Effizienter Nachrichtenaustausch auf speichergekoppelten Rechnerverbundsystemen mit SCI Verbindungsnetz

Von der Fakultät für Elektrotechnik und Informationstechnik
der Rheinisch-Westfälischen Technischen Hochschule Aachen
genehmigte Dissertation zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften

vorgelegt von

Diplom-Ingenieur Joachim Worringen
aus Oberhausen

Berichter:
Universitätsprofessor Dr. rer.nat. habil. Thomas Bemmerl
Universitätsprofessor Dr. rer.nat. Friedel Hoßfeld

Tag der mündlichen Prüfung: 17. Juli 2003

„Until I get mad I don't know my own brains!"

Donald Duck (WDC 47)

ZUSAMMENFASSUNG

Ein Rechnerverbundsystem (engl. *Cluster*) ist eine Anordnung von unabhängigen Rechenknoten, die über ein Verbindungsnetz kommunizierend zur Lösung parallelisierter Probleme eingesetzt werden können. Zielplattform dieser Arbeit sind Cluster, deren Knoten über ein Hochleistungs-Verbindungsnetz gemäß des IEEE 1596-Standards (*Scalable Coherent Interface, SCI)* gekoppelt sind. SCI ermöglicht den transparenten Zugriff einer CPU auf Speicher in entfernten Knoten. In der vorliegenden Arbeit wird auf dieser Plattform das Programmiermodell des Nachrichtenaustauschs über die Programmierschnittstelle *Message Passing Interface (MPI)* mit neuartigen Verfahren umgesetzt, um durch die optimierte Kommunikationsleistung einen möglichst hohen Teil der Gesamtleistung des Systems zu erreichen.

Zunächst wird die Obergrenze der Kommunikationsleistung für unterschiedliche Kommunikationsabläufe und SCI-Topologien in einem solchen Rechnerverbundsystem analytisch ermittelt. Entsprechend dieser Charakteristik werden die erforderlichen Varianten der Punkt-zu-Punkt-Kommunikationsprotokolle entwickelt und mit einem neu eingeführten, einfachen Effizienzmodell analysiert. Zur Abwicklung kollektiver Operationen, bei denen eine Gruppe von Prozessen untereinander Daten austauscht, werden neue Kommunikationsalgorithmen vorgestellt, die durch die Ausnutzung der speziellen Eigenschaften des SCI-Verbindungsnetzes ein Vielfaches der Leistung generischer Algorithmen erreichen. Auch die Leistung bestehender, SCI-optimierter Verfahren wird teilweise weit übertroffen.

Bei der Übertragung nichtzusammenhängender Speicherbereiche erhöht ein innovatives Verfahren die Übertragungsbandbreite auf bis zu 100% des Vergleichswerts für zusammenhängende Speicherbereiche. Gleichzeitig wird der Speicherverbrauch verringert und somit die Lokalität verbessert. Auch für einseitige Kommunikation werden Kommunikations- und Synchronisationsverfahren vorgestellt, die unter allen Bedingungen effizient einsetzbar sind. Im Gegensatz zu bisher verbreiteten Methoden erreichen sie durch asynchrone Synchronisierung auch bei nicht-synchronem Ablauf der beteiligten Prozesse eine hohe effektive Leistung, was in typischen Anwendungsfällen die Ausführungszeit halbiert. Schließlich werden Kommunikationsverfahren und darauf aufbauende Protokolle vorgestellt, die die Überlappung von Kommunikation und Berechnung effizient unterstützen. Dadurch kann die effektive Kommunikationszeit zu über 90% reduziert werden.

Die im Rahmen dieser Arbeit entwickelte Software verbindet die zahlreichen neuartigen Algorithmen und Verfahren zu einer außergewöhnlich leistungsfähigen und flexiblen MPI-Bibliothek. Sie ist frei im Quelltext verfügbar, so daß die gewonnenen Erkenntnisse zur effizienten Parallelisierung auf Clustern genutzt, aber auch auf andere Anwendungsbereiche übertragen werden können.

ABSTRACT

A *cluster* is a group of independent compute nodes which are coupled by an interconnection network. They can be used to execute parallel applications, consisting of processes which communicate via the interconnection network. This work deals with clusters coupled by a high-performance, memory-coupling *Scalable Coherent Interface (SCI, IEEE 1596* standard*)* interconnect. SCI enables transparent access to remote memory locations. This platform serves as a basis for the implementation of a message-passing programming model using the *Message Passing Interface (MPI)* application programming interface specification. A number of new techniques is developed and applied for this implementation to make a maximal fraction of the potential performance available to the MPI-based parallel application.

The upper limit of the communication performance of an SCI interconnect with n-dimensional torus topology is analytically determined for different typical communication scenarios. The required variants of the basic point-to-point communication protocols are designed accordingly. They are evaluated by a new multi-level efficiency model. Next to this, new communication algorithms and protocols for collective operations (which involve an arbitrary number of communicating processes) are presented. They achieve a multiple of the performance of generic techniques, and also surpass the performance of an available SCI-optimized implementation.

An innovative technique for the transmission of non-contiguous memory areas increases the transfer bandwidth for this application of MPI derived data types. Up to 100% of the transfer bandwidth of contiguous memory areas have been achieved, while reducing the working set size and thus increasing the locality. For one-sided communication, means for communication and synchronization have been developed. They perform well even for loosely coupled, non-phased communication scenarios by the use of asynchronous synchronisation. Finally, advanced communication protocols and new data transfer mechanisms are shown to make efficient overlapping of communication and computation possible. This allows to hide up to 90% of the communication time.

The software package that has been developed with this work merges the new techniques and algorithms into an MPI library of outstanding performance and flexibility. It is freely available including the source code. Therefore, it can easily be used for the efficient employment of clusters and for transferring the findings to other areas of parallel processing.

INHALT

KAPITEL 1

Einleitung

Datenverarbeitungssysteme, die mehrere Prozessoren zugleich und koordiniert einsetzen, um ein Problem zu lösen oder einen Vorgang zu verarbeiten, gibt es seit Jahrzehnten in verschiedenen Gattungen unter verschiedenen Bezeichnungen. Ihre Struktur und Organisation ist jedoch einem steten Wandel unterworfen, der sowohl in der fortschreitenden Entwicklung, der wachsenden Verfügbarkeit und dem sinkendem Preis der Technologie als auch in den sich ändernden Einsatzgebieten solcher Systeme begründet ist. Neben dem Einsatz von solchen parallel arbeitenden Systemen zur Maximierung der Verfügbarkeit oder des Durchsatzes von einzelnen Transaktionen ist die Maximierung der Gesamtrechenleistung eine sehr wichtige Aufgabe. Diese Rechenleistung wird benötigt, um immer komplexere Simulationen im technischen und wissenschaftlichen Bereich in möglichst kurzer Zeit durchzuführen.

Prozessorleistung. Hierbei spielt zunächst die Leistung der Prozessoren die wichtigste Rolle. Die Rechenleistung von Mikroprozessoren, die in gängigen Arbeitsplatzrechnern oder einfachen Servern verwendet werden, reicht immer näher an die Rechenleistung der in Hochleistungsrechnern verwendeten Prozessoren heran. Zumindest im Verhältnis von Preis zu nominaler Maximalleistung schneiden die massenhaft hergestellten Standardprozessoren von Herstellern wie Intel, AMD oder HP bereits jetzt besser ab als die spezialisierten Prozessoren etwa von Fujitsu, Hitachi oder Cray. Dies führte dazu, dass Hersteller von Hochleistungsrechnern dazu übergingen, in ihren Produkten Standardprozessoren zu verwenden, um so die Entwicklungszeiten und damit auch die Kosten für die Systeme zu reduzieren. Zu dieser Maßnahme wurden sie auch durch die zunehmende Verbreitung von Rechnerverbundsystemen motiviert, die sich zu einer technisch und wirtschaftlich ernstzunehmenden Konkurrenz entwikkelt haben.

Rechnerverbundsysteme. Das Prinzip des Zusammenschlusses autarker Rechnersysteme zu Rechnerverbundsystemen (i. A. als *Cluster* bezeichnet), in dem die einzelnen Rechnersysteme (*Knoten)* gemeinsam zur Lösung einer Aufgabe verwendet werden, wird seit vielen Jahren genutzt. Der Einsatz von Rechnerverbundsystemen wurde jedoch verstärkt durch zwei wesentliche Entwicklungen der letzten Jahre:

- *Preisgünstige Hardware.* Sowohl Prozessoren als auch Arbeitsspeicher, Festplatten und alle anderen Komponenten zum Aufbau eines Rechners mit hoher Leistung und Kapazität sind zunehmend preiswerter geworden und in hoher Qualität und Kompatibilität einfach zu beschaffen. Somit ist sowohl der Aufbau eines einzelnen, ausreichend dimensionierten Knotens eines Clusters als auch der Aufbau des gesamten Clusters aus der benötigten Anzahl von Knoten kostengünstig und mit geringem Beschaffungsaufwand zu bewerkstelligen.
- *Verfügbare Software.* Neben der Hardware spielt die verbesserte Verfügbarkeit von geeigneter Software sowohl zum Betrieb eines Clusters als auch zur Entwicklung von Applikationen und Werkzeugen eine wichtige Rolle. Zweifellos war der entscheidende Schritt zu dieser Verfügbarkeit die Entwicklung des Internets zu einem überall verfügbaren, einfach benutzbaren Kommunikationsmedium. Der elementare Software-Baustein für einen Cluster ist das Betriebssystem, dessen kostenfreie und offene Verfügbarkeit durch Entwicklungen wie Linux oder verschiedene Varianten von BSD Unix gewährleistet ist.

Eines der bekanntesten Projekte, das in diesem Umfeld von preisgünstiger Hardware und verfügbarer Software entstanden ist, ist das Beowulf Projekt *[1]*. Es entstand 1995 am Goddard Space Flight Center der NASA und bestand im wesentlichen aus 16 schnellen PCs aus Standardbauteilen, die unter Linux betrieben wurden. Zur Erhöhung der Kommunikationsbandbreite zwischen den einzelnen Systemen kam ein spezieller Treiber zum Einsatz, der den Datenstrom

über mehrere Ethernetkarten verteilen konnte. Die hiermit begründete Klasse der Beowulf-Cluster ist also gekennzeichnet durch die durchgehende Verwendung von preisgünstigen Standardkomponenten für die Hardware und kostenfreier, im Quelltext frei verfügbarer Software. Technisch gesehen lag hier keine Revolution vor, doch in der Verfügbarkeit von leistungsfähigen Parallelrechnern vollzog sich ein Wandel: Als Alternative zu der Nutzung von beantragten und zugeteilten Ressourcen großer Systeme in entfernten Rechenzentren bot sich die nunmehr erprobte Variante des Aufbaus und der selbstverwalteten Nutzung des eigenen, dedizierten Parallelrechners an. Diesen kann sich etwa ein Lehrstuhl einer Universität oder eine Abteilung eines Unternehmens kostengünstig aufbauen und so die Vorteile der Parallelverarbeitung unabhängig und flexibel nutzen. So sind derartige Systeme heute häufig anzutreffen und werden inzwischen auch wiederum als Platzbedarf-minimierte Komplettsysteme von großen PC-Herstellern angeboten.

Kommunikationsleistung. Derartige Cluster, deren Knoten nur aus Komponenten bestehen, die auch in einem einzeln betriebenen Rechner mit Anschluß an ein lokales Netz (LAN) zum Einsatz kommen, werden üblicherweise als COTS-Cluster[1] bezeichnet. Die Knoten kommunizieren hierbei i.d.R. über das TCP/IP Protokoll auf der Basis von Fast Ethernet (100 Mb/s) oder seltener, weil mehrfach teurer, Gigabit Ethernet (1 Gb/s). Für parallele Applikationen mit sehr geringer Kopplung zwischen den Teilaufgaben reicht die Kommunikationsleistung solcher Cluster aus, um eine ausreichende Gesamtleistung zu erzielen. Viele parallele Applikationen erfordern jedoch Kommunikation mit höherer Bandbreite und niedrigerer Latenz, um insbesondere mit zunehmendem Parallelitätsgrad effizient ausgeführt zu werden. Dieser Bedarf kann von reinen COTS-Clustern nicht befriedigt werden - die Ethernet-basierten Kommunikationskanäle müssen durch andere Systeme ersetzt werden. Hierzu wurden Lösungen entwickelt, die nun zwar nicht mehr als Standardkomponenten in jedem Herstellerregal zu finden sind, aber doch gegenüber den integrierten Lösungen einzelner Hersteller ein gutes Preis-/Leistungsverhältnis bieten. Hier sind Lösungen wie MemoryChannel, ServerNet, Giganet cLan, Quadrix, Myrinet und SCI zu nennen.

SCI (*Scalable Coherent Interface*) ist ein als IEEE Standard definiertes Verbindungsnetz. Es koppelt verteilte Rechensysteme, indem ihnen der (optional Cache-kohärente) gegenseitige Zugriff auf ihre Adreßräume ermöglicht wird. Physikalisch entfernter Speicher wird über den globalen SCI Adreßraum in den lokalen Adreßraum eines Prozesses eingeblendet. Somit besteht die Kommunikation lediglich aus Lade- und Speichervorgängen auf Speicheradressen. Diese Vorgänge können von beliebigen Initiatoren ausgehen, seien es CPUs oder DMA-Vorrichtungen eines Rechensystems. Die Möglichkeiten zur Kommunikation entsprechen daher in hohem Maße denen innerhalb eines abgeschlossenen Rechensystems und erfolgen mit geringer Latenz und hoher Bandbreite, da keinerlei Protokoll in Software abgearbeitet werden muß.

Programmiermodelle. Neben leistungsfähiger Hardware ist zur Nutzung eines parallelen Rechensystems als Hochleistungsrechner ein effizientes Programmiermodell mit entsprechender Programmierschnittstelle erforderlich, um Applikationen den Zugang zu den Ressourcen des Systems möglichst einfach zu machen. Als Programmiermodell hat sich im Bereich des Hochleistungsrechnens *Message Passing* (Nachrichtenaustausch) durchgesetzt: Parallel laufende Prozesse tauschen Daten durch explizite Sende- und Empfangsoperationen aus. Der Grund für den Erfolg dieses Modells ist, daß es sich auf allen verschiedenen Plattformen für Parallelrechner (gemeinsamer oder verteilter Speicher) effizient implementieren läßt. Nachdem viele Jahre lang jeder Hersteller seine eigene Variante dieses Programmiermodells als Programmierschnittstelle für seine Systeme angeboten hat, wurde 1994 das *Message Passing Interface (*MPI*)* in der Version 1 vorgestellt *[63]*, das zwei Jahre später stark erweitert zur Version 2 wurde *[64]*. Ver-

1. COTS: ***C****omponents* ***O****ff* ***T****he* ***S****helf*

sion 1 legt alle Operationen zum paarweisen und kollektiven Nachrichtenaustausch fest; Version 2 definiert umfassende Erweiterungen im Bereich Ein-/Ausgabe (MPI-IO) und Prozeßverwaltung und führt einseitige Kommunikation ein. Obwohl diese Spezifikationen nicht von einer Normungsorganisation standardisiert wurden, sind sie als MPI-1 respektive MPI-2 Standard weltweit akzeptiert. Durch MPI ist die theoretische Portierbarkeit von parallelen Programmen gemäß dem Message-Passing Programmiermodell nun auch faktisch gegeben, da jedes parallele Rechensystem zumindest MPI-1 konforme Programme verarbeiten kann (mit zunehmender Unterstützung von MPI-2).

Diese Arbeit. In dieser Arbeit wird eine MPI-Implementation vorgestellt und untersucht, die den vollständigen MPI-1 Standard und Teile des MPI-2 Standards auf Clustern mit SCI-Verbindungsnetz verfügbar macht. Es wird dargestellt, wie unter Ausnutzung der speziellen Eigenschaften von SCI effiziente Verfahren zum Nachrichtenaustausch entwickelt wurden. Vielfach bieten sich für eine Aufgabe verschiedene Verfahren an, deren spezifische Eigenschaften im Modell und experimentell untersucht werden.

Im folgenden Kapitel wird zunächst als Voraussetzung und Motivation für das Thema dieser Arbeit der Stand der Technik auf dem Gebiet der Kommunikation in Rechnerverbundsystemen dargestellt. Die technische Basis der in dieser Arbeit vorgestellten und untersuchten Entwicklungen, Rechnerverbundsysteme aus SCI-vernetzten Rechenknoten mit der zum Betrieb nötigen Software, wird in Kapitel 3 detaillierter beschrieben. Kapitel 4 beschreibt und analysiert die Entwicklungen, die im Bereich des grundlegenden Nachrichtenaustausches durchgeführt wurden und analysiert die Effizienz der Protokolldefinition und -implementation. Optimierte und neuartige Wege, die die speziellen Eigenschaften der Kommunikation in speichergekoppelten Rechnerverbundsystemen nutzen, sind Grundlage der in Kapitel 5 dargestellten Verfahren zur Abwicklung von einseitiger und nicht-zusammenhängender Kommunikation. In Kapitel 6 werden Algorithmen vorgestellt und deren Implementierung analysiert, die die Abwicklung von kollektiven Operationen in MPI optimieren. Kapitel 7 stellt Verfahren vor, mit denen die Punkt-zu-Punkt-Kommunikation asynchron, mit minimaler CPU-Belastung und unter Vermeidung von Zwischenkopien abgewickelt werden kann, und untersucht die Auswirkungen dieser Verfahren. Abschließend fasst Kapitel 8 die gesamte Arbeit zusammen und zeigt weitere Entwicklungs- und Anwendungsmöglichkeiten der präsentierten Verfahren auf den speichergekoppelten Rechnerverbundsystemen auf. Das Glossar im Anhang stellt die verwendeten Begriffe, Abkürzungen und Formelzeichen zusammen.

KAPITEL 2

Rechnerverbundsysteme: Stand der Technik

Die Entwicklung von Clustern hat sowohl im Bereich der Hardware als auch der Software in den letzten Jahren eine stürmische Entwicklung vollzogen. Es gilt daher, zunächst den aktuellen Stand der Technik im Bereich der COTS-Cluster zu beschreiben, um daraus die Motivation für die in dieser Arbeit vorgestellten Entwicklungen abzuleiten. Eine weitergehende Beschreibung von relevanten Rechnerarchitekturklassen im Bereich der Parallelverarbeitung findet sich in *[158,159]*. Die Prinzipien von CPU-Architektur und Speichersystemen werden in *[157]* detailliert erläutert. Eine Zusammenfassung des Stands der Technik bei Rechnerverbundsystemen findet sich in einer Veröffentlichung der *IEEE Taskforce for Cluster Computing [5]*.

Cluster im Sinne dieser Arbeit sind im Gegensatz zu *Networks of Workstations* (NOW *[2]*) homogen aus einer Zahl gleichartiger Rechensysteme (den *Knoten*) aufgebaut. Da bei COTS-Clustern die Anschaffungskosten vergleichsweise niedrig sind und die Systeme als dedizierte Parallelrechner betrieben werden, wird zumeist von Beginn an die für die gewünschte Leistung erforderliche Anzahl an Knoten installiert. So vermeidet man Probleme wie Lastungleichgewicht aufgrund unterschiedlicher Leistungsfähigkeit der Knoten, Unterschiede in der Datendarstellung, Softwareinkompatibilität und -unverfügbarkeit, die in heterogenen Systemen auftreten können. Die Zahl der installierten Knoten in einem Cluster reicht von minimal zwei bis hin zu Tausenden von Knoten *[3,4]*. Eine möglicherweise erforderliche Erhöhung der Leistungsfähigkeit eines Clusters durch Hinzufügung von identischen Knoten ist aufgrund der beschränkten zeitlichen Verfügbarkeit eines bestimmten Stands (Typ und Art) der beteiligten Komponenten nur für einen Zeitraum von etwa einem Jahr nach Anschaffung eines Clusters möglich. Danach führt eine Erweiterung eines Clusters zumeist zu der Installation eines parallel operierenden Systems, das noch mit dem alten System gemeinsam verwaltet, aber üblicherweise nicht gemeinsam zur Ausführung einer Applikation genutzt wird.

Die wesentlichen Merkmale eines Clusters sind der einzelne Knoten als Hardwarebasis, das Verbindungsnetz sowie die Softwareumgebung, bestehend aus Betriebssystem und unterstützenden Bibliotheken.

2.1 Rechenknoten

Die wichtigsten Elemente eines Rechenknotens sind die CPUs, der Hauptspeicher und seine Anbindung, das Ein-/Ausgabe-System (E/A-System) sowie als Software das Betriebssystem.

2.1.1 CPU

Da in einem Rechensystem alle Daten letztendlich nur bewegt werden, um von der CPU verarbeitet zu werden, ist diese das zentrale Element, dem die anderen Teilsysteme zuarbeiten. Die akkumulierte Maximalleistung eines SMP[1]-Systems skaliert linear mit der Zahl der CPUs. Der Preis steigt jedoch bei COTS-Rechenknoten sub-linear, da im wesentlichen nur die zusätzlichen CPUs erworben werden müssen. Die Kosten für die peripheren Komponenten bleiben konstant. Dieser Effekt macht SMP-Knoten, die sich nur durch die Zahl der CPUs am Systembus von dem Uniprozessor-Pendant unterscheiden, relativ zur maximalen CPU-Leistung preiseffektiver als Uniprozessor-Knoten. Die Gesamtleistung eines solchen SMP-Systems für Applikationen, die nicht vollständig im Cache ablaufen können, steigt jedoch aufgrund des überlasteten Speichersystems (siehe Kapitel 2.1.2) nur sublinear mit der Zahl der CPUs. Um auch für diese Fälle eine lineare Skalierung zu erreichen, ist ein Speicher- und E/A-System erforderlich, das jeder CPU,

1. *Symmetric Multi-Processing* - Rechensystem mit zwei oder mehr gleichberechtigt arbeitenden CPUs

auch bei gleichzeitiger Aktivität aller CPUs, die volle Bandbreite für den Zugriff auf den Hauptspeicher und angeschlossene E/A-Geräte ermöglicht. Bei den E/A-Geräten selber wiederum ist zu berücksichtigen, daß die von ihnen zur Verfügung gestellte Bandbreite etwa zur Kommunikation nach außen, aber auch bei Zugriff auf lokale Ressourcen, von allen aktiven Prozessen geteilt werden muß (siehe dazu auch die folgenden Kapitel).

Der damit verbundene zusätzliche Aufwand führt dazu, daß bei zunehmender Zahl von CPUs pro Knoten der Preis pro CPU zum Erreichen einer bestimmten Rechenleistung über dem Preis einer äquivalenten größeren Zahl von Knoten mit weniger CPUs pro Knoten liegt. Für dieses Problem eine optimale Lösung zu finden (nämlich das optimale Preis/Leistungs-Verhältnis), hängt von dem zu bearbeitenden Problem und den zu seiner Lösung verfügbaren Softwarearchitekturen und nutzbaren Kommunikationsmodellen ab. Im Rahmen dieser Arbeit werden die besonderen Eigenschaften von gekoppelten SMP-Systemen bezüglich der Inter- und Intra-Knotenkommunikation implizit berücksichtigt, da sich die vorgestellten Kommunikationsprotokolle (siehe Kapitel 4) gleichermaßen für beide Kommunikationsvarianten nutzen lassen.

2.1.2 Hauptspeicher

Die Leistungsfähigkeit des Hauptspeichers (bzw. des gesamten Speichersystems) kann die effektive erreichte Leistung eines Rechensystems stärker beeinflussen als die Leistungsfähigkeit der CPU, da die CPU nie schneller Daten verarbeiten kann als diese vom Hauptspeicher zur Verfügung gestellt werden. Dies trifft auf alle Applikationen zu, die bei ihrer Abarbeitung eine geringe Datenlokalität haben. Dies bewirkt eine geringe Trefferquote in den angeschlossenen Caches, so daß wiederholt Daten aus dem Hauptspeicher gelesen werden müssen. Dabei läuft CPU im Leerlauf, weil notwendige Daten nicht zur Verfügung stehen.

Ein verbreiteter Benchmark, der das Verhalten der Vektorverarbeitung von Datensätzen (mit entsprechend geringer Datenlokalität) und damit die Leistungsfähigkeit des Hauptspeichers, untersucht, ist STREAM *[142]*. Er untersucht die Speicherbandbreite für Kopieroperationen und verschiedene arithmetische Operationen. Zur Evaluierung der Speichersysteme wurden zwei typische COTS-Knoten der aktuellen und vorhergehenden Generation ausgewählt. Zur Einord-

	SW LE[a]		Intel NX[b]			Fire 6800[c]		
Op	**1**	**2**	**1**	**2**	**4**	**1**	**2**	**4**
K	270,9	171,1	289,4	146,6	71,4	386,3	376,6	359,2
A	226,1	171,8	280,1	149,4	69,4	374,1	364,3	345,1
S	339,0	195,9	321,1	162,2	76,4	305,6	300,9	292,1
T	334,3	197,8	299,8	141,8	79,4	288,1	284,1	277,0

Tabelle 2.1: Speicherbandbreiten (pro Prozess) gemäß STREAM Benchmark für verschiedene Intel- und Sparc-basierte Architekturen

a. SuperMicro 370 DLE, 2 x Intel Pentium-III, 800 MHz, mit ServerWorks ServerSet-III LE Chipsatz, PC133 Speicher, 2 Bänke bestückt
b. Dell PowerEdge 6300, 4 x Intel Pentium-III Xeon, 550 MHz, mit Intel NX-Chipsatz, PC66 Speicher, 2 Bänke bestückt.
c. Sun Fire 6800, 24 x UltraSparc-III, 600 MHz, Sun-spezifizierter Speicher (System war nicht exklusiv für den Test verfügbar, da eine CPU anderweitig belegt war.)

nung der Ergebnisse wurde zusätzlich ein Vergleich mit einer typischen Server-Architektur vorgenommen, indem auf einem System von Sun die gleichen Testdurchläufe vermessen wurden. Der STREAM Benchmark für auf zwei Vektoren *A* und *B* (und ggf. einen Skalar *c*) die Operationen *Kopieren* ($A[i] = B[i]$, in der Tabelle 2.1 mit **K** abgekürzt), *Addieren* ($A[i] = A[i] + B[i]$, **A**), *Skalieren* ($A[i] = c \cdot B[i]$, **S**) und *Triade* ($A[i] = A[i] + c \cdot B[i]$, **T**) durch. Die Ergebnisse sind in Tabelle 2.1 dargestellt. Für die Untersuchungen wurde der Benchmark mit voller plattformspezifischer Optimierung durchgeführt. Es wurden dazu jeweils Compiler vom CPU-Hersteller (Intel bzw. Sun) verwendet. Die Ergebnisse verdeutlichen, daß die Bandbreite der Speichersysteme von COTS-Knoten nicht mit der Zahl der CPUs skaliert, was auf ihre einfache Systemarchitektur mit einem gemeinsamen Bus zwischen CPUs und Speicher zurückzuführen ist. Im Gegensatz dazu zeigen die Resultate der Server-Plattform, daß mit höherem technischen Einsatz skalierende Speichersysteme möglich sind. Dieser Vorteil muß jedoch zu einem hohen Preis erworben werden: die Kosten pro CPU liegen für den Server von Sun um den Faktor 10 höher.

Die technische Entwicklung der Speichersysteme von PCs und Workstations verlief bezüglich der erreichten Leistungswerte Bandbreite, Skalierbarkeit und insbesondere Latenz mit deutlich geringeren Steigerungsraten als bei CPUs *[44]*. Diese gewachsene Differenz der Verarbeitungsgeschwindigkeit der CPUs und der Bandbreite bzw. Latenz der Speichersysteme führt zu sinkender Effizienz der CPUs für reale Applikationen. Durch Entwicklungen alternativer Architekturen von dynamischem Speicher wie RAMbus, die deutlich höhere Bandbreiten ermöglichen, werden inzwischen die etablierten Architekturen durch schnellere Taktung zu höheren Bandbreiten befähigt *[36]*. Dies wirkt sich jedoch kaum auf die Zugriffslatenz aus, die nur unwesentlich von der Taktrate des Speicherbusses abhängt.

Zusammenfassend kann festgestellt werden, daß die Speichersysteme von Systemen, wie sie in dieser Arbeit betrachtet werden, gegenüber der Leistungsfähigkeit der CPUs auf absehbare Zeit relativ leistungsschwach sein werden. Ein Zunahme der Zahl von CPUs pro Knoten ist in diesem Segment daher nicht zu erwarten, da selbst bei Nutzung von zwei CPUs pro Knoten das Speichersystem bei Applikationen, die nicht auf maximale Datenlokalität optimiert sind, den Leistungsengpaß darstellt. Dies hat Einfluß auf das Kommunikations- und Programmiermodell und dessen Implementation. Diese sind von diesem Problem abhängig, können es jedoch auch durch ihre Art der Speicherzugriffe und Datenübertragung beeinflussen bzw. umgehen, wie in den Kapiteln 5.1 und 7.2 gezeigt wird.

2.1.3 E/A-System

Neben dem Laden und Speichern von Daten im Hauptspeicher ist die Kommunikation der CPUs mit Ein-/Ausgabegeräten (E/A-Geräten) elementar für ein Rechensystem, da ansonsten Daten nicht dauerhaft gespeichert und nicht mit anderen, externen Systemen ausgetauscht werden könnten. Die Anbindung von E/A-Geräten erfolgt inzwischen fast ausschließlich über den PCI-Bus (*Peripheral Component Interface)*. PCI wurde als offener Industriestandard maßgeblich von Intel als Nachfolger des ISA-Busses für PCs entwickelt. Wegen der technischen Eigenschaften hat sich der PCI-Bus inzwischen bei allen Herstellern von PCs und Servern durchgesetzt.

Der aktuelle PCI-Standard in der Version 2.3 *[14]* definiert unterschiedliche Busbreiten und Taktfrequenzen, die in Tabelle 2.2 zusammen mit der resultierenden maximalen Bandbreite und der maximalen Zahl von Steckplätzen (*Slots)* pro Bus aufgeführt sind. Der Nachfolgestandard von PCI, PCI-X, wurde bereits im Jahr 2000 verabschiedet *[15,17]*; damit ausgestattete Komponenten für COTS-Systeme sind inzwischen verfügbar. Es wird jedoch noch einige Zeit dauern, bis PCI-X-fähige Erweiterungen auf breiter Basis verfügbar sind. Wie die Berechnungen in Kapitel 3.3.2 zeigen werden, ist der aktuelle PCI-Bus bei sorgfältiger Implementation in den

Typ	**Frequenz** [MHz]	**Datenbreite** [Bit]	**Bandbreite** [MB/s]	**max. Slots / Bus**
PCI 2.2	33	32	132	6
	33	64	264	4
	66	64	532	2
PCI-X	66 / 100 / 133	64	max. 1064	4 / 2 / 1

Tabelle 2.2: Definierte Varianten der PCI- und PCI-X-Standards

kommenden Jahren für die meisten Anwendungsfälle noch nicht der Flaschenhals für die Kommunikation in COTS-Clustern.

2.1.4 Betriebssystem

Neben der Hardware spielt für die Nutzbarkeit eines Clusters, bestehend aus den beschriebenen Rechenknoten und den in Kapitel 2.2 behandelten Verbindungsnetzen, die Software eine entscheidende Rolle. Grundlage für alle anderen Entscheidungen im Bereich der Software ist die Wahl des Betriebssystems (BS). Kommerzielle Cluster von Systemherstellern wurden bislang ausschließlich mit dem herstellerspezifischen BS betrieben. Dessen spezifische Eigenschaften wie etwa Ausfallsicherheit, besondere Sicherheitsmerkmale, Unterstützung wesentlicher Applikationen (*"Killerapplikationen")* und nicht zuletzt überdurchschnittliche Skalierbarkeit der Leistung einer bestimmten Ressource (Speicher, E/A-Geräte) im Zusammenspiel mit proprietärer Hardware dienen dem Hersteller als unterscheidendes Merkmal. Demgegenüber haben freie BS wie Linux und verschiedene BSD-Varianten, die im Quelltext vorliegen, zusammen mit dem großen Angebot an frei verfügbarer Software die Entwicklung der COTS-Cluster erst ermöglicht. Das Prinzip der freien und offenen Software im Allgemeinen und Linux als BS für COTS-Cluster im Besonderen erlaubt die Anpaßbarkeit der Software für die individuellen Zwecke, zum Testen neuer Konzepte und zur Erweiterung der vorgegebenen Möglichkeiten. Zudem können in diesem Rahmen durchgeführte Entwicklungen ohne Einschränkungen veröffentlicht und weitergegeben werden, was in dem wissenschaftlich-akademischen Umfeld, in dem diese Systeme entwickelt und betrieben wurden, ein wesentlicher Vorteil ist. Diesem Modell haben sich inzwischen auch Hersteller wie IBM angeschlossen, die neben ihren proprietären BS mit hohem Aufwand Linux weiterentwickeln und auch Linux-basierte Systeme vermarkten.

Obwohl die proprietären Lösungen sowohl im Software- als auch im damit verbundenen Hardwarebereich weiterhin Eigenschaften haben, die bei freier Software nicht oder nicht in diesem Reifegrad verfügbar sind, hat die erwiesene Nutzbarkeit von freier Software für eine Vielzahl von Aufgaben starke Veränderungen im jeweiligen Bereich bewirkt: die Software muß nicht mehr gekauft werden, vielmehr können das notwendige Wissen zum Einsatz der Software oder komplette Lösungen erworben werden - die minimal notwendigen Investitionen zum Betrieb eines Systems mit freier Software liegen jedoch bei Null, was eine starke Verbreitung fördert und proprietäre Lösungen verdrängt.

2.2 Verbindungsnetze

Es gibt kaum einen Einsatzzweck eines Rechensystems, bei dem nicht die Vernetzung mit anderen Rechensystemen, externen E/A-Geräten oder anderen Datenquellen erforderlich ist. Dazu wurde eine Vielzahl von verschiedenen Kommunikationssystemen und -protokollen entwickelt, die entweder für einen bestimmten Einsatzzweck optimiert sind oder als universell einsetzbare

Technologien bestimmt sind. Eine detaillierte Übersicht dazu findet sich in *[21]* (Hardware) und *[162]* (Software).

Neben der Leistungsfähigkeit der Knoten rückt bei Clustern immer stärker die Leistungsfähigkeit des Verbindungsnetzes *(Interconnect)* in den Vordergrund. Während sich die Hardware der Rechenknoten verschiedener Cluster nicht grundlegend bezüglich spezieller Fähigkeiten oder um Größenordnungen bezüglich der erzielten Leistung voneinander unterscheidet, bietet sich im Bereich der Verbindungsnetze ein anderes Bild. Verschiedene Technologien haben sehr unterschiedliche Niveaus in Preis und Leistung und auch im zugrundeliegenden Kommunikationsmodell (siehe Kapitel 2.3). Gemeinsam ist ihnen die Anbindung an die Rechenknoten über den PCI-Bus. Die Eigenschaften des PCI-Bus beeinflussen daher die effektive Leistung der Verbindungsnetze, wie auch für diese Arbeit insbesondere in Kapitel 5 gezeigt wird.

Zwei wesentliche Parameter, die die Leistung eines Verbindungsnetzes beschreiben, sind die *Bandbreite* und die *Latenz*. Die Bandbreite beschreibt die Datenmenge, die pro Zeiteinheit von der Quelle zum Ziel transferiert werden kann. Die Bandbreite eines Verbindungsnetzes ist kein fester Wert, sondern ist abhängig von der Größe des zu transferierenden Datenblocks. Dies ist begründet durch die minimale Latenz, die zur Übertragung eines Datums von der Quelle zum Ziel benötigt wird.

2.2.1 Standardtechnologien

Ethernet *[18]* ist die Standardtechnologie im Bereich lokaler Netze. Es wurde über mehrere Generationen entwickelt und hat mittlerweile eine physikalische Datenrate von 1 Gbit/s (*Gigabit-Ethernet*, GE *[19]*) erreicht. Typischerweise ist jedoch *Fast Ethernet* (FE *[18]*) mit 100 Mbit/s im Einsatz. Damit werden unter Verwendung des TCP/IP Protokolls etwa 11 MB/s Bandbreite bei einer Latenz von etwas über $100\mu s$ erreicht.

Der Einsatz von GE bewirkt jedoch nur für große Datenmengen die Verzehnfachung der Bandbreite gegenüber FE. Die Ursache ist der hohen Softwareaufwand, den das verwendete TCP/IP Protokoll verursacht. Die Latenz von GE steigt gegenüber FE sogar an, da zur Erzielung einer höheren Bandbreite die Größe der Ethernet-Rahmen vervielfacht wurde. Eine Darstellung der erreichbaren Werte für Latenz und Bandbreite für TCP/IP-Kommunikation über FE und GE findet sich in Abb. 2.1.

Durch Umgehung des TCP/IP Protokolls und Einsatz einer speziellen Treibersoftware können mit Ethernetadaptern deutlich bessere Werte für Bandbreite und Latenz erreicht werden *[6,7,11,12]*. Problematisch ist bei diesen Verfahren jedoch, daß diese Softwareentwicklungen nur

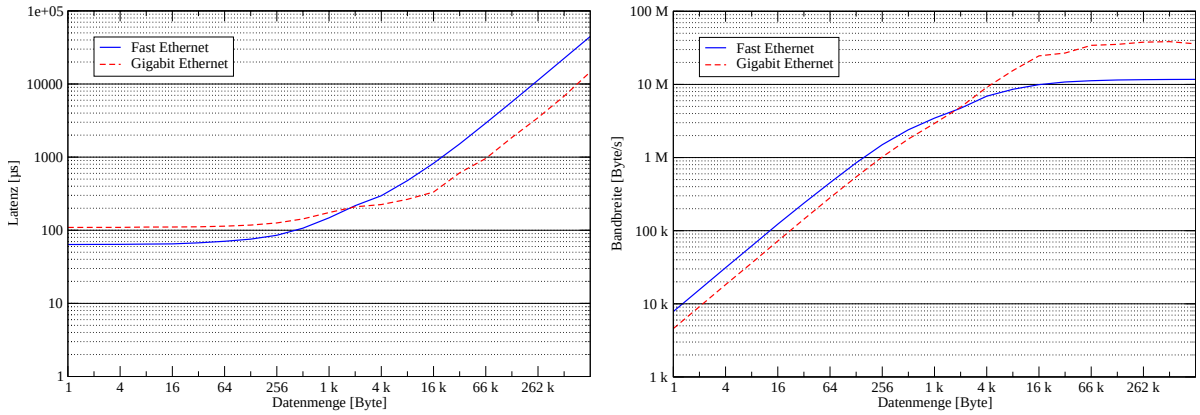


Abbildung 2.1: Latenz und Bandbreite für TCP/IP-Kommunikation zwischen zwei Knoten über *Fast-Ethernet* und *Gigabit-Ethernet* (*links*: Latenz. *rechts*: Bandbreite)
(P3-Plattform mit Intel EtherExpress Fast-Ethernet bzw. Gigabit Ethernet Adaptern)

mit bestimmten Typen von Ethernetadaptern zusammenarbeiten und häufig Einschränkungen bezüglich der Sicherheit, Kompatibilität, Mehrbenutzerfähigkeit und Stabilität haben. Diese Probleme scheinen durchaus relevant zu sein, da trotz der offensichtlichen Vorteile bezüglich der Leistung derartige Systeme nicht verbreitet im Einsatz sind.

Als Kommunikationsnetz zum Nachrichtenaustausch in Clustern ist Ethernet daher nur bedingt geeignet, wie etwa in *[155]* untersucht wurde. Nur besonders kommunikationsarme, rechenzeit-gebundene Applikationen erreichen damit ein gutes Skalierungsverhalten. Dieses wird dann jedoch zu sehr niedrigen Kosten und, aufgrund der vollen Integration des Netzes in das BS, mit einfacher Nutzung erzielt.

Bei höheren Anforderungen spielen jedoch zum einen der hohe zeitliche Aufwand zur Abarbeitung des TCP/IP-Protokolls und (bei FE) die relativ niedrige Kommunikationsleistung bereits auf der physikalischen Ebene eine Rolle. Weiterhin ist bei hoher Knotenzahl nicht mehr die Verbindung über einen einzigen Switch möglich, so daß mehrere Switches kaskadiert werden müssen. Neben der internen Bandbreite und Vermittlung der Switches kommt in diesem Fall die Bandbreite zwischen den Switches, die für einen derartigen Einsatz nur bedingt ausgelegt sind[1], als limitierender Faktor hinzu. Verfahren wie die *flat neighbourhood* Topologie *[20]* verbessern die Skalierbarkeit hinsichtlich der Bisektionsbandbreite. Dies ist jedoch mit höherem Hard- und Softwareaufwand verbunden, der sich in Anschaffungspreis und individueller Punkt-zu-Punkt-Kommunikationsleistung (weiter erhöhte Latenz) niederschlägt. Für Kommunikationsanforderungen, die mit dem standardmäßigem Einsatz von Ethernetnetzen nicht befriedigt werden können, sollten andere Technologien in Betracht gezogen werden, die im folgenden vorgestellt werden.

2.2.2 Systemnetze

Als Systemnetze werden Netztechnologien bezeichnet, die dazu bestimmt sind, räumlich dicht angeordnete, häufig gleichartige Knoten mit hoher Leistungsfähigkeit (hohe Bandbreite und niedrige Latenz) kommunizieren zu lassen. So können eng gekoppelte Prozesse auf diesen Knoten effizient ausgeführt werden. Die Bandbreite eines typischen Systemnetzes liegt in der Größenordnung der Bandbreite der E/A-Busse der Knoten, während die Latenz eine Größenordnung niedriger als bei der Standardtechnologie Ethernet ist.

2.2.2.1 Myrinet

Aus verschiedenen Forschungsvorhaben sowie Erfahrungen aus realisierten MPP-Systemen wurde Myrinet *[23]* entwickelt, das heute eines der erfolgreichsten Verbindungsnetze zum Aufbau von Clustern hoher Leistungsfähigkeit ist.

Es besteht aus Netzadaptern mit einem eingebetten Prozessor und eigenem Speicher, der die Generierung, Annahme und Weiterleitung von Datenpaketen mit Hilfe verschiedener DMA-Bausteine steuert. Damit sind DMA-Transfers zwischen dem Hauptspeicher des Knotens und dem lokalen Speicher des Netzadapters einerseits sowie dem Kommunikationsmedium und dem lokalen Speicher des Netzadapters andererseits möglich. Das Kommunikationsmedium überträgt die Daten in Form von beliebig langen Datenpaketen mit gegenwärtig 2 Gb/s physikalischer Linkbandbreite zwischen dem Netzadapter und einem Switchport.

Die Switches wiederum können zu verschiedenen Topologien verbunden werden; üblich sind hier *q-näre k-Bäume [21]*, typischerweise als *Clos*-Topologie, die bei relativ hohem Aufwand bezüglich der Zahl der Switchports die volle Bisektionsbreite zwischen beliebig gewählten

1. Die Verbindung mehrere Switches über den internen Bus ist nur für eine begrenzte Zahl von Switches möglich; zudem skaliert die maximale Gesamtbandbreite nicht mit der Zahl der Switches, da der interne Bus eine feste maximale Bandbreite hat. Für die effiziente Verbindung über mehrere normale Switchports fehlt häufig die geeignete Routingfähigkeit.

Knotenmengen gewährleistet. Die Wegewahl für die Pakete wird bereits am Quellknoten festgelegt, was ein einfaches, aber statisches Verfahren ist, das eine Kenntnis der gesamten Systemtopologie und aufwendige Maßnahmen bei dem Ausfall bzw. Wiedereintritt eines Knotens oder Switches in das System erfordert.

2.2.2.2 QS-Net

QS-Net [24] wurde von der Firma Quadrics basierend auf der Technologie des Verbindungsnetzes der Meiko Parallelrechner zum Aufbau von hoch-skalierenden Cluster-Systemen entwickelt. Es ist ein technisch ausgesprochen aufwendiges Verbindungsnetz und besteht aus Netzadaptern mit mehreren programmierbaren Mikroprozessoren und eigenem Speicher sowie komplexen Switches, die zu *fat tree [21]* Topologien verbunden werden. Die Netzadapter enthalten eine unabhängige Verwaltung des virtuellen Speichers des Knotens (gekoppelt an die Verwaltung des Betriebssystems), so daß vom Netzadapter direkt auf virtuelle Speicheradressen zugegriffen werden kann. Auf diese Art kann eine begrenzte Speicherkopplung der beteiligten Knoten erfolgen.

Die Integration von Quadrics-Systemen erfolgt stets durch den Hersteller. Insofern ist es kein typisches Produkt für den Einsatz in COTS-Clustern. Dies wird auch durch den hohen Preis deutlich, der etwa das Dreifache anderer Systemnetze beträgt.

2.2.2.3 Giganet cLan

Aufbauend auf ATM-Übertragungsprotokollen war Giganet *cLan* eines der ersten Verbindungsnetze, das mit Hardwareunterstützung für die Virtual-Interfache-Architektur (VIA *[9]*, siehe Kapitel 2.3.2) konzipiert wurde. Somit ist es als universelles Hochleistungsnetz mit jeder VIA-konformen Software verwendbar und erreicht Bandbreiten von bis zu 100 MB/s bei minimalen Latenzen von $30\mu s$.

Mittlerweile wird dieses Produkt nicht mehr als VIA-konformer Netzadapter angeboten, sondern wird nur noch als Komponente eines Massenspeichernetzes vertrieben. Es hat somit seine Bedeutung als universell nutzbares Verbindungsnetz verloren. Dies macht deutlich, daß die Anforderungen an eine standardisierte Schnittstelle für Hochleistungskommunikation in Clustern durch dieses System nicht erfüllt werden konnten. Wenn man den Erfolg des *Message Passing Interface* (MPI, siehe Kapitel 2.4.3) betrachtet, kann jedoch davon ausgegangen werden, daß für eine solche Schnittstelle auf Ebene der Verbindungsnetze durchaus ein Bedarf vorhanden ist. Denn genau wie MPI würde eine solche Schnittstellte den Einsatz einer einmal entwickelten Software auf einer Vielzahl von Systemen mit unterschiedlichen Verbindungsnetzen erlauben.

Daher muß angenommen werden, daß VIA, anders als MPI im Bereich der Programmiermodelle, keine geeignete Schnittstelle in diesem Sinne darstellt, oder daß Probleme mit den existierenden Implementationen von VIA einen breiteren Erfolg verhindert haben (siehe dazu auch *[10]*). Beide Annahmen treffen zu: Zum einen definiert die VIA-Spezifikation Verfahren, die einen generellen Einsatz in Clustern mit einer großen Anzahl von Knoten erschweren. Zum anderen ist die generische Verwaltung der Kommunikationsressourcen umständlicher als mit manchen Hardware-spezifischen Schnittstellen. Daneben enthält die VIA-Spezifikation, um flexibel zu bleiben, einige Spielräume durch Verwendung von Empfehlungen anstelle von Vorschriften, so daß sich der Funktionsumfang und die Nutzbarkeit einzelner VIA-Implementierungen stark unterscheiden können. Womöglich ist aber die Entwicklung der generischen Hochleistungs-Verbindungsnetze für Cluster noch nicht fortgeschritten genug, um eine allgemein effizient implementierbare Schnittstelle zu definieren.

2.2.3 Speicherkoppelnde Netze

Als *speicherkoppelnde Netze* werden Netze bezeichnet, bei denen der Kommunikationsvorgang zwischen zwei Prozessen auf entfernten Knoten aus transparenten Lade- bzw. Speicheroperationen auf bestimmte Adressen im Adreßraum eines Prozesses besteht. Diese

Kommunikation ist, sobald die Konnektivität zwischen den Prozessen hergestellt ist, die effizienteste Art des Datenaustauschs bezüglich des erforderlichen Aufwands für den Zugriff auf die Kommunikationsressourcen.

2.2.3.1 SCI

Das *Scalable Coherent Interface (SCI) [25]* nimmt unter den Verbindungsnetzen eine Sonderstellung ein, da es zunächst als IEEE-Standard definiert wurde, bevor es von verschiedenen Firmen in Hardware implementiert wurde. Weiterhin ist es ungewöhnlich flexibel, da es auf verschiedenen Ebenen eingesetzt werden kann: von der Integration von Komponenten auf einer Systemplatine über die Cache-kohärente Kopplung von SMP-Bausteinen *[46]*, transparente Kopplung von PCI-Bussen, Nutzung als Verbindungsnetz zum Aufbau von NUMA[1]-Systemen bis hin zur Implementierung von effizienten Systemen zum Nachrichtenaustausch, wie in dieser Arbeit vorgestellt. Eine ausführliche Darstellung der Eigenschaften von SCI insbesondere im Hinblick auf die Verwendung als Verbindungsnetz zum Nachrichtenaustausch wird in Kapitel 3 gegeben.

2.2.3.2 MemoryChannel

Der *MemoryChannel [22]* ist eine auf einem proprietären Protokoll basierende Netztechnologie, die ähnlich dem SCI-Standard eine Speicherkopplung herstellt. Jedoch ist dieses Protokoll nicht so generisch ausgelegt wie SCI, so daß der Einsatz auf Kommunikation zwischen Rechenknoten beschränkt ist. Die zugehörigen Produkte werden vom Hersteller (DEC, inzwischen Teil von Hewlett-Packard) nur in Komplettsystemen vertrieben und nur von eigener Software unterstützt, so daß es in COTS-Clustern keine Verwendung findet.

2.3 Kommunikationsmodelle

Kommunikation zwischen getrennten Rechnersystemen erfordert zunächst ein Netz, bestehend aus den physikalischen Leitungen und den angeschlossenen Verteilern (*Switches, Router*) und Endgeräten in den Rechnersystemen. Diese Endgeräte müssen in das Rechnersystem eingebunden werden, so daß Kern- und Benutzerprozesse darüber Daten versenden und empfangen können. Dazu muß weiterhin ein Protokoll vereinbart werden, nach dem Kommunikationskanäle auf- und abgebaut werden und Daten übertragen werden können.

Bezüglich der Einbindung der Endgeräte und des verwendeten Protokolls gibt es drei wesentliche Verfahren, die in Abb. 2.2 dargestellt sind und in den folgenden Abschnitten erläutert werden. Welches Prinzip genutzt werden kann, hängt von den Eigenschaften der Netztechnologie sowie des verwendeten BS ab. Grundsätzlich sind die verschiedenen Verfahren jedoch in ihrer Anwendbarkeit hierarchisch angeordnet: Systeme, auf denen ein direkteres (schlankeres) Verfahren möglich ist, können jeweils alle weniger direkten Verfahren durch zusätzliche Softwareschichten unterstützen. Umgekehrt ist dies nicht möglich.

2.3.1 Kommunikation durch Kerntreiber

Der übliche Ansatz zur Einbindung von Hardware (also auch der hier betrachteten Netzadapter*)* in ein BS besteht darin, die Möglichkeit des direkten Zugriffs auf die Hardware auf den BS-Kern zu beschränken (siehe Abb. 2.2 *a)*). Ein Benutzerprozeß muß für einen Zugriff eine Funktion des BS-Kerns aufrufen. Dadurch ist gewährleistet, daß kein Benutzerprozeß unzulässige Vorgänge auslöst (wie etwa den Zugriff auf beliebige physikalische Adressen durch einen DMA-Baustein), oder mehrere Benutzerprozesse gleichzeitig versuchen, dieselben Hardwareressourcen zu nutzen. Stattdessen leitet jeder Benutzerprozeß seine Sende- und Empfangsanfor-

1. *Non-Uniform Memory Access* - die Zugriffslatenz und -bandbreite einer CPU ist nicht für alle Bereiche des Adreßraums identisch. Speicher auf entfernten Knoten ist transparent, aber nur mit verringerter Leistung nutzbar.

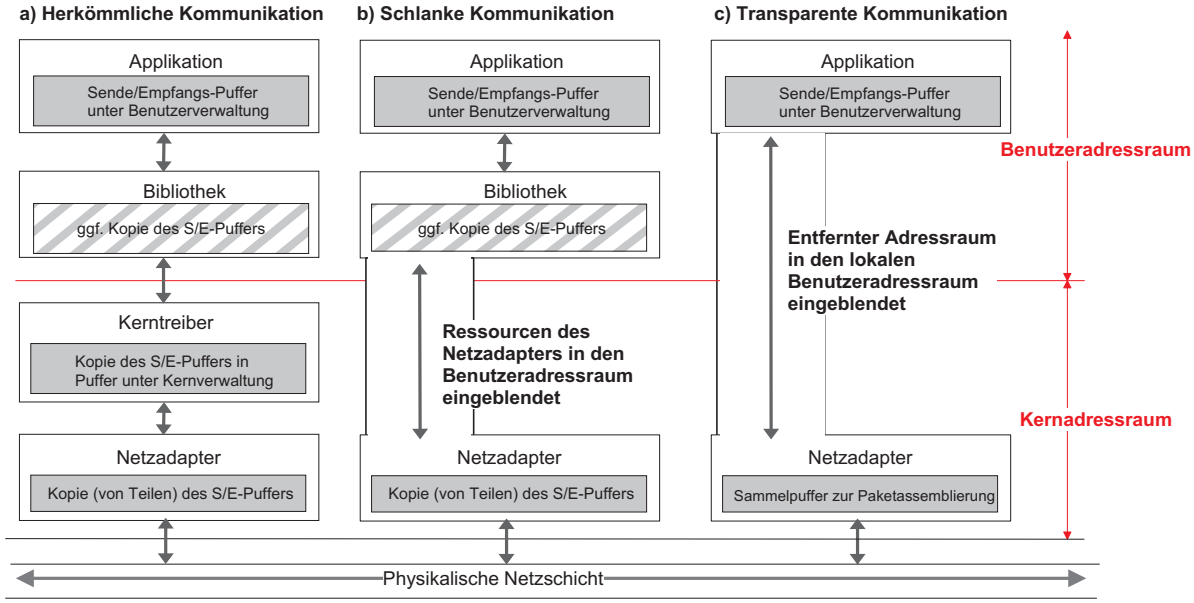


Abbildung 2.2: Verfahren für Interknoten-Kommunikation:
a) herkömmliche Kommunikation durch Kerntreiber
b) schlanke Kommunikation auf Benutzerebene
c) transparente Kommunikation durch Speicherkopplung

derungen durch eine Softwareschnittstelle zum Kern. Dort werden die eingehenden Daten vom Netzadapter (NA) den Empfangsanforderungen zugeordnet und die Sendeanforderungen durch Versand der angegebenen Daten nacheinander abgewickelt. Dies ist das übliche Kommunikationsverfahren für herkömmliche NA wie in Kapitel 2.2.1 vorgestellt.

Diese Vermittlung erfordert jedoch, daß die Daten mindestens einmal zwischen dem vom Benutzer mit der Sende- bzw. Empfangsanforderung bereitgestellten Puffer und dem Kernpuffer, auf den der NA zugreift, kopiert werden. Für komplexe, generische Kommunikationsprotokolle wie etwa TCP/IP können weitere Kopieroperationen notwendig werden. In jedem Fall verringern diese Kopieroperationen die effektive Bandbreite, die bei der Kommunikation über das Netz erreicht werden kann.

2.3.2 Schlanke Kommunikation

Je leistungsfähiger die unteren (Hardware-basierten) Schichten des Kommunikationsstapels werden, desto stärker fällt der (Software-basierte) Protokollaufwand in den oberen Schichten relativ ins Gewicht, indem die Latenz erhöht und die Bandbreite verringert wird. Zur Verbesserung der Kommunikationsleistung wurden Verfahren entwickelt, die auf *schlanken Protokollen* (Abb. 2.2 *b)*) basieren. Dabei wird nur noch zur Einrichtung von Kommunikationskanälen die Unterstützung des BS-Kerns benötigt, der die benötigten Ressourcen des NA in den Adreßraum des Prozesses einblendet. Die eigentliche Kommunikation, also der Datentransfer, kann vollständig innerhalb des Benutzeradressraums erfolgen *[41]*. Dies umgeht komplexe Protokolle im BS-Kern und ist daher leistungsfähiger als die beschriebene Kommunikation durch Kerntreiber. Jedoch liegt die Speicherpufferverwaltung und auch das eigentliche Protokoll, teilweise inklusive der Flußkontrolle, in der Verantwortung des Benutzerprozesses.

Eine der ersten Implementierungen eines solchen Verfahrens ist U-Net *[6,7]*. Dessen Prinzipien wurden später in die Definition der *Virtual Interface Architecture* (*VIA*) *[9]* eingebracht. Der VIA-Standard soll eine einheitliche Schnittstelle zur schlanken Kommunikation für unterschiedliche NA darstellen, so daß Applikationen und Bibliotheken, die schlanke Kommunikation nutzen, portabel werden. Der VIA-Standard ist teilweise jedoch nicht eindeutig definiert und enthält

konzeptionelle Probleme für die Skalierungsfähigkeit der Zahl der kommunizierenden Knoten *[10]*, so daß er sich gegenüber proprietären Schnittstellen noch nicht durchsetzen konnte. Auch der einzige spezifisch für VIA entwickelte NA *Giganet cLan* ist vom Markt verschwunden, so daß VIA zumeist nur noch optional als Softwareemulation (mit zumeist unvollständiger Implementation) zu den generischen Schnittstellen von NA verfügbar ist *[11,13]*. Alle in Kapitel 2.2.2 vorgestellten Systemnetze bieten das Verfahren der schlanken Kommunikation, werden aber in erster Linie über proprietäre Schnittstellen genutzt, die die speziellen Eigenschaften des jeweiligen NA am besten nutzbar machen.

Eine neue Entwicklung im Bereich der Verbindungsnetze und aufsetzender schlanker Kommunikation in Weiterführung der Prinzipien von VIA ist *Infiniband [42]*. Der Rückzug maßgeblicher industrieller Partner (Intel) aus der Entwicklung von Produkten dazu und die fortschreitende, kompatible Weiterentwicklung von PCI macht es jedoch zweifelhaft, ob sich Infiniband mittelfristig durchsetzen kann.

2.3.3 Transparente Kommunikation

Eine weitere Reduzierung des Protokollaufwands zur Kommunikation stellt die *transparente Kommunikation* (Abb. 2.2 *c)*) dar. Wie bei der schlanken Kommunikation wird zur Einrichtung eines Kommunikationskanals weiterhin die Unterstützung des BS-Kerns benötigt. Der Datentransfer erfordert dann jedoch keine Funktionsaufrufe mehr, da beim Einrichten des Kommunikationskanals die Speicherbereiche zum Empfang bzw. Versand von Daten auf dem entfernten Knoten in den Adreßbereich der anderen Prozesse eingeblendet werden. Der schreibende oder lesende Zugriff auf diese Daten kann durch gewöhnliche Speicher- bzw. Ladeoperationen der CPU erfolgen. Dies verringert insbesondere die Latenz für feingranulare Kommunikation und bewirkt, daß die maximale Bandbreite, die über den NA übertragen werden kann, schon für kleine Datenmengen erreicht wird (siehe Abb. 3.9 für Kommunikation über SCI).

Transparente Kommunikation kann über speicherkoppelnde Netze durchgeführt werden, wie sie in Kapitel 2.2.3 vorgestellt wurden. Auf diesem Kommunikationsverfahren basieren die grundlegenden in den Kapiteln 4, 5 und 6 vorgestellten Techniken zum Nachrichtenaustausch. Es wird jedoch auch schlanke Kommunikation verwendet, wenn deren Eigenschaften für den speziellen Anwendungsfall vorteilhaft sind, wie in Kapitel 7 und in Teilen von Kapitel 6 gezeigt wird.

2.4 Parallele Programmiermodelle

Während bei Problemen, die mit einem sequentiellen Programm gelöst werden sollen, bereits die Wahl der geeigneten Programmiersprache und Implementierungsebene (durch Festlegung auf die Nutzung bestimmter Programmierschnittstellen) komplex werden kann, ist der Parameterraum für diese Wahl bei einer mit einem parallel arbeitenden Programm zu lösenden Aufgabe noch weitaus größer[1].

2.4.1 Alternative Modelle

Ausgehend von den Merkmalen der zugrundeliegenden Hardwareplattformen, die insbesondere im Bereich der Kommunikation sehr unterschiedlich sein können, wurden und werden darauf abgestimmte Programmiermodelle entwickelt. Diese Modelle wiederum können über die vollständige Neudefinition einer Programmiersprache, über Erweiterungen bestehender Programmiersprachen oder über eine Programmierschnittstelle, die für eine oder mehrere Programmiersprachen definiert wird, nutzbar gemacht werden.

1. Verteilte Applikationen sollen hier nicht behandelt werden.

2.4.1.1 Gemeinsamer Speicher

Bei physikalisch gemeinsamem Speicher aller Prozessoren kann die Kommunikation und Synchronisation der Verarbeitungsstränge über Speicherstellen oder über die gemeinsame Instanz des Betriebssystems erfolgen. Gängige Programmiermodelle sind hierzu Threadparallelisierung (explizit durch den Programmierer oder implizit durch einen automatisch parallelisierenden Compiler) oder direktivengesteuerte Parallelisierung, wie sie durch den OPENMP Standard *[54]* als Spracherweiterung von Fortran und C definiert wird. Das Problem von Architekturen für gemeinsamen Speicher sind jedoch die relativ zur Zahl der CPUs superlinear steigenden Kosten, wenn für eine höhere Zahl von Prozessoren eine gleichbleibende, hohe Zugriffsgeschwindigkeit auf den gemeinsamen Speicher gewährleistet sein soll.

2.4.1.2 Virtuell gemeinsamer Speicher

Ein immer wieder unternommener Versuch, die beschränkte Skalierbarkeit von Systemen mit physikalisch gemeinsamem Speicher zu umgehen und dennoch das grundlegende Programmiermodell von gemeinsamem Speicher beizubehalten, ist die softwaremäßige Emulation eines gemeinsamen Adreßraums von Prozessen, die auf Knoten mit verteiltem Speicher ablaufen *[55,56]*. Diese Knoten sind über ein Netzwerk verbunden, über das sich Nachrichten verschicken lassen. Durch die Einblendung von Pseudospeicher in den Adreßraum der Prozesse wird der Anschein eines gemeinsamen Adreßraums aus der Sicht des Prozesses geschaffen. Die Ausnahmezustände, die ein Zugriff auf den Pseudospeicher durch einen Prozeß auslöst, werden softwaremäßig mittels Anforderung der fehlenden Daten von einem anderen Knoten über das Netzwerk aufgelöst. Diese Ansätze sind aber umso ineffizienter, desto transparenter sie arbeiten. Die Gewährleistung *sequentieller Konsistenz* bedingt starken Protokollaufwand zwischen den Knoten mit verteiltem Speicher. Weniger strikte Konsistenzmodelle bewirken unter bestimmten Umständen Leistungssteigerungen durch Reduzierung der Kommunikation. Sie erfordern andererseits aber mehr Rücksichtnahme und Kenntnis durch den Programmierer. Hinzu kommt die, verglichen mit der Zykluszeit einer CPU, extrem langsame Bereitstellung der Daten von entfernten Systemen. Entgegen dem eigentlich transparenten Ansatz, den das Modell verfolgt, ist es für den Programmierer zum Erreichen hoher Rechenleistung unabdingbar, sich über die Auswirkungen der einzelnen Zugriffe auf gemeinsame Datenstrukturen im klaren zu sein. Dies wird jedoch gerade durch transparente Abwicklung erschwert. Daher erwies sich dieses Programmiermodell nur in wenigen Fällen als effizient. Die Unterstützung durch Erweiterung einer Programmiersprache ist geeignet, die Effizienz zu erhöhen. Dies wurde etwa bei der Erweiterung von Fortran 77 zu SVM-Fortran dargestellt *[57]*, und auch OPENMP arbeitet nach diesem Prinzip. Die Akzeptanz solcher Erweiterungen ist trotz der Vorteile, etwa der Parallelisierung nur einzelner Abschnitte eines Programms ohne eine explizite Datenverteilung, jedoch gering. Die Ursache dafür muß vorwiegend in der erzielten Leistung einer Applikation auf einem gegebenen System gesucht werden, deren Maximierung vorrangiges Ziel ist. Erst OPENMP hat durch breitere Sprach- und Herstellerunterstützung eine signifikante Verbreitung gefunden, wird jedoch vorwiegend auf Systemen mit physikalisch gemeinsamem Speicher eingesetzt.

2.4.2 Nachrichtenaustausch

Unter dem Eindruck der nur unter hohem Kosteneinsatz gegebenen Skalierbarkeit von Systemen mit physikalisch gemeinsamem Speicher mit UMA oder CC-NUMA Charakteristik und der für die meisten Applikationen geringen Effizienz von Modellen mit virtuell gemeinsamem Speicher hat das Modell des Nachrichtenaustauschs große Bedeutung gewonnen. Es basiert auf der Vorstellung von verteiltem Speicher. Da die Daten dabei explizit zwischen den Prozessen partitioniert werden, entsteht in vielen Fällen eine höhere Lokalität. Die explizite Kommunikation in Form von Nachrichtenaustausch ermöglicht eine Implementation auf allen Plattformty-

pen, insbesondere auf Systemen mit physikalisch verteiltem Speicher. Die Aufteilung der Prozessor-Speicher-Module, innerhalb derer ein Prozeß abläuft, auf unabhängige Knoten erlaubt eine lineare Skalierung des Gesamtsystems mit einem festen Verhältnis von CPU- zu Speicherleistung bei gleichzeitig ebenfalls linearem Kostenverlauf. Die Kosten pro CPU liegen bei diesem Modell niedriger als bei SMP-Systemen mit entsprechender Zahl von CPUs, da Komponenten vom Massenmarkt verwendet werden können und die hohen Basiskosten für ein entsprechendes SMP-System (Gehäuse, Speicher-Interconnect, skalierte E/A-Anbindung) nicht anfallen.

2.4.3 Message Passing Interface

Entscheidend für den Erfolg (gemessen an der Häufigkeit der Verwendung) eines Programmiermodells ist die Existenz eines Standards, der eine Programmierschnittstelle festlegt. Damit sich eine Programmierschnittstelle als Standard etablieren kann, muß sie auf möglichst vielen Plattformen in einer effizienten und standard-konformen Implementation vorliegen und damit die Portierbarkeit für darauf aufbauende Bibliotheken und Anwendungen gewährleisten. Zum Zeitpunkt der Definition des *Message Passing Interface* (MPI) gab es eine Reihe weiterer Schnittstellendefinitionen (*Application Programming Interface*, API) und Bibliotheken zum Nachrichtenaustausch (Chameleon, CHIMP, CMMD, Express, MPL, NX, P4, PARMACS, PVM und weitere). Diese APIs waren entweder auf bestimmte Hardwareplattformen zugeschnitten (z.B. NX für Intel Paragon Systeme, CMMD für Thinking Machine CM-5) oder möglichst flexibel und dynamisch für den Einsatz auf verteilten, heterogenen Systemen (PVM, PARMACS) einsetzbar, was jedoch aufgrund des höheren Softwareaufwands bei der Kommunikation (Verwendung von Routerprozessen, Datenkonvertierung) mit sub-optimaler Leistung einhergeht. Auch in weiteren Merkmalen wie komplexe kollektive Operationen, Definition von Prozeßgruppen oder Nachrichtenkontexten, Vereinbarung von virtuellen Topologien, Einbindung von dynamischer Prozeßverwaltung oder paralleler Ein-/Ausgabe unterschieden sich die vorhandenen APIs. MPI mußte möglichst alle Vorteile und speziellen Eigenschaften dieser Systeme erhalten, eine effiziente Implementierung auf allen Plattformen erlauben und gleichzeitig portabel bleiben, um von den Nutzern und Herstellern angenommen zu werden. Daher wurde vom dazu eingerichteten *MPI Forum* *[63]* unter Beteiligung möglichst vieler interessierter Parteien mit der Entwicklung einer Standardschnittstelle begonnen *[61]*.

Der entscheidende Schritt gelang 1994 mit der Verabschiedung des *Message Passing Interface* (MPI) in der Version 1.0. MPI hat sich seitdem fest etabliert und ist der Standard zur Programmierung von Parallelrechnern im technisch-wissenschaftlichen Bereich geworden. Keine parallele Plattform wird heute mehr ohne eine angepaßte Implementation von MPI ausgeliefert. Die Entwicklung von MPI selber ist nach 1994 weitergegangen, über kleinere Anpassungen und Klarstellungen (Standard der Version 1.1) hin zur Version 2 *[64,65]*. Diese enthält Korrekturen und Ergänzungen des nunmehr als MPI-1 bezeichneten MPI-Standards (Version 1.2). Vor allem definiert dieser Standard eine Zahl von Erweiterungen, in ihrer Gesamtheit als MPI-2 bezeichnet, um die Bereiche parallele Ein-/Ausgabe, einseitige Kommunikation (siehe Kapitel 5.2), erweiterte Datentypen und kollektive Operationen, Prozeßverwaltung, verbesserte Sprachanbindung sowie einige andere Details.

Neben diesem Hauptstandard von MPI wurden auch spezielle Ausrichtungen und Erweiterungen im Zusammenhang mit MPI definiert und z.T. implementiert. Hier sind zu nennen *Realtime-MPI* (MPI/RT *[67]*) zur Kommunikation unter (weichen) Echtzeitbedingungen, *Fault-Tolerant MPI* (FT-MPI *[68]*) als Spezifikation eines MPI, das den Ausfall von Prozessen toleriert, sowie *Interoperable MPI* (IMPI *[69,70]*), eine Spezifizierung der untersten Kommunikationsebene, um die direkte Kopplung von Systemen mit unterschiedlichen MPI-Implementationen zu ermöglichen. Diese Varianten bzw. Erweiterungen bauen auf den Prinzipien von MPI auf und erweitern

es um für den jeweiligen Einsatzzweck erforderliche Eigenschaften.

Die Gründe dafür, daß MPI sich schnell zum Standard entwickelt hat, sind vielfältig. Zunächst einmal war von Seiten der Nutzer, also der Programmierer von parallelen Applikationen, der Bedarf für eine portable Schnittstelle vorhanden. Trotz der umfangreichen Schnittstelle[1] läßt sich ein MPI-Programm, das eine einfache Kommunikationssemantik aufweist, mit einer Handvoll von MPI-Funktionen erstellen. Die große Zahl von Funktionen kommt daher, daß für nahezu jede denkbare Kommunikationssituation und -abfolge eine spezielle Sende- und Empfangsfunktion sowie verschiedene Möglichkeiten zur Eingangs- und Fertigstellungsprüfung zum Empfang respektive Versand von Nachrichten definiert sind. Dies ermöglicht zum einen dem Hersteller eines Systems, mittels dieser Funktionen optimierte Kommunikationsleistung bereitzustellen (siehe Kapitel 7.2), zum anderen kann der Benutzer hiermit auch komplexe Kommunikationsmuster abbilden. Somit wird durch MPI weder der Benutzer noch der Hersteller eines Systems eingeschränkt. Andererseits ist in jedem Fall die volle Portabilität einer MPI-Applikation gewährleistet, da die erwähnten speziellen Kommunikationsfunktionen nicht zwangsläufig mit den entsprechenden speziellen Eigenschaften ausgestattet sein müssen, sondern ggf. auch leicht auf die Standardfunktionen abgebildet werden können. Dies vereinfacht die Implementation von MPI auf vielen Plattformen. Zu dieser einfachen (initialen) Implementation und damit zur starken Verbreitung von MPI trug auch die schon kurz nach der Verabschiedung des Standards frei verfügbare und portable MPI-Implementation MPICH *[72,71]* bei. Diese kann von Herstellern genutzt werden, um MPI auf ihren Systemen anzubieten, da dazu nur die unterste Schicht mit den grundlegenden Kommunikationsfunktionen angepaßt werden muß.

Eine solche Anpassung kann mit wenig Aufwand erfolgen, wenn die neue Zielplattform bereits ähnliche Methoden des Nachrichtenaustauschs anbietet. Für die Integration der NX-Schnittstelle in MPICH waren beispielsweise nur etwa 100 Zeilen Code erforderlich, der die unteren Sende- und Empfangsroutinen von MPICH auf die NX-Schnittstelle abbildet. Diese Art der Portierung von MPICH auf eine neue Plattform liefert schnell eine vollständig nutzbare MPI-Schnittstelle. Zur optimalen Nutzung der Systemressourcen, insbesondere des Verbindungsnetzes, ist jedoch eine weitergehende, spezifische Anpassung notwendig. Dies gilt umso mehr, wenn das Verbindungsnetz Möglichkeiten bietet, die über das einfache Senden und Empfangen von Datenblöcken (mittels Funktionen wie `send()` und `recv()`) hinausgehen.

2.5 Motivation

Die vorhergehenden Ausführungen machen deutlich, daß kostengünstiges Hochleistungsrechnen am ehesten durch Nutzung von Rechnerverbundsystemen (*Clustern)* erreicht werden kann, auf denen Anwendungen über Nachrichtenaustausch kommunizieren. Entscheidend für die Gesamtleistung für alle nicht trivial-parallelen Anwendungen ist bei einem derartigen System, daß die Leistung der Komponente zum Nachrichtenaustausch proportional zu der akkumulierten Rechenleistung der Knoten skaliert. Gleichzeitig soll die absolute Leistung des Nachrichtenaustauschs auf einem Niveau liegen, das die Ausführung einer parallelen Applikation in Relation zur theoretischen Maximalleistung so wenig wie möglich bremst. Dabei müssen als weiterer Faktor die Kosten berücksichtigt werden: idealerweise sollen die Kosten für die Kommunikationskomponente ebenso linear mit der Zahl der Knoten steigen wie die Kosten für die Knoten selber.

Rechnerverbundsysteme, die mittels SCI über eine Speicherkopplung verfügen, entsprechen diesen Kriterien sowohl von der grundlegenden Kommunikationsleistung als auch von der Kostenskalierung. Dabei wird SCI *nicht* als Grundlage für ein System verwendet, auf dem ein Pro-

1. MPI in der Version 1.0 definiert mehr als 130 Funktionen, 90 Konstanten und 7 Datentypen (in C). Version 2.0 definiert mehr als 200 Funktionen und 100 Konstanten.

grammiermodell des gemeinsamen Speichers verwendet werden soll, wie dies in *[50,51,147]* geschah. Der hohe Aufwand, der in diesen Fällen betrieben werden mußte, um eben *keine* Zugriffe auf entfernten Speicher durchzuführen, hat die Problematik dieses Ansatzes verdeutlicht. Die gleiche Problemstellung (Simulation von Molekulardynamik), die in *[147]* gelöst wurde, wurde mehrfach auch mit Anwendungen basierend auf dem Modell des Nachrichtenaustauschs mit überzeugender Skalierungscharakteristik gelöst *[148,149]*.

Diese Arbeit präsentiert und untersucht daher Verfahren, die entwickelt wurden, um die hardwaremäßige, durch die SCI-Kopplung verfügbare Kommunikationsleistung im größtmöglichen Maß für die Kommunikation mittels MPI zu nutzen. Dazu gehört auch, die Skalierung der Gesamtleistung der MPI-basierten Kommunikation mit der Zahl der Knoten zu gewährleisten.

KAPITEL 3

Speichergekoppelte Rechnerverbundsysteme

Unter den Rechnerverbundsystemen nehmen die Systeme mit *Speicherkopplung* eine besondere Stellung ein. Mit Speicherkopplung wird die Fähigkeit bezeichnet, von einem Knoten aus ohne Mitwirkung von Software Speicherstellen auf einem anderen Knoten zu lesen oder zu schreiben. Dies steht im Gegensatz zu rein *nachrichtengekoppelten* Rechnerverbundsystemen, bei denen der Austausch von Informationen zwischen entfernten Knoten nur über den Versand und Empfang von Nachrichten möglich ist, die der Sender zusammenstellen muß und die vom Empfänger weiterverarbeitet werden müssen. Neben proprietären Lösungen wie MemoryChannel *[22]* oder Cray SHMEM *[94]* gibt es einen maßgebenden Standard zur Speicherkopplung: *IEEE 1596 Scalable Coherent Interface* (*SCI*).

3.1 SCI Standard und Implementierung

Der SCI Standard entstand aus der Notwendigkeit, ein Verbindungsnetz ohne inhärente Skalierungsschranken (wie bei Bus-basierten Netzen), aber dennoch mit Bus-typischen Eigenschaften wie transparentem und optional Cache-kohärentem Zugriff auf entfernte Speicherbereiche zu entwerfen. SCI wurde entwickelt, um in einem Multiprozessorsystem die Prozessoren, Speicher, E/A-Geräte und Busse miteinander zu verbinden. Die bislang verwendeten Backplane-Busse waren dazu insbesondere durch die steigende Prozessorleistung nicht mehr in ausreichendem Maße in der Lage *[26]*.

3.1.1 IEEE 1596 Standard

Der resultierende Standard wurde 1992 als IEEE Standard Nummer 1596 verabschiedet *[25]*. Dieser Standard definiert die Protokolle und Paketformate zum Datentransport, das Cachekohärenz-Protokoll und die physikalischen Ebenen der Signalisierung und Verbindungshardware.

Die wichtigsten Eigenschaften der SCI-Protokolle sind:

- Physikalische Punkt-zu-Punkt Verbindungen, um so eine einfache Arbitrierung des Mediums auch bei hohen Frequenzen zu ermöglichen. Aus dieser Basistopologie und geeigneten Switches lassen sich alle weiteren Topologien ableiten.
- Inhärente Skalierbarkeit durch den Aufbau aus unabhängigen Punkt-zu-Punkt Verbindungen, da jede neue Verbindung zu einem weiteren Endgerät die verfügbare Gesamtbandbreite erhöht und es keine zentralen Punkte gibt, die alle Daten passieren müßten.
- Pakete von verschiedener, kurzer Länge (max. 256 Byte Nutzlast), um niedrige Latenzen zu gewährleisten und Sättigung eines Links durch eine einzige Datenquelle zu vermeiden (siehe Abb. 3.1, links).
- *Split Transactions* (geteilte Transaktionen, siehe Abb. 3.1, rechts), um die Effizienz der Nutzung der verfügbaren Bandbreite zu erhöhen: Zwischen der Initiierung und der Bestätigung einer Transaktion (etwa bei einem Schreibzugriff: Senden der Daten in einem *write*-Paket und Empfangsbestätigung durch ein *echo*-Paket) wird das Medium für andere Transaktionen freigegeben.
- Globaler 64-bit Adreßraum, in den die Adreßräume der angeschlossenen Endgeräte eingeblendet werden können. Somit wird der transparente Zugriff auf entfernten Speicher ermöglicht, wie in Abb. 3.2 dargestellt: Prozeß p_1 auf Knoten K_1 exportiert ein lokales Speichersegment, daß aus dem lokalen physikalischen Adreßraum mittels der *Memory Management Unit (MMU)* in seinem virtuellen Adreßraum eingeblendet wird (MAP), in den globalen SCI-Adreßraum. Dabei werden die lokalen physikalischen Adressen durch Addition der 16 Bit

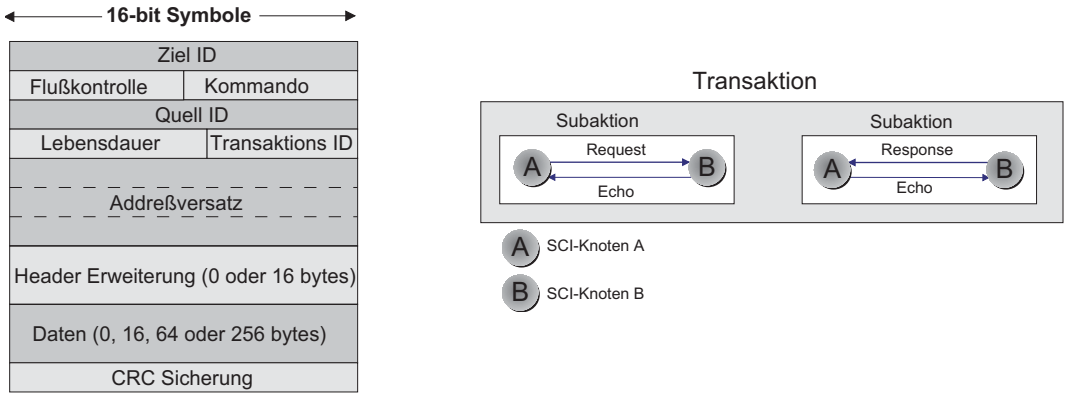


Abbildung 3.1: *Links:* Aufbau von SCI-Paketen
Rechts: Geteilte Transaktionen *(split transactions)* bei SCI

breiten Knotenkennung (ID) in den 64 Bit breiten SCI-Adreßraum transformiert. Von dort importiert Prozeß p_2 auf Knoten K_2 das Speichersegment und blendet es in den lokalen physikalischen Adreßraum ein. In diesem Fall findet die Transformation der Adressen aus dem SCI-Adreßraum in den lokalen physikalischen Adreßraum durch die in der SCI-Hardware enthaltene Adreßübersetzungstabelle (*Adress Translation Table*, ATT) seitenweise statt. Am Ende wird der Speicherbereich wiederum mit Hilfe der MMU in den Prozeßadreßraum eingeblendet, so daß Prozeß p_2 transparent auf den entfernten Speicher zugreifen kann.

3.1.2 SCI Linkcontroller

Die Implementierung von SCI besteht maßgeblich aus zwei Komponenten: die Anbindung eines Endgeräts an das SCI Netz durch einen sogenannten *Linkcontroller* (LC) sowie die Integration dieses Linkcontrollers in das interne Kommunikationssystem des Endgeräts. Diese Aufteilung in zwei Komponenten ist sinnvoll, da die Aufgaben des Linkcontrollers für verschiedenartige Endgeräte gleich bleiben (die Verarbeitung von SCI-Paketen), wohingegen die Anbindung des Linkcontrollers abhängig ist vom internen Aufbau des Endgerätes.

Der erste kommerziell verfügbare Linkcontroller wurde von *Dolphin Interconnect Solutions* in Oslo, Norwegen entwickelt (die auch heute noch den weit überwiegenden Teil von SCI-Komponenten entwickeln und fertigen). Die Entwicklung ging über mehrere Generationen (LC-1 und LC-2) bis zum derzeit verfügbaren LC-3 *[31]*.

Der prinzipielle Aufbau des LC-3 ist in Abb. 3.3 dargestellt. Auf der Knotenseite kommuniziert der LC über einen 64-bit breiten Bus (B-LINK), der die SCI-Pakete in einem SCI-ähnlichen Format überträgt. An den B-LINK können bis zu 8 Geräte angeschlossen werden: ein Schnittstel-

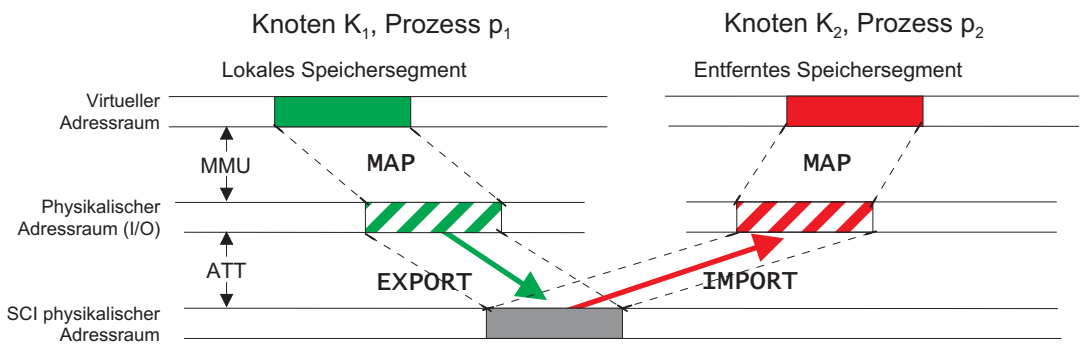


Abbildung 3.2: Prinzip der transparenten Kommunikation über eingeblendete Speichersegmente

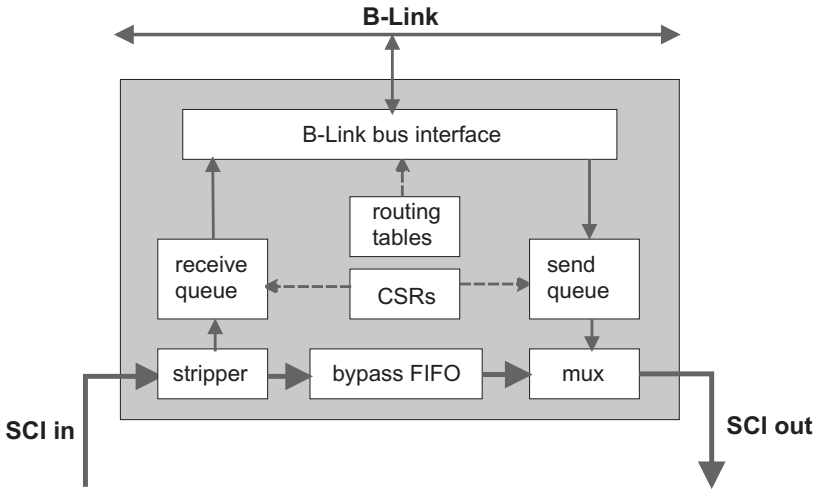


Abbildung 3.3: Blockschaltbild eines SCI Linkcontrollers (Dolphin Architektur, nach *[31]*)

lenbaustein für die Kommunikation mit dem Knoten und mindestens ein Linkcontroller für den Zugriff auf das SCI-Netz. Mehr als ein Linkcontroller ist erforderlich für den Aufbau mehrdimensionaler SCI-Topologien, in denen Pakete von einer Dimension in eine andere geleitet werden müssen. Dies bedeutet das Weiterleiten eines Pakets vom Eingang eines Linkcontrollers über den B-LINK zu einem anderen Linkcontroller und von dort auf den SCI-Link einer anderen Dimension. SCI-seitig sind ein Eingang und ein Ausgang für den SCI-Link vorhanden. Am Eingang werden die Adressen der eingehenden Pakete dekodiert (*stripper)*. Pakete, die gemäß ihrer Ziel-ID für das lokale Endgerät bestimmt sind oder entsprechend der Wegetabelle *(routing table)* in diesem Knoten auf einen anderen SCI-Link geleitet werden müssen, werden in den Eingangspuffer *(receive queue)* geschrieben. Alle anderen Pakete kommen in den Bypasspuffer *(bypass* FIFO), um von dort über einen Multiplexer zusammen mit abgehenden Paketen aus dem Ausgangspuffer *(send queue)* zum nächsten Endgerät weitergeleitet zu werden. Die Pakete im Eingangspuffer werden gemäß der Einstellungen in den Konfigurationsregistern (CSR) behandelt und auf den B-LINK gelegt, von wo sie entweder auf den System- bzw. E/A-Bus des Knotens oder von einem anderen Linkcontroller auf einen anderen SCI-Link geleitet werden. Jeder LC wiederum liest vom B-LINK Pakete, die gemäß ihrer Ziel-ID und der *routing table* auf den betreffenden SCI-Link des LC geleitet werden müssen. Der LC-3 kann den SCI-Link mit Taktfrequenzen zwischen 100 und 250 MHz und den B-LINK mit 80 Mhz betreiben.

3.1.3 Integrierte Systeme

Auf Basis der Linkcontroller produzieren verschiedene Hersteller integrierte Multiprozessorsysteme, in denen mehrere Speichermodule über SCI zu einem globalen cache-kohärenten Hauptspeicher mit nicht-uniformer Zugriffscharakteristik (CC-NUMA) verbunden werden. Das erste verfügbare System war die *Convex Exemplar*, die später von Hewlett Packard zur *X-Serie* weiterentwickelt wurde. Weitere Systeme wurden von Data General (heute IBM) *[46,47]* und Sequent entwickelt. Sie sind vorwiegend für Datenbanksysteme konzipiert, die von dem großen gemeinsamen Speicher profitieren. Der Aufbau all dieser Systeme ist ähnlich: mehrere Prozessoren teilen sich ein lokales Speichermodul, das wiederum aus mehreren Bänken besteht (Steigerung der Bandbreite durch Interleaving). Mehrere solcher Prozessor-Speicher-Module werden wiederum über unabhängige SCI-Verbindungen zusammengeschaltet. Hierbei kommt das verteilte SCI-Cachekohärenz-Protokoll zum Einsatz. Allerdings wirkt dieses nicht direkt auf die Caches der Prozessoren, sondern wird innerhalb eines speziellen Caches für entfernten Speicher, der Teil des lokalen Hauptspeichers ist, abgewickelt, so daß für alle Prozessoren ein

gemeinsamer Cache für entfernten Speicher zur Verfügung steht. Dies vereinfacht den Aufbau des Speichersystems und verhindert zusätzliches Datenaufkommen durch Cachekohärenz-Datenverkehr auf dem Bus zwischen lokalem Speicher und Prozessoren.

3.1.4 PCI-SCI-Adapter

Integrierte Systeme auf Basis von SCI haben überzeugende Leistungs- und Systemeigenschaften, sind jedoch aufgrund der geringen Stückzahl und der hohen Entwicklungs- und Fertigungskosten vergleichsweise teuer. Die Anschaffung lohnt sich also nur für Nutzer, die genau diese Eigenschaften benötigen, um Anwendungen zu realisieren, die eine entsprechende Wertschöpfung haben oder die auf anderen Systemen nicht umgesetzt werden können. Dies kann z.B. an ungenügender Speicherkapazität, der zwingenden Verfügbarkeit von physikalisch gemeinsamen Speicher oder mangelnder Verfügbarkeit liegen. Ein Beispiel für diesen Fall sind hochverfügbare Datenbankanwendungen.

Viele Nutzer benötigen jedoch nicht das gesamte Spektrum an Eigenschaften für ihre Anwendungen, oder die Wertschöpfung der Anwendungen soll durch niedrigere Kosten erhöht werden. Aus Standardkomponenten gebaute System auf Basis von PC-Technologie bieten in bestimmten Bereichen ausreichende Leistung zu sehr niedrigeren Kosten. Der Standard-E/A-Bus bei PCs ist der PCI-Bus, so daß zur Nutzung von SCI mit PCs ein PCI-SCI-Adapter (PSA) die universellste Lösung ist.

3.1.4.1 Aufbau und Einbindung

PCI-SCI-Adapter (PSA) werden ebenfalls von der Firma Dolphin entwickelt *[27]*. Die Anbindung des Linkcontrollers an den PCI-Bus erfolgt dabei über eine PCI-SCI-Bridge (PSB)[1] *[28,29,30]*, die zusammen mit einem oder mehreren SCI-Linkcontrollern einen PSA bildet (siehe Abb. 3.4).

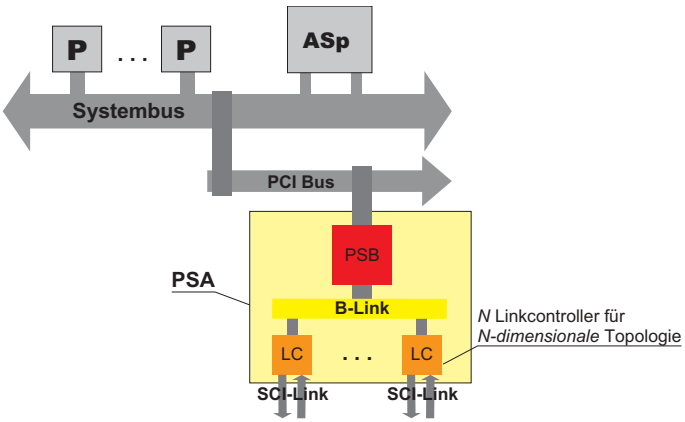


Abbildung 3.4: Anbindung eines Rechenknotens an SCI mittels PSA (PCI-SCI-Adapter)

Die PSB setzt eine oder mehrere ausgehende PCI-Transaktionen in SCI-Transaktionen um. Diese können aus einem oder mehreren SCI-Paketen bestehen. Umgekehrt generiert die PSB aus eingehenden SCI-Paketen PCI-Transaktionen. Bei ausgehenden Datentransfers transformiert die PSB die Adressen mittels des ATT zwischen dem lokalen PCI-Adreßraum (der wiederum in den physikalischen Adreßraum der Prozessoren eingeblendet ist) und dem globalen SCI-Adreßraum. Um die Umwandlung zwischen PCI- und SCI-Transaktionen möglichst effizient zu vollziehen, wird eine Reihe von Verfahren wie *Agressive Prefetching, Speculative Read,*

1. Obwohl der PSB keine direkte SCI-Anbindung hat (sondern an den B-LINK angeschlossen wird), wurde diese Bezeichnung gewählt, da über den B-LINK mit einer übergeordneten Form von SCI-Paketen kommuniziert wird. Dies ist nicht zu verwechseln mit einem PCI-SCI-*Adapter*, der die gesamte Anbindung von PCI-Bus an den wirklichen SCI-Link darstellt.

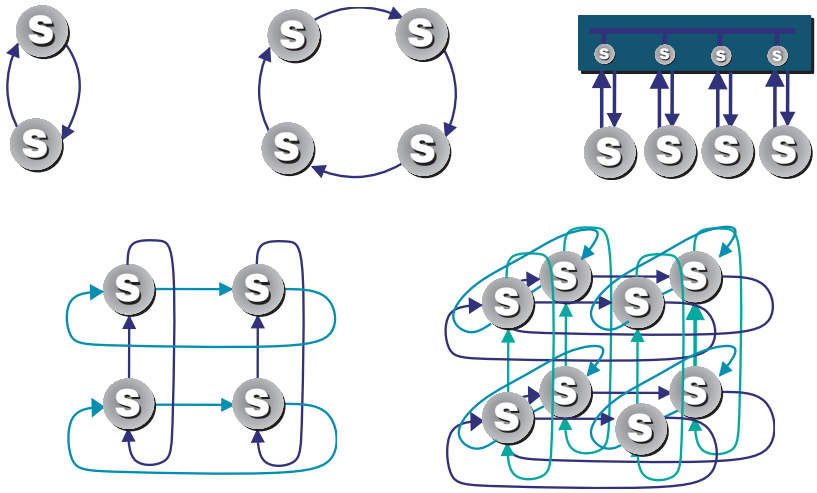


Abbildung 3.5: Mögliche Topologien von SCI-verbundenen Knoten.
V.l.n.r.: minimale 2-Knoten Verbindung, Ring, Switch, 2-D Torus, 3-D Torus

Write Gathering und *PCI Burst Transfers* *[27]* eingesetzt. Neben der Umsetzung von PCI-Transaktionen, die von anderen PCI-Geräten oder den Prozessoren generiert wurden, kann der PSB mittels einer DMA-Einheit auch eigenständig Datentransfers vom lokalen Adreßraum in den globalen SCI-Adreßraum (oder auch wieder in den lokalen Adreßraum) durchführen. Der PSB wird über einen umfassenden Satz von Kontrollregistern gesteuert, die gemäß dem CSR-Standard IEEE 1212-1991 (*Standard for Control- and Status Register Architecture for Microcomputer Buses*) organisiert sind. *[27]* bietet eine umfassende Beschreibung des Aufbaus von PCI-SCI-Adaptern der Firma Dolphin.

3.1.4.2 Topologien

Ein Knoten kann mehrere PSA beinhalten, und jeder PSA kann wie beschrieben mehrere LCs enthalten. Die Adressierung von Paketen erfolgt mittels der eindeutigen Kennung (*Adapter*-ID) jedes PSA im SCI-Netz. Mit den Punkt-zu-Punkt-Verbindungen zwischen zwei LCs können verschiedene Topologien aufgebaut werden (siehe Abb. 3.5). Diese Topologien unterscheiden sich zunächst nicht in ihrer Funktionalität, da die Topologie für die Anwendung transparent ist. Jedoch wird erst mit der Switch-Topologie eine Ausfallsicherheit eingeführt, bei der der Ausfall eines Knotens nicht zur Unterbrechung der gesamten SCI-Kommunikation führt. Bei Ausfall einer beliebigen Zahl von Knoten bleiben alle anderen Knoten weiterhin erreichbar, ohne daß Modifikationen in den einzelnen LCs erfolgen müssen.

Dadurch, daß ein SCI-Knoten mehrere LC betreiben kann, sind mehrdimensionale Topologien wie zwei- und dreidimensionale Tori möglich. Auch bei den mehrdimensionalen Torus-Topologien ist ein Knoten über mehr als einen Weg erreichbar, so daß hier ebenfalls eine Ausfallsicherheit erreicht werden kann. Dazu muß bei Ausfall eines Knotens das Routingverfahren angepaßt werden, wodurch je nach Dimension des Torus der Ausfall einer bestimmten maximalen Zahl von Knoten bei bestehender Erreichbarkeit aller anderen Knoten kompensiert werden kann.

3.2 Softwarearchitektur

Zur Nutzung der vom PSA bereitgestellten Ressourcen stehen Softwareschnittstellen zur Verfügung. Diese sind auf verschiedenen Ebenen des Betriebssystems angelegt und unterscheiden sich in der Abstraktion der Funktionalität, der Art der Kommunikation mit dem Nutzer und dem Kontext, in dem sie ablaufen. Ein Überblick der genutzten Infrastruktur ist in Abb. 3.6 gegeben.

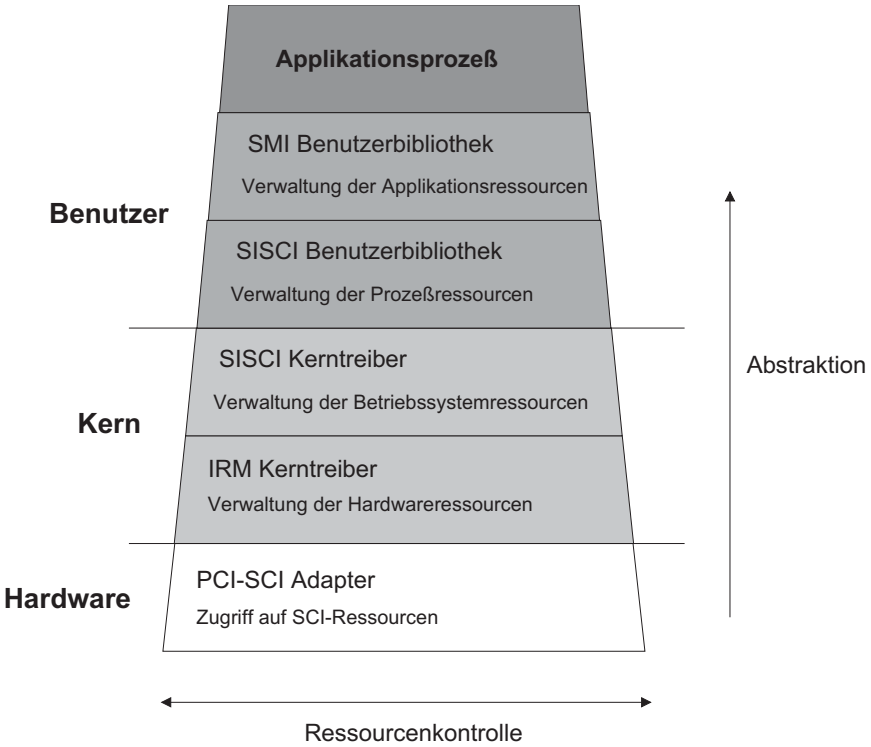


Abbildung 3.6: SCI Softwarearchitektur (Dolphin / Lehrstuhl für Betriebssysteme)

3.2.1 IRM Kerntreiber

Die Einbindung der Ressourcen der in einem Rechensystem installierten PSA erfolgt über einen Gerätetreiber im Kern des Betriebssystems, der als *Interconnect Ressource Manager* (IRM) bezeichnet wird. Seine Aufgaben sind die Verwaltung der Ressourcen des PSA und die Bereitstellung einer hardwareunabhängigen Schnittstelle zum Zugriff auf diese Ressourcen. Der IRM Treiber ist größtenteils unabhängig vom Betriebssystem; nur der Zugriff auf die PCI-Ressourcen des Systems und die Anforderung von Speicherplatz erfordert die Nutzung von Betriebssystemfunktionen.

3.2.2 SISCI Kerntreiber und Benutzerbibliothek

Die übergeordnete Steuerung von komplexeren Vorgängen auf SCI-Ebene, die eine erweiterte Unterstützung durch das Betriebssystem erfordern (etwa DMA-Operationen und Unterbrechungen), und die Verwaltung der allokierten SCI-Ressourcen erfolgt im SISCI (*Software Infrastructure for* SCI*)* Kerntreiber. Diese Funktionalität wird über eine Schnittstelle zur Verfügung gestellt, die eng auf die Anforderungen der SISCI Benutzerschnittstelle (siehe nächster Abschnitt) abgestimmt ist.

Die Kerntreiber IRM und SISCI bieten den gesamten Funktionsumfang des PSA an, sind jedoch in der Handhabung umständlich, da sie nur über *iocontrol Aufrufe [161]* genutzt werden können. Dies erfordert zur Abwicklung einfacher Aufgaben (wie etwa das Anlegen und Exportieren eines Speichersegments) in mehreren Schritten die Initialisierung und Übergabe komplexer Datenstrukturen. Um die Einrichtung und Handhabung von SCI-Ressourcen für den Programmierer im Benutzer-Adreßraum zu vereinfachen, wurde die SISCI-Schnittstelle (erste Implementation *[37]*, EU-Projekt *Software Interface for SCI [38]*, aktuelle Definition in *[39]*) definiert und als Kombination von Benutzer-Bibliothek und oben beschriebenem Kern-Treiber entwickelt. Die SISCI Benutzerbibliothek läuft im Benutzeradreßraum ab und setzt die Aufrufe von SISCI Funktionen in *iocontrol* Aufrufe an den SISCI Kerntreiber um, oder führt Datentrans-

fers direkt über den globalen SCI-Adreßraum durch.

Über die SISCI-Schnittstelle enthält der Benutzer Zugriff auf den Großteil der möglichen SCI-Ressourcen[1] (im folgenden SISCI-Objekte genannt), mit denen die Kommunikation in einem SCI-Cluster abgewickelt werden kann.

3.2.2.1 Lokale Segmente

Um lokalen Speicher für entfernte Knoten verfügbar zu machen, muß ein Prozeß ein *lokales Speichersegment (*LS*)* anlegen. Dies bedeutet, daß über den Kerntreiber ein entsprechend großes Stück *physikalisch zusammenhängenden, nicht auslagerbaren Speichers* reserviert wird, dem von einem PSA eine *Segment*-ID zugeordnet ist. Über das Tupel <*Adapter*-ID, *Segment*-ID> kann dieser Speicher im gesamten System eindeutig referenziert werden.

Die beiden Bedingungen, die an den lokalen Speicher gestellt werden, um als LS verwendet werden zu können, machen diesen zu einer stärker begrenzten Ressource als gewöhnlichen Arbeitsspeicher. Sie sind aber aufgrund der Architekturmerkmale der aktuellen PSA notwendig:

1. Der PSA greift direkt über physikalische Adressen auf den Speicher zu. Daher besteht keine Möglichkeit, ggf. vom BS ausgelagerte virtuelle Seiten zuvor wieder einzulagern. Es würden daher andere virtuelle Seiten, die vom Betriebssystem auf diese physikalischen Seiten gelegt wurden, überschrieben bzw. falsche Daten gelesen.
2. Die Abbildung des gesamten lokalen Speichers des Speichersegments in den SCI-Adreßraum erfolgt beim Export direkt: Mit einem Offset, der sich aus der Adapter-ID des exportierenden PSA ergibt, werden eingehende Adressen aus dem SCI-Adreßraum auf die lokalen physikalischen Adressen abgebildet. Bei physikalisch nicht zusammenhängendem Speicher wäre diese einfache Abbildung nicht möglich, sondern müßte für jede Seite getrennt erfolgen.

Anforderung 1 ist bei allen E/A-Geräten mit DMA-Unterstützung zu erfüllen. Anforderung 2 kann durch einen anderen Entwurf des PSA entbehrlich werden *[59]*. Mit den in dieser Arbeit verwendeten PSAs ist zwar auch die Abbildung von nicht-zusammenhängenden physikalischem Speicher in den SCI-Adreßraum möglich, wie es in *[53]* für eine Shared-Memory Infrastruktur realisiert wurde. Dies wurde ähnlich in *[52]* für eine generelle Erweiterung des IRM vorgeschlagen, bislang jedoch noch nicht umgesetzt. Das Verfahren ist mit höherem Softwareaufwand und PSA-Ressourcenverbrauch verbunden.

3.2.2.2 Lokale registrierte Segmente

Die Allokation des Speichers für ein LS erfolgt durch den IRM, wodurch es nicht möglich ist, auf diese Weise bereits allokierten Speicher oder bestimmte, ausgezeichnete Speicherbereiche (etwa den Adreßraum eines Grafikadapters oder anderen E/A-Geräts) in den SCI-Adreßraum für den Zugriff entfernter Prozesse nutzbar zu machen. Dazu muß ein *lokales registriertes Speichersegment (*LRS*)* eingerichtet werden. Bei einem LRS wird der Speicher durch das BS als nicht auslagerbar gekennzeichnet, womit Anforderung 1 aus dem vorigen Unterkapitel erfüllt ist. Die entsprechende Anforderung 2 kann man jedoch nicht nachträglich erfüllen: Speicher, der innerhalb von Benutzerprozessen allokiert wurde, kann nicht als physikalisch zusammenhängend angenommen werden[2]. Dies bedeutet Einschränkungen für die Nutzbarkeit des Spei-

1. Ausgehend von den fundamentalen Ressourcen *lokal eingeblendeter entfernter Speicher* wurden im Zuge der Weiterentwicklung der Implementation der SISCI-Bibliothek immer weitere SCI-Ressourcen verfügbar gemacht, beispielsweise *asynchrone Rückrufe (Callbacks)* oder durch Schreibzugriffe ausgelöste Unterbrechungen (*conditional interrupts)*, die entweder der Leistungsoptimierung oder der Funktionalitätserweiterung dienen, aber das grundsätzliche Betriebsmodell und die SISCI-Schnittstelle nicht berühren.
2. Tatsächlich wurde in *[52]* ermittelt, daß es sogar sehr unwahrscheinlich ist, daß innerhalb eines zusammenhängenden Bereichs virtuellen Speichers überhaupt zwei aufeinanderfolgende virtuelle Seiten auch physikalisch aufeinanderfolgend sind.

chersegments für entfernte Zugriffe; im Rahmen dieser Arbeit wurde jedoch die Nutzbarkeit von LRS als Quelle und Ziel für DMA-Transfers implementiert sowie ein Verfahren für die Nutzbarkeit durch entfernte PIO-Zugriffe vorgeschlagen (siehe auch *[52]*).

3.2.2.3 Entfernte eingeblendete Segmente

Ein LS, das von einem Prozeß auf einem Knoten angelegt wurde, kann durch andere Prozesse auf demselben oder anderen Knoten als *entferntes eingeblendetes Speichersegment (*EES*)* in den eigenen Adreßraum eingeblendet werden. Dazu wird über einen lokalen PSA mittels des Tupels <*Adapter*-ID, *Segment*-ID> eine Verbindung zu dem entfernten Speichersegment aufgebaut. Ein entsprechend angelegter Bereich im lokalen physikalischen PCI-Adreßraum wird über einen ATT-Eintrag in den globalen Adreßraum abgebildet. Lokal wird der physikalische PCI-Adreßraum über die MMU in den virtuellen Adreßraum der Prozesse eingeblendet, so daß Prozesse darauf mit virtuellen Adressen zugreifen können.

3.2.2.4 Entfernte verbundene Segmente

Sobald ein PSA zu einem entfernten LS eine Verbindung aufgebaut hat, gilt dieses als *entferntes verbundenes Speichersegment (*EVS*).* Obwohl der zugehörige Speicher nicht in den lokalen Adreßraum eingeblendet ist, können zwischen diesem PSA und dem EVS Daten ausgetauscht werden, da der PSA alle notwendigen Information dazu hat. Dadurch ist es für reine DMA-Transfers zwischen lokalem und entferntem Speicher ausreichend, den entfernten Speicher als EVS einzurichten. Zwischen diesem und einem LS oder LRS können dann, unter Berücksichtigung der erforderlichen Ausrichtung für die Adressen und die Länge des Transfers, Daten in beide Richtungen transferiert werden.

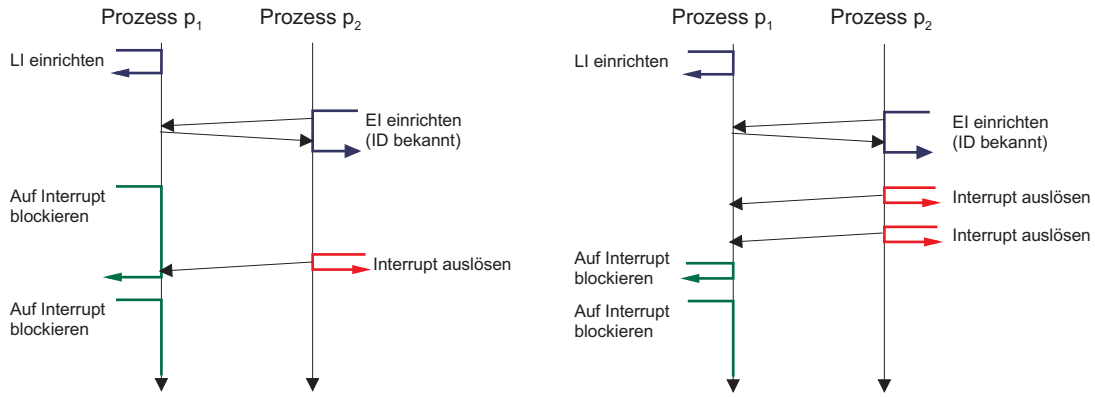
Für die in dieser Arbeit untersuchten Kommunikationsmodelle ist besonders der Fall der Datenübertragung zwischen zwei LRS auf verschiedenen Knoten, von denen eines als EVS genutzt wird, interessant (siehe Kapitel 7.2).

3.2.2.5 Lokale und entfernte Interrupts

Die Kommunikation über gemeinsamen Speicher, wie sie durch die in den vorangegangenen Unterkapiteln beschriebenen SISCI-Objekte möglich ist, erlaubt nur ein *Polling*-basiertes Kommunikationsmodell: der Empfänger einer Nachricht muß durch aktive Abfrage eines Zustandes, in diesem Fall einer Speicherstelle, überprüfen, ob diese oder irgendeine neue Nachricht vorliegt. Im Gegensatz dazu erlauben *Interrupts (Unterbrechungen)* ein *asynchrones* Kommunikationsmodell. Hierbei wird der Empfänger der Nachricht aktiv aus seinem aktuellen Zustand heraus zur Abfrage einer neu eingetroffenen Nachricht veranlaßt. Der aktuelle Zustand eines Prozesses kann die Abarbeitung beliebiger Instruktionen oder das *Blockieren* bis zum eintreffen einer Unterbrechung sein. Im Gegensatz zu einem Polling-Prozeß verbraucht ein blockierter Prozeß bei Warten auf den Eingang der Nachricht keine CPU-Ressourcen.

Die in dieser Arbeit genutzte SCI-Architektur erlaubt das Auslösen von Unterbrechungen auf einem entfernten Knoten (siehe Abb. 3.7). Dazu wird zunächst auf dem Zielknoten für die Unterbrechungen ein *lokaler Interrupt* (LI) eingerichtet werden. Dieser erhält eine Interrupt-ID, die zusammen mit der zugehörigen PSA-ID von einem anderen Prozeß als Tupel <*Interrupt*-ID, PSA-ID> genutzt wird, um einen mit dem lokalen Interrupt verbundenen *entfernten Interrupt* (EI) einzurichten. Über den EI kann die Unterbrechung auf dem entfernten Knoten bei dem Prozeß, der die Unterbrechung eingerichtet hat, ausgelöst werden.

Dabei müssen bei zwei Prozessen p_1 und p_2, die sich über eine von p_1 eingerichtete Unterbrechung synchronisieren, *synchrone* und *asynchrone* Unterbrechungen und deren Behandlung unterschieden werden. Bei einer synchronen Unterbrechung blockiert p_1 bereits auf der lokalen Unterbrechung, wenn p_2 sie auslöst, und deblockiert entsprechend (Abb. 3.7 a). Falls p_1 jedoch erst dann auf der Unterbrechung blockiert, wenn diese von p_2 bereits mindestens einmal ausgelöst wurde (asynchrone Unterbrechung), deblockiert er *einmal* (Abb. 3.7 b), d.h. es wird nur das



a) Synchrone Unterbrechungsauslösung b) Asynchrone Unterbrechungsauslösung

Abbildung 3.7: Interruptmodell für SVS: Einrichtung einer entfernten Unterbrechung und a) Blockierung von p_1 beim Warten auf Unterbrechung durch p_2 (*synchron*) b) Speicherung eingegangener Unterbrechungen von p_2 für p_1 (*asynchron*)

Eintreffen einer entfernten Unterbrechung gespeichert, aber nicht deren Zahl. Dieses Verhalten muß berücksichtigt werden, wenn durch eine Unterbrechung Operationen auf gleichzeitig übertragenen Daten ausgeführt werden sollen (siehe Kapitel 7.1).

3.2.2.6 DMA-Übertragungen

Neben der transparenten Datenübertragung durch Lade- und Speicheroperationen auf eingeblendeten Speichersegmenten kann ein PSA auch pseudo-schlanke Kommunikation[1] mittels DMA-Übertragung durchführen, so daß die CPU keine Daten kopieren muß. Dazu wird ein DMA-Deskriptor mit der Beschreibung der zu übertragenden Bereiche in Quell- und Zielspeichersegment zum PSA übermittelt werden; der DMA-Baustein des PSA wickelt den Transfer anschließend ohne CPU-Mitwirkung ab.

Da der DMA-Baustein mit physikalischen und nicht mit virtuellen Adressen arbeitet, müssen die beteiligten Speichersegmente nur verbunden, nicht jedoch eingeblendet sein. Dieser Zusammenhang wird in Abb. 3.8 verdeutlicht: Wenn etwa Prozeß p_0 Daten von seinem lokalen Speichersegment *B* in das entfernte Speichersegment *C* von Prozeß p_3 übertragen will, so findet diese Übertragung bei Nutzung von DMA ausschließlich in den physikalischen Adreßräumen

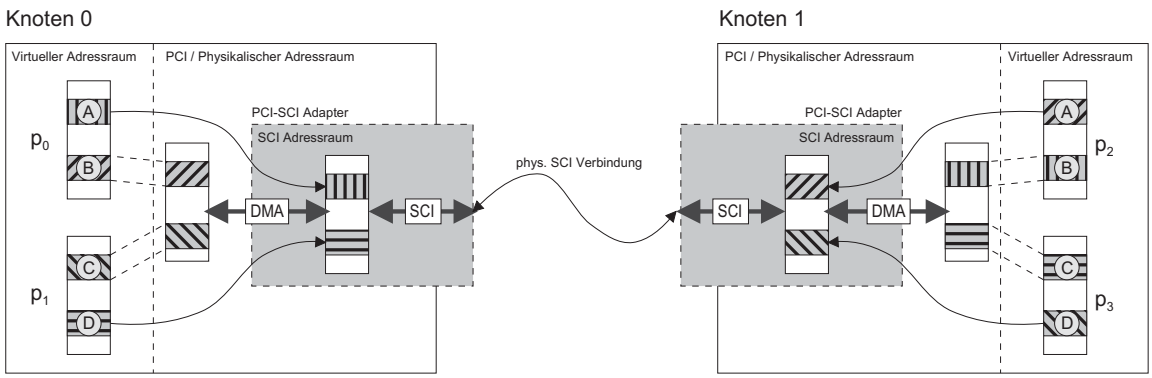


Abbildung 3.8: DMA-Modell für SVS mit Dolphin PCI-SCI-Adaptern

1. Wegen fehlender Möglichkeiten zur Überprüfung der Gültigkeit der Quell- und Zieladressen muß der Deskriptor durch einen Treiberaufruf zum PSA übertragen werden, so daß keine schlanke Kommunikation im Sinne der Definition in Kapitel 2.3.2 vorliegt. Da jedoch ohne Zwischenkopien direkt zwischen Speicherbereichen im Benutzeradreßraum übertragen wird, kann von pseudo-schlanker Kommunikation gesprochen werden.

der beteiligten Knoten 0 und 1 statt.

3.2.3 SMI Bibliothek

Obgleich die Handhabung von SCI-Ressourcen mit der SISCI-Schnittstelle gegenüber der direkten Ansprache des IRM deutlich vereinfacht wurde, fehlen dieser einige grundlegende Funktionen, die zur Erstellung einer parallelen Applikation erforderlich sind:

- *Applikationsstart*: Es ist nicht festgelegt, wie die auf verschiedenen Knoten gestarteten Prozesse, die zusammen eine Applikation bilden sollen, sich miteinander synchronisieren können. Die Nutzung von SCI-Speichersegmenten mit statisch gewählter Segment-ID ist nicht sinnvoll, da die Verfügbarkeit einer bestimmten Segment-ID für ein einzurichtendes Speichersegment nicht allgemein gewährleistet ist. Für einen allgemeingültigen Ansatz, der es auch erlaubt, mehrere unabhängige Instanzen einer Applikation auf einem Knoten laufen zu lassen, müssen die Segment-IDs dynamisch gewählt werden.
- *Synchronisation*: Basisdienste zur Synchronisation wie gegenseitiger Ausschluß und Barrieren oder auch Signalisierung sind in SISCI nicht spezifiziert. Diese müssen basierend auf gemeinsamem Speicher bzw. SCI-Interrupts implementiert werden.
- *Speicherverwaltung*: SISCI bietet alle Funktionen, um SCI-Speichersegmente zu allokieren, zu exportieren und zu importieren. Jedoch ist dazu ein Informationsaustausch zwischen den Prozessen erforderlich, den SISCI wiederum nicht spezifiziert. Auch die dynamische Speicherverwaltung von Speicherblöcken aus einem angelegten (importierten oder exportierten) SCI-Speichersegment wird für viele Applikationen benötigt.

Diese Funktionalität erbringt, neben weiteren Bereichen, die SMI Bibliothek, die zunächst als Programmiermodell zur Parallelisierung von Anwendungen nach dem Modell des gemeinsamen Speichers unter besonderer Berücksichtigung von NUMA Speicherarchitekturen entwickelt wurde *[48,49]*. Das Prozeßmodell hat die gleiche Semantik wie MPI; die angebotenen Funktionen dienen jedoch nicht dem Nachrichtenaustausch, sondern der Einrichtung von gemeinsamen Speicherregionen der Prozesse. Dazu kommen Schleifenparallelisierungs-Konstrukte, die eine ähnliche Semantik wie OPENMP haben, jedoch durch explizite Funktionsaufrufe gesteuert werden. Auf der Grundlage dieses Programmiermodells wurden verschiedene Anwendungen auf der in dieser Arbeit behandelten Plattform parallelisiert. Besonders umfangreiche Anwendungen waren ein System zur Ermittlung von Flugplänen *[50]* und die Simulation von Molekulardynamik GROMOS96 *[147,51]*. Im Rahmen der in diesen Untersuchungen betrachteten Systeme, die aus maximal 6 Prozessen auf 6 Knoten bestanden, konnten diese Anwendungen mit guter Effizienz parallel ausgeführt werden.

Im Rahmen dieser Arbeit wurde die SMI-Bibliothek um Funktionalität erweitert, die sie als generelle, komfortable Schnittstelle zu SCI qualifiziert *[58]*. Neben den abstrakten, aber statischen und rein kollektiven[1] Operationen zur Ressourcenverwaltung, die für das vorgestellte Programmiermodell bei den untersuchten Parallelitätsgraden genügten, wurden neue Typen von gemeinsamen Speicherregionen eingeführt, die dynamisch und asynchron eingerichtet und auch wieder abgebaut werden können. Dazu kommen weitere Funktionen, um neben dem transparenten Zugriff auf entfernten Speicher Ressourcen wie DMA-Übertragungen und Signale an entfernte Prozesse nutzen zu können. Somit kann die komplette Funktionalität von SCI als Netz in einem Rechnerverbundsystem genutzt werden, ohne auf die komfortable Abstraktion zu verzichten. Insbesondere kann die SMI-Bibliothek nun auch zur Unterstützung anderer Bibliotheken dienen, die die SCI-Funktionalität nutzen wollen, wie es in dieser Arbeit geschieht.

