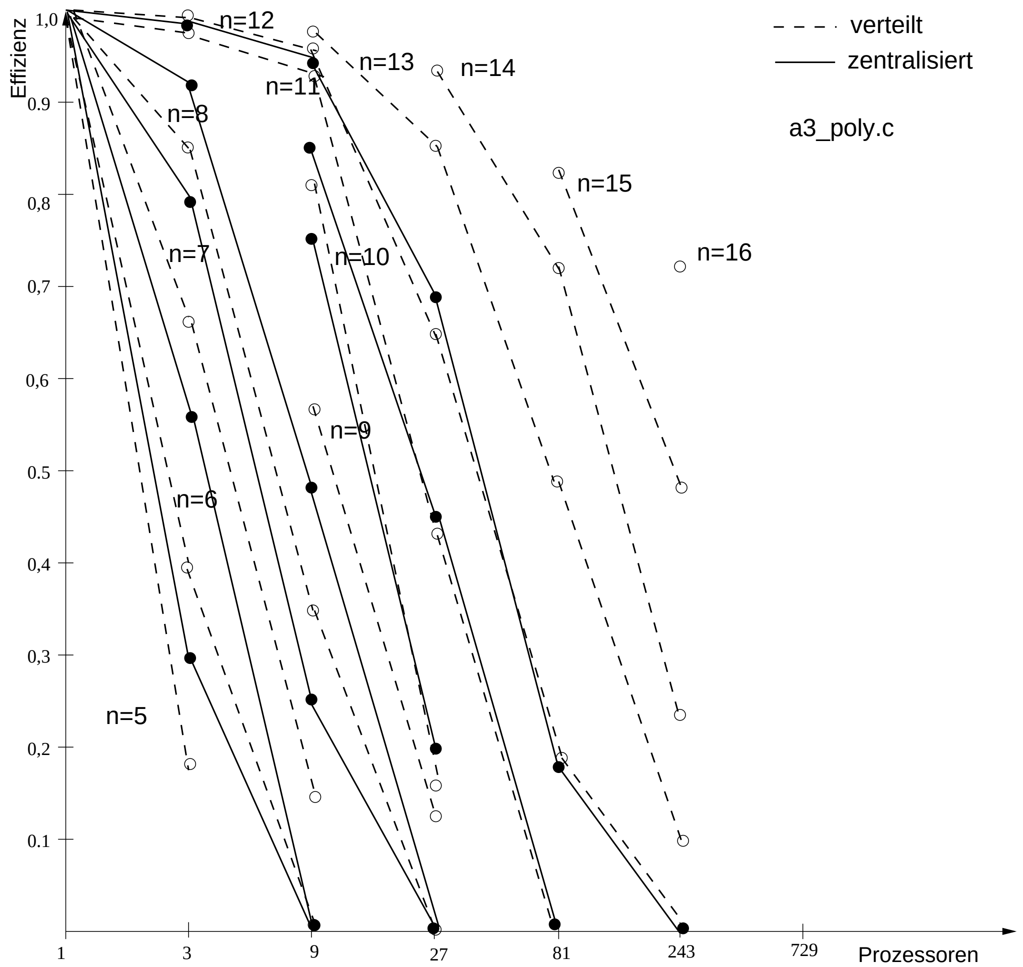



(a) Effizienz für zentralisierte/verteilte Ein/Ausgabe

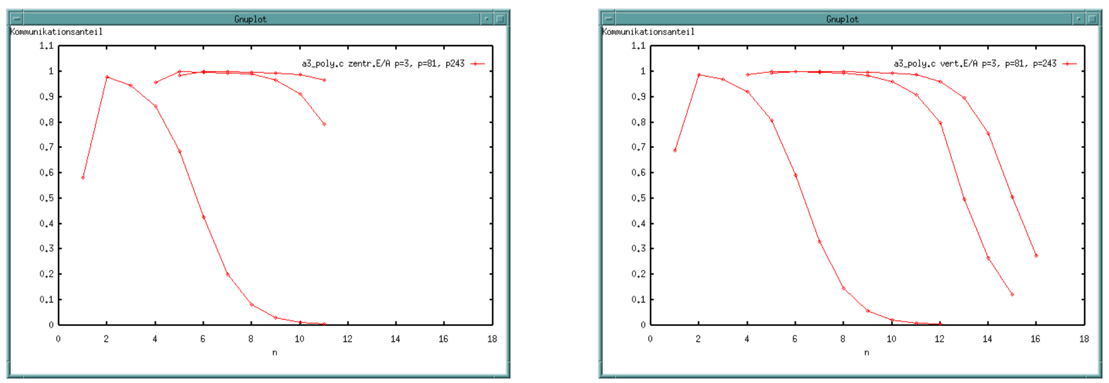



(b) Kommunikationsanteil für zentralisierte/verteilte Ein/Ausgabe

Abbildung A.1: Scan ($k = 2$) mit virtuellen Prozessoren

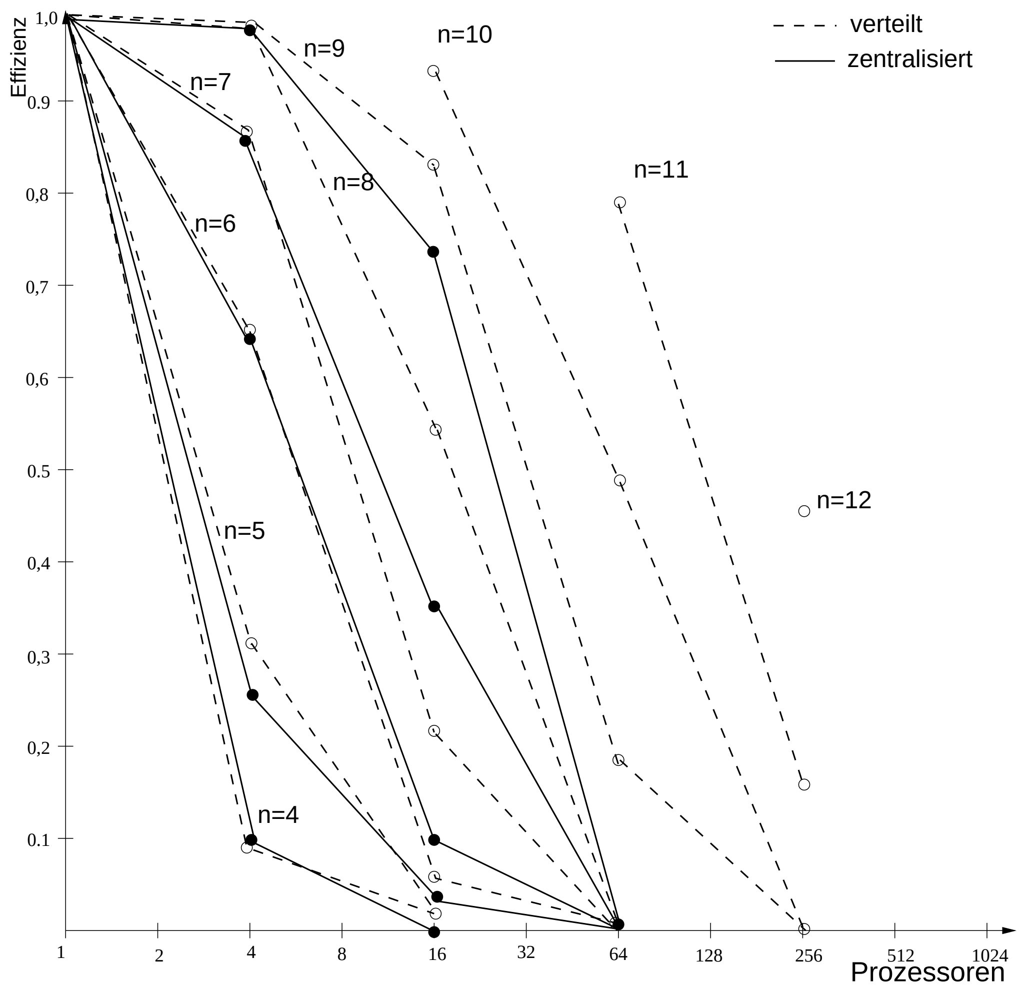



(a) Effizienz für zentralisierte/verteilte Ein/Ausgabe

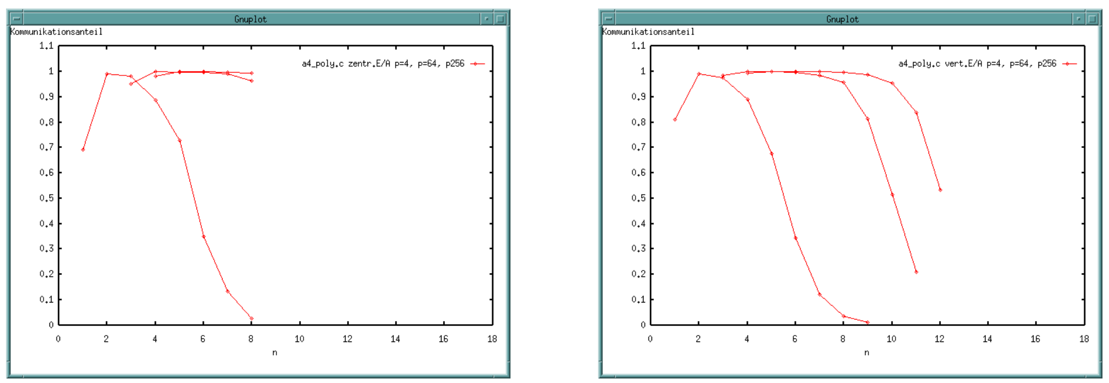


(b) Kommunikationsanteil für zentralisierte/verteilte Ein/Ausgabe

Abbildung A.2: Polynomprodukt ($k = 3$) mit virtuellen Prozessoren




(a) Effizienz für zentralisierte/verteilte Ein/Ausgabe




(b) Kommunikationsanteil für zentralisierte/verteilte Ein/Ausgabe

Abbildung A.3: Polynomprodukt ($k = 4$) mit virtuellen Prozessoren

| n | Komm. | Teilung | Basisfall | | Zus.setzung | Komm. | Gesamt | |
|---|---|---|---|---|---|---|---|---|
| | | | bs | vp | | | bs | vp |
| 11 | 3.9712 | 0.0141 | 0.1184 | 0.5537 | 0.0071 | 3.2163 | 7.3271 | 7.7624 |
| 12 | 3.4693 | 0.0278 | 0.3745 | 1.6666 | 0.0140 | 3.4685 | 7.3541 | 8.6462 |
| 13 | 2.5915 | 0.0549 | 1.1432 | 5.0075 | 0.0283 | 2.5271 | 6.3451 | 10.2094 |
| 14 | 2.1815 | 0.1124 | 3.4511 | 15.1044 | 0.0560 | 2.8151 | 8.6161 | 20.2694 |
| 15 | 2.4207 | 0.2233 | 10.3778 | 45.2251 | 0.1120 | 3.0499 | 16.1837 | 51.0310 |
| 16 | 3.3976 | 0.4459 | 31.2733 | 135.6753 | 0.2244 | 4.1241 | 39.4654 | 143.8674 |
| 17 | 4.8473 | 0.9028 | 94.2090 | 407.0259 | 0.4484 | 6.5318 | 106.9393 | 419.7568 |

Abbildung A.4: Absolute Meßergebnisse in s für das Polynomprodukt ($k = 3$)

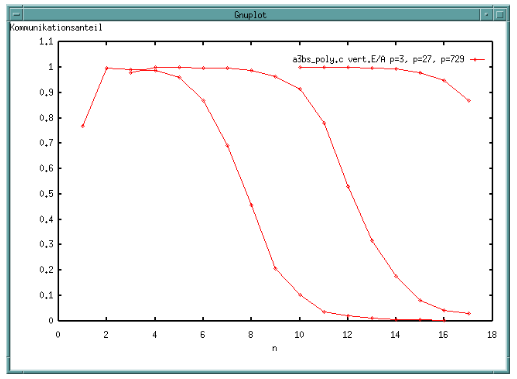



Abbildung A.5: Anteil der Kommunikation für das Polynomprodukt ($k = 3$), verteilte Ein/Ausgabe, der beste seq. Algorithmus

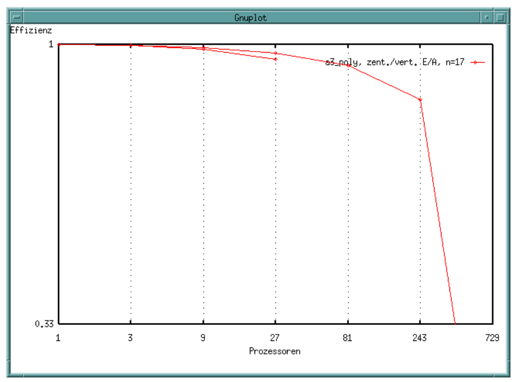



Abbildung A.6: Effizienzvergleich für das Polynomprodukt ($k = 3$) mit zentralisierte/verteilte Ein/Ausgabedaten, der beste seq. Algorithmus

