# Latenzzeitverbergung
# in datenparallelen Sprachen

Zur Erlangung des akademischen Grades eines

**Doktors der Ingenieurwissenschaften**

der Fakultät für Informatik

der Universität Karlsruhe (Technische Hochschule)

genehmigte

## Dissertation

von

**Matthias M. Müller**

aus Weingarten, Württemberg

| | |
|---|---|
| Tag der mündlichen Prüfung: | 2. Februar 2000 |
| Erster Gutachter: | Prof. Dr. Walter F. Tichy |
| Zweiter Gutachter: | Prof. Dr. Theo Ungerer |

## Zusammenfassung

Das ungünstige Verhältnis von Kommunikations- zu Rechenleistung fast aller Parallelrechner, das sich in Kommunikationslatenzzeiten von mehreren hundert bis tausend Prozessortaktzyklen manifestiert, verhindert in vielen Fällen die effiziente Ausführung von kommunikationsintensiven feingranularen datenparallelen Programmen.
Zur Lösung dieses Problems untersucht diese Arbeit Techniken zur Latenzzeitverbergung, die durch Vorladeoperationen die Kommunikationszeit des Netzwerkes verdecken. Der vorgeschlagene Ansatz *VSCAP* (Software Controlled Access Pipelining with Vector commands) erweitert bestehende Techniken um Vektorbefehle und kann die anfallenden Latenzzeiten für eine große Anzahl von Anwendungen fast vollständig verbergen.
Meine Beiträge sind:

- Modellierung von *VSCAP*, einer Erweiterung von SCAP mit Vektorbefehlen.
- Entwurf von Konzepten, mit denen Kommunikationsaufträge in datenparallelen Programmen in Datenfließbänder des VSCAP-Verfahrens überführt werden können.
- Implementierung dieser Konzepte und Integration in den Prototypübersetzer KarHPFn.

Die Leistungen von *VSCAP* bei der Latenzzeitverbergung wurden durch Modellierung und Laufzeittests von 25 Programmen, darunter 3 kompletten Anwendungen, untersucht. Die Ergebnisse sind:

- Nachweis der praktischen Einsetzbarkeit von VSCAP (und damit als Spezialfall auch SCAP) auf einem realen Rechner.
- Berechnung des Grades der Latenzzeitverbergung von VSCAP und Bestätigung der Modellierung durch automatisch generierte Programme.
- Bestätigung der Beschleunigung von VSCAP gegenüber SCAP um einen Faktor gleich der Vektorlänge $L$ durch Modellierung und Messungen.
- Erster Übersetzer auf Parallelrechnerarchitekturen mit gemeinsamem Adreßraum, der zur Kommunikation nur Vorladeoperationen einsetzt.
- Nachweis der automatischen, für den Programmierer transparenten und effizienten Übersetzung von datenparallelen Applikationen in Programme, die zur Kommunikation das VSCAP-Verfahren anwenden, am Beispiel von HPF.
- Vergleichbare Leistung von KarHPFn-generiertem VSCAP und der hochoptimierten Kommunikationsbibliothek auf der Cray T3E, bei dynamischen Kommunikationsmustern sogar ein mehr als 6-facher Laufzeitgewinn von VSCAP.
- 3- bis mehr als 5-facher Laufzeitgewinn von KarHPFn-generiertem VSCAP gegenüber Portland Group HPF beim Test von drei Applikationen (Veltran, FIRE und PDE1) auf bis zu 128 Prozessoren mit identischen HPF-Quellen, bei Programmen mit großem Kommunikationsaufwand sogar mehr als ein Faktor 15.

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

## 1.1 Motivation

In fast allen kommerziell erhältlichen Parallelrechnern werden heutzutage Standardprozessoren eingesetzt, da diese gegenüber Spezialprozessoren ein besseres Preis-Leistungs-Verhältnis und eine kürzere Weiterentwicklungszeit bieten. Die Uniprozessorarchitektur wird um eine Schnittstelle erweitert, welche die Kommunikation zwischen den einzelnen Prozessoren realisiert. Bei einem Gleichgewicht von Rechen- und Kommunikationsleistung können dann Parallelrechner vom stetigen Leistungszuwachs der einzelnen Knoten profitieren. Dieses Gleichgewicht ist jedoch in Richtung der Rechenleistung verschoben und manifestiert sich in Kommunikationslatenzzeiten von mehreren tausend Prozessortaktzyklen bei fast allen kommerziell erhältlichen Parallelrechnern.
Das Mißverhältnis wird sich auch in den nächsten Jahren noch verstärken, da sich die Kommunikationsleistung der Netzwerke nicht im selben Maße wie die Prozessorleistung weiterentwickelt. Erst optoelektronische Netzwerke scheinen die Schere zwischen Rechen- und Kommunikationsleistung zu schließen, da sie mit ihren hohen Taktraten den Durchsatz des Netzwerkes wesentlich erhöhen. Doch werden auch hier in großen Systemen bedingt durch die Lichtgeschwindigkeit und die Schnittstellenlatenzen der einzelnen Hardwarebausteine Latenzzeiten von hundert und mehr Prozessortaktzyklen die Regel sein, welche die Rechenleistung beeinträchtigen.
Um trotzdem von der hohen Rechenleistung heutiger Standardprozessoren in der Parallelverarbeitung profitieren zu können, müssen anfallende Latenzzeiten verdeckt werden, d.h. während das Netzwerk eine Kommunikationsanforderung des Prozessors bearbeitet, führt dieser weitergehende Berechnungen aus, die zur Lösung des Problems beitragen. Latenzzeitverbergende Mechanismen stehen dabei am Ende der Optimierungskette, die durch hardwareseitige Latenzzeitreduktion und softwareseitige Lokalitätsoptimierungen geprägt ist.

## 1.2 Ansatz

Der hier verfolgte Ansatz zur Latenzzeitverbergung verdeckt anfallende Kommunikationszeiten nicht nur mit überlappender Berechnung, sondern es werden auch einzelne Kommunikationsoperationen untereinander überlappt ausgeführt. Das zugrundeliegende Prinzip ist dabei recht einfach: Nicht-lokale Datenelemente werden vor der eigentlichen Verwendung gezielt vorgeladen, um so die Verzögerungszeit des Kommunikationsnetzwerkes beim Zugriff

auf nicht-lokale Datenelemente zu verbergen. Die so gewonnenen Datenfließbänder nennt Warschko *SCAP*: *Software Controlled Access Pipelining* [120].
Er beschreibt neben den Netzwerkanforderungen die Programmtransformation, mit der datenparallele Schleifen in Fließbänder des SCAP-Verfahrens überführt werden können. In einer simulierten Prozessorumgebung bestätigte er die analytischen Modellierungen, die je nach Kommunikationsmuster eine 90%-ige Verdeckung der Latenzzeit erwarten ließen. Als weitere Arbeiten auf diesem Feld nennt Warschko drei Bereiche:

1. Den Entwurf und die Realisierung einer Rechnerarchitektur, auf der das SCAP-Verfahren angewendet werden kann

2. Die Validation der SCAP-Simulationsergebnisse auf einem realen Parallelrechner

3. Die Integration des SCAP-Verfahrens in einen optimierenden Übersetzer

Der erste Punkt wurde bereits von den Ingenieuren von SGI Cray mit dem Bau der T3E erledigt, wobei dort allerdings eine von SCAP unabhängige Zielsetzung verfolgt wurde. Die Einsatzmöglichkeit des SCAP-Modells auf einem realen Parallelrechner wurde mit Hilfe von handgeschriebenen Programmen gezeigt [94]. Die dort gewonnenen Ergebnisse bestätigten das SCAP-Modell und motivierten die Studie von Vektorbefehlen (*Vektor-SCAP* oder kurz *VSCAP*), mit denen $L$ nicht-lokale Kommunikationsanforderungen vom Prozessor auf einmal abgesetzt werden können. Damit wird das SCAP-Verfahren zum Spezialfall von VSCAP, wenn die dort benützten Vektoren die Länge eins haben.
Die Resultate aus [94] werden hier, obwohl sie ein Teil dieser Arbeit waren, nicht weiter aufgeführt, denn sie sind nur eine kleine Teilmenge der in Kapitel 5 aufgeführten Ergebnisse. Der dritte Punkt, die Integration des SCAP-Verfahrens, wird dahingehend erweitert, daß nicht das SCAP-Verfahren, sondern die hier untersuchte Obermenge VSCAP in einen optimierenden Übersetzer eingebaut wird.

## 1.3 Ziel der Dissertation

Die Dissertation beschäftigt sich mit zwei Fragestellungen. Die erste untersucht die Gültigkeit des VSCAP-Modells auf einem realen Rechner. Dabei soll folgendes gezeigt werden:

**These 1**
*Mit dem maschinenunabhängigen analytischen Netzwerkmodell von VSCAP kann in einer gegebenen Parallelrechnerarchitektur der Kommunikationsaufwand einer datenparallelen Schleife abgeschätzt werden.*

Dazu wird die Arbeitsweise von VSCAP modelliert und mit Laufzeitmessungen auf der Cray T3E verglichen. Dieser Vergleich beinhaltet nicht nur den Nachweis der Korrektheit des Modells, sondern es werden auch die Modellaussagen über den möglichen Grad der Latenzzeitverbergung von VSCAP und der berechnete Vorteil der Vektorbefehle gegenüber den einelementigen Operationen von SCAP bestätigt.
Die zweite Fragestellung beschäftigt sich mit der Integration des VSCAP-Verfahrens in einen optimierenden Übersetzer.

**These 2**
*Die Transformation einer datenparallelen Schleife in ein softwaregesteuertes Datenfließband des VSCAP-Verfahrens kann durch einen Übersetzer geschehen. Mit dieser Programmtransformation können die Kommunikationskosten einer datenparallelen Anwendung ohne zusätzliche Intervention des Programmierers gesenkt werden.*

Die Transparenz des VSCAP-Verfahrens soll so weit erreicht werden, daß der Programmierer kein Wissen über die Kommunikationstechnik besitzen muß, um sie effektiv einsetzen zu können. Dazu wurden ein prototypischer HPF-Übersetzer namens KarHPFn implementiert und die Laufzeit der damit übersetzten Programme mit der T3E-eigenen hochoptimierten Kommunikationsbibliothek und dem kommerziellen HPF-Übersetzer von Portland Group verglichen.

## 1.4 Gliederung

Direkt im Anschluß stellt Kapitel 2 die Grundlagen und das SCAP-Modell vor. Es schließt mit der Abgrenzung von SCAP (und damit auch VSCAP) zu bereits existierenden Arbeiten über Latenzzeitverbergung.
In Kapitel 3 wird das VSCAP-Modell vorgestellt. Neben der prinzipiellen Arbeitsweise werden einige Fließbandstrategien besprochen, die sich aus dem unterschiedlichen Einsatz der Vektorbefehle ergeben. Die Analyse der Systemlaufzeiten gibt einen Überblick über zu erwartende Latenzzeitreduktionen. Die Berechnung der optimalen Vektorlänge beschließt die Betrachtungen.
Die Eingliederung von VSCAP in einen Übersetzer behandelt Kapitel 4. Neben allgemeinen Transformationsschemata werden auch spezifische Aspekte der VSCAP-Implementierung auf der T3E besprochen, die in eine Übersicht über den KarHPFn-Übersetzer münden.
Kapitel 5 widmet sich schließlich dem Vergleich der automatisch übersetzten VSCAP-Programme mit der T3E-eigenen Kommunikationsbibliothek und dem HPF-Übersetzer von Portland Group. Dabei werden auch die Kommunikationszeiten der generierten Programme mit den Abschätzungen des VSCAP-Modells verglichen.
Kapitel 6 schließt die Dissertation mit einer Zusammenfassung der erreichten Ergebnisse und bietet Ausblicke auf weitere Einsatzmöglichkeiten von VSCAP.

# Kapitel 2

# Grundlagen und Vorbetrachtungen

Im Grundlagenteil dieses Kapitels (Abschnitt 2.1) werden die Zusammenhänge zwischen Parallelrechnerarchitektur und Programmiertechnik dargestellt. Die Vorbetrachtungen erklären das SCAP-Modell von Warschko (Abschnitt 2.2). Dieser Abschnitt vermittelt die Konzepte von SCAP und bildet die Grundlage für das Verständnis der weiteren Arbeit. Der letzte Teil (Abschnitt 2.3) stellt alternative Techniken zur Latenzzeitverbergung vor.

## 2.1 Grundlagen der Parallelverarbeitung

Die Vorstellung der Parallelrechnerarchitekturen und die Unterscheidung der Programmiermodelle nach Parallelitätsmodell und Rechnerarchitektur ist aus [120].

### 2.1.1 Parallelrechnerarchitekturen

Die Einteilung der Parallelrechnerarchitekturen richtet sich nach der logischen Struktur des Speichersubsystems (verteilter oder gemeinsamer Speicher), der Definition des Adreßraums (global oder prozessorlokal) sowie nach der Kommunikationsstruktur zwischen Prozessor und Speichersubsystem (implizite oder explizite Kommunikation).
Für ein System mit $P$ Prozessoren lassen sich mit Hilfe dieser Kriterien die drei gebräuchlichsten Architekturmodelle definieren:

**Architekturen mit verteiltem Speicher** (Nachrichtenkopplung, *Message Passing*) sind durch einen physikalisch verteilten Speicher, $P$ prozessorlokale Adreßräume sowie explizite Kommunikationsoperationen, die nicht auf den Speicher anderer Prozessoren zugreifen können, gekennzeichnet. Bei den Kommunikationsoperationen handelt es sich um Sende- und Empfangsoperationen, die nach dem Rendevousprinzip nachrichtengekoppelter Architekturen arbeiten. Im Kommunikationsfall müssen dabei beide beteiligte Prozessoren aktiv werden (*zweiseitige Kommunikation*).

Zu den Vertretern dieses Architekturprinzips zählen z. B. IBM-SP2 [5] sowie Workstationfarmen wie ParaStation [120] oder Berkeley NOW.

**Architekturen mit gemeinsamem Adreßraum** (*Shared Address Space*) sind durch einen physikalisch verteilten Speicher, einen globalen Adreßraum sowie explizite Kommunikationsoperationen, die im Gegensatz zu Zugriffen auf den prozessoreigenen Speicher auf

den Speicher anderer Prozessoren zugreifen, charakterisiert. Die Kommunikationsoperationen sind Schreib- und Lesezugriffe, wobei das Rendevousprinzip der Systeme mit verteiltem Speicher entfällt (*einseitige Kommunikation*).

Zu den Vertretern dieses Architekturprinzips zählen alle SIMD-Rechner sowie Cray T3D und T3E [96, 75, 97].

**Architekturen mit gemeinsamem Speicher** (Speicherkopplung, *Shared Memory*) stellen einer Anwendung *eine* Sicht auf den globalen Speicher zur Verfügung. Daß dabei der Speicher physikalisch verteilt sein kann, bleibt vor der Anwendung verborgen. Charakteristisch für diese Rechnerarchitektur ist der globale Adreßraum sowie *implizite* Kommunikationsoperationen, die über das Verbindungsnetzwerk transparent auf alle Speicherstellen zugreifen. Die Unterscheidung zwischen lokalem und entferntem Speicher wird dabei allein anhand der Speicheradresse getroffen. Bei Architekturen mit gemeinsamem Speicher wird zusätzlich das Speicherkonsistenzmodell als Unterscheidungsmerkmal herangezogen, wodurch sich eine weitere Unterteilung in Architekturen mit und ohne Speicherkonsistenz ergibt. In dieser Arbeit werden jedoch nur Architekturen mit konsistenten gemeinsamen Speichern betrachtet (SMPs[1] und CC-NUMA[2] Architekturen).

Zu den Vertretern des Architekturprinzips mit Konsistenzhaltung gehören z. B. alle cache-kohärenten, busbasierten Multiprozessoren der verschiedensten Hersteller (DEC, Sun, SGI, IBM usw.).

### 2.1.2 Modelle der Parallelität

Es gibt zwei prinzipielle Modelle, mit denen Parallelität spezifiziert werden kann:

**Prozeßparallelität** basiert auf dem Prinzip der kommunizierenden sequentiellen Prozesse [64]. Die Charakteristika dieses Modells sind die unabhängigen und im allgemeinen verschiedenen sequentiellen Prozesse, die *asynchron* zueinander laufen. Die Synchronisation und damit die zeitliche Ordnung der beteiligten Prozesse geschieht entweder implizit über den Austausch von Nachrichten oder explizit durch das Einfügen von Synchronisationsprimitiven.

**Datenparallelität** [62] ist durch *synchron* arbeitende Prozessoren charakterisiert, die ein und dasselbe Programm synchron auf verschiedenen Daten abarbeiten. Die beteiligten Programminstanzen müssen vom Programmierer nicht explizit synchronisiert werden, da die synchrone Abarbeitung bereits vom Modell garantiert wird.

Die modellinhärente Synchronität und die aus ihr resultierende Datenkonsistenz machen die Datenparallelität zum gebräuchlicheren Parallelitätsmodell in der Algorithmentechnik.

### 2.1.3 Programmiermodelle

Die Aufgabe eines Programmiermodells ist es, die Annahmen eines Parallelitätsmodells mit den gegebenen, eventuell konträren Eigenschaften einer Parallelrechnerarchitektur zu verbinden.

[1] Symmetric Multiprocessors
[2] Coherent-cache nonuniform memory-access

Tabelle 2.1: Programmiermodelle

| **Architektur- / Parallelitätsmodell** | **verteilter Speicher** | **gemeinsamer Adreßraum** | **gemeinsamer Speicher** |
|---|---|---|---|
| **Prozeßparallelität (asynchron)** | Nachrichtenkopplung | Cray-Modell | gemeinsame Variable |
| **Datenparallelität (synchron)** | SPMD | Datenparallel SIMD | PRAM |

Tabelle 2.1 zeigt die aus der Kombination von Parallelitäts- und Architekturmodell resultierenden Programmiermodelle. Das PRAM-Programmiermodell besitzt in der Theorie und der Algorithmentechnik eine weite Verbreitung, da es neben der Synchronität der Operationen auch einen gemeinsamen Speicher voraussetzt, auf den in $O(1)$ zugegriffen werden kann und der explizite Datenverteilungen überflüssig macht. Das PRAM-Programmiermodell findet sich ansatzweise in einigen Programmiersprachen wieder, wie z. B. in HPF und dessen Varianten sowie in C$^\star$ und in Modula-2$^\star$ [99].
Um Datenparallelität auf existierenden Architekturen zu ermöglichen, müssen datenparallele Applikation auf die verschiedenen Parallelrechner abgebildet werden. Im folgenden sollen die Programmiertechniken der drei vorgestellten Architekturmodelle sowie ihre Vor- und Nachteile erklärt werden.

### 2.1.4 Programmierung von Architekturen mit gemeinsamem Adreßraum

Nach der Einteilung von 2.1.1 fallen in die Klasse der Architekturen mit gemeinsamem Adreßraum hauptsächlich SIMD-Rechner. In ihnen arbeiten Tausende von schwachen Spezialprozessoren im Gleichschritt den selben Befehlsstrom des zentralen Kontrollprozessors ab, wenden ihn aber auf jeweils prozessoreigene Daten an. Dabei emuliert jeder Prozessor $V = \frac{N}{P}$ logische Prozessoren, die sich aus der Verteilung der $N$ unabhängigen Datensätze auf die $P$ physikalischen Prozessoren ergibt. Die Bereitstellung der $V$ Prozessoren auf jedem Prozessor wird *Virtualisierung* genannt. Zugriffe auf nicht-lokale Datenelemente werden auf speziell zugeschnittene prozessoreigene atomare Kommunikationsoperationen ($get/put$) abgebildet, die direkt aus dem Speicher eines beliebigen Prozessors lesen bzw. in den Speicher eines beliebigen Prozessors schreiben können. Zusammen mit einer identischen Belegung des Speichers auf allen Prozessoren (uniformes Speicherlayout), wobei parallele Variablen auf allen Prozessoren dieselbe Adresse im Hauptspeicher belegen, entspricht dies einem systemweit gemeinsamen Adreßraum, bei dem die Speicheradressen in eine Prozessorkennung und einen prozessorlokalen Anteil zerfallen, siehe Abbildung 2.1.

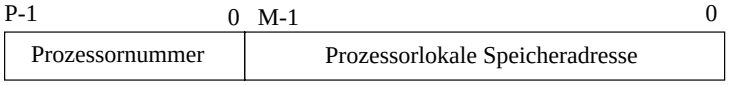



Abbildung 2.1: Adresse in einem gemeinsamen Adreßraum

Die Unteilbarkeit der Kommunikationsoperationen führt jedoch zu einer ineffizienten Programmausführung, da während einer Kommunikationoperation *alle* Prozessoren (Synchronität) auf deren Fertigstellung warten und in dieser Zeit nichts berechnen können. Damit

addiert sich die Netzwerklatenzzeit bei jeder Kommunikationsoperation zur Laufzeit des Programms.
Zu den typischen Vertretern des datenparallelen Programmiermodells zählen die datenparallelen Programmiersprachen auf der Basis von FORTRAN oder C und deren Derivate (MasPar Fortran [86], CM-Fortran [114], Vienna-Fortran [27, 15], Fortran-D [45, 63], Fortran-90 [90], HPF [42, 43], C* [113], DPC (Data Parallel C) [57, 58], MPL (MasPar Programming Language) [85] uvm.).

### 2.1.5 Programmierung von Architekturen mit verteiltem Speicher

Bei diesem sogenannten SPMD-Modell (*Single Program Multiple Data*), siehe Tabelle 2.1, entlehnt man sich aus der Datenparallelität der Idee, daß alle Prozesse zwar dasselbe Programm abarbeiten, sie dabei aber nur auf ihrem lokalen Anteil der verteilten Datenfelder operieren. Dies erleichtert die Programmierung von großen Systemen, da der Parallelitätsgrad und damit die Anzahl der Prozesse sich an der Größe der Datenfelder orientieren.
Beim SPMD-Modell müssen Kommunikationsanforderungen auf Sende- und Empfangsoperationen der nachrichtengekoppelten Zielarchitektur abgebildet werden, da die Prozessoren jetzt nur noch ihren lokalen Speicher adressieren können. Dazu müssen die zu kommunizierenden Daten auf der Senderseite eingepackt und verschickt und auf der Empfängerseite empfangen und ausgepackt werden.
Zusätzlich zur Latenzzeit des Netzwerks addiert sich nun der Aufwand für Verpacken, Senden, Empfangen und Auspacken zur Programmlaufzeit. Wenn für die Kommunikation eine standardisierte Kommunikationsbibliothek wie PVM [46] oder MPI [32] verwendet wird, erhöht sich der Laufzeitverlust um das Zehnfache der eigentlichen Netzwerklatenzzeit [17, 102].
Typische Vertreter dieser Richtung sind Programmierumgebungen wie PVM, P4 [26], MPI, Express [98], Parmacs [59] uvm., die auf verschiedenen Parallelrechner- und Workstationfarmenarchitekturen verfügbar sind, die NX/2-Umgebung auf Intel-Parallelrechnern (iPSC, Delta, Paragon [69, 71, 65, 20, 34, 22]) sowie OCCAM [68] auf Transputersystemen.

### 2.1.6 Programmierung von Architekturen mit gemeinsamem Speicher

Explizite Kommunikationsoperationen gibt es in Architekturen mit gemeinsamem Speicher nicht, denn die physikalische Verteilung der Daten wird durch das Speichersubsystem für die Prozesse transparent verdeckt. Jeder Prozessor führt somit nur Schreib- und Lesezugriffe aus, die vom Speichersystem dann an der entsprechenden Stelle ausgeführt werden. Dies hat zur Folge, daß für den Prozessor (nicht sichtbare) entfernte Datenzugriffe genauso effizient abgesetzt werden können wie Zugriffe in den lokalen Speicher. Doch bedeutet dies nicht, daß alle Speicherzugriffe gleich lange dauern: Denn liegt die Quelle einer Leseoperation im lokalen Zwischenspeicher, so kann der Zugriff wesentlich effizienter durchgeführt werden, als wenn das Datum aus dem Hauptspeicher oder einem anderen Cache über das Verbindungsnetzwerk geladen werden müßte.
Während sich Caches in Uniprozessorensystemen hervorragend zur Leistungssteigerung eignen, ist dies bei Parallelrechnerarchitekturen nur bedingt der Fall. Die Ursachen liegen in unerwünschten Nebeneffekten, wie zum Beispiel dem Überschneidungsproblem (*false sharing*) und der Cachekohärenzproblematik, die lediglich in Parallelrechnerarchitekturen zu finden sind. Das Überschneidungsproblem tritt auf, wenn zwei an sich unkorrelierte Datenelemente, auf die von zwei verschiedenen Prozessoren lesend und schreibend zugegriffen wird, zufällig

in derselben Cachezeile liegen, denn dann muß die jeweilige Cachezeile ständig zwischen den beteiligten Caches hin- und herwandern, was erhebliche Leistungseinbußen nach sich zieht. Die Vertreter des Programmiermodells der gemeinsamen Variablen sind typischerweise Programmierumgebungen (Betriebssystemerweiterungen und sogenannte *Thread*-Bibliotheken) auf den Multiprozessorsystemen der verschiedensten Hersteller (Sun, DEC, SGI, HP, Sequent uvm.).

### 2.1.7 Zusammenfassung

Die Vor- und Nachteile der verschiedenen Architekturmodelle bei der Kommunikation sind architekturgegeben und lassen sich in allen Anwendungsbereichen wiederfinden. Tabelle 2.2 faßt noch einmal die relevanten Punkte zusammen.

Tabelle 2.2: Vor- und Nachteile der Architekturmodelle bei der Kommunikation

| | **verteilter Speicher** | **gemeinsamer Adreßraum** | **gemeinsamer Speicher** |
|---|---|---|---|
| **Vorteile** | explizite Datenverteilung | • Kommunikation mit geringem Prozessoraufwand<br>• explizite Datenverteilung | kein Prozessoraufwand für nicht-lokale Speicherzugriffe |
| **Nachteile** | • Software-Mehraufwand<br>• Netzwerklatenz | • atomare Kommunikations-operationen<br>• Netzwerklatenz | • Netzwerkbelastung durch Kohärenzprotokoll<br>• Netzwerklatenz |

Alle drei Architekturmodelle werden durch die Netzwerklatenzzeit in ihrer Ausführungsgeschwindigkeit behindert. Auch erhöhen weitere architekturspezifische Probleme, wie z.B. der hohe Softwareaufwand für das Senden bzw. Empfangen von Nachrichten in nachrichtengekoppelten Architekturen oder die implizite Netzwerkbelastung in Architekturen mit gemeinsamem Speicher, die Laufzeit einer parallelen Applikation. Das im nächsten Abschnitt vorgestellte SCAP-Vefahren profitiert von der expliziten Datenverteilung und vom geringen Kommunikationsaufwand der Architekturen mit gemeinsamem Adreßraum und eliminiert den Nachteil der atomaren Kommunikationsoperationen.

## 2.2 Das SCAP-Verfahren

Betrachtet man die Vor- und Nachteile der Architekturmodelle bei der Kommunikation von Tabelle 2.2, so fällt auf, daß in Architekturen mit gemeinsamem Adreßraum (SIMD-Architekturen) wegen der synchronen Abarbeitung und der unteilbaren (atomaren) Kommunikationsoperationen keine Latenzzeitverbergung stattfinden kann. Verzichtet man jedoch auf die Synchronität der SIMD-Architekturen zugunsten einer Abarbeitung asynchroner Prozesse mit einer schnellen Barrierensynchronisation (*Cray-Modell* von Tabelle 2.1) und spaltet die atomaren Leseoperationen in eine Vorlade- und eine Zugriffsanweisung auf, so können in diesem Architekturmodell Daten vorgeladen werden. Um jedoch eine vollständige Verdeckung

der Netzwerklatenz von mehreren 100 bis 1000 Prozessortakten erreichen zu können, muß die zeitliche Spanne zwischen Vorladeanweisung und Zugriff (*Vorladedistanz*) entsprechend groß gewählt werden. Diese Distanz ist durch Daten- und Kontrollabhängigkeiten beschränkt und im allgemeinen zu gering, um Zugriffszeiten dieser Größenordung überdecken zu können. Deshalb werden nur datenparallele Anwendungen betrachtet. Diese stellen durch die Virtualisierung genug entfernte Datenzugriffe bereit, mit denen durch Vorladeoperationen die Zugriffszeiten auf das erste und alle weiteren nicht-lokalen Datenelemente verdeckt werden können. Dies erweitert bisherige Vorladetechniken, die bisher nur Kommunikation mit Berechnung verdeckten, um die Überlappung von Kommunikation mit dem weiteren Absetzen von Vorladeanweisungen, d.h. weiteren Kommunikationsaufrufen. Im Idealfall reduziert sich somit der Kommunikationsaufwand auf die Zeit, die zum Absetzen der einzelnen Vorladeanweisungen benötigt wird.
Die hier vorgestellten Konzepte und Ergebnisse basieren auf der Arbeit von Warschko [120].

### 2.2.1 Das Netzwerkmodell und Arbeitsweise des SCAP-Verfahrens

Ein *überlappendes* Netzwerk zeichnet sich im Gegensatz zum *blockierenden* Netzwerk durch die parallele Abarbeitung *mehrerer* Kommunikationsaufträge des selben Prozessors aus. Der prinzipielle Unterschied zwischen blockierenden und überlappenden Operationen in einem Kommunikationsnetzwerk ist in Abbildung 2.2 dargestellt. Die verwendeten Parameter sind in Tabelle 2.3 zusammengefaßt.

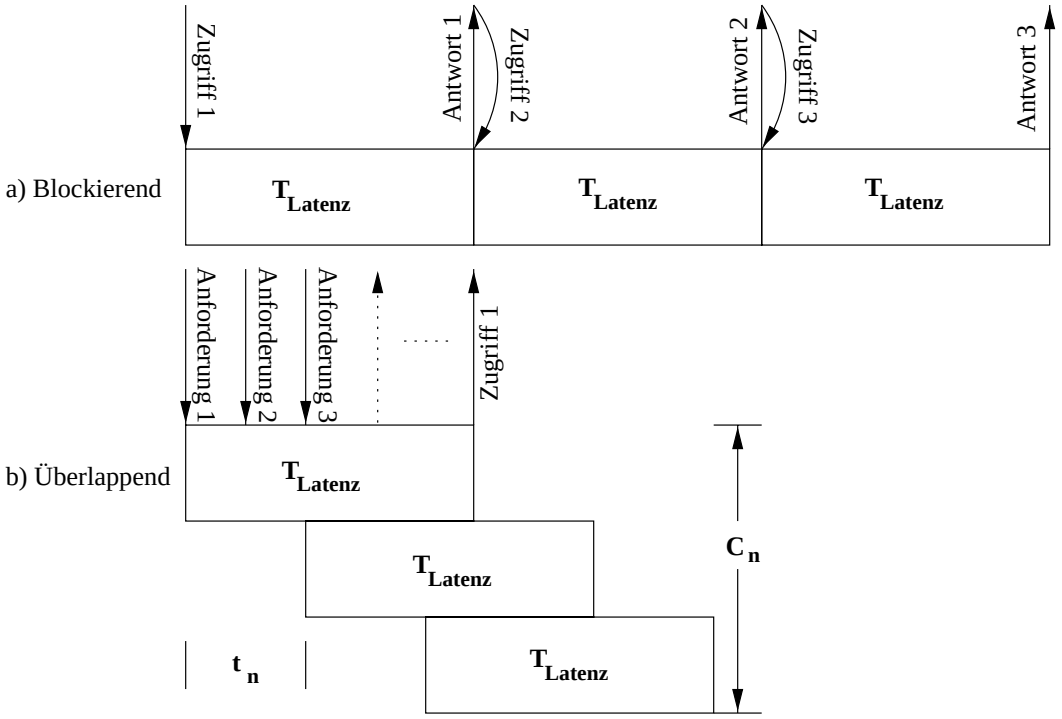



Abbildung 2.2: Netzwerkmodelle

**Blockierend:** Ein Kommunikationsauftrag blockiert den Prozessor so lange, bis die Daten vom Netzwerk geliefert werden können. Dadurch kann keine Überlappung von

Kommunikations- und Berechnungszeiten stattfinden, und die Wartezeit des Prozessors entspricht exakt der Latenzzeit des Netzwerkes. Nach dem Zugriff $i$ entsteht für den Prozessor eine Wartezeit von $T_{\text{Latenz}}$, bis das Netzwerk die Antwort $i$ liefern kann. Erst dann kann der Prozessor den nächsten Zugriff $i+1$ absetzen.

**$(C_N, t_n)$-Überlappend:** Das Netzwerk ist in der Lage, alle $t_n > 0$ Taktzyklen eine Kommunikationsoperation einzuleiten, wobei die Anzahl an gleichzeitig ausstehenden Anfragen pro Knoten im Netzwerk auf $C_N > 1$ beschränkt ist. Der Prozessor kann nun vom Netzwerk entkoppelt alle Kommunikationsanforderungen auf einmal absetzen. Erst danach versucht er, auf das erste nicht-lokale Element zuzugreifen. Sollte die Anforderung bis dahin vom Netzwerk noch nicht erledigt sein, so wird der Prozessor bis zur Bereitstellung angehalten (gestrichelte Linie in Abbildung 2.2).

Tabelle 2.3: SCAP Modellparameter

| **Parameter** | **Beschreibung** |
|---|---|
| **Applikationsparameter** | |
| $t_v$ | Zeit für Vorladeanweisung |
| $t_z$ | Zeit für Zugriff |
| $K$ | Anzahl der nicht-lokalen Zugriffe |
| **Architekturparameter** | |
| $C_V$ | Größe des Vorladepuffers |
| $T_{\text{Latenz}}$ | Latenzzeit des Netzwerkes |
| $t_n$ | Zykluszeit des Netzwerkes |
| $C_N$ | Kapazität des Netzwerkes pro Knoten |

Die Leistungsfähigkeit eines $(C_N, t_n)$-überlappenden Netzwerkes hängt vor allem von der Bandbreite und des Durchsatzes ab. Während die Bandbreite bereits beim Entwurf einer Architektur festgelegt wird, ist der Durchsatz von dynamischen Einflußfaktoren abhängig, wie z. B. der Anzahl der beteiligten Knoten oder der Last des Netzwerkes, und er variiert daher mit den aktuellen Gegebenheiten. Die gewünschten $C_N$ überlappenden Kommunikationsoperationen pro Knoten erfordern ein Netzwerk, das sich auch in Hochlastsituationen relativ robust zeigt. Die beiden Parameter $C_N$ und $t_n$ sind dabei insofern korreliert, als das Produkt der beiden Werte gerade der Latenzzeit entspricht:

$$T_{\text{Latenz}} = C_N * t_n \tag{2.1}$$

Die Netzwerkkapazität $C_N$ hängt von der Anzahl der Netzwerkhops ab, da sie sich mit steigender Entfernung zweier Netzwerknoten vergrößert.
In einem überlappenden Netzwerk werden Kommunikationsoperationen in eine Vorladeanweisung (*prefetch*) und den eigentlichen Zugriff (*access*) aufgespalten, zwischen denen der Prozessor beliebige weitere Anweisungen durchführen kann.

Innerhalb einer Vorladeschleife generiert der Knotenprozessor alle $t_v$ Zeitschritte eine Vorladeoperation für ein entferntes Datenelement, die einem Vorladepuffer mit Kapazität $C_V$ übergeben wird. Das Netzwerk holt sich alle $t_n$ Zeitschritte einen Auftrag aus dem Vorladepuffer und führt ihn aus. Die Abarbeitung der Vorladeschleife ist von der Arbeitsweise des Netzwerkes durch den Einsatz des Vorladepuffers zeitlich entkoppelt, so daß $t_v$ in keinerlei Bezug zu $t_n$ steht. Nach der Vorladeschleife greift der Prozessor alle $t_z$ Zeitschritte auf die Elemente des Vorladepuffers innerhalb der Zugriffsschleife zu. Ist die Anzahl der Kommunikationsaufträge $K$ größer als die Kapazität des Vorladepuffers $C_V$, so muß ein Überlaufen des Vorladepuffers vermieden werden. Dazu wird eine mittlere Schleife eingefügt, die zwischen Zugriffs- und Vorladeoperationen abwechselt.
Es wird davon ausgegangen, daß die Kommunikationsoperationen vom überlappenden Netzwerk in der Anforderungsreihenfolge ausgeführt werden. Die Reihenfolgetreue der Operationen ist für die Korrektheit nicht zwingend, sie garantiert aber die ideale Überlappung der Kommunikationsoperationen und damit eine optimale Ausnutzung der Latenzzeitverbergung. Damit für den Prozessor beim Zugriff auf das erste nicht-lokale Element keine Wartezeiten entstehen, muß der Vorladepuffer

$$C_V \geq \frac{T_{\text{Latenz}}}{t_v} \tag{2.2}$$

Einträge fassen können. Mit dieser minimalen Größe des Vorladepuffers kann der Prozessor die Latenzzeit des Netzwerkes vollständig mit Vorladeanweisungen überlappen und verzögerungsfrei auf das erste und weitere Elemente zugreifen. Weiterhin wird davon ausgegangen, daß der Vorladepuffer wesentlich mehr Vorladeoperationen zwischenspeichern als das Netzwerk überlappen kann, d. h. es gilt $C_V \gg C_N > 1$.
Die Einträge des Vorladepuffers erhalten Gültigkeitskennungen, welche Prozessor und Netzwerk synchronisieren. Fordert der Prozessor ein entferntes Datum an, so wird ein Eintrag im Vorladepuffer reserviert und seine Kennung gelöscht. Sobald das Netzwerk die Operation ausgeführt hat, schreibt es das Ergebnis in den reservierten Eintrag und setzt die Kennung. Wenn der Prozessor auf einen Eintrag mit gelöschter Kennung zugreift, wird er angehalten, bis das Netzwerk die Operation abschließt. Sollte beim Zugriff die Kennung jedoch gesetzt sein, so kann der Prozessor ohne Verzögerung den Wert auslesen.

### 2.2.2 Die Transformationsregel

Aus Ungleichung (2.2) läßt sich berechnen, wieviel nicht-lokale Datenzugriffe der Prozessor braucht, um die Netzwerklatenz mit Vorladebefehlen zu überbrücken. Beträgt die Netzwerklatenz 1000 Prozessortakte und der Prozessor kann in jedem Takt eine Vorladeoperation ausführen ($t_v = 1$), so müssen 1000 Anforderungen abgesetzt werden, bevor auf das erste nicht-lokale Datum verzögerungsfrei zugegriffen werden kann. Im allgemeinen wird es jedoch nicht möglich sein, aus einem Programm diese erforderliche Anzahl von entfernten Datenelementen zu erhalten, ohne dabei bestehende Datenabhängigkeiten zu verletzen. Wenn jedoch datenparallele Schleifen betrachtet werden, so kann mit Hilfe der Abbildung der Problemgröße auf die vorhandenen Prozessoren, den *Virtualisierungsschleifen* [99], die benötigte Anzahl an entfernten Elementen erhalten werden.

Die Transformation wird an folgender datenparallelen Schleife beschrieben:

```
FORALL i = 0 TO N-1 DO
  A[i] := B[q(i)];
END FORALL
```

Obiges Forall ist eine datenparallele Zuweisung von Feld $B$ an Feld $A$, die in disjunkten Speicherbereichen liegen. Auf die Elemente von $B$ wird indirekt durch $q$ zugegriffen.

Zu Beginn der Transformation wird die Problemgröße $N$ auf die $P$ realen Prozessoren abgebildet. Dabei emuliert jeder der $N$ physikalischen $V = \frac{N}{P}$ logische Prozessoren.[3] Nach dieser Virtualisierung werden die entfernten Datenzugriffe eingefügt. Da $q$ in diesem Beispiel nicht zur Übersetzungszeit berechnet werden kann, wird jeder Datenzugriff in eine entfernte Leseoperation übersetzt.

```
V := N/P;
FORALL j = 0 TO P-1 DO
  FOR k = j*V TO (j+1)*V-1 DO
    addr := calculate_address(B[q(k)]);
    A[k] := remote_read(addr);
  END FOR
END FORALL
```

Obiges Programmfragment ist gleichbedeutend mit der Ausführung in einem blockierenden Netzwerk. Der Prozessor berechnet die Adresse von $B[q(k)]$ und greift auf sie zu. In *remote_read* ist ein entfernter Datenzugriff verborgen, der den Prozessor gegebenenfalls für die Dauer der Netzwerklatenz anhält, wenn das Element $B[q(k)]$ auf einem anderen Knoten liegt.

Der nächste Transformationsschritt spaltet die bisher blockierende Leseoperation in eine Vorlade- (*prefetch*) und eine Zugriffsoperation (*access*) auf. Bei dieser Trennung muß ein Überlaufen des Vorladepuffers vermieden werden. Deshalb werden höchstens $C_V$ Elemente in der ersten, der *Vorladeschleife* angefordert. Danach wird auf das erste Element innerhalb der integrierten *Vorlade-* und *Berechnungsschleife* zugegriffen. Der freigewordene Eintrag des Vorladepuffers wird wieder benützt, um das nächste Element vorzuladen. Damit wird erreicht, daß zwischen Vorladezeitpunkt und Zugriff immer $C_V$ Anforderungen stehen und die Latenzzeit optimal verborgen wird. In der dritten Schleife, der *Berechnungsschleife* werden die restlichen $C_V$ angeforderten Elemente zugewiesen:

[3] In diesem Beispiel wird vereinfachend $N \ mod \ P \equiv 0$ angenommen.

```
V := N/P;
FORALL j = 0 TO P-1 DO
                        /* Vorladeschleife */
  FOR k = j*V TO j*V + MIN (V, C_V) -1 DO
    addr := calculate_address(B[q(k)]);
    prefetch(addr);
  END FOR
                        /* Integrierte Berechnungs- und Vorladeschleife */
  FOR k = j*V TO (j+1)*V-1-C_V DO
    addr := calculate_address(B[q(k)]);
    A[k] := access(addr);
    addr := calculate_address(B[q(k+C_V)]);
    prefetch(addr);
  END FOR
                        /* Berechnungsschleife */
  FOR k = j*V + MAX (0, V-C_V+1) TO (j+1)*V-1 DO
    addr := calculate_address(B[q(k)]);
    A[k] := access(addr);
  END FOR
END FORALL
```

In der vorgestellten dreischleifigen Abarbeitung ist noch die Berechnung der Schleifengrenzen eingefügt, denn der Übersetzer kann den Wert von $V$ statisch nicht berechnen. Wenn die Problemgröße zur Übersetzungszeit jedoch bekannt und $V \leq C_V$ ist, so kann die mittlere Schleife entfallen.
Vor der Kommunikation muß eine globale Barrierensynchronisation durchgeführt werden, damit die asynchronen Prozesse eine konsistente Sicht auf die verteilten Daten haben. Dies führt zu einem Programmablauf in dem sich Phasen mit lokaler Berechnung, Synchronisation und Kommunikation abwechseln. Diese Arbeitsweise ist mit dem BSP-Modell von Valiant (siehe 2.3.4) vergleichbar, das ein Programm in Superschritte aufteilt, die durch globale Synchronisationsbarrieren getrennt sind. Die Synchronisation der Prozesse nach oder vor jeder Ausführung einer datenparallelen Operation ist jedoch aus Effizienzgesichtpunkten nicht akzeptabel, sodaß notwendige Synchronisationspunkte gefunden werden müssen, ohne dabei die semantische Korrektheit zu gefährden [100, 99].

### 2.2.3 Grenzen des SCAP-Verfahrens

Das SCAP-Verfahren stößt an seine Grenzen, wenn eine Abhängigkeit zwischen aufeinanderfolgenden Kommunikationsoperationen vorliegt. Benötigt zum Beispiel Kommunikationsoperation $B$ das Ergebnis einer vorangegangenen Kommunikationsoperation $A$ zur Berechnung der Adresse im gemeinsamen Adreßraum des Parallelrechners, dann können diese beiden Kommunikationsoperationen nicht auf ein und diesselbe Vorladeschleife abgebildet werden. Eine solche Situation kann nicht zwischen aufeinanderfolgenden Iterationen der Virtualisierungsschleife eintreten, denn das würde der Definition der Virtualisierung in Form von unabhängigen konzeptuellen Prozessen widersprechen. Findet sich dagegen eine solche Situation innerhalb eines Iterationsschritts der Virtualisierungsschleife, dann muß zwischen den abhängigen Kommunikationsoperationen eine Synchronisationsoperation liegen, denn sonst

wäre die semantische Korrektheit der Virtualisierungsschleife nicht gewährleistet. Als Einschränkung für des SCAP-Verfahren folgt daraus, daß es sich nicht über Synchronisationsgrenzen hinweg einsetzen läßt. Eine Lösung des Problems besteht darin, die ursprüngliche Virtualisierungsschleife an der Synchronisationsgrenze in zwei Virtualisierungsschleifen aufzuteilen, für die dann jeweils wieder das SCAP-Verfahren angewendet werden kann.
Im Fall von parallelen Fließbandtechniken, bei denen Prozessor $i$ die Eingabedaten für einen im Fließband nachfolgenden Prozessor $j$ berechnet, läßt sich das SCAP-Verfahren nicht einsetzen. Dies stellt allerdings keine Einschränkung des SCAP-Verfahrens dar, da die Fließbandverarbeitung von sich aus in der Lage ist, Kommunikationszeiten zu verdecken, wie es in [23] am Beispiel des folgenden ADI-Fragments gezeigt wird:

```
DISTRIBUTE (*,BLOCK) :: A, B
FOR j = 2 TO N DO
  FOR i = 1 TO N DO
    A[i,j] := A[i,j] - A[i,j-1] * B[i,j];
  END FOR
END FOR
```

### 2.2.4 Bisherige Ergebnisse

Die simulativ gewonnenen Ergebnisse zeigen, daß der kombinierte Einsatz von Software- (Vorladeoperationen) und Hardwaretechniken (Vorladepuffer) zum Aufbau eines Datenfließbandes zur Überlappung von Kommunikations- und Berechnungsoperationen dazu benützt werden kann, die Kommunikationszeiten in Parallelrechnern fast vollständig zu verdecken.
Warschko befaßt sich zunächst mit der analytischen Modellierung des Verhaltens von Prozessor und Netzwerk bei der Ausführung eines Datenfließbandes. Seine Modellierung wich dabei weniger als 1.4% von den Simulationen durch den DLX-Simulator [60] ab, der die Funktionsweise des DLX-Prozessors auf Instruktionsebene nachbildet. Weiterhin untersuchte er SCAP auf die Fähigkeit, die großen in der Parallelverarbeitung vorkommenden Latenzzeiten zu verbergen. Als Testprogramme benutzte er die Livermore-Kerne [89], Jacobi-Iteration und Red-Black-SOR. Er simulierte eine Umgebung mit 1024 parallel eingesetzten Prozessoren und einem 100mal langsameren Netzwerk von 1000 Prozessortakten Latenzzeit. Bis auf Reduktionsoperationen, bei denen SCAP lediglich 46% bis 76% der Latenzzeit verbarg, konnten schon bei kleinen Virtualisierungen (zwischen 3 und 57) 85% der Latenzzeit verborgen werden. Voraussetzung für eine gute Latenzzeitverbergung ist jedoch ein hoher Anteil an entfernten Datenzugriffen. Fehlt dieser Anteil oder ist er nur sehr klein, so verringern sich auch die Fähigkeiten von SCAP, große Latenzzeiten zu überbrücken.
In seinen Tests beschränkt sich Warschko auf Simulationsergebnisse. Dadurch bleibt die Frage offen, in wie weit sich das analytische Modell SCAP auf eine existierende Parallelrechnerarchitektur übertragen läßt. Weiterhin sind seine Testprogramme handprogrammiert. Für den Einsatz von SCAP ist es jedoch notwendig, die Effektivität der Datenfließbänder zu testen, wenn sie automatisch vom Übersetzer generiert werden. Als letzter Punkt zur Abgrenzung gegenüber der Arbeit von Warschko sei die hier vorgenommene Erweiterung von SCAP mit Vektorbefehlen erwähnt (VSCAP). Durch diese Erweiterung wird SCAP zum Spezialfall der überlappenden Kommunikation, wenn man in VSCAP die Vektorlänge $L$ auf eins setzt. Vektorbefehle bedeuten eine weitere Reduzierung des Vorladeaufwandes und verringern somit den Kommunikationsaufwand, was zu einer Beschleunigung gegenüber dem SCAP-Verfahren führt.

## 2.3 Alternative Mechanismen der Latenzzeitverbergung

Die Verbergung von Speicher- oder Netzwerkszugriffszeiten zur Verringerung der Laufzeit sequentieller und paralleler Applikationen ist nicht neu, deshalb stellt dieser Abschnitt existierende Techniken zur Latenzzeitverbergung vor. Dabei werden nicht nur die Probleme sequentieller Verfahren in der Parallelverarbeitung und die Nachteile speziell für die Parallelverarbeitung entwickelter Techniken aufgezeigt sondern auch Unterschiede zum SCAP-Verfahren und damit auch zu VSCAP hervorgehoben.

### 2.3.1 Vorladen von Datenelementen

Diese Technik versucht durch Erhöhung der Lokalität entfernter Daten die Zugriffszeit für den Prozessor zu verringern. Die drei Hauptrichtungen *hardwaregesteuert*, *softwaregesteuert* und *hybride Techniken* werden nach dem Maß der Hardwareunterstützung für die Vorladeoperationen unterschieden. Die Zielarchitekturen dieser Techniken sind Uniprozessor- oder Parallelrechnerarchitekturen mit gemeinsamem (konsistenten) Speicher. Damit Speicherzugriffe effektiv durch Berechnung verdeckt werden können, setzen alle Mechanismen eine überlappende Ausführung ausstehender Anforderungen durch das Speichersubsystem voraus.

#### 2.3.1.1 Hardwaregesteuertes Vorladen

Beim hardwaregesteuerten Vorladen entscheidet ausschließlich die Hardware welche Speicherstellen vorzeitig angefordert werden sollen. Ihr Vorteil gegenüber softwaregesteuerten Mechanismen ist ihre Transparenz für einen Programmierer oder Übersetzer, da sie ohne Änderung des Programmtextes arbeiten. Ihr Nachteil liegt in ihrer Schwäche, irreguläre Zugriffsmuster weder zu erkennen noch auszunutzen.

**One Block Lookahead Vorladestrategien (Uniprozessor):** Techniken mit dem geringsten Hardwareaufwand sind die sogenannten *one block lookahead (OBL)* Vorladeschemata des Zwischenspeichers. Sie sind dadurch geprägt, daß sie beim Zugriff auf Cachezeile $i$ nur jeweils die nächste Cachezeile $i+1$ als Kandidat zum Vorladen betrachten.
Smith unterscheidet in [107] drei verschiedene Techniken:

- *prefetch always* lädt bei *jedem* Zugriff auf Zeile $i$ die nächste Zeile $i+1$ vor.
- Bei der *prefetch on miss*-Strategie wird beim Fehlen einer Zeile die nächste gleich mitgeladen, sofern sie nicht schon vorhanden ist. Dies halbiert die Anzahl der Fehlzugriffe beim sukzessiven Durchlaufen der Speicherstellen.
- Beim *tagged prefetch* bekommt jede Zeile eine Kennung (*tag*). Sie wird zurückgesetzt sobald eine Zeile vorgeladen wird. Beim ersten Zugriff auf eine vorhandene Zeile wird die Kennung gesetzt. In diesem Fall wird die darauffolgende Zeile geladen, falls sie nicht schon vorhanden ist. Mit dieser Technik lassen sich alle Fehlzugriffe bei einem sukzessiven Zugriff der Speicherstellen vermeiden.

*Prefetch-always* reduziert die Zugriffsfehler um 50-90% gegenüber einem nicht vorladenden Zwischenspeicher. *Tagged-prefetch* erreicht fast die gleichen Zahlen während *Prefetch-on-miss* lediglich nur halb so gut in der Fehlervermeidung ist. Der Vorteil von *Tagged-prefetch*

gegenüber *Prefetch-always* liegt in der geringeren Anzahl von unnötig in den Zwischenspeicher geladenen Zeilen.
Der Nachteil der OBL-Techniken zeigt sich zum einen bei größerer Latenzzeit, wenn eine Zeile des Zwischenspeichers nicht ausreicht, um die Speicherlatenz zu überbrücken, und zum anderen wenn die Schrittweite der Speicherzugriffe größer als die Zeilenlänge ist. Das SCAP-Verfahren hingegen hat durch die Virtualisierung genügend Information über nicht-lokale Datenzugriffe, so daß für die Verdeckung der Wartezeiten auf die einzelnen nicht-lokalen Datenzugriffe genügend Vorladeoperationen bereitstehen.

**Reference Prediction Table (Uniprozessor):** Größere Latenzzeiten und Schrittweiten versuchen Baer und Chen mit ihrer *Reference Prediction Table (RPT)* [12, 30] in den Griff zu bekommen. Die RPT ist eine Tabelle, die als Einträge die Adresse des Programmzählers (der Indizierungsschlüssel), eine Referenzadresse, eine Schrittweite und Statusinformation enthält. Zusätzlich wird ein *vorausschauender Programmzähler (LA-PC)* eingeführt, der vor dem Programmzähler das Programm durchläuft und die RPT modifiziert. Dabei werden nur Lese- und Schreibzugriffe von und zum Speicher betrachtet. Trifft der LA-PC auf eine solche Operation, wird in der RPT die Zeile mit der entsprechenden Schlüsseladresse ausgewählt. Danach wird die Differenz zwischen Zugriffs- und Referenzadresse gebildet. Wenn die Differenz und die gespeicherte Schrittweite identisch sind und die Zeile nicht im Zwischenspeicher steht, so wird sie beim Speicher angefordert.
Da der LA-PC dem Programmzähler voraus läuft (ähnlich dem Prinzip der entkoppelten Prozessoren, Abschnitt 2.3.2.3), können eine größere Vorladedistanz erreicht und größere Speicherlatenzzeiten überbrückt werden. Wenn der LA-PC jedoch Daten zu früh in den Zwischenspeicher lädt, so kann es sein, daß sie bis zum Gebrauch wieder verdrängt werden. Außerdem ist bei größerer Vorladedistanz eine größere Last des Speicherbuses gegeben, die sich nachhaltig auf die Zugriffszeiten auswirken kann. Aus diesem Grund wird ein LA-PC-Limit eingeführt, das den maximalen Abstand zum Programmzähler angibt.
Mit der RPT können Zugriffsfehler im Zwischenspeicher bei regulären Mustern um 97% gesenkt werden. Bei irregulären Zugriffen spielt das LA-PC-Limit eine bestimmende Rolle. Ein zu großer Wert kann vorhandene Zeilen verdrängen, während ein zu kleiner Wert nicht ausreicht, um die Speicherlatenz zu verdecken.
Baer und Chen können mit ihrer RPT größere Speicherlatenzen und Schrittweiten der Speicherzugriffe gut behandeln. Aber auch ihr Ansatz scheitert bei irregulären Zugriffsmustern, die jedoch für das SCAP-Verfahren durch das softwaregesteuerte Vorladen kein Problem darstellen. Ein weiterer Nachteil der RPT ist auch die Tatsache, daß die RPT erst einmal gefüllt werden muß, bevor sie Speicherlatenzen verbergen kann. Das SCAP-Verfahren kann hingegen sofort mit der Latenzzeitverbergung beginnen, da die Information über nicht-lokale Datenzugriffe bereits durch die Virtualisierung zur Übersetzungszeit feststeht.

**Stream Buffer (Uniprozessor):** Einen vollkommen anderen Ansatz als die bisher vorgestellten Techniken verfolgen die *Stream Buffer* von Jouppi [73]. Dort wird versucht eine hohe Speicherbandbreite auszunutzen, um gleich mehrere, aufeinanderfolgende Zeilen des Zwischenspeichers auf einmal aus dem Hauptspeicher vorzuladen. Die vorgeladenen Zeilen werden durch die Stream-Buffer verwaltet.
Bei einem Zugriffsfehler des Zwischenspeichers schalten sich die Puffer zwischen Zwischenspeicher und Hauptspeicher und können dann in wenigen Takten die geforderte Zeile bereitstellen.

Im Ansatz von Jouppi kann jedoch nur mit dem ersten Puffereintrag verglichen werden. Dadurch entstehen zusätzliche Wartezeiten, wenn auf nachfolgende Zeilen des Puffers zugegriffen wird. Denn dann werden die Zeilen aus dem Hauptspeicher geladen, obwohl sie bereits im Puffer stehen. Heutige Stream-Buffer haben jedoch eine höhere Assoziativität, so daß dieses Problem nicht mehr eintritt. Sie werden z.B. in Mehrprozessorsystemen eingesetzt, bei denen ein großer dritter Zwischenspeicher außerhalb eines Prozessors Kühlprobleme verursachen würde [97]. Sie verkürzen dort Speicherzugriffe wissenschaftlicher Anwendungen in den lokalen Hauptspeicher, die vorwiegend Zugriffe mit regulärer Schrittweite aufweisen.
Der Stream-Buffer kann 25% der Zwischenspeicherfehler beseitigen, indem er die fehlende Zeile innerhalb eines Taktes zur Verfügung stellt. Diese Anzahl kann noch auf 43% erhöht werden, wenn mehrere (in diesem Fall vier) unabhängige Stream-Buffer verwendet werden. So können die verschiedenen Adreßströme getrennt und eine bessere Latenzzeitverdeckung erreicht werden.