1. *Kollektiv* bedeutet, daß alle Prozesse den jeweiligen Funktionsaufruf tätigen müssen, bevor er abgeschlossen werden kann

3.3 Nutzungsmodell des gemeinsamen Speichers

Die vorgestellte Hardware/Software-Architektur ermöglicht es, entfernte Speichersegmente in den Adreßraum eines Prozesses einzublenden. Die CPU kann auf die Adressen dieser Speichersegmente wie auf lokalen Speicher zugreifen. Dennoch gibt es eine Reihe von Unterschieden zwischen lokalem und entferntem Speicher, die bei der Nutzung des entfernten Speichers berücksichtigt werden müssen, um die erwarteten Ergebnisse bei hoher Leistung zu erhalten.

3.3.1 Leistungscharakteristik von Punkt-zu-Punkt-Kommunikation

Zunächst wird bei der Untersuchung der Leistungseigenschaften von Netzen die Punkt-zu-Punkt-Leistung betrachtet, also die Kommunikation zwischen zwei Endpunkten unabhängig von möglicher anderer Kommunikation im Gesamtnetz. Die Werte zu Bandbreite und Latenz der verwendeten PSA und PCI-Implementation (siehe Kapitel 3.5) sind in Abbildung 3.9 dargestellt.

Bei der Kommunikation zwischen durch PSA gekoppelten Knoten entsteht der maßgebliche Anteil der Latenz von minimal 1,5µs bei der Datenübertragung über den PCI-Bus. Die Latenz, die durch das Weiterleiten der Pakete über mehrere Knoten hinweg entsteht, liegt auch bei großen SCI-Netzen bei Weiterleitung über 16 Knoten unterhalb 1µs und somit unterhalb der Zeit zur Übertragung der Daten innerhalb des Quell- und Zielknotens.

Zur Datenübertragung definiert SCI verschiedene Transaktionen, die unterschiedliche Datenmengen transportieren können. Die Betrachtungen beschränken sich auf Schreibvorgänge zwischen PSA-gekoppelten Knoten; die dazu relevanten Transaktionstypen sind in Tabelle 3.1 aufgeführt. Für jeden Transaktionstyp ist die Nutzdatenmenge, die Größe des Feldes im Paket für diese Datenmenge sowie die Effizienz als Verhältnis der Nutzdatenmenge zur Gesamtgröße des Pakets aufgeführt. Diese ergibt sich als Summe der Größe des Datenfelds und der Größe der Kontrolldaten, die für alle Pakete konstant 16 Byte beträgt.

Die maximale Bandbreite kann nur erreicht werden, wenn die SCI-Transaktionen mit dem maximalen Verhältnis von Nutzlast zu Kontrolldaten verwendet werden. Dies sind je nach Version des LC *nwrite64* oder *nwrite128* Pakete mit 64 bzw. 128 Byte Nutzlast gegenüber den 16 Byte Kontrolldaten. Der PCI-Bus kann jedoch in einer Transaktion nur maximal 8 Byte Daten übertragen. Daher müssen die ausgehenden Daten auf dem PSA gesammelt werden, um optimale SCI-Transaktionen generieren zu können. Dieses *write gathering* (Sammeln von Schreibdaten) wird in sogenannten *stream buffers (SB)* durchgeführt, die entsprechend der maximalen Größe

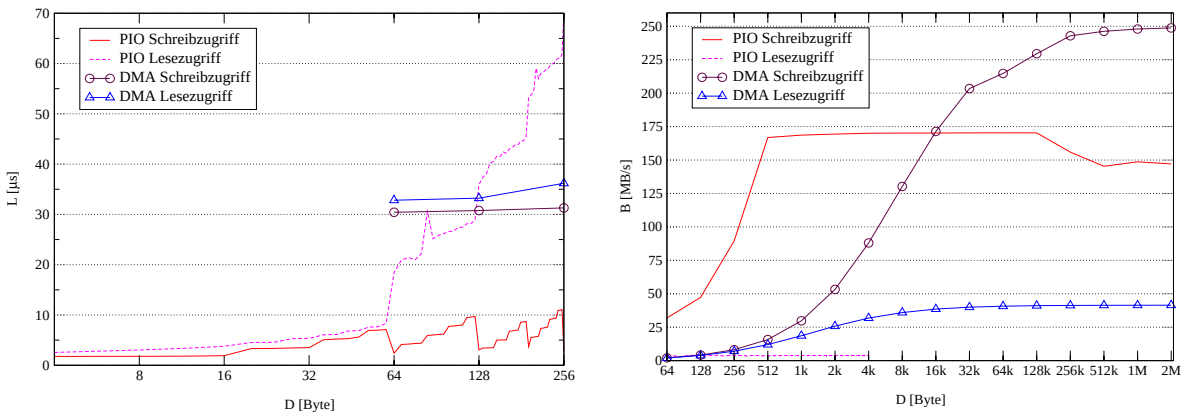


Abbildung 3.9: Latenz (*links*) und Bandbreite (*rechts)* für PIO- und DMA-basierte Lese- und Schreibzugriffe auf entfernten Speicher über PCI-SCI-Adapter.
P3 Plattform: PSA D330, 64-bit/66 MHz PCI-Bus, Chipsatz Serverworks Serverset III-LE

Typ	Nutzdaten	Datenfeld	Effizienz	Verwendung
movesb	1 - 16	16	< 50%	Schreiben von Daten an eine spezifizierte Adresse (Byte-Granularität)
nwrite16	16	16	50%	Schreiben von 16 Byte an 16-Byte ausgerichtete Adresse
nwrite64	64	64	80%	Schreiben von 64 Byte an 64-Byte ausgerichtete Adresse
nwrite128[a]	128	128	88,8%	Schreiben von 128 Byte an 128-Byte ausgerichtete Adresse
nwrite256[b]	256	256	94,1%	Schreiben von 256 Byte an 256-Byte ausgerichtete Adresse

Tabelle 3.1: Verfügbare Transaktionstypen zum Schreiben in entfernten Speicher. Für eine auf *D* Byte ausgerichtete Adresse *A* gilt $A \lozenge D = 0$.

a. Dieser Transaktionstyp ist im ursrprünglichen SCI-Standard nicht enthalten und wurde von Dolphin mit dem Linkcontroller LC-3 eingeführt.
b. Dieser Tranaktionstyp ist in den gegenwärtigen PSA-Implementierungen nicht verfügbar.

einer *nwrite*-Transaktion dimensioniert sind. Um diese optimal nutzen zu können, müssen die folgenden Bedingungen bei der Generierung von SCI-Transaktionen berücksichtigt werden:

- Eine SCI-Transaktion kann nur zusammenhängende Datenblöcke transportieren, die sich beim Empfänger durch Angabe einer einzigen Adresse speichern lassen. Daher müssen die Transaktionen, die in einem SB gesammelt werden sollen, auf *kontinuierlich aufsteigende* Adressen durchgeführt werden.
- Die Adresse des Datenblocks, der in einer SCI-Transaktion übertragen werden soll, muß entsprechend der Größe des Datenblocks ausgerichtet sein (siehe Tabelle 3.1). Gemäß der Adresse eines geschriebenen Datums wird ein SB allokiert, in den die nachfolgenden Daten, sofern sie kontinuierlich aufsteigend geschrieben werden, gesammelt werden.
  Daten, die nicht an einer der durch die *nwrite*-Größen vorgegebenen Grenze ausgerichtet sind, müssen mit vergleichsweise ineffizienten *movesb*-Transaktionen übertragen werden.
- Aus den Daten in einem SB wird die für die Übertragung der enthaltenen Daten notwendige Zahl von SCI-Transaktionen (im Optimalfall eine einzige) generiert, wenn eine der folgenden Bedingungen eintritt:
  - Ein Schreibvorgang wird einem SB zugewiesen, der bereits Daten enthält, jedoch schließt die Adresse des neuen Datums nicht an die vorhandenen Daten an.
  - Ein Schreibvorgang erreicht die obere Grenze eines SB.
  - Der Knoten erzwingt zur Herstellung eines konsistenten Speicherzustands die Ausführung aller ausstehenden (gesammelten) Schreibvorgänge (*flush)* oder wartet zusätzlich auf deren Abschluß durch den Empfänger (*store barrier)*.
  - Auf einen SB wurde über einen vorgegebenen Zeitraum kein Schreibvorgang ausgeführt (*timeout)*.

Aus diesen Randbedingungen ergibt sich die Notwendigkeit einer angepaßten Auslegung von Größe und Lage der Zugriffe auf entfernten Speicher, um eine optimale Bandbreite zu erhalten. Wenn etwa zwischen mit LC-3 basierten PSA verbundenen Knoten die Grenzen von Kommunikationspuffern auf 128 Byte ausgerichtet sind sowie die Größen der übertragenen Daten ein Vielfaches von 128 betragen, wird die optimale Bandbreite erreicht. Falls hingegen für kleinere

Datenübertragungen eine optimale Latenz erreicht werden soll, sollte ein solcher Schreibvorgang an einer 128-Byte-Grenze enden, so daß die Daten aus dem zugewiesenen SB unmittelbar zum Zielknoten übertragen werden, ohne daß ein *flush* oder eine *store barrier* durchgeführt werden muß oder etwa auf den *timeout* gewartet wird.

3.3.2 Skalierungseigenschaften der Bandbreite

Neben der Betrachtung der Punkt-zu-Punkt-Leistung zweier speichergekoppelter Knoten muß auch die Leistung des SCI-Netzes eines Rechnerverbundsystems als Ganzes betrachtet werden. Damit ein System als Ganzes leistungsfähig ist, reicht es nicht aus, daß zwei Prozesse mit hoher Leistung kommunizieren können, sondern gegebenenfalls müssen alle Prozesse gleichzeitig kommunizieren. Um auch in diesem Fall eine hohe Kommunikationsbandbreite zu gewährleisten, muß die Leistung des Netzes skalieren, also mit der Zahl der Knoten zunehmen. Diese Eigenschaft wird durch die *Bisektionsbandbreite* quantifiziert. Dazu muß die Topologie des Gesamtsystems betrachtet werden - so hat ein Ring von gerichteten Punkt-zu-Punkt-Verbindungen nie eine höhere Bisektionsbandbreite als die doppelte Punkt-zu-Punkt-Bandbreite eines SCI-Linksegments $2 \cdot B_{slink}$, das zwei Knoten miteinander verbindet. Ab einer bestimmten Zahl von Knoten K mit jeweils maximaler Ein-/Ausgabebandbreite B_{int} wird diese Bisektionsbandbreite zu einem Engpaß (nämlich dann, wenn $K \cdot B_{int} > 2 \cdot B_{slink}$). In einem solchen Fall müssen andere Topologien eingesetzt werden. SCI bietet hier vor allem die Möglichkeit, mehrdimensionale Torustopologien zu realisieren.

Neben der unterschiedlichen Ausfallsicherheit ist die Skalierbarkeit das wesentliche Unterscheidungskriterium der Topologien. Die Skalierbarkeit soll hier ausgedrückt werden als verfügbare Bandbreite pro Knoten für den Fall, daß alle Knoten die maximale Bandbreite nutzen wollen, um Daten zu einem anderen Knoten zu übertragen. Maximal kann ein Knoten Daten mit der Bandbreite B_{int} versenden, die knotenintern zwischen Hauptspeicher und PSA verfügbar ist. B_{int} wird zumeist durch die Bandbreite des E/A-Busses begrenzt, aber auch der Hauptspeicher kann für die Begrenzung verantwortlich sein. Weiterhin verfügt jeder Knoten über einen Anteil an der festen Bandbreite B_{slink} der SCI-Linksegmente, an die er mit seinen Linkcontrollern angeschlossen ist. Bezüglich dieser Bandbreite muß berücksichtigt werden, daß ein ringförmiger SCI-Link kein Bus ist, sondern aus unabhängigen Linksegmenten besteht. Eine weitere Bandbreite, die limitierend wirken kann, ist die Bandbreite B_{blink} des B-LINKs. Über ihn müssen, wie über den E/A-Bus, alle ein- und ausgehenden Pakete dieses Knotens fließen, zusätzlich aber noch alle Pakete, die in diesem Knoten die Dimension wechseln. Die Zahl dieser Pakete nimmt, ausgehend von 0 Paketen bei einem eindimensionalen Torus, mit der Zahl der Dimensionen zu. Die Zahlenwerte für diese Bandbreiten in aktuellen Systemen sind in Tabelle 3.2 aufgeführt und sind die Basis für die in den folgenden Abschnitten durchgeführten Skalierungsberechnungen.

Bandbreite	Typ	Bitbreite	Frequenz (MHz)	Bruttowert (MB/s)
B_{int}	PCI	64	66	532
B_{blink}	B-LINK	64	80	640
B_{slink}	SCI	16	166[a]	684

a. Das SCI-Signal überträgt Daten auf beiden Taktflanken

Tabelle 3.2: Bandbreitenkennwerte der aktuellen PCI-SCI-Systeme für die Elemente des Datenpfads bei der Kommunkation zwischen zwei entfernten Knoten.

Die Effizienz der Datenübertragung auf B-LINK und SCI-Link hat sich durch die Einführung von 128-Byte Datenpaketen (Typ *nwrite128*) gegenüber *[40]*, wo die Pakete vom Typ *nwrite64* diejenigen mit der größten Nutzlast waren, ebenfalls geändert: In Tabelle 3.3 ist die Nutzung der Taktzyklen bzw. der damit verbundenen Übertragungskapazität auf dem B-LINK und dem SCI-Link für die Übertragung eines solchen *nwrite128*-Pakets dargestellt. Wie daraus ersichtlich ist, hat sich durch das bessere Verhältnis von Nutzlast- zu Steuerzyklen die Effizienz von 50% auf 66,6% erhöht.

Mit diesen Informationen lassen sich Aussagen zu der Skalierbarkeit von SCI-Torus-Topologien unterschiedlicher Dimension machen. Es sollen drei Kommunikationsfälle betrachtet werden:

- Jeweils zwei Knoten tauschen miteinander Daten aus, d.h. ein Knoten schreibt Daten auf den jeweils anderen (*pair*-Kommunikation). Da hier eine Paarbildung erfolgt, hat die Anordnung der Knoten innerhalb der Topologie des SCI-Netzes zueinander Einfluß auf die effektive Bandbreite. Dabei ist die Kommunikation symmetrisch, so daß eine asymmetrische Positionierung der Knoten zueinander (gemessen an der Zahl der SCI-Linksegmente, die ein Paket von einem Knoten zum anderen passieren muß) keine positiven Effekte erwarten läßt.
- Mehrere Paare tauschen miteinander Daten aus (*exchange*-Kommunikation), dies entspricht mehreren unabhängigen Fällen von *pair*-Kommunikation. Durch die Unabhängigkeit der einzelnen Operationen ist es möglich, diese zu parallelisieren und dadurch die Nutzung der einzelnen Segmente des SCI-Links zu optimieren.
- Jeder Knoten schreibt Daten auf jeden anderen Knoten (*alltoall*-Kommunikation). Diese Kommunikation stellt die höchsten Anforderungen an das Netz. Die Anordnung der Knoten zueinander ist hierbei aufgrund der *alltoall*-Charakteristik unerheblich.

Entscheidend ist dabei jeweils, wieviel Bandbreite jedem Knoten bleibt, um seine Daten in das

B-Link (Zyklen)		**SCI Link (Bytes)**	
Request (Daten)	19	Request (Daten)	144
Master	1	Daten Leerlauf	4
Response	3	Echo	8
Master	1	Echo Leerlauf	4
Summe (Zyklen)	24	Response	16
Summe (Bytes)	192	Response Leerlauf	4
Nutzlast (Bytes)	128	Echo	8
Effizienz	**66,6%**	Echo Leerlauf	4
		Summe	192
		Nutzlast	128
		Effizienz	**66,6%**

Tabelle 3.3: Effizienz der Datenübertragung auf B-LINK und SCI-Link

Netz zu injizieren (*Injektionsbandbreite*). Es sollen Torustopologien der Dimension d und vom Grad[1] g betrachtet werden. Ein System solcher Topologie hat stets

$$(3.1)\quad K = g^d$$

Knoten, die mit

$$(3.2)\quad S = d \cdot K$$

SCI-Linksegmenten zu einem SCI-Netz verbunden sind.

3.3.2.1 *Kommunikation innerhalb eines SCI-Rings*

Die *effektive Linkbandbreite* B_{eff} in einem Ring ist die *akkumulierte Segmentbandbreite* B_{sgmt} aller SCI-Linksegmente in einem Ring. Sie kann sich für die Kommunikation von einem Knoten zu jeweils einem anderen Knoten zwischen zwei Extrema bewegen:

- der *maximale* Wert wird erreicht, wenn jeder Knoten mit seinem direkten Nachbarn in Linkrichtung kommuniziert (*maximale Ringkommunikation*);
- der *minimale* Wert wird erreicht, wenn jeder Knoten mit seinem direkten Nachbarn in gegensinniger Linkrichtung kommuniziert (*minimale Ringkommunikation*).

Abbildung 3.10 stellt diese beiden Fälle für einen im Uhrzeigersinn verbundenen SCI-Ring mit $K = 4$ Knoten und $S = 4$ Linksegmenten dar: Links ist der maximale Fall dargestellt, in dem die Prozesse Daten für ihren logischen Nachbarn direkt zu ihrem physikalischen Nachbarn in Richtung des SCI-Rings schicken können, so daß jedes Linksegment die Daten von nur einem Knoten übertragen muß. Rechts hingegen (minimaler Fall) sind die logischen Nachbarn physikalisch im SCI-Ring am weitesten entfernt, und über jedes Linksegment verläuft der Datenverkehr von $K - 1$ Knoten.

Im maximalen Fall traversieren die Datenpakete der Länge D_D Byte[2] nur das eine verbindende Linksegment zwischen den beiden Knoten. Nur die kleineren Response-Pakete der Länge D_R Byte gelangen über die restlichen $S - 1$ Linksegmente zurück zum Sender. Dazu kommen noch die Echopakete (Länge D_E Byte) und die zwei Leerlaufphasen (äquivalente Länge D_L Byte) für jede der beiden Subaktionen, jeweils vom Empfänger zum Sender und umgekehrt. Damit überträgt jedes SCI-Linksegment

$$(3.3)\quad D_{max} = [D_D + D_E + 2D_L] + [(S-1)(D_R + D_E + 2D_L)]$$
$$= D_D + S \cdot (D_E + 2D_L) + (S-1) \cdot D_R$$

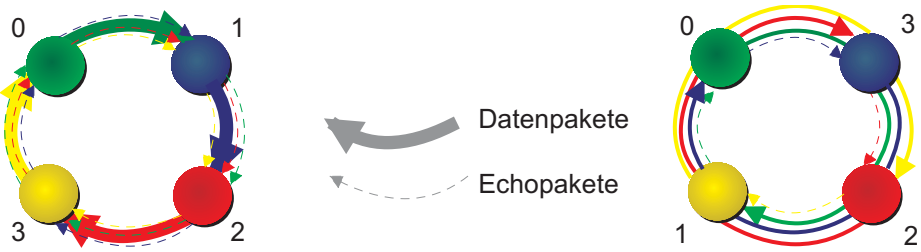


Abbildung 3.10: Extremfälle der effektiven Linkbandbreite auf *S*-teiligem SCI-Ring (im Uhrzeigersinn gerichtet):
*Links: maximale Ringkommunikation (*jedes Datenpaket traversiert nur 1 Linksegment)
*Rechts: minimale Ringkommunikation (*jedes Datenpaket traversiert S-1 Linksegmente)

1. Der *Grad* ist die in jede Dimensionsrichtung identische Ausdehnung des Torus.
2. *nwrite128* Request-Pakete (Schreiben in entfernten Speicher)

Byte pro Transaktion, mit der D Byte Nutzlast übertragen werden.

Im minimalen Fall verhält es sich genau umgekehrt, und die Datenpakete müssen $S-1$ Linksegmente traversieren, um vom Sender zum Empfänger zu kommen. Dies erfordert die Übermittlung von

$$(3.4)\quad D_{min} = [D_R + D_E + 2D_L] + [(S-1)(D_D + D_E + 2D_L)]$$
$$= D_R + S \cdot (D_E + 2D_L) + (S-1) \cdot D_D$$

Byte pro transportierten D Byte.

Mit den Werten für die Paketlängen, wie sie sich aus dem SCI-Standard und der aktuellen Implementation bestimmen ($D_D = D + 16 = 128 + 16 = 144$, $D_R = 16$, $D_E = 8$, $D_L = 4$, siehe auch Tabelle 3.4) ergibt sich der minimale und maximale Wert von B_{sgmt} für die Ringkommunikation (in Abhängigkeit von S bzw. K) wie in Abb. 3.11 *(links)* dargestellt. Die Multiplikation mit der Zahl der Knoten im Ring ergibt daraus B_{eff} (Abb. 3.11 *rechts*). Die Kapazität des SCI-Rings bei maximaler Ringkommunikation um ein Vielfaches höher als bei minimaler Ringkommunikation. Für die derzeitige SCI-Technologie gilt $B_{eff} = 2,2 GB/s$ im maximalen Fall gegenüber $B_{eff} = 0,51 GB/s$ im minimalen Fall (jeweils für Knotenzahl $K \to \infty$).

Die *Injektionsbandbreite* B_i eines Knotens ist das Minimum von B_{int}, B_{blink} und B_{sgmt} des genutzten SCI-Linksegments. Die verfügbaren Bandbreiten des PCI-Busses (B_{int}) und des B-LINKS (B_{blink}) sind in einem Ring konstant und liegen für kleine Knotenzahlen unterhalb von B_{sgmt}. Ab einer bestimmten Zahl von Knoten kommt es in beiden Fällen jedoch zu Sättigungseffekten auf den SCI-Linksegmenten, welche die Injektionsbandbreiten der Knoten im Ring limitieren. Auf der verwendeten Plattform kommt es bei maximaler Ringkommunikation für DMA-Transfers bei $K \geq 6$, für PIO-Transfer bei $K \geq 12$ zu Sättigungseffekten. Bei minimaler Ringkommunikation liegen diese Grenzen bei $K \geq 3$ respektive $K \geq 5$.

3.3.2.2 Pair-Kommunikation

In einem System mit $K = 2 \cdot K'$ Knoten sind $(K-1)!$ verschiedene Paarbildungen möglich. Bei einem eindimensionalen Torus spielt es jedoch keine Rolle, welche Paare miteinander Daten austauschen, da in jedem Fall alle Pakete für die Kommunikation zwischen den beiden Knoten eines Paares einmal den gesamten Ring umrunden müssen. Bei K Knoten und K' Paaren sind somit auf jedem Linksegment K' Pakete zu übertragen, womit jedem Knoten eine Bandbreite von

$$(3.5)\quad B_{s\,pair\,1} = 2\frac{B_{slink}}{K}$$

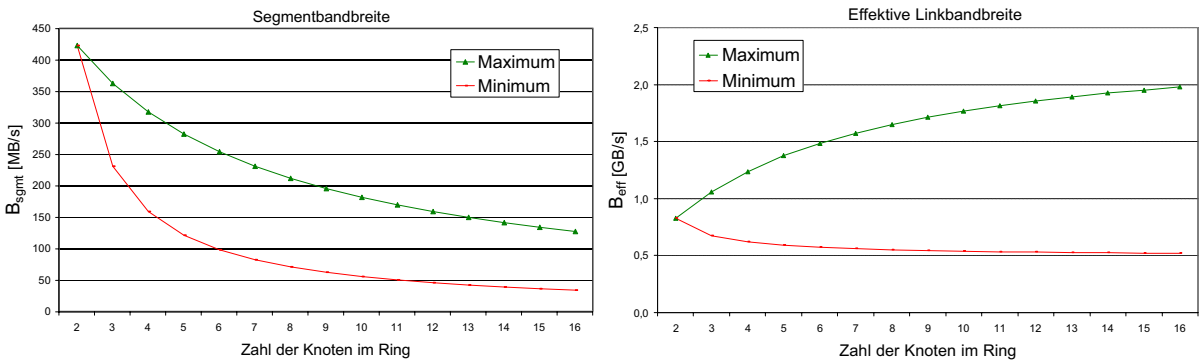


Abbildung 3.11: Minimale und maximale Werte der Segmentbandbreite *(links)* und der effektiven Linkbandbreite *(rechts)* für die Kommunikation in einem SCI-Ring.

zur Verfügung steht. Über jeden B-LINK müssen zwei Pakete (ein- und ausgehende Daten) übertragen werden, so daß in diesem Fall

$$(3.6)\quad B_{b\,pair\,1} = \frac{B_{blink}}{2}$$

der B-LINK-Bandbreite verfügbar sind.

Bei einer mehrdimensionalen Topologie müßte jedoch für jedes Paket, das von einem Knoten eines Paars zu dem jeweils anderen versandt wird, die Zahl der notwendigen Dimensionswechsel berücksichtigt werden, um die Belastung der B-LINKs zu ermitteln. Dies ist einerseits sehr aufwendig und würde andererseits für eine allgemeine Betrachtung kaum weiterhelfen, da sich die unterschiedlichen Ergebnisse keinen allgemeinen Kommunikationsfällen zuordnen ließen. Jedoch ist die Betrachtung des besten und schlechtesten Falles von Interesse, um daraus Methoden zur Optimierung und Bewertung der Positionierung der Paare und der Abwicklung der Kommunikation abzuleiten. Der schlechteste Fall ist dann gegeben, wenn jedes Paket $d-1$ Mal die Dimension wechseln muß[1]. Dieser Fall ist für zwei- und dreidimensionale Topologien in Abb. 3.12 dargestellt: ein System der Dimension d wird in 2^d Quadranten unterteilt; anschließend wird jedes Paar (k, k') zweier Knoten aus zwei Quadranten (Q, Q') gebildet, die jeweils in jeder Dimension gegeneinander versetzt sind. Jedes der K Pakete, die von den K Knoten zu ihrem Partner geschickt werden, traversiert im Mittel

$$(3.7)\quad s_{pair\,d} = d \cdot \frac{g}{2}$$

Linksegmente, so daß auf jedem Linksegment

$$(3.8)\quad n_{sp} = \frac{dg \cdot K}{2} \cdot \frac{1}{d \cdot K} = \frac{g}{2}$$

Pakete zu übertragen sind. Jedem Knoten steht also die Linkbandbreite

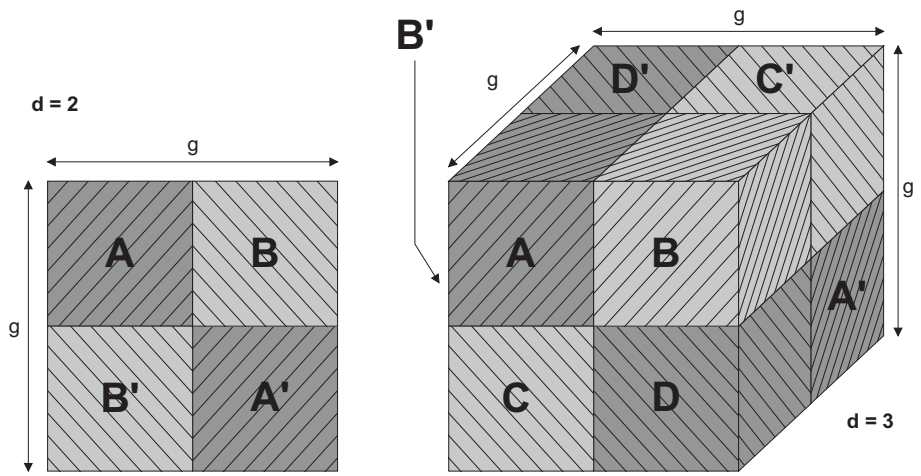


Abbildung 3.12: Schlechtester Fall der *Pair*-Kommunikation: alle Paare von Knoten (k, k') werden jeweils aus den Quadranten (Q, Q') gebildet.
Links: 2-dimensionaler Fall *Rechts:* 3-dimensionaler Fall

1. Tatsächlich hat der schlechteste Fall zusätzlich die Bedingung, daß alle g Knoten eines Ringes mit g Knoten eines anderen Ringes, welcher in eine andere Dimensionsrichtung verläuft, kommunizieren, und das Routingverfahren so gewählt ist, daß alle Pakete in dem bzw. den selben Knoten die Dimension wechseln. Dieser eher pathologische Fall wird hier nicht betrachtet, da von keiner besonderen Anordnung der Knotenpaare ausgegangen wird.

$$(3.9)\quad B_{sp} = 2\frac{B_{slink}}{g}$$

zur Verfügung. Dies ist die allgemeine Form von (3.5) und bedeutet, daß für die *pair*-Kommunikation die Belastung aller SCI-Ringe nur von dem Grad, nicht von der Dimension eines Torus abhängt. Bei Dimensionen d > 1 kommt aber eine zusätzliche Belastung der B-LINKS durch die Dimensionswechsel der Pakete hinzu. Jedes der K Pakete wechselt $d-1$ Mal die Dimension. Da diese Dimensionswechsel entsprechend der angenommenen Paarbildung gleichmäßig über die Knoten des Netzes verteilt erfolgen, überträgt jeder der K B-LINKS zusammen mit dem aus- und eingehendem Paket

$$(3.10)\quad n_{bp} = 2 + d - 1$$

Pakete. Die pro Knoten verfügbare B-LINK-Bandbreite beträgt demnach

$$(3.11)\quad B_{bp} = B_{blink} \cdot \frac{1}{n_{bp}} = \frac{B_{blink}}{2 + d - 1}$$

was wiederum die Verallgemeinerung von (3.6) darstellt. Die Bandbreitenwerte, die sich für diesen Kommunikationsfall und die gegebenen Randbedingungen bezüglich der limitierenden Bandbreiten ergeben, sind in Abb. 3.13 für die B-LINK- und SCI-Link-Bandbreiten dargestellt. Diese Werte beschreiben den *schlechtesten* Fall der *pair*-Kommunikation. Der *beste* Fall kann direkt aus diesen Werten abgeleitet werden: Da die Kommunikation auf den verschiedenen Ringen eines Torus unabhängig ist, ist dieser Fall identisch zu dem Fall eines eindimensionalen Torus des gleichen Grades.

Daraus abgeleitet sind die Darstellungen in Abb. 3.14, die die effektive Bandbreite pro Knoten als Minimum der B-LINK- und SCI-Link-Bandbreite sowie die akkumulierte Bandbreite des Gesamtsystems zeigen. Diese Gesamtbandbreite wird zusätzlich mit der verfügbaren PCI-Bandbreite der Knoten verglichen.

Die ermittelten Werte zeigen, daß für kleine Grade (ungefähr $g < 4 + 2d$) die B-LINK-Bandbreite die effektive Injektionsbandbreite limitiert. Erst für höhere Grade ist die SCI-Link-Bandbreite der limitierende Faktor.

3.3.2.3 Exchange-Kommunikation

Wenn Vorgänge in realen (zweidimensionalen) Gebieten oder (dreidimensionalen) Räumen simuliert werden, spiegelt sich die Aufteilung der Gebiete bzw. Räume in vielen Verfahren in

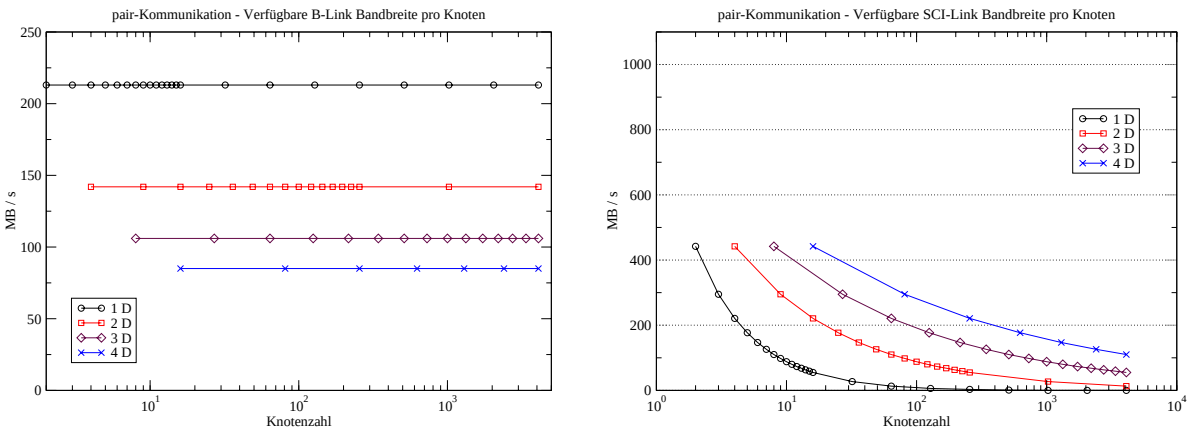


Abbildung 3.13: Limitierung der Injektionsbandbreite durch B-LINK-Bandbreite (*links*) und SCI-Link-Bandbreite (*rechts*) in Abhängigkeit von Knotenzahl und Torus-Dimension für *pair*-Kommunikation.

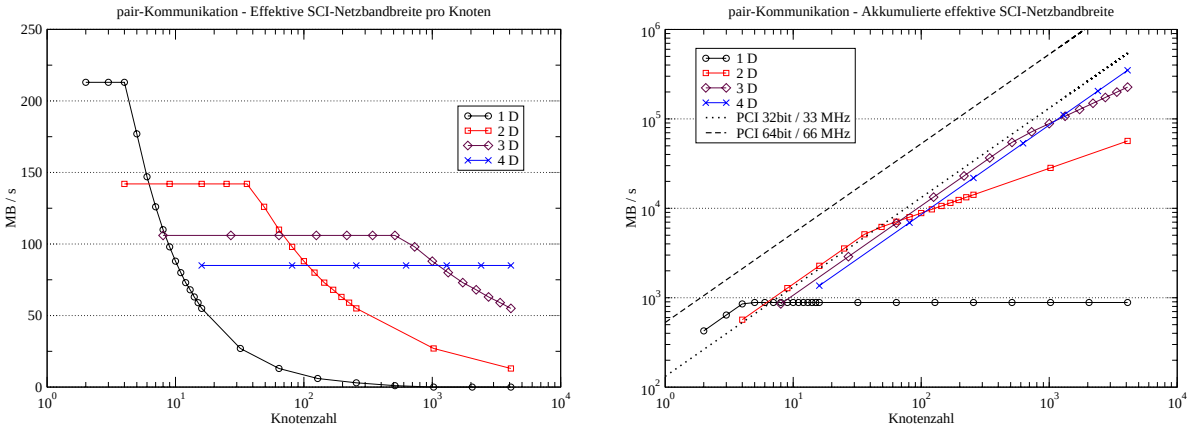


Abbildung 3.14: Effektive Injektionsbandbreite als Minimum der B-LINK- und SCI-Bandbreite pro Knoten (links) und für das Gesamtnetz akkumuliert (rechts) in Abhängigkeit von Knotenzahl und Torus-Dimension für *pair*-Kommunikation. Akkumulierte Bandbreite zusätzlich verglichen mit linear skalierender PCI-Bandbreite der Knoten.

einer entsprechenden Verteilung der Daten wider. Nach den einzelnen Iterationsschritten der Lösungsberechnung müssen zwischen diesen Teilgebieten Randwerte ausgetauscht werden, was einer Punkt-zu-Punkt-Kommunikation zwischen den beiden Besitzern solcher überlappender Teilgebiete entspricht. Diesen Fall kann man auf verschiedene Arten modellieren:

- Aufbauend auf Überlegungen zur *pair*-Kommunikation kann dieser Fall als *n*-fache *pair*-Kommunikation betrachtet werden. Dabei ist jedoch offen, in wievielen Dimensionen sich die Positionen der *n* Knoten, mit denen jeder Knoten kommuniziert, unterscheiden. Hier ist eine schlechtestmögliche bzw. bestmögliche Abschätzung oder eine statistische Mittelung denkbar. Dieser Fall soll *generelle exchange-Kommunikation* genannt werden.
- Entsprechend der Aufteilung der Daten entlang von ein, zwei oder drei Dimensionen kommuniziert ein Knoten (sofern er nicht Daten am Rand des Gebiets bearbeitet) mit 2, 4 oder 6 Nachbarn. Diese Fälle können bezogen auf eine bestimmte physikalische Topologie untersucht werden. Als Bedingung wird dabei angenommen, daß jeweils beide Partner für einen Austausch von Daten in einer Dimensionsrichtung zusammen mit dem betrachteten Knoten in einem gemeinsamen SCI-Ring liegen, wie es in Abb. 3.15 dargestellt ist. Dieser Fall soll als *topologische exchange-Kommunikation* bezeichnet werden.

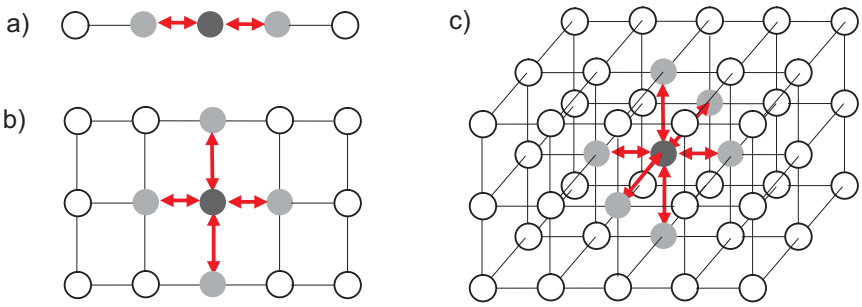


Abbildung 3.15: Anordnung der kommunizierenden Partner bei ein-, zwei- und drei-dimensionaler *topologischer exchange*-Kommunikation (Diagramme a) bis c)). Jeweils die beiden Partner für die Kommunikation in einer Dimension liegen im gleichen SCI-Ring.

Eine niedrigere logische Dimension läßt sich dabei direkt auf eine höhere physikalische Dimension abbilden; der umgekehrte Weg ist ebenfalls mit gewissen Randbedingungen möglich. Auf letzteren Fall wird an dieser Stelle jedoch nicht eingegangen, da von identi-

schen logischen und physikalischen Dimensionen ausgegangen wird.

Generelle exchange-Kommunikation. Die generelle *exchange*-Kommunikation mit n Partnern kann als sequentielle Abfolge von *pair*-Kommunikationsfällen wie oben beschrieben angesehen werden. Dazu muß jedoch davon ausgegangen werden, daß die Positionen aller Knoten der Paarungen die gleiche Zahl von Dimensionen auseinanderliegen. Dann ergibt sich das gleiche Skalierungsverhalten wie zuvor für die Fälle von *pair*-Kommunikation mit maximaler und minimaler Differenz der Dimensionen zwischen den Knotenpositionen.

Wenn im Allgemeinen jedoch nicht von dieser konstanten Dimensionsdifferenz der Knotenpositionen ausgegangen werden soll, kann diese statistisch ermittelt werden. Sei für einen d-dimensionalen Torus p_i die Wahrscheinlichkeit, daß ein Paket zwischen zwei zufälligen Knoten i-mal die Dimension wechseln muß. Dann ist die mittlere Dimensionsdifferenz Δ_k der Knotenpositionen von zwei zufällig ausgewählten Knoten

$$(3.12)\ \Delta_k = \sum_{i=0}^{d-1} p_i \cdot i$$

Die Wahrscheinlichkeiten p_i lassen sich als das Verhältnis der Zahl der Knoten, die mit i Dimensionswechseln erreicht werden können zur Gesamtzahl der Knoten im System ausdrücken. Die Gesamtzahl der Knoten ist stets g^d. Für $i = 0$ müssen die Zielknoten eines Pakets im selben Ring liegen wie der Quellknoten, so daß die Zahl der entsprechenden Zielknoten die Zahl der Knoten in einem Ring mal der Zahl der Ringe ist, an die der Quellknoten angeschlossen ist (was die Zahl der Dimensionen ist). Dabei darf der Quellknoten selber aber nur einmal berücksichtigt werden. Es ergibt sich daher für p_0:

$$(3.13)\ p_0 = \frac{d \cdot g - (d-1)}{g^d} = \frac{d \cdot g - d + 1}{g^d} = \frac{d(g-1)+1}{g^d}$$

Damit ein Zielknoten mit einem Dimensionswechsel ($i = 1$) erreicht werden kann, muß er in einer der drei Ebenen liegen, die zwei der Ringe, an die der Quellknoten angeschlossen ist, aufspannen. Von dieser Zahl müssen noch die Knoten in den selben Ringen des Quellknotens (die ja ohne Dimensionswechsel erreichbar sind) abgezogen werden. Damit ergibt sich für p_1:

$$(3.14)\ p_1 = \frac{d \cdot g^2 - d \cdot g - (d(g-1)+1)}{g^d} = \frac{d(g^2 - 2g + 1) + 1}{g^d} = \frac{d(g-1)^2 + 1}{g^d}$$

Die erkennbare Vorschrift aus (3.13) und (3.14) wiederholt sich auch für die höheren Dimensionen: Um einen Knoten innerhalb eines Torus der Dimension d in i Dimensionswechseln erreichen zu können, müssen diese Zielknoten innerhalb der $(i+1)$-dimensionalen Räume liegen, die von einem Knoten erreichbar sind. Da diese Räume sich überschneiden, muß sichergestellt sein, daß jeder der enthaltenen Knoten nur einmal gezählt wird. Darüberhinaus dürfen die Knoten, die in weniger als i Dimensionswechseln erreicht werden können, ebenfalls nicht berücksichtigt werden. Allgemein sind in einem Torus der Dimension d von einem Knoten aus eine bestimmte Zahl e_i von Räumen mit i Dimensionswechseln erreichbar. Diese Räume sind $(i+1)$-dimensional, daher ist e_i die Zahl der möglichen verschiedenen Ausrichtungen dieser Räume in dem gegebenen Torus. Dies entspricht einer Kombination ohne Wiederholung *[166]*:

$$(3.15)\ e_i = \binom{d}{i+1} = \frac{d!}{(i+1)!(k-i-1)!}$$

Die i-dimensionalen Überschneidungen dieser Räume sind gleichförmig mit einer konstanten Zahl von Knoten

$$(3.16)\ k_i = g^i$$

Da jeder hinzukommende Raum mit allen bereits markierten Räumen jeweils eine neue Überschneidung bildet, gibt es insgesamt

$$(3.17)\ n_k = \sum_{i=1}^{e_i} j = \frac{e_i(e_i+1)}{2}$$

dieser Überschneidungen mit zusammen

$$(3.18)\ n_e = g^i \cdot \frac{e_i(e_i+1)}{2}$$

Knoten.

Als allgemeine Formel ergibt sich zur Berechnung von p_i, wenn (3.15) in (3.18) eingesetzt wird, dies von der Zahl der Knoten in dem betrachteten Raum subtrahiert wird, ins Verhältnis zur Gesamtzahl der Knoten im Torus gesetzt wird und davon schließlich die Wahrscheinlichkeit für den niedrigeren Dimensionswechsel subtrahiert wird:

$$(3.19)\ p_i = \frac{g^{i+1} - \frac{g^i}{2} \cdot \binom{d}{i+1}\left(\binom{d}{i+1}+1\right)}{g^d} - p_{i-1} \text{ mit } p_{-1} = \frac{d-1}{g^d}$$

Damit lassen sich die maximalen effektiven Bandbreiten für den B-LINK und die SCI-Linksegmente berechnen, die in Abb. 3.17 (*links* und *rechts*) dargestellt sind.

Das Minimum von B-LINK- und SCI-Linksegmentbandbreite bestimmt die maximale Injektionsbandbreite pro Knoten, die in Abb. 3.16 *links* dargestellt ist. Die Akkumulation dieser Bandbreiten ergibt die Gesamtbandbreite des Netzes (siehe Abb. 3.16 *rechts*).

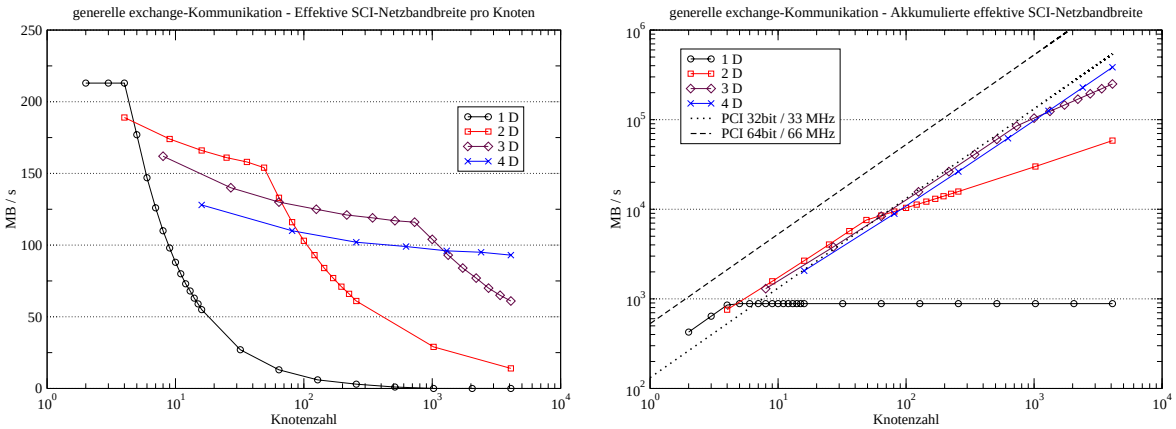


Abbildung 3.16: Effektive Injektionsbandbreite als Minimum der B-LINK- und SCI-Bandbreite pro Knoten (links) und für das Gesamtnetz akkumuliert (rechts) in Abhängigkeit von Knotenzahl und Torus-Dimension für *generelle exchange*-Kommunikation. Akkumulierte Bandbreite zusätzlich verglichen mit linear skalierender PCI-Bandbreite der Knoten.

Topologische exchange-Kommunikation. Die topologische exchange-Kommunikation für *d* Dimensionen läßt sich direkt auf die Ring-Kommunikation aus Kapitel 3.3.2.1 zurückführen, indem *d* Kommunikationsschritte nacheinander in jeweils eine Dimensionsrichtung durchgeführt werden. In jedem Kommunikationschritt findet jeweils eine minimale und eine maximale Ringkommunikation mit den beiden benachbarten Knoten im jeweiligen Ring statt. Damit ergibt sich unabhängig von der physikalischen Torusdimension eine Bandbreite, die dem arithmetischen Mittel von minimaler und maximaler Ringkommunikation entspricht. Die maximale Injektionsbandbreite pro Knoten und für das Gesamtnetz sind in Abb. 3.18 dargestellt.

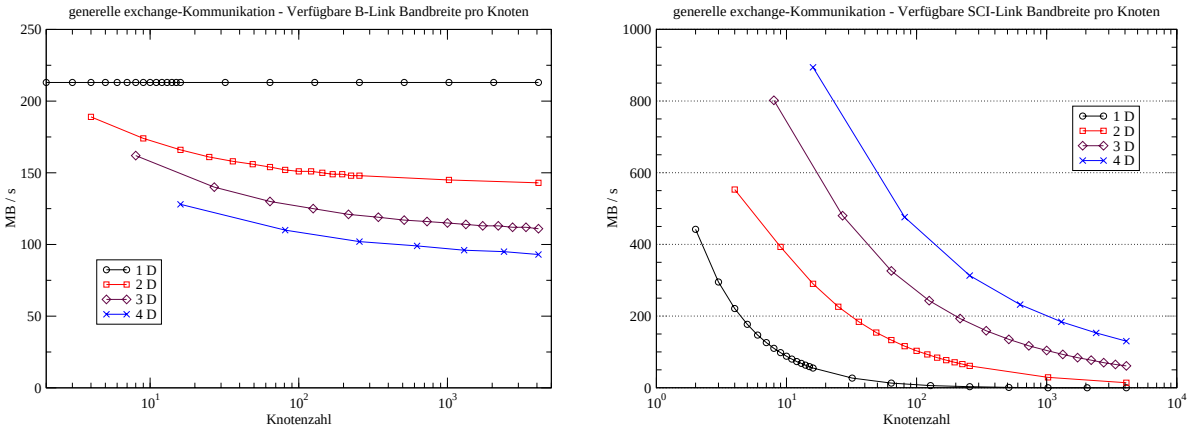


Abbildung 3.17: Limitierung der Injektionsbandbreite durch B-LINK-Bandbreite (*links*) und SCI-Link-Bandbreite (*rechts*) in Abhängigkeit von Knotenzahl und Torus-Dimension für *generelle exchange*-Kommunikation.

Der Vergleich der Leistung von genereller und topologischer exchange-Kommunikation zeigt, daß sich die Abstimmung von physikalischer und logischer Plazierung der kommunizierenden Prozesse um so stärker durch eine erhöhte Gesamtbandbreite bezahlt macht, je größer die Zahl der kommunizierenden Knoten pro Dimensionsrichtung ist. Dabei kommt es bei der generischen exchange-Kommunikation zu Sättigungseffekten, die bei der topologischen Exchange-Kommunikation nicht auftreten.

3.3.2.4 Alltoall-Kommunikation

In *[40]* wurde die Skalierbarkeit von Torustopologien für *alltoall*-Kommunikation untersucht, jedoch basierend auf Hardwareparametern, die inzwischen überholt sind. Daher wurden für diese Arbeit die Zahlenwerte neu berechnet. Abb. 3.19 zeigt den Verlauf der Bandbreitenbegrenzung durch die B-LINK- und die SCI-Link-Bandbreite gemäß den Überlegungen aus *[40]* mit den Werten aus Tabelle 3.2. Der B-LINK jedes Knotens wird durch Pakete, die von einer in eine andere Dimension des Torus wechseln müssen, mit zunehmender Zahl der Dimension des Torus immer stärker belastet. Beim eindimensionalen Fall bleibt diese Bandbreite konstant, da kein Paket eine Dimension wechseln muß. Der verfügbare Anteil an der SCI-Link-Bandbreite hingegen wächst mit steigender Torusdimension, da sich bei gleicher Gesamtknotenzahl jeweils weniger Knoten in einem gemeinsamen SCI-Link befinden.

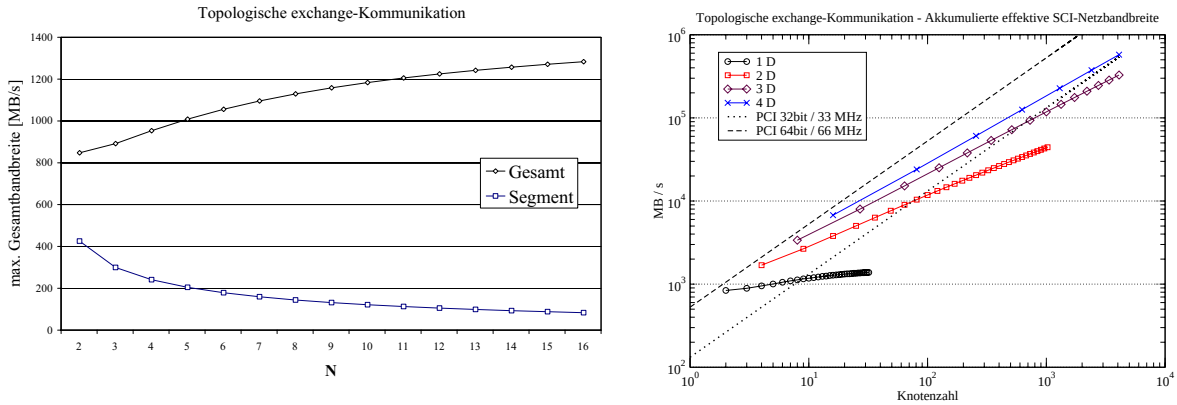


Abbildung 3.18: Topologische exchange-Kommunikation:
Links: Maximale Injektionsbandbreite auf die SCI-Linksegmente für die topologische Exchange-Kommunikation pro Segment und für das Gesamtnetz.
Rechts: Akkumulierte Gesamtbandbreite für verschieden-dimensionale Netze

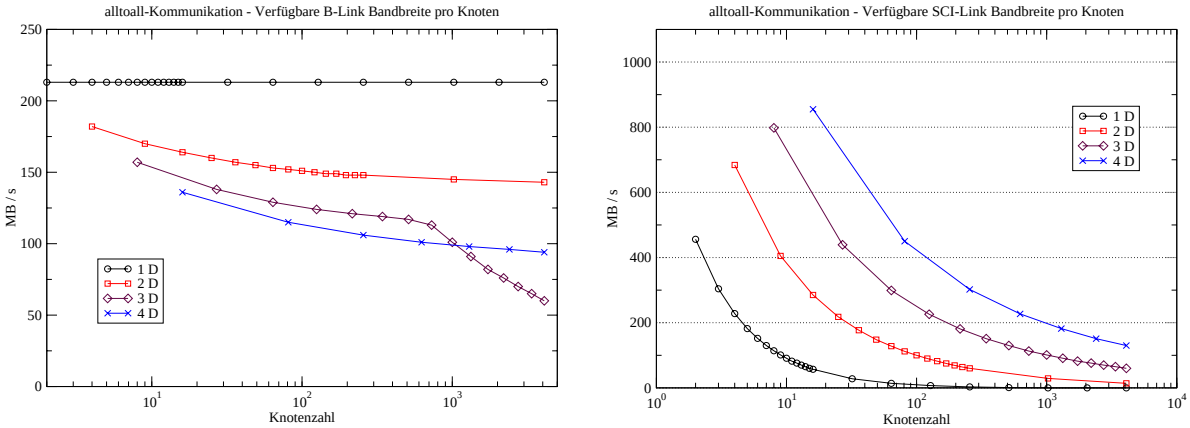


Abbildung 3.19: Limitierung der Injektionsbandbreite durch B-LINK-Bandbreite (*links*) und SCI-Link-Bandbreite (*rechts*) in Abhängigkeit von Knotenzahl und Torus-Dimension für *alltoall*-Kommunikation.

Aus diesem gegenläufigen Verhalten von maximalen Bandbreiten in Abhängigkeit von der Torusdimension ergibt sich, wie in Abb. 3.20 dargestellt ist, daß die optimale Topologie bezüglich des beschriebenen Skalierungskriteriums von der Zahl der Knoten in einem System abhängt. Bis zu 6 Knoten ist ein einfacher Ring optimal; bis zu 100 Knoten sollte ein zweidimensionaler Torus gewählt werden. Ein 3D-Torus ist bis 1000 Knoten dem bislang hypothetischem 4D-Torus überlegen.

Wird die Bandbreite pro Knoten mit der Zahl der Knoten im System multipliziert, erhält man die in Abb. 3.20 rechts dargestellte akkumulierte effektive Gesamtbandbreite für den *alltoall*-Kommunikationsfall, die natürlich nicht mit der akkumulierten Bandbreite aller Linksegmente gleichgesetzt werden darf. Letztere hat, da sie einen sehr speziellen Kommunikationsfall bedingt, weniger allgemeingültigen Wert. Zusätzlich sind in dieser Darstellung die akkumulierten Bandbreiten der knoteninternen PCI-Busse aufgetragen, einmal den leistungsschwächsten PCI-Bus (32 Bit Breite, 33 MHz Taktfrequenz) mit 132 MB/s Bandbreite und zum anderen die momentan leistungsstärkste Ausführung des PCI-Busses (64 Bit Breite, 66 MHz Taktfrequenz) mit 532 MB/s Bandbreite. Es wird offensichtlich, daß die leistungsschwache PCI-Variante (deren Maximalbandbreite in den verfügbaren Systemen darüberhinaus nur zu maximal 75% ausge-

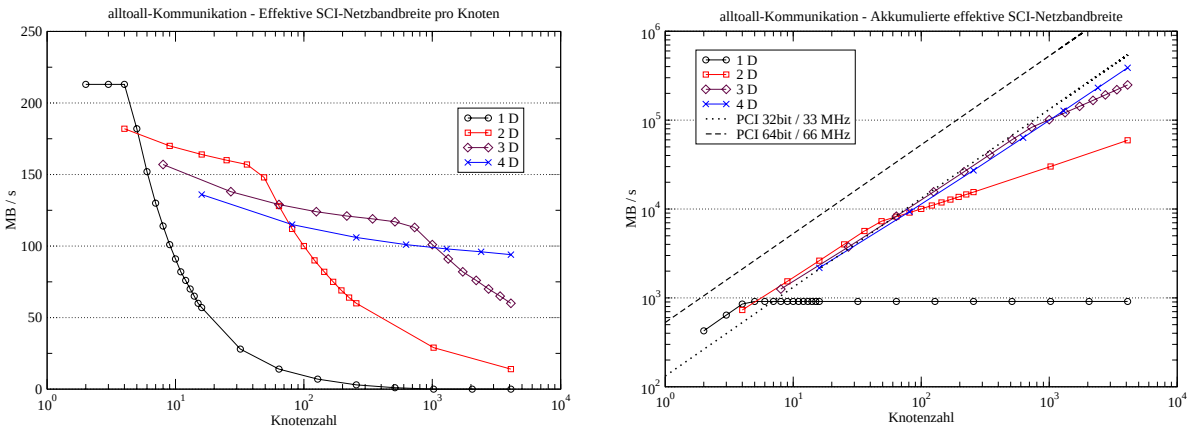


Abbildung 3.20: Effektive Injektionsbandbreite als Minimum der B-LINK- und SCI-Bandbreite pro Knoten (links) und für das Gesamtnetz akkumuliert (rechts) in Abhängigkeit von Knotenzahl und Torus-Dimension für *alltoall*-Kommunikation. Akkumulierte Bandbreite zusätzlich verglichen mit linear skalierender PCI-Bandbreite der Knoten.

nutzt werden kann) bis zu einer Knotenzahl von weit über 100 das SCI-Netz nicht auslasten kann. Andererseits zeigt die Variante des leistungsstärksten PCI-Busses, daß für den vorliegenden Anwendungsfall, der als typischer Belastungsfall von Nachrichtenaustausch-basierten Applikationen gelten kann, der Entwicklungsbedarf bezüglich dieser Skalierungseigenschaften zur Zeit noch nicht primär im Bereich der E/A-Bandbreite der Knoten, sondern im Bereich der Netzbandbreite liegt.

3.3.3 Kohärenz, Konsistenz und Integrität der Datenübertragung

Durch die Anbindung von SCI über den PCI-Bus ist es nicht möglich, entfernten Speicher im CPU-Cache zu puffern, da Geräte im PCI-Bus nicht am Cachekohärenz-Protokoll des Knotens teilnehmen können. Sie erhalten keine Informationen über Transaktionen, die die CPUs auf Hauptspeicher durchführen und die vom Prozessorcache abgewickelt werden. Insbesondere kann ein Datum, das eine CPU aus einem entfernten Speicherbereich gelesen hat und anschließend im Cache hält, nicht aktualisiert oder invalidiert werden, wenn es an anderer Stelle von einem anderen Prozessor geändert wird, da ein PSA keine Kenntnis von dem Zustand der Daten in den lokalen CPU-Caches hat.

Generelle hardwaregesteuerte Cachekohärenz über den globalen Adreßraum ist bei SCI, wenn es über den PCI-Bus angebunden ist, also nicht möglich. Jedoch besteht die Möglichkeit, die Kohärenz auf einer höheren Ebene, also softwaremäßig, sicherzustellen. Dies erfordert entsprechende Zwischenschichten, die nach oben hin ein angepaßtes Programmiermodell anbieten *[53]*. Bei solchen Modelle muß die Kohärenz jedoch explizit hergestellt werden.

Um aus ausgehenden PCI-Transaktionen möglichst effiziente SCI-Pakete (siehe Tabelle 3.1) zu generieren, werden im PSB Zwischenpuffer (sogenannte *stream buffer*) verwendet, um ausgehende Daten zu sammeln und schließlich zu einem SCI-Paket zusammenzusetzen (sogenanntes *write gathering*). Bei Lesezugriffen auf entfernten Speicher, die sich entsprechend der CPU-Befehlssätze immer auf einzelne Worte beziehen, dienen die *stream buffer* dazu, größere Datenpakete spekulativ anzufordern *(speculative read)* und anschließend lokal zur Bedienung der erwarteten weiteren Lesezugriffe bereitzuhalten. Diese Verfahren sind wichtig zur Leistungssteigerung. Sie erfordern jedoch eine explizite Konsistenzkontrolle, um sicherzustellen, daß geschriebene Daten zu einem bestimmten Zeitpunkt im entfernten Speicher angekommen sind bzw. gelesene Daten dem Zustand der Daten im entfernten Speicher zu einem bestimmten Zeitpunkt entsprechen. Dies erfolgt durch die Synchronisationskonstrukte *Speicherbarriere* (*store barrier*) und *Ladebarriere* (*load barrier*), bei denen auf bestimmte Kontrollregister (CSR) auf dem PSA zugegriffen wird.

Weiterhin sind nicht alle PCI-Implementationen vollständig korrekt, so daß bei der hohen Last, die die Kommunikation über einen oder mehrere PSA auf dem PCI-Bus erzeugt, Blockierungen auftreten können, bei deren Auflösung durch den PSA der Verlust von PCI-Transaktionen möglich ist. Der PSA erkennt solche Situationen, kann sie aber nicht beheben. Daher muß jeder Datentransfer[1] mittels eines sogenannten *sequence checks* auf Integrität überprüft werden. Auch hierbei wird ein bestimmtes Statusregister auf dem PSA abgefragt. Im Falle einer zwischenzeitlich aufgetretenen Blockierung mit Verlust von PCI-Transaktionen müssen die Daten erneut übertragen werden. Diese Maßnahme kann auf erwiesenermaßen korrekten PCI-Implementationen entfallen.

Für zuverlässige Datenübertragungsprotokolle müssen diese Eigenschaften und die notwendigen Maßnahmen in das Protokoll integriert werden, wie in Kapitel 4.2 dargestellt wird.

1. Die Granularität dieser Prüfung ist frei wählbar, solange nötigenfalls alle Datentransfers seit der letzten erfolgreichen Prüfung wiederholt werden können.

3.3.4 Verfügbarkeit der Kommunikationsressourcen

Im Gegensatz zu lokalem Speicher, dessen Verfügbarkeit durch die Software zu 100% angenommen wird, kann für ein entferntes eingeblendetes oder verbundenes Speichersegment nicht von dieser absoluten Verfügbarkeit ausgegangen werden, was ebenfalls durch *sequence checks* überprüft werden muß. Für Nicht-Verfügbarkeit gibt es zwei mögliche Ursachen:

- *Softwarebedingter Ausfall:* Der entfernte Prozeß entfernt sein lokales Speichersegment und somit den physikalischen Speicher hinter dem eingeblendeten oder verbundenen Speichersegment.

Auf diese Situation kann ein Übertragungsprotokoll vorbereitet sein und eine andere Protokollvariante wählen oder die entfernte Ressource erneut anfordern.

- *Hardwarebedingter Ausfall:* Durch einen Hardwarefehler ist der entfernte Knoten nicht mehr betriebsbereit, oder die Netzverbindung ist nicht mehr verfügbar, weil ein Knoten oder Switch, durch den die Pakete geleitet werden müssen, nicht mehr betriebsbereit ist.

Im Falle einer Störung der Netzverbindung kann gewartet oder ein anderer Weg für die Pakete bestimmt werden (*rerouting*). Nötigenfalls kann die Applikation geordnet abschließen.

3.4 Ressourcenbegrenzungen

Die einzige Ressourcenbegrenzung für lokalen Speicher ist seine Größe: Derjenige Speicher, der im System installiert ist, kann von einer oder mehreren Anwendungen vollständig genutzt werden, solange keine expliziten Beschränkungen vom Betriebssystem vorgesehen sind.

Die Nutzung von entferntem Speicher innerhalb der vorgestellten Architektur ist von der Verfügbarkeit von weitaus mehr Ressourcen abhängig. Diese Abhängigkeiten müssen entweder der Applikation bekannt sein oder von den Softwareschichten unterhalb der Applikation verwaltet werden. Ein entsprechendes Verfahren wird in Kapitel 4.4 für MPI-Applikationen vorgestellt.

3.4.1 Physikalische Ressourcen

Auf unterster Ebene werden Ressourcen benötigt, die durch die Architektur der genutzten Systemplattform und die konkrete Ausstattung des genutzten Systems bestimmt werden.

3.4.1.1 Physikalischer Speicher

Jedes SCI-Speichersegment muß auf dem Knoten, auf dem es angelegt wurde, mit physikalischem Speicher (PS) hinterlegt sein. Dieser Speicher darf nicht durch die virtuelle Speicherverwaltung (MMU) ausgelagert werden, da der PSA keinen Einfluß auf die MMU hat. Somit bestünde keine Möglichkeit, bei einem externen Zugriff auf ein lokales SCI-Speichersegment die ggf. ausgelagerte zugehörige Speicherkachel wieder vom Hintergrundspeicher in den Arbeitsspeicher zu transferieren. Um diese Situation zu verhindern, werden die Seiten entweder als *nicht auslagerbar* gekennzeichnet oder aus einem Pool von nicht auslagerbarem Speicher allokiert. Alternativ kann auch die virtuelle Speicherverwaltung vollständig auf Auslagerung verzichten, was aber aufgrund der möglichen drastischen Auswirkungen bei eintretendem Mangel an physikalischem Arbeitsspeicher (Abbruch von Applikationen) üblicherweise nicht praktiziert wird.

Die derartig als nicht-auslagerbar gekennzeichneten Speicherbereiche gehen der Speicherverwaltung, auf die ein Prozeß direkten Zugriff hat, verloren, so daß sich der den Applikationsprozessen zur Verfügung stehende Arbeitsspeicher verringert. Die Ausstattung mit Arbeitsspeicher muß bei der Dimensionierung des Pools nicht-auslagerbaren Speichers daher berücksichtigt werden.

3.4.1.2 Lokaler Adreßraum (LA)

Bei der Nutzung des durch SCI vorgegebenen globalen Adreßraums stellt möglicherweise die

Größe des auf einem Knoten verfügbaren Adreßraums eine signifikante Beschränkung dar. Während SCI pro Knoten einen Adreßraum von 2^{48} Byte vorsieht, sind aktuelle Systeme, die zum Aufbau von Clustern der in dieser Arbeit behandelten Art geeignet sind, durch die von der CPU bestimmte Systemarchitektur auf einen Adreßraum von 2^{32} Byte beschränkt. CPU-Architekturen der *64-Bit-Klasse* adressieren einen deutlich größeren Adreßraum, wodurch das Problem für zukünftige Anwendungsfälle nicht mehr relevant sein wird.

Innerhalb der Knoten wird jedoch ein weiterer Adreßraum bei der Nutzung des PCI-Busses wirksam, da dieser auf eigenen Adressen arbeitet.

3.4.2 PCI-SCI-Adapterressourcen

Neben den durch den SCI-Standard definierten Grenzen für den globalen SCI-Adreßraum und die Adressierung von SCI-Knoten und deren Speicherbereich existieren auch in der Implementation des Standards in Form der in dieser Arbeit genutzten PSA weitere Limitierungen *[31]*, die sich jedoch aufgrund der typischen Knotenzahl in heutigen Clustern, die in der weitaus überwiegenden Zahl von Fällen deutlich unter 1000 liegt, auf die Nutzung in diesen Systemen nicht auswirken und daher hier nicht behandelt werden.

Eine elementare Ressource auf den PSA ist die Adreßübersetzungstabelle (*Address Translation Table,* ATT*)*, deren limitierte Zahl von Einträgen (*Entries,* ATT-Eintrag) tatsächlich ein Problem darstellt.

3.4.2.1 Adreßübersetzungstabelle

Die ATT sorgt für die Transformation von Knoten-lokalen physikalischen Adressen in den globalen SCI-Adreßraum (der beim Zugriff auf entfernten Speicher verwendet werden muß). Die notwendigen Einträge liegen zum schnellen Zugriff in einem statischen RAM-Baustein. Die Kapazität dieses Speichers ist limitiert, wodurch eine maximale Zahl N_{ATT} von ATT-Einträgen bedingt ist. Im Gegensatz zu einer Speicherung der ATT im Hauptspeicher, die auf früheren PSA-Modellen noch möglich war *[28]* und eine praktisch unbegrenzte Größe des ATT erlaubte, ist diese Art der Speicherung deutlich schneller. Da bei jedem Zugriff auf entfernten Speicher mindestens ein ATT-Eintrag gelesen werden muß, ist dies ein wichtiger Leistungsfaktor.

Die momentane Treibersoftware kann die für ein EES notwendige Zahl von ATT-Einträgen nur als zusammenhängenden Block verwalten. Eine mögliche Fragmentierung der ATT durch wiederholtes Allokieren und Freigeben verschieden großer Blöcke kann so zu der Situation führen, daß trotz einer ausreichenden Zahl von freien ATT-Einträgen eine notwendige Allokation nicht möglich ist. Dieses ist jedoch kein durch die Architektur bedingtes Problem, sondern nur durch die softwaremäßige Verwaltung der ATT-Einträge bedingt. Ein Konzept zur Umgehung dieses Problems bei der Datenübertragung zum Nachrichtenaustausch wird in Kapitel 7.2 vorgestellt.

3.4.2.2 Maximal einblendbarer Speicher

Jeder ATT-Eintrag bestimmt die Transformation eines Adreßbereichs fester Größe, die als ATT-Seitengröße bezeichnet wird. Diese ATT-Seitengröße S_{ATT} kann auf Werte von 2^n, $12 \le n \le 20$, eingestellt werden. Die maximale Größe S_{EES} des einblendbaren entfernten Speichers berechnet sich dann als Produkt der ATT-Seitengröße und Anzahl der verfügbaren ATT-Einträge:

$$(3.20)\quad S_{EES} = N_{ATT} \cdot S_{ATT}$$

Diese Größe ist auf einen PSA bezogen, mehrere PSA in einem Knoten erhöhen diesen Wert entsprechend. Es müssen sich jedoch alle Prozesse auf einem Knoten diese verfügbare Größe teilen, auch wenn sie die gleichen entfernten Speicherbereiche einblenden, da die ATT-Einträge pro eingerichtetem EES verwaltet werden und eine gemeinsame Nutzung von ATT-Einträgen

für verschiedene EES, die sich auf die gleichen physikalischen Speicherseiten auf einem entfernten Knoten beziehen, von der Treibersoftware nicht unterstützt wird (obgleich durch die Architektur möglich).

3.4.3 Softwareressourcen

Neben den beschriebenen hardwarebedingten Limitierungen von Ressourcen, die zur Nutzung von SCI in Clustern erforderlich sind, existieren auch in den darüberliegenden Softwareschichten Limitierungen. Diese sind durch die nötige Einbindung des PSA in das Betriebssystem mittels der Treibersoftware bedingt. Es müssen dabei existierende Mechanismen und Datenstrukturen genutzt werden, um einen sicheren und verläßlichen Betrieb des PSA in allen möglichen Nutzungssituationen zu gewährleisten. Dazu müssen durch das Betriebssystem vorgegebene Verfahren genutzt werden, um einen geregelten Zugang zu den Hardwareressourcen zu gewährleisten, bis diese schließlich (etwa im Fall von eingeblendetem Speicher bei SCI) eingerichtet und direkt vom Benutzerprozeß genutzt werden können.

3.4.3.1 Physikalisch zusammenhängender Speicher

Ein Speicherbereich, der als SCI-Speichersegment exportiert werden soll, muß gegenwärtig zwei Eigenschaften erfüllen:

- Die physikalischen Speicherseiten dürfen nicht von der Speicherverwaltung ausgelagert werden (siehe Kapitel 3.4.1.1)
- Die physikalischen Speicherseiten, aus denen sich der Speicherbereich zusammensetzt, müssen zusammenhängend sein, d.h. der Speicherbereich muß durch eine physikalische Adresse und die Länge vollständig beschrieben sein (*physikalisch zusammenhängender Speicher,* PZS*).*

PZS wird vom BS-Kern verwaltet und wird üblicherweise nur in relativ kleinen Mengen als Puffer für herkömmliche E/A-Geräte benötigt, die ihre Daten über DMA-Operationen in den Kernadreßraum übertragen.

Die Menge an PZS, die standardmäßig zur Verfügung steht, ist daher für die Nutzung von SCI im größeren Maßstab unzureichend. Für das BS Linux wird daher eine Modifikation des Kerns verwendet, die beim Start des Systems einen physikalisch zusammenhängenden Speicherbereich fester Größe S_{PZS} allokiert und der Speicherverwaltung des Systems entzieht. Stattdessen kann aus diesem Bereich über andere Kernfunktionen Speicher mit der benötigten Eigenschaft allokiert werden. S_{PZS} stellt somit die obere Grenze für die Größe aller SCI-Speichersegmente dar, die von einem Knoten exportiert werden können. Dies bedeutet auch, das Speicher für lokal anzulegende SCI-Speichersegmente grundsätzlich durch den Treiber allokiert werden muß.

Mit erhöhtem Softwareaufwand, der sich allerdings auch in einer Leistungseinbuße bei der Kommunikation niederschlägt, läßt sich jedoch sowohl diese Obergrenze für die Größe der lokalen SCI-Speichersegmente aufheben als auch die erforderlich Speicherallokation durch den SCI-Treiber umgehen. Dies wurde für PIO-basierte Kommunikation in *[53]* gezeigt; für DMA-basierte Kommunikation wird eine Lösung in Kapitel 7.2 vorgestellt, die im Zusammenhang mit *[52,113]* entwickelt wurde. Beide Verfahren lassen sich zu einem generellen Lösungsverfahren vereinigen.

3.4.3.2 Segmente von gemeinsamem Speicher

Sowohl lokale als auch entfernte SCI-Speichersegmente, die in den Adreßraum eingeblendet werden, sind im Sinne des BS gemeinsamer Speicher (*shared memory*). Die Größe und Zahl von *shared-memory*-Segmenten ist durch Parameter im Kern des BS nach oben begrenzt. Diese Grenzen lassen sich jedoch durch eine entsprechende Systemkonfiguration soweit erhöhen, daß sie in Relation zum maximalen physikalischen Adreßraum eines Prozesses auf 32-Bit-Systemen keine Einschränkung mehr darstellen.

3.4.3.3 Gerätedeskriptoren

Die Schnittstelle zwischen dem SCI-Treiber im Kernadreßraum und den Prozessen im Benutzeradreßraum wird durch ein sogenanntes Gerät (*device)* gebildet, das sich unter Unix-nahen BS über spezielle Dateien ansprechen läßt. Zur Kommunikation mit dem SCI-Treiber, um etwa ein SCI-Speichersegment anzulegen oder sich an ein entferntes SCI-Speichersegment zu verbinden, öffnet ein Prozeß eine solche Datei und schreibt Kommandos in diese Datei bzw. liest Antworten des SCI-Treibers aus derselben. Beim Öffnen einer solchen Datei erhält ein Prozeß einen *SCI-Deskriptor* (SD) zurück, mit dem er sich auf die zugehörigen Ressourcen beziehen kann. Jedoch kann über jeden Deskriptor nur eine lokale und eine entfernte (auf einem anderen Knoten allokierte) Ressource angesprochen werden. Zusätzlich ist die Zahl N_{SD} der maximal gleichzeitig verfügbaren Deskriptoren für ein Gerät auf einem Knoten beschränkt.

SCI-Objekt	PZS	PS	LA	ATT	SD[a]
LS	~ Größe	~ Größe	~ Größe	0	lokal
LRS	0	~ Größe	~ Größe	0	lokal
EES	0	0	~ Größe	~ Größe (> 0)	entfernt
EVS	0	0	0	0	entfernt
LI	-	-	-	-	lokal
EI	-	-	-	-	entfernt

Tabelle 3.4: Verbrauch von Systemressourcen durch die verschiedenen SCI-Objekte.

a. 'lokal' bzw.'entfernt' bedeutet, daß jeweils ein lokaler bzw. entfernter Anteil eines SCI-Deskriptors verwendet wird.

3.4.4 Ressourcenverbrauch von SCI-Objekten

Basierend auf den im vorhergehenden Abschnitt dargestellten Limitierungen der Systemressourcen für die Nutzung von SCI wird in Tabelle 3.4 dargestellt, in welchem Maße die in Kapitel 3.2.2 aufgeführten SISCI-Ressourcen zur SCI-Kommunikation (SCI-*Objekte*) diese Systemressourcen benötigen bzw. verbrauchen. Die aufgeführten SCI-Objekte sind lokale SCI-Speichersegmente (LS), lokale registrierte SCI-Speichersegmente (LRS), entfernte eingeblendete Speichersegmente (EES), entfernte verbundene Speichersegmente (EVS) sowie lokale und entfernte SCI-Interrupts (LI bzw. EI). Alle anderen durch die SISCI-Schnittstelle verfügbaren SCI-Objekte verbrauchen keine der angeführten Systemressourcen, so daß sie hier nicht dargestellt werden. Zusätzlich zu den angeführten Systemressourcen verbrauchen jedoch alle in der Tabelle aufgeführten SCI-Objekte lokalen virtuellen Speicher für die notwendigen Datenstrukturen zur Verwaltung. Der Verbrauch dieser Systemressource ist jedoch für die Betrachtungen in diesem Zusammenhang unwesentlich und wird daher nicht dargestellt.

3.5 Entwicklungs- und Untersuchungssystem

Die Entwicklungen und Untersuchungen, die in dieser Arbeit vorgestellt werden, entstanden auf einem am Lehrstuhl für Betriebssysteme gebauten SCI-gekoppelten Cluster. Zur Referenz wird das System mit den relevanten technischen Daten vorgestellt.

3.5.1 Systemplattform

Das System (im folgenden mit dem Kürzel P3 bezeichnet) entspricht dem aktuellen Stand der Technik für preisgünstige Clustersysteme. Es besteht aus 8 identischen Knoten, die mittels eines SCI-Rings gekoppelt sind. Alle Messungen und Leistungswerte im Rest der Arbeit, bei denen keine Angabe zum verwendeten System gemacht sind, wurden auf diesem System durchgeführt bzw. gewonnen.

Konfiguration der Knoten:

- Dual-Prozessor-Systeme
  - Systemplatine Supermicro 370DLE *[35]*
  - Chipsatz Serverworks ServerSet-III LE (Revision 6) *[34]*
- je 2 mal Intel Pentium-III CPU (800 MHz), Coppermine Kern *[33]*
  - 16kB 1st-Level-Cache (jeweils Daten und Code)
  - 256kB 2nd-Level-Cache (mit vollem CPU-Takt betrieben)
- 512 MB Hauptspeicher (synchrones DRAM, PC133-ECC)
- PCI-Bus mit 64 Bit Datenbreite, 66 MHz Taktfrequenz (max. 532 MB/s Bandbreite)
- Betriebssystem Linux, Kernelversion 2.4.4

Konfiguration des SCI-Verbindungsnetzes:

- PCI-SCI-Adapter D330 (Dolphin Interconnect Solutions *[30,31]*) mit LC-3 Linkcontroller
  - SCI-Linkbandbreite 800 MB/s, B-LINK-Bandbreite 640 MB/s
- Ringtopologie

3.5.2 Verfügbare Ressourcen für SCI

Für die Nutzung eines SCI-gekoppelten Clusters ist die Kenntnis der verschiedenen SCI-relevanten Ressourcen wichtig. Daher sind in Tabelle 3.5 die auf der Plattform P3 verfügbaren Ressourcen, die für die Nutzung von SCI bedeutsam sind, aufgeführt. Die Handhabung dieser Ressourcen wird in Kapitel 4.4 vertieft behandelt.

Ressource	Einheit	P3	Kürzel
Physikalischer Hauptspeicher	MB	512	SPS
Prozeßadressraum	MB	2048	SPA
Nutzbare ATT-Einträge	-	8192	NATT
ATT-Seitengröße	kB	64	SATT
Physikalisch zusammenhängender Speicher	MB	64	PZS
SCI-Deskriptoren	-	256	NSD
Virtuelle Kanäle	-	1024	NVC

Tabelle 3.5: Verfügbare SCI-relevante Ressourcen für Clusterplattform P3

KAPITEL 4

Grundlagen des Nachrichtenaustauschs

Kommunikation ist die Grundlage für die Lösung von Problemen durch Parallelverarbeitung, sofern die durch Zerlegung des Hauptproblems entstandenen Teilprobleme nicht völlig unabhängig voneinander sind. Dies ist etwa bei der Suche in Parameterräumen der Fall; diese Klasse Problemen soll hier jedoch nicht betrachtet werden.

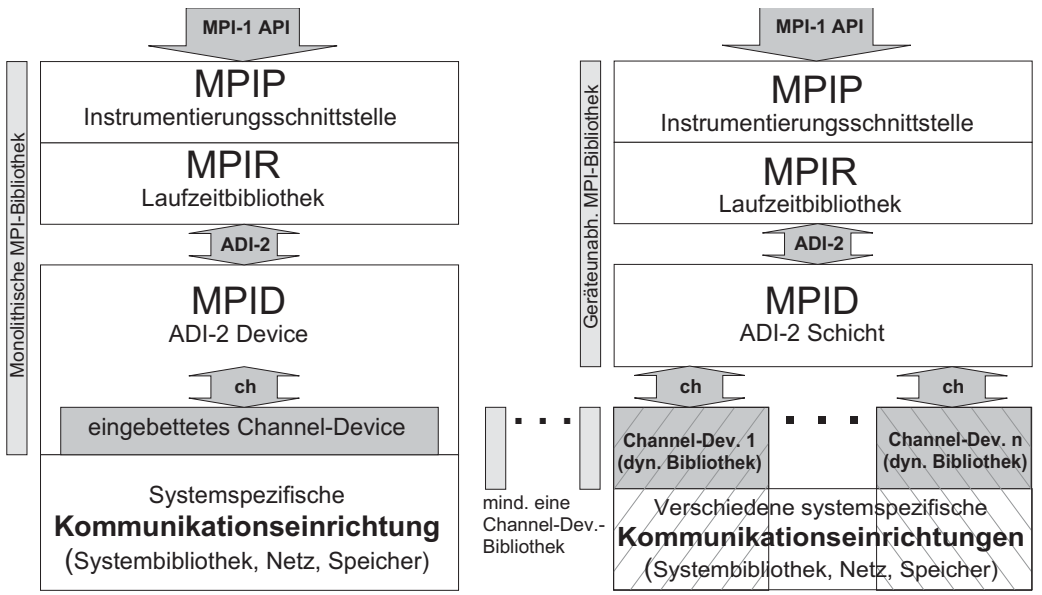
Sobald die Kommunikation von parallelen Ausführungssträngen nicht mehr über den Zugriff auf gemeinsame Datenstrukturen erfolgt, müssen dazu Nachrichten eingesetzt werden, die versandt und empfangen werden. Dazu ist eine definierte Schnittstelle erforderlich, bestehend aus einer Zahl von Funktionen, Datentypen und Konstanten. In dieser Arbeit wird die MPI-Schnittstelle für den Einsatz auf Clustern mit SCI-Verbindungsnetz implementiert. Dabei werden zahlreiche Optimierungen durch Ausnutzung der speziellen Eigenschaften dieses Verbindungsnetzes realisiert.

4.1 MPI-Implementation SCI-MPICH

Die Basis für die Entwicklungen dieser Arbeit im Bereich des Nachrichtenaustauschs ist die frei im Quelltext erhältliche MPI-1 Implementation MPICH *[71,72]*. Sie diente als Referenzimplementation bei der Einführung des MPI-Standards und ist bei stetiger Weiterentwicklung *[73]* die am meisten verwendete MPI-Implementierung. Dies gilt um so mehr, als auch einige kommerzielle *[80]* bzw. plattformspezifische MPI-Implementierungen *[85]* auf MPICH basieren.

4.1.1 Aufbau von MPICH

Der Aufbau der zentralen Bibliothek von MPICH, welche die gesamte MPI-1-Funktionalität anbietet, ist in Abb. 4.1 (a) dargestellt. Die Bibliothek ist intern in drei Schichten sowie eine Pseudoschicht unterteilt, die durch festgelegte Schnittstellen miteinander kommunizieren.



(a) Aufbau von MPICH

(b) Allgemeiner Aufbau von MP-MPICH

Abbildung 4.1: Aufbau der MPI-Bibliothek und Einbindung der Kommunikationsmodule in MPICH und SCI-MPICH

4.1.1.1 Profilingschicht MPIP

Die MPIP Schicht ist eine Pseudoschicht, die es ermöglicht, daß sämtliche MPI-Aufrufe durch eine andere Bibliothek geleitet werden. Diese fremde Bibliothek kann die im Aufruf enthaltenen Informationen sammeln und verwerten und ruft anschließend die Originalfunktion auf. Dieses Verfahren findet vorwiegend in der automatischen Instrumentierung Anwendung, beispielsweise in MPE *[71]* und Vampir *[77]*.

Das API der MPIP Schicht ist identisch mit dem von der unterliegenden MPIR Schicht implementierten Teil des MPI Standards.

4.1.1.2 Laufzeitschicht MPIR

Die MPIR Schicht bildet alle verfügbaren Funktionen des MPI API auf die Punkt-zu-Punkt Sende- und Empfangs-Funktionen der ADI-2 Schnittstelle ab. Darin ist auch eine generische Implementation der kollektiven Operationen enthalten, die auf Punkt-zu-Punkt-Kommunikationsoperationen abgebildet werden. Die MPIR-Schicht kommuniziert mit der unterliegenden MPID-Schicht über die ADI-2-Schnittstelle (s.u).

Daneben wickelt die MPIR-Schicht alle weiteren Funktionen ab, die nicht unmittelbar zu einem Datentransfer führen, etwa die Verwaltung von Kommunikatoren und Datentypen.

4.1.1.3 Kommunikationsschicht MPID

Die Aufgaben der MPID Schicht sind die Verwaltung von laufenden sowie persistenten[1] Sende- und Empfangsoperationen, Datenkonvertierung zur Kommunikation mit Systemen mit anderer Zahlendarstellung sowie die Zusammenfassung nicht-zusammenhängender Sendedaten in einen durchgängigen Sendepuffer. Basierend auf diesen Informationen überprüft die MPID-Schicht die untergeordneten Kommunikationskanäle auf eingehende Nachrichten und verschickt ausgehende Nachrichten über den jeweils geeigneten Kanal mittels der Funktionen der *CH-Device*-Schnittstelle.

Obwohl die MPID-Schicht in den elementaren Datenstrukturen eine beliebige Zahl von Kommunikationsmechanismen in Form von *CH-Devices* (s.u.) unterstützt, ist deren Einbindung an vielen Stellen derart gestaltet, daß der Betrieb nur mit einem einzigen CH-Device möglich ist. Dies trifft insbesondere auf die Initialisierung und Abfrage der verschiedenen CH-Devices zu. Letztendlich wird der Prozeß erst durch die Initialisierung des CH-Devices Teil der MPI-Applikation, da dort die Synchronisation mit den anderen Prozessen stattfindet. Die Art und Weise, wie dies geschieht, ist aber nicht definiert, so daß verschiedene CH-Devices miteinander unverträgliche Verfahren verwenden können, was auch die Bestimmung des eigenen Rangs des Prozesses innerhalb der Applikation anbelangt. Bei der Abfrage der CH-Devices auf neu eingegangene Nachrichten werden z.T. blockierende Aufrufe verwendet, so daß beim Warten auf eine neue Nachricht in einem CH-Device nicht auf eine neue Nachricht reagiert werden kann, die über ein anderes CH-Device eintrifft.

4.1.1.4 Geräteschicht CH

Die CH-Schicht[2] ist die unterste Schicht von MPICH und interagiert direkt mit jeweils einem verfügbaren Kommunikationsmechanismus des Systems. Dies kann über eine Benutzer- oder Systembibliothek oder auch direkt über die Hardware erfolgen. Die Funktionalität, die über die CH-Schnittstelle angeboten wird, besteht aus Punkt-zu-Punkt Sende- und Empfangsoperationen, basierend auf bis zu drei verschiedenen Protokollen, sowie einer Funktion zur Prüfung auf neu eingegangene Nachrichten. Dazu kommen Funktionen zum regulären (kooperativen) Herunterfahren der Kommunikation sowie zum Prozeß-individuell motivierten Abbruch der Appli-

1. Zu wiederholten Verwendung eingerichtete Sende- oder Empfangsoperationen.
2. CH steht sowohl für *Chameleon*, eine andere Kommunikationsbibliothek der Autoren, auf der MPICH konzeptionell aufbaut, als auch für *channel* als Konzept des minimalen plattformabhängigen Kommnikationskerns, und hat MPICH den Namen gegeben.

kation im Fehlerfall.

Die Sende- und Empfangsprotokolle werden entsprechend der Größe der zu versendenden Nachricht ausgewählt (siehe Kapitel 4.2.1). Jedes Protokoll wiederum beinhaltet Funktionen zum blockierenden (synchronen) und nicht-blockierenden (potentiell asynchronen) Versand und Empfang von Nachrichten. Nicht-blockierende Kommunikation wiederum besteht jeweils aus zwei Funktionen zum Start des Vorgangs und zur Überprüfung auf Fertigstellung.

4.1.2 Erweiterungen in SCI-MPICH

Gegenüber einer völligen Neuentwicklung ist der Vorteil der erfolgten Weiterentwicklung von MPICH zu SCI-MPICH, daß die Entwicklung schrittweise erfolgen kann und dennoch stets eine vollständige MPI-1 Implementation vorhanden ist, die ohne Einschränkungen mit vorhandenen Testprogrammen auf Korrektheit und Leistung getestet und ebenso mit Applikationen verwendet werden kann. Da die Struktur von MPICH einer Erweiterung der Funktionalität nicht im Wege steht (wie später gezeigt wird), konnte auf diese Weise die erneute Implementation von Funktionalität in Bereichen vermieden werden, für die getesteter Code existiert und in denen alternative Implementierungen keine relevanten Leistungsverbesserungen versprechen.

Auf die Einbindung von SCI als Kommunikationsmedium als Teil der Erweiterung von MPICH zu SCI-MPICH wird im weiteren Verlauf dieser Arbeit detailliert eingegangen. Viele der Vorbedingungen für diese Einbindung sind jedoch außerhalb des eigentlichen SCI-Kommunikationsmoduls (CH-Device) geschaffen worden und daher als allgemeine Erweiterung auch für andere Kommunikationsdevices nutzbar.

4.1.2.1 Multidevice-Unterstützung

In einigen grundlegenden Datenstrukturen ist MPICH bereits auf den Einsatz von mehr als einem Kommunikationsdevice zugleich vorbereitet. Ein solcher Betrieb ist beispielsweise sinnvoll, um zum Betrieb von gekoppelten SMP-Knoten die Intra-Knoten-Kommunikation über gemeinsamen Speicher und die Inter-Knoten-Kommunikation über ein getrenntes Device für das jeweilige Verbindungsnetz abzuwickeln *[79]*. Trotz der konzeptionellen Vorbereitung fehlten für die tatsächliche Umsetzung dieses Konzepts Verfahren für die Initialisierung und Abfrage mehrerer Devices. Durch die Einführung dynamisch nachladbarer CH-Devices und die durchgängig nicht-blockierende Abfrage der aktiven CH-Devices auf neue Nachrichten wurden diese Probleme gelöst. Dazu war auch eine Umstrukturierung der Codebasis in voneinander unabhängig übersetz- und bindbare Module notwendig, wie dies in Abb. 4.1 dargestellt ist. Die monolitische MPI-Bibliothek von MPICH (*links*) wurde aufgeteilt in einen geräte- und betriebssystemunabhängigen Teil (MPIP-, MPIR- und MPID-Schichten, *rechts*), der nun die MPI-Bibliothek darstellt. Dazu kommen beliebig viele CH-Devices, die zum Betrieb ein bestimmtes Kommunikationsgerät voraussetzen.

4.1.2.2 Einseitige Kommunikation

Die im MPI-2-Standard definierte Schnittstelle zur *einseitigen Kommunikation [64]* wurde implementiert. Dazu gehören Funktionen zur Verwaltung der für den Zugriff durch entfernte Prozesse freigegebenen Speicherbereiche und das vollständige API in der MPIR-Schicht sowie die zugehörigen Kommunikations- und Synchronisationsroutinen in der MPID-Schicht. Der wesentliche Teil zur Implementation von einseitiger Kommunikation liegt jedoch im CH-Device, da die möglichst effiziente Durchführung der Kommunikationsvorgänge eine enge Anbindung an das verwendete Kommunikationsgerät erfordert. Im Rahmen dieser Arbeit erfolgte eine Implementierung von einseitiger Kommunikation über gemeinsamen Speicher und SCI (siehe Kapitel 5.2).

4.1.2.3 Datentypkommunikation im CH-Device

Die Kommunikation in MPI basiert auf *Datentypen*, welche die zu versendenden oder zu empfangenden Speicherstellen im Adreßraum des Prozesses beschreiben. Es gibt eine Reihe von

standardmäßig definierten Datentypen, von denen man neue, applikationsspezifische Datentypen ableiten kann. Diese können auch nicht-zusammenhängende Speicherbereiche (NZS) beschreiben. Um solche NZS zu verschicken, kopiert MPICH die enthaltenen Daten in einen entsprechend großen zusammenhängenden Speicherbereich um ("packen" genannt). Anschließend werden sie wie üblich verschickt, um beim Empfänger wiederum "entpackt" zu werden, um so die gleiche Darstellung wie beim Sender wiederzuerlangen.

Für viele Kommunikationsgeräte ist dies der einzig sinnvolle Weg, solche NZS zu übertragen. Für andere Kommunikationsgeräte, die auch kleine Dateneinheiten mit hoher Bandbreite übertragen können, bedeuten die zusätzlichen Kopiervorgänge beim Packen und Entpacken jedoch unnötigen Aufwand. SCI-MPICH ermöglicht es daher solchen Kommunikationsgeräten, direkt auf die NZS zuzugreifen und die enthaltenen Daten auf die optimale Weise zu übertragen.

Ein solches Verfahren, wiederum für die Kommunikation über gemeinsamen Speicher und SCI, wird in Kapitel 5.1 vorgestellt.

4.1.2.4 Kommunikatorinitialisierung

Das Konzept der *Kommunikatoren* ist ein zentraler Bestandteil von MPI, da sich alle Kommunikationsfunktionen auf einen Kommunikator beziehen. Für die MPI-Applikation bezeichnet ein Kommunikator ein Teilmenge von Prozessen aus der Gesamtmenge der Prozesse, welche die Applikation bilden.

Innerhalb der MPI-Bibliothek beinhaltet ein Kommunikator eine Vielzahl von Informationen, beispielsweise Attribute und Fehlerbehandlungsroutinen. Jeder Kommunikator kann für die enthaltenen Prozesse eigene Funktionen als Implementation von kollektiven Operationen bereitstellen, welche die generischen Implementationen in der MPI-Bibliothek ersetzen und für ein bestimmtes Kommunikationsgerät oder die aktuelle Topologie der Prozesse im Kommunikator (bezogen auf ihre Plazierung in einem hierarchischen System wie etwa gekoppelten SMP-Knoten) optimiert sind. Daher wird bei der Initialisierung eines neuen Kommunikators dieser auch an alle beteiligten CH-Devices gereicht, die dann die erforderlichen Einträge im Kommunikator vornehmen können.

4.1.2.5 Benutzergesteuerte Speicherverwaltung und persistente Kommunikation

Im Zusammenhang mit einseitiger Kommunikation ist die Implementation von MPI-eigenen Funktionen zur Speicherallokierung (`MPI_Alloc_mem` und `MPI_Free_mem`) wichtig. Dadurch kann die Applikation mittels der MPI-Bibliothek Speicher allokieren, der aufgrund systemspezifischer Eigenschaften für die einseitige Kommunikation besonders geeignet ist.

Es macht aber auch Sinn, für herkömmliche (zweiseitige) Kommunikation den Speicher für häufig genutzte Sende- und Empfangspuffer über diese Funktionen zu allokieren, um dem Kommunikationsgerät einen besonders effizienten Umgang mit der Speicherressource zu ermöglichen. Alternativ dazu kann auch *persistente Kommunikation* eingerichtet werden, bei der Sende- oder Empfangspuffer von der Applikation in der MPI-Bibliothek angemeldet werden, wenn sie für viele gleichartige Kommunikationsoperationen verwendet werden sollen. Dies ermöglicht der MPI-Bibliothek beispielsweise, den Speicher an das Kommunikationsgerät zu "binden", um Sende- und Empfangsoperationen effizienter abwickeln zu können.

Beide Vorgänge, Speicherallokation und Verwaltung persistenter Kommunikation, können in SCI-MPICH vom CH-Device abgewickelt werden, wenn dieses die nötige Funktionalität anbietet (ansonsten werden generische Funktionen verwendet). Ein Beispiel für die Nutzung dieser Möglichkeit wird in Kapitel 7.2 sowie in Kapitel 5.2 gegeben.

4.1.2.6 Modifikationen der internen Schnittstellen

Die oben vorgestellten Maßnahmen sowie weitere Veränderungen, die hier nicht beschrieben werden, ließen sich nicht ohne Änderungen an den Schnittstellendefinitionen realisieren. Dabei konnte der Teil der ADI-2 Schnittstelle, der von der MPIR-Schicht genutzt wird, unverändert ge-

lassen werden, wodurch die Kompatibilität zwischen diesen Schichten erhalten blieb[1]. So können weiterhin Entwicklungen in der MPIR-Schicht von MPICH ohne Änderungen in SCI-MPICH übernommen werden. Jedoch wurde jeder der drei Funktion zum allgemeinen Senden und Empfangen von zusammenhängenden Daten eine entsprechende Funktion für den Fall nicht-zusammenhängender Datentypen zur Seite gestellt. Letztere werden verwendet, wenn das CH-Device mit nicht-zusammenhängenden Datentypen umgehen kann, und reichen den Datentyp entsprechend an das CH-Device durch.

An der CH-Schnittstelle zum CH-Device waren jedoch zahlreiche Änderungen und Erweiterungen möglich, um die erweiterte Funktionalität des CH-Device für die oberen Schichten nutzbar zu machen. Insgesamt bedeuten diese Entwicklungen natürlich, daß der Anteil der Funktionalität der gesamten MPI-Implementation, der im CH-Device liegt, gegenüber der generischen Funktionalität in der MPI-Bibliothek erhöht wird. Dies wiederspricht auf den ersten Blick dem Ansinnen, Middleware-Softwareschichten, wie MPI sie darstellt, möglichst portabel zu halten. Bei genauerer Betrachtung ist dies bei dem vorgestellten Konzept jedoch aus mehreren Gründen nicht der Fall. Zunächst ist die erweiterte Funktionalität des CH-Devices optional: Weiterhin besteht die hinreichende minimale Funktionalität aus *Senden & Empfangen einer Kontrollnachricht, Prüfen auf neue Kontrollnachricht* sowie *Senden & Empfangen eines zusammenhängenden Datenblocks.* Um im Fall von MPI jedoch möglichst hohe Bandbreiten, niedrige Latenz und gute Skalierung zu erreichen, müssen die verfügbaren Kommunikationsgeräte spezifisch genutzt werden. Dies führt tatsächlich dazu, daß große Teile eines solchen optimierten CH-Devices nur im Zusammenhang mit diesem Kommunikationsgerät nutzbar sind. Es sind zwar wiederum Softwareschichten denkbar, die die Softwareschnittstellen eines Kommunikationsgeräts auf einem anderen Kommunikationsgerät anbieten[2]. Hierbei gehen häufig die speziellen Leistungseigenschaften des originären Kommunikationsgeräts verloren, für die das CH-Device ja optimiert wurde. Dieses Problem ist nicht länger relevant, wenn das Kommunikationsgerät selber portabel wird. Dies ist für den Bereich von Verbindungsnetzen für PC/Workstation-Cluster durch die Homogenisierung der Systemanbindung über den PCI-Bus der Fall: Sowohl SCI als auch bespielsweise Myrinet sind für eine Vielzahl von Betriebssystem- und Prozessorplattformen verfügbar. Dies wird auch dadurch begünstigt, daß die erforderlichen Softwareschichten (Treiber) zur Einbindung des Kommunikationsgeräts in das System entsprechend den in Kapitel 2.3 vorgestellten Prinzipien der schlanken Kommunikation recht dünn sind und daher relativ leicht zu portieren sind. Dies trifft insbesondere auf die leistungsrelevanten Bereiche der Datenübertragung zu.

Dieses Konzept der Erweiterung der potentiellen Funktionalität des CH-Devices wird auch bei der parallel laufenden Entwicklung der nächsten Generation der ADI-Schnittstelle (ADI-3, *[76]*) berücksichtigt, indem entsprechend zum CH-Device ein MPID-*Core* verwendet wird.

In SCI-MPICH wurde die CH-Schnittstelle folgendermaßen erweitert und modifiziert:

- Alle Sende- und Empfangsfunktionen der Protokolle können optional den MPI-Datentyp der zu sendenden bzw. empfangenden Daten erhalten (siehe Kapitel 4.1.2.3 und 5.1).
- Über die `comm_init()`- und `comm_free()`-Funktionen kann ein CH-Device Kenntnis von der Einrichtung und Löschung von Kommunikatoren erhalten. Kollektive Operationen, die auf das genutzte Kommunikationsgerät optimiert sind, können im Zuge der Initialisierung

1. Teile der ADI-2-Schnittstelle werden nicht von der MPIR-Schicht, sondern nur von den obersten Schichten des MPID aufgerufen. Insofern sind sie eigentlich nicht wirklich Bestandteil der ADI-2-Schnittstelle, sind aber in deren Definition enthalten.
2. Zumindest für Kommunikationsgeräte, die nach ähnlichen Prinzipien arbeiten. Eine Softwareschicht, die etwa SCI auf einem Myrinet-System emuliert, wäre durch den hohen Softwareaufwand zur Emulation von entfernten Speicherzugriffen ausgesprochen ineffizient.

eines neuen Kommunikators über die `collops_init()`-Funktion aus dem CH-Device erbracht werden (siehe Kapitel 4.1.2.4 und 6).

- Das CH-Device kann über die Funktionen `alloc_mem()` und `free_mem()` Dienste zur Speicherverwaltung anbieten. Dies ermöglicht es der Applikation, Speicherbereiche mit spezieller Charakteristik für optimierte Leistung bei ein- und zweiseitiger Kommunikation zu verwenden (siehe Kapitel 4.1.2.5 und 7.2.2).
- Die Einrichtung und Löschung einer persistenten Kommunikation kann über die `persistent_init()`- und `persistent_free()`-Funktionen durch das CH-Device geleitet werden, so daß dieses die bezogenen Kommunikationspuffer vorbereiten kann (siehe Kapitel 7.2.2.3)
- Das Widerrufen (*Canceln)* von gestarteten Kommunikationsoperationen, wie es mittels `MPI_Cancel()` möglich ist, ist eine Funktionalität des CH-Devices, das dazu das passende Protokoll aussuchen muß und darauf basierend entscheiden muß, ob der Widerruf möglich ist. Daher wurde eine `cancel()`-Funkion in die CH-Schnittstelle aufgenommen, entgegen der ausschließlichen Implementation in den Protokollen bei MPICH.
- Die Zeitmessung kann vom CH-Device über die `wtime()`-Funktion angeboten werden. Dies ist insbesondere dann sinnvoll, wenn über das Verbindungsnetz eine global synchronisierte Zeitmessung für alle Prozesse erbracht werden kann.

4.2 Übertragungsprotokolle für SCI

Der Aufbau des CH-Device *ch_smi* im Hinblick auf die Nutzung von SCI ist in Abb. 4.2 dargestellt. Es gibt 3 verschiedene Zugriffspfade auf die SCI-Hardware, die je nach Aufgabe genutzt werden müssen. Das ch_smi-Device jedoch sieht nur zwei unterschiedliche Zugriffspfade, nämlich Aufrufe von Funktionen der SMI Bibliothek oder Speicherzugriffe auf lokalen oder entfernten Speicher. Für eine möglichst effiziente Anwendung von SCI mittels der SMI-Bibliothek ist es jedoch wichtig, die Arbeitsweisen verschiedener Operationen zu kennen, um deren Aufwand abschätzen zu können.

Kernoperationen. Bestimmte Operationen wie das Einrichten eines lokalen SCI-Speichersegments, das Verbinden mit einem entfernten SCI-Speichersegment, das Auslösen einer entfernten Unterbrechung sowie die Ausführung eines DMA-Transfers erfordern den Zugriff auf geschützte Bereiche des PSA oder Datenstrukturen im Kern und können daher nur unter Mitwirkung des SCI-Kerntreibers durchgeführt werden[1]. Dies impliziert jeweils einen erhöhten Aufwand, da zumindest ein Kontextwechsel (Systemaufruf) des Prozesses erfolgen muß und

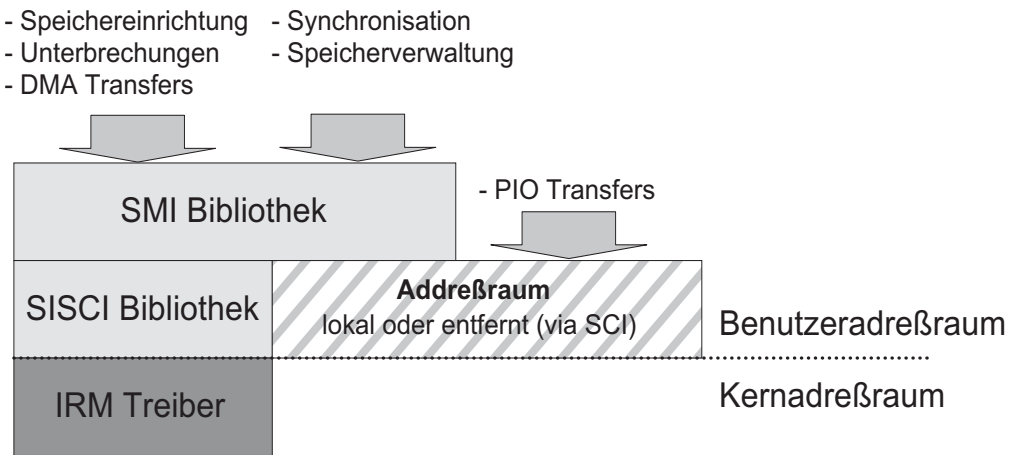


Abbildung 4.2: Aufbau des CH_SMI Channel-Device

ggf. auf den PSA zugegriffen werden muß. Die Messung der Zeitdauer dieser beiden Operationen zeigt eine untere Grenze der Zeitdauer allgemeiner Zugriffe über diesen Pfad, die unter $2\mu s$ liegt.

SMI-Dienste. Die Nutzung von anderen Diensten, die die SMI-Bibliothek anbietet, erfordert keinen Systemaufruf, sondern wird innerhalb der lokalen Instanz der SMI-Bibliothek oder ggf. durch Kommunikation über gemeinsamen Speicher mit anderen Instanzen der SMI-Bibliothek abgewickelt. Solche Dienste sind in erster Linie die Synchronisation sowie die Speicherverwaltung von Regionen gemeinsamen Speichers.

Kommunikation im gemeinsamen Speicher. Zur Übertragung von Daten zwischen lokalem Speicher und einem entfernten Speichersegment wird der transparente CPU-gesteuerte Zugriff über den SCI-Adreßraum verwendet. Dies ist mit dem geringstmöglichen Aufwand verbunden und bildet die Basis für Nachrichtenaustausch mit niedriger Latenz und hoher Bandbreite. Da hier SCI-typisch kein zusätzlicher Aufwand anfällt, erfolgen diese Zugriffe gemäß den Leistungswerten, wie sie in Kapitel 3.3.1 vorgestellt wurden.

4.2.1 Protokollstufen für Nachrichtenaustausch

Wie im vorhergehenden Abschnitt erwähnt, bietet MPICH die Möglichkeit, zum Nachrichtenaustausch bis zu drei Protokolle zu verwenden. Auch andere MPI-Implementation verfolgen das Konzept, mehrere Protokolle zu verwenden. Die MPI-Implementationen für die T3D von Cray *[83]* und ScaMPI für SCI von Scali *[81,82]* kennen ebenfalls drei verschiedene Protokolle. Hingegen verwendet etwa MPI-LAPI für die IBM RS/6000 SP-Systeme *[84]* nur zwei unterschiedliche Protokolle. Es ist jedoch keine MPI-Implementation bekannt, die weniger als zwei oder mehr als drei verschiedene Protokolle zum zweiseitigen Nachrichtenaustausch verwendet. Welche Prinzipien motivieren also zu der Realisierung des Nachrichtenaustauschs mit zwei oder drei Protokollen?

Zunächst soll dazu verdeutlicht werden, warum keine Implementierung nicht nur ein einziges Protokoll verwendet, um Daten zu übertragen. Dies ist nicht selbstverständlich, schließlich bietet es sich an, etwa eine weitverbreitete Kommunikationsschnittstelle wie *sockets [163]*, bei der mit einer einzigen Funktion Daten an einen anderen Prozeß gesendet bzw. von einem anderen Prozeß empfangen werden können, in eben dieser Weise zu verwenden. Dies wäre durchaus möglich, indem alle eingehenden Daten gelesen und die daraus extrahierten Nachrichten zunächst in Zwischenpuffer gespeichert werden. Sobald die Applikation durch den Aufruf einer Empfangsfunktion anzeigt, welche Benutzerpuffer für eingehende Daten bereitstehen, können schließlich schon vorhandene Daten aus den Zwischenpuffern in die passenden Benutzerpuffer kopiert werden. Andere Empfangsanforderungen müssen zurückgestellt werden, bis die verlangten Daten tatsächlich vom Sender geliefert werden.

Es muß daher unterschieden werden zwischen *erwarteten* und *unerwarteten* Nachrichten, die bei einem Prozeß eingehen. Für eine erwartete Nachricht liegt eine Empfangsanforderung des Empfängers vor, die den Benutzerpuffer beschreibt. Eine solche Nachricht kann wie beschrieben direkt vom Kommunikationsgerät (bzw. der entsprechenden Schnittstellenfunktion) in den Benutzerpuffer transferiert werden. Für eine unerwartete Nachricht liegt noch keine Empfangsanforderung vor, so daß sie nach dem Eintreffen in einem temporären Puffer zwischengespeichert werden muß. Sobald eine passende Empfangsanforderung vorliegt, kann sie in den darin beschriebenen Benutzerpuffer kopiert werden. Dieser zusätzliche Kopiervorgang belastet den Empfänger und kostet Leistung, da in dieser Zeit keine anderen Nachrichten empfangen werden können und Rechenzeit verbraucht wird. Darüberhinaus setzt dieses Verfahren, um für beliebig

1. Das Verfahren zur Auslösung einer entfernten Unterbrechung durch einen Schreibzugriff auf eine bestimmte entfernte Speicherstelle war zum Zeitpunkt der Erstellung dieser Arbeit noch nicht nutzbar.

große Nachrichten verwendet werden zu können, voraus, daß im Kommunikationssystem entsprechend beliebig große Puffer vorhanden sind, um den von einem möglichen Empfangsvorgang unabhängigen Sendevorgang einer Nachricht in jedem Fall abschließen zu können. In realen Systemen ist dies nicht gegeben, was dazu führt, daß bei einem solchen System der Sender ab einer gewissen Nachrichtengröße beim Schreiben der Daten blockiert. Um dies zu vermeiden, müßte für jede Nachricht ein 3-Wege-Protokoll verwendet werden, wie es in Abb. 4.3 dargestellt ist.

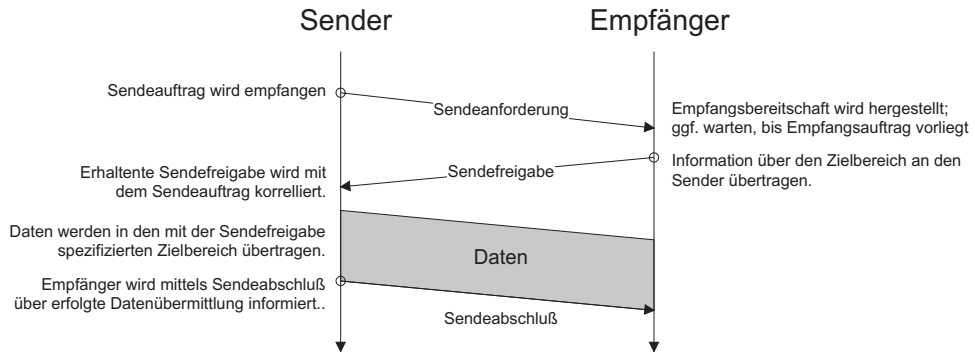


Abbildung 4.3: Prinzip des 3-Wege Protokolls zum Nachrichtenaustausch

Zunächst wird eine Nachricht vom Sender beim Empfänger angekündigt und um Anwort gebeten, sobald die Nachricht empfangen werden kann (*request*). Sobald der Empfänger nun bereit ist, weil die Applikationen eine passende Empfangsfunktion aufgerufen hat, wird diese Antwort gesendet (*confirm*). Sodann sendet der Sender die Daten (*transfer).* Mit diesem Protokoll können die beschriebenen Probleme vermieden werden. Jedoch muß der Sender bei diesem Verfahren warten, bis seine Nachricht nicht mehr unerwartet ist, und kann in dieser Zeit nichts anderes tun. Für Nachrichten bis zu einer systemabhängigen oberen Grenze würden die Ressourcen jedoch durchaus ausreichen, um die Nachricht im Kommunikationssystem zu puffern. Dies ist sinnvoll, wenn der Überhang beim Empfänger für das Zwischenspeichern der Nachricht klein ist gegenüber der potentiellen Wartezeit des Senders. Daher ist es sinnvoll, mindestens zwei verschiedene Protokolle zu unterscheiden:

- Bis zu einer systemabhängigen oberen Grenze der Nachrichtenlänge wird eine Nachricht (nach Möglichkeit) sofort verschickt, ohne Kenntnis darüber, ob sie beim Empfänger erwartet ist oder nicht. Dieses Protokoll wird als *eager* bezeichnet[1].
- Oberhalb dieser Grenze wird durch Verwendung eines 3-Wege-Protokolls sichergestellt, daß der Empfänger die Nachricht erwartet und sie beim Eintreffen der Daten einem Benutzerpuffer zuordnen kann. Dieses Protokoll ist als *rendez-vous* bekannt[2].

Der Sinn eines dritten Protokolls neben *eager* und *rendez-vous* hängt von der Charakteristik des Kommunikationssystems ab. Falls es eine feste minimale Größe für ein Datenpaket zur Kommunikation zwischen zwei Prozessen gibt, oder es aufgrund der Latenzcharakteristik für die Kommunikation mit kleinen Datenpaketen nur unwesentlich länger als die minimale Latenz L_{min} (siehe Abb. 4.4 *links*) dauert, eine feste minimale Größe D_{opt} zu verwenden (etwa eine Cachezeile oder ein Paket in einem Verbindungsnetz), läßt sich ein drittes Protokoll vereinbaren, das im Vergleich zu den beiden vorgestellten Protokollen bessere Leistungseigenschaften für kleine Nachrichtengrößen bietet.

1. *eager* = begierig: der Sender ist begierig, die Nachricht so schnell wie möglich (ohne Wissen des Empfängers) zu verschicken.
2. *rendez-vous* = ein Treffen [verabreden]: die beteiligten Parteien einigen sich auf einen Zeitpunkt der Kommunikation.

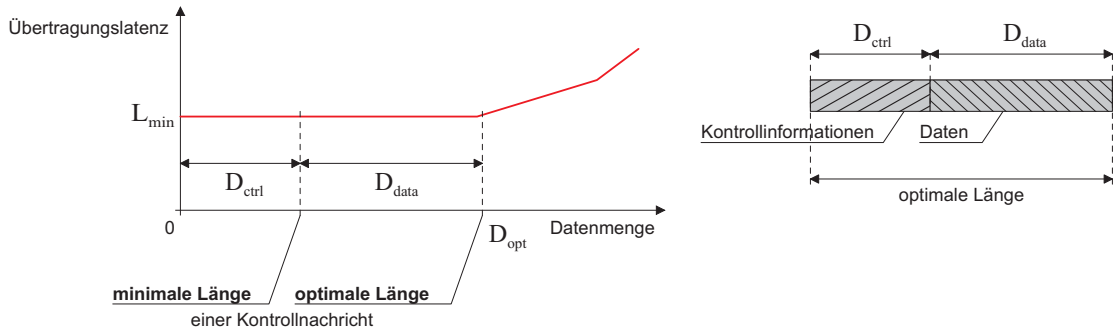


Abbildung 4.4: Typische Latenzcharakteristik für Übertragung kleiner Datenmengen *(links)*; darauf aufbauende Kapselung von Daten innerhalb einer Kontrollnachricht *(rechts)*.

Zu einer Übermittlung einer MPI-Nachricht sind bestimmte Informationen erforderlich: mindestens *Länge in Bytes*, *Tag* und *Kontext* zur Zuordnung der Nachricht beim Empfänger, sowie abhängig von dem gewählten, systemspezifischen Kommunikationsverfahren Absenderrang global oder innerhalb des Kontexts, Typkennung der Nachricht und weitere spezifische Informationen. Das dritte Protokoll basiert darauf, daß in dem beschriebenen Fall auch die Nachrichten, die zur Abwicklung etwa des *rendez-vous* Protokolls dienen (sogenannte *Kontrollnachrichten*) stets diese minimale Größe haben. Der für den Transport der beschriebenen Informationen benötigte Platz D_{ctrl} liegt häufig unter der gegebenen Optimallänge D_{opt} einer Kontrollnachricht, so daß es möglich ist, innerhalb dieser Kontrollnachricht eine bestimmte Menge Daten D_{data} zu transportieren (siehe Abb. 4.4 *rechts*). Ein solches Protokoll wird *short*-Protokoll[1] genannt.

Durch den Transport der Daten innerhalb der Kontrollnachricht fällt die Startlatenz L_{min}, die für jeden Datentransport minimal erforderlich ist, nur einmal an, während sie beim *eager*-Protokoll zweimal und beim *rendez-vous*-Protokoll dreimal anfällt. Jedoch ist es beim *short*-Protokoll erforderlich, die Kontrollinformationen und die Daten in einen gemeinsamen Speicherbereich zu kopieren. Um die Menge an Daten zu ermitteln, die unter diesen Bedingungen maximal mit dem *short*-Protokoll transportiert werden sollte, bevor das *eager*-Protokoll zum Einsatz kommt, müssen die Nachrichtenlatenzen verglichen werden. Wenn t_{copy} die Zeitdauer zum lokalen Kopieren eines Bytes ist, und t_{send} die Dauer zum Senden eines Bytes an den anderen Prozeß ist, ergibt sich

$$(4.1)\quad \underbrace{L_{min} + t_{copy} \cdot D_{data} + t_{send} \cdot (D_{ctrl} + D_{data})}_{\textit{short}\text{-Protokoll}} = \underbrace{L_{min} + t_{send} \cdot D_{ctrl} + L_{min} + t_{send} \cdot D_{data}}_{\textit{eager}\text{-Protokoll}}$$

was als Bestimmungsgleichung für D_{data} ergibt:

$$(4.2)\quad D_{data} = \frac{L_{min}}{t_{copy}}$$

Hierbei ist zu berücksichtigen, daß t_{copy} bei einem nicht-linearen Zusammenhang zwischen Datenmenge und Latenz ein von D_{data} abhängiger Parameter ist. In einem solchen Fall ist die gekoppelte Gleichung

$$(4.3)\quad D_{data} \cdot t_{copy}(D_{data}) = L_{min}$$

für D_{data} zu lösen. Dies entfällt, wenn man den Wertebereich von D_{data} so einschränkt, daß

1. *short*: dieses Protokoll ist nur für relative kurze Nachrichten geeignet.

von einem linearen Zusammenhang ausgegangen werden kann. Sobald also (4.2) für D_{data} einen Wert von mehr als der Zahl der Bytes einer Fließkommazahl liefert, ist die Verwendung eines *short*-Protokolls sinnvoll. Bei einer angenommen minimalen Kommunikationslatenz von $L_{min} = 15\mu s$ und einer Byte-Kopierlatenz von $20ns$ für kleine Blockgrößen wäre dies beispielsweise bis zu einer Nachrichtenlänge von 750 Byte der Fall.