# Anhang B

# Programmlisting

```
/* ----------------------------------------------------------------------------
 *                                FILE sk_typedefs.h
 *                                ^^^^^^^^^^^^^^^^^^
 * Makrodefinitionen und Typdefinitionen fuer die sk_mesh.c Datei.
 * Diese Datei wird von sk.h included.
 *
 * --------------------------------------------------------------------------*/


/* ---------------------------------------------------------------------------
 *                                   INCLUDES
 * --------------------------------------------------------------------------*/

#include "../tmp/types.h"   /*   Typdefinitionen fuer die Algorithmendatei    */

/* ---------------------------------------------------------------------------
 *                           MACROS FOR SKELETON FUNCTIONS
 * --------------------------------------------------------------------------*/

/*                Aufruf der send/receive-Systemfunktionen                    */

#define  RECV_FROM_LEFT(dim, addr, size)  RecvLink(param.link[dim][0],addr,size)
#define  RECV_FROM_RIGHT(dim, addr, size) RecvLink(param.link[dim][1],addr,size)
#define  SEND_TO_LEFT(dim, addr, size)    SendLink(param.link[dim][0],addr,size)
#define  SEND_TO_RIGHT(dim, addr, size)   SendLink(param.link[dim][1],addr,size)

/*           Komponenten der Variable in_vec vom Typ INVECTOR*                */

#define   DIV_V                           in_vec->divided   /*vom Typ INTYPE* */
#define   DIV_V_SIZE                      in_vec->divsize   /* vom Typ int    */
#define   UNDIV_V                         in_vec->undivided /*vom Typ INTYPE* */
#define   UNDIV_V_SIZE                    in_vec->undivsize /* vom Typ int    */

/*           Komponenten der Variable out_vec vom Typ OUTVECTOR*              */

#define   COM_V                           out_vec->combined   /* Typ OUTTYPE* */
#define   COM_V_SIZE                      out_vec->comsize    /*  Typ int     */
#define   UNCOM_V                         out_vec->uncombined /* Typ OUTTYPE* */
#define   UNCOM_V_SIZE                    out_vec->uncomsize  /*  Typ int     */

/* Komponenten der Variablen aux_in_vec vom Typ INVECTOR und
    aux_out_vec vom Typ OUTVECTOR sie enthalten Vektoren, die
    die virtuellen Prozessoren repraesentieren                                */

#define   VP_DIV                          aux_in_vec.divided     /*  INTYPE*  */
#define   VP_D_SIZE                       aux_in_vec.divsize     /*  Typ int  */
#define   VP_UNDIV                        aux_in_vec.undivided   /*  INTYPE*  */
#define   VP_UND_SIZE                     aux_in_vec.undivsize   /*  Typ int  */

#define   VP_COM                          aux_out_vec.combined /*Typ OUTTYPE* */
#define   VP_C_SIZE                       aux_out_vec.comsize    /*  Typ int  */
```

```
#define   VP_UNCOM                        aux_out_vec.uncombined /* OUTTYPE*  */
#define   VP_UNC_SIZE                     aux_out_vec.uncomsize  /*  Typ int  */

/*       Koordinaten der Gitterknoten mit input- und output-Daten            */

#define   RIGHTPOS(k)                     ((k)/2)                /*  int k  */
#define   LEFTPOS(k)                      RIGHTPOS(k)-1          /*  int k  */

/*       Koordinatenbestimmung im n-dim k-Gitter:
         - id virtuelle Prozessornummer
         - k  Problemteilungsgrad
         - offset Abstand zwischen zwei Nachbarprozessoren in einer Dimension */

#define   COORDINATE(id,offset,k)         ((id)/(offset))%(k)


/* --------------------------------------------------------------------------
 *                          TYPES FOR SKELETON FUNCTIONS
 * --------------------------------------------------------------------------*/

#ifndef HOST           /* Ausgeschlossen von der Compilation der host.c Datei */
typedef struct {
           long    divsize;    /* Laenge des geteilten Vektors               */
           INTYPE *divided;    /* geteilter Vektor                           */
           long    undivsize;  /* Laenge des ungeteilten Vektors             */
           INTYPE *undivided;  /* ungeteilter Vektor                         */

} INVECTOR;

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

typedef struct {
           long     comsize;    /* Laenge des zusammengesetzten Vektors      */
           OUTTYPE *combined;   /* zusammengesetzter Vektor                  */
           long     uncomsize;  /* Laenge des nicht zusammengesetzten Vektors */
           OUTTYPE *uncombined; /* nicht zusammengesetzter Vektor            */

} OUTVECTOR;
#endif

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

typedef struct {
           int         *p;          /* Prozessortabelle                      */
           long         vpid;       /* Virtuelle Prozessornummer             */
           long         dim;        /* Anzahl der Dimensionen                */
           long         n;          /* Potenz der Datengroesse size = 2^n    */
           int          cent_in;    /* Art der Eingabe: -1 cent., -0 vert.   */
           int          cent_out;   /* Art der Ausgabe: -1 cent., -0 vert.   */
           int          part;       /* Partitionsnummer                      */
           long         exper;      /* Anzahl der Experimente                */
```

```
          long        k;          /* Problemteilungsgrad                 */
          long        max_vpid;   /* Anzahl der gebrauchten Prozessoren   */
#ifndef HOST
          LinkCB_t  ***link;       /* Linktabelle                         */
#endif
} INFO;

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

                                /* Code fuer die Ausgabezeiten, fuer die    */
typedef enum   { I      ,       /* input() Funktion                         */
                 D      ,       /* divide() Funktion                        */
                 CD     ,       /* divide Kommunikation                     */
                 VPBAS  ,       /* basic() Funktion                         */
                 SEQBAS ,       /* sequential() Funktion                    */
                 C      ,       /* combine() Funktion                       */
                 CC     ,       /* combine Kommunikation                    */
                 O              /* output() Funktion                        */

} PATTERN;

/* ---------------------------------------------------------------------------
 *                                  END
 * -------------------------------------------------------------------------*/
```

```
/* ---------------------------------------------------------------------------
 *                                  FILE sk.h
 *                                  ^^^^^^^^^
 * Makrodefinitionen und Definitionen externer Schnittstellenfunktionen.
 * Diese Datei wird vom Skelett und von jeder customizing-Datei included.
 *
 * --------------------------------------------------------------------------*/



/* ---------------------------------------------------------------------------
 *                                  INCLUDES
 * --------------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/root.h>
#include <sys/memory.h>
#include <sys/rrouter.h>
#include "/usr/include/memory.h"

#include "sk_typedefs.h"

#include "design.c"

/* ---------------------------------------------------------------------------
 *                                   MACROS
 *                          FOR CUSTOMIZING FUNCTIONS
 * --------------------------------------------------------------------------*/


#define STRLEN                    30 /* Laenge einer Zeile des Ausgabestrings */
                                     /*drei Faelle fuer partition in combine()*/
#define CENT                      (-1)           /* -zentralisierte Ausgabe  */
                                                 /* -verteilte Ausgabe:      */
#define RPOS                      ((degree)/2)   /* --rightpos               */
#define LPOS                      ((degree)/2-1) /* --leftpos                */

                                             /* Zeiger auf FILE:            */
#define IN_FILE                   fin_file    /* -form. Parameter von INPUT() */
#define OUT_FILE                  fout_file   /* -form. Parameter von OUTPUT()*/

                                        /* Zeiger auf INVECTOR und OUTVECTOR */
#define IN_VEC                    fin_vec   /* form. Parameter der customizing*/
#define OUT_VEC                   fout_vec  /*           functions            */

                                  /* Zugriff auf die Laenge des Vektors vector*/