Alle bisher vorgestellten Techniken können keine irregulären Zugriffsmuster erkennen und effizient vorladen. Es gibt Ansätze in der Literatur, wie z.B. [31], die Registerinhalte auswerten um dynamische Schrittweiten behandeln zu können. Diese Techniken sind jedoch mit erheblicher Zusatzlogik verbunden, so daß sie bisher nur Vorhersagen treffen können, die eine Schleifeniteration voraus liegen. Somit können große Speicherlatenzen nicht vollständig verdeckt werden, was jedoch im SCAP-Verfahren durch die Virtualisierung möglich ist. Weiterhin fordern alle untersuchten hardwaregesteuerten Techniken ganze Zeilen des Zwischenspeichers an. Dies ist für die Verwaltung einfacher, da sie orthogonal zum bestehenden System arbeiten. Es resultiert jedoch in erhöhter Last des Speicherbuses, welche die Zugriffszeiten noch einmal vergrößern kann, besonders wenn die vorgeladenen Zeilen nicht gebraucht werden. Der Einsatz eines Vorladepuffers im SCAP-Verfahren reduziert diesen zusätzlichen Busverkehr, da nur die nicht-lokalen Datenelemente kommuniziert werden müssen, die vom Prozessor auch angefordert wurden.

#### 2.3.1.2 Softwaregesteuertes Vorladen

Die Ineffizienz hardwarebasierter Vorladetechniken bei irregulären Speicherzugriffen, ist bei softwaregesteuerten Techniken nicht gegeben. Denn der Übersetzer oder Programmierer hat mehr Wissen über die Zugriffsstruktur als die Hardware, welche ihre Information aus dem Adreßstrom des Programms extrahieren muß. Weiterhin haben softwaregesteuerte Vorladetechniken minimale Hardwarevoraussetzungen, da sie lediglich nicht-blockierende Zwischenspeicher und eine nicht-blockierende Vorladeanweisung voraussetzen. Ihr wesentlicher Nachteil gegenüber Hardwaretechniken liegt im Mehraufwand für den Prozessor, der die zusätzlichen Anforderungen ausführen muß. Die Vorladeanweisungen bewirken auch eine nicht unerhebliche Vergrößerung des Programmtextes. Ein weiterer Nachteil stellt die mangelnde Portierbarkeit eines Programms dar. Denn Parameter zur effektiven Generierung der Vorladebefehle sind z.B. Zeilengröße und Kapazität des Zwischenspeichers sowie die Speicherlatenz. Sie sind charakteristische Hardwarekonstanten, welche sich schlecht oder gar nicht von einer auf die andere Architektur übertragen lassen. Aus diesem Grund muß ein gegebenes Programm mit den neuen Parametern übersetzt werden, damit die Vorteile des Vorladens ausgenutzt werden können. Softwaregesteuerte Vorladetechniken betrachten vor allem Schleifen, denn in ihnen wird ein Großteil der Programmausführung verbracht und die Vorladeanweisungen können dort oftmals weit genug im voraus abgesetzt werden, um auch

größere Latenzzeiten überbrücken zu können.
Softwaregesteuertes Vorladen wurde in vielen Projekten untersucht. Hier sollen exemplarisch zwei Arbeiten vorgestellt werden.

**Block-Vorladen (Multiprozessor):** Gornish, Granston und Veidenbaum benützen eine *Block-Vorladeanweisung*, die einen größeren Speicherblock anfordert [50]. Die Autoren begründen ihren asynchronen Blocktransfer mit einer höheren erreichbaren Speicherbandbreite, als wenn jedes Speicherwort einzeln angefordert würde. Die Ergebnisse erhalten sie durch Simulation eines speichergekoppelten Mehrprozessorsystems mit 32 Prozessoren, welche durch ein mehrstufiges Omega-Netzwerk miteinander verbunden sind. Vorgeladen wird zum frühest möglichen Zeitpunkt im Programmtext, den Kontroll- und Datenabhängigkeiten zulassen. Weiterhin nehmen sie an, daß ein Prozessor eine unbeschränkte Anzahl an Speicheranforderungen absetzen kann. Die Autoren betrachten in ihren Tests zwei verschiedene Ausführungsmodi:

- Im **nicht-überlappenden** Modus wird die nächste Operation erst ausgeführt, wenn die vorherige abgeschlossen wurde, während
- im **überlappenden** Fall Speicherzugriffe und Berechnung parallel ausgeführt werden können.

Die Autoren haben die Kombinationen *nicht-überlappend/nicht-vorladen*, ihr Basismodell, sowie *überlappend/nicht-vorladen* und *überlappend/vorladen* miteinander verglichen. Sie erreichten mit *überlappend/vorladen* einen Geschwindigkeitsgewinn gegenüber ihrem Basismodell zwischen 8 und 61%. Sie maßen auch die anfallenden Speicherlatenzen und Datenankunftszeiten in den einzelnen Fällen. Dabei stellte sich heraus, daß sich trotz geringerer Prozessorwartezeiten (sie wurden zwischen 32 und 97% reduziert) die Speicherlatenz bei *überlappend/vorladen* um über eine Größenordnung erhöhte und sich die Ankunftszeiten der Daten mindestens verdoppelte. Sie begründen die hohe Netzwerklast durch ihr simuliertes mehrstufiges Netzwerk und verweisen auf eine bessere Netzwerkanbindung zur Behebung des Problems.

**SUIF-Projekt (Uni- und Multiprozessor):** Der zweite hier vorgestellte Ansatz von Mowry [93, 92, 83] entstand im Zusammenhang mit dem *SUIF*-Projekt der Stanford Universität.
Im Gegensatz zur Arbeit von Gornish, Granston und Veidenbaum beschränkt sich Mowry auf das Vorladen von Zeilen des Zwischenspeichers. Nach einer Lokalitätsanalyse, die Zwischenspeichergröße und Datenwiederverwendung beachtet, werden diejenigen Adressen vorgeladen, die wahrscheinlich einen Zugriffsfehler im Zwischenspeicher auslösen. Die Programmrestrukturierung führt zur bestehenden Schleife einen Prolog, der erforderliche Daten vorlädt, und einen Epilog ein, welcher lediglich auf Daten zugreift. Die ursprüngliche Schleife wird um Vorladeanweisungen erweitert, damit nachfolgende Iterationen ihre Daten ebenfalls lokal im Zwischenspeicher finden.
Die Vorladedistanz $d$ wird in Schleifeniterationen angegeben und wird durch $d = \lceil \frac{l}{s} \rceil$ berechnet. Dabei bezeichnet $l$ die Speicherlatenz und $s$ die Länge des kürzesten Pfades durch den Schleifenkörper.

Der vorgeschlagene Algorithmus unterdrückt unnötige Vorladeanweisungen bis zu einem Faktor von 21 verglichen mit einem naiven Ansatz, der alles vorlädt. Dabei blieben die wieder neu auftauchenden Zugriffsfehler unter 10%.
Waren bisher die Speicherzugriffe auf affine Funktionen beschränkt, so betrachtet [92] auch indirekte Zugriffe und das Vorladen in speichergekoppelten Mehrprozessorsystemen mit konsistenten Zwischenspeichern.
Das Problem indirekter Zugriffe ist, daß die Adresse nicht zur Übersetzungszeit bestimmt werden kann. Ein Lokalitätstest zur Laufzeit wäre mit zu großem Mehraufwand verbunden; deshalb werden alle Speicherstellen indirekt adressierter Felder vorgeladen. Dies erweitert die Vorladestrategie, da auch die Adresse des Indexfeldes vorgeladen werden muß. Dieser Mehraufwand war nicht unerheblich, denn er verhinderte für Testprogramme mit Problemgrößen, die vollständig in der Zwischenspeicher paßten, einen zusätzlichen Geschwindigkeitsgewinn gegenüber dem originalen Vorladealgorithmus.
Weitere Probleme treten bei einem Wechsel von einem Uniprozessor- zu einem speichergekoppelten Mehrprozessorsystem auf. Da stellen sich z.B. die Fragen:

1. Wann wird ein Wert an eine Adresse gebunden? D.h. wann darf er von fremden Prozessoren noch verändert werden, wenn er einmal geladen wurde? (Überschneidungsproblem)

2. Wann sollen gemeinsam benutzte Daten vorgeladen werden?

Es wurde eine nicht-bindende Vorladestrategie gewählt, bei welcher der Wert erst beim *Zugriff* auf die Adresse an dieselbe gebunden wird. Der Zugriff auf gemeinsam benützte Variablen wird durch Barrieren synchronisiert. Trifft der Algorithmus auf eine Barriere, so geht er davon aus, daß die lokalen Daten ungültig wurden und setzt für sie neue Vorladeanweisungen ab. Weiterhin wird zwischen Lese- und Schreibzugriffen unterschieden, so daß zu schreibende Adressen *exklusiv* vorgeladen werden, d.h. alle Vorkommen dieser Daten in fremden Zwischenspeichern werden auf ungültig gesetzt.
Trotz dieser Unzulänglichkeiten konnte Mowry die Wartezeiten auf entfernte Daten zwischen 31 und 88% reduzieren.
Die angeführten Mehrprozessorprobleme treten beim SCAP-Verfahren nicht auf, denn durch die Beschränkung auf Systeme verteiltem nicht-konsistentem Speicher ist zu jedem Zeitpunkt genau bekannt, welcher Prozessor welches Datum besitzt. Damit können zum Beispiel Überschneidungsprobleme überhaupt nicht auftreten.
Im Gegensatz zu Gornish, Granston und Veidenbaum beschränken sich Mowrys Untersuchungen auf die algorithmische Seite des Problems, deshalb finden sich bei ihm auch keine Resultate oder Anmerkungen über eine erhöhte Netzwerklast im Mehrprozessorfall. Dieses Problem ist durch das Vorladen in einen separaten Vorladepuffer beim SCAP-Verfahren nicht gegeben, denn es werden nur die angeforderten Daten vom Netzwerk bereitgestellt.

#### 2.3.1.3 Hybride Vorladetechiken

Hybride Vorladetechniken sind durch eine Hardwareunterstützung der softwarebasierten Techniken charakterisiert, die über einen Vorladebefehl und einen nicht-blockierenden Zwischenspeicher hinausgeht.

**Speculative Loads (Uniprozessor):** In der Arbeit von Rogers und Li werden *speculative loads* vorgeschlagen [103]. Spekulative Ladebefehle arbeiten auf Registerwortbreite und werden bereits in mehreren Architekturen wie z.B. Motorola 88100 [67], IBM RS6000 [66], Intel i860 [70] oder IA-64 [37] verwendet. Ihr Einsatz ist jedoch beschränkt, da keine Seitenfehler des Hauptspeichers abgefangen werden und sie somit nicht über Sprunggrenzen hinweg vorgezogen werden können.
Dieses Problem versuchen Rogers und Li mit Statusbits für jedes Register in den Griff zu bekommen, die z.B. anzeigen ob ein Registerinhalt gültig ist oder ob ein Seitenfehler auftrat. Letzterer wird dann entsprechend behandelt.
Spekulative Ladeoperationen werden nun so weit wie möglich im Programmtext vorgeschoben, wie es bestehende Datenabhängigkeiten zulassen. Sie laden einen Wert aus dem Hauptspeicher in das angegebene Register am Zwischenspeicher vorbei. Mit Hilfe der Statusbits werden nachfolgende Zugriffe auf das entsprechende Register so lange blockiert, bis der Wert gültig ist. Andere Berechnungen oder Zugriffsoperationen auf den Zwischenspeicher werden nicht behindert. Die Register erfüllen damit die Funktionalität eines Vorladepuffers, wie er in Abschnitt 2.2.1 vorgestellt wurde. Ihre Ergebnisse erhielten Rogers und Li durch die Simulation eines MIPS Prozessors.
Der Vorteil spekulativer Ladebefehle liegt an der geringeren Last des Speicherbuses, da nur Werte vorgeladen werden, die tatsächlich benötigt werden. Allerdings bringt die frühe Bindung der Werte an Register einen gravierenden Nachteil: Sie erhöht die Lebenszeit der Register auf die Dauer der Speicherlatenz, da sie von der spekulativen Ladeoperation bis zum letzten Gebrauch belegt sind. Dadurch wird die notorische Registerknappheit in Prozessoren noch vergrößert, die sich in zusätzlichen Lese- und Schreiboperationen der übrigen Register äußert.[4] Rogers und Li beobachteten diesen Effekt bei Testprogrammen mit registerintensiven Schleifenkörpern. Trotzdem konnten sie die Anzahl der Zyklen pro Speicherzugriff bei den meisten ihrer Testprogramme konstant halten, sobald die Problemgröße die Kapazität des Zwischenspeichers überstieg. Bei den anderen Tests verhinderten Datenabhängigkeiten oder die geringe Bandbreite des Speicherbuses bessere Ergebnisse, da sie ein weiter entferntes Vorladen nicht zuließen bzw. die damit verbundenen Vorteile durch eine erhöhte Speicherlatenz zunichte machten.

**Fetchbuffer (Uniprozessor):** Den negativen Registerdruck versuchen Klaiber und Levy durch einen *Fetchbuffer* zu umgehen [77]. Sie schlagen ein Vorladen in einen separaten Zwischenspeicher, den Fetchbuffer vor. Dieser hat 128 Einträge von jeweils 16 Byte Länge. Zur leichteren Verwaltung der angeforderten Daten laden sie ganze Zeilen des Zwischenspeichers vor. Der Fetchbuffer ist vollassoziativ und seine Einträge können nach dem Vorladen nur gelesen werden. Die Verwaltung übernimmt eine zusätzliche Vorladeeinheit, die für das Absetzen der Anforderungen verantwortlich ist und bei Schreibzugriffen die entsprechenden Zeilen des Puffers zuvor in den Zwischenspeicher schreibt. Weiterhin wird zu jedem Speichermodul ein Vorladepuffer mit $q$ Einträgen modelliert, so daß es bei einer $n$-fachen Verschränkung des Speichers $q * n$ ausstehende Anforderungen geben kann. Die Berechnung der Vorladedistanz $\delta$ berücksichtigt diese Anzahl, denn ein Überlauf der Puffer würde in ihrem Modell weitere Vorladeanforderungen unterdrücken, das sich in schlechteren Speicherzugriffszeiten wiederspiegeln würde. Um dem logischen Mehraufwand des Fetchbuffers auf dem Prozessor gerecht zu werden, hat ihre Basisarchitektur einen Zwischenspeicher mit doppelter Zeilenlänge.
Aus Effizienzgründen nehmen sie an, daß die Schrittweite der Anforderungen innerhalb einer Schleife konstant ist. Weiterhin berechnen sie jede Adresse doppelt, um den Registerdruck,

[4] sogenannter *spill code*

der beim Zwischenspeichern der Werte entstehen würde, so gering wie möglich zu halten. Außerdem bauen sie kein Softwarefließband wie Mowry und SCAP auf, so daß in den ersten $\delta$ Iterationen auf nicht lokale Daten zugegriffen und in den letzten $\delta$ Iterationen unnötig Daten vorgeladen werden.
Ihre Resultate zeigen, daß die Anzahl der Zyklen für einen Speicherzugriff bei Programmen mit wenig Lokalität von der Verschränkung des Speichers und damit von der Anzahl der ausstehenden Vorladeoperationen abhängt. Bei Programmen mit großer Lokalität spielt die Anzahl der Speichermodule eine geringere Rolle. Hier konnten die Zyklen pro Speicherzugriff mindestens halbiert werden. Sie maßen weiterhin eine geringfügige Mehrbelastung des Speicherbuses, die jedoch asymptotisch nicht ins Gewicht fiel. Diese begründen sie zum einen mit den halb so großen Zeilen des Zwischenspeichers, die bei nicht vorgeladenen Daten die Zugriffsfehler erhöhen, und zum anderen führen die letzten $\delta$ Iterationen einer Schleife mit ihren unnötigen Anforderungen zu erhöhtem Busverkehr. Dieser Effekt tritt beim SCAP-Verfahren nicht auf, denn in der Zugriffsschleife werden nur auf Elemente im Vorladepuffer zugegriffen und keine weiteren vorgeladen.

**Streams (Uniprozessor):** Die Arbeit von McKee u.a. [88] ist die einzige hier vorgestellte Arbeit, die ein existierendes System getestet hat. Ihr Ansatz ist an Jouppis Stream-Buffer-Konzept angelehnt (vgl. 2.3.1.1). Auch sie versuchen eine hohe Speicherbandbreite auszunutzen, um Daten vor ihrem Gebrauch lokal zum Prozessor zu bringen. Ihre *Ströme* werden durch den Übersetzer erkannt und in Befehle für den *Stream Memory Controller (SMC)* übersetzt. Dies erspart dem Prozessor Adreßberechnungen zur Laufzeit. Er initialisiert am Schleifenanfang die Ströme und greift in jeder Iteration auf das vorderste Element des Stroms zu. Die Aufgaben des SMC sind

1. die geforderten Daten so schnell wie möglich aus dem Hauptspeicher zu lesen, und

2. die Daten verzögerungsfrei an den Prozessor weiterzugegeben.

Für die Tests erweiterte McKee einen Intel-i860 mit einer Taktfrequenz von 40 MHz mit einem ebensoschnellen SMC-Board. Der SMC-Prototyp hatte vier verschiedene Puffer zur Verwaltung der Ströme mit jeweils 16 Einträgen. Die gemessenen Ergebnisse bestätigten die Simulationen, die eine Warmlaufzeit des SMC prognostizierten, bevor er seine maximale Durchsatzrate erreicht. Dies ist auf die mehreren Puffer zurückzuführen, die der SMC zuerst laden muß, bevor er wieder beim ersten Puffer fortfahren kann.
Die durchgeführten Tests konzentrierten sich leider nur auf die Erhöhung der Speicherbandbreite des Prozessors, so daß Daten über die erzielte Laufzeitbeschleunigung nicht vorliegen. Ein Nachteil dieses Ansatzes ist die Beschränkung auf vektorisierbare Schleifen, denn Schleifen mit Speicherzugriffen, die ihre Schrittweite dynamisch verändern, können damit nicht optimiert werden, wie dies im SCAP-Verfahren möglich ist.

### 2.3.2 Vielfädige und entkoppelte Prozessoren

Das Prinzip der vielfädigen (*multithreaded*) Prozessoren basiert auf der Idee, die Speicherzugriffszeiten durch schnelle Kontextwechsel zwischen verschiedenen Instruktionsströmen zu verdecken. Die Bereitstellung der multiplen Kontexte wurde ursprünglich durch die Hardware

übernommen, doch gibt es auch Bestrebungen dieses Verhalten mit handelsüblichen Prozessoren in Software nachzubilden. Bei entkoppelten Prozessoren wird die zentrale Ausführungseinheit in eine Speicherzugriffs- und eine arithmetische Einheit zerlegt. Durch das Vorauseilen der Speicherzugriffseinheit gegenüber der Berechnungseinheit können erhöhte Speicherzugriffszeiten toleriert werden.

#### 2.3.2.1 Fein- und blockgranulare vielfädige Prozessoren

Je nach Wechselstrategie, mit der zwischen den einzelnen Kontexten gewechselt wird, unterscheidet man zwischen *feingranularen* oder *blockgranularen* vielfädigen Prozessoren. Bei feingranularen Prozessoren wird bei jedem Prozessortakt ein Befehl eines anderen Kontrollfadens in die Prozessorpipeline eingefüttert (*Cycle-by-Cycle-Interleaving-Technik*). Der Nachteil dieses Verfahrens liegt zum einen in der geringen Leistung, falls die Applikation nur wenige Kontexte als Last zur Verfügung stellt, und zum anderen im Hardwareaufwand, der durch die Bereitstellung einer ausreichend großen Anzahl von Kontexten entsteht. Denn geht man von den Leistungsdaten einer Cray T3D (150MHz Prozessortakt und $4\mu s$ Zugriffszeit auf nicht-lokale Datenelemente) aus, so müßte ein feingranularer Prozessor über 600 Kontexte unterstützen [120].
Bei blockgranularen Prozessoren werden die Befehle eines Kontextes solange aufeinanderfolgend ausgeführt, bis ein Ereignis eintritt, das zu Wartezeiten führt (*Block-Multithreading-Technik*). Erst dann wird auf einen anderen Kontext gewechselt. Dieses Ereignis kann ein entfernter Speicherzugriff oder wie beim Sparcle-Prozessor [4] ein Cache-Fehlzugriff oder eine fehlgeschlagene Synchronisation sein. Der Nachteil dieser Technik ist, daß solche Ereignisse erst spät in der Pipeline erkannt werden und nachfolgende bereits in der Pipeline vorhandene Befehle nicht verwendet werden können. Daraus resultiert z.B. beim Sparcle-Prozessor ein Kontextwechselaufwand in der Größenordnung von 14 Takten.
Der Vorteil des SCAP-Verfahrens gegenüber vielfädigen Prozessoren liegt darin, daß die Virtualisierung nicht nur zur Latenzzeitverbergung zum Zeitpunkt des Zugriffs auf ein Datenelement verwendet wird, sondern daß die Virtualisierung dazu dient, Datenelemente vor dem eigentlichen Zugriffszeitpunkt gezielt vorzuladen. Dadurch erhält man die erforderliche Menge an entfernten Datenzugriffen, mit der erst die Verdeckung großer Latenzzeiten möglich wird.
Architekturen mit feingranularen Prozessoren sind z.B. die SB-PRAM der Universität Saarbrücken [1, 11], Tera-MTA [8, 7], HEP [108], Horizon [79, 116]. Blockgranulare Prozessoren sind z. B. Sparcle [2, 3], EMC-Y [111] und Rhamma [74].

#### 2.3.2.2 Emulierte vielfädige Prozessoren

Anstatt mehrere Kontexte allein durch die Hardware bereitzustellen, versuchen Grävinghoff und Keller dieses Verhalten mit Software auf einem herkömmlichen Prozessor zu emulieren [53, 54, 55]. Durch den Verzicht auf Spezialhardware können sie auf existierende Architekturen aufsetzen und damit vom stetigen Leistungszuwachs moderner Prozessoren profitieren. Für ihre Tests wählten sie Alpha Architektur [35] von Digital Equipment, die neben einem einfach zu speichernden Registersatz auch einen für die Parallelverarbeitung wichtigen großen virtuellen Adreßraum bereitstellt. Weiterhin ist die Alpha-Architektur (21064, 21164 und 21264) seit ihrer Einführung im Jahre 1992 laut den SPEC-Testprogrammen die leistungsfähigste Prozessorfamilie.

Für die Emulation wird für jeden Instruktionsstrom ein Kontext bereitgestellt, der den Programmzähler, den Status, Kontrollinformation und die Register hält. Jede Instruktion wird nun in einem simulierten Umfeld ausgeführt, aus dem die benötigten Register und Kontrollinformationen gelesen werden. Beim Vergleich von einer nichtemulierten zu einer emulierten Ausführung mit einem Kontext, zeigten die Simulationen eine Verdreifachung der Zyklen pro Instruktion und eine sechsfache Vergrößerung des Programmtextes.
Werden für die Simulationen Systemparameter der Cray T3E vorausgesetzt (180-900 Takte Latenzzeit), dann läuft die Emulation schneller (Latenzzeit von $\geq 350$ Takten und mehr als 16 Instruktionsströmen pro Prozessor), wenn mindestens jeder vierte Speicherzugriff entfernt ist. Da sich in einer parallelen Applikation Bereiche mit und ohne Kommunikation abwechseln, bleibt es bisher offen, wie sich die Emulation bei konkreten Problemen verhält.
Im Vergleich zum SCAP-Verfahren, das zur parallelen Ausführung einer Applikation nur den Mehraufwand der Vorladeanweisungen hinzufügt, wird beim Emulieren der Vielfädigkeit jede Instruktion mit einem dreifachen Ausführungsaufwand belegt, der sich erst bei größeren Latenzzeiten amortisiert. Weiterhin wird bei SCAP die Emulation konzeptioneller Prozessoren (Kontexte) durch die Virtualisierung erreicht, wobei sich die Kontextwechselzeit auf die Ausführung der Schleifenkontrollen beschränkt. Daher kann man davon ausgehen, daß das SCAP-Verfahren bei der Ausführung einer datenparallelen Applikation effizienter ist.

#### 2.3.2.3 Entkoppelte Prozessorarchitekturen

Prozessoren mit entkoppelten Speicherzugriffs- und Ausführungseinheiten (*Decoupled Access/Execute Architecture*) trennen einen sequentiellen Instruktionsstrom in einen Strom für Speicherzugriffe und einen für arithmetische Operationen. Diese Instruktionsströme werden von den getrennten Ausführungseinheiten unabhängig voneinander ausgeführt, wobei über interne Puffer Daten ausgetauscht werden. Durch diese Puffer kann die Speicherzugriffseinheit der arithmetischen Einheit vorauseilen und Daten weit vor ihrem Gebrauch in den Prozessor laden. Somit verkürzt sich die Wartezeit der Berechnungseinheit beim Zugriff der Daten. Auf diese Weise können Speicherzugriffszeiten toleriert und bei ausreichendem Vorlauf der Zugriffseinheit sogar verdeckt werden, was im Vergleich zu herkömmlichen Prozessoren zu einer Beschleunigung der gemessenen Applikationen führt [109, 122].
Probleme treten bei entkoppelten Prozessorarchitekturen bei der Ausführung von Kontrollstrukturen, insbesondere bei bedingten Sprüngen auf, da dann die Zugriffseinheit auf Ergebnisse der hinterherlaufenden Berechnungseinheit wareten muß.
Die Arbeitsweise des SCAP-Verfahren ist dem der entkoppelten Prozessoren recht ähnlich. Das Netzwerk läuft durch die Vorladeoperationen des Prozessors diesem voraus, so daß letzterer auf entfernte Elemente ohne Wartezeit zugreifen kann. Das Problem der Synchronisation bei bedingten Verzweigungen existiert beim SCAP-Verfahren nicht, denn durch die Virtualisierung, sind die entfernten Datenelemente genau beschrieben.
Vertreter dieser Prozessorarchitektur sind die Prototypen PIPE [49], ZS-1 [110] und ACRI-1 [117].

### 2.3.3 Maschinenmodell LogP und planbare Algorithmen

Das erstmals von Culler et al. vorgestellte LogP-Modell [36], beschreibt eine nachrichtengekoppelte Parallelrechnerarchitektur mit Hilfe der vier Parameter

**Latenzzeit L** (*latency*) Sie ist eine obere Zeitschranke für das Versenden eines Datenelements (oder einer kleinen Menge von Datenelementen) von einem Quell- zu einem Zielprozessor.

**Kommunikationsaufwand o** (*overhead*) Sie ist die Zeit, die ein Prozessor benötigt, um eine Nachricht zu senden oder zu empfangen. In dieser Zeit kann der Prozessor keine weitere Arbeit verrichten.

**Kommunikationstotzeit g** (*gap*) Sie ist die minimale Zeitspanne zwischen aufeinanderfolgenden Datenübertragungen von und zum Prozessor.

**Prozessoren P** Die Anzahl der beteiligten Prozessoren.

Die ersten drei Parameter müssen für die Instanziierung des LogP-Modells für eine gegebene Architektur gemessen werden, damit sie zur Berechnung der Kommunikationskosten herangezogen werden können. Mit Hilfe dieses Wissens wird schließlich ein Kommunikationsplan erstellt, der die anfallenden Kommunikationskosten einer datenparallelen Anwendung minimiert. In diesem Kommunikationsplan werden Sende- und Empfangsoperationen so weit wie möglich zeitlich voneinander getrennt, so daß die Latenzzeit (fast) vollständig mit anderen Operationen überdeckt werden kann. In diesem Zusammenhang sei vor allem auf die theoretischen Arbeiten von Loewe und Zimmermann [82, 123, 124] verwiesen.
Ein Nachteil des LogP-Modells ist die Beschränkung auf planbare Algorithmen (*oblivious programs*), d.h. Algorithmem deren Kommunikationsverhalten zur Übersetzungszeit bekannt ist. Dies hat zur Folge, daß dynamische Zugriffsmuster nicht behandelt werden können. Weiterhin müssen für die statische Analyse die Problemgröße und die Anzahl der an der Berechnung beteiligten Prozessoren bekannt sein. Weitere Nachteile sind die Dauer der Analyse, die der Ausführung des Programms auf einem Uniprozessorsystem gleichkommt, und die Programmgröße, die durch Schleifenausrollen und das Ausfalten von Folgen von Verzweigungen exponentiell wachsen kann.
Eisenbiegler übertrug diesen Ansatz in einen optimierenden Übersetzer [38]. Er benutzte die Abschätzungen des LogP-Modells, um zeitintensive Kommunikation mit redundanter Berechnung zu ersetzen, d.h. verschiedene Prozessoren führen identische Berechnungen aus und sparen dadurch Kommunikation. Laufzeittests mit und ohne redundante Berechnung zeigen eine Zeitersparnis von bis zu Faktor sechs. Er hat jedoch auch Probleme mit den erwähnten langen Übersetzungszeiten und möglicherweise exponentiell wachsenden Programmgrößen.
Der Vorteil von SCAP gegenüber dem LogP-Modell liegt zum einen an der Übersetzung dynamischer Kommunikationsmuster und zum anderen an der geringeren Übersetzungszeit. Denn da in das SCAP-Transformationsschema nur das Wissen über nicht-lokale Datenelemente eingeht, welches bereits in jedem parallelisierenden Übersetzer bei der Generierung der Kommunikation vorliegt, ist der Zusatzaufwand für die Erstellung des Softwarefließbandes konstant.

### 2.3.4 Programmiermodell BSP

Das BSP-Modell von Valiant [119] und seine Varianten [6, 48, 33, 95, 29, 39, 87] bestehen aus einer beschränkten, asynchron zueinander arbeitenden Anzahl $P$ von Komponenten, die aus einem Prozessor, lokalem Speicher und einer Netzwerkschnittstelle zusammengesetzt sind. Die Kommunikation zwischen den Komponenten findet über das Netzwerk mit Hilfe von entfernten Speicherzugriffen statt.

Der Ablauf des Programms wird in Superschritte (*supersteps*) unterteilt. Am Ende jedes Superschrittes findet durch eine Synchronisationsbarriere eine Synchronisation aller Prozessoren statt. Nach einer solchen Synchronisation sind alle entfernten Speicherzugriffe, die vor der Barriere angestoßen wurden, auf allen lokalen Speichern und für alle Komponenten sichtbar durchgeführt. Innerhalb der Superschritte tauscht jeder Prozessor nach der Berechnung $V \geq log(P)$ Datenelemente mit den anderen Prozessoren aus. Valiant zeigt, daß sich diese $P * log(P)$ Kommunikationsoperationen bei geeignetem Netzwerk und einer geeigneten Wegwahlfunktion mit hoher Wahrscheinlichkeit in $O(log(P))$ ausführen lassen. Verfügt ein Programm über eine genügend große Virtualisierung (*parallel slackness*), so läßt sich der BSP-Algorithmus ohne wesentliche Effizienzverluste ausführen. Valiants Vorgehensweise ist sehr theoretisch und er führt nicht aus, woher ein Algorithmus die Virtualisierung bekommt oder wie man die Supersteps konstruiert.
Die Unterteilung einer parallelen Applikation durch Superschritte, die durch Synchronisationsbarrieren von einander getrennt sind, entspricht der Einteilung einer SCAP-Applikation, bei der ebenfalls vor der Kommunikation die Konsistenz der verteilten Daten sichergestellt werden muß. Auf der anderen Seite überläßt das BSP-Modell die Art und Weise der entfernten Speicherzugriffe dem Programmierer, so daß für die Softwarefließbänder des SCAP-Verfahrens im BSP-Modell kein Äquivalent existiert. Somit kann das SCAP-Verfahren als Instanz des BSP-Modells gesehen werden, das konkrete Angaben über die Durchführung der Kommunikation gibt.

### 2.3.5 Übersetzungstechnik: Inspector-Executor

Im *Inspector-Executor*-Ansatz wird eine parallele Schleife in eine Inspektor- und eine Executor-Schleife aufgeteilt. Der Inspektor untersucht die Schleife auf Datenabhängigkeiten oder auf nicht-lokale Datenzugriffe, die zur Übersetzungszeit nicht festgestellt werden können und somit eine parallele Ausführung der Schleife verhindern oder nur sehr ineffizient ermöglichen. Der Exekutor führt anhand der Analyse des Inspektors die Schleife parallel aus. Das *Inspector-Executor*-Modell wird vor allem in parallelisierenden Übersetzern, bei der effizienten Abarbeitung geschachtelter Schleifen sowie der Aggregation von Kommunikationsoperationen [104, 78, 106, 80, 118] eingesetzt.
Der Nachteil des *Inspector-Executor*-Modells im Bereich der SPMD-Programmierung liegt im Zeitaufwand für das Ausführen des Inspektors. Dort müssen vor allem lokale und nicht-lokale Datenzugriffe erkannt und Kommunikationspläne erstellt werden, mit denen schließlich die nicht-lokalen Datenelemente zu den jeweiligen Prozessoren befördert werden. Im Laufe dieser Analyse werden Datenstrukturen aufgebaut und gegebenenfalls Daten ausgetauscht, damit Quell- und Zielprozessoren für den Kommunikationsplan bestimmt werden können. Dies führt zu einem erheblichen Zeitverlust, da ein Großteil der Laufzeit im Inspektor verbracht wird.
Um die zeitaufwendige Analyse des Inspektors so wenig wie möglich ausführen zu müssen, werden mit Hilfe von Datenflußanalyse solche Schleifen markiert, bei denen sich das Kommunikationsmuster zur Laufzeit nicht ändert. Für diese Schleifen muß der Inspektor nur noch beim ersten Erreichen ausgeführt werden. Bei allen weiteren Vorkommen wird der bereits erstellte Kommunikationsplan verwendet und somit Laufzeit gespart. Dieser Ansatz der Kommunikationsplanwiederverwendung (*schedule reuse*) kann automatisch geschehen oder durch Programmiererannotationen unterstützt werden [101, 16, 13].
Das Softwarefließband des SCAP-Verfahrens kann bei dynamischen Kommunikationsmustern

im Prinzip als Inspektor aufgefaßt werden, denn in der Vorladeschleife (vgl. 2.2.2) wird für jedes Datenelement die Lokalität überprüft und gegebenenfalls durch eine Vorladeoperation angefordert. Im Vergleich zum Inspektor ist der Laufzeitaufwand jedoch geringer, da durch die einseitige Kommunikation in Architekturen mit gemeinsamen Adreßraum kein Kommunikationsplan erstellt werden muß und die Vorladeoperationen effizient ausgeführt werden können.

### 2.3.6 Überblick und Einordnung

Tabelle 2.4 gibt einen Überblick über die verwandten Arbeiten auf dem Gebiet der latenzzeitverbergenden Mechanismen und faßt die Unterschiede der einzelnen Techniken zum SCAP-Verfahren zusammen. Die verwendeten Abkürzungen werden in Tabelle 2.5 erklärt.

Tabelle 2.4: Vergleich von VSCAP und SCAP mit anderen Vorladetechniken

| Arbeit | Art der Technik | Rechner-architektur | Hardware-Aufwand | Größe der Vorladeeinheit | Vorladedistanz | Verhalten bei großer Latenz | Verhalten bei irreg. Zugriffen |
|---|---|---|---|---|---|---|---|
| Smith (OBL) | HW | U | G | Z | Z | - | - |
| Baer, Chen (RPT) | HW | U | Z | Z | DY | + | - |
| Jouppi (Stream Buffer) | HW | U | P | Z | 4Z | + | - |
| Gornish u.a. (Block-Vorladeanweisung) | SW | S | G | SB | DA/KO | 0 | * |
| Mowry (SUIF) | SW | U/S | G | Z | AR | + | 0 |
| Rogers, Li (spec. Loads) | HY | U | G | R | DA | 0 | * |
| Klaiber, Levy (Fetchbuffer) | HY | U | Z | Z | DA | + | * |
| McKee (Streams) | HY | U | P | R | 16R | 0 | - |
| versch. Autoren (vielfädige Proz.) | HW | M | S | R | HK | + | + |
| Grävinghoff, Keller (emulierte vielf. Proz.) | SW | M | K | R | SK | 0 | 0 |
| versch. Autoren (entkoppelte Proz.) | HY | U | S | R | DY/KO | + | + |
| Culler u.a. (LogP, planbare Alg.) | S | N | K | - | - | + | - |
| Valiant (BSP) | S | M/S | K | - | - | 0 | 0 |
| versch. Autoren (Inspector-Executor) | S | M/N/S | K | - | - | + | 0 |
| Warschko (SCAP) | HY | M | P/Ü | R | $C_V$*R | + | + |
| Müller (VSCAP) | HY | M | P/Ü | R | $C_V$*R | ++ | ++ |

Tabelle 2.5: Verwendete Abkürzungen

| | |
|---|---|
| **Art der Technik**<br>HW: Hardware<br>SW : Software<br>HY : Hybrid | **Rechner Architektur**<br>U : Uniprozessor<br>N : Nachrichtengekoppelte Mulitproz.<br>M: Multiproz. mit globalem Adreßraum<br>S : Speichergekoppelte Multiproz. |
| **Hardware Aufwand**<br>G: Gering<br>Z : Zweiten Zwischenspeicher<br>S : Spezialprozessoren<br>P : Puffer außerhalb des Prozessors<br>Ü: Überlappendes Netzwerk<br>K: Keiner | **Größe der Vorladeeinheit**<br>Z : Zeile des Zwischenspeichers<br>SB: Speicherbereich<br>R : Registerwort |
| **Vorladedistanz**<br>R : Registerwort<br>Z : Zeile des Zwischenspeichers<br>DY: Dynamisch<br>DA: Datenabhängig<br>KO: Kontrollabhängig<br>AR: Architekturabhängig<br>HK: Anzahl Hardware-Kontexte<br>SK : Anzahl Software-Kontexte<br>$C_V$ : Größe Vorladepuffer | **Bewertung**<br>++: sehr gut<br>+ : gut<br>0 : neutral<br>- : schlecht<br>* : keine Angaben |

## 2.4 Zusammenfassung

In diesem Kapitel wurden die grundlegenden Zusammenhänge zwischen Parallelrechnerarchitektur und Programmiertechnik vermittelt. Die Vorstellung des SCAP-Verfahrens diente als Grundlage für die Vorarbeiten auf der Cray T3E und vermittelte eine Übersicht für das grundlegende Verständnis der Arbeitsweise eines überlappenden Netzwerkes.
Die Abgrenzung des SCAP-Verfahrens zu existierenden Vorlademechanismen motivierte den Einsatz eines Vorladepuffers, mit dem erstens der Prozessor vom Netzwerk entkoppelt wird und zweitens im Gegensatz zum Vorladen von Cachezeilen nur die Daten vorgeladen werden, die auch später zur Berechnung benötigt werden. Weiterhin ermöglicht das softwaregesteuerte Vorladen ein gezieltes Anfordern von dynamischen Zugriffen, die in hardwarebasierten Vorladetechniken nicht oder nur schlecht behandelt werden können.

# Kapitel 3

# Das Vektor-SCAP Modell

Dieses Kapitel stellt die Erweiterung des SCAP-Modells um Vektorbefehle vor: *VSCAP*. Die Vorstellung beinhaltet neben der Einführung (Abschnitt 3.1) in die Terminologie und Arbeitsweise von VSCAP auch Effizienzbetrachtungen (Abschnitt 3.2). Mit letzteren kann nicht nur der Grad der Latenzzeitverbergung von VSCAP gegenüber einer blockierenden Kommunikation für eine gegebene Parallelrechnerarchitektur bestimmt werden, sondern es können auch generelle Aussagen über die zu erwartende Reduktion der Kommunikationskosten beim Einsatz von Vektorbefehlen der Länge $L$ gemacht werden, wenn statt SCAP die Erweiterung VSCAP eingesetzt wird. Im Zuge der Effizienzbetrachtungen wird auch ein Bereich von sinnvollen Vektorlängen berechnet, in dem eine maximale Prozessorauslastung mit optimalem Netzwerkdurchsatz erreicht werden kann.

## 3.1 Das VSCAP-Modell

Dieser Abschnitt stellt das VSCAP-Verfahren vor und behandelt in 3.1.1 zunächst grundlegende Fragen, die sich beim Einsatz von Vektorbefehlen stellen. In 3.1.2 werden einige konzeptionelle Entscheidungen bei der Implementierung von VSCAP in einem optimierenden Übersetzer getroffen. Danach stellt 3.1.3 die Modellparameter vor, mit denen die späteren Effizienzaussagen gewonnen werden. An die Vorstellung der Arbeitsweise von VSCAP in 3.1.4, die nicht nur die theoretische Sichtweise sondern auch eine mögliche Implementierung eines VSCAP Fließbandes beinhaltet, reiht sich 3.1.5 mit den Definitionen von Kommunikationsmuster, Vorlade- und Vektorstrategie, mit denen die Differenzierung von *allgemeiner* Vorladestrategie zu *spezieller* Vektorstrategie ermöglicht wird.

### 3.1.1 VSCAP: Eine Erweiterung des SCAP-Verfahrens

Die Erweiterung des SCAP-Verfahrens liegt im Prinzip darin, $L$ verschiedene Elemente in einer einzigen Operation vorzuladen und darauf zuzugreifen. Dieses Vorgehen wird *Vektor-SCAP mit Vektorlänge $L$* oder kurz *VSCAP* bezeichnet.
Der Einsatz der Vektorbefehle hat mehrere Konsequenzen auf die bisher vorgestellte überlappende Netzwerkarchitektur:

**Länge der Vektoren:** Es wird davon ausgegangen, daß die Größe des Vorladepuffers die Vektorlänge bei weitem übersteigt: $C_V \gg L > 1$, mit $L = 2^n$ und $n \leq 5$. Denn zu große Vektoren erhöhen den Druck des Prozessors auf das Netzwerk, das die einzelnen

Kommunikationsaufträge nicht mehr rechtzeitig vor dem Zugriff bereitstellen kann, was zu längeren Wartezeiten für den Prozessor und somit zur Verlängerung der Kommunikationszeit führt. Die Berechnung der optimalen Vektorlänge zeigt 3.2.4. Dort werden die hier gemachten Annahmen bestätigt.

**Funktionalität der Netzwerkschnittstelle:** Die Netzwerkschnittstelle muß bei einem Vektorbefehl in der Lage sein, die der Startadresse nachfolgenden $L-1$ Adressen berechnen zu können. Diese Adressen $A_i$ könnten z.B. am einfachsten durch das Aufaddieren einer konstanten Schrittweite $c$ generiert werden:

$$A_i = A_{i-1} + c, \text{ mit } 1 \leq i \leq L-1. \tag{3.1}$$

Dynamische Schrittweiten $c_i$ oder komplexere Berechnungsvorschriften würden die Antwortzeit des Netzwerkes weiter vergrößern, was sich zum Vorhaben der Latenzzeitverbergung kontraproduktiv erweist. Deshalb wird im folgenden davon ausgegangen, daß die Abstände einzelner Elemente innerhalb eines Vektors konstant sind und sich die Adressen nach (3.1) berechnen lassen.

**Erweiterung des Befehlssatzes:** Weiterhin muß der Netzwerkschnittstelle neben der Startadresse des Vektors die Berechnungsvorschrift mitgeteilt werden. Dies ist im einfachsten Fall der Abstand $c$ der einzelnen Vektorelemente. Dies kann Mehraufwand für den Prozessor bedeuten, der nun nicht nur den Vorladebefehl sondern auch noch weitere Vektorparameter übermitteln muß. Deshalb lohnen sich Vektorbefehle erst, wenn sie schneller als $L$ Einzelelementbefehle ausgeführt werden können, d.h. wenn $t_{\mathrm{v}}^L \leq L * t_{\mathrm{v}}$ gilt, mit $t_{\mathrm{v}}$ und $t_{\mathrm{v}}^L$ als Zeitspannen für das Absetzen eines einelementigen bzw. eines Vektorvorladebefehls. Andererseits können Vektorbefehle auch nicht schneller als Einzelelementbefehle abgesetzt werden $t_{\mathrm{v}} \leq t_{\mathrm{v}}^L$, so daß folgendes gilt:

$$t_{\mathrm{v}} \leq t_{\mathrm{v}}^L \leq L * t_{\mathrm{v}}$$

Beim Zugriff mit einem Vektorbefehl wird nachfolgend davon ausgegangen, daß der Prozessor so lange blockiert wird, bis alle Elemente eines Vektors im Vorladepuffer liegen.

**Größe des Vorladepuffers:** Eine weitere Konsequenz ergibt sich für die Größe $C_V$ des Vorladepuffers, der nun mindestens

$$C_V \geq L * \frac{T_{\mathrm{Latenz}}}{t_{\mathrm{v}}^L} \tag{3.2}$$

Einträge fassen sollte, da der Prozessor zur Verdeckung der Latenzzeit jetzt $L$-mal mehr Elemente vorladen kann. Mit der vorgeschlagenen Größe kann selbst bei einem langsamen Netzwerk $t_{\mathrm{v}} < t_{\mathrm{n}}$ der Zugriff auf das erste nicht lokale Element überbrückt werden. Der Vorladepuffer wird aber etwas größer als in (3.2) gewählt, damit er auch die Wartezeit auf den ersten Vektor überbrücken kann, da auf einen Vektor per Vektorbefehl erst wartezeitfrei zugegriffen werden kann, wenn alle $L$ Vektorelemente vorhanden sind.

Wird der Vorladepuffer mit Vektorbefehlen gefüllt, so erhält das Netzwerk zum Zeitpunkt $t_{\mathrm{v}}^L$ den ersten Vektorauftrag. Nach $t_{\mathrm{v}}^L + (L-1) * t_{\mathrm{n}}$ Schritten hat das Netzwerk die erste Vektorvorladeanweisung entgegengenommen, denn es bekommt zum Zeitpunkt $t_{\mathrm{v}}^L$ den Vektorbefehl und kann danach alle $t_{\mathrm{n}}$ Schritte die restlichen $L-1$ Vektorelemente verarbeiten. Der Prozessor kann damit zum Zeitpunkt

$$T_{\mathrm{Netzwerk}} = T_{\mathrm{Latenz}} + t_{\mathrm{v}}^L + (L-1) * t_{\mathrm{n}}$$

wartezeitfrei darauf zugreifen. Der Vorladepuffer ist dann groß genug, wenn das Füllen mit Vektorbefehlen mindestens so lang wie die Antwortzeit des Netzwerkes dauert:

$$\frac{C_V - L}{L} * t_{\mathrm{v}}^L \geq T_{\mathrm{Latenz}} + t_{\mathrm{v}}^L + (L-1) * max(t_{\mathrm{n}}, t_{\mathrm{v}}) \tag{3.3}$$

In obiger Gleichung werden nur $C_V - L$ Einträge des Vorladepuffers gefüllt, denn die restlichen $L$ Einträge sind für die maximal $m = L-1$ nicht-lokalen Datenzugriffe reserviert, mit denen kein Vektor mehr gefüllt werden konnte. Der Term $max(t_{\mathrm{n}}, t_{\mathrm{v}})$ beachtet, daß der Arbeitstakt $t_{\mathrm{n}}$ des Netzwerkes durch die eventuell längere Vorladedauer $t_{\mathrm{v}}$ bei einelementigen Vorladebefehlen des Prozessors vergrößert wird. Dieser Fall wird betrachtet, wenn der Vorladepuffer mit einelementigen Operationen gefüllt aber mit Vektorbefehlen geleert wird. Dann gilt mit der Zeitbeschränkung der Vektorbefehle durch $t_{\mathrm{v}} \leq t_{\mathrm{v}}^L \leq L * t_{\mathrm{v}}$ insbesondere, daß das Füllen des Vorladepuffers mit einelementigen Vorladebefehlen länger dauert als mit Vektorbefehlen:

$$(C_V - L) * t_{\mathrm{v}} \geq T_{\mathrm{Latenz}} + t_{\mathrm{v}} * (L-1) * max(t_{\mathrm{n}}, t_{\mathrm{v}}) \tag{3.4}$$

**Verwaltung des Vorladepuffers:** Beim SCAP-Verfahren konnten die Einträge des Vorladepuffers über die Adresse des nicht-lokalen Datenzugriffs angesprochen werden, d.h. `prefetch(B[i])` und `A[j] := access(B[i])` opererierten auf demselben Eintrag $B[i]$ des Vorladepuffers. Für Vektorbefehle ist dies nun nicht mehr ohne weiteres möglich, denn Vektorzugriffe benötigen neben den Quelladressen auch die Zieladressen, d.h. in einen Vektor können die Elemente $B[i + s * \mathrm{Inc_B}]$, mit $0 \leq s \leq L-1$, vorgeladen worden sein, die an Adressen $A[j + s * \mathrm{Inc_A}]$ geschrieben werden sollen. Dabei können $i \neq j$ und $\mathrm{Inc_A} \neq \mathrm{Inc_B}$ sein. Eine Möglichkeit der Adressierung der Einträge des Vorladepuffers mit Vektorbefehlen wäre die Übergabe beider Schrittweiten, d.h. ein Vektorbefehl der Form

`vector_access(A[i],`$\mathtt{Inc_A}$`,B[j],`$\mathtt{Inc_B}$`)`

der die Inhalte der Vorladepuffereinträge $B[i + s * \mathrm{Inc_B}]$ an die Adressen $A[j + s * \mathrm{Inc_A}]$ kopiert. Diese Funktionalität ist für affine nicht-lokale und affine lokale Datenzugriffe ausreichend, aber dieses Schema scheitert bei der Adressierung von Einträgen des Vorladepuffers, die aus nicht-lokalen Datenzugriffen mit variabler Schrittweite resultieren, denn dann reicht $\mathrm{Inc_B}$ als einziger Parameter zur Berechnung der Adressen nicht

aus. Da auch in diesem Fall Vektorzugriffe möglich sein sollen, wird zusätzlich eine Reihenfolge auf die Einträge des Vorladepuffers definiert, d.h. der Vorladepuffer bekommt implizite Kennungen $0 \ldots C_V - 1$. Aufeinanderfolgende Vorladebefehle belegen aufsteigende Einträge innerhalb des Vorladepuffers, d.h. `prefetch(B[i])` steht an Position $p$ und `prefetch(B[i+Inc`$_B$`])` wird an Position $p+1$ abgelegt. Dabei bestimmt die Reihenfolge der Vorladebefehle die Reihenfolge innerhalb des Vorladebuffers. Vektorvorladebefehle adressieren den Vorladepuffer nach dem gleichen Prinzip, d.h. `vector_prefetch(B[i],Inc`$_B$`)` füllt die Einträge $p \ldots p + L - 1$ mit den Inhalten der Adressen $B[i + s * \mathrm{Inc_B}]$. Im weiteren wird davon ausgegangen, daß der Programmierer oder der Übersetzer ein Überlaufen am Ende des Vorladepuffers vermeidet.

Ein Vektorzugriff `vector_access(A[j],Inc`$_A$`,B[i],Inc`$_B$`)` bestimmt nun durch $B[i]$ das erste Vektorelement an der Stelle $p$ und kopiert dieses und die folgenden $L-1$ Einträge $p+1 \ldots p+L-1$ an die Adressen $A[j + s * \mathrm{Inc_A}]$.

Neben diesen Netzwerkerweiterungen gibt es auch bei der Implementierung von VSCAP innerhalb eines Übersetzers Änderungen gegenüber SCAP.

### 3.1.2 Einbettung von VSCAP in eine automatische Übersetzung

Warschko kombinierte den Zugriff und die Benutzung der Datenelemente innerhalb der Berechnungsschleife. Dies kann bei Vektorzugriffen jedoch nicht mehr geschehen, denn Vektorbefehle kopieren Daten und führen keinerlei Operationen auf diesen aus. Anstatt nun VSCAP wie SCAP direkt in die Berechnungsvorschrift der datenparallelen Operation zu integrieren, wird die datenparallele Schleife in einen Kommunikations- und einen Berechnungsteil aufgespalten. Im Kommunikationsteil werden die entfernten Datenelemente durch VSCAP in lokale Felder kopiert, die dann im Berechnungsteil als Operanden verwendet werden. Dieses Vorgehen erleichtert auch die Integration von VSCAP in ein bestehendes datenparalleles Übersetzungssystem, da dort jetzt nur noch die Kommunikationsoperationen lokal durch die Datenfließbänder von VSCAP ersetzt werden müssen.
So wird folgendes Programmfragment

```
FORALL i = 1 TO N DO
  A[i] := A[i] + B[q(i)];
END FORALL
```

in eine Kommunikation und eine Berechnung aufgeteilt, wobei letztere nur noch auf lokale Daten zugreift

```
  /* Kommunikationsschleife */
FORALL i = 1 TO N DO
  TMP[i] := B[q(i)];
END FORALL

  /* Berechnungsschleife */
FORALL i = 1 TO N DO
  A[i] := A[i] + TMP[i];
END FORALL
```

Die weiteren Übersetzungsschritte bei der Kommunikationsgenerierung bauen auf der *Kommunikationsschleife* auf, so daß dort unabhängig von der übrigen Berechnung die verschiedenen Kommunikationsstrategien eingesetzt werden können.
Das VSCAP-Transformationsschema verdeckt anfallende Kommunikationszeiten und steht somit ganz am Ende der Optimierungskette. Es arbeitet orthogonal zu bestehenden Optimierungstechniken, wie z.B. der Lokalitätsoptimierung von Daten [99], so daß es den Einsatz dieser Verfahren im Hinblick auf die Reduktion der Kommunikationszeit nicht negiert sondern sinnvoll erweitert.

### 3.1.3 Modellparameter

Die für die Modellierung der Systemlaufzeiten eines blockierenden Netzwerkes, SCAP und VSCAP notwendigen Parameter sind in Tabelle 3.1 zusammengefaßt. Zur Vollständigkeit sind auch die Parameter aus Tabelle 2.3 mit aufgeführt.

Tabelle 3.1: VSCAP Modellparameter

| **Parameter** | **Beschreibung** |
|---|---|
| | **Applikationsparameter** |
| $t_{\mathrm{v}}^1 = t_{\mathrm{v}}$ | Zeit für einelementige Vorladeanweisung |
| $t_{\mathrm{v}}^L$ | Zeit für vektorwertige Vorladeoperation mit Vektor der Länge $L$ |
| $t_{\mathrm{z}}^1 = t_{\mathrm{z}}$ | Zeit für einelementigen Zugriff |
| $t_{\mathrm{z}}^L$ | Zeit für vektorwertigen Zugriff mit Vektor der Länge $L$ |
| $K$ | Anzahl der nicht-lokalen Zugriffe |
| $m$ | Rest der Division von $K$ durch $L$: $m = K \mod L$ |
| | **Architekturparameter** |
| $L$ | Vektorlänge |
| $C_V$ | Größe des Vorladepuffers |
| $t_{\mathrm{s}}$ | Schleifenaufwand |
| $T_{\mathrm{Latenz}}$ | Latenzzeit des Netzwerkes |
| $t_{\mathrm{n}}$ | Zykluszeit des Netzwerkes |
| $C_N$ | Kapazität des Netzwerkes pro Knoten |

Die Applikationsparameter erklären die reine Rechenzeit des Prozessors. Dies sind die Dauer für das Absetzen einer Einzel- oder Vektorvorladeanweisung $t_{\mathrm{v}}$ bzw. $t_{\mathrm{v}}^L$ sowie die Dauer für die entsprechenden Zugriffsoperationen $t_{\mathrm{z}}$ bzw. $t_{\mathrm{z}}^L$. Diese Zeiten beinhalten den Aufwand für die Adreßberechnung, das Absetzen der Operation, den Schleifenaufwand für die Vorlade- bzw. Zugriffsschleife und im Falle von Vektoranweisungen noch die Übermittlung der Berechnungsvorschrift für die weiteren $L-1$ Adressen. Diese vier Zeiten müssen für jedes Kommunikationsfließband gemessen werden und können innerhalb einer Applikation mit verschiedenen Kommunikationsmustern variiern. Wenn der Aufwand der Adreßberechnung und der Schleifenkontrolle abgezogen wird, so dauert das reine Absetzen einer Vorlade- oder Zugriffsanweisung gleichlang. Es ist also $t_{\mathrm{v}} = t_{\mathrm{z}}$ und $t_{\mathrm{v}}^L = t_{\mathrm{z}}^L$ für identische Adreßberechnungen.

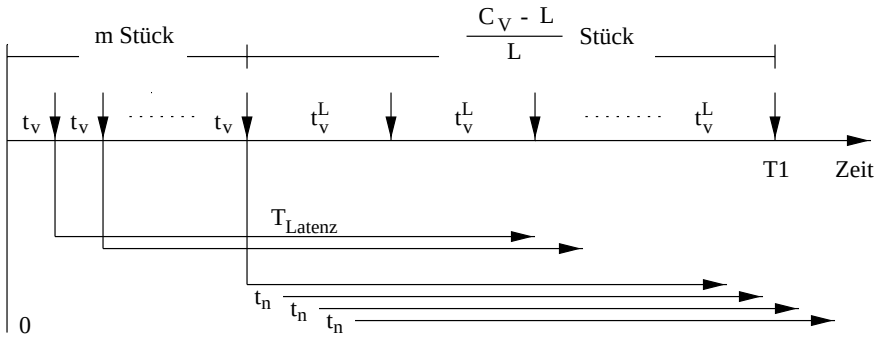
Die Anzahl der entfernten Zugriffe innerhalb eines Fließbandes wird mit $K$ gekennzeichnet. Da im allgemeinen

$$0 \not\equiv K \bmod L \tag{3.5}$$

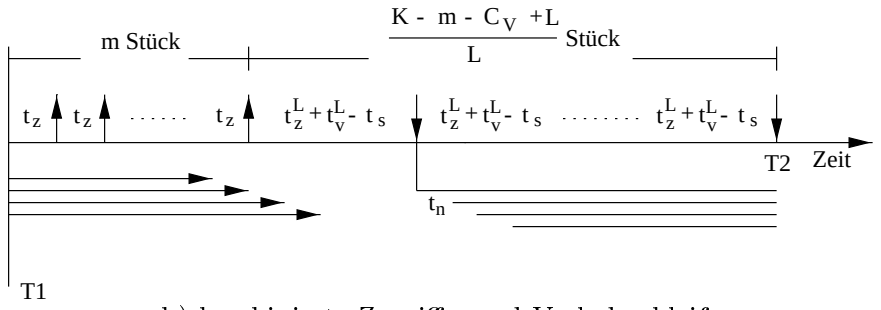
gilt, müssen die restlichen $m = K \bmod L$ nicht-lokalen Datenelemente, die keinen Vektor mehr füllen ($0 \leq m \leq L-1$), mit einzelnen Vorlade- und Zugriffsbefehlen kopiert werden.
Die Architekturparameter von Tabelle 3.1 beschreiben die anwendungsinvarianten Konstanten einer Parallelrechnerarchitektur. Zu diesen Konstanten gehören die Vektorlänge $L$, die Größe des Vorladepuffers $C_V$ und die Dauer für die Schleifenkontrollen $t_\mathrm{s}$. Die Latenzzeit des Netzwerkes ist mit $T_\mathrm{Latenz}$ beschrieben. Das Netzwerk kann alle $t_\mathrm{n}$ Zeitschritte einen Kommunikationsauftrag vom Vorladepuffer entgegennehmen. Damit dauert das Abholen eines Vektors der Länge $L$ durch das Netzwerk $L * t_\mathrm{n}$. Abschließend beschreibt $C_N$ die Kapazität des Netzwerkes, d. h. die Anzahl der Kommunikationsoperationen, die das Netzwerk pro Knoten gleichzeitig bearbeiten kann.
Mit einem überlappenden kann ein blockierendes Netzwerk simuliert werden, wenn die Größe des Vorladepuffers nicht ausgenützt und nur ein Eintrag verwendet wird. Damit wird die blockierende zum Spezialfall der überlappenden Kommunikation.