4.2.2 Kontroll- / Short-Protokoll

Das Kontroll-Protokoll erfüllt zunächst den Zweck des Austauschs von Kontrollnachrichten, die zur Steuerung aller anderen Protokolle und der Interprozeßkommunikation genutzt werden. Jedoch ist die Paketgröße dieses Protokolls stets ausreichend, um neben den Informationen der Kontrollnachrichten auch Daten im selben Paket zu transportieren. Im Falle einer solchen Nutzung des Kontroll-Protokolls wird dieses als *short*-Protokoll bezeichnet. Das *short*-Protokoll arbeitet mit den selben Datenstrukturen und Speicherbereichen wie das Kontroll-Protokoll und hat die zusätzliche Funktionalität, neben den Kontrolldaten auch Nutzdaten in die Nachrichten zum Versand einzubetten und beim Empfang wieder zu extrahieren.

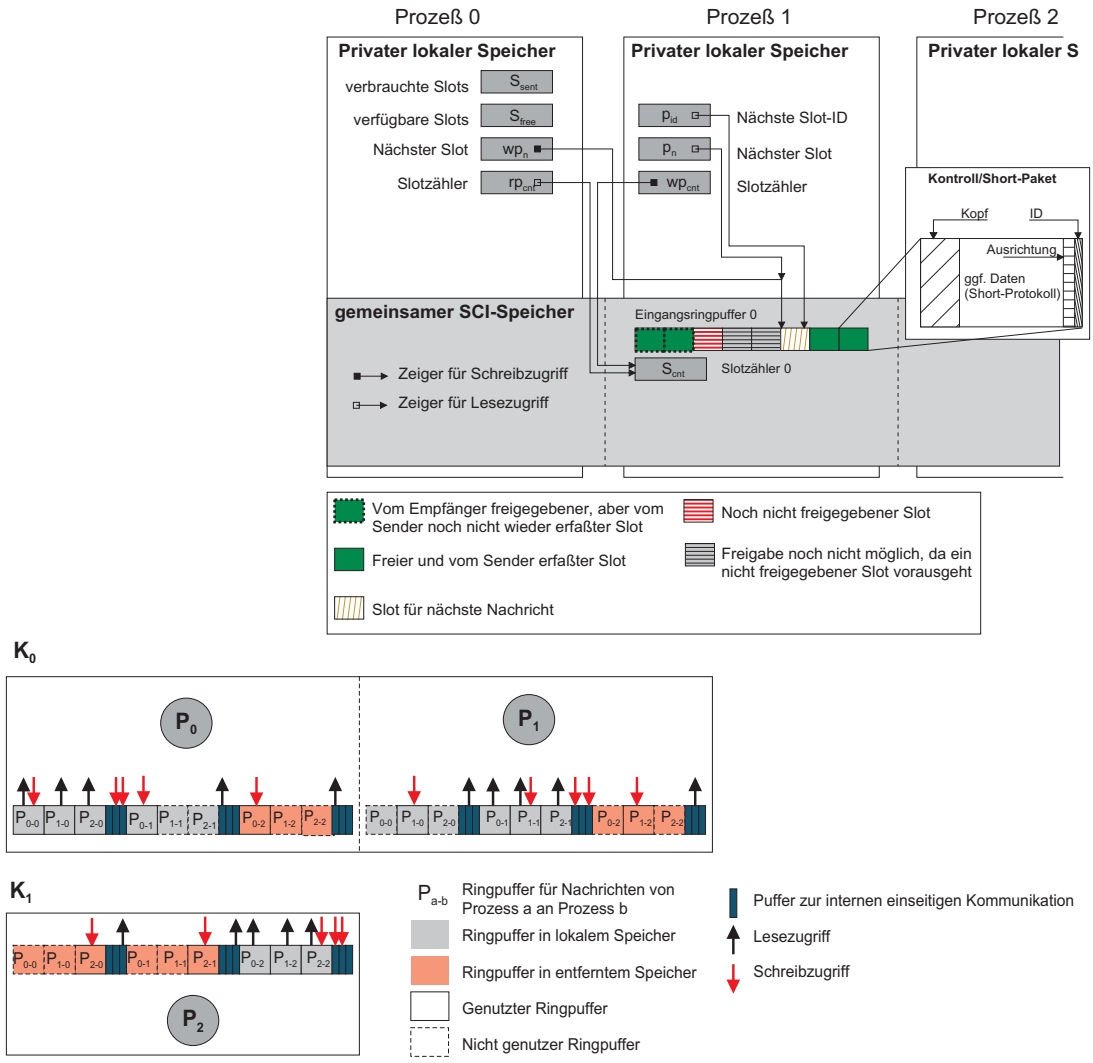


Abbildung 4.5: Datenstrukturen für das Kontroll/*short*-Protokoll zur Kommunikation von Prozeß 0 mit Prozeß 1 *(oben)* und SCI-Speichersegment-Konnektivität für 2 Knoten mit 3 Prozessen *(unten)*.

4.2.2.1 Kommunikationsprinzip und Datenstrukturen

Ziel des Short-Protokolls ist also eine minimale Latenz der Datenübertragung, um die Bandbreite sowohl bei Nutzung des Short-Protokolls für Datentransport als auch bei der Nutzung als Kontrollprotokoll zur Steuerung anderer Protokolle zu maximieren. Daher muß vollständig auf (zeitaufwendige) Synchronisation zwischen den mehreren Prozessen, die dem selben Prozeß eine Nachricht senden, verzichtet werden. Dies wurde erreicht, indem jedes Paar von Prozessen ein eigenes Paar von Ringpuffern nutzt, auf die nur der Sender schreibend zugreift. Da so jede Speicherstelle nur von maximal einem Prozeß verändert wird, kann dieser Zugriff unsynchronisiert erfolgen. In Abb. 4.5 *oben* sind alle Elemente der Datenstrukturen abgebildet, die erforderlich sind, um eine Nachricht von Prozeß 0 an Prozeß 1 zu senden. In den entsprechenden Spalten sind dabei die Speicherbereiche der Prozesse 0, 1 und 2 abgebildet, die jeweils aus privatem Speicher und exportierten SCI-Speichersegmenten (*short*-Segmente) der einzelnen Prozesse bestehen. Jeder Prozeß importiert dabei die SCI-Speichersegmente aller anderen Prozesse, so daß sich der dargestellte gemeinsame Speicherbereich ergibt. Jeder Prozeß legt in seinem lokalen SCI-Speichersegment einen Eingangs-Ringpuffer für alle anderen Prozesse (und sich selbst) an. Ein Beispiel für eine Konfiguration dieser Ringpuffer bei einer Applikation aus drei Prozessen, die auf zwei Knoten laufen, ist in Abb. 4.5 *unten* gegeben. Sie zeigt die verteilten Ringpuffer aus der Sicht der drei Prozesse und illustriert, warum diese Konfiguration selbstsynchronisierende Nachrichtenpakete erlaubt: auf jeden Ringpuffer hat stets nur ein Prozeß Schreibzugriff, während im Gegenzug nur ein Prozeß Daten aus diesem Ringpuffer lesen kann. Dies trifft auch auf die Ringpuffer zu, mit denen ein Prozeß Nachrichten an sich selbst sendet. Durch Prozeß-interne Synchronisation ist gewährleistet, daß innerhalb der Prozesse bei Verwendung mehrerer Threads der Ringpufferzugriff ein kritischer Abschnitt ist, in dem jeweils nur ein Thread aktiv ist.

Neben den Ringpuffern für Kontroll/*short*-Nachrichten liegen in den *short*-Segmenten noch Speicherbereiche, die zur internen (d.h. nicht direkt durch die Applikation veranlaßten) *einseitigen* Kommunikation dienen. Jeder Prozeß legt in diesen Bereich Datenstrukturen mit Inhalten, die zur Abwicklung der anderen Protokolle dienen. Wiederum wird für jeden anderen Prozeß ein eigener Bereich definiert. Im Gegensatz zur Kommunikation über Kontrollnachrichten, die ja auch zu interner Kommunikation eingesetzt werden, kann diese einseitige Kommunikation direkt durch SCI-Lesezugriffe erfolgen.

Nach Möglichkeit soll eine Nachricht mit nur einer SCI-Transaktion übertragen werden, um die Latenz zu minimieren. Daher wird als minimale Paketgröße (die Bruttogröße der Nachricht) die maximale verfügbare SCI-*nwrite*-Paketgröße[1] $nwrite_{max}$ gewählt. Diese Einheit wird im Rahmen dieses Protokolls als *Slot* (Größe S_{slot}) bezeichnet. Zur Erkennung einer neuen Nachricht prüft der empfangende Prozeß das letzte Wort des Speicherplatzes im Eingangspuffer ab, an den die nächste Nachricht vom Sender geschrieben werden muß. Dort prüft der Empfänger nach, ob die erwartete Kennung für die nächste Nachricht vom Sender geschrieben wurde. Die Kennung id_{msg} ist eine Ganzzahl und wird von Sender und Empfänger, die über einem Ringpuffer mit *S* Slots kommunizieren, für die *n*-te Nachricht gemäß

$$id_{msg}(n) = n \lozenge m + 1 \qquad ; k > S \tag{4.4}$$

berechnet. *m* ist eine beliebige Konstante, die der Bedingung $m > S$ genügt. Da diese Bedingung aufgrund

1. Die minimale Kommunikationslatenz für Nutzdatenmengen von mehr als 8 Byte wird bei dieser Paketgröße erreicht (siehe SCI Latenzdiagramm, Abb. 4.6). Bei LC-2 Linkcontrollern beträgt diese Größe 64 Byte, LC-3 Linkcontroller können bis zu 128 Byte in einer *nwrite*-Transaktion übertragen. Der SCI-Standard definiert *nwrite*-Transaktionen mit einer Länge bis maximal 256 Byte.

$$(4.5)\quad id_{msg}(n+i) = (n+i) \lozenge m + 1 \neq id_{msg}(n) \qquad (\text{für } 0 < i \leq S)$$

garantiert, daß der gleiche Slot niemals zweimal hintereinander die gleiche Kennung erhält, braucht die Kennung nach dem Lesen vom Empfänger nicht invalidiert zu werden, was einen Schreibzugriff erspart. Die Addition von 1 in (4.4) ist erforderlich, da der Ringpuffer zu Beginn durchgehend mit 0 initialisiert ist.

Die Verwaltung des Speichers der Ringpuffer des Empfängers erfolgt nach einem Kreditverfahren auf Basis der genutzten Slots. Die Zahl *S* der Slots in jedem Ringpuffer ist bekannt und fest. Der Sender führt für jeden Empfänger zwei Zähler (siehe Abb. 4.5 *oben*): S_{sent} ist die Zahl der für versandte Nachrichten verwendeten Slots, die dem Sender bekannt ist; S_{free} ist die Zahl der verfügbaren Slots ab dem Schreibzeiger wp_n. Zu Beginn gilt $S_{free} = S$; mit jeder verschickten Nachricht wird S_{free} entsprechend den verbrauchten Slots dekrementiert. Auf der Gegenseite zählt der Empfänger die Zahl der Slots, aus denen er Nachrichten vom Sender gelesen und verarbeitet hat, in S_{cnt}, einer im gemeinsamen Speicher angelegten Variable. Sobald S_{free} kleiner ist als die Zahl der für die nächste Nachricht benötigten Slots, liest der Sender S_{cnt} aus und berechnet gemäß

$$(4.6)\quad S_{free} = S_{sent} - S_{cnt}$$

die Zahl der effektiv freien Slots.

Die volle Konnektivität der Ringpuffer unter den N Prozessen bedeutet, daß der Speicherbedarf M_{short} für die Ringpuffer quadratisch mit der Zahl der Prozesse wächst. Jeder Prozeß exportiert ein lokales Speicherelement der Größe M_{short} für den Empfang von Nachrichten und importiert $N-1$ entfernte Speichersegmente zum Versand von Nachrichten. Jedes der Segmente hat die Größe von *S* Slots, so daß sich als Größe M_{short} ergibt:

$$\begin{aligned}(4.7)\quad M_{short} &= M_{local} + M_{remote} \\ &= N \cdot S \cdot nwrite_{max} + N(N-1) \cdot S \cdot nwrite_{max} \\ &= N^2 \cdot S \cdot nwrite_{max}\end{aligned}$$

Das Protokoll könnte auch mit nur einem Slot pro Ringpuffer ($S = 1$) genutzt werden. Dies würde aber erfordern, daß nach jeder gesendeten Kontroll- oder Shortnachricht der Empfänger diese verarbeiten und freigeben müßte, bevor dieser Sender eine weitere Nachricht an diesen Empfänger schicken kann. Dies wird in einer korrekten MPI-Applikation auch so sein, jedoch erhöht sich dadurch einerseits die durchschnittliche Latenz einer Übertragung, zum anderen führt dies zu einer sehr starken Kopplung des Fortschritts der beiden Prozesse und somit potentiell zu Wartezeiten beim Sender.

Es kann aber durchaus erforderlich sein, die Zahl der Slots bei steigender Prozesszahl zu reduzieren. Welchen Einfluß dies auf die Leistung hat, wird in Kapitel 4.3 durch Modellierung untersucht. Es ist jedoch in jedem Fall sinnvoll, die volle Konnektivität statisch zwischen allen Prozessen einzurichten. Eine dynamische Einrichtung von entfernten Speicherressourcen, wie sie für das *eager*- und *rendez-vous*-Protokoll (siehe Kapitel 4.4) wichtig ist, ist für den vergleichsweise geringen Speicherverbrauch und die hohe Wahrscheinlichkeit, daß an einen Prozeß eine Short- oder Kontrollnachricht geschickt werden muß, nicht gewinnbringend. Die nötigen Ressourcen für diese Basiskommunikation müssen in jedem Fall gegeben sein.

4.2.2.2 Datenintegrität

Das vorgestellte Verfahren zur Synchronisation hat das Problem, daß nicht grundsätzlich davon ausgegangen werden kann, daß beim Erscheinen des adreßmäßig letzten Wortes eines Pa-

kets alle vorhergegangenen Worte des Pakets ebenfalls im Eingangspuffer eingetroffen sind. Durch Übertragungsfehler, die vom SCI-Protokoll entdeckt werden, kann es vorkommen, daß Teile des SCI-Pakets in getrennten Transaktionen wiederholt übertragen werden müssen. Der im vorhergehenden Abschnitt erläuterte Protokollaufbau, der ähnlich auch in *[103]* vorgeschlagen wird, muß daher erweitert werden.

Die volle Integrität könnte nur durch eine Aufteilung der Übertragung durch den Sender in drei Schritte sichergestellt werden:

1. Übertragung des Kontroll- und Datenanteils des Pakets.
2. Durchführung einer Speicherbarriere zur Sicherstellung der vollständigen Übertragung.
3. Notifikation des Empfängers durch Schreiben des letzten Wortes (oder durch Senden einer Unterbrechung).

Diese Lösung würde die Latenz jedoch mindestens um den Faktor 3 erhöhen, da statt *einer* mindestens *zwei* SCI-Transaktionen verwendet werden müßten, und zusätzlich eine Speicherbarriere zwischen diesen beiden Transaktionen durchgeführt werden müßte, deren Zeitbedarf in der Größenordnung einer Datenübertragung liegt.

Daher wird zur Wahrung der Datenintegrität eine Prüfsumme genutzt, die im Paket übertragen wird. Ergibt der Vergleich der mitgeschickten und der berechneten Prüfsumme beim Empfänger eine Differenz, muß der Empfänger solange erneut die Prüfsumme berechnen und vergleichen, bis die Datenintegrität festgestellt werden kann. Dies wird immer nach endlicher Zeit der Fall sein, da das SCI-Protokoll die Übertragung der Daten garantiert. Als Prüfsummenverfahren kommt ein CRC-Verfahren (*Cyclic Redundancy Check*) zum Einsatz. Dieses Verfahren wird auch zur Sicherung der SCI-Pakete auf der Linkebene eingesetzt, wobei dort eine Wortlänge von 16 Bit zur Codierung verwendet wird (CRC-16). Bis zu einer Datenlänge von 256 Byte würde daher mit einer CRC-16 Codierung die gleiche Datensicherheit wie auf der Linkebene erreicht. Da Nachrichten im Short-Protokoll jedoch eine theoretisch beliebige Länge haben können (siehe Kapitel 4.2.2.3), und Längen bis zu 1024 Byte bezgl. der erreichten Latenz im Verhältnis zum Eager-Protokoll sinnvoll sein können, ist eine Verwendung größerer Wortlängen notwendig, um die gleiche Datensicherheit wie auf der Linkebene gewährleisten zu können. Es wird daher standardmäßig das CRC-32 Verfahren (Wortlänge 32 Bit) zur Codierung der Fehlersicherung verwendet.

4.2.2.3 Lange Short-Nachrichten

In Kapitel 4.2.1 wurde die bessere Effizienz eines *short*-Protokolls gegenüber einer entsprechenden Variante des *eager*-Protokolls für Kommunikationsgeräte mit einem typischen Verlauf (gemäß Abb. 4.4) von Übertragungslatenz zu übertragender Datenmenge in einem bestimmten Bereich der Größe der Daten belegt. In Kapitel 3.3.1 wurden die Leistungseigenschaften der PSA in den hier genutzten SVS vorgestellt. Die dort entnommenen relevanten Latenzmessungen in Abb. 4.6 zeigen, daß der Verlauf der PCI-SCI-Übertragungslatenzen diesem typischen Verlauf für Datenmengen, die ein Vielfaches von $nwrite_{max}$ betragen, entspricht: die Latenz nimmt im Bereich kleiner Datenmengen sublinear mit der Datenmenge zu, so daß die Übertragung von einem größeren Datenblock deutlich effizienter ist als die Übertragung der gleichen Menge Daten in mehreren kleinen Blöcken.

In ein Kontrollpaket der Minimalgröße $nwrite_{max}$ können, je nach Version des Linkcontrollers, 32 oder 92 Byte an Nutzdaten für das *short*-Protokoll eingebettet werden. Wie die Darstellung der reinen SCI-Übertragungslatenz und der Nachrichtenlatenz in Abb. 4.6 vermuten läßt, ist die durch die Minimalgröße $nwrite_{max}$ vorgegebene Nutzdatenlast für das *short*-Protokoll nicht der gemäß (4.2) bestimmbare optimale Übergangspunkt D_{data}. Zur Überprüfung dieser Vermutung können die in Kapitel 4.2.1 vorgestellten Zusammenhänge auf diesen Fall übertragen werden. Die minimale Latenz L_{min} läßt sich den Messungen entnehmen. Für die P3-Platt-

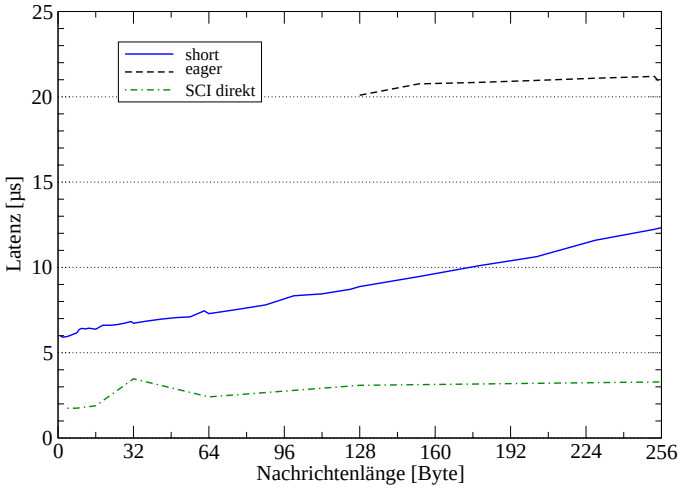


Abbildung 4.6: Latenz der Datenübertragung über PCI-SCI-Adapter für kleine Datenmengen: Latenz von *short*- und *eager*-Protokoll sowie direkten Schreiboperationen via SCI.

form gilt $L_{min} = 2\mu s$ Die Zeitdauer t_{copy} fällt für das *short*-Protokoll in dieser Form jedoch nicht an, da hier die Daten nicht mit dem Nachrichtenkopf zu einem lokalen Datenblock zusammengestellt werden, der dann versandt wird. Stattdessen werden die Nutzdaten direkt in den Eingangspuffer des Empfängers geschrieben, womit t_{copy} entfällt. Stattdessen fällt jedoch für alle Nutzdaten die Erstellung der Prüfsumme an, mit der die Integrität der Übertragung gesichert wird. Diese Zeitdauer kann als äquivalent gewertet werden. Gemäß den durch Instrumentierung gewonnenen Werten gilt $t_{copy} = 40ns/byte$. Mit diesen beiden Werten ergibt sich für die P3-Plattform die optimale Maximallänge für das *short*-Protokoll zu $D_{dataP3} = 500\text{Byte}$. Dieser Wert liegt oberhalb der Menge an Daten, die in ein Kontrollpaket eingebettet werden können.

Es wurde daher ein Verfahren entwickelt, mit dem innerhalb des Kontroll-Protokolls überlange Nachrichten verschickt werden können, die eine entsprechend höhere Nutzlast im *short*-Protokoll befördern können *[90]*. Die Brutto-Länge dieser überlangen Kontroll-Nachrichten beträgt ein Vielfaches von $nwrite_{max}$. Für den Versand einer solchen Nachricht wird die minimal notwendige Zahl von Slots aus dem Eingangspuffer des Empfängers reserviert. Gegenüber einer (einfacheren) statischen Verwaltung auf Basis der (festen) maximalen Paketgröße, die größer als ein Slot sein kann, hat die Verwaltung auf Basis der einzelnen Slots den Vorteil, daß eine Nachricht je nach Nutzdatenmenge nur die minimal notwendige Zahl von Slots verwendet. Dieses dynamische Allokationsverfahren führt, wie in *[90]* gezeigt wurde, gegenüber dem statischen Verfahren zu einer niedrigeren mittleren Latenz, da die Zahl der Leseoperationen auf entfernten Speicher zur Aktualisierung von S_{free} reduziert wird.

4.2.3 Eager-Protokoll

Zum Transport von Daten mit einer Größe oberhalb von D_{data} wird gemäß den Überlegungen in Kapitel 4.2.1 das *eager*-Protokoll verwendet. Eine Implementation dieses Protokolls ist für die Kommunikation über gemeinsamen Speicher besonders sinnvoll. Es ist hierbei möglich, daß Daten ohne Mitwirkung des empfangenden Prozesses dort hinterlegt werden, so daß er sie aus dem lokalen Speicher in den Empfangspuffer kopieren kann sobald die Empfangsbereitschaft hergestellt ist. Dies ist auch mit dem *short*-Protokoll möglich, jedoch ist dies für Nachrichtenlängen im Bereich von bis zu 100 kB nicht mehr effizient: Zum einen muß für die gesamte Nutzlast eine Prüfsumme berechnet werden, zum anderen müßten die Eingangspuffer eine Größe

haben, die bei der erwünschten statischen Konnektivität der Skalierung im Wege stünde.

Für die Kommunikation über SCI wurden zwei verschiedene Varianten des *eager*-Protokolls entwickelt, die sich in der Flußkontrolle und in der Speicherverwaltung sowie in dem Verhalten für unerwartete Nachrichten unterscheiden. Beiden Varianten gemein ist, daß jeder Prozeß einen fest dimensionierten Speicherbereich pro Prozeß für eingehende Nachrichten innerhalb eines lokalen SCI-Speichersegments allokiert. Dieses Segment wird exportiert, so daß sich jeder Prozeß den Teil des entsprechenden Segments einrichten kann, den er zum Versand von *eager*-Nachrichten an den Besitzer des Segments benötigt. Da für diese Eingangspuffer, je nach Konfiguration des Protokolls und Zahl der Prozesse in der Applikation, der entsprechende Speicherbedarf, der durch das zu exportierende SCI-Speichersegment gedeckt werden muß, nicht unerheblich ist, mußte ein Konfigurations- und Verbindungsverfahren entwickelt werden, das eventuell notwendige Abweichungen von der vorgegebenen Konfiguration erlaubt. Solche Abweichungen können erforderlich werden, wenn die vorhandenen SCI-Ressourcen eines Knotens nicht ausreichen, ein entsprechend großes SCI-Speichersegment anzulegen. In diesem Fall kann dieser Prozeß die Größe seines SCI-Speichersegments bis auf Null reduzieren. Um die solchermaßen dynamisch gewonnene Konfiguration den anderen Prozessen bekannt zu machen, schreibt jeder Prozeß die Kennwerte *Größe* und *Anzahl* seiner lokalen *eager*-Puffer in den Bereich zur internen einseitigen Kommunikation, wo sie von den anderen Prozessen ausgelesen werden können.

Die grundlegenden Datenstrukturen der beiden Protokolle sind in Abb. 4.7 dargestellt. Wie bereits bei der Darstellung des *short*-Protokolls werden jeweils die Elemente aufgeführt, die nötig sind, um von Prozeß 0 eine Nachricht an Prozeß 1 zu schicken. Wiederum hat jeder Prozeß Elemente in seinem privaten Speicher, während die zur Kommunikation erforderlichen Elemente in SCI-Speichersegmenten liegen, die von den einzelnen Prozessen exportiert werden. Im Gegensatz zum *short*-Protokoll verwendet das *eager*-Protokoll jedoch keine statische Konnektivität, da nicht notwendigerweise jeder Prozeß jedem anderen eine Nachricht über das *eager*-Protokoll sendet. Eine statische Konnektivität bindet eine große Menge von SCI-Ressourcen[1], ohne daß diese notwendigerweise im Laufe der Ausführung der Applikation zur Kommunikation benötigt werden. Daher werden die entfernten SCI-Speichersegmente, welche die Eingangspuffer enthalten (*eager*-Segmente), dynamisch ein- und ausgeblendet (siehe dazu Kapitel 4.4). Neben den Eingangspuffern müssen für das Protokoll aber auch noch Daten zur Flußkontrolle zwischen den kommunizierenden Prozessen geteilt werden. Wenn diese in den gleichen Segmenten wie die Eingangspuffer lokalisiert wären, müßte nicht nur der Sender das *eager*-Segment des Empfängers, sondern auch der Empfänger das *eager*-Segment des Senders einblenden. Um dies zu vermeiden, sind die entsprechenden Datenstrukturen in dem Segment für das *short*-Protokoll abgelegt, das statisch eingeblendet ist.

Statische Variante. Bei der *statischen Variante* des *eager*-Protokolls (siehe Abb. 4.7 *oben*) wird der Speicherbereich für den Nachrichteneingang in eine Zahl von Eingangspuffern fester Größe unterteilt. Jeder dieser Puffer kann genau eine Nachricht aufnehmen. Die Verwaltung dieser Puffer erfolgt durch einen Ringpuffer von Zeigern auf diese Puffer, der beim Sender im Speicherbereich für einseitige interne Kommunikation liegt. Der Sender liest aus diesem Ringpuffer den Zeiger auf den nächsten freien Eingangspuffer, während der Empfänger Zeiger auf Eingangspuffer empfangener *eager*-Nachrichten freigibt, indem er einen Zeiger auf diesen Puffer zurück in den Ringpuffer des Senders schreibt. Wenn der Sender einen Zeiger auf einen Ein-

1. Ab einer gewissen Zahl von Prozessen übersteigt der Ressourcenbedarf einer statischen Konnektivität grundsätzlich die verfügbaren Ressourcen und verhindert somit den Start der Applikation.Dieses Problem trifft im Prinzip auch auf das *short*-Protokoll zu, jedoch liegt hier die kritische Zahl von Prozessen aufgrund des niedrigeren Ressourcenbedarfs sehr viel höher und stellt für die betrachteten Systeme keine relevante Einschränkung dar (siehe auch Kapitel 4.4).

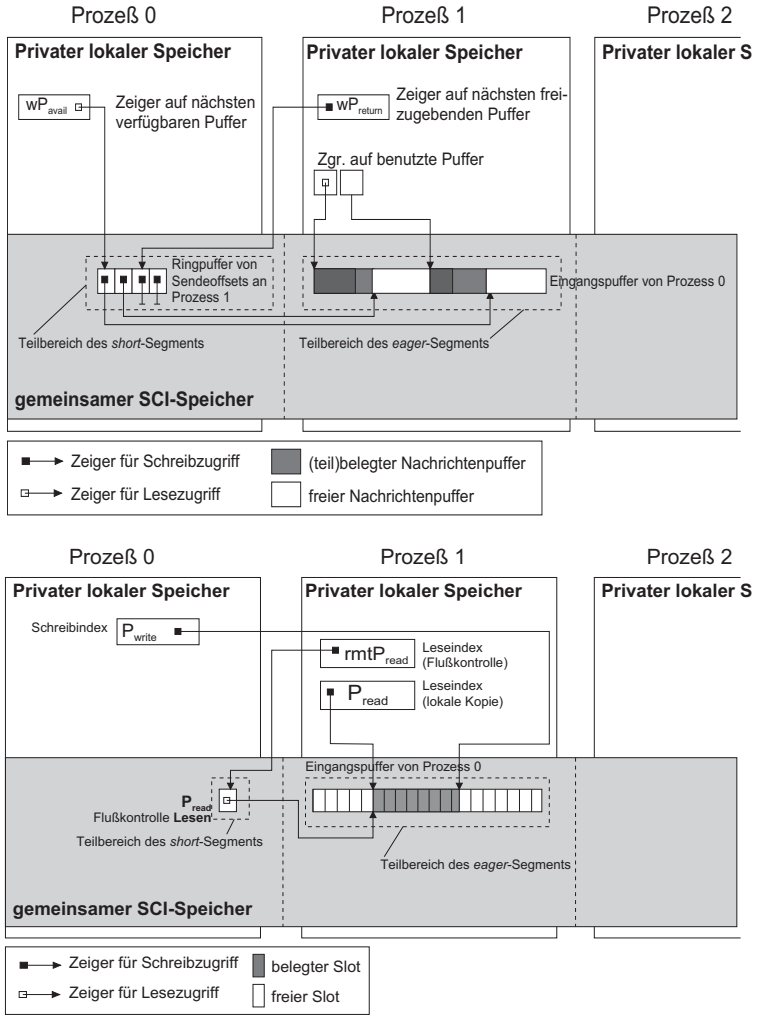


Abbildung 4.7: Datenstrukturen für das *eager*-Protokoll zur Kommunikation von Prozeß 0 mit Prozeß 1 *oben:* statische Variante, *unten:* dynamische Variante

gangspuffer gelesen hat, invalidiert er ihn im Ringpuffer und schreibt die Daten an die vom Zeiger angegebene Adresse. Anschließend wird der Empfänger durch eine Kontrollnachricht, welche die Adresse des Eingangspuffers enthält, über die eingegangene Nachricht informiert.

Der Vorteil dieser Variante ist, daß der Empfänger unerwartete Nachrichten in den Eingangspuffern belassen kann, bis die passende Empfangsanforderung verfügbar ist. Dies erspart in diesem Fall einen Kopiervorgang der Daten. Nachteilig wirkt sich hingegen die statische Dimensionierung der Eingangspuffer aus: Unabhängig von der Größe der *eager*-Nachrichten kann nur eine feste Zahl vom Sender an den gleichen Empfänger geschickt werden, ohne daß dieser die eingegangenen Nachrichten verarbeiten muß. Die Speicherausnutzung ist also nicht in jedem Fall optimal.

Dynamische Variante. Bei der *dynamischen Variante* (siehe Abb. 4.7 *unten*) ist der gesamte Speicherbereich für den Nachrichteneingang eines Prozesses als ein einziger Ringpuffer ausgelegt. Zur Leistungsoptimierung der Schreibvorgänge ist der Ringpuffer in *Slots* der Länge $nwrite_{max}$ eingeteilt, so daß ausschließlich Datenblöcke von der Größe $n \cdot nwrite_{max}$ geschrieben werden. Zur Flußkontrolle werden ein Schreib- und ein Lesezeiger verwendet, wobei der Lesezeiger im gemeinsamen Speicher des Senders liegt (wiederum im Speicherbereich für einseitige interne Kommunikation). Der Sender setzt den Schreibzeiger auf das Ende der von ihm geschriebenen Daten, während der Empfänger den Lesezeiger auf das Ende des von ihm

ausgelesenen Speicherbereichs setzt. Zur Flußkontrolle liest der Sender den Lesezeiger, um zu verhindern, daß noch nicht gelesene Daten mit neuen Nachrichten überschrieben werden. Die Flußkontrolle des Empfängers erfolgt hingegen über Kontrollnachrichten, die der Sender nach dem Schreiben der Daten einer *eager*-Nachricht dem Empfänger schickt, um diesen über die neue Nachricht zu informieren.

Gegenüber der statischen Variante ist die Speicherausnutzung (unter Vernachlässigung des Verschnitts durch die Slot-Einteilung) optimal. Jedoch ist es bei der verwendeten Flußkontrolle nicht möglich, eingegangene unerwartete Nachrichten bis zur Verfügbarkeit einer passenden Empfangsanforderung im Eingangspuffer zu belassen. Somit ist bei dieser Protokollvariante für unerwartete Nachrichten stets ein zusätzlicher Kopiervorgang in einen Zwischenpuffer erforderlich.

Der Einfluß auf die Leistung, den die unterschiedlichen Eigenschaften der Protokolle haben, ist abhängig vom Kommunikationsmuster einer Applikation. Entscheidend ist zunächst die Verfügbarkeit von Speicherplatz in den Puffern, um überhaupt eine *eager*-Nachricht schicken zu können. Falls aus Speicherplatzmangel keine *eager*-Nachricht geschickt werden kann, muß das *rendez-vous*-Protokoll verwendet werden, das im Größenbereich des *eager*-Protokolls eine deutlich höhere Latenz aufweist. Daneben ist auch der Aufwand beim ggf. erforderlichen Zwischenspeichern von unerwarteten Nachrichten zu berücksichtigen. Für eine Applikation, die viele *eager*-Nachrichten ähnlicher Größe verschickt oder in der die Prozesse zum großen Teil unsynchronisiert miteinander kommunizieren, ist die statische Variante in einer auf diese Nachrichtengröße angepaßten Konfiguration der Eingangspuffergröße günstiger. Eine Applikation mit stark variierender Größe der *eager*-Nachrichten, die zudem ein synchronisiertes Kommunikationsverhalten (etwa durch kollektive Operationen) aufweist, kann hingegen von der besseren Speicherausnutzung der dynamischen Variante profitieren.

4.2.4 Rendez-vous-Protokoll

Das *rendez-vous*-Protokoll ist durch sein 3-Wege-Verfahren (siehe Abb. 4.3) das flexibelste Protokoll. Es ermöglicht, vor der eigentlichen Datenübertragung einen Modus zu vereinbaren, der sich aus der angestrebten Maximierung der Bandbreite, der Verfügbarkeit der dynamisch zu allokierenden Ressourcen sowie speziellen Anforderungen wie etwa der asynchronen Übertragung der Daten ergibt. Zusätzlich wird dadurch, daß der Empfänger den Transfer der Daten bestätigen muß, sichergestellt, daß keine Nutzdaten übertragen werden, bevor der Empfänger die Nachricht nicht erwartet[1]. Dies wird zur Implementierung von synchronen MPI-Sendemodi verwendet. Darüberhinaus bedeutet dies, daß der Sender bei einem blockierenden (synchronen) Funktionsaufruf mit dem Empfänger synchronisiert wird, was beim *eager*- und *short*-Protokoll nicht der Fall ist.

Um eine Nachrichtenübertragung mit dem *rendez-vous*-Protokoll durchzuführen, formuliert zunächst der Sender eine Sendeanfrage, in der die Größe der Nutzdaten S_{msg} sowie Hinweise zum erwünschten Übertragungsverfahren enthalten sind. Sobald die Sendeanfrage (in Form einer Kontrollnachricht vom Typ `REQUEST_TO_SEND`) beim Empfänger eingetroffen ist und dort eine passende Empfangsanforderung vorliegt, versucht der Empfänger die notwendigen Ressourcen gemäß dem vom Sender vorgeschlagenen Übertragungsverfahren zu allokieren. Standardmäßig ist diese Ressource ein Eingangspuffer der Größe S_{in} (mit $S_{in} = min(S_{msg}, S_{rndv})$), der dynamisch aus einem exportierten SCI-Speichersegment (*rendez-vous*-Speicherpool) der Größe S_{pool} allokiert wird. S_{rndv} ist die maximale Allokationsgröße mit $S_{pool} \geq n \cdot S_{rndv}$ Dies

1. Es wäre zwar auch möglich, daß der Empfänger die Daten einer Nachricht anfordert, bevor die Empfangsanforderung vorliegt. Dies würde jedoch stets einen zusätzlichen Kopiervorgang der Daten durch den Empfänger bedeuten, da die Daten zunächst in einen Zwischenpuffer geschrieben werden müßten.

gewährleistet, daß ein Prozeß *n* Empfangsvorgänge parallel durchführen kann. Die Sendeaufforderung (Kontrollnachricht vom Typ `OK_TO_SEND`), die anschließend zum Sender zurückgeschickt wird, enthält die erforderlichen Informationen (SCI-Segment-ID des Speicher-pools sowie Größe und Lage des Eingangspuffers darin), um dem Sender Zugriff auf diesen Puffer zu ermöglichen. Sobald der Sender die Sendeaufforderung erhalten hat und der Eingangspuffer in seinem Adreßraum verfügbar ist, kann die Übertragung der Nutzdaten durchgeführt werden. Für diese Übertragung wurden zwei Verfahren mit unterschiedlicher Flußkontrolle entwickelt, die im folgenden erläutert werden.

4.2.4.1 Flußkontrolle durch Kontrollnachrichten

Für den Fall $S_{msg} \leq S_{in}$ können die Daten in einer Operation aus dem Sendepuffer in den Eingangspuffer transferiert werden. Anschließend signalisiert der Sender dem Empfänger durch eine Kontrollnachricht vom Typ `CONTINUE`, daß der Inhalt des Eingangspuffers in den Empfangspuffer übertragen werden kann. Falls jedoch $S_{msg} > S_{in}$ gilt, müssen die Daten in $\lceil S_{msg}/S_{in} \rceil$ Operationen transferiert werden, die jeweils durch Kontrollnachrichten vom Typ `CONTINUE` abgeschlossen und, falls noch Daten zu übertragen sind, vom Empfänger nach Übertragung der Daten vom Eingangs- in den Empfangspuffer durch eine Antwort vom Typ `OK_TO_SEND` gestartet werden. Mit diesem Verfahren lassen sich die Daten übertragen, jedoch ist die resultierende Transferbandbreite nicht optimal, da die beiden notwendigen Kopieroperationen durch den Sender und Empfänger vollständig sequentialisiert sind. Als resultierende Transferbandbreite B_{rndv} ergibt sich aus der Sende- und Empfangsbandbreite B_{send} und B_{recv}

$$(4.8) \quad B_{rndv} = \left[\frac{1}{B_{send}} + \frac{1}{B_{recv}} \right]^{-1}$$

Durch Überlappung der Transferoperationen des Senders und Empfängers läßt sich im günstigen Fall die Zeitdauer derjenigen Operation, die mit höherer Bandbreite arbeitet, vollständig verdecken, und die resultierende Transferbandbreite entspricht der langsameren der beiden Transferoperationen. Dieses Verfahren ist ein Pipelining der beiden Operationen, hier als *Transferpipelining* bezeichnet. Es läßt sich durch die Verwendung eines weiteren Typs von Kontrollnachrichten (`BLOCK_READY`) realisieren: Der Sender kopiert die Daten nicht mehr in einer einzigen Operation der Größe S_{in} in den Eingangspuffer, sondern in Teilschritten der Größe S_{block}. Nach jedem Block wird eine `BLOCK_READY`-Nachricht geschickt, nach deren Eingang der Empfänger den entsprechenden Block aus dem Eingangspuffer in den Empfangspuffer kopieren kann. Ein exemplarischer Ablauf dieser Variante des *rendez-vous*-Protokolls ist in Abb. 4.8 dargestellt.

Die Übertragung ist dabei nicht an einen einzelnen Partner gebunden. Durch die Flußkontrolle via Kontrollnachrichten werden die Eingangspuffer für Kontrollnachrichten ständig abgefragt, so daß ggf. auch Sendeanfragen von anderen Prozessen bearbeitet und im weiteren Verlauf die zugehörigen Daten übertragen werden können. Dies kann einerseits zu einer optimierten Auslastung der Kommunikationskanäle (und damit einer Leistungssteigerung) dienen, andererseits können durch diese Steigerung der Zahl der gleichzeitigen Kommunikationspartner jedoch auch Probleme entstehen: Einerseits steigt der Ressourcenbedarf an, zum anderen kann es bei einer Sättigung der Kommunikationskanäle zu einer Leistungsverminderung kommen. Aufgrund dieser Fähigkeit, während der Datenübertragung von bzw. zu einem Prozeß auch andere Kontrollnachrichten und die zugehörigen Vorgänge fortzuführen, wird dieses Protokoll als *nichtblockierendes rendez-vous*-Protokoll bezeichnet.

4.2.4.2 Flußkontrolle durch Ringpuffer

Die Flußkontrolle der Schreib/Lesevorgänge in den Empfangspuffer kann anstatt mittels Kontrollnachrichten auch durch Zeiger im gemeinsamen Speicher erfolgen. Die Datenübertragung erfolgt dabei, analog der dynamischen Variante des *eager*-Protokolls, in einem Ringpuffer. Al-

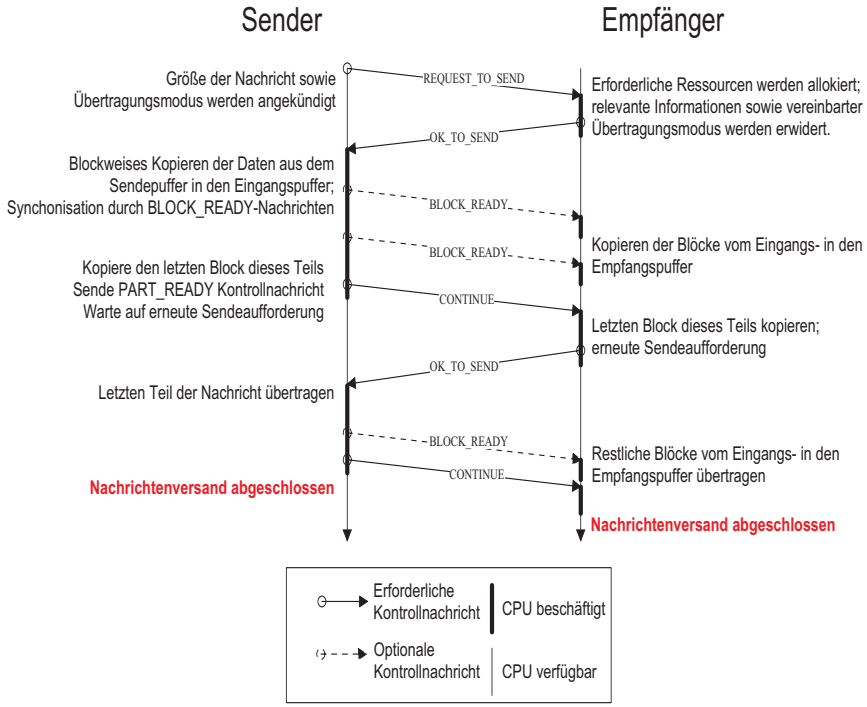


Abbildung 4.8: Exemplarischer Ablauf des *rendez-vous*-Protokolls bei Flußkontrolle durch Kontrollnachrichten und Transferpipelining

lerdings sind die Randbedingungen anders: Der Ringpuffer ist nicht statisch, sondern dynamisch allokiert. Daher können die zur Flußkontrolle erforderlichen Schreib/Lesezeiger nicht vorab statisch allokiert werden, sondern müssen im Bereich des Ringpuffers plaziert werden. Sie liegen daher beide im SCI-Speichersegment des Empfängers, so daß der Sender bei der Abfrage des Lesezeigers einen entfernten Lesezugriff durchführen muß (siehe Abb. 4.9).

Da bei diesem Verfahren zur Flußkontrolle anstelle der Zusammenstellung, des Versands und der Bearbeitung einer Kontrollnachricht nur der Zugriff auf eine entfernte Speicherstelle nötig ist, um den Stand der Schreib- und Lesepositionen zu synchronisieren, reduziert sich der Aufwand und steigt die effektive Bandbreite der Kommunikation.

Dieses in der Übertragung der Daten effizientere Protokoll hat jedoch auch Nachteile gegenüber der Flußkontrolle durch Kontrollnachrichten: Von Anfang bis Ende der Übertragung der Nachricht sind Sender und Empfänger exklusiv mit dieser Übertragung beschäftigt und können nicht auf eventuell eingehende Kontrollnachrichten eingehen, so daß keine Überlappung mehrerer nebenläufiger Kommunikationsoperationen möglich ist. Da die lokale Lesebandbreite ty-

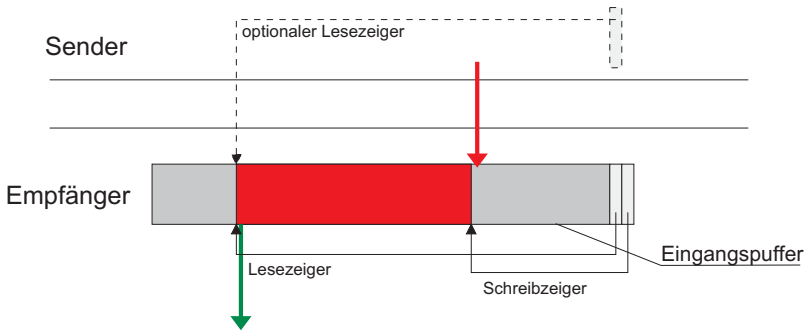


Abbildung 4.9: Ringpufferprinzip zur Flußkontrolle über gemeinsamen Speicher ohne gegenseitigen Ausschluß

pischerweise höher ist als die Schreibbandbreite in entfernten Speicher, muß der Empfänger hier also auf den Sender warten, ohne in dieser Wartezeit die Möglichkeit zu haben, Daten für einen anderen Kommunikationsvorgang zu transferieren. Daher wird dieses Protokoll als *blokkierendes rendez-vous* Protokoll bezeichnet.

Entsprechend dieser engen Kopplung bei der Datenübertragung müssen sich Sender und Empfänger zur Übertragung der Nachricht synchronisieren. Dies kann bei diesem Protokoll in der jetzigen Form zu einer Verklemmung führen: Nachdem ein Prozeß eine Sendeanforderung erhalten hat, allokiert er den lokalen Ringpuffer und antwortet auf die Anforderung mit einer Sendeaufforderung. Von diesem Zeitpunkt an findet die weitere Kommunikation für diese Nachricht nur noch über die Veränderungen der Schreib/Lesezeiger am Ende des Ringpuffers statt. Daher muß der Empfänger unmittelbar nach Abschicken der Sendeaufforderung auf den Beginn der Übertragung warten, indem er auf eine Inkrementierung des Schreibzeigers prüft. Auch wenn es weiterhin möglich ist, in Intervallen auf neue Kontrollnachrichten zu prüfen, kann es bereits bei der gleichzeitigen Kommunikation von zwei Prozessen zu einer Verklemmung führen, wie in Abb. 4.10 dargestellt ist: zwei Prozesse 0 und 1 wollen gleichzeitig jeweils dem anderen Prozeß eine Nachricht über das blockierende *rendez-vous*-Protokoll schicken und generieren dazu eine entsprechende `REQUEST_TO_SEND` Kontrollnachricht. Diese treffen gleichzeitig ein und werden von jedem Prozeß mit einer `OK_TO_SEND` Kontrollnachricht als Sendeaufforderung beantwortet. Darauf beginnen beide Prozesse, im gerade lokal reservierten Ringpuffer den Schreibzeiger vom anderen Prozeß abzufragen, um die Datenübertragung durchzuführen. Die entstehende Verklemmung kann auch nicht durch Kontrollnachrichten aufgelöst werden, da im Protokoll keine weiteren Kontrollnachrichten vorgesehen sind.

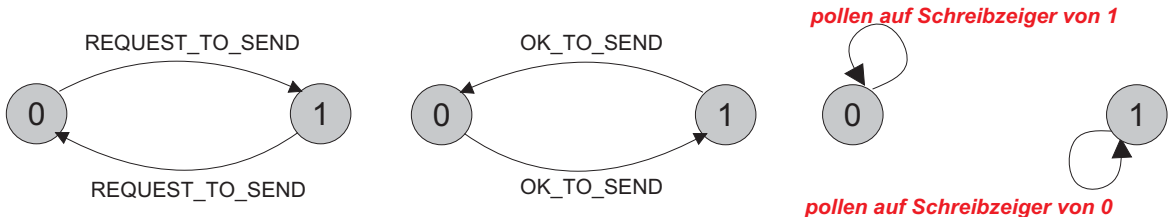


Abbildung 4.10: Bildung einer Verklemmung beim blockierenden *rendez-vous*-Protokoll durch fehlende Synchronisation der Prozesse auf den primären Transfer

Diese Verklemmungen können auch in Zyklen mit mehr als zwei Prozessen auftreten. Um sie grundsätzlich zu vermeiden, wurde ein Algorithmus entwickelt, der darauf basiert, zur Bestimmung der erforderlichen Maßnahmen nicht nur eine Unterscheidung zwischen sendendem und empfangendem Prozeß zu machen, sondern auch die Relation des lokalen Prozeßranges zum Prozeßrang des aktuellen Kommunikationspartners als Entscheidungskriterium heranzuziehen.

Ohne die Allgemeingültigkeit zu beeinflussen, wird davon ausgegangen, daß im folgenden Sendeoperationen stets blockierend ausgeführt werden und Sendeanfragen stets erwartet sind[1]. Zunächst möge ein Prozeß *i* eine Nachricht an Prozeß *j* senden wollen und generiert dazu eine `REQUEST_TO_SEND` Kontrollnachricht, um anschließend auf die Antwort zu warten. Jeder Prozeß, der eine `REQUEST_TO_SEND` Sendeanfrage empfängt und auswertet, allokiert einen Eingangspuffer und schickt eine `OK_TO_SEND` Sendeaufforderung zurück, um anschließend den Schreibzeiger des Eingangspuffers abzufragen. Jeder Prozeß, der eine `OK_TO_SEND` Sendeaufforderung empfängt und auswertet, beginnt damit, den Eingangspuffer mit Daten zu füllen.

1. Die Allgemeingültigkeit ist trotz dieser Bedingung gegeben, weil nicht-blockierende Sendeoperationen oder unerwartete Sendeanfragen nur eine größere zeitliche Trennung von Generierung bzw. Empfang der Kontrollnachricht und der dadurch bedingten Aktion (Warten auf Sendeaufforderung bzw. Generierung der Sendeaufforderung) erwirken.

Sowohl Sender als Empfänger können nun verklemmen:

- Wenn der Lesezeiger eines Eingangspuffers nicht durch den Empfänger weiterbewegt wird (weil dieser keine Daten aus diesem Puffer liest), blockiert der Sender.
- Wenn der Schreibzeiger eines Eingangspuffers nicht durch den Sender weiterbewegt wird (weil dieser keine Daten in diesen Puffer schreibt), blockiert der Empfänger.

Die Auflösung der Verklemmung kann durch das Verarbeiten neu eingegangener Kontrollnachrichten gelingen. In dem o.a. Beispiel wird etwa aus einem der beiden Empfänger durch die Verarbeitung der inzwischen eingegangenen Sendeaufforderung ein Sender, der den anderen Empfänger aus seiner Verklemmung befreit. Dazu muß jedoch sichergestellt sein, daß nicht eine neue Verklemmung entsteht, indem beide Empfänger zu Sendern werden. Daher gilt als erste Regel in diesem Algorithmus:

(4.9) Wenn ein Prozeß i beim Empfang einer Nachricht von Prozeß j verklemmt, prüft er auf neu eingegangene Nachrichten, wenn gilt: $i < j$.

Analog muß ein Sender verfahren, um eine Verklemmung aufzulösen:

(4.10) Wenn ein Prozeß i beim Senden einer Nachricht an Prozeß j verklemmt, prüft er auf neu eingegangene Nachrichten, wenn gilt: $i > j$.

Auf diese Weise werden paarweise Verklemmungen aufgelöst. Dieses Verfahren kann jedoch weiterhin verklemmen, wenn mehr als zwei Prozesse miteinander kommunizieren. In einem solchen Fall, wenn n Prozesse in einer beliebigen Zuordnung Sendeanfragen und -aufforderungen ausgetauscht haben, ist die Reihenfolge, in der die zugehörigen Kontrollnachrichten bei jedem Prozeß verarbeitet werden, völlig unbestimmt. Ein Vorgehen allein nach dem o.a. Verfahren könnte dazu führen, daß innerhalb des entsprechend dem Eingang der Kontrollnachrichten aufgebauten Stapels von Funktionsaufrufen (der Sende- bzw. Empfangsfunktionen) der einzelnen Prozesse zueinander passende Sende- und Empfangsoperationen unsynchronisiert bleiben. Da diese innerhalb der Aufruffolge bereits erfaßt wurden, würde sich nach dem Eingang der letzten Kontrollnachricht wiederum eine Verklemmung einstellen.

Zur Vermeidung dieser Verklemmung existiert in jedem Prozeß eine Protokoll-globale Datenstruktur einer Schlange, in die jeder Empfänger vor dem Prüfen auf neu eingegangene Nachrichten seinen Empfangsstatus speichert. Kann ein Empfänger nur erfolglos auf neue Nachrichten prüfen, beginnt er, Empfangsvorgänge aus dieser Schlange verarbeiten zu lassen. Durch diese Wiederaufnahme von Empfangsvorgängen aus höheren Aufrufebenen wird letztlich die potentielle globale Verklemmung aufgelöst, da es immer mindestens einen Sender gibt, der Aufgrund Regel 4.10 an einen beliebigen Empfänger sendet.

4.3 Protokollmodellierung

Die Leistung, die eine Implementation eines Protokolls auf einer bestimmten Plattform erbringt, wird von zwei Komponenten bestimmt:

- *Bandbreite bzw. Latenz der Datentransfers auf unterster Ebene (Transferanteil)*. Gemäß den Vorschriften des Protokolls müssen zur Übertragung der Daten von dem Sendepuffer in den Empfangspuffer der beteiligten Prozesse eine Reihe von Datentransfers (mindestens einer) durchgeführt werden. Diese Leistungswerte dieser Transfers werden bestimmt durch die Eigenschaften des verwendeten Kommunikationsgeräts und dessen Anbindung an den Rechenknoten.
- *Aufwand zur Verwaltung des Transfers (Verwaltungsanteil)*. Neben dem reinen Datentransfer erfordert die allgemeine Kommunikation in einem komplexen System, wie MPI es darstellt, verschiedene Verwaltungsaufgaben. Dazu gehören insbesondere die Ressourcen-

verwaltung sowie weitere Aufgaben wie die lokale Einordnung des Transfers in die Anzahl gleichzeitig zu verarbeitender Transfers oder Datenkonvertierung (in heterogenen Systemen).

Während der absolute Betrag des Transferanteils bei korrekter Nutzung des Kommunikationsgeräts (etwa durch Ausrichtung der Kommunikationspuffer an Speichergrenzen oder optimierter Kopieroperationen auf Ebene von Maschinenbefehlen) eine durch die Eigenschaften des verwendeten Kommunikationsgeräts und dessen Anbindung an den Rechenknoten definierte untere Grenze hat, ist eine solche Grenze beim Verwaltungsanteil weniger klar definiert. Da dieser Anteil auf in Software implementierten Verfahren beruht, besteht hier ein großer Gestaltungsspielraum.

Die effektive Bandbreite B_{eff} eines Protokolls für eine Nachricht mit einer Nutzlast von *D* Bytes ist die Bandbreite, die ein Applikationsprozeß wahrnimmt, und bestimmt sich als

$$(4.11)\ B_{eff}(D) = \frac{D}{t_{Transfer} + t_{Verwaltung}}$$

Dabei stellt $t_{Transfer}$ die Summe der Zeitdauern aller gemäß dem Protokoll notwendigen Datentransfers dar. Im Zusammenhang mit asynchroner Kommunikation muß beachtet werden, daß zu $t_{Transfer}$ nur die Zeiten gerechnet werden, während derer die Applikation aufgrund des Kommunikationsvorganges blockiert ist, also keine anderen Aufgaben durchführen kann. Entsprechend werden Phasen, in denen ein Datentransfer ohne CPU-Belastung außerhalb des eigentlichen Aufrufs der Kommunikationsfunktion stattfindet, nicht gerechnet[1]. Ergänzend stellt $t_{Verwaltung}$ die gesamte Zeitdauer dar, die während der Kommunikation mit der Verwaltung der Kommunikationsoperation benötigt wird. Hierbei wird davon ausgegangen, daß keine Synchronisationsverluste zwischen Sender und Empfänger auftreten, d.h. daß Anfragen eines Prozesses (etwa beim *rendez-vous*-Protokoll) von dem Partnerprozeß in der minimal möglichen Zeit beantwortet werden. Dies setzt vorraus, daß beim Empfänger eingehende Nachrichten stets *erwartete Nachrichten* sind. Diese Voraussetzung ist bei der Bestimmung der maximalen Bandbreite inhärent erfüllt, da die maximale Bandbreite nur dann erzielt werden kann, wenn beide Kommunikationspartner bereit sind, die Kommunikationsoperation durchzuführen.

4.3.1 Ebenen der Effizienz bei der Kommunikation

Zum zuverlässigen Austausch von Daten über ein Medium muß stets ein Protokoll definiert sein, mit dem die Kommunikationspartner in die Lage versetzt werden, den Zugriff auf die notwendigerweise gemeinsam genutzten Ressourcen so zu synchronisieren, daß die Daten unter Beibehaltung der Integrität aus dem vom Sender kontrollierten Speicherbereich in einen vom Empfänger kontrollierten Speicherbereich gelangen. Ein derartiges Protokoll wiederum muß implementiert werden, so daß schließlich der Datenaustausch tatsächlich erfolgen kann. Die Einführung jeder dieser Ebenen bewirkt (unvermeidbare) Leistungsverluste gegenüber der Leistung (Datenrate), die auf dem Medium bei einem Zugriff ohne jedes Protokoll erreicht werden kann. Zur Bewertung der Qualität von Protokoll und Implementierung bezüglich der Leistungsausbeute müssen diese Verluste quantifiziert werden.

4.3.1.1 Implementationseffizienz

Die Effizienz der Implementierung eines Protokolls (*Implementationseffizienz)* wird bestimmt durch die Größe des Verwaltungsanteils, da der Transferanteil von der Implementierung nicht

1. Andere Einflüsse des asynchronen Datentransfers auf die Ausführung der Applikation, wie etwa eine reduzierte Bandbreite beim Zugriff auf den Hauptspeicher durch die gemeinsame Nutzung des Datenbusses durch CPU und Kommunikationsgerät sollen hier nicht berücksichtigt werden, da sie nicht zwangsläufig relevant sind, sondern stark von der Charakteristik der Applikation abhängen.

minimiert werden kann. Damit bestimmt sich die Implementierungseffizienz aus dem Verhältnis von effektiver Bandbreite zur gemäß der Protokolldefinition maximal möglichen Bandbreite B_{prot} als

$$(4.12)\ \varepsilon_{imp}(D) = \frac{B_{eff}(D)}{B_{prot}(D)} = \frac{B_{eff}(D)}{D/t_{transfer}} = t_{Transfer} \cdot \frac{B_{eff}(D)}{D}$$

In dieser Gleichung ist die Nutzlastgröße D gegeben, und B_{eff} läßt sich für das gegebene D experimentell bestimmen. Es gibt jedoch verschiedene Definitionen von Bandbreite der Nachrichtenübertragung zwischen zwei Prozessen. In diesem Fall ist die Bandbreite über die Zeitdauer definiert, die zwischen dem Aufruf der Sendefunktion durch den Sender und dem Abschluß der Empfangsfunktion beim Empfänger verstreicht. Dies entspricht einer Ende-zu-Ende-Kommunikation zwischen den benutzerdefinierten Sende- und Empfangspuffern. Um diese Zeitdauer ohne eine globale (zwischen allen Prozessen synchronisierte) Uhr zu messen, wird ein sogenannter Ping-Pong-Zyklus verwendet, wie er in Abb. 4.11 dargestellt ist. Aus der Ping-Pong-Zykluszeit t_{pp} berechnet sich B_{eff} als

$$(4.13)\ B_{eff}(D) = 2 \cdot \frac{D}{t_{pp}}$$

Mit Hilfe der derartig bestimmten Implementationseffizienz läßt sich das Optimierungspotential des Verwaltungsanteils quantifizieren.

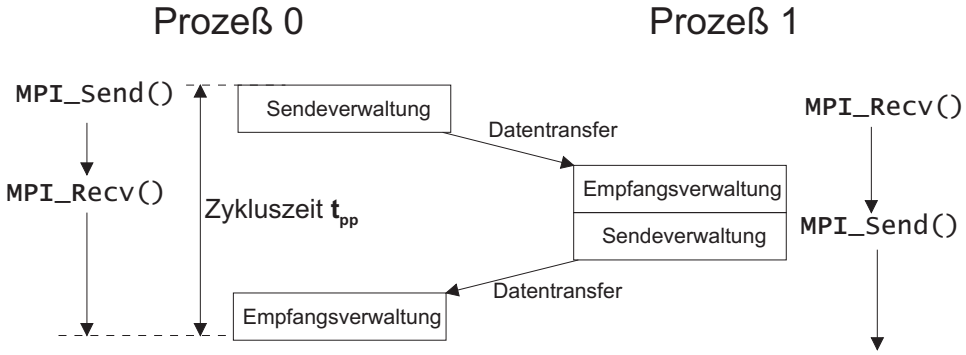


Abbildung 4.11: Definition Ping-Pong-Zyklus in MPI für blockierende Kommunikation

4.3.1.2 Protokolleffizienz

Analog zur Implementationseffizienz kann auch die Effizienz der Protokolldefinition (*Protokolleffizienz*) definiert werden. Sie bestimmt, wie gut ein Protokoll ein Kommunikationsgerät ausnutzt, und ist das Verhältnis der gemäß der Protokolldefinition und Systemeigenschaften maximal möglichen Bandbreite B_{prot} und der maximalen Bandbreite B_{max}, mit der Daten zwischen zwei Prozessen transferiert werden können, ohne daß die Übertragung im Kontext einer Applikation, Bibliothek o.ä. stattfindet (sogenannte *Peakbandbreite* des Kommunikationsgeräts). Dies ergibt

$$(4.14)\ \varepsilon_{prot}(D) = \frac{B_{prot}(D)}{B_{max}(D)}$$

Dabei muß B_{max} wiederum experimentell bestimmt werden, während B_{prot} analog zu Kapitel 4.3.1.1 bestimmt wird.

Mit Hilfe der Protokolleffizienz lassen sich die hinsichtlich der Bandbreite optimalen Grenzen zwischen den verschiedenen Protokollen bestimmen. Es läßt sich zudem ermitteln, wie gut ein Protokoll auf die speziellen Leistungscharakteristika einer Plattform abgestimmt ist, und durch Variation der Parameter bestimmen, wie stark sich einzelne Leistungsparameter auf die Proto-

kolleffizienz auswirken.

4.3.1.3 Kommunikationseffizienz

Mit den gewonnenen Werten für die Implementations- und Protokolleffizienz läßt sich schließlich auch die für den Nutzer eines Systems sichtbare *Kommunikationseffizienz* bestimmen, die die effektive Kommunikationsbandbreite der Implementation zu der Peakbandbreite des Kommunikationsgeräts ins Verhältnis setzt:

$$(4.15)\ \varepsilon_{komm} = \frac{B_{imp}}{B_{max}} = \varepsilon_{imp} \cdot \varepsilon_{prot}$$

4.3.2 Modellierung der Kommunikationsprotokolle in SCI-MPICH

Die Modellierung der Kommunikationsprotokolle, die in SCI-MPICH verwendet werden, erfüllt mehrere Zwecke. Zunächst erlaubt sie anhand des Vergleich der so ermittelten Protokolleffizienz und der gemessenen Implementationseffizienz eine Beurteilung darüber, ob die Implementation eines Protokolls effizient ist bzw. ob eine weitere Optimierung der Implementierung lohnenswert ist. Ebenso wird sichtbar, welche Auswirkung die Umgestaltung eines Protokolls hat, die beispielsweise eine weitere Latenz im Protokollablauf verursacht. Analog kann ermittelt werden, welchen Einfluß eine geänderte Leistungseigenschaft einer Plattform auf die resultierende Leistung eines beliebigen Protokolls hat. Die somit mögliche quantitative Ermittlung und Beurteilung von "Flaschenhälsen" erlaubt eine Optimierung der involvierten Komponenten.

Weiterhin ist mit der Erfassung der Protokolle in einem Modell, basierend auf einigen wesentlichen, schnell erfaßbaren Leistungsmerkmalen einer Plattform, eine Optimierung der Leistungseigenschaften *Latenz* bzw. *Bandbreite* der Protokolle für jede Plattform möglich, ohne dazu die einzelnen Protokolle für alle relevanten Parameter experimentell zu vermessen. Ein experimentelles Vorgehen ist sehr viel zeitintensiver; insbesondere nimmt die benötigte Zeit mit der verlangten Genauigkeit der Ergebnisse zu, da entsprechend mehr Testiterationen mit feiner verteilten Testpunkten der Optimierungsparameter durchgeführt werden müssen. Wenn ein Protokoll für p unabhängige Parameter mit jeweils i_p Testwerten optimiert werden soll, verlangt dies bereits

$$(4.16)\ N_{test} = \prod_{j=1}^{p} i_j$$

Testdurchläufe. Hingegen kann mit der Modell-gestützten Simulation der Protokolle im Rechner sehr viel schneller ein Ergebnis erzielt werden, das aus der optimierten Besetzung der Einflußparameter des Protokolls besteht. Voraussetzung dazu ist eine ausreichende Übereinstimmung des Modells mit dem Verhalten der Implementation auf beliebigen Plattformen (mit unterschiedlichen Kombinationen von Leistungsparametern), die im weiteren ebenfalls überprüft wird.

4.3.2.1 Modellparameter

Zur Modellierung der Protokolle sind Protokollparameter erforderlich, die für bestimmte Leistungscharakteristika des Systems stehen, auf dem das Protokoll abgewickelt wird. Durch Änderung dieser Parameter sind Aussagen über das Verhalten eines Protokolls auf beliebigen (realen oder virtuellen) Systemen möglich. Weiterhin kann durch gezielte Variation eines Parameters dessen Einfluß auf das Protokoll evaluiert werden, um so den günstigsten Ansatz für eine Optimierung des Protokolls oder der Systemplattform zu finden. In Tabelle 4.1 sind die erforderlichen Parameter zur Modellierung der vorgestellten Protokolle aufgeführt. Neben der Bezeichnung und Beschreibung der Parameter sind jeweils noch die Peakwerte dieses Parameters (d.h. maximale Bandbreite oder minimale Latenz) für die Systemplattform P3 (siehe

Kapitel 3.5) aufgeführt. Dabei ist zu beachten, daß viele der nicht-konstanten Parameter (Bandbreite) auch nicht-linear bezüglich der Datenmenge sind. Zur Berechnung eines Modells für eine bestimmte Datenmenge müssen daher experimentell ermittelte Meßkurven bzw. künstliche Verläufe verwendet werden.

Parameter	Beschreibung	Maximalwert (P3)
$B_{lr}(N)$	Bandbreite für *sequentielles Lesen* von *N* Byte aus lokalen Speicher	385,5 MB/s (aus Cache) 227,9 MB/s (aus Hauptspeicher)
$B_{cl}(N)$	Bandbreite zum *Kopieren* von *N* Byte im lokalen Speicher	3368 MB/s (aus Cache) 226,9 MB/s (aus Hauptspeicher)
$B_{cr}(N)$	Bandbreite für das *Kopieren* von *N* Byte aus lokalen Speicher in entfernten Speicher *(via PIO)*	170,2 MB/s (aus Cache) 145,6 MB/s (aus Hauptspeicher)
$B_{rr}(N)$	Bandbreite für das *Kopieren* von *N* Byte aus entferntem Speicher in lokalen Speicher *(via PIO)*	$3{,}5\,MB/s$
l_{rw}	Ende-zu-Ende-Latenz, um ein Wort in entfernten Speicher zu *schreiben*	$1{,}5\,\mu s$
l_{rs}	Lokale Latenz beim Sender, um ein Wort in entfernten Speicher zu *schreiben*	$0{,}5\,\mu s$
l_{rr}	Minimale Latenz, um ein Wort aus entferntem Speicher zu *lesen*	$3{,}2\,\mu s$
l_{lr}	Minimale Latenz einer *lokalen Leseoperation* aus dem Hauptspeicher	$31\,ns$
l_{sb}, l_{lb}	Latenz einer *store-barrier* bzw. *load-barrier* für entfernten Speicher (Sicherstellung der Speicherkonsistenz)	$3{,}5\,\mu s$

Tabelle 4.1: Modellparameter zur Bestimmung der Protokolleffizienz

4.3.2.2 short-Protokoll

Die Übertragung von Daten im *short*-Protokoll (oder einer Kontrollnachricht ohne Daten) setzt sich zusammen aus Aktionen des Senders:

- Allokation der Slots im Eingangspuffer des Empfängers (t_{alloc}).
- Berechnung der Prüfsumme über den Nachrichtenkopf und die Daten (t_{check}).
- Schreiben des Nachrichtenkopfs und der Daten in den Eingangspuffer (t_{send}).

Der Empfänger führt folgende Aktionen zum Abschluß der Nachrichtenübertragung aus:

- Erkennung der neuen Nachricht (t_{new})
- Prüfen (impliziert das Lesen) des Nachrichtenkopfs und der Daten (t_{check}).
- Schreiben der Daten in den Empfangspuffer (t_{store}).

Die Gesamtlatenz (L_{short}) ist die Summe aus diesen Zeiten, die im folgenden basierend auf den Modellparametern bestimmt werden. Zu den allgemeinen Modellparametern aus Tabelle 4.1 kommen für dieses Protokoll noch weitere hinzu, die in Tabelle 4.2 aufgeführt sind.

Parameter	Beschreibung	P3
$B_{check}(N)$	Bandbreite der Prüfsummenberechnung	87,8 MB/s
S_{header}	Größe des Nachrichtenkopfs von *short*- und Kontrollnachrichten	32 Byte
S_{short}	Maximale Bruttogröße einer *short*-Nachricht	2^n Byte; $S_{short} \geq S_{slot}$
n_{short}	Zahl der *short*-Nachrichten maximaler Größe, die ein Eingangspuffer speichern kann.	> 0

Tabelle 4.2: Besondere Parameter für das *short*-Protokoll

Slotallokation. Der Eingangspuffer des Empfängers besteht aus einer festen Anzahl an Slots:

$$(4.17)\ n_{inslots} = \frac{S_{short} \cdot n_{short}}{S_{slot}}$$

Jede Kontrollnachricht belegt genau einen Slot. Eine *short*-Nachricht belegt je nach Länge der Nutzdaten mindestens einen und maximal S_{short}/S_{slot} Slots. Diese Zahl $n_{msgslots}$ ergibt sich aus der Nutzdatenlänge N (für $N > S_{slot} - S_{header}$) wie folgt:

$$(4.18)\ n_{msgslots}(N) = 1 + \left\lceil \frac{N - (S_{slot} - S_{header}) + S_{ID}}{S_{slot}} \right\rceil$$

Jeweils nach der Allokation von $n_{inslots}$ Slots muß im Rahmen des in Kapitel 4.2.2 beschriebenen Kreditverfahrens der Zähler für freigegebene Slots im Speicher des Empfängers ausgelesen werden. Durch die variable Anzahl der Slots pro Nachricht ist es nicht möglich, eine allgemeine Relation zwischen der Zahl der versandten Nachrichten und der dafür notwendigen Zahl n_{credit} dieser Aktualisierungen herzustellen; diese ist abhängig von der Kommunikationscharakteristik der Applikation. Stattdessen kann nur das Minimum und Maximum sowie der Wert für ein bestimmtes N von *notwendigen Kreditaktualisierungen pro Nachrichtenversand* (c) bestimmt werden.

Die minimale Zahl von Kreditaktualisierungen pro Nachrichtenversand c_{min} liegt für den Fall $S_{short} = S_{slot}$ vor und beträgt $c_{min} = (n_{inslots})^{-1}$. Entsprechend beträgt der maximale Wert, der für $n_{short} = 1$ auftritt, $c_{max} = 1$. Für ein bestimmtes N ergibt sich:

$$(4.19)\ c(N) = \frac{1}{n_{msgslots}(N)}$$

Die Dauer einer Kreditaktualisierung beträgt jeweils l_{rr}, somit ergibt sich für t_{alloc}:

$$(4.20)\ t_{alloc_{min}} = c_{min} \cdot l_{rr} \qquad t_{alloc_{max}} = c_{max} \cdot l_{rr} \qquad t_{alloc}(N) = c(N) \cdot l_{rr}$$

Prüfsummenberechnung. Die Dauer der Prüfsummenberechnung ergibt sich direkt aus dem entsprechenden Protokollparameter, angewandt auf die Länge des Nachrichtenkopfs und der Nutzdaten:

$$(4.21)\ t_{check}(N) = \frac{S_{header}}{B_{check}(S_{header})} + \frac{N}{B_{check}(N)}$$

Die in (4.21) bestimmte Zeit kann auf den Sender wie auf den Empfänger angewandt werden.

Schreiben in Eingangspuffer. Im Eingangspuffer wird immer eine ganze Zahl von Slots beschrieben, so daß sich t_{send} unmittelbar aus der Zahl der Slots berechnet, die eine Nachricht belegt. Allerdings wird der erste Slot in einer separaten Operation kopiert, so daß aufgrund des nicht-linearen Bandbreitenverlaufs geschrieben werden muß:

$$(4.22)\ t_{send}(N) = \frac{S_{slot}}{B_{cr}(S_{slot})} + \frac{S_{slot} \cdot (n_{msgslots}(N) - 1)}{B_{cr}(n_{msgslots}(N) - 1)}$$

Nachrichtenerkennung. Die Nachrichtenerkennung durch den Empfänger besteht bei dem verwendeten selbstsynchronisierenden Protokoll aus dem Lesevorgang eines Wortes aus dem lokalen Speicher:

$$(4.23)\ t_{new} = l_{lr}$$

Schreiben in Empfangspuffer. Die Dauer des Kopiervorgangs der empfangenen Daten vom Eingangspuffer in den Empfangspuffer kann ebenfalls direkt aus den Protokollparametern bestimmt werden:

$$(4.24)\ t_{store}(N) = \frac{N}{B_{cl}(N)}$$

Aus (4.20) bis (4.24) ergibt sich somit für die Zeitdauer vom Start des Sendevorgangs bis zum Abschluß des Empfangsvorgangs (Ping-Pong-Latenz) einer *short*-Nachricht mit Nutzdatenlänge *N:*

$$(4.25)\ L_{short-pp}(N) = t_{alloc}(N) + 2 \cdot t_{check}(N) + t_{send}(N) + t_{new} + t_{store}(N)$$

Während eine *short*-Nachricht von variabler Länge, entsprechend der Nutzlast, sein kann, hat eine Kontrollnachricht stets eine fixe (die minimale) Länge. Daher kann für diese eine gesonderte Bezeichnung verwendet werden:

$$(4.26)\ L_{ctrl-pp} = L_{short-pp}(0)$$

Neben der Latenz für einen vollständigen Kommunikationsablauf über das *short*-Protokoll, bestehend aus Sende- und Empfangsvorgang, ist für bestimmte Fälle auch die Zeit relevant, die der Sender alleine für das Abschicken einer Nachricht benötigt. Hierbei werden also nur die lokalen Operationen des Senders berücksichtigt. Dies erfordert aber auch, daß eine lokale Operation, die Fehlerkontrolle für die Datenübertragung, hinzukommt. Diese konnte in (4.25) vernachlässigt werden, da die Daten mit sehr hoher Wahrscheinlichkeit[1] bereits *vor* dem lokalen Abschluß der Fehlerkontrolle auf der Senderseite beim Empfänger ankommen, sich also die Dauer für die Fehlerkontrolle t_{errSCI} für den Empfänger nicht bemerkbar macht, sofern kein Fehler auftrat. Da die Fehlerkontrolle aber durchgeführt werden muß, bestimmt sie die senderlokale Ausführungszeit einer Nachrichtenübermittlung im *short*-Protokoll mit. Dies ergibt als Dauer eines Sendevorgangs (Ping-Latenz)

$$(4.27)\ L_{short-p}(N) = t_{alloc}(N) + t_{check}(N) + t_{send}(N) + t_{errSCI}$$

1. Empirisch wurde eine Fehlerwahrscheinlichkeit bei *short*- und Kontrollnachrichten von unter 0,01% festgestellt.

die für eine Kontrollnachricht wiederum bestimmt ist als

$$(4.28)\quad L_{ctrl-p} = L_{short-p}(0)$$

Aus den so bestimmten Latenzen ergeben sich die Bandbreiten zu

$$(4.29)\quad B_{short-pp}(N) = \frac{N}{L_{short-pp}(N)} \text{ bzw. } B_{short-p}(N) = \frac{N}{L_{short-p}(N)}$$

4.3.2.3 *eager*-Protokoll

Die beiden Varianten des *eager*-Protokolls können identisch modelliert werden. Im Ablauf gemeinsam ist das Schreiben der Daten in den entfernten Speicher und das Verschicken der Kontrollnachricht. Auch die unterschiedlichen Verfahren der Verwaltung des Eingangspuffers; beinhalten beide die gleichen Speicherzugriffe, nämlich das Schreiben eines Wortes in den entfernten Speicher des Senders. Somit setzt sich die Latenz für den vollständigen Transfer einer *eager*-Nachricht aus der Schreiboperation in den entfernten Speicher, dem Versand einer Kontrollnachricht, der lokalen Kopieroperation des Empfängers in den Empfangspuffer und dem Freigeben des Speichers im Eingangspuffer durch Zurückschreiben eines Wortes in den Speicher des Senders:

$$(4.30)\quad L_{eager}(N) = \frac{N}{B_{cr}(N)} + L_{ctrl-pp} + \frac{N}{B_{cl}(N)} + l_{rw}$$

Daraus ergibt sich die Bandbreite des *eager-Protokolls:*

$$(4.31)\quad B_{eager}(N) = \frac{N}{L_{eager}(N)}$$

4.3.2.4 *rendez-vous*-Protokoll

Die Datenübertragung mittels des *rendez-vous*-Protokolls in der in diesem Kapitel vorgestellten Variante setzt sich zusammen aus der initialen Kontrollkommunikation (Sendeanfrage und -aufforderung) durch $n_{init} = 2$ Kontrollnachrichten sowie einer anschließenden Zahl $n_{block} \geq 1$ von Schreiboperationen für Datenblöcke des Senders, die teilweise von Leseoperationen des Empfängers zeitlich überlappt werden, sowie aus einer Zahl n_{ctrl} von Kontrollnachrichten zur Flußkontrolle der Übertragung. Diese beiden Parameter lassen sich nicht allein durch die Nachrichtengröße *N* als Parameter bestimmen. Zusätzlich werden dazu als Parameter benötigt:

- die Größe S_{in} des dynamisch allokierten Eingangspuffers,
- die gewählte Pipeline-Blockgröße S_{pipe}, mit der Schreib- und Lesevorgänge überlappt werden.

Mit diesen Parametern läßt sich zunächst die Zahl n_{part} der *vollständigen* Teilübertragungen bestimmen:

$$(4.32)\quad n_{part} = \left\lfloor \frac{N}{S_{in}} \right\rfloor$$

Eine Teilübertragung wird als vollständig bezeichnet, wenn sie S_{in} Byte überträgt. Dies ist für die letzte Teilübertragung möglicherweise nicht der Fall. Nach den vollständigen Teilübertragungen verbleibt noch eine Menge an Daten, deren Größe durch

$$(4.33)\quad S_{rest} = N - n_{part} \cdot S_{in}$$

bestimmt ist.

Zur Flußkontrolle wird jede Teilübertragung durch eine Kontrollnachricht an den Empfänger abgeschlossen. Für den Fall $N/S_{in} > 1$ fallen zusätzlich n_{part} Kontrollnachrichten an den Sender an. Im Rahmen des Pipelining werden zusätzlich vom Sender

$$(4.34)\quad n_{pipe} = n_{part} \cdot \frac{S_{in}}{S_{pipe}} + \left\lfloor \frac{N - n_{part} \cdot S_{in}}{S_{pipe}} \right\rfloor$$

Kontrollnachrichten verschickt, von denen allerdings nur die Ping-Latenz berücksichtigt wird, da der Empfänger sie nicht quittieren muß.

Bei der Zählung der Speichertransferoperationen muß ebenfalls das Pipelining berücksichtigt werden, das eine zeitliche Überlappung eines Großteils der Schreib- und Leseoperationen durch den Sender und Empfänger bewirkt. Bei jeder Teilübertragung erfolgt das Füllen und Entleeren der Pipeline nicht-überlappend durch eine Schreiboperation des Senders bzw. durch eine Leseoperation des Empfängers. Alle anderen Schreib- und Leseoperationen überlappen sich, so daß jeweils nur die längerdauernde dieser beiden Operationen berücksichtigt werden muß. Es wird davon ausgegangen, daß dies die Schreiboperation ist, und zwar in einem Maße, daß diese Überlappung auch den Empfang der zugehörigen Kontrollnachricht überlappt. Dieser Ablauf ist in Abb. 4.12 für eine Teilübertragung bei einem Verhältnis $S_{in}/S_{pipe} = 5$ dargestellt.

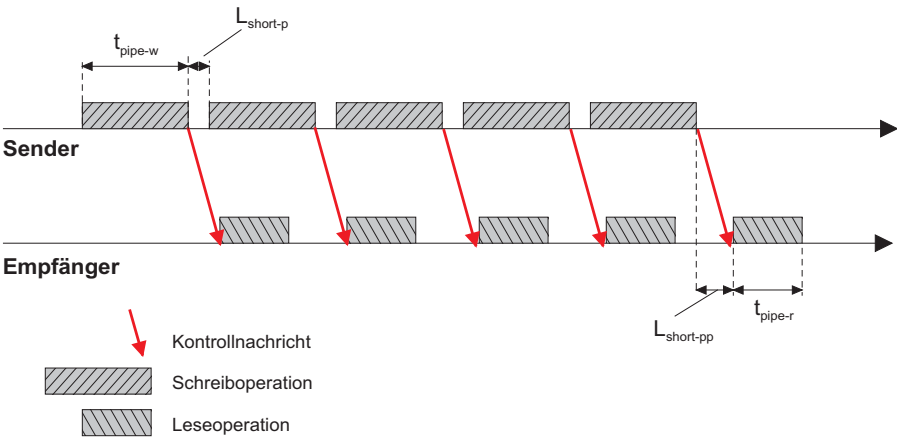


Abbildung 4.12: Pipelining von Schreib- und Leseoperationen beim *rendez-vous*-Protokoll

Gemäß diesen Überlegungen ergibt sich für die Zeitdauer einer vollständigen Teilübertragung

$$(4.35)\quad L_{part} = t_{pipe-w} + \left(\frac{S_{in}}{S_{pipe}} - 1\right) \cdot (L_{ctrl-p} + t_{pipe-w}) + L_{ctrl-pp} + t_{pipe-r}$$

Die Zeitdauern für die Schreib- und Leseoperationen ergeben sich als Verhältnis von Blockgröße zu zugehöriger Bandbreite.

Die Latenz L_{rest} einer potentiellen unvollständigen Teilübertragung muß als Fallunterscheidung abhängig von S_{rest} berechnet werden:

$$(4.36)\quad L_{rest} = 0 \qquad \text{für } S_{rest} = 0$$

$$L_{rest} = \frac{S_{rest}}{S_{pipe}} \cdot t_{pipe-w} + L_{short-pp} + \frac{S_{rest}}{S_{pipe}} \cdot t_{pipe-r} \qquad \text{für } S_{rest} < S_{pipe}$$

$$L_{rest} = t_{pipe-w} + \left(\left\lfloor \frac{S_{rest}}{S_{pipe}} \right\rfloor - 1\right) \cdot (L_{ctrl-p} + t_{pipe-w}) + L_{ctrl-pp} + t_{pipe-r} \qquad \text{sonst}$$

Für die Latenz L_{r-v} der vollständigen Übertragung einer Nachricht im *rendez-vous*-Protokoll ergibt sich in Abhängigkeit von N, S_{in} und S_{pipe} mit (4.32) bis (4.36)

$$(4.37)\ L_{r-v}(N, S_{in}, S_{pipe}) = 2 \cdot L_{ctrl-pp} + n_{part} \cdot L_{part} + L_{rest}$$

Die Bandbreite ist entsprechend

$$(4.38)\ B_{r-v}(N) = \frac{N}{L_{r-v}(N)}$$

4.3.3 Quantifizierung der Effizienz von Protokoll und Implementierung

Gemäß der Modellierung wurde ein Protokollsimulator entwickelt, mit dem anhand der gemessenen Leistungsparameter die theoretische Leistung der Protokolle auf der verwendeten P3-Plattform bestimmt wurde.

4.3.3.1 short-Protokoll

Die Ergebnisse der Simulation sowie Messungen des *short*-Protokolls sind in Abb. 4.13 zum einen als absolute Latenzwerte für die Übermittlung von Nachrichten mit Längen zwischen 0 und 2048 Byte, zum anderen ausgedrückt als Protokoll-, Implementations und Kommunikationseffizienz gemäß Kapitel 4.3.1 dargestellt.

Bei den gemessenen Latenzen fallen starke Unregelmäßigkeiten im Bereich der Nachrichtenlängen von 280 bis 750 Byte auf, die die doppelte Latenz aufweisen, als zu erwarten wäre. Die Ursache für dieses Verhalten konnte nicht geklärt werden und hängt nicht mit der Protokollim-

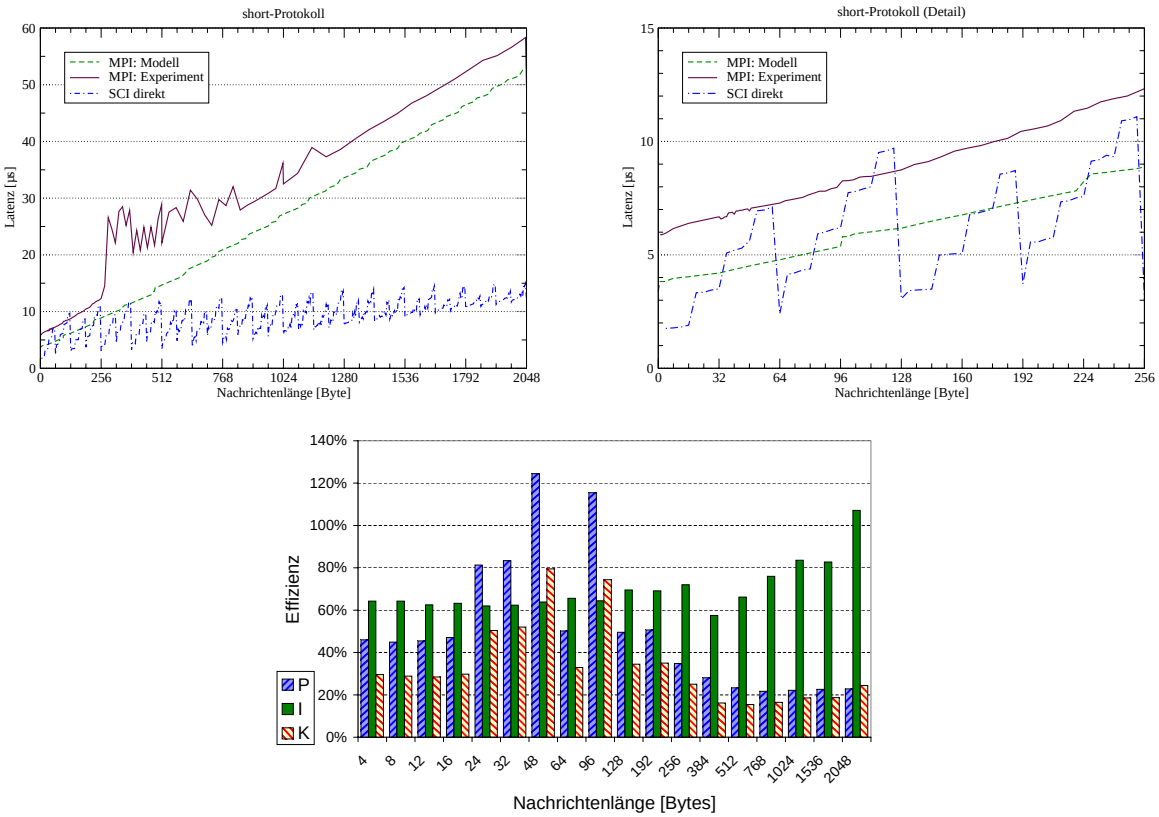


Abbildung 4.13: Simulation des *short*-Protokolls
Oben: Latenz von SCI-Übertragung, Protokollsimulation und -implementation (Übersicht (*links*) und Detail (*rechts*))
Unten: Protokoll-, Implementations und Kommunikationseffizienz (P, I und K)

plementation zusammen, da dieses Verhalten auf anderen Plattformen nicht zu beobachten ist. Da zudem nur bestimmte feste Nachrichtenlängen diese Abweichungen aufweisen, liegt die Ursache vermutlich im Bereich der Datenübertragung auf dem PCI-Bus der P3-Plattform.

Protokolleffizienz. Die Protokolleffizienz liegt im Mittel bei etwa 50%, fällt ab 256 Byte Nachrichtenlänge auf knapp 25% ab. Die relativ niedrige mittlere Effizienz macht deutlich, daß ein erheblicher Überhang im Protokoll vorhanden ist. Ein Großteil davon liegt in der Prüfsummenberechnung. Deren im Vergleich zur zunehmenden SCI-Übertragungsbandbreite geringe Bandbreite führt schließlich ab 256 Byte zu einem weiteren Rückgang der Effizienz.
Die Werte oberhalb von 100% sind in den stark gestuften SCI-Übertragungslatenzen begründet, die nicht in dieser Auflösung in den Modellrechnungen berücksichtigt wurden. Stattdessen nutzt die Implementation die Granularität der Kopieroperationen, um einen gleichmäßigen und teilweise niedrigeren Latenzverlauf zu erzeugen als er bei der reinen SCI-Übertragung zu beobachten ist.

Implementationseffizienz. Die Implementationseffizienz liegt konstant bei über 60% und steigt für lange Nachrichten an bis an 100%. Die Verwendung von mehr als einem SCI-Paket für Nachrichtenlängen senkt die Effizienz nicht. Der konstante Überhang der Implementation, der bei den niedrigen absoluten Latenzen die Effizienz vergleichsweise stark senkt, hat für größere Nachrichten einen abnehmenden Einfluß auf die Effizienz. Die Implementation entspricht daher den Erwartungen, die die Protokolldefinition gegeben hat.

Kommunikationseffizienz. Bei den sehr kurzen Latenzen für die reine Übertragung von einzelnen SCI-Paketen entspricht die resultierende Kommunikationseffizienz von über 25% den Erwartungen. Schnellere Prozessoren und höhere Speicherbandbreiten werden die Prüfsummenberechnung beschleunigen, was sich positiv auf diese Effizienz auswirken wird.

4.3.3.2 eager-Protokoll

Die Ergebnisse der Simulation und Messung für das *eager*-Protokoll sind in Abb. 4.14 dargestellt. Bei der Effizienzanalyse wurde neben den 2-exponentiellen Werten auch die Nachrichtenlänge’500’ berücksichtigt, um das Verhalten bei nicht ausgerichteten Längen zu überprüfen.

Protokolleffizienz. Die Protokolleffizienz liegt für Nachrichtenlängen bis 512 Byte bei 20%, steigt im weiteren Verlauf auf bis 90% an, um anschließend wieder abzufallen. Dieses Verhalten liegt in dem relativ hohen Aufwand begründet, den (mindestens) drei SCI-Transaktionen und eine Speicherbarriere für kleine Nachrichten darstellen. Dieser Aufwand wird im weiteren Verlauf mit zunehmender Zeitdauer für die Kopieroperationen kompensiert. Im Gegenzug verringern die beiden sequentiellen Kopieroperationen der gesamten Nachricht zunächst beim Sen-

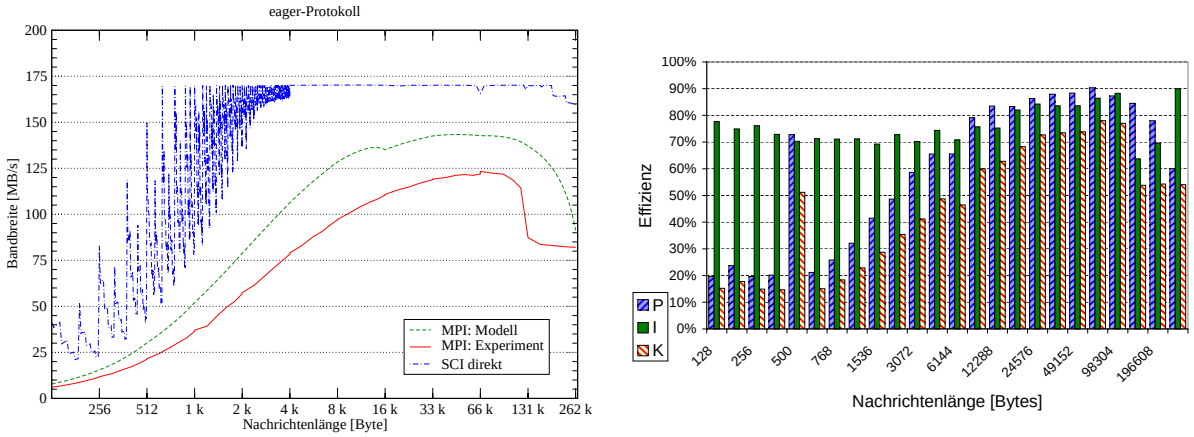


Abbildung 4.14: Simulation des *eager*-Protokolls
Links: Latenz von SCI-Übertragung, Protokollsimulation und -implementation
Rechts: Protokoll-, Implementations- und Kommunikationseffizienz (P, I und K)

der und anschließend beim Empfänger die effektive Bandbreite wieder, sobald die lokale Kopierbandbreite mit steigender Nachrichtengröße zurückgeht. Dieser Effekt setzt ab einer Nachrichtengröße von 128 kB, die der halben Größe des Level-2 Caches entspricht, besonders deutlich ein.

Implementationseffizienz. Die Implementationseffizienz liegt durchgehend über 70% und steigt maximal auf 90%. Einzig bei der Nachrichtengröße 128 kB sinkt die Effizienz deutlich unter 70% ab. Dieser Effekt liegt in dem Einbruch der Bandbreite, der früher auftritt als im Modell vorhergesagt. Die Ursache dafür ist, daß im Modell der Cache allein den Nachrichtendaten zur Verfügung steht, im Experiment jedoch auch der Code und andere Programmdaten in den Cache geladen werden. Daher tritt der *Trashing-Effekt* (gegenseitiges Verdrängen) bei den lokalen Kopieroperationen im Experiment früher auf als im Modell. Sobald dieser Effekt auch im Modell wirksam wird, steigt die Implementationseffizienz, bedingt durch die niedrigere Kopierbandbreite, wieder bis über 90% an. Die Implementation des *eager*-Protokolls ist daher sehr effizient.

Kommunikationseffizienz. Die resultierende Kommunikationseffizienz beträgt für Nachrichtenlängen von 8 kB bis zu 256 kB mehr als 50%; das Maximum liegt mit 78% bei 64 kB Nachrichtenlänge. Eine höhere lokale Speicherbandbreite wird diese Effizienz für größere Nachrichten steigern; für kleine Nachrichten unter 8 kB Länge wäre dieser Effekt geringer, da dort die Konsistenzoperationen für die Speichertransfers in entfernten Speicher den maßgeblichen Zeitanteil haben. Bei unausgerichteten Nachrichtenlängen ist jedoch auch bei kurzen Nachrichten die Effizienz deutlich höher, da das *eager*-Protokoll immer Blöcke ausgerichteter Länge kopiert.

4.3.3.3 rendez-vous-Protokoll

Für das *rndv*-Protokoll wurden die Simulation und Messung für beide Protokollvarianten, blockierend und nicht-blockierend, durchgeführt. Die Ergebnisse sind zusammen mit den Effizienzberechnungen in Abb. 4.15 dargestellt. Wiederum wurde ebenfalls die Nachrichtenlänge’500’ berücksichtigt.

Protokolleffizienz. Im Gegensatz zum *eager*-Protokoll steigt die Protokolleffizienz (für ausgerichtete Nachrichtenlängen) monoton an, auch für Nachrichtenlängen jenseits der Cachegrößen. Von unter 10% Effizienz für kleine Nachrichten steigt die Effizienz bis auf 100% (nicht-blokkierend) und darüber hinaus auf 110% (blockierend). Dieser Effekt ist durch die Nutzung des Pipelining erklärbar: die Kopieroperationen sind auf die Blockgröße der Pipeline beschränkt, die in diesem Fall deutlich kleiner als die Cachegröße ist. Dadurch nutzt das Protokoll höhere Kopierbandbreiten, als sie für die jeweilige Nachrichtengröße bei direktem Kopiervorgang (wie beim *eager*-Protokoll) entstehen. Die blockierende Protokollvariante ist dabei durch die Flußkontrolle mittels Zeiger im entfernten Speicher effizienter als die nicht-blockierende Variante. Bei dieser muß der Sender nach dem kompletten Auffüllen des Eingangspuffers auf die Lesebestätigung des Empfängers warten; diese Wartezeit fällt bei der blockierenden Variante mit den kontinuierlichen leichtgewichtigen Kontrollnachrichten nicht an.

Implementationseffizienz. Die Implementationseffizienz liegt überwiegend zwischen 80 und 100%, wobei die leicht schwankenden Werte für große Nachrichten immer weiter ansteigen. Für kleinere Nachrichten liegt die Effizienz der nicht-blockierenden Variante höher.

Kommunikationseffizienz. Bemerkenswert ist der Anstieg der Kommunikationseffizienz auf über 100% beim blockierenden *rendez-vous*-Protokoll. Dabei muß jedoch berücksichtigt werden, daß die blockierende Variante des Protokolls nur dann effektiv eingesetzt werden kann, wenn zu einem Zeitpunkt jeweils genau zwei Prozesse darüber kommunizieren. Ansonsten kommt es zu Leistungsrückgang durch die Verklemmungsauflösung und Synchronisationsverzögerungen bei den dadurch nicht kommunizierenden Prozessen. Mehrere überlappende Kommunikationsvorgänge wickelt das nicht-blockierende Protokoll durch das den *split transactions*

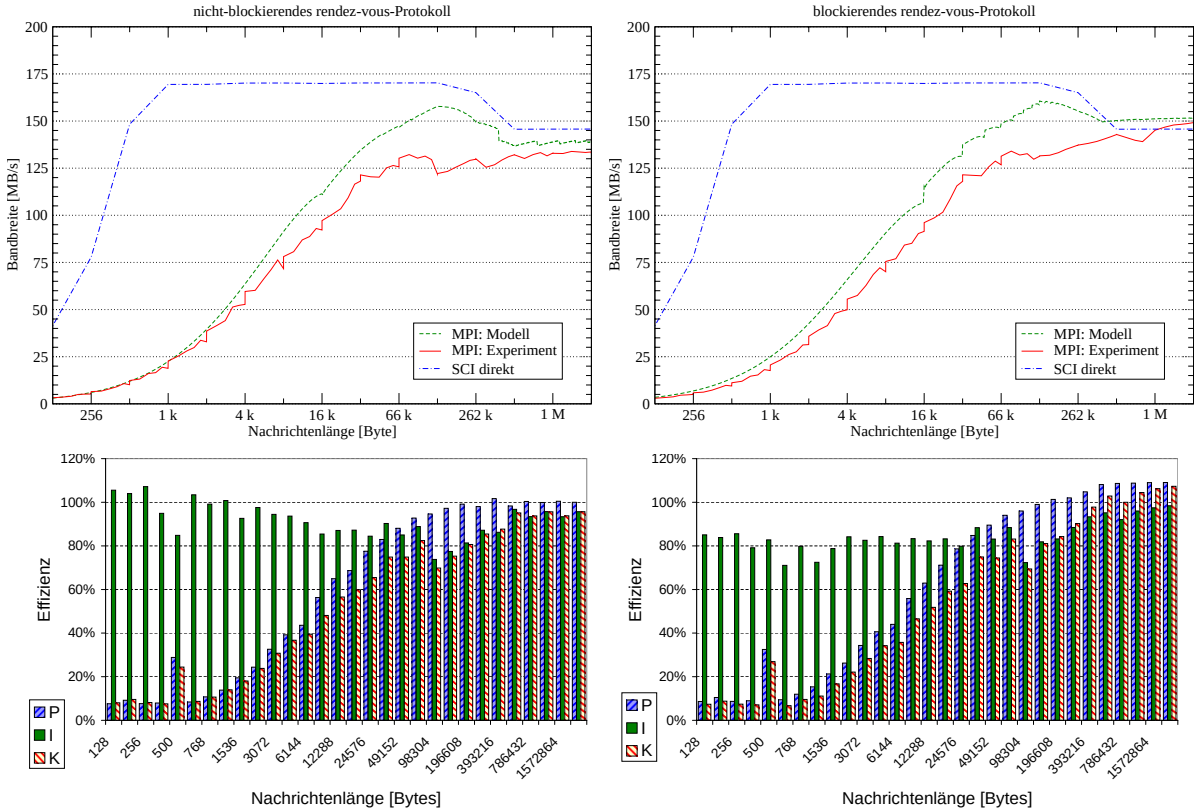


Abbildung 4.15: Simulation des nicht-blockierenden (*links*) und blockierenden (*rechts) rendez-vous*-Protokolls
Oben: Latenz von SCI-Übertragung, Protokollsimulation und -implementation
Unten: Protokoll-, Implementations und Kommunikationseffizienz (P, I und K)

ähnliche Prinzip deutlich effektiver ab. Daher ist das blockierende Protokoll standardmäßig nur für sogenannte *ready-sends* vorgesehen, bei denen laut MPI-Standard der Empfänger bereits auf den Eingang der Nachricht warten muß.

Die vorgestellten Basis-Kommunikationsprotokolle in SCI-MPICH weisen eine hohe Effizienz auf, die insbesondere in der Implementation den Erwartungen durch die Protokolldefinition gerecht wird. Dies ist insbesondere im Hinblick auf die verwendete dynamische Ressourcenverwaltung (siehe nächstes Kapitel) bemerkenswert, da diese den Aufwand zum Start einer Kommunikation erhöht und in der Protokollmodellierung nicht berücksichtigt wurde.

4.4 Ressourcenverwaltung

Im Laufe der Ausführung einer SCI-MPICH-Applikation werden verschiedene SCI-Ressourcen potentiell zur Kommunikation verwendet:

- *lokale SCI-Speichersegmente* (LS): Speicher, auf den entfernte Prozesse transparent zugreifen sollen, muß lokal als SCI-Speichersegment allokiert werden.
- *eingeblendete entfernte Speichersegmente* (EES)*:* Das eager-Protokoll, das ausschließlich mit PIO-Transfers arbeitet, sowie das rendez-vous-Protokoll im PIO Modus müssen die jeweiligen Zielpuffer in den lokalen Adreßraum einblenden, um Daten übertragen zu können. Das Einblenden von entfernten Speichersegmenten ist eine zeitaufwendige Operation, die neben einem festen Aufwand auch von der Größe des entfernten Segments abhängt sowie elementare Ressourcen belegt.
- *entfernte verbundene Speichersegmente* (EVS): Für DMA-Transfers muß ein entferntes Speichersegment nicht in den Adreßraum eingeblendet werden, da der Transfer nicht von der

CPU, sondern vom DMA-Baustein des PSA durchgeführt wird. Neben dem verringerten Aufwand beim Einrichten einer solchen Verbindung ist auch der Ressourcenverbrauch geringer.

- *lokale registrierte Speicherbereiche* (LRS): Um benutzer-allokierte Speicherbereiche als Quelle oder Ziel von DMA-Transfers verwenden zu können, müssen die virtuellen Speicherseiten dieses Bereichs als *nicht auslagerbar* gekennzeichnet werden und so an ihren aktuellen physikalischen Adressen fixiert werden (siehe Kapitel 7.2). Diese sogenannte *Registrierung* benötigt Zeit, die der Segmentgröße proportional ist.

Die von diesen Kommunikationsressourcen benötigten internen Ressourcen der PCI-SCI-Architektur wurden bereits in Tabelle 3.4 aufgeführt. Die begrenzten Systemressourcen werden ab einer gewissen Zahl von Prozessen zu einem Problem, das die Ausführung eines Kommunikationsvorgangs blockieren kann. Für die Ausführung einer MPI-Applikation ist es jedoch erforderlich, daß eine angestoßene Sende- oder Empfangsoperation tatsächlich mit endlicher Verzögerung ausgeführt wird, wenn zu einer Sendeoperation eine passende Empfangsoperation angestoßen wurde. Für den Fall, daß dies nicht möglich ist, muß die Applikation abgebrochen werden, da es sonst zu einer Verklemmung käme.

Eine stabile und skalierbare MPI-Implementation darf ein solches Verhalten auch bei begrenzten Ressourcen nicht zeigen. Die dazu erforderliche Flexibilität der Ressourcennutzung soll keinen leistungsmindernden Einfluß haben. Mit dieser Zielvorgabe wurden für SCI-MPICH Verfahren entwickelt, implementiert und integriert, die die vorhandenen Ressourcen effizient und unter Beibehaltung hoher Leistung nutzen.

4.4.1 Quantifizierung des Ressourcenverbrauchs

Eine Blockade durch Ressourcenmangel tritt ein, wenn eine Nachricht an einen Prozeß nicht verschickt werden kann, weil beispielsweise kein SCI-Deskriptor mehr frei ist. Bei maximal 256 verfügbaren Deskriptoren ist dies ein reales Szenario. Zur exakten Quantifizierung des Ressourcenverbrauchs werden folgende Größen verwendet:

N	Zahl der Prozesse, die die parallele Applikation bilden.
K	Zahl der Knoten, auf denen Prozesse der Applikation ablaufen.
k	Zahl der Prozesse pro Knoten (*Prozeßdichte*), entsprechend N/K.

Der Aufbau der SCI-Basiskommunikation ist für den Fall $N = 5$ und $K = 3$ in Abb. 4.16 dargestellt. Sie zeigt den SCI-bezogenen Verbindungs- und Segmenteinblendungszustand, der statisch durch die Initialisierung der SCI-MPICH-Bibliothek (die die Initialisierung SMI-Bibliothek impliziert) erreicht werden muß, damit die Prozesse auf MPI-Basis kommunizieren können. Segmente sind in der Grafik durch Kästen, SCI-Deskriptoren durch Pfeile dargestellt. Dabei stellen einfache Pfeile Verbindungen dar, wohingegen Doppelpfeile anzeigen, daß zusätzlich noch eine Speichereinblendung über diese Deskriptor erfolgt.

Damit läßt sich der Verbrauch der SCI-Ressourcen für eine Applikation aus N Prozessen, die auf K Knoten ausgeführt werden, ermitteln. Die Prozesse können beliebig auf die Knoten verteilt sein. In der Praxis wird aber stets eine gleichmäßige Verteilung verwendet, da pro CPU maximal ein Prozeß auf einem Knoten aktiv sein sollte. Aufgrund der homogenen Charakteristik der Rechenknoten in Clustern der hier betrachteten Art kann von einer gleichen Anzahl von CPUs pro Knoten ausgegangen werden. Es ist daher keine unzulässige Einschränkung, wenn vorausgesetzt wird, das K echter Teiler von N ist und jeder Knoten die gleiche Prozeßdichte k aufweist. Mit dieser Randbedingung stellt sich der Ressourcenverbrauch für eine SCI-MPICH-Applikation gemäß Tabelle 4.3 dar.

Zur Berechnung des globalen Ressourcenverbrauchs einer Applikation muß sowohl die *Allokation* der Ressourcen auf einem Knoten als auch die Nutzung (*Konnektierung*) entfernter Res-

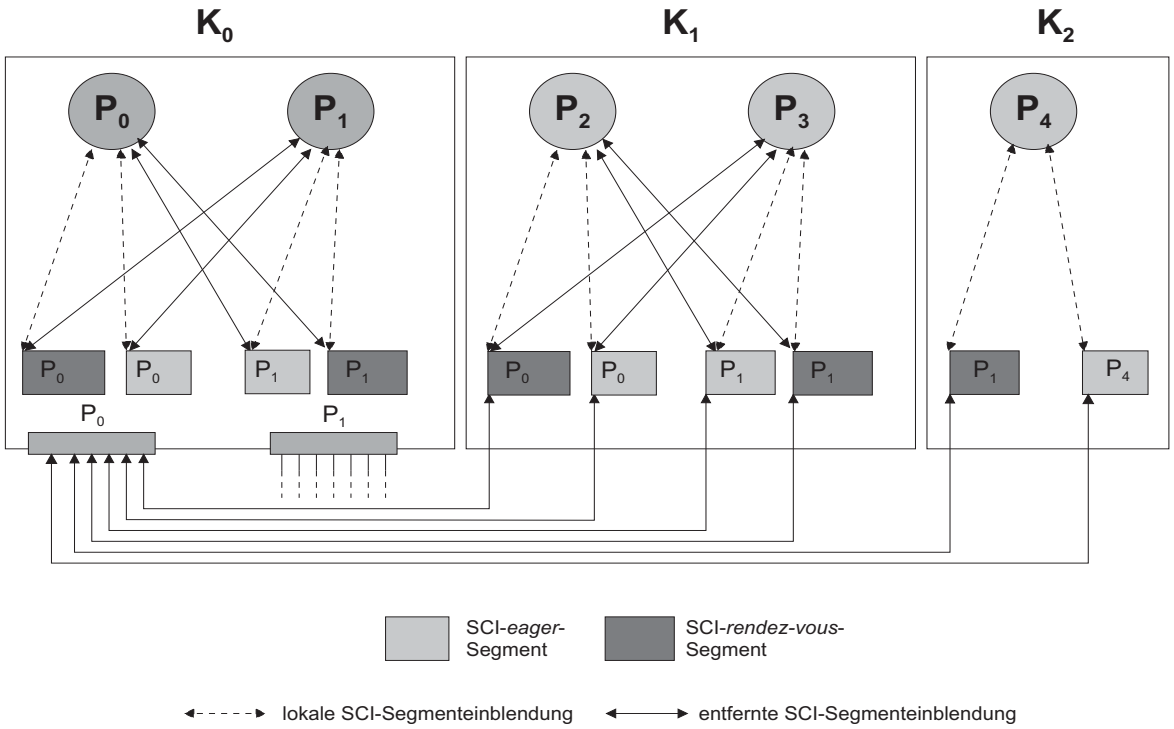


Abbildung 4.16: Kommunikationsressourcen für das *eager*- und *rendez-vous*-Protokoll für den statisch-direkten Fall. Wiederum sind nur die Inter-Knoten-Verbindungen von Prozeß 0 dargestellt (Prozesse 1 bis 4 analog).

	Beschreibung	Typ	Instanz	Größe
1	SMI: Basissegment	Speichersegment	Knoten 0	$S_{SMI_{base}}$
2	SMI: Internsegment	Speichersegment	pro Knoten	$S_{SMI_{intern}}$
3	SMI: Signal	Unterbrechung	pro Prozeß	-
4	ch_smi: *short*-Protokoll	Speichersegment	pro Prozeß oder pro Knoten	$S_{short} \cdot k \cdot (N-k)$
5	ch_smi: *eager*-Protokoll	Speichersegment	pro Prozeß oder pro Knoten	$S_{eager} \cdot k \cdot (N-k)$
6	ch_smi: *rendez-vous*-Protokoll	Speichersegment	pro Prozeß	S_{r-v}

Tabelle 4.3: Ressourcenbedarf für eine SCI-MPICH Applikation.
Die Elemente 1 bis 4 werden statisch allokiert und konnektiert, während die schattierten Elemente 5 und 6 statisch allokiert und dynamisch konnektiert werden.

sourcen durch Verbindung und ggf. Einblendung berücksichtigt werden. Die dazu erforderlichen Ressourcen ergeben sich aus den Tabellen 3.4 und 3.5 und werden im folgenden für die vollständige Verbindung zur Kommunikation von N Prozessen auf K Knoten gemäß

Tabelle 4.3 bestimmt.

Allokation. Zur Allokation der angegebenen Ressourcen werden *pro Knoten* folgende SCI-Ressourcen verbraucht:

- Physikalisch zusammenhängender Speicher (gleich dem benötigten Prozeßadreßraum):

$$(4.39)\ KS_{PZS} = S_{SMI_{intern}} + k \cdot ((S_{short} + S_{eager}) \cdot (N - k) + S_{r-v})$$

$$= KS_{PA_{lokal}}$$

Auf Knoten 0 zusätzlich für das *SMI*-Basissegment:

$$KS_{PZS_0} = S_{SMI_{base}} + S_{SMI_{intern}} + k \cdot ((S_{short} + S_{eager}) \cdot (N - k) + S_{r-v})$$

- lokale SCI-Deskriptoren:

$$(4.40)\ KN_{SD_{lokal}} = 3 + 2 \cdot k$$ (+ 1 auf Knoten 0 für das SMI-Basissegment)

Konnektierung. Zur Nutzung der entfernten Ressourcen werden *pro Knoten* folgende SCI-Ressourcen benötigt:

- Prozeßadreßraum:

$$(4.41)\ PS_{PA_{entf}} = k \cdot (K - 1) \cdot (S_{SMI_{base}} + S_{SMI_{intern}} + S_{short} + S_{eager} + S_{r-v})$$

(auf Knoten 0 entfällt der Anteil für $S_{SMI_{base}}$)

- ATT-Einträge:

$$(4.42)\ KN_{ATT} = \frac{K_{PA_{entf}}}{S_{ATT}}$$

- entfernte SCI-Deskriptoren (gleich der Zahl der Virtuellen Kanäle):

$$(4.43)\ KN_{SD_{lentf}} = 5 \cdot k \cdot (K - 1) + k^2 \cdot (K - 1)$$

$$= KN_{VC}$$

Für das *statische-direkte Modell* des Kommunikationsaufbaus ergibt dies für eine typische Konfiguration von 64 Prozessen in einer Applikation, von denen jeweils ein oder zwei auf einem Knoten laufen, der Ressourcenverbrauch pro Knoten wie in Tabelle 4.4 dargestellt. Eine solche Konfiguration benötigt weitaus mehr ATT-Einträge und SCI-Deskriptoren als im System verfügbar sind. Das Problem verschärft sich, wenn mehr als ein Prozeß pro Knoten gestartet wird, da es sich hier um Knotenressourcen[1] handelt. Auch der Anteil des Prozeßadreßraums, der für Kommunikationsressourcen belegt wird, ist beträchtlich.

Mit diesem Modell könnten nur Applikationen mit maximal etwa 50 (ein Prozeß pro Knoten) oder 27 (zwei Prozesse pro Knoten) Prozessen betrieben werden.

1. Die ATT-Einträge sind PSA-Ressourcen, was jedoch bei einem PSA pro Knoten gleichbedeutend ist.

N = 64	k = 1	k = 2
KS_{PZS}	9,4 MB	18,6 MB
PS_{PA}	604 MB	603 MB
KN_{ATT}	**9514**	**18712**
$KN_{SD_{lokal}}$	5	9
$KN_{SD_{lentf}}$	**317**	**571**

Tabelle 4.4: Ressourcenverbrauch für das statisch-direkte Modell

4.4.2 Maßnahmen zur Reduzierung des Ressourcenverbrauchs

Zur Reduzierung des Ressourcenverbrauchs wurde das statisch-direkte Modell optimiert (*statisch-optimiertes Modell*). Die einzelnen Maßnahmen werden im folgenden beschrieben.

4.4.2.1 Intra-Knoten Kommunikation

Die Kommunikation von Prozessen, die auf dem selben Knoten ausgeführt werden (*intra-Knoten Kommunikation)*, kann über die gleichen (SCI-)Ressourcen abgewickelt werden wie die Kommunikation zwischen Prozessen auf unterschiedlichen Knoten (*inter-Knoten Kommunikation)*. Ein solches Verfahren würde die Ressourcenverwaltung für alle Kommunikationspaarungen identisch halten, würde andererseits aber unnötigerweise SCI-Ressourcen verbrauchen und sich auch negativ auf die Kommunikationsleistung auswirken. Letzteres ist der Fall, weil ein Schreibvorgang auf ein lokal von Prozeß *i* angelegtes LS, das von Prozeß *j* auf dem selben Knoten in seinen Adreßraum eingeblendet wird, über den PCI-Bus abgewickelt wird. Dieser Kommunikationsweg ist natürlich weniger leistungsfähig als der direkte Zugriff über den Systembus.

Zur Vermeidung des unnötigen Verbrauchs von SCI-Ressourcen zur intra-Knoten Kommunikation wird der gemeinsame Speicher in diesem Fall durch entsprechende Mechanismen des Betriebssystems angelegt, die keinerlei SCI-Ressourcen benötigen. Durch die Nutzung der SMI-Bibliothek muß für dieses Verfahren nur ein anderer Typ für die einzurichtende Region von gemeinsamem Speicher angegeben werden. Neben diesem Unterschied bei der Initialisierung wird bei der Kommunikation (den Schreiboperationen in den Speicher eines anderen Prozesses) jeweils unterschieden, ob es sich um entfernten oder lokalen Speicher handelt, um das jeweils optimale Verfahren zum Datentransfer zu wählen. Die Protokolle bleiben davon jedoch unberührt, da diese Unterscheidung in einer tieferen Ebene angewandt wird.

4.4.2.2 Gemeinsame Nutzung von SCI-Ressourcen

SCI-Ressourcen, die zur Kommunikation mit Prozessen auf anderen Knoten verwendet werden müssen, können auf SMP-Knoten gemeinsam genutzt werden. So werden die Eingangspuffer für das *short*- und das *eager*-Protokoll für alle Prozesse auf einem Knoten in jeweils einem LS angelegt.

Der Pool für die dynamisch zu allokierenden Eingangspuffer im *rendez-vous*-Protokoll kann jedoch nicht mittels eines einzigen LS pro Knoten realisiert werden, da dies zu Blockaden führen kann, wenn ein Prozeß keinen Speicher mehr allokieren kann, obwohl er zu diesem Zeitpunkt keine *rendez-vous*-Übertragung durchführt.