                                                    /* vector ist vom Typ:  */
#define D_SIZE(vector)            ((vector)->divsize) /* Zeiger auf INVECTOR  */
#define UND_SIZE(vector)          ((vector)->undivsize)
```

```
#define C_SIZE(vector)            ((vector)->comsize) /* Zeiger auf OUTVECTOR */
#define UNC_SIZE(vector)          ((vector)->uncomsize)

                                  /*               Kopieren                  */
                                  /* des linken Teils des ungeteilten Vektors */
                                  /* in den geteilten Vektor                 */

#define COPY_L                    memcpy(fin_vec->divided, fin_vec->undivided, \
                                                sizeof(INTYPE)*D_SIZE(fin_vec))

                                  /* des rechten Teils des ungeteilten Vektors*/
                                  /* in den geteilten Vektor                 */

#define COPY_R                    memcpy(fin_vec->divided,\
                                         &fin_vec->undivided[D_SIZE(fin_vec)],\
                                         sizeof(INTYPE)*D_SIZE(fin_vec))

                                 /*Speicherplatzreservierung analog zu malloc */
                                 /*fuer einen OUTVECTOR table der Laende size */

#define ALLOC(table, size)        (table) =(OUTTYPE *)\
                                            MALLOC(sizeof(OUTTYPE)*(size))

                                  /*Anfangsadressen der vier Vektorkomponenten*/

#define DIVIDED(vector)           ((vector)->divided)
#define UNDIVIDED(vector)         ((vector)->undivided)
#define COMBINED(vector)          ((vector)->combined)
#define UNCOMBINED(vector)        ((vector)->uncombined)

                                  /* Anfangsadressen der Vektorkomponenten,   */
                                  /* die um n*I_LEN von der Vektoranfangs-    */
                                  /* adresse  entfernt sind                   */

#define DIV(n)                    (fin_vec->divided[I_VAR + (n)*I_LEN])
#define UNDIV(n)                  (fin_vec->undivided[I_VAR + (n)*I_LEN])

#define COM(n)                    (fout_vec->combined[I_VAR + (n)*I_LEN])
#define UNCOM(n)                  (fout_vec->uncombined[I_VAR + (n)*I_LEN])

                                  /* Zugriff auf ein Vektorelement in basic() */

#define BCOM                      (fout_elem[0])                  /* OUTTYPE */
#define BDIV                      (fin_elem)                      /* INTYPE  */

                                                          /*  Iteratoren   */

                                               /* fuer die divide()-Funktion */

#define DIV_ITER(body)            {long I_VAR, I_LEN = D_SIZE(fin_vec);  \
                                        for(I_VAR = 0; I_VAR < I_LEN; I_VAR++)\
```

```
                                          (body);}

                                             /* fuer die combine()-Funktion */
                                             /* zentralisierte Ausgabe      */

#define COM_CENT_ITER(body)       {long I_VAR, I_LEN = C_SIZE(fout_vec)/2 ; \
                                        for(I_VAR = 0; I_VAR < I_LEN; I_VAR++)\
                                            (body);}

                                             /* fuer die combine()-Funktion */
                                             /* verteilte Ausgabe           */

#define COM_DIST_ITER(body)       {long I_VAR, I_LEN = C_SIZE(fout_vec);\
                                        for(I_VAR = 0; I_VAR < I_LEN ; I_VAR++)\
                                           (body);}

                                             /*  fuer die input()-Funktion  */

#define IN_ITER(body)             DIV_ITER(body)

                                     /*  Laenge des iterierten Eingabevektors */

#define INITERLEN                 D_SIZE(IN_VEC)

                                  /* Ueberlesen der Daten aus der input-Datei */
                                  /* condition bedeutet die Anzahl der        */
                                  /* ueberlesenen Daten                       */

#define OVERREAD(condition)       {long I_VAR; \
                                    for(I_VAR = 0; I_VAR < condition ; I_VAR++)\
                                        fscanf(IN_FILE, "%d ");}


                                            /*  fuer die output()-Funktion  */

#define OUT_ITER(body)            {long I_VAR, I_LEN = C_SIZE(fout_vec); \
                                    for(I_VAR = 0; I_VAR < I_LEN ; I_VAR++)\
                                           (body);}

                                                   /* customizing functions */

#define INPUT(body)               void input( FILE *fin_file, INFO fparam,\
                                     INVECTOR *fin_vec ){\
                                     long I_VAR;  switch(NOTEXPERIMENT(fparam))\
                                                        {body};}

#define OUTPUT(body)              void output( FILE *fout_file, INFO fparam, \
                                       OUTVECTOR *fout_vec ){\
                                       long I_VAR; body}

#define BASIC(body)               void basic( INTYPE fin_elem,\
```

```
                                                    OUTTYPE *fout_elem )\
                                        {body}

#define SEQUENTIAL(body)          void sequential(INVECTOR *fin_vec,\
                                                   OUTVECTOR *fout_vec )\
                                        {body}

#define DIVIDE(body)              void divide( long branch, INVECTOR *fin_vec )\
                                        {switch(branch){ body } ;}

#define COMBINE(body)             void combine( long partition,\
                                                     OUTVECTOR *fout_vec )\
                                        {switch(partition){ body } ;}

#define IN_V                      DIV(0) /* Anfangsadresse des input-Vektors  */
                                         /*             INTYPE               */

#define OUT_V                     COM(0) /* Anfangsadresse des output-Vektors */
                                         /*             OUTTYPE              */

                                         /*          Zeitmessungen           */
#ifdef TIME
#define S_TIME(t)                 ((t) = TimeNowLow())
#define SECONDS(t1, t2)           (((double)((t2)-(t1)))/((double)15625.0))
#else
#define S_TIME(t)                       /*falls keine Zeitmessungen gewuenscht*/
#define SECONDS(t1, t2)                 /*wird es mit leeren Zeichen ersetzt  */
#endif                                  /*und von der Compilation ausgeschl.  */

                                /* Zugriff auf die Komponenten der INFO struct*/

#define NOTEXPERIMENT(x)         !((x).exper)
#define IN_CENT                   (fparam.cent_in)
#define IN_DIST                  !(fparam.cent_in)
#define PARTITION                 (fparam.part)
#define DIM                       (fparam.dim)

#define PRINT(file, format, elem) fprintf(file, format, elem)

                                 /* Die Daten koennen gelesen oder            */
                                 /* geschrieben werden ohne expliziete Eingabe*/
                                 /* der Ein- Ausgabedatei                     */
#define SCAN_IN(var)              fscanf(fin_file, "%d ", var)
#define PRINT_OUT(var)            fprintf(fout_file, "%d ", var)

/* ---------------------------------------------------------------------------
 *                                CUSTOMIZING FUNCTIONS
 * --------------------------------------------------------------------------*/

/* ---------------------------------------------------------------------------
 *  NAME       : input()
```

```
 *
 *  PARAMETER   : FILE      *in_file
 *                INFO       fparam
 *                INVECTOR  *fin_vec
 *
 *  DESCRIPTION : initialisiert einen Eingabevektor fin_vec der Laenge INITERLEN
 *                mit beliebigen Werten. Auf die Elemente des Eingabevektors
 *                kann mit IN_V innerhalb von IN_ITER(body) zugegriefen werden.
 *                Der Typ der Vektorelemente muss in types.h definiert werden.
 *                Falls man den Experimente-modus waehlt ( case = 0 ),
 *                kann der Vektor direkt initialisiert werden, sonst wird es von
 *                einer Eingabedatei gelesen, auf die ein Zeiger in_file
 *                zur Verfuegung steht. Die Eingabedatei muss bereits existiere
 *                und Werte beinhalten die kompatibel mit dem Typ der Elemente