### 3.1.4 Arbeitsweise des VSCAP-Verfahrens

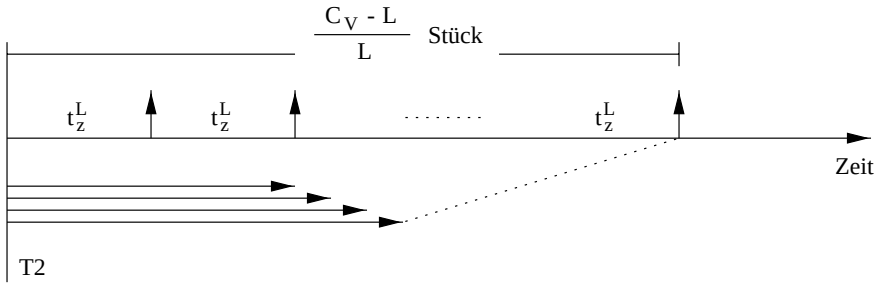
Mit den Parametern aus dem vorherigen Abschnitt kann die prinzipielle Arbeitsweise des VSCAP-Verfahrens modelliert werden, siehe Abbildung 3.1.
In der *Vorladephase* (oder auch *Vorladeschleife*) werden vom Prozessor $m$ Einzel- und $\frac{C_V - L}{L}$ Vektorvorladeoperationen abgesetzt, Abbildung 3.1.a. Die einelementigen werden vor den Vektoroperationen ausgeführt, damit bei wenigen nicht-lokalen Datenzugriffen die Berechnung der Schleifengrenze der Vektorvorladeschleife verdeckt werden kann, da sie im Vergleich zur einelementigen Vorladeschleife etwas aufwendiger ist.
Das Netzwerk nimmt alle $t_\mathrm{n}$ Schritte eine Anforderung entgegen und verarbeitet sie. Dabei bestimmen $max(t_\mathrm{v}, t_\mathrm{n})$ die Zykluszeit für die Abarbeitung der Einzelelemente und $t_\mathrm{n}$ die Zeit für die Bearbeitung einzelner Vektorelemente. Die Vorladephase ist zum Zeitpunkt $T1 = m * t_\mathrm{v} + \frac{C_V - L}{L} * t_\mathrm{v}^L$ abgeschlossen, wenn der Prozessor die $m$ Einzel- und die $\frac{C_V - L}{L}$ Vektorvorladebefehle der Dauer $t_\mathrm{v}$ bzw. $t_\mathrm{v}^L$ abgesetzt hat. Die Arbeitszeit des Netzwerkes spielt für die Ausführung der Vorladephase des Prozessors keine Rolle, da hier keine Zugriffsoperationen stattfinden.
Danach beginnt die Phase in denen sich die Zugriffs- und Vorladeoperationen abwechseln (*kombinierte Zugriffs- und Vorladeschleife*), siehe Abbildung 3.1.b. Sie wird benötigt, wenn die Anzahl $K$ der entfernten Elemente die Kapazität $C_V$ des Vorladepuffers übersteigt. Am Anfang dieser Phase wird mit Einzelzugriffen auf die ersten $m$ vorgeladenen Elemente zugegriffen. Danach wechseln sich $\frac{K - m - C_V + L}{L}$ Vektorzugriffs- und Vektorvorladeoperationen ab, bis alle $K$ Elemente vorgeladen wurden. Der Aufwand jeder der $\frac{K - m - C_V + L}{L}$ Iterationen der kombinierten Zugriffs- und Vorladeschleife beträgt $t_\mathrm{z}^L + t_\mathrm{v}^L - t_\mathrm{s}$, denn durch die Addition von $t_\mathrm{z}^L$ und $t_\mathrm{v}^L$ wird eine Schleifenkontrolle $t_\mathrm{s}$ zuviel berechnet, die nachträglich abgezogen werden muß. Die kombinierte Vorlade- und Zugriffsschleife ist ohne Wartezeiten des Prozessors zum Zeitpunkt $T2 = T1 + m * t_\mathrm{z} + \frac{K - m - C_V + L}{L} * (t_\mathrm{z}^L + t_\mathrm{v}^L - t_\mathrm{s})$ abgeschlossen.
Sollten beim Zugriff des Prozessors auf ein Element oder einen Vektor diese noch nicht im Vorladepuffer liegen, dann wird er solange angehalten, bis das Netzwerk alle ausstehenden




a) Vorladeschleife




b) kombinierte Zugriffs- und Vorladeschleife




c) Zugriffsschleife

Abbildung 3.1: Arbeitsweise des VSCAP-Verfahrens

Daten liefern konnte. Das bedeuted, daß im Falle eines Vektorzugriffs alle Vektorelemente im Vorladepuffer liegen müssen, damit der Prozessor verzögerungsfrei darauf zugreifen kann. Die möglichen Wartezeiten werden jedoch erst in 3.2.2 bei der Berechnung der Systemlaufzeiten betrachtet und sind hier noch nicht relevant.
Auf die zweite Phase folgt die reine Zugriffsphase (oder *Zugriffsschleife*), welche die letzten $C_V - L$ Elemente mit $\frac{C_V - L}{L}$ Vektoroperationen der Dauer $t_z^L$ in Empfang nimmt, siehe Abbildung 3.1.c. Das VSCAP-Verfahren ist mit dem Ende der Zugriffsphase zum Zeitpunkt $T2 + \frac{C_V - L}{L} * t_z^L$ abgeschlossen.
Die reine Prozessorlaufzeit des VSCAP-Verfahrens $T_{\mathrm{VSCAP}}$ beträgt somit

$$\begin{aligned} T_{\mathrm{VSCAP}} &= T2 + \frac{C_V - L}{L} * t_z^L \\ &= m * (t_v + t_z) + \frac{K - m}{L} * (t_v^L + t_z^L) - \frac{K - m - C_V + L}{L} * t_s. \end{aligned} \tag{3.6}$$

In einem ausführbaren Programm wird das VSCAP-Verfahren durch fünf separate Schleifen implementiert, die nachfolgend diskutiert werden. Es wird das Prinzip vorgestellt, mögliche Optimierungen sind nicht aufgeführt.
Als Ausgangspunkt dient eine Kopieroperation in einem blockierenden Netzwerk:

```
                    /* Kopieren im blockierenden Netzwerk */
FOR i = 0 TO K-1 DO
  addr := calculate_address(B[Start_B+i*Inc_B]);
  A[Start_A+i*Inc_A] := remote_access(addr);
END FOR
```

Obiges Programmstück kopiert $K$ Elemente von Feld $B$ nach Feld $A$. Die Indexmengen von $A$ und $B$ sind durch die Startadressen $Start_A$ bzw. $Start_B$ und den Abständen zwischen den einzelnen Elementen $\mathrm{Inc_A}$ bzw. $\mathrm{Inc_B}$ beschrieben. Nach der Ausführung des Programmstückes gilt $A[Start_A + i * \mathrm{Inc_A}] = B[Start_B + i * \mathrm{Inc_B}]$, für $0 \leq i < K$. Für VSCAP werden zwei separate Zeiger auf die zu kopierenden Adressen benötigt. Dies sind der aktuelle Vorladeindex $\mathrm{I_{Prefetch}}$ und der Index für die Zugriffsoperationen $\mathrm{I_{Access}}$. $\mathrm{I_{Prefetch}}$ indiziert die nicht-lokalen Quell- und $\mathrm{I_{Access}}$ die lokalen Zieladressen. $\mathrm{I_{Prefetch}}$ läuft $\mathrm{I_{Access}}$ um die Anzahl $C_V - L$ der Vorladepuffereinträge voraus.
In der ersten Schleife werden die $m$ nicht-lokalen Datenelemente vorgeladen, mit denen kein ganzer Vektor der Länge $L$ gefüllt werden kann.

```
                    /* Schleife 1 */
                    /* Einzelelemente vorladen */
m := MOD(K,L);
I_Prefetch := Start_B;
FOR i = 0 TO m-1 DO
  addr := calculate_address(B[I_Prefetch]);
  prefetch(addr);
  I_Prefetch := I_Prefetch + Inc_B;
END FOR
```

In der zweiten Schleife, der Vorladeschleife, wird der Vorladepuffer vollständig mit Vektorvorladebefehlen gefüllt. In ihr muß darauf geachtet werden, daß bei $K-m > C_V-L$ nicht-lokalen Datenzugriffen, ein Überlaufen des Vorladepuffers vermieden wird. Deshalb ist die Anzahl der Iterationen durch $\frac{min(K-m,C_V-L)}{L}$ gegeben, mit $K-m$ nicht-lokalen Zugriffen und $C_V-L$ als zulässige Größe des Vorladepuffers.

```
                    /* Schleife 2 */
                    /* Vektorvorladen */
FOR i = 0 TO MIN(K-m, C_V-L)/L-1 DO
  addr := calculate_address(B[I_Prefetch]);
  vector_prefetch(addr,Inc_B);
  I_Prefetch := I_Prefetch + Inc_B*L;
END FOR
```

Danach werden die $m$ vorgeladenen Datenelemente in den lokalen Speicher geschrieben. Der

Ausdruck $\mathrm{Start_B} + (I_{\mathrm{Access}} - \mathrm{Start_A})/\mathrm{Inc_A} * \mathrm{Inc_B}$ berechnet vom Zugriffszähler $\mathrm{I_{Access}}$ ausgehend, den für den Vorladebefehl benützten Index $\mathrm{I_{Prefetch}}$, damit mit `access` der richtige Eintrag im Vorladepuffer angesprochen werden kann.

```
                    /* Schleife 3 */
                    /* Einzelelemente zugreifen */
I_Access := Start_A;
FOR i = 0 TO m-1 DO
  addr := calculate_address(B[Start_B+(I_Access-Start_A)/Inc_A*Inc_B]);
  A[I_Access] := access(addr);
  I_Access := I_Access + Inc_A;
END FOR
```

Die anschließende kombinierte Zugriffs- und Vorladeschleife wechselt schließlich Vektorzugriffe und Vektorvorladeoperationen ab. Sie wird $\frac{K-m-C_V-L}{L}$-mal ausgeführt. Der Schleifenkopf überwacht auch die Fälle, in denen die kombinierte Schleife nicht ausgeführt werden muß. Dies kommt vor, wenn die $K-m$ nicht-lokalen Datenzugriffe vollkommen in den Vorladepuffer passen, d.h. wenn $K-m \leq C_V - L$ ist. Dann wird $\frac{K-m-C_V+L}{L} - 1$ negativ und die Schleife wird nicht betreten.

```
                    /* Schleife 4 */
                    /* kombinierte Zugriffs- und Vorladeschleife */
FOR i = 0 TO (K-m-C_V+L)/L-1 DO
  addr := calculate_address(B[Start_B+(I_Access-Start_A)/Inc_A*Inc_B]);
  vector_access(A[I_Access],Inc_A,addr);
  I_Access := I_Access + Inc_A*L;

  addr := calculate_address(B[I_Prefetch]);
  vector_prefetch(addr,Inc_B);
  I_Prefetch := I_Prefetch + Inc_B*L;
END FOR
```

In der letzten Schleife werden die verbliebenen Einträge des Vorladepuffers in den Hauptspeicher geschrieben. Dabei muß wieder darauf geachtet werden, ob die $K-m$ nicht-lokalen Datenzugriffe den Vorladepuffer vollständig füllen konnten, denn dann müssen nur diese $K-m$ Elemente zurückgeschrieben werden. Ansonsten muß der gesamte Vorladepuffer geleert werden. Damit iteriert die Zugriffsschleife $min(K-m, C_V - L)/L$-mal.

```
                    /* Schleife 5 */
                    /* Zugriffsschleife */
FOR i = 0 TO MIN(K-m,C_V-L)/L-1 DO
  addr := calculate_address(B[Start_B+(I_Access-Start_A)/Inc_A*Inc_B]);
  vector_access(A[I_Access],Inc_A,addr);
  I_Access := I_Access + Inc_A*L;
END FOR
```

### 3.1.5 Kommunikationsmuster, Vorlade- und Vektorstrategien

Zum besseren Verständnis der Anwendung und der Implementierung des VSCAP-Modells werden hier die Begriffe des Kommunikationsmusters, der Vorlade- und der Vektorstrategie definiert und ihre Unterschiede aufgezeigt.

**Definition 1 (Kommunikationsmuster)**
Ein *Kommunikationsmuster* $\mathcal{K}$ ist eine Abbildung der Menge (oder Teilmenge) der virtuellen Prozessoren $\mathcal{V}$ auf sich selber (bzw. auf eine eventuell andere Teilmenge). Dabei bedeutet $\mathcal{K}(i) = j$, daß Prozessor $i$ ein Datum von Prozessor $j$ benötigt.

*Statische Kommunikationsmuster* können bereits zur Übersetzungszeit erkannt werden, z.B. alle affinen Kommunikationsmuster $\mathcal{K}(i) = a * i + b$ mit Laufzeitkonstanten $a, b \in \mathbb{N}$. *Dynamische Kommunikationsmuster* sind durch Größen gekennzeichnet, deren Wert erst zur Laufzeit feststeht, z.B. $\mathcal{K}(i) = g(i)$ und $g$ ist eine Funktion oder ein Feld. Im Unterschied zu *planbaren Algorithmen* (siehe 2.3.3) wird hier die Anzahl $V$ der virtuellen Prozessoren zur Bestimmung des Kommunikationsmusters nicht weiter berücksichtigt.
Die nicht-lokalen Datenzugriffe eines Kommunikationsmusters können durch verschiedene Vorladestrategien ausgeführt werden.

**Definition 2 ($(C_V, L)$-Vorladestrategie)**
Eine $(C_V, L)$-Vorladestrategie ist durch die Größe des Vorladepuffers $C_V$ und durch die Vektorlänge $L$ der eingesetzten Vektoren bestimmt.

Blockierende Kommunikation ist demnach eine $(1, 1)$-, das SCAP-Verfahren eine $(C_V, 1)$, mit $C_V > 1$, und VSCAP eine $(C_V, L)$-Vorladestrategie mit $C_V > 1$ und $L > 1$. Während $(1, 1)$- und $(C_V, 1)$-Vorladestrategien nicht weiter verfeinert werden können, ergeben sich bei einem $(C_V, L)$-Vorladen in Abhängigkeit vom Kommunikationsmuster mehrere Möglichkeiten.

**Definition 3 ($(a, z)$-Vektorstrategie)**
Die $(a, z)$-*Vektorstrategie* bestimmt den Einsatz der Vektorbefehle für Vorlade- ($a \geq 1$) und Zugriffsoperationen ($z \geq 1$). Aus der Kombination von Einzel- und Vektoroperationen für die Vorlade- und Zugriffsbefehle ergeben sich vier Vektorstrategien

| Vektorstrategie | Beschreibung |
|---|---|
| ($L$,$L$) | Vektorbefehle für Vorlade- und Zugriffsoperationen |
| (1,$L$) | Einzelelement Vorladebefehle und Vektorzugriffe |
| ($L$,1) | Vektorvorladebefehle und einelementige Zugriffe |
| (1,1) | Einelementige Vorlade- und Zugriffsoperationen |

Die vier Vektorstrategien sind wie folgt charakterisiert:

**($L$,$L$)-Vektorstrategie:** Wenn das Kommunikationsmuster zur Übersetzungszeit berechnet werden kann und sich die Nummern der Quell- und Zielprozessoren konstant vergrößern, z.B. bei $\mathcal{K}(i) = a * i + b$, so kann eine $(L, L)$-Vektorstrategie eingesetzt werden. Bei der ($L$,$L$)-Vektorstrategie werden wie in 3.1.3 beschrieben, nur $m$ Datenelemente mit Einzeloperationen entfernt gelesen. Die restlichen $K - m$ Elemente werden durch Vektoroperationen kopiert. Das Kommunikationsmuster ist in Abbildung 3.2 veranschaulicht.

Eine mögliche Implementierung der ($L$,$L$)-Vektorstrategie wurde im vorangegangenen Abschnitt gezeigt. Die Laufzeit einer ($L$,$L$)-Vektorstrategie mit $K > C_V - L$ nicht-lokalen Datenzugriffen gibt (3.6) an.

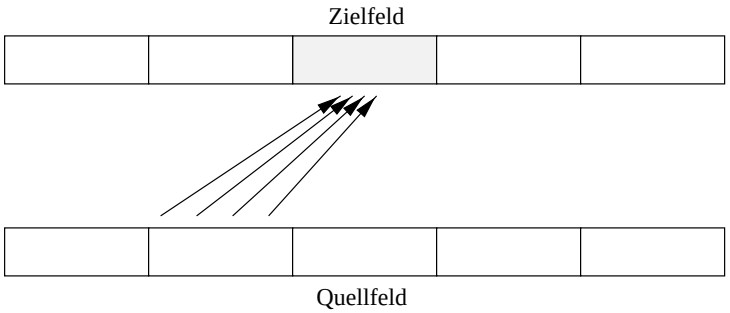



Abbildung 3.2: Kommunikationsmuster bei ($L$,$L$)-Vektorstrategie

**(1,$L$)-Vektorstrategie:** Diese Vektorstrategie kommt bei dynamischen Kommunikationsmustern zum Einsatz, wenn der Indexausdruck der rechten Seite einer Zuweisung nicht zur Übersetzungszeit berechnet werden kann, wie z.B. bei `A[i] := B[q(i)]` ($\mathcal{K}(i) = q(i)$). In diesem Fall werden die nicht-lokalen Daten einzeln vorgeladen und durch Vektorbefehle in den lokalen Speicher geschrieben. Das Kommunikationsmuster ist in Abbildung 3.3 dargestellt.

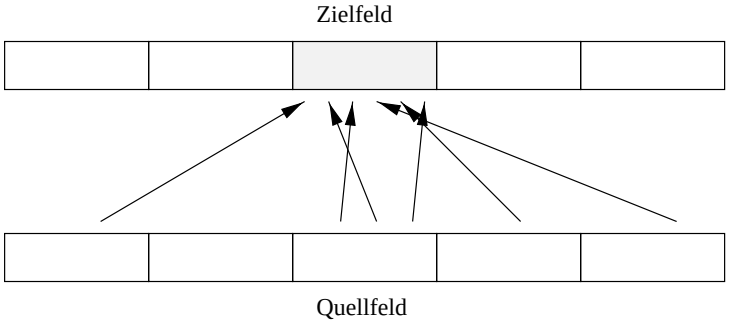



Abbildung 3.3: Kommunikationsmuster bei (1,$L$)-Vektorstrategie

Das folgende Programmstück zeigt eine mögliche Implementierung der (1,$L$)-Vektorstrategie. Es wurde vereinfachend $m = 0$ angenommen. Das Programmstück zeigt die Permutation `A[i]=B[q(i)]`. Der Iterationsraum ist durch die Startadresse $\text{Start}_\text{A}$ und die Schrittweite $\text{Inc}_\text{A}$ gegeben. Die Wahl der Schleifengrenzen orientiert sich an den Schleifen 2, 4 und 5 von 3.1.4, mit $m = 0$.

In der ersten Schleife wird durch einelementige Vorladeoperationen der Vorladepuffer gefüllt. Für jede der $MIN(K, C_V - L)$ Iterationen benötigt der Prozessor $t_\text{v}$ Schritte.

```
                    /* Schleife 1 */
I_Prefetch := Start_A;
FOR i = 0 TO MIN(K,C_V-L)-1 DO
  addr := calculate_address(B[q(I_Prefetch)]);
  prefetch(addr);
  I_Prefetch := I_Prefetch + Inc_A;
END FOR
```

Die mittlere Schleife greift auf die einzelnen Elemente mit Vektorbefehlen zu. Eine innere Schleife füllt die freigewordenen $L$ Einträge des Vorladepuffers wieder durch einelementige Operationen. Da immer $L$ Elemente vorgeladen werden müssen, reicht die Endebedingung im Schleifenkopf der äußeren Schleife. Jede Iteration dauert $t_{\mathrm{z}}^{L} + L * t_{\mathrm{v}}$.

```
                    /* Schleife 2 */
I_Access := Start_A;
FOR i = 0 TO (K-C_V+L)/L-1 DO
  addr := calculate_address(B[q(I_Access)]);
  vector_access(A[I_Access],Inc_A,addr);
  I_Access := I_Access + L*Inc_A;
  FOR j = 0 TO L-1 DO
    addr := calculate_address(B[q(I_Prefetch)]);
    prefetch(addr);
    I_Prefetch := I_Prefetch + Inc_A;
  END FOR
END FOR
```

In der letzten Schleife werden schließlich die verbliebenen Elemente aus dem Vorladepuffer in den lokalen Speicher geschrieben. Jede Iteration dauert $t_{\mathrm{z}}^{L}$.

```
                    /* Schleife 3 */
FOR i = 0 TO MIN(K,C_V-L)/L-1 DO
  addr := calculate_address(B[q(I_Access)]);
  vector_access(A[I_Access],Inc_A,addr);
  I_Access := I_Access + L*Inc_A;
END FOR
```

Damit beträgt die Laufzeit des VSCAP-Verfahrens für eine (1,$L$)-Vektorstrategie bei $K > C_V - L$ nicht-lokalen Zugriffen ohne weitere Wartezeiten

$$
\begin{aligned}
T_{\mathrm{VSCAP}} &= T_{\mathrm{Schleife\ 1}} + T_{\mathrm{Schleife\ 2}} + T_{\mathrm{Schleife\ 3}} \\
&= (C_V - L) * t_{\mathrm{v}} + \frac{K - C_V + L}{L} * (t_{\mathrm{z}}^{L} + L * t_{\mathrm{v}}) + \frac{C_V - L}{L} * t_{\mathrm{z}}^{L} \\
&= K * t_{\mathrm{v}} + \frac{K}{L} * t_{\mathrm{z}}^{L} \qquad (3.7)
\end{aligned}
$$

**($L$,1)-Vektorstrategie:** Bei dieser Strategie werden Vektorvorlade- und einelementige Zugriffsbefehle benützt. Da sie vor allem bei verteilenden Kommunikationsmustern der Form $\mathcal{K}(g(i)) = i$ vorkommt und überlappende Netzwerke beim Versenden von Daten in nachrichtengekoppelten Parallelrechnerarchitekturen schon bekannt sind, wird diese Vektorstrategie hier nicht weiter untersucht.

**(1,1)-Vektorstrategie:** Diese Strategie entspricht dem SCAP-Verfahren. Sie wird bei Kommunikationsmustern eingesetzt, die den Einsatz von Vektorbefehlen überhaupt nicht zulassen. Das sind zum einen überwachte Zuweisungen, bei denen ein Prädikat die Zuweisung dominiert (siehe auch Abbildung 3.4). Dies hat zur Folge, daß das Prädikat $P(i)$ für jeden Index $i$ separat auf Gültigkeit untersucht werden muß.

```
FORALL i = 0 TO N-1 DO
  IF P(i) THEN
    A[i] := B[q(i)];
  END IF
END FORALL
```

Weitere Einsatzgebiete der (1,1)-Vektorstrategie sind auch einige Kombinationen von Datenverteilung, Iterationsraum und Indexausdruck, bei denen statisch nicht beschreibbare Indizierungs- und Kommunikationsmuster entstehen, die dann mit Hilfe von Speichertabellen berechnet werden müssen (vgl. 4.1.4.3). Die entfernten Datenzugriffe und die lokalen Schreibvorgänge können dann nur mit einelementigen Operationen vollzogen werden, weshalb auch hier der Einsatz von Vektorbefehlen nicht möglich ist.

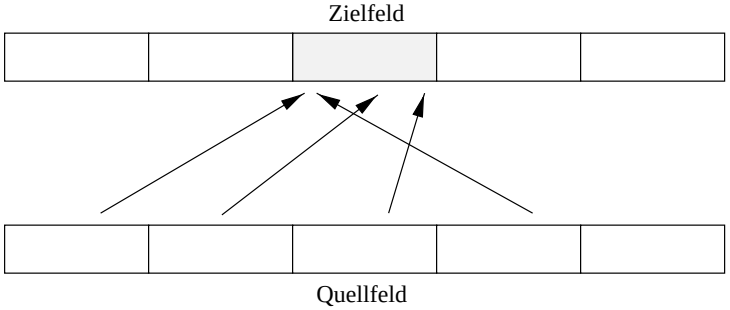



Abbildung 3.4: Kommunikationsmuster bei (1,1)-Vektorstrategie

Die Vorstellung des SCAP-Verfahrens zeigte in 2.2.2 eine mögliche Implementierung der (1,1)-Vektorstrategie. Die Laufzeit berechnet sich aus der Laufzeit (3.6) der ($L$,$L$)-Vektorstrategie, wenn man $m = 0$, $L = 1$, $t_{\mathrm{v}}^{L} = t_{\mathrm{v}}$ und $t_{\mathrm{z}}^{L} = t_{\mathrm{z}}$ setzt.

Die Beschreibung der Systemlaufzeiten von Abschnitt 3.2.2 wird die ($L$,$L$)- und (1,$L$)-Vektorstrategien genauer untersuchen. Die (1,1)-Vektorstrategie ist mit der $(C_V, 1)$-Vorladestrategie des SCAP-Verfahrens identisch, so daß sie hier nicht weiter betrachtet wird, denn der Teil dieser Strategie mit $t_{\mathrm{v}} \leq t_{\mathrm{z}}$ wurde schon durch Warschko beschrieben und die Fälle mit $t_{\mathrm{v}} > t_{\mathrm{z}}$ sind in den Betrachtungen von 3.2.2.2 enthalten.

## 3.2 Effizienzbetrachtungen

In diesem Abschnitt sollen Fragen über die Leistungsfähigkeit des VSCAP-Verfahrens beantwortet werden. Dies beinhaltet zum einen den Grad der Latenzzeitverbergung bei der Ausführung des VSCAP-Verfahrens gegenüber einem blockierenden Netzwerk. Der zweite damit eng verbundene Themenkomplex beschäftigt sich mit den Laufzeitvorteilen gegenüber SCAP, die sich aus dem Einsatz der Vektorbefehle ergeben. Diese Fragestellung wird mit dem Vergleich der VSCAP- zu den SCAP-Laufzeiten nachgegangen. Dabei ist es interessant festzustellen, ob sich die intuitive Reduktion der Kommunikationszeit um den Faktor $L$ bestätigt, wenn statt einelementigen die Vektorbefehle des VSCAP-Verfahrens eingesetzt werden. Der dritte Schwerpunkt beschäftigt sich mit der optimalen Vektorlänge. Denn es wird sich herausstellen, daß lange Vektoren zwar den Prozessoraufwand reduzieren aber gleichzeitig auch den Druck auf die Netzwerkschnittstelle erhöhen, welche die Unmengen von Kommunikationsanforderungen nicht rechtzeitig beantworten kann, so daß es zu längeren Wartezeiten seitens des Prozessors kommt.
Der erste Teil dieses Abschnitts bespricht die Laufzeiten blockierender Netzwerke. In 3.2.2 geht es um die Laufzeiten des VSCAP-Verfahrens. Die Betrachtungen sind in die ($L$,$L$)- und (1,$L$)-Vektorstrategie aufgeteilt, da der verschiedene Einsatz der Vektorbefehle innerhalb dieser Strategien zu unterschiedlichem Laufzeitverhalten führt. Im darauffolgenden Abschnit 3.2.3 werden die Laufzeiten der verschiedenen Vorladestrategien mit den oben erwähnten Fragestellungen untersucht. Am Ende der Effizienzbetrachtungen steht in 3.2.4 die Frage nach der optimalen Vektorlänge.

### 3.2.1 Kommunikationszeiten blockierender Netzwerke

Nach der Definition 2 ist ein blockierendes Netzwerk als (1,1)-Vorladestrategie ein Spezialfall eines überlappenden Netzwerkes. Berechnet man die Laufzeiten einer blockierenden Kommunikation (BLOCK), so müssen die Adreßberechnungszeiten des Prozessors und seine Wartezeiten auf die Bereitstellung eines nicht-lokalen Datums vom Netzwerk addiert werden. Die Kommunikation läuft dabei unabhängig vom Kommunikationsmuster immer nach dem gleichen Schema ab, da nur Einzelelementbefehle benützt werden. Für jeden nicht-lokalen Datenzugriff berechnet der Prozessor die entfernte Quelladresse und die lokale Zieladresse, was pro Iteration einen Aufwand von $(t_\mathrm{v} + t_\mathrm{z} - t_\mathrm{s})$ bedeutet. Der Schleifenaufwand $t_\mathrm{s}$ muß nachträglich abgezogen werden, da er durch $t_\mathrm{v}$ und $t_\mathrm{z}$ doppelt berechnet wurde. Die Rechenzeit beträgt damit für $K$ nicht-lokale Zugriffe $K * (t_\mathrm{v} + t_\mathrm{z} - t_\mathrm{s})$. Die Wartezeit für jeden nicht-lokalen Datenzugriff ist in dieser Arbeit und im Vergleich zu Warschko nicht mit der Netzwerklatenz $T_\mathrm{Latenz}$ identisch, denn das blockierende Netzwerk wird durch ein überlappendes Netzwerk simuliert. Das bedeutet, daß die Adreßberechnung für die lokale Zieladresse mit der Wartezeit auf das entfernte Datum überlappt wird. Dies führt zu einer Reduktion der Wartezeit um die Adreßberechnung. Sie dauert letztendlich $T_\mathrm{Latenz} - (t_\mathrm{z} - t_\mathrm{s})$ für jeden der $K$ nicht-lokalen Zugriffe.
Damit ist die Laufzeit $T_\mathrm{BLOCK}$ eines blockierenden Netzwerkes

$$\begin{aligned} T_\mathrm{BLOCK} &= K * (t_\mathrm{v} + t_\mathrm{z} - t_\mathrm{s}) + K * (T_\mathrm{Latenz} - (t_\mathrm{z} - t_\mathrm{s})) \\ &= K * (t_\mathrm{v} + T_\mathrm{Latenz}) \end{aligned} \tag{3.8}$$

Der folgende Abschnitt zeigt die Berechnung der Kommunikationszeiten des VSCAP-Verfahrens für die verschiedenen Vektorstrategien.

### 3.2.2 Berechnung der Systemlaufzeiten für VSCAP

Die Kommunikationszeiten der einzelnen Vektorstrategien von VSCAP setzen sich aus dem Laufzeitaufwand $T_{\mathrm{L,VSCAP}}$ (Vorlade- und Zugriffsaufwand) sowie anfallender Wartezeiten $T_{\mathrm{W,VSCAP}}$ zusammen:

$$T_{\mathrm{VSCAP}} = T_{\mathrm{L,VSCAP}} + T_{\mathrm{W,VSCAP}} \tag{3.9}$$

$T_{\mathrm{L,VSCAP}}$ und $T_{\mathrm{W,VSCAP}}$ hängen jeweils von der gewählten Vektorstrategie ab, die sich durch den unterschiedlichen Einsatz der Vektorbefehle für Vorlade- und Zugriffsoperationen auszeichnen. Dies ermöglicht keine allgemeingültige Berechnung, sondern erfordert eine Instanziierung von $T_{\mathrm{L,VSCAP}}$ und $T_{\mathrm{W,VSCAP}}$ für jede Vektorstrategie.
Da sich die betrachteten Vektorlängen im Bereich von $L = 2^n$, mit $n \leq 5$, bewegen (siehe 3.1.1), werden für die folgenden Berechnungen nur Problemgrößen $K$ betrachtet, die ausschließlich mit Vektorbefehlen behandelt werden können, d.h. $K \equiv 0 \mod (L)$, denn der Anteil der Kommunikationszeit für die höchstens $L-1$ Elemente wird sich bei $K > C_V$ nicht-lokalen Zugriffen unterhalb der Meßtoleranz befinden. Für die Betrachtungen gilt somit $m = 0$.
Dies ändert die Arbeitsweise von VSCAP, das nun nicht wie in 3.1.4 vorgestellt fünf Schleifen für die Kommunikation benötigt. Die Schleifen 1 und 3 entfallen, da sie lediglich für das Vorladen und den Zugriff auf die höchstens $L-1$ restlichen Datenelemente gebraucht würden. Das hier beschriebene VSCAP setzt sich aus drei Schleifen zusammen: In der ersten Schleife (Schleife 2 von 3.1.4) wird der Vorladepuffer mit Vektorbefehlen gefüllt. In der zweiten Schleife, der kombinierten Zugriffs- und Vorladeschleife (Schleife 4 von 3.1.4) werden abwechseln Zugriffs- und Vorladeoperationen abgesetzt. Die letzte Schleife (Schleife 5 von 3.1.4) greift auf die verbliebenen Daten des Vorladepuffers zu.
Die Wartezeiten für VSCAP $T_{\mathrm{W,VSCAP}}$ setzen sich aus der Summe der Wartezeiten auf die $\frac{K}{L}$ Vektoren $T^L_{\mathrm{Warten}}(j)$ zusammen:

$$T_{\mathrm{W,VSCAP}} = \sum_{j=1}^{\frac{K}{L}} T^L_{\mathrm{Warten}}(j) \tag{3.10}$$

Die folgenden Abschnitte betrachten nacheinander die $(L,L)$- und $(1,L)$-Vektorstrategien. Sie berechnen den Laufzeitaufwand $T_{\mathrm{L,VSCAP}}$ und die anfallenden Wartezeiten $T_{\mathrm{W,VSCAP}}$ und leiten daraus nach (3.9) die Laufzeit von VSCAP ab.

#### 3.2.2.1 $(L,L)$-Vektorstrategie

Die $(L,L)$-Vektorstrategie benützt Vektorbefehle für Vorlade- und Zugriffsoperationen. Das entsprechende Programmstück zeigte 3.1.4. Der Aufwand für die Durchführung eines VSCAP Fließbandes bei dem die Anzahl $K$ der nicht-lokalen Datenzugriffe die Kapazität des Vorladepuffers überstieg ($K > C_V - L$) ist durch Gleichung (3.6) gegeben. Beachtet man noch die Fälle, in denen der Vorladepuffer nicht vollständig gefüllt wird ($K \leq C_V - L$) und die Ausführung der kombinierten Zugriffs- und Vorladeschleife entfällt, so ergibt sich die folgende VSCAP-Kommunikationszeit:

$$T_{\mathrm{VSCAP}} = \frac{K}{L} * (t_{\mathrm{v}}^{L} + t_{\mathrm{z}}^{L}) - max(0, \frac{K - C_V + L}{L}) * t_{\mathrm{s}} + T_{\mathrm{W,VSCAP}} \tag{3.11}$$

Die Gleichung berechnet den Laufzeitaufwand für eine zwei- und eine dreischleifige Ausführung des VSCAP-Verfahrens. Der Term $max(0, \frac{K-C_V+L}{L}) * t_{\mathrm{s}}$ zieht die bei einer dreischleifigen Ausführung zuviel berechneten Schleifenkontrollen nachträglich ab.
Die $(L,L)$-Vektorstrategie ist durch einen identischen Zeitaufwand für die Vorlade- und Zugriffsanweisungen charakterisiert, da zur Adreßberechnung die gleichen Berechnungsvorschriften ausgeführt werden müssen. Damit gilt $t_{\mathrm{v}}^{L} = t_{\mathrm{z}}^{L}$ für Vektorbefehle.
Für das weitere Vorgehen sind das Verhältnis von Netzwerktakt $t_{\mathrm{n}}$ zur Vorladegeschwindigkeit $t_{\mathrm{v}}^{L}$ des Prozessors und die Anzahl der nicht-lokalen Datenzugriffe entscheidend. Denn bei $K > C_V - L$ entfernten Zugriffen wird die Kapazität des Vorladepuffers überschritten und das VSCAP-Verfahren wechselt von einer zwei- auf eine dreischleifige Abarbeitung, wobei in der kombinierten Zugriffs- und Vorladschleife Einträge des Vorladepuffers freigeräumt und anschließend wieder belegt werden.
Tabelle 3.2 stellt die verschiedenen Modellfälle für die $(L,L)$-Vektorstrategie vor.

Tabelle 3.2: Parameterraum für $(L,L)$-Vektorstrategie

| | | $L * t_{\mathrm{n}} \leq t_{\mathrm{v}}^{L}$ | $t_{\mathrm{v}}^{L} < L * t_{\mathrm{n}}$ |
|---|---|---|---|
| $K \leq C_V - L$ | $\frac{K}{L} * t_{\mathrm{v}}^{L} < T_{\mathrm{Latenz}} + (L-1) * t_{\mathrm{n}}$ | Fall 1 | Fall 4 |
| | $\frac{K}{L} * t_{\mathrm{v}}^{L} \geq T_{\mathrm{Latenz}} + (L-1) * t_{\mathrm{n}}$ | Fall 2 | Fall 5 |
| $K > C_V - L$ | | Fall 3 | Fall 6 |

Die Spalten von Tabelle 3.2 beschreiben das Verhältnis von Netzwerkzykluszeit $t_{\mathrm{n}}$ zum Prozessoraufwand für das Absetzen der Vorladeanweisungen $t_{\mathrm{v}}^{L}$. In der ersten Spalte kann das Netzwerk einen Vektor schneller bearbeiten als ihn der Prozessor vorladen kann $L * t_{\mathrm{n}} \leq t_{\mathrm{v}}^{L}$. Das Netzwerk wartet nach der Bearbeitung eines Vektors auf den nächsten Kommunikationsauftrag des Prozessors. Die zweite Spalte beinhaltet alle Fälle in denen das Netzwerk einen Vektor vergleichsweise langsamer als der Prozessor abarbeiten kann $t_{\mathrm{v}}^{L} < L * t_{\mathrm{n}}$. Die Zeilen unterscheiden das Verhältnis von entfernten Datenzugriffen $K$ und Vorladepuffergröße $C_V - L$. Die erste Zeile $K \leq C_V - L$ stellt eine zweischleifige Abarbeitung des VSCAP-Verfahrens dar, wobei in der ersten Schleife der Vorladepuffer gefüllt und in der zweiten Schleife auf die Einträge zugegriffen wird. Die erste Zeile wird weiterhin nach der Fragestellung unterteilt, ob beim Zugriff auf den ersten Vektor mit Wartezeiten zu rechnen ist $\frac{K}{L} * t_{\mathrm{v}}^{L} < T_{\mathrm{Latenz}} + (L-1) * t_{\mathrm{n}}$ oder ob der erste Zugriff ohne Wartezeit durchgeführt werden kann $\frac{K}{L} * t_{\mathrm{v}}^{L} \geq T_{\mathrm{Latenz}} + (L-1) * t_{\mathrm{n}}$.
Insgesamt können sechs verschiedene Fälle für die $(L,L)$-Vektorstrategie unterschieden werden, siehe Tabelle 3.2. Im folgenden werden die Fälle 1 bis 3 detailiert vorgestellt. Auf eine Diskussion der Fälle 4 bis 6 wird verzichtet, denn zum einen werden sie bei der späteren Validierung nicht benötigt und zum anderen liefern sie keine neuen Einblicke in die Arbeitsweise des VSCAP-Verfahrens. Sie wurden jedoch durchgerechnet, und ihre Laufzeiten sind in Abschnitt 3.2.2.3 zusammengefaßt.

**Fall 1**

($K \leq C_V - L, \frac{K}{L} * t_v^L < T_{\text{Latenz}} + (L-1) * t_n, L * t_n \leq t_v^L$, siehe Tabelle 3.2)
Dieser Fall ist durch eine geringe Anzahl an nicht-lokalen Datenzugriffe geprägt. Sie reicht zum einen nicht aus, um den Vorladepuffer zu füllen $K \leq C_V - L$, dies resultiert in einer zweischleifigen Abarbeitung, und zum anderen kann mit den $\frac{K}{L}$ Iterationen die Wartezeit $T_{\text{Latenz}}$ auf den ersten Vektor nicht verdeckt werden ($\frac{K}{L} * t_v^L < T_{\text{Latenz}} + (L-1) * t_n$).
Der Prozessor lädt alle $K$ nicht-lokalen Datenelemente vor und greift danach mit einem Vektorbefehl auf den ersten Vektor zu. Das geschieht zum Zeitpunkt

$$T_{\text{Prozessor}} = \frac{K}{L} * t_v^L + t_z^L.$$

Das Netzwerk bearbeitet den ersten Auftrag zum Zeitpunkt $t_v^L$ und hat den ersten Vektor zum Zeitpunkt

$$T_{\text{Netzwerk}} = t_v^L + T_{\text{Latenz}} + (L-1) * t_n$$

bereitgestellt. Damit beträgt die Wartezeit auf den ersten Vektor

$$\begin{aligned} T_{\text{Warten}}(1) &= T_{\text{Netzwerk}} - T_{\text{Prozessor}} \\ &= T_{\text{Latenz}} + (L-1) * t_n - \frac{K}{L} * t_v^L. \end{aligned}$$

Weitere Wartezeiten gibt es nicht ($T_{\text{Warten}}(j) = 0$, für $1 < j \leq \frac{K}{L}$), denn das Netzwerk kann den nächsten Vektor vor dem nächsten Zugriff des Prozessors bereitstellen $L * t_n \leq t_v^L = t_z^L$. Damit ist nach (3.10) $T_{\text{W,VSCAP}} = T_{\text{Warten}}(1)$, und die Laufzeit des VSCAP-Verfahrens berechnet sich mit (3.11) zu

$$\begin{aligned} T_{\text{VSCAP}} &= \frac{K}{L} * (t_v^L + t_z^L) - max(0, \frac{K - C_V + L}{L}) * t_s + T_{\text{Warten}}(1) \\ &= \frac{K}{L} * (t_v^L + t_z^L) + T_{\text{Latenz}} + (L-1) * t_n - \frac{K}{L} * t_v^L \\ &= \frac{K}{L} * t_z^L + T_{\text{Latenz}} + (L-1) * t_n \end{aligned} \tag{3.12}$$

**Fall 2**

($K \leq C_V - L, \frac{K}{L} * t_v^L \geq T_{\text{Latenz}} + (L-1) * t_n, L * t_n \leq t_v^L$, siehe Tabelle 3.2)
Das VSCAP-Verfahren hat nach wie vor eine zweischleifige Abarbeitung $K \leq C_V - L$, doch reichen nun die $\frac{K}{L}$ nicht-lokalen Datenzugriffe aus, um die Wartezeit $T_{\text{Latenz}} + (L-1) * t_n$ auf den ersten Vektor zu überbrücken. Alle weiteren Vektorzugriffe sind wie in Fall 1 wartezeitfrei, so daß $T_{\text{Warten}}^L(j) = 0$. Damit berechnet sich die Laufzeit des VSCAP-Verfahrens mit $T_{\text{W,VSCAP}} = 0$ und (3.11) zu

$$\begin{aligned} T_{\text{VSCAP}} &= \frac{K}{L} * (t_v^L + t_z^L) - max(0, \frac{K - C_V + L}{L}) * t_s + T_{\text{W,VSCAP}} \\ &= \frac{K}{L} * (t_v^L + t_z^L) \end{aligned} \tag{3.13}$$

**Fall 3**

($K > C_V - L, L * t_n \leq t_v^L$, siehe Tabelle 3.2)
In diesem Fall wechselt VSCAP von einer zwei- auf eine dreischleifige Abarbeitung. Da der Vorladepuffer komplett gefüllt werden kann ($K > C_V - L$), gilt nach (3.3) $\frac{C_V - L}{L} * t_v^L > T_{\text{Latenz}} + (L-1) * t_n$, und es gibt keine Wartezeit auf den ersten Vektor. Weitere Wartezeiten treten wegen des schnellen Netzwerktaktes $L * t_n \leq t_v^L$ auch nicht auf, so daß $T_{\text{Warten}}^L(j) = 0$ und somit $T_{\text{W,VSCAP}} = 0$ ist. Nach (3.11) berechnet sich die Laufzeit des VSCAP-Verfahrens zu

$$\begin{aligned} T_{\text{VSCAP}} &= \frac{K}{L} * (t_v^L + t_z^L) - max(0, \frac{K - C_V + L}{L}) * t_s + T_{\text{W,VSCAP}} \\ &= \frac{K}{L} * (t_v^L + t_z^L) - \frac{K - C_V + L}{L} * t_s \end{aligned} \tag{3.14}$$

#### 3.2.2.2 (1,$L$)-Vektorstrategie

Die (1,$L$)-Vektorstrategie wird bei dynamischen Kommunikatinsmustern eingesetzt bei denen die nicht-lokalen Datenzugriffe zur Übersetzungszeit nicht festgestellt werden können. Aus diesem Grund werden die entfernten Datenelemente durch einelementige Operationen vorgeladen. Der Zugriff erfolgt durch Vektorbefehle. Die Laufzeit des VSCAP-Verfahrens ist durch die $K$ Vorladebefehle der Dauer $t_v$ und die $\frac{K}{L}$ Vektorzugriffe der Dauer $t_z^L$ nach (3.7) gegeben. Wenn man eventuelle Wartezeiten $T_{\text{W,VSCAP}}$ addiert, so wird

$$T_{\text{VSCAP}} = K * t_v + \frac{K}{L} * t_z^L + T_{\text{W,VSCAP}} \tag{3.15}$$

In der obigen Gleichung beinhalten $t_v$ und $t_z^L$ die richtigen Zeiten für die Schleifenkontrollen, denn bei einer dreischleifigen Abarbeitung werden die $L$ einzelnen Vorladoperationen in einer eigenen Schleife bearbeitet (siehe dazu das Programmstück aus 3.1.5).
Die (1,$L$)-Vektorstrategie ist durch aufwendigere Adreßberechnungen für die nicht-lokalen Datenzugriffe gekennzeichnet, da nun z.B. Indirektionen in den Feldzugriffen aufzulösen sind. Deshalb dauern $L$ einzelne Vorladebefehle länger als ein Vektorzugriff $L * t_v > t_z^L$. Dies ist auch die Charakteristik der Applikationsparameter für die (1,$L$)-Vektorstrategie.
Für das weitere Vorgehen ist wie bei der ($L$,$L$)-Vektorstrategie das Verhältnis von Netzwerktakt $t_n$ zur Vorladegeschwindigkeit $t_v$ des Prozessors und die Anzahl der nicht-lokalen Datenzugriffe $K$ entscheidend. Mit der Anzahl der Datenzugriffe $K$ wird über die Ausführung einer mittleren Schleife entschieden, in der sich Zugriffs- und Vorladeoperationen ablösen ($K > C_V - L$).
Tabelle 3.3 stellt die verschiedenen Modellfälle für die (1,$L$)-Vektorstrategie vor.
Die Spalten von Tabelle 3.3 unterscheiden analog zu Tabelle 3.2 das Verhältnis von Netzwerkzykluszeit $t_n$ zur Vorladezeit des Prozessors $t_v$. In der ersten Spalte ist mit $t_n \leq t_v$ ein gegenüber dem Prozessor schneller arbeitendes Netzwerk aufgetragen, d.h. bevor der Prozessor den nächsten Vorladebefehl absetzen kann, hat das Netzwerk den letzten Kommunikationsauftrag bearbeitet und wartet auf den nächsten. In der zweiten Spalte werden entsprechend langsamere Netzwerke berücksichtigt $t_v < t_n$. Die Zeilen unterscheiden zwischen einer zweischleifigen VSCAP-Abarbeitung $K \leq C_V - L$ und einer dreischleifigen $K > C_V - L$, bei welcher der Vorladepuffer nicht alle entfernten Datenzugriffe aufnehmen kann. Die erste Zeile

Tabelle 3.3: Parameterraum für (1,$L$)-Vektorstrategie

| | | $t_n \leq t_v$ | $t_v < t_n$ |
|---|---|---|---|
| $K \leq C_V - L$ | $K * t_v < T_{Latenz} + (L-1) * max(t_v, t_n)$ | Fall 1 | Fall 4 |
| | $K * t_v \geq T_{Latenz} + (L-1) * max(t_v, t_n)$ | Fall 2 | Fall 5 |
| $K > C_V - L$ | | Fall 3 | Fall 6 |

betrachtet wieder den Fall, daß die Wartezeit auf den ersten Vektor durch die Ausführung der Vorladschleife verdeckt werden kann $K*t_v \geq T_{Latenz}+(L-1)*max(t_v, t_n)$ oder ob beim Zugriff auf den ersten Vektor mit einer Wartezeit zu rechnen ist $K*t_v < T_{Latenz}+(L-1)*max(t_v, t_n)$. Es können sechs verschiedene Fälle für die (1,$L$)-Vektorstrategie unterschieden werden. Wobei analog zur ($L$,$L$)-Vektorstrategie nur die Fälle 1 bis 3 detailiert vorgestellt werden. Die Ergebnisse der anderen Fälle sind in Abschnitt 3.2.2.3 zusammengefaßt.

**Fall 1**

($K \leq C_V - L, K * t_v < T_{Latenz} + (L-1) * t_v, t_n \leq t_v$, siehe Tabelle 3.3 )
Dieser Fall ist analog zu Fall 1 der ($L$,$L$)-Vektorstrategie durch eine zweischleifige Ausführung des VSCAP-Verfahrens gekennzeichnet $K \leq C_V - L$. Weiterhin reicht die Anzahl $K$ der entfernten Datenzugriffe nicht aus, um die Wartezeit $T_{Latenz}+(L-1)*t_v$ auf den ersten Vektor zu überbrücken. Der Prozessor lädt alle $K$ entfernten Datenelemente in den Vorladepuffer und versucht, auf den ersten Vektor zuzugreifen, das geschieht zum Zeitpunkt

$$T_{Prozessor} = K * t_v + t_z^L.$$

Das Netzwerk liefert das $L$-te Element des ersten Vektors zum Zeitpunkt

$$T_{Netzwerk} = t_v + T_{Latenz} + (L-1) * t_v.$$

Daraus entsteht eine erste Wartezeit für den Prozessor von

$$\begin{aligned} T_{Warten}(1) &= T_{Netzwerk} - T_{Prozessor} \\ &= T_{Latenz} + L * t_v - K * t_v - t_z^L \end{aligned}$$

Alle weiteren $\frac{K}{L} - 1$ Vektorzugriffe erfahren ebenfalls eine Wartezeit von $L * t_v - t_z^L$, da das Netzwerk die Vektoren nicht schneller liefern kann, als sie der Prozessor vorgeladen hat, nämlich in Abständen von $L * t_v < t_z^L$. Die einzelnen Wartezeiten sind durch $T_{Warten}^L(j)$ noch einmal zusammengefaßt:

$$T_{Warten}(j) = \begin{cases} T_{Latenz} + L * t_v - K * t_v - t_z^L, & j = 1 \\ L * t_v - t_z^L, & 1 < j < \frac{K}{L} \end{cases}$$

Um die gesamte Wartezeit des Prozessors zu berechnen, müssen die Wartezeiten $T^L_{\text{Warten}}(j)$ gemäß (3.10) aufaddiert werden:

$$\begin{aligned} T_{\text{W,VSCAP}} &= \sum_{j=1}^{\frac{K}{L}} T^L_{\text{Warten}}(j) \\ &= T_{\text{Latenz}} + L * t_{\text{v}} - K * t_{\text{v}} - t^L_{\text{z}} + (\frac{K}{L} - 1) * (L * t_{\text{v}} - t^L_{\text{z}}) \\ &= T_{\text{Latenz}} - \frac{K}{L} * t^L_{\text{z}} \end{aligned}$$

Für die Berechnung der Laufzeit des VSCAP-Verfahrens wird $T_{\text{W,VSCAP}}$ in (3.15) eingesetzt:

$$\begin{aligned} T_{\text{VSCAP}} &= K * t_{\text{v}} + \frac{K}{L} * t^L_{\text{z}} + T_{\text{W,VSCAP}} \\ &= K * t_{\text{v}} + \frac{K}{L} * t^L_{\text{z}} + T_{\text{Latenz}} - \frac{K}{L} * t^L_{\text{z}} \\ &= K * t_{\text{v}} + T_{\text{Latenz}} \end{aligned} \tag{3.16}$$

**Fall 2**

($K \leq C_V - L, K * t_{\text{v}} \geq T_{\text{Latenz}} + (L-1) * t_{\text{v}}, t_{\text{n}} < t_{\text{v}}$, siehe Tabelle 3.3)
Das VSCAP-Verfahren ist immer noch durch eine zweischleifige Abarbeitung gekennzeichnet, doch reichen jetzt die $K$ nicht-lokalen Datenzugriffe aus, um mit dem Aufwand der Vorladeschleife die Wartezeit auf den ersten Vektor zu überbrücken ($K * t_{\text{v}} \geq T_{\text{Latenz}} + (L-1) * t_{\text{v}}$). Es gilt somit $T_{\text{Warten}}(1) = 0$. Der Prozessor greift jedoch schneller auf die Daten zu, als sie das Netzwerk liefern kann. Dadurch kann es in der zweiten Schleife beim Zugriff auf einen Vektor $x > 1$ zum Zeitpunkt

$$T_{\text{Prozessor}}(x) = K * t_{\text{v}} + x * t^L_{\text{z}}$$

passieren, daß der Prozessor auf das Netzwerk warten muß, das die einzelnen Vektoren nur in größeren Abständen von $L * t_{\text{v}} > t^L_{\text{z}}$ liefern kann. Dieser Punkt ist dann erreicht, wenn die Ausführungszeit des Prozessors $T_{\text{Prozessor}}(x)$ die Lieferzeit

$$T_{\text{Netzwerk}}(x) = T_{\text{Latenz}} + x * L * t_{\text{v}}$$

des Netzwerkes für den Vektor $x$ unterschreiten würde. Gesucht ist das kleinste ganzzahlige $x \in \{2 \ldots \frac{K}{L}\}$ mit

$$\begin{aligned} T_{\text{Prozessor}}(x) &< T_{\text{Netzwerk}}(x) \\ K * t_{\text{v}} + x * t^L_{\text{z}} &< T_{\text{Latenz}} + x * L * t_{\text{v}} \\ K * t_{\text{v}} - T_{\text{Latenz}} &< x * (L * t_{\text{v}} - t^L_{\text{z}}) \\ \frac{K * t_{\text{v}} - T_{\text{Latenz}}}{(L * t_{\text{v}} - t^L_{\text{z}})} &< x \end{aligned}$$

Daraus folgt als Lösung

$$\left\lceil \frac{K * t_{\mathrm{v}} - T_{\mathrm{Latenz}}}{(L * t_{\mathrm{v}} - t_{\mathrm{z}}^{L})} \right\rceil = x \text{ mit } x \in \{2 \dots \frac{K}{L}\}$$

Ab dem Vektor $x$ sind alle weiteren Vektorzugriffe mit Wartezeiten behaftet. Die erste Wartezeit beträgt $T_{\mathrm{Warten}}(x) = T_{\mathrm{Netzwerk}}(x) - T_{\mathrm{Prozessor}}(x)$. Alle weiteren Wartezeiten betragen $T^{L}_{\mathrm{Warten}}(j) = L * t_{\mathrm{v}} - t_{\mathrm{z}}^{L}$, für $x < j \leq \frac{K}{L}$.

$$T^{L}_{\mathrm{Warten}}(j) = \begin{cases} 0, & 1 \leq j < x \\ T_{\mathrm{Latenz}} - K * t_{\mathrm{v}} + x * (L * t_{\mathrm{v}} - t_{\mathrm{z}}^{L}), & j = x \\ L * t_{\mathrm{v}} - t_{\mathrm{z}}^{L}, & x < j \leq \frac{K}{L}. \end{cases}$$

Das ergibt nach (3.10) insgesamt eine Wartezeit $T_{\mathrm{W,VSCAP}}$ von

$$\begin{aligned} T_{\mathrm{W,VSCAP}} =& \sum_{j=1}^{\frac{K}{L}} T^{L}_{\mathrm{Warten}}(j) \\ =& T_{\mathrm{Latenz}} - K * t_{\mathrm{v}} + x * (L * t_{\mathrm{v}} - t_{\mathrm{z}}^{L}) + (\frac{K}{L} - x) * (L * t_{\mathrm{v}} - t_{\mathrm{z}}^{L}) \\ =& T_{\mathrm{Latenz}} - \frac{K}{L} * t_{\mathrm{z}}^{L} \end{aligned}$$

Dies ist die gleiche Wartezeit wie für Fall 1, denn die Abstände $t_{\mathrm{v}}$, in denen die Vorladeoperationen abgesetzt werden, sind in beiden Fällen gleich und sie dominieren die Zugriffszeit $L * t_{\mathrm{v}} > t_{\mathrm{z}}^{L}$. Im Falle einer Wartezeit ist somit die Laufzeit des VSCAP-Verfahrens mit Fall 1 identisch und berechnet sich nach (3.16). Sollte $x$ jedoch nicht im erlaubten Bereich sein, d.h. $x \notin \{2 \dots \frac{K}{L}\}$, so erfährt VSCAP keine Wartezeit und es ist $T_{\mathrm{W,VSCAP}} = 0$. Die Laufzeit $T_{\mathrm{VSCAP}}$ beträgt dann nach (3.15)

$$\begin{aligned} T_{\mathrm{VSCAP}} &= K * t_{\mathrm{v}} + \frac{K}{L} * t_{\mathrm{z}}^{L} + T_{\mathrm{W,VSCAP}} \\ &= K * t_{\mathrm{v}} + \frac{K}{L} * t_{\mathrm{z}}^{L} \end{aligned} \tag{3.17}$$

**Fall 3**

($K > C_V - L, t_{\mathrm{n}} \leq t_{\mathrm{v}}$, siehe Tabelle 3.3)
Im Gegensatz zu Fall 2 übersteigt hier die Anzahl der entfernten Datenzugriffe die Kapazität des Vorladepuffers, und die Abarbeitung wechselt von einem zwei- auf ein dreischleifiges Fließband. Da der Vorladepuffer komplett gefüllt werden kann ($K > C_V - L$), gilt nach (3.4)

$$(C_V - L) * t_{\mathrm{v}} > T_{\mathrm{Latenz}} + (L - 1) * t_{\mathrm{v}} \tag{3.18}$$

und es gibt keine Wartezeit auf den ersten Vektor. Jede Iteration der kombinierten Zugriffs- und Berechnungsschleife dauert $t_{\mathrm{z}}^{L} + L * t_{\mathrm{v}}$ und jede Iteration der Zugriffsschleife dauert $t_{\mathrm{z}}^{L}$.