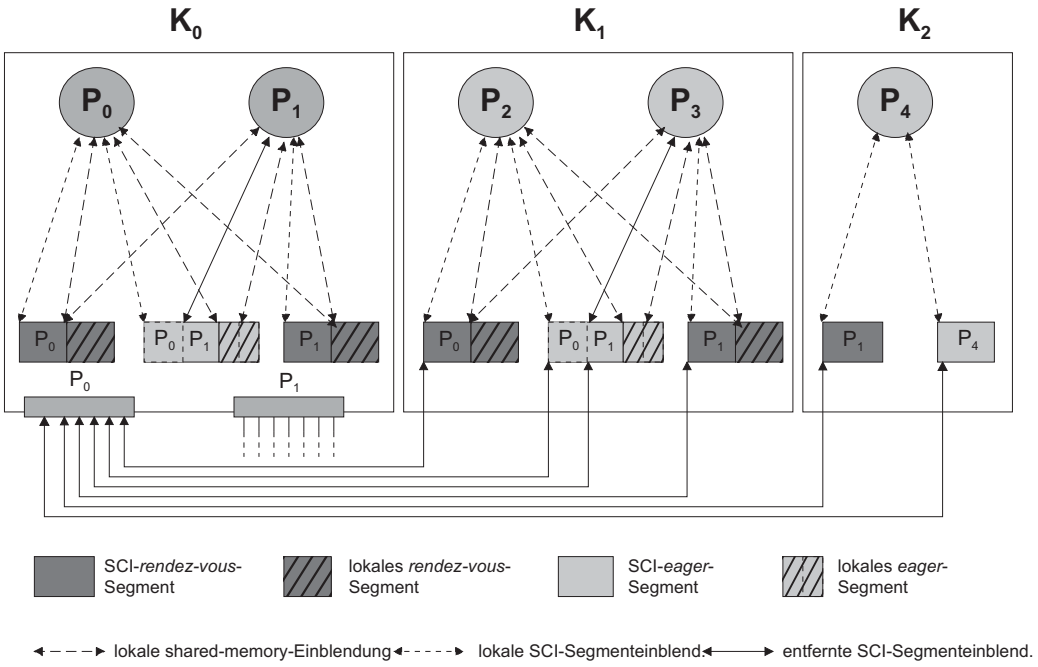


Abbildung 4.17: Kommunikationsressourcen für das *eager*- und *rendez-vous*-Protokoll für das statisch-optimierte bzw. dynamische Modell.
Wiederum sind nur die Inter-Knoten-Verbindungen von Prozeß 0 dargestellt (Prozesse 1 bis 4 analog).

4.4.2.3 Teileinblendung von entfernten Segmenten

Zur Kommunikation über das *eager*-Protokoll muß der sendende Prozeß nur auf denjenigen Teil der Eingangspuffer des Empfängers zugreifen, die seinem Prozeßrang zugeordnet sind. Alle anderen Teile des entsprechenden Segments sind für diesen Prozeß nicht relevant und brauchen nicht eingeblendet werden, was den Verbrauch an ATT-Einträgen und Prozeßadreßraum verringert.

Dies trifft grundsätzlich auch auf das *rendez-vous*-Protokoll zu. Jedoch ist hier der Eingangspuffer im entsprechenden Segment des Empfängers nicht fest angeordnet, sondern kann für jede Übertragung an einer anderen Stelle im Segment liegen. Daher ist es hier vorteilhafter, das gesamte Segment einzublenden zu versuchen. Nur dann, wenn dies nicht möglich ist, wird eine Teileinblendung durchgeführt. Die entstehende Segmentstruktur ist in Abb. 4.17 dargestellt.

4.4.2.4 Quantifizierung der Verringerung des SCI-Ressourcenverbrauchs

Für die bereits bekannte Konfiguration von $N = 64$ Prozessen auf $K = 64$ bzw. $K = 32$ Knoten ergibt sich mit diesen Optimierungen ein Verbrauch an SCI-Ressourcen wie in Tabelle 4.5 dargestellt. Durch die Optimierung des statisch-direkten Modells konnte der Verbrauch an ATT-Einträgen sowie der erforderliche Adreßraum pro Prozeß deutlich unter die maximal verfügbaren Ressourcen gesenkt werden. Der Verbrauch an SCI-Deskriptoren ist jedoch weiterhin zu hoch, so daß dieses *statisch-optimierte Modell* nicht mehr Prozesse in einer Applikation ausführen kann als das statisch-direkte Modell. Jedoch können die Prozesse größere bzw. mehr Eingangspuffer allokieren, was die Leistung der Applikation erhöhen kann.

4.4.3 Dynamische Verwaltung

Trotz der Maßnahmen zur Reduzierung des Ressourcenverbrauchs bei den statischen Modellen bleibt das Problem der unzureichenden Skalierbarkeit bestehen. Hier kann eine dynamische Verwaltung der Ressourcen Abhilfe schaffen.

Der direkte Ansatz dazu ist, für jeden Datentransfer die notwendigen Ressourcen zu allokieren und nach Abschluß des Datentransfers wieder freizugeben. Dies kann nur für entfernte Ressour-

N = 64	k = 1	k = 2
KS_{PZS}	9,3 MB	18,3 MB
PS_{PA}	116 MB	117 MB
KN_{ATT}	1764	3348
$KN_{SD_{lokal}}$	5	7
$KN_{SD_{lentf}}$	**315**	**439**

Tabelle 4.5: Ressourcenverbrauch für das statisch-optimierte Modell

cen derartig gehandhabt werden, da die lokal allokierten Ressourcen den anderen Prozessen jederzeit als Eingangspuffer verfügbar sein müssen. Ohnehin haben die lokalen Ressourcen nur einen geringen Anteil am Gesamtressourcenverbrauch. Auch für das *short*-Protokoll ist die dynamische Konnektierung nicht sinnvoll, da dies die Latenz stark erhöhen würde. Daher können nur die entfernten Ressourcen für das *eager*- und *rendez-vous*-Protokoll dynamisch allokiert werden. Die Umsetzung dieses *dynamischen Modells* für das bekannte Szenario resultiert in dem in Tabelle 4.6 dargestellten Ressourcenverbrauch

Das dynamische Modell kann mit den verfügbaren Ressourcen einen Ablauf der Applikation gewährleisten. Zusätzliche Ressourcen werden jeweils temporär für Kommunikation im *eager*- oder *rendez-vous*-Protokoll benötigt. Für diese Protokolle sinkt durch dieses Konzept jedoch die Leistung, also Bandbreite bzw. Latenz der Datenübertragung, nicht unerheblich, da der Zeitbedarf der Allokation und Deallokation im Verhältnis zur erforderlichen Zeit zur Übertragung keinesfalls vernachlässigt werden kann. In Abb. 4.18 sind einerseits der Zeitbedarf zur Einrichtung eines EES und der darüber abgewickelten PIO-Datenübertragung und andererseits der Zeitbedarf zur Einrichtung von VES und LRS und der darüber abgewickelten DMA-Datenübertragung dargestellt. Der Zeitbedarf aller dieser Vorgänge ist abhängig von der Größe der bezogenen Speichersegmente und ist daher für den Bereich von 4 kB bis 16 MB angegeben. Die Untergrenze von 4 kB entspricht der durch die Seitengröße der virtuellen Speicherverwaltung vorgegebenen minimalen Größe eines EES; die Obergrenze ist frei gewählt: Die Größe von EES, wie sie

N = 64	k = 1	k = 2
KS_{PZS}	9,3 MB	18,3 MB
PS_{PA}	49 MB	88 MB
KN_{ATT}	630	1116
$KN_{SD_{lokal}}$	5	7
$KN_{SD_{lentf}}$	189	253

Tabelle 4.6: Ressourcenverbrauch für das dynamische Modell

zum Nachrichtenaustausch verwendet werden, ist typischerweise kleiner. Die Größe von VES bei diesem Verwendungszweck entspricht der Größe der zu übertragenden Nachricht und ist somit theoretisch unbegrenzt (siehe Kapitel 7.2).

Um den Auf- und Abbau einer Verbindung für jeden Datentransfer zu vermeiden, müssen einmal aufgebaute Verbindungen daher möglichst lange bestehen bleiben und wiederholt genutzt werden (*Ressourcen-Caching*). Bei ungenügenden Ressourcen zum Aufbau einer neuen Verbindung muß jedoch eine andere Verbindung aufgelöst bzw. eine Ressource deallokiert werden, um die nötigen Ressourcen für den anstehenden Datentransfer zu erhalten. Dazu muß bekannt sein, welche Ressourcen durch welche andere Verbindungen allokiert sind und welche davon in diesem Moment nicht benötigt werden. Von diesen deallokierbaren Ressourcen muß nun die geeignete Ressource zur Freigabe ausgewählt werden. Ziel der Ressourcenverwaltung ist es also, neben einer Schnittstelle zur Allokation und Freigabe der Ressourcen beim Verbindungsaufbau und -abbau (wozu die SMI-Bibliothek eine geeignete Schnittstelle anbietet) ein möglichst effizientes Caching der Ressourcen durchzuführen und gleichzeitig sicherzustellen, daß eine Applikation nicht durch einen Ressourcenengpaß abgebrochen werden muß.

4.4.4 Ressourcenallokation, -freigabe und -verwaltung

Voraussetzung dafür, daß ein Ressourcenmanagement (RM) die formulierten Aufgaben erfüllen kann, ist die Information, welche Ressourcen allokiert und welche davon momentan in Gebrauch sind. Dies muß die übergeordnete Schicht, die die Ressourcen nutzt, im Falle von SCI dem RM explizit mitteilen, da die Nutzung der Ressource selbst, etwa eines EES durch PIO-Zugriffe, für das RM nicht ersichtlich ist.

Vor jeder Verwendung einer dieser Ressourcen muß durch einen Aufruf der zugehörigen Anforderungsfunktion sichergestellt werden, daß die Ressource auch verfügbar ist. Das RM gibt auf die Anforderung eine Referenz auf die angeforderte Ressource zurück, die es entweder einem internen Verzeichnis der bereits allokierten Ressourcen entnommen hat (falls die Ressource bereits einmal angefordert wurde) oder durch Allokation der angeforderten Ressource erhalten hat. Das interne Verzeichnis besteht aus zwei Hashtabellen *In Gebrauch* und *Verfügbar*, in denen die Referenzen auf momentan angeforderte (in Gebrauch befindliche) Ressourcen sowie auf die bereits allokierten, aber momentan nicht verwendeten Ressourcen abgelegt werden. Letztere Ressourcen sind verfügbar sowohl zur erneuten Verwendung als auch zur Deallokation und bilden damit den Cache des RM (*Ressourcencache)*. Der Schlüssel, über den die Referenzen auf die Ressourcen indiziert werden, wird aus der SCI-Segment-ID der Ressource,

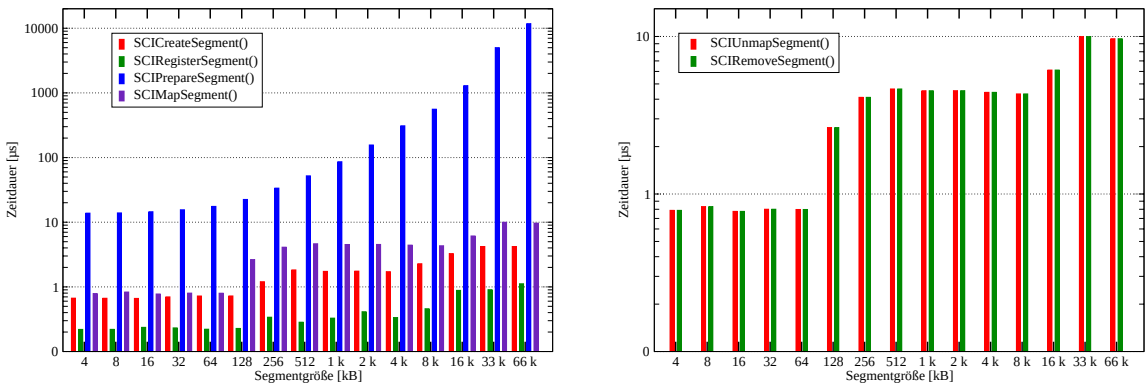


Abbildung 4.18: Zeitbedarf zur Einrichtung (*links*) und Entfernung (*rechts*) einer Kommunikationsressource vom Typ EES (*SCIMapSegment*()) und LRS (*SCIPrepareSegment()*) sowie Verbindung an ein entferntes LRS zur Erstellung eines VES (*SCIConnectSegment()*)

dem Prozeßrang des Prozesses, der die Ressource erzeugt hat, sowie aus dem Typ der Ressource erzeugt und ist somit eindeutig im ganzen System. Die Referenzen auf die Ressourcen enthalten alle Informationen, die zur Nutzung der Ressource erforderlich sind, sowie weitere Informationen, die für den Verdrängungsalgorithmus des Ressourcencache verwendet werden können. Neben diesen Datenstrukturen richtet ein anderes Modul, der *Ressourcenscheduler*, weitere Datenstrukturen ein, um die Steuerung (das *Scheduling*) der Deallokation von Ressourcen, was der Verdrängung aus dem Ressourcencache entspricht, durchführen zu können. Nach Abschluß der Nutzung einer Ressource muß diese durch Aufruf der entsprechenden Freigabefunktion freigegeben werden, um nötigenfalls deallokiert werden zu können. Schließlich kann eine Ressource explizit deallokiert werden, so daß sie anschließend nicht mehr im Ressourcencache zur Verfügung steht.

Die Schritte, die das RM bei jeder dieser Aktionen durchführt, sind in Abb. 4.19 dargestellt. Bei jedem Zugriff auf das RM muß zunächst der Schlüssel für die betreffende Ressource erstellt werden. Bei der Anforderung wird zunächst geprüft, ob ob die Ressource momentan in Gebrauch ist oder sich im Ressourcencache befindet. In beiden Fällen kann die Ressource, nachdem die internen Verwaltungsstrukturen in der Referenz auf die Ressource aktualisiert wurden, unmittelbar genutzt werden. Falls die Ressource allokiert werden muß, wird dies versucht. Falls dies mißlingt, liegt ein Ressourcenengpaß vor, der durch Deallokation einer anderen Ressource zu beheben versucht wird. Dazu wird der Ressourcenscheduler aufgerufen, der gemäß des gewählten Verdrängungsverfahrens aus den verfügbaren Ressourcen diejenige aussucht, die als nächste deallokiert werden soll, um die für die Erfüllung der Anforderung notwendige Alloka-

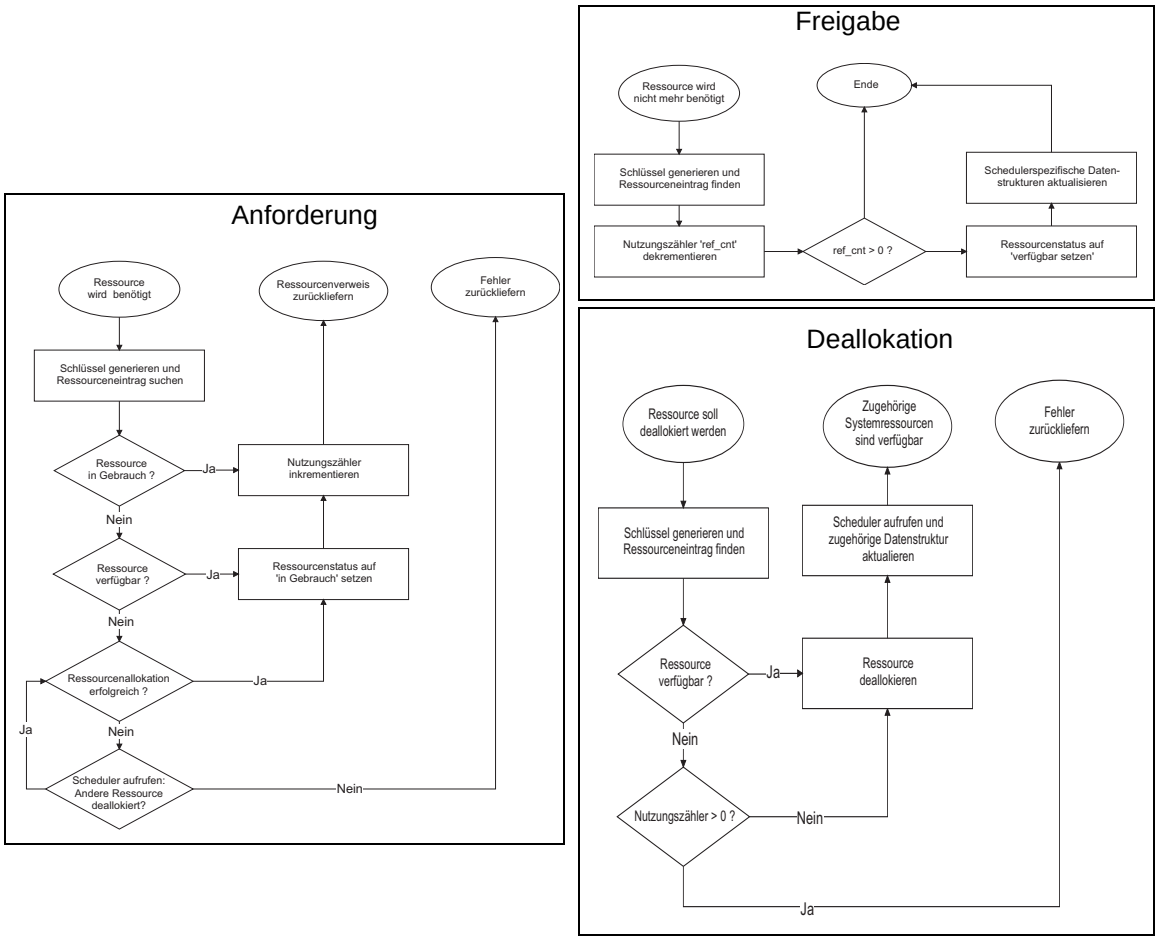


Abbildung 4.19: Anforderung (*links*), Freigabe (*rechts oben*) und Deallokation (*rechts unten*) einer Ressource durch den Ressourcenmanager

tion durchführen zu können. Dies wird so oft wiederholt, bis diese Allokation erfolgreich war. Ein endgültiges Fehlschlagen der Allokation, das dann zu konstatieren ist, wenn trotz Deallokation aller "verfügbaren" Ressourcen die notwendige Allokation nicht durchgeführt werden kann, kann in Verbindung mit den Maßnahmen zur Sequentialisierung (siehe Kapitel 5.1) nicht auftreten, wenn jeweils mindestens *eine* Anforderung für einen Datentransfer über das *rendezvous*-Protokoll erfüllt werden kann.

Auf SMP-Knoten, auf denen mehrere Prozesse einer MPI-Applikation ausgeführt werden, kommt eine weitere Möglichkeit hinzu, wie es zu Ressourcenengpässen in einem Prozeß kommen kann. Da die Prozesse unabhängig voneinander Ressourcen allokieren, kann es zu der Situation kommen, daß ein Prozeß P_i keine Ressourcen mehr freigeben kann und dennoch eine Anforderung nicht befriedigen kann, weil ein anderer Prozeß P_j einen entsprechend größeren Anteil der auf dem Knoten verfügbaren Ressourcen allokiert hat. Da der Prozeß P_j nicht veranlaßt ist, Teile seiner Ressourcen zu deallokieren, solange er sie nicht selber für andere Zwecke benötigt, würden so für den Fall der SMP-Nutzung durch mehrere Prozesse zusätzliche Ressourcen-bedingte Verklemmungen (mit der Folge des Abbruchs der Ausführung der Applikation) entstehen. Um derartige Verklemmungen aufzulösen, wurde ein inter-Prozeß-Ressourcenausgleichsprotokoll entwickelt. Dies ermöglicht dem blockierten Prozeß P_i, Prozeß P_j sowie alle weiteren Prozesse, die auf dem lokalen Knoten ausgeführt werden, mittels einer Kontrollnachricht vom Typ `RSRC_REQ` zur Freigabe einer benötigten Ressource aufzufordern. Der Prozeß, der eine solche Aufforderung erhält, versucht ihr nachzukommen und teilt das positive oder negative Ergebnis dieses Versuchs als Quittung in Form einer Kontrollnachricht vom Typ `RSRC_OK` mit. Je nachdem, ob diese Quittung positiv oder negativ ausfällt, versucht Prozeß P_i einen erneuten Versuch zur Allokation der benötigten Ressource oder fordert den nächsten lokalen Prozeß zur Deallokation einer Ressource auf. Erst wenn alle lokalen Prozesse negative Quittungen erteilt haben und die benötigte Ressource weiterhin nicht allokiert werden kann, muß die Ausführung der Applikation abgebrochen werden.

Bei der Freigabe einer Ressource wird die Referenz auf die Ressource von *In Gebrauch* nach *Verfügbar* überführt. Dabei wird wiederum der Ressourcenscheduler aufgerufen, der die neue verfügbare Ressource in die Verwaltungsstrukturen des Ressourcencaches aufnimmt. Bei einer expliziten Deallokation wird eine Ressource, sofern sie verfügbar ist, unmittelbar deallokiert und dabei aus allen Datenstrukturen des RM gelöscht.

4.4.5 Leistungsevaluation

Es wurden analog zu Speicher- und E/A-Caches verschiedene Verdrängungsverfahren implementiert, darunter LRU (*least recently used*) und ein Pseudoverfahren NULL, daß kein Caching durchführt, sondern jede Ressource unmittelbar freigibt und so das Caching deaktiviert. Dies erlaubt die direkte Messung des Leistungsgewinns durch das Ressourcencaching, das im folgenden für eine Latenzmessung zwischen zwei Prozessen durchgeführt wurde.

Das Diagramm in Abb. 4.20 zeigt die drastische Erhöhung der Latenz einer Nachrichtenübertragung bei dynamischer Ressourcenverwaltung ohne Caching. Bei kleinen Nachrichten im *eager*-Protokoll erhöht sich die Latenz um den Faktor 50; mit zunehmender Nachrichtenlänge verringert sich dieser Faktor durch die zunehmende Zeit für die Speicherzugriffe auf den entfernten Speicher. Da die Ressourcen für das *short*-Protokoll statisch allokiert werden, hat das Ressourcencaching hier keinen Einfluß.

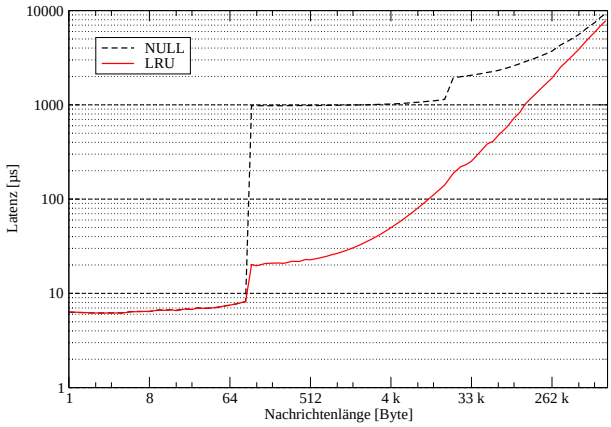


Abbildung 4.20: Auswirkung des Ressourcencaching auf die Nachrichtenlatenz (NULL: *ohne* Ressourcencaching, LRU: *mit* Ressourcencaching)

KAPITEL 5

Direkter Zugriff auf entfernten Speicher

Eine der elementaren Eigenschaften der speichergekoppelten Rechnerverbundsysteme ist der transparente Zugriff auf entfernte eingeblendete Speichersegmente, der mit sehr niedriger Verzögerung erfolgt. Neben dem effizienten Nachrichtenaustausch, wie in Kapitel 4 beschrieben, erlaubt diese Eigenschaft die Optimierung bestehender Kommunikationsverfahren und die effiziente Implementierung neuer Kommunikationsprinzipien, wie in diesem Kapitel für die Übertragung nicht-zusammenhängender Datenbereiche und die einseitige Kommunikation gezeigt wird.

5.1 Übertragung nicht-zusammenhängender Daten

Die Daten, die in MPI-Applikationen zwischen Prozessen ausgetauscht werden, sind immer Teil der globalen, auf die einzelnen Prozesse aufgeteilten Datenstrukturen, durch die das zu lösende Problem beschrieben wird. Diese globalen Datenstrukturen sind häufig mehrdimensionale Felder, entsprechend den Dimensionen des realen Problems. So werden beispielsweise bei der Simulation von Vorgängen in räumlichen Gebieten, etwa Ozeanen, dreidimensionale Felder verwendet. Die Aufteilung eines solchen globalen Feldes erfolgt häufig in zwei Dimensionen, um die Größen der Schnittflächen zu minimieren. Ein solcher Fall ist in Abb. 5.1 dargestellt: Das dreidimensionale Feld A (in Fortran-Indizierung) mit den Indizes *I*, *J* und *K*, das zur Simu-

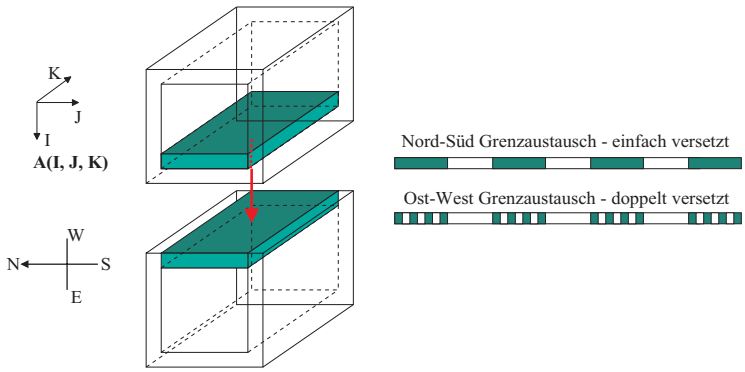


Abbildung 5.1: Austausch nichtkontinuierlicher Datenbereiche bei der Gebietszerlegung (aus *[136]*)

lation eines Gebiets eines Ozeans dient, wird sowohl in Nord-Süd- als auch in Ost-West-Richtung aufgeteilt, so daß die abgebildeten Teilgebiete mit überlappenden Randgebieten entstehen *[136]*. Die Inhalte der Randgebiete müssen nach jeder Iteration zwischen den Prozessen ausgetauscht werden. Die Daten dieser Randgebiete liegen im Speicher jedoch aufgrund der Tatsache, daß die Daten im eindimensional adressierbaren Speicher nur in einer Dimensionsrichtung abgelegt werden können, nicht als kontinuierlicher Block vor, sondern als einfach oder doppelt versetzte Folge von separaten Datenblöcken. Durch eine andere Gebietsaufteilung, die in diesem Fall jedoch aus algorithmischen Gründen nicht angebracht war, hätte der doppelt versetzte Randbereich vermieden werden können; nicht-kontinuierliche Datenbereiche erhält man jedoch grundsätzlich bei der Aufteilung von Gebieten entlang mehr als einer Dimension.

Um einen solchen Datenbereich mittels MPI an einen anderen Prozeß zu schicken, gibt es drei Möglichkeiten:

- *Einzelnachrichtenversand.* Für jeden kontinuierlichen Block von Daten wird eine einzelne Nachricht versandt. Dieses Vorgehen führt zu einer drastisch reduzierten effektiven Bandbreite, da für jede Nachricht aufs neue die Startlatenz anfällt.

- *Manuelles Packen.* Der Benutzer kopiert die Daten in einen separaten Puffer so um, daß ein kontinuierlicher Datenblock entsteht, und verschickt diesen mit einer einzigen Nachricht. Dies erfordert Aufwand vom Benutzer und bedingt einen zusätzlichen Kopiervorgang im lokalen Speicher, der bei einem im Verhältnis zur Speicherbandbreite leistungsfähigen Verbindungsnetz die Bandbreite deutlich reduziert.
- *Abgeleitete Datentypen.* Der Benutzer definiert in MPI einen *abgeleiteten Datentyp*, der die nicht-kontinuierliche Datenverteilung beschreibt. Unter Verweis auf diesen Datentyp kann der gesamte Datenbereich als eine einzige Nachricht verschickt werden. Es ist Aufgabe der MPI Bibliothek, die Daten möglichst effizient zu übertragen.

Die naive Methode zur Übertragung, die daraus besteht, die Lücken zwischen den Datentypen einfach mit in einer Nachricht zu übertragen, ist nicht nur wegen der zusätzlich transportierten Daten ineffizient, sondern auch generell unzulässig, da so beim Empfänger die Daten überschrieben würden, die nicht Teil der empfangenen Nachricht sind.

Von den drei vorgestellten zulässigen Methoden ist die der abgeleiteten Datentypen vorzuziehen, da diese für den Benutzer am wenigsten Aufwand impliziert und vor allem der MPI-Bibliothek die Gelegenheit gibt, den Datentransfer auf die bestmögliche Weise abzuwickeln.

5.1.1 Abgeleitete Datentypen

Aus einer Reihe von Basis-Datentypen, die im wesentlichen den verfügbaren Datentypen in den Programmiersprachen entsprechen, für die MPI spezifiziert ist, lassen sich beliebig komplexe abgeleitete Datentypen zusammensetzen. Neben der Definition der Abfolge der Basis-Datentypen (den *Elementen*) in dem abgeleiteten Datentyp ist es wesentlich, den Abstand vor, zwischen und nach den Elementen festlegen zu können, um so eine nicht-kontinuierliche Datenverteilung wie in dem obigen Beispiel beschreiben zu können. Abgeleitete Datentypen selber können wiederum zur Definition weiterer Datentypen verwendet werden.

Zur Definition der abgeleiteten Datentypen stehen Funktionen zur Verfügung, die für typische Fälle (etwa Vektoren wie im obigen Beispiel) eine vereinfachte Vorgehensweise zulassen. Neben derartigen regelmäßigen Strukturen kann jede denkbare Anordnung von Daten im Speicher als abgeleiteter Datentyp definiert werden. Dies geschieht auf der Basis der initialen Plazierung eines Elements im Datentyp (*offset)*, seiner Wiederholungszahl (*count*) und der Plazierung der wiederholten Elemente relativ zum vorhergehenden Element (*stride*). Ein Beispiel für den Aufbau im Speicher und die Definition in MPI für einen abgeleiteten, nicht-zusammenhängenden Datentyp ist in Abb. 5.2 gegeben. Eine Struktur aus einer `integer`-Variablen und folgend fünf

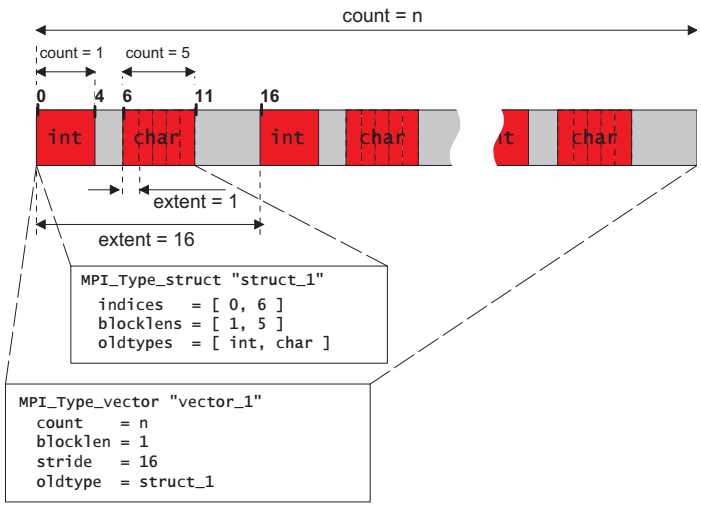


Abbildung 5.2: Aufbau und Definition eines nicht-zusammenhängenden Datentyps in MPI

`character`-Variablen wird mittels `MPI_Type_struct` zu einem neuen Datentyp *struct_1* definiert. Im Speicher liegen *n* Instanzen dieses Datentyps alle 16 Byte hintereinander, die mittels `MPI_Type_vector` zu einem Datentyp *vector_1* definiert werden.

Die naheliegende Datenstruktur zur Repräsentation eines abgeleiteten Datentyps ist ein Baum, dessen Blätter die einzelnen Elemente sind. In den Knoten sind Angaben zur Plazierung und Wiederholung gespeichert. Die Wurzel eines solchen Datentypbaumes kann wiederum Blatt eines übergeordneten Datentyps sein. Ein solcher Baum kann mit bekannten, rekursiven Verfahren traversiert werden (z.B. Tiefensuche *[156]*), um auf alle seine Elemente zuzugreifen und so den Datentyp zu lesen oder zu schreiben. Viele MPI-Implementationen verwenden daher einen Baum zur internen Repräsentation von Datentypen (siehe Abb. 5.3 *links*).

Alternativ kann ein Datentyp auch *flach* dargestellt werden (*datatype flattening*), indem alle Elemente eine verkettete Liste bilden, in der jeder Knoten Plazierung und Länge eines Speicherbereichs beschreibt. Diese Art von Beschreibung kann auch von einfachen Kommunikationsschnittstellen wie etwa *sockets [163]* oder DMA-Bausteinen auf Netzwerkkarten interpretiert werden. Der offensichtliche Nachteil dieser Darstellung ist jedoch der benötigte Zeitbedarf und vor allem der Speicherplatzbedarf, um eine solche Darstellung zu erzeugen. Beide Ressourcen wachsen linear mit der Zahl n_b der nicht-zusammenhängenden Datenblöcke des Datentyps, so daß im ungünstigen, aber nicht seltenen Extremfall eines Vektors von nicht-zusammenhängenden Basis-Datentypen die Darstellung des Datentyps mehr Speicherplatz verbraucht als die eigentlichen Daten selber (siehe Abb. 5.3 *rechts*).

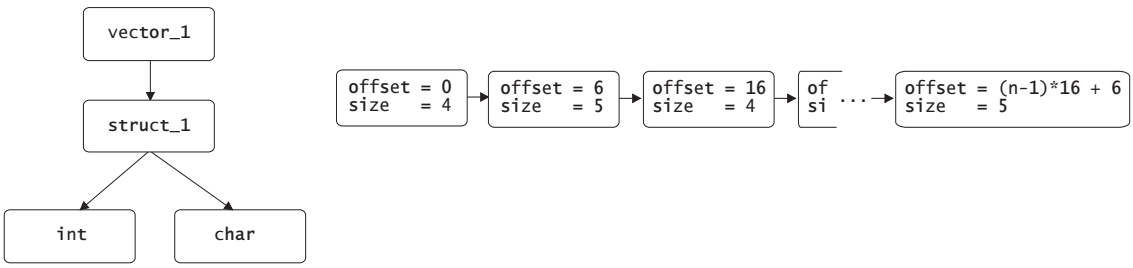


Abbildung 5.3: Interne Repräsentation des Datentyps aus Abb. 5.2 als Baum (*links*) und Liste (*rechts*)

5.1.2 Kommunikation mit abgeleiteten Datentypen

Bei der Definition eines abgeleiteten Datentyps kann die MPI-Bibliothek feststellen, ob dieser als zusammenhängender Block von Daten beschrieben werden kann oder nicht. Im ersten Fall erfolgt der Versand und Empfang der Daten auf die herkömmliche Weise. Im Fall von nicht-zusammenhängenden Datentypen muß ein Weg gefunden werden, diese Daten effizient über das zugrundeliegende Netz zu übertragen. Die meisten Kommunikationsschnittstellen für Netze basieren auf dem Konzept eines durchgängigen Datenstroms (*stream*) und verwenden daher als Beschreibung der zu kommunizierenden Daten ein Tupel {*Adresse, Länge*}, das nicht-zusammenhängende Daten nicht als Ganzes beschreiben kann. Ergänzend dazu wird verschiedentlich auch eine Beschreibung der Daten als verkette Liste (flache Beschreibung) verwendet. Aufgrund der beschriebenen Probleme bei der flachen Darstellung eines Datentyps als verkettete Liste verwenden die meisten MPI-Bibliotheken daher das Verfahren des *expliziten Packens / Entpackens* (siehe Abb. 5.4 (*links*)). Dabei werden die zu versendenden nicht-zusammenhängenden Daten zunächst in einen lokalen temporären Puffer *gepackt*, so daß sie dort einen zusammenhängenden Datenblock bilden, der durch ein Tupel {*Adresse, Länge*}vollständig beschrieben ist und somit über jede verfügbare Kommunikationsschnittstelle übertragen werden kann. Auf der Empfängerseite werden die im Eingangspuffer empfangenen, gepackten Daten entsprechend des Empfangsdatentyps in den Empfangspuffer *entpackt*. Für die bei längeren Nachrichten übliche Fragmentierung einer Nachricht bei der Übertragung über das Netz müssen

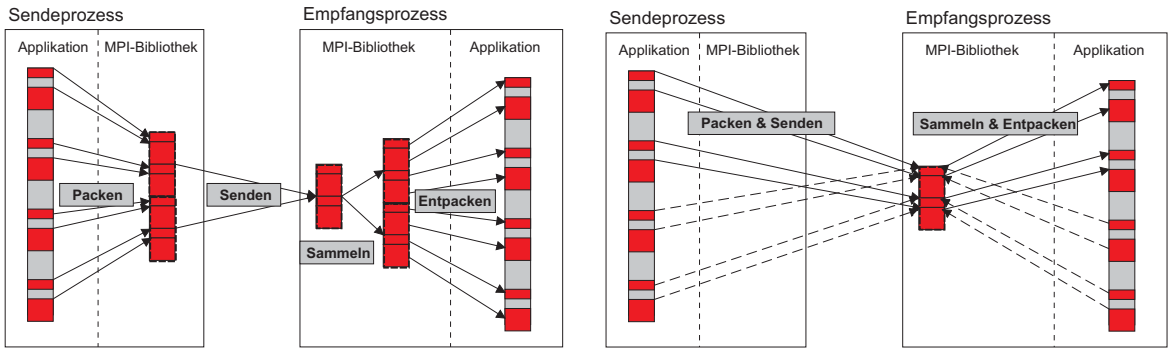


Abbildung 5.4: Übertragung von nicht-zusammenhängenden Datentypen mittels *expliziten Packens / Entpackens (links)* und mittels *direkten Packens/Entspackens (rechts)* (hier bei fragmentierter Übertragung).

die empfangenen Fragmente zunächst wiederum in einem temporären Puffer *gesammelt* werden, um für den Entpackvorgang die gepackten Daten vollständig und zusammenhängend vorliegen zu haben.

Insgesamt erfordert dieses Verfahren bei fragmentierter Übertragung mindestens drei lokale Kopieroperationen sowie die eigentliche Übertragung der Daten über das Netz. Bei nicht-fragmentierter Übertragung, etwa im *short*- oder *eager*-Protokoll, kommen zu der eigentlichen Übertragung der Daten zwei lokale Kopieroperationen hinzu. Es ist offensichtlich, daß durch die zusätzlichen Kopieroperationen, die bis auf das Sammeln sequentiell abgewickelt werden müssen, die effektive Bandbreite reduziert wird. Ein Verfahren zur Optimierung der Leistung muß entsprechend die zusätzlichen Kopiervorgänge eliminieren.

5.1.3 Optimierungspotential bei der Kommunikation

In *[87]* wird erwähnt, daß durch *flattening* des Datentyps bei seiner Erzeugung beim anschließenden Umgang damit effizienter gearbeitet werden kann, da statt der rekursiven Darstellung eine flache Beschreibung verwendet wird. Allerdings wird nicht auf den Aufbau der verwendeten Datentypen eingegangen, wodurch auch die Frage nach dem Speicherplatzverbrauch der angelegten Tabelle unbeantwortet bleibt. Das Packen der Daten in einen zusammenhängenden Puffer kann mit dieser Methode allein nicht umgangen werden[1]. Auch für den Fall der *shared-memory*-Kommunikation wird weiterhin gepackt. Vielmehr ist dort das Ziel gewesen, die Kommunikation unter Nutzung von Datentypen genauso effizient zu gestalten wie das manuelle Packen der Daten durch den Benutzer und anschließende Versenden des kontinuierlichen Puffers. Dies ist für die untersuchten Beispiele annähernd gelungen, doch ein Vergleich der erzielten Ergebnisse mit der Leistung beim Versand eines äquivalenten kontinuierlichen Puffers wird nicht gemacht.

Die Leistungscharakteristik der Kommunikation über SCI mit den sehr kurzen Startlatenzen zum Übertragen eines Datums läßt es effektiv erscheinen, auf das lokale Packen der Daten zu verzichten und stattdessen direkt in den Eingangspuffer des Empfangsprozesses zu packen, d.h. die Daten dort kontinuierlich abzulegen. Da dies ein kontinuierlicher Schreibvorgang in den entfernten Speicher ist (nur mit einem nicht-kontinuierlichen Quellpuffer), können die leistungssteigernden Verfahren für die SCI-Übertragung (Sammeln der Daten zur Generierung von SCI-Paketen maximaler Größe) weiterhin wirken. Auf der Empfängerseite sollten sodann die Daten direkt vom Eingangspuffer in den Empfangspuffer entpackt werden, so daß insgesamt

1. Dies ist eine Frage der Fähigkeiten des Verbindungsnetzes, daß derartige Transportvorgänge durchführen muß.

wie bei der Übertragung kontinuierlicher Daten zwei Kopiervorgänge erforderlich sind (siehe Abb. 5.4 *rechts*).

Das wesentliche Problem bei diesem Verfahren tritt bei der fragmentierten Übertragung von großen Nachrichten mit dem *rendez-vous*-Protokoll auf: In diesem Fall muß sowohl der Pack- als auch der Entpackalgorithmus in der Lage sein, Teile der Nachricht stückweise zu packen bzw. zu entpacken, je nachdem, welcher Teil der Nachricht gerade gesendet bzw. empfangen wird. Der Algorithmus muß also, im Gegensatz zu den existierenden Pack/Entpackalgorithmen, *neustartbar* sein, und zwar an jeder beliebigen Position innerhalb der nicht-kontinuierlichen oder gepackten Daten, und von dieser Position an eine beliebige Menge an gepackten Daten erzeugen bzw. entpacken. Dabei ist wichtig, daß dieser Neustart in möglichst kurzer Zeit erfolgt und dazu möglichst wenig Speicher verwendet werden muß, um negative Leistungseinflüsse zu vermeiden.

Im weiteren wird zunächst anhand einer Modellierung untersucht, welche Leistungsgewinne durch das beschriebene *direkte Packen* zu erwarten sind. Im Anschluß wird beschrieben, wie der erforderliche neustartbare Packalgorithmus und die zugehörigen Datenstrukturen gestaltet sind. Schließlich wird der erzielte Leistungsgewinn, auch im Hinblick auf den Einfluß des Packens/Entpackens auf den Zustand der CPU-Caches, untersucht.

5.1.3.1 Modellierung

Für die Modellierung des Versands nichtzusammenhängender Daten wird von einem *Vector*-Datentyp ausgegangen, der aus n_b Blöcken der Größe l_b mit einem Stride $s_b > 1$ aufgebaut ist. Dieser Datentyp hat entsprechend eine Daten-Nutzlast $D = n_b \cdot l_b$. Die Verwendung einer einzelnen, festen Blockgröße und eines beliebigen Strides ist zum einen sinnvoll, da dies gemäß dem beschriebenen typischen Einsatzgebiet von nicht-zusammenhängenden Datentypen (die Kommunikation der Ausschnitte von mehrdimensionalen Feldern) der häufigste auftretende Datentyp ist. Darüberhinaus ließen sich beliebig aufgebaute Datentypen nur statistisch modellieren, ohne daß diese Modellierung andere Erkenntnisse liefern würde. Dies liegt darin begründet, daß sich aus einer Überlagerung einer Zahl von Vektordatentypen jeder andere Datentyp (analog zur Fourieranalyse) erzeugen läßt.

Die erzielbare Latenz $L_{expl}(n_b,l_b)$ für den Versand von Daten eines solchen Datentyps mittels expliziten Packens/Entpackens kann abgeleitet werden aus der Latenz des Packens (beim Sender) und Entpackens (beim Empfänger), der Übertragung des Datenblocks in den entfernten Speicher sowie das dortige Sammeln der Daten durch eine weitere Kopieroperation[1]. Da diese Vorgänge sequentiell ausgeführt werden, können die einzelnen Latenzen zur Gesamtlatenz addiert werden:

$$(5.1)\quad L_{expl}(n_b,l_b) = \frac{n_b}{B_{cl}(l_b)} + \frac{1}{B_{cr}(D)} + \frac{1}{B_{cl}(D)} + \frac{n_b}{B_{cl}(l_b)} = \frac{2 \cdot n_b}{B_{cl}(l_b)} + \frac{1}{B_{cr}(D)} + \frac{1}{B_{cl}(D)}$$

Die Bandbreite ist entsprechend

$$(5.2)\quad B_{expl}(n_b,l_b) = \frac{D}{L_{expl}(n_b,l_b)}$$

Im Gegensatz dazu fällt beim direkten Versand nur das Kopieren der einzelnen Blöcke direkt aus dem Sendepuffer in den entfernten Eingangspuffer und von dort in den Empfangspuffer an. Dies bedeutet, daß in den bestehenden Protokollen zur Nachrichtenübertragung nur die Kopierfunktion für zusammenhängende Datenblöcke durch die spezielle, *direkte* Pack/Entpack-Funk-

1. Obwohl nur die Übertragung eines Blocks berücksichtigt wird, ist dieses Modell auch für *rendez-vous* Übertragungen gültig, wenn davon ausgegangen wird, daß die Bandbreite für Blöcke der Größe *N* bereits maximal ist.

tion ersetzt werden muß. Somit bleibt die Charakteristik der Protokolle, etwa das Pipelining beim *rendez-vous*-Protokoll, vollständig erhalten. Zur Modellierung reicht es daher aus, die Funktionen zur Bandbreite von Kopieroperationen eines zusammenhängenden Datenblocks der Länge D in den entfernten Speicher ($B_{cr}(D)$) bzw. innerhalb des lokalen Speichers ($B_{cl}(D)$) durch erweiterte Funktionen zum Kopieren von nichtzusammenhängenden Datentypen zu ersetzen. Für die hier angestrebte Modellierung können diese Funktionen als $B_{cr_d}(n_b,l_b)$ bzw. $B_{cl_d}(n_b,l_b)$ formuliert werden. Diese lassen sich beschreiben als

$$(5.3)\quad B_{cr_d}(n_b,l_b) = n_b \cdot l_b \Big/ \left(n_b \cdot \frac{l_b}{B_{cr}(l_b)} \right) = B_{cr}(l_b) \text{ und entsprechend}$$

$$B_{cl_d}(n_b,l_b) = B_{cl}(l_b)\,.$$

In Abb. 5.5 sind die nach diesem Modell erzielbaren Bandbreiten für die Übertragung von nicht-kontinuierlichen Daten über die variable Blocklänge l_b für ein festes Gesamtdatenvolumen aufgetragen. Dabei wurde die Bandbreite B_{expl} sowohl gemäß dem Modell als auch als experimentell bestimmter Wert aufgetragen. Die Bandbreite B_{direkt} für die direkte Übertragung der nicht-zusammenhängenden Daten wurde bestimmt, indem die modifizierte Kopierbandbreite gemäß (5.3) in die Protokollmodelle aus Kapitel 4 eingesetzt wurde. Als Referenzwert ist die Bandbreite B_{kont} eingetragen, die für einen zusammenhängenden Vektor der Länge D erreicht wird.

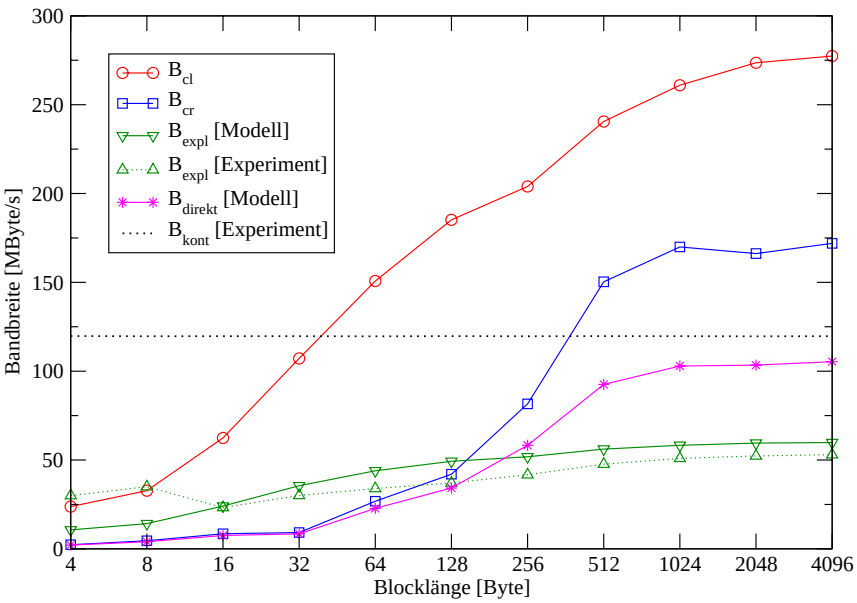


Abbildung 5.5: Potentieller Leistungsgewinn bei der Übertragung von nicht-kontinuierlichen Daten durch Direktübertragung gegenüber explizitem Packen/Entpacken

Es wird deutlich, daß für Blocklängen oberhalb eines Schwellwerts (hier: etwa 256 Byte) die Bandbreite für das direkte Packen höher liegt als für das explizite Packen. Es lohnt sich also, diesen Ansatz zu verfolgen. Es muß jedoch versucht werden, auch für kleine Blocklängen mindestens die Leistung des expliziten Packens zu erreichen. Durch einen hybriden Ansatz, in dem kleine Sub-Blöcke explizit gepackt werden und anschließend direkt übertragen werden, läßt sich die Leistung möglicherweise weiter erhöhen.

5.1.4 Direkter Pack/Entpackalgorithmus mit transparentem Zugriff

Um den potentiellen Leistungsgewinn zu realisieren, wurde für SCI-MPICH ein neuartiger Pack- und Entpackalgorithmus unter der Bezeichnung *direktes Verfahren* (*direct_ff*, *[141]*) ent-

wickelt, der beim Packen die Daten direkt in den Eingangspuffer des Empfängers schreibt, von wo sie mit dem gleichen Verfahren direkt in den Empfangspuffer übertragen werden. Um dies effizient und flexibel zu implementieren, konnte nicht auf die vorhandenen Verfahren aufgebaut werden.

Nachrichten im *short*- und *eager*-Protokoll werden stets in einer einzelnen Kopieroperation in einen bereitstehenden Eingangspuffer bzw. den Datenbereich des Kontrollpakets kopiert (vergl. Kapitel 4.2.2 und 4.2.3), so daß in diesen Fällen auch der Datentypbaum rekursiv durchlaufen werden könnte, um einen nicht-zusammenhängenden Datentyp *direkt* zu kopieren. Bei der Übertragung einer Nachricht im *rendez-vous*-Protokoll muß jedoch damit gerechnet werden, daß diese *fragmentiert*, also in mehreren Teilen mit dazwischen abzuwickelnder Flußkontrolle, übertragen wird (vergl. Kapitel 4.2.4). Zwischen den Übertragungen dieser Fragmente wird der Kontext der Übertragung, also auch der Kontext der Pack-Funktion verlassen, womit der Rekursionsvorgang zum Traversieren des Datentypbaumes abgebrochen würde. Ein Wiederaufsetzen für die Übertragung des nächsten Fragments erforderte das erneute rekursive Traversieren des Baumes, bis die Schreibposition innerhalb der linearen Darstellung des Datentyps am Ende des zuvor übertragenen Fragments angelangt ist. Von dort ab könnte sodann das folgende Fragment in den Eingangspuffer geschrieben werden. Der Zeitbedarf zum Wiederfinden der Schreibposition wächst hierbei linear mit der Komplexität (Tiefe und Breite) des Datentypbaums, den Wiederholungszahlen (für Teile) des Datentyps sowie in erster Ordnung arithmetisch mit der Zahl der Fragmente für die Übertragung.

Es wird klar, daß mit dieser Methode der Aufwand für die direkte Übertragung schnell über dem für explizites Packen/Entpacken liegt. Aber auch für die anderen Protokolle verspricht eine Datenstruktur, die bei ihrer Nutzung weniger Funktionsaufrufe und Zeiger-Dereferenzierungen benötigt als ein rekursiv abzuarbeitender Datentypbaum, eine höhere Kopierleistung.

5.1.4.1 Datenstruktur

Statt der intuitiven und naheliegenden Baumstruktur muß eine andere Darstellung des Datentyps gefunden werden, die folgende Kriterien erfüllt:

- Der zeitliche Aufwand zur Erstellung und der Speicherbedarf zur Speicherung der Darstellung ist proportional zu der Zahl der nicht-kontinuierlichen Datenblöcke des Datentyps innerhalb einer Instanz des Datentyps. Die Zahl der Wiederholungen von Elementen innerhalb des Datentyps oder des Datentyps selber darf keinen Einfluß haben. Aus mehreren Elementen bestehende zusammenhängende Datenblöcke sollen als solche behandelt werden.
- Das Wiederaufsetzen an einer beliebigen Position in der linearen Darstellung des Datentyps im Speicher soll nicht von der Wiederholungszahl von Elementen des Datentyps oder des Datentyps insgesamt abhängig sein; sonstige Abhängigkeiten sollen maximal linear sein.
- Optimierungen der Kopiervorgänge in den entfernten Speicher, wie sie gemäß der Leistungscharakteristik des Verbindungsnetzes erforderlich sein können, sollen möglich sein. Dies erfordert eine Sortierung der Elemente des Datentyps nach ihrer Größe.

Wie bereits dargelegt, werden diese Bedingungen von einer Baum- oder linearen Listendarstellung nicht erfüllt. Stattdessen wurde eine Darstellung in Form einer Liste entwickelt, deren Elemente nicht die einzelnen zu kopierenden Datenblöcke sind, sondern Beschreibungen des Vorkommens eines (sich wiederholenden) zusammenhängenden Datenblocks im gesamten Datentyp. Diese Beschreibungen enthalten die Information, die der Rekursions-Stack bei der Traversierung eines Datentypbaums enthält, wenn die Rekursion an dem Blatt eines Baumes angelangt ist (vergl. *[156]*) und sind entsprechend als *Stack* (Stapel) abgelegt. Somit enthält die Liste für jedes Blatt des entsprechenden Datentypbaums einen Stack (*Stackliste*). Der Algorithmus zum Aufbau dieser *Stackliste* ist in Abb. 5.6 als Flußdiagramm dargestellt.

Die Anwendung dieses Algorithmus auf den Datentyp-Baum für den Beispieldatentyp aus

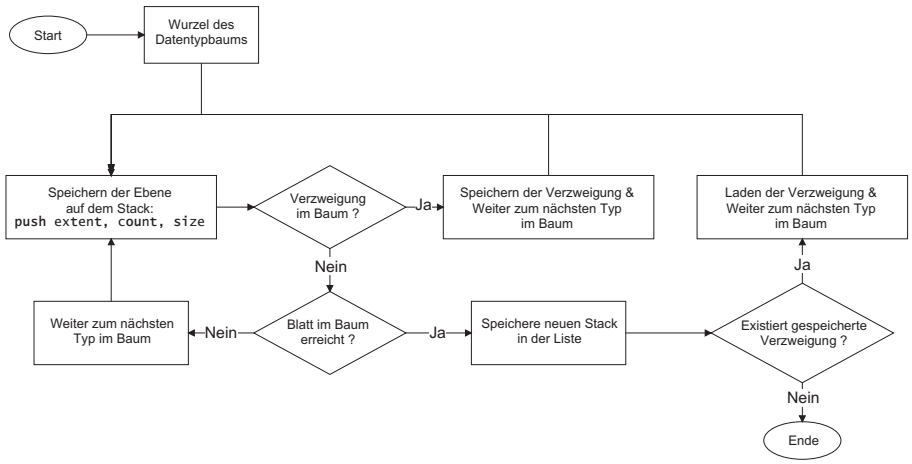


Abbildung 5.6: Algorithmus zum Aufbau der Datentyp-Stackliste aus dem Datentyp-Baum

Abb. 5.2 ergibt eine Stackliste wie in Abb. 5.7 (*links*) dargestellt. Die zwei Elemente der Liste, entsprechend den beiden Basisdatentypen `int` und `char` als Blätter des Datentyp-Baums aus Abb. 5.3 (*links*), enthalten die Größe und Position des erstmaligen Auftretens dieses Basisdatentyps und den Verweis auf den Stack, der das Wiederholungsmuster dieses Datenblocks beschreibt. Dabei enthalten die Stacks die gleiche Zahl von Ebenen wie der Datentyp-Baum. Ein Teil dieser Ebenen enthält häufig redundante Informationen, wie z.B. die oberen beiden Ebenen des Stacks für den Basisdatentyp `int`. In einem weiteren Schritt, genannt *merging*, werden daher die redundanten Informationen aus den Stacks entfernt und diese entsprechend vereinfacht. Dabei werden zwei Regeln angewandt:

Alle Ebenen eines Stacks mit einem Zähler von '1' ($count = 1$) sind redundant, solange noch weitere Ebenen vorhanden sind. Diese Regel gilt, weil die Gesamtgröße des durch diesen Basisdatentyps gebildeten zusammenhängenden Datenblocks, der durch diesen Stack beschrieben wird, das Produkt der *count*-Werte aller Ebenen des Stacks ist. Da '1' das neutrale Element der Multiplikation ist, tragen diese Ebenen nicht zur Gesamtgröße bei.

Alle Ebenen, für die $extent = size$ gilt, sind redundant, wenn die im Listeneintrag angegebene Größe des Basisblocks contig_size gemäß $contig\_size = size \cdot count_{top}$ (mit $count_{top}$ als count-Wert des obersten Eintrags auf dem Stack) ersetzt wird. Derartige Ebenen in einem Stack (für die $count \neq 1$ gilt, da die Ebene in dem Fall $count = 1$ schon gemäß der ersten Regel entfernt worden wäre) stellen zusammenhängende Datenblöcke aus Vektoren des Basisdatentyps dar, die jedoch nicht als zusammenhängend definiert wurden. Die Stackliste für den Beispieldatentyp nach dem *merging* ist in Abb. 5.7 (*rechts*) dargestellt.

5.1.4.2 Kopierverfahren

Der Kopiervorgang für einen nicht-zusammenhängenden Datentyp mit Stacklisten-Beschreibung besteht auf oberster Ebene aus drei Schritten:

1. *Wiederaufsetzen*: Es wird der Stack in der Stackliste bestimmt, der bei einem ggf. vorhergegangenen Kopierschritt eines Fragments nicht vollständig abgearbeitet (d.h. übertragen) wurde.
2. *Restkopie*: Der Rest des zuvor bestimmten Stacks wird kopiert.
3. *Stackkopie*: Weitere Stacks werden kopiert, bis die Kapazität des Zielpuffers erschöpft ist.

Als Pseudocode sind diese Vorgänge in Abb. 5.8 dargestellt. Die Schritte 1 und 2 können nur beim fragmentierten Kopieren im Rahmen des *rendez-vous*-Protokolls auftreten, bei denen das Wiederaufsetzen erforderlich ist.

Wiederaufsetzen. Beim Wiederaufsetzen muß anhand einer Byte-genauen Angabe der bereits

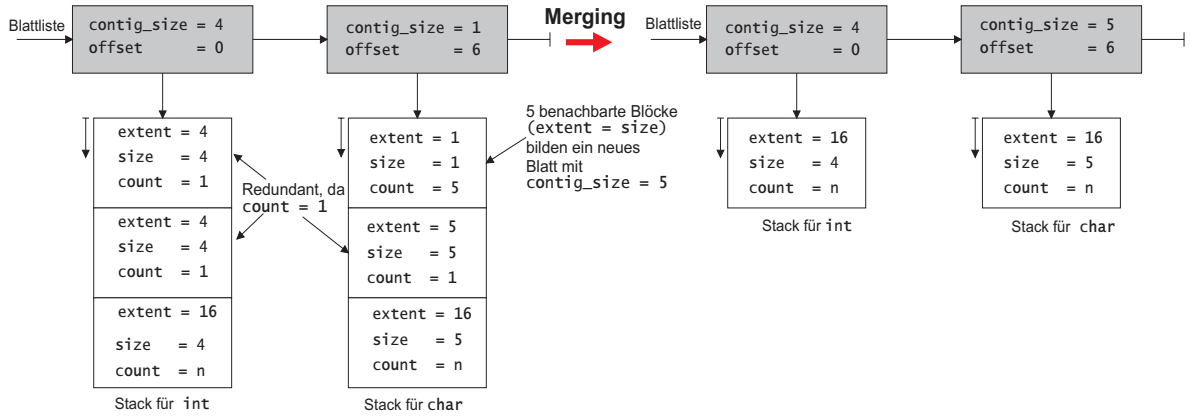


Abbildung 5.7: Darstellung des Datentyps aus Abb. 5.2 als Liste von Blättern mit Stacks zur direkten Übertragung.
Links: Unmittelbar nach dem Ableiten von der Baumstruktur
Rechts: Nach dem Zusammenfassen und Entfernen redundanter Einträge (*merging*)

kopierten Datenmenge die Position innerhalb des Datentyps (d.h.seiner Darstellung in Form der Stackliste) gefunden werden, ab der mit dem Kopieren fortgefahren werden kann. Dazu wird zunächst derjenige Stack gesucht, der diese Position beinhaltet. Diese Suche gestaltet sich einfach, weil bei der Erstellung jedes Stacks die Gesamtmenge der Daten, die durch diesen Stack repräsentiert wird, im zugehörigen Listenelement gespeichert wird. Die Zahl der Stacks in der Liste entspricht der Zahl N_{bb} der *unterschiedlichen* Basisblöcke in diesem Datentyp und ist üblicherweise nicht groß (d.h. $N_{bb} < 10$); im Schnitt werden daher $\frac{1}{2} N_{bb}$ Vergleiche benötigt; der Zeitbedarf dazu ist entsprechend gering. Die exakte Bestimmung der Position innerhalb des Stacks erfordert für jede Ebene des Stacks vier arithmetische Ganzzahloperationen. Da die Zahl E_{stack} der Ebenen in einem Stack durch das *merging* üblicherweise ebenfalls klein ist (d.h. $E_{stack} < 5$), ist das Wiederaufsetzen eine Operation mit sehr kurzer Ausführungszeit, deren genaue Länge entsprechend der Anforderung nur von der Zahl der unterschiedlichen Basisblöcke, aber nicht von der Länge des Datentyps oder der Zahl von Wiederholungen, abhängt.

Stackkopie. Beim Kopieren eines Datentyps werden die in der Liste gespeicherten Stacks der Basisblöcke der Reihe nach abgearbeitet und die zugehörigen Basisblöcke in den Eingangspuffer des Empfängers übertragen. Beim Kopieren eines Stacks werden zwei wichtige Optimierungen durchgeführt:

- *Sammeln kleiner Blöcke*: Die Modellierung zur Leistung der direkten Übertragung nicht-

```
int pos, start, size_left, counter_array[];
stacklist_t *ff;

/* Wiederaufsetzen für fragmentierte Übertragung (rendez-vous) */
pos = find_pos (&start, &ff, counter_array);
if (start > 0)
     copy_splitted_block (ff, ff->total_size - start);

/* Kopieren vollständiger Stacks */
do {
     size_left -= copy_basic_block (&pos, ff);
     ff = ff->next;
} while (ff != NULL && size_left > 0);
```

Abbildung 5.8: Direkter Kopiervorgang (eines Fragments) eines nicht-zusammenhängenden Datentyps

zusammenhängender Daten hat gezeigt, daß diese erst ab einer bestimmten Blockgröße effizienter ist als die explizit packende Übertragung. Die Sortierung der Stackliste nach der Größe der Basisblöcke in abfallender Richtung erlaubt eine einfache Optimierung: sobald die Größe eines Basisblocks unter den Schwellwert nc_{min} fällt, wird ein temporärer Puffer allokiert, in den die verbleibenden Basisblöcke gepackt werden. Anschließend wird der gepackte Puffer kopiert. Der Schwellwert nc_{min} ist abhängig von dem Verhältnis der Latenzen der Speicherkopieroperationen in lokalen und in entfernten Speicher und ist daher einstellbar. Der Wert läßt sich in Abb. 5.5 als Schnittpunkt der Kurven für die direkte und explizit packende bestimmen.

- *Ausrichten von Blöcken*: Die Bandbreite von Speicherkopieroperationen in entfernten Speicher sinkt stark unter den Maximalwert ab, wenn die Zugriffe nicht auf Zieladressen einer bestimmten Ausrichtung erfolgen. Eine Darstellung dieses Effektes ist in Abb. 5.9 gegeben und zeigt ein Absinken der Bandbreite für die Übertragung von Blöcken identischer Größe um mehr als 80% für den Fall, daß die Zieladressen nicht optimal ausgerichtet sind. Die Ursache dieses Effekts liegt nicht in der Übertragung der Daten über das SCI-Netz: Die Schreibvorgänge sind ja, nach einer möglichen Fehlausrichtung zu Beginn, im Laufe der weiteren Übertragung kontinuierlich und aufsteigend. Vielmehr führt eine fehlende Ausrichtung zu ineffizienten Transaktionen sowohl auf dem Systembus als auch vor allem auf dem PCI-Bus, so daß die Daten zu langsam zum PSA gelangen. Für die in dieser Arbeit verwendete Hardwareplattform beträgt die Ausrichtung 32 Byte, was durch die Nutzung eines *write-combining*-Puffers (Datensammlung beim Schreiben) bedingt ist.
  Um diesen leistungsmindernden Effekt so stark wie möglich zu vermeiden, wird bei nicht ausgerichteten Zieladressen zunächst ein Teil eines Basisblocks kopiert, womit eine ausgerichtete Adresse erreicht wird. Der restliche Teil des Basisblocks wird direkt anschließend mit optimaler Ausrichtung kopiert, so daß die Gesamtlatenz für die Übertragung des Basisblocks (Mittel aus der Latenz des nicht-ausgerichteten und des ausgerichteten Teils der Übertragung in Abhängigkeit der Größe der beiden Teile des Basisblocks) deutlich über dem Minimalwert der nicht-ausgerichteten Übertragung liegen kann.

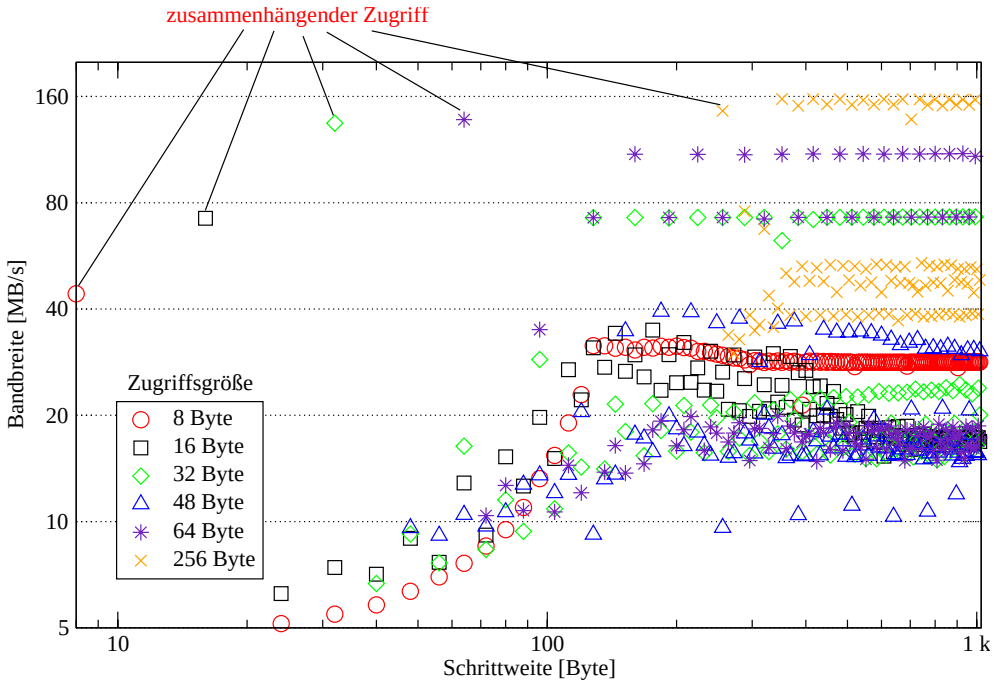


Abbildung 5.9: Abhängigkeit der reinen SCI-Übertragungsbandbreite von der Ausrichtung der Datenzugriffe (PIO-Schreibzugriffe auf entfernten Speicher).

5.1.4.3 Type Matching

Jeder abgeleitete Datentyp besteht letztendlich aus Basistypen (in diesem Zusammenhang können Basistypen und Basisblöcke äquivalent verwendet werden) und ist somit bezüglich der Verteilung der Daten im Speicher definiert durch die *Typabbildung (type map)*. Die Typabbildung ist die flache Darstellung des Datentyps, also die Sequenz von Paaren <*Basistyp, Versatz*>. Zusammen mit einer Startadresse ist damit die Abbildung einer Instanz des Datentyps im Speicher vollständig. Die *Typsignatur (type signature)* eines Datentyps ist die Sequenz der Typabbildung, jedoch ohne die Versatzinformation (also schlicht <*Basistyp*>). Beim expliziten Packen/Entpacken ist die Typsignatur des Datentyps im entpackten wie im gepackten Zustand identisch, da das Packen rein sequentiell entlang der Typabbildung erfolgt.

Durch den nicht strikt sequentiellen Zugriff auf die hintereinander liegenden Basisblöcke des nicht-zusammenhängenden Datentyps kommt es beim direkten Packen jedoch dazu, daß bei Datentypen, die aus verschiedenen Basisblöcken zusammengesetzt sind, diese im Eingangspuffer in einer anderen Reihenfolge vorliegen, als es durch die Datentypdefinition vorgegeben ist. Diese Konstellation ist in Abb. 5.10 beispielhaft für einen Datentyp aus *integer*- und *double*-Basisblöcken (bzw. Basistypen) dargestellt.

Dieser Effekt verändert die Typsignatur der gepackten Darstellung des Datentyps im Vergleich zum ungepackten Datentyp. Nach einem Aufruf von `MPI_Send` wird im Eingangspuffer des Empfängers also ein Datentyp mit veränderter Typsignatur abgelegt. Solange die gepackte Darstellung des Datentyps jedoch mit dem gleichen Datentyp oder mit einem Datentyp identischer Typsignatur wieder in die ungepackte Darstellung des Datentyps überführt wird (etwa bei einem Aufruf von `MPI_Recv`), entsteht kein Unterschied zum expliziten, lokalen Packen und Entpacken.

Falls jedoch der Empfänger davon ausgeht, daß die Typsignatur der Daten im Eingangspuffer unverändert ist zur Typsignatur des Datentyps in der Darstellung im Speicher des Senders und direkt mit den Basistypen in der Reihenfolge der angenommenen Typsignatur auf den Puffer (mittels einer Empfangsfunktion) zugreift, werden die so empfangenen Daten nicht mit der Darstellung beim Sender übereinstimmen.

Dies ist für abgeleitete Datentypen gemäß der *type matching* Regel des MPI-Standards zulässig. Eine praktische Anwendung, die eine derartige Vorgehensweise *erfordert*, ist jedoch schwer vorstellbar, und sie empfiehlt sich auch aus Effizienzgründen nicht: Ein nichtzusammenhängender Datentyp sollte immer so umfassend wie möglich definiert werden, um der MPI-Bibliothek (wie in diesem Fall) das maximale Optimierungspotential zu erhalten. Das beschriebene Problem läßt sich so ohne Verlust an Funktionalität umgehen. SCI-MPICH kann jedoch auch angewiesen werden, sich strikt MPI-konform zu verhalten. In diesem Fall werden Daten-

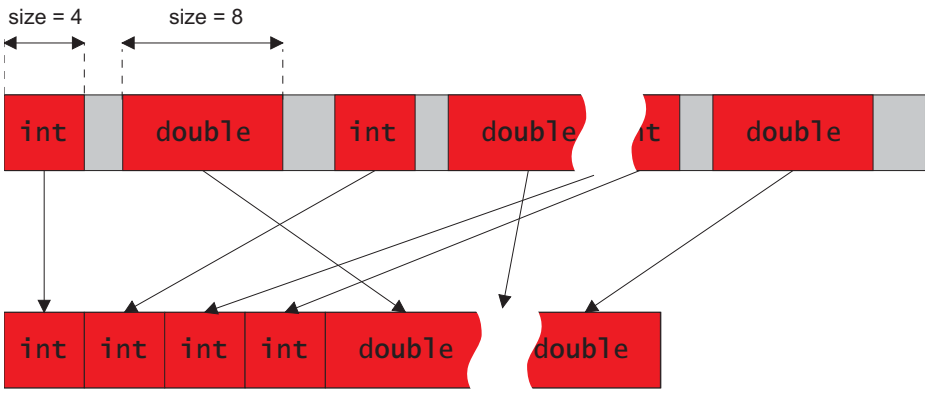


Abbildung 5.10: Änderung der Abfolge der Basisblöcke beim *direct_ff*-Kopieren vom Sendepuffer (*oben*) in den Eingangspuffer (*unten*).

typen, die mittels `MPI_Type_indexed`, `MPI_Type_hindexed` oder `MPI_Type_struct` definiert wurden, auf die herkömmliche Weise übertragen. Für Vektortypen (definiert mittels `MPI_Type_vector` oder `MPI_Type_hvector`) kann dieses Problem nicht auftreten, da hier nur gleichartige Basisblöcke aneinandergereiht werden.

5.1.5 Andere Arbeiten zu nicht-zusammenhängenden Datentypen

In der Literatur finden sich nur wenige Arbeiten, die die Nutzung von abgeleiteten, nicht-zusammenhängenden Datentypen behandeln. In *[138]* beschreiben Träff et. al. die Implementierung eines optimierten Packverfahrens insbesondere für die NEC SX-Serie von parallelen Vektorrechnern, die spezielle Hardwareunterstützung für derartige Speicherkopieroperationen hat. Auf dem dort beschriebenen Verfahren baut das in dieser Arbeit vorgestellte Verfahren auf. Jedoch wird dort nicht direkt in den Eingangspuffer des Empfängerprozesses gepackt, sondern weiterhin beim Sender gepackt, kontinuierlich versandt und beim Empfänger mit dem selben beschleunigten Verfahren entpackt. Der Packvorgang selbst wird jedoch durch die effiziente Nutzung der Hardware stark beschleunigt.

In *[137]* wird ein Verfahren vorgestellt, daß die Kommunikation mit Vektortypen beschleunigt, indem das Wiederholmuster in diesem Fall so gespeichert wird, daß kein ständiges traversieren des Typbaums erforderlich ist, sondern die Kopieroperationen unmittelbar innerhalb einer Schleife durchgeführt werden können. Diese Optimierung ist allerdings auf einfach verschachtelte Vektortypen beschränkt und bringt auch nur bei kleinen Basisblockgrößen einen Vorteil, wenn die Zeit zum Traversieren des Baumes hoch ist im Vergleich zur einzelnen Kopieroperation, die ebenfalls durch direkte Zuweisungen ersetzt wird, falls für die benötigte Blocklänge äquivalente vordefinierte Datentypen existieren. Das Verfahren ist in den aktuellen MPICH-Versionen, also auch in dem Standardverfahren von SCI-MPICH, integriert.

Ein Vergleich zwischen der Übertragung nicht-zusammenhängender Daten mit Hilfe von MPI Datentypen einerseits und mit Packen der Daten durch den Benutzer andererseits wurde in *[140]* für unterschiedliche Plattformen (Cray T3E, SGI Origin 2000 sowie Linux Cluster mit Fast Ethernet- und mit Myrinet-Kopplung) durchgeführt. Die Untersuchung zeigt, daß die Nutzung von MPI Datentypen jeweils etwas, aber nicht wesentlich höhere Leistung erbringt als das Packen der Datentypen durch den Nutzer. Allein der Vektor-Datentyp auf der Cray T3E und SGI Origin 2000 zeigt für bestimmte Längen der Elemente einen deutlichen Leistungsgewinn (vergl. auch Ergebnisse in Kapitel 5.1.6.2). Ansonsten zeigt die Untersuchung jedoch nur, daß die getesteten MPI-Implementationen gegenüber dem Packen durch den Benutzer keine Leistung verlieren. Davon sollte jedoch generell ausgegangen werden können. Da diese Untersuchung die Bandbreite für die Übertragung nicht-zusammenhängender Datentypen jedoch nicht ins Verhältnis setzt zur Übertragung der gleichen Datenmenge in Form eines zusammenhängenden Datentyps, sind die Ergebnisse nur bedingt zum Vergleich mit den Ergebnissen dieser Arbeit geeignet.

Insgesamt gibt es also wenige Aussagen zur Leistung der Kommunikation mit nicht-zusammenhängenden Datentypen. Daher ist bei den Programmierern von MPI-Applikationen oft Unwissen über den Einsatz von abgeleiteten Datentypen anzutreffen. Mitunter wurden auch schlechte Erfahrungen mit deren Leistungseigenschaften gemacht, was etwa durch *[136]* bestätigt wird. Dies führt dazu, daß abgeleitete Datentypen nicht verwendet werden, wenn es sinnvoll wäre.

5.1.6 Leistungsevaluierung

Für die vergleichende Leistungsevaluierung wurde eine Reihe von typischen Rechnerverbundsystemen untersucht, die auf unterschiedlicher Hardware mit jeweils vom Hersteller auf das Verbindungsnetz angepaßten MPI-Implementation operieren[1]. Dazu kommt ein etabliertes

Maschine	Verbindungsnetz	MPI-Implementation	Kennung
Cray T3E-1200	spezifisch	Cray MPI	C-C
Sun Fire 6800	Gigabit Ethernet	Sun HPC 3.1	F-G
24-fach SMP, 750 MHz, 64 bit PCI	shared memory		F-s
Pentium III Dual SMP	SCI LC-3	SCI-MPICH 1.3.0	M-S
800 MHz, 64 bit PCI	shared memory		M-s
Pentium III Xeon	fast ethernet	LAM 6.5.4	X-f
Quad SMP, 550 MHz	shared memory		X-s
Pentium II Dual SMP	Myrinet 1280	SCore 2.4.1	S-M
400 MHz, 32 bit PCI	shared memory		S-s

Tabelle 5.1: Übersicht der evaluierten Rechnerverbundsysteme / MPP-Systeme

massiv-paralleles Systems, daß bezüglich der Hardware-Architektur jedoch in vielen Aspekten Ähnlichkeit mit modernen Hochleistungs-Rechnerverbundsystemen hat. Eine Aufstellung der untersuchten Systeme findet sich in Tabelle 5.1. Zur einfacheren Referenzierung und besseren Übersicht in den Diagrammen dieses Kapitels wurden die Systeme jeweils mit einer Kennung nach dem Schema *Rechenknoten-Verbindungsnetz* versehen.

Das offensichtliche Ziel des vorgestellten Verfahrens ist die Maximierung der Kommunikationsbandbreite. Jedoch hat das Verfahren einen weiteren Einfluß auf die Applikationsleistung, wie sich in Kapitel 5.1.6.3 zeigt.

5.1.6.1 Einfluß auf die Übertragungsbandbreite

Zunächst wird die Implementation des *direct_ff* Verfahrens für SCI mittels der erreichten Ping-Pong-Bandbreite für verschiedene nicht-zusammenhängende Datentypen evaluiert. Dazu sind in Abb. 5.11 die Ergebnisse für einen einfach nicht-zusammenhängenden Vektor mit Blocklängen von $D_{block} = 2^n$ Byte (*links*) und für einen komplexeren Vektor mit Blocklängen von $D_{block} = 3^n$ und $D_{block} = 7 \cdot 3^n$ Byte (*rechts*) dargestellt.

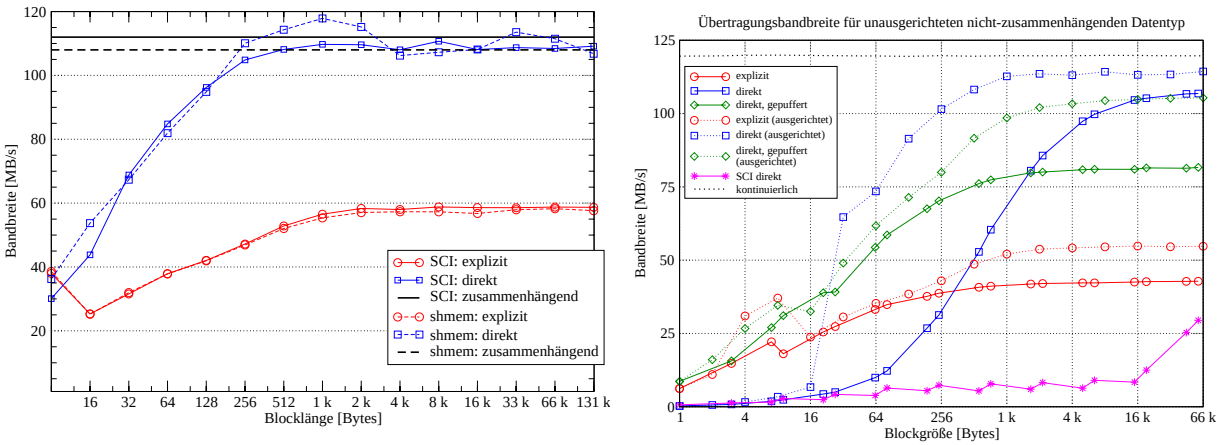


Abbildung 5.11: Übertragung nicht-zusammenhängender Datentypen mit explizitem und direktem Verfahren.
Links: einfach nicht-zusammenhängender Vektor (via SCI und gemeinsamen Speicher)
Rechts: komplexer, nicht ausgerichteter nicht-zusammenhängender Datentyp (via SCI)

1. Für TCP/IP-Kommunikation ist eine derartige Optimierung i.d.R. nicht auf ein spezielles System beschränkt, sondern allgemein für alle TCP/IP-Protokollstacks gültig, die die entsprechenden Verfahren bzw. Optionen unterstützen.

Bei der Übertragung des einfach nicht-zusammenhängenden Vektors (*links*) erreicht das direkte Verfahren für $D_{block} > 8$ eine höhere Bandbreite als das explizite Verfahren. Dieses enthält bereits ein optimiertes Pack-Verfahren, daß jedoch nur auf Vektoren anwendbar ist *[137]*, das für $D_{block} = 4$ und $D_{block} = 8$ die Blöcke mittels direkter Zuweisungen anstelle von Kopieroperationen packt. Für diese kleinen Blocklängen erreicht das direkte Verfahren über SCI durch die vergleichsweise hohe Latenz nur 75% der Leistung des optimierten expliziten Verfahrens. Bei der Intra-Knoten-Kommunikation über lokalen gemeinsamen Speicher (*shmem*) sind beide Verfahren für $D_{block} = 8$ gleichwertig. Für $D_{block} > 8$ ist das direkte Verfahren deutlich schneller und erreicht für $D_{block} > 256$ mehr als 95% der Bandbreite für die Übertragung der gleichen Datenmenge in einem zusammenhängenden Datentyp. Bei der Intra-Knoten-Kommunikation übertrifft die Übertragungsbreite des nicht-zusammenhängenden Vektors sogar die Vergleichsbandbreite des zusammenhängenden Datentyps. Dieser Effekt ist auch auf anderen Plattformen (z.B. SCI-MPICH auf Sun Fire) zu beobachten. Die Ursache dafür muß in den kürzeren Blocklängen zu suchen sein, die in den einzelnen Kopieroperationen verwendet werden, wodurch die Cachenutzung verbessert wird. Indiz dafür ist, daß der Effekt jeweils bei $D_{block} > 2^{12}$ und $D_{block} > 2^{18}$ nicht mehr auftritt, was auf der P3-Plattform jeweils der halben Größe des Level-1- bzw. Level-2-Caches entspricht.

Die Übertragungsbandbreite für einen nicht an 8-Byte-Grenzen ausgerichteten nicht-zusammenhängenden Datentyps unterscheidet sich deutlich von der eines entsprechend ausgerichteten Datentyps, wie in Abb. 5.11 (*rechts)* sichtbar wird. Insbesondere wird im Vergleich (durchgezogene vs. gepunktete Linien) zweierlei deutlich:

- Insgesamt erreicht die Bandbreite für diesen Datentyp nur etwa $\frac{2}{3}$ (direktes Verfahren) bis $\frac{3}{4}$ (explizites Verfahren) der Vergleichsbandbreite für einen ausgerichteten Datentyp.
- Die einfache Optimierung für Vektoren gemäß *[137]* zeigt einen deutlich schwächeren Effekt, da die Zuweisungs- bzw. Kopier-Operationen für $D_{block} = 7, 9, \ldots$ deutlich höhere Latenzen aufweisen.
- Die Zwischenpufferung beim direkten Verfahren ist elementar, um eine maximale Übertragungsbandbreite zu erreichen. Diese liegt für $D_{block} > 4$ über der Leistung des expliziten Verfahrens und für $D_{block} > 128$ beim doppelten Wert. Erst für $D_{block} > 2^{10}$ übersteigt der Aufwand für die Zwischenpufferung den Gewinn durch die erzielte höhere Kopierbandbreite, und das ungepufferte direkte Verfahren zeigt die beste Leistung.

5.1.6.2 Effizienzvergleich existierender Implementationen

Um die Eigenschaften verschiedener MPI-Implementationen, zusammen mit dem verwendeten Kommunikationsmedium, zu beurteilen, ist der Vergleich der erzielten absoluten Bandbreite nicht aufschlußreich. Das geeignete Beurteilungskriterium ist vielmehr die Effizienz ε_{nz} der Übertragung nicht-zusammenhängender Datentypen, bezogen auf die Übertragung eines zusammenhängenden Datentyps mit der gleichen Datenmenge *D*:

$$(5.4) \quad \varepsilon_{nz} = \frac{B_{nz}(D)}{B(D)}$$

Diese Metrik erlaubt Vergleiche zwischen Systemen mit unterschiedlichen Parametern bezüglich der Kommunikations- und Speicherleistung.

Natürlich kann auch mit dem herkömmlichen Verfahren des expliziten Packens/Entpackens eine hohe Effizienz erreicht werden, wenn das Speichersystem relativ viel leistungsfähiger ist als das Kommunikationssystem. Dies ist etwa bei einem Rechnerverbundsystem aus modernen Rechenknoten, die über Fast-Ethernet gekoppelt sind, der Fall. Dies muß bei der Bewertung berücksichtigt werden.

Die Systeme aus Tabelle 5.1 wurden mit einem einfach nicht-zusammenhängenden Vektor ge-

testet. Dies ist der einfachste nicht-zusammenhängende Datentyp: Wenn eine MPI-Implementation überhaupt Optimierungen für nicht-zusammenhängende Datentypen hat, sollten sie hier am deutlichsten sichtbar werden. Ansonsten ist davon auszugehen, daß auch komplexere Datentypen nicht optimiert sind. In Abb. 5.12 sind die absoluten Bandbreiten (*oben*) und die daraus bestimmte Effizienz ε_{nc} (*unten*) aufgetragen.

Die Ergebnisse erlauben aufgrund der Differenz der Bandbreiten für die Übertragung der gleichen Menge Daten in zusammenhängender und nicht-zusammenhängender Darstellung den Schluß, daß keines der betrachteten Systeme eine effektive und flexible Optimierung der Übertragung nicht-zusammenhängender Datentypen bietet.

Das Cray-System kann die Effizienz für $D_{block} < 2^{17}$ auf Werte etwas über 100% steigern, um jedoch für $D_{block} > 2^{17}$ wieder unter 50% Effizienz aufzuweisen. Ebenso ist die Effizienz für $D_{block} = 8$ um Größenordnungen höher als die für $D_{block} = 16$. Hier ist fraglich, warum die offensichtlich in Grundzügen vorhandene Optimierung nicht für alle Blocklängen umgesetzt wurde. Ein ähnliches Verhalten zeigt die Sun Fire-Plattform für die Kommunikation über gemeinsamen Speicher: beim Übergang $D_{block} = 2^{13}$ auf $D_{block} = 2^{14}$ steigt die Effizienz von zuvor konstanten 50% auf 100%.

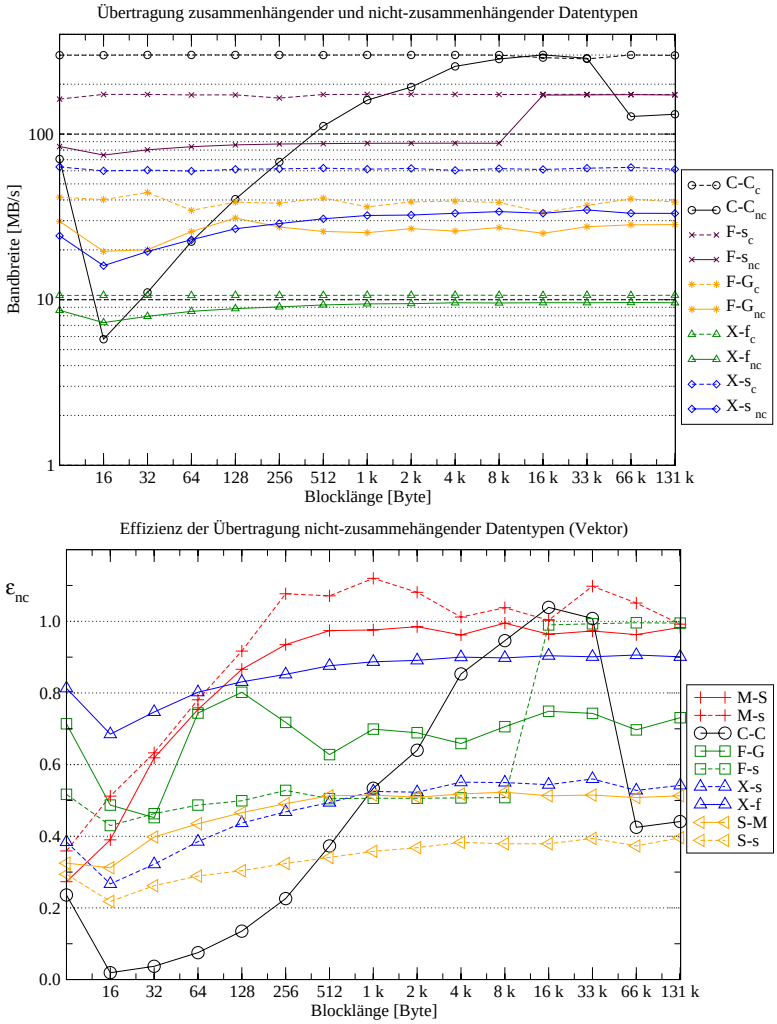


Abbildung 5.12: Übertragung nicht-zusammenhängender Datentypen (hier: einfach nicht-zusammenhängender Vektor) auf einer Auswahl von Systemen und MPI-Implementationen (Zuordnung der Kürzel in der Legende s. Tabelle 5.1).
Oben: Bandbreite (Index c: zusammenhängend, Index nc: nicht-zusammenhängend)
Unten: Effizienz (inkl. SCI-MPICH)

Die Effizienz des *direct_ff* Verfahrens via SCI liegt für $D_{block} > 64$ über allen anderen Vergleichswerten. Einzig bei sehr kleinen Blocklängen liegt die Effizienz von Systemen mit vergleichsweise langsamem Verbindungsnetz (Fast Ethernet) und mit leistungsfähigem gemeinsamem Speicher (Sun Fire) erwartungsgemäß darüber.

5.1.6.3 Steigerung der räumlichen Lokalität

Beim expliziten Packen und Entpacken ist es sowohl beim Sender als auch beim Empfänger erforderlich, einen temporären Puffer zu allokieren, der die zu übertragenden Daten zusammenhängend (gepackt) aufnehmen kann (siehe Abb. 5.4). Dies vergrößert generell den Speicherbereich, der für die Abarbeitung der Applikation benötigt wird und verringert gleichzeitig die räumliche Lokalität der Zugriffe. Dies bedingt eine geringere Trefferquote in den Prozessorcaches, was allgemein eine Verringerung der Applikationsleistung unabhängig von der eigentlich Datenübertragung (und den dabei übertragenen Daten) bedingen kann. Am deutlichsten wird dieser Effekt, wenn direkt nach dem Versand von Daten, die durch einen nicht-zusammenhängenden Datentyp beschrieben werden, auf diese Daten zugegriffen wird. Zur Quantifizierung dieses Effekts wurde ein synthetischer Benchmark *nc_cache* entwickelt, der die Bandbreite beim Zugriff auf zuvor versendete nicht-kontinuierliche Daten (Vektor) testet. Das Ergebnis dieses Benchmarks für unterschiedliche Puffergrößen von 0 bis 590kB (entspricht der Datenmenge des Vektors) ist in Abb. 5.13 dargestellt. Deutlich ist für alle Kurven zu sehen, daß ab einer bestimmten Puffergröße der Zugriff auf die enthaltenen Daten stark verlangsamt wird, da sie nicht mehr vollständig im Cache gehalten werden können. Dieser Effekt setzt für das explizite Packen jedoch bereits bei einer Puffergröße entsprechend der halben Cachegröße (128 kB) ein, da ein ebenso großer Puffer für die gepackten Daten benötigt wird. Beim direkten Packen hingegen steht die gesamte Cachegröße für die Benutzerdaten zur Verfügung; der Rückgang der Zugriffsbandbreite setzt erst ab einer Puffergröße von 256 kB ein. Die gepufferte Variante des direkten Packens hat einen etwas höheren lokalen Speicherverbrauch (zugunsten der höheren Kommunikationsbandbreite), der einen Rückgang der Zugriffsbandbreite bereits ab 200 kB Puffergröße bewirkt.

5.1.6.4 Kommunikation und Nutzung von Matrizenausschnitten

In einem anwendungsnahen Beispiel können die Auswirkungen der beschriebenen Effekte auf die Kommunikationsbandbreite und die räumliche Lokalität der Speicherzugriffe in Kombination beurteilt werden. Daher wurden zwei Benchmarks *column* und *triangle* entwickelt, die die Leistung beim Versand und Zugriff auf Teile einer Matrix untersuchen.

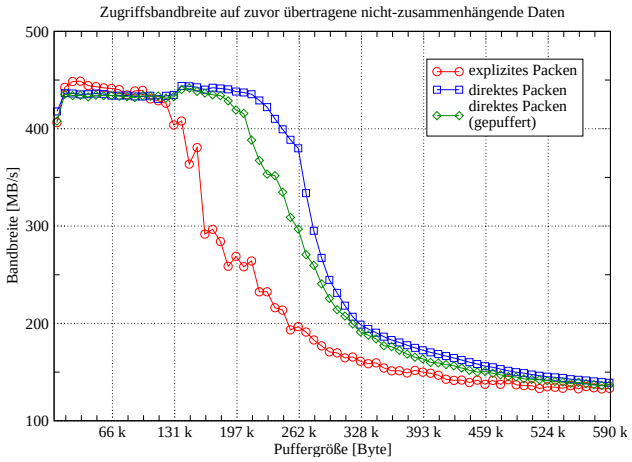


Abbildung 5.13: Quantifizierung der Cacheverschmutzung (*nc_cache* Benchmark): Zugriffsbandbreite auf zuvor übertragene nicht-zusammenhängende Daten mit explizitem Packen sowie direktem und direktem-gepuffertem Packen.

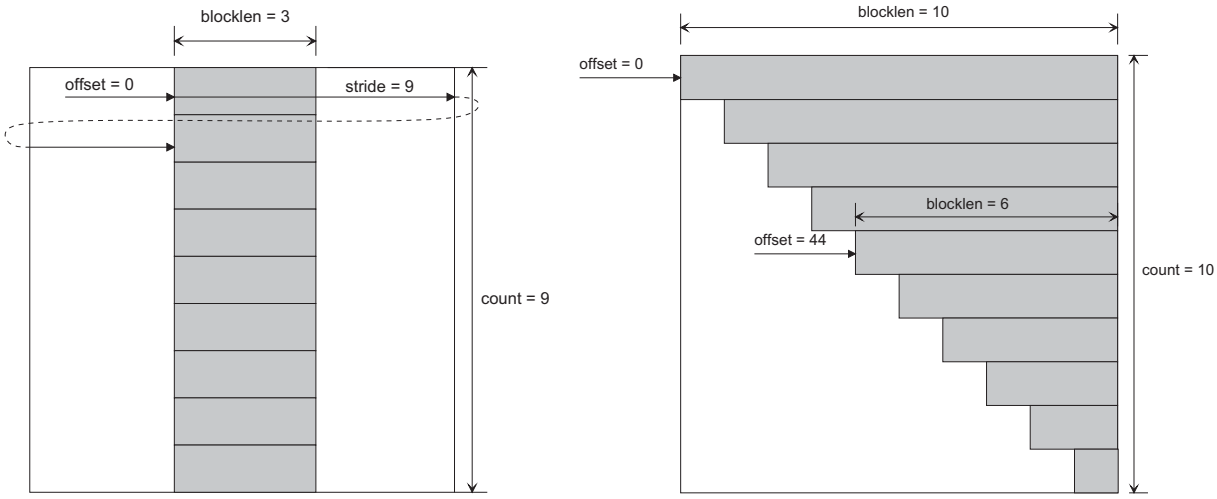


Abbildung 5.14: Benchmarks zum Zugriff und Versand von nicht-zusammenhängenden Datentypen
Links: column, Beschreibung von Spalten einer Matrix
Rechts: triangle, Beschreibung einer oberen Dreiecksmatrix

column. Der Benchmark *column* (siehe Abb. 5.14) definiert eine Zahl von Spalten einer Matrix als einen Datentyp. Da in der verwendeten Programmiersprache C Matrizen zeilenweise im Speicher abgelegt werden, ist dieser Datentyp nicht-zusammenhängend und kann als *vector* beschrieben werden. Der Benchmark kann mit $N > 1$ Prozessen ausgeführt werden. Die Matrix der Dimension *M* wird in *P* Blöcke mit jeweils *M/N* Spalten aufteilt. Prozeß 0 skaliert die gesamte Matrix und verschickt die Spaltenblöcke anschließend an alle anderen Prozesse[1]. Ein Beispiel für diesen Datentyp ist in Abb. 5.14 (*links*) für $M = 9$ und $N = 3$ dargestellt.

triangle. Der Benchmark *triangle* verwendet einen Datentyp, der eine obere Dreiecksmatrix beschreibt, und besteht daraus, wiederholt die Elemente dieser oberen Dreiecksmatrix mit einem Skalar zu verknüpfen und anschließend die obere Dreiecksmatrix zu versenden. Das Abbild dieses *hindexed*-Datentyps im Speicher für eine Matrix der Dimension 10×10 ist ebenfalls mit den für die Definition des Datentyps relevanten Parametern *blocklen* (Länge der einzelnen Blöcke), *offset* (Lage der Blöcke relativ zum Beginn der Instanz des Datentyps) und *count* (Zahl der Blöcke im Datentyp) in Abb. 5.14 (*rechts*) abgebildet.

Gemessen wird in beiden Benchmarks die Zeit, die in Abhängigkeit des Sendemodus (explizit oder direkt gepackt) und der Matrixdimension *M* für einen Versende- und Skalierungsvorgang benötigt wird. Dabei ist t_{ss_d} die Zeit, die bei direktem Packen benötigt wird, während t_{ss_e} entsprechend die benötigte Zeitdauer für explizites Packen/Entpacken darstellt. t_{ss} wird sowohl von der erzielten Übertragungsbandbreite für den Sendevorgang als auch von der Zugriffszeit auf die Elemente der Matrix bei der Skalierung bestimmt; die einzelnen Anteile lassen sich durch Bestimmung der Zeitdauern t_{scale} für die Skalierung und t_{send} für den Sendevorgang bestimmen. Entscheidend für die Applikationsleistung ist jedoch die angestrebte Verkürzung des Gesamtvorgangs, der entsprechend eines Speedups als Verhältnis der beiden Zeiten bestimmt werden kann. Beide Zeiten werden aufgrund der Abhängigkeit der Trefferquote im Cache und der Größe der zu übertragenden Basisblöcke von der Matrixdimension unterschiedlich mit *M* wachsen, so daß sich ein nicht-konstantes Verhältnis in Abhängigkeit von *M* ergibt. Dieses Verhältnis ist in (5.5) dargestellt.

1. In realen Applikationen würde stattdessen eher ein Alle-an-Alle-Austausch stattfinden, der zusätzlich das Netz maximal belastet. Die Auswirkungen der direkten Kommunikation nicht-zusammenhängender Datentypen sollen jedoch unabhängig vom Verhalten des Kommunikationsnetzes untersucht werden.

$$(5.5)\quad S_{ss}(M) = \frac{t_{ss_e}(M)}{t_{ss_d}(M)}$$

Die Ergebnisse für den *column*-Benchmark sind in Abb. 5.15 dargestellt. Der Test wurde in zwei Variationen durchgeführt: einerseits wurde die Dimension der Matrix konstant gehalten, und die Breite der kommunizierten und skalierten Spalte variiert (Fall *S*); andererseits wurde die Matrixdimension variiert, und die Spaltenbreite entsprach jeweils der Hälfte der Matrixdimension (Fall *D*).

Für beide Fälle ist der Effekt zu beobachten, daß die Sendebandbreite für das direkte Verfahren stark von der zusammenhängenden Blocklänge abhängt: Die maximale Bandbreite wird erreicht, wenn die Blocklänge ein Vielfaches von 32 darstellt. Ansonsten sinkt die Bandbreite durch das notwendige Zwischenpuffern (zur Vermeidung unausgerichteter *SCI*-Schreibvorgänge) um bis zu 40% ab. Für die optimalen Blocklängen liegt die Bandbreite für das direkte Verfahren ab einer Spaltenbreite von 8 über der Bandbreite des expliziten Verfahrens; ab einer Spaltenbreite von 112 ist dies für alle Blocklängen der Fall.

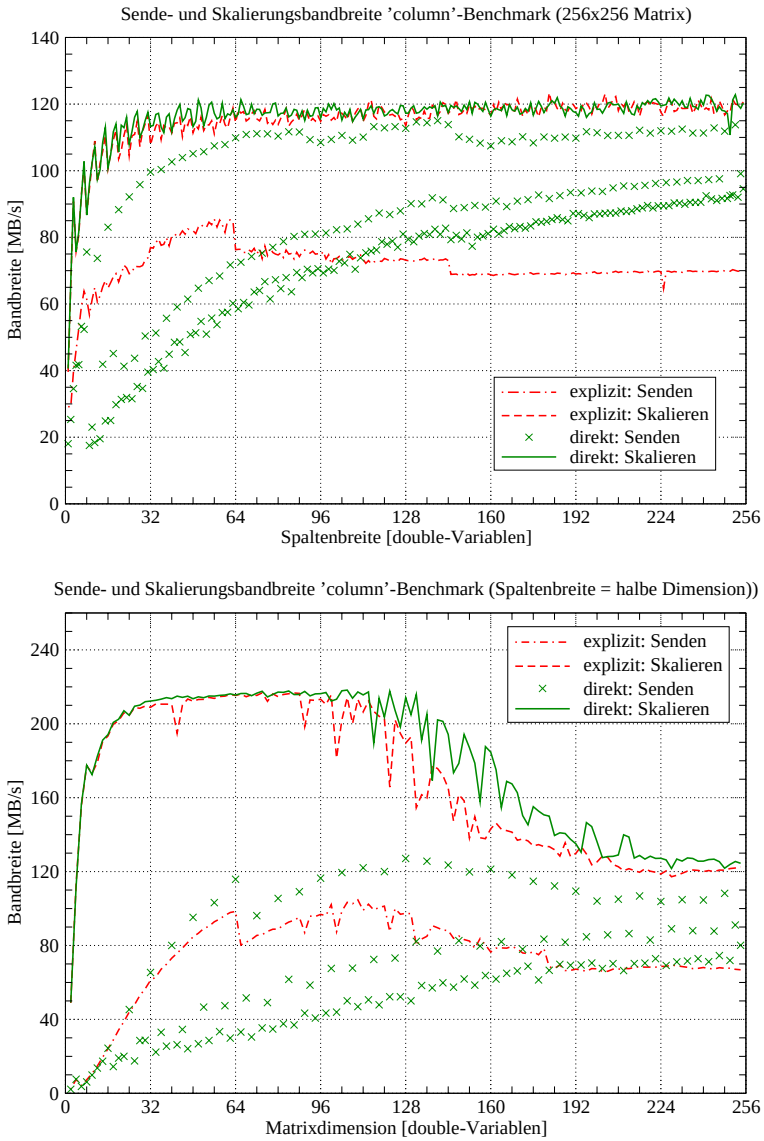


Abbildung 5.15: Skalierungs- und Sende-Bandbreite beim *column*-Benchmark
Oben: Feste Matrixdimension, variable Spaltenbreite (Fall *S*)
Unten: Variable Matrixdimension, proportionale Spaltenbreite (Fall *D*, 50% Matrixdimension)

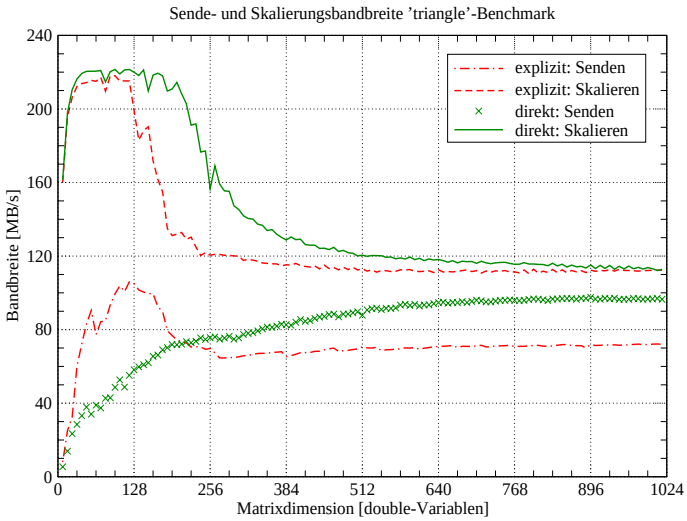


Abbildung 5.16: Übertragungs- und Skalierungsbandbreite für den *triangle*-Benchmark (Fall *T*).

Die beiden untersuchten Fälle unterscheiden sich vornehmlich bei der erreichten Skalierungsbandbreite. In Fall *S* erreichen beide Verfahren die gleiche Skalierungsbandbreite, obwohl das *working set* für die maximale Dimension eine Größe von 256 kB hat. Beim expliziten Verfahren sollte die Skalierungsbandbreite für ein working set oberhalb der halben Cachegröße zurückgehen (siehe Kapitel 5.1.6.3). Dieser Effekt tritt jedoch nur für den Fall *D* auf: hier entsteht ein Bereich, in dem die Skalierungsbandbreite beim direkten Verfahren um bis zu 40% höher liegt als beim expliziten Verfahren. Aus dem Vergleich der Skalierungsbandbreite der beiden Fälle wird ersichtlich, daß die durchgehend geringere Skalierungsbandbreite in Fall *S* die Ursache in der größeren Matrixdimension hat: Bei gleicher Größe des working set liegen die Daten im Speicher räumlich stärker verteilt. Obwohl die einzelnen Blocklängen nicht unterhalb der Cachezeilenlänge liegen, tritt dieser Effekt auch auf anderen Testplattformen auf. Neben der Größe der zu versendenden Datenblöcke ist für den durch das direkte Übertragungsverfahren zu erwartenden Gewinn auch die Lage der Daten im Speicher zu berücksichtigen. Fall *D* zeigt jedoch, daß die Erhöhung der räumlichen Lokalität durch das direkte Verfahren in Anwendungsszenarien zur Geltung kommt, bei denen die Daten so angeordnet sind, daß der Cache überhaupt wirksam werden kann.

Während beim *column*-Benchmark der sehr einfache Vektor-Datentyp zum Einsatz kam, ist der Datentyp beim *triangle*-Benchmark sehr komplex, da jedes Element eine andere Länge und explizite Plazierung besitzt. Der Vektor-Datentyp läßt sich daher beim direkt-Verfahren äußerst kompakt darstellen, während der beim *triangle*-Benchmark verwendete *hindexed*-Datentyp in der Stackliste einen separaten Eintrag pro Element benötigt. Daher ist interessant, wie sich dies auf die Leistung der beiden Übertragungsverfahren auswirkt. Die Ergebnisse des *triangle*-Benchmarks (Fall *T*) sind in Abb. 5.16 dargestellt.

Tatsächlich erreicht die Übertragungsbandbreite für das direkte Verfahren mit maximal 100 MB/s nicht die Maximalwerte der Fälle *D* und *S* von etwa 120 MB/s, während das explizite Verfahren in allen Fällen maximal etwa 70 MB/s erreicht. Für Matrixdimensionen bis 256 liegt die Übertragungsleistung des expliziten Verfahrens über der des direkten Verfahrens, da hier beim Packen der Daten der Cache nützlich ist, während beim direkten Verfahren ein relativ großer Anteil nicht optimal ausgerichteter Daten die Übertragungsleistung reduziert. Dieser Vorteil des expliziten Verfahrens wird jedoch zum Vorteil des direkten Verfahrens, sobald die Datenmenge steigt: Die Skalierungsbandbreite bleibt länger auf dem hohen Niveau, und auch die Übertragungsbandbreite des direkten Verfahrens übertrifft die des expliziten Verfahrens.

Der unmittelbare Vergleich der Leistung der beiden Verfahren ist in Abb. 5.17 gegeben, ausgedrückt als die Effizienz gemäß (5.5) für die Fälle *D* und *T*. Neben der Effizienz für den Gesamtvorgang sind auch die entsprechenden Werte für die Effizienz der Einzelvorgänge, Senden und Skalieren, angegeben.

Der Vergleich zeigt, daß das explizite Verfahren für die Fälle, in denen die Datenmenge signifikant wird, dem herkömmlichen Verfahren in der Gesamtleistung um etwa 20% überlegen ist. Dies gilt gleichermaßen für einfache wie auch komplexe Datentypen. Falls bei einfachen Datentypen auf eine Ausrichtung der Daten geachtet wird, kann die Leistungssteigerung noch höher liegen; in den untersuchten Fällen bis zu 40%.

5.1.6.5 Leistung bei Intra-Knoten-Kommunikation

Das vorgestellte Verfahren für die Inter-Knoten-Kommunikation via SCI ist direkt übertragbar auf die Intra-Knoten-Kommunikation über gemeinsamen Speicher. Durch den direkten Zugriff auf den gemeinsamen lokalen Speicher über den Systembus (anstelle über den PCI-Bus) entsteht eine von der Ausrichtung der Daten unabhängige Zugriffscharakteristik. Dies wird in den Ergebnissen des *column*- und *triange*-Benchmarks für die Intra-Knoten-Kommunikation auf der P3-Plattform deutlich (Abb. 5.18).

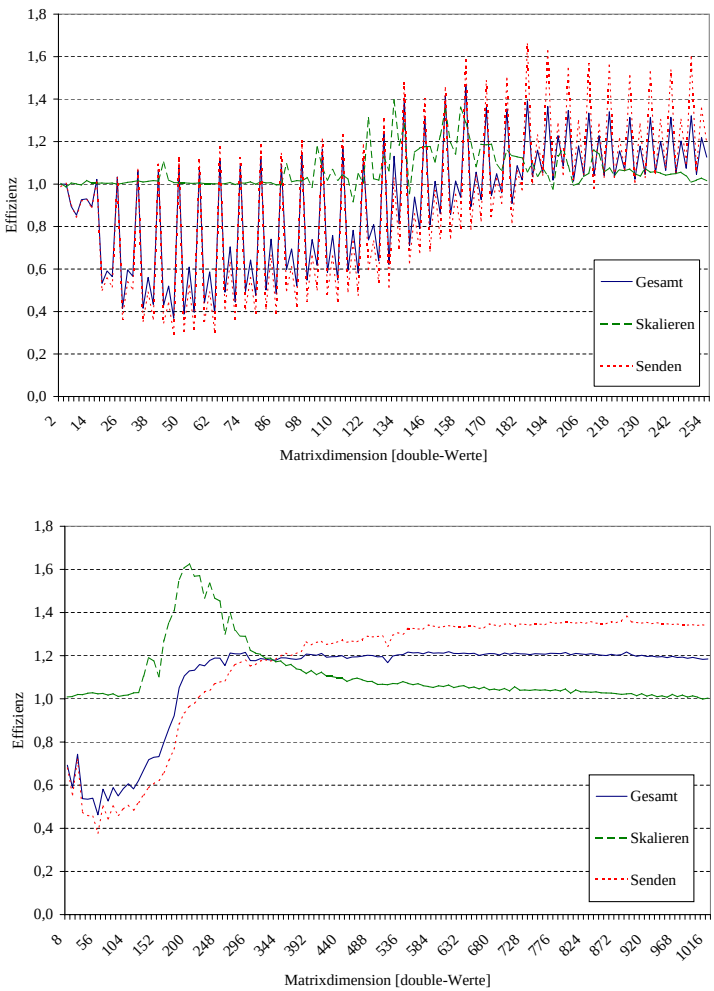


Abbildung 5.17: Relative Effizienz des direkten Verfahrens gegenüber dem expliziten Verfahren zur Übertragung und Skalierung nicht-zusammenhängender Datentypen via SCI
Oben: Fall D; *Unten:* Fall T

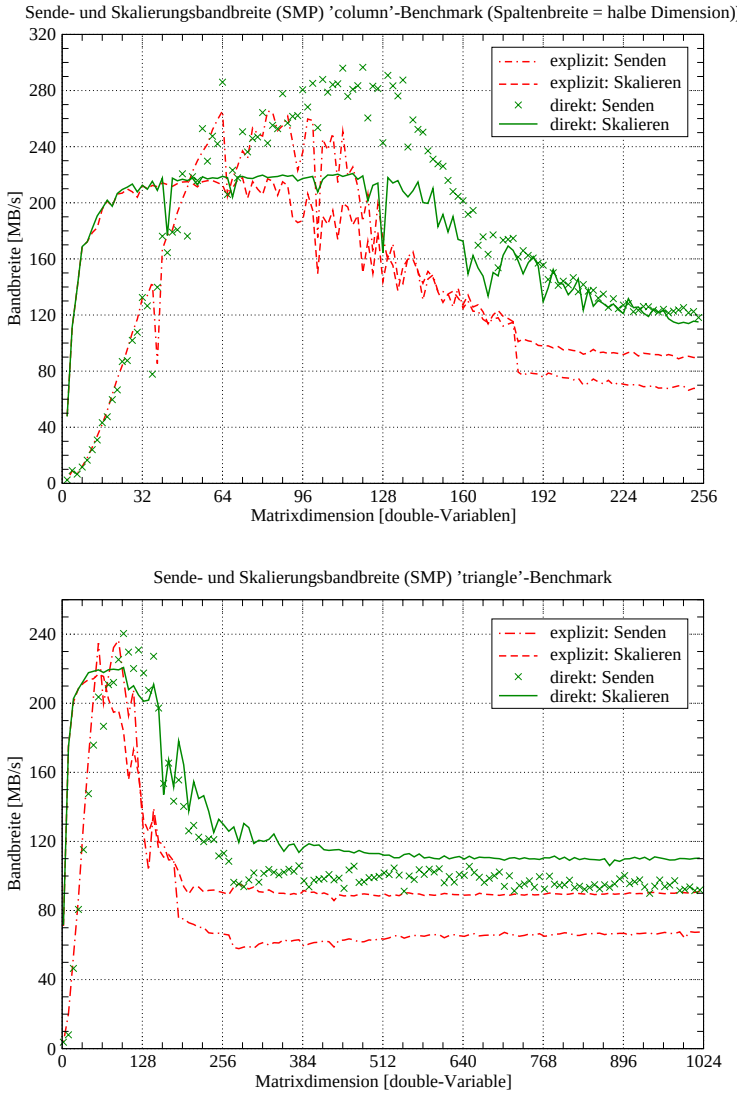


Abbildung 5.18: Skalierungs- und Sende-Bandbreite von nicht-zusammenhängenden Datentypen für Intra-Knoten-Kommunikation
Oben: column-Benchmark mit variabler Matrixdimension
Unten: triangle-Benchmark mit variabler Matrixdimension

Der Vergleich der Ergebnisse für die Kommunikation über lokalen gemeinsamen Speicher mit denen für die Kommunikation über SCI zeigt mehrere Unterschiede. Zunächst ist festzustellen, daß aufgrund der weitaus geringeren Kommunikationslatenz das direkte Verfahren dem expliziten Verfahren auch bei kleinen oder (für Kommunikation via SCI) ungünstigen Blockgrößen überlegen ist. Ein qualitativer Unterschied ist darin erkennbar, daß die Skalierungsbandbreite für das direkte Verfahren auch dann über derjenigen des expliziten Verfahrens liegt, wenn die zu transportierende Datenmenge die Cachegröße übersteigt (jeweils für Matrixdimensionen > 128). Dieser Effekt ist darauf zurückzuführen, daß der Eingangspuffer des empfangenden Prozesses für Intra-Knoten-Kommunikation in einem Speicherbereich liegt, der vom Cache mittels *write-back*-Verfahren beim Schreiben gepuffert wird. Im Gegensatz dazu liegt der Eingangspuffer für Inter-Knoten-Kommunikation via SCI in einem *write-combining*-Speicherbereich, wodurch bei Schreibvorgängen längere PCI-Übertragungsblöcke (*bursts*, siehe Kapitel 2.1.3) erzeugt werden. Jedoch geschehen Schreibvorgänge in derartige Speicherbereiche am Cache vorbei.

Diese Versuche machen deutlich, daß Optimierungen im Nachrichtenaustausch für PIO-Übertragungen via SCI ebenso Optimierungen für Nachrichtenaustausch über lokalen gemeinsamen Speicher mit erhöhter Leistung sind.

5.2 Einseitige Kommunikation

Gemäß dem MPI-1 Standard erfordert jeder Punkt-zu-Punkt Kommunikationsvorgang eine Sendeoperation eines Prozesses und eine dazu passende Empfangsoperation eines weiteren (auch desselben) Prozesses. Dieses Modell wird als *zweiseitige Kommunikation* bezeichnet. Der MPI-2 Standard definiert neben diesem Modell die *einseitige Kommunikation.* Hierbei kann ein Kommunikationsvorgang allein mittels der Angaben, die ein einzelner Prozeß spezifiziert, abgeschlossen werden. Dieses Modell ähnelt der Kommunikation über gemeinsamen Speicher und hat gegenüber der zweiseitigen Kommunikation den Vorteil, daß es potentiell auftretende Wartezeiten eines Prozesses, der Daten eines anderen Prozesses benötigt, vermeidet, indem er alleine in der Lage ist, sich die erforderlichen Daten ohne aktive Mitwirkung des anderen Prozesses zu beschaffen. In Abb. 5.19 ist dargestellt, wie Prozeß B mittels herkömmlicher (zweiseitiger) und mittels einseitiger Kommunikation an Daten aus dem Adreßraum von Prozeß A gelangt. Bei der Nutzung von zweiseitiger Kommunikation muß B eine blockierende MPI-Emp-

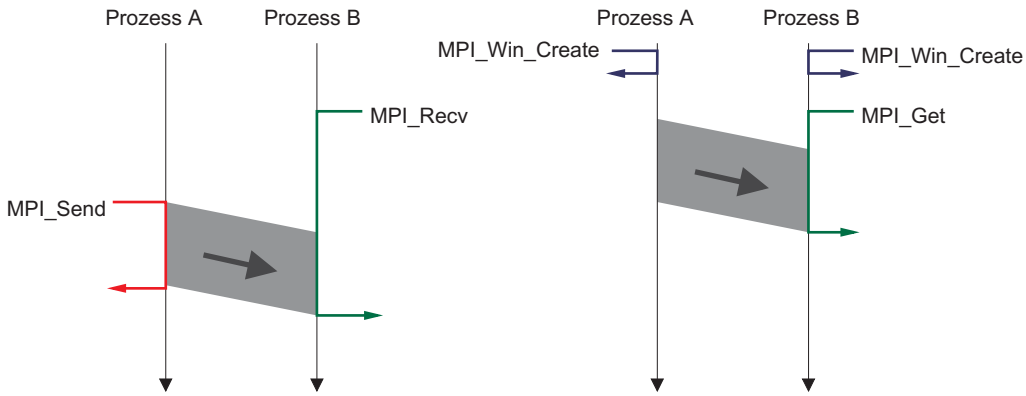


Abbildung 5.19: Prinzip der zweiseitigen (links) und einseitigen (rechts) Kommunikation in MPI

fangsfunktion (`MPI_Recv`) aufrufen, nach deren Rückkehr die Daten lokal vorliegen. Dazu ist es jedoch erforderlich, daß A eine passende Sendeoperation (im Beispiel `MPI_Send`) auslöst. A muß also wissen, daß B Daten erwartet (und dazu auch, *welche* Daten). Bei einseitiger Kommunikation verständigen sich A und B vorab mittels `MPI_Win_create` über Bereiche ihres lokalen Adreßraums, auf den der jeweils andere Prozeß Zugriff erhalten soll (sogenannte *Windows*). Anschließend können die einseitigen Kommunikationsoperationen `MPI_Get`, `MPI_Put` und `MPI_Accumulate` genutzt werden, um ohne explizite Mitwirkung des Partners Daten aus dessen Window zu lesen, dort Daten zu schreiben oder dort vorhandene Daten mit eigenen Daten zu verknüpfen. Dabei ist der Prozeß, der die Zugriffsfunktion aufruft, der *Ursprungsprozeß* (UP, *origin process*), während der Prozeß, auf dessen im Window festgelegter lokaler Speicher der Zugriff durchgeführt wird, als *Zielprozeß* (ZP, *target process*) bezeichnet wird.