 *                des Eingabevektors sind.
 *                Bsp.: INPUT(case 0: DIV_ITER( IN_V.a = 1 ); break;)
 * ---------------------------------------------------------------------------*/
extern
void input( FILE *in_file, INFO fparam, INVECTOR *fin_vec );

/* ----------------------------------------------------------------------------
 *  NAME        : output()
 *
 *  PARAMETER   : FILE      *out_file
 *                INFO      fparam
 *                INVECTOR  *fout_vec
 *
 *  DESCRIPTION : schreibt den Inhalt des output-Vektors OUT_V auf eine Datei.
 *                Es steht ein Zeiger out_file auf die Ausgabedatei zur
 *                Verfuegung.
 *                Bsp.: OUTPUT( OUT_ITER( PRINT_OUT( OUT_V.scan) ) )
 * ---------------------------------------------------------------------------*/
extern
void output( FILE  *out_file , INFO  fparam, OUTVECTOR *fout_vec );

/* ----------------------------------------------------------------------------
 *  NAME        :  basic()
 *
 *  PARAMETER   : INTYPE      fin
 *                OUTTYPE    *fout
 *
 *  DESCRIPTION : loest das einfache Problem. Die Loesung wird in das
 *                Vektorelement fout gespeichert. Das Argument ist
 *                ein Vektorelement fin.
 *                Bsp.: BASIC( BCOM.scan = BDIV.a;
 *                             BCOM.sum  = BDIV.a; )
 * ---------------------------------------------------------------------------*/
extern
void basic( INTYPE   fin, OUTTYPE   *fout );

/* ----------------------------------------------------------------------------
```

```
 *  NAME        :  sequentiell()
 *
 *  PARAMETER   : INVECTOR    *fin_vec
 *                OUTVECTOR   *fout_vec
 *
 *  DESCRIPTION : loest das einfache Problem mit einem sequentiellen Algorithmus
 *                Es stehen zwei Vektoren zur Verfuegung:
 *                 - fin_vec als Argument
 *                 - fout_vec als Ergebnis
 * --------------------------------------------------------------------------*/
extern
void sequential( INVECTOR    *fin_vec, OUTVECTOR   *fout_vec );

/* ---------------------------------------------------------------------------
 *  NAME        : divide()
 *
 *  PARAMETER   : long       branch
 *                INVECTOR  *fin_vec
 *
 *
 *  DESCRIPTION : entspricht der g Funktion im Pseudomorphismus-Modell,
 *                implementiert die inter-Vektorkommunikation.
 *                Teilt einen Vektor UNDIVIDED(fin_vec) der Laenge
 *                UND_SIZE(fin_vec) in einen Vektor DIVIDED(fin_vec) der Laenge
 *                D_SIZE(fin_vec).
 *                Auf die Vektorelemente kann man mit DIV() und UNDIV()
 *                innerhalb von DIV_ITER(body) zugreiffen. Es muessen k Faelle
 *                beruecksichtigt werden.
 *
 * --------------------------------------------------------------------------*/
extern
void divide( long  branch , INVECTOR *fin_vec );

/* ---------------------------------------------------------------------------
 *  NAME        : combine()
 *
 *  PARAMETER   : long partition   -1: centralized output
 *                                 0, 1: distr. output
 *                OUTVECTOR  *fout_vec
 *
 *  DESCRIPTION : entspricht der h Funktion im Pseudomorphismus-Modell,
 *                implementiert die inter-Vektorkommunikation.
 *                Setzt einen Vektor COMBINED(fout_vec) der Laenge
 *                C_SIZE(fout_vec) aus k-Teilen zusammen.
 *                Es gibt zwei Moeglichkeiten:
 *                - zentralisierte Ausgabedaten : partition  = CENT
 *                - verteilte      Ausgabedaten : partition  = LPOS, RPOS
 *                Auf die Vektorelemente kann man mit COM() und UNCOM()
 *                innerhalb von  COM_DIST_ITER(body) und COM_CENT_ITER(body)
 *                zugreiffen.
 * --------------------------------------------------------------------------*/
```

```
extern
void combine( long partition,  OUTVECTOR  *fout_vec );

/* ----------------------------------------------------------------------------
 *                                OTHER FUNCTIONS
 * --------------------------------------------------------------------------*/

/* ----------------------------------------------------------------------------
 *  NAME        : MALLOC()
 *
 *  PARAMETER   : long l
 *
 *  DESCRIPTION : identisch mit malloc(), gibt zusaetzlich Fehlermeldung aus,
 *                falls der Speicher ueberlaeuft und bricht die Berechnung ab.
 * --------------------------------------------------------------------------*/
extern
char *MALLOC( long l );

/* ----------------------------------------------------------------------------
 *  NAME        : ERROR()
 *
 *  PARAMETER   :  char *s
 *
 *  DESCRIPTION :  gibt eine Fehlermeldung s auf den Bildschirm aus und bricht
 *                 die Berechnung ab.
 * --------------------------------------------------------------------------*/
extern
void  ERROR(char *s);

/* ----------------------------------------------------------------------------
 *  NAME        : OPEN_FILE()
 *
 *  PARAMETER   : char *f_name,
 *                char *access_mod
 *
 *  DESCRIPTION : oeffnet eine Datei f_name mit Zugriffrecht access_mode
 *                identisch mit fopen(), gibt zusaetzlich Fehlermeldung aus,
 *                falls die Datei nicht geoeffnet werden kann und bricht die
 *                Berechnung ab.
 * --------------------------------------------------------------------------*/
extern
FILE *OPEN_FILE(char *f_name, char *access_mode);

/* ----------------------------------------------------------------------------
 *  NAME        : LOG()
 *
 *  PARAMETER   : int base,
 *                long x
 *
 *  DESCRIPTION : berechnet den int-Logarithmus von x bei base.
 * --------------------------------------------------------------------------*/
```

```
extern
long LOG(int base, long x);

/* ---------------------------------------------------------------------------
 *                                    END
 * --------------------------------------------------------------------------*/
```

```
/* --------------------------------------------------------------------------
 *                                  FILE sk_mesh.c
 *                                  ^^^^^^^^^^^^^^
 * Implementierung des balanced FDDC-Skeletts auf ein n-dim k-Gitter.
 *
 * --------------------------------------------------------------------------*/


/*---------------------------------------------------------------------------
                                 INCLUDES
-----------------------------------------------------------------------------*/

#include "sk.h"
#include "sk_funcdefs.c"


/*---------------------------------------------------------------------------
                               MAIN PROGRAM
-----------------------------------------------------------------------------*/

main(void) {

    INFO        param ;
    INVECTOR    in_vec ;
    OUTVECTOR   out_vec;
    FILE       *file, *in_file, *out_file ;

    int         i, map, in_f, seq ;
    long        vpid=-1L;
    char        s[10], s_in[20], s_out[20], s_time[100], *res_string , *tstr;

   /*-----------Lesen der Parameter von der control-Datei---------------------*/

    file = OPEN_FILE("../custom/control", "r");
    fscanf(file, "-d%d -n%d -i%d -o%d -x%ld -m%d -if%d -s%d\n",
           &param.dim, &param.n, &param.cent_in, &param.cent_out,
                                 &param.exper, &map, &in_f, &seq);
    fclose(file);

   /*--------------------------Problemteilungsgrad-------------------------- */

    param.k = degree;          /* degree ist ein in types.h definiertes Makro */

   /*----------Prozessorenanzahl, die an der Berechnung beteiligt sind--------*/

     param.max_vpid=pow(param.k, param.dim);

   /*-----------Speicherplatzreservierung fuer die Prozessortabelle---------- */

    param.p =( int* )MALLOC( PC_nProcs * sizeof( int ));
```

```
   /*-----------Speicherplatzreservierung fuer die Prozessorlinks------------ */

   /* Es werden 2*param.dim Links fuer die mesh-Verbindung und zwei Links     */
   /* fuer die chain-Verbindung gehalten;  ein Link ist vom Typ LinkCB_t*     */

    param.link =( LinkCB_t***)MALLOC((param.dim+1)*sizeof( LinkCB_t**));
    for(i=0;i<=param.dim;i++)
        param.link[i] =( LinkCB_t**)MALLOC(2*sizeof( LinkCB_t*));

   /* ------Mapping der virtuellen Prozessoren auf die reale Prozessoren -----*/

     mapping( map, param.p);

   /*--------------- Bestimmung der virtuellen Prozessornummer ---------------*/

     param.vpid = find_vpid( param.p);