Da der Vorladepuffer vollständig gefüllt wird, und nach dem Vorladen jedes Vektors zuerst auf alle anderen Vektoren zugegriffen wird, verstreicht zwischen jedem Vorladen eines Vektors und dem zugehörigen Zugriff mindestens eine Zeitspanne von $\frac{C_V - L}{L} * t_z^L$. Diese ist nach (3.4) größer als die Lieferzeit des Netzwerkes $T_{\text{Latenz}} + L * t_v$, was dazu führt, daß das Netzwerk jeden Vektor *vor* dem Zugriff des Prozessors im Vorladepuffer bereitstellen kann. Damit sind alle Zugriffe des Prozessors wartezeitfrei $T_{\text{W,VSCAP}} = 0$, und die Laufzeit des VSCAP-Verfahrens berechnet sich für diesen Fall nach Gleichung (3.17).

#### 3.2.2.3 Zusammenfassung der Systemlaufzeiten

Die Tabellen 3.4 und 3.5 fassen die Laufzeiten des VSCAP-Verfahrens für die ($L$,$L$)- bzw. die (1,$L$)-Vektorstrategie zusammen.

Tabelle 3.4: Laufzeiten des VSCAP-Verfahrens bei einer ($L$,$L$)-Vektorstrategie

| **Fall** | | **Laufzeit** |
|---|---|---|
| 1 | | $\frac{K}{L} * t_z^L + T_{\text{Latenz}} + (L-1) * t_n$ |
| 2 | | $\frac{K}{L} * (t_v^L + t_z^L)$ |
| 3 | | $\frac{K}{L} * (t_v^L + t_z^L) - \frac{K - C_V + L}{L} * t_s$ |
| 4 | | $T_{\text{Latenz}} + t_v + (K-1) * t_n$ |
| 5 | $T_{\text{W,VSCAP}} > 0$ | $T_{\text{Latenz}} + t_v + (K-1) * t_n$ |
| | $T_{\text{W,VSCAP}} = 0$ | $\frac{K}{L} * (t_v^L + t_z^L)$ |
| 6 | $T_{\text{W,VSCAP}} > 0$ | $T_{\text{Latenz}} + t_v + (K-1) * t_n$ |
| | $T_{\text{W,VSCAP}} = 0$ | $\frac{K}{L} * (t_v^L + t_z^L) - \frac{K - C_V + L}{L} * t_s$ |

Die Fälle 4 bis 6 wurden bei den jeweiligen Strategien nicht besprochen, da ihre genaue Analyse keine weiteren Einsichten in die Arbeitsweise des VSCAP-Verfahrens bietet. Sie zeichnen sich durch ein gegenüber dem Prozessor langsameres Netzwerk aus ($t_n > t_v$). Dieses Verhältnis führt dazu, daß sobald der Prozessor eine Wartezeit erfährt, die Ausführungszeit durch die Arbeitszeit des Netzwerkes bestimmt wird, und somit der Einsatz der Vektorbefehle überflüssig macht. Wann sich der Einsatz in Abhängigkeit von Vorladedauer zu Netzwerktakt lohnt, bespricht 3.2.4. Die Laufzeiten für die einzelnen Strategien sind in der entsprechenden Tabelle aufgeführt.

### 3.2.3 Analyse der Systemlaufzeiten

Dieser Abschnitt beschäftigt sich mit dem Vergleich der Laufzeiten von BLOCK und SCAP mit denen von VSCAP. Der erste Vergleich von BLOCK und VSCAP zeigt die Reduktion der Latenzzeit, wenn statt eines blockierenden ein überlappendes Netzwerk mit Vektorbefehlen benützt wird. Der zweite Vergleich von SCAP und VSCAP zeigt den Gewinn der Kommunikationszeit der durch die Verwendung der Vektorbefehle entsteht. Es wird erwartet, daß der Einsatz von Vektorbefehlen im günstigsten Fall zu einer $L$-fachen Beschleunigung gegenüber

Tabelle 3.5: Laufzeiten des VSCAP-Verfahrens bei einer (1,$L$)-Vektorstrategie

| **Fall** | | **Laufzeit** |
|---|---|---|
| 1 | | $K * t_{\mathrm{v}} + T_{\mathrm{Latenz}}$ |
| 2 | | $K * t_{\mathrm{v}} + \frac{K}{L} * t_{\mathrm{z}}^{L}$ |
| 3 | | $K * t_{\mathrm{v}} + \frac{K}{L} * t_{\mathrm{z}}^{L}$ |
| 4 | | $T_{\mathrm{Latenz}} + t_{\mathrm{v}} + (K-1) * t_{\mathrm{n}}$ |
| 5 | $T_{\mathrm{W,VSCAP}} > 0$ | $T_{\mathrm{Latenz}} + t_{\mathrm{v}} + (K-1) * t_{\mathrm{n}}$ |
| | $T_{\mathrm{W,VSCAP}} = 0$ | $K * t_{\mathrm{v}} + \frac{K}{L} * t_{\mathrm{z}}^{L}$ |
| 6 | $T_{\mathrm{W,VSCAP}} > 0$ | $T_{\mathrm{Latenz}} + t_{\mathrm{v}} + (K-1) * t_{\mathrm{n}}$ |
| | $T_{\mathrm{W,VSCAP}} = 0$ | $K * t_{\mathrm{v}} + \frac{K}{L} * t_{\mathrm{z}}^{L}$ |

SCAP führt, denn der Vorlade- und Zugriffsaufwand reduziert sich um den Faktor $L$ der Vektorlänge.
Es werden nacheinander die ($L$,$L$)- und die (1,$L$)-Vektorstrategien besprochen. Die Laufzeit von BLOCK berechnete (3.8). Die Laufzeiten von VSCAP sind in den Tabellen 3.4 und 3.5 zusammengefaßt. Mit $L = 1$, $t_{\mathrm{v}} = t_{\mathrm{v}}^{L}$ und $t_{\mathrm{z}} = t_{\mathrm{z}}^{L}$ erhält man aus den VSCAP-Laufzeiten die die Kommunikationszeiten für das SCAP-Verfahren.
Um die maximale Laufzeitreduktion von VSCAP berechnen zu können, wird von einem minimalen Aufwand für das Absetzen einer Vektorvorlade- und Zugriffsoperation ausgegangen ($t_{\mathrm{v}}^{L} = t_{\mathrm{v}}$ und $t_{\mathrm{z}}^{L} = t_{\mathrm{z}}$).

#### 3.2.3.1 Vorbetrachtungen

Für die späteren Berechnungen muß noch die Existenz dreier Konstanten gezeigt werden. Die Voraussetzung für die Berechnung der Fälle 1 bis 3 bei der ($L$,$L$)- und (1,$L$)-Vektorstrategie war, daß die Arbeitsfrequenz des Netzwerkes geringer als die Anforderungsfrequenz der Vorladeschleife ausfällt ($t_{\mathrm{n}} \leq t_{\mathrm{v}}$). Auf der anderen Seite läßt sich $t_{\mathrm{v}}$ nach oben durch die Latenzzeit des Netzwerkes abschätzen ($t_{\mathrm{v}} \leq T_{\mathrm{Latenz}}$). Wäre das nicht der Fall, dann ließe sich zeigen, daß die Netzwerklatenzzeit unterhalb der Ausführungszeit einer Adreßberechnung sowie dem Absetzen einer Vorladeoperation liegen müßte. Das widerspricht jedoch der Basisannahme, daß die Latenzzeit mehrere hundert bis tausend Prozessortakte beträgt. Mit (2.1) von Seite 11 folgt damit die Existenz einer Konstanten $c_1$, die $t_{\mathrm{v}}$ in Beziehung zur Latenzzeit setzt.

$$t_{\mathrm{n}} \leq t_{\mathrm{v}} \leq T_{\mathrm{Latenz}} = C_N * t_{\mathrm{n}} \Rightarrow t_{\mathrm{v}} = \frac{T_{\mathrm{Latenz}}}{c_1} \text{ mit } 1 \leq c_1 \leq C_N \tag{3.19}$$

Mit derselben Begründung existiert ebenfalls eine Konstante $c_2$, welche die Zeit für das Absetzen einer Zugriffsoperation $t_{\mathrm{z}}$ in Beziehung zur Latenzzeit setzt.

$$t_{\mathrm{n}} \leq t_{\mathrm{z}} \leq T_{\mathrm{Latenz}} = C_N * t_{\mathrm{n}} \Rightarrow t_{\mathrm{z}} = \frac{T_{\mathrm{Latenz}}}{c_2} \text{ mit } 1 \leq c_2 \leq C_N \tag{3.20}$$

Bei der Berechnung der VSCAP-Laufzeiten für die $(L,L)$-Vektorstrategie werden zuviel berechnete Schleifenkontrollen mit dem Aufwand $t_s$ abgezogen. Die Schleifenkontrollen bestehen aus einem arithmetischen Befehl und einem bedingten Sprung, so daß sie sehr schnell ausgeführt werden können. Deshalb wird davon ausgegangen, daß die Bearbeitung der Schleifenkontrollen schneller ist als der Netzwerktakt ($t_s \leq t_n$). Daraus folgt die Existenz einer dritten Konstanten, mit der auch $t_s$ in Beziehung zur Latenzzeit gesetzt werden kann.

$$t_s \leq t_n \leq T_{\text{Latenz}} = C_N * t_n \Rightarrow t_s = \frac{T_{\text{Latenz}}}{c_3} \text{ mit } c_3 \geq C_N \tag{3.21}$$

Analog zu den Berechnungen der Kommunikationszeiten werden auch hier bei den einzelnen Vektorstrategien nur die Fälle 1 bis 3 betrachtet.

#### 3.2.3.2 $(L,L)$-Vektorstrategie

Bei der $(L,L)$-Vektorstrategie werden Vektorvorlade- und Zugriffsoperationen eingesetzt. Durch die minimale Dauer der Vektoroperationen ($t_v = t_v^L$ und $t_z = t_z^L$) wird mit einer $L$-fachen Reduktion der VSCAP-Laufzeiten im Vergleich zu SCAP gerechnet.
Durch die Charakteristik der Applikationsparameter ($t_v = t_z$ und $t_v^L = t_z^L$) und dem minimalen Aufwand der Vektoroperationen gilt $t_v = t_z = t_v^L = t_z^L$. Im folgenden wird für Vorlade- und Zugriffsoperationen beim Vereinfachen der Term $t_v$ verwendet.

**Fall 1:** Der Laufzeitvorteil von VSCAP gegenüber BLOCK ergibt sich aus der Subtraktion der Gleichungen (3.8) und (3.12):

$$\begin{aligned} T_{\text{BLOCK}} - T_{\text{VSCAP}} &= K * t_z + K * T_{\text{Latenz}} - (\frac{K}{L} * t_z^L + T_{\text{Latenz}} + (L-1) * t_n) \\ &= K * t_v + K * T_{\text{Latenz}} - (\frac{K}{L} * t_v + T_{\text{Latenz}} + (L-1) * t_n) \\ &= (K-1) * T_{\text{Latenz}} + \frac{K}{L} * (L-1) * t_v - (L-1) * t_n \end{aligned}$$

Ersetzt man $t_n$ durch $t_v$ so wird die rechte Seite kleiner und mit $K \geq L$ gilt:

$$\begin{aligned} T_{\text{BLOCK}} - T_{\text{VSCAP}} &> (K-1) * T_{\text{Latenz}} + (\frac{K}{L} - 1) * (L-1) * t_v \\ &> (K-1) * T_{\text{Latenz}} \end{aligned}$$

Die Laufzeit von BLOCK wird um mehr als das $(K-1)$-fache der Netzwerklatenz verringert, denn die Zugriffsschleife wird durch die Vektoroperationen $L$-mal weniger ausgeführt.

Der Laufzeitvorteil der Vektorbefehle berechnet sich durch die Differenz der SCAP und der VSCAP-Laufzeit:

$$\begin{aligned} T_{\mathrm{SCAP}} - T_{\mathrm{VSCAP}} &= K * t_{\mathrm{z}} + T_{\mathrm{Latenz}} - (\frac{K}{L} * t_{\mathrm{z}}^{L} + T_{\mathrm{Latenz}} + (L-1) * t_{\mathrm{n}}) \\ &= K * t_{\mathrm{v}} + T_{\mathrm{Latenz}} - (\frac{K}{L} * t_{\mathrm{v}} + T_{\mathrm{Latenz}} + (L-1) * t_{\mathrm{n}}) \\ &= \frac{K}{L} * (L-1) * t_{\mathrm{v}} - (L-1) * t_{\mathrm{n}} \\ &> (\frac{K}{L} - 1) * (L-1) * t_{\mathrm{v}} \end{aligned}$$

Damit wird schon bei wenigen nicht-lokalen Datenzugriffen fast eine Laufzeitreduktion um den Faktor $L$ erreicht. Daß die $L$-fache Reduktion nicht ganz erreicht wird, liegt an der ersten längeren Wartezeit von VSCAP ($(L-1) * t_{\mathrm{n}}$), wenn auf die Bereitstellung des ganzen Vektors gewartet werden muß.

**Fall 2:** Zur Berechnung der Laufzeitdifferenz von BLOCK und VSCAP sind die Gleichungen (3.8) und (3.13) zu subtrahieren. Unter Berücksichtigung von Gleichung (3.19) ergibt sich:

$$\begin{aligned} T_{\mathrm{BLOCK}} - T_{\mathrm{VSCAP}} &= K * t_{\mathrm{v}} + K * T_{\mathrm{Latenz}} - \frac{K}{L} * (t_{\mathrm{v}}^{L} + t_{\mathrm{z}}^{L}) \\ &= K * t_{\mathrm{v}} + K * T_{\mathrm{Latenz}} - \frac{K}{L} * (t_{\mathrm{v}} + t_{\mathrm{v}}) \\ &= K * T_{\mathrm{Latenz}} + \frac{K}{L} * (L-2) * t_{\mathrm{v}} \\ &= (1 + \frac{L-2}{L * c_1}) * K * T_{\mathrm{Latenz}} \qquad 1 < c_1 \leq C_N \end{aligned}$$

Der Laufzeitvorteil von VSCAP ist durch die Vektorlänge und die Kapazität $C_N$ des Netzwerkes bestimmt. Der Spezialfall $C_N = 1$ wird ausgeschlossen, da dies einem blockierenden Netzwerk entsprechen würde. Es fällt auf, daß der Faktor der Latenzzeitverbergung $(1 + \frac{L-2}{L*c_1}) \geq 1$ für Vektorlängen $L \geq 2$ ist. Damit werden bei Vektoren der Länge 2 100% der Latenzzeit verborgen. Für längere Vektoren wird noch zusätzlicher Aufwand durch die Adreßberechnung eingespart, was die Einsparungen von über 100% erklärt.

Der Vorteil von VSCAP zu SCAP berechnet sich wieder aus der Differenz der beiden Laufzeiten:

$$\begin{aligned} T_{\mathrm{SCAP}} - T_{\mathrm{VSCAP}} &= K * (t_{\mathrm{v}} + t_{\mathrm{z}}) - \frac{K}{L} * (t_{\mathrm{z}}^{L} + t_{\mathrm{v}}^{L}) \\ &= K * (t_{\mathrm{v}} + t_{\mathrm{v}}) - \frac{K}{L} * (t_{\mathrm{v}} + t_{\mathrm{v}}) \\ &= \frac{L-1}{L} * K * (t_{\mathrm{v}} + t_{\mathrm{v}}) \\ &= \frac{L-1}{L} * T_{\mathrm{SCAP}} \end{aligned}$$

Das entspricht einer exakten $L$-fachen Reduktion der Kommunikationszeit des SCAP-Verfahrens. Das ist gleichbedeutend mit einer 50% Laufzeitreduktion für die kleinsten Vektoren der Länge 2. Auf der Cray T3E ist durch die Vektoren der Länge 8 mit einer 8-fachen (oder 87.5%) Reduktion der Kommunikationszeit zu rechnen.

**Fall 3:** Die Laufzeitdifferenz von BLOCK und VSCAP ergibt sich durch Subtraktion der Gleichungen (3.8) und (3.14). Mit den Gleichungen (3.19) und (3.21) gilt:

$$\begin{aligned}
T_{\text{BLOCK}} - T_{\text{VSCAP}} &= K * t_{\text{v}} + K * T_{\text{Latenz}} - (\frac{K}{L} * (t_{\text{v}}^L + t_{\text{z}}^L) - \frac{K - C_V + L}{L} * t_{\text{s}}) \\
&= K * t_{\text{v}} + K * T_{\text{Latenz}} - (\frac{K}{L} * (t_{\text{v}} + t_{\text{v}}) - \frac{K - C_V + L}{L} * t_{\text{s}}) \\
&= K * T_{\text{Latenz}} + \frac{K}{L}(L-2) * t_{\text{v}} - \frac{K * (1 - \frac{C_V - L}{K})}{L} * t_{\text{s}} \\
&= K * T_{\text{Latenz}} * (1 + \frac{L-2}{L * c_1} - \frac{1 - \frac{C_V - L}{K}}{L * c_3})
\end{aligned}$$

Mit $K > C_V - L$ ist $1 - \frac{C_V - L}{K} < 1$ und es wird

$$T_{\text{BLOCK}} - T_{\text{VSCAP}} > K * T_{\text{Latenz}} * (1 + \frac{L-2}{L * c_1} - \frac{1}{L * c_3}) \qquad 1 < c_1 \leq C_N \leq c_3$$

Die Reduktion der Kommunikationszeit hängt von der Vektorlänge $L$ und den Konstanten $c_1$ und $c_3$ ab. Bei einer Vektorlänge $L = 2$ und einer Netzwerkkapazität $C_N = 2$ ($c_3 = 2$) ist mit einer Latenzzeitreduktion von 75% zu rechnen, bei einer Netzwerkkapazität $C_N = 10$ sogar um 95%.

Der Vorteil der Vektorbefehle wird durch die Laufzeitdifferenz von SCAP und VSCAP berechnet:

$$\begin{aligned}
T_{\text{SCAP}} - T_{\text{VSCAP}} =& K * (t_{\text{v}} + t_{\text{z}}) - (K - C_V + L) * t_{\text{s}} \\
& - (\frac{K}{L} * (t_{\text{v}}^L + t_{\text{z}}^L) - \frac{K - C_V + L}{L} * t_{\text{s}}) \\
=& K * (t_{\text{v}} + t_{\text{v}}) - (K - C_V + L) * t_{\text{s}} \\
& - (\frac{K}{L} * (t_{\text{v}} + t_{\text{v}}) - \frac{K - C_V + L}{L} * t_{\text{s}}) \\
=& K * \frac{L-1}{L} * (t_{\text{v}} + t_{\text{v}}) - \frac{L-1}{L} * (K - C_V + L) * t_{\text{s}} \\
=& \frac{L-1}{L} * (K * (t_{\text{v}} + t_{\text{v}}) - (K - C_V + L) * t_{\text{s}}) \\
=& \frac{L-1}{L} * T_{\text{SCAP}}
\end{aligned}$$

Dies entspricht wie bei Fall 2 einer $L$-fachen Reduktion der Kommunikationszeit. Dieser Laufzeitvorteil ist nicht nur für wenige nicht-lokale Datenzugriffe sondern für einen Großteil der Probleme erreichbar, bei denen die entfenten Datenzugriffe die Größe des Vorladepuffers bei weitem übersteigen ($K \gg C_V - L$).

Die Fälle 4-6 werden nicht gesondert untersucht, da dort das Netzwerk langsamer als der Prozessor ($L * t_{\mathrm{n}} > t_{\mathrm{v}}^{L}$) ist. Dies führt dazu, daß bei einer Wartezeit des Prozessors seine Ausführungszeit mit der des Netzwerkes identisch ist. Die Ausführungszeit ist in diesem Fall unabhängig von der eingesetzten Vorladestrategie, und der Einsatz von VSCAP gegenüber SCAP ergibt keinen Vorteil. Erfährt der Prozessor jedoch keine Wartezeit, so sind die Vorteile aus den bisherigen Untersuchungen ersichtlich.
Die gezeigten Betrachtungen der Laufzeiten des VSCAP-Verfahrens im Vergleich mit BLOCK und SCAP wurden bewußt allgemein gehalten, um ein möglichst großes Einsatzspektrum von VSCAP abzudecken. Bei einer ($L$,$L$)-Vektorstrategie bedeutet die Vektorlänge $L$ eine $L$-fache Beschleunigung der Kommunikationszeit selbst wenn der Vorladepuffer noch nicht vollständig gefüllt werden konnte. Für die T3E bedeutet dies, daß schon bei weniger als $C_V = 128$ nicht-lokalen Datenzugriffen durch den Einsatz der Vektorbefehle eine Beschleunigung von Faktor 8 erreicht werden kann. Im Vergleich zu BLOCK ergibt sich auf der T3E für viele entfernte Datenzugriffe $K \gg C_V$ und einer Netzwerkkapazität $C_N = 112$ eine Latenzzeitverbergung von mehr als 99%.

#### 3.2.3.3 (1,$L$)-Vektorstrategie

Bei der (1,$L$)-Vektorstrategie werden die nicht-lokalen Datenzugriffe durch einelementige Operationen vorgeladen. Der Zugriff erfolgt durch Vektorbefehle. Durch die komplexere Adreßberechnung beim Vorladen der Elemente war die Parametercharakteristik der (1,$L$)-Vektorstrategie durch das Verhältnis $t_{\mathrm{z}}^{L} < L * t_{\mathrm{v}}$ gegeben. Mit der minimalen Dauer der Vektorzugriffe gilt $t_{\mathrm{v}} > t_{\mathrm{z}} = t_{\mathrm{z}}^{L}$. Für die Vereinfachung wird für die Dauer der Vorladeoperationen $t_{\mathrm{v}}$ und die der Zugriffe $t_{\mathrm{z}}$ verwendet.

**Fall 1:** Der Vorteil von VSCAP und BLOCK berechnet sich aus der Diffenerenz der Gleichungen (3.8) und (3.16):

$$\begin{aligned} T_{\mathrm{BLOCK}} - T_{\mathrm{VSCAP}} &= K * t_{\mathrm{v}} + K * T_{\mathrm{Latenz}} - (K * t_{\mathrm{v}} + T_{\mathrm{Latenz}}) \\ &= (K-1) * T_{\mathrm{Latenz}} \end{aligned}$$

Wie man sieht, beträgt der Laufzeitvorteil von VSCAP das $(K-1)$-fache der Netzwerklatenz. Daß durch die Vektorbefehle kein gößerer Gewinn erzielt wird, liegt an der gegenüber der Zugriffsschleife vergleichsweise langsamen Vorladeschleife, die durch die Dauer der Vorladeoperationen dominiert ($t_{\mathrm{v}} > t_{\mathrm{z}}$) wird.

Aus diesem Grund lohnt sich der Einsatz der Vektorbefehle von VSCAP gegenüber SCAP nicht, und es ist unabhängig von der Vektorlänge $L$:

$$\begin{aligned} T_{\mathrm{SCAP}} - T_{\mathrm{VSCAP}} &= K * t_{\mathrm{v}} + T_{\mathrm{Latenz}} - (K * t_{\mathrm{v}} + T_{\mathrm{Latenz}}) \\ &= 0 \end{aligned} \tag{3.22}$$

**Fälle 2 und 3:** Der Laufzeitvorteil von VSCAP gegenüber BLOCK ergibt sich aus der Differenz der Gleichungen (3.8) und (3.17). Mit Gleichung (3.20) gilt:

$$\begin{aligned} T_{\text{BLOCK}} - T_{\text{VSCAP}} &= K * t_{\text{v}} + K * T_{\text{Latenz}} - (K * t_{\text{v}} + \frac{K}{L} * t_{\text{z}}^{L}) \\ &= K * t_{\text{v}} + K * T_{\text{Latenz}} - (K * t_{\text{v}} + \frac{K}{L} * t_{\text{z}}) \\ &= K * T_{\text{Latenz}} - \frac{K}{L} * t_{\text{z}} \\ &= K * T_{\text{Latenz}} * (1 - \frac{1}{L * c_2}) \qquad 1 < c_2 \leq C_N \end{aligned}$$

Der Laufzeitvorteil hängt nur von der Vektorlänge und der Kapazität $C_N$ des Netzwerkes ab. Bei einer minimalen Vektorlänge von $L = 2$ und $c_2 = 2$ verdeckt VSCAP 75% der Latenzzeit, bei $c_2 = 5$ sogar 90%.

Den Laufzeitgewinn durch den Einsatz der Vektorbefehle zeigt die Differenz von SCAP und VSCAP:

$$\begin{aligned} T_{\text{SCAP}} - T_{\text{VSCAP}} &= K * t_{\text{v}} + K * t_{\text{z}} - (K * t_{\text{v}} + \frac{K}{L} * t_{\text{z}}^{L}) \\ &= K * t_{\text{v}} + K * t_{\text{z}} - (K * t_{\text{v}} + \frac{K}{L} * t_{\text{z}}) \\ &= \frac{L-1}{L} * K * t_{\text{z}} \end{aligned}$$

Setzt man die Vorladedauer $t_{\text{v}}$ mit $u * t_{\text{z}} = t_{\text{v}}$ ins Verhältnis zur Zugriffsdauer $t_{\text{z}}$, mit $u > 1$, so gilt weiter:

$$\begin{aligned} T_{\text{SCAP}} - T_{\text{VSCAP}} &= \frac{L-1}{L} * K * \frac{t_{\text{v}} + t_{\text{z}}}{u+1} \\ &= \frac{L-1}{L*(u+1)} * K * (t_{\text{v}} + t_{\text{z}}) \\ &= \frac{L-1}{L*(u+1)} * T_{\text{SCAP}} \end{aligned}$$

Der Laufzeitvorteil von VSCAP hängt nur von der Vektorlänge $L$ und dem Verhältnis $u$ von Vorlade- und Zugriffszeit ab. Für ein günstiges Verhältnis von $u = 2$ und einer minimalen Vektorlänge $L = 2$ kann die Kommunikationszeit von SCAP um ca. 17% reduziert werden. Bei einem schlechteren Verhältnis $u$ verringert sich der Vorteil, da der Laufzeitanteil der Vorladeschleife entsprechend größer wird und der Gewinn durch die Vektorbefehle nicht mehr nennenswert ins Gewicht fällt.

Wie im vorherigen Abschnitt wird auch hier auf eine gesonderte Betrachtung der Fälle 4-6 verzichtet, da sich bei einer Wartezeit die Laufzeit des Prozessors ebenfalls der des Netzwerkes angleicht.
Der berechnete Laufzeitvorteil von VSCAP fällt bei einer $(1,L)$-Vektorstrategie nicht so groß aus als bei der $(L,L)$-Vektorstrategie. Dies liegt zum einen am reduzierten Einsatz der Vektorbefehle, die nur die Zugriffszeit gegenüber SCAP verkürzen können, und zum anderen an der

aufwendigen Adreßberechnung für die Vorladeoperationen, welche die Abstände dominieren, mit denen der Prozessor auf die Daten zugreifen kann.
Dies führt dazu, daß sich für wenig nicht-lokale Zugriffe (auf der T3E sind dies 3-4 Stück, vgl. 5.2.3) Vektorbefehle im Vergleich zu SCAP überhaupt nicht lohnen. Bei mehreren entfernten Zugriffen gibt $1 - \frac{1}{L*C_N}$ die Latenzzeitverdeckung von VSCAP gegenüber BLOCK an. Für eine Vektorlänge von $L = 2$ und einer Kapazität von $C_N \geq 2$ sind dies mindestens 75%. Auf der T3E mit $L = 8$ und $C_N = 112$ kann sogar mit einer Reduktion der Kommunikationszeit von mehr als 99% gerechnet werden.
Den Vorteil, der aus dem Einsatz der Vektorbefehle für große Datenmengen ($K > C_V - L$) erwächst, gibt $\frac{L-1}{L*(u+1)}$ an. So spart eine Vektorlänge von $L = 2$ und ein Verhältnis $u = 2$ von Vorladezeit $t_{\mathrm{v}}$ zu Zugriffszeit $t_{\mathrm{z}}$ ca. 17% der Laufzeit. Auf die T3E übertragen, bedeutet das einen Gewinn von 25% bei einem Verhältnis von $u = 2.5$ (siehe Tabelle 5.5) und einer Vektorlänge $L = 8$.

### 3.2.4 Berechnung der Vektorlänge

Die Laufzeitanalyse von VSCAP im vorherigen Abschnitt zeigte, daß durch die Vektorbefehle der Länge $L$ mit einer $L$-fachen Beschleunigung gegenüber SCAP gerechnet werden kann. Die Bedingung für diese Laufzeitreduktion war ein entsprechend schnelles Netzwerk, das die Vektorelemente vor dem Absetzen des nächsten Vektorbefehles bearbeitet hatte ($L * t_{\mathrm{n}} \leq t_{\mathrm{v}}^L$). Dieser Abschnitt untersucht den Einfluß des Verhältnisses von Vorladegeschwindigkeit $t_{\mathrm{v}}^L$ und Netzwerkarbeitstakt $t_{\mathrm{n}}$ auf die Vektorlänge. Damit können Entscheidungen über sinnvolle Vektorlängen getroffen werden, die bei der Abarbeitung nicht in zusätzlichen Wartezeiten münden.
Der Ausgangspunkt der Analyse ist die ($L$,$L$)-Vektorstrategie, bei der mit Vektorbefehlen vorgeladen und zugegriffen wird. Der vorherige Abschnitt verlangte $L * t_{\mathrm{n}} \leq t_{\mathrm{v}}^L$ als Mindestdauer für die Bearbeitung eines Vektors durch das Netzwerk. Mit dieser Zeitspanne wurde garantiert, daß der Prozessor durch das Netzwerk keine Wartezeiten erfährt. Mit der Wahl von $L = \frac{t_{\mathrm{v}}^L}{t_{\mathrm{n}}}$ ist garantiert, daß bei wenigen nicht-lokalen Datenzugriffen $K \leq C_V - L$ für den Prozessor keine Wartezeiten entstehen, wenn die erste Wartezzeit durch Vektorvorladebefehle verdeckt werden konnte ($\frac{K}{L} * t_{\mathrm{v}}^L \geq T_{\mathrm{Latenz}} + (L-1) * t_{\mathrm{n}}$), siehe Tabelle 3.2.
Die Wahl der Vektorlänge mit $L = \frac{t_{\mathrm{v}}^L}{t_{\mathrm{n}}}$ ist jedoch eine untere Schranke, wie sich im folgenden für mehrere nicht-lokale Datenzugriffe $K \gg C_V - L$ herausstellt. Denn wird der Vorladepuffer vollständig gefüllt ($K \geq C_V - L$), so greift der Prozessor ohne Wartezeit auf den ersten Vektor zu (vgl. (3.3) von Seite 31). Damit der Prozessor auch beim $i$-ten ($i \gg 1$) Durchlauf des Vorladepuffers innerhalb der kombinierten Zugriffs- und Vorladeschleife auf alle Vektoren verzögerungsfrei zugreifen kann, muß das Netzwerk alle $C_V - L$ Einträge innerhalb einer Zeit von $\frac{C_V - L}{L} * (t_{\mathrm{z}}^L + t_{\mathrm{v}}^L - t_{\mathrm{s}})$ bearbeitet haben. Das ist nämlich genau die Zeit, die der Prozessor für $\frac{C_V - L}{L}$ Iterationen der kombinierten Zugriffs- und Vorladeschleife benötigt.
Damit gilt für die obere Schranke der Vektorlänge

$$
\begin{aligned}
T_{\mathrm{Prozessor}} &\geq T_{\mathrm{Netzwerk}} \\
\frac{C_V - L}{L} * (t_{\mathrm{z}}^L + t_{\mathrm{v}}^L - t_{\mathrm{s}}) &\geq (C_V - L) * t_{\mathrm{n}} \\
\frac{t_{\mathrm{z}}^L + t_{\mathrm{v}}^L - t_{\mathrm{s}}}{t_{\mathrm{n}}} &\geq L
\end{aligned}
$$

Mit dieser Vektorlänge sind die Forderungen der Laufzeitberechnungen im vorherigen Abschnitt mit $L * t_{\mathrm{n}} \leq t_{\mathrm{v}}^L$ nicht falsch, denn für $L = \frac{t_{\mathrm{v}}^L + t_{\mathrm{z}}^L - t_{\mathrm{s}}}{t_{\mathrm{n}}}$ werden in der Zugriffsschleife Wartezeiten für den Prozessor erwartet, die sich jedoch durch die große Anzahl $K \gg C_V - L$ an nicht-lokalen Datenzugriffen nicht weiter negativ auswirkt.
Damit erhält man mit $t_{\mathrm{z}}^L = t_{\mathrm{v}}^L$ für identische Adreßberechnungen und der Tatsache daß $t_{\mathrm{s}}$ nur wenige Prozessortakte beansprucht, folgenden Bereich für die optimale Vektorlänge.

**Satz 1 (Bereich für Vektorlängen)**
*In einem $(C_N, t_n)$-überlappenden Netzwerk sind die möglichen Vektorlängen $L$ einer $(C_V, L)$-Vorladestrategie mit einem Vorladeaufwand von $t_{\mathrm{v}}^L$ durch*

$$\frac{t_{\mathrm{v}}^L}{t_n} \leq L \leq 2 * \frac{t_{\mathrm{v}}^L}{t_n} \tag{3.23}$$

*beschränkt.*

Ob nun $L = \frac{t_{\mathrm{v}}^L}{t_{\mathrm{n}}}$ oder $L = 2 * \frac{t_{\mathrm{v}}^L}{t_{\mathrm{n}}}$ gewählt wird, hängt von der Größe $C_V$ des Vorladepuffers ab. Denn ist $C_V$ verhältnismäßig klein ($C_V \approx C_N$), so werden die meisten Anwendungen mehr als $C_V - L$ nicht-lokale Datenzugriffe aufweisen, so daß sich längere Vektoren lohnen. Sollte jedoch $C_V \gg C_N$ sein, so ist anzunehmen, daß der Vorladepuffer nicht immer vollständig gefüllt wird, so daß eine eher kürzere Vektorlänge vorteilhaft ist.
In jedem Fall muß für eine minimale Vektorlänge die Netzwerkarbeitszeit $t_{\mathrm{n}}$ schneller als die Vorladezeit $t_{\mathrm{v}}^L$ des Prozessors sein ($t_{\mathrm{n}} < t_{\mathrm{v}}^L$). Das schränkt aber den Einsatz des hier vorgestellten VSCAP-Verfahrens auf dedizierte Parallelrechnerarchitekturen ein, denn nur dort kann durch eine proprietäre Netzwerkarchitektur dieses Verhältnis geschaffen werden.
Für den allgemeinen Einsatz des VSCAP-Verfahren muß auf optoelektronische Verbindungsnetzwerke gewartet werden, in denen mit Netzwerkarbeitszeiten im Bereich der Prozessortaktraten entsprechend günstige Verhältnisse $\frac{t_{\mathrm{v}}^L}{t_{\mathrm{n}}} \geq 1$ vorgefunden werden. Die gleichzeitige Reduktion der Latenzzeit macht den Einsatz des VSCAP-Verfahrens nicht überflüssig, denn die Analyse der Systemlaufzeiten zeigte, daß selbst bei sehr wenigen nicht-lokalen Zugriffen schon mit einer deutlichen Reduktion der Kommunikationszeit gerechnet werden kann.

## 3.3 Zusammenfassung

Das SCAP-Verfahren war der Ausgangspunkt dieses Kapitels. Seine Erweiterung um Vektorbefehle ließ daraus eine ganze Familie von Kommunikationsmodi wachsen, die eine Einteilung nach Größe des Vorladepuffers und eingesetzter Vektorlänge für die Operationen nötig machte. So ist jetzt eine detailiertere Unterteilung in $(C_V, L)$-Vorladestrategien möglich, Abbildung 3.5.
Die Berechnung der Laufzeitvorteile von VSCAP gegenüber SCAP prognostizieren einen $L$-fachen Laufzeitvorteil durch den Einsatz der Vektorbefehle. VSCAP läßt weiterhin gegenüber BLOCK schon bei sehr kleinen Vektoren ab $L = 2$ und einer Netzwerkkapazität von $C_N = 10$ eine 95% Reduktion der Latenzzeit erwarten. Überträgt man das Modell auf die T3E, so kann VSCAP eine Latenzzeitreduktion von mehr als 99% erreichen.
Jedoch ist der Preis für diesen Laufzeitgewinn sehr hoch, denn VSCAP wird sich in den nächsten Jahren nur in Parallelrechnern mit einem integrierten Hochleistungsnetzwerk lohnen,

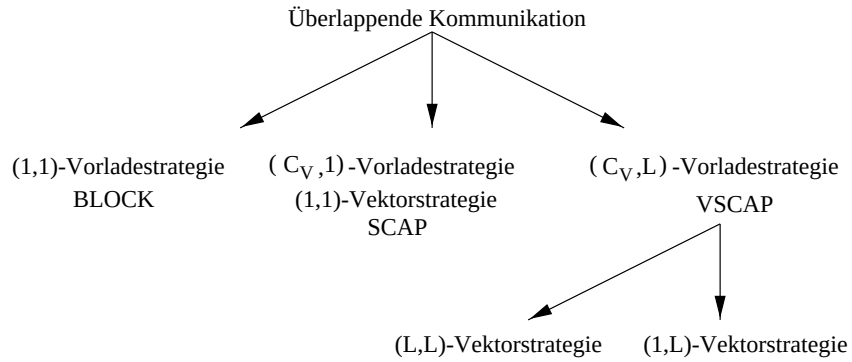



Abbildung 3.5: Strategien der überlappenden Kommunikation

wenn der Netzwerktakt mit der Prozessorarbeitsgeschwindigkeit mithalten kann, wie es zum Beispiel bei der Cray T3E der Fall ist, siehe Tabelle 5.3 von Seite 90.
Abhilfe werden auf diesem Gebiet optoelektronische Netzwerke schaffen, die VSCAP auch für weitläufige Rechnerverbunde rentabel gestalten.

# Kapitel 4

# Transformationsmuster für VSCAP

Dieses Kapitel zeigt in Abschnitt 4.1 wie die Vektorstrategien von VSCAP auf Transformationsmuster in einem optimierenden Übersetzer abgebildet werden können. Abschnitt 4.2 skizziert die Architektur der Cray T3E und geht auf einige Details der E-Registerprogrammierung ein. Danach stellt Abschnitt 4.3 den Prototypübersetzer KarHPFn vor, der HPF (High Performance Fortran) in Programme übersetzt, die zur Kommunikation ausschließlich die Kommunikationsfließbänder des VSCAP-Verfahrens benützen.

## 4.1 Transformationsmuster für Vektorstrategien

Dieser Abschnitt zeigt, wie die Vektorstrategien von 3.1.5 auf Datenfließbänder des VSCAP-Verfahrens abgebildet werden können. An die allgemeinen Vorbetrachtungen von 4.1.1 über eine mögliche Implementierung schließt die Beschreibung der verschiedenen Transformationsmuster an. Sie orientiert sich an der Art der eingesetzten Vektorstrategie. Abschnitt 4.1.5 bildet schließlich Datenverteilungen und Kommunikationsmuster auf die einzelnen Transformationsmuster ab.

### 4.1.1 Vorbetrachtungen

Für die Implementierung von VSCAP in einen optimierenden Übersetzer sind folgende Betrachtungen wichtig:

- Die Implementierung ist architekturunabhängig, denn die Datenfließbänder des VSCAP-Verfahrens setzen lediglich Vorlade- und Zugriffsoperationen voraus. Bei der Portierung eines Übersetzers auf eine andere Architektur müssen damit nur diese Anweisungen auf die Zieloperationen abgebildet werden. Eine aufwendige Reimplementierung der Datenfließbandgenerierung entfällt somit, da nur das entsprechende Modul für die Maschinenschnittstelle ausgetauscht werden muß.

- Die einseitigen Kommunikationsprimitive verringern den Aufwand für die Analyse der nicht-lokalen Speicherzugriffe, denn es müssen nicht wie in nachrichtengekoppelten Systemen Sende- und Empfangsmengen berechnet werden [78, 76, 14, 115] (zweiseitige Kommunikation), da jeder Prozessor selbständig Daten aus dem gemeinsamen Adreßraum lesen kann. Dies verringert nicht nur den Analyseaufwand, sondern es reduziert

auch bei dynamischen Kommunikationsmustern die Anzahl der Kommunikationsoperationen.

- Die Datenfließbänder werden direkt in den Programmtext des Zielprogramms geschrieben. Damit können zum einen Optimierungen für entfernte Adressberechnungen vorgenommen werden und es besteht zum anderen die Möglichkeit, daß die so erhaltenen Programme durch den Knotenübersetzer weiter optimiert werden können. Dies vergrößert zwar das ausführbare Programm geringfügig, doch fällt dieser Nachteil durch den Laufzeitgewinn nicht ins Gewicht.

- Die Datenfließbänder werden ausschließlich zur Kommunikation verwendet. Eine zusätzliche Verknüpfung mit lokaler Berechnung, wie sie Warschko vorgeschlagen hatte, entfällt, wie es bereits in 3.1.2 begründet wurde. Deshalb wird im weiteren von der Kommunikation auch als *datenparalleler Zuweisung* gesprochen.

- Für die Generierung der Fließbänder muß die Verteilung der beteiligten Datenfelder bekannt sein. Dies ist für Sprachen ohne dynamische Datenumverteilung kein Problem, da hier das Feld einmal mit einer fest definierten und nicht veränderbaren Verteilung angelegt wird. Problematisch sind Programmiersprachen, die mehrere Verteilungen eines Feldes zur Laufzeit zulassen, denn dann muß mit Datenflußtechniken die aktuelle Verteilung bestimmt werden. Sollte die Datenverteilung nicht eindeutig sein, d.h. sollte ein Feld an einer Stelle im Programmtext mehrere verschiedene Verteilungen besitzen können, so muß durch Klonen der entsprechenden Programmstelle für jede mögliche Datenverteilung ein eigenes Fließband generiert werden. Danach wird erst zur Laufzeit anhand der aktuellen Datenverteilung das passende Fließband gewählt. Der in dieser Arbeit entwickelte Übersetzer geht von einer statischen Verteilung aus, die zur Laufzeit nicht geändert werden kann (siehe 4.3.2).

- Die Auswahl der lokal zu emulierenden Prozessoren geschieht mit Techniken aus [121]. Die Menge der aktiven Prozessoren wird mit einem Trippel $(Start, Stop, Inc)$ beschrieben, das den ersten $Start$, den letzten $Stop$ und den Abstand $Inc$ zwischen den einzelnen Prozessoren beschreibt. Die Anzahl der aktiven Prozessoren $length$ berechnet sich durch

$$length = \frac{Stop - Start}{Inc} + 1$$

  Das Tripel $(Start, Stop, Inc)$ ist mit dem lokalen Iterationsraum identisch und kann daher zur Berechnung der globalen Adressen verwendet werden. Es werden auch zwei weitere Bezeichner eingeführt, welche die aktuelle Vorladeadresse $I_{Prefetch}$ und Zugriffsadresse $I_{Access}$ speichern.

Die folgenden Abschnitte komplettieren die Betrachtungen zur Implementierung des VSCAP-Verfahrens.

### 4.1.2 $(L,L)$-Vektorstrategie

$(L, L)$-Vektorstrategien werden bei statischen Kommunikationsmustern eingesetzt, d.h. wenn die entfernten Datenzugriffe zur Übersetzungszeit bestimmt werden können. Die Schrittwei-

te braucht dabei noch nicht festzustehen; es genügt, wenn sie konstant ist. Die *Einblock*-Transformation bildet die Grundlage für $(L, L)$-Vektorstrategien. Sie kopiert einen zusammenhängenden Datenblock von *einem* entfernten Knoten in den lokalen Speicher. Auf ihr bauen die *Mehrblock*-Transformation und Reduktionen auf.

#### 4.1.2.1 Einblock-Transformation

Mit dem *Einblock*-Fließband wird ein Feld von Datenelementen von genau einem entfernten Prozessor gelesen. Abbildung 4.1 zeigt das Kommunikationsmuster für eine *Einblock*-Transformation.

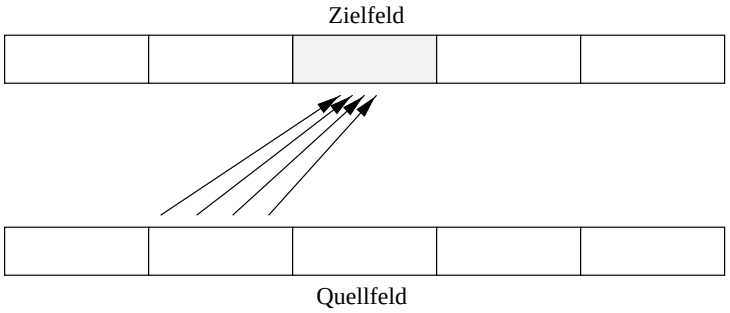



Abbildung 4.1: Kommunikationsmuster für *Einblock*-Transformation

Das Datenfließband enthält nur zwei Zeiger, einen auf die Quell- und einen auf die Zieladresse, die dann durch die jeweiligen Schrittweiten erhöht werden. Zur Laufzeit müssen dazu die lokale und entfernte Startadresse, die lokale und die entfernte Schrittweite und die Anzahl der entfernten Zugriffe bestimmt werden. Die Nummer des Quellknotens ist bereits durch die entfernte Startadresse beschrieben (vgl. Abbildung 2.1). Das entsprechende Fließband wurde in 3.1.4 vorgestellt.
Die *Einblock*-Transformation wird direkt bei n-dimensionaler Nachbarschaftskommunikation mit blockweise verteilten Feldern eingesetzt, bei denen die Ränder angrenzender Felder in Überlappungsbereiche [47] (*overlap area*) kopiert werden. Dazu werden die lokalen Felder um einen konstanten Bereich vergrößert, der dann die entfernten Datenelemente aufnimmt.

#### 4.1.2.2 Mehrblock-Transformation

Die *Mehrblock*-Transformation wird bei Kommunikationsmustern eingesetzt, bei denen die Daten auf mehrere entfernte Knoten verteilt sind. Dabei wird für jeden entfernten Knoten das *Einblock*-Fließband separat eingesetzt. Die *Mehrblock*-Transformation ist somit eine geblockte Version der *Einblock*-Transformation mit dem Grad der Virtualisierung als Blockgröße. Abbildung 4.2 zeigt das entsprechende Kommunikationsmuster, bei dem über mehrere Prozessoren verteilte Datenelemente in den lokalen Speicher kopiert werden müssen. Die gestrichelten Linien bedeuten lokale Speicherzugriffe, die eine Aufspaltung der Kommunikation in drei separate Kopiervorgänge nach sich ziehen (nicht-lokale, lokale und nicht-lokale Datenzugriffe).
Vor jedem Gebrauch des *Einblock*-Fließbandes werden die Startadressen von Quell- und Zielfeld und die Nummer des Netzwerkknotens neu berechnet.

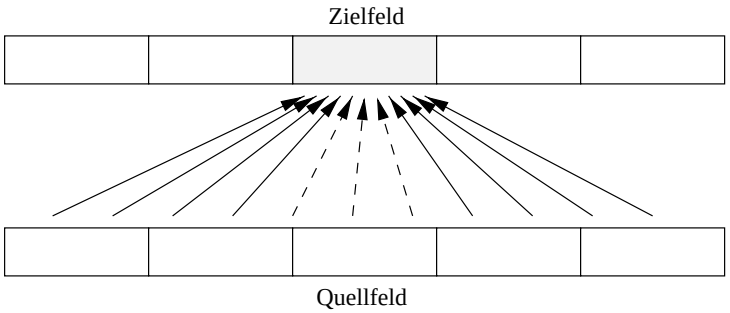



Abbildung 4.2: Kommunikationsmuster für *Mehrblock*-Transformation

Die *Mehrblock*-Transformation wird vorwiegend bei affinen Indexausdrücken und blockweise verteilten Feldern verwendet.
Ein Spezialfall der *Mehrblock*-Transformation ist die 1-dimensionale Nachbarschaftskommunikation, bei der durch die geringe Anzahl an entfernten Datenzugriffen ($\mathcal{K}(i) = i + c$ mit $c \leq C_V$ als Konstante), alle nicht-lokalen Daten auf einmal in den Vorladepuffer geladen werden können. Damit können auch Latenzzeiten von Zugriffen verschiedener Netzwerkknoten miteinander überlappt werden, was in der originalen *Mehrblock*-Transformation nicht möglich war.

#### 4.1.2.3 Reduktionen

Bei einer Reduktion berechnet zunächst jeder Prozessor für sich ein lokales Ergebnis. Die $P$ Einzelergebnisse der Prozessoren werden dann in $log_2(P)$ Schritten zu einem globalen Ergebnis zusammengefaßt. Am Ende greift jeder Prozessor auf das Endergebnis zu und lädt es zu sich lokal in den Hauptspeicher. Den entsprechenden Kommunikationsplan zeigt Abbildung 4.3.
Warschko vergrößert den Verzweigungsgrad $f$ der Reduktion, um mehr entfernte Datenzugriffe für SCAP zu erhalten, mit denen die Latenzzeit verdeckt werden kann. Setzt man in seinen Ansatz die Parameter der Cray T3E ein, so erhält man einen optimalen Verzweigungsgrad von $f = 16$, der ein Kompromiß zwischen Latenzzeitverbergung und Laufzeiteffizienz darstellt. Eine Vergrößerung von $f$ für VSCAP und Messungen auf einer verschiedenen Anzahl an Prozessoren führte jedoch durch zu große Schwankungen in den Meßergebnissen zu keinem brauchbarem Ergebnis, so daß für alle in dieser Arbeit durchgeführten Reduktionen ein Verzweigungsgrad von $f = 16$ benutzt wurde.
Die $log_f(P)$ Kommunikationsschritte werden durch *Einblock*-Fließbänder übernommen, bei denen nicht wie bisher die prozessorlokale Speicheradresse sondern die Prozessornummer der globalen Adresse aus Abbildung 2.1 verändert wird. Nachdem die $P$ Einzelergebnisse auf einem Prozessor $i$ zusammengefaßt worden sind, greifen alle anderen Prozessoren auf das berechnete Endergebnis von Prozessor $i$ zu. Damit ergibt sich bei einer Reduktion ein Kommunikationsvolumen von $f * log_f(P) + 1$ nicht-lokalen Datenzugriffen.

### 4.1.3 (1,$L$)-Vektorstrategie

Die $(1, L)$-Vektorstrategie wird bei dynamischen Kommunikationsmustern eingesetzt, bei denen die entfernten Datenzugriffe willkürlich über den gesamten gemeinsamen Adreßraum verstreut sind, die einzelnen Daten jedoch lokal mit fester Schrittweite gebraucht werden. Das

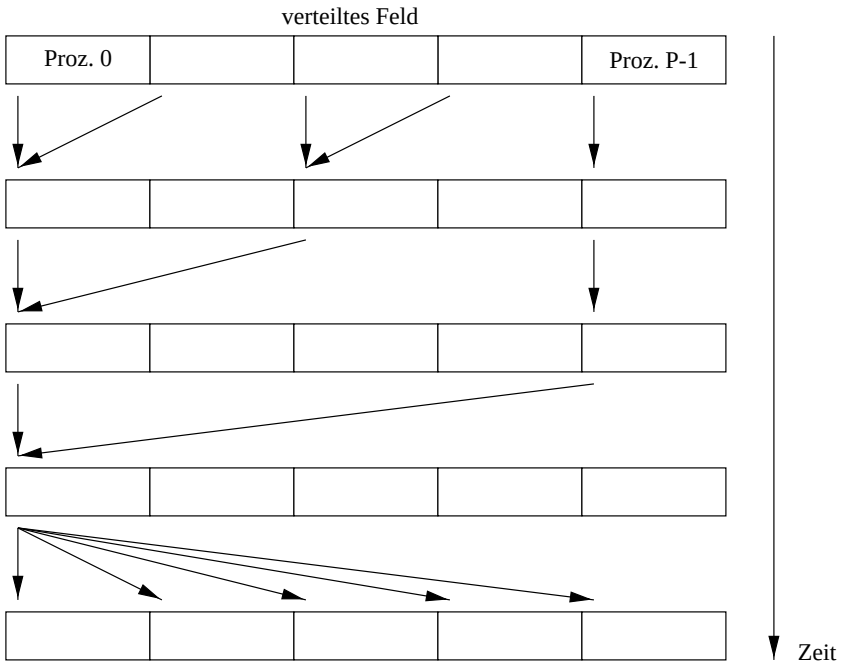



Abbildung 4.3: Reduktion mit Verzweigungsgrad 2

entsprechende Transformationsmuster heißt *Einsammeln* (*gather*), siehe Abbildung 4.4.

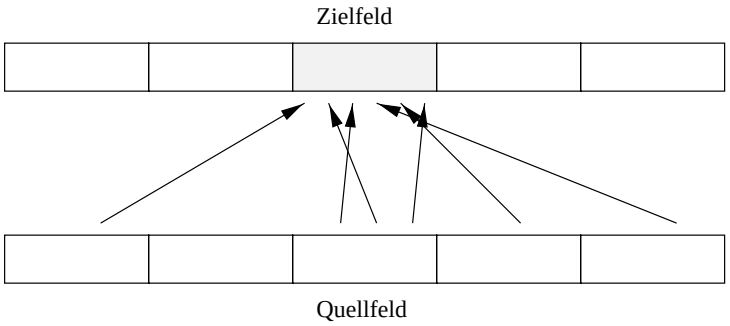



Abbildung 4.4: Kommunikationsmuster für Transformation *Einsammeln*

Für Netzwerkschnittstellen, die den gesamten globalen Adreßraum (auch den lokalen) abdecken, können Vektorbefehle für die Zugriffsoperationen verwendet werden. Ansonsten muß mit einem Laufzeittest die Lokalität der Daten getestet werden. Dies resultiert jedoch in einer $(1, 1)$-Vektorstrategie, da der gleichbleibende Abstand der entfernten Daten beim Rückschreiben nicht mehr gewährleistet werden kann. Diese Vektorstrategie wird im nächsten Abschnitt besprochen. Das entsprechende Fließband ist in 3.1.5 bei der $(1,L)$-Vektorstrategie gezeigt.

### 4.1.4 (1,1)-Vektorstrategie

Eine $(1,1)$-Vektorstrategie wird bei dynamischen Vorlade- und Zugriffsoperationen gewählt. Statisch nicht berechenbare lokale und nicht-lokale Datenzugriffe treten bei einem *Softwaretest*, der die Lokalität der Zugriffe überprüft, bei *maskierten Zuweisungen* und bei der Übersetzung von *block-zyklisch* verteilten Feldern auf. Die in diesem Abschnitt vorgestellten Transformationsmuster behandeln zueinander orthogonale Problemstellungen, so daß eine VSCAP-Implementation beliebige Kombinationen unterstützen muß.

#### 4.1.4.1 Softwaretest

Ein *Softwaretest* ist nötig, wenn die Netzwerkschnittstelle nicht in den lokalen Speicher greifen kann, so daß vor jedem Zugriff die Lokalität des Datums überprüft werden muß. Für die Abarbeitung des Fließbandes wird ein zusätzlicher Puffer benötigt, der den Index entfernter Datenelemente für den späteren Zugriff zwischenspeichert. Das prinzipielle Vorgehen ohne die mittlere Schleife ($length \leq C_V$) zeigt folgendes Fließband am Beispiel der Zuweisung `A[i]=B[q(i)]`:

Counter := 0;
$I_{Prefetch}$ := $Start_A$;
**FOR** i = 0 **TO** length-1 **DO**
  addr := calculate_address(B[q($I_{Prefetch}$)]);
  **IF** PeNr(addr) $\neq$ MyPe **THEN**
    prefetch(addr);
    IndexBuffer[Counter] := $I_{Prefetch}$;
    Counter := Counter + 1;
  **ELSE**
    A[$I_{Prefetch}$] := addr;
  **END IF**
  $I_{Prefetch}$ := $I_{Prefetch}$ + $Inc_A$;
**END FOR**

Zuerst wird die Zieladresse *addr* berechnet. Der anschließende Test (PeNr(addr) $\neq$ MyPe) unterscheidet zwischen lokalen und nicht-lokalen Zugriffen. Bei nicht-lokalen Zugriffen wird das Datum vorgeladen, der aktuelle Index in `IndexBuffer` zwischengespeichert und der Zähler für die entfernten Zugriffe erhöht. Andernfalls wird das Datum aus dem lokalen Hauptspeicher gelesen (A[$I_{Prefetch}$] := addr).
Die Größe von `IndexBuffer` ist mit der Kapazität des Vorladepuffers identisch, da für jeden Eintrag des Vorladepuffers ein separater Index $I_{Prefetch}$ existiert.