Alle Datenzugriffe auf Windows müssen beim UP innerhalb einer *Zugriffsepoche* (*access epoch)* erfolgen. Die Zugriffsepoche wird eröffnet und abgeschlossen durch zueinander passende Synchronisationsaufrufe. Die Datenzugriffe innerhalb einer Zugriffsepoche sind erst dann abgeschlossen, wenn die Zugriffsepoche abgeschlossen ist. Das heißt, daß die Empfangs- bzw. Sendepuffer für die Datenzugriffe die angeforderten Daten noch nicht enthalten bzw. die zu schreibenden Daten nicht verändert werden dürfen, bis die abschließende Synchronisation der Zugriffsepoche erfolgt ist. Es liegt also keine sequentielle, sondern eine gelockerte Konsistenz vor: Obwohl die Kommunikationsaufrufe blockierend ausgeführt sind, liegt das Ergebnis der Operation nicht unmittelbar nach deren Abschluß vor. Die Reihenfolge der Aufrufe bleibt jedoch für das Ergebnis bestimmend. Auf die zugehörigen Synchronisationsaufrufe wird in Kapitel 5.2.2 eingegangen.

5.2.1 Benachrichtigungs- und Übertragungsmöglichkeiten in SCI-MPICH

SCI bietet durch das Prinzip der Speicherkopplung gute Vorraussetzungen, die beschriebene einseitige Kommunikation effizient zu implementieren. Ein Window, das auf einem SCI-Speichersegment basiert, kann potentiell von jedem anderen Prozeß im System transparent beschrieben und ausgelesen werden. Diesem einfachen Lösungsansatz des *direkten Zugriffs* stehen jedoch verschiedene Probleme gegenüber:

- Die Lesebandbreite nimmt bereits bei Datenmengen von einigen Dutzend Byte gegebenüber der Schreibbandbreite rapide ab. Dies wirkt sich direkt auf die Bandbreite für `MPI_Get`-Operationen aus.
- Der Benutzer sollte jeden beliebigen Speicherbereich als Window für andere Prozesse zugreifbar machen können - auch Bereiche, die nicht innerhalb eines SCI-Speichersegments allokiert wurden.
- Selbst wenn der Benutzer versucht, Windows innerhalb von SCI-Speichersegmenten anzulegen, kann dies aufgrund der begrenzten Größe von SCI-Speichersegmenten pro Knoten fehlschlagen, und es muß privater Speicher verwendet werden.

Es müssen also Verfahren entwickelt werden, die die beschriebenen Probleme umgehen und eine optimale Abwicklung von einseitiger Kommunikation unter allen potentiell auftretenden Umständen erlauben. Die erste Implementation von einseitiger Kommunikation in SCI-MPICH wurde in *[102]* vorgenommen, bot jedoch nur die grundlegende Funktionalität ohne weitergehende Leistungsoptimierungen.

5.2.1.1 Direkter Zugriff

Wenn der Speicher, der einem Window bei seiner Erstellung zugewiesen wird, über `MPI_Alloc_mem` als *shared* Speichertyp allokiert wurde (siehe Kapitel 7.2.2), kann ein entfernter Prozeß auf die Daten direkt lesend oder schreibend zugreifen. Die verschiedenen Zugriffsvarianten laufen dann so ab, daß der UP in der Window-Datenstruktur die SCI-Kennung des entfernten Speichersegments sucht und über die Ressourcenverwaltung den Zugriff auf den zugehörigen Speicherbereich sicherstellt. Auf die resultierende Adresse zuzüglich des spezifizierten Offsets wird sodann der Speicherzugriff durchgeführt. Dieses Konzept ermöglicht Zugriffe gemäß der SCI-Leistungscharakteristik (vergleiche Abb. 3.9): minimale Latenz und maximale Bandbreite für Schreibzugriffe (`MPI_Put`) und eine stark steigende Latenz und beschränkte Bandbreite für Lesezugriffe (`MPI_Get`). Die Verknüpfungsoperationen bei `MPI_Accumulate` erfordern beim direkten Zugriff zunächst eine Leseoperation für den entfernten Operanden und anschließend eine Schreiboperation für das Ergebnis.

Neben der Größe eines einzelnen Speicherzugriffs spielt für die Leistung des direkten Zugriffs auch der Abstand zwischen einzelnen, aufeinander folgenden Zugriffen eine Rolle, wie bereits in Kapitel 5.1.4.2 im Zusammenhang mit dem direkten Verfahren zur Übertragung nicht-zusammenhängender Datentypen dargestellt wurde (siehe auch Abb. 5.9). Dort hat die MPI-Bibliothek die Kontrolle über die auszuführenden Speicherzugriffe und kann die Zugriffe durch Ausrichtung optimieren. Diese Maßnahmen sind hier ebenso erforderlich.

In jedem Fall wird der Zugriff vollständig innerhalb der (blockierenden) Kommunikationsaufrufe abgewickelt: Die Daten stehen bei der Rückkehr der Funktion im angegebenen Zielpuffer. Dies ist ein strengeres Konsistenzmodell als vom Standard gefordert und entspricht dem SHMEM-Konsistenzmodell *[94]*.

5.2.1.2 Emulierter Zugriff

Da aus den oben beschriebenen Gründen der direkte Zugriff durch Lese- bzw. Schreiboperationen auf gemeinsame SCI-Segmente nicht immer die optimale Leistung liefert oder gar nicht möglich ist, wurde als alternatives Verfahren der *emulierte Zugriff* entwickelt. Zum Datentransport werden wie bei der zweiseitigen Kommunikation Nachrichten verwendet. Hierbei muß un-

terschieden werden zwischen *emulierten* Lese- und Schreibzugriffen. Während es bei einem Schreibzugriff ausreicht, die Daten von UP zum ZP zu übertragen (was für die reine Übertragung nicht zwangsläufig eine Mitwirkung des Zielprozesses erfordert), muß beim Lesezugriff der ZP veranlaßt werden, die Daten zum UP zu übertragen. Analog zum direkten Zugriff, nur auf höherer Ebene, sind daher beim Lesezugriff höhere Latenzen zu erwarten.

Datenanforderung und -übertragung. Für die Datenübertragung beim emulierten Zugriff ist es sinnvoll, die Protokolle der zweiseitigen Kommunikation zu nutzen, da diese für die jeweilige Datenmenge optimale Übertragungsmethoden anbieten. Es besteht jedoch das Problem, daß nach dem Prinzip der zweiseitigen Kommunikation jeder Sendeoperation eine passende Empfangsoperation gegenüberstehen muß, damit die Kommunikation insgesamt abgeschlossen werden kann. Dies ist bei einseitiger Kommunikation naturgemäß nicht der Fall. Daher verschickt der UP zusätzliche Kontrollnachrichten, die dem ZP den anstehenden Datentransfer bekannt macht (`MPI_Put` und `MPI_Accumulate`) bzw. ihn zu dem erforderlichen Datentransfer veranlassen (`MPI_Get`). Der jeweilige Ablauf von Kontrollnachrichten, Empfangsanforderungen und Datenübertragungen ist in Abb. 5.20 dargestellt.

Abbildung 5.20: Ablauf der verschiedenen Typen einseitiger Kommunikation bei *emuliertem Zugriff*

Gemäß diesem Modell sind zur Abwicklung einer *put*-Operation stets eine Kontrollnachricht und eine Datennachricht erforderlich. Analog zum *short*-Protokoll wurde auch hier zur Minimierung der Latenz beim Transfer kleiner Datenmengen die Möglichkeit geschaffen, diese Daten zusammen mit der Kontrollnachricht zu versenden. Dadurch ist in diesem Fall eine einzige Nachricht ausreichend, um eine *put*-Operation mit emuliertem Zugriff abzuwickeln, was die Latenz der Operation effektiv halbiert.

Während bei *put*- und *accumulate*-Operationen Nachrichten im *short*- und *eager*-Protokoll *unmittelbar* an den Empfänger ausgeliefert werden, ist dies bei Nutzung des *rendez-vous*-Protokoll nicht der Fall. Ebenso kann bei *get*-Operationen innerhalb der Operation unmittelbar auf den Empfang der angeforderten Daten gewartet werden oder der Empfang erst beim Abschluß der Zugriffsepoche *verzögert* durchgeführt werden. Die Unterscheidung in beiden Fällen erfolgt in beiden Fällen anhand der Verwendung von blockierenden oder nicht-blockierenden zweisei-

tigen Kommunikationsoperationen. Der Vorteil des unmittelbaren Verfahrens ist der mögliche Verzicht auf explizite Synchronisation zum Ende der Zugriffsepoche. Nachteile entstehen dabei potentiell durch die notwendige Synchronisation mit dem ZP[1]. Diese kann durch eine zusätzliche Signalisierung erzwungen werden (siehe Kapitel 7.1). Dies macht die umittelbare Operation vorwiegend geeignet zur Übertragung größerer Datenmengen; bei kleinen Datenmengen sollte möglichst mit Direkt- oder Sammelzugriff (siehe Kapitel 5.2.1.4) gearbeitet werden. Eine quantitative Untersuchung dieser Zusammenhänge erfolgt am Ende des Kapitels.

Datentypbehandlung. Gemäß dem MPI-2-Standard muß ein Datentyp, mit dem der UP eine einseitige Kommunikation durchführt, dem ZP nicht bekannt sein (da dieser in die Kommunikation nicht explizit involviert ist). Diese Situation kann eintreten, da die Definition eines Datentyps eine Prozeß-lokale Operation ist. Bei direktem Zugriff tritt dieses Problem nicht auf, da der Zugriff auf den entfernten Speicher vom UP durchgeführt wird, der alle nötigen Informationen besitzt. Beim emuliertem Zugriff wird der Zugriff jedoch effektiv vom ZP ausgeführt. Falls der Datentyp einen einzigen, zusammenhängenden Speicherbereich beschreibt (*zusammenhängender Datentyp*), reicht dazu die in jedem Fall übertragene Information *<Lage der Daten im Window, Länge der Daten in Byte>*, damit der ZP die empfangenen Daten in den Speicherbereich des Windows übertragen oder von dort versenden kann.

Sobald es sich bei dem Zieldatentyp jedoch um einen *nicht-kontinuierlichen Datentyp* handelt, müssen die vollständigen Informationen zum Aufbau des Datentyps sowie dessen interne Kennung vom UP an den ZP übermittelt werden, da der UP nicht davon ausgehen kann, daß der Datentyp dem ZP bekannt ist. Zudem haben die Datentypen zwischen den Prozessen keine einheitliche Kennung, so daß die Feststellung, ob ein Datentyp beiden Prozessen bekannt ist, die Übertragung der vollständigen Beschreibung des Datentyps erfordert. Um jedoch diese Datentypbeschreibung nicht bei jeder Kommunikation übermitteln zu müssen, wird bei der lokalen Beschreibung eines Datentyps vermerkt, welchem Prozeß diese Beschreibung bereits übermittelt wurde. Wenn eine Datentypbeschreibung einem Prozeß bereits übermittelt wurde, wird bei der nächsten Kommunikation mit diesem Datentypen nur noch mit der initialen Kontrollnachricht die inzwischen dem UP *und* dem ZP bekannte Kennung übertragen.

5.2.1.3 Remote-put-Zugriff

In einem besonderen Fall kann eine Mischung aus direktem und emuliertem Zugriff die optimale Leistung bringen: Wenn bei einer *get*-Operation die Adresse beim UP innerhalb eines Bereichs vom gemeinsamen Speicher liegt, ist es ab einer bestimmten Datenmenge effektiver, den ZP analog zum emulierten Zugriff zur Übertragung der Daten aufzufordern (*remote-put*-Verfahren). Um die *get*-Operation bei diesem Verfahren analog zum direkten Zugriff ebenfalls unmittelbar abzuschließen, wird dem ZP diese Aufforderung signalisiert, um so die Bearbeitung zu veranlassen. Dieser überträgt die Daten dann jedoch nicht mittels zweiseitiger Kommunikation, sondern durch direkten Zugriff auf den Speicherbereich des UP.

Die Datenmenge D_{rmtput}, ab der das *remote-put*-Verfahren eine kürzere Latenz als der direkte Lesezugriff des UP bewirkt, bestimmt sich zu

$$(5.6)\quad D_{rmtput} > (L_{ctrl-pp} + l_{int}) \cdot \frac{B_{rr}(D_{rmtput}) \cdot B_{cr}(D_{rmtput})}{B_{cr}(D_{rmtput}) - B_{rr}(D_{rmtput})}$$

5.2.1.4 Sammelzugriff

Beim zuvor beschriebenen emulierten Zugriff wird für jeden Aufruf einer einseitigen Kommunikationsoperation unmittelbar eine Kommunikation über Kontrollnachrichten und zugehöri-

1. Trotz der Verwendung von blockierenden Sendeaufrufen ist auch bei großen Datenmengen keine Verklemmungsgefahr gegeben, da die vorhergehende Kontrollnachricht zuvor stets eine passende Empfangsaufforderung beim ZP generiert.

gen Datentransport ausgelöst. Dies bedeutet, daß jede Kommunikationsoperation isoliert ausgeführt wird, auch wenn sie zusammen mit weiteren Operationen in der gleichen Zugriffsepoche liegt. Diese müssen gemeinsam erst mit dem Abschluß der Zugriffsepoche beendet sein. Dieses Konsistenzmodell bietet somit die Möglichkeit, die Ausführung von Kommunikationsoperationen innerhalb einer Zugriffsepoche zu verzögern und zum Abschluß der Zugriffsepoche die angefallenen Kommunikationsoperationen gemeinsam auszuführen. Diese Zugriffsform wurde ebenfalls für SCI-MPICH realisiert und wird als *Sammelzugriff* bezeichnet.

Mögliche Leistungsgewinne. Der Sammelzugriff kann sowohl als Alternative zum direkten als auch zum emulierten Zugriff Vorteile bringen. Als Alternative zum direktem Zugriff wird statt der entsprechenden Anzahl von Schreibvorgängen in den entfernten Speicher eine einzige zusammenhängende Nachricht übertragen. Als alternative Nutzung zum emulierten Zugriff kann der Sammelzugriff noch höhere Gewinne bringen, da die Startlatenz für eine Nachricht weitaus höher liegt als für einen Schreibzugriff. Nachteilig wirkt sich jedoch der notwendige lokale Kopiervorgang der Daten beim UP aus, der notwendig ist, um die Daten zu einer zusammenhängenden Nachricht zusammenzusetzen. Dies ist auf den ersten Blick vergleichbar mit der Übertragung nicht-kontinuierlicher Daten wie in Kapitel 5.1 beschrieben. Jedoch sind die dort beschriebenen Verfahren nicht auf den Sammelzugriff übertragbar. Um wie bei der direkten Übertragung nicht-zusammenhängender Daten vorgehen zu können, müßte für jeden Abschluß einer Zugriffsepoche zunächst ein eigener *hindexed*-Datentyp erstellt werden, der die nicht-regelmäßige Lage der Daten im Speicher des UP beschreibt. Der ZP müßte jedoch mit einem anderen Datentyp arbeiten, der sich aus der Plazierung der Daten in seinem Window ergibt. Da diese Datentypen unterschiedliche Typsignaturen haben können, würde aus den in Kapitel 5.1.4.3 beschriebenen Gründen eine fehlerhafte Datenübertragung erfolgen.

Implementation. Eine einseitige Kommunikationsoperation innerhalb eines Windows läßt sich beschreiben durch *Adresse*, *Zahl* und *Datentyp* der Elemente beim UP, *Rang* des ZP sowie *Plazierung* (relativ zum Window-Beginn), *Datentyp* und *Zahl* der Elemente beim ZP. Wenn von einem System mit homogener Datendarstellung ausgegangen wird (wie es bei den betrachteten Clustersystemen zulässig ist), und die Kommunikation sich auf zusammenhängende Datentypen beschränkt, reichen zur Beschreibung *Adresse*, *Länge* (in Byte), *Plazierung* und *Rang* des Zielprozesses aus. Der ZP benötigt zur Durchführung der Operation nur *Länge* und *Plazierung* sowie (bei *put*- und *accumulate*-Operationen) die Daten selber. Bei *accumulate*-Operationen ist zusätzlich der Typ der Verknüpfungsoperation erforderlich. Diese Minimierung der zu übermittelnden Information ist wichtig, um die Effizienz der Datenübertragung beim Sammelzugriff zu maximieren. Da sich die Zusammenfassung von Kommunikationsoperationen gerade bei kleinen Datenmengen für die einzelnen Operationen lohnt, darf die zu kommunizierende Datenmenge möglichst wenig durch Kontrollinformationen vergrößert werden.

Das Schema der Implementation ist in Abb. 5.21 dargestellt. Für jede Kommunikationsoperation werden die notwendigen Informationen in einer *einseitigen Transaktion* zusammengefaßt. Diese wird in einem nach der Adresse beim UP sortierten, ausgeglichenen Baum gespeichert. Für jeden Prozeß und für jede der drei Operationen *get*, *put* und *accumulate* werden getrennte Bäume angelegt. Für die *accumulate*-Operation wird für jeden Baum der Typ der Verknüpfungsoperation *aller* gespeicherten einseitigen Transaktionen gespeichert[1]. Das Beispiel in Abb. 5.21 zeigt den Fall, in dem Prozeß 1 einer Applikation mit vier Prozessen *get*- und *put*-Operationen mit ZP 2 durchgeführt hat. Zusätzlich wurden *accumulate*-Operationen (Verknüpfungstyp `MPI_MAX` bzw. `MPI_SUM`) mit den ZP 0 und 3 durchgeführt.

1. Falls in einer Zugriffsepoche *accumulate*-Operation mit unterschiedlichen Verknüpfungstypen zum gleichen ZP ausgeführt werden, werden die bislang gespeicherten Transaktionen eines Typs ausgeführt, bevor die nächste Transaktion gespeichert wird.

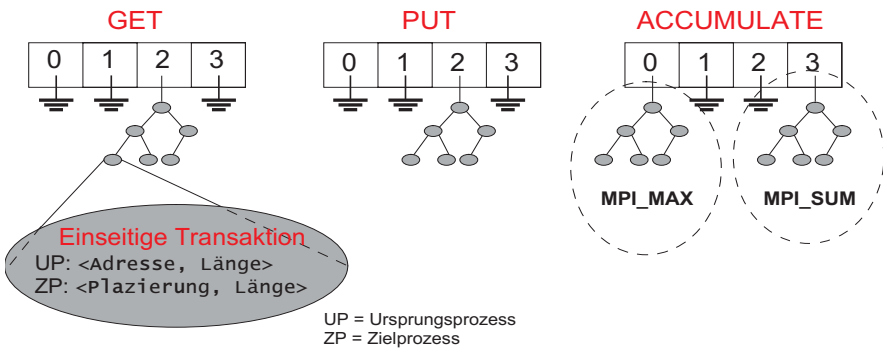


Abbildung 5.21: Datenstrukturen zur verzögerten Ausführung einseitiger Kommunikationsoperationen beim Sammelzugriff

Die Sortierung nach Adressen im UP dient dazu, beim Erstellen der Sammelnachricht sequentiell den Speicher traversieren zu können. Dabei können einzeln gespeicherte Transaktionen, die sowohl beim UP als auch beim ZP zusammenhängend sind, zusammengefaßt werden. Beim ZP ist die räumliche Lokalität durch den Empfang der Daten als *eine* Nachricht automatisch gegeben. Diese Sortierung ist auch konform mit dem MPI-Standard, der für die Abwicklung der Zugriffe innerhalb einer Zugriffsepoche keine Einhaltung der zeitlichen Reihenfolge garantiert. Dies ist relevant für *accumulate*-Operationen mit überlappenden Adreßbereichen im Ziel-Window.[1]

Die Kosten für die zusätzliche Kopieroperation auf die Daten bei der Erzeugung der Sammelnachricht steigen relativ zur Größe der Daten; gleichzeitig sinkt der durch die Verringerung der Startlatenzen erzielte relative Leistungsgewinn mit der Menge der Daten pro Operation. Daher kann die Obergrenze der Datenmenge pro Operation, bis zu der eine Kommunikationsoperation im Sammelzugriff durchgeführt wird, eingestellt werden.

5.2.2 Synchronisationsverfahren

Um die Implementation von einseitiger Kommunikation auch auf Systemen ohne die Möglichkeit des direkten Zugriffs aller Prozesse auf gemeinsamen (entfernten) Speicher effizient möglich zu machen, wurde im MPI-2 Standard die Synchronisation der Zugriffe von den Funktionsaufrufen für die Auslösung der Zugriffe getrennt. Im Gegensatz zu Zugriffsmodellen mit impliziter Synchronisation wie etwa SHMEM *[94]* ist dieses Modell der expliziten Synchronisation für den Programmierer komplexer zu handhaben. Neben dem eigentlichen Zugriff muß durch die Synchronisation sichergestellt werden, daß die Daten tatsächlich übertragen wurden. Bei `MPI_Get` heißt das, die Daten befinden sich nun im spezifizierten lokalen Puffer; bei einem `MPI_Put` oder `MPI_Accumulate` heißt das, die Daten wurden in den Speicher des entfernten Windows geschrieben. Der Vorteil für die MPI-Bibliothek auf Systemen ohne direkten Zugriff auf entfernten Speicher ist bei expliziter Synchronisation, daß ein vom Benutzer veranlaßter Zugriff nicht unmittelbar ausgeführt werden muß, sondern längstens bis zum Aufruf der nächsten auf dieses Window bezogenen Synchronisationsfunktion verzögert werden kann. Dies bietet die Möglichkeit zu Optimierungen wie beim Sammelzugriff in Kapitel 5.2.1.4 beschrieben.

Der Zugriff auf Daten in einem Window muß stets innerhalb einer Zugriffsepoche erfolgen, deren Beginn und Ende durch den Aufruf von Synchronisationsfunktionen gekennzeichnet ist. Der MPI-2-Standard definiert dazu drei unterschiedliche Synchronisationsmodelle, die sich in die Kategorien *aktives Ziel (active target)* und *passives Ziel (passive target)* einteilen lassen. Beim aktiven Ziel sind sowohl der UP als auch der ZP durch den Aufruf von Synchronisations-

1. Analoge überlappende *put*- und *get*-Operationen erlaubt der MPI-Standard nicht.

funktionen an der Definition der Zugriffsepoche beteiligt[1]. Beim passiven Ziel ist hingegen der ZP passiv, d.h. er beteiligt sich nicht durch Funktionsaufrufe an der Definition der Zugriffsepoche oder der Abwicklung der Kommunikation. Im folgenden werden die drei Synchronisationsmodelle im Hinblick auf ihre Umsetzung für den direkten, emulierten und gesammelten Zugriff in SCI-MPICH und die daraus entstehenden Randbedingungen zur Durchführung von Kommunikation und Synchronisation untersucht.

5.2.2.1 Zaunsynchronisation

Ähnlich zur Barrierensynchronisation ist die *Zaunsynchronisation* (`MPI_Win_Fence`) ein kollektiver Aufruf, der solange blockieren *kann*, bis alle Prozesse der Gruppe, die mit dem Window assoziiert ist, diese Funktion aufgerufen haben. Alle beteiligten Prozesse müssen diese Funktion gleich oft aufrufen, um eine Blockade zu verhindern. Somit ist dies eine Synchronisation der Kategorie *aktives Ziel*. Tatsächlich garantiert die Rückkehr dieser Funktion, daß alle einseitigen Kommunikationsoperationen, die vor dem Aufruf von `MPI_Win_fence` abgesetzt wurden, abgeschlossen sind. Kommunikationsoperationen, die nach dem Aufruf von `MPI_Win_fence` gestartet wurden, werden erst dann beim ZP ausgeführt, wenn dieser ebenfalls `MPI_Win_fence` aufgerufen hat.

Zum Eintritt in eine Zugriffsepoche kann `MPI_Win_fence` als einfache Barriere implementiert werden. Die Implementation von `MPI_Win_fence` als Abschluß einer Zugriffsepoche muß zwei Dinge sicherstellen:

- Alle Kommunikationsoperationen, die für dieses Window von diesem Prozeß als UP initiiert wurden, müssen abgeschlossen sein. Kommunikationsoperationen, die über direkten oder ummittelbaren emulierten Zugriff abgewickelt werden, sind bereits vor Aufruf der Synchronisation abgeschlossen. Der Abschluß von Operationen über verzögerten emulierten Zugriff kann durch den Abschluß der dazu initiierten Sende- bzw. Empfangsoperationen festgestellt werden.
- Die initiierten Kommunikationsoperationen müssen auch beim ZP abgeschlossen sein, bevor die Zaunsynchronisation abschließen darf. Dies betrifft *put*- und *accumulate*-Operationen, die über emulierten oder gesammelten Zugriff abgewickelt werden. Um sicherstellen zu können, daß alle anstehenden, von anderen Prozessen initiierten Operationen abgeschlossen wurden, muß von allen anderen Prozessen in der zum Window gehörigen Gruppe bekannt sein, daß sie sich innerhalb der Zaunsynchronisation befinden. Sobald dann alle eingegangenen Nachrichten abgearbeitet wurden, müssen bei allen Prozessen die Job-Zähler für das betroffene Window auf Null stehen (nachdem diese vorher für jede ausstehende Operation inkrementiert wurden). Dies gilt für den Fall, daß die Synchronisation der Prozesse untereinander durch Kontrollnachrichten erfolgt, da sich Nachrichten nicht überholen können und somit das Eintreffen der Kontrollnachricht der Zaunsynchronisation bedeutet, daß keine weiteren Kommunikationsanforderungen mehr eintreffen werden.

Die Erfüllung der ersten Bedingung kann ein Prozeß lokal feststellen. Die zur Erfüllung der zweiten Bedingung notwendige Art der Synchronisation kann durch eine Barriere erfolgen. Bei der Nutzung einer Barriere, die ohne Kontrollnachrichten arbeitet (siehe Kapitel 6.5), kann es hier durch die Nutzung eines unabhängigen Kommunikationskanals (nämlich den direkten Speicherzugriffen auf die gemeinsamen Datenstrukturen der Barriere) jedoch zu einer *Race-Condition* kommen: Während der Barrierensynchronisation können bei einem ZP Anforderungen für emulierte Zugriffe eintreffen, die von dem UP vor dessen Eintritt in Zaunsynchronisation abgesetzt wurden. Die Barrierensynchronisation "überholt" in diesem Fall die entsprechenden Kontrollnachrichten, was aufgrund der Kommunikationsabwicklung in ge-

1. Die durch den ZP definierte Zugriffsepoche wird auch als *Öffnungsepoche (exposure epoch)* bezeichnet.

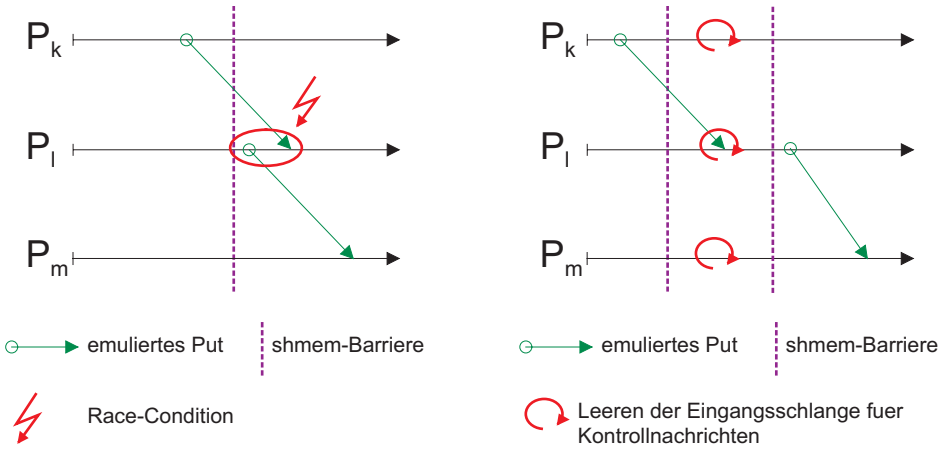


Abbildung 5.22: Mögliche *Race-Condition* bei der Zaunsynchronisation mittels shared-memory-Barriere (*links*) und Sicherung der Synchronisation mittels doppelter Barriere (*rechts*)

trennten Kanälen, also innerhalb und außerhalb des Protokolls für Kontrollnachrichten, möglich ist. Wenn diese Anforderungen innerhalb der Zaunsynchronisation nicht bearbeitet werden, entspricht diese nicht den Anforderungen des MPI-2-Standards. Als Folge kann, wie in Abb. 5.22 (*links*) dargestellt, Prozeß P_l ein Datum verschicken, das gemäß dem Algorithmus zuvor von Prozeß P_k geschrieben worden sein sollte. Tatsächlich wurde der Schreibvorgang, übermittelt durch eine emulierte *put*-Operation, von Prozeß P_l noch gar nicht erkannt, da dieser auch bei der Zaunsynchronisation noch keine neu eingegangene Nachricht vorgefunden hatte.

Hier sind zwei Lösungen möglich:

- *Doppelbarriere (Abb. 5.22, rechts):* Es werden *zwei* Barrierensynchronisationen durchgeführt: Nach der ersten Synchronisation wird keiner der Prozesse eine weitere Kommunikationsanforderung absetzen, so daß alle Prozesse die bis dahin eingegangenen Nachrichten abarbeiten können, bis die Eingangsschlange für Kontrollnachrichten geleert ist und alle lokalen Empfangsoperationen für das Window abgearbeitet sind. Anschließend wird die zweite Barrierensynchronisation durchgeführt, bevor die Prozesse die Zaunsynchronisation durch Verlassen von `MPI_Win_fence` abschließen.
- *Nachrichtenbarriere:* Alternativ wird statt der (schnelleren) Barriere mittels gemeinsamen Speichers die herkömmliche nachrichtenbasierte Barriere verwendet. Da sich dabei die Nachrichten nicht überholen können, sind nach Abschluß der Barriere und aller lokalen Empfangsoperationen die Bedingungen für die Zaunsynchronisation erfüllt.

Obwohl bei der Doppelbarriere *zwei* Barrierensynchronisationen nötig sind, ist dies aufgrund der deutlich niedrigeren Latenz für die Synchronisation über gemeinsamen Speicher (siehe Kapitel 6.5, Abb. 6.10) die schnellere Variante. Der Vorteil der Nachrichtenbarriere ist jedoch, daß jeder Prozeß nach dem Abschluß der Barrierensynchronisation und der Abarbeitung der lokalen, auf das Window bezogenen Empfangsoperationen unabhängig von allen anderen Prozessen die Zaunsynchronisation abschließen kann. Dies kann bei unausgeglichener Verteilung der Kommunikationsanforderungen auf die Prozesse relevante Zeitunterschiede zum Abschluß der Zaunsynchronisation zwischen den Prozessen erzeugen. Da jedoch in typischen iterativen Applikationen alle Prozesse spätestens bei der nächsten Zaunsynchronisation wieder synchronisiert werden, ist es unwahrscheinlich, daß eine Applikation insgesamt dadurch einen Laufzeitvorteil erreicht. In SCI-MPICH wird daher die Zaunsynchronisation mittels der Doppelbarriere über gemeinsamen Speicher durchgeführt.

5.2.2.2 Gruppensynchronisation

Die *Gruppensynchronisation* ermöglicht die Beschränkung der Konnektivität der Synchroni-

sation auf eine Teilgruppe der zum Window gehörigen Prozesse (also minimal ein UP und ein ZP) und stellt gegenüber der Zaunsynchronisation die generalisierte Form der aktives-Ziel-Synchronisation dar. Dies ermöglicht eine Entkopplung der Synchronisation und somit einen Leistungsgewinn in Applikationen, in denen immer nur Untergruppen von Prozessen des Window-Kommunikators miteinander kommunizieren. Eine Kommunikator-globale Barriere wie bei der Zaunsynchronisation ließe hier einen Teil der Prozesse unnötig warten.

Ein Prozeß eröffnet für die betroffene Teilgruppe von Prozessen eine *Öffnungsepoche* auf sein lokales Window durch den Aufruf von `MPI_Win_post` und beendet diese Epoche durch den Aufruf von `MPI_Win_wait`. In dieser Zeit werden alle Zugriffe auf das Window, die von den UPs in ihrer Zugriffsepoche (gekennzeichnet durch den Aufruf von `MPI_Win_start` und `MPI_Win_complete`, bezogen auf dieses Window) gestartet wurden, ausgeführt. Die dadurch mögliche zeitlich unabhängige Definition der Öffnungs- und Zugriffsepochen durch ZPs und UPs kann von der MPI-Bibliothek unterschiedlich gehandhabt werden, wie in Abb. 5.23 dargestellt ist:

- Die Rückkehr von `MPI_Win_start` kann verzögert werden, bis alle betroffenen ZPs den Zugriff auf ihre Windows durch `MPI_Win_post` freigegeben haben (*strenge* Gruppensynchronisation).
- Die Ausführung der Kommunikationsoperationen der UPs wird bei den ZPs verzögert, bis diese den Zugriff auf ihr lokales Window freigegeben haben (*lockere* Gruppensynchronisation).

Abbildung 5.23 zeigt diese zwei mögliche Varianten der Gruppensynchronisation, die beide MPI-Standard-konform sind. Bei der strengen Gruppensynchronisation kann die Zugriffsphase beim UP nicht eröffnet werden, bevor beim ZP die Öffnungsepoche begonnen hat, und die Zugriffsphase kann nicht abgeschlossen werden, bevor die Zugriffsoperation beim ZP durchgeführt wurde. Dies führt zu einer engen zeitlichen Kopplung der Prozesse durch die Gruppensynchronisation.

Die lockere Gruppensynchronisation kann jedoch nur dann verwendet werden, wenn in der Zugriffsepoche *alle* Kommunikationsoperationen im Sammelzugriff abgewickelt werden - ein direkter Zugriff auf ein entferntes Window, dessen Öffnungsepoche noch nicht begonnen hat, ist nicht zulässig. Der emulierte Zugriff wäre zwar möglich, doch die Nachrichten müßten beim ZP bis zum Beginn der Öffnungsepoche zwischengespeichert werden - dies hätte zur Folge, daß die volle Zahl der Startlatenzen *und* die zusätzlichen Kopieroperationen anfallen.

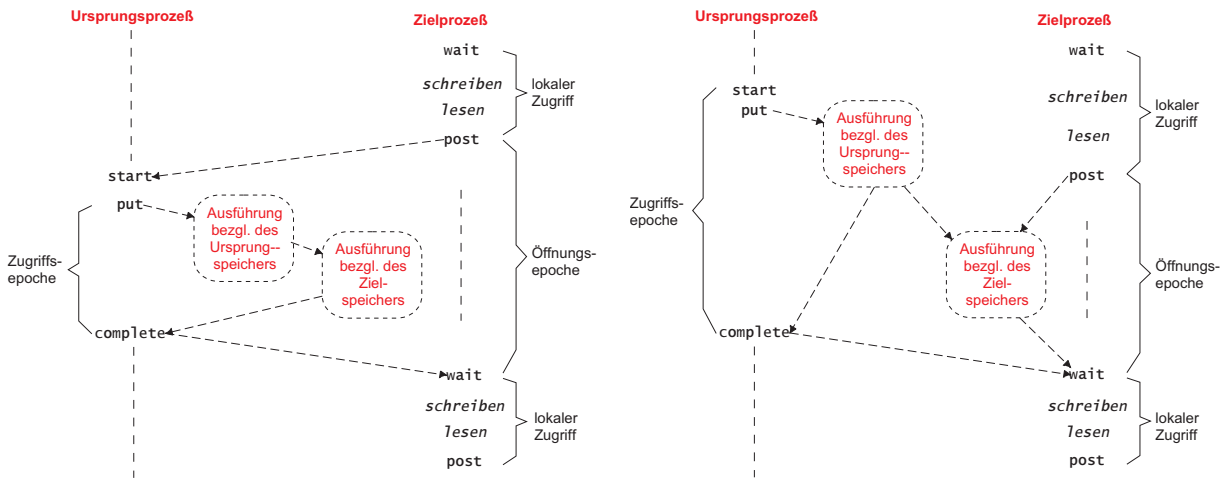


Abbildung 5.23: Mögliche MPI-konforme zeitliche Abläufe der Gruppensynchronisation (nach *[64]*). Die gestrichelten Pfeile stellen zeitliche Abhängigkeiten dar.
Links: Strenge Gruppensynchronisation *Rechts:* Lockere Gruppensynchronisation

Beide Arten der Gruppensynchronisation werden durch Nachrichtenversand zwischen Ursprungs- und Zielprozessen innerhalb eines sequentiellen Baumes (vergl. Kapitel 6.1.1) durchgeführt. Dadurch ist es für sehr große Gruppen vorteilhafter, die Zaunsynchronisation zu verwenden, da die dort verwendete Barrierensynchronisation logarithmisch statt linear mit der Zahl der Prozesse skaliert (vergl. Kapitel 6.5.3). Für die eigentliche Anwendung der Gruppensynchronisation, nämlich die entkoppelte einseitige Kommunikation innerhalb von Untergruppen eines Kommunikators, ist dieses Synchronisationsverfahren jedoch aufgrund des blockierungsfreien Versands von Kontrollnachrichten besser geeignet.

5.2.2.3 Passivsynchronisation

Die Synchronisation mit passivem ZP wird als *Passivsynchronisation* bezeichnet. Dies wird dadurch erreicht, daß jeder UP die Zugriffsepoche auf ein Window eines bestimmten Zielprozesses durch den Aufruf von `MPI_Win_lock` eröffnen kann. Dabei kann das Window *exclusive*, d.h. zum alleinigen Schreib- und Lesezugriff, oder *shared* zum gemeinsamen, nebenläufigen Lesezugriff durch mehrere UP reserviert werden. Die Zugriffsepoche wird durch Aufruf von `MPI_Win_unlock` abgeschlossen; dann sind alle Zugriffe auf das Window sowohl beim UP also auch beim ZP abgeschlossen.

Die Möglichkeit zu dieser Art der Synchronisation darf gemäß MPI-2 Standard auf Windows beschränkt sein, deren Speicher über `MPI_Alloc_mem` allokiert wurde. Ohne gemeinsamen Speicher (der oft nur mit `MPI_Alloc_mem` allokiert werden kann) zur Abwicklung der Synchronisation und der Datenzugriffe wird aufgrund des passiven Verhaltens des ZP ein aktiver *Handler*, entsprechend dem Device-Thread in SCI-MPICH (siehe Kapitel 7.1), benötigt. Ein solcher Handler kann jedoch nicht generell vorausgesetzt werden.

In SCI-MPICH kann jedoch jedes Window passiv synchronisiert werden. Dazu werden *Locks* sowohl für *exclusive*- als auch für *shared*-Zugriffe in separat verwaltetem gemeinsamem Speicher verwendet, die bei dem Erstellen eines Windows angelegt werden. Dabei werden die Locks für jeden Teil eines Windows beim jeweiligen Besitzer lokalisiert. Dadurch arbeiten diese Locks mit kurzen Latenzen, solange die Last auf einem einzelnen Lock nicht zu hoch wird. Diese Situation kann eintreten, wenn viele Prozesse gleichzeitig exklusiven Zugriff auf den Window-Teil eines Prozesses haben möchten. Jedoch ist ein solcher *hot-spot* für die Skalierung immer ein Problem und sollte bereits vom Algorithmus durch Verteilung der Daten auf mehrere Windows, die unabhängig voneinander synchronisiert werden können, vermieden werden. Für über alle Teile des Windows verteilte nicht-exklusive oder kurze Zugriffe mit geringer Frequenz bieten Locks im gemeinsamen Speicher durch kurze Latenzen gute Leistungseigenschaften.

Beim Abgeben eines Locks wird zuvor sichergestellt, daß alle ausstehenden Zugriffe auf entfernte Window-Teile bzw. Zugriffe entfernter Prozesse auf den lokalen Window-Teil abgeschlossen sind. Die Feststellung, daß auch beim ZP alle Operationen (wie emulierte *put*- und *accumulate*-Operationen) abgeschlossen sind, erfolgt über einen Job-Zähler im gemeinsamen Speicher beim UP, auf den der ZP zugreifen kann. Bei jeder vom UP veranlaßten Operation, die der ZP ausführt, inkrementiert er diesen Zähler. Der UP wartet solange, bis dieser Zähler den Stand des identisch inkrementierten lokalen Referenzzählers der an den ZP abgesandten Jobs erreicht hat. Zusätzlich wird der ZP signalisiert, um so die Abarbeitung der Eingangsschlange zu erzwingen.

5.2.2.4 Synchronisationsloser Betrieb

Die Erforderlichkeit von expliziter Synchronisation hat zum einen Einfluß auf die Leistung einer Applikation, da jede Synchronisation zwischen Prozessen Wartezustände einzelner Prozesse impliziert. Zum anderen ist das Modell der Sicherstellung der Speicherkonsistenz durch explizite Synchronisation komplexer in Applikationen einzusetzen als Modelle mit impliziter Synchronisation, wie etwa SHMEM. Dies ist möglicherweise ein Grund, warum die SHMEM-

Kommunikation vergleichsweise häufiger verwendet wird *[93]* als einseitige Kommunikation gemäß MPI-2. Dies, obwohl SHMEM im Gegensatz zu MPI-2 kein vereinbarter Standard ist und über die Systeme von Cray/SGI hinaus kaum unterstützt wird.

Tatsächlich entspricht die explizite Synchronisation zur Eröffnung einer Zugriffs- bzw. Öffnungsepoche, die zur Sicherstellung der Speicherkonsistenz erforderlich ist, den üblichen Programmiergewohnheiten für nebenläufigen Zugriff auf gemeinsame Daten. Dies trifft jedoch nicht zu für das Modell, daß die Daten nach Abschluß einer Kommunikationsoperation noch nicht am Bestimmungsort angelangt sein müssen, sondern daß es dazu einer weiteren expliziten Synchronisationsoperation bedarf. Dies hat beispielsweise zur Folge, daß eine einfache Folge *Lesen-Manipulieren-Zurückschreiben* auf ein Datum in einem entfernten Window nicht atomar ausgeführt werden kann.

Es liegt daher nahe, einen Modus des *unmittelbaren Betriebs* der einseitigen Kommunikation zu implementieren und zu evaluieren. In diesem Modus sind Aufrufe der Synchronisationsfunktionen zur tatsächlichen Übertragung der Daten einer Kommunikationsoperation nicht erforderlich; stattdessen wird eine Kommunikationsoperation innerhalb des originären Aufrufs so weit bearbeitet, daß bei *put*- und *accumulate*-Operationen der Quelldatenbereich vom UP beim ZP angelangt ist, und die angeforderten Daten bei einer *get*-Operation im spezifizierten Puffer des UP vorliegen.

Ein solcher Modus wurde in SCI-MPICH implementiert. Bei direktem Zugriff ist er ohnehin gegeben; bei emulierten Zugriffen ist es erforderlich, daß Sende- bzw. Empfangsoperationen unmittelbar im Funktionsaufruf der einseitigen Kommunikation abgeschlossen werden. Bei Empfangsoperationen wird dies durch einen blockierenden Empfangsaufruf gewährleistet. Bei Sendeoperationen jedoch reicht der Schreibvorgang in den Eingangspuffer des ZP nicht aus - die Daten müssen im Zielfenster gespeichert werden, damit die Semantik des Zugriffs gemäß MPI-Standard erfüllt ist, ohne daß dies in einer Synchronisation geschieht. Auch eine anschließende Signalisierung ist nicht ausreichend, da der UP auch dann keine Sicherheit über den Zeitpunkt des Abschlusses der Datenübertragung beim ZP hat. Dies kann nur durch Verwendung von synchronen Sendevorgängen erreicht werden; hierbei werden alle Daten mit dem *rendezvous*-Protokoll übertragen, so daß der UP die Bestätigung hat, daß der ZP mit dem Empfangsvorgang begonnen hat und ihn abschließen wird, bevor er eine anderen Kommunikationsoperation durchführt. Durch Signalisierung wird der ZP zur unmittelbaren Abarbeitung der Kommunikationsoperation gezwungen.

Die einzelne Operation dauert beim unmittelbaren Betrieb mit emuliertem Zugriff also länger. Dafür muß bei der Synchronisation zum Abschluß der Zugriffsepoche keine Kommunikation mehr abgeschlossen werden und insbesondere nicht auf die Kommunikationspartner gewartet werden.

5.2.3 Modellierung und Evaluierung

Analog zur Modellierung und Evaluierung der zweiseitigen Kommunikationsprotokolle werden die vorgestellten Varianten der einseitigen Kommunikation (direkter und emulierter Zugriff sowie Sammelzugriff) modelliert. Anschließend werden diese erwarteten Leistungswerte mit den durch die Evaluierung der Implementation ermittelten Werten verglichen.

Die Evaluierung der Leistung einseitiger Kommunikation ist schwierig, da das Konzept der nicht-sequentiellen Konsistenz mit Zugriffsepochen und deren Abschluß durch explizite Synchronisation es nicht erlaubt, eine einzelne Kommunikationsoperation zu betrachten. Stattdessen muß stets die gesamte Epoche inklusive der Synchronisation betrachtet werden. Wenn dabei die Kommunikation jedoch über emulierten Zugriff für Datenmengen oberhalb des *eager*-Protokolls abgewickelt wird, ist zusätzlich entscheidend, ob der ZP die Daten selbständig entgegennimmt (weil er gerade die Eingangsschlange für Kontrollnachrichten abfragt), oder ob er

dazu durch eine entfernte Unterbrechung veranlaßt wird.

Diese Abhängigkeiten machen es bei einseitiger Kommunikation noch schwieriger als bei zweiseitiger Kommunikation, die Leistungswerte aus synthetischen Benchmarks auf die erwartete Leistung von Applikationen zu übertragen. Jedoch können die gewonnenen Erkenntnisse vom Programmierer bzw. Nutzer einer Applikation dazu genutzt werden, aus verschiedenen Möglichkeiten zur Abwicklung der Kommunikation das optimale Verfahren zu wählen und die Parameter sowohl der Applikation (z.B. Größe und Lage der Daten, Art der Speicherallokation, Synchronisationsverfahren) als auch von SCI-MPICH (Wahl der Zugriffsart für unterschiedliche Datenmengen, Nutzung des unmittelbaren Betriebs) aufeinander abzustimmen.

Für große Datenmengen ist die Bandbreite für einseitige Kommunikation bei Messungen, die analog zum Ping-Pong-Verfahren erfolgen, ähnlich wie für zweiseitige Kommunikation. Dies liegt dran, daß dann für *get*- und *accumulate*-Operationen die Übertragung ohnehin mit den Protokollen der zweiseitigen Kommunikation abgewickelt wird. Nur für *put*-Operationen, deren Ziel-Window im gemeinsamen Speicher liegt, wird ein direkter Zugriff durchgeführt. Da die Anteile des Protokolls an den Übertragungszeiten großer Datenmengen immer geringer werden, dominiert die Zeit, die für die Datenübertragung benötigt wird. Diese ist bei allen Zugriffsverfahren annähernd gleich. Die ermittelten Bandbreitenwerte für die unterschiedlichen Zugriffsverfahren sind in Abb. 5.24 dargestellt. Die weitere Modellierung und Evaluierung wird sich daher auf die Latenz für die Übertragung kleiner Datenmengen beschränken. Dies ist für einseitige Kommunikation auch die häufigere Verwendung, wenn in irregulären Problemen (wie etwa Molekulardynamik) Datensätze für einzelne Elemente im System übertragen werden. Diese Datensätze bestehen zumeist aus einer kleinen Zahl von Fließkommawerten, so daß sich Datenmengen von weniger als 1 kB ergeben *[151]*.

Die Modellierung der einseitigen Kommunikation erfolgt mit den gleichen Zielen wie die der zweiseitigen Kommunikation (vergl. Kapitel 4.3). Allerdings kann für die einseitige Kommunikation nicht jeder Punkt im Protokoll, wie bei der zweiseitigen Kommunikation, einzig anhand der Datenmenge D modelliert werden: Zusätzlich ist noch die Zahl R der gleichartigen Operationen in der betrachteten Zugriffsepoche erforderlich. Nur beim synchronisationslosen Betrieb können die Zugriffsoperationen isoliert betrachtet werden. Als Synchronisationslatenz wird l_{wsync} eingesetzt, die durch die Synchronisationsmodelle gemäß Kapitel 5.2.3.4 bestimmt werden kann. Die im Zuge des Abschlusses einer Zugriffsepoche abzuwickelnden Kommunikationsoperationen brauchen hier nicht berücksichtigt zu werden. Diese werden bereits für den Zugriff modelliert, was für die Gesamtdauer der Epoche zum gleichen Ergebnis führt.

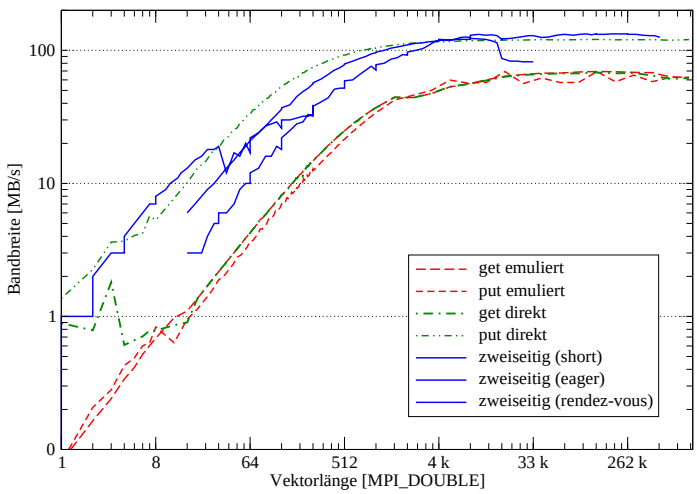


Abbildung 5.24: Ping-Pong-Bandbreite für ein- und zweiseitiger Kommunikation

5.2.3.1 Direkter Zugriff

Die untere Schranke der Zeitdauer für einen direkten Zugriff entspricht den Leistungsdaten eines direkten PIO-Zugriffs auf entfernten Speicher inklusive der erforderlichen Speicherbarriere. Für *get*- und *put*-Operationen (*accumulate*-Operationen werden stets mit emuliertem Zugriff durchgeführt) ergibt sich damit für die Bandbreiten $B_{get_{direkt}}$ und $B_{put_{direkt}}$ mit den Größen aus Tabelle 4.1:

$$(5.7)\quad B_{get_{direkt}}(D,R) = \frac{R \cdot D}{R \cdot \left(\frac{D}{B_{rr}(D)} + l_{lb}\right) + l_{wsync}}$$

$$B_{put_{direkt}}(D,R) = \frac{R \cdot D}{R \cdot \left(\frac{D}{B_{rw}(D)} + l_{sb}\right) + l_{wsync}}$$

Das *remote-put*-Verfahren zur Verbesserung der Leistung von *get*-Operationen im direkten Zugriff erfordert den Versand einer Kontrollnachricht, begleitet von einer entfernten Unterbrechung, sowie den Schreibvorgang in den entfernten Speicher, womit sich ergibt

$$(5.8)\quad B_{get_{direkt-write}}(D,R) = \frac{R \cdot D}{R \cdot \left(L_{ctrl-pp} + l_{sint} + \frac{D}{B_{rw}(D)}\right) + l_{wsync}}$$

Der Einfluß der Ausrichtung der Zieladressen auf die Übertragungsbandbreite sind im Modell nicht abgebildet, sollten bei der Wahl des geeigneten Übertragungsprotokolls für einseitige Kommunikation jedoch mit berücksichtigt werden. Die Probleme, die mit einer ungeeigneten Ausrichtung verbunden sind, treten beim emulierten Zugriff sowie beim Sammelzugriff nicht auf. Die Abbildungen 5.26 und 5.25 zeigen den Verlauf von Latenz und Bandbreite im Vergleich des Modells und des Experiments.

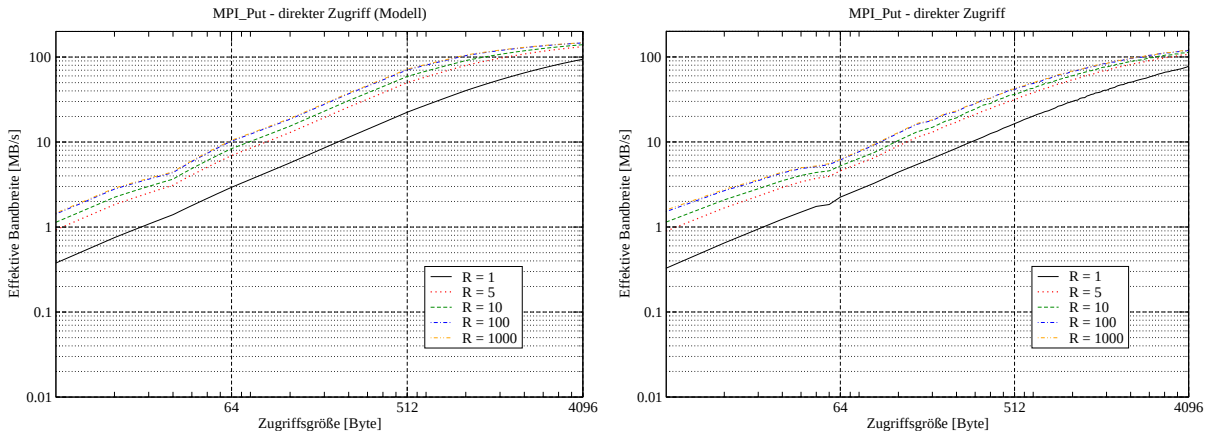


Abbildung 5.25: Modell und Experiment für den direkten Zugriff bei einseitiger Kommunikation: *put*-Operationen
(Parameter *R* für Zahl der Zugriffe pro Epoche)

5.2.3.2 Emulierter Zugriff

Der emulierte Zugriff wird über Nachrichten abgewickelt, wobei die der Datenmenge entsprechenden Protokolle verwendet werden. Neben den Kontrollnachrichten zur Anforderung (*get*) bzw. Ankündigung (*put, accumulate*) der Daten und zum Transport der Daten selber können noch weitere Nachrichten zur Übermittlung der Beschreibung des Datentyps erforderlich sein,

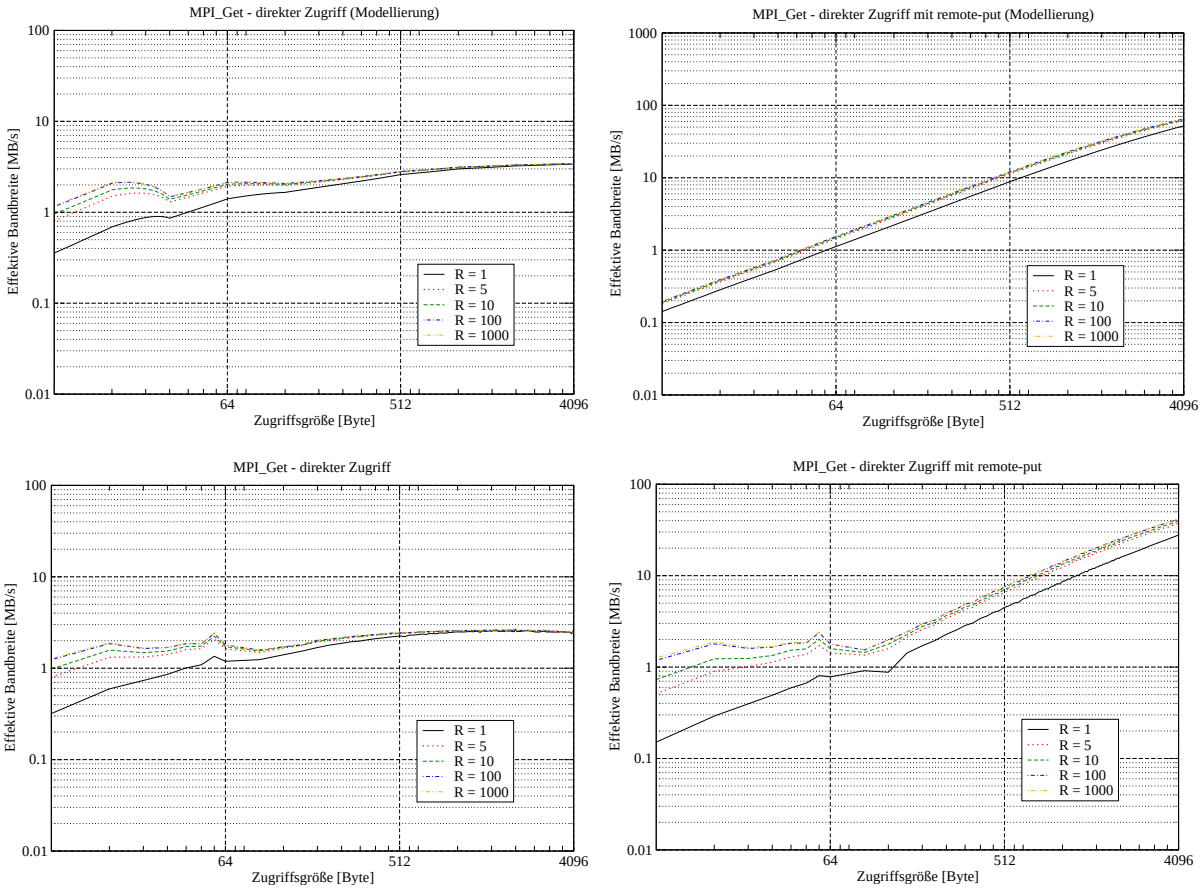


Abbildung 5.26: Modell und Experiment für den direkten Zugriff bei einseitiger Kommunikation (Parameter *R* für Zahl der Zugriffe pro Epoche)
Links: direkte *get*-Operationen; *Rechts: get*-Operationen mit *remote-put*-Verfahren

sofern es sich um einen nicht-kontinuierlichen Datentyp handelt. Diese Datentypnachrichten werden jedoch in der Modellierung nicht berücksichtigt, da sie pro nicht-kontinuierlichem Datentyp nur einmalig für alle folgenden Kommunikationsoperationen versandt werden müssen. Neben den zu versendenden Nachrichten muß für den Fall der signalisierten Übertragung die Latenz der dazu generierten entfernten Unterbrechung berücksichtigt werden.

Zusätzlich muß die Latenz $l_{job} = l_{rw} + l_{sb}$, die jeder ZP nach Bearbeitung einer eingegangenen *put*- oder *accumulate*-Operation zur Aktualisierung des Job-Zählers beim UP benötigt, berücksichtigt werden. Damit ergeben sich für die verschiedenen Arten des emulierten Zugriffs die im folgenden aufgeführten Bandbreiten:

Unsignalisierte *get*- und *put*-Operation[1]:

$$(5.9)\quad B_{get_{emu}}(D,R) = \frac{R \cdot D}{R \cdot (L_{ctrl-pp} + L_{msg}(D)) + l_{wsync}}$$

$$B_{put_{emu}}(D,R) = \frac{R \cdot D}{R \cdot (L_{ctrl-p} + L_{msg}(D) + l_{job}) + l_{wsync}}$$

1. *accumulate*-Operationen werden als *put*-Operation modelliert, was aufgrund der ermittelten Bandbreiten für reine Kopier- und Skalierungsoperationen (Kapitel 2.1.2, Tabelle 2.1) in diesem Rahmen zulässig erscheint.

Signalisierte *Get*- oder *Put*-Operation:

$$(5.10)\quad B_{get_{emu-sig}}(D,R) = \frac{R \cdot D}{R \cdot (L_{ctrl-pp} + l_{int} + L_{msg}(D)) + l_{wsync}}$$

$$B_{put_{emu-sig}}(D,R) = \frac{R \cdot D}{R \cdot (L_{ctrl-p} + l_{int} + L_{msg}(D) + l_{job}) + l_{wsync}}$$

put-Operation im *short*-Protokoll ("inline") ohne und mit Signalisierung:

$$(5.11)\quad B_{put_{inl}}(D,R) = \frac{D}{L_{short}(D) + \frac{l_{wsync}}{R}}$$

$$B_{put_{inl-sig}}(D,R) = \frac{D}{L_{short}(D) + l_{int} + \frac{l_{wsync}}{R}}$$

Bei den *get*-Operationen fällt jeweils die Ende-zu-Ende-Latenz von Kontrollnachrichten bzw. Unterbrechungen an, da der ZP diese Ereignisse zunächst verarbeiten muß, bevor er mit dem Versand der Daten reagieren kann. Bei *put*-Operationen hingegen überlappen sich diese Vorgänge beim ZP mit dem Versand der Daten durch den UP, so daß die Latenz entsprechend kürzer ausfällt. Mit zunehmender Nachrichtengröße wird dieser Unterschied vernachlässigbar. Ausreichend kleine Datenmengen können bei *put*-Operationen analog zum *short*-Protokoll zusammen mit den Kontrollinformationen des Protokolls in einer Nachricht versandt werden. Dadurch entfällt die Sendelatenz einer Kontrollnachricht.

5.2.3.3 Sammelzugriff

Beim Sammelzugriff wird eine beliebige Zahl von *get*-, *put*- oder *accumulate*-Operationen, die von einem UP zu einem ZP während einer Epoche initiiert wird, am Ende der Epoche durch einen Kommunikationsvorgang (je Operationstyp) zwischen diesen Prozessen abgewickelt. Der Zeitbedarf zur Abwicklung dieses Kommunikationsvorgangs (nur dieser wird im Modell erfaßt) hängt entsprechend ab von

- der Summe der Größe der bei den einzelnen Operationen involvierten Daten, die die Größe der Datennachricht des Kommunikationsvorgangs bestimmt;
- der Zahl der Operationen und der Größe der involvierten Daten, da diese Daten beim UP durch eine entsprechende Zahl von Kopieroperationen in einen zusammenhängenden Speicherbereich kopiert werden müssen, um beim ZP nach der Übertragung aus der eingegangenen Nachricht zur Zieladresse kopiert zu werden.

Für einen einzelnen Zugriff innerhalb einer Epoche bringt der Sammelzugriff keinen Gewinn, da dabei die selben Kommunikationsschritte wie beim emulierten Zugriff anfallen. Der Aufwand der zusätzlichen Kopieroperationen lohnt sich erst, wenn er durch den Wegfall der Latenzen einer Zahl von kürzeren Nachrichten zugunsten der einzelnen Latenz einer Nachricht wettgemacht wird. Zusätzlich zu den Daten muß für jeden der gesammelten Zugriffe eine Beschreibung $V = <Plazierung, Länge>$ über Lage und Größe der Daten der einzelnen Operationen im Window des ZP mitgeschickt werden. Beim emulierten Zugriff ist diese Beschreibung in der Kontrollnachricht enthalten, beim Sammelzugriff wird die entsprechende Zahl von Beschreibungen zusammen mit den Daten (*put*-Operation) oder als getrennte Nachricht mit der Anforderung (*get*-Operation) versandt. Im Modell drückt sich dies folgendermaßen für R Zugriffe der Größe $D(i)$ aus:

$$(5.12)\quad B_{put_{sammel}}(D,R) = \frac{\sum_{i=0}^{R} D(i)}{L_{ctrl-p} + 2\cdot\sum_{i=0}^{R}\frac{D(i)}{B_{cl}(i)} + L_{msg}\left(R\cdot V + \sum_{i=0}^{R} D(i)\right) + \frac{l_{wsync}}{R}}$$

Die Modellierung der Bandbreite für eine entsprechende *get*-Operation sieht sehr ähnlich aus, jedoch werden V und die zugehörigen Daten in zwei Nachrichten übertragen: die erste vom UP zum ZP, die zweite in die Gegenrichtung.

$$(5.13)\quad B_{get_{sammel}}(D,R) = \frac{\sum_{i=0}^{R} D(i)}{L_{ctrl-pp} + L_{msg}(V\cdot R) + L_{msg}\left(\sum_{i=0}^{R} D(i)\right) + 2\cdot\sum_{i=0}^{R}\frac{D(i)}{B_{cl}(i)} + \frac{l_{wsync}}{R}}$$

Die für den Sammelzugriff ermittelten Werte aus Modell und Experiment sind in Abb. 5.27 dargestellt.

5.2.3.4 Synchronisation

Neben der Kommunikation fallen für jede Zugriffsepoche Synchronisationsoperationen an. Für eine vollständige Modellierung der einseitigen Kommunikation muß die damit verbundene Latenz ebenfalls berücksichtigt werden, wenn nicht aus bestimmten Gründen vollständig auf Synchronisation verzichtet werden kann (siehe Kapitel 5.2.2.4). Wie stark diese Latenzen sich auf die Gesamtleistung der einseitigen Kommunikation auswirken, hängt vom Verhältnis der für den Datenaustausch benötigten Zeit zu der Synchronisationsverzögerung ab.

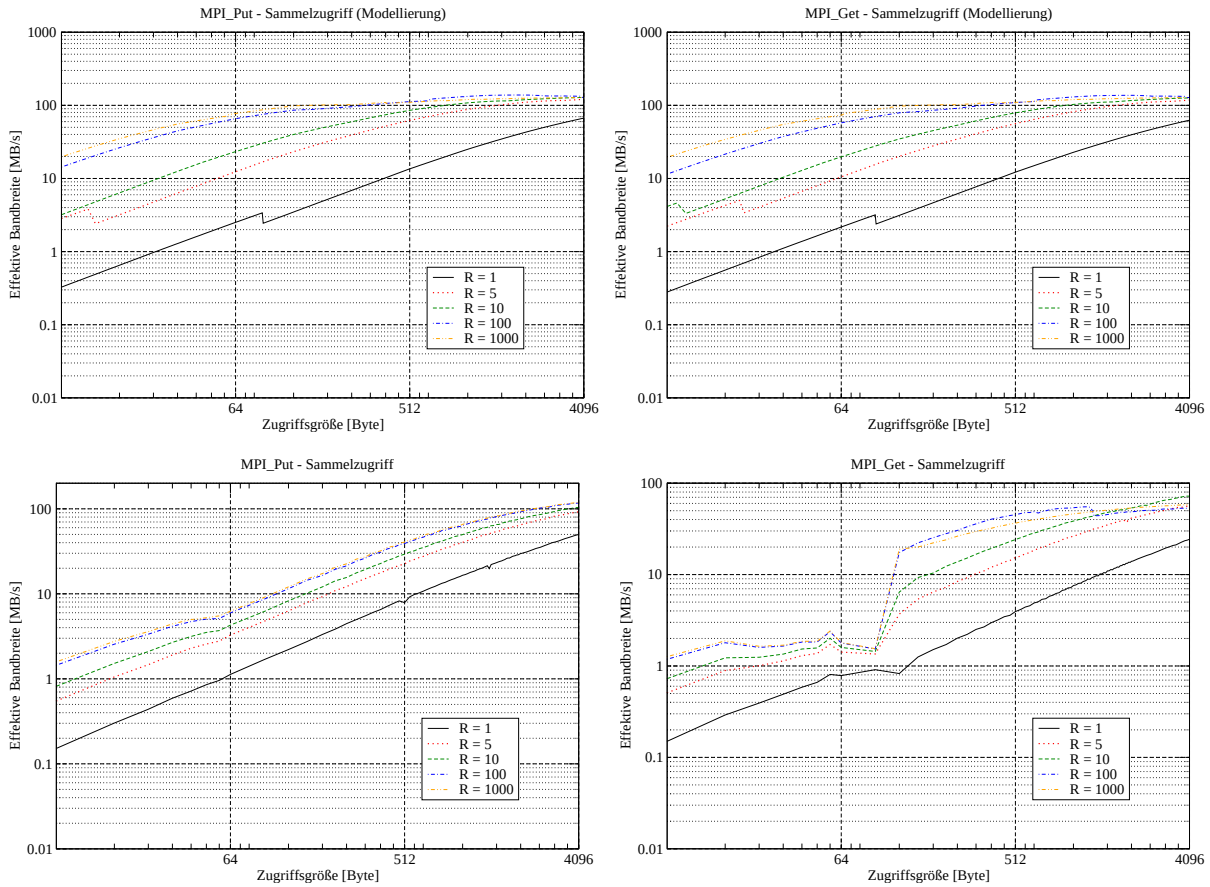


Abbildung 5.27: Modell und Experiment für den Sammelzugriff bei einseitiger Kommunikation (Parameter R für Zahl der Zugriffe pro Epoche)
Links: *put*-Operationen; *Rechts: get*-Operationen

Zaunsynchronisation. Die Zaunsynchronisation erfolgt mittels Barrierensynchronisation, wobei beim Eintritt eine und beim Abschluß zwei Synchronisationen erforderlich sind. Abhängig von der Zahl *N* der Prozesse im Kommunikator (bei der Zaunsynchronisation ist jeder Prozeß sowohl UP als auch ZP) ergibt dies (zur Barrierenverzögerung $\tau_{B_{shmem}}$ vergl. Kapitel 5.2.2.1):

$$(5.14)\ l_{zsync}(N) = 3 \cdot \tau_{B_{shmem}}(N)$$

Gruppensynchronisation. Bei der Gruppensynchronisation öffnen N_{ZP} Zielprozesse ihr lokales Window für Zugriffe von N_{UP} Ursprungsprozessen. Die dazu notwenige Kommunikation wird als sequentieller Baum abgewickelt. Am Beginn der Epoche schickt jeder ZP an alle UP eine Kontrollnachricht *is_posted*, während zum Abschluß der Epoche jeder UP allen ZP über eine Kontrollnachricht *is_complete* den Abschluß seiner Zugriffsepoche mitteilt. Es wird davon ausgegangen, daß die Sendeoperationen jedes Prozesses (vergl. Kapitel 6.5.2, Konflikte einzelner SCI-Pakete) unabhängig voneinander sind und sich somit vollständig überlappen. Damit ergibt sich für die Zeitdauer der Gruppensynchronisation:

$$(5.15)\ l_{gsync} = (N_{UP} + N_{ZP}) \cdot L_{ctrl-pp}$$

Passivsynchronisation. Die Passivsynchronisation wird stets zwischen einem UP und einem ZP durchgeführt und besteht zum Beginn und zum Abschluß der Zugriffsepoche aus jeweils einem Lock-Erwerb bzw. dessen Abgabe. Die Locks in SCI-MPICH arbeiten auf gemeinsamem Speicher nach dem Algorithmus von Lamport *[104]* und weisen die folgenden minimalen Latenzen auf:

Erwerb eines exklusiven Locks: $l_{lock_{excl}} = l_{lb} + l_{sb} + 3 \cdot (l_{rr} + l_{rs})$

Erwerb eines Lese-Locks: $l_{lock_{read}} = 2 \cdot (l_{lb} + l_{sb}) + 3 \cdot l_{rr} + 6 \cdot l_{rs}$

Abgabe eines exklusiven Locks: $l_{unlock_{excl}} = l_{sb} + l_{rr} + 2 \cdot l_{rs}$

Abgabe eines Lese-Locks: $l_{unlock_{read}} = l_{lock_{read}}$

Diese Latenzen sind identisch mit den Latenzen für den Beginn und Abschluß einer Zugriffsepoche.

Synchronisation einer Zugriffsepoche mit Schreib- und Lesezugriff eines einzelnen UP:

$$(5.16)\ l_{psync_{excl}} = l_{lock_{excl}} + l_{unlock_{excl}}$$

Synchronisation einer Zugriffsepoche mit ausschließlichem Lesezugriff durch mehrere UP:

$$(5.17)\ l_{psync_{read}} = 2 \cdot l_{lock_{read}}$$

Dies ergibt für die Plattform P3 Werte von $l_{psync_{excl}} = 30,8\mu s$ sowie $l_{psync_{read}} = 53,2\mu s$.

Synchronisationsverzögerung. Die Verzögerungen, die allein durch die Synchronisation (also ohne die Abwicklung von Kommunikation innerhalb der Synchronisation) entstehen, wurden für verschiedene Prozeßzahlen und Synchronisationsbeziehungen experimentell ermittelt. Während bei der Zaunsynchronisation stets alle an das Window gekoppelten Prozesse miteinander synchronisiert werden, ist die Zahl der tatsächlich miteinander zu synchronisierenden Prozesse bei den anderen Synchronisationsvarianten variabel. Folgende Fälle wurden untersucht:

- *Passivsynchronisation 1-auf-1:* Prozeß p, $0 \le p < n$, nutzt einen Lock auf das Window bei

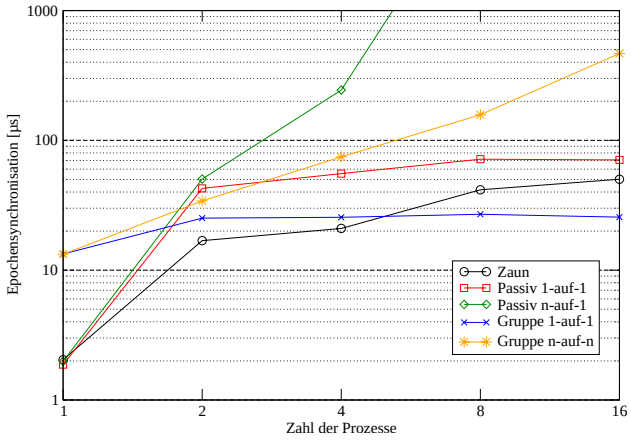


Abbildung 5.29: Gesamtzeit für Epochensynchronisation für unterschiedliche Synchronisationsverfahren bei einseitiger Kommunikation

Prozesse	**Passivsynchr.**		**Gruppensynchr.**		**Zaunsynchr.**
n	1-auf-1	n-auf-1	1-auf-1	n-auf-n	n
1	1,9	1,9	13,3	13,3	2,0
2	42,6	50,3	25,2	34,2	16,9
4	55,4	244,1	25,5	74,7	21.0
8	71,6	11000	27,0	157,5	41,6
16	70,7	-	25,6	466,7	50,1

Tabelle 5.2: Skalierung der unterschiedlichen Synchronisationsmechanismen für einseitige Kommunikation in SCI-MPICH ohne implizite Kommunikation (alle Angaben in μs)

Prozeß $p + 1 \lozenge n$

- *Passivsynchronisation n-auf-1:* Alle Prozesse nutzen einen Lock auf das Window bei Prozeß 0
- *Gruppensynchronisation 1-auf-1:* Prozeß p, $0 \leq p < n$, öffnet sein Window für Prozeß $p + n - 1 \lozenge n$ und erwirbt Zugriff auf das Window bei Prozeß $p + 1 \lozenge n$
- *Gruppensynchronisation n-auf-n:* Jeder Prozeß öffnet sein Window für alle anderen Prozesse, und erwirbt entsprechend Zugriff auf das Window bei allen Prozessen.

Die Ergebnisse sind in Tabelle 5.2 und graphisch in Abb. 5.29 dargestellt. In den Zeiten ist jedoch ein wesentlicher Anteil an Synchronisationsverzögerungen nicht erfaßt, die in Applikationen auftreten. Dies sind die Wartezeiten auf Prozesse, die noch nicht mit der Synchronisation begonnen haben. Daher sind diese Zahlen vor allem als Orientierung zu sehen, wie hoch der Aufwand für ein bestimmtes Synchronisationsverfahren ist, jedoch nicht, wie lange die Ausführung im Kontext einer Applikation dauern wird.

Im Vergleich mit den Werten aus der Modellierung liegen die gemessenen Werte im erwarteten Bereich. Unter den zuvor gemachten Einschränkungen, die im folgenden Abschnitt vertieft behandelt werden, ist die Zaunsynchronisation ein günstiger Synchronisationsmechanismus, dem nur die Gruppensynchronisation mit eingeschränkter Gruppengröße vorzuziehen ist. Dies ist wichtig, da die Zaunsynchronisation wegen ihrer einfachen Handhabung analog zur Barriere

oft eingesetzt wird. Wenn Gruppensynchronisation analog zur Zaunsynchronisation verwendet wird, steigt die Verzögerungszeit superlinear mit der Zahl der Prozesse an, obwohl die Kommunikationskomplexität nur linear wächst. Dies deutet auf Sättigungseffekte hin; hier könnte wahrscheinlich noch optimiert werden. Dies ist jedoch nicht die vorgesehene Einsatzart für die Gruppensynchronisation. Die Passivsynchronisation hat eine relativ hohe Verzögerung. Vor allem ist es wichtig, daß es nicht zum Wettbewerb mehrerer Prozesse um einen Lock kommt, da die Synchronisationszeit bei mehr als vier Prozessen extrem ansteigt.

5.2.4 Kommunikation bei gleichzeitiger Berechnung

Die rein synthetischen Messungen der Übertragungsprotokolle für einseitige Kommunikation im vorhergehenden Abschnitt erfolgten unter der Bedingung, daß beide Prozesse zugleich kommunizieren bzw. innerhalb der MPI-Bibliothek ein- und ausgehende Nachrichten verarbeiten. Dieses Muster soll als *enge Kopplung*[1] bezeichnet werden (siehe Abb. 5.30 *links*). Ein derartiger Ablauf mit Phasen von ausschließlicher Kommunikation könnte auch mittels herkömmlicher, zweiseitiger Kommunikation erfolgen, indem die Nachrichten mit den auszutauschenden Daten um die notwendigen Informationen wie Auswahl einer bestimmten Datenstruktur (entsprechend dem Window) und Plazierung innerhalb dieser Struktur ergänzt werden. Insofern nimmt die Implementation einseitiger Kommunikation innerhalb der MPI-Bibliothek dem Programmierer in erster Linie Arbeit ab, ohne jedoch einen Leistungsgewinn zu bringen. Im Gegenteil, häufig ist die anfallende einseitige Kommunikation langsamer als der Austausch der gleichen Daten mittels zweiseitiger Kommunikation (sofern der ZP weiß, welche Daten der UP benötigt).

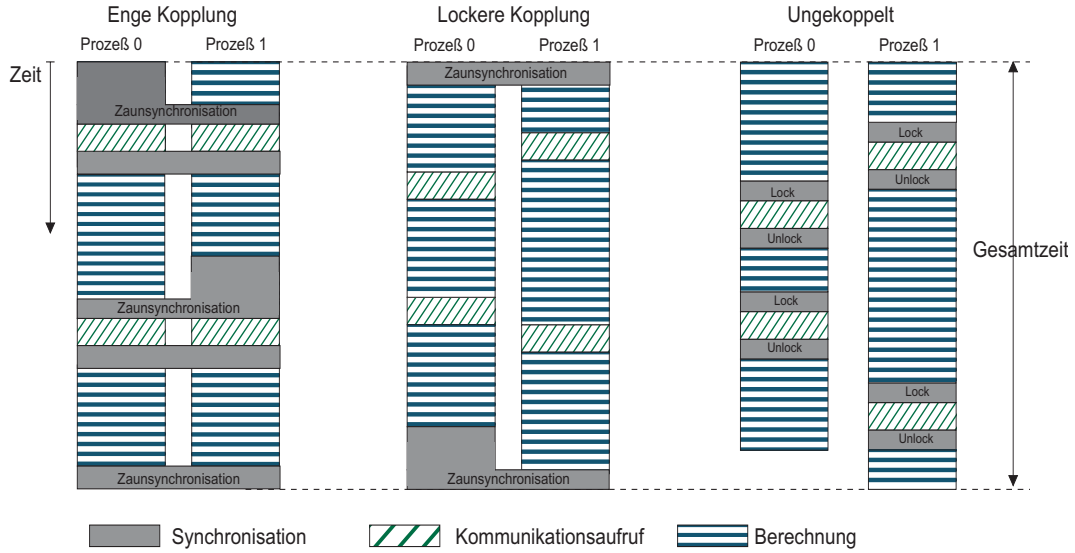


Abbildung 5.30: Unterschiedliche Muster des Ablaufs von Berechnung, Kommunikation und Synchronisation für einseitige Kommunikation
Links: enge Kopplung *Mitte:* Lockere Kopplung *Rechts* Ungekoppelt

Abbildung 5.30 zeigt zwei weitere Muster des Ablaufs von Berechnung, Kommunikation und Synchronisation, die bei Nutzung von einseitiger Kommunikation möglich sind:

- Falls während der Berechnung Datensätze anfallen, die von anderen Prozessen erst zu einem späteren Zeitpunkt benötigt werden, oder falls Anforderungen für Datensätze ermittelt werden, die ebenfalls erst zu einem späteren Zeitpunkt benötigt werden, kann von der engen zur

1. Die Kopplung bezieht sich hier auf die Synchronisation durch Mechanismen der einseitigen Kommunikation.

lockeren Kopplung (Abb. 5.30 *mitte*) übergegangen werden. Die Lockerung besteht darin, daß die Kommunikationsaufrufe unmittelbar abgesetzt werden, die Synchronisation jedoch erst später erfolgt. Erst dann ist garantiert, daß die Kommunikationsoperationen abgeschlossen sind. Sowohl die enge als auch die lockere Kopplung lassen sich mit aktiver Synchronisation, etwa der Zaunsynchronisation, umsetzen.

- Falls die Kommunikationsanforderungen bei der lockeren Kopplung jedoch unmittelbar umgesetzt werden sollen (weil die Daten von dem jeweiligen Prozeß zur Weiterverarbeitung benötigt werden), ohne daß es zu einer expliziten Synchronisation mit dem ZP kommen soll, muß passive Synchronisation verwendet werden. Für diesen Fall besteht keine Kopplung zwischen den Prozessen (Abb. 5.30 *rechts*). Damit läßt sich ein Programmiermodell realisieren, bei dem alle Prozesse auf gemeinsamen Datenstrukturen arbeiten, wie es auch bei gemeinsamem Speicher der Fall ist (siehe Kapitel 2.4.1).

Zur Evaluierung der Leistung der vorgestellten Verfahren in SCI-MPICH in diesen Nutzungsfällen der einseitigen Kommunikation wurde der Benchmark *progress* entwickelt, der es erlaubt, diese Fälle zu simulieren. Dazu wird als Berechnung die Skalierung eines Vektors[1] durchgeführt, die sowohl das Speichersystem als auch die CPU belastet. Der Benchmark führt eine Zahl von Zyklen durch. Pro Zyklus wird für die lockere Kopplung eine Zugriffsepoche mit wiederum einer bestimmten Zahl von Kommunikationsaufrufen abgewickelt; für den ungekoppelten Fall wird in dem Zyklus die entsprechende Zahl von passiv synchronisierten Zugriffen durchgeführt. Neben den Kommunikationsaufrufen finden abwechselnd Berechnungsphasen mit einer vorgegebenen mittleren Zahl von Iterationen statt. Um die für dynamische, nicht eng gekoppelte Anwendungen typischen Lastungleichgewichte zu simulieren, wird von der mittleren Zahl von Iterationen regelmäßig nach oben und unten abgewichen. Gemessen wird jeweils die Gesamtausführungszeit (die letztlich relevant ist) sowie die akkumulierten Zeiten zur Abwicklung der Kommunikationsaufrufe und der Synchronisation.

Für die im folgenden vorgestellten Ergebnisse des *progress*-Benchmarks für 2 Prozesse wurden 10 Zyklen mit jeweils 10 *put*-Zugriffen von 32 Byte Größe (entspricht 4 `double`-Werten) durchgeführt. Die mittlere Zahl der Iterationen pro Berechnungsphase wurde von 10^1 bis $3 \cdot 10^4$ variiert. Die Kommunikation wurde über direkten Zugriff auf SCI-Speichersegmente (in den Diagrammen als D abgekürzt), über emulierten Zugriff ohne und mit Signalisierung (E bzw. ES) sowie über Sammelzugriff ohne und mit Signalisierung (S bzw. SS) abgewickelt. Als aktives Synchronisationsverfahren kam die Zaunsynchronisation zum Einsatz.

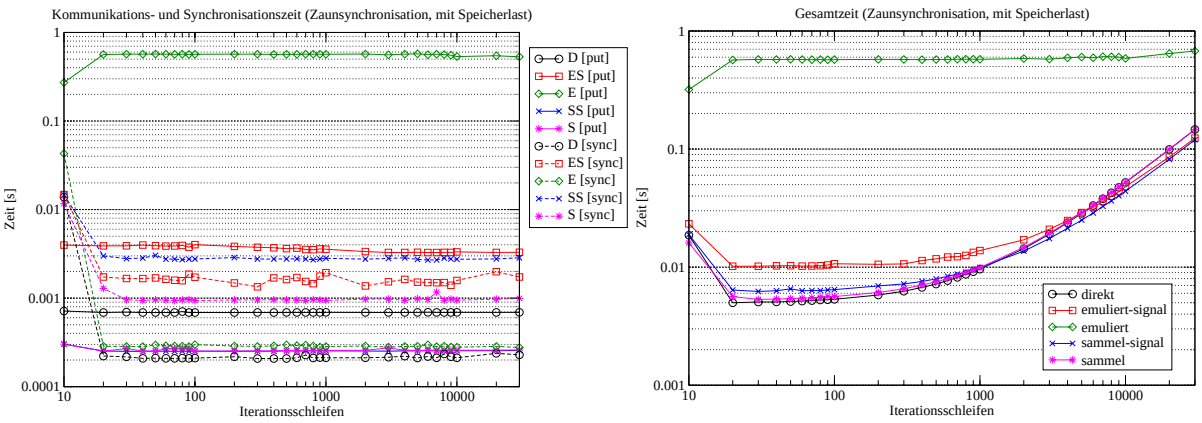


Abbildung 5.31: Ergebnisse des *progress*-Benchmark für lockere Kopplung mit Zaunsynchronisation
Links: Latenzen für Kommunikation und Synchronisation *Rechts:* Gesamtabarbeitungszeit für Kommunikation, Synchronisation und DAXPY-Berechnung

1. Sogenannte DAXPY Operation; entspricht der Triade beim STREAM Benchmark, siehe Kapitel 2.1.2

In Abb. 5.31 sind links die Zeiten zur Abwicklung der Kommunikationsaufrufe (durchgezogene Linie) und der Synchronisationsvorgänge (entsprechende Symbole mit gestrichelter Linie) dargestellt. Diese Zeiten sind nicht unmittelbar abhängig von der Zahl der DAXPY-Iterationen, da der einzelne Kommunikationsaufruf beim UP jedesmal gleichartig, ohne direkten Einfluß des ZP, abläuft. Nur die Synchronisationsaufrufe benötigen beim ersten Aufruf eine höhere Zeitdauer, da in diesem Moment die notwendigen entfernten Speichersegmente für die Job-Zähler eingeblendet werden. Die kürzesten Zeiten für die Kommunikationsaufrufe treten bei den beiden Varianten des Sammelzugriffs auf. Tatsächlich werden hierbei ja nur lokal die Referenzen auf die bei der Synchronisation abzuwickelnde Kommunikation gespeichert. Dies dauert je Aufruf etwa $1{,}5\mu s$. Die Zeit für die Abwicklung der direkten Kommunikation liegt mit $7\mu s$ darüber; allerdings ist danach das Datum bereits im Window des ZP abgelegt. Der emulierte Zugriff mit Signalisierung benötigt pro Aufruf etwa $40\mu s$, wobei etwa $35\mu s$ auf die Signalisierung entfallen. Extrem hoch ist mit $5500\mu s$ die Zeit, die beim emulierten Zugriff pro Aufruf benötigt wird.

Bei den Synchronisationszeiten ist der direkte Zugriff mit $20\mu s$ am schnellsten, da hierbei nur zwei Barrierensynchronisationen erfolgen müssen. Die Jobzähler stehen jeweils auf 0, so daß hier keine Wartezeiten auftreten. Mit $30\mu s$ ähnlich kurz ist die Synchronisation für den emulierten Zugriff, da hier die Kommunikation bereits abgewickelt ist, wenn die Synchronisation durchgeführt wird. Gleiches ist gerade auch beim emulierten Zugriff mit Signalisierung der Fall. Die höhere Synchronisationsdauer von gut $100\mu s$ kann daher nicht in der Durchführung der Kommunikation innerhalb des Synchronisationsaufrufs begründet sein. Vielmehr müssen dafür Eintrittsverzögerungen in die Barriere (siehe Kapitel 6.5) verantwortlich sein, die durch die mittels Signalisierung ausgelösten Aktivitäten des Device-Threads (siehe Kapitel 7.1.1) entstehen, da dieser sich dabei mit dem Applikations-Thread (der ja auch Sendeoperationen ausführen muß) beim Zugriff auf interne Datenstrukturen synchronisieren muß. Die im Gegensatz zu allen anderen Synchronisationsverfahren nicht-konstanten Synchronisationszeiten stützen diese Vermutung. Die Synchronisationszeiten für die Sammelzugriffe liegen etwas höher, da hier die gesamte Kommunikation abgewickelt werden muß - jedoch in diesem Fall in Form jeweils nur einer Nachricht, die zuvor beim UP gepackt und beim ZP wieder entpackt werden muß. Zusammen mit den Barrierensynchronisationen und Zugriffen auf die Job-Zähler dauert dies ohne Signalisierung etwa $95\mu s$. Die Signalisierung ist in diesem Fall überflüssig, da der jeweilige ZP bei der Zaunsynchronisation ohnehin auf eingehende Nachrichten prüft, wenn der UP die gesammelten Kommunikationszugriffe überträgt.

Da die Zeit für die Berechnungen in allen Fällen gleich ist, spiegeln die unterschiedlichen Gesamtzeiten in Abb. 5.31 *rechts* die Differenzen in den Summen aus Kommunikation und Synchronisation wider. Der direkte Zugriff weist die kürzeste Gesamtlaufzeit aus, wobei die Sammelzugriffe nur unwesentlich mehr Zeit benötigen. Dies liegt in der höheren Bandbreite der einzelnen Sammelnachricht gegenüber der größeren Zahl von direkten Zugriffen begründet. Mit zunehmender Größe der Daten der einzelnen Zugriffe wird sich der Vorteil des direkten Verfahrens erhöhen, da sich dann die Differenz der Bandbreiten für die einzelnen direkten Zugriffe und die Sammelnachricht verringert. Bei den signalisierenden Verfahren kommt es durch die Überlappung von Kommunikation (durch den Device-Thread) und Berechnung (durch den Applikations-Thread) auf den verwendeten SMP-Knoten dazu, daß mit zunehmender Zahl der Iterationen der zusätzliche Aufwand der Signalisierung kompensiert wird.

Während bei der lockeren Kopplung die Synchronisation weiterhin kollektiv durch alle beteiligten Prozesse erfolgt, ist dies beim ungekoppelten Verfahren nicht der Fall. Zur Absicherung des kritischen Abschnitts, des Zugriffs auf das Window beim ZP, kommt dazu bei allen Verfahren ein Lock im gemeinsamen Speicher zum Einsatz. Eine derartig eingeleitete Zugriffsepoche kann jedoch beim UP nicht abgeschlossen werden, solange der Zugriff nicht auch tatsächlich

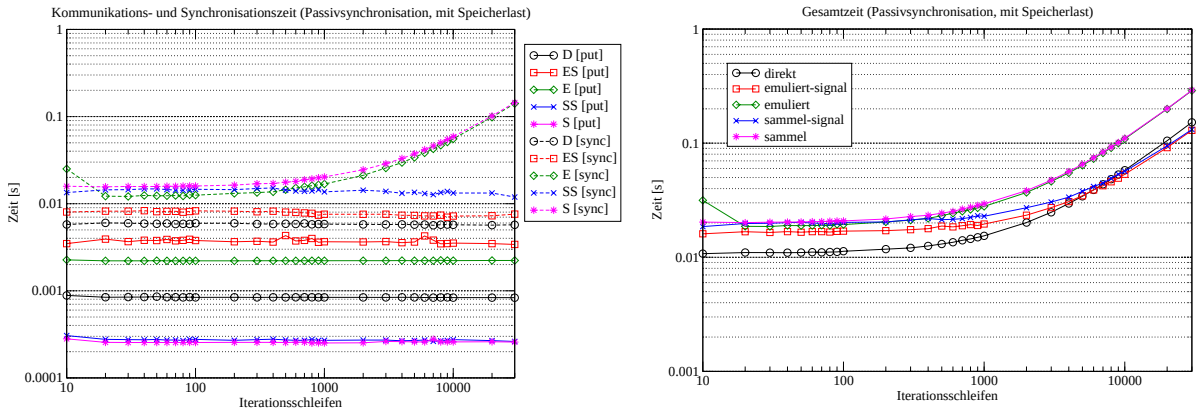


Abbildung 5.32: Ergebnisse des *progress* Benchmark für ungekoppelten Ablauf mit Passivsynchronisation
Links: Latenzen für Kommunikation und Synchronisation *Rechts:* Gesamtabarbeitungszeit für Kommunikation, Synchronisation und DAXPY-Berechnung

im Window des ZP vollzogen wurde. Ohne die Mitwirkung des ZP kann dies nur beim direkten Zugriff erfolgen. Die Zeiten für Kommunkationsaufrufe und Synchronisation für diesen Fall (Abb. 5.32 *links*) machen dies deutlich: Während die Dauer der Kommunikationsaufrufe gegenüber der lockeren Kopplung nahezu gleich bleibt, steigen für die nicht-signalisierenden Varianten des emulierten Zugriffs und des Sammelzugriffs die Synchronisationszeiten proportional zur Zahl der Iterationen bei der DAXPY-Berechnung an. Dies liegt daran, daß der UP jeweils warten muß, bis der ZP seinerseits einen Kommunikationsaufruf durchführt, dabei die eingegangenen Nachrichten verarbeitet und den Job-Zähler beim UP inkrementiert. Allgemein sind die Synchronisationsverzögerungen höher als bei der Zaunsynchronisation, da gemäß der Modelle für die Synchronisationsverfahren (Kapitel 5.2.3.4) ein Lock langsamer abgewickelt wird als eine Barriere.

Dies wirkt sich unmittelbar auf die mit den verschiedenen Verfahren erreichte Gesamtzeit aus: Für kleine Iterationszahlen wird mit direktem Zugriff die deutlich kürzeste Gesamtzeit erreicht. Mit zunehmender Iterationszahl amortisiert sich der zusätzliche Aufwand der signalisierenden Verfahren, und diese erreichen die gleichen Gesamtzeiten wie der direkte Zugriff. Die Differenz zwischen der Gesamtzeit dieser und der anderen, nicht-signalisierenden Verfahren steigt jedoch bei großen Iterationszahlen proportional an.

Diese Untersuchungen haben gezeigt, daß für komplexere, nicht gekoppelte Fälle von Berechnung und einseitiger Kommunikation der direkte Zugriff auf entfernten Speicher besonders bei fein-granularem Zugriff deutliche Vorteile bringt. Die Signalisierung im Zusammenhang mit herkömmlichen, auf zweiseitiger Kommunikation aufbauenden Verfahren wird erst bei grob-granularem Zugriffsmuster mit einem geringen Zeitanteil der Kommunikation ähnlich effizient. Herkömmliche Verfahren, die einzig auf die zweiseitige synchrone Kommunikation bauen, sind nicht geeignet, einseitige Kommunikation so einzusetzen, wie es das Programmiermodell des Arbeitens auf gemeinsamen Datenstrukturen erfordert.

5.2.5 Andere Arbeiten zu einseitiger Kommunikation

Das Problem, ohne direkten Zugriff auf entfernten Speicher *und* ohne Signalisierung entfernter Prozesse effiziente einseitige Kommunikation zu implementieren, wurde bereits in *[95]* erkannt. Die dort vorgestellte Implementation von *put*/*get*-Operationen auf einem Myrinet-gekoppelten SMP-Cluster benötigt einen Thread, der in Konkurrenz zur Applikation den Netzwerkadapter auf eingehende Nachrichten überprüft. Dies stellt keine zufriedenstellende Lösung dar. Auch die in *[92]* vorgestellte, ebenfalls auf Myrinet basierende Lösung kann einseitige Kommunikation ausschließlich durch Nutzung zweiseitiger Kommunikation emulieren, was zu hohen La-

tenzen bei kleinen Zugriffsgrößen führt. Eine Einbindung in MPI ist dabei nicht erfolgt.

In *[97]* wird die Implementation von einseitiger Kommunikation in der MPI-Implementation MPI/SX für die SX-Vektorrechner von NEC beschrieben. Diese Systeme verfügen ebenfalls über die Möglichkeit, zwischen den Rechenknoten direkte Speicherzugriffe auf speziell eingerichtete Speicherbereiche durchzuführen. Die verwendeten Verfahren sind ähnlich, unterscheiden sich jedoch in wichtigen Details. So werden keine Job-Zähler im gemeinsamen Speicher verwendet; stattdessen wird beim Abschluß einer Zugriffsepoche mit aktiver Synchronisation die Zahl der ausstehenden Transaktionen mittels Aufruf von `MPI_Allreduce` ermittelt. Dies ist bei N Prozessen ein zusätzlicher Aufwand der Ordnung $2 \cdot \log N$. Ein Verfahren, das dem hier vorgestellten Sammelzugriff entspricht, wurde nicht implementiert; jeder Zugriff auf nichtglobalen Speicher generiert eine einzelne Nachricht. Bei der Leistungsevaluierung erkennen die Autoren ebenfalls das Problem, daß die Leistung einseitiger Kommunikation nicht eindeutig definiert ist. Sie messen den Datenaustausch eines Prozesses mit n Nachbarprozessen und gehen dabei von dem Modell der engen Kopplung aus. Dies führt dazu, daß in dem Vergleich mit zweiseitiger Kommunikation diese zumeist deutlich schneller abgewickelt wird. Bei kleinen Datenmengen ist zweiseitige Kommunikation um eine Größenordnung schneller. Dies macht erneut deutlich, daß dies nicht das geeignete Kommunikationsmuster für einseitige Kommunikation ist. Da in MPI/SX jedoch auch keine Signalisierung entfernter Prozesse implementiert ist und Lock-Operationen über Nachrichten und nicht über gemeinsamen Speicher abgewickelt werden, würde die Leistung dieser Implementation bei lockerer Kopplung oder kopplungslosem Betrieb die in Kapitel 5.2.4 gezeigten Effekte aufweisen. Von der gleichen Gruppe wird in *[98]* die Implementation von einseitiger Kommunikation auf einem VIA-gekoppelten SMP-Cluster vorgestellt, die analog zu MPI/SX implementiert und evaluiert wurde.

Die Implementation von einseitiger Kommunikation in den *Sun Clustertools [43]*, der MPI-Implementation von Sun, wurde generisch mittels zweiseitiger MPI-Kommunikation realisiert *[99,100]*. Dazu wurde ebenfalls ein Thread verwendet. Da sich die Leistungsevaluation durch Nutzung des *Pallas MPI Benchmarks (PMB) [153]* jeweils auf reine Ping-Pong-Transfers mit Zaunsynchronisation beschränkt, ist die Aussagekraft der ermittelten Werte beschränkt. Die Tatsache, daß bei diesen Tests jedoch die Latenz für eine *put*-Operation um bis zu zwei Größenordnungen oberhalb einer äquivalenten *send-recv*-Operation liegt, zeigt, daß der generische Implementationsansatz in Verbindung mit der Nutzung von TCP/IP zur Intra-Knoten-Kommunikation effiziente einseitige Kommunikation nicht ermöglicht.

In *[96]* wird die Leistung von einseitiger Kommunikation via SHMEM und MPI-2 auf zwei Systemen verglichen, die globale Adreßräume mit transparentem Zugriff unterstützen (Cray T3E und SGI Origin 2000). Allerdings wird in dieser Untersuchung nicht auf Implementationsaspekte eingegangen. Die Messungen erfolgen wiederum nur im eng gekoppelten Betrieb mit Zaunsynchronisation. Bei den Untersuchungen wird deutlich, daß der Betrieb ohne explizite Synchronisation, wie ihn SHMEM ermöglicht, deutliche Leistungsvorteile bietet. Für die Implementationen desselben Kommunikationsmusters auf identischer Hardware war die Nutzung des SHMEM-API bei kleinen Nachrichtenlängen teilweise um Größenordnungen schneller. Hier stellt sich die Frage, warum die MPI-Implementation selbst offensichtlich keinen (effizienten) Gebrauch des SHMEM-API macht. Dies trifft insbesondere für die Tests auf der SGI Origin 2000 zu.

5.2.6 Einfluß einseitiger Kommunikation auf Applikationsleistung

Molekulardynamik-Simulationen sind ein geeigneter Applikationstyp für einseitige Kommunikation. Die Moleküle werden häufig in unstrukturierten Gittern gespeichert, und für die einzelnen Moleküle werden die Wechselwirkungskräfte mit anderen, mehr oder weniger weit entfernten Nachbarmolekülen berechnet. Dazu ist ein Datenaustausch zwischen den durch Par-

titionierung entstandenen Teilen des gesamten Simulationsraumes notwendig. Es existieren dazu parallel arbeitende Verfahren sowohl basierend auf dem Programmiermodell des gemeinsamen Speichers *[147]* als auch des Nachrichtenaustauschs *[148]*.

In *[151]* wird das System *Protomol* zum Aufbau von Molekulardynamik-Simulationen vorgestellt, auf dessen Basis ein Simulationscode *Springs* entstanden ist. Dieser Code kann sowohl zweiseitige als auch einseitige Kommunikation nutzen. Die Evaluierung des Skalierungsverhaltens dieses Codes bei der Lösung derselben Probleme mit den unterschiedlichen Kommunikationsmodellen in *[150]* zeigt, daß der effiziente, nämlich nicht gekoppelte Betrieb von einseitiger Kommunikation für dieses Problem Leistungsgewinne von 10 bis 70% erbringt. Die Gewinne fallen um so größer aus, desto irregulärer die Zugriffe sind. Dies belegt, daß das Leistungspotential von einseitiger Kommunikation nicht in höherer Kommunikationsleistung des einzelnen Zugriffs liegt, sondern in der Vermeidung von Synchronisationsverlusten, die bei zweiseitiger Kommunikation in diesen Fällen auftreten.

KAPITEL 6

Kollektive Operationen

Eine *kollektive Operation* im Sinne des MPI-Standards ist ein Kommunikationsvorgang, bei dem Daten nicht nur von einem Quellprozeß zu einem anderen Zielprozeß bewegt werden, sondern bei dem eine Gruppe von *N* Prozessen Daten nach einer bestimmten Vorschrift austauscht. Die *1:1*-Kommunikation von Sender und Empfänger wird also erweitert auf *1:N*-, *N:1*- oder *N:N*-Kommunikation. Eine kollektive Operation erfordert die Beteiligung aller Prozesse des verwendeten Kommunikators und kann daher für mindestens einen Prozeß nicht beendet sein, bevor nicht alle Prozesse der Gruppe in die kollektive Operation eingetreten sind. Jedoch kann eine kollektive Operation für einen Prozeß früher abgeschlossen sein als für einen anderen.

Die Leistung der kollektiven Operationen einer MPI-Implementierung kann einen wichtigen Einfluß auf die Leistung einer parallelen Applikation haben. Kollektive Operationen werden u.a. häufig in einem entsprechend regulären Algorithmus verwendet, um zwischen den Iterationsstufen Ergebnisse unter den Prozessen auszutauschen. Oftmals handelt es sich hierbei um große Datenmengen. Die kollektiven Operationen stellen hohe Anforderungen an das Kommunikationssystem, da in einem Moment viele Prozesse gleichzeitig Daten senden (möchten). Die hierbei auftretende Last kann zu Leistungseinbrüchen führen, wenn einzelne Komponenten des Kommunikationsnetzes überlastet werden. Andererseits bietet dieses Szenario auch vielfältige Möglichkeiten zur algorithmischen und implementationstechnischen Optimierung.

Es soll von *N* Prozessen ausgegangen werden, die an der kollektiven Operation teilnehmen. *N* läßt sich darstellen als

$$(6.1)\quad N = P + Q, \text{ mit } (P, Q, n \in \mathbb{N}),\ P = 2^n,\ 2^{n+1} > N \text{ und } P > Q$$

Diese *N* Prozesse haben Prozeßränge *r* mit $r \in \{0, \ldots, N-1\}$. Bei *1:N*- und *N:1*-Kommunikation wird der einzelne Prozeß, der Datenquelle oder -senke der Operation ist, als *Wurzelprozeß* bezeichnet und habe den Rang $r = 0$. Im Falle eines Wurzelprozesses mit Rang $r \neq 0$ kann durch eine einfache Abbildung eine neue Rangverteilung erfolgen, so daß diese Bedingung keine Einschränkung der Allgemeingültigkeit darstellt. Im Fall $Q = 0$ soll von einer *2-exponentiellen Prozeßzahl* gesprochen werden. Der Fall $Q = 2 \cdot p$ $(p \in \mathbb{N})$ ist entsprechend eine *gerade Prozeßzahl*, während für $Q = 2 \cdot p + 1$ eine *ungerade Prozesszahl* vorliegt.

Weiterhin wird davon ausgegangen, daß ein Prozeß alle Sendevorgänge sequentiell hintereinander durchführt, so daß keine Überlappung von mehreren Sendevorgängen auftritt. Auch diese Voraussetzung beinhaltet keine Einschränkung der Allgemeingültigkeit, da jeder Prozeß tatsächlich nur einen Datenstrom zu einem Zeitpunkt erzeugen kann. Eine Zahl von *s* ineinander verschachtelten Sendevorgänge würde daher (mindestens) die gleiche Zeit wie *s* sequentiell ausgeführte Sendevorgänge benötigen, sofern alle Zielprozesse empfangsbereit sind.

6.1 Basisalgorithmen

Zur Abwicklung von kollektiven Operationen existiert eine Reihe von Basisalgorithmen, die das Kommunikationsmuster beim Datenaustausch zwischen den Prozessen festlegen *[118,120]*. Diese werden hier mit ihrer allgemeinen Charakteristik für die Anwendung in einem SVS beschrieben; eine genauere Untersuchung erfolgt jeweils im Zusammenhang mit ihrer Anwendung für eine spezifische kollektive Operation.

6.1.1 Sequentieller Baum

Bei *1:N*-Kommunikation in Form eines sequentiellen Baumes schickt der Wurzelprozeß die

Daten nacheinander an alle Zielprozesse, während bei *N:1*-Kommunikation alle Prozesse mit den Quelldaten diese an den Wurzelprozeß senden, der diese nacheinander empfängt. Die Reihenfolge, in der die Daten bei der *N:1*-Kommunikation im Falle des *short*- oder *eager*-Protokolls tatsächlich in den Puffern des Empfangsprozeß ankommen, ist jedoch unbestimmt und hängt im wesentlichen vom Zeitpunkt ab, zu dem die einzelnen Prozesse die Daten abgesendet haben. Wenn in diesem Fall eine große Zahl von Prozessen zur gleichen Zeit Nachrichten an einen Prozeß schickt, kann es zur Ausbildung von kritischen Segmenten bzw. zur Überlastung des E/A-Systems des Zielknotens kommen, wodurch die Leistung reduziert wird. Die algorithmische Laufzeit dieses Verfahrens beträgt $O(N)$, da $N-1$ Sendevorgänge erforderlich sind.

6.1.2 Verketteter Baum

Der verkettete Baum läßt sich vorwiegend bei *1:N*-Kommunikation einsetzen. Dabei schickt der Wurzelprozeß seine Daten an Prozeß 1, dieser an Prozeß 2 usw., bis schließlich Prozeß $N-1$ die Daten von Prozeß $N-2$ empfängt. Die algorithmische Laufzeit dieses Verfahrens beträgt aufgrund der $N-1$ Sendevorgänge wiederum $O(N)$, wenn ein Prozeß eine empfangene Nachricht erst dann weiterleitet, wenn sie vollständig von ihm empfangen wurde. Für mehrere aufeinander folgende Operationen ergibt sich jedoch ein *Pipeline*-Effekt, da jeder Prozeß nur eine Nachricht verschicken muß. Dieser Fall dürfte jedoch in Applikationen eher selten auftreten. Besonders wirksam würde der Pipeline-Effekt jedoch, wenn er auch bei der Übertragung einer einzelnen Nachricht wirksam würde. Ein solches Verfahren wurde in bekannten MPI-Bibliotheken noch nicht implementiert. Für Rundsende- und Reduktionsoperationen in SCI-MPICH werden derartige Verfahren jedoch in Kapitel 6.6 und 6.7 vorgestellt.

6.1.3 Binärbaum

Ein Binärbaum kann sowohl für *1:N*- als auch für *N:1*-Kommunikation verwendet werden. Im Falle der *1:N*-Kommunikation empfängt außer dem Wurzelprozeß jeder Prozeß von einem anderen Prozeß und sendet an bis zu zwei weitere Prozesse. Bei einer 2-exponentiellen Prozeßzahl ergeben sich $2 \cdot ld(N+1)$ Kommunikationschritte; für allgemeine Prozeßzahlen ergeben sich $2 \cdot \lfloor ld(N+1) \rfloor - n \ (n \in \{1,2\})$ Kommunikationsschritte. Damit ist beim binären Baum jeder Prozeß in zwei Kommunikationsschritten ein Sendeprozeß, solange es genügend Prozesse gibt, die noch keine Daten empfangen haben. Die Zahl N_{send} der Sendeprozesse in jedem Kommunikationsschritt *i* beträgt daher (mit $N_{send}(1) = 1$ und $N_{send}(2) = 2$):

$$N_{send}(i) = N_{send}(i-1) + N_{send}(i-2) \text{ für } (i \geq 3) \quad (6.2)$$

Bei der Anwendung eines Binärbaums zur *N:1*-Kommunikation gelten bezüglich der Zahl der Kommunikationsschritte die gleichen Überlegungen, wenn die Sendevorgänge von jeweils zwei Quellprozessen an einen Zielprozeß sequentiell abgewickelt werden, d.h. die doppelte Zeit eines einzelnen Sendevorgangs benötigen.

6.1.4 Binomialbaum

Binomialbäume erreichen eine stärkere Steigerung der Zahl der sendenden Prozesse pro Kommunikationsschritt, indem ein Prozeß in jedem Kommunikationsschritt sendet, solange es Empfänger gibt. Um dies zu ermöglichen, wird jeder Empfänger im nächsten Kommunikationsschritt zum Wurzelprozeß eines Teilbaums. Damit verdoppelt sich in jedem Kommunikationsschritt i $(i \geq 1)$ die Zahl der sendenden Prozesse gemäß

$$N_{send}(i) = 2^{i-1} \quad (6.3)$$

Der entstehende Baum ist ein asymmetrischer Binärbaum, der jedoch für 2-exponentielle Prozeßzahlen nur $ld(N)$ Kommunikationsschritte benötigt. Die Bestimmung der Empfangsprozesse in jeder Stufe kann auf verschiedene Weisen erfolgen, die sich leicht in der Zahl der resultierenden Kommunikationsschritte für nicht-2-exponentielle Prozeßzahlen unterscheiden. (für ein Beispiel siehe Kapitel 6.6).

6.2 Austauschkommunikation

In vielen Fällen von kollektiver Kommunikation tritt das Kommunikationsmuster des *Austausches* auf. Dies bedeutet, das zwei Prozesse miteinander Daten austauschen müssen, die sie in gegenseitigen Sende- und Empfangsoperationen übertragen. Für dieses Kommunikationsmuster sind zwei unterschiedliche Abläufe denkbar, von denen der leistungsfähigste verwendet werden soll. Die Leistungsfähigkeit hängt jedoch von der genutzten Systemplattform, der Datenmenge und dem gewählten Übertragungsprotokoll ab und muß daher experimentell bestimmt werden.

Die beiden unterschiedlichen Kommunikationsabläufe sind einerseits das gleichzeitige (*parallele)* und andererseits das zeitlich nacheinander ablaufende *(sequentielle)* Senden und Empfangen beider Prozesse. In beiden Fällen wird vorausgesetzt, daß jeder Prozeß getrennte Sende- und Empfangspuffer hat. Die Übertragungsverfahren sind zum einen das *short*- und *eager*-Protokoll und zum anderen die verschiedenen Varianten des *rendez-vous*-Protokolls. Das nichtblockierende (PIO-Modus) und das direkt-übertragende *rendez-vous*-Protokoll (DMA-Modus) können sowohl für den sequentiellen als auch für den parallelen Ablauf verwendet werden, während das blockierende *rendez-vous*-Protokoll nur für den sequentiellen Ablauf verwendet werden kann. Zudem kann neben der Inter-Knoten-Kommunikation noch die Intra-Knoten-Kommunikation über lokalen gemeinsamen Speicher erfolgen.

```
falls (mein_rang < partner_rang)
     MPI_Send(sende_puffer)
     MPI_Recv(empfangs_puffer)
sonst
     MPI_Recv(empfangs_puffer)
     MPI_Send(sende_puffer)
```

```
MPI_Irecv(empfangs_puffer)
MPI_Isend(sende_puffer)
MPI_Waitall()
```

Abbildung 6.1: MPI-Umsetzung von sequentieller (*links*) und paralleler (*rechts*) Austauschkommunikation

Abbildung 6.1 zeigt die Umsetzung der beiden Abläufe in MPI-Kommunikationsaufrufe, mit denen die Leistungsfähigkeit untersucht wurde. Die Leistung wird als Bandbreite gemessen, indem die gesamte Datenmenge als Summe der zwei identisch großen Nachrichten der beiden Prozesse durch die Zeit geteilt wird, die vom Aufruf der ersten bis zum Abschluß der letzten Kommunikationsoperation vergeht. Die Ergebnisse der Messungen auf der P3-Systemplattform für die oben beschriebenen Kommunikationsabläufe und -protokolle sind in Abb. 6.2 dargestellt. Dabei wurden die Grenzen der verfügbaren Kommunikationsprotokolle *short*, *eager* und *rendez-vous* teilweise überlappt. Es wird offensichtlich, daß die parallele Austauschkommunikation in allen untersuchten Fällen überlegen ist. Dies trifft insbesondere auf die DMA-Übertragung großer Datenmengen mittels *rendez-vous*-Protokoll zu, ebenso auf das *eager*-Protokoll bis zu einer Puffergröße von 128kB gegenüber dem PIO-basierten *rendez-vous*-Protokoll in diesem Bereich. Dies ist auf die Verluste durch Synchronisationszeiten zurückzuführen, die beim *rendez-vous*-Protokoll auftreten.

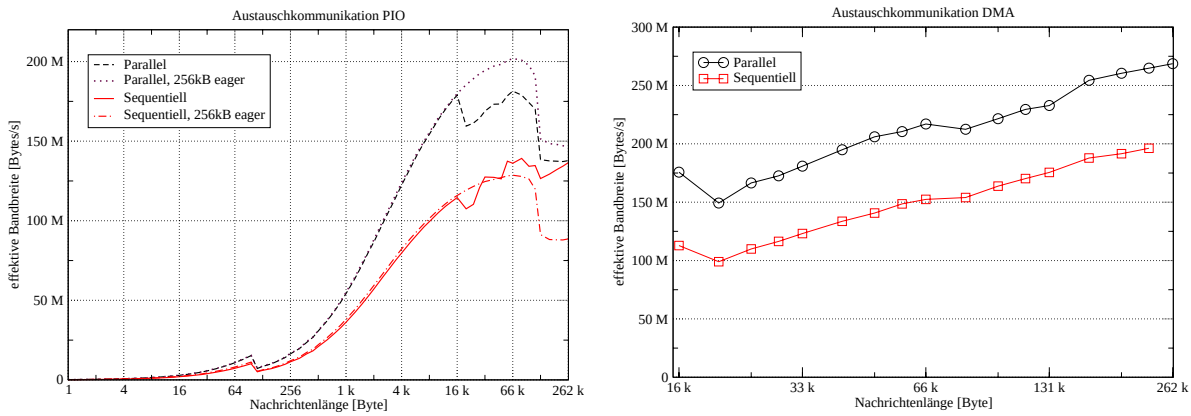


Abbildung 6.2: Bandbreite für Austauschkommunikation in Abhängigkeit von der Nachrichtengröße, dem Kommunikationsablauf und dem Übertragungsprotokoll
Links: PIO-Transfers mit Standardprotokollen; Variation mit 256kB großen Puffern im *eager*-Protokoll
Rechts: DMA-Transfers mit *rendez-vous*-Protokoll

6.3 Optimierungspotential

Für kollektive Operationen im allgemeinen (vergl. *[132,133]*) und auf SCI-gekoppelten Systemen im speziellen gibt es eine Reihe von potentiellen Optimierungsstrategien, die jeweils bei der Optimierung einer bestimmten kollektiven Operation berücksichtigt werden sollten. Der Grad der Wirksamkeit der möglichen Maßnahmen hängt jedoch auch immer stark von den Leistungseigenschaften des betrachteten Systems ab, insbesondere der Latenz und der Bandbreite der Kommunikationskanäle auf den verschiedenen Ebenen (Inter- und Intra-Knoten-Kommunikation). Auch spezielle, nicht erwünschte Eigenschaften im Verhalten der Kommunikationskanäle unter bestimmten Lastprofilen haben Einfluß. So kann eine Maßnahme, die auf einem System eine Leistungssteigerung bringt, auf einem anderen System kontraproduktiv sein.

6.3.1 Speicherhierarchie

Wenn die verbundenen Knoten Multiprozessorsysteme darstellen, kann man zwei Ebenen des Nachrichtenaustauschs zwischen Prozessen betrachten: Austausch zwischen Prozessen auf einem Knoten (Intra-Kommunikation) und Austausch zwischen Prozessen auf getrennten Knoten (Inter-Kommunikation). Die Intra-Kommunikation ist leistungsfähiger als die Inter-Kommunikation; nur bei der Kommunikation von Datenblöcken, die größer als die Speicher-Caches sind, kann die Inter-Knoten-Bandbreite höher sein als die Intra-Knoten-Bandbreite. Daneben können die beiden Kommunikationsformen weitgehend parallel ausgeführt werden, da sie verschiedene Ressourcen benutzen (E/A-Bus vs. Speicherbus) bzw. die gemeinsam benutzten Ressourcen wie der Speicherbus quasi-parallel benutzt werden können. Dies bietet die Möglichkeit, die kollektiven Operationen aufzuteilen in Intra- und Inter-Knoten-Operationen und diese geeignet zu überlappen. Dieser Effekt wird um so stärker, je größer die Zahl der Prozessoren auf einem SMP-Knoten ist, und je größer der Leistungsunterschied zwischen Intra- und Inter-Knoten-Kommunikation ausfällt. Bei COTS-Clustern sind jedoch üblicherweise nicht mehr als zwei Prozessoren pro Knoten anzutreffen, und das Speichersystem skaliert typischerweise auch nicht mit diesen zwei Prozessoren. Damit liegt die Leistung der Inter-Knoten-Kommunikation in der gleichen Größenordnung und zum Teil noch über der Leistung der Intra-Knoten-Kommunikation. Dadurch erzielen Verfahren, die sich für gekoppelte SMP-Knoten mit bis zu 64 Prozessoren pro Knoten *[121]* oder gar für über Weitverkehrsnetze gekoppelte Systeme *[116]* eignen, auf COTS-Clustern weniger Wirkung. Stattdessen gilt es in erster Linie, zusätzliche Kopien der

Nachrichteninhalte zu vermeiden, da sich diese desto stärker leistungshemmend auswirken, je näher die Netzbandbreite an die Speicherbandbreite heranreicht.