   /*-----------------Aufbau der virtuellen Gitter-Topologie------------------*/

     make_mesh( &(param));

   /*-----------------Aufbau der virtuellen Ketten-Topologie------------------*/

   /*         wird mitcompiliert nur wenn die Zeitausgabe gefordert           */

#ifdef TIME
     make_chain( &(param));
#endif

   /*--Der folgende Code wird nur von den beteiligten Prozessoren augefuehrt--*/

     if(param.vpid < param.max_vpid && param.vpid >= 0){


      /*--------------------Partitionierung des Gitters ----------------------*/

      /* Es muessen Partitionen gefunden werden, die im Fall der verteilten   */
      /* Eingabe die input-Daten lesen                                        */

        param.part = partition(param);

      /*---------- Speicherplatzreservierung fuer den Ausgabestring-----------*/

        res_string = (char *)MALLOC((4*param.dim+3)*sizeof(char)*STRLEN);
        tstr =  res_string;                     /* Anfangsadresse des Strings */

      /*----------------------Oeffnen der input-Datei-------------------------*/

        if(in_f < 0)   /* Falls Experimente durchgefuehrt werden, wird keine  */
                       /* input-Datei geoeffnet, die Eingabe erfolgt direkt   */
                       /* in der input-Funktion                               */
```

```
        in_file = (FILE *)NULL;
        else {
              sprintf(s_in, "../custom/in%d", in_f);
              in_file = OPEN_FILE(s_in, "r");
              fscanf(in_file, "%d: ");
       }

       /*----------------------Oeffnen der output-Datei------------------------*/

         out_file =(FILE*) NULL;

       /*---------------------------------------------------------------------
                    DO-WHILE-Schleife fuer die Anzahl der Experimente
        --------------------------------------------------------------------*/
         i = 0;
         do{
            i++;                                        /*  Experimentzaehler  */

           /*---------------------- Lesen der Daten ---------------------------*/

            sk_input(&(res_string), in_file, param, &(in_vec));

           /*-------------------------- divide-Phase --------------------------*/


            if(param.dim)             /* nur fuer den nicht-sequentiellen Fall */
              sk_divide(&(res_string), param, &(in_vec));

           /*-------------------------- basic-Phase--------------------------- */

            if(seq) sk_sequential(&(res_string), param, &(in_vec), &(out_vec));
                                         /* der beste sequentielle Algorithmus */

            else sk_basic(&(res_string), param, &(in_vec), &(out_vec));
                                      /* Simulation mit virtuellen Prozessoren */

           /*-------------------------- combine-Phase--------------------------*/

             if(param.dim)            /* nur fuer den nicht-sequentiellen Fall */
               sk_combine(&(res_string), param, &(out_vec));

           /*-----------------------Schreiben der Daten-----------------------*/

             sk_output(&(res_string), out_file, param, &(out_vec));

           /*--synchrones Schreiben der Zeitmessungen auf eine output-Datei----*/

#ifdef TIME
             write_sync(param, tstr);
             res_string = tstr;
#endif
```

```
            param.n++; /* Nach jedem Experiment Problemgroesse dekrementieren */

       } /* end do */

       while(i < param.exper);

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

     /*------------Schliessen der input- und output-Datei---------------------*/

       fclose(out_file);
       fclose(in_file);

    }/* end if */

   /* ------------------Freigabe der virtuellen Gitter-Topologie -------------*/

       free_mesh(&(param));

   /* ------------------Freigabe der virtuellen Ketten-Topologie -------------*/

#ifdef TIME
       free_chain(&(param));
       free(res_string);
#endif

   /* --------------------------Schpeicherplatzfreigabe-----------------------*/
       free(param.link);
       free(param.p);
} /*endmain*/

/*-----------------------------------------------------------------------------
                                    END
-----------------------------------------------------------------------------*/
```

```
/* ---------------------------------------------------------------------------
 *                              FILE sk_funcdefs.c
 *                              ^^^^^^^^^^^^^^^^^^
 * Funktionsdefinitionen fuer die sk_mesh.c Datei.
 * Diese Datei wird von sk_mesh.c included.
 *
 * -------------------------------------------------------------------------*/

#include "design.c"

/* ---------------------------------------------------------------------------
 *                              SKELETON FUNCTIONS
 * -------------------------------------------------------------------------*/

/* ---------------------------------------------------------------------------
 *  NAME        : mapping()
 *
 *  PARAMETER   : int  m    Nummer der mapping-Datei
 *                int *fp   Zeiger auf die Prozessortabelle
 *
 *  DESCRIPTION : offnet eine Mapping-Datei map# mit der Nummern map,
 *                liesst sie, schreibt ihren Inhalt in die Prozessortabelle p
 *                und schliesst diese Datei.
 * -------------------------------------------------------------------------*/

void mapping(int m, int *fp) {

/* ---------------------------------------------------------------------------
 *  NAME        :  find_vpid()
 *
 *  PARAMETER   :  int *fp   Zeiger auf die Prozessortabelle
 *
 *  DESCRIPTION :  sucht die vpid-Nummer jedes Prozessors und liefert sie
 *                 zur"uck.
 * -------------------------------------------------------------------------*/

long find_vpid(int *fp) {

/* ---------------------------------------------------------------------------
 *  NAME        : partition()
 *
 *  PARAMETER   : INFO fparam
 *
 *  DESCRIPTION : sucht die Prozessoren, auf die die Wurzel des  call graphs
 *                abgebildet wird, und berechnet die Partitionsnummer
 *                (von 0 bis 2^d - 1) , fuer alle anderen Prozessoren -1.
 *                Der berechneter Wert wird zurueckgeliefert.
 * -------------------------------------------------------------------------*/

int partition( INFO fparam ){
```

```
/* ---------------------------------------------------------------------------
 *  NAME        : make_mesh()
 *
 *  PARAMETER   : INFO *fparam
 *
 *  DESCRIPTION : verbindet die Prozessoren in ein n-dimensionales k-Gitter
 *                mit Links, ueber die Nachrichten gesendet und empfangen
 *                werden k"onnen. Entlang einer Dimension d=0, .., n-1 werden
 *                zwei Nachbarn verbunden (d.h. Prozessoren mit vpid und
 *                vpid+k^d). Prozessoren mit vpid % k=0 haben keinen linken
 *                Nachbarn, Prozessoren mit vpid % k=k-1 haben keinen
 *                rechten Nachbarn. Der reale pid = p[vpid].
 * -------------------------------------------------------------------------*/

void make_mesh( INFO *fparam ) {

/* ---------------------------------------------------------------------------
 *  NAME        : make_chain()
 *
 *  PARAMETER   : INFO *fparam
 *
 *  DESCRIPTION : verbindet die Prozessoren mit Links in eine Kette von
 *                vpid=0 bis vpid=k^d - 1.
 * -------------------------------------------------------------------------*/

void make_chain( INFO *fparam ) {

/* ---------------------------------------------------------------------------
 *  NAME        : free_mesh()
 *
 *  PARAMETER   : INFO *fparam
 *
 *  DESCRIPTION : baut die von make_mesh() aufgebaute virtuelle Topologie ab.
 * -------------------------------------------------------------------------*/

void free_mesh( INFO *fparam ) {

/* ---------------------------------------------------------------------------
 *  NAME        :  free_chain()
 *
 *  PARAMETER   :  INFO *fparam
 *
 *  DESCRIPTION :  baut die von make_chain() aufgebaute virtuelle Topologie ab.
 * -------------------------------------------------------------------------*/

void free_chain( INFO *fparam ) {

/* ---------------------------------------------------------------------------
 *  NAME        : write_sync()
 *
 *  PARAMETER   : INFO   param
```

```
 *               char  *time_str  Zeiger auf den Ausgabestring
 *
 *  DESCRIPTION : synchronisiert die Ausgabe von allen Prozessoren.
 *               wenn ein Prozessor eine Nachricht vom linken Nachbarn erhaelt, *                oeffnet er eine Datei mit Deskriptor time_file , schreibt
 *               den Ausgabestring time_str darauf, schliesst sie