**FOR** i = 0 **TO** Counter-1 **DO**
  addr := calculate_address(B[q(IndexBuffer[Counter])]);
  A[IndexBuffer[Counter]] := access(addr);
**END FOR**

Die Zugriffsschleife läuft dann noch über die *Counter* entfernten Zugriffe. Sollte die Netzwerkschnittstelle jedoch auch auf den lokalen Speicherbereich zugreifen können, so muß über den Vorzug des *Softwaretests* gegenüber *Einsammeln* durch einen Effizienzvergleich entschieden werden.

#### 4.1.4.2 Maskierte Zuweisungen

Maskierte Zuweisungen sind durch ein Prädikat `Mask` geprägt, das den Zugriff auf entfernte Daten überwacht:

**FORALL** i = 0 **TO** N-1 **DO**
  **IF** Mask(i) **THEN**
    A[i] := B[q(i)];
  **END IF**
**END FORALL**

Im Fließband entscheidet nun nicht die Lokalität über Netzwerkzugriff sondern der Wert des Prädikats:

Counter := 0;
$I_{Prefetch}$ := $Start_A$;
**FOR** i = 0 **TO** length-1 **DO**
  **IF** Mask($I_{Prefetch}$) **THEN**
    addr := calculate_address(B[q($I_{Prefetch}$)]);
    prefetch(addr);
    IndexBuffer[Counter] := $I_{Prefetch}$;
    Counter := Counter + 1;
  **END IF**
  $I_{Prefetch}$ := $I_{Prefetch}$ + $Inc_A$;
**END FOR**

Wie beim Softwaretest werden durch die Abfrage Mask($I_{Prefetch}$) innerhalb der Vorladeschleife nicht-lokale Datenzugriffe selektiert, das entsprechende Element wird vorgeladen, der Index wird zwischengespeichert und der Zähler für vorgeladene Daten wird erhöht. Die Zugriffsschleife kann vom vorherigen Beispiel übernommen werden.
Die Ausführung der obigen Vorladeschleife setzt eine Netzwerkschnittstelle voraus, die auf den gesamten globalen Adreßraum und damit auch auf den prozessorlokalen Speicher zugreifen kann. Sollte dies nicht der Fall sein, so müssen der Softwaretest vom vorherigen Abschnitt und die maskierte Zuweisung kombiniert werden.

#### 4.1.4.3 Block-zyklische Verteilungen

Das Problem bei block-zyklischen Verteilungen mit einer Blockgröße $k > 1$ (`CYCLIC(k)`) sind Adreßmengen (Mengen aktiver lokaler Prozessoren), die bei der Übersetzung nicht berechnet werden können. Tabelle 4.1 zeigt die Zugriffsfolge bei einer Blockgröße von $k = 4$, bei drei physikalischen Prozessoren und einer Schrittweite von $Inc = 5$. Die Kästchen markieren die aktiven Indizes während der Zuweisung.
In der Literatur gibt es eine Vielzahl von Algorithmen mit den verschiedensten Ansätzen, um die Zugriffsfolgen für einen Prozessor effizient zu berechnen. Techniken aus der linearen Algebra [9, 91, 56, 112, 14] benötigen für die Adreßgenerierung eine Schleifenschachtel der Tiefe 2, die sich nicht weiter reduzieren läßt. Damit können diese Techniken nicht ohne weiteren Laufzeitaufwand für das VSCAP-Verfahren eingesetzt werden. In [84] wird jede Adresse auf ihre Zugehörigkeit zum Iterationsraum getestet, was für eine effiziente Ausführung

Tabelle 4.1: Adressen bei einer block-zyklischen Verteilung mit Blockgröße $k = 4$, mit $P = 3$ Prozessoren und einer Schrittweite von $Inc = 5$

| Prozessor 0 | | | | Prozessor 1 | | | | Prozessor 2 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| [0] | 1 | 2 | 3 | 4 | [5] | 6 | 7 | 8 | 9 | [10] | 11 |
| 12 | 13 | 14 | [15] | 16 | 17 | 18 | 19 | [20] | 21 | 22 | 23 |
| 24 | [25] | 26 | 27 | 28 | 29 | [30] | 31 | 32 | 33 | 34 | [35] |
| 36 | 37 | 38 | 39 | [40] | 41 | 42 | 43 | 44 | [45] | 46 | 47 |
| 48 | 49 | [50] | 51 | 52 | 53 | 54 | [55] | 56 | 57 | 58 | 59 |
| [60] | 61 | 62 | 63 | 64 | [65] | 66 | 67 | 68 | 69 | [70] | 71 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

zu Aufwendig ist. Bei den tabellengesteuerten Methoden [28, 76, 115] wird der Umstand ausgenützt, daß sich die Adreßfolge für einen Prozessor nach $k$ Zugriffen wiederholt. Zur Laufzeit werden diese Adressen berechnet und jeweils die Differenz von Adresse $i$ auf Adresse $i-1$ in einer Tabelle `Delta` gespeichert. Zu diesen Deltas wird die Start- und die maximale lokale Adresse berechnet. Die Adreßfolge wird durch Aufaddieren der Deltas auf einen Zeiger generiert, wobei die Deltas modulo der Tabellengröße $k$ indiziert werden.

Da die Anzahl der entfernten Elemente nicht mehr geschlossen berechnet werden kann, denn sie hängt von den einzelnen Deltas ab, zählt die Vorladeoperationen nur noch die Vorladeschleife. Dadurch wird ein Überlaufen des Vorladepuffers vermieden. Die Abbruchbedingungen der beiden weiteren Schleifen ergeben sich aus dem Überschreiten der letzten Adresse. Als Beispiel dient wieder die Permutation `A[i]=B[q(i)]`:

PrefetchCounter := 0;
$\mathrm{I_{Prefetch}}$ := StartAddress;
**WHILE** PrefetchCounter $< C_V \wedge \mathrm{I_{Prefetch}} \leq$ LastAddress **DO**
  addr := calculate_address(B[q($\mathrm{I_{Prefetch}}$)]);
  prefetch(addr);
  $\mathrm{I_{Prefetch}}$ := $\mathrm{I_{Prefetch}}$ + Delta[MOD(PrefetchCounter,k)];
  PrefetchCounter := PrefetchCounter + 1;
**END WHILE**

Die Vorladeschleife iteriert solange wie der Vorladepuffer noch nicht gefüllt ist (PrefetchCounter $< C_V$) und die Vorladeadresse $\mathrm{I_{Prefetch}}$ die letzte Adresse noch nicht überschritten hat ($\mathrm{I_{Prefetch}} \leq$ LastAddress). Im Schleifenrumpf wird das entfernte Datenelement vorgeladen und die Vorladeadresse durch Aufaddieren eines Deltas erhöht. Die Deltas werden modulo der Blockgröße $k$ indiziert, da sie sich nach $k$ Zugriffen wiederholen:

  $\mathrm{I_{Prefetch}}$ := $\mathrm{I_{Prefetch}}$ + Delta[MOD(PrefetchCounter,k)];

Die kombinierte Zugriffs- und Vorladeschleife wird solange ausgeführt, bis auch das letzte Element vorgeladen wurde ($\mathrm{I_{Prefetch}} \leq$ LastAddress). Die Adreßfolge der Zugriffe wird nach dem gleichem Schema wie die Vorladeadreßfolge generiert.

```
AccessCounter := 0;
I_Access := StartAddress;
WHILE I_Prefetch ≤ LastAddress DO
  addr := calculate_address(B[q(I_Access)]);
  A(I_Access) := access(addr);
  I_Access := I_Access + Delta[MOD(AccessCounter,k)];
  AccessCounter := AccessCounter + 1;

  addr := calculate_address(B[q(I_Prefetch)]);
  prefetch(addr);
  I_Prefetch := I_Prefetch + Delta[MOD(PrefetchCounter,k)];
  PrefetchCounter := PrefetchCounter + 1;
END WHILE
```

Die Zugriffsschleife lädt zum Schluß alle im Vorladepuffer verbliebenen Elemente in den lokalen Speicher.

```
WHILE I_Access ≤ LastAddress DO
  addr := calculate_address(B[q(I_Access)]);
  A(I_Access) := access(addr);
  I_Access := I_Access + Delta[MOD(AccessCounter,k)];
  AccessCounter := AccessCounter + 1;
END WHILE
```

#### 4.1.4.4 Vergleich mit *Inspector-Executor*

Die vorangegangenen Transformationsmuster zeigten recht deutlich den Unterschied zum *Inspector-Executor*-Modell von 2.3.5. Die Vorladeschleife des VSCAP-Verfahrens *inspiziert* nicht nur die Datenelemente, sondern sie setzt auch die für die Kommunikation notwendigen Vorladebefehle ab. Im *Inspector-Executor*-Modell sind jedoch Inspektion, Kommunikation und Ausführung drei von einander getrennte Programmeinheiten, die auch zeitlich nacheinander ausgeführt werden. Durch die einseitigen Kommunikationsprimitive von VSCAP müssen auch vor der Ausführung des Datenfließbandes keine weiteren Elemente kommuniziert werden, was bei der zweiseitigen Kommunikation nachrichtengekoppelter Systemen nötig war, um Quell und Zielprozessor einer Nachricht bestimmen zu können.
Zusammengefaßt ist VSCAP eine sehr einfache Variante des *Inspector-Executor*-Modells, das im Vergleich nicht nur geringeren Berechnungsaufwand sondern auch durch die einseitigen Kommunikationsprimitive weniger Kommunikationsaufträge zu bewältigen hat.

### 4.1.5 Übersetzung verschiedener Datenverteilungen und Kommunikationsmuster

Tabelle 4.2 zeigt die Zuordnung von Datenverteilung und Kommunikationsmuster auf die beschriebenen Transformationsstrategien. Die Selektion orientiert sich nach einem Mustervergleich, bei dem die Indexausdrücke der verteilten Felder ausgewertet und miteinander

verglichen werden [81]. Die Indexanalyse beinhaltet auch verschiedene Ausrichtungen (*Alignment*) der verteilten Felder untereinander, so daß nur die resultierenden Muster aufgeführt sind. Die Beschreibung der Kommunikationsmuster ist aus [21] übernommen.

Tabelle 4.2: Abbildung der Kommunikationsmuster auf Transformationsregeln

| **Kommunikationsmuster** | **Beschreibung** | **Transformationsmuster** |
|---|---|---|
| Datenverteilung: *BLOCK* oder *BLOCK(k)* | | |
| $\mathcal{K}(i)=b^*$ | Verteilen | Einblock |
| $\mathcal{K}(i)=i+c^\dagger$ | Überlappendes Kopieren | Einblock |
| $\mathcal{K}(i)=i+b$ | Temporäres Kopieren | Mehrblock |
| $\mathcal{K}(i)=a*i+b$ | Affines Kopieren | Mehrblock |
| $\mathcal{K}(i)=V(i)$ | Indirekter Zugriff | Einsammeln |
| $\mathcal{K}(i)=f(i)$ | Funktionsaufruf | Einsammeln |
| Datenverteilung: *CYCLIC(1)* | | |
| $\mathcal{K}(i)=b$ | Verteilen | Einblock |
| $\mathcal{K}(i)=i+c$ | Überlappendes Kopieren | Einblock |
| $\mathcal{K}(i)=i+b$ | Temporäres Kopieren | Einblock |
| $\mathcal{K}(i)=a*i+b$ | Affines Kopieren | Einsammeln, Einblock |
| $\mathcal{K}(i)=V(i)$ | Indirekter Zugriff | Einsammeln |
| $\mathcal{K}(i)=f(i)$ | Funktionsaufruf | Einsammeln |
| Datenverteilung: *CYCLIC(K)* | | |
| $\mathcal{K}(i)=b$ | Verteilen | Einblock |
| $\mathcal{K}(i)=i+c$ | Überlappendes Kopieren | Mehrblock, Einblock |
| $\mathcal{K}(i)=i+b$ | Temporäres Kopieren | Mehrblock |
| $\mathcal{K}(i)=a*i+b$ | Affines Kopieren | Einsammeln |
| $\mathcal{K}(i)=V(i)$ | Indirekter Zugriff | Einsammeln |
| $\mathcal{K}(i)=f(i)$ | Funktionsaufruf | Einsammeln |

$^*b$ ist eine beliebige Variable
$^\dagger c$ ist eine Konstante

Die oberen beiden Abschnitte von Tabelle 4.2 zeigen für *BLOCK*- (oder *BLOCK(k)*- mit $k \geq N/P$) und *CYCLIC(1)*-Verteilungen speziell optimierte Transformationsmuster, da diese beiden Verteilungen in der Praxis so häufig anzutreffen sind, so daß sich eine spezielle Betrachtung lohnt. Das untere Drittel betrachtet allgemeine *CYCLIC(k)*-Verteilungen. Diese Verteilungen können bei Schrittweiten größer eins aufgrund ihrer lokalen Adreßzuordnung nur mit einer $(1,1)$-Vektorstrategie behandelt werden, so daß ihr Einsatz bei der Kommunikation verglichen mit den Spezialfällen Effizienzeinbußen zur Folge hat.
Die einseitigen Kommunikationsprimitve resultieren bei Verteiloperationen in einem geringerem Kommunikationsaufwand als in nachrichtengekoppelten Architekturen. Denn mußten dort $log(P)$ Schritte durchgeführt werden, um ein Datum auf alle $P$ beteiligten Prozessoren zu verteilen, so kann dies nun in $O(1)$ geschehen, da jeder Prozessor das Datenelement zu sich in den lokalen Speicher liest.
Die vorgestellten Abbildungen beziehen sich auf nicht maskierte Datenzugriffe. Sollten die Datenzugriffe jedoch durch ein Prädikat gesteuert werden, so wird für die Kommunikation die maskierte Zuweisung mit einer $(1,1)$-Vektorstrategie eingesetzt.

### 4.1.6 Zusammenfassung

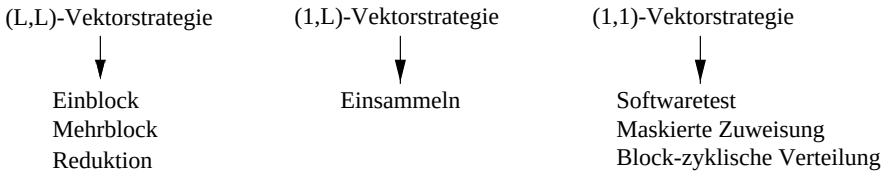



Abbildung 4.5: Abbildung der Vektorstrategien auf die Transformationsmuster

Abbildung 4.5 faßt die Zuordnung der Vektorstrategien auf die einzelnen Transformationsmuster noch einmal zusammen.

## 4.2 Die Architektur der Cray T3E

Dieser Abschnitt stellt die Architektur der Cray T3E vor. Der Schwerpunkt liegt auf der Netzwerkschnittstelle, den *E-Registern*, die erst den Einsatz von VSCAP auf der T3E ermöglichen. Nachdem 4.2.1 die Wahl der Zielarchitektur begründet, stellt 4.2.2 die hier relevanten Charakteristiken der Zielarchitektur vor. Der Kern dieses Abschnitts bildet 4.2.3 mit der Programmierung der E-Register. Hier werden architekturspezifische Zusammenhänge erklärt, die das Verständnis der Ergebnisse von Kapitel 5 erleichtern und die die Implementierung der Datenfließbänder in 4.2.4 beeinflußten.

### 4.2.1 Entscheidung für die Cray T3E

Für die Wahl der Zielplattform standen zwei Architekturen zur Auswahl, die einen Vorladepuffer für entfernte Speicherzugriffe bereitstellen. Das sind die Parallelrechner T3D und T3E von SGI/Cray. Mit dem Vorladepuffer der T3D können 16 entfernte Schreib- und Lesezugriffe überlappt werden [10]. Dies reicht aus, um die Netzwerklatenz zu einem Knoten zu verdecken. In datenparallelen Anwendungen können jedoch nicht-lokale Speicherzugriffe mehrere Knoten weit entfernt sein. Dies hat zur Folge, daß viel größere Netzwerklatenzen verdeckt werden müssen, für die der Vorladepuffer der T3D zu klein ist.
Die T3E stellt im Vergleich dazu einen Vorladepuffer mit 480 Einträgen zur Verfügung: Die sogenannten *E-Register*.[1] Ein entfernter Datenzugriff über die E-Register kann in eine Vorlade- und Zugriffsoperation aufgespalten werden, zwischen denen beliebige weitere Operationen vom Prozessor ausgeführt werden können. Die Aufspaltung der Leseoperationen und die Anzahl der E-Register ermöglichen es, ein Datenfließband aufzubauen, mit dem die Latenzzeiten datenparalleler Anwendungen verdeckt werden können. Damit fiel die Wahl auf die Cray T3E.

### 4.2.2 Überblick

Die Cray T3E [97] ist ein MIMD-Rechner [41] mit physisch verteiltem Speicher und gemeinsamen Adreßraum. Jeder Knoten besteht aus einem DEC Alpha EV5 (21164 Implementierung) Mikroprozessor, zwischen 64 bis 2048 MB Hauptspeicher und einer lokalen Netzwerkverbindung, siehe Abbildung 4.6.

[1] E steht dabei für *Extern*, da sie außerhalb des Prozessors liegen.

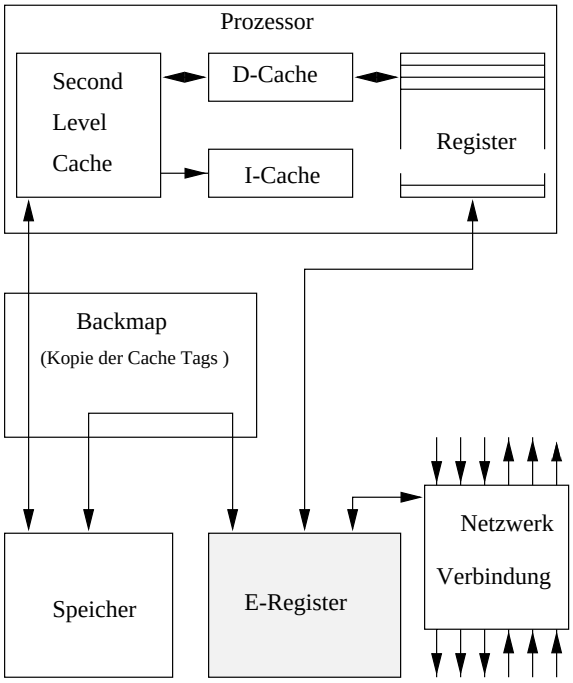



Abbildung 4.6: Aufbau eines Knotens der Cray T3E

Weiterhin besitzt jeder Knoten einen eigenen Satz von E-Registern. Sie sind die einzige Möglichkeit, um Daten von einem entfernten Speicher zu lesen oder in einen entfernten Speicher zu schreiben. Die Konsistenz von lokalem Hauptspeicher und Prozessorcache wird durch die *Backmap* erreicht, die eine Kopie der Einträge des Prozessorcaches führt. Durch diese Einträge werden bei Schreibzugriffen entfernter Prozessoren auf den lokalen Speicher die entsprechenden Zeilen des lokalen Prozessorcaches auf ungültig gesetzt. Der Zugriff (lesend und schreibend) auf entfernte Speicherstellen und die Konsistenthaltung des lokalen Prozessorcaches wird vollständig von der Netzwerkschnittstelle übernommen (*einseitige Kommunikation*).
Das Netzwerk ist ein 3-dimensionaler Torus mit Transferraten von ca. 500 MB/s bidirektional in jede der drei Koordinatenrichtungen. Das ergibt eine Bandbreite von 3 GB/s pro Netzwerkknoten. Die $P$ Netzwerkknoten einer Applikation sind von 0 bis $P-1$ durchnumeriert.
Die Programmierung der T3E folgt dem SPMD-Programmiermodell nachrichtengekoppelter Architekturen, doch entfällt hier der Softwareaufwand für das Ein- und Auspacken der zu versendenden Daten (*Cray-Modell* von Tabelle 2.1). Die Synchronisation der einzelnen Netzwerkknoten wird durch ein im Netzwerk integriertes *Barrier/Eureka*-Netz gewährleistet.

### 4.2.3 Die Netzwerkschnittstelle

Die E-Register sind die einzige Möglichkeit zur Kommunikation in der T3E. Sie liegen zwar physikalisch außerhalb des Prozessors (Abbildung 4.6) aber logisch im adressierbaren Ein/-Ausgabebereich. Dies ermöglicht den Einsatz herkömmlicher Prozessoren und läßt die Rechnerleistung vom stetigen Zuwachs der Prozessoren profitieren. Befehle vom Prozessor zu den E-Registern sind Schreib- und Leseoperationen. Sie gehen am Prozessorcache vorbei.

Schreib- und Leseoperationen der E-Register über das Netzwerk werden *put* und *get* genannt. Mit ihnen kann der ganze gemeinsame Adreßraum (auch der lokale) angesprochen werden. Es gibt insgesamt 512 E-Register auf jedem Knoten, von denen 480 frei verfügbar sind und als Vorladepuffer eingesetzt werden können. Sie werden durch ihren Index voneinander unterschieden. Um einen größeren Durchsatz bei der Kommunikation zu bekommen, können genau 8 E-Register zu einem Vektor zusammengefaßt und gemeinsam angesprochen werden. Weiterhin können mehrere E-Register lesend auf dieselbe Speicherstelle zugreifen. Mehrere Schreiboperationen auf dieselbe Speicherstelle sollten jedoch vermieden werden, da das Resultat unbestimmt ist. Damit hat die T3E einen CREW-Speicher.

#### 4.2.3.1 Ablauf einer get-Operation

*Get*-Operationen auf der T3E sind in eine Vorlade- und eine Zugriffsanweisung aufgeteilt. Zwischen diesen beiden Operationen kann der Prozessor beliebige weitere Anweisungen ausführen. Dies beinhaltet auch get-Operationen mit anderen E-Registern. Nach dem Absetzen einer entfernten Leseanfrage wird das zugehörige E-Register auf *leer* gesetzt. Der Zustand wechselt auf *voll*, sobald das Netzwerk die entfernte Leseoperation ausführen konnte. Greift der Prozessor früher auf dieses E-Register zu, so wird er solange blockiert, bis es seinen Zustand wechselt. Liest der Prozessor ein bereits gefülltes E-Register, so kann er ohne Verzögerung weiterrechnen.

#### 4.2.3.2 Programmierung der E-Register

Bei *get*- und *put*-Operationen führt der Prozessor Schreibanweisungen in seinen Ein/Ausgabebereich aus

```
E-Register_Befehl(RegisterNummer) = Index.
```

Die linke Seite der Zuweisung beschreibt den Befehl und indiziert das entsprechende E-Register mit der `RegisterNummer`. Dabei ist für jedes E-Register und jeden Befehl eine eigene Adresse reserviert. Die rechte Seite, der `Index`, zeigt auf die Quelle eines *get* bzw. das Ziel eines *put*. Die Berechnung des Knotens und der lokalen Adresse auf diesem Knoten erfolgt in zwei Schritten in der sogenannten *Hardware-Zentrifuge* [105]. Dazu wird ein Block von vier weiteren E-Registern benötigt. Dieser Block wird durch einen Bereich im oberen Teil des *Indexes* adressiert (grauer Bereich in Abbildung 4.7).

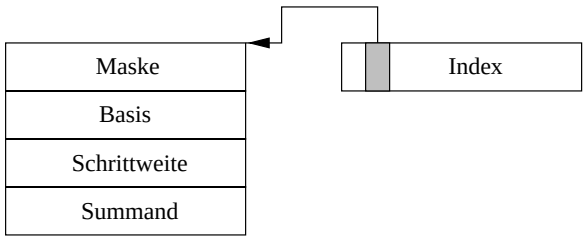



Abbildung 4.7: E-Register für entfernten Speicherzugriff

Die *Maske*, die *Basis* und die *Schrittweite* werden für die Berechnung der Knotennummer und der knotenlokalen Adresse gebraucht.

Nach der Adressierung des zusätzlichen E-Registerblockes wird der Zeiger aus dem *Index* entfernt.

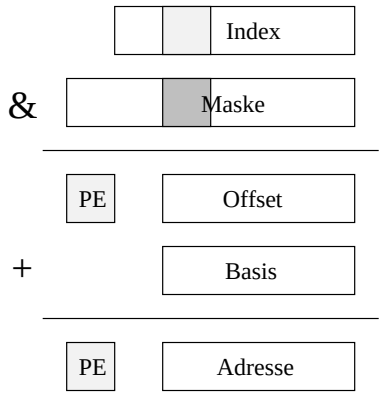



Abbildung 4.8: Adreßberechnung in der *Hardware-Zentrifgue*

Im ersten Schritt der Adreßumsetzung, siehe Abbildung 4.8, wird aus dem verbliebenen[2] *Index* durch die gesetzten Bits der *Maske* die Knotennummer selektiert. Das Ergebnis ist der *Offset* und die virtuelle Knotennummer *PE*. Mit dem zweiten Schritt, der Addition von *Offset* und *Basis*, erhält man schließlich die virtuelle *Adresse*. Die weitere Abbildung der virtuellen Knotennummer und Adresse auf physikalische Größen ist hier nicht weiter wichtig und wird deshalb nicht betrachtet.
Bei Vektorkommandos gibt die *Schrittweite* den Abstand zwischen den einzelnen Vektorelementen an. Sie wird vor der Selektion der Knotennummer zum *Index* hinzuaddiert, der dann zur Berechnung der weiteren Vektoradressen zwischengespeichert wird. Der *Summand* wird für atomare Lese-und-Addiere Operationen (*fetch-and-add*) gebraucht. Er wird hier jedoch nicht verwendet.
Um Adreßberechnungen von der Software in die Hardware verlagern zu können, zeigt die *Basis* auf den knotenlokalen Anfang eines verteilten Datenfeldes. Der *Index* enthält dann nur noch den globalen Index der datenparallelen Zuweisung, der für die Berechnung der Knotennummer und des knotenlokalen Offsets relevant ist. Die *Maske* wird so intialisiert, daß sie die für die Knotenummer signifikanten Bits selektiert. Die Programmierung der *Maske* zeigt 4.3.4.
In einem Datenparallelen Programm existieren damit prinzipiell für jedes verteilte Datenfeld ein separater Satz von vier E-Registern. Sollte die Anzahl der E-Register nicht ausreichen, so muß der separate E-Registerblock einmal vor jeder Ausführung eines Datenfließbandes entsprechend initialisiert werden.

### 4.2.4 Auswirkungen auf VSCAP

Die Programmierung der E-Register hat mehrere Auswirkungen auf die Realisierung der Datenfließbänder:

- Der Vorladepuffer muß explizit verwaltet werden. Das Modell machte bisher keine konkreten Angaben über die Zuordnung von Vorlade- und Zugriffoperation auf den

[2] Die Größe der einzelnen Adreßbereiche ist für die Arbeit nicht weiter wichtig, sie wird lediglich schematisch dargestellt.

gleichen Eintrag des Vorladepuffers. Auf der T3E werden die E-Register von 0 bis 511 indiziert. Um eine optimale Latenzzeitverbergung zu erreichen, werden die E-Register reihum (*round-robin*) adressiert. Nach jeder Operation wird der Zähler modulo der eingesetzten E-Register erhöht. Um diese Berechnung effizient ausführen zu können, werden immer $2^n$ (in KARHPFN ist $n = 8$) E-Register adressiert, um für die Modulo-Arithmetik ein logisches UND mit Maske $2^n - 1$ verwenden zu können.

- Vor dem Einsatz von VSCAP wird neben der *Maske* und der *Basis* auch die *Schrittweite* (vgl. Abbildung 4.7) gesetzt. Da Vektorbefehle vom Prozessor genauso schnell abgesetzt werden können wie einzelne E-Registerbefehle, führt dies zu einer Laufzeitbeschleunigung von Faktor 8.

- Im vorherigen Abschnitt wurde erwähnt, daß die E-Registerbefehle Speicheradressen darstellen. Es gibt für jeden Befehl einen dedizierten Speicherbereich, in dem die einzelnen E-Register separat angesprochen werden. Damit stehen die Adressen einer Vorlade- und einer Zugriffsanweisung für dasselbe E-Register an unterschiedlichen Stellen im Speicher. Damit ergibt sich bei einer dreischleifigen VSCAP- (und natürlich auch SCAP-)Abarbeitung ein Problem in der kombinierten Zugriffs- und Vorladeschleife, in der Zugriff- und Vorladeanweisung auf dasselbe E-Register stattfinden. Denn die Logik des prozessorinternen Schreibpuffers, der den Prozessor vom langsameren Bustakt entkoppelt, erlaubt, daß sich Schreibvorgänge auf verschiedene Zieladressen überholen dürfen. Da E-Registerbefehle Schreibzugriffe sind, kann somit die Vorladeanweisung der mittleren Schleife den vorherigen Zugriff auf das gleiche (!) E-Register überholen. Für den Erhalt der Reihenfolge könnte man eine Speicherbarriere einfügen, doch die würde den Schreibpuffer erst einmal leer laufen lassen, was unnötig Zeit kosten würde. Eine andere Möglichkeit ist die zeitliche Entkopplung der Zugriff- und Vorladeoperation auf das gleiche E-Register. Dazu werden statt einem zwei Zähler, ein Vorlade- und ein Zugriffszähler, eingesetzt, siehe Abbildung 4.9.

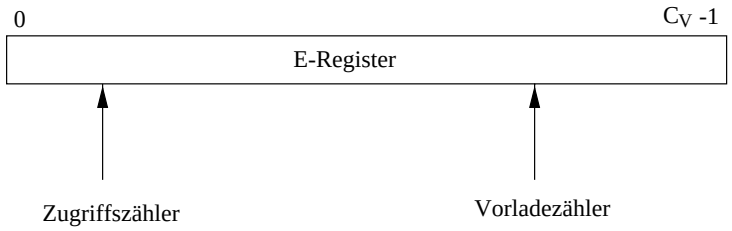



Abbildung 4.9: E-Registeradressierung mit zwei Zählern

Beide Zähler adressieren die E-Register reihum, wobei der Vorladezähler immer die Vorladedistanz vorausläuft. Der Abstand zwischen Nachlade und Vorladezähler modulo $C_V$ ist die *Nachladedistanz* mit der die Zugriffe auf das gleiche E-Register entzerrt werden.

- Die *Hardware-Zentrifuge* kann dazu benützt werden, die Umrechnung der globalen in lokale Adressen in die Hardware zu verlagern. Dies stellt jedoch gewisse Anforderungen an die Anzahl der beteiligten Prozessoren und die Problemgröße, weshalb dieser Einsatz nicht allgemeingültig ist. Diese spezielle Eigenschaft der T3E wird in Abschnitt 4.3.4 genauer erklärt.

## 4.3 KarHPFn

Dieser Abschnitt stellt den im Rahmen dieser Arbeit entwickelten Prototypübersetzer KarHPFn (*Kar*lsruher *H*igh *P*erformance *F*ortra*n*) vor.[3] Nach einer kurzen Übersicht 4.3.1 zeigt 4.3.2 den Stand der Implementierung und den von KarHPFn unterstützten Sprachumfang. Die Kommandozeilenoptionen zur Steuerung der Kommunikationsgenerierung listet 4.3.3 auf, und 4.3.4 widmet sich der speziellen Programmierung der T3E eigenen Hardware-Zentrifuge.

### 4.3.1 Übersicht

Der Einsatz der VSCAP-Verfahrens ist in datenparallelen Sprachen begünstigt, da dort durch die Virtualisierung genügend entfernte Datenzugriffe vorhanden sind, mit denen die Netzwerklatenzzeit verdeckt werden kann. Die datenparallelen Zuweisungen finden sich zum Beispiel in parallelen Schleifen naturwissenschaftlicher Anwendungen. Da hier Fortran und im immer größeren Maße auch HPF [42, 43] (High Performance Fortran) eingesetzt wird, wurde entschieden, das Verhalten eines automatisch generierten VSCAP am datenparallelen Forall-Konstrukt von HPF zu testen.
Abbildung 4.10 gibt einen Überblick über den Prototypübersetzer KarHPFn.


Abbildung 4.10: Der KarHPFn Übersetzer

KarHPFn akzeptiert ein datenparalleles HPF Programm und gibt ein SPMD Programm aus, das zur Kommunikation das VSCAP-Verfahren mit den E-Register Operationen enthält. Danach wird mit dem *Fortran 90*-Übersetzer auf der T3E ein ausführbares Knotenprogramm erzeugt.
Als Grundlage von KarHPFn dient das Frontend von *ADAPTOR*, einem von Thomas Brandes an der GMD entwickelten Übersetzungssystem [24]. Auf dem von ihm berechneten attributierten Strukturbaum setzen die weiteren Phasen zur Generierung des Datenfließbandes auf. Die Implementierung erfolgt mit dem Übersetzerwerkzeug *PUMA* [51] aus der *Cocktail-Toolbox* [52].
Parallelismus wird in KarHPFn ausschließlich durch das Forall-Konstrukt ausgedrückt. Mit der Datenverteilung und einer Indexanalyse beteiligter Datenfelder wird anhand von Tabelle 4.2 das entsprechende Transformationsmuster gewählt. Die Datenfließbänder werden dann direkt in das Knotenprogramm eingefügt. Um die Kommunikation so effizient wie möglich zu gestalten, wurde auf die Implementierung einer eigenen Kommunikationsbibliothek verzichtet, was die Komplexität der Fließbandgenerierung zwar erhöht, sich aber in einer höheren Effizienz im ausführbaren Programm niederschlägt (vgl. 4.1.1). Eine Ausnahme bilden die Ein- und Mehrblock-Transformationen, denn sie hängen nur von der Anzahl der nicht-lokalen Datenzugriffe, der Virtualisierung und dem Abstand der einzelnen Elemente ab, so daß sie als Makros realisiert wurden.

[3] sprich Karpfen

Das Ziel der Arbeit war kein Übersetzer, der den kompletten Sprachumfang von HPF unterstützt. Das Hauptaugenmerk lag auf der Generierung der Kommunikationsfließbänder, um die Leistungsfähigkeit von VSCAP bei einer Vielzahl von Kommunikationsmustern zu testen. Die Fließbänder werden von KarHPFn automatisch und ohne weitere Unterstützung vom Programmierer generiert, so daß der Einsatz von VSCAP zur Kommunikation vollkommen transparent bleibt.
Der nächste Abschnitt gibt einen kurzen Überblick über den von KarHPFn unterstützten Sprachumfang.

### 4.3.2 Stand der Implementierung

KarHPFn unterstützt *Subset*-HPF [42]. Subset-HPF wurde eigens zum Zweck definiert, um Firmen oder Forschergruppen einen Standard für erste Übersetzer bereitzustellen. Zum vollen HPF-Sprachumfang fehlen Subset-HPF Direktiven zur dynamischen Umverteilung (*Redistribute*) und Ausrichtung (*Realign*-Direktiven) von verteilten Feldern. Eine Einschränkung des Subset-HPF-Standards gibt es in KarHPFn, denn in Align-Direktiven können keine verschiedenen Ausrichtungen der verteilten Felder zueinander angegeben werden. Dies stellt jedoch keine Einschränkung dar, da diese Ausrichtungen auch durch ein Vergrößern der jeweiligen Felder erreicht werden können [21].
KarHPFn wurde zwar unter der Vorgabe entwickelt, daß eine möglichst große Menge an Kommunikationsmustern unterstützt werden sollte, doch gibt es auch dort Einschränkungen, die jedoch in späteren KarHPFn-Versionen beseitigt werden:

- Bei n-dimensionaler Nachbarschaftskommunikation ($n \geq 2$) mit blockweiser Verteilung werden lediglich die Ränder, nicht aber die Ecken angrenzender Felder in den lokalen Speicher kopiert.
- Die von KarHPFn unterstützten Kommunikationsmuster zeigt Tabelle 4.3.

Tabelle 4.3: Von KarHPFn Unterstützte Kommunikationsmuster

| **Kommunikationsmuster** | **Beschreibung** | **Transformationsmuster** |
|---|---|---|
| Datenverteilung: *BLOCK(k)* | | |
| $\mathcal{K}(i)=b^*$ | Verteilen | Einblock |
| $\mathcal{K}(i)=i+c^\dagger$ | Überlappendes Kopieren | Einblock |
| $\mathcal{K}(i)=i+b$ | Temporäreres Kopieren | Mehrblock |
| Datenverteilung: *CYCLIC(1)* | | |
| $\mathcal{K}(i)=b$ | Verteilen | Einblock |
| $\mathcal{K}(i)=i+c$ | Überlappendes Kopieren | Einblock |
| $\mathcal{K}(i)=i+b$ | Temporäres Kopieren | Einblock |

*$b$ ist eine beliebige Variable
†$c$ ist eine Konstante

Alle anderen Kommunikationsmuster von Tabelle 4.2 werden mit *Einsammeln* übersetzt.

- KarHPFn erkennt Felder mit entfernten Datenzugriffen und fügt nötigenfalls temporäre Felder in die Berechnung ein. Sollte sich jedoch die Anzahl der Dimensionen

vom temporären zum Ausgangsfeld unterscheiden, versagt die automatische Ersetzung, da KarHPFn diesen Fall nicht behandeln kann. Dies ist vor allem bei Verteiloperationen wichtig, da dort durch die Verteiloperation die Anzahl der Dimensionen der temporären Felder kleiner ist als bei den Ausgangsfeldern. Hier ist es die Aufgabe des Programmieres, die temporären Felder korrekt in den Programmtext einzufügen.

- KarHPFn unterstützt die meisten der Fortran 90 Reduktionsfunktionen. Eine Ausnahme bilden die HPF-Reduktionen (XXX_PREFIX, XXX_SCATTER, XXX_SUFFIX), die nicht übersetzt werden können.

Wie die Vorladestrategie für einen Übersetzungslauf gewählt werden kann, zeigt der nächste Abschnitt.

### 4.3.3 Kommandozeilenoptionen

KarHPFn unterstützt eine Reihe von Kommandozeilenoptionen mit denen die Generierung der Kommunikationsfließbänder beeinflußt werden kann:

**-P BLOCK|SCAP|VSCAP|SHMEM** selektiert eine Vorladestrategie.

**BLOCK** ist die $(1, 1)$-Vorladestrategie eines blockierenden Netzwerkes.

**SCAP** sind die Datenfließbänder des SCAP-Verfahrens: $(C_V, 1)$-Vorladestrategie.

**VSCAP** sind Datenfließbänder mit einer $(C_V, 8)$-Vorladestrategie. Die gewählte Vektorstrategie und das korrespondierende Transformationsmuster ist aus Tabelle 4.3 ersichtlich. Bei maskierten Berechnungen wird die $(C_V, 1)$-Vorladestrategie von SCAP gewählt.

**SHMEM** benützt die *shared-memory* Bibliotheksfunktionen zur Kommunikation. Da diese Funktionen keine dynamischen Kommunikationsmuster unterstützen, kommt dieser Kommunikationsmodus bei diesen Mustern einer blockierenden Ausführung gleich. Die *Einblock*-Transformation wird auf `shmem_iget` abgebildet.

**-RUNTIME** erzwingt einen *Softwaretest* bei dynamischen Kommunikationsmustern. Da die E-Register auf den gesamten globalen Speicher zugreifen können, wird bei dynamischen Kommunikationsmustern die $(1, 8)$-Vektorstrategie mit der Transformation *Einsammeln* gewählt.

**-HWC** In besonderen Fällen kann mit dieser Option die Adreßberechnung durch die Hardware-Zentrifuge erfolgen. Ihre Programmierung und die Anwendung der Option erklärt der nächste Abschnitt.

### 4.3.4 Programmierung der Hardware-Zentrifuge

In 4.2.3.2 wurde die Programmierung der E-Register und die Adreßberechnung der Hardware-Zentrifuge auf der Cray T3E gezeigt. Dieser Abschnitt erklärt wie dieser Mechanismus in KarHPFn verwendet wird, um für bestimmte Problemgrößen die Berechnung der Knotennummern und lokalen Adressen von der Soft- in die Hardware auszulagern.
Die Benennung der beteiligten E-Register erfolgt nach Abbildung 4.7 von Seite 73. Die Umsetzung der globalen Adresse erfolgt in zwei Schritten, vgl. Abbildung 4.8 von Seite 74,

wobei im ersten Schritt die Selektion der Knotennummer aus dem *Index* vorgenommen wird. Die entsprechenden Bits werden durch die *Maske* angezeigt. Für eine beliebige block-zyklische Verteilung mit Blockgröße $k$ ist der Aufbau der *Maske* in Abbildung 4.11 gezeigt.

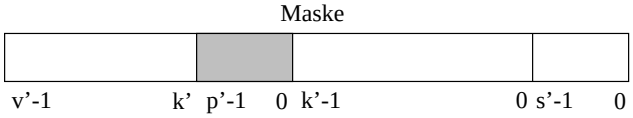



Abbildung 4.11: Berechnung der *Maske* des E-Registerblocks aus Abb. 4.7 für eine CYCLIC(k)-Verteilung, mit $k' = log_2(k)$, $p' = log_2(P)$, $s' = log_2(sizeof(Datatype))$ und $v' = log_2(V)$.

Die *Maske* wird dazu in vier Blöcke unterteilt. Der erste Block ganz rechts beinhaltet die Anzahl Bits $s' = log_2(sizeof(Datatyp))$, die für die richtige Ausrichtung des verwendeten Datentyps notwendig sind. Danach folgen die $k' = log_2(k)$ Bits für die Blockgröße $k$. Die Bits der ersten beiden Blöcke sind nicht gesetzt. Die $p' = log_2(P)$ Bits des dritten Blocks sind gesetzt, denn sie selektieren die Knotennummer. Die restlichen $v' - k'$ Bits ($v' = log_2(V)$) des vierten Blocks sind wieder gleich null. Diese Programmierung der *Maske* erspart damit die Berechnung der Prozessornummer durch die Software, so daß beim Einsatz dieses Adressierungsschemas mit einem Laufzeitgewinn innerhalb eines Kommunikationsfließbandes zu rechnen ist.
Die Adreßberechnung mit der Hardware-Zentrifuge ist natürlich nur möglich, wenn $k$ und $P$ 2-er Potenzen sind. Bei blockweiser Verteilung ist auch $V$ eine 2-er Potenz, da $V = k$. Dies sind die notwendigen Bedingungen für einen korrekten Einsatz der Option `-HWC`, die KarHPFn auf eine Adreßberechnung mit Hardware-Zentrifuge umschaltet. Für die Berechnung der Maske im mehrdimensionalen Fall wird nur einmal am Anfang die Größe des Datentyps berechnet und sonst für jede Dimension das oben beschriebene Verfahren angewendet.
Sind die obigen Bedingungen nicht erfüllt, so muß die Knotennummer $nr$ und die lokale Adresse $addr$ aus dem globalen Feldindex $F$ durch die Software berechnet und der *Index* aus diesen beiden Werten, wie in Abbildung 2.1 von Seite 2.1 gezeigt, aufgebaut werden:

$$nr = \left\lfloor \frac{F \mod P * k}{k} \right\rfloor$$

$$addr = F \mod k + \left\lfloor \frac{F}{P * k} \right\rfloor * k$$

In 5.5.2 sind die Vorteile der Adreßumsetzung durch die Hardware-Zentrifuge an einem Testprogramm aufgeführt.

## 4.4 Zusammenfassung

Dieses Kapitel stellte in 4.1 Transformationsmuster vor, mit denen abhängig von der Datenverteilung und des Kommunikationsmusters datenparallele Zuweisungen auf Fließbänder des VSCAP-Verfahrens abgebildet werden können. Dabei wurden für block und zyklisch verteilte Datenfelder speziell optimierte Transformationen (Ein- und Mehrblock) angegeben, da diese Verteilungen in der Praxis häufig verwendet werden. Generelle block-zyklische Verteilungen

können aufgrund ihrer statisch nicht berechenbaren Prozessorzuordnung teilweise nur mit einer (1,1)-Vektorstrategie übersetzt werden.
Die Fließbänder werden direkt in den späteren ausführbaren Programmtext geschrieben. Dies ermöglicht weitere Optimierungen durch eventuelle zusätzliche Übersetzerläufe.
Der zweite große Abschnitt 4.2 beschäftigte sich mit der Architektur der Cray T3E und ihren E-Registern, die durch die Aufspaltung einer entfernten Leseoperation in Vorlade- und Zugriffsanweisungen als Vorladepuffer für das VSCAP-Verfahren benützt werden können.
Der letzte Abschnitt 4.3 stellte den im Rahmen dieser Arbeit entwickelten Prototypübersetzer KarHPFn vor. KarHPFn akzeptiert HPF-Programme und generiert die im ersten Abschnitt dieses Kapitels vorgestellten Kommunikationsfließbänder. KarHPFn ist somit das erste uns bekannte System, das Programme mit automatisch generierten Vorladeoperationen für Architekturen mit gemeinsamem Adreßraum erstellt. Die Übersetzung läuft dabei für den Programmierer vollkommen transparent, der somit bequem in seinem von HPF bereitgestellten datenparallelen Programmiermodell arbeiten kann.

# Kapitel 5

# Validation

In diesem Kapitel werden die Resultate von Laufzeittests verschiedener Programme vorgestellt, mit denen die in Abschnitt 1.3 aufgestellten Thesen quantitativ bestätigt werden.
Abschnitt 5.1 stellt die untersuchten Testprogramme vor und unterteilt sie in verschiedene Algorithmenklassen mit unterschiedlichen Kommunikationsmustern und unterschiedlichem Verhältnis von Kommunikations- zu Berechnungsaufwand. Dadurch lassen sich Aussagen über das Laufzeitverhalten eines Vertreter auf alle weiteren Programme dieser Klasse übertragen. Danach vergleicht 5.2 die in Kapitel 3 gewonnen Laufzeitaussagen von VSCAP mit den auf der Cray T3E gemessenen Werten. Dies wird für die $(L,L)$- und die $(1,L)$-Vektorstrategie durchgeführt. In 5.3 wird der Kommunikationsvorteil von VSCAP anhand der einzelnen Algorithmenklassen gezeigt. Abschnitt 5.4 zeigt die Laufzeittest dreier Applikationen, die auf bis zu 128 Prozessoren ausgeführt wurden. Zum Schluß werden in 5.5 T3E spezifische Tests wie der Einsatz eines Softwaretests oder der Hardware-Zentrifuge gezeigt.

## 5.1 Auswahl der Testprogramme

Dieser Abschnitt stellt die in dieser Arbeit getesteten Programme vor. Um eine differenzierte Aussage über die automatisch generierten VSCAP-Programme zu erhalten, wurden als Testprogramme die parallelisierten Versionen der *Livermore Loops* (LL) [89], einige zusätzliche Basisalgorithmen (Jacobi-Iteration, Laplace-PDE, Indirekt, PDE1) und zweier Applikationen (Veltran-Operator und FIRE) herangezogen. Die Livermore-Kerne wurden dabei untersucht, damit ein Vergleich mit den Ergebnissen von Warschko möglich ist.
Die Livermore-Kerne wurden in [120, 61, 40, 44] parallelisiert, so daß sie teilweise nur noch von C nach HPF übersetzt werden mußten. Bis auf LL17 und LL20, die nicht auflösbare Abhängigkeiten zwischen einzelnen Schleifeniterationen haben, lagen alle Livermore-Kerne in parallelisierte Form vor. LL16 wird ebenfalls nicht betrachtet, da sich durch das Monte-Carlo-Suchverfahren keine allgemeingültigen Aussagen ableiten lassen. Bis auf LL13 und LL14 wurden alle Programme von KarHPFn ohne weitere Hilfestellung übersetzt. Die Reduktionen in LL13 und LL14 sind die von KarHPFn nicht unterstützten SUM_SCATTER Funktionen von HPF. Diese Reduktionen wurden von Hand in den übersetzten Programmtext eingefügt, wobei auf eine möglichst getreue Nachbildung einer automatischen Übersetzung geachtet wurde.
Die folgende Charaketerisiserung der Testprogramme und ihre Klassifikation richtet sich nach Warschkos Arbeit.

### 5.1.1 Charakterisierung der Testprogramme

Die Charakterisierung der Testprogramme erfolgt nach zwei unterschiedlichen Kriterien: zum einen Aufgrund des Verhältnisses zwischen Berechnungs- und Kommunikationsaufwand, das drei Einteilungen liefert

$$T_{\text{Berechnung}} > T_{\text{Kommunikation}}$$
$$T_{\text{Berechnung}} \sim T_{\text{Kommunikation}}$$
$$T_{\text{Berechnung}} < T_{\text{Kommunikation}}$$

und zum anderen bezüglich des Kommunikationsverhaltens, wobei sich sechs unterschiedliche Basisvarianten ausgeprägt haben, siehe Tabelle 5.1.

Tabelle 5.1: Charakterisierung der Beispielprogramme

| Beispielprogramm | | Kommunikations-verhalten | Kommunika-tionsaufwand pro Prozessor | Berechnungs-aufwand pro Prozessor |
|---|---|---|---|---|
| LL1 | hydro fragment | indiziertes Feld | $min(11, \frac{N}{P})$ | $\frac{N}{P}$ |
| LL2 | ICCG excerpt | Reduktion | $log(N) * \frac{N}{P}$ | $log(N) * \frac{N}{P})$ |
| LL3 | inner product | Reduktion | $log(P)$ | $\frac{N}{P} + log(P)$ |
| LL4 | banded linear equation | Reduktion | $2 * log(P)$ | $2 * (\frac{N}{P} + log(P))$ |
| LL5 | tri-diagonal elimination | Reduktion | $2 * log(N) * \frac{N}{P}$ | $2 * \frac{N}{P} + 4 * log(N) * \frac{N}{P}$ |
| LL6 | general linear recurrence equation | Verteiloperation, indiziertes Feld | $N * (1 + \frac{N}{P})$ | $\frac{N^2}{P}$ |
| LL7 | equation of state fragment | indiziertes Feld | $min(6, \frac{N}{P})$ | $\frac{N}{P}$ |
| LL8 | ADI integration | indiziertes Feld | 6 | $\frac{N}{P}$ |
| LL9 | integrate predictors | | 0 | $\frac{N}{P}$ |
| LL10 | difference predictors | | 0 | $\frac{N}{P}$ |
| LL11 | first sum | Reduktion | $log(P)$ | $\frac{N}{P} + log(P)$ |
| LL12 | first difference | indiziertes Feld | 1 | $\frac{N}{P}$ |
| LL13* | 2D particle in cell | indirekt indiziertes Feld, Reduktion | $min(\frac{N}{P}, 64^2)$ | $\frac{N}{P}$ |

*weiter auf der nächsten Seite*

*fortgesetzt von letzter Seite*

| Beispielprogramm | | Kommunikations-verhalten | Kommunika-tionsaufwand pro Prozessor | Berechnungs-aufwand pro Prozessor |
|---|---|---|---|---|
| LL14$^\star$ | 1D particle in cell | indirekt indiziertes Feld, Reduktion | $2 * \frac{N}{P} * (1 + P)$ | $12 * \frac{N}{P} + 2 * N$ |
| LL15 | casual Fortran | indiziertes Feld | 3 | $12 * \frac{N}{P}$ |
| LL18 | 2D explicit hydrodynamics fragment | indiziertes Feld | 5 | $8 * \frac{N}{P}$ |
| LL19 | general linear recurrence equations | Reduktion | $3 * log(N) * \frac{N}{P}$ | $\frac{N}{P} * (4 + 7 * log(N))$ |
| LL21 | matrix-matrix produkt | Verteiloperation | $\frac{N}{P} * \sqrt{P}$ | $(\frac{N}{P})^3 * \sqrt{P}$ |
| LL22 | Planckian distribution | | 0 | $\frac{N}{P}$ |
| LL23 | 2D implicit hydrodynamics fragment | indiziertes Feld, Reduktion | $2 * log(N) * \frac{N}{P}$ | $10 * \frac{N}{P} * (2 + log(N))$ |
| LL24 | first minimum | Reduktion | $log(P)$ | $\frac{N}{P} + log(P)$ |
| | Jacobi-Iteration | 2D-Gitter Nachbar-Nachbar | $4 * \sqrt{\frac{N}{P}}$ | $2 * \frac{N}{P}$ |
| | Laplace PDE | 2D-Gitter Nachbar-Nachbar, Reduktion | $4 * \sqrt{\frac{N}{P}} + log(P)$ | $2 * \frac{N}{P}$ |
| | Rotieren | indiziertes Feld | $\frac{N}{P}$ | $\frac{N}{P}$ |
| | Indirekt | indirekt indiziertes Feld | $\frac{N}{P}$ | $\frac{N}{P}$ |
| | PDE1 | 3D-Gitter Nachbar-Nachbar | $6 * \sqrt[3]{\frac{N}{P}}$ | $4 * \frac{N}{P}$ |
| | Fire | indirekt indiziertes Feld, Reduktion | $\frac{N}{P}$ | $\frac{N}{P}$ |
| | Veltran-Operator | Verteiloperation | $\frac{N}{P}$ | $7 * \frac{N}{P}$ |

$\star$ Die Reduktion ist eine SUM_SCATTER Funktion in HPF, die KarHPFn nicht unterstützt. Sie wurde nachträglich in den Programmtext eingefügt, wobei auf die möglichst genaue Nachbildung einer automatischen Übersetzung geachtet wurde.

**Keine Kommunikation:** Sogenannte trivialparallele Programme, die vollkommen ohne Kommunikationsoperationen auskommen.

**Reduktionen:** Hierzu zählen Programme, die hauptsächlich auf einem Reduktionsbaum arbeiten, und Vektorsummen, Minima bzw. Maxima eines Vektors und die Prä- bzw. Postfixsummen berechnen. Während für blockierende Netzwerke ein binärer Reduktionsbaum die besten Laufzeiten erwarten läßt, wird im Rahmen des VSCAP-Verfahrens die in Abschnitt 4.1.2.3 angedeutete Vergrößerung des Verzweigungsgrades eingesetzt.

**Indizierte Feldzugriffe:** Die Kommunikationsoperationen der Programme lassen sich auf Zugriffe unterschiedlicher Feldelemente (z.B. $\mathcal{K}(i) = i + b$) im Umfeld der Datenverteilung zurückführen. Nach der Erkennung nicht-lokaler Datenelemente zur Übersetzungszeit können diese mittels VSCAP vorgeladen werden. Dabei treten bei blockweiser Verteilung der Felder höchsten $b$ nicht-lokale Feldzugriffe auf.

**Indirekt indizierte Feldzugriffe:** Hierbei ergeben sich die Kommunikationsoperationen aus Feldzugriffen der Art $\mathcal{K}(i) = g(i)$, wobei es sich bei $g(i)$ um eine a priori unbekannte oder datenabhängige Permutationsfunktion handelt. VSCAP verwendet bei diesem dynamischen Kommunikationsmuster eine $(1, L)$-Vektorstrategie, bei denen mit einer Wahrscheinlichkeit von $\frac{P-1}{P}$ $g(i)$ nicht-lokale Datenelemente adressiert werden.

**nD-Gitter, Nachbar-Nachbar-Kommunikation:** Diese Programme sind durch indizierte Feldzugriffe auf n-dimensionale Felder charakterisiert. Ein Spezialfall dieses Kommunikationsmusters ist die 2-dimensionale nächste Nachbarschaftskommunikation, bei dem das typische NEWS-Kommunikationsmuster (Nord, Ost, West und Süd) auftritt. Bei einer blockweisen Verteilung aller Dimensionen sind lediglich die Ränder des $\frac{N}{P} \times \frac{N}{P}$ großen lokalen Rechtecks in Kommunikationsoperationen involviert. Diese Oberflächenelemente werden vor der Berechnung durch VSCAP in Überlappungsbereiche (siehe Abschnitt 4.1.2.1) kopiert.

**Verteiloperationen:** Dieses Kommunikationsmuster ist in Architekturen mit zweiseitigen Kommunikationsprimitiven durch einen Verteilbaum charakterisiert, mit dem Datenelemente an alle Prozessoren verteilt werden. Durch die einseitige Kommunikation im VSCAP-Verfahren ist diese Technik jedoch nicht nötig, da jeder Prozessor die benötigten Daten selbständig anfordert. Voraussetzung hierfür ist, daß die Parallelrechnerarchitektur mehrere Leseanforderungen an die gleiche Speicherstelle unterstützt (CR-Speicher). Damit vergrößert sich zwar der gesamte Kommunikationsaufwand von $O(log(P))$ auf $(O(P))$, im Gegenzug verringert sich jedoch der Aufwand des einzelnen Prozessors auf $O(1)$ Kommunikationsschritte. Dies resultiert in einer Ausführung der Verteilung in $O(1)$, anstatt wie bisher in $O(log(P))$. Voraussetzung ist allerdings, daß die Parallelrechnerarchitektur mehrere Leseanforderungen an dieselbe Speicherstelle zuläßt, wie es zum Beispiel bei der Cray T3E der Fall ist, siehe 4.2.3.

In den meisten der untersuchten Testprogramme aus Tabelle 5.1 treten die Kommunikationsmuster singulär auf. Bei mehreren Kommunikationsmustern (LL6, LL13, LL14, LL23, Laplace-PDE und FIRE) zeigt der Berechnungsaufwand, welches Kommunikationsmuster überwiegt. So dominiert die Reduktionsoperation die Testprogramme LL13, LL14 und LL23, während dies bei LL6 und Laplace-PDE die indizierten und bei FIRE die indirekt indizierten Zugriffe sind.