6.3.2 Datensammlung

Auch im Zusammenhang mit der Ausnutzung der Speicherhierarchie sollte berücksichtigt werden, daß in einem bestimmten Größenbereich ein Datenpaket der Größe $n \cdot D$ mit geringerer Latenz übertragen werden kann als n Datenpakete der Größe D. Es bietet sich also an, mehrere kleine Nachrichten zu einer einzelnen, größeren Nachricht zusammenzufassen. Diese zusammenzufassenden Nachrichten stammen dabei jeweils von unterschiedlichen Prozessen. Dem Bandbreitengewinn durch die größere Datenmenge in einer Nachricht muß jedoch der zusätzliche Aufwand beim lokalen Zusammenführen der Nachrichten durch Intra-Knoten-Kommunikation entgegengestellt werden. Geschieht dieses Zusammenführen durch normales Verschicken und Empfangen von MPI-Nachrichten, fallen zwei zusätzliche Kopiervorgänge an. Dieses Prinzip wird in *[91]* im Zusammenhang mit der Kopplung von räumlich weit verteilten Rechnersystemen eingehend behandelt. Dort ist der Leistungsunterschied zwischen Intra- und Inter-System-Kommunikation jedoch so groß, so daß sich auch aufwendigere Verfahren lohnen können. Die dort vorgestellten Verfahren lassen sich für die hier betrachtete Plattform daher nicht direkt übertragen.

6.3.3 Interknotenkommunikation

Wenn gleiche Daten an mehrere Prozesse auf einem Knoten geschickt werden müssen (möglich bei *1:N*- Kommunikation), ist es effizienter, diese nur einmal in einen Puffer auf diesem Knoten zu schreiben, von wo aus alle Prozesse die Daten lesen können. Dadurch wird die Belastung des Verbindungsnetzes reduziert und gleichzeitig die Parallelität der Kommunikation erhöht: Während bereits Daten auf den nächsten Knoten geschrieben werden, können die Prozesse auf dem vorherigen Knoten die Daten in die Benutzerpuffer kopieren. Bei dieser Optimierung wird also keine Inter-Prozeß-Kommunikation mehr betrieben; stattdessen wird die nächsthöhere Organisationsebene im Cluster verwendet, um Inter-Knoten-Kommunikation zu betreiben.

Eine Kombination von Datensammlung und Inter-Knoten-Kommunikation kann zur Optimierung bei *N:1*-Kommunikation verwendet werden: Mehrere Prozesse von verschiedenen Knoten schreiben Daten in getrennte Bereiche eines gemeinsamen Puffers auf einem Knoten, die sodann als zusammenhängende Nachricht weitergeschickt werden können.

6.3.4 Kommunikationsparallelität

Eine optimale Implementation einer kollektiven Operation nutzt die verfügbaren Ressourcen maximal aus. D.h. alle potentiell nutzbaren unabhängigen Kommunikationskanäle sind gleichzeitig aktiv, wodurch die maximale Leistung erzielt wird (unter der Voraussetzung, daß keine redundanten Daten versendet werden). Es muß vermieden werden, daß ein Kommunikationskanal bei Nutzung durch zu viele Prozesse über seine Kapazität hinaus beansprucht wird, da dadurch in der Regel die effektive Bandbreite sinkt. Der Zugriff auf derartige Ressourcen muß in solchen Fällen synchronisiert werden, um die maximale Leistung zu erzielen.

6.3.5 Kommunikationslokalität

In einem SCI-Netz sind die Punkt-zu-Punkt-Verbindungen unabhängige Kommunikationskanäle und sollten entsprechend der oben beschriebenen Parallelität genutzt werden. Dabei sollte jedoch berücksichtigt werden, daß keine vollständig verbundene Topologie vorliegt, sondern bei der Kommunikation zwischen mehreren Knotenpaaren potentiell Kommunikationskanäle mehrfach genutzt werden, wodurch die verfügbare Bandbreite pro Knoten reduziert wird.

Ebenso sollte versucht werden, das Caching von aufgebauten Inter-Prozeß-Verbindungen, wie es die in Kapitel 4.4 beschriebene Ressourcenverwaltung betreibt, effizient zu nutzen, indem auch bei der Nutzung dieser Ressourcen eine maximale Wiederverwendung zu erreichen versucht wird.

6.3.6 Gemeinsamer Speicher

Bei lokalem gemeinsamem Speicher und auch bei SCI steht neben explizitem Nachrichtenaustausch auch die direkte Kommunikation über gemeinsamen Speicher zur Verfügung, der von mehreren Prozessen unter Umgehung der Primitive des Nachrichtenaustauschs gleichzeitig gelesen bzw. geschrieben werden kann. Die Verwendbarkeit von gemeinsamem Speicher für kollektive Operationen sollte bei jeder Art der Operation geprüft werden.

6.4 SCI-orientierte Topologie

Um kollektive Operationen über das SCI-Netz mit maximaler Effizienz durchführen zu können, muß das Kommunikationsmuster der beteiligten Prozesse, von denen es jeweils Sender und Empfänger innerhalb von Punkt-zu-Punkt-Transferoperationen gibt, auf die Topologie des SCI-Netzes abgestimmt sein. Dazu werden hier entsprechende Vereinbarungen getroffen, auf die die Algorithmen der kollektiven Operationen aufbauen können.

6.4.1 SCI-Segmentsättigung

Wie in Kapitel 3.3 erläutert, steht auf jedem SCI-Linksegment zwischen zwei Knoten die volle Link-Bandbreite B_{slink} zur Verfügung. Diese reicht je nach E/A-Bandbreite B_{int} der Knoten aus, eine bestimmte Anzahl n_{sat} von Datenströmen ohne Einbußen in der Bandbreite der einzelnen Datenströme gleichzeitig zu übertragen. Für eine größere Zahl von Datenströmen reicht die Bandbreite auf einem oder mehreren *kritischen Segmenten* nicht mehr aus, und die Bandbreite pro Datenstrom sinkt unter die maximale Bandbreite, die ein *einzelner* Datenstrom im System erreichen kann[1]. Um diese Überlegung zu validieren und ein gültiges n_{sat} zu bestimmen, wurde ein entsprechender *Segmentsättigungstest* durchgeführt. Bei diesem Test senden in einem SCI-Ring mit *2K* Knoten eine steigende Anzahl von *1...K* Knotenpaaren Daten von einem Sender zu einem anderen Empfänger. Dabei werden die Paare auf dem Ring so angeordnet, daß ein einzelnes SCI-Segment sämtliche Datenströme übertragen muß und somit zum kritischen Segment wird. Die Darstellung in Abb. 6.3 zeigt den Aufbau in einem Ring mit 8 Knoten mit dem Segment zwischen Knoten 4 und 5 als kritischem Segment. Um tatsächlich die Limitierung durch das Segment und nicht etwa Limitierungen durch die E/A-Bandbreite der Knoten zu beobachten, ist es wichtig, daß jeder Knoten entweder als Sender oder Empfänger von Daten fungiert (oder gar nicht kommuniziert). Ein Knoten, der Daten empfängt, kann nicht gleichzeitig mit der maximalen Rate Daten senden.

Für das P3-System mit 8 Knoten in einem SCI-Ring, auf dem der Test durchgeführt wurde, sind die Werte für B_{slink} und B_{int} in Tabelle 6.1 dargestellt. Die E/A-Bandbreite B_{int} der Kno-

Linkfrequenz	B_{slink}	$B_{int_{PIO}}$	$B_{int_{DMA}}$
166 MHz	664 MB/s	170 MB/s	248 MB/s

Tabelle 6.1: Bandbreitenparameter für Segmentsättigungstest

1. Diese Maximum hängt nicht nur von der Bandbreite des SCI-Linksegments, sondern auch von den E/A-Bussen, deren Einbindung in die Knoten und sonstigen Knoteneigenschaften ab.

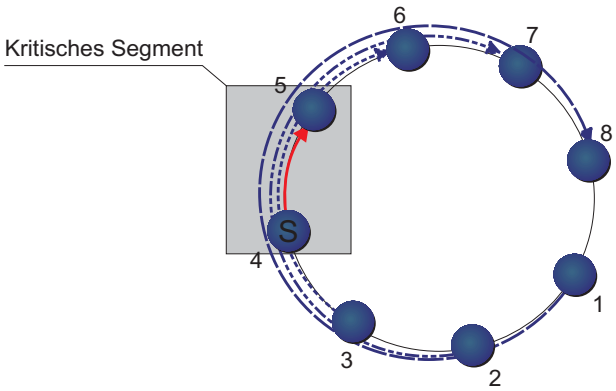


Abbildung 6.3: Datentransfer in einem SCI-Ring über ein kritisches Segment

ten hat verschiedene Werte, je nachdem ob die Daten mittels PIO- oder DMA-Übertragung gesendet werden. Die angegebenen maximalen Werte beziehen sich auf die Übertragung zwischen zwei Knoten auf einem SCI-Link ohne weiteren Datenverkehr und sind für die untersuchten SCI-Linkbandbreiten konstant, da die Limitierung für PIO-Transfers auf dem PCI-Bus auftritt, für den DMA-Transfer im DMA-Baustein auf dem PSA.

Die Ergebnisse des Tests sind in Abb. 6.4 in zwei verschiedenen Ansichten dargestellt. Zunächst ist die Bandbreite aufgetragen, die jeder Knoten im jeweils gewählten Szenario in das Netz einspeisen konnte bzw. aus dem Netz erhalten hat (*Injektionsbandbreite*). Daraus abgeleitet ist die *Segmentbandbreite*, die akkumulierte Bandbreite, die ein Segment im jeweiligen Fall überträgt. Die maximale Injektionsbandbreite beträgt 225 MB/s[1], wenn Daten zwischen zwei Knoten übertragen werden. Kommen weitere Knotenpaare hinzu, bleibt diese Bandbreite beim äußersten Paar weiterhin erhalten. Die inneren Paare erreichen jedoch eine zunehmend geringere Injektionsbandbreite. Dies bedeutet, daß Pakete auf dem SCI-Link Vorrang haben vor Paketen, die aus einem Knoten auf den SCI-Link geleitet werden.

Die akkumulierten Werte der Injektionsbandbreiten pro Segment ergeben die Segmentbandbreiten. Der höchste Wert von 445 MB/s wird für die Kommunikation von zwei Paaren erreicht. Dies entspricht mit hoher Genauigkeit dem theoretischen Maximalwert gemäß Kapitel 3.3.2 (Tabellen Tabelle 3.2 und 3.4), der sich zu 455 MB/s errechnet. Für eine größere Zahl von kommunizierenden Paaren sinkt der Wert auf unter 400 MB/s ab. Dies bedeutet, daß bei Überlastung des Links die effektive Bandbreite absinkt und Überlastungen zur Erreichung der maximalen Leistung daher möglichst zu vermeiden sind. Das Verhalten ist jedoch gutmütig insofern, als daß die effektive Bandbreite stabil auf dem niedrigeren Niveau verbleibt und sublinear mit dem Maß der Überlastung korreliert.

6.4.2 Topologieregeln

Die paarweisen Kommunikationsvorgänge, die in dem oben angeführten Segmentsättigungstest abgelaufen sind und zur Ausbildung von kritischen Segmenten geführt haben, hätten in dem gleichen SCI-Ring auch ohne die beobachteten Sättigungseffekte durchgeführt werden können. Dazu hätten die kommunizierenden Prozesse so auf Knoten des Rings plaziert sein müssen, daß nicht mehr als n_{sat} Datenströme über ein SCI-Linksegment geleitet werden. Bei unabhängiger Punkt-zu-Punkt-Kommunikation sind derartige Optimierungen nicht allgemein möglich, da keines der beteiligten Paare von Kommunikationspartnern von den gleichzeitig stattfindenden Operationen Kenntnis hat[2]. Zudem wäre auch keine Abhilfe möglich, da es keine Alternative

1. Die Messungen wurden via MPI durchgeführt, so daß die absoluten Werte etwas niedriger liegen als bei unmittelbarer SISCI-Nutzung.

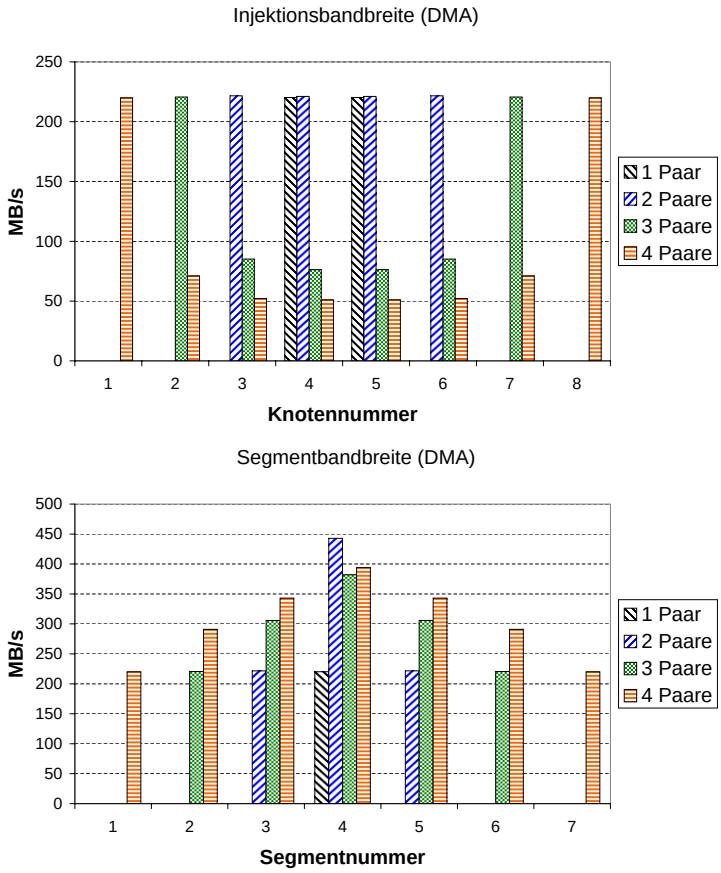


Abbildung 6.4: Maximale Injektions- und Segmentbandbreite für Plattform P3 (8-Knoten, Ringtopologie)

zu dem durch die Sende- und Empfangsaufforderungen eingeprägten Kommunikationsmuster gibt. Durch Sequentialisierung können zwar Sättigungseffekte und ein damit verbundener Leistungsverlust vermieden werden. Diese Maßnahme nimmt jedoch keine Rücksicht auf Datenströme, die gleichzeitig aufgrund anderer Kommunikationsvorgänge über die betroffenen SCI-Linksegmente laufen. Dazu wäre eine separate globale Koordinierung erforderlich.

Bei kollektiven Operationen hingegen sind alle Kommunikationspartner der Operation in Kenntnis über die erforderlichen Datentransfers. Die globale Koordinierung ist durch die Semantik der kollektiven Operation implizit, so daß das genutzte Kommunikationsmuster hinsichtlich der Segmentsättigung optimiert werden kann. Dazu ist als Vorraussetzung jedoch erforderlich, daß die Prozesse die Lage aller anderen Prozesse innerhalb der Topologie des SCI-Netzes kennen. Die Knoten, auf denen die Prozesse laufen, lassen sich im SCI-Netz nur durch ihre SCI-Knoten-ID differenzieren. Daher wurde als erste Regel definiert:

> Die MPI-Ränge der Prozesse in der Applikation werden in aufsteigender Reihenfolge mit den aufsteigenden SCI-Knoten-IDs vergeben, auf denen die Prozesse ausgeführt werden.

Damit die somit festgelegte Zuweisung der MPI-Ränge an die Prozesse diesen auch Rückschlüsse erlaubt auf die Anordnung der SCI-Segmente, mit denen sie untereinander verbunden sind, müssen auch die SCI-Knoten-IDs nach einer festen Regel vergeben werden:

> Innerhalb eines SCI-Ringes werden den Knoten die SCI-IDs aufsteigend in Richtung des SCI-Datenverkehrs zugewiesen.

2. Zumindest läßt sich diese Information nicht ohne unerheblichen Aufwand, insbesondere bei steigender Prozeßzahl, kommunizieren.

Für eindimensionale SCI-Torustopologien ist mit diesen Regeln eine vollständige Beschreibung der SCI-Kommunikationstopologie der Prozesse untereinander gegeben. Für mehrdimensionale Torustopologien mit d Dimensionen $\{d_0,\ldots,d_{d-1}\}$ ist dies ebenfalls möglich. Ausgehend von einer Betrachtung des Systems als eine Zahl $R_0 = \sum_{i=1}^{d-1} d_i$ von SCI-Ringen (Ordnungszahl $r \in [0\ldots R_o]$) in Dimension 0 mit d_0 Knoten (Ordnungszahl $k \in [0\ldots d_0]$), die untereinander mit entsprechend $d_0 \cdot (d-1)$ weiteren SCI-Ringen in die weiteren $d-1$ Dimensionen verknüpft sind, gilt als Präzisierung der genannten Regeln gemäß *[31]*:

(6.4) Pro SCI-Ring dürfen maximal $K_{max} = 15$ Knoten verwendet werden, und innerhalb des SCI-Ringes r erhalten die Knoten die Knoten-ID
$ID_K = r \cdot (K_{max} + 1) + 4 \cdot (k + 1)$

Damit ist ein fester Zusammenhang zwischen SCI-Topologie und MPI-Prozeßrang erreicht, wie es in Abb. 6.5 beispielhaft für ein System aus 12 Knoten mit zweidimensionaler SCI-Netztopologie, auf dem eine MPI-Applikation mit 6 Prozessen abläuft, dargestellt ist.

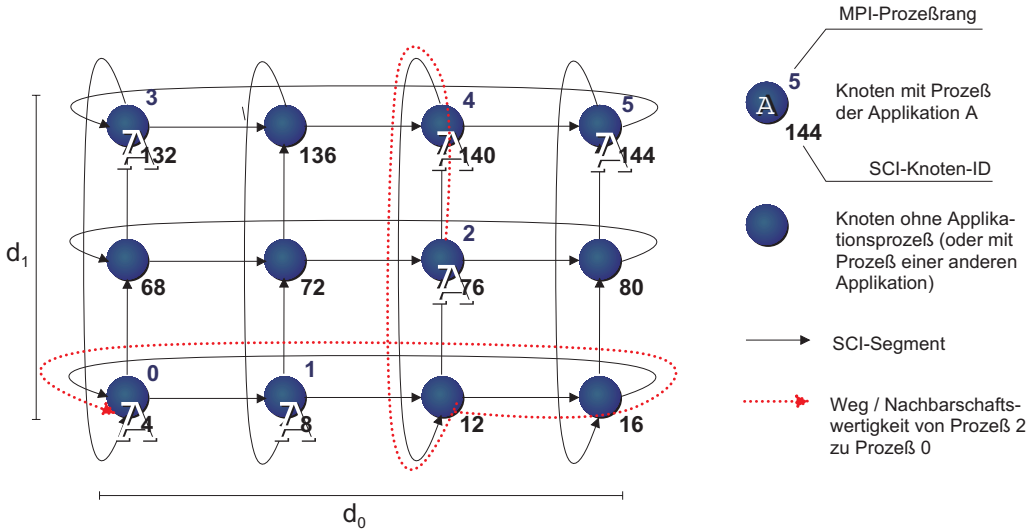


Abbildung 6.5: Zweidimensionales SCI-Netz (D = 2 mit $\{d_0, d_1\}$):
Verhältnis der SCI-Knoten-IDs zur SCI-Netztopologie einerseits und der MPI-Prozeßränge zu den SCI-Knoten-IDs andererseits.

Basierend auf dieser Topologievorschrift bestimmt SCI-MPICH dynamisch die physikalische Lage des Knotens, auf dem ein Prozeß abläuft, relativ zu den anderen Prozessen im Kommunikator. Somit können die im folgenden vorgestellten Verfahren die notwendige Punkt-zu-Punkt-Kommunikation optimieren, indem die Kommunikation über den Zeitraum der kollektiven Operation möglichst konfliktfrei über die SCI-Linksegmente verteilt wird.

6.5 Barrierensynchronisation

Bei der Barrieresynchronisation (kurz: *Barriere*), die durch die MPI-Funktion `MPI_Barrier` ausgeführt wird, handelt es sich um eine kollektive Operation, bei der keine Daten transportiert werden. Stattdessen dient sie dazu, sicherzustellen, daß alle beteiligten Prozesse sich zu einem Zeitpunkt im gleichen Zustand befinden (nämlich innerhalb der Barriere). Somit wird eine globale Synchronisation aller Prozesse im Kommunikator erreicht. Obgleich durch den Nachrichtenaustausch eine implizite Synchronisation stattfindet, die eine derartige explizite Synchronisation oftmals entbehrlich machen könnte, enthalten Applikationen oftmals eine große Zahl von (häufig jedoch unnötigen) Barrieren *[117]*.

Die globale Synchronisation impliziert jedoch nicht, daß die Prozesse zum gleichen Zeitpunkt die Barriere betreten bzw. verlassen, wie aus Abb. 6.6 deutlich wird: Durch die Einnahme des gleichen Zustands zum Zeitpunkt $T_{Barrier}$ ist sichergestellt, daß kein Prozeß eine Operation *hinter* der Barriere ausführen kann, bevor nicht alle anderen Prozesse alle Operationen *vor* der Barriere abgeschlossen haben. Die Zeiten, zu denen ein Prozeß lokal in die Barriere eintritt, sind nicht miteinander verknüpft. In Abb. 6.6 sind die Zeitpunkte für Prozeß P_p als T_{Pp} gekennzeichnet. Die Differenz zwischen dem Zeitpunkt des eigenen Eintritts T_P in die Barriere und dem frühesten Eintrittszeitpunkt eines der beteiligten Prozesse T_{P_min} ist die *Eintrittsverzögerung*,

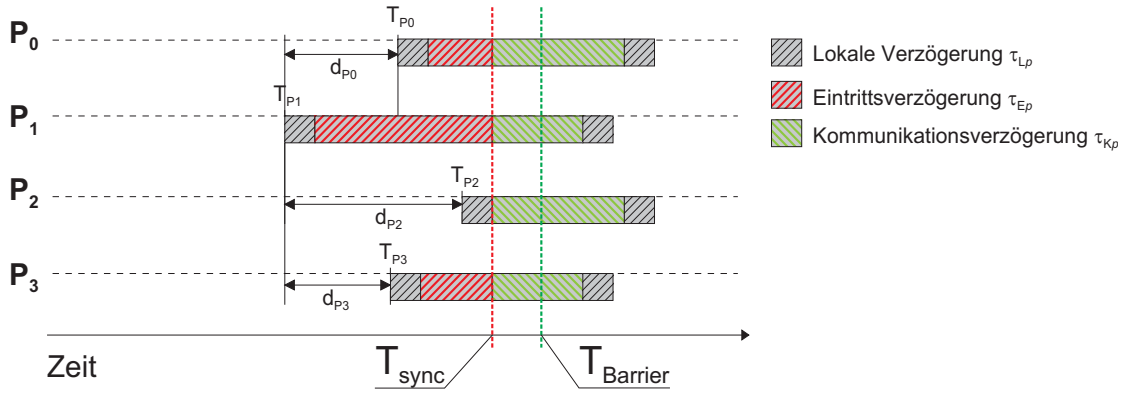


Abbildung 6.6: Möglicher zeitlicher Ablauf einer Barrierensynchronisation und Zeitanteile der einzelnen Phasen durch verschiedene Verzögerungen

Die Gesamtzeit, die ein Prozeß in einer Barriere verbringt, setzt sich aus verschiedenen Anteilen zusammen, wie in Abb. 6.6 dargestellt.:

- *Lokale Verzögerung* τ_L*:* Dies ist die Zeit, die ein Prozeß vom Eintritt in die Funktion `MPI_Barrier` bis zum Aufruf der ersten Kommunikationsfunktion, bzw. vom Abschluß der letzten Kommunikationsfunktion bis zum Verlassen von `MPI_Barrier`, verbringt. Diese Zeit ist abhängig von der Komplexität der zu durchlaufenden Funktionen in der MPI-Bibliothek und entsprechend von der Leistung der lokalen Knotenkomponenten CPU und Speicher.
- *Eintrittsverzögerung* τ_E*:* Nach der lokalen Verzögerung eines Prozesses kann es *minimal* bis zum Aufruf der ersten Kommunikationsfunktion (Ende der lokalen Verzögerung) des Prozesses mit dem spätesten Eintritt in die Barriere (der Prozeß mit dem maximalen Wert von T_{Pp}) dauern, bis die *abschließende* Kommunikation zur Barrierensynchronisation beginnen kann (Zeitpunkt T_{sync}). Diese Verzögerung ist in erster Linie von der zeitlichen Abfolge des Barriereneintritts der beteiligten Prozesse, sowie von dem Kommunikationsmuster (Synchronisationsalgorithmus) und der entsprechenden Kommunikationslatenz abhängig. Diese Zeitspanne entspricht der bereits oben definierten Eintrittsverzögerung.
- *Kommunikationsverzögerung* τ_K*:* Die Abwicklung der Kommunikation, die zur Sicherstellung der Barrierensynchronisation aller Prozesse zum Zeitpunkt $T_{Barrier}$ führt, stellt die Kommunikationsverzögerung dar. Deren Dauer kann, je nach verwendetem Synchronisationsalgorithmus, für die einzelnen Prozesse unterschiedlich sein. Sie hängt natürlich von der Kommunikationslatenz sowie auch von dem Anteil der Kommunikation ab, der bereits während der Eintrittsverzögerung abgewickelt werden konnte.

6.5.1 Bestehende Algorithmen zur Barrierensynchronisation

Da zur Barrierensynchronisation keine Nutzdaten transportiert werden müssen, ist einzig die Latenz der dazu durchgeführten Kommunikation entscheidend. Die Barriere beinhaltet, daß alle

Prozesse ihren Eintritt in die Barriere allen anderen Prozessen mitteilen. Wenn dies durch direkten Austausch von Nachrichten erfolgen würde, müßten dazu insgesamt $(N-1)^2$ Nachrichten verschickt werden und eine Laufzeit $O(2N)$ in Kauf genommen werden. Stattdessen werden üblicherweise die Nachrichten über den Eintritt in die Barriere über eine baumartige Kommunikationsstruktur zu einem Prozeß konzentriert (*check-in)*, der anschließend allen Prozessen das Signal zum Verlassen der Barriere über entgegensetzt gerichtete Nachrichten gibt (*check-out*). Bei Prozeßzahlen, die nicht ein ganzzahliges Vielfaches des *fan-out* bzw. *fan-in*, also der Zahl der Söhne jedes Vaterknotens im Baum, sind, ist eine zusätzliche Stufe im Baum erforderlich. Bei Nutzung einer binären Baumstruktur erfordert dies $2N$ Nachrichten und eine Laufzeit $O(2 \cdot \lceil ld(N) \rceil)$. Die Baumtiefe, und damit die Laufzeit, kann durch Wahl eines anderen *fan-in* variiert werden. Dadurch kann der *check-in* schneller erfolgen, der *check-out* erfordert dann jedoch von jedem Vaterknoten entsprechend mehr nacheinander versandte Nachrichten. Bei einer gegenüber der Verzögerung zum Erfassen neuer Nachrichten und dem Versand *einer* Nachricht geringen minimalen Verzögerungszeit zwischen dem Verschicken von *zwei* Nachrichten kann dies insgesamt effizienter sein.

Eine weitere Variante der Barrierensynchronisation wird über *binären Austausch* (BAB) realisiert. Dieses Verfahren kann optimal für 2-exponentielle Prozeßzahlen implementiert werden: In n Schritten mit $i = 1 \ldots 2^{n-1}$ tauscht jeweils Prozeß p mit Prozeß p' eine Synchronisationsnachricht aus, wobei sich der Rang p' als binäre *exklusiv-oder*-Verknüpfung von p und i ergibt. Mit diesem Verfahren ist nach $O(ld(P))$ jeder Prozeß mit jedem anderen synchronisiert. Für alle Fälle mit $Q \neq 0$ fallen konstant zwei weitere Schritte an, in denen sich die Q Prozesse mit jeweils einem der P Prozesse synchronisieren. Damit ergibt sich für diese Fälle eine nötige Zahl von Kommunikationsschritten $\lfloor ld(P) \rfloor + 2$. Das Kommunikationsmuster der Barrierensynchronisationen über binären Austausch ist in Abb. 6.7 für die Fälle von 6 und 8 Prozessen dargestellt. Es wird deutlich, daß die Synchronisation für 6 Prozesse durch die notwendige prä- und post-Synchronisation mehr Stufen erfordert und somit länger dauert als bei 8 Prozessen, da keine durchgängige Paarbildung (über alle Stufen) möglich ist.

Ein anderes Verfahren, in dem ein Prozeß p in jedem Kommunikationsschritt n ($n \geq 0$) eine Nachricht an Prozeß $(p + n + 1) \lozenge N$ schickt und eine entsprechende Nachricht empfängt, kommt auch bei nicht-2-exponentiellem N mit $\lfloor ld(N) \rfloor + 1$ Schritten aus.

6.5.2 Hierarchische Barriere auf gemeinsamem Speicher

Trotz der geringen Latenz der Nachrichtenübertragung mit SCI-MPICH ist der direkte Zugriff auf lokalen oder entfernten Speicher durch den Wegfall sämtlichen softwaremäßigen Protokollüberhangs schneller. Da zur Synchronisation nur die Übermittlung einer minimalen Information erforderlich ist, und somit der relative Zeitgewinn durch Wegfall des Protokollüberhangs maximal ist, ist der direkte Speicherzugriff für SVS die optimale Methode zur Barrierensynchro-

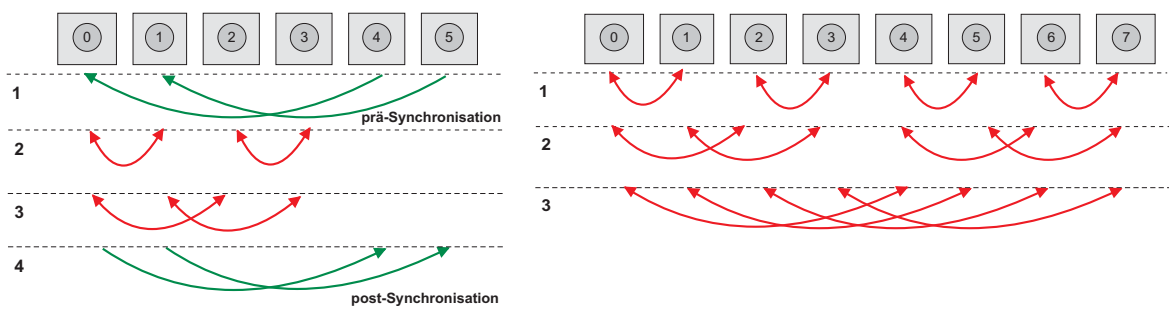


Abbildung 6.7: Kommunikationsmuster (Nachrichtenversand) für die Barrierensynchronisation über binären Austausch für 6 *(links)* und 8 Prozesse *(rechts)*

nisation. Daher wurde eine hierarchische Barriere auf gemeinsamem Speicher (*hierarchische shmem-Barriere*, HSB) entwickelt, bei der sich sowohl die Prozesse innerhalb eines Knotens (*Intra-Knoten-Synchronisation*) als auch die beteiligten Knoten, vertreten durch jeweils einen Prozeß pro Knoten (*Inter-Knoten-Synchronisation*), über gemeinsame Speicherstellen (*Flaggen*) synchronisieren.

Die Intra-Knoten-Synchronisation ist als sequentieller Baum organisiert, während die Inter-Knoten-Synchronisation als *n-ärer* Baum organisiert ist. Daher muß auf jedem beteiligten Knoten über die Prozeßränge der lokalen Prozesse ein *Barrierenprozeß* bestimmt werden, der die Kommunikation mit den anderen Knoten abwickelt; unter allen Knoten wiederum wird ein *Barrierenknoten* bestimmt, der die Wurzel des Baumes bildet. Jeder Barrierenprozeß allokiert auf dem lokalen Knoten SCI-Speicher in einem durch die SMI-Bibiothek global bekannten Speicherbereich, um die in Abb. 6.8 dargestellte Datenstruktur anzulegen. Diese enthält jeweils einen Bereich für die Intra- und die Inter-Knoten-Synchronisation, jeweils wiederum unterteilt in Bereiche für *check-in* und *check-out*. Die Barrierenprozesse organisieren untereinander den Aufbau des Knotenbaums.

Beim *check-in* inkrementiert jede Instanz (Prozeß oder Knoten, vertreten durch den Barrierenprozeß) den Wert der Prozeß-*check-in*-Flagge, während der Barrierenprozeß fortlaufend überprüft, ob alle lokalen Prozesse und untergeordneten Knoten diese Inkrementierung vorgenommen haben. Sobald dies der Fall ist, führt der Prozeß beim übergeordneten Knoten über die Knoten-check-in-Flagge auf dieselbe Weise den *check-in* durch. Sobald der Barrierenprozeß auf dem Wurzelknoten den Abschluß des *check-in* auf seinem Knoten festgestellt hat, startet er den *check-out*, der sich über das Inkrementieren der Knoten-*check-out*-Flagge auf allen Knoten und dort über die auf die Inkrementierung dieser Flagge wartenden Barriereprozesse auf die übrigen Prozesse fortsetzt, indem die Prozeß-*check-out*-Flagge inkrementiert wird. Alle lokalen Prozesse warten auf die Inkrementierung dieser Flagge, die damit eine Barrierensynchronisation abschließt.

Die Anordnung der Prozeß-*check-in*- und -*check-out*-Flaggen in der Datenstruktur wurde optimiert auf eine Minimierung der anfallenden Datentransfers, die durch das knoten-interne Cache-Kohärenz-Protokoll erzeugt werden: Die Flagge, deren Inkrementierung durch den Barrierenprozeß den lokalen Prozessen den erfolgten *check-out* signalisiert, liegt in einer eigenen Cache-Zeile getrennt von den Flaggen, auf deren Inkrementierung der Barrierenprozeß beim *check-in* wartet. Dadurch werden bei P_l lokalen Prozessen $P_l - 1$ Schreibzugriffe sowie maximal $P_l - 1$ Cache-Zeilen-Transfers beim *check-in* und 1 Schreibzugriff mit wiederum $P_l - 1$ Cache-Zeilen-Transfers beim *check-out* erforderlich.

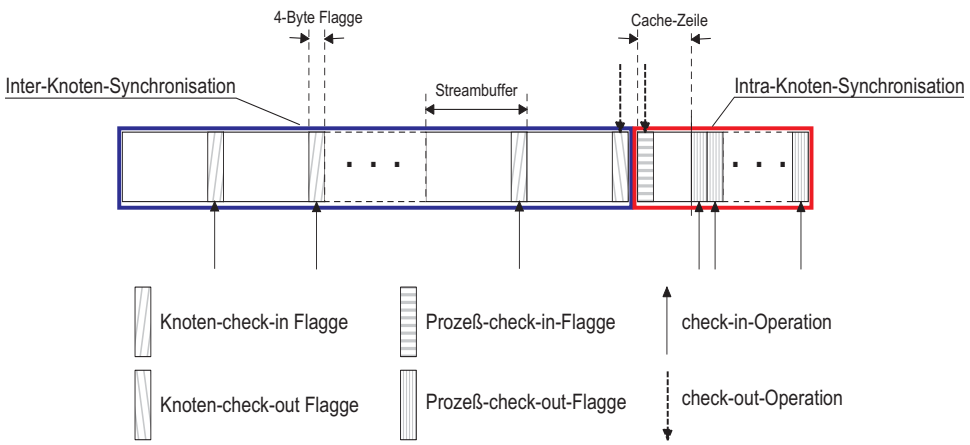


Abbildung 6.8: Datenstruktur der *hierarchischen shmem-Barriere* zur Barrierensynchronisation über direkten Zugriff auf gemeinsamen Speicher.

Der Knotenbaum wurde allgemein als *n-ärer* Baum konzipiert, da der *fan-in* bzw. *fan-out* des Baums größer als 2 gewählt werden sollte. Diese Maßnahme ist sowohl bei der Betrachtung des *check-in* als auch des *check-out* zur Reduzierung der Tiefe des Baums sinnvoll:

- Beim *check-in* ist ein Konflikt der einzelnen SCI-Pakete, mit denen die Knoten-check-in-Flaggen inkrementiert werden, bis zu einer zu bestimmenden Obergrenze des *fan-in* unkritisch. Jedes eingehende Paket ist 16 Symbole lang und belegt somit 8 Takte auf dem SCI-Link, was bei 166MHz Linkfrequenz einer Zeit von $T_{SCI_{write}} = 48ns$ entspricht (dies gilt analog für die Echo- und Response-Pakete). Hinzu kommt beim Schreibvorgang die Transaktion auf dem PCI-Bus.
  Der zeitliche Versatz des Eintritts der Prozesse in die Barriere wird in der Regel deutlich größer sein als die Kommunikationszeit $T_{SCI_{write}}$ auf dem SCI-Link: bei einer typischen Zykluszeit der CPU von etwa 1*ns* und einer Verzögerung von mehreren Hundert *ns* durch einen notwendigen Zugriff auf den Hauptspeicher ist es sehr unwahrscheinlich, daß zwei Prozesse innerhalb von $T_{SCI_{write}}$ einen Schreibzugriff auf den selben Knoten durchführen. Der nicht-deterministische Charakter des zeitlichen Ablaufs des *check-in* und die geringe Zahl von SCI-Paketen, die bei einem *check-in* anfallen erlaubt keine aussagefähige Modellierung dieses Vorgangs (aus der Sicht des Ziels der Schreiboperationen) bezüglich möglicher Effekte gleichzeitig eintreffender Pakete. Stattdessen kann die erwähnte Obergrenze des *fan-in* experimentell bestimmt werden.
- Beim *check-out* erfolgt durch die Vergrößerung des *fan-out* zwar eine Sequentialisierung der *check-out*-Operationen auf den untergeordneten Knoten. Da die Anordnung der Knoten-check-out-Flaggen (wie auch der Knoten-*check-in*-Flaggen) in der Datenstruktur bezüglich der *streambuffer* so ist, daß ein Schreibzugriff auf eine der Flaggen auf einem entfernten Knoten unmittelbar zur Generierung eines SCI-Pakets zur Ausführung der Transaktion führt, ist auch für *n* aufeinanderfolgende derartige Schreibzugriffe nur eine einzige *store-barrier* (siehe Kapitel 3.3.3) erforderlich, um den Abschluß der Transaktion festzustellen.

Das Kommunikationsmuster für die HSB für 9 Prozesse auf 6 Knoten mit einem *fan-in* von 3 ist in Abb. 6.9 (*links)* dargestellt. Dabei ist zu beachten, daß die Kommunikation nicht auf dem Versand von Nachrichten beruht, sondern aus *store*-Operationen eines Wortes in entfernten Speicher.

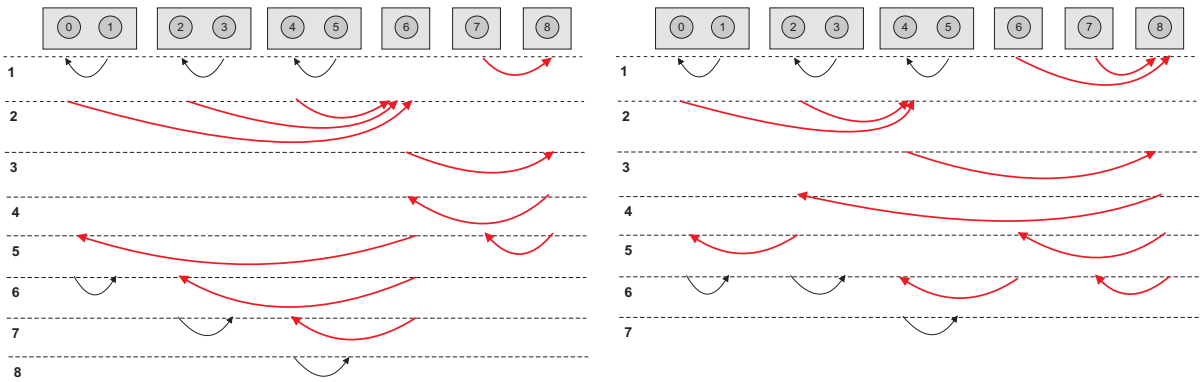


Abbildung 6.9: Kommunikationsmuster (store-Operationen) für die hierarchische shmem-Barrier mit 9 Prozessen auf 6 Knoten (dünne Pfeile: intra-Knoten, dicke Pfeile: inter-Knoten)
links: fester *fan-in* von 4 und identischer *fan-out*
rechts: adaptiver *fan-in* (hier: 3) und reduzierter *fan-out* (hier: 2)

Die Parallelität der Kommunikation beim *check-out* ließe sich erhöhen, wenn der verwendete *fan-out* kleiner gewählt wird als der *fan-in*. Die Modellierung und Evaluierung im weiteren wird klären, wie stark sich die geringere Parallelität beim check-out auswirkt.

Ein weiterer Ansatz zur Optimierung ist die Anpassung des vorgegebenen *fan-in* an die tatsächliche Anzahl von Prozessen. Wie aus Abb. 6.9 ersichtlich ist, synchronisieren sich bei der ersten Intra-Knoten-Synchronisation bei einem *fan-in* von 4 in der linken Gruppe entsprechend 4 Prozesse, in der rechten Gruppe hingegen befinden sich nur noch 2 Prozesse. Dieses Ungleichgewicht, das sich vor allem beim check-out bemerkbar macht, kann durch einen *adaptiven fan-in* vermieden werden, bei dem der vorgegebene *fan-in* auf eine optimale Gleichverteilung der Gruppengrößen hin optimiert (reduziert) wird. Für den gegebenen Fall beträgt dieser optimierte *fan-in* 3, wie in Abb. 6.9 *rechts* dargestellt. Für die Barrierensynchronisation von N Prozessen und einen vorgegeben *fan-in* f_{in} berechnet sich der derartig optimierte *fan-in* f_{opt} gemäß

$$(6.5)\quad f_{opt} = f_{in} - \left\lfloor \frac{\left\lceil \frac{N}{f_{in}} \right\rceil \cdot f_{in} - N}{\left\lceil \frac{N}{f_{in}} \right\rceil} \right\rfloor$$

6.5.3 Modellierung und Evaluierung der Barrierensynchronisation

Zur Beurteilung der Wirksamkeit und Effizienz der vorgestellten Lösung werden die Modelle der Verfahren und die daraus ermittelte Leistung den Experimenten gegenübergestellt.

6.5.3.1 Modellierung der Nachrichtenbarriere mit binärem Austausch

Die Modellierung der herkömmlichen Barrierensynchronisation durch binären Austausch von Nachrichten wurde bereits in Kapitel 6.5.1 durchgeführt und muß daher nur noch mit der gemäß (4.26) bestimmten Latenz für Kontrollnachrichten skaliert werden, um die Barrierenverzögerung $\tau_{B_{bin}}$ für N Prozesse zu ermitteln:

$$(6.6)\quad \tau_{B_{bin}}(N) = \tau_l + ld(N) \cdot L_{ctrl-pp} \text{ für } N = 2^n$$

$$\tau_{B_{bin}}(N) = \tau_l + (\lfloor ld(N) \rfloor + 2) \cdot L_{ctrl-pp} \text{ für } N \neq 2^n$$

6.5.3.2 Modellierung der hierarchischen shmem-Barriere

Die Modellierung der Abarbeitungszeit der hierarchischen shmem-Barriere erfolgt für eine konstante Prozeßdichte k. Die mathematische Modellierung beliebiger asymmetrischer Verteilungen würde eine sehr komplexe Beschreibung der konkreten Verteilung erfordern, brächte jedoch keine weiteren Erkenntnisse: Wie gezeigt wird, wird die Kommunikationsverzögerung von der Inter-Knoten-Kommunikation dominiert; bei der Intra-Knoten-Kommunikation dominiert die Eintrittsverzögerung gegenüber der Kommunikationsverzögerung. Weiterhin werden zur Modellierung benötigt:

- Die Wahl des *fan-in* f_{in} und des *fan-out* f_{out}. Für den Fall $f_{in} \neq f_{out}$ soll gelten, daß f_{out} echter Teiler von f_{in} ist, was durch das *fan-Verhältnis*

  $$(6.7)\quad r_{fan} = \frac{f_{in}}{f_{out}}$$

  beschrieben wird. Die Implementation der shmem-Barriere in SCI-MPICH arbeitet mit $r_{fan} = 1$.
- Die Latenzen für die Schreib- und Lesevorgänge in lokalem und entferntem Speicher sowie einer *store-barrier* gemäß Tabelle 4.1 auf Seite 73.
- Die *lokale Verzögerung* vom Eintritt in `MPI_Barrier` bis zum ersten Schreibvorgang sowie dem Verlassen aller Funktionen zurück zur Aufrufebene oberhalb von `MPI_Barrier`.

Die lokale Verzögerung ist eine konstante, systemabhängige Zeit τ_l, die experimentell durch Aufruf von `MPI_Barrier` ohne Durchführung von Kommunikation bestimmt werden muß.

Der Intra-Knoten *check-in* besteht aus $s_{in} = k - 1$ Schreibvorgängen in eine Cache-Zeile und mindestens einem Lesevorgang zur Überprüfung des *check-in*-Zählers durch den Barrierenprozeß. Die genaue Zahl der notwendigen Lesevorgänge hängt von den Eintrittsverzögerungen der beteiligten Prozesse ab.

Bei K Knoten und einem *fan-in* f_{in} bzw. *fan-out* f_{out} sind S_{in} bzw. S_{out} Stufen der Intra-Knoten-Kommunikation zum *check-in* bzw. *check-out* erforderlich (jeweils f_x Knoten sind einem weiteren Knoten zugeordnet). Für $f_{in} = f_{out} = 2$ kann direkt gerechnet werden:

$$(6.8)\quad S_{in_2} = \lfloor ld(K) \rfloor \text{ bzw. } S_{out_2} = \lfloor ld(K) \rfloor + 1$$

Für höhere *fan-in*-Werte muß die Anzahl der Knoten K_S, die über S Intra-Knoten-Kommunikationsschritte synchronisiert werden können, über eine Reihenentwicklung bestimmt werden:

$$(6.9)\quad S(K_S) = j, \text{ für das gilt } K_S \le \sum_{i=0}^{i \le j} f_{in}^{\,i} \quad , \quad K_S > \sum_{i=0}^{i \le j-1} f_{in}^{\,i}$$

Damit berechnet sich die Zahl der Inter-Knoten-Kommunikationsschritte für den *check-in* und *check-out* für $f_{in} = f_{out} = f$ für K_S Knoten:

$$(6.10)\quad S_{in_f} = S(K_s) \text{ bzw. } S_{out_f} = S(K_S) + f - 1$$

Der anschließende Intra-Knoten-*check-out* besteht für $k > 1$ aus einem Schreibvorgang des Barrierenprozesses und einem anschließenden (parallelen) Lesevorgang aller $k - 1$ weiteren Prozesse.

Insgesamt ergibt sich als Barrierenverzögerung τ_B in Abhängigkeit von N, k, f_{in} und f_{out}:

$$(6.11)\quad \tau_{B_{shmem}} = \tau_l + 2 \cdot (l_{lw} + l_{lr}) + S_{in}(K, f_{in}) \cdot (l_{rw} + l_{lr}) + S_{out}(K, f_{out}) \cdot l_{rs} + l_{sb}$$

In (6.11) beschreibt

- der erste Summand die lokale Verzögerung;
- der zweite Summand die Intra-Knoten-*check-in*- und -*check-out*-Vorgänge. Dabei wird davon ausgegangen, daß es durch die unterschiedlichen Ankunftszeiten der Prozesse an der Barriere nicht zu weiteren Verzögerungen im Bereich des Systembus kommt;
- der dritte Summand den Inter-Knoten-*check-in*-Vorgang. Obwohl hierbei Schreibvorgänge auf entfernten Speicher durchgeführt werden, braucht die zugehörige *store-barrier* nicht berücksichtigt zu werden, da durch die entsprechende Speicherausrichtung das Datum direkt nach dem lokalen Schreibvorgang zum Empfänger übertragen wird. Die durchgeführte *store-barrier* dient der Sicherung der Integrität der Übertragung;
- der vierte und fünfte Summand schließlich den Inter-Knoten-*check-out*-Vorgang. Hier müssen die lokalen Lesevorgänge nicht berücksichtigt werden, da diese sich mit dem nächsten entfernten Schreibvorgang (bzw. der abschließenden *store-barrier*) überlappen. Wiederum wird hier eine abschließende *store-barrier* zur Sicherung der Integrität *aller* vorhergehenden Übertragungen durchgeführt.

6.5.3.3 Leistungsbewertung und Vergleich von Modell und Implementation

Die gemäß dem vorgestellten Modell sowie experimentell ermittelten Werte für die Barrierensynchronisation sind in Abb. 6.10 dargestellt; es wurden die möglichen *fan-ins* für Konfigura-

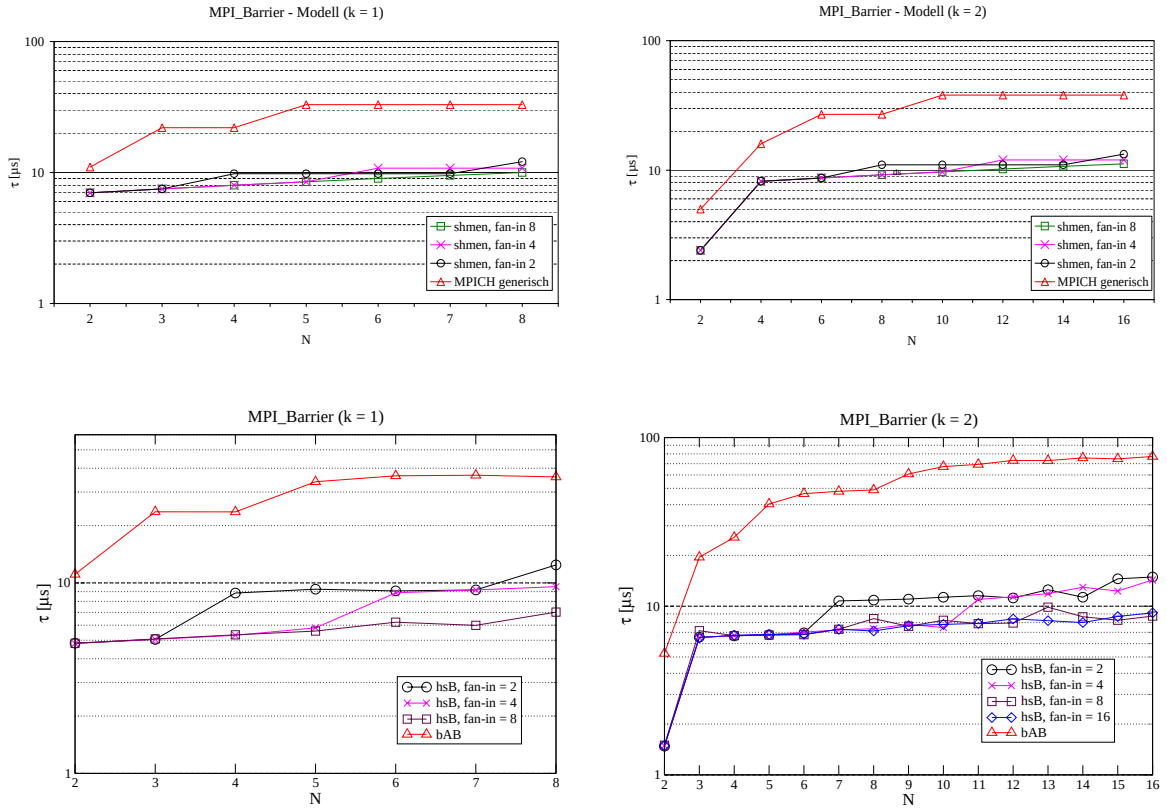


Abbildung 6.10: Simulativ (*oben*) und experimentell (*unten*) ermittelte Latenz für Barrierensynchronisation mittels Nachrichtenaustausch oder mittels hierarchischer Speicherbarriere (bei unterschiedlichem *fan-in)*
Links: Prozeßdichte *k = 1 Rechts:* Prozeßdichte *k= 2*

tionen mit Prozeßdichten von $k = 1$ und $k = 2$ auf bis zu 8 Knoten verwendet.

Die Vorteile der hierarchischen shmem-Barriere werden aus den Diagrammen deutlich.

Latenz. Die Leistung der Barrierensynchronisation für eine feste Zahl von Prozessen ist direkt proportional zur Latenz. Die absolute Latenz zur Synchronisation liegt bei der HSB in den untersuchten Fällen bei maximaler Zahl von Prozessen um fast eine Größenordnung niedriger als bei der BAB (8,7μs gegenüber 70,1μs bei 16 Prozesse auf 8 Knoten). Dies bestätigt die initiale Motivation der Verbesserung der Leistung durch Nutzung der direkten Speicherzugriffe.

Intra-Knoten-Skalierbarkeit. Die relative Zunahme der Latenz für ein wachsendes *k* ist das Maß für die *Intra-Knoten-Skalierbarkeit*, die um so besser ist, je geringer diese relative Zunahme ausfällt. Hier hat die HSB ebenfalls Vorteile gegenüber der BAB, wie Tabelle 6.2 zeigt. Gegenüber dem Faktor 1,38 der BAB weist die HSB einen Faktor 1,20 auf. Die explizite Berücksichti-

Barrierentyp	**Latenz τ für $k = 1$**	**Latenz τ für $k = 2$**	$\tau_{k=2} / \tau_{k=1}$
BAB	*34,46 µs*	*47,59 µs*	**1,38**
HSB ($f_{in} = 8$)	*7,03 µs*	*8,44 µs*	**1,20**

Tabelle 6.2: Vergleich der Intra-Knoten-Skalierbarkeit von BAB und HSB für $N = 8$ Prozesse

gung der Hierarchie der Kommunikationsleistung im Verfahren der HSB erweist sich hier als wirksam.

Inter-Knoten-Skalierbarkeit. Die relative Zunahme der Synchronisationslatenz für eine wachsende Zahl K von Knoten mit Prozessdichte k, ist das Maß für die *Inter-Knoten-Skalierbarkeit.* Diese wurde in Tabelle 6.3 für die Knotenzahlen 4 und 8 bestimmt.

Barrierentyp	k	**Latenz τ für $K = 4$**	**Latenz τ für $K = 8$**	$\tau_{K=8} / \tau_{K=4}$
BAB	*1*	*22,25 µs*	*34,46 µs*	**1,55**
	2	*47,59 µs*	*70,97 µs*	**1,49**
HSB ($f_{in} = 8$)	*1*	*5,34 µs*	*7,03 µs*	**1,31**
	2	*8,44 µs*	*8,74 µs*	**1.04**

Tabelle 6.3: Vergleich der Inter-Knoten-Skalierbarkeit von BAB und HSB für $K = 4$ auf $K = 8$ Knoten

Für die Inter-Knoten-Skalierbarkeit, die wegen $K \gg k$ (in typischen Konfigurationen) wichtiger ist als die Intra-Knoten-Skalierbarkeit, wirkt sich die Berücksichtigung der Kommunikationshierarchie ebenfalls günstig aus. Während für $K = 4$ die dadurch notwendige zusätzliche Kommunikationsstufe noch deutlich ins Gewicht fällt (vergleiche Latenzen für $k = 1$ und $k = 2$), fällt dies mit zunehmendem K immer weniger ins Gewicht, erbringt aber günstige absolute Latenzen.

Modell vs. Experiment. Zwischen den Werten, die über das Modell ermittelt wurden, und den experimentell ermittelten Werten zeigt sich sich eine vollständige qualitative Übereinstimmung. Die quantitative Übereinstimmung für die HSB bei Knotenzahlen, die mehr als einen Inter-Knoten-*check-in* erfordern, ist ebenfalls sehr hoch und belegt die Korrektheit des Modells für diese Fälle. Es ist daher zulässig, damit das Skalierungsverhalten der HSB jenseits der experimentell validierten Knotenzahlen zu prognostizieren.

Jedoch sind die tatsächlichen Latenzen für die HSB bei weniger als zwei Inter-Knoten-check-*ins* um maximal $2\mu s$ niedriger als mittels des Modells berechnet. Hier findet offenbar eine stärkere Überlappung der einzelnen Schritte statt als im Modell formuliert. Andererseits sind die berechneten Latenzen für die BAB für $k > 1$ deutlich niedriger als im Experiment bestimmt. Da das Modell das Kommunikationsmuster korrekt beschreibt und seine Werte für $k = 1$ sehr gut mit dem Experiment übereinstimmen, ist dieser Effekt auf Engpässe bei der Intra-Knoten-Kommunikation zurückzuführen, die nicht weiter untersucht werden konnten.

6.6 Rundsenden

Die *1:N*-Kommunikation wird in MPI als *Broadcast* bezeichnet und mittels `MPI_Bcast()` implementiert. Jeder Prozeß erhält vom Wurzelprozeß die gleichen Daten zugestellt. Das Kommunikationsvolumen beträgt $N \cdot D$ Bytes, wobei die D Bytes des Wurzelprozesses identisch zu den N Prozessen geschickt werden.

Viele Arbeiten im Bereich der kollektiven Operationen beschäftigen sich mit derartigen *multicasts*, bei denen eine Nachricht von einem Knoten in einem Netz zu einer Menge von anderen Knoten übertragen wird *[125,123]*, mehrere solcher *multicasts* parallel abgewickelt werden *[126]* oder auch *multi-node multicasts [124]* untersucht werden, wo mehr als ein Knoten eine Nachricht zu einer Menge von anderen Knoten übertragen. Alle diese Untersuchungen gehen wie das

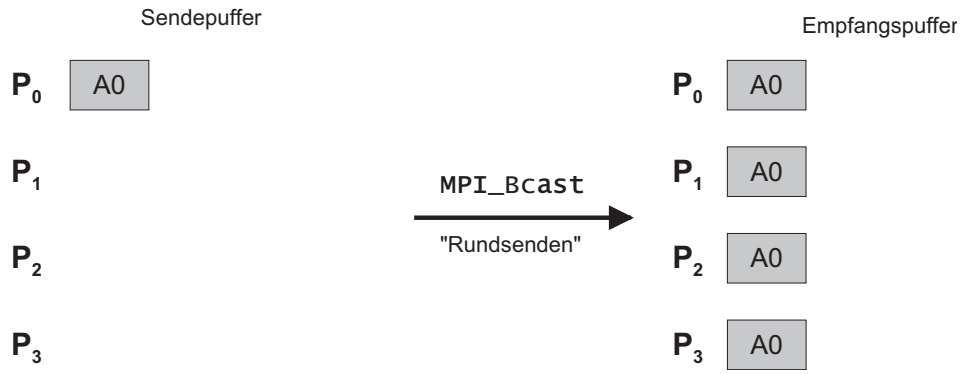


Abbildung 6.11: Datenverteilung bei *Rundsenden*-Kommunikation (`MPI_Bcast`)

Rundsenden davon aus, daß alle Zielknoten von einem Quellknoten die gleichen Daten erhalten. Jedoch machen sie zwei Annahmen, die für die in dieser Arbeit genutzte Kommunikation nicht gemacht werden können:

Die Zeit für lokales Kopieren ist klein oder sogar vernachlässigbar gegenüber der Zeit, die gleiche Datenmenge an einen anderen Prozeß zu senden.

Diese Annahme trifft für die Nutzung von SCI-Kommunikation in einem PC-Cluster nicht zu, wenn die Kopieroperation nicht im Cache abgewickelt wird (siehe Tabelle 4.1 und auch Abb. 7.2).

Der Wurzelprozeß kann mehr als einen weiteren Prozeß mit nahezu gleichbleibender Bandbreite mit Daten versorgen.

Versuche mit entsprechenden mehrdimensionalen Verfahren haben gezeigt, daß die akkumulierte Gesamtbandbreite für derartige Verfahren nicht steigt, da im eindimensionalen Fall durch die Nebenläufigkeit von DMA- und PIO-Transfers bei den empfangenden Prozessen tatsächlich der Wurzelprozeß die geringste Bandbreite aufweist.

6.6.1 Rundsenden über Binomialbaum

Beim Broadcast gilt es, ausgehend vom Wurzelprozeß möglichst schnell die Kommunikationsparallelität zu maximieren. Dies kann durch eine einfache Baumstruktur nicht erreicht werden; der generische MPICH Broadcast-Algorithmus verwendet daher als Kommunikationsmuster einen Binomialbaum der Tiefe $T_{bcast} = \lceil ld(N) \rceil$. Dieser bestimmt den Kommunikationspartner durch folgende Operation auf der Binärdarstellung des eigenen Prozessrangs:

> Der Rang des Empfängerprozesses in der Runde r (beginnend mit $r = 0$ für den ersten Sendeprozeß) ergibt sich durch Invertierung des r-höchstwertigen Null-Bits in der Binärdarstellung des eigenen Prozeßrangs.

Der Ablauf der Kommunikation gemäß dieser Vorschrift für $N = 16$ ist in Abb. 6.12 dargestellt. Prozeß 0 hat die Daten initial (ist somit der Wurzelprozeß) und schickt die Daten in der ersten Stufe zunächst an Prozeß 8. In der nächsten Stufe senden diese beiden Prozesse die Daten an die Prozesse 4 und 12, usw. In der Abbildung sind die Pfeile, die die Sendevorgänge darstellen, jeweils mit der Nummer der Stufe versehen, in der sie ausgeführt werden. Die Tabelle in Abb. 6.12 stellt der Zahl der Prozesse in jeder Stufe, die die Daten verfügbar haben, die Zahl der Sendevorgänge in dieser Stufe gegenüber. Da diese Zahlen gleich sind, wird deutlich, daß dieser Algorithmus für zwei-exponentielle Prozesszahlen eine hohe Kommunikationsparallelität aufweist, die exponentiell mit der jeweiligen Stufe zunimmt. Die SCI-Topologieregeln werden optimal berücksichtigt, da mit zunehmender Kommunikationsparallelität die Distanz zwischen Sender und Empfänger immer weiter zurückgeht und es so zu keinen Kollisionen der Datenströme mit Bildung von kritischen Segmenten kommen kann.

6.6.2 Rundsenden durch Pipelining

Speziell für kleinere Datenmengen ist das Binomial-Baum-Verfahren bereits sehr effizient und kaum weiter zu optimieren. Für das Rundsenden von großen Datenmengen, die mit dem *rendezvous*-Protokoll übertragen werden, kann jedoch durch *Pipelining* die Kommunikationsparallelität maximiert werden, indem maximal *alle* Prozesse gleichzeitig senden und/oder empfangen. Bei den bekannten Verfahren für Rundsenden in MPI *[118,120,121,122,133]* muß ein Prozeß jedoch, bevor er eine Nachricht an einen anderen Prozeß weiterleiten kann, diese vollständig empfangen haben. Dadurch beträgt die Kommunikationsparallelität KP_{bcast} im allgemeinen Fall des vorgestellten Binomial-Baum-Verfahrens für Rundsenden mit N Prozessen

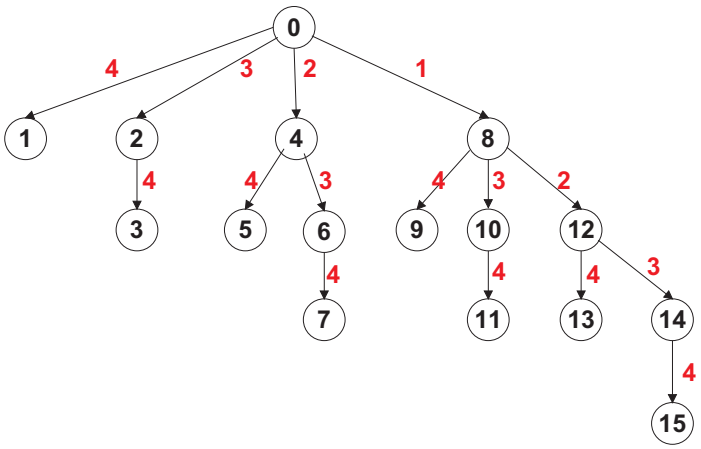
$$(6.12)\quad KP_{bcast_{baum}} = \frac{1}{\lceil ld(N) \rceil} \cdot \sum_{i=0}^{\lceil ld(N) \rceil} 2^i$$

Dies bedeutet für das angegebene Beispiel einen Wert von $\frac{15}{4}$ bei einem theoretischen Maximum von 16, was als Effizienz

$$(6.13)\quad \varepsilon_{KP} = \frac{KP_{bcast}}{KP_{max}}$$

einen Wert von $\varepsilon_{KP_{baum}} = 0,234$ ergibt. Diese Effizienz läßt sich hierbei nur für mehrere unabhängige, hintereinander ausgeführte Rundsendeoperationen steigern, wenn für das Rundsenden ein sequentieller Baum verwendet wird. Für diesen Fall wurde in *[118]* für 96 Knoten eine 70-fache Bandbreite gegenüber einer einmal ausgeführten Rundsendeoperation ermittelt. Dieser Wert ist jedoch von geringer praktischer Bedeutung, da der Fall mehrerer aufeinanderfolgender Rundsendeoperationen mit dem gleichen Wurzelprozeß in der Praxis äußerst selten auftritt. Darüber hinaus hat das erforderliche sequentielle Verfahren für einzelne Rundsendeoperationen eine sehr schlechte Leistungscharakteristik, da es gegenüber den Baum-basierten Verfahren gemäß $O(N)$ anstatt $O(\log(N))$ skaliert.

Wenn hingegen jeder Prozeß die verfügbaren Daten an den nächsten Nachbarn weitersendet, *bevor* die gesamte Nachricht eingetroffen ist, kann die Kommunikationsparallelität nahe an das Maximum herangeführt werden, da mit diesem *Pipelining*-Verfahren für eine ausreichend große Datenmenge alle Prozesse gleichzeitig senden und empfangen. Ein Vergleich des Kommunikationsablaufs der beiden Verfahren ist in Abb. 6.13 für ein Rundsenden mit $N = 8$ Prozessen dargestellt. Das Binomial-Baum-Verfahren benötigt gemäß $ld(N) = 3$ die dreifache Sendezeit einer einzelnen Nachricht. Das Pipeline-Verfahren benötigt hingegen nur die einfache Sendezeit einer Nachricht plus der $N-1$-fachen Verzögerung zum Beginn des



Stufe	Daten verfügbar	Sende -vorgänge
1	1	1
2	2	2
3	4	4
4	8	8

Abbildung 6.12: Ablauf der Kommunikation beim Broadcast für den generischen MPICH-Algorithmus (*links*) und Aufstellung der Kommunikationsparallelität (*rechts*).

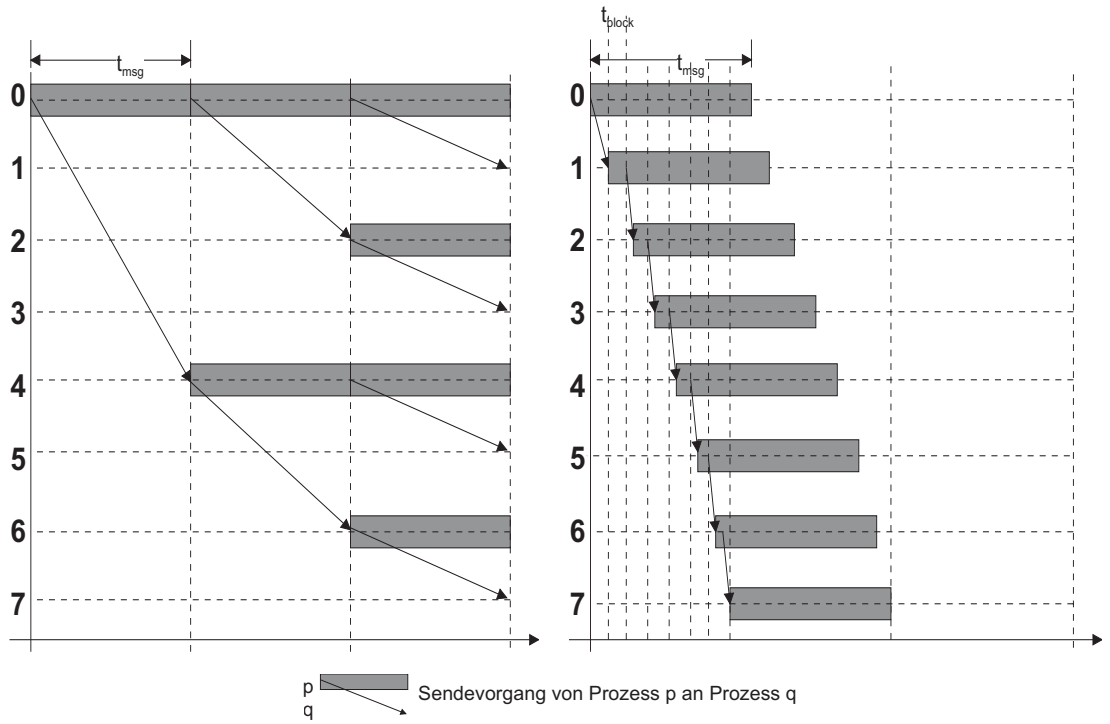


Abbildung 6.13: Zeitbedarf für eine Broadcast-Übertragung für den generischen Binomialbaum-Algorithmus (*links)* und das Pipeline-Verfahren (*rechts).*

Weiterleitens eines Blocks auf jedem Prozeß (Füllzeit der Pipeline).

Aus diesen Darstellungen läßt sich die untere Schranke des Zeitbedarfs t_{bcast} für Rundsenden, gemessen als Zeitdauer vom Start des ersten Sendevorgangs bis zum Abschluß des letzten Empfangsvorgangs, ableiten. Diese basiert auf der Zahl *N* der Prozesse, der Zeitdauer t_{msg} für die Übertragung der gesamten Broadcast-Nachricht sowie der Zeitdauer t_{block} für die Übertragung eines Teils der Nachricht, die beim Pipeline-Verfahren die Blockgröße der Flußkontrolle darstellt. Für den Baum-Algorithmus ist dies

$$(6.14)\ t_{bcast_{baum}} = t_{msg} \cdot \lceil ld(N) \rceil$$

Für das Pipeline-Verfahren ergibt sich hingegen:

$$(6.15)\ t_{bcast_{pipe}} = t_{msg} + (N-1) \cdot t_{block}$$

Für ein hinreichend kleines t_{block}, das durch die Wahl der Blockgröße für die Flußkontrolle bestimmt werden kann, ist der Zeitbedarf für die Broadcast-Übertragung mittels des Pipeline-Verfahrens unabhängig von der Zahl der Prozesse. Der zusätzliche Zeitbedarf für die Durchführung der Flußkontrolle (jeweils eine *store barrier* und ein separater Schreibvorgang für den Schreibzeiger), der pro übertragenem Block anfällt, sowie die sinkende Bandbreite für sehr kleine Blockgrößen lassen die Bestimmung der optimalen Blockgröße zu einem Optimierungsproblem werden, bei dem die optimale Blockgröße nicht die kleinstmögliche sein wird.

6.6.2.1 pcast-Protokoll

Die Implementierung dieses *Pipeline*-Verfahrens ist komplex, da es sich nicht, wie sonst bei der Implementierung von kollektiven Operationen üblich, durch die Verwendung der verfügbaren MPI-Kommunikationsprimitive innerhalb einer höhergelegenen Schicht realisieren läßt. Dieser Ansatz ist hier nicht möglich, weil auf diese Art nur *vollständige* Nachrichten bearbeitet

werden können. Das hieße, daß eine Broadcast-Nachricht in eine entsprechende Zahl von kleineren Nachrichten zerlegt werden müßte. Diese müßten dann einzeln vollständig empfangen werden, bevor sie weitergesendet werden können. Dabei geht viel Leistung durch anfallende Zwischenkopien und den notwendigen Aufwand zur Zuordnung der einzelnen Nachrichten verloren. Entsprechend ist keine Implementation des Pipeline-Verfahrens in der Literatur bekannt, und auch Experimente mit (nicht öffentlich dokumentierten) MPI-Implementationen zeigen stets eine logarithmische Abhängigkeit der Zeitdauer einer Broadcast-Übertragung von der Zahl der beteiligten Prozesse.

Tatsächlich ist es zur effizienten Implementation des Verfahrens erforderlich, die nötigen Kommunikationsoperationen auf möglichst tiefer Ebene abzuwickeln, ohne dabei jedoch den Kontext der MPI-Bibliothek zu verlassen, wie dies bei einem einfachen Hardware-Broadcast der Fall wäre. Für SCI-MPICH wurde daher ein spezielles Rundsende-Protokoll namens *pcast* entwickelt. Als Basis dient das blockierende *rendez-vous*-Protokoll, das durch seine äußerst schlanke Flußkontrolle gut dazu geeignet ist (siehe dazu Kapitel 4.2.4.2). Zunächst besteht hier wie beim blockierenden *rendez-vous*-Protokoll das Problem, daß sich die Sender und Empfänger synchronisieren müssen, um die unterbrechungsfreie Übertragung der Daten im Ringpuffer durchführen zu können. Im Gegensatz zur allgemeinen Punkt-zu-Punkt-Kommunikation bestehen bei einer kollektiven Operation wie der *Broadcast*-Übertragung etwas andere Rahmenbedingungen: Da kollektive Operationen stets blockierend sind, kann zu einem Zeitpunkt nur *eine* kollektive Operation durchgeführt werden. Die MPI-Bibliothek kann so lange in der Operation verweilen, bis diese abgeschlossen ist (was bei einem korrekten MPI-Programm in endlicher Zeit geschieht). Das Problem der Zuordnung mehrerer aktiver Sende- und Empfangsprozesse zueinander entfällt. Somit kann zur Synchronisation ein einzelner globaler Zustand verwendet werden. Sobald ein Prozeß `MPI_Bcast` aufruft, allokiert er einen Eingangspuffer und initialisiert gleichzeitig die Beschreibung des Eingangspuffers des Prozesses, an den er senden muß (das ist der Prozeß mit dem nächsthöheren Rang im Kommunikator[1]), auf einen ungültigen Wert. Anschließend signalisiert er dem nächstniedrigeren Prozeß durch eine `PCAST_READY`-Kontrollnachricht, die die Beschreibung des Eingangspuffers enthält, seine Empfangsbereitschaft. Diese Kontrollnachricht kann im Gegensatz zu der analogen `OK_TO_SEND`-Kontrollnachricht im *rendez-vous*-Protokoll ohne vorhergehende Anforderung durch den Sender verschickt werden, da die Semantik einer kollektiven Operation voraussetzt, daß alle Prozesse bereit sind zum Senden der Nachricht, deren Größe dem Sender und Empfänger im Voraus bekannt ist. Jeder Prozeß prüft nun solange auf eingehende Nachrichten, bis er einerseits eine `PCAST_READY`-Kontrollnachricht erhalten hat, wodurch die zuvor invalidierte Beschreibung des Eingangspuffers des Empfangsprozesses gültige Werte erhält. Sobald dies in der *Broadcast*-Funktion erkannt wird, kann die Datenübertragung beginnen. Der Wurzelprozeß führt dieses Protokoll nur teilweise aus, da er keine `PCAST_READY`-Kontrollnachricht versendet, jedoch auch auf deren Eintreffen warten muß. Entsprechend wartet der *Endprozeß* nicht auf eine `PCAST_READY`-Kontrollnachricht, sondern sofort auf eingehende Daten in seinem Eingangspuffer.

6.6.2.2 Parallele DMA- und PIO-Transfers

Zur weiteren Leistungssteigerung wurde eine nebenläufige Übertragung von Daten mittels DMA durch den PSA und lokale PIO-Transfes durch die CPU entwickelt, die in Abb. 6.14 dargestellt ist. Während der Wurzelprozeß und der Endprozeß die unveränderten Sende- bzw. Empfangsroutinen des blockierenden *rendez-vous*-Protokolls verwenden können, müssen alle übrigen Prozesse, die an der Broadcast-Übertragung beteiligt sind, eine kombinierte Empfangs-

1. Zur Vereinfachung wird hier von einer normalisierten Rangverteilung im Kommunikator ausgegangen, in der der Wurzelprozeß den Rang 0 hat. Alle anderen Fälle, in denen der Wurzelprozeß einen anderen Rang hat, lassen sich durch eine Rangtransformation auf den normalisierten Fall zurückführen.

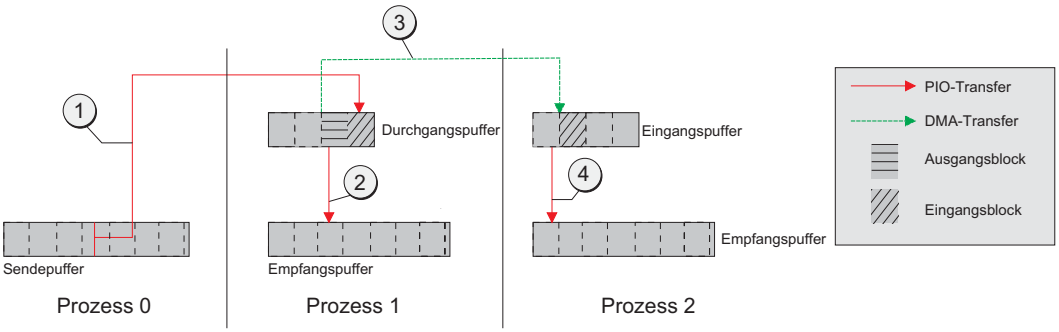


Abbildung 6.14: Datenfluß beim *pcast*-Protokoll zum Pipeline-basierten Rundsenden.

und Senderoutine durchlaufen. Jeder Datenblock, der vom Eingangspuffer in den Empfangspuffer kopiert wird, muß ebenfalls weitergesendet werden in den Eingangspuffer des folgenden Prozesses. Beide Transferoperationen können als PIO-Transfers von der CPU durchgeführt werden, was jedoch eine Sequentialisierung und damit eine Bandbreitenreduktion der beiden Transfers impliziert. Stattdessen kann das Weitersenden der Daten auch als DMA-Transfer gestaltet werden, wodurch die beiden Transfers überlappt ausgeführt werden können. Für den DMA-Transfer sind in diesem Fall die maximalen Bandbreiten zu erwarten, da beide Eingangspuffer, die als Quelle und Ziel der Übertragung dienen, reguläre SCI-Speichersegmente sind und somit eine einzige DMA-Operation die Daten überträgt (siehe Kapitel 7.2.1).

Dieses Vorgehen ist jedoch nur dann effizient, wenn *alle* Prozesse so verfahren können. Da Intra-Knoten-Transfers nicht via DMA abgewickelt werden können, wirkt die sequentielle PIO-Übertragung bei mehr als einem Prozeß der Applikation pro Knoten als limitierender Faktor auf die gesamte Pipeline.

6.6.3 Modellierung und Evaluierung des Pcast-Protokolls

Die Leistung einer Rundsendeoperation wird definiert als Verhältnis von Zeitdauer der gesamten Operation im Verhältnis zur bewegten Datenmenge, wodurch sich eine Bandbreite B_{bcast} ergibt. Auf Basis der eingeführten Größen gilt

$$(\ 6.16\ )\quad B_{bcast} = \frac{D}{t_{bcast}}$$

Die Zeitdauer t_{bcast} wird gemessen als maximale Zeitdauer eines der beteiligten Prozesse für die Ausführung von `MPI_Bcast()`, nachdem alle Prozesse zuvor eine Barrierensynchronisation durchgeführt haben. Andere, unsynchronisierte Meßmethoden würden je nach zeitlicher Abfolge des Aufrufs von `MPI_Bcast()` durch die einzelnen Prozesse zu unterschiedlichen Ergebnissen führen, wodurch keine reproduzierbaren Ergebnisse gewonnen werden könnten.

Die grundlegende Modellierung des Baum- und des Pipeline-Verfahrens zum Rundsenden wurde bereits im vorhergehenden Kapitel durchgeführt. So beschreibt (6.14) die Zeitkomplexität des Baum-Verfahrens in Abhängigkeit von der Nachrichtenlatenz und Prozeßzahl. Die Werte für die Modellrechnung des Pipeline-Verfahrens lassen sich nicht aus gemessenen Werten für Punkt-zu-Punkt-Kommunikation ableiten, da es sich bei diesem Verfahren um ein eigenständiges Protokoll handelt. Stattdessen müssen für die beiden Varianten des Verfahrens (sequentialisierte PIO-Transfers oder überlappende DMA- und PIO-Transfers) Modelle des Protokolls gebildet werden, die diese Transfers berücksichtigen. Die Basis dazu bildet die Zeitkomplexität des Verfahrens gemäß (6.15). Die Nachrichtenlatenz ist hier jedoch nicht die einer normalen Punkt-zu-Punkt-Nachricht: Der Wurzelprozeß versendet nur Daten, der Endprozeß empfängt nur Daten, alle anderen Prozesse empfangen und senden Daten. In der Pipeline bestimmt die langsamste Verarbeitungseinheit die gesamte Verarbeitungsgeschwindigkeit, daher

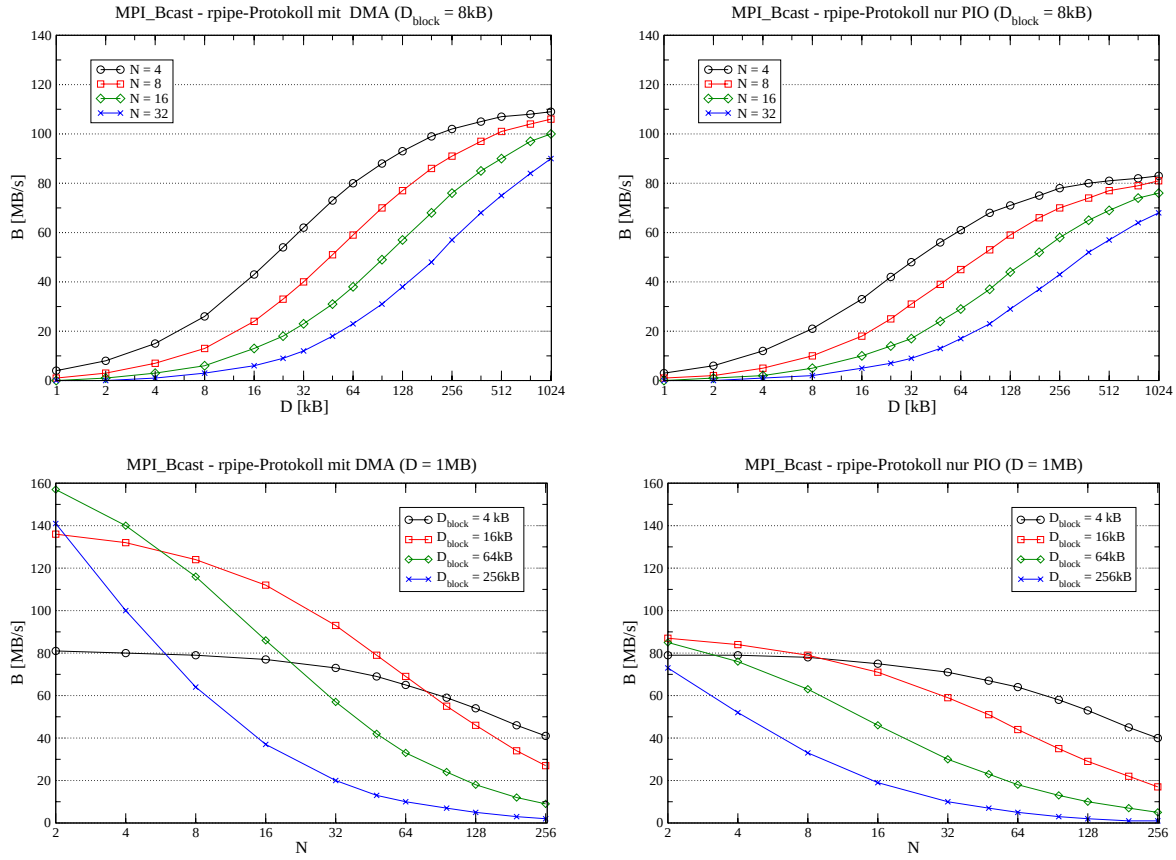


Abbildung 6.15: Bandbreite für das *pcast*-Protokoll gemäß Modell für DMA-Übertragung (*links*) und PIO-Übertragung (*rechts*)
Oben: als Funktion der Nachrichtengröße, Prozeßzahl *N* als Parameter
Unten: als Funktion der Prozeßzahl, Pipeline-Blockgröße D_{block} als Parameter

muß für die Nachrichtenlatenz die Latenz zum kombinierten Senden und Empfangen verwendet werden. Für die reine PIO-Variante ergibt sich diese für eine Blockgröße D_{block} [1] als Produkt aus der Zahl der zu übertragenden Blöcke und der Summe der Latenzen der Kopiervorgänge in den lokalen und entfernten Speicher, einer *store-barrier* und der Aktualisierung des Schreibzeigers zu

$$(6.17)\ t_{pcast_{PIO}} = \left(N + \frac{D}{D_{block}}\right) \cdot \left(\frac{D_{block}}{B_{cl}(D_{block})} + \frac{D_{block}}{B_{cr}(D_{block})} + l_{sb} + l_{rw}\right)$$

Wenn der Kopiervorgang in den entfernten Speicher mittels DMA stattfindet, können sich die beiden Kopiervorgänge überlappen, und die Nachrichtenlatenz ergibt sich zu

$$(6.18)\ t_{pcast_{DMA}} = \left(N + \frac{D}{D_{block}}\right) \cdot \left(l_{DMA} + max\left(\frac{D_{block}}{B_{cl}(D_{block})}, \frac{D_{block}}{B_{DMA}(D_{block})}\right) + l_{sb} + l_{rw}\right)$$

Nach diesem Modell gewonnene Simulationsergebnisse für die DMA- und PIO-Variante des *pcast*-Protokolls sind in Abb. 6.15 in unterschiedlicher Ansicht dargestellt.

Die Ergebnisse des realen Experiments sind in Abb. 6.16 dargestellt. Die höhere Startlatenz

1. Für das Modell wird davon ausgegangen, daß D_{block} ein echter Teiler der Datenmenge D ist. In der Implementation wird die Länge letzten Blocks ggf. entsprechend gekürzt; für das Modell ist dieser Fall jedoch unerheblich.

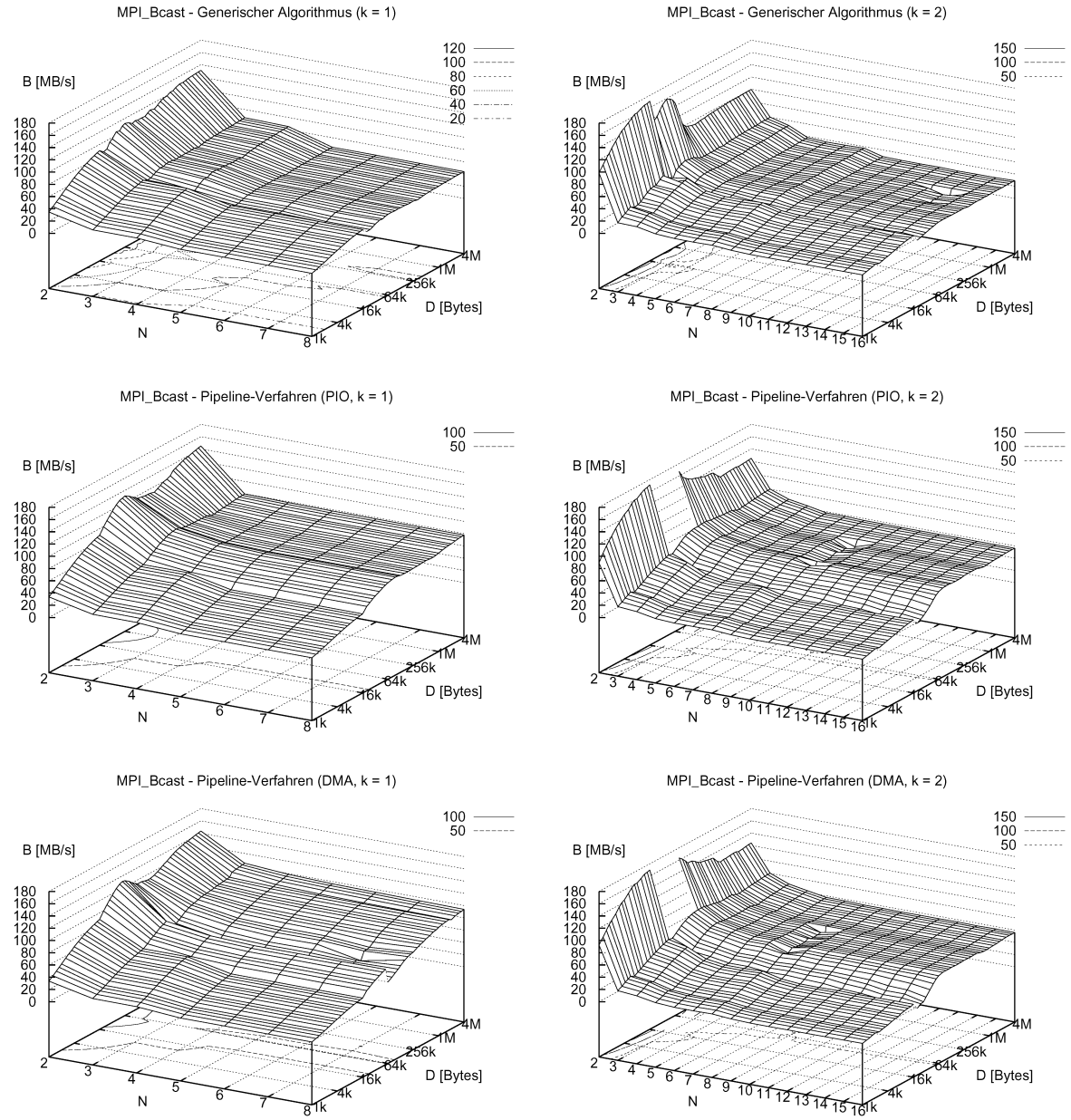


Abbildung 6.16: Experimentell bestimmte Bandbreite des Rundsendens via generischem Algorithmus (MPICH, *oben*), sowie *pcast*-Protokoll (Pipeline-Verfahren) mit PIO-Übertragung (*mitte*) und DMA-Übertragung (*unten*). *Links: k = 1; Rechts: k = 2*

der DMA-Übertragung macht es erforderlich, die Blockgröße zu erhöhen, um die Bandbreite der gesamten DMA-Übertragung nicht zu weit absinken zu lassen. Dadurch sinkt die Effizienz der Pipeline durch Anstieg der Füllzeit ab. Im SMP-Betrieb mit mehr als einem Prozeß pro Knoten hat das überlappende PIO-DMA-Verfahren gegenüber dem sequentiellen PIO-Verfahren keine höhere Bandbreite, da in diesem Fall Intra-Knoten-Transfers, die nicht über DMA abgewickelt werden (können), die langsamste Stufe in der Pipeline sind und somit deren Leistung bestimmen.

Für kleine Nachrichtengrößen im Bereich des *short*-Protokolls ist das *pcast*-Protokoll langsamer als das herkömmliche Verfahren, da hier alleine für die Initialisierung der Datenübertragung durch eine Kontrollnachricht die gleiche Kommunikation wie beim gesamten herkömmlichen Verfahren erforderlich ist. Aus diesem Grund wird das *pcast*-Protokoll erst ab einer minimalen Nachrichtengröße D_{pcast} verwendet. Wenn zur Übertragung der Nachrichten

im herkömmlichen Verfahren das *rendez-vous*-Protokoll verwendet werden muß, ist der Aufwand zur Initialisierung der Übertragung höher als beim *pcast*-Protokoll; gegenüber dem *eager*-Protokoll ist das *pcast*-Protokoll jedoch aufwendiger.

6.7 Reduktionsoperationen

Reduktionsoperationen verbinden den Versand und Empfang von Vektoren mit deren Verknüpfung. Dazu sind in MPI die gängigen arithmetischen und logischen Verknüpfungen vordefiniert; zusätzlich kann der Benutzer eigene Funktionen als Reduktionsoperationen definieren. Reduktionsoperationen in MPI müssen assoziativ sein, damit das Ergebnis der Verknüpfung nicht von der Reihenfolge abhängt, in der die Prozesse miteinander Punkt-zu-Punkt-Kommunikation betreiben, wobei jeweils zwei der Vektoren miteinander verknüpft werden. Die Kommutativität ist eine optionale Eigenschaft, die die freie Wahl der Operanden bei der lokalen Verknüpfung zweier Vektoren ermöglicht, womit beliebige Kommunikationstopologien möglich sind. Alle vordefinierten Reduktionsoperationen sind kommutativ.

6.7.1 Allgemeine Optimierungen

Neben verbesserten Kommunikationsalgorithmen zur Leistungssteigerung der verschiedenen Typen von Reduktionsoperationen wurden zwei allgemein nutzbare Maßnahmen entwickelt und eingeführt, die Reduktionsoperationen beschleunigen.

6.7.1.1 Integration von SIMD-Befehlen

Die Reduktionsoperationen für die MPI-Basistypen sind in MPICH als festverdrahtete C-Funktionen ausgeführt. Sie haben für die Verknüpfung zweier Vektoren A und B mittels eines arithmetischen oder binären Operators ' $\oplus$ ' die allgemeine Form $A[i] = A[i] \oplus B[i]$. Für einen logischen Operator wird stattdessen ein Vergleich mit anschließender Zuweisung durchgeführt. In SCI-MPICH wurden die festverdrahteten Funktionen zunächst durch umsetzbare Funktionszeiger ersetzt, was den Einsatz alternativer Funktionen zur Verknüpfung der Vektoren ermöglicht.

Die SIMD-Erweiterungen des x86-Befehlssatzes für die IA-32-Architektur (MMX und SSE *[33]*) wurden von K. Scholtyssik[1] verwendet, um optimierte Reduktionsoperationen zu entwikkeln, die auch in dieser Arbeit verwendet wurden. Durch die Nutzung von Funktionszeigern für die Reduktionsoperationen können diese für alle Reduktionsverfahren alternativ eingesetzt werden und bringen für die unterstützten Datentypen z.T. erhebliche Leistungsgewinne. Diese rühren von der CPU-internen Parallelverarbeitung sowie von optimierten Speicherzugriffsverfahren (*prefetching*) her. Aktuelle C-Compiler *[167,168]* konnten trotz Unterstützung der SIMD-Instruktionen keinen gleichwertigen Code für die in C notierten Ausdrücke erzeugen, insbesondere nicht für logische und binäre Verknüpfungen. Dies macht die Notwendigkeit von individuell optimiertem Code deutlich. Das Verfahren erlaubt es, Optimierungen für andere CPU-Typen mit entsprechenden Befehlssätzen ebenso einfach einzubinden. Die SIMD-basierten Operationen werden automatisch, also sowohl für alle hier vorgestellten Reduktionsverfahren als auch für die Standardalgorithmen verwendet.

6.7.1.2 Direktreduktion

Bei den bekannten Verfahren zur Durchführung einer Reduktion ist es erforderlich, einen lokal vorhandenen Vektor (abgelegt im *Reduktionspuffer*) mit einem Vektor, der von einem anderen Prozeß empfangen wird, zu verknüpfen, so daß der reduzierte Ergebnisvektor wiederum im Reduktionspuffer liegt. Die herkömmliche Vorgehensweise dazu besteht aus der Allokation eines Zwischenpuffers, in dem der zu empfangende Vektor abgelegt wird, nachdem er vom Sender in den Eingangspuffer geschrieben wurde. Anschließend wird die Verknüpfungsoperation zwi-

1. RWTH Aachen, Lehrstuhl für Betriebssysteme

schen Zwischenpuffer und Reduktionspuffer durchgeführt. Dieses Vorgehen erfordert zwei Kopiervorgänge und eine Verknüpfungsoperation über die gesamte Vektorlänge (siehe Abb. 6.17 *links*).

Für SCI-MPICH wurde das Verfahren der *Direktreduktion* entwickelt und eingeführt. Dabei wird als Zieladresse für den zu empfangenden Vektor nicht die Adresse eines Zwischenpuffers, sondern die Adresse des Reduktionspuffers angegeben. Gleichzeitig wird die Empfangsaufforderung für den eingehenden Vektor so initialisiert, daß sie alle erforderlichen Informationen zur Durchführung der Reduktion enthält. Sobald die Daten im Eingangspuffer vorliegen, überprüft die Funktion, die die Daten von dort zur Zieladresse übertragen soll, ob in der zugeordneten Empfangsaufforderung gültige Reduktionsinformationen eingetragen sind. Wenn dies der Fall ist, werden die Daten nicht mit der herkömmlichen Kopierfunktion übertragen, sondern bereits in diesem Moment mit den Daten des Reduktionspuffers verknüpft (Abb. 6.17 *rechts*). Das Verfahren der Direktreduktion wird von dem statischen *eager*-Protokoll und dem nicht-blockierenden, PIO-gestützten *rendez-vous*-Protokoll unterstützt.

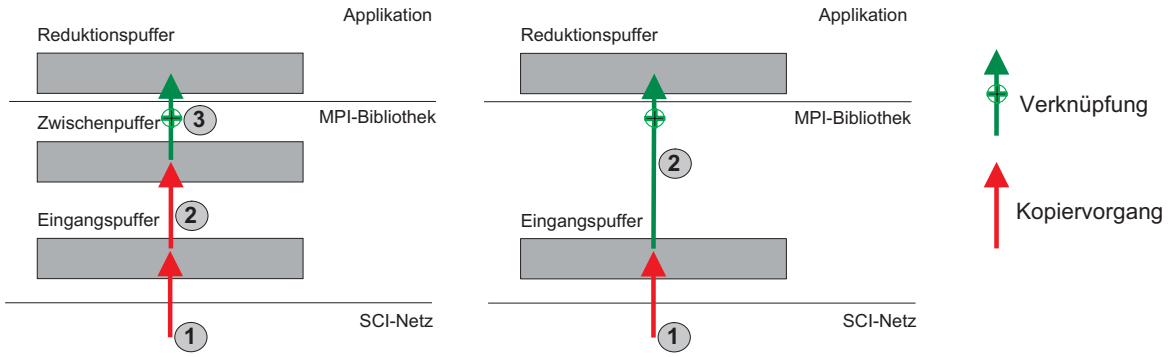


Abbildung 6.17: Herkömmliche Reduktion (*links*) und Direktreduktion in SCI-MPICH (*rechts*)

Die Direktreduktion spart gegenüber dem herkömmlichen Verfahren einen Kopiervorgang. Wie sich dies auf die effektive Leistung des gesamten Vorgangs, den Zeitraum vom Beginn des Versands des Vektors bis zur Ablage des letzten reduzierten Elements im Zielpuffer, auswirkt, läßt sich unterschiedlich bewerten:

- Die *absolute* Verringerung der Zeitdauer des Vorgangs hängt wesentlich von der Größe des Vektors in Relation zu den Caches ab.
- Die *relative* Verringerung hängt zusätzlich von der Netzbandbreite im Verhältnis zu der Kopier- bzw. Reduktionsbandbreite ab.

Falls der Vektor komplett in den CPU-Cache der ersten Ebene[1] paßt, fällt nach der Kopieroperation (in Abb. 6.17 *links* Vorgang '2'), die den Vektor in den Cache lädt, die Reduktionsoperation deutlich weniger ins Gewicht, als wenn bei beiden Vorgängen Daten aus dem Hauptspeicher gelesen werden müssen. Dabei muß unterschieden werden zwischen Vektoren, die mit dem *eager*- oder mit dem *rendez-vous*-Protokoll übertragen werden.

Beim *eager*-Protokoll erfolgt sowohl der Kopiervorgang vom Eingangspuffer in den Zwischenpuffer sowie die Reduktion zwischen Zwischenpuffer und Zielpuffer in jeweils einem Schritt auf dem gesamten Vektor. Sofern der Vektor in den Cache paßt, können von dort auch die Daten für die folgende Reduktionsoperation gelesen werden. Dadurch fällt die absolute Verringerung der Laufzeit durch die Direktreduktion in diesem Fall geringer aus. Erst bei großen Puffern für das *eager*-Protokoll, die nicht mehr vom CPU-Cache der ersten Ebene gehalten wer-

1. Unter Berücksichtigung der höheren Zugriffslatenz gelten die Betrachtungen analog auch für CPU-Caches niedrigerer Ebenen.

den können, wächst der Vorteil der Direktreduktion stärker an.

Beim *rendez-vous*-Protokoll wird der Kopiervorgang in einer Zahl von Kopieroperationen auf kleineren Blöcken durchgeführt (vergl. Kapitel 4.2.4.1), während die Reduktion wiederum in einem Vorgang auf dem gesamten Vektor erfolgt. Dies hat zur Folge, daß trotz des vorhergegangenen Kopiervorgangs der Vektor nicht mehr (vollständig) im Cache liegt, wenn die Reduktion durchgeführt wird. Da der letzte Teil des Vektors zuletzt kopiert wird, die Reduktion aber wieder beim Start des Vektors anfängt, muß der Vektor dazu wieder vollständig aus dem Hauptspeicher gelesen werden. In diesem Fall führt die Direktreduktion zu einer weitaus stärkeren absoluten Verringerung der Zeitdauer.

Im Modell läßt sich die Zeitdauer T_r für die gewöhnliche Reduktion und die Direktreduktion T_{dr} für einen Vektor der Länge D wie folgt als Summe der Latenzen für Datenübertragung und lokale Kopieroperation (entsprechend der Nachrichtenlatenz L_{msg} des jeweiligen Protokolls) und der Latenz der Reduktionsoperation L_{red} beschreiben:

$$(6.19)\quad T_r(D) = L_{msg}(D) + L_{red}(D)$$

Bei der Direktreduktion entfällt der explizite Aufruf der Reduktionsoperation und somit die Latenz L_{red}; stattdessen wird die lokale Kopieroperation ersetzt durch die Reduktionsoperation, die die Daten aus dem Eingangs- und dem Zielpuffer im Zielpuffer miteinander verknüpft:

$$(6.20)\quad T_{dr}(D) = L'_{msg}(D)$$

Wie sehr sich die eingesparte Kopieroperation auf die effektive Bandbreite der Reduktion auswirkt, hängt von der Bandbreite der Datenübertragung im Verhältnis zu lokalen Kopieroperationen bzw. Verknüpfungen ab. Letztere werden beeinflußt durch die Lage der Daten in der Speicherhierarchie, was wiederum abhängig von der Vektorlänge im Verhältnis zur Cachegröße ist. Auch die Frage, wie hoch die Bandbreite einer reinen Kopieroperation im Verhältnis zu einer Verknüpfungsoperation mit implizitem Kopiervorgang bei gleicher Lage der Daten in der Speicherhierarchie ist, spielt eine Rolle, ist aber wiederum abhängig vom Typ der Verknüpfung. Die Vielzahl der Fälle, die bei einer Modellierung unterschieden werden müßten, läßt es sinnvoller erscheinen, das Verhalten anhand einiger Experimente zu analysieren.

Abbildung 6.18 *links* zeigt die verringerte Latenz bei einer Reduktion mit zwei Prozessen für unterschiedliche Vektorlängen. Der Vorteil der Direktreduktion nimmt mit der Vektorlänge zu.

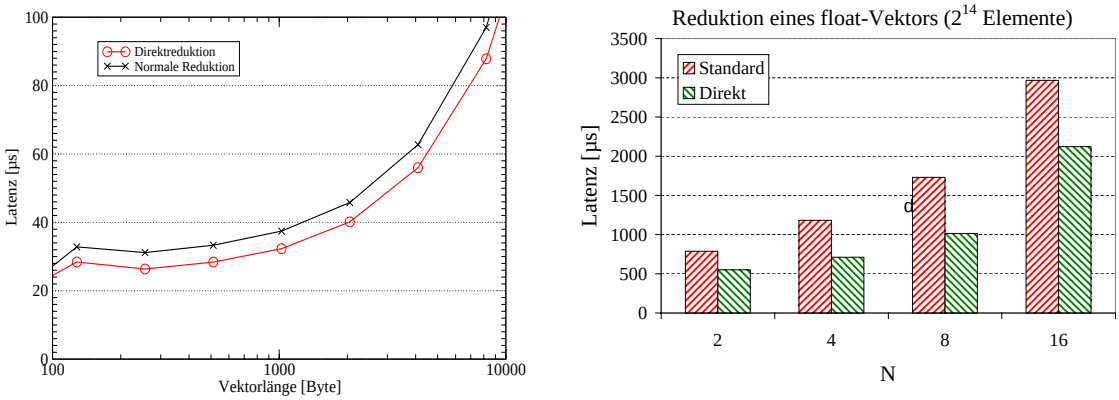


Abbildung 6.18: *Links:* Verringerung der Kommunikationslatenz durch Direktreduktion für Reduktionsoperation zwischen 2 Prozessen
Rechts: Skalierung des Leistungsgewinns mit der Prozeßzahl

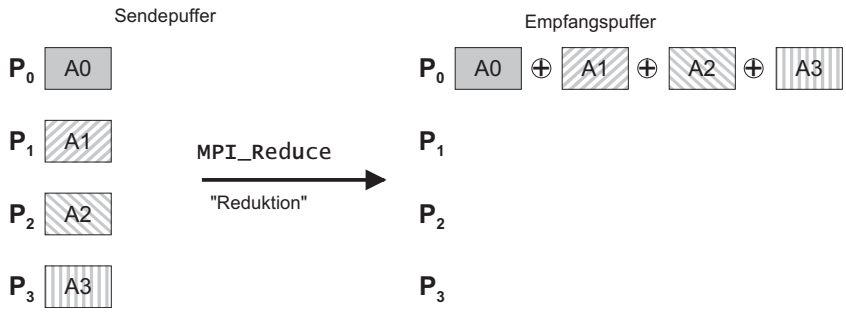


Abbildung 6.19: Datenverteilung bei der Reduktion (`MPI_Reduce`)

Bei einer Reduktion mit *N* Prozessen muß die Reduktionsoperation mehrfach angewandt werden. Arithmetisch sind mindestens $N-1$ Aufrufe der Reduktionsoperation notwendig. Je nach verwendetem Kommunikationsmuster können die Reduktionen parallelisiert werden, so daß beispielsweise nur $ld(N)$ sequentiell ausgeführte Reduktionsoperationen anfallen. In jedem Fall akkumuliert sich der Vorteil der Direktreduktion, so daß bei konstanter Vektorlänge der Gewinn mit zunehmender Zahl der Prozesse zunimmt. Dies wird am Beispiel eines Reduktionsverfahrens mit binärer Baumtopologie deutlich, bei dem ein Vektor von 2^{14} `float`-Zahlen mit gewöhnlicher Reduktion bzw. Direktreduktion addiert wurde (siehe Abb. 6.18 *rechts*).

Ein weiterer positiver Effekt der Direktreduktion neben der Erhöhung der Bandbreite der Reduktion selber ist die Vermeidung von Cacheverschmutzung: Die zusätzliche lokale Kopie beim herkömmlichen Verfahren verdrängt zwangsläufig Daten aus den Caches, die möglicherweise anschließend erneut geladen werden müssen.

Die Nutzung der Direktreduktion ist für den Benutzer transparent, da er nur Quell- und Zielpuffer bei einer Reduktion angibt. Der Zwischenpuffer beim herkömmlichen Verfahren wird von der MPI-Bibliothek allokiert. Jedoch muß die Verknüpfungsoperation kommutativ sein, da die Reihenfolge der Operatoren durch die Pufferanordnung vorgegeben ist. Dies ist jedoch keine wesentliche Einschränkung, da alle vordefinierten Reduktionsoperatoren sowohl assoziativ als auch kommutativ sind. Falls der Benutzer eine nicht-kommutative Reduktionsoperation definiert, wird für deren Nutzung das herkömmliche Verfahren verwendet.

6.7.2 Reduktion kleiner Vektoren

Die einfache Reduktion von Daten, also die Verknüpfung von *N* Vektoren aller *N* Prozesse zu einem Ergebnisvektor, ist eine N:1-Kommunikation: Der Ergebnisvektor liegt zum Abschluß der Operation beim Wurzelprozeß (Abb. 6.19). Zur Optimierung dieser Kommunikationsform wurde unterschieden zwischen "kleinen" und "großen" Vektoren.

Als *kleine Vektoren* sollen in diesem Zusammenhang Vektoren bezeichnet werden, die beim Versand nicht im *rendez-vous*-Protokoll übertragen werden, sondern unmittelbar beim Empfänger abgelegt werden können (*short*- oder *eager*-Protokoll). Für diese Vektoren spielt die benötigte Rechenzeit im Verhältnis zur Kommunikationslatenz eine untergeordnete Rolle: Falls die Daten im CPU-Cache liegen (was bei kleinen Vektoren sehr wahrscheinlich ist, da der Inhalt zumeist zuvor bearbeitet wurde), kann bei einer Taktfrequenz f_{CPU} von $1GHz$ davon ausgegangen werden, daß eine Fließkommaoperation für den betrachteten Vektor nicht mehr als 4 Taktzyklen benötigt, entsprechend $\tau_{FLOP} = 4ns$. Zur Initiierung einer Datenübertragung im *rendez-vous*-Protokoll sind 2 Kontrollnachrichten notwendig, deren Übertragung im Mittel nicht unter $2 \cdot L_{ctrl-pp} = 20\mu s$ (siehe Kapitel 4.2.2) dauert, sofern der Empfänger die Übertragung bereits erwartet. Die typische Latenz τ_{FLsend} zur Übertragung einer Fließkommazahl liegt dabei zwischen und $0,08\mu s$ (für einen langen Vektor) und $10\mu s$ (für eine Nachricht mit einer einzelnen Fließkommazahl). Dies ergibt ein Verhältnis von Kommunikationslatenz zu

Verknüpfungslatenz, definiert als

$$(6.21)\ r_{reduce} = \frac{\tau_{FLsend}}{\tau_{FLOP}},$$

von $2\times10^1 < r_{reduce} < 2,5\times10^3$. Dieser Wert von r_{reduce} macht es für eine hohe effektive Leistung vorrangig, die Kommunikationslatenz zu minimieren, was mit einer Übertragung im *short*- oder *eager*-Protokoll innerhalb eines Baum-basierten Kommunikationsmusters am besten erreicht wird[1]. Hierbei ist zudem für alle Prozesse, die nur als Blätter des Baumes auftauchen, die Abarbeitung der Funktion unabhängig von allen anderen Prozessen möglich.

Der in MPICH implementierte Reduktionsalgorithmus arbeitet mit einer solchen Baum-basierten Kommunikationsstruktur. Für die gegebene Plattform sollte jedoch nicht "abwärts", d.h. in Richtung eines Partnerprozesses mit niedrigerem Rang, sondern stets "aufwärts", also zu einem Partnerprozeß mit höherem Rang, kommuniziert werden, um die Zahl der in eine Kommunikation involvierten SCI-Segmente zu minimieren. Zusätzlich kann es bei kleinen Vektoren sinnvoll sein, den *fan-in* f_{in} des Baums zu erhöhen. Bei kleinen Datenmengen führt die dadurch verstärkte Konzentration der Schreibvorgänge auf einen Knoten kaum zu Konflikten bei der Datenübertragung (siehe dazu auch Kapitel 6.5 zur Barrierensynchronisation), reduziert aber sehr effektiv die Zahl der Kommunikationsstufen S (Tiefe des Baums) gemäß $S = \log_{f_{in}} N$ und verkürzt somit die Kommunikationszeit. Die Tatsache, daß dabei der Empfängerprozeß $f_{in} > 1$ Reduktionsoperationen durchführen muß, fällt bei dem beschriebenen Wert von r_{reduce} kaum ins Gewicht. Abb. 6.21 zeigt einen Vergleich des herkömmlichen Reduktionsverfahrens mit dem optimierten SCI-MPICH-Verfahren mit $f_{in} = 4$ und der Direktreduktion das *eager*- und *rendez-vous*-Protokoll.

Die bessere Skalierung der Reduktionsleistung durch den höheren *fan-in* und die Direktreduktion ist nicht nur für die Latenz kleiner Nachrichten, sondern auch für die Bandbreite langer Vektoren offensichtlich. Die Latenz ist um über 40% geringer, während die effektive Bandbreite in etwa verdoppelt wird. Neben dem Überblick über das Skalierungsverhalten ist die Betrachtung eines ausgewählten Falls mit allen Details aufschlußreich, wie dies anhand Abb. 6.20 möglich ist. Es wurden Vektoren mit $f_{in} \in \{2, 4, 8\}$ jeweils mit und ohne Direktreduktion reduziert. Die Latenz kleiner Vektoren im betrachteten Fall geht proportional zum *fan-in* zurück; auch bei $f_{in} = 8$ ist noch keine Sättigung bei dem jeweils empfangenden Knoten festzustellen.

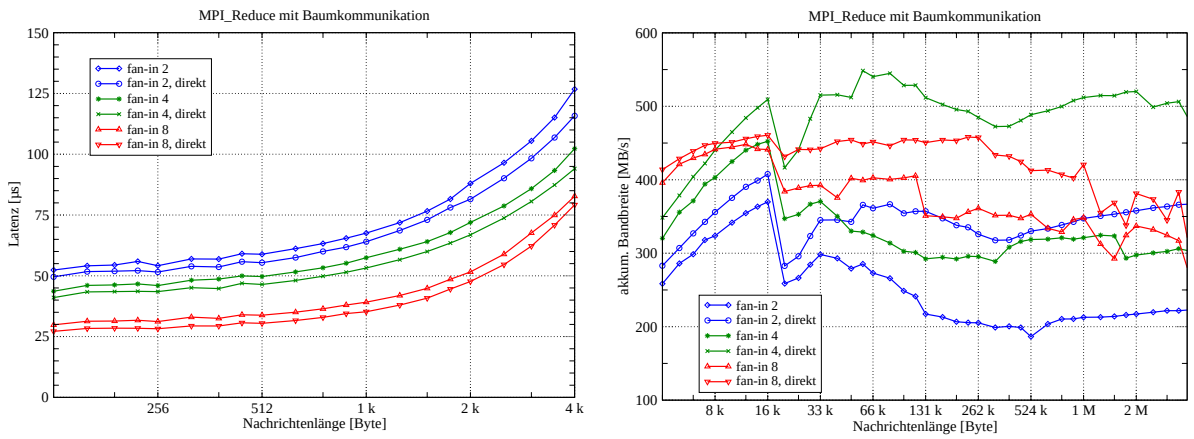


Abbildung 6.20: Reduktion mit verschiedenen *fan-in*-Parametern und ohne/mit Direktreduktion ($N = 8$). *links:* Latenz kleiner Vektoren; *rechts:* Bandbreite großer Vektoren

1. Dieser Vergleich bezieht sich auf die bekannten Standardverfahren zur kollektiven Kommunikation gemäß Kapitel 6.1 sowie die in den folgenden Kapiteln vorgestellten Verfahren.

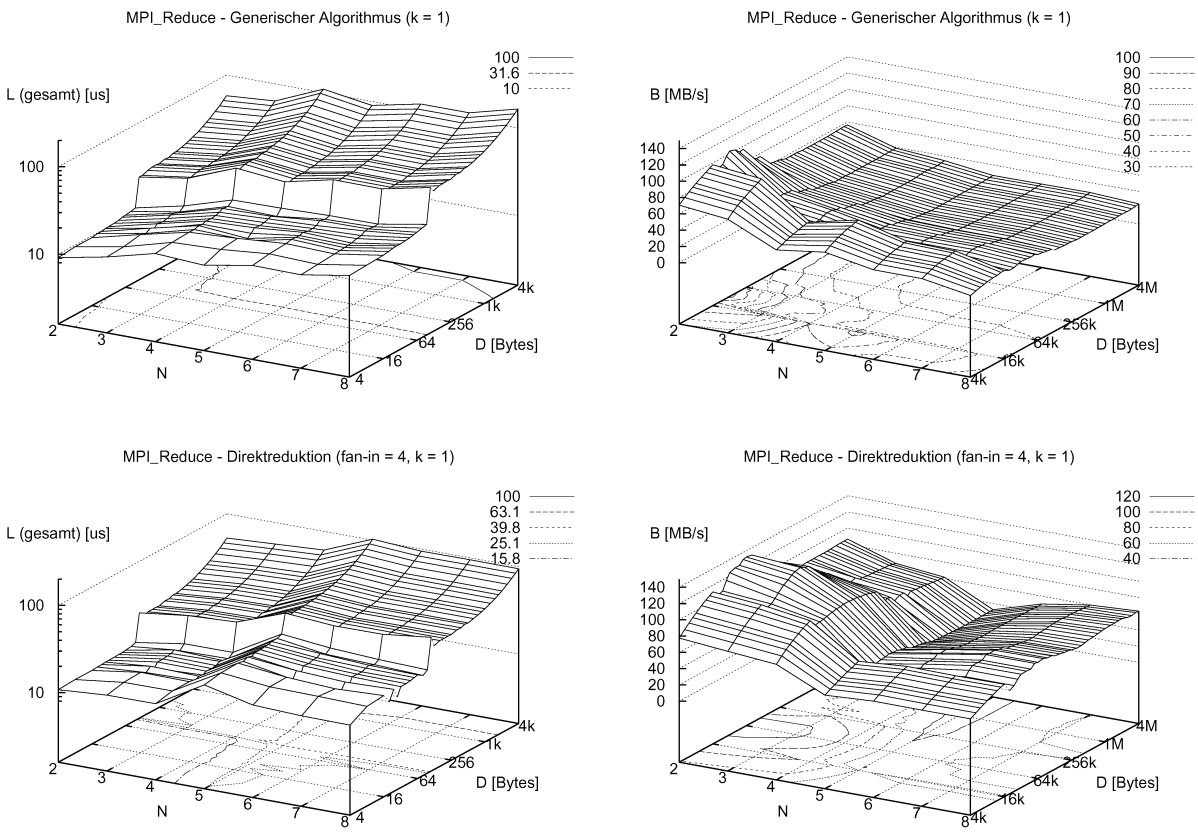


Abbildung 6.21: Herkömmliches und optimiertes Reduktionsverfahren im Vergleich: generischer MPICH-Algorithmus (*oben)*, SCI-MPICH-Verfahren mit $f_{in} = 4$ und Direktreduktion (*unten)*. *Links:* Latenz für kleine Vektoren; *rechts:* Bandbreite für große Vektoren.

Bei der Bandbreite für lange Vektoren wird bei 8 kB Vektorlänge jedoch deutlich, daß es für $f_{in} = 8$ zur Sättigung des betreffenden Knotens kommt; die Bandbreite steigt nicht weiter an. Für große Vektoren ist daher auf der P3-Plattform $f_{in} = 4$ eine gute Wahl.

Die Analyse der Ergebnisse läßt den Schluß zu, daß ein optimaler Baum-basierter Reduktionsalgorithmus den *fan-in* dynamisch entsprechend der Vektorlänge wählt. Dies ist bei SCI-MPICH einfach zu realisieren, da die Kommunikationspartner bei jeder Reduktion dynamisch bestimmt werden. Die Direktreduktion hat sich in jedem Fall als vorteilhaft erwiesen, insbesondere wenn die Vektorlänge oberhalb der Cachegrößen liegt.

6.7.3 Reduktion großer Vektoren

Als *große Vektoren* sollen in diesem Zusammenhang Vektoren bezeichnet werden, die mit dem *rendez-vous*-Protokoll übertragen werden. Für große Vektoren ergibt sich dabei durch Übertragungsbandbreiten zwischen 100 und 200 MB/s ein Wert von r_{reduce}, der zwischen 10 und 20 liegt. Es wird deutlich, daß für diese Fälle die Zeit zur Verknüpfung gegenüber der Kommunikationszeit nicht weiter vernachlässigt werden kann, wie es für kleine Vektoren erlaubt ist.