 *               und sendet eine Nachricht an den rechten Nachbarn.
 *               Am Anfang der Kette (vpid=0) wird keine Nachricht empfangen,
 *               am Ende (vpid=k^d-1) keine Nachricht gesendet.
 *               Falls nur ein Prozessor aktiv werden keine Nachrichten
 *               ausgetauscht.
 * ---------------------------------------------------------------------------*/

void write_sync( INFO param, char *time_str ) {

/* ----------------------------------------------------------------------------
 *  NAME        : sk_input()
 *
 *  PARAMETER   : char    **time_str  Zeiger auf den Ausgabestring
 *               FILE     *in_file   Eingabedatei-Deskriptor
 *               INFO      fparam    andere Parameter
 *               INVECTOR *fin_vec   Zeiger auf den Eingabevektor
 *
 *  DESCRIPTION : reserviert den Speicherplatz fuer den Eingabevektor,
 *               bestimmt seine Laenge in Abhaengigkeit von der Art der
 *               Eingabe, selektiert Prozessoren, die durch den Aufruf von
 *               input() die Eingabedaten lesen und notiert die
 *               Ausf"uhrungszeiten von input() im Ausgabestring.
 * ---------------------------------------------------------------------------*/

void sk_input( char **time_str, FILE *in_file, INFO fparam, INVECTOR *fin_vec ){

/* ----------------------------------------------------------------------------
 *  NAME        : sk_output()
 *
 *  PARAMETER   : char     **time_str   Zeiger auf den Ausgabestring
 *               FILE      *out_file   Ausgabedatei-Deskriptor
 *               INFO       fparam     andere Parameter
 *               OUTVECTOR *fout_vec   Zeiger auf den Ausgabevektor
 *
 *  DESCRIPTION : selektiert Prozessoren, die durch den Aufruf von output()
 *               die Ausgabedaten schreiben, notiert die Ausf"uhrungszeiten
 *               von output() im Ausgabestring und gibt den Speicherplatz frei
 * ---------------------------------------------------------------------------*/

void
sk_output( char **time_str, FILE *out_file , INFO fparam, OUTVECTOR *fout_vec ){

/* ----------------------------------------------------------------------------
 *  NAME        : sk_divide()
 *
 *  PARAMETER   : char    **time_str
```

```
 *                INFO      param
 *                INVECTOR *in_vec
 *
 *  DESCRIPTION : implementiert das absteigende Kommunikationsmuster, verteilt
 *                die parallelen Komponenten des Eingabevektors auf
 *                die Prozessoren und ruft die divide()-Funktion auf.
 *                Diese Funktion entspricht der ersten vernesteten for-Schleife  *                im Schleifenprogramm und simuliert die Rekursion
 *                im funktionalen Programm.
 * -------------------------------------------------------------------------*/

void sk_divide( char **time_str, INFO param, INVECTOR *in_vec){

int       pos, leftpos, rightpos, time, timecom, timediv, addr=0;
long      offset, d;
INTYPE   *ptr;
PATTERN   cpat = CD, dpat = D;

  if(param.cent_in)
     UNDIV_V = (INTYPE *)MALLOC( DIV_V_SIZE*sizeof(INTYPE) );

  else  {
         UNDIV_V_SIZE = DIV_V_SIZE << 1;
         UNDIV_V = (INTYPE *)MALLOC( UNDIV_V_SIZE *sizeof(INTYPE) );
        }

  rightpos = RIGHTPOS(param.k);
  leftpos = LEFTPOS(param.k);

  offset=pow(param.k, param.dim-1);

/****************************************************************************/

  for(d = param.dim-1; d >= 0; d--){

      pos = COORDINATE(param.vpid,offset,param.k);

      if(param.cent_in) {
         ptr = UNDIV_V;
         UNDIV_V = DIV_V;
         DIV_V = ptr;
         UNDIV_V_SIZE = DIV_V_SIZE;
         DIV_V_SIZE = UNDIV_V_SIZE >> 1;
      }

      /**********************************************************************/

      if(pos == leftpos && !param.cent_in){

         S_TIME(time);                            /*Beginn der Zeitmessung */

         SEND_TO_RIGHT(d, DIV_V, DIV_V_SIZE*sizeof(INTYPE));
```

```
   RECV_FROM_RIGHT(d, &UNDIV_V[DIV_V_SIZE], DIV_V_SIZE*sizeof(INTYPE));

   memcpy(UNDIV_V, DIV_V , DIV_V_SIZE*sizeof(INTYPE));

   if( param.k > 3)
      SEND_TO_LEFT(d, UNDIV_V, UNDIV_V_SIZE*sizeof(INTYPE));

   S_TIME(timecom);

   divide( pos, in_vec);
}

/**********************************************************************/

else if(pos == rightpos){

       S_TIME(time);                         /*Beginn der Zeitmessung */


       if(!param.cent_in){
          RECV_FROM_LEFT(d, UNDIV_V, DIV_V_SIZE*sizeof(INTYPE));
          SEND_TO_LEFT(d, DIV_V, DIV_V_SIZE*sizeof(INTYPE));

          memcpy(&UNDIV_V[DIV_V_SIZE], DIV_V , sizeof(INTYPE)*DIV_V_SIZE);
        }
        else {
              SEND_TO_LEFT(d, UNDIV_V, UNDIV_V_SIZE*sizeof(INTYPE));
        }

        if( param.k > 2)
           SEND_TO_RIGHT(d, UNDIV_V, UNDIV_V_SIZE*sizeof(INTYPE));

        S_TIME(timecom);

        divide( pos, in_vec);
}

/**********************************************************************/

else if( pos < leftpos && !param.cent_in ||
                                       pos < rightpos && param.cent_in ){

        S_TIME(time);                         /*Beginn der Zeitmessung */

        RECV_FROM_RIGHT(d, UNDIV_V, UNDIV_V_SIZE*sizeof(INTYPE));

        if( pos !=0 )
           SEND_TO_LEFT(d, UNDIV_V, UNDIV_V_SIZE*sizeof(INTYPE));

        S_TIME(timecom);
```

```
                divide( pos, in_vec);
        }

        /*********************************************************************/

        else if(pos > rightpos){

                S_TIME(time);                       /*Beginn der Zeitmessung */

                RECV_FROM_LEFT(d, UNDIV_V, UNDIV_V_SIZE*sizeof(INTYPE));

                if( pos != param.k-1)
                   SEND_TO_RIGHT(d, UNDIV_V, UNDIV_V_SIZE*sizeof(INTYPE));

                S_TIME(timecom);

                divide( pos, in_vec);
        }

        /*********************************************************************/

        else ERROR("Error in sk_divide: pos invalid");

        S_TIME(timediv);

        offset/=param.k;

#ifdef TIME
     addr= sprintf(*time_str, "p:%4d n:%2d s%d p%d %d:%8.4f \n",
                    param.vpid, param.n, d, pos, cpat, SECONDS(time, timecom));
     (*time_str) += addr;

     addr= sprintf(*time_str,"p:%4d n:%2d s%d p%d %d:%8.4f \n",
                 param.vpid, param.n, d, pos, dpat, SECONDS(timecom, timediv));
     (*time_str) += addr;
#endif

  } /* endfor */

  free(UNDIV_V);

} /* end sk_divide */

/* ---------------------------------------------------------------------------
 *  NAME       : sk_sequential()
 *
 *  PARAMETER   : char     **time_str
 *                INFO       param
 *                INVECTOR  *in_vec
 *                OUTVECTOR *out_vec
 *
```

```
 *  DESCRIPTION :  bietet die M"oglichkeit an, die sequentielle Komponente
 *                 der Problem-Vektoren nicht mit virtuellen Prozessoren,
 *                 sondern mit dem besten sequentiellen Algorithmus zu l"osen.
 *                 Der beste sequentielle Alorithmus wird in sequential()
 *                 implementiert. Die Funktion sk_sequential() nimmt einen
 *                 Vektor als Parameter auf, reserviert den Speicherplatz
 *                 fuer den Ergebnisvektor von sequential(), ruft sequential()
 *                 auf und nimmt die Zeitmessungen vor.
 * ---------------------------------------------------------------------------*/

void
sk_sequential(char **time_str,INFO param, INVECTOR *in_vec, OUTVECTOR *out_vec){

/* ----------------------------------------------------------------------------
 *  NAME        : sk_basic()
 *
 *  PARAMETER   : char     **time_str
 *                INFO       param
 *                INVECTOR  *in_vec
 *                OUTVECTOR *out_vec
 *
 *  DESCRIPTION :  implementiert die Idee mit virtuellen Prozessoren,
 *                 sie entspricht der zweiten vernesteten for-Schleife im
 *                 Schleifenprogram.
 *                 Die sk_basic()-Funktion besteht aus drei Teilen fuer drei
 *                 Bearbeitungsphasen: divide, basic und combine.
 *                 Die Bearbeitung der drei Phasen unterscheidet sich von den
 *                 entsprechenden parallelen Phasen damit, dass die Probleme
 *                 nicht auf die Prozessoren verteilt werden, sondern virtuelle
 *                 Prozessoren als Array-Elemente simuliert werden.
 *                 Die Komunikation zwischen den virtuellen Prozessoren heisst
 *                 lokal, da sie nur auf einem realen Prozessor als
 *                 Speicherzugriffe (Zugriffe auf die Array-Elemente)
 *                 stattfindet.