### 5.1.2 Klassifikation der Testprogramme

Die Charakterisierung der Testprogramme aus Tabelle 5.1 anhand der auftretenden Kommunikationsmuster sowie dem Verhältnis zwischen Berechnungs- und Kommunikationsaufwand wird im folgenden genutzt, um die Programme in Applikationsklassen einzuordnen. Das geschieht mit dem Ziel, ein ähnliches Laufzeitverhalten der Testprogramme innerhalb einer Applikationsklasse nachzuweisen. Gelingt dies, so kann im Gegenzug die Charakterisierung einer Applikation zur Vorhersage des Laufzeitverhaltens herangezogen werden.
Aus der Kombination der Kommunikationsmuster (6 Varianten) sowie dem Verhältnis zwischen Kommunikations- und Berechnungsaufwand (3 Varianten) ergeben sich 18 mögliche Applikationsklassen. Die Unterscheidung zwischen verschiedenen Aufwandsklassen bei trivialparallelen Programmen (keine Kommunikation) ist unerheblich, so daß sich die Zahl auf 16 realistische Applikationsklassen reduziert. Die Klassenbezeichnung basiert primär auf dem Verhältnis zwischen Kommunikations- und Berechnungsaufwand, welches die Basisklasse $Kx$ festlegt, wobei trivialparallele Programme eine Sonderstellung mit eigener Basisklasse $K0$ einnehmen.

$$Kx : \begin{cases} K0, & \text{falls } T_{\text{Kommunikation}} \equiv 0 \\ K1, & \text{falls } T_{\text{Kommunikation}} < T_{\text{Berechnung}} \\ K2, & \text{falls } T_{\text{Kommunikation}} \sim T_{\text{Berechnung}} \end{cases}$$

Auf die Basisklasse $K3$ ($T_{\text{Kommunikation}} > T_{\text{Berechnung}}$) wurde verzichtet, da Prozessoren Datenelemente nur dann austauschen, wenn sie diese auch benötigen. Deshalb ist anzunehmen, daß die Basisklasse $K2$ eine realistische Abschätzung für kommunikationsintensive Applikationen liefert. Damit reduziert sich die Anzahl der verschiedenen Applikationsklassen auf 11. Zusätzlich zu den Basisklassen wird jedes Programm gemäß dem dominierenden Kommunikationsmuster in eine Unterklasse eingeordnet.

$$Kxy : \begin{cases} Kxa, & \text{Reduktion} \\ Kxb, & \text{Indizierte Felder} \\ Kxc, & \text{Indirekt indizierte Felder} \\ Kxd, & \text{n-dimensionale Nachbarschaftskommunikation} \\ Kxe, & \text{Verteiloperation} \end{cases}$$

Wie Tabelle 5.2 zeigt, decken die ausgewählten Testprogramme neun der insgesamt 11 möglichen Klassen ab (die Klasse $K0$ der trivialparallelen Programme wird nicht weiter betrachtet) und repräsentieren ein breites Spektrum paralleler Applikationen. Die Klassen $K1c$ und $K2d$ werden nicht betrachtet, denn für die Klasse $K1c$ konnte kein passendes Program gefunden werden und die Klasse $K2d$ ist natürlicherweise leer, da Programme mit nächster Nachbarschaftskommunikation immer einen im Vergleich zur Kommunikation großen Berechnungsaufwand besitzen.
Mit der Einordnung eines Testprogramms in eine bestimmte Klasse wird implizit eine Annahme über das Laufzeitverhalten getroffen. Bei der folgenden Erklärung wird der Begriff der *PRAM-Effizienz* verwendet, der wie folgt definiert ist

Tabelle 5.2: Einteilung der Testprogramme in Klassen

| Klasse | Aufwandsverhältnis | Kommunikationsmuster | Testprogramme |
|---|---|---|---|
| K0 | $T_{\text{Kommunikation}} \equiv 0$ | Keine Kommunikation | LL9, LL10, LL22 |
| K1a | $T_{\text{Kommunikation}} < T_{\text{Berechnung}}$ | Reduktion | LL3, LL4, LL24 |
| K1b | | Indizierte Felder | LL1, LL7, LL8, LL12, LL15, LL18 |
| K1d | | nD-Gitter | Jacobi, Laplace-PDE, PDE1 |
| K1e | | Verteiloperation | LL21 |
| K2a | $T_{\text{Kommunikation}} \sim T_{\text{Berechnung}}$ | Reduktion | LL2, LL5, LL11, LL13, LL14, LL19, LL23, |
| K2b | | Indizierte Felder | LL6, Rotieren |
| K2c | | Indirekt indizierte Felder | Indirekt, FIRE |
| K2e | | Verteiloperation | Veltran |

$$T_{\text{PRAM}} = T_{\text{Laufzeit ohne Kommunikation}} \tag{5.1}$$

$$T_{\text{PRAM-Effizienz}} = \frac{T_{\text{PRAM}}}{T_{\text{Laufzeit mit Kommunikation}}} \tag{5.2}$$

**Klasse K1a:** Der Kommunikationsaufwand einer Reduktionsoperation in dieser Algorithmenklasse ist abhängig von der Anzahl $P$ der parallel eingesetzten Prozessoren und kann mit $O(log(P))$ Kommunikationschritten ausgeführt werden. Für wachsende Problemgrößen stehen dem konstanten Kommunikationsaufwand bei gleichviel eingesetzten Prozessoren ein wachsender Berechnungsaufwand gegenüber, so daß bei großen Virtualisierungen die Kommunikationszeit keinen großen Einfluß auf die Gesamtlaufzeit der Applikation hat. Dies führt zu einer PRAM-Effizienz von über 90%, wenn der lokale Berechnungsanteil groß genug gewählt wird.

**Klasse K1b:** Der Kommunikationsaufwand von Programmen mit indizierten Feldzugriffen ist in dieser Klasse nach oben durch eine Konstante beschränkt, so daß sich nur bei kleinen Problemgrößen Vorteile aus der Latenzzeitreduktion ergeben. Mit wachsender Virtualisierung verringert sich der Anteil der Kommunikation an der Gesamtlaufzeit, so daß auch hier, wie bei Algorithmenklasse $K1a$, mit PRAM-Effizienzen von 90% und mehr zu rechnen sind. VSCAP kann in dieser Klasse den Kommunikationsaufwand durch die Vektorbefehle nicht weiter reduzieren, da die Anzahl der entfernten Datenzugriffe nicht ausreicht, um die Latenzzeit auf das erste Datenelement vollständig zu überbrücken.

**Klasse K1d:** Bei Programmen mit $n$-dimensionaler Nachbarschaftskommunikation steigen die Kommunikationskosten mit der $(n-1)$-ten Wurzel der Größe des lokal zu bearbeitenden Datenbereichs, da nur die Ränder benachbarter Bereiche kopiert werden müssen. Der Vorteil von VSCAP gegenüber SCAP ist auf kleine Virtualisierungen beschränkt, da mit steigender Problemgröße der Berechnungsaufwand in der $n$-ten Potenz wächst und somit bei großen Problemen der reduzierte Kommunikationsaufwand durch die Vektorbefehle nicht mehr ins Gewicht fällt.

**Klasse K1e:** Programme dieser Algorithmenklasse enthalten eine Verteiloperation. Da jeder Prozessor die Daten autonom zu sich in den lokalen Speicher kopieren kann, bedeutet dies einen Kommunikationsaufwand von $O(1)$ für jeden nicht-lokalen Datenzugriff. In dieser Klasse überwiegt der Anteil der Berechnung, so daß für große Virtualisierungen der Anteil der Kommunikation nicht mehr ins Gewicht fällt und der Vorteil von VSCAP durch die Vektoroperationen nicht mehr meßbar wird.

**Klasse K2a:** Bei Programmen aus dieser Klasse lassen sich die Reduktionsoperationen aufgrund algorithmischer Beschränkungen nicht über $N/P$ lokale Vorberechnungen sowie eine Reduktionsoperation über $P$ Prozessoren abhandeln, sondern die Reduktion muß für alle $N/P$ lokalen Elemente durchgeführt werden. Eine Optimierung der Stufe $i$ der Reduktionsoperation bietet sich insofern an, als bei einer blockweisen Verteilung der Daten lediglich $i$ statt $N/P$ ($i < N/P$) Kommunikationsoperationen benötigt werden. Bei großen Datensätzen kommt diese Optimierung allerdings nur in den ersten $log(N/P)$ der insgesamt $log(N)$ Stufen zum Tragen. In allen anderen Stufen werden $N/P$ Kommunikationsoperationen benötigt. Das VSCAP-Verfahren nutzt die große Anzahl an Kommunikationsoperationen zur Latenzzeitverbergung, und es wird erwartet, daß mit den Vektorbefehlen ein weiterer Laufzeitgewinn gegenüber SCAP erreicht werden kann. Die mit der Problemgröße wachsende Anzahl an Kommunikationsoperationen verhindert jedoch große PRAM-Effizienzen, da die Ausführung des Datenfließbandes einen relativ großen Anteil an der Gesamtlaufzeit ausmachen wird.

**Klasse K2b:** Bei dieser Algorithmenklasse ist das Kommunikationsmuster durch indirekte Zugriffe charakterisiert. Die große an Anzahl nicht-lokalen Datenzugriffe resultiert entweder aus der Wahl der Datenverteilung oder sie hängt von einer Größe ab, die erst zur Laufzeit bekannt ist und die sich innerhalb des Programmablaufs ändern kann. VSCAP kann durch den großen Kommunikationsanteil dieser Programme einen großen Nutzen ziehen und im Vergleich zu SCAP die Kommunikationszeit deutlich reduzieren. Dieser Vorteil wird sich auch bei großen Virtualisierungen bemerkbar machen, da der Kommunikations- mit dem Berechnungsaufwand wächst.

**Klasse K2c:** Die Quelladressen indirekt indizierter Feldzugriffe müssen zur Laufzeit berechnet werden, da sie statisch nicht bestimmt werden können. Daher wird bei VSCAP eine (1,L)-Vektorstrategie zum Kopieren der Daten eingesetzt. Für jedes Element wird der Prozessor und die knotenlokale Adresse berechnet und durch Vorladebefehle angefordert. Anstatt spekulativ vorzuladen kann auch ein Softwaretest eingesetzt werden, mit dem die Lokalität der Daten zuerst geprüft wird, so daß wirklich nur nicht-lokale Datenelemente vorgeladen werden. Die Untersuchung in wieweit sich die (1,L)-Vektorstrategie und der Softwaretest zur Laufzeitreduktion eignen zeigt 5.5.1. Prinzipiell verhindern die vielen nicht-lokalen Datenzugriffe eine effiziente PRAM-Auslastung, jedoch kann mit einem Laufzeitgewinn von VSCAP gegenüber SCAP durch den Einsatz der Vektorbefehle beim Zurückschreiben der Datenelemente gerechnet werden. Die *shared-memory*-Versionen der getesteten Programme werden ähnliche Laufzeiten wie BLOCK aufweisen, da dynamische Kommunikationsmuster durch die *shared-memory*-Bibliothek nicht unterstützt werden. Somit kann keine Überlappung einzelner Kommunikationsaufrufe stattfinden, da jedes Element durch einen eigenen Aufruf von `shmem_get` kopiert werden muß.

**Klasse K2e:** Die Kommunikationszeiten der Pogramme aus dieser Algorithmenklasse werden durch eine Verteiloperation dominiert, deren Aufwand mit dem Berechnungsaufwand gleichzusetzen ist. Daher wird sich der Vorteil der Vektoroperationen gegenüber SCAP nicht nur auf kleine Virtualisierungen beschränken, sondern der Vorteil wird sich auch auf große Probleme durchschlagen. Durch den hohen Kommunikationsaufwand werden sich die für die Basisklasse $K2$ typisch niedrigen PRAM-Effizienzen ergeben.

Die Betrachtung dieser Algorithmenklasse wird bei veränderten Problemgrößen und einer konstanten Anzahl an parallel eingesetzten Prozessoren keine neuen Einblicke liefern, weshalb sich die Analyse des einzigen Vertreters, des Veltran-Operators, auf eine konstante Problemgröße bei variierender Prozessorzahl in 5.4.3 beschränkt.

## 5.2 Validierung analytisches Modell

Der vorliegende Abschnitt befaßt sich mit der Bestätigung der Laufzeitanalyse von 3.2.2 und 3.2.3. Hierfür werden die Abschätzungen der (1,8)- und (8,8)-Vektorstrategien mit den auf der T3E gemessenen Laufzeiten von KARHPFN übersetzten Programmen verglichen. Weiterhin werden die Laufzeitdifferenzen der Vorladestrategien BLOCK und SCAP mit der (128,8)-Vorladestrategie von VSCAP betrachtet. Der erste Vergleich zeigt die Latenzzeitreduktion innerhalb des VSCAP-Verfahrens, während der zweite Vergleich die Zeiteinsparungen der Vektorbefehle verdeutlicht.
Die Validierung wird für jede Vektorstrategie an jeweils einem Testprogramm vorgenommen. Das ist für die (8,8)-Vektorstratgie *Rotieren* und für die (1,8)-Vektorstrategie *Indirekt*. Für beide Programme wurden jeweils die applikationsspezifischen Parameter $t_{\mathrm{v}}$, $t_{\mathrm{z}}$, $t_{\mathrm{v}}^{L}$ und $t_{\mathrm{z}}^{L}$ gemessen. Anhand dieser Parameter wurden die entsprechenden Modellklassen von 3.2.2 selektiert, mit denen die Abschätzungen vorgenommen wurden.
In Abschnitt 5.2.1 wird die Vorgehensweise bei den Tests beschrieben. Die beiden verbleibenden Abschnitte zeigen den Vergleich vom Modell zur T3E für die (8,8)- und (1,8)-Vektorstrategie.

### 5.2.1 Beschreibung der Testumgebung

Getestet wurde auf einer Cray T3E-900, wie sie in Abschnitt 4.2.2 beschrieben wurde. Gemessen wurde mit der Vektorlänge $L = 8$ und einem Netzwerktakt von $13.3ns$.
Es wurden 256 E-Register mit einer Vorladepuffergröße von $C_V = 128$ (Nachladedistanz ist 128 Elemente, siehe Abbildung 4.9) eingesetzt. Der Schleifenaufwand wurde mit $t_{\mathrm{s}} = 44ns$ gemessen. Für die Latenzzeit wurden zwei verschiedene Werte verwendet. Für die Vektorstrategien wurde $T_{\mathrm{Latenz}} = 1480ns$ als Wert für die Latenzzeit verwendet. Für BLOCK wurde dieser Wert um $400ns$ ($T_{\mathrm{Latenz}} = 1880$) erhöht, da sich nach jedem Datenzugriff eine Wartezeit für das Rückschreiben des E-Registerwertes in den lokalen Speicher ergab.
Die verbleibenden Parameter sind applikationsabhängig und werden bei den entsprechenden Tests aufgeführt. Als Uhr wurde die *RTC* (*real time clock*) der T3E eingesetzt. Ein Problem bei der Messung der Applikationsparameter waren die unterschiedlichen Zeiten für verschiedene Iterationen der einzelnen Schleifen. Um stabile Werte zu bekommen, wurden die Programme zum einen mit ausgeschalteten Optimierungen `-g` übersetzt, zum anderen sind die angegebenen Zeiten der Applikationsparameter gemittelte Werte über die Anzahl der ausgeführten Iterationen der Vorlade- bzw. Zugriffsschleifen. Die beiden Schleifen wurden für $t_{\mathrm{v}}$

und $t_z$ 128-mal und für $t_v^L$ und $t_z^L$ 16-mal iteriert. Die Anzahl der Iterationen entspricht einem vollständigen Füllen des Vorladepuffers. Für jeden Test wurden 2 Prozessoren eingesetzt, da jeder Prozessor für sich lokal das VSCAP-Verfahren anwendet, und mehr Prozessoren sich nur in einer Vergrößerung der Latenzzzeit des Netzwerkes widerspiegeln würden. Für die Messungen wurden die Virtualisierungen $V$ variiert ($8 \leq V \leq 4096$), denn es sollten die Modellabschätzungen in Abhängigkeit der nicht-lokalen Datenzugriffe gezeigt werden.
Die Präsentation der Testergebnisse ist in einen Vergleich von BLOCK gegenüber VSCAP und von SCAP gegenüber VSCAP aufgeteilt. Der erste Vergleich zeigt die Vorteile von VSCAP gegenüber einem auf der T3E simulierten blockierenden Netzwerk (Vorladedistanz $d = 1$), während der zweite Vergleich die Vorteile der Vektorbefehle von VSCAP gegenüber der einelementigen Befehle von SCAP demonstriert. Jeder Vergleich wird durch zwei Grafiken präsentiert, die auf der x-Achse die Virtualisierung und auf der y-Achse die Zeiten in *ms* auftragen. Die linke Grafik zeigt die Laufzeiten des jeweiligen Testprogramms und die des Modells, während die rechte Grafik die Differenzen der Laufzeiten zwischen BLOCK und VSCAP oder SCAP und VSCAP zeigt. Alle Testversionen wurden durch KarHPFn übersetzt. Die Vorladestrategie wurde durch Kommandozeilenoptionen gewählt.
Im VSCAP-Modell wurden kleine Modifikationen dahingehend vorgenommen, daß verschiedene Zeiten für die VSCAP und SCAP Vorlade- und Zugriffsoperationen zugelassen wurden. Das Modell ging bisher von identischen Zeiten für gleiche Adreßberechnungen aus. So wurden für die Berechnung der Differenz von BLOCK und SCAP zu VSCAP nicht die Modellabschätzungen von 3.2.3 eingesetzt. Es wurden vielmehr die mit den unterschiedlichen Parametern berechneten Laufzeiten voneinander abgezogen. Die Grafiken der Laufzeitgewinne von VSCAP zeigen die so erhaltenen Werte.
Getestet wurden die ($L$,$L$)- und die (1,$L$)-Vektorstrategien mit den Programmen *Rotieren* bzw. *Indirekt*. Die Abschätzungen beinhalten auch die (1,1)-Vektorstrategie des SCAP-Verfahrens, so daß sie nicht separat betrachtet werden mußte.

### 5.2.2 ($L$,$L$)-Vektorstrategie

Die ($L$,$L$)-Vektorstrategie wird bei Kommunikationsmustern eingesetzt, bei denen die äquidistanten nicht-lokalen Datenzugriffe zur Übersetzungszeit bestimmt werden können. Das zugehörige Einblock-Fließband (siehe Abschnitt 4.1.2.1) kopiert einen Datenblock von *einem* entfernen Prozessorknoten in den lokalen Speicher. Sie berechnet einmal vor der Ausführung des Datenfließbandes die Nummer des entfernten Prozessors und die entfernten bzw. lokalen Schrittweiten. Durch Vektorbefehle können so die Daten zeitsparend vorgeladen und in den Speicher zurückgeschrieben werden.
Für die Messungen auf der T3E wurde das Programm *Rotieren* ausgesucht. Es implementiert ein zyklisches Rotieren der Elemente eines Vektors:

```
t := A[0];
FORALL i = 0 TO N-2 DO
  A[i] := A[i+1];
END FORALL
A[N-1] := t;
```

Um den Anteil der Kommunikation zu erhöhen, wurden die Vektoren CYCLIC(1)-Verteilt. Damit resultiert jeder indizierte Feldzugriff `A[i+1]` in einer Kommunikationsoperation. Durch

die CYCLIC(1)-Verteilung werden alle entfernten Datenelemente vom selben Prozessor geladen. Dazu wird die Einblock-Transformation eingesetzt.
Die zugehörigen Modellparameter zeigt Tabelle 5.3.

Tabelle 5.3: Modellparameter für Kopieren

| **Parameter** | **Wert** |
|---|---|
| $t_{\mathrm{v}}$ | 148 ns |
| $t_{\mathrm{v}}^{L}$ | 146 ns |
| $t_{\mathrm{z}}$ | 148 ns |
| $t_{\mathrm{z}}^{L}$ | 144 ns |

Sie beinhalten die Zeit für eine Iteration der Vorladeschleife oder der Zugriffsschleife innerhalb des Datenfließbandes. Daraus ergibt sich die Ungleichung

$$8 * t_{\mathrm{n}} = 106.4ns < t_{\mathrm{v}}^{L} = 146ns < 8 * t_{\mathrm{v}} = 1184ns.$$

Damit wird die T3E in die Fälle 1 bis 3 des VSCAP-Modells von 3.2.2.1 eingeordnet (siehe auch Tabelle 3.4). Nach Fall 2 ($K \leq 128$ und $\frac{K}{L} * t_{\mathrm{v}}^{L} \geq 1571$), bei dem der Vorladepuffer nicht ganz gefüllt wird und es keine Wartezeit auf das erste nicht-lokale Element gibt, kann ab 88 nicht-lokalen Datenzugriffen mit einer 8-fachen Reduzierung der Kommunikationszeit beim Wechsel von SCAP zu VSCAP gerechnet werden. Denn bei 88 nicht-lokalen Datenzugriffen überdeckt VSCAP mit den Vektorbefehlen die gesamte Latenzzeit ($\lceil \frac{T_{\mathrm{Latenz}} + (L-1) * t_{\mathrm{n}}}{t_{\mathrm{v}}^{L}} \rceil * 8 = 88$).
Weitere Wartezeiten entstehen nicht, da $8 * t_{\mathrm{n}} < t_{\mathrm{v}}^{L}$. Der Kommunikationsaufwand von SCAP wird damit um den Faktor $L = 8$ der Vektorlänge reduziert.
Abbildung 5.1 zeigt auf der linken Seite die Laufzeiten von BLOCK und VSCAP, gewonnen durch Laufzeittests und durch die Modellabschätzungen.

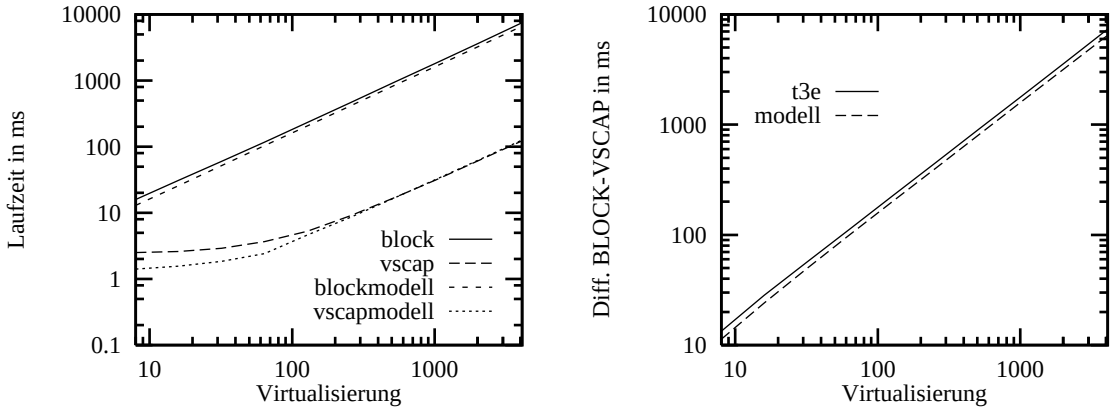



Abbildung 5.1: Vergleich der Laufzeiten von VSCAP und BLOCK

Die Grafik auf der linken Seite zeigt den linearen Anstieg der Ausführungszeit von BLOCK in Abhängigkeit von der Virtualisierung $K$, denn bei jedem nicht-lokalen Datenzugriff wird die Netzwerklatenz zur Ausführungszeit hinzuaddiert. Die Laufzeitabschätzungen von VSCAP

zeigen eine kleine Differenz für kleine Virtualisierungen, eine sogenannte *Warmlaufphase*. Diese *Warmlaufphase* des VSCAP-Verfahrens, das sich in schwankenden Zeiten für die Vorlade- und die Zugriffsopeationen widerspiegelt, kann durch die fest gewählte Vorladezeit nicht wiedergegeben werden, so daß sich diese Abweichung von Modell und Maschine einstellt. Erst wenn genügend viele nicht-lokale Datenzugriffe vorliegen ($K \geq 100$ oder $\frac{K}{L} \geq 13$ Vektorzugriffe) schließt sich die Lücke zwischen Modell und Maschine.

Der Vergleich der Laufzeitgewinne beim Wechsel von BLOCK zu VSCAP auf der rechten Seite von Abbildung 5.1 zeigen den beobachteten Warmlaufeffekt des VSCAP-Verfahrens nicht, denn mit einer Reduktion der Kommunikationszeit um 90% schon für kleine Datenzugriffe ($K = 8$), fällt diese Schwankung nicht ins Gewicht. Die Latenzzeitverbergung von VSCAP erhöht sich bis auf 96% für 4096 nicht-lokale Datenzugriffe und ist im Bereich des theoretischen Maximums von über 99%.

Der Vorteil der Vektorbefehle von VSCAP gegenüber dem SCAP-Verfahren zeigt Abbildung 5.2.

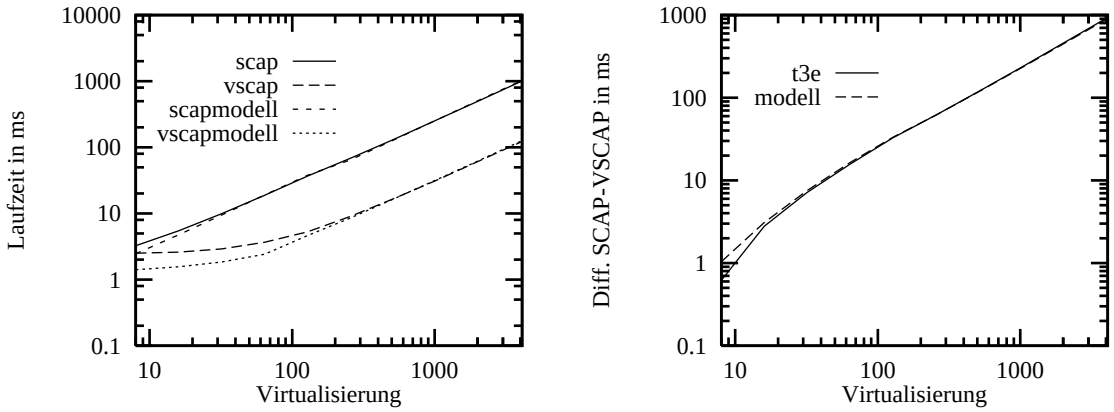



Abbildung 5.2: Vergleich der Laufzeiten von VSCAP und SCAP

Die Grafik auf der linken Seite zeigt für VSCAP diesselbe Laufzeitkurve wie für den vorherigen Vergleich mit BLOCK. Die SCAP Laufzeiten zeigen einen ähnlichen Warmlaufeffekt wie beim VSCAP-Verfahren, doch ist er hier nicht so stark ausgeprägt, da sich das Fließband bei $K = 8$ Zugriffen schon stärker gefüllt hat.

Die rechte Grafik zeigt die genauen Modellabschätzungen für die Differenz von SCAP und VSCAP. Das Modell berechnet ab 88 Datenelementen eine 8-fache Laufzeitreduktion. Diese Reduktion wird durch die T3E für größere Virtualisierungen erreicht, siehe Tabelle 5.4.

Die mittlere Spalte zeigt die vom Modell berechneten Beschleunigungen. Nach 3.2.3.2 müßte $L = 8$ die maximale Beschleunigung von VSCAP gegenüber SCAP sein. Sie wird übertroffen, denn im Modell wurden identische Vorlade- und Zugriffszeiten von SCAP und VSCAP vorausgesetzt. Diese Zusicherung ist hier verletzt, da nach Tabelle 5.3 die Vektorbefehle schneller als einelementige Operation abgesetzt werden können ($t_{\mathrm{v}}^L < t_{\mathrm{v}}$ und $t_{\mathrm{z}}^L < t_{\mathrm{z}}$). Die dritte Spalte zeigt die auf der T3E gemessenen Beschleunigungen, die sich ab 512 nicht-lokalen Datenzugriffen oder 64 Vektoroperationen nahe dem theoretischen Maximum befinden, das für 4096 entfernte Datenzugriffe oder 512 Vektorzugriffe erreicht wird.

Tabelle 5.4: Laufzeitbeschleunigung von VSCAP gegenüber SCAP

| Virtualisierung | Modell | T3E |
|---|---|---|
| 64 | 7.8 | 5.2 |
| 128 | 8.2 | 7.0 |
| 256 | 8.0 | 7.6 |
| 512 | 8.0 | 7.9 |
| 1024 | 8.0 | 8.1 |
| 2048 | 8.2 | 8.1 |
| 4096 | 8.2 | 8.2 |

### 5.2.3 (1,$L$)-Vektorstrategie

Die (1,$L$)-Vektorstrategie wird bei dynamischen Kommunkationsmustern eingesetzt, bei denen die nicht-lokalen Datenzugriffe erst zur Laufzeit bestimmt werden können. Vertreter dieses Kommunikationsmusters sind zum Beispiel indirekt indizierte Feldzugriffe `A[i]=B[q(i)]` bei denen der Wert von $q$ erst zur Laufzeit bestimmt werden kann. Innerhalb des Datenfließbandes muß für jeden Datenzugriff separat die Prozessornummer und die knotenlokale Adresse berechnet werden. Sind die Elemente einmal vorgeladen, so können sie durch Vektorbefehle in den lokalen Speicher zurückgeschrieben werden. Dieses spekulative Vorgehen, denn jeder Zugriff wird als entfernter Zugriff behandelt, setzt allerdings vorraus, daß die Netzwerkschnittstelle auch auf den lokalen Speicher zugreifen kann. Die E-Register erfüllen diese Bedingung, so daß auf der T3E die (1,8)-Vektorstrategie für dynamische Zugriffe angewendet werden kann.
Für die Validierung des Modells mit der (1,8)-Vektorstrategie wird das Testprogramm *Indirekt* benützt. Es implementiert die Permutation eines Datenfeldes durch eine Funktion oder einen Vektor $q$, dessen Wert erst zur Laufzeit festgestellt werden kann.

**FORALL** i = 0 **TO** N-1 **DO**
  A[i] := B[q(i)];
**END**

Die Modellparameter von *Indirekt* zeigt Tabelle 5.5.

Tabelle 5.5: Modellparameter für Indirekt

| Parameter | Wert |
|---|---|
| $t_{\mathrm{v}}$ | 462 ns |
| $t_{\mathrm{z}}$ | 156 ns |
| $t_{\mathrm{z}}^{L}$ | 183 ns |

Für die Vorladeoperationen gibt es nur einen Parameter $t_{\mathrm{v}}$, der von allen drei Vorladestrategien BLOCK, SCAP und VSCAP benützt wird. Die zu Tabelle 5.3 unterschiedlichen Zugriffszeiten kommen dadurch zustande, daß jetzt von KARHPFN direkt in den Programmtext eingefügte Kommunikationsfließbänder verwendet werden, während das Einblock-Fließband

als Makro implementiert ist. Mit den Parametern aus Tabelle 5.5 gilt

$$t_z^L = 183 < 8 * t_z = 1248 < 3696 \text{ und } t_n = 13.3 < t_v = 462.$$

Nach Tabelle 3.3 von Seite 47 wird die T3E in die Modellfälle 1, 2 und 3 eingeordnet. Das Modell prognostiziert für VSCAP bei $K \geq 128$ nicht-lokalen Datenzugriffen bis zu 99% Latenzzeitverbergung. Den Vorteil der Vektorbefehle gegenüber dem SCAP-Verfahren schätzt das Modell auf 25%.
Die Laufzeiten von *Indirekt* für BLOCK und VSCAP zeigt Abbildung 5.3.

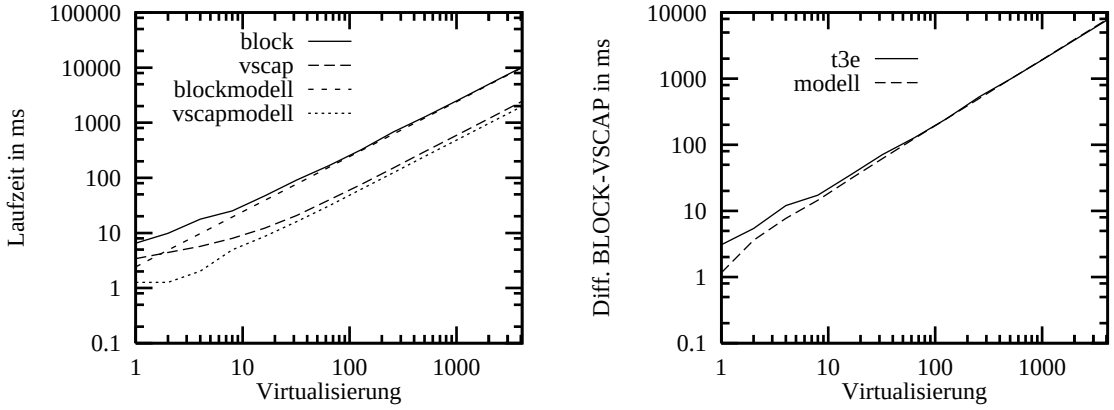



Abbildung 5.3: Vergleich der Laufzeiten von VSCAP und BLOCK

In der linken Grafik kann wieder der lineare Anstieg der Kommunikationszeit bei BLOCK für das Modell und die T3E beobachtet werden. Es ist auch der schon früher betrachtete Warmlaufeffekt bei den VSCAP Zeiten zu sehen. Die Grafik auf der rechten Seite zeigt die recht genauen Abschätzungen für die eingesparte Laufzeit beim Wechsel von BLOCK zu VSCAP. VSCAP erreicht ab $K = 128$ nicht-lokalen Zugriffen eine Latenzzeitverbergung von 100%, die sehr nahe am geschätzten Wert von 99% des Modells liegen
Die Laufzeiten für den Vergleich von VSCAP und SCAP zeigt Abbildung 5.4.

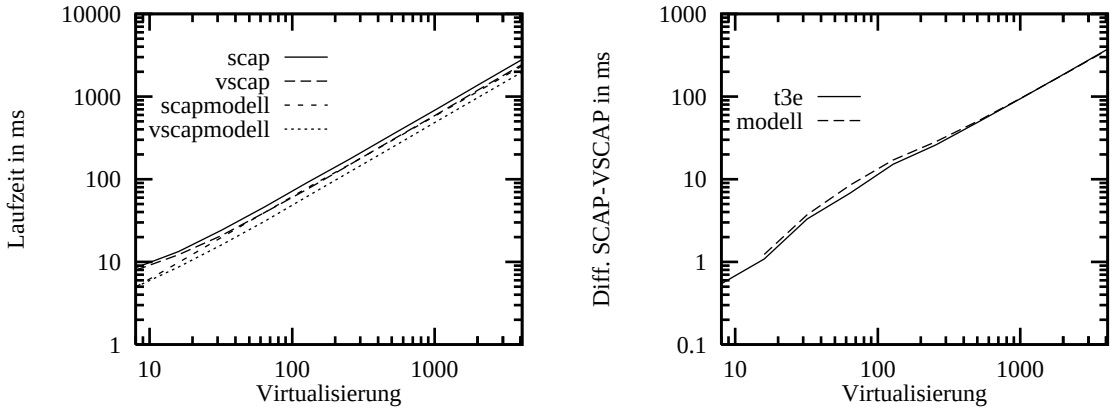



Abbildung 5.4: Vergleich der Laufzeiten von VSCAP und SCAP

Die Grafik auf der linken Seite zeigt die Laufzeiten der T3E und des Modells für SCAP und VSCAP. Die Kurven auf der rechten Seite zeigen die guten Modellabschätzungen der

Maschine. Die Laufzeitreduktion von VSCAP wurde bei $K = 4096$ mit 13% gemessen, was deutlich vom geschätzten Wert von 25% abweicht. Der Grund für diese Abweichung liegt in der größeren Laufzeit von VSCAP auf der T3E, die vom Modell um 20% abweicht. Die Ursache für diese Abweichung liegt an der Ausführungszeit der kombinierten Zugriffs- und Vorladeschleife (Schleife 2 der (1,$L$)-Vektorstrategie von 3.1.5), denn sie braucht nicht wie berechnet $t_z^L + L * t_v = 3879ns$ sondern $900ns$ länger. Dies liegt an der inneren Schleife, die pro Iteration nicht $t_v = 462$ sondern etwa $100ns$ länger dauert, da jetzt nicht wie bei den Messungen 128 sondern nur 8-mal iteriert wird. Rechnet man diese Zeit in den Laufzeitgewinn mit ein, so erhält man einen Vorteil der Vektorbefehle von 29%, was den Modellaussagen entspricht.
Diese Abweichung von Maschine von Modell ist auf eine Schwäche der Analytik zurückzuführen, die Aufgrund der Vereinfachung des Parameterraums, in der nur *ein* Wert für Vorlade- und Zugriffsoperationen zugelassen wurde, keine exakte Wiedergabe der Verhältnisse auf der T3E ermöglichte. Aber gerade wegen dieser Widrigkeiten konnten tiefere Einblicke in das VSCAP-Verhalten auf der T3E gewonnen werden, die eine genauere Modellierung der Vorlade- und Zugriffszeiten nach sich ziehen.

## 5.3 Leistungsvergleich

Im Leistungsvergleich werden die von KarHPFn übersetzten Programme nicht nur in Relation zueinander gesehen, sondern es werden auch Vergleiche zu den hochoptimierten *shared-memory*-Funktionen (SHMEM) und dem kommerziell erhältlichen HPF-Übersetzer von Portland Group [19] (PGHPF) gezogen. Der Vergleich zu SHMEM zeigt die reine Kommunikationsleistung, der von KarHPFn generierten Datenfließbänder, während der Vergleich zu PGHPF, die Effizienz gegenüber einem anderen Übersetzungssystem zeigt.
Zunächst wird die Testumgebung und das Format der Abbildungen und Tabellen beschrieben. Danach werden die in 5.1.2 vorgestellten Algorithmenklassen am Beispiel eines speziellen Vertreters besprochen.

### 5.3.1 Beschreibung der Testumgebung

Jedes Programm wird in fünf verschiedenen Versionen getestet.

**BLOCK:** (1,1)-Vorladestrategie

**SCAP:** (128,1)-Vorladestrategie

**VSCAP:** (128,8)-Vorladestrategie mit (8,8)- oder (1,8)-Vektorstrategie

Es werden in KarHPFn die Transformationsmuster für Vektorstrategien aus Tabelle 4.3 von Seite 77 gewählt.

**Shared-memory Bibliothek (SHMEM):** In dieser Version wird die Kommunikation ausschließlich mit den *shared-memory* Bibliotheksfunktionen ausgeführt. Allerdings werden keine dynamischen Kommunikationsmuster unterstützt, so daß in diesem Fall jedes Element einzeln angefordert werden muß, was letztendlich einer blockierenden Kommunikation gleichkommt.

**PGHPF:** Als Vergleich zu einem existierenden datenparallelen Übersetzungssystem werden die HPF-Quellen noch mit dem HPF-Übersetzer von Portland Group übersetzt [19]. KarHPFn und PGI HPF übersetzen identische HPF Programme.

Die ersten vier Versionen wurden mit KarHPFn übersetzt. Die Wahl der Vorladestrategie erfolgte mit Kommandozeilenoptionen (siehe Abschnitt 4.3.3). Alle Programme wurden mit den Standardoptimierungen `-O2` übersetzt.
Die Tests wurden mit einer konstanten Anzahl an Prozessoren durchgeführt, wobei der Grad der Virtualisierung geändert wurde, denn das Verhalten von VSCAP sollte in Abhängigkeit der lokalen Problemgröße getestet werden. Die Diskussion jeder Algorithmenklasse erfolgt an einem Vertreter. Die Ergebnisse werden in drei Abbildungen gezeigt. Die ersten beiden Abbildungen zeigen links die Laufzeit und rechts die Beschleunigung der einzelnen Versionen relativ zu BLOCK. Die x-Achse zeigt die verschiedenen Virtualisierungen. Sie wurden mit eins beginnend in 2er-Potenzen variiert. Die y-Achse führt die Laufzeiten in $ms$ bzw. die Beschleunigungen in Faktoren auf. Beschleunigungen über eins bedeuten eine schnellere Ausführung um den entsprechenden Faktor und Zahlen kleiner eins zeigen eine entsprechend langsamere Ausführung an.
Die dritte Abbildung zeigt die PRAM-Effizienz der getesteten Versionen. Die x-Achse zeigt wie vorhin die Größe der lokalen Datenfelder und die y-Achse repräsentiert die PRAM-Effizienz in Prozent. Die PRAM-Effizienz ist dabei nach (5.1) und (5.2) definiert. Die PRAM-Version der einzelnen Programme entstanden durch Streichen der Datenfließbänder und der globalen Synchronisationsbarrieren. Die Kommunikationsmuster wurden von Hand auf Zugriffe lokaler Daten umgebogen, so daß das Programm auf einem Prozessor ausführbar wurde.
Da die Tests auf einer festen Architektur durchgeführt wurden, stellt die Variation der Virtualisierung eine der zwei veränderbaren Größen dar. Die zweite veränderbare Größe ist die Anzahl der parallel eingesetzten Prozessoren. Sie wird in 5.4 betrachtet.

### 5.3.2 Agorithmenklasse K1a

Klasse $K1a$ umfaßt Algorithmen, in denen hauptsächlich Reduktionsoperationen Anwendung finden. Die Anzahl der nicht-lokalen Kommunikationszugriffe beträgt $f * log_f(P) + 1$. Als Vertreter dieser Klasse wird im folgenden der Kern LL3 verwendet, der das Skalarprodukt zweier Vektoren $x$ und $y$ berechnet.
Abbildung 5.5 zeigt das Laufzeitverhalten des Kerns LL3 für die unterschiedlichen Kommunikationsmodi. Als Verzweigungsgrad für die Reduktion wurde $f = 16$ eingesetzt. LL3 wurde auf 64 Prozessoren ausgeführt. KarHPFn setzte für BLOCK, SCAP und VSCAP das in 4.1.2.3 beschriebene Transformationsmuster mit der Einblock-Transformation ein. Für SHMEM wurde die `sum_to_all`-Reduktionsfunktion gewählt.
Wie erwartet, können die überlappenden Kommunikationsstrategien nur bei einer kleinen Virtualisierung ($V \leq 1000$) durch die Latenzzeitverbergung profitieren. Danach überwiegt der lokale Berechnungsaufwand, und es wird unerheblich, auf welche Art und Weise die zur Reduktion erforderliche Kommunikation ausgeführt wird. VSCAP und SCAP sind beide etwa 2.1-mal schneller als BLOCK. Daß VSCAP keine weiteren Vorteile durch die Vektorbefehle erlangt, liegt an der geringen Anzahl $f = 16$ an entfernten Datenzugriffen pro Reduktionsschritt, mit denen die Latenzzeit selbst bei SCAP nicht vollständig überdeckt werden kann. Überraschend ist das Abschneiden von SHMEM, das zwar 40% schneller als BLOCK aber 56%

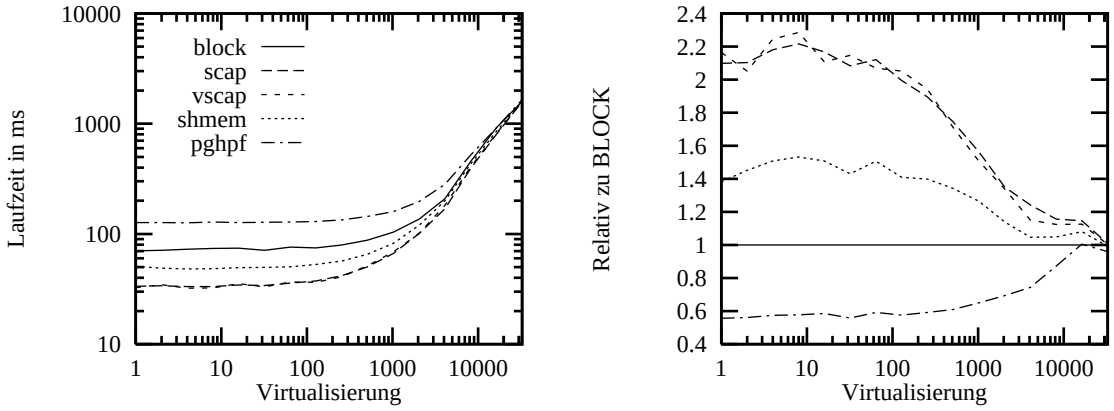



Abbildung 5.5: Laufzeiten und Beschleunigung von LL3

langsamer als VSCAP oder SCAP ist. PGHPF ist bis auf große Virtualisierungen etwa 80% langsamer als BLOCK und fast 3-mal langsamer als VSCAP und das bei $log_{16}(64)*16+1 = 25$ nicht-lokalen Datenzugriffen.

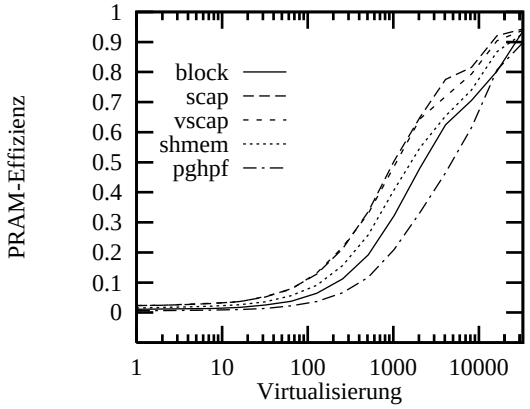



Abbildung 5.6: PRAM-Effizienz von LL3

Abbildung 5.6 zeigt die PRAM-Effizienzen der einzelnen Programmvarianten. Es ist ganz deutlich das prognostizierte Verhalten zu beobachten. Für kleine Problemgrößen überwiegt der Kommunikationsaufwand und die PRAM-Effizienz ist kleiner als 5%. Erst mit steigender Virtualisierung verschiebt sich das Kommunikations-Berechnungsverhältnis zu ungunsten der Kommunikation, so daß der Kommunikationsmehraufwand im Vergleich zur Gesamtlaufzeit nicht mehr ins Gewicht fällt. So erreichen alle Varianten eine PRAM-Effizienz von mindestens 90%, VSCAP und SCAP sogar an die 96%.

Die anderen Vertreter der Algorithmenklasse $K1a$ zeigen im wesentlichen dasselbe Verhalten wie der ausgewählte Vertreter LL3 (siehe Tabelle 5.6).

Zum einen werden die maximalen Beschleunigungen gegenüber BLOCK bei recht geringen Virtualisierungen erzielt (LL4 bei 128 lokalen Elementen und LL24 bei 512) und zum anderen erreichen alle mit KarHPFn generierten Versionen eine maximale PRAM-Effizienz von deutchlich über 90%. Nur PGHPF erreicht nur knapp die 80% Grenze.

Tabelle 5.6: Ergebnisse der weiteren Testprogramme aus der Algorithmenklasse K1a

| relativ zu BLOCK | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Programm** | **Anzahl Proz.** | | **SCAP** | | **VSCAP** | | **SHMEM** | | **PGHPF** | |
| | | | Bschl. | Virt. | Bschl. | Virt. | Bschl. | Virt. | Bschl. | Virt. |
| LL4 | 32 | | 2.1 | 128 | 2.2 | 128 | 1.4 | 128 | 0.9 | 16384 |
| LL24 | 32 | | 2.6 | 512 | 2.4 | 512 | 2.5 | 512 | 0.8 | 32768 |
| **PRAM-Effizienz** | | | | | | | | | | |
| | **BLOCK** | | **SCAP** | | **VSCAP** | | **SHMEM** | | **PGHPF** | |
| | % | Virt | % | Virt. | % | Virt. | % | Virt. | % | Virt. |
| LL4 | 92 | 32768 | 95 | 32768 | 95 | 32768 | 94 | 32768 | 78 | 32768 |
| LL24 | 92 | 32768 | 96 | 32768 | 93 | 32768 | 95 | 32768 | 76 | 32768 |

### 5.3.3 Agorithmenklasse K1b

Klasse $K1b$ spezifiziert Algorithmen, deren Kommunikationsbedarf sich aus indizierten Feldzugriffen ergibt. Als Vertreter dieser Klasse wird im folgenden der Kern LL1 verwendet. Der Kern stammt aus einer Hydrodynamik-Berechnung, deren Kommunikationsverhalten sich bei einer blockweisen Datenverteilung auf maximal 11 nicht-lokale Feldzugriffe beschränkt. KarHPFn setzt in diesem Fall die spezielle Mehrblock-Transformation aus 4.1.2.2 ein, bei der *alle* entfernten Datenelemente auf einmal in den Vorladpuffer geladen werden können ($11 \leq C_V$). Bei SHMEM wird die allgemeine Mehrblock-Transformation eingesetzt. LL1 wurde auf 32 Prozessoren getestet.
Abbildung 5.7 zeigt das Laufzeitverhalten des Kerns LL1 für die unterschiedlichen Programmvarianten.

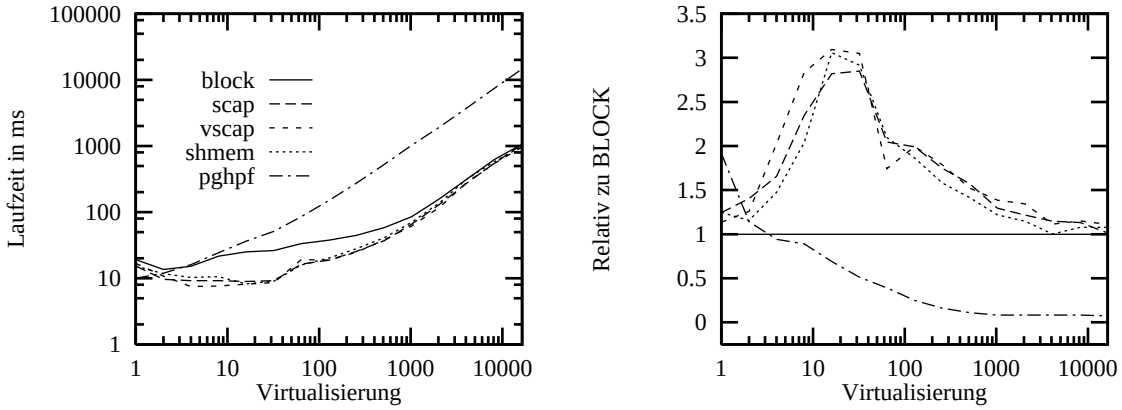



Abbildung 5.7: Laufzeiten und Beschleunigung von LL1

Die mit KarHPFn generierten Programmvarianten zeigen das erwartete Verhalten. Für kleine Problemgrößen profitieren VSCAP, SCAP und SHMEM durch die Verkleinerung des Kommunikationsaufwandes. Erst bei größeren Virtualisierungen ($V \geq 1000$) fällt die Einsparung durch Latenzzeitverbergung nicht mehr ins Gewicht und alle vier KarHPFn-Versionen haben die gleichen Laufzeiten. Der Vorteil der Latenzzeitverbergung zeigt sich beim größten Verhältnis von Kommunikations- zu Berechnungsaufwand und einer maximalen Anzahl von nicht-lokalen Datenzugriffen. Dieser Punkt ist bei einer lokalen Problemgröße von 11 Elementen erreicht. VSCAP und SHMEM sind dann etwa 3-mal schneller als BLOCK. SCAP ist etwa 2.7-mal schneller als BLOCK und erreicht, wie vermutet, fast die Leistung von VSCAP

und SHMEM. PGHPF zeigt ein nicht erklärbares Verhalten, das bei einem relativen Laufzeitvorteil von Faktor 1.8 beginnt und in einer Verlangsamung um den Faktor 13.7 endet.
Obwohl VSCAP die optimierte Mehrblock-Strategie ausführt, treten ab einer Virtualisierung von 11 Elementen keine weiteren Vorteile gegenüber SHMEM auf, da ab 11 Elementen alle nicht-lokalen Datenelemente vom denselben Prozessor geladen werden müssen und die Mehrblock-Strategie bei SHMEM intern zur Einblock-Strategie degeniert.

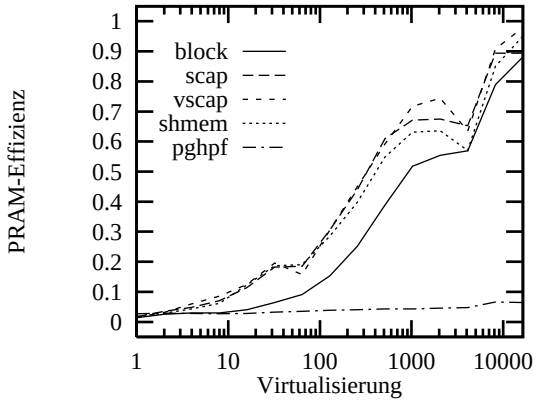



Abbildung 5.8: PRAM-Effizienz von LL1

Abbildung 5.8 zeigt die PRAM-Effizienzen der verschiedenen Programmversionen. Bei BLOCK zeigt sich erst ein Anstieg der PRAM-Effizienz, nachdem die Virtualisierung größer als 11 Elemente ist. Sie steigt am Ende bis auf 90%. Bei VSCAP, SHMEM und SCAP steigt die PRAM-Effizienz schon etwas früher, ab einer lokalen Problemgröße von etwa $V = 8$, weil dort die Latenzzeitverbergung schon erheblich die Ausführungszeit verringert (Faktor 2 gegenüber BLOCK, siehe Abbildung 5.7). Am Ende ($V = 16384$) werden von den drei überlappenden Vorladestrategien PRAM-Effizienzen von deutlich über 90% erreicht, da der lokale Berechnungsaufwand die Gesamtlaufzeit von LL1 bestimmt. Die Knicke in der PRAM-Effizienz bei einer Virtualisierung von $V = 4096$ sind auf Zugriffsfehler im 8KByte großen L1-Cache zurückzuführen.
Tabelle 5.7 zeigt ein ähnliches Verhalten der weiteren Vertreter aus Algorithmenklasse $K1b$. Auffallend sind zum einen die identischen Laufzeiten von BLOCK, SCAP, VSCAP und SHMEM bei LL8. Dies liegt am einzigen nicht-lokalen Datenzugriff, der eine Latenzzeitverbergung durch das vorgestellte VSCAP-Verfahren nicht möglich macht. Zum anderen fallen die PRAM-Effizienzen von über 100% auf, die ausschließlich auf Cacheeffekte zurückzuführen sind. Dies trifft auch auf die Beschleunigungen bei LL12 zu, die bei einer recht großen Virtualisierungen $V = 2048$ erreicht werden.

### 5.3.4 Agorithmenklasse K1d

Die Klasse $K1d$ umfaßt Algorithmen, die als Kommunikationsmuster die Nachbar-Nachbar-Kommunikation in einem $n$-dimensionalen Gitter verwenden. Als Vertreter dieser Klasse wird die 2-dimensionale Jacobi-Iteration verwendet, die pro Feldelement das arithmetische Mittel der vier Nachbarelemente berechnet. Jacobi-Iterationen werden zum Beispiel in iterativen Lösungsverfahren für partielle Differentialgleichungen eingesetzt.
KarHPFn setzt für die Kommunikation der Ränder die Einblock-Transformation ein. Die Jacobi-Iteration wurde auf 64 Prozessoren getestet.

Tabelle 5.7: Ergebnisse der weiteren Testprogramme aus der Algorithmenklasse K1b

| **relativ zu BLOCK** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Prog.** | **Anzahl Proz.** | **SCAP** | | **VSCAP** | | **SHMEM** | | **PGHPF** | |
| | | Bschl. | Virt. | Bschl. | Virt. | Bschl. | Virt. | Bschl. | Virt. |
| LL7 | 64 | 2.0 | 4 | 2.1 | 4 | 1.9 | 8 | 0.1 | 32768 |
| LL8 | 32 | 1.0 | 64 | 1.0 | 64 | 1.0 | 64 | 0.8 | 1024 |
| LL12 | 32 | 1.2 | 2048 | 1.2 | 2048 | 1.2 | 2048 | 1.0 | 32768 |
| LL15 | 32 | 2.0 | 1 | 2.0 | 1 | 2.0 | 1 | 1.2 | 8 |
| LL18 | 32 | 1.9 | 2 | 2.0 | 4 | 2.0 | 4 | 1.0 | 32768 |
| **PRAM-Effizienz** | | | | | | | | | |
| | **BLOCK** | | **SCAP** | | **VSCAP** | | **SHMEM** | | **PGHPF** |
| | % | Virt | % | Virt. | % | Virt. | % | Virt. | % | Virt. |
| LL7 | 96 | 8192 | 96 | 32768 | 97 | 32768 | 97 | 32768 | 10 | 32768 |
| LL8 | 104* | 32768 | 100 | 16384 | 103 | 32768 | 104 | 32768 | 74 | 1024 |
| LL12 | 93 | 8192 | 106 | 4096 | 108 | 4096 | 110 | 4096 | 94 | 32768 |
| LL15 | 98 | 32768 | 98 | 32768 | 97 | 32768 | 98 | 32768 | 99 | 32768 |
| LL18 | 96 | 4096 | 96 | 4096 | 95 | 4096 | 98 | 16384 | 102 | 16384 |

*Prozentangaben über 100% sind auf Cacheeffekte zurückzuführen.

Abbildung 5.9 zeigt das Laufzeitverhalten der verschiedenen Versionen. Die Virtualisierung gibt die Größe der lokalen 2-dimensionalen Matrizen an.

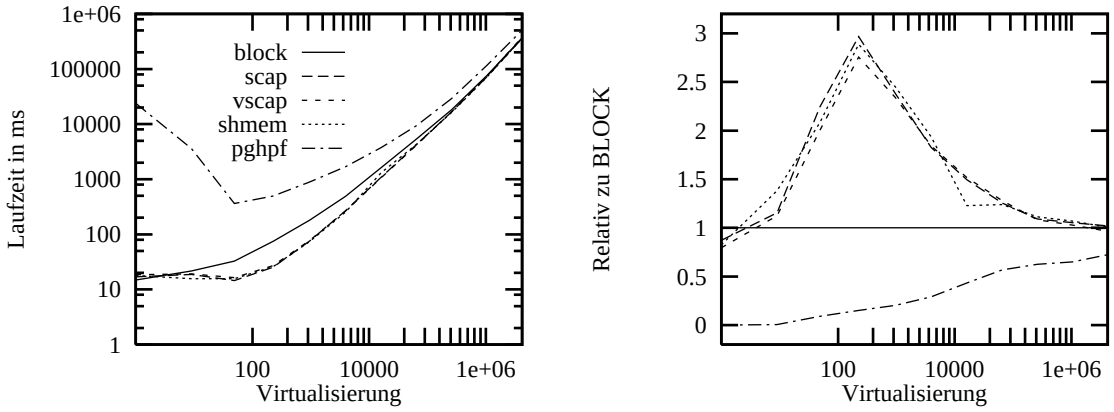



Abbildung 5.9: Laufzeiten und Beschleunigung von Jacobi

In der linken Grafik fallen drei verschiedene Laufzeitkurven auf. Die oberste Kurve zeigt das Laufzeitverhalten von PGHPF, die mittlere Kurve zeigt BLOCK und in der untersten Kurve liegen VSCAP, SCAP und SHMEM. Die unterste Kurve zeigt das erwartete Verhalten, bei dem ein Laufzeitvorteil für relativ kleine Virtualisierungen erwartet wurde ($V \leq 100^2$). Danach überwiegt der quadratische Berechnungsaufwand und der Kommunikationsaufwand fällt nicht mehr ins Gewicht.
Dasselbe Verhalten zeigt auch die rechte Grafik mit den Beschleunigungen der einzelnen Programmvarianten gegenüber BLOCK. VSCAP, SHMEM und SCAP erreichen bei $V = 15^2$ fast identische Beschleunigungen von Faktor 2.6 bis 2.8. Bei größeren Virtualisierungen $V = 1023^2$ sind alle Versionen fast genauso schnell wie BLOCK. Ein Rätsel bleibt die Laufzeitkurve von PGHPF, das auch für große Virtualisierungen 38% langsamer als BLOCK ist.