Dies hat Einfluß auf die Wahl der Kommunikationsstruktur: Bei einem Baum mit *fan-in* f_{in} sind für den Wurzelprozeß $S_{Baum} = \log_{f_{in}} N$ Kommunikationsschritte erforderlich, wobei für diesen Prozeß $V_{Baum} = (f_{in} - 1) \cdot \log_{f_{in}} N$ Vektorverknüpfungen anfallen, um den Ergebnisvektor zu erhalten. Ein höherer *fan-in* wirkt sich hier also kontraproduktiv aus; im Gegenteil muß versucht werden, die Verknüpfungsoperationen so gleichmäßig wie möglich auf alle beteiligten Prozesse zu verteilen, um so eine Minimierung der Zeit zu erreichen, die ein Prozeß (der Wurzelprozeß) für die Durchführung der Reduktion benötigt. Trotz der Leistungsgewinne

durch Direktreduktion und erhöhten *fan-in* bei der Baumtopologie ist dieses Verfahren somit für lange Vektoren durch das *store-and-forward*-Prinzip benachteiligt. Daher wurden in SCI-MPICH spezielle Verfahren für große Vektoren entwickelt.

Dazu wurde zunächst der Algorithmus von R. Rabenseifner *[122]* adaptiert. Dieser Algorithmus erreicht bei den entsprechend längeren Rechenzeiten dieser Vektoren (im Verhältnis zur zugehörigen Kommunikationslatenz) durch die bessere Parallelisierung von Vektorverknüpfungen und Kommunikation trotz höherem Kommunikationsaufkommen einen höheren Durchsatz als Baum-basierte Algorithmen. Anstelle einer Komplexität von $O(\log N) \cdot O(D)$ für die Reduktion von Vektoren der Länge D mit N Prozessen erreicht dieser Algorithmus eine Komplexität von $O(\log N) + O(D)$. In SCI-MPICH wurde durch Verzicht auf eine Kopierstufe des Rabenseifner-Algorithmus der Durchsatz erhöht; die Daten werden nun analog zur Direktreduktion direkt in den Kommunikationspuffern miteinander verknüpft, wenn die Verknüpfungsoperation kommutativ ist. Eine Überlappung von Kommunikation und Berechnung findet bei diesem Algorithmus jedoch nicht statt. Dies ist aufgrund der Abhängigkeit der Kommunikationspuffer von vorhergegangenen Berechnungen selbst unter Verwendung der in Kapitel 7 vorgestellten Verfahren nicht möglich.

Eine nahezu optimale Parallelisierung von Kommunikation und Berechnung kann jedoch, aufbauend auf das in Kapitel 6.6 vorgestellte Rundsende-Verfahren, wiederum durch *Pipelining* erreicht werden. Dazu wurde aufbauend auf dem *pcast*-Protokoll das *rpipe*-Protokoll (*r*educe *pipe*line) entwickelt, dessen Datenfluß für eine Reduktion von 3 Prozessen mit Wurzelprozeß 0 in Abb. 6.22 dargestellt ist. Zur Minimierung der SCI-Segmentnutzung verläuft die Kommunikation auch in diesem Fall in Richtung aufsteigender Prozeßränge, so daß für Wurzelprozeß P_w die Pipeline beim Sendeprozeß $P_s = (P_w + 1) \lozenge N$ beginnt. Analog zum *pcast*-Protokoll überträgt dieser seine Daten in den *Durchgangspuffer* des nächsten Prozesses (Punkt 1 in Abb. 6.23). Im Unterschied zum *pcast*-Protokoll beinhaltet dieser Puffer nicht zwei, sondern drei Pipelinestufen: einen *Eingangsblock* für die eintreffenden Daten, einen *Verknüpfungsblock*, in dem die im Schritt zuvor eingetroffenen Daten mit dem entsprechenden Teil des Vektors im Sendepuffer gemäß der Reduktionsoperation verknüpft werden (2), und einen *Ausgangsblock*, aus dem die im Schritt zuvor verknüpften Daten in den *Durchgangs-* bzw. *Eingangspuffer* des nächsten Prozesses weitergesendet werden (3). Während die Verknüpfung durch die CPU erfolgen muß, erfolgt die Weiterleitung der Daten überlappend durch einen DMA-Transfer. Der Wurzelprozeß am Ende der Pipeline verknüpft zunächst die eingegangenen Daten aus dem Verknüpfungsblock mit dem Vektor aus seinem Sendepuffer (4) und kopiert das Ergebnis von dort in den Empfangspuffer (5). Diese zusätzliche Kopie ist nötig, da die Verknüpfungsoperationen so definiert sind, daß das Ergebnis stets einen der Vektoren, aus dem die Quelldaten der Verknüpfung kommen, überschreibt.

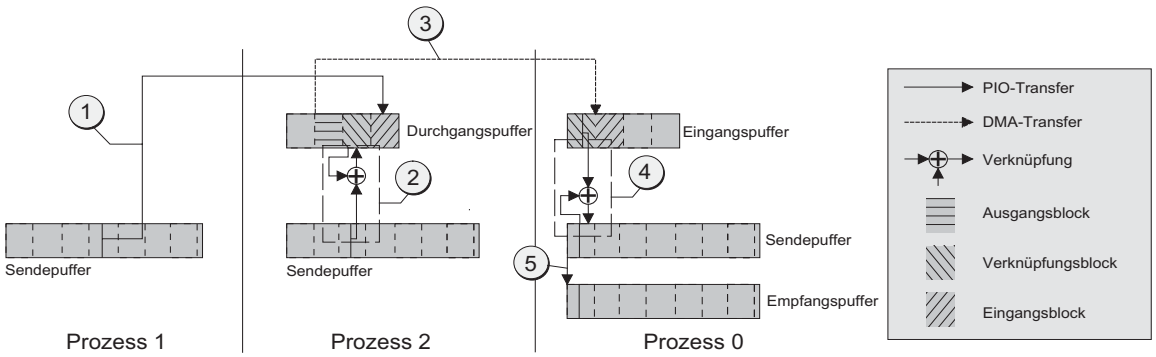


Abbildung 6.22: Datenfluß beim *rpipe*-Protokoll zur Reduktion im Pipeline-Prinzip. Hier: Reduktion von 3 Prozessen mit Prozeß 0 als Wurzelprozeß.

Gemäß dem Protokoll ist zu erwarten, daß der kritische Abschnitt der Pipeline beim Wurzelprozeß liegt, da dieser für jeden Block eine Verknüpfung und einen lokalen Kopiervorgang durchführen muß. Der Sendeprozeß muß nur einen Kopiervorgang in entfernten Speicher durchführen, während die Weiterleitungsprozesse eine lokale Verknüpfung überlappend mit einem DMA-Transfer in entfernten Speicher durchführen. Eine Analyse der tatsächlichen Verhältnisse im Rahmen der Evaluierung muß zeigen, ob diese Vermutung zutrifft. Dazu werden in Abb. 6.23 die Bandbreiten für die Reduktion großer Vektoren zwischen einer festen Zahl von Prozessen für das herkömmliche MPICH-Verfahren, den Rabenseifner-Algorithmus sowie das vorgestellte *rpipe*-Protokoll mit und ohne DMA-Unterstützung verglichen. Die Ergebnisse werden in Kapitel 6.7.6.1 zusammen mit den Ergebnissen der Globalreduktion diskutiert.

6.7.4 Globalreduktion großer Vektoren

Die Globalreduktion (MPI-Funktion `MPI_Allreduce`) arbeitet wie die Reduktion, jedoch liegt der Ergebnisvektor nach Abschluß der Reduktion in den Empfangspuffern *aller* Prozesse (Abb. 6.24). Da dazu, wie bei der Reduktion, alle zusammengehörigen Elemente der Vektoren (oder deren Verknüpfungen) über *einen* Prozeß geführt werden können, besteht eine mögliche Implementation der Globalreduktion aus der Hintereinanderausführung einer optimalen Reduktion und Rundsendeoperation. In diesem Fall bedeutet dies die Verwendung der vorgestellten Pipeline-Verfahren.

6.7.4.1 Globalreduktion durch Pipelining

Gegenüber der isolierten Ausführung der Pipeline-gestützen Funktionen `MPI_Reduce` und `MPI_Bcast` ist eine Verbindung dieser beiden Operationen zu einer Operation mit einer einzi-

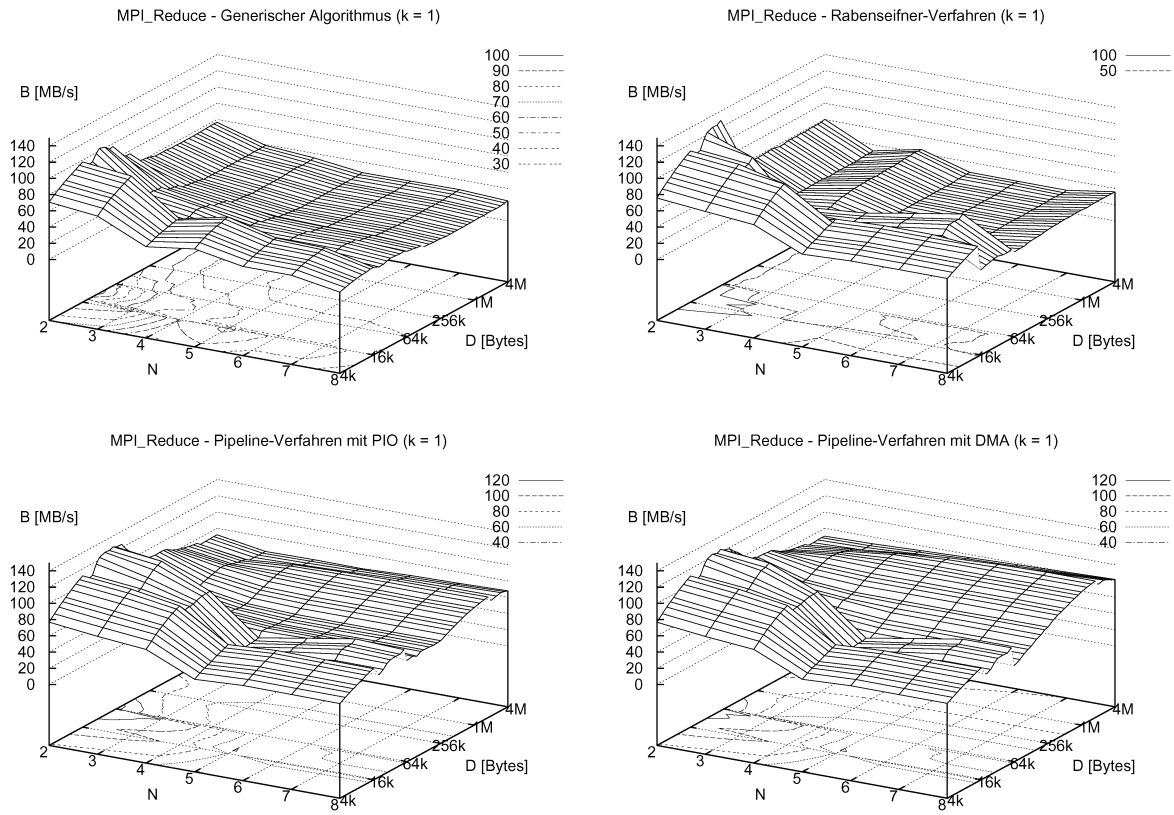


Abbildung 6.23: Vergleich der Reduktionsbandbreite von MPICH- und Rabenseifner-Algorithmus mit dem *rpipe*-Protokoll (mit und ohne DMA-Unterstützung).
oben: generischer MPICH-Algorithmus (binomial-Baum), Rabenseifner
unten: Pipeline-Verfahren (*rpipe*-Protokoll) ohne und mit DMA-Transfers

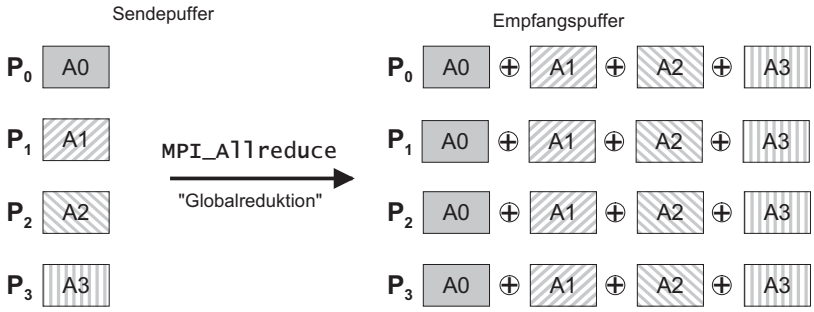


Abbildung 6.24: Datenverteilung bei der Globalreduktion (`MPI_Allreduce`)

gen Pipeline, die die doppelte Länge der einzelnen Pipelines hat, günstiger. Ansonsten fällt die Füllung der Pipeline zweimal an, nämlich beim Start der Reduktionspipeline und erneut beim Start der Rundsendepipeline. Bei Verwendung einer einzigen Pipeline wird beim Wurzelprozess der Reduktion nicht der komplette Vektor gesammelt, bevor er rundgesendet wird, sondern direkt nach der Verknüpfung wieder in die Rundsende-Pipeline geschrieben. Sofern die Datenmenge im Verhältnis der Blockgröße der Pipeline ausreichend groß ist, um eine Pipeline mit *2N* Verarbeitungseinheiten zu füllen, würde eine Globalreduktion eines Vektor der Länge *D* mit *N* Prozessoren unwesentlich länger dauern als eine Reduktion dieses Vektors auf *N* Prozessoren. Dieses Leistungsverhalten wird sich auf der gegebenen Plattform eines typischen Clusters jedoch nicht einstellen, da die für den Betrieb von zwei parallelen Pipelines erforderlichen Ressourcen fehlen. Schon bei der Pipeline zur Reduktionsoperation sind alle zum Datentransport nutzbaren Ressourcen (CPU, DMA-Baustein des PSA, Bandbreite des E/A-Systems) in Verwendung. Daher werden die Reduktion und das Rundsenden nacheinander in zwei unabhängigen Pipelines durchgeführt, womit die Ausführungszeit der Global-reduktion mittels Pipeline die Summe der Ausführungszeiten einer Reduktion und eines Rundsendens ist.

6.7.4.2 Globalreduktion gemäß Rabenseifner-Algorithmus

Bei 2-exponentiellem *N* ist für diese Operation der Algorithmus, der von R. Rabenseifner als generische Implementation vorgestellt wurde *[122]*, ebenfalls gut geeignet, da er die Reduktionsoperationen in diesem Fall optimal parallelisiert. Dies wird dadurch erreicht, daß die Prozesse paarweise jeweils die Hälfte ihres aktuellen Vektors gemäß dem binären Austauch (siehe auch Barrierensynchronisation Abb. 6.7) miteinander austauschen, und die so reduzierten Daten im nächsten Schritt wiederum zur Hälfte mit dem nächsten Partner austauschen. Am Ende dieser *ld(N)* Schritte hat jeder Prozeß ein *N*-tel des Ergebnisvektors bei sich liegen. Das Zusammenführen dieser Teilvektoren zum vollständigen Ergebnisvektor, der bei allen Prozessen vorliegt, erfolgt mit dem gleichen Verfahren wie in Kapitel 6.8 beschrieben. Die Darstellung in Abb. 6.25 illustriert das Verfahren für den Fall von 8 Prozessen. Fälle mit nicht-2-exponentiellen Prozeßzahlen erfordern zwei zusätzliche Kommunikationsschritte vor und nach den hier gezeigten Kommunikationsschritten, in denen jeweils der Quell- und Ergebnisvektor mit den ansonsten inaktiven Prozessen ausgetauscht wird.

Somit liegt die Zahl der Kommunikationsschritte in diesem Verfahren bei $2 \cdot ld(N)$, das maximale Kommunikationsvolumen für einen Prozeß bei $2 \cdot \frac{N-1}{N} \cdot D$, und das maximale Verknüpfungsvolumen bei $\frac{N-1}{N} \cdot D$. Das baum-basierte Verfahren benötigt bei einem *fan-in* von 2 ebenfalls $2 \cdot ld(N)$ Kommunikationsschritte; das maximale Kommunikationsvolumen für einen Prozeß liegt jedoch bei $2 \cdot ld(N) \cdot D$, und dieser Prozeß muß auch Vektoren mit einer Datenmenge von $ld(N) \cdot D$ miteinander verknüpfen. Dies macht offensichtlich, daß das Rabenseifner-Verfahren effizienter ist. Beim Pipeline-Verfahren liegt das Kommunikationsvolumen pro Prozeß bei $2D$ und das Verknüpfungsvolumen bei D, was etwas über dem entsprechenden Volumen beim Rabenseifner-Verfahren liegt. Jedoch kann die Reduktion beim

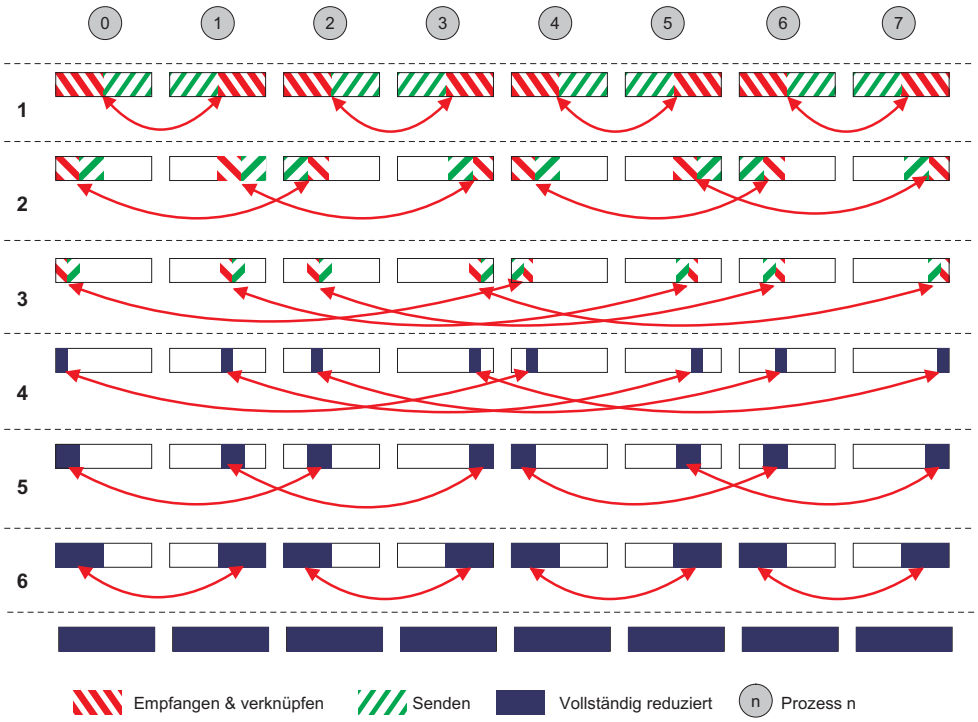


Abbildung 6.25: Globalreduktion nach Rabenseifner-Verfahren für 8 Prozesse.

Pipeline-Verfahren überlappt mit der Kommunikation abgewickelt werden. Zudem kann die Zahl der Kommunikationschritte nicht miteinander verglichen werden, da diese beim Pipeline-Verfahren nicht in der gleichen Form zu quantifizieren sind. Es müssen daher die beiden Modelle aus Kapitel 6.6.3 (Rundsenden) und Kapitel 6.7.6 (Reduktion) verwendet werden, um eine quantitative Aussage zur Leistung der Globalreduktion im Pipeline-Verfahren zu treffen.

6.7.5 Präfixreduktion

Die *Präfixreduktion* (MPI-Funktion `MPI_Scan`) ist eine Variante der Reduktion, bei der stets der Prozeß mit dem höchsten Rang in der Gruppe als Wurzelprozeß festgelegt ist. Im Unterschied zur Reduktion hat jeder Prozeß nach Abschluß der Präfixreduktion die Verknüpfung seines Vektors mit den Vektoren aller Prozesse mit niedrigerem Rang in seinem Empfangspuffer vorliegen (siehe Abb. 6.27).

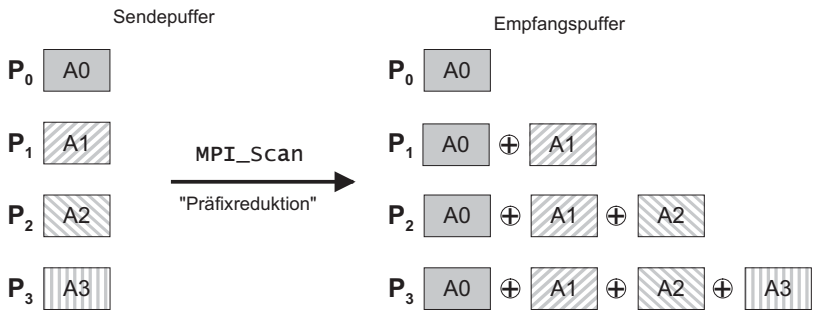


Abbildung 6.27: Datenverteilung bei der Präfixreduktion (`MPI_Scan`)

Eine naive Implementierung der Präfixreduktion wäre ein verketteter Baum, in dem jeweils Prozeß p an Prozeß $p+1$ sendet, was zu einer Laufzeit linearer Ordnung ($O(N \cdot D)$) führt. In MPICH wird jedoch ein optimierter Algorithmus verwendet *[135]*, der eine Laufzeit logarithmischer Ordnung aufweist ($O(\log(N \cdot D))$) und im folgenden als generischer MPICH-Algorith-

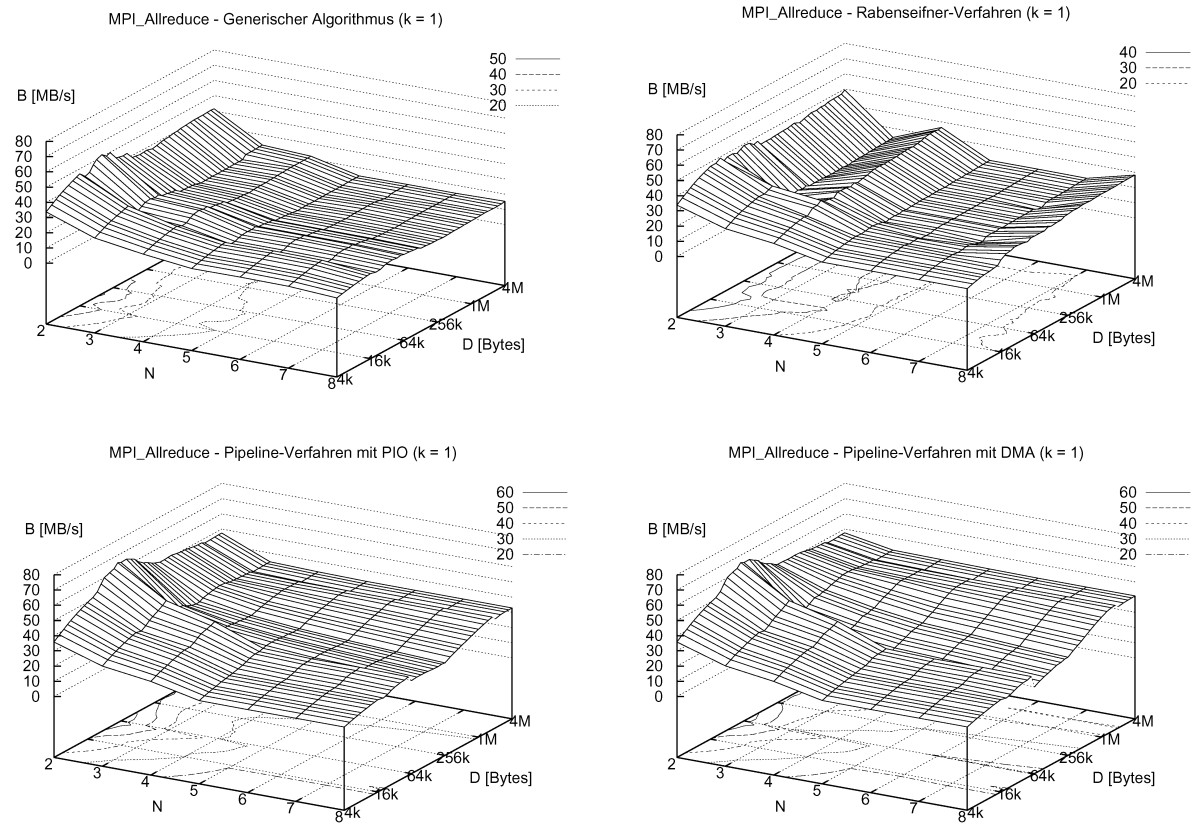


Abbildung 6.26: Vergleich der Bandbreite pro Prozeß für verschiedene Verfahren zur Globalreduktion
oben: Generischer MPICH-Algorithmus, Rabenseifner-Verfahren
unten: Pipeline-Verfahren mit PIO und DMA

mus bezeichnet wird.

Die Präfix-Reduktion läßt sich wiederum effizient mit Hilfe des Pipeline-Verfahrens implementieren. Gegenüber dem *rpipe*-Protokoll zur Reduktion ist nur eine Erweiterung erforderlich: Jeder Ausgangsblock muß nicht nur an den nächsten Prozeß weitergeleitet werden, sondern zusätzlich noch in den lokalen Empfangspuffer des Prozesses kopiert werden. Zur Umsetzung dieser Modifikation reicht ein zusätzlicher Parameter im *rpipe*-Protokoll aus. Da die zu kopierenden Daten nach der Verknüpfungsoperation bereits im Cache stehen, fällt diese zusätzliche Kopieroperation wenig ins Gewicht.

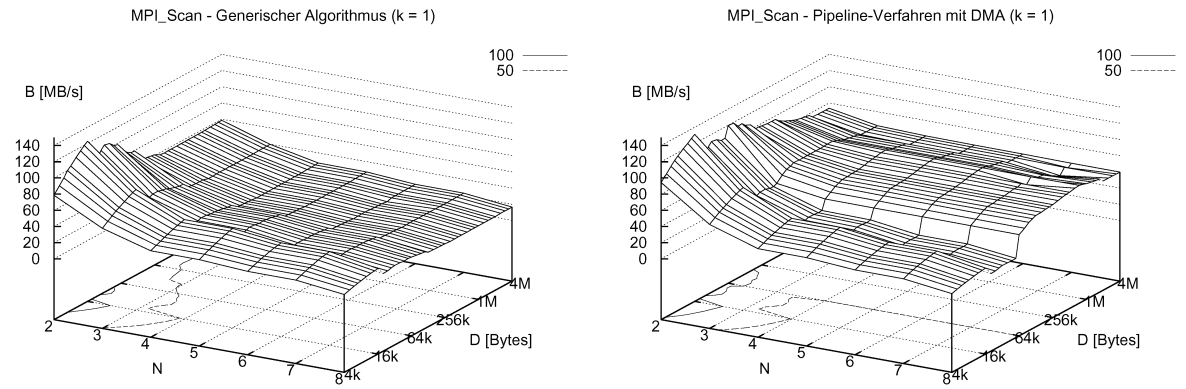


Abbildung 6.28: Bandbreite pro Prozeß bei der Präfixreduktion
links: Generischer MPICH-Algorithmus *rechts: rpipe*-Protokoll mit DMA-Unterstützung

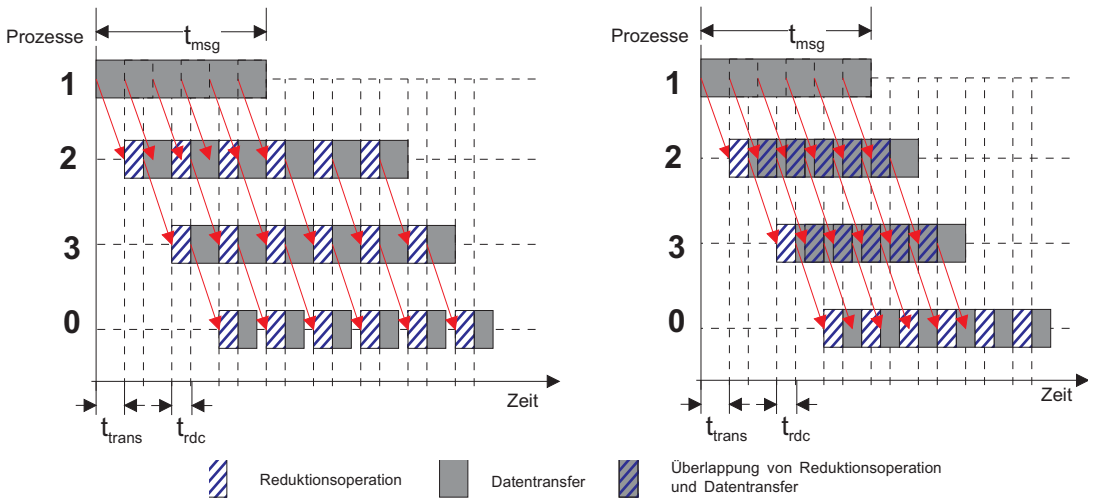


Abbildung 6.29: Ablauf einer Reduktion mit 4 Prozessen (Wurzelprozess 0) via *rpipe*-Protokoll.
Links: Reiner PIO-Datentransfer, keine Überlappung von Reduktion und Datentransfer
Rechts: DMA-Datentransfer ermöglicht Überlappung von Reduktion und Datentransfer

Die Ergebnisse dieser Optimierung sind in Abb. 6.28 dargestellt. Die erzielten Bandbreiten weisen das von der Reduktion bekannte Verhalten auf. Das generische Verfahren zu `MPI_Scan` fällt mit unter 20 MB/s für 8 Prozesse und 4 MB Vektorlänge um 20% hinter das generische Reduktionsverfahren zurück. Auch die Leistung für Präfixreduktion via DMA-unterstütztem *rpipe*-Protokoll liegt aufgrund der zusätzlichen Kopieroperation um 25% unter dem vergleichbaren Verfahren zur Reduktion, erbringt mit 60 MB/s jedoch die dreifache Leistung des generischen Verfahrens.

6.7.6 Modellierung des *rpipe*-Protokolls

Das Modell des *rpipe*-Protokolls hat aufgrund des gemeinsamen Pipelineprinzips starke Ähnlichkeit mit dem des *pcast*-Protokolls. Gegenüber dem *pcast*-Protokoll hat das *rpipe*-Protokoll jedoch pro Prozeß neben den Stufen für eingehende und ausgehende Daten eine weitere Stufe, in der die Daten mit dem entsprechenden Teil des lokalen Vektors verknüpft werden (und bei der Präfixreduktion zusätzlich in den lokalen Zielpuffer kopiert werden).

Dies führt dazu, daß die Füllatenz der Pipeline (die Zeit, bis der Wurzelprozeß den ersten Datenblock zu empfangen beginnt) höher ist als beim Rundsenden (siehe auch Abb. 6.29). Nachdem die Pipeline gefüllt ist, hat die Art des Datentransfers entscheidenden Einfluß auf die erzielbare Bandbreite: PIO-Datentransfers führen dazu, daß die Reduktionsoperationen und der Datentransfer eines Blocks sequentialisiert werden, nur der Empfang des nächsten Blocks verläuft (je nach Zustand des zugehörigen Sendeprozesses) parallel. Datentransfers via DMA hingegen erlauben es, bei der Übertragung von Block n bereits den Block $n+1$ zu reduzieren. Falls die Reduktion eines Blocks kürzer dauert als seine Übertragung, wird damit bei gefüllter Pipeline die gleiche Bandbreite wie beim Rundsenden erreicht.

Die in Abb. 6.29 dargestellten Größen sind t_{msg} als reine Übertragungsdauer des zu reduzierenden Vektors, t_{trans} als Übertragungsdauer für einen Pipeline-Block der Größe D_{block} und t_{rdc} als Dauer der Reduktion eines Pipeline-Blocks. Die Übertragungsdauer bestimmt sich zu

$$(6.22)\quad t_{trans} = \frac{D_{block}}{B_{cr}(D_{block})}$$

Die Reduktionsdauer ist abhängig vom Datentyp und dem Typ der Reduktionsoperation und kann nicht allgemeingültig angegeben werden. Die Reduktion eines Vektors der Länge D mit N

Prozessen hat somit für PIO-Datentransfers eine Latenz $t_{rdc_{rpipePIO}}$ von

$$(6.23)\ t_{rdc_{rpipePIO}} = t_{trans} + \left(N + \frac{D}{D_{block}} - 3\right) \cdot (t_{rdc} + t_{trans} + l_{sb} + l_{rw}) + \left(t_{rdc} + \frac{D_{block}}{B_{cl}(D_{block})}\right)$$

Der erste Summand ist der initiale Datentransfer. Im Anschluß müssen die $N-2$ Prozesse, die Empfänger *und* Sender sind, zum Füllen der Pipeline jeweils einen Block empfangen, reduzieren und weitersenden. Danach, bei gefüllter Pipeline, erfolgt dies weitere $D/D_{block}-1$ Mal. Am Ende muß schließlich der Wurzelprozeß den letzten Block reduzieren und lokal kopieren.

Für den Fall, daß die Datenübertragung mittels DMA-Transfers abgewickelt wird, ist nach dem Füllen der Pipeline aufgrund der Überlappung nur noch das Maximum von t_{trans} und t_{rdc} relevant:

$$\begin{aligned}(6.24)\ t_{rdc_{rpipeDMA}} = {} & t_{trans} + (N-2)(t_{rdc} + t_{trans} + l_{sb} + l_{rw}) \\ & + \left(\frac{D}{D_{block}} - 1\right)(max(t_{rdc}, t_{trans}) + l_{sb} + l_{rw}) + t_{rdc} + \frac{D_{block}}{B_{cl}(D_{block})}\end{aligned}$$

Die Ergebnisse der Simulation dieses Modells sind qualitativ den Ergebnissen der Simulation des *pcast*-Protokolls sehr ähnlich. Die jeweils erzielte Bandbreite ist jedoch geringer, da die Füllzeit der Pipeline aufgrund der doppelten Tiefe länger ist.

6.7.6.1 Vergleich der Verfahren zur Reduktion und zur Globalreduktion

Die experimentellen Ergebnisse für die Reduktion sind in Abb. 6.23 und für die Globalreduktionen in Abb. 6.26 als effektive Bandbreite pro aktivem Prozeß dargestellt. Die Leistungseckwerte, nämlich die Bandbreite für die maximal untersuchte Zahl von Prozessen und Nachrichtengröße, sind in Tabelle 6.4 dargestellt.

Operation	Generisch	Rabenseifner	*rpipe* PIO	*rpipe* DMA
Reduktion	24,7	35,9	68,7	82,0
Globalreduktion	15,3	28,1	32,8	40,8

Tabelle 6.4: Leistungs-Eckwerte (MB/s/Prozeß) der untersuchten Verfahren zur Reduktion und zur Globalreduktion (effektive Bandbreite für 8 Prozesse, 4MB Vektorlänge)

Die generische Implementation mit Baumtopologie ist in allen Fällen deutlich das leistungsschwächste Verfahren. Die Ursache dafür ist die nicht optimale Parallelisierung der Verknüpfungsoperation und die Notwendigkeit, jeweils vollständige Nachrichten sequentiell zwischen immer weniger aktiven Prozessen zu übertragen.

Das Rabenseifner-Verfahren erhöht demgegenüber durch die optimale Verteilung der Verknüpfungsarbeit (zum Preis eines höheren, dafür gleichmäßig verteilten Kommunikationsvolumens) insbesondere für 2-exponentielle Prozeßzahlen den Durchsatz. Während es jedoch etwa bei der Globalreduktion für 8 Prozesse maximal 28 MB/s pro Prozeß erreicht, sinkt dieser Wert bei bezüglich der Kommunkation ungünstigen Prozeßzahlen (wie etwa 5, 6 oder 7 Prozesse) um 25% auf 21 MB/s ab.

Das *rpipe*-Protokoll erreicht für alle Prozeßzahlen die höchsten Werte. Besonders deutlich ist dies bei der Reduktion und beim Einsatz der DMA-Übertragung. Dabei wird das 2,3-fache der Bandbreite des Rabenseifner-Verfahrens erreicht. Bei der Globalreduktion ist die Bandbreite des *rpipe*-Protokolls nur um das 1,45-fache höher, da die beiden getrennten Pipelines die doppelte Füllzeit bewirken.

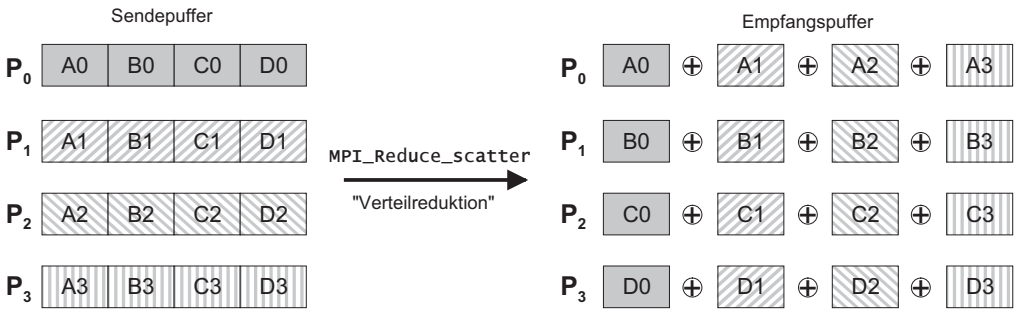


Abbildung 6.30: Datenverteilung bei der Verteilungsreduktion (`MPI_Reduce_scatter`)

Sowohl beim Rabenseifner-Verfahren als auch beim *rpipe*-Protokoll sind die Verknüpfungsoperationen optimal auf die Prozesse verteilt. Bezüglich des Verknüpfungs- und Kommunikationsaufwandes sind Rabenseifner- und *rpipe*-Verfahren gleich. Die Ursache für den Leistungsvorsprung des *rpipe*-Protokolls sind also an anderer Stelle zu suchen. Neben dem unbalancierten Kommunikationsmuster beim binären Austausch mit nicht-2-exponentiellen Prozeßzahlen spielen zwei weitere Gründe eine Rolle für die bessere Leistung des *rpipe*-Protokolls:

1. Überlappung von Kommunikation und Berechnung
   Das *rpipe*-Protokoll mit DMA-Unterstützung kann gleichzeitig verschiedene Teilvektoren kommunizieren und miteinander verknüpfen. Dies ist mit dem Rabenseifner-Verfahren selbst bei Nachrichtenversand mittels DMA nicht möglich, da die Verknüpfung und Kommunikation sequentiell ausgeführt werden müssen.
2. Segmentbelastung
   In der untersuchten Konfiguration muß ein SCI-Linksegment beim Rabenseifner-Verfahren bis zu 4 Datenströme transportieren. Beim *rpipe*-Protokoll transportiert jedes SCI-Linksegment zu jedem Zeitpunkt maximal *einen* Datenstrom, so daß es nicht zur Segmentsättigung kommen kann (siehe auch Abb. 6.4).

6.7.7 Verteilungsreduktion

Bei der Verteilungsreduktion (MPI-Funktion `MPI_Reduce_scatter`) hat im Gegensatz zur Globalreduktion nicht jeder Prozeß den gleichen, vollständigen Vektor als Ergebnis der Verknüpfung der Vektoren aller beteiligten Prozesse im Empfangspuffer, sondern nur einen bestimmten Teil dieses Vektors. Die Darstellung der entsprechenden Datenverteilung in Abb. 6.30 zeigt eine starke Ähnlichkeit mit der Datenverteilung bei der Alle-an-alle-Kommunikation.

Der Unterschied liegt darin, daß der bei den N Prozessen am Ende vorliegende Vektor im Mittel nur $1/N$ der Länge des Ausgangsvektors hat, da die N Teilvektoren miteinander verknüpft werden. Zur Implementierung bieten sich mehrere Verfahren an:

- *Reduktion-Verteilung:* Es wird eine gewöhnliche Reduktion des gesamten Vektors auf einen beliebigen Wurzelprozeß durchgeführt, der anschließend die Teilvektoren an alle anderen Prozesse verteilt. Dies ist die generische Implementation in MPICH. Die Komplexität dieses Verfahrens ist $O(D \cdot ld(N) + N \cdot (D/N))$. Der zweite Summand in diesem Term berücksichtigt, daß das Versenden von N Nachrichten der Gesamtlänge D unterschieden werden muß vom Versand einer einzelnen Nachricht der Länge D.
- *Alle-an-alle:* Die Prozesse führen eine Alle-an-Alle-Kommunikation durch und verknüpfen die empfangenen Teilvektoren anschließend lokal. Dies erfordert bei angenommener Gleichverteilung der Längen der Teilvektoren N Nachrichten der Länge D/N, die von allen Prozessen parallel versandt werden, was zu einer Komplexität $O(N \cdot (D/N))$ führt.

- *Pipeline-Verteilung:* Die Prozesse führen zunächst eine Pipeline-Reduktion aus; anschließend versendet der Wurzelprozess sequentiell die einzelnen Teilvektoren. Dies entspricht nominell dem Kommunikationsaufwand der *Alle-an-Alle*-Variante zuzüglich der Pipeline-Reduktion mit einem zu *D* proportionalen Aufwand.
- *Multi-Pipelines*: Jeder der *N* Prozesse ist Wurzelprozeß einer von *N* parallel ablaufenden Reduktionspipeline für die *N* Teilvektoren. Hier fällt nominell eine Gesamtoperation der Komplexität $O(D)$ an.

Wie bereits jedoch bei der einfachen Reduktion (Kapitel 6.7.2) dargelegt wurde, ist für kleine Vektoren eine Pipelineverarbeitung (und mithin auch eine Multi-Pipelineverarbeitung) nicht sinnvoll. Daher wird in diesem Falle das *Alle-an-Alle*-Verfahren die niedrigste Latenz und somit die beste Leistung erbringen. Gegenüber der *Alle-an-Alle*-Kommunikation kommt in diesem Fall die Verknüpfungsoperation hinzu, die zwischen dem lokalen Teil des Vektors und den empfangenen Vektorteilen durchgeführt werden muß. Der direkte Ansatz für die Umsetzung des Verfahrens wäre, zunächst die eigenen *N* Empfangsbereitschaften herzustellen, anschließend die *N* Teilvektoren zu verschicken, um schließlich die $N-1$ Verknüpfungsoperationen durchzuführen. Dies hätte jedoch zur Folge, daß die gesamte Kommunikation und die gesamte Berechnung (zur Durchführung der Verknüpfungen) nacheinander von allen Prozessen zugleich durchgeführt wird. Die Überlappung dieser Vorgänge zwischen den Prozessen würde jedoch zu einer Verringerung der Netzbelastung führen: Während ein Teil der Prozesse kommuniziert, führt ein anderer Teil die Verknüpfungen durch.

Dazu wurde ein spezielles Kommunikationsmuster entwickelt, das auf der Aufteilung der Prozesse in die Gruppen der Prozesse mit geradem und ungeradem Rang basiert. Diese wickeln ihre Kommunikation, bestehend aus dem Versand von Teilvektoren und den entsprechenden Reduktionsoperationen, nach einer jeweils unterschiedlichen Vorschrift ab. Beide Vorschriften haben gemeinsam, daß zunächst der Austausch mit einem unmittelbaren Nachbar stattfindet und anschließend zunächst mit der jeweils anderen Gruppe von Prozessen kommuniziert wird. Im Anschluß daran findet der Austausch mit den Prozessen aus der eigenen Gruppe statt. Dies ist in Abb. 6.31 für eine Verteilungsreduktion mit $N = 8$ Prozessen dargestellt und wird im folgenden für die beiden Gruppen im einzelnen beschrieben.

Gerade Prozesse: Ein Prozeß mit Rang p_e ($p_e \lozenge 2 = 0$) sendet zunächst an Prozeß $(p_e + 1) \lozenge N$, dann an sich selbst. Darauf folgen $N/2 - 1$ Iterationen, in denen jeweils ein Teilvektor an den nächsten Prozeß mit ungeradem Rang gesendet wird. Anschließend wird von dessen ungeradem Vorgänger ein Teilvektor empfangen, der mit dem lokalen Teilvektor reduziert wird. Danach erfolgt in weiteren $N/2 - 1$ Iterationen der Versand an die Prozesse aus der eigenen Gruppe in absteigender Reihenfolge (ausgehend vom eigenen Rang) und die Reduktion der empfangenen Teilvektoren in aufsteigender Reihenfolge.

Ungerade Prozesse: Die ungeraden Prozesse p_o ($p_o \lozenge 2 \neq 0$) schicken zunächst einen Teilvektor an sich selbst. Anschließend werden ebenfalls in $N/2 - 1$ Iterationen die Teilvektoren von den geraden Prozessen in absteigender relativer Reihenfolge empfangen und verknüpft. Weitere $N/2 - 1$ Iterationen senden an die ungeraden Prozesse die Teilvektoren in absteigender Reihenfolge und reduzieren die von diesen empfangenen Teilvektoren in aufsteigender Reihenfolge.

In Abb. 6.31 ist dieser Algorithmus graphisch dargestellt. Für die Prozesse 0 und 7 wurden die abzuarbeitenden Kommunikationsvorgänge hervorgehoben. Die durchgezogenen Pfeile verbinden den Zeitpunkt des Versands von Teilvektoren mit dem Zeitpunkt, an dem diese von dem Empfängerprozeß benötigt werden. Umgekehrt geben die gestrichelten Pfeile die Abhängigkeit einer Reduktionsoperation von dem Versand eines Teilvektors durch einen entfernten Prozeß an. Kennzeichnend für die Effizienz des Verfahrens ist dabei der Verlauf der beschriebenen Ab-

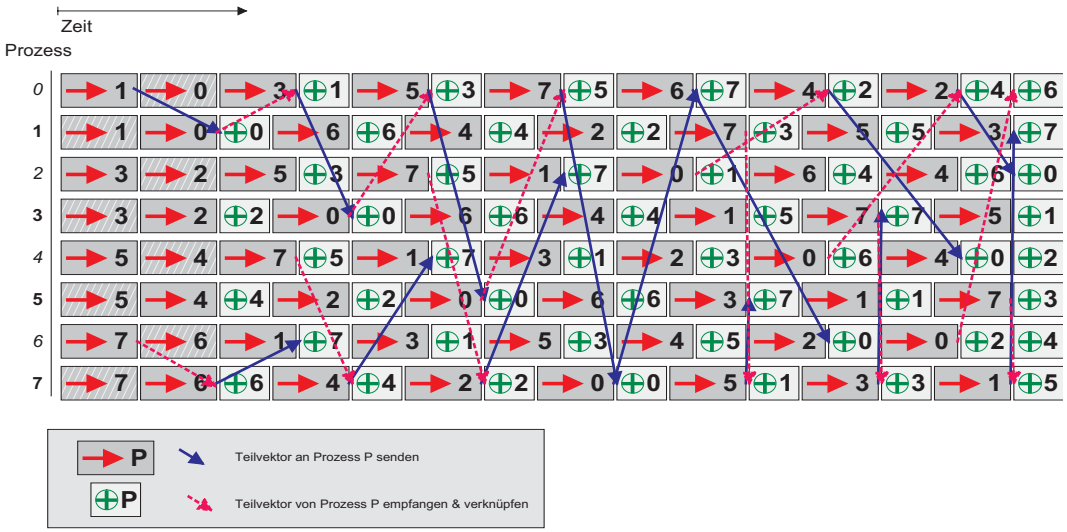


Abbildung 6.31: Globale Überlappung von Kommunikation und Berechnung bei Verteilungsreduktion mit kleinen Vektoren. Für jeweils einen geraden Prozeß (hier: 0) und einen ungeraden Prozeß (hier: 7) sind die Kommunikationsvorgänge und -abhängigkeiten mit allen anderen Prozessen dargestellt.

hängigkeiten in Richtung der Zeitachse. Ein Verlauf in dieser Richtung bedeutet, das nicht auf das Eintreffen von Daten gewartet werden muß. Dies trifft für alle Prozesse zu, da alle das gleiche Kommunikationsmuster, bezogen auf ihren Prozeßrang, abwickeln. Ein Verlauf gegen die Richtung der Zeitachse würde bedeuten, daß der Zeitpunkt, zu dem die Daten beim Empfänger benötigt werden, *vor* dem Zeitpunkt des Versands liegt. Dies hätte Wartezeiten beim Empfänger zur Folge.

Dieses Verfahren ist besonders geeignet für das *eager*-Protokoll, das es dem Sender erlaubt, den Teilvektor ohne Mitwirkung des Empfänger bei diesem zu hinterlegen. Die obere Grenze der Vektorlänge D für die Verwendung des *eager*-Protokolls liegt hierbei nicht bei S_{eager} (der Größe der Eingangspuffer im *eager*-Protokoll), sondern bei $N \cdot S_{eager}$, da jeder Prozeß nur diese Teilvektoren versendet. Die maximale Länge eines Vektors, für den dieses Verfahren verwendet werden kann, wächst daher proportional mit der Zahl der Prozesse, die an der Verteilungsreduktion teilnehmen.

Der experimentelle Vergleich dieses neuartigen Reduktionsverfahren mit dem generischen MPICH-Verfahren in Abb. 6.32 zeigt, daß die vorgestellten Überlegungen tatsächlich einen starken leistungsfördernden Effekt haben. Zunächst ist qualitativ ein Unterschied im Skalierungsverhalten offenbar; etwa die Steigerung der Latenz bei einer Vektorlänge von 2^{16} Byte. Beim generischen Verfahren ist diese Steigerung für den Übergang von 4 auf 8 aktive Prozesse mit einer Zunahme von $727\mu s$ deutlich stärker als beim überlappenden Verfahren mit einer Zunahme von $371\mu s$[1]. Dieser quantitative Leistungsunterschied wird auch in den absoluten Zahlen deutlich. Für den gesamten wirksamen Bereich des überlappenden Verfahrens wird eine geringere Latenz erzielt. Der Unterschied fällt um so stärker aus, je länger die Vektoren sind, da mit der Länge der Vektoren auch die Effekte der Segmentsättigung und der Überlastung einzelner PSA zunehmen. Da das überlappende Protokoll diese Effekte reduziert, wächst der relative Leistungsgewinn bei 8 Prozessen vom Faktor 1,29 für einen Vektor von 256 Byte (entsprechend 4 `double`-Werten pro Prozeß) auf 2,95 für einen Vektor von 2^{16} Byte.

1. Die Prozeßzahlen von 2 und 3 stellen im überlappenden Verfahren Sonderfälle dar und können daher nicht als Vergleichswerte für den zugrundeliegenden Algorithmus verwendet werden.

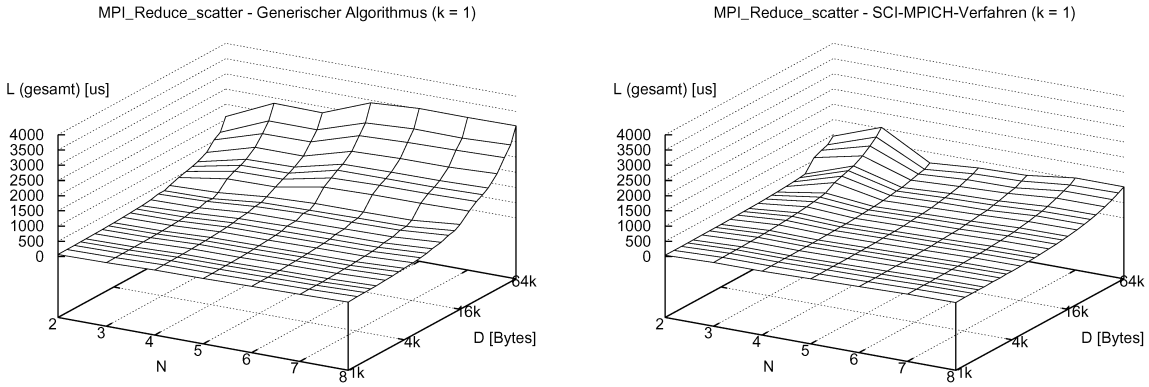


Abbildung 6.32: Vergleich der Latenz der Verfahren zur Verteilungsreduktion
links: Generischer MPICH Algorithmus
rechts: Überlappende Verteilungsreduktion in SCI-MPICH

6.8 Alle-sammeln-Kommunikation

Die Alle-sammeln-Kommunikation (Funktion `MPI_Allgather`()) führt Daten der Größe D Bytes von N Prozessen auf allen N Prozessen zusammen, so daß anschließend jeder Prozeß die Daten aller anderen Prozesse vorliegen hat. Die Variante `MPI_Allgatherv`() erlaubt jedem Prozeß die individuelle Spezifikation der Datenmenge und der Plazierung der Daten im Empfangspuffer. Die Plazierung spielt für die Optimierung jedoch keine Rolle, und da die Dauer für die kollektive Operation im wesentlichen von der Dauer für den Transfer des größten Datenblocks abhängt, stellt eine Beschränkung auf die Funktion `MPI_Allgather`() keine Einschränkung der Allgemeingültigkeit des vorgestellten Verfahrens dar. Ein Beispiel für die Datenverteilung durch eine Alle-sammeln-Operation ist in Abb. 6.33 dargestellt. Im Gegensatz zur Alle-an-alle-Kommunikation hat jeder Prozeß p_i dieselben Daten von einem bestimmten Prozeß p_j, was bestimmte Optimierungen ermöglicht. Das Kommunikationsvolumen beträgt also insgesamt $N^2 \cdot D$ Bytes, jedoch werden nur $N \cdot D$ Bytes unterschiedlicher Daten bewegt.

Abbildung 6.33: Datenverteilung bei *Alle-sammeln*-Kommunikation (`MPI_Allgather`)

Der naive Algorithmus startet zunächst N nicht-blockierende Empfangsvorgänge für die Daten von den N Prozessen, anschließend werden N nicht-blockierende Sendeoperationen gestartet, um schließlich auf den Abschluß all dieser Operationen zu warten. Dieses Verfahren mit unkoordinierten Transfers kann bei kleinen Datenmengen (bis zu wenigen kB pro Prozeß) gute Leistungen bringen; bei größeren Datenmengen kommt es jedoch zu Segmentsättigung und damit reduzierter effektiver Bandbreite. Dieses Problem verstärkt sich, wenn in einem SMP-Knoten mehrere Prozesse über einen PSA kommunizieren. Hinzu kommt die erforderliche volle Kon-

nektivität bei dem Datenaustausch zwischen *allen* Prozessen.

In *[119]* wird ein zyklisches Verfahren vorgestellt (das auch in MPICH verwendet wird), bei dem abwechselnd mit den benachbarten Prozessen Daten durch Senden und Empfangen ausgetauscht werden. Dadurch werden die Übertragungen koordiniert und es wird keine volle Konnektivität zwischen allen Prozessen benötigt. Für SMP-Knoten mit mehr als einem Prozeß werden alle lokalen Daten bei einem Prozeß auf dem Knoten gesammelt, von diesem übertragen und anschließend die empfangenen Daten an alle lokalen Prozesse verteilt, um die zeitliche Mehrfachnutzung eines PSA durch mehrere Prozesse zu vermeiden. Hierbei tritt jedoch das Problem auf, daß die Daten zwischen den Prozessen auf einem Knoten vollständig nach der Inter-Knoten-Kommunikation übertragen werden. Zudem werden bei jeder Sende-/Empfangsoperation nur Daten *eines* Prozesses übertragen.

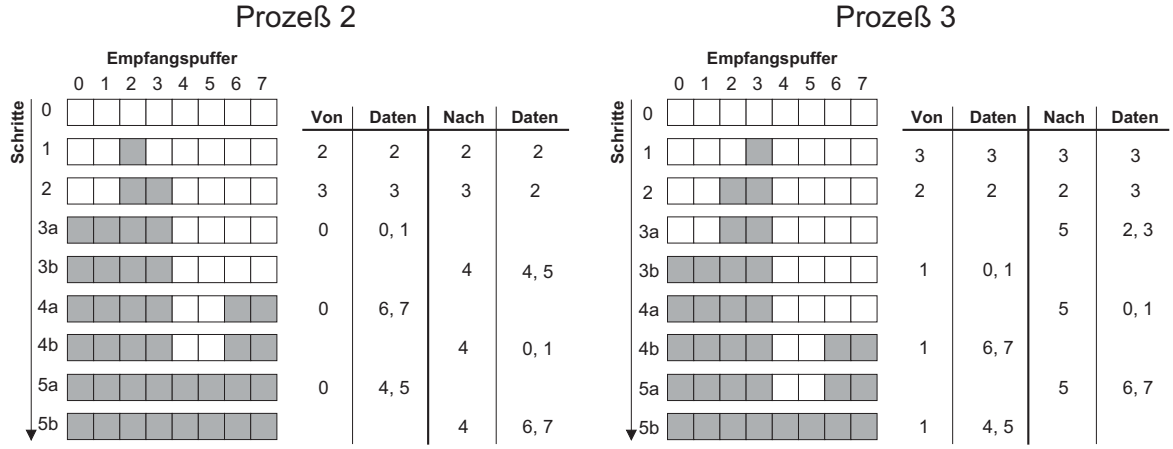


Abbildung 6.34: Unidirektionales Schiebe-Verfahren zur Alle-Sammeln-Kommunikation für 8 Prozesse auf 8 Knoten. Dargestellt sind für die Prozesse 2 und 3 die Zustände der Empfangspuffer und die Sende- und Empfangsoperationen.

6.8.1 Unidirektionales Schiebe-Verfahren

Diese Nachteile vermeidet das für SCI-MPICH entwickelte unidirektionale Schiebe-Verfahren (*unidirectional shift*), daß sich bei geraden Prozeßzahlen besonders vorteilhaft einsetzen läßt. Die Daten werden, bis auf einen Austausch, stets nur in die Vorzugsrichtung des SCI-Rings übertragen. Bei dem Verfahren aus *[119]* ergibt sich im SMP-Betrieb der Effekt, daß sich die Größe der Nachrichten, mit denen Daten versandt werden, proportional mit der Zahl der Prozesse pro Knoten vergrößert und die Nachrichten dadurch mit höherer Bandbreite übertragen werden. Dieser Effekt ergibt sich beim vorgestellten neuen Verfahren bereits im UP-Betrieb mit einer Verdoppelung der Nachrichtengröße. Das Verfahren wird für den UP-Betrieb an einem Beispiel für 8 Prozesse auf 8 Knoten anhand der benachbarten Prozesse 2 und 3 in Abb. 6.34 erläutert. Die graphische und die tabellarische Darstellung der Kommunikationsoperationen zeigen die sukzessiven Füllung des Empfangspuffers in jedem Schritt. Zunächst sendet sich jeder Prozeß die eigenen Daten (Schritt 1 in der Abbildung); anschließend tauscht jeder Prozeß mit Rang r, $r \lozenge 2 = 0$, seine Daten mit Prozeß $r' = (r + 1) \lozenge N$ aus (Schritt 2). Nach dieser Initialisierung folgt die unidirektionale Verschiebung von Daten: in der Abbildung sind dies die Schritte 3a bis 5b. Jeder Prozeß r' schickt abwechselnd die beiden Datenblöcke, die er zuletzt empfangen hat, an den Prozeß $(r' + 1) \lozenge N$ und empfängt von Prozeß r die nächsten beiden Datenblöcke in jeweils einer Sende- und Empfangsoperation, die parallel abgewickelt werden können. Für gerade Prozeßzahlen, für die das Verfahren optimiert ist, ist die Kommunikation nach $1 + N/2$ Schritten abgeschlossen. Bei ungerader Prozeßzahl bleibt der Prozeß $N - 1$ untätig, bis er zum Abschluß von Prozeß $N - 2$ die gesamten benötigten Daten in einem Transfer zuge-

stellt bekommt.

6.8.2 Evaluierung des unidirektionalen Schiebe-Verfahrens

Die Ergebnisse eines Tests mit 8 Prozessen sind in Abb. 6.35 dargestellt. Die verringerte Zahl von Sende- und Empfangsoperationen macht sich bei kleinen Nachrichten in einer deutlichen Verringerung der Latenz bemerkbar, die beim unidirektionalen Schiebe-Verfahren um bis zu 43% gegenüber dem generischen Schiebe-Verfahren zurückgeht. Auch bei großen Datenmengen im Bereich des *rendez-vous* Protokolls liegt der Durchsatz des unidirektionalen Schiebe-Verfahrens um etwa 7% über dem generischen Schiebe-Verfahren, was auf die Erhöhung der Kommunikationseffizienz durch Verwendung größerer Datenblöcke und verringerte Segmentsättigungseffekte zurückzuführen ist.

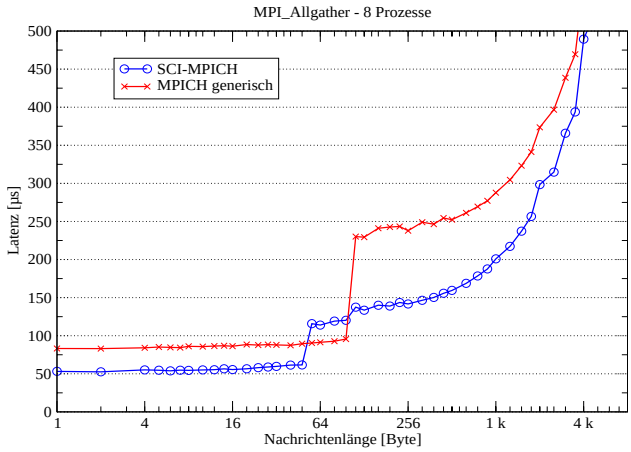


Abbildung 6.35: Vergleich der Latenz von generischem Schiebe-Verfahren und unidirektionalem Schiebe-Verfahren mit Nachrichtenvergrößerung (8 Prozesse auf 8 Knoten)

6.9 Verteilen

Beim *Verteilen* sendet der Wurzelprozeß Teile eines Vektors an alle anderen Prozesse, wobei jeder Prozeß einen disjunkten Teil des Vektor erhält (siehe Abb. 6.36). Bei der G*leichverteilung* (via `MPI_Scatter`) erhält jeder Prozeß einen gleichgroßen Anteil des Vektors. Die *Indexverteilung* (via `MPI_Scatterv`) hingegen erlaubt die Prozeß-individuelle Spezifikation der Position und Länge der an jeden Prozeß zu übermittelnden Daten. Somit beträgt die zu kommunizierende Datenmenge bei einem Vektor der Datenmenge *D* ebenfalls *D*.

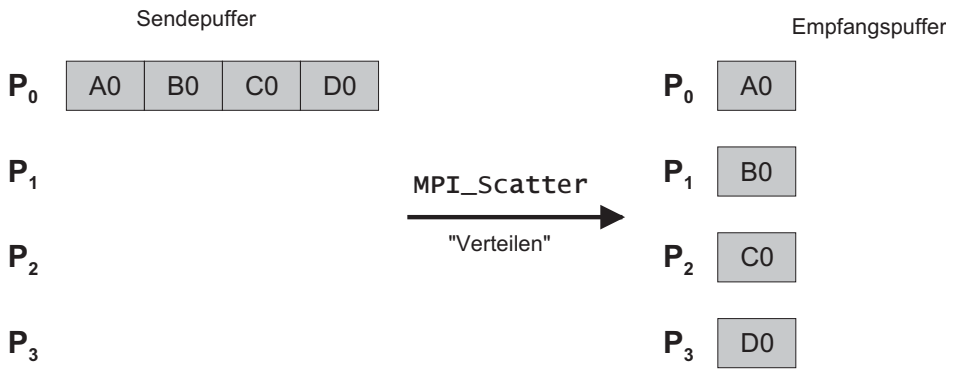


Abbildung 6.36: Verteilen eines Vektors (`MPI_Scatter`)

6.9.1 Gleichverteilung

Der naive Algorithmus der Gleichverteilung besteht aus einem sequentiellen Baum mit N sequentiellen Sendevorgängen an die N Prozesse. Dieses Verfahren hat entsprechend eine Laufzeit $O(D \cdot N)$. Eine Optimierung dieses Vorgangs scheint zunächst schwer möglich, da sich die Menge der zu kommunizierenden Daten, die vom Wurzelprozeß versandt werden muß, nicht reduzieren läßt und beim Versand auch keine Konflikte im Netz oder bei den Empfangsprozessen zu erwarten sind, die sich durch eine andere Koordination der Vorgänge vermeiden ließen.

Wenn man jedoch den Verlauf der Latenz von kleinen Nachrichten betrachtet (siehe Abb. 4.6), entsteht durchaus eine Optimierungsmöglichkeit: Die Latenz der Nachrichten steigt sublinear mit der Datenmenge an. So sind größere Nachrichten effizienter als kleinere Nachrichten. Anstelle der sequentiellen Sendevorgänge kann ein Binomialbaum als Kommunikationsmuster verwendet werden. Dabei sendet der Wurzelprozeß einen Teilvektor an einen Zielprozeß, der die Daten für diesen Prozeß sowie für eine Zahl weiterer Prozesse enthält, die anschließend die Daten von diesem Zielprozeß erhalten werden. Die insgesamt kommunizierte Datenmenge erhöht sich durch dieses Verfahren. Jedoch muß der Wurzelprozeß anstelle von N nur noch $ld(N)$ Nachrichten verschicken, und durch die höhere Effizienz der versendeten Nachrichten verringert sich die benötigte Zeit im Bereich kleiner Nachrichten. Für den Bereich großer Nachrichten, wenn der Versand einer Nachricht der Länge D/N die gleiche Bandbreite erzielt wie alle anderen Nachrichten mit einer Länge zwischen D/N und D, ist dieses Verfahren aufgrund der redundanten Kommunikation nicht geeignet. In diesem Fall verhält sich die Laufzeit für *alle* Prozesse so wie im schlechtesten Fall des naiven Verfahrens, da *jeder* Prozeß seine Daten erst nach dem Versand von $\frac{N-1}{N} \cdot D$ Daten seinen Teil des Vektors erhält.

Die Ergebnisse dieser Optimierung sind in Abb. 6.37 dargestellt. Daraus wird deutlich, daß die

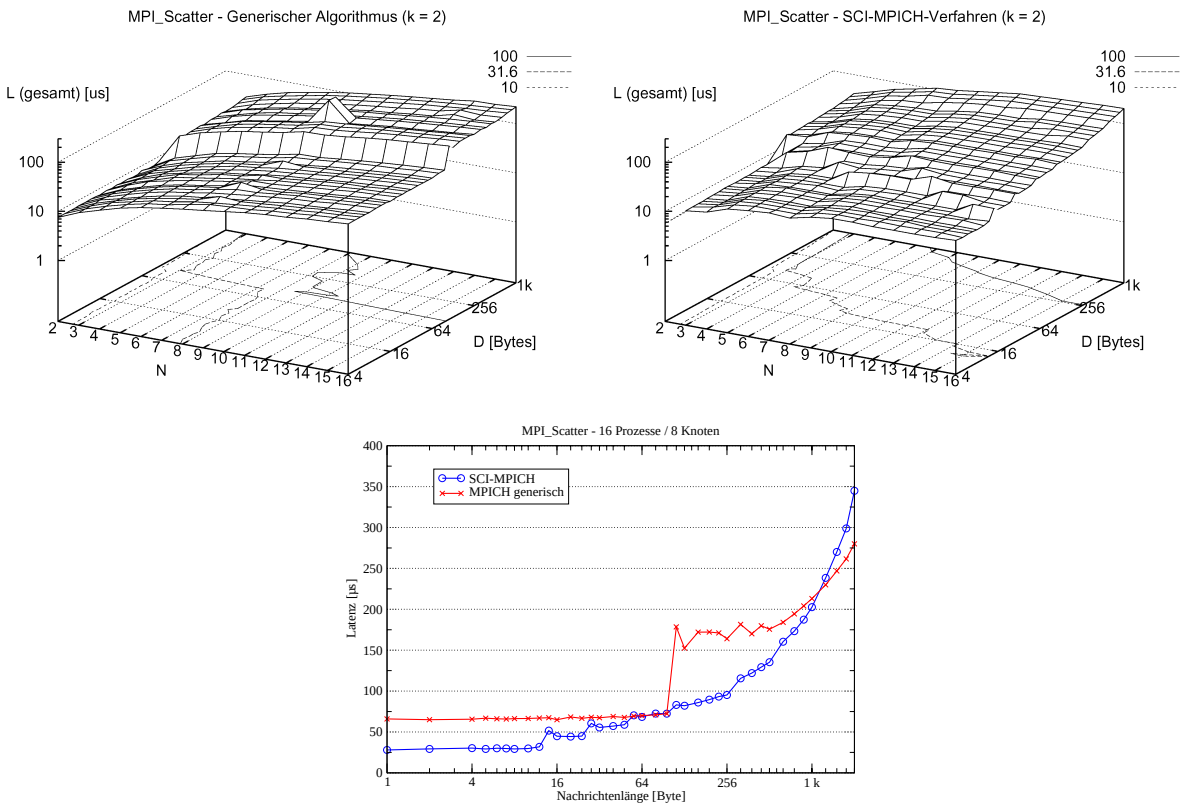


Abbildung 6.37: Vergleich generischen MPICH-Algorithmus und optimierten SCI-MPICH-Algorithmus für `MPI_Scatter`
Oben: Unterschiedliches Skalierungsverhalten
Unten: Latenzvergleich für kleine Nachrichten

Leistungssteigerung relativ zum generischen Algorithmus mit der Zahl der Prozesse *N* skaliert, wie dies die Zahl der notwendigen Nachrichten *O(N)* gegenüber *O(ld N)* erwarten läßt. Ebenso bestätigt sich, daß das Verfahren nur für kleine Nachrichten effizient ist, bei denen die Latenz einer relativ längeren Nachricht nur sublinear mit der Längendifferenz skaliert.

6.9.2 Indexverteilung

Für die Optimierung der Indexverteilung (via `MPI_Scatterv`) kann das vorgestellte Verfahren nicht verwendet werden, da nur dem Wurzelprozeß die gesamte (potentiell ungleichmäßige) Verteilung des Vektors auf die Prozesse bekannt ist. Alle anderen Prozesse kennen nur die Menge der Daten, die sie selber empfangen. Ohne jedoch zu wissen, welcher Prozeß welchen Teil des Vektors bekommt, kann ein Prozeß einen empfangenen Teilvektor nicht weiterverteilen, wie es bei der Gleichverteilung möglich ist.

6.10 Andere Arbeiten

Zur Implementation von kollektiven Operationen gibt es eine große Zahl von Arbeiten allgemeiner Art, die unter Berücksichtigung der gegebenen Netztopologie und -technologie, Kommunikationsprotokolle, Übertragungsverfahren und anderer Randbedingungen optimierte oder optimale Algorithmen vorstellen *[123,124,125,126,127,128,129,130]*. Aufgrund der darin gemachten Annahmen, wie etwa "Zeitaufwand für lokales Kopieren von Daten ist vernachlässigbar" oder "Netzadapter kann Daten von beliebig vielen Sendern mit voller Bandbreite empfangen" sind diese kaum oder gar nicht auf eine tatsächliche Implementation in MPI zu übertragen. Da die Verfahren zumeist theoretisch oder in Simulation evaluiert werden, ist ein Vergleich nur bedingt möglich.

Andere Arbeiten beziehen sich auf generische Algorithmen und deren Implementation *[115]* und die Optimierung der Kommunikationsparameter dieser Verfahren *[120]*. Diese Verfahren waren in MPICH zum großen Teil bereits implementiert; die Optimierung der Eingangsparameter innerhalb sinnvoller Werte konnte in dem verwendeten Testmaßstab keine deutlichen Veränderungen bringen.

L.P. Huse hat jedoch in *[118,119]* Optimierungen der kollektiven Operationen in SCAMPI *[81]* beschrieben. Teilweise beruhten die darin erzielten Verbesserungen auf der Vermeidung besonders ungünstiger Betriebsmodi für die damals aktuelle Generation von PSA; es sind aber auch allgemeine Verfahren beschrieben, die auf die Nutzung von SCI optimiert und in SCAMPI implementiert sind. Es war möglich, auf einem 8-Knoten-Cluster einen direkten Vergleich der Leistung von SCAMPI und SCI-MPICH vorzunehmen, dessen wichtigste Ergebnisse im folgenden vorgestellt werden.

6.10.1 Punkt-zu-Punkt-Kommunikation

Die Leistung der Punkt-zu-Punkt-Kommunikation ist als Ping-Pong-Test und Austauschkommunikation in Abb. 6.38 dargestellt. Die schwächere Leistung von SCI-MPICH ist vorwiegend auf die fehlende Optimierung auf diese spezielle Plattform (Intel Pentium 4 mit i860 Chipsatz) zurückzuführen, auf der die in SCI-MPICH verwendeten Speicherkopieroperationen in entfernten Speicher zum Testzeitpunkt nicht die maximal mögliche Leistung erbrachten.

6.10.2 Kollektive Operationen

Die Messung der kollektiven Operationen erfolgte mit PMB bei acht Prozessen auf acht Knoten. Die Ergebnisse für `MPI_Allreduce`, `MPI_Bcast`, `MPI_Reduce`, `MPI_Allgather` und `MPI_Reduce_scatter` und sind in Abb. 6.39 dargestellt.

Obwohl bei `MPI_Allreduce` sowohl SCAMPI als auch SCI-MPICH den gleichen Algorithmus

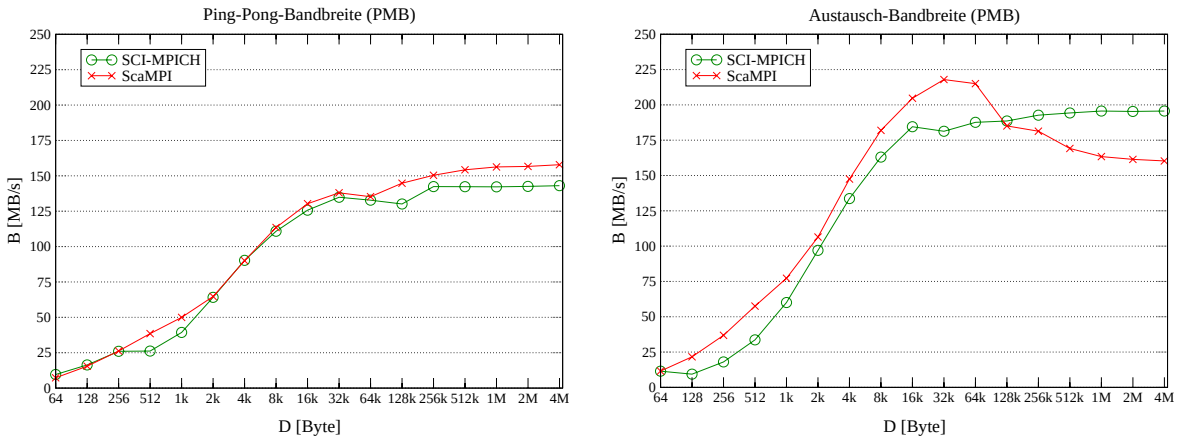


Abbildung 6.38: Vergleich zwischen ScaMPI und SCI-MPICH: Bandbreite der Ping-Pong-Kommunikation (*links*) und Austausch-Kommunkation (*rechts*) für 2 Prozesse.

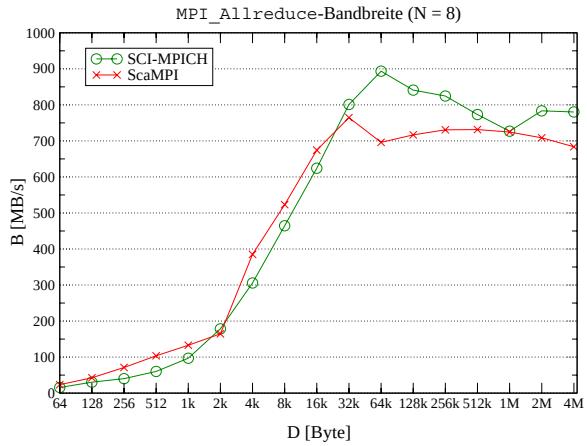
[122] verwenden und die Punkt-zu-Punkt-Bandbreite von SCI-MPICH niedriger ist, führen die in Kapitel 6.7.4.2 beschriebenen Optimierungen dazu, daß SCI-MPICH für lange Vektoren eine höhere Bandbreite erzielt. Bei `MPI_Reduce` zeigt sich klar die Wirkung der Direktreduktion bei kurzen Vektoren und der Nutzung des Pipelining bei langen Vektoren. Das Pipelining sorgt auch bei `MPI_Bcast` dafür, daß die akkumulierte Bandbreite von SCI-MPICH 50% höher liegt als der entsprechende Wert von ScaMPI.[1] Die um über 300% höhere Maximalbandbreite, die SCI-MPICH gegenüber ScaMPI bei `MPI_Reduce_scatter` erreicht, belegt die hohe Effizienz dieses neuen Verfahrens. Während ScaMPI bei `MPI_Allgather` die etwas höhere Maximalbandbreite erzielt, kann SCI-MPICH die erreichte Bandbreite auch für lange Vektoren beibehalten und liegt so schließlich oberhalb der Bandbreite von ScaMPI.

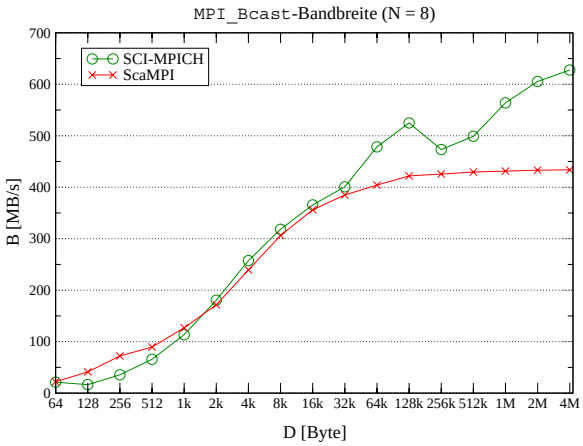
Sowohl ScaMPI als auch SCI-MPICH nutzen den direkten Zugriff auf gemeinsamen Speicher zur Barrierensynchronisation. Die Verzögerung, die dabei entsteht, ist in Tabelle 6.5 aufgeführt. Deutlich wird hier, daß sich durch den vergrößerten *fan-in* bei SCI-MPICH die Latenz beim Übergang von zwei auf vier Knoten kaum erhöht. Auch die relative Steigerung der Verzögerung beim Übergang von vier auf auf Prozesse fällt mit 27% bei SCI-MPICH besser aus als bei ScaMPI mit 53%.

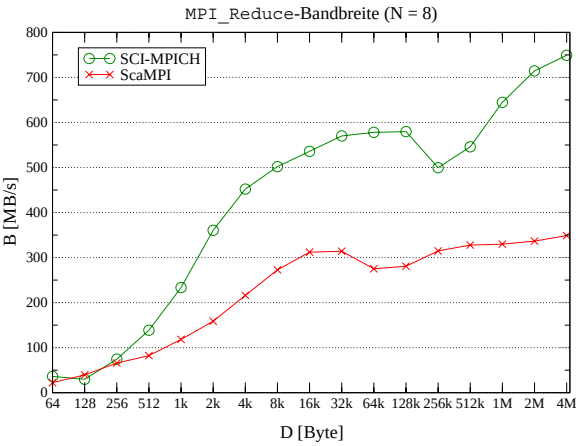
Prozesse	ScaMPI	SCI-MPICH
2	7,75µs	6,06µs
4	15,24µs	6,61µs
8	23,41µs	8,43µs

Tabelle 6.5: Barrierenverzögerung von ScaMPI und SCI-MPICH

1. Die Leistungssteigerung von SCI-MPICH wäre noch deutlicher, aber die DMA-Bandbreite auf dem Testsystem ist auf 150 MB/s beschränkt (gegenüber 245MB/s auf P3).







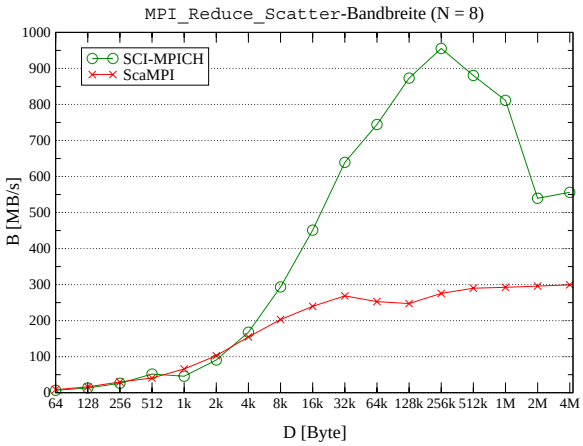
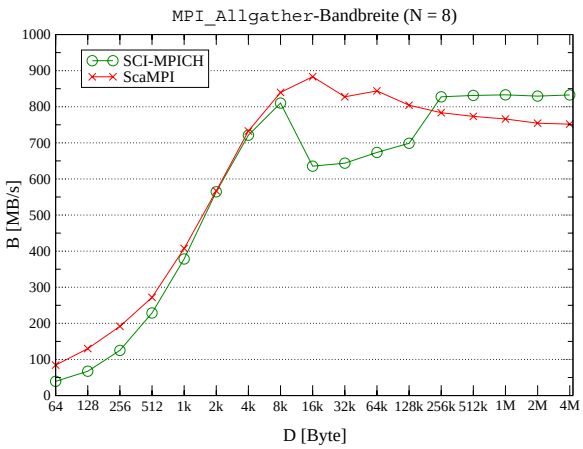


Abbildung 6.39: Vergleich ausgewählter kollektiver Operationen zwischen SCI-MPICH und ScaMPI.



KAPITEL 7

Asynchrone und direkte Datenübertragung

Die in Kapitel 4 vorgestellten Protokolle zur Nachrichtenübertragung orientieren sich in ihren Eigenschaften an den Möglichkeiten, die unmittelbare Lade- und Speicheroperationen auf (gemeinsamem) Speicher bieten. Speichergekoppelte Rechnerverbundsysteme bieten jedoch als weiteren Kommunikationsmechanismus die DMA-Übertragung, die Daten ohne Mitwirkung der CPU kopiert. Zur Synchronisation können neben unterschiedlichen Zuständen von Speicherstellen auch Unterbrechungen entfernter Prozesse dienen, was Benachrichtigungsverfahren wiederum ohne aktive Mitarbeit der entfernten CPU (in Form von Ladevorgängen auf die entsprechenden Speicherstellen) ermöglicht. Die Verwendung dieser beiden Verfahren ermöglicht die Realisation von Kommunikationsprotokollen, die durch Asynchronität und geringe CPU-Belastung andere Leistungseigenschaften als die zuvor vorgestellten Protokolle aufweisen.

7.1 Unterbrechungsgesteuerte Eingangsprüfung

Zum Erreichen einer minimalen Kommunikationslatenz wird zur Abfrage auf neu eingegangene Nachrichten (*Eingangsprüfung)* in SCI-MPICH standardmäßig ein *Polling-Verfahren* verwendet (siehe Beschreibung des *short*-Protokolls in Kapitel 4.2.2). Dabei prüft ein Prozeß fortlaufend die Speicherstellen ab, die bei einer Veränderung die Ankunft einer neuen Kontrollnachricht anzeigen. Dieses Verfahren führt dazu, daß die gesamte Leistung einer CPU für diese fortlaufende Prüfung verwendet wird. Für die typische Nutzung eines Systems durch MPI-Applikationen stellt dies kein Problem und auch keine Verschwendung von Rechenzeit dar, da die Prozesse der MPI-Applikationen im Verhältnis 1:1 auf die verfügbaren CPUs verteilt werden. Nur so erhält jeder Prozeß für seine eigentliche Aufgabe, die Durchführung von Berechnungen, die maximale CPU- und Speicherleistung.

Wenn jedoch das System neben der Bereitstellung von CPU-, Speicher- und Kommunikationsressourcen für die Prozesse der MPI-Applikationen noch andere Aufgaben zu erfüllen hat, wie es etwa bei der parallelen Nutzung von Knoten als Rechen- und E/A-Knoten der Fall ist, ist eine Blockade einer CPU zur Prüfung auf neue Nachrichten nicht erwünscht. Ein anderer Fall, bei dem die Nutzung der vollen Leistung einer CPU zur Minimierung der Latenz nicht eine optimale Gesamtleistung liefert, ist die Nutzung von mehr als einem Thread innerhalb eines MPI-Prozesses. Hier würde dieses Verfahren dazu führen, daß bei t aktiven Threads ein Thread, der auf den Eingang neuer Nachrichten wartet, den $t-1$ anderen aktiven Threads, die weiterhin Berechnungen durchführen können, mindestens $100 \cdot \left(1 - \frac{t-1}{t}\right)$ % der CPU-Zeit entzieht.

7.1.1 Eingangsprüfung durch Device-Thread

Um in derartigen Situationen eine CPU-schonende Prüfung auf eingehende Nachrichten durchzuführen, wurde in SCI-MPICH alternativ zum Polling-Verfahren zur Eingangsprüfung eine *unterbrechungsgesteuerte* Eingangsprüfung (*Unterbrechungs-Verfahren*) entwickelt. Nach dem Versand jeder Kontrollnachricht muß ein Sendeprozeß dazu beim Empfangsprozeß eine Unterbrechung auslösen. Im Kommunikations-Device *ch_smi* muß zur Verarbeitung dieser Unterbrechung ein unabhängiger Thread eingeführt werden (der *Device-Thread*, DT), der auf diese von einem entfernten Prozeß ausgelöste Unterbrechung wartet. Nach deren Auftreten prüft er die Eingangspuffer für Kontrollnachrichten und verarbeitet die vorgefundenen Kontrollnachrichten. Dabei muß mindestens eine Kontrollnachricht gefunden werden, aufgrund derer die Unterbrechung ausgelöst wurde. Weitere Kontrollnachrichten können im Verlauf der Prüfung der Eingangspuffer und der Verarbeitung eintreffen und werden ebenfalls abgearbeitet.

Die Abfrage der Eingangspuffer Applikationsthreads (AT) muß für die unterbrechungsgesteu-

erte Eingangsprüfung verhindert werden, da diese weiterhin das in diesem Fall nicht erwünschte Polling-Verfahren anwenden würden. Dazu erhält der DT eine spezifische Kennung, die am Beginn der Eingangsprüfung abgefragt wird. Andere Threads werden aufgrund des Fehlens dieser Kennung dazu veranlaßt, die CPU abzugeben und anschließend die Eingangsprüfungsfunktion zu verlassen. Auf diese Weise können weiterhin mehrere unabhängige Channel-Devices parallel betrieben werden, was bei einem blockierenden Warten innerhalb des *ch_smi*-Devices nicht möglich wäre. Somit sind keine Änderungen in der Semantik der Eingangsprüfung in höheren Schichten erforderlich.

Jeder AT kann jedoch weiterhin Nachrichten verschicken. Die Verarbeitung eingehender Nachrichten in jedem Protokoll sowie die Abarbeitung des *rendez-vous*-Protokolls erfolgen aber ausschließlich durch den DT. Dabei prüft dieser, sobald er einmal durch eine eingegangene Unterbrechung zur Eingangsprüfung veranlaßt wurde, solange auf neue Nachrichten, bis nach einer durchgehenden Überprüfung der Eingangspuffer für alle entfernten Prozesse in keinem dieser Puffer neue Nachrichten gefunden wurden. Dadurch entsteht ein hybrides Polling- und Unterbrechungs-Verfahren zur Eingangsprüfung, das bei hoher Frequenz von eingehenden Nachrichten die Leistungscharakteristik des Polling-Verfahrens hat, bei niedriger Frequenz jedoch mit maximaler Schonung der CPU-Ressourcen arbeitet.

Die Latenz zur Abarbeitung von einzelnen Nachrichten bei niedriger Eingangsfrequenz steigt so natürlich an, wie in Abb. 7.1 für einen Ping-Pong-Test zwischen zwei Prozessen gezeigt wird: Sobald eine eingegangene Nachricht bearbeitet wurde, werden bei der unmittelbar folgenden Überprüfung aller (zwei) Eingangspuffer keine neuen Nachrichten entdeckt, so daß der Device-Thread erst wieder durch eine Unterbrechung zur Prüfung auf neue Nachrichten veranlaßt werden kann. Dadurch wird die Ping-Pong-Latenz von der Latenz für entfernte Unterbrechungen dominiert.

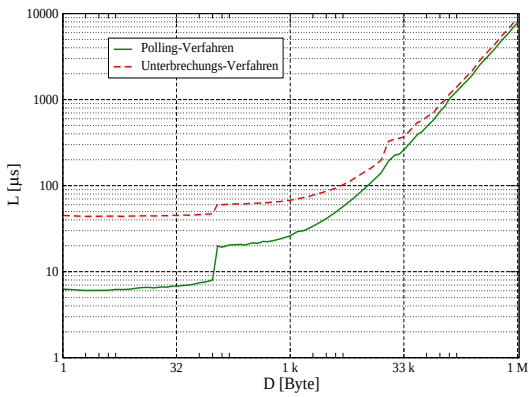


Abbildung 7.1: Ping-Pong Latenz bei Polling- und Unterbrechungs-Verfahren zur Eingangsprüfung

7.1.2 Statistische Reduzierung der Nachrichtenlatenz

Die Ping-Pong-Latenz für diesen Fall kann verringert werden, indem der Prozeß nicht unmittelbar nach Abfrage der Eingangspuffer auf die nächste Unterbrechung wartet, sondern zunächst eine gewisse Zeit lang weiterhin auf neu eingehende Nachrichten prüft. Dabei sind sowohl die Dauer als auch die Art dieser fortgeführten Eingangsprüfung zu wählen. Die verschiedenen möglichen Arten sind

- *Eingangsprüfung mit fortgesetztem Polling* (EFP): unter Nutzung aller verfügbaren Zyklen einer CPU wird weiterhin für einen festen Zeitraum t_{poll} auf eingehende Nachrichten geprüft.
- *Eingangsprüfung mit unterbrochenem Polling (EUP):* der Thread gibt für eine bestimmte

Zeitdauer t_{pause} die CPU ab, um nach Ablauf dieser Zeit erneut eine Eingangsprüfung durchzuführen.

Die EFP verringert den Effekt des CPU-schonenden Verfahrens zur Eingangsprüfung und sollte daher nur für kurze Zeiträume verwendet werden. Die EUP erzielt einen ähnlichen Effekt wie die ausschließlich unterbrechungsgesteuerte Eingangsprüfung; jedoch ist hier entscheidend, wie das Warten für einen bestimmten Zeitraum vom BS implementiert ist. Zum einen ist die Genauigkeit der Zeitdauer im μs-Bereich zumeist relativ gering, was aber für diese Anwendung unkritisch ist. Entscheidend ist, was das BS während dieser Zeitdauer mit dem Thread tatsächlich tut: Nur wenn es tatsächlich einen anderen Thread zur Ausführung bringt, wird so eine CPU-schonendere Eingangsprüfung als bei der EFP erreicht.

Eine obere Grenze für die Dauer des *Pollings* t_{poll} bei der EFP ist die doppelte Latenz l_{int} von entfernten Unterbrechungen. Die Entdeckung einer neu eingegangenen Nachricht nach fortgesetztem Polling über einen Zeitraum von $2 \cdot l_{int}$ (oder länger) bringt gegenüber der reinen unterbrechungsgesteuerten Eingangsprüfung im Mittel keinen Vorteil. Ohne Betrachtung einer bestimmten Applikation ist nicht vorhersagbar, wann ein Sendeprozeß eine Nachricht an einen Empfangsprozeß absendet. Wenn der Empfangsprozeß diese Nachricht im Rahmen einer EFP mit einem Pollintervall t_{poll} entdeckt, wird dies im Mittel nach $t_{poll}/2$ geschehen. Für $t_{poll} = 2 \cdot l_{int}$ entsteht somit eine Latenz von l_{int}, die auch mit rein unterbrechungsgesteuerter Eingangsprüfung hätte erreicht werden können. Analog gilt diese Betrachtung auch für die EUP für das Pauseintervall t_{pause}.

7.1.3 Evaluierung der Eingangsprüfungsverfahren

Relevant wird die Wirkung dieser unterschiedlichen Eingangsprüfungsverfahren, wenn über die Betrachtung des reinen Ping-Pong-Verhaltens zwischen zwei Prozessen hinaus komplexere Kommunikationsmuster zwischen mehr als zwei Prozessen berücksichtigt werden. Dies wurde anhand zweier Kernel-Benchmarks überprüft.

7.1.3.1 Kernel-Benchmarks LU und CG

Entscheidend für die Eignung eines Verfahrens ist die mit dem jeweiligen Verfahren erzielte Leistung in einer Applikation. Dazu wurden zwei Benchmarks der *NAS Parallel Benchmarks* (NPB, *[146]*) ausgewählt, CG und LU. Der CG-Benchmark ist ein nicht-stationärer iterativer Gleichungslöser nach der Methode der konjugierten Gradienten. Der LU-Benchmark löst ein lineares Gleichungssystem, das im Rahmen der Bearbeitung eines Strömungsproblems nach Navier-Stokes aufgestellt wurde. Beide Benchmarks sind kommunikationsintensiv, wobei LU sehr viele kleine Nachrichten austauscht und CG irregulär (d.h. mit einem großen Teil der Prozesse in unregelmäßiger Folge) kommuniziert.

In *[105]* wurde ebenfalls eine derartige Untersuchung mit diesen beiden Benchmarks durchgeführt, die zum Ergebnis hatte, daß die Art der Eingangsprüfung (Polling- oder Unterbrechungs-Verfahren) keinen nennenswerten Einfluß auf die erzielte Leistung hat. Die Allgemeingültigkeit dieser Erkenntnis wird hier geprüft, indem die Hard- und Softwareumgebung vollständig geändert wird. Nur die Merkmale der beiden unterschiedlichen Methoden der Eingangsprüfung sowie die Benchmarkcodes entsprechen der genannten Untersuchung.

Die Ergebnisse als erreichte Gesamtrechenleistung in MFLOPS in Tabelle 7.1 zeigen, daß sich die höheren Latenzen durch das Unterbrechungs-Verfahren zur Eingangsprüfung in kommunikationsintensiven Anwendungen um so stärker auf die erzielte Leistung auswirkt, je mehr Prozesse ein gegebenes Problem lösen. Während die Leistungseinbuße beim CG-Benchmark für 2 Prozesse nur 1,5% beträgt, wächst diese bei 8 Prozessen auf 9,3%. Beim LU-Benchmark fällt dieser Rückgang absolut schwächer aus, obwohl hier bei 8 Prozessen jeder Prozeß rund 31000 Nachrichten sendet und empfängt, während dies beim CG-Benchmark nur rund 3000 Nachrich-

	LU			CG		
Prozesse	**Polling**	**Unterbr.**	**relativ**	**Polling**	**Unterbr.**	**relativ**
2	211,1	210,7	99,8%	87,97	86,68	98,5%
4	419,4	417,1	99,4%	163,9	155,75	95,0%
8	826,7	816,9	98,8%	297,10	269,56	90,7%

Tabelle 7.1: Erreichte Rechenleistung in MFLOPS von LU- und CG-Benchmark für unterschiedliche Eingangsprüfungsverfahren

ten sind. Dies weist darauf hin, daß beim LU-Benchmark mehr als eine Nachricht pro Unterbrechung verarbeitet wird. Daneben ist der Anteil der Kommunikation an der Gesamtlaufzeit beim LU-Benchmark geringer.

Es zeigt sich in beiden Fällen, daß Leistungs*gewinne*, wie in *[105,106]* ermittelt, nicht allgemein durch Ersetzung der Verfahren zur Eingangsprüfung (Unterbrechung anstatt Polling) erwartet werden können. Das Unterbrechungs-Verfahren sollte daher nicht pauschal verwendet werden, sondern nur, wenn weniger als eine CPU pro Prozeß zur Verfügung steht. Neben dem reinen Unterbrechungs- bzw. Polling-Verfahren sind jedoch auch hybride Lösungen denkbar, die die Vorteile beider Verfahren nutzen. Eine derartige Lösung wird in Kapitel 7.3 vorgestellt.

7.2 DMA-gestützte Direktübertragung

Die in Kapitel 4 beschriebenen Protokolle zum Nachrichtenversand arbeiten auf der Seite des empfangenden Prozesses alle mit einem Eingangspuffer, in den die Daten vom Sender geschrieben werden. Von dort muß der Empfänger die Daten in den Empfangspuffer übertragen. Dies hat Vorteile, wenn beispielsweise beim *eager*-Protokoll der Sender seine Daten unmittelbar, ohne Mitwirkung des Empfängers, übertragen kann, weil eben dieser Eingangspuffer bereitsteht und dem Sender bekannt ist. Auch die effektive Bandbreite kann trotz des zusätzlichen Kopiervorgangs zwischen dem Eingangspuffer und dem Benutzerpuffer durch Pipelining nahe an das theoretische Maximum geführt werden, wie beim *rendez-vous*-Protokoll gezeigt wurde. Dies gelingt jedoch nur durch aktive Mitwirkung der CPU des Empfängers.

Die bisher vorgestellten Protokolle arbeiten dabei alle mit PIO-Datentransfers. Diese Übertragungsart hat Eigenschaften, die sich nachteilig auf die effektive Leistung einer Applikation auswirken können:

- PIO-Datentransfers ermöglichen sehr niedrige Latenzen für kleinere Nachrichtengrößen. Bei größeren Nachrichten geht diese geringe Startlatenz jedoch nur noch zu einem geringen Teil in die Gesamtlatenz der Nachricht ein. Hier ist für Anwendungen oftmals entscheidend, wie hoch die CPU-Last beim Transport der Daten ist (siehe dazu auch Kapitel 7.3).
- Datenzugriffe der CPU auf den Hauptspeicher, wie sie bei PIO-Datentransfer inhärent sind, wirken sich stets auch auf den Zustand der CPU-Caches aus und verdrängen dort möglicherweise Daten, die kurz darauf wieder benötigt werden. Dieser Effekt tritt bei PIO-Datentransfers über SCI beim Sendeprozeß durch das Lesen der Quelldaten und beim Empfangsprozeß durch das Kopieren des Eingangspuffers in den Empfangspuffer auf. Zudem ist beim Sendevorgang mittel PIO-Datentransfer die Bandbreite davon abhängig, ob die Quelldaten bereits im Cache liegen oder nicht. Diese Effekte treten bei der Übertragung der Daten via DMA nicht auf, da dort der Datentransfer vom PSA an den CPU-Caches vorbei betrieben wird.
- Je nach verwendeter Plattform liegt die Bandbreite von DMA-Datentransfers oberhalb der von PIO-Datentransfers. In jedem Fall ist sie unabhängig von der CPU-Leistung. Abbildung

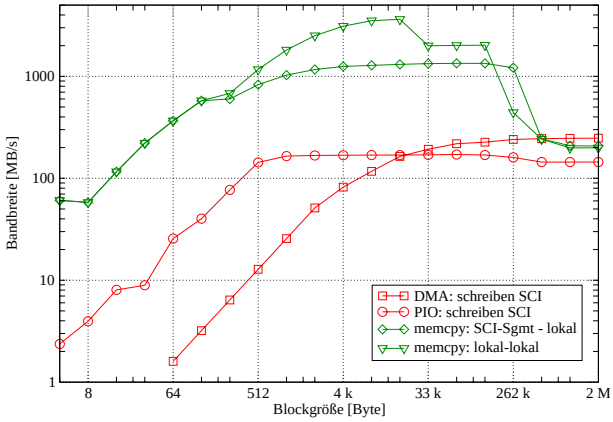


Abbildung 7.2: Bandbreite für Intra- und Inter-Knoten-Speicherkopieroperationen (P3-Plattform)

7.2 zeigt die Bandbreite für Intra- und Inter-Knoten-Speicherkopieroperationen auf der P3-Plattform. Insbesondere wird sichtbar, daß die DMA-Datentransfers eine von dem Cache-Zustand unabhängige Bandbreite bei höherer Latenz aufweisen.

- DMA-Übertragungen via SCI benötigen ein EVS und ein LRS, insgesamt also weniger SCI-Ressourcen als die PIO-Übertragung, die ein EES mit zugehörigen ATT-Einträgen und gemeinsamen Adreßraum benötigt. Wenn also eine PIO-Übertragung aus Ressourcenmangel nicht stattfinden kann, ist eine DMA-Übertragung eventuell noch möglich. Bei DMA-Übertragungen ist durch den geringeren Ressourcenverbrauch die Wahrscheinlichkeit, daß eine andere Ressource de-allokiert werden muß, um die Übertragung durchzuführen, geringer. Jedoch kann eine DMA-Übertragung aufgrund unzulässiger Ausrichtung der Startadressen der Kommunikationspuffer scheitern.

7.2.1 Direktübertragung mit DMA

Datenübertragung mittels DMA ist bei den meisten Typen von Netzadaptern gängige Praxis; der direkte Zugriff auf entfernten Speicher ist ja ein spezifisches Merkmal von SCI. Jedoch kommt die übliche Übertragung mittels DMA, etwa auch bei Ethernet-Netzen, nicht ohne lokale Kopiervorgänge durch die CPU aus. Diese kopieren die Daten zwischen dem durch den Nutzer spezifizierten Puffer und einem vom Treiber des Netzadapters allokierten Speicherbereich. Dies ist notwendig, da der Netzadapter nur auf nicht auslagerbare Speicherbereiche zugreifen kann, die zudem möglichst physikalisch zusammenhängend sind[1] und deren physikalische Adresse bekannt ist. So sind beim Sender und beim Empfänger jeweils mindestens ein Kopiervorgang erforderlich. Dies begrenzt die effektive Bandbreite, die für die Datenübertragung erreicht werden kann. Zugriff auf Arbeitsspeicher durch andere Hardware als der CPU ist nur dann möglich, wenn die entsprechenden Speicherseiten *blockiert* wurden, d.h. für das BS als nicht auslagerbar und nicht verschiebbar gekennzeichnet wurden. Ein solches Verfahren ist möglich und macht die beschriebenen Kopiervorgänge entbehrlich.

Der potentielle Leistungsgewinn wird in Abb. 7.3 für ein Beispiel gezeigt, bei dem lokale Kopiervorgänge mit einer Bandbreite von 200 MB/s abgewickelt werden können. Das Blockieren von Speicher kann hingegen mit einer Bandbreite von 4 GB/s erfolgen. Die Netzbandbreite B_{Netz} variiert von 0…300 MB/s (diese Werte sind angelehnt an die Werte der P3-Plattform). Die Leistung wird ausgedrückt als Anteil der effektiven Ende-zu-Ende-Bandbreite an B_{Netz}, also als Effizienz. Dargestellt sind drei Fälle:

- *C1:* zwei lokale Kopieroperationen sowohl beim Sender als auch beim Empfänger

1. Dies vereinfacht die Adressierung von Datenbereichen beliebiger Größe innerhalb dieses Bereichs.

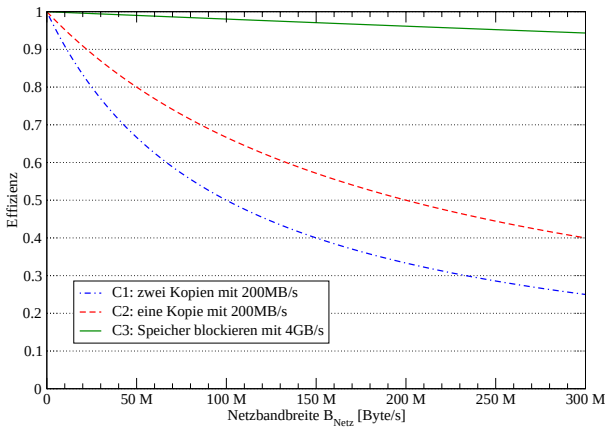


Abbildung 7.3: Effizienz der Datenübertragung über eine Netzverbindung in Abhängigkeit der notwendigen lokalen Kopieroperationen

- *C2:* eine lokale Kopieroperation beim Sender oder beim Empfänger
- *C3:* Verzicht auf lokale Kopieroperationen; stattdessen Blockierung des Speichers für direkten DMA-Zugriff.

Es wird deutlich, daß die Kopieroperationen für die Datenübertragung über ein Fast-Ethernet-Netz mit $B_{Netz} \approx 10\text{MB}/\text{s}$ die Effizienz nicht wesentlich reduzieren. Für eine Netzbandbreite von 100 MB/s führen 2 Kopieroperationen (C1) jedoch zu einer Effizienz von nur noch 50 %, während die Blockierung des Speichers und ein direkter Zugriff für dieses Beispiel eine Effizienz von unwesentlich unter 100 % bieten würde. Dies deckt sich mit den Auswertungen der ersten Implementation der Nutzung von DMA-Übertragungen in SCI-MPICH *[112]*.

Auf die beschriebenen Kopiervorgänge, und somit auf die aktive Mitwirkung des Empfängers an der Datenübertragung kann verzichtet werden, wenn der Sender seine Daten direkt in den benutzerdefinierten Empfangspuffer schreibt. Dies ist jedoch für SCI zunächst nicht möglich. Dazu muß dieser Puffer über den SCI-Adreßraum exportiert werden, um den Sender diesen in seinen Adreßraum einblenden zu lassen. Dies ist in der existierenden Implementation von PSA und der Treibersoftware nur dann möglich, wenn der Puffer über den SCI-Treiber allokiert wurde. Falls der Puffer jedoch, wie es in Applikationen die Regel ist, vom Heap oder Stack des Prozesses allokiert wurde oder im Datensegment des Prozesses liegt, ist dies aus zweierlei Gründen nicht möglich:

- In diesem Fall haben die physikalischen Speicherseiten, die den Sendepuffer bilden, nichtfortlaufende Adressen, wie in Abb. 7.4 dargestellt. Alle Datentransport- und -zugriffsverfahren im SCI-Treiber gehen jedoch davon aus, daß sich alle Adressen eines Segments über eine einzige Startadresse und die Länge des Datenblocks vollständig beschreiben lassen.
- Die physikalischen Speicherseiten dürfen für die Dauer der Datenübertragung, und für wiederholte Übertragungen auch darüber hinaus, nicht von der Speicherverwaltung des Betriebssystems ausgelagert werden. Ansonsten könnten falsche Daten gelesen oder überschrieben werden, da die physikalischen Speicherseiten, auf die die Hardware des PSA zugreift, inzwischen andere virtuelle Adressen repräsentieren können.

Ein Export eines derartig gebildeten Puffers für den entfernten Zugriff über SCI ist unter Berücksichtigung dieser Erfordernisse seitens des PSA jedoch sowohl für DMA- als auch für PIO-Datentransfer möglich, wie im Rahmen der Entwicklung von SCI-MPICH ermittelt wurde *[52]*. Die Umsetzung dieses Verfahrens für DMA-Übertragungen ist weniger komplex zu implementieren als für PIO-Übertragungen, da dazu der entfernte Speicher nicht (unter Nutzung der ATT-Einträge) in den Adreßraum eingeblendet werden muß, sondern lediglich die Adressen der

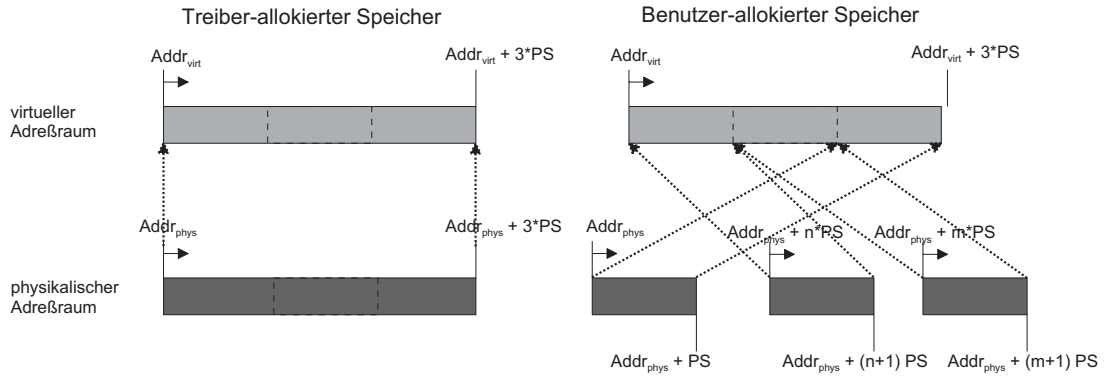


Abbildung 7.4: Zuordnung physikalischer Seiten zu virtuellen Seiten für Treiber- und Benutzer-allokierten Speicher (PS ist die Seitengröße der Speicherverwaltung).

Speicherseiten auf dem entfernten Knoten bekannt sein müssen. Daher wurde im Zusammenhang mit dieser Arbeit gemeinsam mit Friedrich Seiffert[1] eine Erweiterung des SCI-Treibers entwickelt, so daß dieser auch mit physikalisch nicht-zusammenhängenden Segmenten umgehen kann *[52]*. Diese Erweiterung baut auf der Funktionalität eines Kernmoduls (LMM, *locked memory manager [60]*) für das BS Linux auf, das die Auslagerbarkeit von physikalischen Seiten Zeit-effizient steuert. Das im BS-Kern laufende Modul kann dynamisch festlegen, daß Speicherseiten nicht ausgelagert werden dürfen. Dieses Blockieren kann auch verschachtelt und für überlappende Bereiche erfolgen, so daß ein robustes Verfahren zur sogenannten *Registrierung* von Speicher zur Nutzung via SCI zur Verfügung steht.

Mit dieser Erweiterung ist es nun möglich, beliebige Benutzer-allokierte Speicherbereiche als Quelle für DMA-Transfers zu verwenden. Außerdem können solche Speicherbereiche als SCI-Segment exportiert werden, so daß ein Prozeß auf einem entfernten Knoten diesen Speicherbereich als Quelle oder Ziel für DMA-Operationen nutzen kann. Derartige SCI-Speichersegmente werden als *Benutzersegmente* bezeichnet. Die beschriebenen unterschiedlichen Eigenschaften von DMA- und PIO-Datentransfers lassen eine alternative Verwendung von *beiden* Transferarten als wünschenswert erscheinen, um je nach Situation und Anwendungsfall die optimale Art des Datentransfers nutzen zu können. Dies bietet zudem auch die Möglichkeit, den Grad der Auswirkung der beschriebenen Effekte auf die effektive Leistung einer Applikation zu beurteilen, wenn alternativ die eine oder die andere Übertragungsmethode für den gleichen Anwendungsfall verwendet werden kann.

7.2.2 Speicherallokation

Sobald nicht nur die CPU, sondern auch E/A-Hardware wie hier ein PSA auf Speicherbereiche zugreifen soll, sind, wie oben beschrieben, besondere Bedingungen zu beachten. Wie gut diese Bedingungen erfüllt werden können, hängt auch von der Art der Speicherallokation und Bereitstellung der Kommunikationspuffer ab. MPI bietet unterschiedliche Verfahren an, deren Nutzung die Sicherstellung dieser Bedingungen vereinfachen. Da es Ziel dieser Arbeit ist, mit beliebig spezifizierten Kommunikationspuffern die optimale Kommunikationsleistung zu erzielen, beinhaltet dies auch, via DMA auf diese Puffer zuzugreifen.

7.2.2.1 MPI Speicherverwaltung

Die bezüglich der erreichbaren Kommunikationsleistung effektivste Methode, in einem MPI-Programm Speicher für Kommunikationspuffer zu allokieren, ist die Verwendung der MPI-eigenen Speicherallokations-Funktionen `MPI_Alloc_mem` und `MPI_Free_mem`. Ein solches Vorgehen ermöglicht der MPI-Bibliothek entsprechend vorgegebener Regeln oder durch Hin-

1. TU Chemnitz, Professur für Rechnerarchitektur

weise des Benutzers, bestimmte Typen von Speicher zur Befriedigung der Anforderung einzusetzen.

SCI-MPICH verfolgt hierbei ein mehrstufiges, von der Größe der Speicheranforderung abhängiges Allokationsverfahren. Speicheranforderungen unterhalb des Schwellwerts $D_{alloc_{pool}}$ werden aus dem allgemeinen Heap des Prozesses bedient. Dies ist sinnvoll, da sich für derartig kleine Kommunikationspuffer die Direktübertragung via DMA aufgrund der hohen Startlatenz nicht rentiert. Derartige Datenmengen lassen sich weitaus effektiver mit dem PIO-gestützten *short*- oder *eager*-Protokoll übertragen. Speicheranforderungen mit einer Größe zwischen $D_{alloc_{pool}}$ und $D_{alloc_{sgmt}}$ werden aus einem im Bedarfsfall anzulegenden Pool, bestehend aus einem SCI-Speichersegment, bedient. Dadurch wird vermieden, für viele relativ kleine Speicheranforderungen jeweils ein neues SCI-Speichersegment anlegen zu müssen und damit den Ressourcenverbrauch unnötig zu erhöhen. Für Speicheranforderungen mit einer Größe oberhalb von $D_{alloc_{sgmt}}$ wird jedoch ein eigenes SCI-Speichersegment angelegt. Dies ist erforderlich, da Speicher-Pools nur eine feste maximale Größe haben können. Diese darf im Verhältnis zum gesamten verfügbaren SCI-Segmentspeicher des Knotens nicht zu groß werden, um andere Speicheranforderungen (ggf. von anderen Prozessen auf demselben Knoten) nicht zu blockieren.

Neben diesen impliziten Regeln zur Speicherallokation kann der Benutzer explizit wählen, welchen Typ von Speicher mit welchen Eigenschaften er erhält. Dies geschieht mit Hilfe von *Attributen*, die bei der Speicheranforderung übergeben werden. Jedes Attribut besteht aus einem Schlüssel und ggf. einem zugehörigen Wert. Was ein gültiger Schlüssel und Wert ist, ist abhängig von der jeweiligen MPI Implementation. Im Bereich der Speicherallokation gibt es bisher keine verbindlichen Attribute. Daher wurden für SCI-MPICH folgende Attribute definiert und implementiert:

type. Der Schlüssel *type* bestimmt den Typ des Speichers, mit dem die Anforderung bedient werden soll. Gültige Werte für diesen Schlüssel sind

- *shared:* Anforderung *muß* aus gemeinsamem Speicher, in diesem Fall einem SCI-Speichersegment, bedient werden. Dies ist für Kommunikationspuffer wichtig, mit denen die maximal mögliche Leistung erreicht werden soll.
- *private:* Anforderung *muß* aus privatem Speicher des Prozesses bedient werden. Dies ist dann sinnvoll, wenn es sich um einen großen Puffer handelt, der nicht mit der maximalen Leistung übertragen werden muß. Somit kann vermieden werden, daß eine Anforderung einen großen Teil der SCI-Speicherressourcen erschöpft. Dennoch hat auf diese Art die MPI-Bibliothek die Kontrolle über den Speicherbereich.
- *default*: Die MPI-Bibliothek verfährt bei der Wahl des Speichertyps gemäß der beschriebenen integrierten Regeln.

alignment. Die Leistung bei der Übertragung eines Puffers kann durch ungünstige Ausrichtung der Adressen negativ beeinflußt werden. Der Wert zum Schlüssel *alignment* gibt an, auf welche Grenze der zu allokierende Speicher ausgerichtet werden soll. Bei großen Puffern macht es durchaus Sinn, hier als Grenze die Größe einer Speicherseite zu wählen, da der Verschnitt dabei relativ gering bleibt.

7.2.2.2 Nutzung von benutzerallokiertem Speicher

Jeder Adreßbereich, der vom Benutzer als Sende- oder Empfangspuffer angegeben wird, wird zunächst daraufhin überprüft, ob er Teil einer Speicherregion unter Verwaltung der SMI-Bibliothek ist. Ist dies der Fall, handelt es sich entweder um ein LS oder ein LRS; in beiden Fällen kann der Puffer unmittelbar als Quell- oder Zielpuffer einer DMA-Übertragung genutzt werden. Steht der Bereich nicht unter Verwaltung der SMI-Bibliothek, muß zunächst eine entsprechende SMI-Speicherregion erstellt werden, die den Bereich als LRS verwaltet. Diese Vorgänge werden

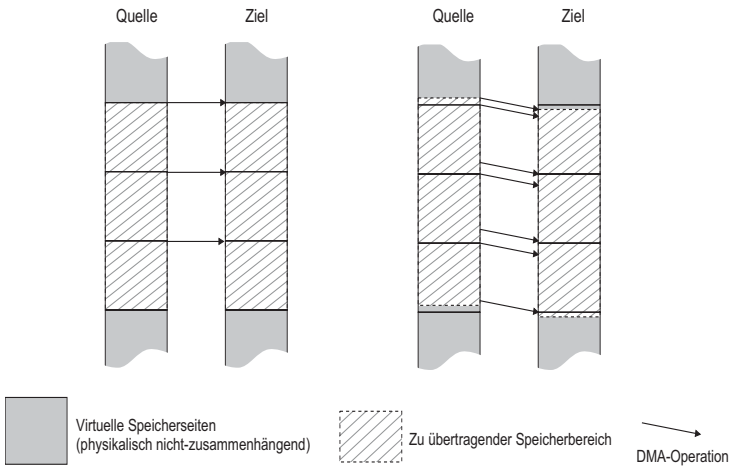


Abbildung 7.5: Erforderliche DMA-Operationen für eine Datenübertragung von 3 im virtuellen Adreßraum zusammenhängenden Speicherseiten zwischen zwei benutzer-allokieren Speicherbereichen
Links: Optimalfall, 3 DMA-Operationen *Rechts:* Schlechtester Fall, 7 DMA-Operationen

über die in Kapitel 4.4 vorgestellte Ressourcenverwaltung vorgenommen.

Anschließend können Daten aus diesem Bereich aufgrund der beschriebenen Erweiterungen des SCI-Treibers via DMA gesandt oder empfangen werden. Die hierbei erzielte Bandbreite liegt jedoch unterhalb der Bandbreite, die bei der DMA-Übertragung zwischen zusammenhängend allokierten SCI-Segmenten erreicht wird. Dies liegt daran, daß aufgrund der Zerstückelung der Speicherbereiche auf Seitengranularität zur Übertragung der gleichen Datenmenge eine größere Zahl von DMA-Operationen durchgeführt werden muß, die jeweils entsprechend kleinere Datenmengen bewegen. Die Extremfälle, die dabei auftreten können, sind in Abb. 7.5 dargestellt: Links ist die ideale gegenseitige Ausrichtung der virtuellen Start- und Zieladresse auf Seitengrenzen abgebildet. Der Inhalt der drei Speicherseiten kann in 3 DMA-Transfers mit jeweils der gleichen Länge von einer Seite übertragen werden. Rechts ist der schlechtest denkbare Fall dargestellt: die Quelladresse hat minimalen negativen Versatz zur Seitengrenze, die Zieladresse hat minimalen positiven Versatz zu einer Seitengrenze. Dadurch ist der virtuell zusammenhängende Speicherbereich so auf physikalische Seiten verteilt, daß insgesamt 7 DMA-Transfers erforderlich sind, von denen 4 sehr klein sind und so eine geringe effektive Bandbreite erzielen.

Die Auswirkungen auf die erzielte effektive DMA-Bandbreite sind in Abb. 7.6 dargestellt. Die Bandbreite im schlechtesten Fall (entsprechend Abb. 7.5 *rechts*) beträgt nur etwa 50% vom Optimum. Bereits eine Ausrichtung an 128-Byte-Grenzen steigert die erreichte Bandbreite auf 80% des Maximalwerts. Dies macht deutlich, daß die Ausrichtung der Kommunikationspuffer, wie sie mit der MPI-Speicherallokations-Funktion möglich ist, bei Nutzung von DMA zur Erzielung der optimalen Kommunikationsleistung wichtig ist.