 * ---------------------------------------------------------------------------*/

void sk_basic(char **time_str, INFO param, INVECTOR *in_vec, OUTVECTOR *out_vec)

/* ----------------------------------------------------------------------------
 *  NAME        : sk_combine()
 *
 *  PARAMETER   : char     **time_str
 *                INFO       param
 *                OUTVECTOR *out_vec
 *
 *  DESCRIPTION : implementiert das aufsteigende Kommunikationsmuster, setzt
 *                die parallelen Komponenten des Ausgabevektors zu einem
 *                Ergebnisvektor zusammen und ruft die combine()-Funktion auf.
 *                Diese Funktion entspricht der dritten vernesteten for-Schleife *                im Schleifenprogramm und simuliert die Rueckwaertsrekursion im *                funktionalen Programm.
 * ---------------------------------------------------------------------------*/
```

```
void sk_combine(char **time_str, INFO param, OUTVECTOR *out_vec){

int      pos, leftpos, rightpos, time, timecom, timec, addr = 0 ;
long     offset, d ;
OUTTYPE *ptr;
PATTERN  cpat = C, ccpat = CC;

  rightpos = RIGHTPOS(param.k);
  leftpos = LEFTPOS(param.k);

  if(param.cent_out){

     UNCOM_V =
      (OUTTYPE *)MALLOC(sizeof(OUTTYPE)*(COM_V_SIZE* param.k) << (param.dim-1));
     ptr = COM_V;
     COM_V = (OUTTYPE *)MALLOC(sizeof(OUTTYPE)*(COM_V_SIZE << param.dim));
     memcpy(COM_V, ptr, sizeof(OUTTYPE)*COM_V_SIZE);
     free(ptr);
  }
  else{
       UNCOM_V = (OUTTYPE *)MALLOC(sizeof(OUTTYPE)*(COM_V_SIZE*param.k));
  }

  offset=1;

  /*****************************************************************************/

  for(d=0; d < param.dim;d++){

      pos = COORDINATE(param.vpid,offset,param.k);

      UNCOM_V_SIZE = COM_V_SIZE*param.k;

      memcpy(&UNCOM_V[pos*COM_V_SIZE], COM_V ,sizeof(OUTTYPE)*COM_V_SIZE);

      /*************************************************************************/

      if(pos > rightpos) {

        S_TIME(time);                         /*Beginn der Zeitmessung */

        if(pos != param.k-1 )
           RECV_FROM_RIGHT(d, &UNCOM_V[COM_V_SIZE*(pos+1)] ,
                                   COM_V_SIZE*(param.k-1-pos)*sizeof(OUTTYPE));

           SEND_TO_LEFT(d, &UNCOM_V[COM_V_SIZE*pos] ,
                                   COM_V_SIZE*(param.k-pos)*sizeof(OUTTYPE));

           S_TIME(timecom);
      }
```

```
/*********************************************************************/

else if( pos < leftpos && !param.cent_out ||
                                      pos < rightpos && param.cent_out ){

       S_TIME(time);                         /*Beginn der Zeitmessung */

       if(pos != 0)
          RECV_FROM_LEFT(d, UNCOM_V, pos*COM_V_SIZE*sizeof(OUTTYPE));

       SEND_TO_RIGHT(d,  UNCOM_V , COM_V_SIZE*(pos+1)*sizeof(OUTTYPE));

       S_TIME(timecom);
}

/*********************************************************************/

else if(pos == leftpos && !param.cent_out){

        S_TIME(time);                         /*Beginn der Zeitmessung */

        if( param.k > 3)
           RECV_FROM_LEFT(d, UNCOM_V , pos*COM_V_SIZE*sizeof(OUTTYPE));

        SEND_TO_RIGHT(d, UNCOM_V , COM_V_SIZE*(pos+1)*sizeof(OUTTYPE));
        RECV_FROM_RIGHT(d, &UNCOM_V[(pos+1)*COM_V_SIZE],
                           COM_V_SIZE*(param.k-1-pos)*sizeof(OUTTYPE));

        S_TIME(timecom);

        combine(pos, out_vec);

        S_TIME(timec);

#ifdef TIME
            addr = sprintf( *time_str,"p:%4d n:%2d s%d p%d %d:%8.4f \n",
                    param.vpid, param.n, d, pos, cpat, SECONDS(timecom, timec));
            (*time_str) += addr;
#endif
      }

/*********************************************************************/

else if(pos == rightpos){

        S_TIME(time);                         /*Beginn der Zeitmessung */

        if(param.k > 2)
           RECV_FROM_RIGHT(d, &UNCOM_V[COM_V_SIZE*(pos+1)],
                            COM_V_SIZE*(param.k-1-pos)*sizeof(OUTTYPE));
```

```
              RECV_FROM_LEFT(d, UNCOM_V, pos*COM_V_SIZE*sizeof(OUTTYPE));

              if(!param.cent_out)
                SEND_TO_LEFT(d, &UNCOM_V[pos*COM_V_SIZE],
                                    COM_V_SIZE*(param.k-pos)*sizeof(OUTTYPE));

              if(param.cent_out)
                 COM_V_SIZE <<= 1;

              S_TIME(timecom);

              combine((param.cent_out ? -1: pos), out_vec);

              S_TIME(timec);
#ifdef TIME
             addr = sprintf( *time_str,"p:%4d n:%2d s%d p%d %d:%8.4f \n",
                    param.vpid, param.n, d, pos, cpat, SECONDS(timecom, timec));
   (*time_str) += addr;
#endif
      }

      /*********************************************************************/

      offset*=param.k;

#ifdef TIME
      addr = sprintf( *time_str,"p:%4d n:%2d s%d p%d %d:%8.4f \n",
                    param.vpid, param.n, d, pos, ccpat, SECONDS(time, timecom));
      (*time_str) += addr;
#endif
  } /* endfor  */

  free(UNCOM_V);

} /* end sk_combine */

/* ---------------------------------------------------------------------------
 *                                     END
 * ---------------------------------------------------------------------------*/
```

```
/*-----------------------------------------------------------------------------

   customizing functions fuer das Polynomprodukt mit k=3 (Karatsuba-Verfahren)

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

#include "../src/sk.h"

/*******************************************************************************
INPUT( case x: actions; break;)       x = 0 : fuer Experimente
                                      x = 1 : fuer Lesen von IN_FILE
*******************************************************************************/

INPUT(case 0: IN_ITER( (IN_V.a = 1,
                        IN_V.b = 1 )); break;
      case 1:
              OVERREAD( PARTITION * INITERLEN );

              IN_ITER( SCAN_IN( &IN_V.a ) );:
              if(IN_DIST)
                 OVERREAD( ((1 << DIM)-1) * INITERLEN );

              IN_ITER( SCAN_IN( &IN_V.b ) ); break; )


/********************************************************************************
OUTPUT(actions;)   schreibt die Ausgabe auf OUT_FILE
*******************************************************************************/

OUTPUT( OUT_ITER( PRINT_OUT( OUT_V.h) );
        fprintf(OUT_FILE, " \n" );
        OUT_ITER( PRINT_OUT( OUT_V.l) );
        fprintf(OUT_FILE, " \n" );
        fflush(stdout);)

/********************************************************************************
BASIC(actions;) END loest das einfache Problem
Argument: BDIV.Komponente
Ergebnis: BCOM.Komponente
*******************************************************************************/

BASIC( BCOM.h = 0;
       BCOM.l = BDIV.a * BDIV.b;)

/********************************************************************************
DIVIDE( case x: actions; ) teilt den Eingabevektor
Argument: UNDIV(n)
Ergebnis: DIV(n)
COPY_L := DIV(0) = UNDIV(0);
COPY_R := DIV(0) = UNDIV(1);
*******************************************************************************/
```

```
DIVIDE( case 0: COPY_L; break;
        case 1: DIV_ITER( (DIV(0).a = UNDIV(0).a + UNDIV(1).a,
                           DIV(0).b = UNDIV(0).b + UNDIV(1).b) ); break;
        case 2: COPY_R; break; )

/********************************************************************************
COMBINE( case x: actions; ) setzt den Ausgabevektor aus Teilvektoren zusammen
Argument: UNCOM(n)
Ergebnis: COM(n)
********************************************************************************/

COMBINE( case  LPOS:
          COM_DIST_ITER( (COM(0).h = UNCOM(2).h,
                          COM(0).l = UNCOM(2).h +
                                     UNCOM(1).l - UNCOM(0).l - UNCOM(2).l) );
               break;
         case  RPOS:
          COM_DIST_ITER( (COM(0).l = UNCOM(0).l,
                          COM(0).h = UNCOM(2).l +
                                     UNCOM(1).h - UNCOM(0).h - UNCOM(2).h) );
               break;
         case CENT:
          COM_CENT_ITER( (COM(1).l = UNCOM(0).l,

                          COM(0).l = UNCOM(0).h +
                                     UNCOM(1).l - UNCOM(0).l - UNCOM(2).l,
                          COM(0).h = UNCOM(2).h,
                          COM(1).h = UNCOM(2).l +
                                     UNCOM(1).h - UNCOM(0).h - UNCOM(2).h ));
               break; )

/*------------------------------------------------------------------------------
                                       END
------------------------------------------------------------------------------*/
```