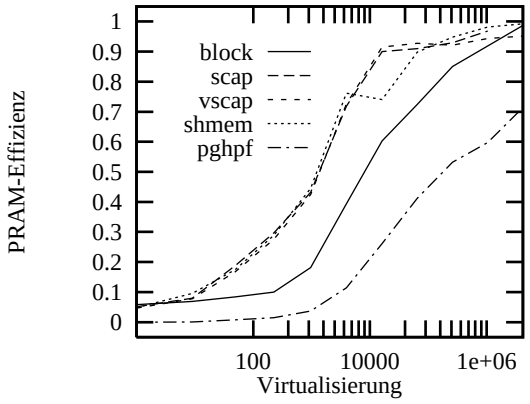



Abbildung 5.10: PRAM-Effizienz von Jacobi

Die PRAM-Effizienzen der einzelnen Versionen zeigt Abbildung 5.10. Dort ist die Dominanz des quadratischen Berechnungsaufwandes ersichtlich, denn für Virtualisierungen $V \leq 100^2$ erreichen VSCAP und SCAP eine PRAM-Effizienz von 90%. Der Knick in der SHMEM-Kurve, der auch schon in der rechten Grafik von 5.9 erkennbar war, ist auf Hardwareeffekte zurückzuführen. Für große Virtualisierungen $V \geq 2047^2$ erreichen alle KarHPFn-Programme Effizienzen von 95% und mehr.

Tabelle 5.8: Ergebnisse der weiteren Testprogramme aus der Algorithmenklasse K1d

| **relativ zu BLOCK** | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Programm** | **Anzahl Proz.** | | **SCAP** | | **VSCAP** | | **SHMEM** | | **PGHPF** | |
| | | | Bschl. | Virt. | Bschl. | Virt. | Bschl. | Virt. | Bschl. | Virt. |
| Laplace | 64 | | 2.4 | $15^2$ | 2.4 | $15^2$ | 2.0 | $15^2$ | 0.9 | $2047^2$ |
| **PRAM-Effizienz** | | | | | | | | | | |
| | **BLOCK** | | **SCAP** | | **VSCAP** | | **SHMEM** | | **PGHPF** | |
| | % | Virt | % | Virt. | % | Virt. | % | Virt. | % | Virt. |
| Laplace | 96 | $2047^2$ | 99 | $1023^2$ | 95 | $2047^2$ | 99 | $1023^2$ | 83 | $2047^2$ |

Tabelle 5.8 zeigt die maximalen Laufzeitvorteile und PRAM-Effizienzen des anderen Programms aus der Algorithmenklasse $K1d$. Das Laufzeitverhalten von PDE1 wird in 5.4.2 bei der Variation der Prozessorzahl gezeigt.

### 5.3.5 Agorithmenklasse K1e

Die Algorithmenklasse $K1e$ umfaßt Programme, deren Laufzeit durch eine Verteiloperation bestimmt ist. Dabei kopiert jeder Prozessor die Daten zu sich in den lokalen Speicher, ohne daß dabei ein Reduktionsbaum aufgebaut wird. Ist der entfernte Datenblock groß genug, so kann VSCAP durch die Vektorbefehle genug Kommunikationszeit gegenüber SCAP einsparen. Das Laufzeitverhalten dieser Klasse wird an LL21 gezeigt. LL21 ist eine Matrixmultiplikation zweier blockweise verteilten Matrizen identischer Größe.
KarHPFn verwendet für die Kommunikation der Teilmatrizen die Einblock-Transformation. Die Tests wurden auf 64 Prozessoren ausgeführt. Abbildung 5.11 zeigt die Laufzeiten und Beschleunigungen der einzelnen Programmvarianten von LL21. Die Virtualisierung gibt die Größe der lokalen Matrizen an.

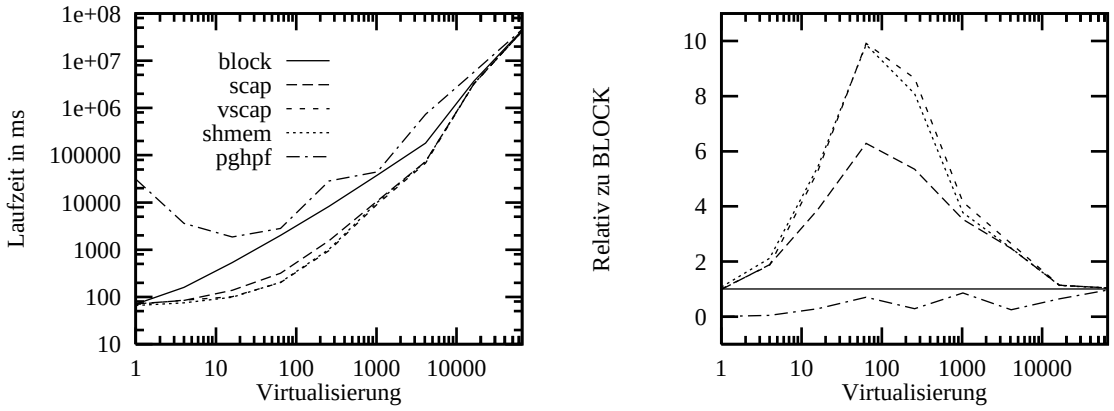



Abbildung 5.11: Laufzeiten und Beschleunigung von LL21

Die Laufzeitkurven auf der linken Seite, zeigen die Dominanz des kubischen Berechnungsaufwandes der Matrixmultiplikation, denn schon bei einer lokalen Größe von $V = 256^2$ betragen die Laufzeiten ca. 40s. Die Beschleunigungen der einzelnen Programmvarianten zeigen die Laufzeitvorteile durch die überlappende Kommunikation nur bei kleinen Virtualisierungen. Bei einer Virtualisierung von $V = 8^2$ sind VSCAP, SHMEM und SCAP 9.9, 9.8 bzw. 6.3-mal schneller als BLOCK. Danach nimmt der Anteil der Kommunikation an der Gesamtlaufzeit immer mehr ab und letztendlich ist die gewählte Art der Kommunikation nicht mehr relevant. Daß der Vorteil der Vektoroperationen von VSCAP und SHMEM so groß ausfällt, obwohl die lokalen Matrizen nur einen Vektor füllen ($\sqrt{V} = 8 = L$), liegt an der Tatsache, daß die gesamte entfernte Matrix auf einmal kopiert wird. Damit erhält man $8^2 = 64$ nicht-lokale Datenzugriffe bei denen die Kommunikation mit Vektoroperationen mehr als 5-mal schneller ist als eine (1,1)-Vektorstrategie (siehe Tabelle 5.4).
Die Abbildung 5.12 mit den PRAM-Effizienzen untermauert den großen Berechnungsanteil bei LL21.

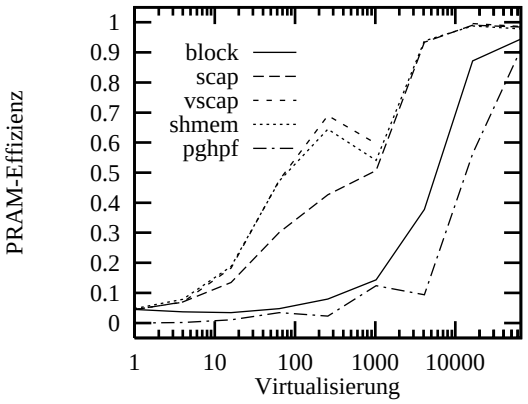



Abbildung 5.12: PRAM-Effizienz von LL21

Bei kleinen Virtualisierungen $V \leq 32^2$ erreichen BLOCK und PGHPF kaum 10% PRAM-Effizienz, während die überlappenden Kommunikationsstrategien bereits über 50% erreichen. Ab einer Virtualisierung von $V \geq 64^2$ erreichen sie über 90% (ab $V \geq 128$ sogar 99%).

BLOCK und PGHPF erreichen die 90% Schwelle erst bei $V = 256^2$. Die Knicke in den PRAM-Effizienzen von VSCAP und SCAP bei einer Virtualisierung von $V = 32^2$ sind auf Zugriffsfehler innerhalb des L1-Caches zurückzuführen.
Weitere Vertreter der Algorithmenklasse K1e gibt es nicht.

### 5.3.6 Agorithmenklasse K2a

Die Basisklasse $K2$ spezifiziert Algorithmen, die prinzipiell einen höheren Kommunikationsaufwand aufweisen als die bisher präsentierten Vertreter der Basisklasse $K1$. Bei der Unterklasse $K2a$ handelt es sich um Algorithmen, deren Kommunikationsstrukturen sich aus Reduktionsoperationen ergeben. Durch den hohen Kommunikationsanteil kann VSCAP vom Einsatz der Vektorbefehle profitieren. Als Vertreter dieser Klasse wird der Kern LL5 verwendet, der einen linearen Gleichungslöser für Bandmatrizen entstammt.
In KarHPFn werden auf die nicht-lokalen Datenzugriffe mit der Mehrblock-Transformation zugegriffen. Für die Tests wurden 32 Prozessoren eingesetzt. Abbildung 5.13 zeigt die Laufzeiten und Beschleunigungen der einzelnen Programmvarianten.

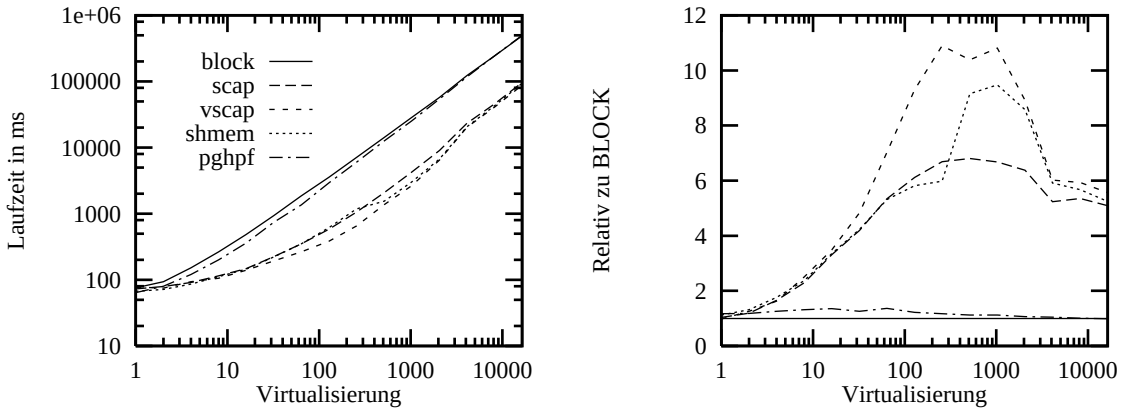



Abbildung 5.13: Laufzeiten und Beschleunigung von LL5

Die fünf Programmversionen sind in der linken Grafik in zwei Klassen aufgeteilt, die nahezu identische Laufzeiten vorweisen. Dies sind zum einen BLOCK und PGHPF und zum anderen VSCAP, SHMEM und SCAP. Es fällt auf, daß sich nun nicht wie in Basisklasse $K1$ die Laufzeiten der einzelnen Programme für größere Probleme annähern, sondern daß sie sich deutlich unterscheiden. Dieses Verhalten liegt an dem mit der zur Problemgröße linear ansteigenden Kommunikationsaufwand. Dementsprechend groß fallen auch die Beschleunigungen der Programmversionen mit überlappender Kommunikation für größere Virtualisierungen aus. Denn für $V = 16384$ ist SCAP noch 5-mal schneller als BLOCK. Die maximalen Laufzeitvorteile von VSCAP, SHMEM und SCAP werden bei Virtualisierungen zwischen $128 \leq V \leq 1024$ erreicht. VSCAP, SHMEM und SCAP sind dann 10.9, 9.5 bzw. 6.8-mal schneller als BLOCK. Der plötzliche Einbruch von VSCAP und SHMEM bei $V = 4096$ ist auf Cacheeffekte zurückzuführen. Der Anstieg der Beschleunigung von SHMEM bei $V = 128$ ist in der Implementierung der `shmem_iget` Funktion begründet, bei der wahrscheinlich[1] ab einer bestimmten Datengröße ein anderes Fließband benützt wird. Dieser Effekt ist auch in

[1] Cray hält sich bei der Programmierung der E-Register sehr bedeckt.

den Beschleunigung von SHMEM beim Testprogramm Rotieren zu beobachten (siehe Abbildung 5.15). Bei VSCAP wird von Anfang an die $(L,L)$-Vektorstrategie eingesetzt, so daß sich der Laufzeitvorteil durch die Vektorbefehle gegenüber SCAP kontinuierlich erhöht.
Abbildung 5.14 zeigt die PRAM-Effizienzen der einzelnen Programmversionen.

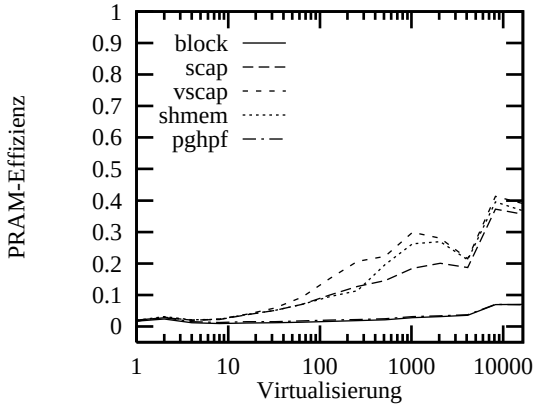



Abbildung 5.14: PRAM-Effizienz von LL5

Die Kurven spiegeln deutlich den hohen Anteil an zusätzlicher Kommunikationszeit wider, denn VSCAP, SHMEM und SCAP erreichen jetzt eine PRAM-Effizienz von knapp 40%. In der Basisklasse $K1$ waren noch an die 90% PRAM-Effizienz die Regel. BLOCK und PGHPF erreichen knapp 10%.

Tabelle 5.9: Ergebnisse der weiteren Testprogramme aus der Algorithmenklasse K2a

| **relativ zu BLOCK** | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Programm** | **Anzahl Proz.** | | **SCAP** | | **VSCAP** | | **SHMEM** | | **PGHPF** | |
| | | | Bschl. | Virt. | Bschl. | Virt. | Bschl. | Virt. | Bschl. | Virt. |
| LL2 | 32 | | 19.4 | 32768 | 30.0 | 32768 | 31.1 | 32768 | 1.7 | 32768 |
| LL11 | 32 | | 8.5 | 1024 | 15.9 | 1024 | 14.5 | 1024 | 1.2 | 1 |
| LL13 | 64 | | 6.4 | 4 | 6.7 | 4 | 6.4 | 4 | 4.6 | 256 |
| LL14 | 32 | | 7.9 | 4 | 7.9 | 4 | 6.5 | 4 | 0.2 | 64 |
| LL19 | 32 | | 6.0 | 512 | 9.4 | 512 | 8.0 | 512 | 2.1 | 32 |
| LL23 | 32 | | 5.4 | 256 | 7.1 | 512 | 6.6 | 512 | 0.6 | 2048 |
| **PRAM-Effizienz** | | | | | | | | | | |
| | **BLOCK** | | **SCAP** | | **VSCAP** | | **SHMEM** | | **PGHPF** | |
| | % | Virt | % | Virt. | % | Virt. | % | Virt. | % | Virt. |
| LL2 | 1 | 8192 | 15 | 8192 | 24 | 8192 | 24 | 8192 | 1 | 8192 |
| LL11 | 6 | 16384 | 40 | 16384 | 47 | 16384 | 47 | 16384 | 4 | 16384 |
| LL13 | 6 | 32768 | 18 | 32768 | 22 | 32768 | 5 | 32768 | 10 | 32768 |
| LL14 | 17 | 1 | 37 | 2 | 34 | 4 | 38 | 1 | 2 | 1 |
| LL19 | 10 | 32768 | 46 | 32768 | 50 | 32768 | 49 | 32768 | 15 | 32768 |
| LL23 | 17 | 16384 | 77 | 16384 | 91 | 1024 | 89 | 1024 | 11 | 16384 |

In Tabelle 5.9 sind die weiteren Vertreter der Algorithmenklasse $K2a$ aufgeführt. Es fallen die sehr großen Beschleunigungen der $(L,L)$-Vektorstrategien bei LL2 und LL11 auf (ca. 30-mal bzw. 15-mal schneller als BLOCK). Bei den PRAM-Effzienzen werden kaum 50% erreicht, bis auf den Ausreißer LL23, der durch seinen etwas größeren Berechnungsanteil mit VSCAP doch noch 90% erreicht. Besondere Beachtung muß LL13 und LL14 geschenkt werden, denn zum einen haben sie dynamische Zugriffsmuster, daher die geringe PRAM-Effizienz von SHMEM

von 5%, und zum anderen dominiert eine kommunikationsintensive Reduktion ihre Laufzeit, bei der $P$ lokale Felder zu einem globalen Datenfeld vereint werden müssen (SUM_SCATTER-Funktion in HPF). Während der Reduktion werden Einblock-Fließbänder eingesetzt, so daß SHMEM durch die Vektoroperationen wieder Laufzeit gewinnen kann. Dies erklärt die guten Beschleunigungen gegenüber BLOCK.

### 5.3.7 Agorithmenklasse K2b

Die Programm aus Klasse $K2b$ sind durch indizierte Feldzugriffe geprägt, die aufgrund der Datenverteilung oder des Zugriffsmusters einen hohen Kommunikationsanteil besitzen. Als Vertreter dieser Algorithmenklasse wird Rotieren herangezogen. Rotieren implementiert das in 5.2.2 vorgestellte zyklische Rotieren eines Datenfeldes. Um einen hohen Kommunikationaufwand zu erzielen, wurden die Felder zyklisch verteilt.
KarHPFn erkennt den indirekten Zugriff und setzt Aufgrund der zyklischen Datenverteilung ein Einblock-Fließband für das Kopieren der Daten ein (siehe Tabelle 4.3). Die Tests wurden auf 32 Prozessoren ausgeführt.
Abbildung 5.15 zeigt die Laufzeiten und Beschleunigungen der einzelnen Programmvarianten.

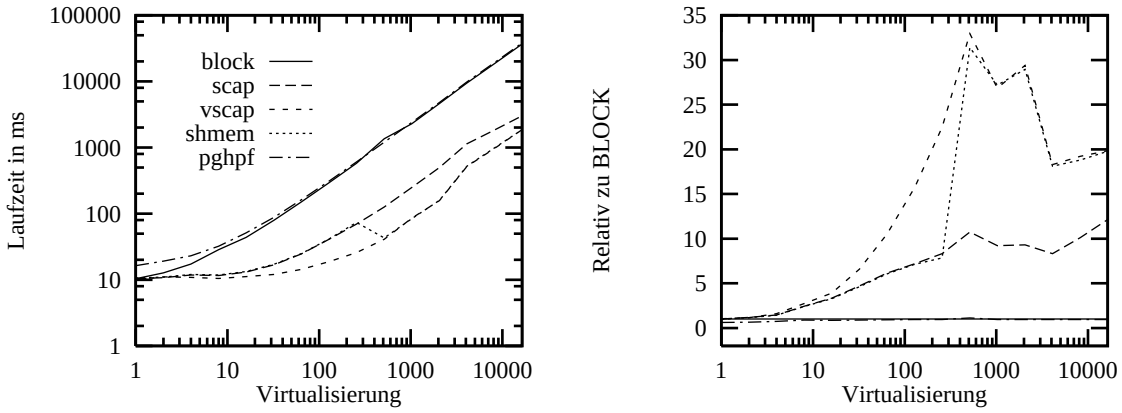



Abbildung 5.15: Laufzeiten und Beschleunigung von Rotieren

Die verschiedenen Programme lassen sich in drei Laufzeitklassen einteilen. Die Programme mit der längsten Laufzeit sind BLOCK und PGHPF. SCAP erzielt im Vergleich dazu einen deutlichen Laufzeitgewinn und VSCAP ist auch noch einmal deutlich schneller als SCAP. SHMEM hat bis zu einer Virtualisierung von $V = 128$ fast dieselben Laufzeiten wie SCAP, erfährt dann aber einen Leistungszuwachs und verhält sich danach wie VSCAP. Die Beschleunigungen der einzelnen Programme im Vergleich zu BLOCK sind beachtlich. VSCAP und SHMEM sind 33 bzw. 31.4-mal schneller als BLOCK. SCAP erreicht eine Beschleunigung von Faktor 10.7. Es ist wieder der Knick in der Beschleunigung von SHMEM zu sehen, der wahrscheinlich, wie im vorherigen Abschnitt bei der Klasse $K2a$ angesprochen, an einem internen Wechsel des Fließbandmodus bei `shmem_iget` liegt. Auffallend ist auch das Verhalten von PGHPF, das gegenüber BLOCK trotz des sehr strukturierten Kommunikationsmusters keinen nennenswerten Laufzeitgewinn erreicht.
Die PRAM-Effizienzen zeigt Abbidlung 5.16.
Bei der Betrachtung der PRAM-Effizienzen fällt zum einen wieder der Laufzeitgewinn von SHMEM bei einer Virtualisierung von $V = 128$ auf, und zum anderen werden von VSCAP

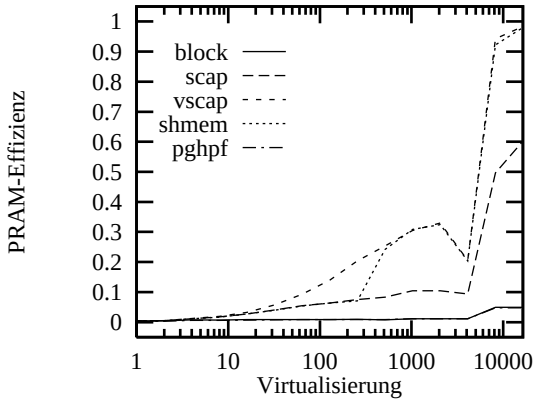



Abbildung 5.16: PRAM-Effizienz von Rotieren

und SHMEM nicht zu erwartend hohe PRAM-Effizienzen von über 95% erreicht. Dies liegt an der Tatsache, daß die Kommunikation am lokalen Datencache des Prozessors vorbeigeht und der Aufwand der Vektorvorlade- und Zugriffsoperation fast dem Zeitverlust für das Laden der Cachezeilen bei der PRAM-Version entspricht.

Tabelle 5.10: Ergebnisse der weiteren Testprogramme aus der Algorithmenklasse K2b

| **relativ zu BLOCK** | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Programm** | **Anzahl Proz.** | | **SCAP** | | **VSCAP** | | **SHMEM** | | **PGHPF** | |
| | | | Bschl. | Virt. | Bschl. | Virt. | Bschl. | Virt. | Bschl. | Virt. |
| LL6 | 32 | | 11.4 | 2048 | 22.3 | 2048 | 20.0 | 2048 | 1.1 | 8 |
| **PRAM-Effizienz** | | | | | | | | | | |
| | **BLOCK** | | **SCAP** | | **VSCAP** | | **SHMEM** | | **PGHPF** | |
| | % | Virt | % | Virt. | % | Virt. | % | Virt. | % | Virt. |
| LL6 | 1 | 1024 | 13 | 2048 | 25 | 1024 | 22 | 2048 | 1 | 1024 |

Tabelle 5.10 zeigt mit LL6 den anderen Vertreter der Algorithmenklasse *K2b*. Es werden wie bei Rotieren ähnlich große Beschleunigungen von VSCAP gegenüber SCAP und BLOCK erreicht, doch fallen die PRAM-Effizienzen nun nicht mehr so groß aus, da LL6 die Daten auch zur Berechnung benötigt und sie bei jeder Programmversion in den Prozessorcache geladen werden müssen.

### 5.3.8 Agorithmenklasse K2c

Klasse *K2c* spezifiziert Algorithmen, deren Kommunikationsoperationen sich aus indirekt indizierten Feldzugriffen ableiten und sich in der Regel nicht statisch vorberechnen lassen. VSCAP und SCAP setzten eine spekulative Vorladestrategie ein, bei der jedes Element über die E-Register vorgeladen wird, da mit einer Wahrscheinlichkeit von $\frac{P-1}{P}$ die Zugriffe ein nicht-lokales Datenelement referenzieren. Als Vertreter dieser Klasse wird im folgenden *Indirekt* verwendet, das die Zuweisung `A[i]:=B[q(i)]` implementiert. *Indirekt* wurde auf 32 Prozessoren getestet.
Abbildung 5.17 zeigt das Laufzeitverhalten von *Indirekt* für die verschiedenen Programmversionen.
Es ergibt sich die erwartete Zweiteilung der von KarHPFn übersetzten Programmversionen.

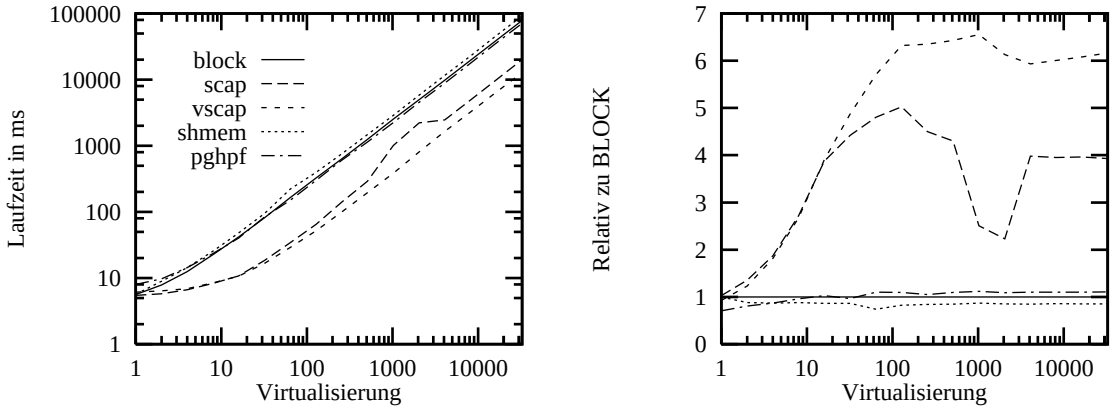



Abbildung 5.17: Laufzeiten und Beschleunigung von Indirekt

Während BLOCK und SHMEM annähernd identische Laufzeiten aufweisen, können VSCAP und SCAP durch das spekulative Vorladen die Latenzzeiten der einzelnen Kommunikationszugriffe verdecken. Dabei ist VSCAP 6.6-mal und SCAP 5.0-mal schneller als BLOCK. SHMEM ist im Schnitt etwa 15% langsamer, während PGHPF bis zu 10% schneller ist als BLOCK. Der Knick in der SCAP-Kurve ist durch die blockweise Verteilung der Datenfelder und der damit verbundenen Adreßberechnung bedingt, denn weder bei einem Softwaretest (siehe Abschnitt 5.5.1) noch bei einer zyklischen Verteilung tritt dieser Effekt auf.

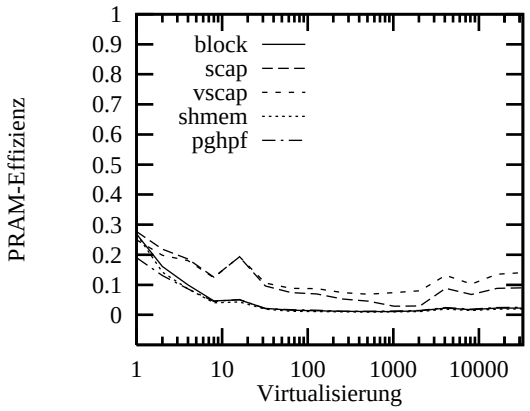



Abbildung 5.18: PRAM-Effizienz von Indirekt

Abbildung 5.18 zeigt die PRAM-Effizienz der getesteten Versionen. Auffallend sind die geringen erzielten Auslastungen, die für alle Programme und Virtualisierungen von $V \leq 4$ zwar jenseits der 10% liegen, doch für größere Probleme erreichen BLOCK, SHMEM und PGHPF nur noch 2-3%. Nur VSCAP mit 14% und SCAP mit 9% erreichen bei $V = 32768$ noch eine nennenswerte PRAM-Effizienz. Dieses Abschnieden liegt am erheblichen Zusatzaufwand durch die Berechnung der Knoten- und knotenlokalen Adressen. Wenn dieser Aufwand in die Hardware verlagert werden könnte, so würde sich noch ein weiterer Laufzeitgewinn einstellen. Die T3E bietet einen solchen Mechanismus mit der Hardware-Zentrifuge an (siehe Abschnit 4.2.3.2). Welchen Laufzeitgewinn sich durch die Einsparung der Adreßbrechnung noch erzielen lassen zeigt 5.5.2.

Ein sehr interessanter Effekt trat bei der Programmierung der PRAM-Version von *Indirekt* auf, denn für Virtualisierungen von mehr als 4096 Elementen war VSCAP schneller als die PRAM-Version. Dies lag an Cachefehlern, denn durch die dynamischen Zugriffe bei großen Virtualisierungen dominierten sie die Laufzeit der PRAM-Version. Bei VSCAP traten diese Fehler nicht auf, denn die dynamischen Zugriffe erfolgten über die E-Register, die am Prozessorcache vorbei gehen. Für die PRAM-Version ergab sich dann die Notwendigkeit, für große Probleme auf die E-Registerprogrammierung von VSCAP zurückzugreifen, so daß die Laufzeiten der PRAM-Version ab einer Virtualisierung von $V = 4096$ durch eine (1,8)-Vektorstrategie ohne weitere Adreßberechnung entfernter Datenelemente erhalten wurde.

### 5.3.9 Agorithmenklasse K2e

Auf die Diskussion der Algorithmenklasse *K2e* wird hier verzichtet, da ihr einziger Vertreter, der Veltran-Operator, bei den Tests mit variierender Prozessorzahl in Abschnitt 5.4.3 betrachtet wird.

## 5.4 Variation der Prozessorzahl

Bei den folgenden drei Tests werden bei fester Problemgröße die Anzahl der Prozessoren variiert. Dabei werden die Tests mit bis zu 128 Prozessoren ausgeführt. Damit soll die Leistungsfähigkeit von VSCAP in einem massiv parallelen Umfeld gezeigt werden, denn die bisherigen Tests benützten maximal 64 Prozessoren.
Bei diesen Tests werden Anwendungen aus der Strömungslehre (*FIRE*), Geophysik (*Veltran*) und ein Gleichungslöser für partielle Differentialgleichung (*PDE1*) betrachtet, da nicht nur das Einsatzspektrum von KarHPFn sondern sondern auch die Auswirkungen einer schnelleren Kommunikation auf das Gesamtlaufzeitverhalten einer kompletten Applikation gezeigt werden soll.
Die Darstellung der Testergebnisse folgt zwar dem in 5.3.1 erklärten Schema, doch wird hier nicht die Problemgröße sondern die Anzahl der Prozessoren variiert.

### 5.4.1 FIRE

*FIRE* ist ein Programmpacket aus der Strömungslehre, das von der AVL List Gmbh in Graz entwickelt wurde. Die hier verwendete Version von *FIRE*ist ein nach der Methode der konjugierten Gradienten auf unstrukturierten Netzen arbeitender Löser GCCG [25].
Die Kommunikation in *FIRE* besteht aus einem Einsammeln der an der Berechnung beteiligten Zellen `DIREC1`, deren Indizes in `LCC` gespeichert sind

```
FORALL J = 1 TO 6 DO
  FORALL NC = 1 TO NNINTC DO
    IF LCCMASK(J,NC) THEN
      DIREC1V(J,NC) := DIREC1(LCC(J,NC));
    END IF
  END FORALL
END FORALL
```

Das Programm konnte ohne größere Modifikation von KARHPFN übersetzt werden. Für den maskierten indirekt indizierten Zugriff setzte KARHPFN die Transformation für ein maskiertes Fließband ein (siehe 4.1.4.2). Dies hat zur Folge, daß eine (1,1)-Vektorstrategie gewählt wird, so daß überhaupt keine Vektorbefehle für die Kommunikation zum Einsatz kommen und VSCAP dasselbe Fließband wie SCAP benützt. Aus diesem Grund wurde auf eine VSCAP-Version für *FIRE* verzichtet.
Der Kommunikationsaufwand wächst linear mit der Problemgröße, so daß *FIRE* in die Algorithmenklasse $K2c$ einzuordnen ist. Bei den Tests blieb die Problemgröße mit 318045 Partikeln konstant. Die Anzahl der an der Berechnung beteiligten Prozessoren variierte von 2 bis 128. Alle beteiligten Felder wurden blockweise verteilt.
Abbildung 5.19 zeigt die Laufzeiten und Beschleunigungen der einzelnen Versionen in Abhängigkeit der eingesetzten Prozessoren.

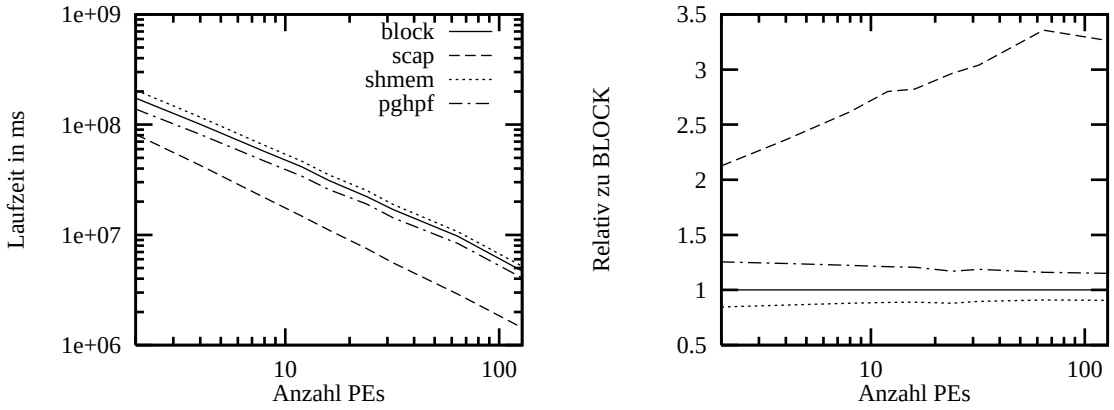



Abbildung 5.19: Laufzeiten und Beschleunigung von *FIRE*

Die Laufzeiten von BLOCK, SHMEM und PGHPF bewegen sich ungefähr im selben Bereich. Lediglich SCAP zeigt einen deutlichen Laufzeitgewinn gegenüber BLOCK. Diese Beobachtung untermauert die rechte Grafik mit den Beschleunigungen der Programmversionen gegenüber BLOCK. Der Vorteil von SCAP bewegt sich bei wenigen Prozessoren ($P = 2$, $V = 159023$) bei einem Laufzeitvorteil von Faktor 2.1. Dieser steigert sich mit zuhnemender Prozessorzahl auf einen maximalen Faktor von knapp 3.5 bei 64 Prozessoren ($V = 4970$). Die Beschleunigungen von PGHPF und SHMEM sind dagegen eher vernachlässigbar (PGHPF ca. 30% schneller und SHMEM ca. 15% langsamer). Das schlechte Laufzeitverhalten von SHMEM liegt am nicht unterstützten dynamischen Kommunikationsmuster. Das schwache Abschneiden von PGHPF liegt am *Inspector-Executor*-Modell, das die Kommunikationszeit durch zusätzlichen Verwaltungsaufwand erhöht (siehe 2.3.5).
Abbildung 5.20 zeigt die PRAM-Effizienzen der getesteten Versionen.
Es fällt die sehr gute PRAM-Auslastung von SCAP für wenig Prozessoren auf. Dies liegt am dynamischen Kommunikationsmuster. Denn die PRAM-Ausführung muß jeden Zugriff über den Datencache des Prozessors ausführen, was bei indirekt indizierten Zugriffen zu häufigen Zugriffsfehlern des Caches führt, die sich zu Lasten der Laufzeit bemerkbar machen. SCAP dagegen greift in der maskierten Kommunikation nur mit regulären Zugriffen auf `LCCMASK` und `LCC` zu. Die indirekt indizierten Zugriffe gehen über die E-Register und damit am Cache vorbei, so daß sich hier Cacheeffekte nicht negativ auswirken können. Tatsächlich mußte für eine optimale PRAM-Laufzeit bis 12 Prozessoren auf die (1,1)-Vektorstrategie innerhalb der

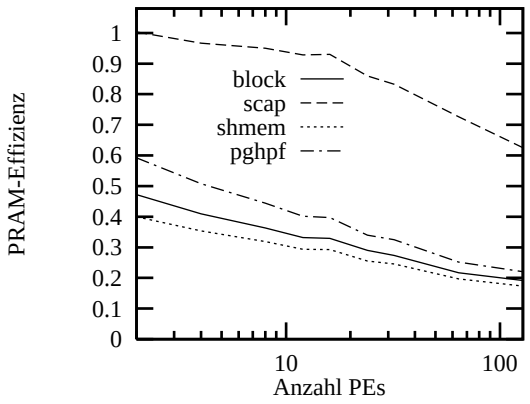



Abbildung 5.20: PRAM-Effizienz von *FIRE*

PRAM-Version zurückgegriffen werden, um schnellere Laufzeiten als SCAP zu erhalten. Erst danach waren die reinen Cachezugriffe schneller. Dieser Effekt erklärt die PRAM-Effizienz von SCAP von mehr als 95% bis 12 Prozessoren ($V \geq 26504$). Selbst bei 128 Prozessoren wird noch eine PRAM-Effizienz von 62% erreicht. Dies entspricht einem Speed-Up von 79 bei 128 Prozessoren. Die anderen drei Versionen erreichen dagegen einen Speed-Up von 25.

### 5.4.2 PDE1

Das Testprogramm *PDE1* (*partial differential equation*) ist ein 3-dimensionaler Poisson-Löser mit einer Red-Black-Relaxation. Bei diesem Löser partieller Differentialgleichungen wird das 3-dimensionale Gitter in disjunkte *rote* und *schwarze* Punkte aufgeteilt. Im ersten Schritt werden mit Hilfe der 6 Nachbarn die neuen Werte der roten Elemente ausgerechnet. Mit diesem Ergebniss werden schließlich nach dem gleichen Schema die schwarzen Punkte bestimmt. *PDE1* gehört zur Algorithmenklasse $K1d$. Bei der Kommunikation müssen die Ränder der lokalen Würfel von logisch benachbarten Prozessoren kopiert werden. Der Kommunikationsaufwand wächst damit quadratisch zur Kantenlänge der lokalen Würfel während der Berechnungsaufwand kubisch dazu ist. Das bedeutet, daß bei einer kleinen Anzahl an Prozessoren und großer Virtualisierung die gewählte Kommunikationsart bei der Ausführungszeit nicht ins Gewicht fällt, während bei mehreren Prozessoren und einer kleineren Virtualisierung die überlappenden Kommunikationsstrategien einen Vorteil durch den verhältnismäßig geringeren Berechnungsaufwand besitzen.
Bei der Übersetzung von *PDE1* erweitert KarHPFn die lokalen Würfel um einen Überlappungsbereich in jeder Dimension, in den die Nachbarelemente kopiert werden (siehe 4.1.2.1). Dazu wird das Einblock-Fließband verwendet, denn durch den in jeder Dimension blockweise verteilten Würfel liegen die Nachbarelemente eines Randes nur auf *einem* entfernten Prozessorknoten. Die Tests wurden auf 4 bis 128 Prozessoren bei einer festen globalen Würfelgröße von $N = 128^3$ Punkten durchgeführt.
Abbildung 5.21 zeigt die Laufzeiten und Beschleunigungen der einzelnen Programmversionen. Man sieht das für die Algorithmenklasse $K1d$ typische Laufzeitverhalten. Bei wenig Prozessoren $P = 4$ und großer Virtualisierung sind die Laufzeiten durch den Berechnungsaufwand geprägt. Erst mit steigenden Prozessorzahlen und damit kleineren lokalen Würfelgrößen macht sich der Vorteil der schnelleren Kommunikation von VSCAP, SHMEM und SCAP bemerkbar.

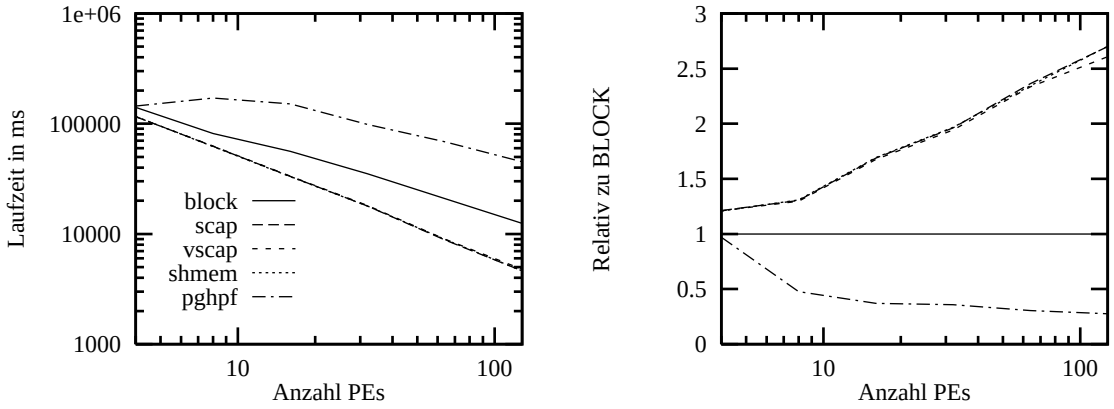



Abbildung 5.21: Laufzeiten und Beschleunigung von *PDE1*

Die fast identischen Laufzeiten der eben genannten Programmversionen liegt an der kleinen Seitenlänge der lokalen Würfel (bei 128 Prozessoren 16 Elemente), bei denen Vektorbefehle keine oder nur eine sehr kleine Ersparnis der Kommunikationszeit liefern. Die Beschleunigungen der drei Programmversionen mit überlappender Kommunikation sind daher nahezu identisch. Bei 4 Prozessoren haben sie einen Laufzeitgewinn von 20% gegenüber BLOCK der sich bis zu einem Faktor von 2.7 bei 128 Prozessoren erhöht. PGHPF zeigt das schon bei der Diskussion der Algorithmenklasse $K1d$ von 5.3.4 beobachtete Verhalten, daß die zu BLOCK relative Leistung bei kleinen Virtualisierungen gering ist. PGHPF ist bei 128 Prozessoren etwa 3.7-mal langsamer.
Abbildung 5.22 zeigt die PRAM-Effizienzen der getesteten Versionen.

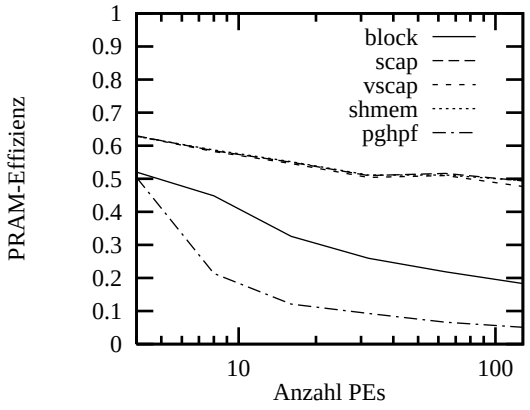



Abbildung 5.22: PRAM-Effizienz von *PDE1*

Für 4 Prozessoren erreichen VSCAP, SHMEM und SCAP eine PRAM-Effizienz von 62%. BLOCK und PGHPF erreichen jeweils 50%. Sie verringert sich bei den Versionen mit überlappender Kommunikation auf knapp 50%, während sich die PRAM-Auslastung von BLOCK auf 19% und die von PGHPF auf 5% reduziert. Eigentlich hätte man bei großen Virtualisierungen PRAM-Effizienzen von ca. 90% erwartet, doch ist der Aufwand für die Kommunikation größer als dies zum Beispiel bei den anderen Vertretern der Algorithmenklasse $K1d$ der Fall war.

### 5.4.3 Veltran-Operator

Der *Veltran-Operator* (*Veltran*) ist eine Anwendung aus der Geophysik, in der durch die Analyse von Wellengeschwindigkeiten die Zusammensetzung von Erdschichten berechnet wird [72], dabei wird nach der Methode der konjugierten Gradienten vorgegangen.
Während der Berechnung müssen 2-dimensionale Bereiche der 3-dimensionalen lokalen Datenfelder an alle Prozessoren verteilt werden. Da der Berechnungsaufwand mit dem Aufwand der Verteiloperation gleichzusetzen ist, wird *Veltran* in die Algorithmenklasse $K2e$ eingeordnet. Damit ist bei wenig Prozessoren und großer Virtualisierung mit einem Gewinn durch die überlappenden Kommunikationsstrategien zu rechnen.
Als HPF-Quelle diente der Programmtext aus Anhang A.1, in dem die Parallelität durch die zwei *DO-Independent*-Schleifen ausgedrückt wird. KarHPFn parallelisiert jedoch nur Forall-Schleifen, so daß der Quelltext in ein äquivalentes Programmstück umgeformt werden mußte, in dem Parallelität ausschließlich durch Foralls ausgedrückt wird, siehe Anhang A.2. Ein erstaunlicher Effekt dieser Umformulierung war eine Reduktion der Laufzeit um einen Faktor 2, als beide Programme mit PGHPF übersetzt und getestet wurden, obwohl in der Forall-Variante zusätzliche lokale Datenfelder angelegt werden müssen. Der Laufzeitvorteil liegt an der einfacheren Analyse der nicht-lokalen Datenzugriffe, die nun auf *ein* Forall beschränkt ist und in einer effektiveren Kommunikation der benötigten Daten mündet. Diese Laufzeitreduktion rechtfertigt die Umformung und zeigt, daß der resultierende Programmtext bei der Ausführung nicht langsamer geworden ist.
Zum Kopieren der entfernten Daten wird von KarHPFn das Einblock-Fließband verwendet. Das entsprechende Forall ist zwar durch ein Prädikat bewacht, doch hängt es nicht von allen Forall-Indizes ab, so daß durch eine geeignete Reihenfolge der Virtualisierungsschleifen immer auf ganze Blöcke zugegriffen werden kann. Getestet wurde auf 2 bis 128 Prozessoren bei einer Problemgröße von 3240000 Punkten. Die Felder wurden in den letzten beiden Dimensionen blockweise verteilt.
Abbildung 5.23 zeigt die Laufzeiten und Beschleunigungen der einzelnen Versionen.

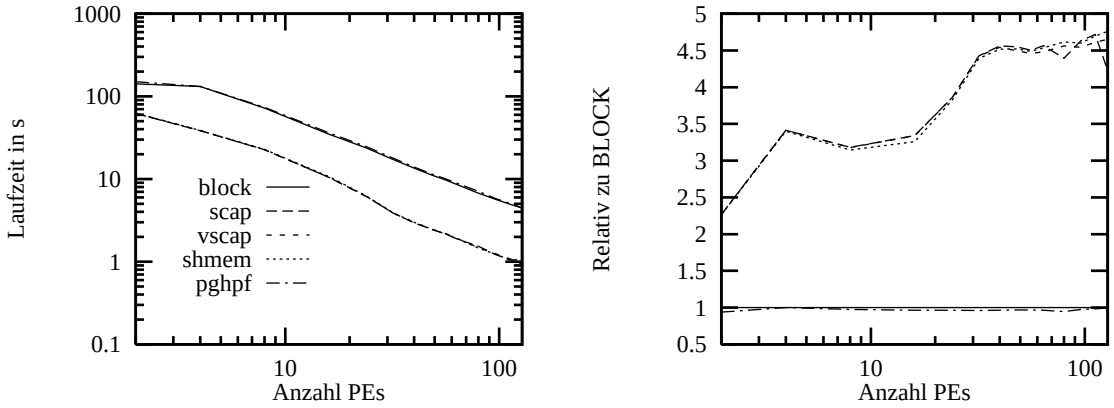



Abbildung 5.23: Laufzeiten und Beschleunigung von *Veltran*

Durch die Laufzeiten sind die verschiedenen Programmversionen in zwei Klassen aufgeteilt. In der einen Klasse liegen BLOCK und PGHPF (obere Kurve) und in der anderen Klasse befinden sich VSCAP, SHMEM und SCAP (untere Kurve). Erstaunlich sind die nahezu identischen Laufzeiten von SCAP und VSCAP, obwohl letztere Programmversion durch die

Vektorbefehle bei wenigen Prozessoren und großen Virtualisierungen einen deutlichen Laufzeitgewinn erreichen sollte.

Man könnte vermuten, daß dieser Effekt durch eine Überlastung des Netzwerkknotens entsteht, auf den alle anderen Prozessoren auf einmal zugreifen. Dies hätte eine Vergrößerung der Netzwerklatenzzeit zur Folge. Durch eine Vergrößerung der Vorladedistanz sollte sich diese Ursache jedoch beheben lassen, doch zeigen Messungen mit 192 vorgeladenen E-Registern und 24 E-Registervektoren (anstatt 128 bzw. 16) keinen Laufzeitvorteil. Eine weitere Ursache könnten zu kurze Speicherbereiche sein, die durch die zu geringen nicht-lokalen Zugriffe den Einsatz der Vektorbefehle aufheben. Doch sind bei 16 Prozessoren 270 nicht-lokale aufeinanderfolgende Datenelemente zu kopieren. Mit dieser Anzahl an Elementen müßte sich jedoch innerhalb der Kommunikation ein meßbarer Kommunikationsvorteil von Faktor 8 ergeben. Messungen zeigen aber, daß die Kommunikation bei einer (1,1)-Vektorstrategie (SCAP) und bei einer ($L$,$L$)-Vektorstrategie (VSCAP) jeweils ca. $40\mu s$ dauert. Cacheeffekte können ebenfalls nicht die Ursache für die unerwartet hohe Laufzeit von VSCAP sein, denn die Kommunikation geht über die E-Register am internen Prozessorcache vorbei. Als Fazit bleibt, daß der Effekt von der Hardware verursacht wird, da zwei voneinander unabhängige Programmversionen VSCAP und SHMEM davon betroffen sind.

Die Beschleunigungen zeigen den Vorteil der Programmversionen mit überlappender Kommunikation gegenüber BLOCK. Sie sind für 2 Prozessoren 2.2-mal schneller. Dieser Faktor erhöht sich für SHMEM auf 4.8 bei 128 Prozessoren. VSCAP erreicht 4.7, und SCAP ist bei 40 Prozessoren 4.6-mal schneller als BLOCK. PGHPF ist bis auf 5% genauso schnell wie BLOCK.

Abbildung 5.24 zeigt die erreichten PRAM-Effizienzen der verschiedenen *Veltran* Versionen.

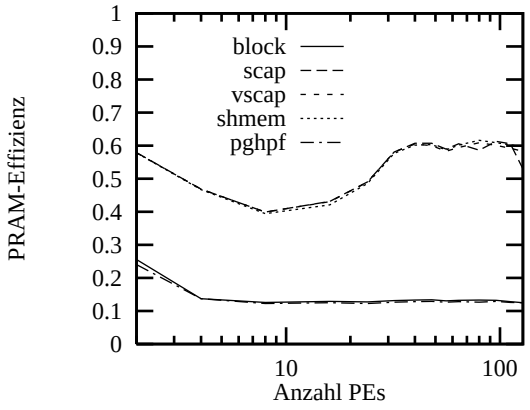



Abbildung 5.24: PRAM-Effizienz von *Veltran*

Das Schaubild zeigt für VSCAP, SHMEM und SCAP eine PRAM-Effizienz von ca. 60% für wenig ($P = 2$) und für viele ($P \geq 32$) Prozessoren (obere Kurve). PGHPF und BLOCK erreichen für $P \geq 4$ Prozessoren ca. 15% (untere Kurve). Der Rückgang der PRAM-Effizienz um bis zu 20% bei den Programmen mit überlappenden Kommunikationsstrategien liegt an der Verteilung der beteiligten Felder, da zuerst in die Dimension parallelisiert wurde, bei der sich der Kommunikationsaufwand nur unmerklich reduziert. Erst ab $P = 32$ reduziert sich der Kommunikationsaufwand und der Anteil der Kommunikation an der Berechnung sinkt entsprechend.

## 5.5 Weitere Tests

Im folgenden werden weitere T3E-spezifische Test wie der in 4.1.4.1 vorgeschlagene Softwaretest (Abschnitt 5.5.1) und die Benutzung der Adreßberechnung durch die Hardware-Zentrifuge (Abschnitt 5.5.2) besprochen. Der letzte Abschnitt 5.5.3 zeigt die Laufzeiten von VSCAP bei der Kommunikation im Falle einer allgemeinen block-zyklischen Verteilung.
Alle Test werden mit dem Testprogramm *Indirekt* ausgeführt (Algorithmenklasse $K2c$), denn dort müssen für jeden nicht-lokalen Datenzugriff die Knotennummer und die knotenlokale Adresse separat berechnet werden, so daß sich dort der Einsatz eines Softwaretests oder der Hardware-Zentrifuge am besten studieren läßt.

### 5.5.1 Softwaretest oder (1,$L$)-Vektorstrategie?

Die Notwendigkeit eines Softwaretestes besteht bei dynamischen nicht-lokalen Zugriffen, wenn zur Übersetzungszeit die Lokalität eines entfernten Datenzugriffs nicht festgestellt werden kann. Diese Entscheidung muß durch die Software geschehen, wenn die Netzwerkschnittstelle keinen Zugriff auf den lokalen Speicher hat. Auf der T3E kann mit Hilfe der E-Register auf den ganzen globalen Adreßraum, also auch auf den lokalen Speicher zugegriffen werden, weshalb dort ein Softwaretest nicht zwingend notwendig ist. Dieser Abschnitt untersucht nun die Frage, ob auf der T3E eine (1,$L$)-Vektorstrategie (Transformationsmuster *Einsammeln*) oder ein Softwaretest mit einer (1,1)-Vektorstrategie besser ist. Eine (1,$L$)-Vektorstrategie hat auf der einen Seite durch den fehlenden Softwaretest weniger Laufzeitaufwand beim Vorladen der Daten. Auf der anderen Seite dauert ein E-Register Vorladebefehl mit einem Vektorzugriff länger als ein entsprechender Zugriff über den Hauptspeicher. Der Nachteil eines Softwaretests sind die im Vergleich zur (1,$L$)-Vektorstrategie ineffizienten einzelnen Zugriffsoperationen.
Der Einsatz eines Softwaretests wird am Beispiel *Indirekt* untersucht, das durch seine dynamischen Zugriffe eine Lokalitätsaussage zur Übersetzungszeit unmöglich macht. Im folgenden werden BLOCK und SCAP ohne und mit Softwaretest getestet (BLOCKSW bzw. SCAPSW). Bei SCAPSW wird das Datenfließband aus 4.1.4.1 eingesetzt. Dazu werden die Ausführungszeiten von VSCAP mit einer (1,$L$)-Vektorstrategie herangezogen. Getestet wurde auf 32 Prozessoren bei einer Variation der lokalen Feldgröße von 1 bis 32768 Elementen. Alle Programme wurden von KarHPFn übersetzt. Für die Generierung der Versionen mit Softwaretest wurde die KarHPFn-Option `-RUNTIME` gewählt, siehe 4.3.3. Die Tests wurden jeweils für eine blockweise und eine zyklische Verteilung der Datenfelder durchgeführt.
Abbildung 5.25 zeigt die Laufzeiten und Beschleunigungen relativ zu BLOCK bei einer blockweisen Verteilung der Datenfelder.
Die Laufzeiten und Beschleunigungen von SCAP, BLOCK und VSCAP sind mit den Daten von Abbildung 5.17 identisch. Neu sind die Kurven von BLOCKSW und SCAPSW. Es fällt auf, daß BLOCKSW keine nennenswerten Laufzeitgewinne duch den Softwaretest erfährt, obwohl bei jedem lokalen Zugriff über den Datencache des Prozessors und nicht über die E-Register zugegriffen wird. Die identischen Laufzeiten liegen zum einen am erhöhten Aufwand für den Softwaretest und zum anderen werden im Mittel nur $\frac{1}{32} * 100\% = 3\%$ der Datenzugriffe über den Cache kopiert. Das bedeutet eine Laufzeiteinsparung von höchstens 3%, die unterhalb der Meßtoleranz liegt. Der Einsatz des Softwaretests bei SCAPSW zeigt, daß sich der Mehraufwand durch den Test im Vergleich zu VSCAP nicht lohnt, da nur 3% der Zugriffe davon betroffen sind, aber *alle* nicht-lokalen Datenelemente mit einelementigen Zugriffsoperationen in den lokalen Speicher kopiert werden müssen. VSCAP ist bei einer lokalen Feldgröße

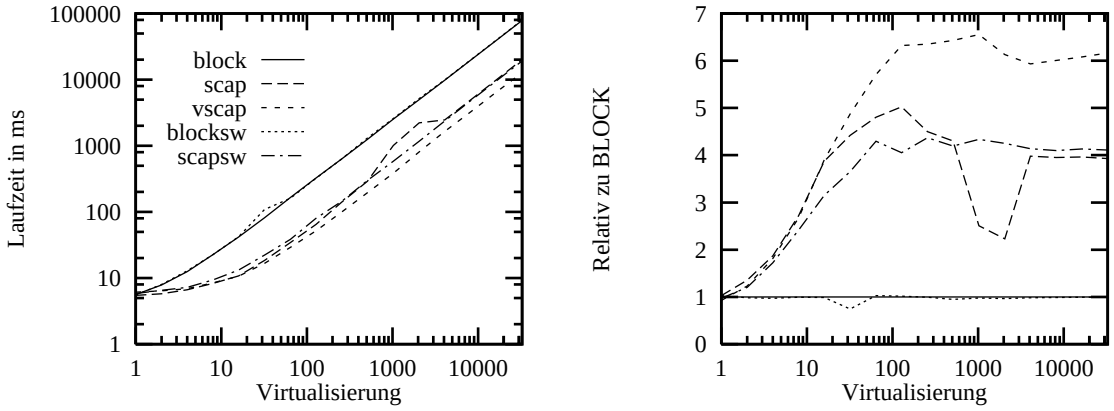



Abbildung 5.25: Laufzeiten und Beschleunigung von *Indirekt* mit und ohne Softwaretest, BLOCK-Verteilung

von $V = 128$ 57% schneller als SCAPSW.
Interessant ist die gleichbleibende Beschleunigung von SCAPSW gegenüber BLOCK, während SCAP bei Virtualisierungen von $512 \leq V \leq 1024$ einen signifikanten Rückgang der zu BLOCK relativen Laufzeit erfährt. Vor diesem Knick ist SCAP etwa 25% schneller und danach etwa 6% langsamer als SCAPSW.
Abbildung 5.26 zeigt mit den Laufzeiten und Beschleunigung bei einer zyklischen Verteilung der Datenfelder ein ähnliches Bild.