# Literaturverzeichnis


[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[Brä93] T. Bräunl. *Parallele Programmierung: eine Einführung.* Vieweg, 1993.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*, chapter 3, pages 41–69. Pitman, 1989.

[GL95] S. Gorlatch and C. Lengauer. Parallelization of divide-and-conquer in the Bird-Meertens formalism. *Formal Aspects of Computing*, 7(6):663–682, 1995.

[Gri91] D. Gries. *The Science of Programming.* Texts and Monographs in Computer Science. Springer-Verlag, 1991.

[HJW92] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language haskell (version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[HL95] C. Herrmann and C. Lengauer. Notes on the space-time mapping of divide-and-conquer recursion. *In Proc. GI/ITG FG PARS 95*, 1995.

[HS78] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Pitman-Press, 1978.

[KO62] A. Karacuba and J. Ofman. Multiplikation vielstelliger Zahlen mit Rechenautomaten. *Akademija Nauk SSSR Moskva: Doklady Akademii Nauk SSSR*, 1,1(145):293–294, 1962.

[LD91] H. L. Lewis and L. Denenberg. *Data Structures and Their Algorithms.* Harper Collins, 1991.

[Lew91] T. G. Lewis. Data parallel computing: An alternative for the 1990s. *IEEE Computer*, 24(2):110–111, 1991.

[LRG+90] V.M. Lo, S. Rajopadhye, S. Gupta, D. Keldsen, M. Mohamed, and J. Telle. Mapping divide-and-conquer algorithms to parallel architectures. In *International Conference on Parallel Architectures*, pages 128–135, 1990.

[MCH92] Z. G. Mou, C. Constantinescu, and H. Hickey. Optimal mappings of divide-and-conquer algorithms to mesh connected parallel architectures. In *Proc. 1992 Int. Computer Symp.*, pages 273–284. Feng Chia University, Taiwan, December 1992.

[Mis94] J. Misra. Powerlist: A structure for parallel recursion. *ACM TOPLAS*, 16(6):1738–1767, 1994.

[Mou89] Z. G. Mou. *A Formal Model for Divide-and-Conquer and Its Parallel Realization.* PhD thesis, Department of Computer Science, Yale University, 1989. Technical Report YALEU/DCS/RR-795.

[OW93] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen.* BI-Wiss.-Verl., 1993.

[Qui88] Michael J. Quinn. *Algorithmenbau und Parallelcomputer.* McGraw-Hill, 1988.

[Ski90] D. B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, 1990.

[Smi87] D. R. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, 8:213–229, 1987.