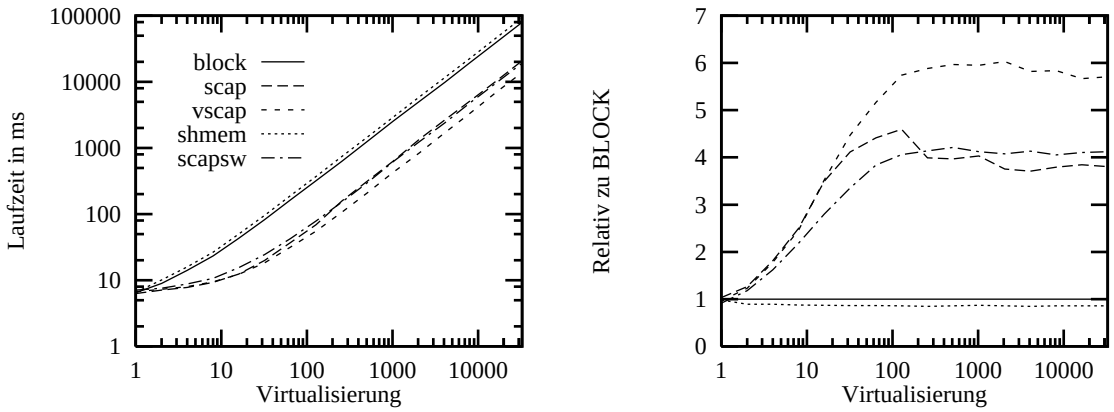



Abbildung 5.26: Laufzeiten und Beschleunigung von *Indirekt* mit und ohne Softwaretest, CYCLIC(1)-Verteilung

Bis zu Virtualisierungen von $V = 128$ ist SCAP maximal 25% schneller als SCAPSW. Danach ist SCAP bis zu 7% langsamer als SCAPSW. VSCAP ist ab einer Virtualisierung von $V \geq 128$ mehr als 50% schneller als SCAPSW.
Die Ursache für den Knick in der Ausführungszeit von SCAP bei einer blockweisen Verteilung ist auf Probleme bei der Berechnung der globalen Quelladresse zurückzuführen, denn bei einer zyklischen Verteilung treten diese Probleme nicht auf.
Als Fazit kann festgestellt werden, daß sich ein Softwaretest bei zufällig verteilten Datenzugriffen als Alternative zur $(1,L)$-Vektorstrategie auf der T3E nicht lohnt, da sich die zwar im Vergleich zu SCAP geringeren aber zu VSCAP doch erheblich zahlreicheren E-Registerzugriffe nicht lohnen.

### 5.5.2 Einsatz der Hardware-Zentrifuge

Mit der in 4.2.3.2 vorgestellten Hardware-Zentrifuge der T3E lassen sich, wie in 4.3.4 beschrieben, in bestimmten Fällen die Adreßberechnungen von der Soft- in die Hardware verlagern. Das verringert den Vorladeaufwand und trägt somit zur Beschleunigung der Kommunikation bei.
Im folgenden soll der Einsatz der Hardware-Zentrifuge am Beispiel *Indirekt* untersucht werden. Es wird erwartet, daß sich die fehlende Adreßumsetzung erheblich auf die Laufzeit der getesteten Programme auswirkt, da durch das dynamische Kommunikationsmuster *jeder* Zugriff von der Adreßberechnung betroffen ist. Für die Test wurde eine blockweise Verteilung der Datenfelder gewählt. Es wurde bei $P = 32$ Prozessoren die Virtualisierung von 1 bis 32768 Elementen variiert. Für den Einsatz der Hardware-Zentrifuge bei einer BLOCK-Verteilung ist eine lokale Problemgröße von $2^n$ notwendig, sonst kann aus dem Feldindex die Knotennummer nicht durch eine Bitmaske selektiert werden. Die Anzahl der Prozessoren ist hingegen beliebig, da nur 1-dimensionale Felder betrachtet werden.
Die BLOCK, SCAP und VSCAP Programme mit Adreßumsetzung durch die Hardware-Zentrifuge wurden mit KarHPFn und der Option `-HWC` übersetzt. Die jeweiligen Versionen heißen BLOCKHWC, SCAPHWC bzw. VSCAPHWC.
Abbildung 5.27 zeigt die Laufzeiten und die Beschleunigungen der einzelnen Programme relativ zu BLOCK.

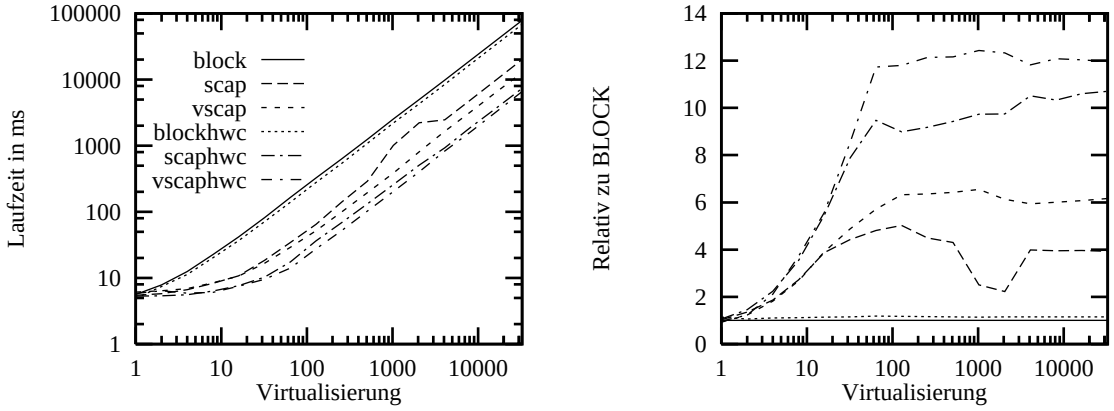



Abbildung 5.27: Laufzeiten und Beschleunigung von *Indirekt* mit und ohne Hardware-Zentrifuge

Wie erwartet erfahren die Programme mit Hardwareunterstützung einen deutlichen Laufzeitgewinn gegenüber ihren Versionen ohne Hardwareunterstützung. So ist VSCAPHWC 1.9-mal, SCAPHWC 2.7-mal und BLOCKHWC 13% schneller. Interessant ist auch, daß SCAPHWC 1.7-mal schneller ist wie VSCAP, obwohl dort für die Zugriffe *keine* Vektorbefehle eingesetzt wurden ((1,1)-Vektorstrategie). Daß SCAPHWC so viel schneller ist als SCAP und VSCAP, liegt an den $V$ zeitaufwendigen Vorladebefehlen, welche die Kommunikationszeit dominieren. Die Zeit für einen SCAP-Vorladebefehl bei 128 nicht-lokalen Datenelementen beträgt $t_v \approx 154ns$, die für SCAPHWC jedoch nur $t_v \approx 38ns$. Das ist eine Laufzeitersparnis beim Vorladen von Faktor 4. Da beide Versionen für einen Zugriff $t_z \approx 53ns$ brauchen, ist SCAPHWC bei 128 nicht-lokalen Zugriffen 2.3-mal schneller. Bei den gezeigten Messungen beträgt der Vorteil zwar nur einen Faktor 1.8, dieser Unterschied ist aber mit dem zusätzlichen Aufwand für die globale Synchronisation der 32 Prozessoren zu erklären. Der Laufzeitvor-

teil bei den Vorladeoperationen ist durch den Wegfall der Division und Modulo-Berechnung bei SCAPHWC bedingt, denn für diese beiden Integeroperationen existieren bei den Alpha-Prozessoren keine speziellen Befehle, so daß sie bei einem statisch nicht berechenbaren Divisor durch eine zeitaufwendige Softwareemulation berechnet werden müssen. Diese Berechnung nimmt, wie oben gesehen, 75% der Vorladezeit bei SCAP in Anspruch.
Weiterhin läßt sich bestätigen, daß der Leistungseinbruch von SCAP bei Virtualisierungen zwischen 128 und 4096 Elementen durch die Adreßberechnungen der blockweisen Verteilung verursacht worden sind, denn SCAPHWC ohne Adreßberechnung erfährt diesen Leistungsverlust nicht.
Für den Vergleich zu den Systemfunktionen SHMEM sei auf die Laufzeiten von *Indirekt* in Abbildung 5.17 hingewiesen. Dort erreichte SHMEM eine mit BLOCK vergleichbare Laufzeit. Verglichen mit der von KarHPFn automatisch generierten VSCAPHWC-Version ist sie 12-mal langsamer.
Dieser Test zeigt deutlich die Notwendigkeit *schneller* Vorladeoperationen. Sie bestimmen den Kommunikationsaufwand und damit die Laufzeit der gesamten Anwendung.

### 5.5.3 CYCLIC(K)-Verteilung

Der letzte Test beschäftigt sich mit der Effizienz der Datenfließbänder für eine allgemeine block-zyklische Verteilung. Das Problem block-zyklischer Verteilungen lag an der statisch nicht berechenbaren lokalen Menge aktiver Prozessoren, die zur Laufzeit mit tabellengesteuerten Methoden vor der Ausführung einer datenparallelen Zuweisung berechnet werden muß (siehe 4.1.4.3). In KarHPFn wurde ein Teil der Methode zur Berechnung block-zyklischer Iterationsräume von [115] implementiert. Für die korrekte Anwendung der Implementierung muß vom Programmierer auf die Bedingung $ggT(\mathrm{Inc}, k) = 1$ geachtet werden. Dabei sind Inc der Abstand der logischen Prozessornummern und $k$ die Blockgröße der Verteilung.
Die Tests wurden an *Indirekt* durchgeführt. Es wurden vier Versionen getestet: BLOCK, SCAP, SHMEM und SCAPHWC. Auf die VSCAP Version wurde verzichtet, da durch die (1,1)-Vektorstrategie SCAP und VSCAP dieselben Datenfließbänder benutzen. Eine vollständig lauffähige PGHPF-Version konnte nicht produziert werden, denn bei größeren Problemen traten E-Registerfehler auf, so daß für PGHPF nur Laufzeitdaten bis zu einer Virtualisierung von $V = 64$ vorliegen. Die erhaltenen Ergebnisse von PGHPF werden in den Grafiken nicht gezeigt. Die Version SCAPHWC berechnet *Indirekt* mit Hilfe der Hardware-Zentrifuge.
Für die Tests wurde eine CYCLIC(8)-Verteilung mit einer Schrittweite von $\mathrm{Inc} = 3$ gewählt. Damit ist $ggT(3, 8) = 1$ und die Zuweisung kann von KarHPFn korrekt übersetzt werden. Das generierte Fließband wurde in 4.1.4.3 vorgestellt.
Abbildung 5.28 zeigt die Laufzeiten und Beschleunigungen bei 32 eingesetzten Prozessoren.
Die Laufzeiten zeigen das erwartete Bild. SCAP und SCAPHWC erreichen einen deutlichen Laufzeitgewinn gegenüber BLOCK und SHMEM. Dieser Gewinn ist durch die Beschleunigungen auf der rechten Seite genauer zu sehen. Für kleine Virtualisierungen $V \leq 16$ sind SCAP und SCAPHWC etwa 33% schneller als BLOCK. Danach erreicht SCAP einen Faktor 4. SCAPHWC ist bei 4096 nicht-lokalen Datenzugriffen gegenüber den anderen Versionen am schnellsten und erreicht im Vergleich zu BLOCK und SCAP einen Laufzeitgewinn von Faktor 9 bzw. 2.3. SHMEM ist im Mittel etwa 10% langsamer als BLOCK.
Bei den Beschleunigungen ist deutlich der zusätzliche Aufwand für die Berechnung der lokalen Adreßfolge zu sehen. Denn für kleine Virtualisierungen $V \leq 8$ erreichen beide SCAP Versionen

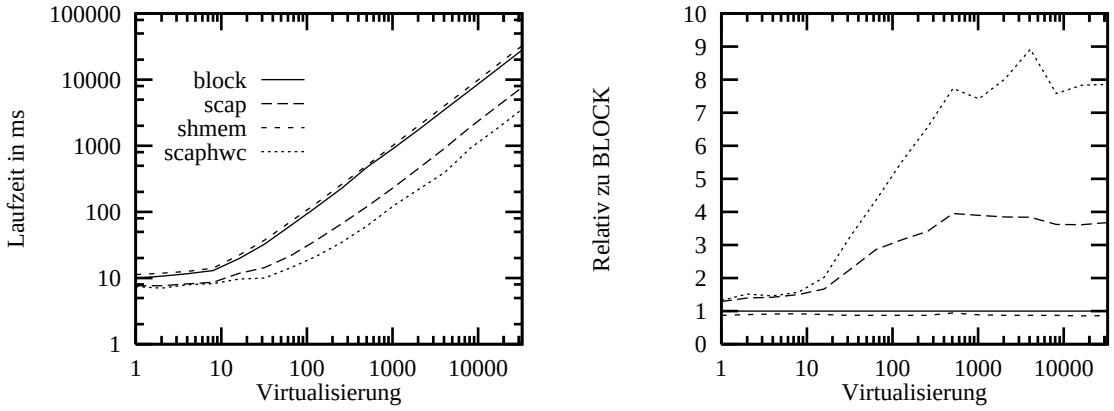



Abbildung 5.28: Laufzeiten und Beschleunigung von *Indirekt* mit CYCLIC(K)-Verteilung

nur einen kleinen konstanten Laufzeitgewinn und der Einsatz der Hardware-Zentrifuge bei SCAPHWC macht sich gegenüber SCAP erst bei größeren Virtualisierungen bezahlt.
PGHPF weist für die erhaltenen Virtualisierungen $V \leq 64$ Laufzeiten wie BLOCK auf. Erstaunlich ist die Tatsache, daß der aufwendigere Vorlade- und Zugriffsaufwand von SCAP gegenüber SCAP mit einer blockweisen Verteilung nur eine Leistungseinbuse von 10% nach sich zieht. Dies liegt an den effizienten Feldzugriffen und der Ersetzung der teuren Modulo-Arithmetik im Fließband von 4.1.4.3 durch eine bedingte Zuweisung.
Das folgende Programmstück wurde durch eine if-Abfrage mit einem zusätzlichen Zähler `BlockCounter` ersetzt:

$I_{Prefetch}$ := $I_{Prefetch}$ + Delta[MOD(PrefetchCounter,k)];
PrefetchCounter := PrefetchCounter + 1;

wird zu

BlockCounter := BlockCounter + 1;
**IF** BlockCounter = k **THEN**
  BlockCounter := 0;
**END IF**
$I_{Prefetch}$ := $I_{Prefetch}$ + Delta[BlockCounter];
PrefetchCounter := PrefetchCounter + 1;

## 5.6 Zusammenfassung

Dieses Kapitel untersuchte das in Kapitel 3 aufgestellte VSCAP-Modell auf der einzigen Parallelrechnerarchitektur, die überlappende Kommunikation im Sinne des VSCAP-Verfahrens ermöglicht. Das Hauptaugenmerk lag nicht nur auf der Bewertung der Effizienzaussagen des VSCAP-Modells gegenüber einer blockierenden und einer Ausführung ohne Vektorbefehle, sondern es wurde auch auf einer breiten Basis von Testprogrammen mit unterschiedlichsten Kommunikationsmustern und -verhalten die Einsatzmöglichkeiten des VSCAP-Verfahrens beleuchtet.

In 5.2 wurden die Aussagen des VSCAP-Modells auf der Cray T3E verifiziert. Es konnte bei einer ($L$,$L$)-Vektorstrategie eine $L$-fache Reduktion der Kommunikationszeit beim Einsatz von VSCAP gegenüber SCAP nachgewiesen werden. Es wurde ebenfalls die Latenzzeitreduktion von VSCAP gegenüber BLOCK bestätigt, die mit 96% fast dem theoretischen Maximum von 99% entsprach. Bei der (1,$L$)-Vektorstrategie konnten für die Latenzzeitverbergung ähnliche Werte gemessen werden. So konnte VSCAP 100% der Latenzzeit verbergen und war damit im vom Modell berechneten Bereich. Der Vorteil der Vektorbefehle bei einer (1,$L$)-Vektorstrategie konnten nicht ohne weiteres auf der T3E bestätigt werden. Diese Abweichung ist auf das vorgestellte Modell zurückzuführen, das durch eine Vereinfachung des Parameterraums keine für die T3E korrekte Modellierung zuließ. Trotz oder gerade wegen dieser Schwierigkeiten ermöglichte der Vergleich des Modells und der Maschine einen genaueren Einblick in die Arbeitsweise von VSCAP auf der T3E.
Insgesamt konnten die Laufzeitabschätzungen des Modells mit den automatisch generierten Fließbändern von KarHPFn nachgewiesen werden, was die These 1 von 1.3 nicht nur bestätigt, sondern noch erweitert, da der Nachweis mit automatisch generierten Kommunikationsfließbändern geschah.
In 5.3 wurde die Leistung der VSCAP-Programme von KarHPFn mit der Kommunikationsbibliothek der T3E und dem kommerziell erhältlichen HPF-Übersetzer von Portland Group verglichen. Der Vergleich zur Kommunikationsbibliotek SHMEM sollte die Effizienz von VSCAP zur bestmöglichst erreichbaren Kommunikationsleistung zeigen. VSCAP ist bei statischen Kommunikationsmustern mit der Leistung von SHMEM vergleichbar. Es konnten sogar Schwächen in der `shmem_iget`-Implementation gezeigt werden, die erst bei mehr als 128 nicht-lokalen Zugriffen ausgeglichen werden (siehe 5.3.4 und 5.3.6). Bei dynamischen Kommunikationsmustern ist die Leistungsfähigkeit von SHMEM im Bereich von BLOCK anzusiedeln, da es keine Unterstützung für diese Art der Kommunikation gibt. In diesem Fall ist sogar mit einem Laufzeitverlust von einem Faktor 6.6 gegenüber VSCAP zu rechnen. Beim Einsatz der Hardware-Zentrifuge, der allerdings nur auf spezielle Datengrößen beschränkt ist, beträgt der Verlust sogar einen Faktor 12.
Der Vergleich zu PGHPF zeigte die Möglichkeiten einer automatischen Übersetzung von VSCAP. Bis auf einige Ausnahmen war das Laufzeitverhalten von PGHPF mit dem von BLOCK vergleichbar. VSCAP war beim Test auf 128 Prozessoren bei *FIRE* knapp 3-mal, bei *Veltran* etwa 4.5-mal und bei *PDE1* mehr als 5-mal schneller als PGHPF. Beide Übersetzer, KarHPFn und PGHPF, bearbeiteten die gleichen HPF-Quellen.
Die Ergebnisse des Vergleichs von VSCAP zu SHMEM und PGHPF erbringt den Nachweis für eine automatische, effiziente und transparente Transformation von datenparallelen Programmen in ausführbare Programme mit Kommunikationsfließbändern von VSCAP und zeigt damit die Gültigkeit von These 2. Bis auf zwei Ausnahmen (LL13 und LL14) wurden *alle* betrachteten Programme ohne weitere Hilfestellung des Programmieres übersetzt. Die Ausnahmen LL13 und LL14 begründen sich auf eine fehlende Unterstützung des gesamten HPF-Standards und nicht auf eine Schwäche der verwendeten Methodik.

# Kapitel 6

# Zusammenfassung und Ausblick

## 6.1 Zusammenfassung

Das Ziel dieser Arbeit war die Verbergung der Latenzzeit bei der Ausführung datenparalleler Programme. Dazu wurde das bekannte SCAP-Modell verwendet und um Vektorbefehle erweitert, so daß sich eine Familie von Vektorstrategien mit SCAP als Spezialfall entwickelte. Die intuitiv verständliche Verringerung der Kommunikationszeit von VSCAP um einen Faktor gleich der Vektorlänge $L$ gegenüber SCAP konnte modelliert und auf der T3E durch Laufzeitmessungen nachgewiesen werden. Durch Messungen konnten ebenfalls die vom Modell abgeschätzten Einsparungen der Latenzzeit von VSCAP bestätigt werden, was These 1 aus 1.3 bestätigt, daß *mit dem maschinenunabhängigen analytischen Netzwerkmodell von VSCAP in einer gegebenen Parallelrechnerarchitektur der Kommunikationsaufwand einer datenparallelen Schleife abgeschätzt werden kann.*

Um die Latenzzeitverbergung von VSCAP dem Programmierer zugänglich zu machen, wurde der Prototypübersetzer KarHPFn implementiert, der HPF nach F90 übersetzt und für die Kommunikation die VSCAP-Datenfließbänder einsetzt. Die Laufzeitmessungen von 25 Testprogrammen zeigte die Leistungsstärke von VSCAP im Vergleich zur hochoptimierten Kommunikationsbibliothek der T3E und zum kommerziellen HPF-Übersetzer von Portland Group. Das von KarHPFn erzeugte VSCAP ist mit der Leistung der Kommunikationsbibliothek nicht nur vergleichbar, sondern bei dynamischen Kommunikationsmustern zeigt sich mit einer bis zu 6-mal schnelleren Ausführungszeit der eindeutige Vorteil der KarHPFn-Programme. Gegenüber PGHPF war VSCAP schon bei kleinen Problemgrößen mehr als 3-mal so schnell und konnte bei großem Kommunikationsaufwand eine Beschleunigung von mehr als Faktor 15 erreichen. Diese Laufzeiteinsparungen wurden in einer für den Programmierer transparenten Transformation erzielt, denn PGHPF und KarHPFn übersetzten identische HPF-Quellen. Damit konnte der Nachweis für These 2 erbracht werden, daß *die Transformation einer datenparallelen Schleife in ein softwaregesteuertes Datenfließband des VSCAP-Verfahrens durch einen Übersetzer geschehen kann und daß mit dieser Programmtransformation die Kommunikationskosten einer datenparallelen Anwendung ohne zusätzliche Intervention des Programmierers gesenkt werden können.*

## 6.2 Ausblick

Das VSCAP-Verfahren ist nicht auf die Cray T3E beschränkt, obwohl es dort in seiner ursprünglichen Form angewendet werden kann. Dieser Abschnitt bespricht Probleme und Vorteile, die sich bei einer Portierung von VSCAP auf verschiedene Parallelrechnerarchitekturen ergeben. Der letzte Abschnitt zeigt eine Erweiterung des VSCAP-Verfahrens um entfernte Schreibzugriffe.

### 6.2.1 VSCAP auf Bündel von Arbeitsplatzrechnern

Der Einsatz von VSCAP stößt in nachrichtengekoppelten Bündel von Arbeitsplatzrechnern auf einige Probleme, die nachfolgend genannt werden. Betrachtet man jedoch Arbeitsplatzrechner, die durch einen virtuellen gemeinsamem Speicher (*virtual shared memory*) verbunden sind, so erschließt sich ein Umfeld, das die effiziente Ausführung von VSCAP ermöglicht.
Das erste Problem auf nachrichtengekoppelten Bündel ergibt sich aus der kurzen Vektorlänge ($L \leq 32$) von VSCAP, mit der im allgemeinen keine effiziente Kommunikation durchgeführt werden kann, da hierfür DMA-Operationen mit großen Anlaufzeiten eingesetzt werden, die sich erst für Blockgrößen von mehreren Kilobytes lohnen. Prinzipiell ist das Problem der effizienten Kommunikation kleiner Blockgrößen jedoch gelöst, wie die Kommunikationsbibliothek von ParaStation [18] zeigt.
Die nächste Schwierigkeit liegt an der Form des Speicherbereichs, der mit DMA-Operationen versendet oder gelesen werden soll. Dieser Bereich muß nämlich zusammenhängend sein. Damit können bisher keine Datenmengen kommuniziert werden, die durch eine Anfangsadresse und eine konstante Schrittweite zwischen den einzelnen Datenelementen beschrieben sind, ohne daß zusätzliche Kopieroperationen zum Ein- und Auspacken oder mehrmaliges Aufsetzen der DMA-Operation benötig werden. Eine Lösung für dieses Problem ist bisher nicht bekannt.
Erweitert man diese Thematik auf dynamische Zugriffe, die zufällig über den verteilten Speicher einer Applikation verstreut sind, so ergibt sich noch das Problem der Umrechnung von logischen zu physikalischen Adressen. Diese Transformation kann sehr zeitaufwendig werden (im Bereich von hundert bis tausend Prozessortakten), da sie nur im Betriebsystemkern ausgeführt werden kann, der die entsprechenden Speichertabellen exklusiv verwaltet. Der Einsatz einer $(1,L)$-Vektorstrategie verspricht in diesem Fall keinen nennenswerten Laufzeitgewinn zum Beispiel gegenüber einer Ausführung nach dem *Inspector-Executor*-Modell, da die Adreßberechnungen innerhalb der Vorladeschleife etwa so lange dauern würden wie der Aufbau der Kommunikationsstruktur im *Inspector* und die anschließende Kommunikation mittels DMA-Operationen innerhalb des *Executors*.
Die Probleme der Adreßumsetzung in Arbeitsplatzrechnern kommen durch die virtuelle Speicherverwaltung des Betriebssystems zustande, das Speicherseiten für die Anwendung transparent austauschen kann, was zu einer veränderten Zuordnung von logischer zu physikalischer Adresse führt. Dieser Austausch ist dann nötig, wenn mehrere Prozesse gleichzeitig ausgeführt werden müssen. Da in Parallelrechnern häufig auf jedem Knoten nur ein Prozess ausgeführt wird, kann auf die Auslagerung von Speicherseiten verzichtet und die damit einfachere Adreßumsetzung von der Kommunikationshardware durchgeführt werden.
Das bisher[1] einzige System, das dieses Problem der Adreßumsetzung in Bündel von Arbeitsplatzrechnern effizient löst, ist der *SCI-* (*Scalable Coherent Interface*) Standard

[1] Stand November 1999

(ANSI/IEEE Std 1596), der aus der Sicht einer Applikation ein System mit virtuellem gemeinsamem Speicher bereitstellt. Die Adreßumsetzung findet in der Hardware statt. Kommunikationsaufträge werden ähnlich wie bei der Cray T3E durch Lese- und Schreibzugriffe in den Ein/Ausgabebereich des Prozessors abgesetzt. Bisher blockiert die Kommunikation den Prozessor jedoch für die Dauer der Netzwerklatenz, so daß in SCI keine Überlappung von Kommunikationsaufträgen möglich ist. Es gibt jedoch Bestrebungen, daß nicht blockierende Lesevorgänge effizient durchgeführt werden können, was den Einsatz von VSCAP mit SCI in nächster Zeit ermöglichen würde.
Für eine Implementierung von VSCAP auf Bündel von Arbeitsplatzrechnern würde man VSCAP aber aus Effizienzgründen um entfernte Schreibzugriffe erweitern und in seiner ursprünglichen Form nur bei dynamischen Kommunikationsmustern einsetzen (siehe Abschnitt 6.2.3).

### 6.2.2 VSCAP in Architekturen mit gemeinsamem Speicher

Der Vorteil des hier vorgestellten VSCAP-Verfahrens beim Einsatz in Architekturen mit gemeinsamem Speicher liegt an der Funktionalität des Vorladepuffers. Denn seine Einträge werden für den jeweiligen Prozessor exklusiv vorgeladen, und nachfolgende Schreibzugriffe auf diese Daten werden die vorgeladenen Elemente nicht verändern. Mit dem Einsatz eines Vorladepuffers können alle Nachteile, die aus dem Cachekohärenzprotokoll entstehen, beseitigt werden (siehe auch 2.3.1.2).
Die Ausführung von VSCAP durch einen softwareemulierten Vorladepuffer ist nicht möglich, denn die Vorladebefehle in Architekturen mit gemeinsamem Speicher verwalten die vorgeladenen Datenelemente unter ihrer ursprünglichen Adresse. Damit erklärt das Cachekohärenzprotokoll bei einem Schreibzugriff auf ein bereits vorgeladenes Datum dieses wieder für ungültig und der nächste Zugriff muß wieder den zuletzt geschriebenen Wert lesen. Dies ist aber das Cachekohärenzproblem, das durch den Vorladepuffer gerade vermieden werden sollte. Wenn der Vorlademechanismus jedoch das vorgeladene Datenelement unter einer anderen als der ursprünglichen Adresse verwalten könnte, so würden spätere Schreibzugriffe ignoriert und die Applikation bestimmte über die Dauer der Gültigkeit des Datums.
Eine weitere Alternative sind Cachezeilen, die der Prozessor als invariant gegenüber dem Cachekohärenzprotokoll deklarieren kann. Damit lädt der Prozessor ganz normal seine Daten in seinen Cache vor und teilt dem Protokoll mit, daß nachfolgende Schreibzugriffe anderer Prozessoren auf diese Zeile ignoriert werden sollen. Damit stehen die Daten exklusiv zur Verfügung und können durch die Speicherzugriffe der anderen Prozessoren nicht mehr verdrängt werden.

### 6.2.3 Kombination von Get- und Put-Semantik

KARHPFN arbeitet ausschließlich mit einer *Get*-Semantik, d.h. jeder Prozessor lädt sich die Daten vor, die er später für die Berechnung benötigt. Der dazu alternative Ansatz ist die *Put*-Semantik, bei der ein Prozessor die Daten in den Speicher des Prozessors schreibt, der sie für seine spätere Berechnung braucht. Es stellt sich die Frage, inwieweit sich die Ausführungszeiten von VSCAP weiter reduzieren lassen, wenn bei statischen Kommunikationsmustern entfernte (einseitige) Schreibzugriffe eingesetzt werden. Der Vorteil der *Put*-Methode ist die verringerte Netzwerklast, denn die Daten müssen nicht wie bei der *Get*-Semantik zweimal durch das Netz propagiert werden (Anfrage und Antwort), sondern es

genügt eine Traversierung. Mit der *Put*-Semantik könnten auch bei großen Datenblöcken effiziente DMA-Operationen eingesetzt werden, wie sie in Bündel von Arbeitsplatzrechnern Verwendung finden.
Prinzipiell ist die *Put*-Semantik eine mögliche Ergänzung des VSCAP-Verfahrens, denn bei dynamischen Kommunikationsmustern ist die Kommunikation mit der *Get*-Semantik nicht ohne zusätzlichen Kommunikationsaufwand zu ersetzen.

# Anhang A

# Transformation des Veltran-Operators

## A.1 Original HPF-Quelltext

Das folgende Programmstück zeigt den Originalquelltext des Berechnungskerns des Veltran-Operators.

```
      SUBROUTINE velsimpI( conj,add,t0,dt,x0,dx,s0,ds,
                           nt,nx,ns,np, modl, data,vel,type)
      INTEGER it,ix,is,conj,add,nt,nx,ns,np,iz,nz,vel,type
      REAL x,s,sx,t,tt,z, dz,wt,t0,dt,x0,dx,s0,ds
      REAL modl(nt,ns,np),data(nt,nx,np)
      INTEGER i,k,l

!HPF$ PROCESSORS SQUARE($PROC,number_of_processors()/$PROC)
!HPF$ ALIGN modl(*,*,l) WITH data (*,*,l)
!HPF$ DISTRIBUTE data (*,block,block) ONTO SQUARE

      nz = nt
      dz= dt                  ! z is travel time depth
!HPF$ INDEPENDENT,NEW(is,s,ix,x,iz,z,tt,sx,it,wt)
      DO ip= 1, np
!HPF$ INDEPENDENT,NEW(s,ix,x,iz,z,tt,sx,it,wt)
         DO is= 1, ns
            s = s0 + (is-1) * ds
            IF(vel.eq.1) THEN
               s = 1./s
            END IF
            DO ix= 1, nx
               x = x0 + (ix-1) * dx
               sx =  abs( s * x)
               DO iz= 1, nz
                  z = t0 + (iz-1) * dz
```

```
               tt  = z * z + sx * sx
               IF ( tt .gt. 0 ) THEN
                  t = sqrt ( tt )
               ELSE
                  t = 0
               END IF
               it = 1.5 + (t - t0) / dt
               IF ( it .le. nt) THEN
                  wt= 1.
                  IF ( type .eq. 1) THEN
                     wt= 0.
                     IF( t .ne. 0.)  THEN
                        wt= (z/t) / sqrt( t)
                     END IF
                  END IF
                  modl(iz,is,ip) = modl(iz,is,ip) +
                                       data(it,ix,ip) * sx * wt
               END IF
            END DO
         END DO
      END DO
   END DO
END DO
END
```

## A.2 HPF-Text für KarHPFn

Dieses Programmstück zeigt den transformierten Quelltext, der KarHPFn und PGHPF als Eingabe diente und mit dem die Testergebnisse von 5.4.3 gewonnen wurden.

```
      SUBROUTINE velsimpI( conj,add,t0,dt,x0,dx,s0,ds,
                           nt,nx,ns,np, modl, data,vel,type)
      INTEGER ix,is,conj,add, nt,nx,ns,np,iz,nz,vel,type,ip
      INTEGER i,j,k,l,it(1:ns)
      REAL x,z,dz,t0,dt,x0,dx,s0,ds,modl(nt,ns,np),data(nt,nx,np)
      REAL s(1:ns),sx(1:ns),t(1:ns),tt(1:ns),wt(1:ns)
      REAL remotedata ( ns,np )

!HPF$ PROCESSORS SQUARE($PROC,NUMBER_OF_PROCESSORS()/$PROC)
!HPF$ DISTRIBUTE data (*,block,block) ONTO SQUARE
!HPF$ ALIGN modl(*,k,l) WITH data (*,k,l)
!HPF$ ALIGN remotedata ( i,l ) WITH data ( *,i,l )
!HPF$ ALIGN s(i) WITH data ( *,i,* )
!HPF$ ALIGN sx(i) WITH data ( *,i,* )
!HPF$ ALIGN t(i) WITH data ( *,i,* )
!HPF$ ALIGN tt(i) WITH data ( *,i,* )
!HPF$ ALIGN it(i) WITH data ( *,i,* )
!HPF$ ALIGN wt(i) WITH data ( *,i,* )

      nz = nt
      dz = dt                  ! z is travel time depth
      IF ( vel .eq. 1 ) THEN
         FORALL ( is=1:ns )
            s(is) = 1/(s0 + (is-1) * ds)
         END FORALL
      ELSE
         FORALL ( is=1:ns )
            s(is) = s0 + (is-1) * ds
         END FORALL
      END IF
      DO ix= 1, nx
         x = x0 + (ix-1) * dx
         FORALL ( is=1:ns )
            sx(is) =  abs( s(is) * x)
         END FORALL
         DO iz= 1, nz
            z = t0 + (iz-1) * dz
            FORALL ( is=1:ns )
               tt(is)  = z * z + sx(is) * sx(is)
            END FORALL
            FORALL ( is=1:ns, tt(is) .gt. 0 )
```

```
         t(is) = sqrt ( tt(is) )
      END FORALL
      FORALL ( is=1:ns, tt(is) .le. 0 )
         t(is) = 0
      END FORALL
      FORALL ( is=1:ns )
         it(is) = 1.5 + (t(is) - t0) / dt
      END FORALL
      wt(:) = 1
      IF ( type .eq. 1) THEN
         wt(:) = 0.
         FORALL ( is=1:ns, (t(is) .ne. 0) )
            wt(is) = (z/t(is)) / sqrt( t(is))
         END FORALL
      END IF
      FORALL (is=1:ns, ip= 1:np, it(is) <= nt )
         remotedata ( is, ip ) = data(it(is),ix,ip)
      END FORALL
      FORALL (is=1:ns, ip= 1:np, it(is) <= nt )
         modl(iz,is,ip) = modl(iz,is,ip) +
                              remotedata(is,ip) * sx(is) * wt(is)
      END FORALL
   END DO
END DO
END
```

# Literaturverzeichnis


[1] F. Abolhassan, J. Keller, and W. J. Paul. On the cost-effectiveness and realization of the theoretical PRAM model. Technical Report 09/91, University of Saarland in Saarbrucken, Comp. Sci. Dept., March 1991.

[2] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, pages 2–13, June 1995.

[3] Anant Agarwal, Geoffrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Daniel Nussbaum, Mike Parkin, and Donald Yeung. The MIT alewife machine : A large-scale distributed-memory multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic, 1991.

[4] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeoung, Geoffrey D'Souza, and M. Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[5] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir. SP2 system architecture. *IBM Systems Journal*, 34(2):152–184, 1995.

[6] A. Aggarwal and A. K. Chandra. Communication complexity of PRAMs. In Timo Lepistö and Arto Salomaa, editors, *Proceedings of the 15th International Colloquium on Automata, Languages and Programming*, volume 317 of *LNCS*, pages 1–17, Berlin, July 1988. Springer.

[7] Gail A. Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton J. Smith. Exploiting heterogeneous parallelism on a multi-threaded multiprocessor. In *6th ACM International Conference on Supercomputing*, pages 188–197, Washington, D.C., July 1992.

[8] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, pages 1–6, 1990.

[9] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[10] Remzi H. Arpaci, David E. Culler, and Arvind Krishnamurthy. Empirical evaluation of the CRAY-T3D: A compiler perspective. Technical report, Computer Science Devision, University of California, Berkeley, 1994.

[11] P. Bach, M. Braun, A. Formella, J. Friedrich, Th. Grün, and C. Lichtenau. Building the 4 processor SB-PRAM prototype. In *30th Hawaii International Conference on System Sciences*, pages 14–23, January 1997.

[12] J.L. Baer and T.F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing '91*, pages 176–186, 1991.

[13] S. Benkner, K. Sanjari, V. Sipkova, and B. Velkov. Parallelizing Irregular Applications with the Vienna HPF+ Compiler VFC. In *International Conference on High Performance Computing and Networking (HPCN '98)*, Amsterdam, April 1998. Springer Verlag.

[14] Siegfried Benkner. Handling block-cyclic distributed arrays in Vienna Fortran 90. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Limassol, Cyprus, June 1995.

[15] Siegfried Benkner, Barbara M. Chapman, and Hans P. Zima. Vienna Fortran 90. In *Scalable High Performance Computing Conference*, pages 51–59, Williamsburg, Va., April 1992.

[16] Siegfried Benkner, Piyush Mehrotra, John Van Rosendale, and Hans Zima. High-level management of communication schedules in HPF-like languages. Technical Report TR-97-46, Institute for Computer Applications in Science and Engineering, September 1997.

[17] J. M. Blum, T. M. Warschko, and W. F. Tichy. PSPVM: Implementing PVM on a high-speed interconnect for workstation clusters. In *Proceedings of the Third Euro PVM Users' Group Meeting*, pages 235–242, München, October 1996.

[18] J. M. Blum, T. M. Warschko, and W. F. Tichy. PULC: ParaStation User-Level Communication. Design and Overview. *Lecture Notes in Computer Science*, 1388:498–517, 1998.

[19] Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young. PGHPF – an optimizing High Performance Fortran compiler for distributed memory machines. *Scientific Programming*, 6(1):29–40, Spring 1997.

[20] Z. Bozkus, S. Ranka, G. Fox, and A. Choudhary. Performance Comparison of the CM-5 and Intel Touchstone Delta for Data Parallel Operations. In *Symposium on Parallel and Distributed Systems (SPDP '93)*, pages 307–311, Los Alamitos, December 1994. IEEE Computer Society Press.

[21] Zeki Bozkus. *Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers.* PhD thesis, Syracuse University, June 1995.

[22] Joe Brandenburg. Technology advances in the Intel Paragon system. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, page 182, Velen, Germany, June 30–July 2, 1993. SIGACT and SIGARCH.

[23] T. Brandes and F. Desprez. Implementing pipelined computation and communication in an HPF compiler. *Lecture Notes in Computer Science*, 1123:459–462, August 1996.

[24] Thomas Brandes. Adaptor: A compilation system for data parallel fortran programs. Technical report, German National Center for Computer Science (GMD), St. Augustin, Germany, 1994. ftp://ftp.gmd.de/GMD/adaptor/docs/adaptor.ps.

[25] P. Brezany, V. Sipkova, B. Chapman, and R. Greimel. Automatic parallelization of the AVL FIRE benchmark for a distributed-memory system. *Lecture Notes in Computer Science*, 1041:50–60, 1996.

[26] Ralph Buttler and Ewing Lusk. User's guide to the p4 parallel programming system. Technical Report ANL-92/17, Argonne National Laboratory, October 1992.

[27] Barbara M. Chapman, Piyush Mehrotra, and Hans P. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.

[28] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and S.-H. Teng. Generating local addresses and communication sets for data-parallel programs. *ACM SIGPLAN Notices*, 28(7):149–158, July 1993.

[29] Thomas Cheatham. Models, languages, and compiler technology for high performance computers. In Igor Prívara, Branislav Rovan, and Peter Ruzicka, editors, *Mathematical Foundations of Computer Science*, volume 841 of *Lecture Notes in Computer Science*, pages 3–26, Kosice, Slovakia, August 1994. Springer.

[30] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, Boston, Massachusetts, October 1992. Also available as U. Washington CS TR 92-06-03.

[31] Chi-Hung Chi and Chin-Ming Cheung. Hardware-driven prefetching for pointer data references. In *12th International Conference on Supercomputing*, Melbourne, 1998.

[32] L. Clarke, I. Glendinning, and R. Hempel. The MPI Message Passing Interface Standard. In K. M. Decker and R. M. Rehmann, editors, *Programming environments for massively parallel distributed systems: working conference of the IFIP WG10.3, April 25–29, 1994, Ascona, Italy*, pages 213–218, Boston, MA, USA, 1994. Birkhäuser.

[33] R. Cole and O. Zajicek. The APRAM: Incorporating Asynchrony into the PRAM Model. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, Santa Fe, NM, June 1989. ACM Press.

[34] Intel Coporation. Paragon XS/P Product Overview, 1991.

[35] Digital Equipment Corporation. *Alpha Architecture Handbook*. Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, 1994.

[36] David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus Erik Schauser, Ramesh Subramonian, and Thorsten von Eicken. Logp: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, November 1996.

[37] Carole Dulong. The IA-64 Architecture at Work. *IEEE Computer*, pages 24–32, July 1998.

[38] Jörn Eisenbiegler. *Optimierung von SIMD-Programmen auf verteilten Systemen.* PhD thesis, School of Computer Science, Universität Karlsruhe, December 1998.

[39] Amr Fahmy and Abdelsalam Heddaya. BSPk: Low overhead communication constructs and logical barriers for bulk synchronous parallel programming. *Bulletin of the IEEE Technical Committee on Operating Systems and Application Environments (TCOS)*, 8(2):27–32, Summer 1996.

[40] J. T. Feo. An analysis of the computational and parallel complexity of the Livermore loops. *Parallel Computing*, 7(2):163–185, June 1988.

[41] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[42] High Performance Fortran Forum. *High Performance Fortran Language Specification 1.1*, November 1994.

[43] High Performance Fortran Forum. *High Performance Fortran Language Specification 2.0*, January 1997.

[44] Ian Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1994.

[45] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. Fortran D language specification. Computer Science Department Report CRPC-TR90079, Rice University, Houston, TX, USA, December 1990.

[46] G. A. Geist, A. Benguelim, J. Dongarra, J. Jiang, R. Manchek, and V. Sunderam. *PVM 3.0 "User's Guide" and "Reference Manual".* Oak Ridge National Laboratory, 1993.

[47] M. Gerndt. Updating distributed variables in local computations. *Concurrency, Practice and Experience*, 2(3):171–194, September 1990.

[48] P. B. Gibbons. A more practical PRAM model. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, Santa Fe, NM, June 1989. ACM Press.

[49] James R. Goodman, Jian tu Hsieh, Koujuchj Liou, Andrew R. Pleszkun, P. B. Schechter, and Honesty C. Young. PIPE: A VLSI decoupled architecture. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 20–27, Boston, Massachusetts, June 17–19, 1985. IEEE Computer Society TCA and ACM SIGARCH.

[50] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessor with memory hierarchies. In *Proceedings 1990 International Conference on Supercomputing*, pages 354–368, Amsterdam, June11–15 1990.

[51] J. Grosch. *Puma - A Generator for the Transformation of Attributed Trees.* GMD, Forschungsstelle an der Universität Karlsruhe, 1991.

[52] Josef Grosch and Helmut Emmelmann. A tool box for compiler construction. In Dieter Hammer, editor, *Compiler Compilers, Third International Workshop on Compiler Construction*, volume 477 of *Lecture Notes in Computer Science*, pages 106–116, Schwerin, Germany, 22–26 October 1990. Springer, 1991.

[53] Andreas Grävinghoff and Jörg Keller. How to emulate fine-grained multithreading. In *Proceedings of the 2nd IASTED Conference on Parallel and Distributed Computing and Networks*. ACTA press, 1998.

[54] Andreas Grävinghoff and Jörg Keller. Virtual duplex systems in embedded environments. Technical report, FernUniversität-GHS Hagen, FB Informatik, 1998.

[55] Andreas Grävinghoff and Jörg Keller. Fine-grained multithreading on the CRAY T3E. In *High-Performance Computing in Science and Engineering 1999*, pages 447–456. Springer, 2000.

[56] S. K. S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. In N. Berra P. Bruce, A. Choudhary, editor, *Proceedings of the 1993 International Conference on Parallel Processing. Volume 2: Software*, page 301, Syracuse, NY, August 1993. CRC Press.

[57] P. J. Hatcher, M. J. Quinn, R. J. Anderso, A. J. Lapadula, B. K. Seevers, and A. F. Bennett. Architecture-independent scientific programming in dataparallel C: three case studies. In *Proceedings of Supercomputing '91*, pages 208–217, Albuquerque, New Mexico, November 1991. IEEE Computer Society Press.

[58] Philip J. Hatcher, Michael J. Quinn, Anthony J. Lapadula, Bradley K. Seevers, Ray J. Anderson, and Robert R. Jones. Data-parallel programming on MIMD computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):377–383, July 1991.

[59] R. Hempel, H.-C. Hoppe, U. Keller, and W. Krotz. *PARMACS V6.0 Specification*. Pallas GmbH, Brühl, Germany, 1993.

[60] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, INC., San Mateo, California, 1990.

[61] Christian G. Herter. *Emulation eines PRAM Shared Memory Modells mit Triton/1*. PhD thesis, School of Computer Science, Universität Karlsruhe, 1995.

[62] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[63] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling fortran d for mimd distributed memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[64] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[65] R. W. Hockney and E. A. Carmona. Comparison of communications on the Intel iPSC/860 and Touchstone Delta (short communication). *Parallel Computing*, 18(9):1067–1072, September 1992.

[66] IBM. IBM J. Research and Development, Speciel Issue on RISC System/6000, January 1990.

[67] Motorola Inc. *MCS88100: RISC Mircoprocessor User's Manual.* Motorola Inc., 1988.

[68] INMOS Limited. OCCAM programming manual. Prentice Hall, 1984.

[69] Intel Corporation, Santa Clara, CA. iPSC system overview, January 1986.

[70] Intel Corporation, Santa Clara, CA. i860(TM) microprocessor programmer's reference manual, 1989.

[71] Intel Corporation, Santa Clara, CA. iPSC/2 and iPSC/860 user's guide, June 1990.

[72] Matthias Jacob, Michael Philippsen, and Martin Karrenbach. Large-scale parallel geophysical algorithms in Java: a feasibility study. *Concurrency: Practice and Experience*, 10(11–13):1143–1153, September 1998. Special Issue: Java for High-performance Network Computing.

[73] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990.

[74] T. Ungerer K. Bittnar, W. Grünewald. Entwurf einer vielfädigen prozessorarchitektur zum einsatz in distributed-shared-memory-systemen. In *PARS-Workshop*, pages 85–94, Potsdam, 19.-20. September 1994.

[75] V. Karamcheti and A. A. Chien. A comparison of architectural support for messaging on the TMC CM-5 and Cray T3D. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 298–307, New York, June 22–24 1995. ACM Press.

[76] K. Kennedy, N. Nedeljkovic, and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. *ACM SIGPLAN Notices*, 30(8):102–111, August 1995.

[77] Alexander C. Klaiber and Henry M. Levy. An architecture for software-controlled data prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 43–53, May 1991.

[78] Charles Koelbel and Piyush Mehrotra. Supporting shared data structures on distributed memory architectures. In *Proceedings of the 2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPOPP*, pages 177–186, March 1990.

[79] J. T. Kuehn and B. J. Smith. The Horizon supercomputing system: architecture and software. In IEEE, editor, *Proceedings, Supercomputing '88: November 14–18, 1988, Orlando, Florida*, volume 1, pages 28–34, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1988. IEEE Computer Society Press.

[80] S. Leung and J. Zahorjan. Improving the performance of runtime parallelization. *ACM SIGPLAN Notices*, 28(7):83–91, July 1993.

[81] Jingke Li and Marina Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[82] Welf Löwe. Optimization of PRAM-programs with input-dependent memory access. In Seif Haridi, Khayri Ali, and Peter Magnusson, editors, *Proceedings of the First International EURO-PAR Conference*, Lecture Notes in Computer Science, pages 243–254, Stockholm, Sweden, August 29–31, 1995. Springer-Verlag.

[83] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Cambridge, Massachusetts, October 1996. ACM Press.

[84] T. MacDonald, D. Pase, and A. Meltzer. Addressing in Cray Research's MPP Fortran. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, pages 161–172, Vienna, Austria, July 1992. Austrian Center for Parallel Computation.

[85] MasPar Computer Corporation. MasPar Parallel Application Language (MPL) reference manual, September 1990.

[86] MasPar Computer Corporation. MasPar Fortran reference manual, July 1992.

[87] W. F. McColl. Universal computing. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Proceedings Euro-Par '96 Parallel Processing*, volume 1123 of *LNCS*, pages 25–36. Springer-Verlag, 1996.

[88] Sally A. McKee, Robert H. Klenke, Kenneth L. Wright, William A. Wulf, Maximo H. Salinas, James H. Aylor, and Alan P. Batson. Smarter memory: Improving bandwith for streamed references. *IEEE Computer*, 31(7):54–63, July 1998.

[89] Frank McMahon. The Livermore FORTRAN Kernels Test of the Numerical Performance Range. In Joanne L. Martin, editor, *Performance Evaluation of Supercomputers*, volume 4 of *Special Topics in Supercomputing*, pages 143–186. Elsevier Science Publishers B.V. (North-Hollan), Amsterdam, 1988.

[90] Michael Metcalf and John Ker Reid. *Fortran 90 explained*. Oxford science publications. Oxford University Press, 1994.

[91] S. P. Midkiff. Local iteration set computation for block-cyclic distributions. In *International Conference on Parallel Processing, Vol.2: Software*, pages 77–84, Boca Raton, USA, August 1995. CRC Press.

[92] T. Mowry. *Tolerating Latency Through Software Controlled Data Prefetching*. PhD thesis, Dept. of Computer Science, Stanford University, March 1994.

[93] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Boston, Massachusetts, October 1992.

[94] Matthias M. Müller, Thomas M. Warschko, and Walter F. Tichy. Prefetching on the Cray-T3E. In *12th International Conference on Supercomputing*, pages 368–375, Melbourne, July 13–17, 1998.

[95] J. M. Nash and P. M. Dew. XPRAM model and programming interface. In Daniel Etiemble and Jean-Claude Syre, editors, *Proceedings of Parallel Architectures and Languages Europe (PARLE '92)*, volume 605 of *LNCS*, pages 981–982, Berlin, Germany, June 1992. Springer.

[96] Wilfried Oed. The Cray Research massively parallel processor system CRAY T3D. Technical report, Cray Research GmbH, München, Germany, November 1993.

[97] Wilfried Oed. Massiv-paralleles Prozessorsystem CRAY T3E. Technical report, Cray Research GmbH, München, November 1996.

[98] Parasoft Corporation, Pasadena. Express C: User's Guide, Reference 3.0, 1992.

[99] Michael Philippsen. *Optimierungstechniken zur Übersetzung paralleler Programmiersprachen.* Number 292 in Serie 10: Informatik/Kommunikationstechnik. VDI-Verlag, 1993.

[100] Michael Philippsen, Ernst A. Heinz, and Paul Lukowicz. Compiling machine-independent parallel programs. *ACM SIGPLAN Notices*, 28(8):99–108, August 1993.

[101] R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Proceedings, Supercomputing '93: Portland, Oregon, November 15–19, 1993*, Supercomputing 1993, pages 361–370. ACM Press, November 1993.

[102] Ralf Reussner, Peter Sanders, Lutz Prechelt, and Matthias Müller. SKaMPI: A detailed, accurate MPI benchmark. In *Recent Advances in Parallel Virtual and Message Passing Interface, 5th European PVM/MPI Users' Group Meeting*, volume 1497 of *Lecture Notes in Computer Science*, Liverpool, UK, September 1998. Springer.

[103] Anne Rogers and Kai Li. Software support for speculative loads. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, Boston, Massachusetts, October 1992.

[104] Joel H. Saltz, Ravi Mirchandaney, and Doug Baxter. Run-time parallelization and scheduling of loops. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 303–312, Santa Fe, New Mexico, June 18–21, 1989. SIGACT and SIGARCH.

[105] Steven L. Scott. Synchronization and communication in the T3E multiprocessor. *ACM SIGPLAN Notices*, 31(9):26–36, September 1996.

[106] S. D. Sharma, R. Ponnusamy, B. Moon, Yuan-Shin Hwang, R. Das, and J. Saltz. Run-time and compile-time support for adaptive irregular problems. In *Proceedings, Supercomputing '94: Washington, DC, November 14–18, 1994*, Supercomputing, pages 97–106. IEEE Computer Society Press, 1994.

[107] A.J. Smith. Cache memories. *Computing Surveys*, pages 473–530, September 1982.

[108] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. *SPIE Real-Time Signal Processing IV*, 298:241–248, 1981.

[109] James E. Smith, Shlomo Weiss, and Nicholas Y. Pang. A simulation study of decoupled architecture computers. *IEEE Transactions on Computers*, 35(8):692–701, August 1986.

[110] J.E. Smith, G.E. Dermer, B.D. Vanderwarn, S.D. Klinger, C.M.Rozewski, D.L. Fowler, D.R. Scidmore, and J.P. Lauden. The ZS-1 central processor. Technical report, Astronautics Corporation of America, Madison, Wisconsin, 1987.

[111] A. Sohn, Y. Kodama, J. Ku, M. Sato, H. Sakane, H. Yamana, S. Sakai, and Y. Yamaguchi. Fine-grain multithreading with the EM-X multiprocessor. In *9th ACM Symposium on Parallel Algorithms and Architectures*, Newport, Rhode Island, June 1997.

[112] J. Stichnoth. Efficient compilation of array statements for private memory multicomputers. Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, February 1993.

[113] Thinking Machines Corporation, Cambridge, Massachusetts. C⋆ Reference Manual, version 4.0, 1987.

[114] Thinking Machines Corporation, Cambridge, Massachusetts. CM Fortran Reference Manual, version 1.0. Cambridge, Massachusetts: MIT Press, Februrary 1991.

[115] Ashwath Thirumalai. Code generation and optimization for High Performance Fortran. Master's thesis, Lousiana State University, August 1995.

[116] M. R. Thistle and B. J. Smith. A processor architecture for Horizon. In IEEE, editor, *Proceedings, Supercomputing '88: November 14–18, 1988, Orlando, Florida*, volume 1, pages 35–41, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1988. IEEE Computer Society Press.

[117] N. P. Topham and K. McDougall. Performance of the decoupled ACRI-1 architecture: the perfect club. In *Proceedings High Performance Computing - Europe*, Milan, Italy, 1995.

[118] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran-D. In *Proceedings of 5th Workshop on Languages and Compilers for Parallel Computing*, 1992.

[119] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[120] Thomas M. Warschko. *Effiziente Kommunikation in Parallelrechnerarchitekturen.* PhD thesis, School of Computer Science, Universität Karlsruhe, 1997.

[121] M. J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, Redwood City, CA, 1996.

[122] W. A. Wulf. Evaluation of the WM architecture. In David Abramson and Jean-Luc Gaudiot, editors, *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 382–391, Gold Coast, Australia, May 1992. ACM Press.

[123] W. Zimmermann and W. Loewe. An approach to machine-independent parallel programming. In *Parallel Processing: CONPAR 94 - VAPP VI*, volume 854 of *Lecture Notes in Computer Science*, pages 277–288, 1994.

[124] W. Zimmermann, M. Middendoff, and W. Loewe. On optimal $\kappa$-linear scheduling of tree-like task graphs for LogP-Machines. In *Europar '98: Parallel Processing*, volume 1470 of *Lecture Notes in Computer Science*, pages 328–336, 1998.