7.2.2.3 Persistente Kommunikation

Neben der Allokation von Speicher durch die MPI-Bibliothek bietet der MPI-Standard eine Möglichkeit, Speicherbereiche explizit für wiederholte Kommunikationsvorgänge zu kennzeichnen. Ein solcher Vorgang wird als *persistente Kommunikation* bezeichnet und besteht daraus, eine Sende- oder Empfangsoperation vollständig vorzubereiten, aber zu diesem Zeitpunkt noch nicht durchzuführen (`MPI_Send_init` bzw. `MPI_Recv_init`). Diese Vorbereitung bindet auch einen Speicherbereich an die Operation. Die Durchführung der derartig vorbereiteten Operation wird durch einen Aufruf von `MPI_Start` ausgelöst. Erst eine Freigabe der persistenten Kommunikation (durch `MPI_Request_free`) läßt zu, daß der Puffer für andere Zwecke

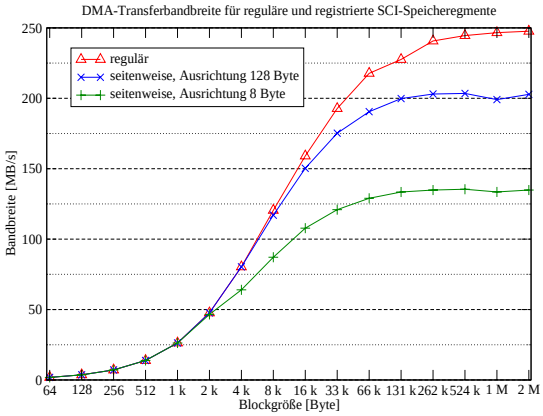


Abbildung 7.6: Abhängigkeit der DMA-Bandbreite zwischen SCI-Segmenten von Benutzer-allokiertem Speicher von der Ausrichtung der Startadressen auf Seitengrenzen (zur Referenz: Übertragung zwischen Treiber-allokierten SCI-Segmenten).

benutzt oder de-allokiert wird.

Diese explizite Bindung der Puffer an Kommunikationsoperationen ermöglicht es der MPI-Bibliothek, potentiell zeitaufwendige Operation durchzuführen, die die Kommunikationsleistung mit diesen Puffern verbessern. SCI-MPICH tut dies durch eine Registrierung der Puffer für die Direktübertragung; der potentielle leistungssteigernde Effekt der persistenten Kommunikation wird in diesem Fall jedoch durch das Caching der Registrierung auch für nicht-persistente Kommunikationsoperationen erzielt, wenn eine Bedingung bezüglich der Speicherfreigabe (siehe nächstes Kapitel) erfüllt ist.

7.2.2.4 Speicherfreigabe

Ein wesentliches Problem der Registrierung von virtuellen Speicherbereichen für den Gebrauch als Kommunikationspuffer im direkten SCI-Zugriff ist die Frage, wann die Registrierung wieder aufgehoben werden muß. Der einfachste Ansatz besteht daraus, vor jeder DMA-Übertragung die relevanten Speicherbereiche zu registrieren, um sie direkt nach der Übertragung wieder zu deregistrieren. Die Deregistrierung ist aus zwei Gründen erforderlich:

- Um das System in einem stabilen Zustand zu halten, dürfen nicht beliebig viele Speicherseiten durch die Registrierung als *nicht auslagerbar* gekennzeichnet werden. Auftretende Speicherengpässe können dann nicht mehr bewältigt werden. Allerdings ist dies je nach Auslegung des Systems mehr oder weniger kritisch; man kann ein System auch vollständig ohne Auslagerungsspeicher betreiben[1].
- Wenn durch eine Deallokation und erneute Allokation von dynamischem Speicher der neu allokierte Speicherbereich die gleichen virtuellen Adressen wie ein zuvor registrierter, dann jedoch de-allokierter Speicherbereich enthält, können diesem Bereich dennoch andere physikalische Speicherseiten zugeordnet sein. Geht der SCI-Treiber aufgrund der identischen virtuellen Adressen von der Zuordnung der virtuellen zu physikalischen Seiten aus, wie sie durch die Registrierung des inzwischen de-allokierten Speichers festgelegt wurde, greift er auf die falschen physikalischen Seiten zu, was fatale Folgen hat.

Diese Probleme treten nicht auf, wenn für jede Übertragung ein vollständiger Registrierung-Deregistrierung-Zyklus durchgeführt wird. Dieser ist jedoch zeitaufwendig; es ist daher wünschenswert, das in Kapitel 4.4 vorgestellte Caching von Ressourcen auch auf diese Ressource anzuwenden.

Eine Nutzung eines solchen *Segmentcaches* dann möglich, wenn bei jeder Deallokation von

1. Dies wird beispielsweise auf Cray-Systemen (UNICOS Betriebssystem) so gehandhabt.

Speicherbereichen überprüft wird, ob dieser Speicherbereich registrierte Speicherseiten enthält, und diese Seiten ggf. deregistriert werden. Diese Funktionalität wurde in `MPI_Free_mem` implementiert. Um jedoch beliebige Applikationen, die dynamischen Speicher nicht mit den entsprechenden MPI-Funktionen verwalten, sicher mit Direktübertragung nutzen zu können, müssen die Funktionen der C-Bibliothek (`malloc` und insbesondere `free`) mit entsprechender Funktionalität ausgestattet werden. In SCI-MPICH wurde dies auf Quellcode-Ebene erreicht, indem diese Funktionen über Makros auf die MPI-Funktionen zur Speicherverwaltung umgelenkt werden. Optional könnten auch die entsprechenden Funktionen der C-Bibliothek durch eigene Funktionen ersetzt werden.

Wenn Speicher freigegeben wird, der Teil eines Benutzersegments ist, muß dieses Speichersegment ebenfalls aufgelöst werden. In diesem Fall ist der entsprechende Eintrag im Segmentcache eines entfernten Prozesses, der zuvor Daten in dieses aufzulösende Speichersegment übertragen hat, nicht mehr gültig. Die Konsistenz dieses Cache wird für diesen Fall über ein Segment-*callback* (Rückruffunktion) gewährleistet, das den entsprechenden Cache-Eintrag invalidiert. Gleiches gilt auch, wenn ein Prozeß ein lokales reguläres SCI-Speichersegment, das zuvor für eine entsprechende Speicherallokation angelegt wurde, auflösen muß, da das Segment-*callback* für alle SCI-Speichersegmente eingerichtet wird, in die mittels Direktübertragung Daten gesendet werden.

7.2.3 Einbindung in das rendez-vous-Protokoll

Die Implementation der Direktübertragung erfolgt als eine Variante des *rendez-vous*-Protokolls. Das Verfahren ist in Abb. 7.7 dargestellt. Innerhalb des Sendeaufrufs (hier: blockierender Versand mittels `MPI_Send`) muß der Sender entscheiden, ob das herkömmliche Protokoll oder die Direktübertragung verwendet werden soll. Diese Entscheidung wird über die Nachrichtengröße getroffen, die über einem Schwellwert D_{direkt} liegen muß. Diesen Schwellwert kann man über die allgemeine Bestimmungsgleichung für die erforderliche Zeitdauer t_{msg} einer Nachrichtenübertragung der Größe D Bytes bestimmen:

$$(7.1)\quad t_{msg} = \alpha + \beta \cdot D$$

Hierbei ist α die Startlatenz und β die Übertragungsdauer pro Byte. Im Vergleich zwischen dem herkömmlichen PIO-*rendez-vous*-Protokoll und der Direktübertragung setzen sich die Koeffizienten im schlechtesten Fall wie folgt zusammen:

$$(7.2)\quad \alpha_{PIO} = 2 \cdot t_{control} + t_{rmtmap}(S_{rv-sgmt})$$

$$\alpha_{direkt} = 2 \cdot t_{control} + 2 \cdot t_{register}(D) + t_{connect}(D)$$

Bei beiden Varianten müssen vor der Datenübertragung zwei Kontrollnachrichten ausgetauscht werden. Zusätzlich muß zur PIO-Übertragung ggf. noch das entfernte SCI-Speichersegment der Größe $S_{rv-sgmt}$ in den lokalen Adreßraum eingeblendet werden. Bei der Direktübertragung müssen zunächst sowohl Sende- als auch Empfangspuffer (hintereinander) registriert werden und anschließend eine Verbindung an den Empfangspuffer hergestellt werden. Sowohl die Einblendung und die Registrierung als auch die Verbindung können jedoch bereits erfolgt sein und aus dem Ressourcen-Cache bedient werden, womit die Zeiten drastisch reduziert werden.

Die Übertragungsbandbreiten β_{PIO} und β_{direkt} sind generell plattformabhängig. Die PIO-Bandbreite ist jedoch zusätzlich noch abhängig von dem Zustand der Caches (Daten aus dem Cache können schneller geschrieben werden) und der Menge der Daten im Bezug zur Cache-

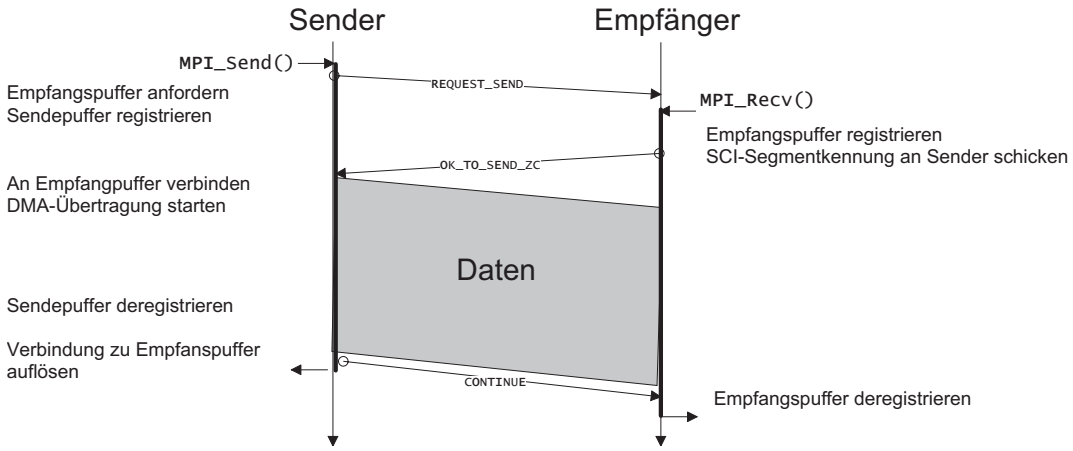


Abbildung 7.7: Variante des *rendez-vous*-Protokolls zur Direktübertragung via DMA

Größe (siehe Kapitel 4.3.2.4).

Die optimale Nachrichtengröße D_{direkt} zum Übergang zur Direktübertragung läßt sich durch Aufstellen und Gleichsetzen zweier Bestimmungsgleichungen gemäß (7.1) unter Berücksichtigung der Parameter aus (7.2) bestimmen. Allerdings ist der Parameterraum, der durch die vielen Varianten bei den Startlatenzen und Bandbreiten, alle abhängig von den Leistungseigenschaften der aktuellen Plattform, aufgespannt wird, derartig groß, daß es sinnvoller erscheint, D_{direkt} für den Optimalfall (alle Ressourcen bereits allokiert und im Cache, Daten im Cache, Sende- und Empfangspuffer auf Seitengrenze ausgerichtet) und den entsprechenden schlechtesten Fall experimentell zu bestimmen und daraus einen geeigneten Übergangspunkt abzuleiten.

Wenn der Sender entschieden hat, daß die Nachricht mittels Direktübertragung versandt werden soll, registriert er den Sendepuffer, was eine notwendige Vorraussetzung für die Direktübertragung ist. Wie beim bisherigen *rendez-vous*-Protokoll informiert der Sender den Empfänger sodann über eine `REQUEST_TO_SEND`-Kontrollnachricht von dem anstehenden Nachrichtenversand. Dabei wird in diesem Fall der Empfänger auch darüber informiert, daß die Übertragung mittels DMA erfolgen soll. Dies veranlaßt den Empfänger, seinerseits den Empfangspuffer zu registrieren und im Erfolgsfall in der `OK_TO_SEND`-Kontrollnachricht die Kennung des dabei eingerichteten Segments zu übermitteln. Nach deren Erhalt verbindet sich der Sender mit dem Empfangpuffer (Erzeugung eines EVS) und startet die DMA-Übertragung. Von deren Abschluß wird der Empfänger durch eine `CONTINUE`-Kontrollnachricht in Kenntnis gesetzt.

Die registrierten Speicherbereiche wie auch die Verbindung zum Empfangspuffer können nun aufgelöst werden, verbleiben jedoch im Cache der Ressourcenverwaltung.

7.2.4 Leistungsevaluation

Zunächst sollen die Auswirkungen der verschiedenen Protokollvarianten auf die erzielte Punkt-zu-Punkt-Bandbreite quantifiziert werden. Deren Einfluß auf die Verarbeitung von Applikationen wird anschließend beispielhaft untersucht.

7.2.4.1 Punkt-zu-Punkt Bandbreite

Die Punkt-zu-Punkt Bandbreite zwischen zwei Prozessen auf unterschiedlichen Knoten wurde mit dem *mpptest*-Benchmark *[144]* für eine Reihe von unterschiedlichen Kommunikationszenarien ermittelt. Diese bestehen daraus, daß innerhalb einer Messung eine unterschiedliche Anzahl von Wiederholungen des Sende-/Empfangsvorgangs durchgeführt wurden (hier: 1, 10 oder 100). Zusätzlich wurde das Caching von Verbindungen zu entfernten Segmenten genutzt oder deaktiviert, und es wurden Kommunikationspuffer verwendet, die mittels `MPI_Alloc_mem` ge-

zielt als gemeinsamer Speicher allokiert wurden, oder die mittels der Standard-Allokations-Funktion `malloc` vom Heap allokiert wurden. Die Ergebnisse dieser Messungen für Nachrichtengrößen von 128 kB, 512 kB und 2 MB sind in Tabelle 7.2 dargestellt. Zusätzlich ist für die Nachrichtengröße von 2 MB die jeweils erreichte Implementationseffizienz angegeben, die sich auf die maximale DMA-Bandbreite von 249 MB/s für reguläre Segmente und 212 MB/s für Benutzersegmente bezieht.

Wdh.	Segment Caching	Segment Typ	128kB Bandbreite	512kB Bandbreite	2MB Bandbreite	2MB Effizienz
1	nein	regulär	78.44	154.80	203.48	81,7%
		registriert	39.77	57.59	64.32	30,3%
	ja	regulär	81.83	158.27	205.65	82,6%
		registriert	43.13	59.87	64.89	30,6%
10	nein	regulär	97.21	172.43	210.28	84,4%
		registriert	55.81	85.92	98.38	46,4%
	ja	regulär	145.66	204.05	222.79	89,5%
		registriert	118.50	149.10	158.22	74,6%
100	nein	regulär	100.45	175.01	211.08	84,8%
		registriert	57.58	89.66	103.73	48,9%
	ja	regulär	159.41	210.37	224.94	90,3%
		registriert	144.20	177.07	185.17	87,3%

Tabelle 7.2: Bandbreite für DMA-Direktübertragung

Wie zu erwarten ist, steigert die Nutzung des Segmentcaches deutlich die erreichte Bandbreite. Die leichte Steigerung, die das Caching bereits bei nur einer Wiederholung erreicht, hat ihre Ursache in der Zeitdauer der Trennung der Verbindung zum entfernten Segment, die in diesem Fall nicht anfällt. Die durchweg höheren Bandbreiten für die Direktübertragung in reguläre Segmente macht den erhöhten Zeitaufwand deutlich, der einerseits bei der Übertragung der physikalischen Adresse jeder einzelnen Seite (und nicht nur des Beginns des Segments) beim Aufbau einer Verbindung zu einem entfernten Benutzersegment, andererseits durch die geringere DMA-Übertragungsrate zwischen Benutzersegmenten anfällt. Ohne den Segment-Cache läge die Übertragungsleistung zwischen Benutzersegmenten um bis zu 60% unterhalb des unter Nutzung des Caches erreichten Werts (128kB Nachrichtenlänge bei 100 Wiederholungen). Die Kommunikationseffizienz des Protokolls steigt unter Nutzung des Ressourcen-Caches auf Werte über 90% an. Bei Benutzersegmenten bewirkt die Nutzung des Segment-Cache annähernd eine Verdoppelung der Kommunikationseffizienz.

Um DMA-Transfers ohne Direktübertragung zu nutzen, also durch Kopieren der Daten in explizit allokierte DMA-Puffer, die statisch miteinander verbunden sind, sind 2 zusätzliche Kopiervorgänge erforderlich (entsprechend dem Kommunikationsmodell aus Kapitel 2.3.1). Zusammen mit den Werten für die herkömmliche PIO-basierte Übertragung und die DMA-Direktübertragung ist die Entwicklung der Bandbreite all dieser Verfahren für Nachrichtenlängen zwischen 64kB und 4MB in Abb. 7.8 dargestellt.

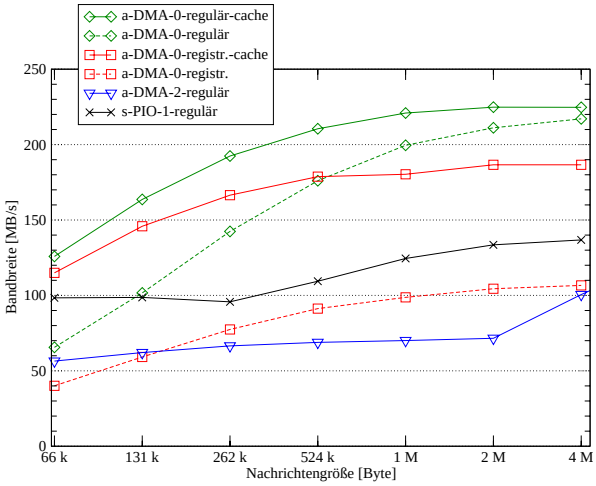


Abbildung 7.8: Bandbreite für unterschiedliche Varianten der DMA-Übertragung sowie PIO-Übertragung im *rendez-vous*-Protokoll

Der Vergleich der Bandbreiten zeigt, daß DMA-Übertragungen zwischen regulären SCI-Speichersegmenten bei Nutzung des Segment-Cache für Nachrichtenlängen oberhalb von 64kB der PIO-Übertragung vorzuziehen sind. Dies ist bei DMA-Übertragungen zwischen Benutzersegmenten ebenfalls der Fall, jedoch nur, wenn eine hohe Trefferquote im Segment-Cache erreicht wird - sonst wird der Aufwand für die Nutzung der Benutzersegmente zu groß gegenüber der PIO-Übertragung.

Unabhängig von der reinen Übertragungsbandbreite hat die DMA-Direktübertragung die weitere positive Eigenschaft, daß die geringe CPU-Last die effiziente Überlappung von Kommunikation und Berechnung ermöglicht (siehe dazu Kapitel 7.3).

7.2.4.2 Einfluß auf Applikationsleistung

Als Beispiel für den Einfluß der erhöhten Bandbreite wurde ein weiterer NPB-Benchmark *[146]* ausgewählt. Der IS-Benchmark sortiert einen Vektor von Integer-Zahlen nach dem Bukket-Sort-Verfahren und tauscht dazu nach dem lokalen Sortieren Teilvektoren der Sortierschlüssel aus. Es wurden zwei verschiedene Klassen W und A untersucht, die sich in der Länge des zu sortierenden Vektors unterscheiden. Tabelle 7.3 zeigt das Kommunikationsverhalten für 4 Prozesse mit PIO-Kommunikation.

Der hohe Kommunikationsanteil an der Laufzeit von etwa einem Drittel macht Verbesserungen in diesem Bereich lohnenswert. Die mittels DMA-Direktübertragung erzielte Leistungssteigerung ist in Tabelle 7.4 für die Nutzung von regulärem SCI-Speicher, der mittels `MPI_Alloc_mem` allokiert wurde, sowie für die Nutzung von Benutzersegmenten dargestellt.

Die Leistungssteigerungen durch die verkürzte Kommunikationszeit, ausgedrückt als Speedup der Gesamtausführungszeit mit DMA-Direktübertragung gegenüber der Gesamtausführungszeit mit PIO-Kommunikation, betragen zwischen 16% und 26%. Wieder wird deutlich, daß für optimierte Kommunikationsleistung der Speicher für die Kommunikationspuffer mittels `MPI_Alloc_mem` allokiert werden sollte. Zusätzlich sollte persistente Kommunikation eingesetzt werden. Beides geschieht in typischen Applikationen, auch in dem getesteten IS-Benchmark in der Originalfassung, nicht. Der Großteil der Entwickler von MPI-Applikationen setzt nur denjenigen kleinen Teil des MPI-API ein, auf den zur Lösung des Problems nicht verzichtet werden kann *[66]*.

7.3 Asynchrone Kommunikation

Der MPI-Standard spezifiziert blockierende und nicht-blockierende Sende- und Empfangsope-

Klasse	Vektorlänge [MB]	Prozesse	Nachrichtenlänge [kB]	Alltoallv Dauer [ms]	% der Gesamtzeit
W	1	4	256	16.4	34.6
A	8	4	2048	123.9	36.2

Tabelle 7.3: Kommunikationsverhalten IS-Benchmark

Klasse	Prozesse	regulär [ms]	Speedup	registriert [ms]	Speedup
W	4	7.578	**1.22**	9.617	**1.16**
A	4	52.415	**1.26**	63.957	**1.21**

Tabelle 7.4: Leistungssteigerung durch DMA-Direktübertragung

rationen. Bei blockierenden Operationen ist der Nachrichtenpuffer bei erfolgreicher Rückkehr der Funktion bereit zur weiteren Verwendung durch die Applikation: Ein Sendepuffer kann verändert werden (da die zu sendenden Daten an anderer Stelle im System gepuffert wurden oder bereits beim Empfänger eingetroffen sind); ein Empfangspuffer enthält den Inhalt der spezifizierten empfangenen Nachricht. Bei der Rückkehr einer nicht-blockierenden Operation ist jedoch über den Inhalt des Nachrichtenpuffers keine Aussage möglich; dieser darf erst wieder von der Applikation genutzt werden, wenn der Abschluß der Operation durch mindestens einen weiteren Funktionsaufruf festgestellt wurde.

Diese Asynchronität der nicht-blockierenden Operationen bietet die Möglichkeit, Kommunikation und Berechnung in einer Applikation zu überlappen. Im Idealfall erfolgt dadurch die Kommunikation nahezu kostenfrei, wenn die Nachricht nach dem Aufruf der nicht-blockierenden Sende- oder Empfangsoperation ohne CPU-Belastung im Hintergrund erfolgt. Die Bandbreite der Übertragung spielt in diesem Fall solange keine Rolle, wie die Applikation nicht auf die Übertragung der Nachricht warten muß, sondern die Übertragungszeit vollständig mit anderen Aufgaben (Berechnungen) überbrückt hat. Um diesen Effekt jedoch möglichst häufig, d.h. bei möglichst kurzen Berechnungsphasen, zu erreichen, muß auch die Bandbreite der asynchronen Übertragung maximiert werden.

7.3.1 Steuerung der Datenübertragung

Gemäß dem MPI-Standard ist es für eine korrekte MPI-Implementation ausreichend, daß Kommunikationsoperationen ausschließlich dann ausgeführt werden, wenn sich die Applikation innerhalb eines MPI-Funktionsaufrufs befindet. Ein solches Vorgehen ist zur Realisierung effizienter asynchroner Kommunikation jedoch unzureichend, wie in Abb. 7.9 (*links*) illustriert ist: Nachdem der Empfänger einen nicht-blockierenden Empfangsaufruf ausgeführt hat und somit den Empfangspuffer für eine passende eingehende Nachricht bereitgestellt hat, trifft eine Sendeanforderung des Senders ein. Diese wird jedoch vom Empfänger nicht bearbeitet, bis er wieder eine MPI-Funktion aufruft (`MPI_Wait`, um den Abschluß der Empfangsoperation festzustellen). Als Folge wird der gesamte Nachrichtentransfer beim Empfänger innerhalb der Funktion `MPI_Wait` abgewickelt. Dieses Problem verschärft sich weiter, wenn der Sender keine blockierende, sondern eine nicht-blockierende Sendeoperation aufgerufen hat, wie in Abb. 7.9 (rechts) dargestellt ist: Die eintreffende Sendebestätigung wird nun wiederum vom

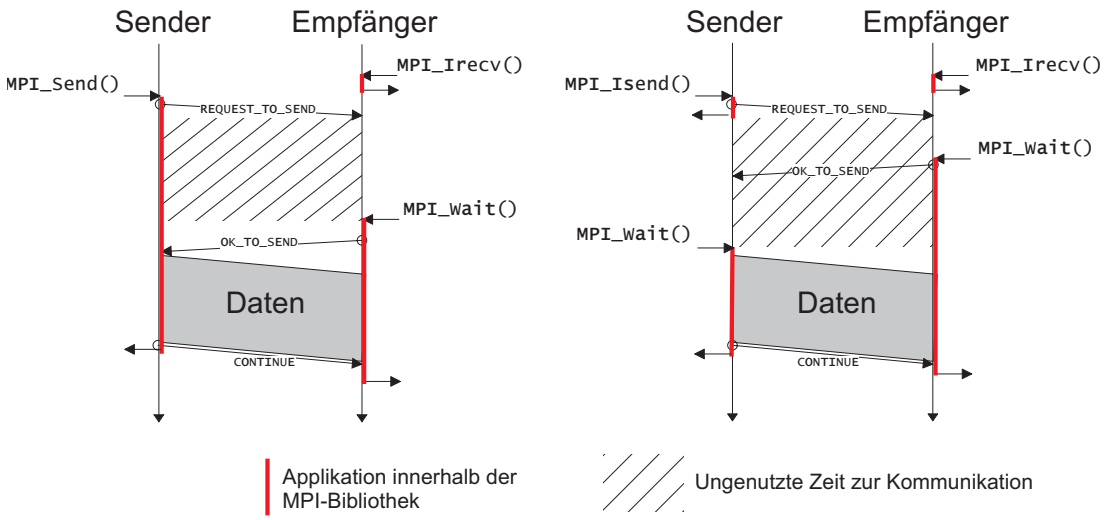


Abbildung 7.9: Problem des Nicht-Fortschreitens der Datenübertragung bei nicht-blockierender Kommunikation.

Sender solange nicht bearbeitet, bis dieser sich durch den Aufruf von `MPI_Wait` erneut innerhalb der MPI-Bibliothek ausstehenden Kommunikationsoperationen widmet.

Als Konsequenz dieser fehlenden unmittelbaren, asynchronen Reaktion auf eingehende Kontrollnachrichten wird eine potentiell zur Übertragung von Daten nutzbare Phase nicht zu diesem Zweck verwendet. Diese Phase beginnt, nachdem eine passende Empfangs- und Sendeaufforderung beim Empfänger einander zugeordnet werden können und reicht bis zum Beginn der eigentlichen Datenübertragung (schraffierter Bereich zwischen den Zeitachsen in Abb. 7.9). Stattdessen findet die vollständige Datenübertragung in der Routine zur Prüfung des Abschlusses der Kommunikation statt.

Um die zur asynchronen Kommunikation erforderliche unmittelbare Reaktion auf eingehende Kontrollnachrichten auszulösen, sind verschiedene Ansätze denkbar, wie sie bereits in Kapitel 7.1 besprochen wurden. Die dort untersuchten alternativen Verfahren zur Eingangsprüfung arbeiten jedoch wiederum ausschließlich mit entfernten Unterbrechungen, was wiederum zu Leistungseinbußen bei gut synchronisierter Kommunikation führt. Daher wurde zur effektiven Abwicklung asynchroner Kommunikation eine hybride Lösung entwickelt, die dem Device-Thread (DT) *nicht* das exklusive Zugriffsrecht auf die Eingangspuffer gewährt: Der DT reagiert weiterhin auf jede eingehende Unterbrechung mit der Prüfung auf neue Nachrichten und deren Abarbeitung; falls der DT aber keinen Zugriff auf die Eingangspuffer benötigt, kann der Applikations-Thread (AT) die Eingangsprüfung mittels Polling effizient durchführen.

Das Prinzip dieser Abwicklung von asynchronen Kommunikationsanforderungen durch einen Device-Thread ist in Abb. 7.10 dargestellt: Nicht-blockierende Sende- und Empfangsoperationen werden durch die DTs von Sender- und Empfängerprozeß abgearbeitet, ohne daß der AT eine MPI-Funktion aufruft. Dazu wird für jede versandte Kontrollnachricht eine Unterbrechung im Zielprozeß der Kontrollnachricht ausgelöst, die vom DT bearbeitet wird. Dadurch kann, wie in diesem Beispiel dargestellt, der gewünschte Effekt der asynchronen Kommunikation erzielt werden: Der Aufruf von `MPI_Wait` muß lediglich den Abschluß der Kommunikation feststellen, aber keine Datenübertragung mehr durchführen. Die eigentliche Datenübertragung fand bereits statt, während der AT beliebige andere Aufgaben bearbeiten konnte.

Das vorgestellte Prinzip zur asynchronen Kommunikation kann auf alle bestehenden Kommunikationsprotokolle angewandt werden. Jedoch ist es ineffizient, asynchrone Kommunikation für kleine Nachrichtengrößen im Bereich des *short*- oder *eager*-Protokolls zu betreiben. Bei die-

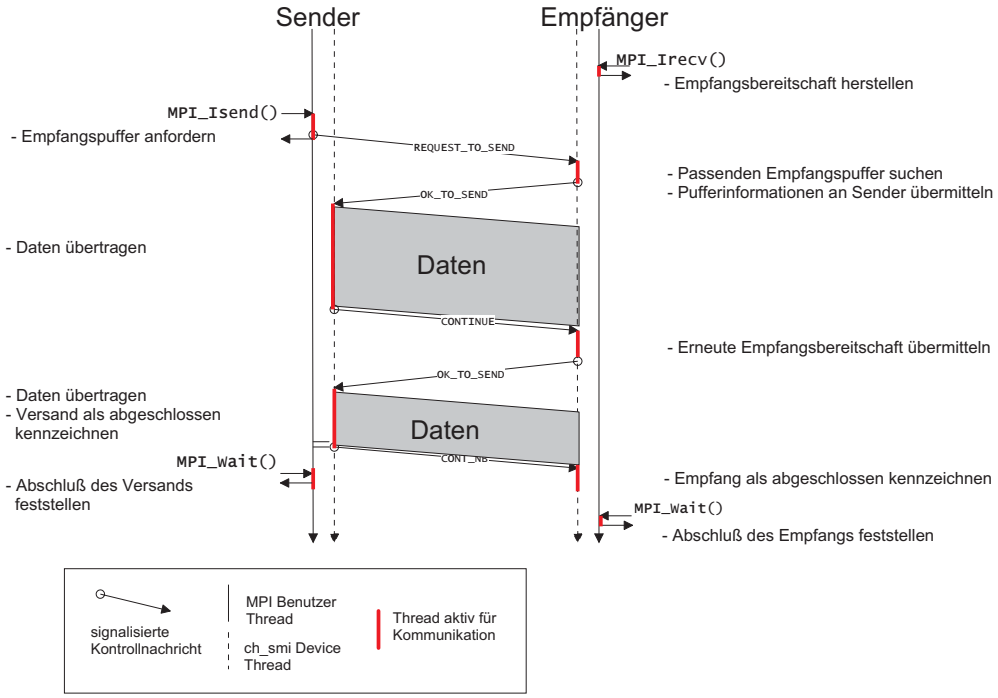


Abbildung 7.10: Prinzip der asynchronen Kommunikation mit Einsatz eines Device-Threads

sen Protokollen findet der Datentransfer auch bei nicht-blockierenden Sendeaufrufen bereits vollständig innerhalb dieses MPI-Aufrufs statt. Dabei hat die eigentliche Datenübertragung einen relativ geringen Anteil an der gesamten Nachrichtenlatenz; die lokale Verwaltung des Sendevorgangs und die Allokation und Freigabe erforderlicher Eingangspuffer hat hingegen einen vergleichsweise hohen Anteil. Dies wird deutlich an der Aufschlüsselung des Zeitbedarfs zum Versand einer *short*-Nachricht von 128 Byte bzw. einer *eager*-Nachricht von 4kB, die in Tabelle 7.5 dargestellt ist. Da zusätzlich die Erzeugung der entfernten Unterbrechung erforderlich ist, ergibt sich für diese Fälle eine Erhöhung der Zeit, die der AT zum Versand der Nachricht in der MPI-Bibliothek verbringen muß. Dieser erhöhten Zeit steht kein Gewinn durch verkürzte Synchronisationspausen gegenüber, da diese beim *short*- und *eager*-Protokoll ohnehin nicht anfallen.

Über die dargestellte fehlende Verkürzung der Sendelatenz hinaus findet auch keine Entla-

***short*-Nachricht 128 Byte**	
Vorgang	**Zeit [μs]**
MPIR/MPID-Durchlauf	3,13
Allokation Eingangspuffer	0,55
Prüfsummenberechnung	0,68
Datentransfer	8,62
Gesamtzeit	**12,98**

***eager*-Nachricht 4kB**	
Vorgang	**Zeit [μs]**
MPIR/MPID-Durchlauf	3,71
Allokation Eingangspuffer	7,89
Datentransfer	35,63
Versand Kontrollnachricht	6,69
Gesamtzeit	**53,92**

Tabelle 7.5: Aufschlüsselung der Anteile einzelner Vorgänge am Versand einer *short*- und *eager*-Nachricht

Transfertyp	128 Byte	4 kB
DMA	35,9μs	47μs
PIO	3,06μs	24μs

Tabelle 7.6: Vergleich von PIO- und DMA-Sendelatenz für direkte SCI-Datentransfers

stung der CPU statt, da die eigentlichen Datentransfers auch bei der asynchronen Kommunikation als PIO-Transfers abgewickelt werden. DMA-Transfers sind bei den relevanten Nachrichtenlängen nicht effizient, wie in Tabelle 7.6 im Vergleich dargestellt ist. Die ermittelten Latenzen für DMA-Transfers kleiner Datenmengen liegen teilweise oberhalb der Abarbeitungszeit der gesamten Nachricht mittels PIO.

Das bedeutet, daß die effektive Nutzung der asynchronen Kommunikation nur mit dem *rendez-vous*-Protokoll möglich ist. Dies ist auch das Protokoll, dessen Nutzung entsprechend der Nachrichtengrößen in einer Applikation am meisten Zeit beansprucht. Insbesondere hat es die beschriebenen Synchronisationsverluste zwischen Sender und Empfänger. Eine Optimierung bezüglich einer Minimierung dieser Verluste ist daher sinnvoll.

7.3.2 Asynchrones rendez-vous-Protokoll

Mit dem vorgestellten Verfahren lassen sich alle Varianten des *rendez-vous*-Protokolls (nichtblockierend und blockierend mit PIO-Übertragung sowie DMA-gestützte Direktübertragung) asynchron betreiben. Da bei aktiver PIO-Übertragung jedoch die Leistung einer CPU gebunden ist, ist asynchrone Kommunikation mit PIO-Übertragung nur für den Fall effizient, wenn dem DT eine eigene CPU zur Verfügung steht. Das ist etwa bei SMP-Knoten der Fall, auf denen weniger Applikationsprozesse ausgeführt werden, als CPUs vorhanden sind. Für den allgemeinen Fall kann dieses Verfahren jedoch nicht effizient eingesetzt werden.

In Kapitel 7.2 wurde bereits die DMA-gestützte Direktübertragung vorgestellt, die hohe Bandbreiten bei niedriger CPU-Last erzielt. Dies sind beides Eigenschaften, die auch für die asynchrone Kommunikation wichtig sind, um eine tatsächlich asynchrone Abwicklung der Kommunikation mit minimaler Beeinflussung des weiterlaufenden ATs zu erreichen. Daher hat diese Variante des *rendez-vous*-Protokolls für die asynchrone Kommunikation eine besondere Bedeutung.

Während die Datenübertragung vom DT durchgeführt wird, kann es dazu kommen, daß der AT bereits auf den Abschluß des Nachrichtenversands bzw. Nachrichtenempfangs wartet. Dies geschieht normalerweise durch Polling auf einer Speichervariablen. Dieses Verfahren belegt die Leistung einer CPU, was den DT bei der Abwicklung des Datentransfers behindert. Daher wurde für diesen Fall statt der beschriebenen Abschlußprüfung mittels Polling eine *blockierende* Variante entwickelt, bei der der AT auf einer Semaphore wartet, die vom DT erst nach Abschluß des zugehörigen Transfers freigegeben wird. Somit verbraucht der AT bei der Abschlußprüfung keine CPU-Zyklen.

Neben den üblichen ungepufferten Sendemodi definiert MPI auch einen *explizit gepufferten* Sendemodus, bei dem eine Nachricht nicht direkt aus dem Benutzerpuffer gesendet wird, sondern in einen lokalen Zwischenpuffer kopiert wird, um dann von dort gesendet zu werden. Der Sendevorgang ist damit aus Sicht des Benutzers nach dem Aufruf der entsprechenden Sendefunktion (`MPI_Bsend` oder `MPI_Ibsend`) abgeschlossen, da sofort wieder auf den Benutzerpuffer zugegriffen werden darf. Dieses Verfahren hat dann Vorteile, wenn die Bandbreite der Übertragung über das Netz deutlich unter der lokalen Kopierbandbreite liegt und die Kommunikation ohne CPU-Mitwirkung im Hintergrund abgewickelt werden kann. Zudem werden da-

mit (für *rendez-vous*-Nachrichten) Synchronisationsverluste vermieden, da nicht auf die Empfangsbereitschaft des Empfängers gewartet werden muß. Der genannte Zwischenpuffer muß vom Benutzer allokiert und an die MPI-Bibliothek übergeben werden. Wenn hierzu, wie in Kapitel 7.2.2 beschrieben, ein physikalisch zusammenhängend allokierter Puffer verwendet wird, kann die asynchrone Übertragung mittels Direktübertragung mit optimaler DMA-Übertragungsleistung erfolgen. Damit wird erreicht, daß für blockierende Sendeaufrufe die effektive Kommunikationsbandbreite so hoch ist wie die lokale Kopierbandbreite und Synchronisationsverluste wie bei der nicht-blockierenden asynchronen Kommunikation vermieden werden.

7.3.3 Überlappung von Kommunikation und Berechnung

Das eigentliche Ziel der asynchronen Kommunikation ist die Überlappung von Kommunikation mit der Ausführung von Berechnungen, wodurch ein möglichst großer Anteil der zur Kommunikation erforderlichen Zeit für die effektive Ausführungszeit der Applikation nicht relevant wird. Die Ausführung von Berechnungen stellt die eigentliche Aufgabe einer MPI-Applikation dar. Hingegen dient die Kommunikation nur dazu, den beteiligten Prozessen die zur Berechnung notwendigen Daten bereitzustellen.

Bei der Betrachtung der angestrebten Überlappung von Kommunikation und Berechnung sind zwei Parameter entscheidend: die Länge D der zu übertragenden Nachricht und die Dauer t_{comp} der auszuführenden Berechnung, die unabhängig von den zu übertragenden Daten ist. Die Zeitdauer t_{cc} zur Ausführung *beider* Operationen (*communication* und *computation*) ist davon abhängig und soll minimiert werden. Der prinzipielle Effekt der Überlappung und der Unterschied, der dabei zwischen synchroner und asynchroner Kommunikation sichtbar wird, ist in Abb. 7.11 dargestellt. Sie zeigt schematisch die Gesamtzeitdauer einer Berechnung der Zeitdauer $t_{comp} \geq 0$ und des nicht-blockierenden Transfers einer Nachricht fester Größe gemäß dem dargestellten Pseudocode. Der Transfer selber hat eine Dauer von $t_{msg} = t_{sync}$ für die synchrone Kommunikation und $t_{msg} = t_{async}$ für die asynchrone Kommunikation. Da die asynchrone Kommunikation in der Regel eine höhere Startlatenz hat, ist in diesem Beispiel der Fall $t_{async} > t_{sync}$ dargestellt. Bei der synchronen Kommunikation ergibt sich die Gesamtzeitdauer t_{cc_sync} aus der Zeitdauer der sequentiellen Ausführung der beiden Operationen *Kommunikation fester Dauer* und *Berechnung variabler Dauer*, so daß die Linie *Berechnung + synchrone Kommunikation* eine Parallele zu *Berechnung* darstellt. Bei der asynchronen Kommunikation ergibt sich ein anderes Bild: die Gesamtzeit t_{cc_async} für *Berechnung und asynchrone Kommunikation* verbleibt zunächst bei t_{async}, um erst zu einem späteren Zeitpunkt linear mit der Berechnungsdauer anzusteigen. Dies bedeutet, daß zunächst nur die Kommunikationszeit die Gesamtzeit bestimmt (die Berechnung wird in diesem Fall vollständig von der Kommunikation verdeckt),

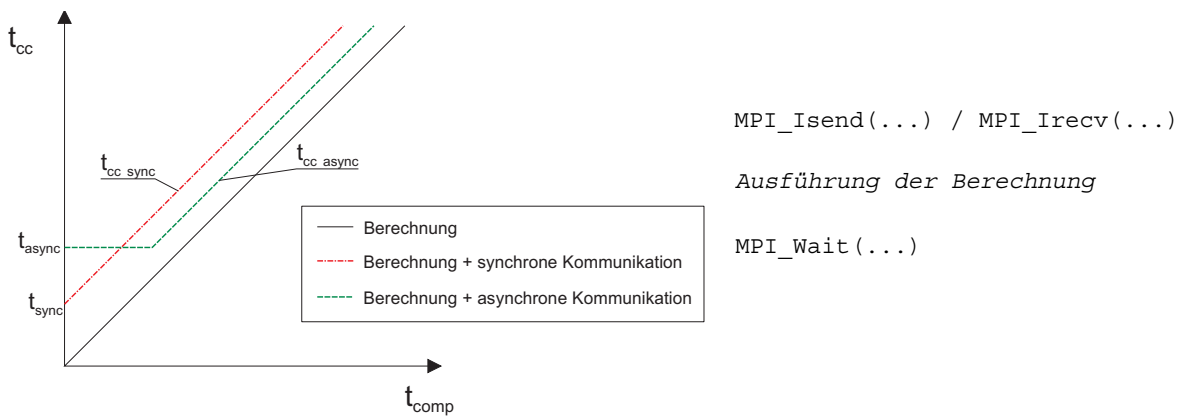


Abbildung 7.11: Unterschiedlicher Effekt der Überlappung von Kommunikation und Berechnung bei synchroner und asynchroner Kommunikation

während im Bereich des linearen Anstiegs die Berechnungszeit über der reinen Datenübertragungszeit der asynchronen Kommunikation liegt. In diesem Bereich kehrt sich die Überlappung um, indem die Berechnung die Kommunikation überlappt. Dies ist der erwünschte Effekt der asynchronen Kommunikation. Jedoch kann diese Überlappung nie vollständig erfolgen, da die lokale Ausführung der nicht-blockierenden Sende- bzw. Empfangsfunktion und der Wartefunktion jeweils synchrone Anteile beinhaltet, die auch bei eigentlich asynchroner Kommunikation sequentiell zur Berechnung ausgeführt werden müssen.

Um das Maß der Überlappung, die durch asynchrone Kommunikation erzielt wird, zu beurteilen, müssen entsprechende Metriken eingeführt werden. Der *Überlappungsgrad* gibt an, welchen Anteil die Berechnung an der Gesamtzeit hat. Ein höherer Anteil für eine feste Berechnungsdauer bedeutet eine bessere Überlappung. Daneben interessiert die *Effizienz* der asynchronen Kommunikation, d.h. welcher Anteil der Kommunikationsoperation tatsächlich asynchron abgearbeitet wird. Je höher dieser Anteil ist, desto effizienter kann die Überlappung aufgrund des entsprechend kleineren verbleibenden synchronen Anteils an der Kommunikation maximal sein. Weiterhin ist von Interesse, wie lang die Berechnung sein muß, bis sie die gegebene asynchrone Kommunikation vollständig überlappt und die gemäß der Effizienz maximale Überlappung eintritt. Dieser Fall, also eine Berechnungsdauer, deren weitere Verlängerung keine bessere Überlappung ergibt, soll als *Sättigung* der asynchronen Kommunikation bezeichnet werden. Die Bestimmung dieser drei Metriken kann anschaulich aus dem in Abb. 7.11 gezeigten Zusammenhängen erfolgen.

Der *Überlappungsgrad* λ ist das Verhältnis von Berechnungszeit zu Gesamtzeit für Berechnung und Kommunikation:

$$(7.3)\quad \lambda = \frac{t_{comp}}{t_{cc}}$$

Angestrebt wird ein Überlappungsgrad möglichst nahe 1. Verschiedene Überlappungsgrade können jedoch nur für gleiche Berechnungszeiten und Nachrichtengrößen verglichen werden, da sonst die Bezugsgrößen nicht übereinstimmen.

Die *Effizienz* $\varepsilon_{overlap}$ ist bestimmt als

$$(7.4)\quad \varepsilon_{overlap} = 1 - \frac{t_{cc_{async}} - t_{comp}}{t_{cc_{sync}} - t_{comp}}$$

und drückt für einen Arbeitspunkt die Effizienz der Überlappung aus, indem sie den erreichten Anteil an der maximal möglichen Überlappung beschreibt. Somit können verschiedene Effizienzwerte verglichen werden, solange sie für eine konstante Berechnungszeit bestimmt wurden. Letztere Randbedingung braucht ebenfalls nicht erfüllt zu sein, wenn die Berechnungszeit oberhalb des *Sättigungspunktes* s liegt. Der Sättigungspunkt gibt diejenige Berechnungszeit an, ab der eine weitere Verlängerung der Berechnungszeit keine weitere Erhöhung der Effizienz mehr ergibt. Seine Bestimmung wird in Abb. 7.12 dargestellt: s ist der kleinste Wert t_{comp}, für den gilt:

$$(7.5)\quad t_{cc}(t_{comp} + \Delta t) > t_{cc}(t_{comp})$$

7.3.4 Evaluierung der implementierten Lösung

Um die implementierten Varianten zur asynchronen Nachrichtenübertragung zu untersuchen, wurde ein synthetischer Benchmark *overlap* geschrieben, der die angestrebten Merkmale der Verfahren untersucht und anhand der eingeführten Metrik quantitative Aussagen zu deren Wirksamkeit erlaubt. Der Aufbau von *overlap* ist in Abb. 7.13 dargestellt. Ein Prozeß empfängt

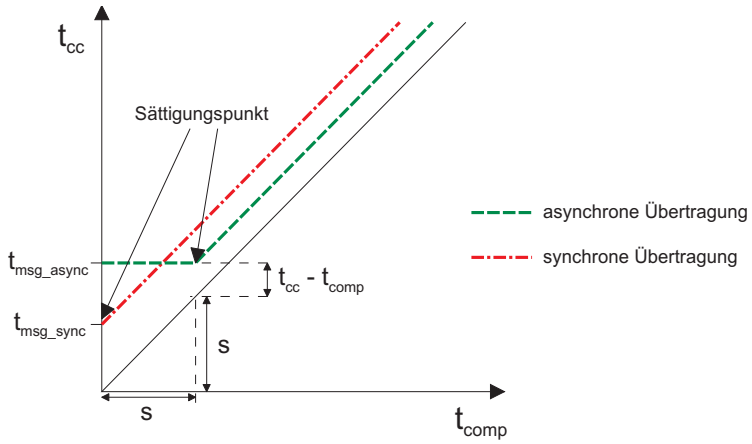


Abbildung 7.12: Bestimmung des Sättigungspunktes *s* bei der Überlappung von Kommunikation und Berechnung

Daten nicht-blockierend, bevor der Sender die Sendeoperation startet und dient so als Datensenke mit minimaler Verzögerung. Der Sender sendet nicht-blockierend eine Nachricht der Größe D_{msg} und beginnt danach mit Berechnungen. Diese können auf zwei verschiedene Arten durchgeführt werden, die in ihren Wirkungen auf den Kommunikationsablauf den beiden wesentlichen Vorgängen entsprechen, die auch in realen Applikationen auftreten:

- ***spin***: Bei der Rechenschleife *spin* werden einzelne Variablen in einer gegebenen Zahl von Iterationen miteinander verknüpft, so daß volle CPU-Belastung, aber keine Belastung des Speichersystems resultiert. Diese Schleife kann auch mit mehreren Threads ausgeführt werden, um auf einem SMP-Knoten mit einem Prozeß eine beliebige Anzahl von CPUs auszulasten.
- ***daxpy***: Bei der Rechenschleife *daxpy* wird im Gegensatz zu *spin* sowohl die CPU als auch das Speichersystem belastet, indem auf zwei Vektoren *x* und *y*, deren Datenmenge jeweils der Größe der versandten Nachricht entspricht, eine Operation der Form $y[i] = A \cdot x[i] + y[i]$ ausgeführt wird.

Mit *spin* läßt sich das Verhalten der asynchronen Kommunikation isoliert auf die Nutzung der CPU beurteilen. Hingegen spiegelt *daxpy* eher die Abläufe in realen Applikationen wider, da ein Rechenaufwand erzeugt wird, der proportional zur Nachrichtenlänge ist und das Speicher-

```
t_cc = MPI_Wtime()
if (sender)
     MPI_Barrier()
     MPI_Isend(msg, D_msg)
```

```
n_iter = vorgegebener_wert
while (n < n_iter)
     spin (with multiple threads)
     n = n + 1
```

```
t_comp = 0
daxpy (D_msg)
t_comp = MPI_Wtime() - t_comp
```

```
     MPI_Wait()
else
     MPI_Irecv(msg, D_msg)
     MPI_Barrier()
     MPI_Wait()
t_cc = MPI_Wtime() - t_cc
```

Abbildung 7.13: Aufbau des *overlap*-Benchmarks (Pseudocode) mit *spin* und *daxpy* Rechenphasen

system belastet.

Mit diesem Benchmark wurden Versuche mit unterschiedlichen Nachrichtenlängen (64 kB und 1 MB), den beiden verfügbaren Typen von Rechenschleifen und mittels DMA-Direktübertragung in reguläre und registrierte SCI-Speichersegmente durchgeführt. Diese Versuche lieferten die gesuchten quantitative Angaben, die die Bestimmung der Effizienz der Protokolle bezüglich der Überlappung von Kommunikation möglich machen. Die Ergebnisse sind grafisch über die Länge der *daxpy*-Schleife in Abb. 7.14 dargestellt. Aus der gleichen Versuchsserie stammen die Daten in Tabelle 7.7, mit denen die Überlappungseffizienz im Bereich der Sättigung bestimmt wird.

Die Entwicklung der Gesamtzeit im Experiment (Abb. 7.14) entspricht dem theoretischen Modell der Überlappung von Kommunikation und Berechnung, wie es in Abb. 7.11 dargestellt ist und mit den nun bestimmten Größen beschrieben werden kann:

$$(7.6)\quad t_{cc} = \begin{cases} t_{msg} & \text{für } t_{comp} > s \\ t_{msg} + (t_{comp} - s) & \text{für } t_{comp} \le s \end{cases}$$

Wenn die Nachrichtenlänge und die Berechnungsaufgabe, und somit t_{msg} und t_{comp}, bekannt sind, kann mittels (7.6) das geeignete Übertragungsprotokoll mit minimalem t_{cc} ausgewählt werden.

Für den Versand eines Vektor von 64 kB Länge liegt der Sättigungspunkt bei Nutzung regulärer SCI-Speichersegmente bei 2×10^3 *daxpy*-Verknüpfungen. Da die Zeit t_{msg} der asynchronen DMA-Direktübertragung in diesem Fall unterhalb der entsprechenden Zeit für synchrone PIO-Übertragung liegt, wird für jede Zahl von *daxpy*-Operationen eine kürzere Gesamtzeit t_{cc} erreicht. Dies trifft bei Nutzung von registrierten SCI-Speichersegmenten erst ab $3,3 \times 10^3$ *daxpy*-Operationen zu. In der Sättigung sparen die asynchronen Verfahren gegenüber dem synchronen Verfahren 0,22 ms bzw. 0,29 ms ein, was eine Effizienz der asynchronen Verfahren von 45,2% bzw. 60,1% bedeutet. Anders ausgedrückt heißt dies, das 54,8% bzw. 39,9% der Übertragungszeit weiterhin wirksam sind, gegenüber 100% beim synchronen Verfahren.

Die Übertragung des gleichen Vektors in Überlappung mit einer *spin*-Berechnung zeigt leichte Unterschiede in den Kennwerten, die auf die etwas längere Ausführungszeit der Rechenschleife zurückzuführen sind. Nur beim synchronen Verfahren steigt jedoch t_{cc} entsprechend an. Die asynchronen Verfahren halten t_{cc} gegenüber der *daxpy*-Berechnung konstant. Dies war abzuse-

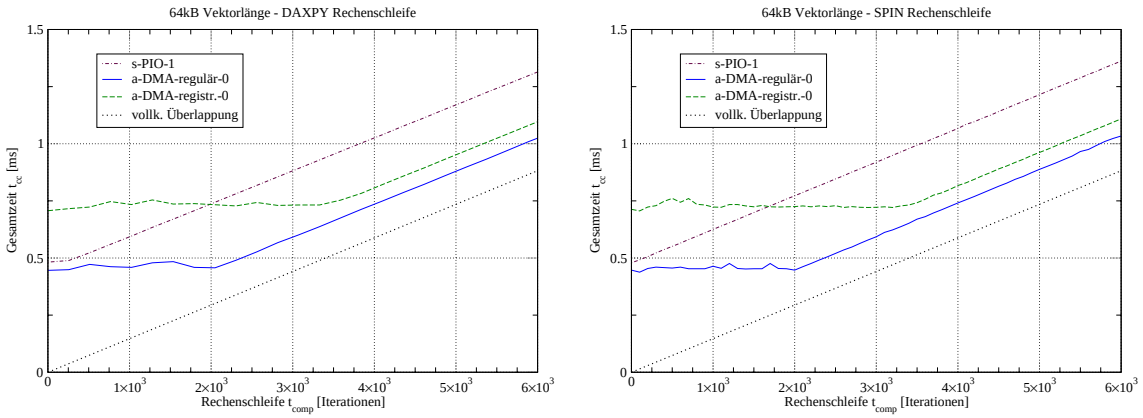


Abbildung 7.14: Überlappung von Kommunikation und Berechnung für unterschiedliche Varianten des *rendez-vous*-Protokolls (*overlap* Benchmark), Nachrichtenlänge 64kB
Links: daxpy-Rechenschleife; *Rechts: spin*-Rechenschleife

Nachricht.-länge	Rechen-schleife	t_{comp}	Protokoll-variante	t_{cc} [ms]	t_{msg} [ms]	s [ms]	$\varepsilon_{overlap}$
64 kB	*daxpy*	4096 Op. entspr. 0,557 ms	a-DMA-0 regulär	0,749	0,446	0,401	60,1%
			a-DMA-0 registriert	0,821	0,707	0.651	45,2%
			s-PIO-1	1,039	0,482	0	0%
64 kB	*spin*	4000 It. entspr. 0,588 ms	a-DMA-0 regulär	0,741	0,447	0,301	68,0%
			a-DMA-0 registriert	0,817	0,713	0,514	52,2%
			s-PIO-1	1,067	0,479	0	0%
1 MB	*daxpy*	100k Op. entspr. 14,149 ms	a-DMA-0 regulär	14,894	4,639	4,065	90,8%
			a-DMA-0 registriert	15,505	5,449	4,065	83,2%
			s-PIO-1	22,209	7,860	0	0%
1 MB	*spin*	100k It. entspr. 14,684 ms	a-DMA-0 regulär	15.104	4,663	4,258	94,5%
			a-DMA-0 registriert	15,699	5,618	4,258	86,7%
			s-PIO-1	22,322	7,638	0	0%

Tabelle 7.7: Überlappungseffizienz für unterschiedliche Kommunikationsfälle

hen, da bei der Vektorlänge von 64kB die *daxpy*-Operationen im Cache ablaufen, so daß das Speichersystem nicht belastet wird. Die Effizienzwerte steigen durch den größeren Wert von t_{comp} entsprechend an.

In Abb. 7.15 ist der analoge Vergleich für die Übertragung eines 1MB langen Vektors dargestellt. Beim Zugriff auf diesen Vektor mittels *daxpy*-Rechenschleife wird aufgrund der Größe des Vektors das Speichersystem belastet. Aber auch hier ist beim Übergang auf die *spin*-Rechenschleife, die wiederum eine längere Ausführungszeit hat, dafür das Speichersystem nicht belastet, sichtbar, daß sich t_{cc} für die asynchronen Verfahren nicht signifikant ändert. Beim synchronen Verfahren fällt die Zunahme von t_{cc} entsprechend der Rechenschleife aus, so daß sich die Effizienz der asynchronen Verfahren weiter verbessert. Insgesamt können diese in der vorgestellten Untersuchung zwischen 83,2% und 94,5% der Rechenzeit mit Kommunikation überlagern.

Das asynchron gesteuerte *rendez-vous*-Protokoll ist zusammen mit der DMA-Direktübertragung von Daten zwischen den Prozessen offensichtlich gut geeignet, das Prinzip der Überlappung von Kommunikation und Berechnung umzusetzen. Der höhere Aufwand zur Vorbereitung und Bereitstellung der Puffer kann durch die CPU-unabhängige Datenübertragung mehr als kompensiert werden. In dieser Evaluierung wurden dabei noch nicht die möglichen weiteren

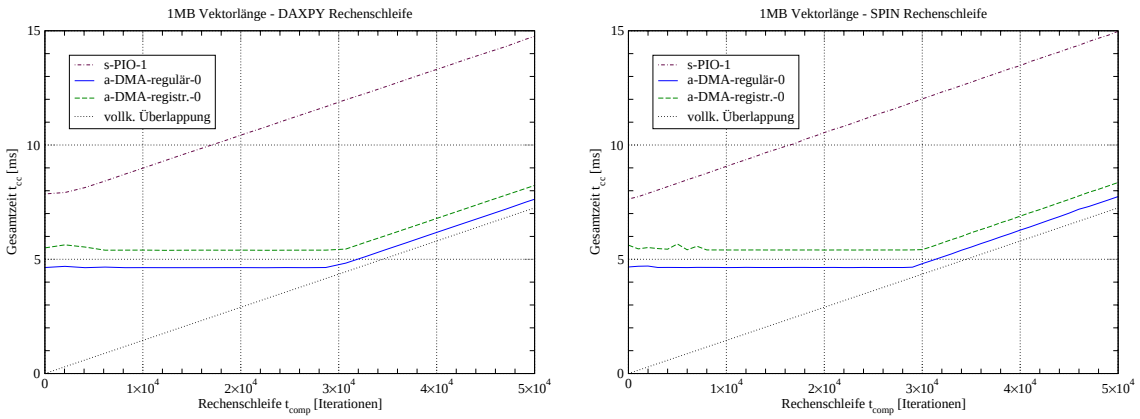


Abbildung 7.15: Überlappung von Kommunikation und Berechnung für unterschiedliche Varianten des *rendez-vous*-Protokolls (*overlap* Benchmark), Nachrichtenlänge 1MB
Links: daxpy-Rechenschleife; *Rechts: spin*-Rechenschleife

Leistungsgewinne in einer Applikation ermittelt, die dadurch entstehen können, daß ein Sendeprozeß nicht auf MPI-Aktivitäten des Empfängerprozesses warten muß, um im Falle der Empfangsbereitschaft dieses Prozesses eine Kommunikation im *rendez-vous*-Protokoll durchzuführen. Dieser Effekt wurde jedoch bereits analog im Zusammenhang mit der Evaluierung der einseitigen Kommunikation in Kapitel 5.2.4 analysiert; die dort gewonnenen Erkenntnisse lassen sich auf diese Situation übertragen.

7.3.5 Einfluß der asynchronen Direktübertragung auf Applikationsleistung

Der Effekt der Überlappung von Kommunikation und Berechnung wurde bereits häufig zur Optimierung der Leistung von Applikationen genutzt. Ein aktuelles Beispiel, dessen Wirksamkeit sich sehr gut auf das in dieser Arbeit vorgestellte Verfahren übertragen läßt, ist in *[114]* beschrieben. In dieser Arbeit wird ein Verfahren vorgestellt, in dem allgemeine Berechnungen über verschachtelte Schleifen parallelisiert werden. Die Implementation des Verfahrens wird auf einem SCI-gekoppelten Cluster evaluiert.

Ein Beispiel für eine mittels dieses Verfahrens parallelisierbare Schleife ist in Abb. 7.16 gegeben. Die Berechnungsschritte in der *n*-dimensionalen Gesamtschleife werden als Indexraum J^n betrachtet und mittels *supernode*-Transformation in *n* unabhängige Gruppen von *Hyperebenen* überführt. Die Berechnung für jeden Schritt sowie das Weiterleiten bzw. Empfangen von Zwischenergebnissen lassen sich überlappen. Dabei kommuniziert jeder Prozeß nur mit einem Nachbarprozeß, was für die Kommunikation via SCI einen günstigen Pipeline-Effekt ergibt. Die Autoren haben dieses Prinzip jedoch nicht als MPI-Programm implementiert[1], sondern direkt mit SISCI-Aufrufen gearbeitet. Die erzielten Leistungssteigerungen für die Gesamtabarbeitungszeit der Schleife liegen für die getesteten Fälle zwischen 33% und 100%. Für eine

```
for (i = 1; i < dim_x; i++)
     for (j = 1; j < dim_y; j++)
          for (k = 1; k < dim_z; k++)
               A[i][j][k] = func(A[i-1][j][k], A[i][j-1][k], A[i][j][k-1]);
```

Abbildung 7.16: Beispiel für eine mittels *supernode*-Transformation parallelisierbare verschachtelte Schleife

1. Die Autoren waren irrigerweise der Annahme, es stünde keine für diese Anwendung geeignete MPI-Implementation zur Verfügung. Eine Umsetzung des Programms auf MPI, die eine Nutzung von SCI-MPICH ermöglicht, war bis zur Erstellung dieser Arbeit noch nicht abgeschlossen.

Implementation dieses Verfahrens als MPI-Programm, das mit SCI-MPICH ausgeführt würde, sind nahezu identische Steigerungsraten zu erwarten, da der Überhang der Ausführung der MPI-Funktionen mit etwa $4\mu s$ gegenüber den direkten SISCI-Aufrufen nur gut 10% der Zeit zur Initiierung eines DMA-Transfers von $35\mu s$ beträgt.

KAPITEL 8

Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurden wesentliche Aspekte des Nachrichtenaustausches gemäß dem *Message Passing Interface (MPI)* Standard auf speichergekoppelten Rechnerverbundsystemen mit SCI-Verbindungsnetz untersucht. Obwohl MPI schon seit einigen Jahren als Standardschnittstelle zum Nachrichtenaustausch in parallelen Applikationen und Bibliotheken im technisch-wissenschaftlichen Bereich etabliert ist, nutzen die Anwender nur zögerlich die Möglichkeiten der Schnittstelle, die über die Grundfunktionalität hinaus gehen. Dies wird etwa darin deutlich, daß die häufig verwendeten und zur Leistungsquantifizierung zitierten Kernel- und Applikationsbenchmarks der *NAS Parallel Benchmark Suite [146]* keine persistente oder einseitige Kommunikation und keine abgeleiteten Datentypen verwenden, keine explizite Überlappung von Kommunikation und Berechnung betreiben, keine Topologiefunktionen verwenden und keine Allokation von Speicher mittels der MPI-Allokationsfunktionen vornehmen.

Daher ist es nicht immer unmittelbar gewinnbringend, aber mittelfristig unabdingbar, daß eine Plattform zur Ausführung von MPI-Programmen die komplette Funktionalität in einer effizienten Implementierung anbietet. Es ist zu erwarten, daß in Zukunft Anwender, die parallele Applikationen oder Bibliotheken erstellen, immer mehr die fortgeschrittenen Möglichkeiten von MPI nutzen werden, die über die minimalistische Nutzung von `MPI_Send` und `MPI_Recv` hinausgehen. Die vorliegende Arbeit beschreibt und evaluiert eine solche Implementierung, die für alle leistungskritischen Bereiche des Nachrichtenaustausches einerseits die Möglichkeiten der zur Verfügung stehenden Hardware optimal zu nutzen, andererseits die gegebenen Limitierungen mit möglichst geringen Einbußen zu umgehen sucht.

Die Voraussetzungen dazu werden in Kapitel 2 für Rechnerverbundsysteme im allgemeinen und in Kapitel 3 für Rechnerverbundsysteme mit Speicherkopplung über SCI im besonderen dargelegt. In Kapitel 4 werden die notwendigen grundlegenden Protokolle zum zweiseitigen Nachrichtenaustausch entwickelt und modelliert, um schließlich Modell und Implementierung in Relation zu setzen. Die speziellen Eigenschaften der Speicherkopplung werden in Kapitel 5 genutzt, um ein verbessertes Verfahren zur Repräsentation und direkten Übertragung nicht zusammenhängender Datentypen zu entwickeln. Daneben werden Verfahren zur effizienten einseitigen Kommunikation vorgestellt, die über die Fähigkeiten der verbreiteten Emulation durch zweiseitige Kommunikation hinausgehen. Kapitel 6 beschreibt die optimale Nutzung der Topologie des SCI-Verbindungsnetzes, der Fähigkeiten der Netzadapter sowie der einseitigen Ablage von Daten und des unmittelbaren Zugriffs auf diese Daten in bereitgestellten Puffern zur Abwicklung kollektiver Operationen. Die vorgestellten Verfahren steigern den Durchsatz des Systems gegenüber der Nutzung generischer Algorithmen deutlich. Die Voraussetzungen zur Überlappung von Kommunikation und Berechnung sowie die damit möglichen Reduzierungen der effektiven Kommunikationszeit um über 90% werden in Kapitel 7 dargestellt.

Das Ziel dieser Arbeit ist in erster Linie, dem Benutzer die maximale Leistung für die Applikation zur Verfügung zu stellen. Genauso wird aber anhand der entwickelten Modelle und der beschriebenen Verfahren deutlich, in welchen Bereichen eine Weiterentwicklung der zugrundeliegenden Hard- und Softwareplattform für den Nachrichtenaustausch am meisten Gewinn bringen wird.

8.1 Gewonnene Erkenntnisse

Die wesentlichen neuen Erkenntnisse dieser Arbeit seien an dieser Stelle zusammengefaßt.

- Die quantitative Analyse der Skalierbarkeit von SCI-Netzen in Torus-Topologie, die über den bereits zuvor analysierten Fall der Alle-an-Alle-Kommunikation hinaus zwei weitere

typische Kommunikationsmuster untersucht, gibt eine obere Grenze der Kommunikationsleistung in SCI-gekoppelten Rechnerverbundsystemen vor. Diese gilt es mit geeigneten Kommunikationsalgorithmen zu einem möglichst hohen Anteil zu erreichen, was in dieser Arbeit theoretisch und praktisch durchgeführt wurde. Die eingeführten Modelle können außerdem dazu verwendet werden, den Einfluß der unterschiedlichen Komponenten in einem SCI-Netz auf die Gesamtleistung zu bestimmen.

- Die Möglichkeiten zur Punkt-zu-Punkt-Kommunikation wurden quantitativ durch Modellbildung ermittelt und die praktisch in den implementierten Protokollen erreichten Werte über ein einfaches, aber aussagefähiges Effizienzschema mit der Zielsetzung in Bezug gesetzt. Dieses Beurteilungsschema, bei dem die reine Kommunikationseffizienz getrennt nach Protokoll- und Implementationseffizienz betrachtet wird, gibt klare Hinweise darauf, an welcher Stelle Änderungen in den Leistungseigenschaften der Knoten- und Netzhardware oder Änderungen im Protokoll eine Steigerung der Kommunikationseffizienz bewirken können. Die Auswirkungen dieser Änderungen können durch die Modellierung unmittelbar quantitativ analysiert werden.
- Die Optimierung der Kommunikation mit nicht-zusammenhängenden Datentypen basiert auf der Abkehr der internen Repräsentation der Datentypen von rekursiven Strukturen oder gar linearen Listen hin zu einer neuartigen kompakten und effizienten stapelorientierten Darstellung. Diese ermöglicht den leistungssteigernden Direktzugriff auf die Elemente des Datentyps. Die sorgfältige Berücksichtigung der SCI-Leistungscharakteristik bereits im Aufbau dieser internen Darstellung ermöglicht eine hohe Kommunikationsleistung auch für ungünstige Datenausrichtung.
- Zur Bewertung der einseitigen Kommunikation wurde eine Klassifikation von Schemata der einseitigen Kommunikation vorgestellt. Deren Berücksichtigung bei der Leistungsevaluation zeigt, daß die bislang verwendeten Verfahren die komplexen Kommunikationsabläufe mit lockererer oder gar nicht vorhandener Kopplung, die dem Prinzip der einseitigen Kommunikation entsprechen, nicht gut wiedergeben. Die Verfahren zur einseitigen Kommunikation in SCI-MPICH bieten auch in diesen Fällen durch asynchronen Betrieb und direkten Speicherzugriff ein hohes Leistungsniveau unter voller Bewahrung der geforderten Synchronisationssemantik.
- Zur Steigerung der Leistungsfähigkeit von kollektiven Operationen wird zunächst die physikalische Topologie der Prozesse im SCI-Netz des Rechnerverbunds automatisch analysiert (SCI bietet hierzu kein Verfahren an). Damit können für Rundsende- und Reduktionsoperationen Pipeline-Verfahren effizient genutzt werden. Während Pipelining als solches ein Grundprinzip zur Leistungssteigerung in Datenverarbeitungssystemen ist, ist die nebenläufige Nutzung von lokalen PIO-Transfers und entfernten DMA-Operationen neuartig und bewirkt insbesondere bei Reduktionsoperationen starke Leistungssteigerungen. Daneben wurde ein hocheffizienter Kommunikationsalgorithmus für die Verteilungsreduktion entwickelt, der von der koordinierten Überlappung von Kommunikation und Berechnung zwischen den Prozessen profitiert.
  Die Überlegenheit der neuartigen Verfahren wurde praktisch im Vergleich mit den generischen Verfahren sowie im direkten Vergleich mit einer optimierten, kommerziellen MPI-Implementation für SCI auf identischer Hardware unter Beweis gestellt.
- Asynchroner Fortschritt von nicht blockierenden Kommunikationsoperationen ist eine bei MPI-Implementationen selten anzutreffende Eigenschaft. Wenn diese Technik verfügbar ist, wird sie durch höhere Latenzen für kurze Nachrichten erkauft. Das neuartige, hybride Eingangsprüfungsverfahren verbindet die niedrige Kommunikationslatenz von SCI-MPICH im Abfragebetrieb (Polling) mit der gewünschten asynchronen Bearbeitung von Kommunikationsoperationen. Dadurch können Berechnungen mit oder ohne Speicherlast zu über 90% mit

Kommunikation überlappt werden.

Neben der Betrachtung der wissenschaftlichen Erkenntnisse ist es wichtig zu erwähnen, daß die im Rahmen dieser Arbeit und im Zusammenhang mit weiteren Projekten am Lehrstuhl *[78]* entstandene Software zum Nachrichtenaustausch, von der SCI-MPICH ein wesentlicher Teil ist, für eine Zahl von Betriebssystemen frei und im Quelltext verfügbar ist[1]. In den vergangenen zwei Jahren wurde die Software bereits von mehr als 1000 Arbeitsgruppen in aller Welt geladen. Dazu zählen Universitäten, Forschungseinrichtungen und Firmen. Die Software muß für den kommerziellen Einsatz lizenziert werden, was ebenfalls bereits wiederholt erfolgte.

Die somit gegebene Offenheit der Forschungsergebnisse dieser Arbeit ermöglicht die unabhängige Verifikation der dargestellten Verfahren und Resultate. Darüberhinaus leistet sie einen Beitrag zur Entwicklung leistungsfähiger und stabiler Middleware zum Nachrichtenaustausch, indem die beschriebenen Verfahren im Detail studiert und verwendet werden können.

8.2 Weitere Entwicklungen

Methoden der numerischen Simulation werden in allen Bereichen der Forschung und Entwicklung im technischen und naturwissenschaftlichen Bereich immer stärker genutzt. Preiswerte parallele Rechensysteme auf Basis von Rechnerverbundsystemen ermöglichen deren unmittelbaren Einsatz auch für kleine Organisationen. Die Kopplung solcher Systeme mit einem SCI-Verbindungsnetz ermöglicht den effizienten Einsatz von kommunikationsintensiven Applikationen, die MPI als Schnittstelle zum Nachrichtenaustausch verwenden. Es ist daher in den kommenden Jahren eine weiter zunehmende Verbreitung der Nutzung von Rechnerverbundsystemen kleiner und mittlerer Größe zu erwarten. Dies wird nicht zwangsläufig zur Ablösung der spezialisierten oder extrem skalierten Systeme führen, die eine höhere Leistung und Zuverlässigkeit zu deutlich höheren Kosten liefern. Der wachsende Markt für paralleles Rechnen und Datenverarbeitung im allgemeinen wird jedoch einen Großteil der Bedürfnisse mit Rechnerverbundsystemen decken können.

Die vorgestellten Verfahren zum Nachrichtenaustausch auf Rechnerverbundsystemen mit SCI-Verbindungsnetz behandeln einen großen Teil dieses Themas. Es ergeben sich jedoch weitere vertikal und horizontal angrenzende Arbeitsgebiete.

Vertikal aufbauend sind weitergehende Untersuchungen und Evaluierungen der vorgestellten Verfahren anhand komplexer, realer Applikationen wünschenswert. Diese können zusätzlich Aufschluß über die Wirkung der vorgestellten Verfahren geben. Die vorliegende Arbeit versucht, diesen Prozeß zu fördern, indem sowohl das Potential zur Leistungssteigerung beim Nachrichtenaustausch als auch der Weg zu seiner Nutzung dargelegt wird. Die vorgestellten fortgeschrittenen Verfahren des Nachrichtenaustauschs mittels MPI werden jedoch bislang von den (frei verfügbaren) relevanten MPI-basierten Applikationen kaum genutzt. Ebenso können Versuche zur Skalierbarkeit auf Rechnerverbundsystemen mit einer größeren Knotenzahl und mehrdimensionaler SCI-Netztopologie im Vergleich mit der Modellierung in Kapitel 3 Hinweise auf weiteres Optimierungspotential geben. Solches Potential ist insbesondere bei den in dieser Arbeit behandelten kollektiven Operationen vorhanden.

Horizontal ergänzende Arbeiten sind etwa im Bereich von paralleler Ein-/Ausgabe sinnvoll. MPI definiert dazu die Schnittstelle MPI-IO, deren Umsetzung auf SCI-gekoppelten Clustern von der transparenten Kopplung der Adreßräume auf Hardwareebene profitieren kann *[32]*. Die erwartete zunehmende Nutzung der gesamten im MPI-Standard beschriebenen Funktionalität erfordert, auch die verbliebenen Kapitel des MPI-2 Standards zu implementieren: *Prozeßmanagement* zur dynamischen Erzeugung und Kopplung von MPI-Prozessen und -Applikationen so-

1. http://www.lfbs.rwth-aachen.de/mp-mpich

wie *erweiterte kollektive Operationen* auf den so erzeugten Interkommunikatoren.

Eine Implementierung des Standards für Echtzeit-Nachrichtenaustausch gemäß MPI/RT ist eine Aufgabe, für deren Umsetzung einige der vorgestellten Verfahren benutzt werden können, insbesondere aus dem Bereich der persistenten Kommunikation sowie der grundlegenden Kommunikationsprotokolle. Zur Erhöhung der Kommunikationsleistung auf SMP-Systemen ist die Parallelschaltung von mehreren PCI-SCI-Adaptern in einem Knoten ein Lösungsweg. Ein geeignetes Verfahren wurde in SCI-MPICH bereits implementiert, konnte aufgrund fehlender Hardwareressourcen jedoch nicht evaluiert werden.

GLOSSAR

Begriffe

Viele Begriffe im Bereich der EDV sind je nach Zusammenhang oder Betrachtungspunkt mehrdeutig. Der in dieser Arbeit verwendete Gebrauch von Bezeichnungen wird hier definiert. Zudem haben sich für viele Vorgänge, Einrichtungen oder Verfahren englische Begriffe auch im Deutschen eingebürgert - diese werden hier aufgeführt, sofern sie verwendet werden. Daneben werden verwendete Abkürzungen und Formelzeichen aufgeführt.

API — Englische Abkürzung für *Application Programming Interface*. Eine definierte Softwareschnittstelle, die zur Programmerstellung genutzt werden kann. Die Schnittstellendefinition umfaßt sowohl syntaktische Funktions- und Typdefinitionen als auch semantische Definitionen, unter welchen Bedingungen und mit welchen übergebenen Parametern ein Funktionsaufruf ein bestimmtes Resultat erwirkt. Ein API wird i.d.R. als Bibliothek implementiert. Ein Programm, das ein bestimmtes API verwendet, ist mit allen Implementationen des APIs quelltextkompatibel.

AUSGANGSPUFFER — Von der MPI-Bibliothek verwalteter Puffer, in dem zu versendende Daten zwischengespeichert werden (etwa zur DMA-Übertragung). Dieser Puffer liegt innerhalb eines SCI-Segments.

BENUTZERSEGMENT — Ein Speicherbereich, der durch Systemfunktionen im privaten Speicher allokiert wurde, oder im statischen Datenbereich eines Prozesses liegt, und durch →Registrierung für die direkte SCI-Kommunikation nutzbar gemacht wurde.

CACHING — (deutsch *Pufferung*) Verwendung eines lokalen Puffers (*Cache*) mit relativ höherer Zugriffsgeschwindigkeit, um Daten aus Bereichen mit relativ geringerer Zugriffsgeschwindigkeit zwischenzulagern. Zugriffe mit zeitlicher oder räumlicher Lokalität werden entsprechend beschleunigt.

EINGANGSPUFFER — Von der MPI-Bibliothek verwalteter Puffer, in den Daten vom Empfänger geschrieben werden. Dieser Puffer liegt innerhalb eines SCI-Segments und wird vom Sender in den eigenen Adreßraum eingeblendet oder (zur DMA-Übertragung) verbunden.

EMPFANGSPUFFER — Durch die Applikation bereitgestellter Puffer, der die zu empfangenden Daten aufnimmt. Dieser Puffer kann wie der →Sendepuffer in beliebigen Speicherbereichen gelegen sein.

KNOTEN — (engl. *node*) In einem Parallelrechner mit verteiltem Speicher: Einheit mit einer oder mehreren →CPUs, Arbeitsspeicher und Anbindung an das interne Kommunikationsnetz. Ggf. auch mit eigenem E/A-System. Auf einem Knoten kann ein oder können mehrere Prozesse einer verteilten Anwendung laufen.

NACHRICHTENAUSTAUSCH — (engl. *message passing*) Das Senden, Empfangen, Lesen oder Schreiben von Daten zwischen Prozessen innerhalb eines Parallelrechners durch Aufruf von System- oder Bibliotheksfunktionen (kein Schreib- oder Lesevorgang in gemeinsamen Speicher).

PARALLELRECHNER — Datenverarbeitungsanlage mit mehr als einer →CPU, auf der ge-

	eignete Programme auf mehr als einer →CPU gleichzeitig verarbeitet werden und ein in Teilprobleme zerlegtes Gesamtproblem bearbeiten.
POLLING	Fortwährende Überprüfung eines Zustandes (Speicherstelle) durch die CPU bis zum Eintritt der erwarteten Zustandsänderung (deutsch: *Abfragebetrieb*).
RECHNERVERBUNDSYSTEM	(engl. *cluster*) Typ eines Parallelrechners, der aus dem Zusammenschluß von autarken, für sich selbst voll funktionsfähigen Rechensystemen mit je einer Instanz eines Betriebssystems pro →Knoten. Oftmals aus industriellen Standardkomponenten preisgünstig aufgebaut.
REGISTRIERUNG	Nachträgliche Nutzbarmachung von bereits allokiertem privatem Speicher zur direkten SCI-Kommunikation (als Ziel und Quelle von DMA-Operationen und Ziel von entfernten PIO-Speicherzugriffen) durch Fixierung der virtuellen Speicherseiten an ihren aktuellen physikalischen Adressen sowie durch Speicherung dieser Adreßinformationen im →SCI-Treiber und Zuweisung einer →SCI-Segmentkennung.
SCI-LINK	Eine geschlossene SCI-Verbindung zwischen zwei →Knoten.
SCI-LINKSEGMENT	Eine Punkt-zu-Punkt-Verbindung zwischen zwei Linkcontrollern (→LC), bestehend aus einer Kabelverbindung zwischen dem Ausgang des einen und dem Eingang des anderen LC.
SCI-NETZ	Die Gesamtheit aller →SCI-Segmente zwischen einer Zahl von →Knoten, wobei jeder Knoten mit jedem anderen Knoten kommunizieren kann.
SCI-RING	Ein geschlossener Ring von Linksegmenten, so daß ein Paket, das von allen angeschlossenen →LCs vom Eingang an den Ausgang zum nächsten →SCI-Segment weitergeleitet wird, wieder beim Absender ankommt.
SCI-SPEICHERSEGMENT	Ein Speicherbereich auf einem Knoten, der durch den SCI-Treiber allokiert wurde. Andere Prozesse auf diesem oder anderen Knoten können sich daran verbinden und es in ihren Adreßraum eingeblenden.
SENDEPUFFER	Durch die Applikation bereitgestellter Puffer, der die zu versendenden Daten enthält. Dieser Puffer liegt in der Regel im privaten Speicher, der ggf. von der MPI-Bibliothek zur SCI-Kommunikation →registriert wird. Er kann aber auch durch die Applikation explizit in einem SCI-Segment allokiert werden.
ÜBERLAPPUNG	(engl. *overlapping*) Nebenläufige Ausführung von Vorgängen innerhalb einer Gesamtaufgabe, i.d.R. von Kommunikation und Berechnung.

Abkürzungen

ATT	**A**ddress **T**ranslation **T**able: Auf dem →PCI-SCI-Adapter integrierte Tabelle mit Einträgen zur Abbildung von ausgehenden (lokalen) physikalischen Speicheradressen auf Speicheradressen im globalen SCI-Adreßraum. Die Größe der Tabelle ist durch die Hardware begrenzt. Ein Eintrag in dieser Tabelle bildet die Adressen eines Speicherbereichs fester Größe (ATT-Seite) ab und wird als ATT-Eintrag bezeichnet.
BS	**B**etriebs**s**ystem: Programmcode zur Steuerung und Verwaltung eines Rechners.
CPU	**C**entral **P**rocessing **U**nit: Zentraler (Mikro-)Prozessor; eine oder mehrere CPUs arbeiten in einem →Knoten.
DMA	**D**irect **M**emory **A**ccess: direkter Speicherzugriff durch →E/A-Gerät ohne →CPU im Datenpfad; entsprechend eine Methode zum Datentransfer.
E/A	**E**in-/**A**usgabe: Übertragung von Daten zwischen Hauptspeicher und anderen Geräten (nicht CPUs)
EES	**E**ntferntes **e**ingeblendetes **S**egment: Auf einem entfernten Knoten angelegtes Segment, auf das nur mittels DMA zugegriffen werden kann.
EVS	**E**ntferntes **v**erbundenes **S**egment: Auf einem entfernten Knoten angelegtes Segment, auf das mittels DMA und PIO zugegriffen werden kann.
FE	**F**ast-**E**thernet (100×10^6 Bit/s physikalische Datenrate)
GE	**G**igabit-**E**thernet (1000×10^6 Bit/s physikalische Datenrate)
ID	*engl. Identifier*: numerische Kennung eines Objekts
IRM	**I**nterconnect **R**esource **M**anager: SCI-Kerneltreiber
LS	**L**okales **S**egment: Auf dem lokalen Knoten (aus der Sicht eines Prozesses) angelegtes Segment.
LC	**L**ink**c**ontroller: Integrierter Schaltkreis, der den Zugriff auf eine physikalischen →SCI-Verbindung (bestehend aus Ein- und Ausgang) steuert.
LRS	**L**okales **r**egistriertes **S**egment (entspricht dem →Benutzersegment)
MMU	**M**emory **M**anagement **U**nit: Abbildung von virtuellen auf physikalische Adressen (notwendig für die Implementation von virtuellem Speicher); inzwischen in die →CPU integrierte Einheit, die entsprechend umfangreiche Tabellen verwaltet.
NUMA	**N**on **U**niform **M**emory **A**ccess - →CPUs des →SMP-System erfahren eine uneinheitliche Zugriffscharakteristik auf verschiedene Bereiche des gemeinsamen Hauptspeichers
CC-NUMA	wie →NUMA, jedoch wird für den gesamten gemeinsame Spei-

	cher des Systems Cache-Kohärenz gewährleistet ("**c**ache-**c**oherent")
NZS	**N**icht-**z**usammenhängender **S**peicherbereich
PC	**P**ersonal **C**omputer: Ursprünglich ein Rechensystem, daß an einem Arbeitsplatz einem Nutzer zur exklusiven Nutzung zur Verfügung steht. In dieser Arbeit soll damit die Technologieplattform eines Rechensystems gemeint sein, die in großen Stückzahlen zu geringen Kosten produziert wird und in unterschiedlicher Ausführung, aber basierend auf der gleichen Architektur und unter Nutzung gleichartiger Baugruppen sowohl von Endkunden und Endverbrauchern erworben und benutzt, als auch im gewerblichen und wissenschaftlichen Bereich eingesetzt wird. PCs dienen als Basis für die Integration von Rechnerverbundsystemen (→ Clustern). In ihrer Nutzung unterscheiden sie sich nicht mehr von → Workstations, da sie in den meisten Bereichen über ähnliche (oder bessere) Leistungseigenschaften verfügen.
PIO	**P**rogrammed **I**/**O**: Abwicklung von Ein-/Ausgabe-Operationen durch Lade-/Speicheroperationen der → CPU; entsprechend eine Methode zum Datentransfer über Ein-/Ausgabeadressen.
PSA	**P**CI-**S**CI-**A**dapter
PSB	**P**CI-**S**CI **B**ridge: Integrierter Schaltkreis, der → PCI-Transaktionen in → SCI-Transaktionen umsetzt (und umgekehrt). Enthält daneben weitere Baugruppen wie DMA-Steuerung und Adreßumsetzung.
SCI	**S**calable **C**oherent **I**nterface
SISCI	**S**oftware **I**nfrastructure for **SCI**: SISCI-Kerneltreiber (oberhalb des → IRM) und SISCI-API Bibliothek
SMP	**S**ymetric **M**ulti-**P**rocessing: Das Vorhandensein bzw. die Nutzung von mehren → CPUs in einem System, die über einen gemeinsamen Kommunikationskanal (i.d.R. Bus oder Kreuzschienenverteiler) auf Hauptspeicher und → E/A-Geräte zugreifen. Jede CPU ist dabei gleich an das System angebunden und kann hard- wie softwaremäßig die gleichen Aufgaben übernehmen, so daß eine Symmetrie zwischen den CPUs vorliegt.
SVS	**S**peichergekoppeltes Rechner**v**erbund**s**ystem
UMA	**U**niform **M**emory **A**ccess - Alle → CPUs des → SMP-Systems erfahren die gleiche Zugriffscharakteristik auf dem gesamten gemeinsamen Hauptspeicher
UP	**U**rsprungs**p**rozeß: Bei einseitiger Kommunikation ruft dieser Prozeß die Kommunikationsoperation auf.
ZP	**Z**iel**p**rozeß: Bei einseitiger Kommunikation ist dieser Prozeß (bzw. das Window in seinem Adreßraum) Ziel der Kommunikationsoperation des → UP

Rechenvorschriften und Formelsymbole

GB, GB, MB, KB, MB, KB	Gigabyte, Gigabit, Megabyte, Kilobyte, Megabit, Kilobit Alle Größenangaben in Verbindung mit Byte und Bit sind in dieser Arbeit zur Basis 2 gegeben, d.h. GB = 2^{30} Byte, MB = 2^{20} Byte, kB = 2^{10} Byte, analog für Bit
n/m	*Ganzzahldivision* für natürliche Zahlen*:* Ergebnis ist das maximale *j,* für das gilt: $m \cdot j \leq n$
$n \lozenge m$	*modulo*-Operation für natürliche Zahlen: Ergebnis ist der Rest der Ganzzahldivision n/m, also $n - m \cdot j$.
$ld(N)$	Logarithmus von *N* zur Basis 2: $ld(N) = \frac{\log N}{\log 2}$
$max(a, b)$	Maximum der Werte *a* und *b*
$min(a, b)$	Minimum der Werte *a* und *b*
B	Bandbreite (allgemein)
B_{slink}	Verfügbare Bandbreite auf der SCI-Linkebene
B_{sgmt}	Genutzte Bandbreite eines SCI-Linksegments
B_{blink}	Bandbreite des B-LINKS
B_{int}	Interne Bandbreite eines Knotens zwischen Hauptspeicher bzw. CPU und dem PSA
B_i	Injektionsbandbreite (eingespeiste Bandbreite) eines Knotens
L	Latenz (allgemein)
N, P, Q	Zahl der Prozesse mit $N = P + Q$ und $ld(P) \in \aleph$
K	Zahl der Knoten
k	Prozeßdichte (Zahl der Prozesse pro Knoten)
D	Datenmenge / Nachrichtengröße (Byte)
S	Zahl der SCI-Linksegmente
l	Zahl der SCI-Linkcontroller pro Knoten
d	Zahl der Dimensionen eines SCI-Netzes in Torus-Topologie
g	Grad (Zahl der Knoten) jeder Dimension
n	Ganzzahlfaktor

LITERATURVERZEICHNIS

Cluster

Entwicklung und Status.

[1] Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, Charles V. Packer: *Beowulf: A Parallel Workstation for Scientific Computation.* In Proc. 24th International Conference on Parallel Processing (ICPP), pp.11-14, 1995.
http://citeseer.nj.nec.com/sterling95beowulf.html

[2] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team: *A case for NOW (Networks of Workstations).* IEEE Micro, Vol.15 (1), January 1995.
http://now.cs.berkeley.edu

[3] N.N.: *Chiba City, the Argonne Scalable Cluster.* Mathematics and Computer Science Division, Argonne National Lab, Chicago, July 2000.
http://www-unix.mcs.anl.gov/chiba

[4] Ron Brightwell, Lee Ann Fisk, David S. Greenberg, Tramm Hudson, Mike Levenhagen, Arthur B. Maccabe, and Rolf Riesen: *Massively Parallel Computing Using Commodity Components.* Parallel Computing, Vol.26 (2-3), February 2000.
http://www.cs.sandia.gov/cplant

[5] Mark Baker (Ed.): *Cluster Computing White Paper.* Special Issue of The International Journal of High Performance Computing Applications, Vol.15 (2), Sage Science Press, Thousand Oaks, July 2001.

Kommunikationsmodelle.

[6] Th. v. Eicken, A. Basu, V. Buch, W. Vogels: *U-Net: A User-Level Network Interface for Parallel and Distributed Computing.* In Proc. 15th ACM Symposium on Operating System Principles. Copper Mountain, 3.-6. December 1995.
http://www.cs.cornell.edu/tve/u-net

[7] Matt Welsh, Anindya Basu, and Thorsten von Eicken: *ATM and Fast Ethernet Network Interfaces for User-level Communication.* In Proc. Third International Symposium on High Performance Computer Architecture (HPCA), San Antonio, February 1997.

[8] Cezary Dubnicki, Angelos Bilas, Yuqun Chen, Stefanos N. Damianakis, Kai Li: *Shrimp Project Update: Myrinet Communication.* IEEE Micro Vol.18 (1), pp. 50-52, January/February 1998.

[9] Compaq, Intel and Microsoft Corporations. *The Virtual Interface Specification. Version 1.0*, December 1997.
http://www.viarch.org

[10] Ron Brightwell and Arthur B. Maccabe: *Scalability Limitations of VIA-Based Technologies in Supporting MPI.* In Proc. Fourth MPI Developer's and User's Conference, March 2000.
ftp://ftp.cs.sandia.gov/pub/papers/bright/via-cplant.pdf

[11] N.N.: *M-VIA: A High Performance Modular VIA for Linux.* National Energy Research Scientific Computing Center, Berkeley, August 2002.
http://www.nersc.gov/research/FTG/via/

[12] G. Ciaccio: *Optimal Communication Performance on Fast Ethernet with GAMMA.* In Proc. PC-NOW'98 (International Workshop on Personal Computers based Networks Of Workstations, in conjunction with IPPS/SPDP 1998), Orlando, LNCS 1388, Springer-Verlag Berlin Heidelberg, April 1998.
http://www.disi.unige.it/project/gamma

[13] Karim Ghouas, Knut Omang, Hakon Bugge: *VIA over SCI - Consequences of a Zero Copy Implementation, and Comparison with VIA over Myrinet.* In Proc. Workshop on Communication Architecture for Clusters 2001, in conjunction with Int'l Parallel and Distributed Processing Symposium (IPDPS '01), San Francisco, April 2001.
http://www.ifi.uio.no/~knuto/Publications

E/A-Systeme und Verbindungsnetze.

[14] PCI Special Interest Group: *PCI 2.3: An Evolution of the Conventional PCI Specification.* PCI-Standard, Version 2.3.
http://www.pcisig.com

[15] PCI Special Interest Group: *PCI-X 2.0: The Next Generation of Backward Compatible PCI.* PCI-X-Standard, Version 2.0.
http://www.pcisig.com

[16] ISSD Technology Communications: *PCI-X: An Evolution of the PCI Bus.* Compaq Computer Coorporation, September 1999.
http://www.compaq.com/support/techpubs/whitepapers/tc990903tb.html

[17] Alan Goodrum: *PCI-X Architecture Tutorial.* Applied Computing Conference 2000, May 2000.

[18] IEEE 802.3 Working Group *IEEE 802-2001® IEEE Standards for Local and Metropolitan Area Networks: Overview and Architecture*. The Institute of Electrical and Electronics Engineers, Inc., 2001.

[19] IEEE 802.3 Working Group : *IEEE Std 802.3z-1998, Gigabit Ethernet.* The Institute of Electrical and Electronics Engineers, Inc., 1998.
http://grouper.ieee.org/groups/802/3

[20] H.G. Dietz and T.I. Mattox: *KLAT2's Flat Neighborhood Network.* In Proc. Extreme Linux track of the 4th Annual Linux Showcase (ALS2000), Atlanta, October 2000.
http://aggregate.org/FNN/

[21] José Duato: *Interconnection Networks.* Morgan Kaufmann Publishers Revised Printing, July 2002.

[22] R. Gillet: *MEMORY CHANNEL Network for PCI: An Optimized Cluster Interconnect.* IEEE Micro, Vol.16 (1), pp.12-18, February 1996.

[23] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, W. Su: *Myrinet - a gigabit-per-second local-area network.* IEEE Micro, Vol.15 (1), pp.26-29, February 1995.

[24] Fabrizio Petrini, Wu-chun Feng, Adolfy Hoisie, Salvador Coll, Eitan Frachtenberg: *The Quadrics Network: High-Performance Clustering Technology.* IEEE Micro, 22(2):46-57, February 2002.

[25] IEEE Std. 1596-1992: *IEEE Standard for Scalable Coherent Interface (SCI).* The Institute of Electrical and Electronics Engineers, Inc., 1992.

[26] Alan Beck: *Dave Gustavson Answers Questions About SCI.* Interview, HPCwire, Articles 10249, 10282, 10316, April 1996.
http://www.hpcwire.com

[27] Marius Christian Liaanen, Hugo Kohmann: *Dolphin SCI Adapter Cards.* In: H.Hellwagner, A. Reinefeld (Eds.): *SCI: Architecture and Software for High-Performance Compute Clusters.* LNCS 1734, pp.71-86, Springer-Verlag Berlin Heidelberg, 1999.

[28] N.N: *PCI-SCI Bridge Function Specification.* Dolphin Interconnect Solutions AS, Oslo, November 1996.

[29] N.N.: *PCI-SCI Bridge Specification - D663-PSB.* Dolphin Interconnect Solutions AS, Oslo, October 1998.

[30] N.N.: *PSB66 Specification - D667.* Version 0.90, Dolphin Interconnect Solutions AS, Oslo, February 2000.

[31] N.N.: *LinkController3^{TM} Specification - D666-LC-3.* Version 0.79, Dolphin Interconnect Solutions AS, Oslo, March 2000.

[32] Jorgen S. Hansen: *I/O Buffer Management for Shared Storage Devices in SCI-based Clusters of Workstations.* In Proc. 4th International Conference on SCI-based Technology and Research (SCI-Europe 2001), Trinity College Dublin, October 2001.
http://wwwbode.cs.tum.edu/events/scieurope2001

Systemarchitektur.

[33] N.N.: *Pentium III Architecture.* Intel Inc., January 2002.
http://www.intel.com/products/desk_lap/processors/desktop/pentiumiii

[34] N.N.: *Serverset Chipset Architecture.* Serverworks Inc., January 2002.
http://www.serverworks.com/products/LE.html

[35] N.N.: *Mainboard 370DLE Manual.* SuperMicro Inc., January 2002.
http://www.supermicro.com/PRODUCT/MotherBoards/RCC_LE/370dle.htm

[36] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge: *High-Performance DRAMs in Workstation Environments.* IEEE Transactions on Computers, Vol.50 (11), November 2001.

[37] Michael Eberl, Hermann Hellwagner, Bjarne Herland, Martin Schulz: *SISCI - Implementing a Standard Software Infrastructure on an SCI Cluster.* Tagungsband 1. Workshop Cluster Computing, TU-Chemnitz, November 1997.

[38] ESPRIT HPCN Project EP23174: *Standard Software Infrastructure for SCI based parallel systems - SISCI*, April 1999 - April 2001.
http://www.cordis.lu/esprit/src/23174.htm

[39] Dolphin Interconnect Solutions AS: *SISCI API Specification.* Version 1.10.4, Oslo, April 2001.

[40] Hakon Bugge: *Affordable Scalibility using Multicubes.* In Proc. SCI-Europe ’98 / EMMSEC ’98, Bordeaux, September 1998.

[41] Justin Rattner: *A Low-Overhead Communication Standard for Volume Server Clusters.* Presentation, Server Architecture Lab, Intel Inc., 1997.

[42] Infiniband Trade Association: *Infiniband Architecture Release 1.0.* Volume 1: General Specifications, October 2000.
http://www.infinibandta.org

[43] Sun Microsystems, Inc.: *Sun Cluster 3 Architecture - A Technical Overview.* Palo Alto, December 2000.
http://www.sun.com/clusters

[44] Ch. Kurmann and T. Stricker: *Characterizing memory system performance for local and remote accesses in high end SMPs, low end SMPs and clusters of SMPs.* In Proc. 7th Workshop on Scalable Memory Multiprocessors held in conjunction with the 25th Annual International Symposium on Computer Architecture (ISCA98), Barcelona, June 1998.

[45] Alan Charlesworth: *The Sun Fireplane SMP Interconnect in the Sun Fire 3800 - 6800.* Technical Whitepaper, Sun Microsystems, Inc., 2001.

[46] N.N.: *The IBM NUMA-Q enterprise server architecture.* Technical Whitepaper, International Business Machines Coorporation, January 2000.

[47] N.N.: *Why NUMA-Q for business intelligence?* Technical Whitepaper, International Business Machines Coorporation, April 2000.

[48] Marcus Dormans, Walter Sprangers, Hubert Ertl, Thomas Bemmerl: *A Programming Interface for NUMA Shared-Memory Clusters.* In Proc. High Performance Computing and Networking (HPCN), pp.698 - 707, LNCS 1225, Springer-Verlag Berlin Heidelberg, 1997.

[49] Marcus Dormans, Karsten Scholtyssik, Thomas Bemmerl: *A Shared Memory Programming Interface for SCI Clusters.* In: H. Hellwagner, A. Reinefeld (Hrsg.): SCI-Based Cluster Computing, Springer-Verlag Berlin Heidelberg, 1999.

[50] M. Dormans, S. Lankes, Th. Bemmerl, G. Bolz, E. Pfeiffle: *Parallelization of an Airline Flight-Scheduling Module on a SCI-Coupled NUMA Shared Memory Cluster.* In Proc. High Performance Computing Systems and Applications (HPCS), Kingston, 1999.

[51] Marcus Dormans: *Shared-Memory Parallelization of the GROMOS96 Molecular Dynamics Code on a SCI-Coupled NUMA Cluster.* In Proc. SCI Europe 98, pp.111 - 119, Bordeaux, September 1998.

[52] Friedrich Seifert, Joachim Worringen, Wolfgang Rehm: *Using Arbitrary Memory Regions for SCI Communication.* In Proc. SCI Europe 2001, Dublin, October 2001.
http://wwwbode.cs.tum.edu/events/scieurope2001/

[53] Martin Schulz: *Shared Memory Programming on NUMA-based Clusters using a General and Open Hybrid Hardware / Software Approach.* Dissertation, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik, Technische Universität München, Juni 2001.
http://wwwbode.cs.tum.edu/~schulzm/

[54] N.N.: *OpenMP C and C++ Application Program Interface.* OpenMP Architecture Review Board, Version 2.0, March 2002.
http://www.openmp.org

[55] Sven M. Paas, Karsten Scholtyssik: *Efficient Distributed Synchronization within an all-software DSM system for clustered PCs.* Tagungsband 1st Workshop Cluster-Computing, TU Chemnitz, November 1997.

[56] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwanepoel: *TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems.* In Proc. Winter 94 Usenix Conference, pp.115-131, January 1994.

[57] Rudolf Berrendorf, Michael Gerndt, Andreas Krumme, Selcuk Özmen: *SVM-Fortran: Eine Programmierumgebung für massiv-parallele Rechner.* Interner Bericht KFA-ZAM-IB-9623, Zentralinstitut für angewandte Mathematik, Jülich, September 1996.

[58] Marcus Dormans und Joachim Worringen: *SMI Shared Memory Interface - User & Reference Manual.* Technischer Bericht, Lehrstuhl für Betriebssysteme, RWTH Aachen, 2001.
http:://www.lfbs.rwth-aachen.de/users/joachim/SMI

[59] Mario Trams, Wolfgang Rehm: *A new generic and reconfigurable PCI-SCI bridge.* In Proc. SCI Europe '99, helt in conjunction with EuroPar '99, pp.3-11, Toulouse, September 1999.
http://wwwbode.cs.tum.edu/events/sci-europe99

[60] Friedrich Seifert, Wolfgang Rehm: *Reliably Locking System V Shared Memory for User Level Communication in Linux.* In Proc. IEEE International Conference on Cluster Computing (CLUSTER '01), IEEE Computer Society Press, October 2001.

Nachrichtenaustausch

MPI-Standard.

[61] Rolf Hempel, David W. Walker: *The Emergence of the MPI Message Passing Standard for Parallel Computing.* Computer Standards & Interfaces, Vol.21, pp.51-62, 1999.

[62] David W. Walker, Jack J. Dongarra: *MPI: A Standard Message Passing Interface.* Supercomputer, Vol.12 (1), pp.56-68, January 1996.

[63] Message Passing Interface Forum: *MPI: A message-passing interface standard.* International Journal of Supercomputing Applications, Vol.8 (3/4), 1994.

[64] Message Passing Interface Forum: *MPI-2 Standard.*
http://www.mpi-forum.org/docs/docs.html

[65] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker and Jack Dongarra: *MPI - The Complete Reference.* Volume 1, The MPI Core, 2nd edition, MIT Press, Cambridge, 1998.
http://www.netlib.org/utk/papers/mpi-book/mpi-book.html

[66] Rolf Rabenseifner: *Automatic Profiling of MPI Applications with Hardware Performance Counters.* In Proc. EuroPVM/MPI '99, LNCS 1697, pp.35-42, Springer-Verlag Berlin Heidelberg, September 1999.
http://www.hlrs.de/people/rabenseifner

[67] N.N.: *Document for the Real-Time Message Passing Interface (MPI/RT 1.1).* Real-Time Message Passing Interface (MPI/RT) Forum, December 2001.
http://www.mpirt.org/MPIRT

[68] Graham E. Fagg, Antonin Bukovsky, Jack J. Dongarra: *HARNESS and fault tolerant MPI.* Parallel Computing, 11(27):1479-1495, Elsevier Science B.V., October 2001.
http://icl.cs.utk.edu

[69] N.N.: *IMPI: Interoperable Message-Passing Interface*. Protocol Version 0.0, IMPI Steering Committee, January 2000.
http://impi.nist.gov/IMPI

[70] William L. George, John G. Hagedorn, and Judith E. Devaney: *IMPI: Making MPI Interoperable.* Journal of Research of the National Institute of Standards and Technology, Vol. 3 (105), May/June 2000.
http://nvl.nist.gov/pub/nistpubs/jres/105/3/cnt105-3.htm

MPI-Implementationen.

[71] W. Gropp, E. Lusk, N. Doss and A. Skjellum: *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard.* Parallel Computing, Vol.22 (6), pp.789-828, September 1996.

[72] MPICH Developers: *MPICH - A portable MPI Implementation.* Mathematics and Computer Science Division, Argonne National Laboratory, Chicago, 2002.
http://www-unix.mcs.anl.gov/mpi/mpich

[73] William Gropp and Ewing Lusk: *Sowing MPICH: A Case Study in the Dissemination of a Portable Environment for Parallel Scientific Computing.* The International Journal of Supercomputer Applications and High Performance Computing, Sage Science Press, Summer 1997.

[74] Ewing Lusk and William Gropp: *The implementation of the second generation MPICH ADI.* MPICH working note (draft), Mathematics and Computer Science Division, Argonne National Laboratory, Chicago, 1997.
http://www.mcs.anl.gov/mpi/mpich/workingnote/adi2impl/note.html

[75] Ewing Lusk and William Gropp: *Creating a new MPICH device using the channel interface.* Technical Report ANL/MCS-TM-213, Mathematics and Computer Science Division, Argonne National Laboratory, Chicago, 1995.

[76] William Gropp, Ewing Lusk: *MPICH Abstract Device Interface*. Version 3.3, Reference Manual, Mathematics and Computer Science Division, Argonne National Laboratory, Chicago, August 2002.
http://www-unix.mcs.anl.gov/mpi/mpich/adi3

[77] W.E. Nagel, A. Arnold, M. Weber, H.C. Hoppe and K.Solchenbach: "*VAMPIR: Visualization and Analysis of MPI Resources.* Supercomputer Vol. 12 (1), pp.69-80, 1996
http://citeseer.nj.nec.com/nagel96vampir.html

[78] Joachim Worringen, Karsten Scholtyssik, Silke Schuch, Martin Pöppe, Thomas Bemmerl: *MP-MPICH - Multi-Platform MPI.* Lehrstuhl für Betriebssysteme, RWTH Aachen, 2002.
http://www.lfbs.rwth-aachen.de/mp-mpich

[79] Sven Schindler: *Entwurf und Implementierung eines ADI-2 Multidevices.* Diplomarbeit, TU Chemnitz, Fakultät für Informatik, Professur für Rechnerarchitektur, Februar 1999.

[80] Genias Software GmbH: *Parallel Tools Environment on NT.* (Stand: August 2000).
http://www.genias.de/products/patent

[81] L.P. Huse, K. Omang, H. Bugge, H. Ry, A.T. Haugsdal, E. Rustad: *ScaMPI - Design and Implementation.* In: H.Hellwagner, A. Reinefeld (Hrsg.): *SCI: Architecture and Software for High-Performance Compute Clusters.* LNCS 1734, pp.249-261, Springer-Verlag Berlin Heidelberg, 1999.

[82] Scali AS: *ScaMPI Users Guide.* Oslo, Version 2.1, 2000.
http://www.scali.com

[83] A. Gordon Smith: *CRI/EPCC MPI for T3D - Implementation Overview.* Edinburgh Parallel Computing Centre (EPCC), The University of Edinburgh, June 1995.

[84] Mohammad Banikazemi, Rama K. Govindaraju, Robert Blackmore, and Dhabaleswar Panda: *MPI-LAPI: An Efficient Implementation of MPI for IBM RS/6000 SP Systems.* IEEE Transactions on Parallel and Distributed Systems, Vol.12 (10), October 2001.

[85] James V. Lawton, John J. Brosnan, Morgan P. Doyle, Seosamh D. Ó Riordáin, Timothy G. Reddin: *Building a High-performance Message-Passing System for MEMORY CHANNEL Clusters.* Digital Technical Journal, Vol.8 (2), Digital Equipment Coorporation, 1996.

[86] David Sitsky, David Walsh, and Chris Johnson: *An efficient implementation of the message passing interface (MPI) on the Fujitsu AP1000.* In Mitsuo Ishii (Eds), Proc. of the Third Parallel Computing Workshop, Kawasaki, November 1994.
http://cap.anu.edu.au/cap/projects/mpi/mpi.html

[87] Rossen Dimitrov and Anthony Skjellum: *Efficient MPI for Virtual Interface (VI) Archtitecture.* In Proc. IPDS '99, Las Vegas, 1999.

[88] Rossen Dimitrov and Anthony Skjellum: *An Efficient MPI Implementation for Virtual Interface (VI) Architecture-Enabled Cluster Computing.* MPI Software Technology, Inc., 1999.
http://www.mpi-softtech.com

[89] Joachim Worringen and Thomas Bemmerl: *MPICH for SCI-Connected Clusters.* In Proc. SCI-Europe 99 (Conference Stream of Euro-Par 99), pp.3-11, Toulouse, August 1999.
http://wwwbode.in.tum.de/events/sci-europe99/proceedings
http://www.lfbs.rwth-aachen.de/users/joachim/SCI-MPICH

[90] Hendrik Meyer: *Optimierung und Bewertung eines Nachrichtenprotokolls für speichergekoppelte PC-Cluster.* Studienarbeit, Lehrstuhl für Betriebssysteme, RWTH Aachen, September 2001.

[91] Jörg Henrichs: *Programmierkonzepte und Lastverteilungsstrategien für heterogene Parallelrechnersysteme.* Forschungszentrum Jülich, Jül-3676, Juli 1999.

Einseitige Kommunikation.

[92] Jarek Nieplocha, Jialin Ju, Edoardo Apra: *One-Sided Communication on the Myrinet-based SMP Clusters using the GM Message-Passing Library.* In Proc. Workshop on Communication Architecture in Clusters (CAC '01), in conjunction with the International Parallel and Distributed Processing Symposium (IPPDS 2001), IEEE Computer Society Press, 2001.

[93] Private Kommunikation mit Jarek Nieplocha (Pacific Northwest Laboratory, USA), Februar 2002.

[94] N.N.: *Cray SHMEM Programming Interface.* Cray T3E™ Fortran Optimization Guide, Cray Inc., Document Nbr. 004-2518-002.
http://www.cray.com/craydoc/

[95] Louis A. Giannini and Andrew A. Chien: *A Software Architecture for Global Address Space Communication on Clusters: Put/Get on Fast Messages.* In Proc. 7th IEEE Int. Symposium on High Performance Distributed Computing (HPDC), pp.330-339, 1998.

[96] Glenn R. Luecke, Silvia Spanoyannis, Marina Kraeva: *The Performance and Scalability of SHMEM and MPI-2 One-Sided Routines on a SGI Origin 2000 and a Cray T3E-600.* Journal of Performance Evaluation and Modelling for Computer Systems (PEMCS), December 2002.

[97] Jesper Larsson Träff, Hubert Ritzdorf, Rolf Hempel. *The Implementation of MPI-2 One-Sided Communication for the NEC SX-5.* In Proc. Supercomputing 2000 Conference, Dallas, November 2000.

[98] Maciej Golebiewski, Jesper Larsson Träff. *MPI-2 One-sided Communications on a Giganet SMP Cluster.* In Proc. EuroPVM/MPI 2001, LNCS 2131, pp.16-23, Springer-Verlag Berlin Heidelberg, 2001.

[99] Stephen Booth, Elson Mourao: *Single sided MPI implementations for SUN MPI.* In Proc. Supercomputing 2000 Conference, Dallas, November 2000.

[100] Elson Mourao, Stephen Booth: *Single Sided Communications in Multi-Protocol MPI.* In Proc. EuroPVM/MPI 2000, LNCS 1908, pp.176-183, Springer-Verlag Berlin Heidelberg, 2000.

[101] Jarek Nieplocha, Jialin Ju, and Edoardo Apra: *One-sided Communication on the Myrinet-based SMP Clusters using the GM Message-Passing Library.* Workshop on Communication Architecture for Clusters (CAC ‘01), in conjunction with Int’l Parallel and Distributed Processing Symposium (IPDPS ’01), San Francisco, April 2001.

[102] Frank Reker: *Integration von einseitiger Kommunikation in eine MPI-Bibliothek für speichergekoppelte PC-Cluster.* Diplomarbeit, Lehrstuhl für Betriebssysteme, RWTH Aachen, September 2001.

[103] Knut Omang: *Synchronization Support in I/O Adapter Based SCI Clusters.* In Proc. Workshop on Communication and Architectural Support for Network-based Parallel Computing, CANPC'97, San Antonio, LNCS 1199, pp.158-172, Springer Verlag Berlin Heidelberg, February 1997.

[104] Leslie Lamport: *A Fast Mutual Exclusion Algorithm.* ACM Transactions on Computer Systems, Vol.5 (1), pp.1-11, February 1987.

Asynchrone & Zero-Copy Nachrichtenübermittlung.

[105] Rossen P. Dimitrov: *Overlapping of Communication and Early Binding: Fundamental Mechanisms for Improving Parallel Performance on Clusters of Workstations.* Dissertation, Mississippi State University, Mississippi, May 2001.

[106] Rossen Dimitrov and Anthony Skjellum: *Impact of Latency on Application’s Performance*. In Proc. Fourth MPI Developer's and User's Conference, Ithaca, New York, April 2000.
`http://www.mpi-softtech.com/publications`

[107] Giuseppe Ciaccio and Gianni Chiola: *GAMMA and MPI/GAMMA on Gigabit Ethernet.* In Proc. EuroPVM/MPI 2000, LNCS 1908, pp.129-136, Springer-Verlag Berlin Heidelberg, September 2000.

[108] Patrick Geoffray, Loic Prylli, Bernard Tourancheau: *BIP-SMP: High Performance Message Passing over a Cluster of Commodity SMPs.* In Proc. Supercomputing ‘99 (SC99).
http://www.supercomp.org/sc99/proceedings/techpap.htm

[109] Francis O’Carroll, Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa: *The Design and Implementation of Zero Copy MPI Using Commodity Hardware with a High Performance Network.* In Proc. ACM SIGARCH ICS ’98, pp.243-250, July 1998.

[110] Yutaka Ishikawa, Hiroshi Tezuka, Atsushi Hori, Shinji Sumimoto, Toshiyuki Takahashi, Francis O’Carroll, Hiroshi Harada: *RWC PC Cluster II and SCore Cluster System Software - High Performance Linux Cluster.*
http://www.pccluster.org

[111] Hiroshi Tezuka, Francis O’Carroll, Atsushi Hori, and Yutaka Ishikawa: *Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication.* In Proc. 12th International Parallel Processing Symposium (IPPS ’98), Orlando, April 1998.

[112] Joachim Worringen: *SCI-MPICH - The Second Generation.* In Proc. SCI-Europe 2000 (Conference Stream of Euro-Par 2000), pp.11-20, München, September 2000.
http://wwwbode.cs.tum.edu/events/scieurope2000

[113] Joachim Worringen, Friedrich Seifert, Thomas Bemmerl: *Efficient Asynchronous Message Passging via SCI using Zero-Copy.* In Proc. SCI Europe 2001, Dublin, October 2001.
http://wwwbode.cs.tum.edu/events/scieurope2001

[114] Aristidis Sotiropoulos, Georgios Tsoukalas and Nectarios Koziris: *Enhancing the Performance of Tiled Loop Execution onto Clusters using Memory Mapped Network Interfaces and Pipelined Schedules.* In Proc. 2nd Workshop on Communication Architecture in Clusters (CAC ’02), in conjunction with IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS ’02), Ft. Lauderdale, April 2002.

Kollektive Operationen.

[115] Robert van de Geijn, David Payne, Lance Shuler, Jerrel Watts: *A Streetguide to Collective Communication and its Application.* January 1996.
http://www.cs.utexas.edu/users/rvdg/

[116] Thilo Kielmann, Rutger F.H. Hofman, Henri E. Bal, Aske Plaat, Raoul A.F. Bhoedjang: *MagPIe: MPI’s Collective Communication Operations for Clustered Wide Area Systems.* ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), Atlanta, Georgia, May 1999.
http://www.cs.vu.nl/albatross

[117] Persönliche Kommunikation mit Hubert Ritzdorf, NEC C&C Research Labs, St. Augustin, Juli 2002.

[118] Lars Paul Huse: *Collective Communication on Dedicated Clusters of Workstations.* In Proc. EuroPVM/MPI ’99, LNCS 1697, pp.469-476, Springer-Verlag Berlin Heidelberg, September 1999.

[119] Lars Paul Huse: *MPI Optimization for SMP Based Clusters Interconnected with SCI.* In Proc. EuroPVM/MPI 2000, LNCS 1908, pp.56-63, Springer-Verlag Berlin Heidelberg, September 2000.

[120] Sathish S. Vadhiyar, Graham E. Fagg and Jack Dongarra: *Automatically Tuned Collective Communications.* In Proc. Supercomputer 2000 (SC2000), Dallas, November 2000.
http://www.sc2000.org/techpapr/#01

[121] Steve Sistare, Rolf van de Vaart, Eugene Loh: *Optimization of MPI Collectives on Clusters of Large-Scale SMP's.* In Proc. Supercomputer 1999 (SC99), Portland, November 1999.
http://www.supercomp.org/sc99/proceedings/techpap.htm

[122] Rolf Rabenseifner: *Efficient MPI_Reduce for large vectors.* High-Performance Computing-Center, Universität Stuttgart, November 1997.
http://www.hlrs.de/organization/par/services/models/mpi/myreduce.html

[123] Yu-Chee Tseng, San-Yuan Wang, and Chin-Wen Ho: *Efficient Broadcasting in Wormhole-Routed Multicomputers: A Network-Partitioning Approach.* IEEE Transactions on Parallel and Distributed Systems, Vol.10 (1), January 1999.

[124] San-Yuan Wang, Yu-Chee Tseng, Ching-Sung Shiu, and Jang-Ping Sheu: *Balancing Traffic Load for Multi-Node Multicast in a Wormhole 2D Torus/Mesh.* In Proc. International Parallel and Distributed Processing Symposium 2000, Cancun, Mexiko, May 2000.
http://ipdps.eece.unm.edu/2000/session.html

[125] David F. Robinson, Philip K. McKinley, and Betty H.C. Cheng: *Optimal Multicast Communication in Wormhole-Routed Torus Networks.* IEEE Transactions on Parallel and Distributed Systems, Vol.6 (10), October 1995.

[126] Ram Kesavan, and Dhabaleswar K. Panda: *Multiple Multicast with Minimized Node Contention on Wormhole k-ary n-cube Networks.* IEEE Transactions on Parallel and Distributed Systems, Vol.10 (4), April 1999.

[127] Yu-Chee Tseng, Sandeep K. S. Gupta, and Dhabaleswar K. Panda: *An Efficient Scheme for Complete Exchange in 2D Tori.* In Proc. 9th International Parallel Processing Symposium (IPPS '95), IEEE Communications, 1995.

[128] Young-Joo Suh, and Sudhakar Yalamanchili: *Algorithms for All-to-All Personalized Exchange in 2D and 3D Tori.* In Proc. 10th International Parallel Processing Symposium (IPPS '96), pp.808-814, IEEE Communications, 1996.

[129] Wenheng Liu, Cho-Li Wang and Viktor K. Prasanna: *Portable and Scalable Algorithms for Irregular All-to-All Communication.* In Proc. 16th International Conference on Distributed Computing Systems (ICDCS '96), pp.428-435, IEEE Communications, 1996.

[130] Young-Joo Suh, and Kang G. Shin: *All-to-all Personalized Communication in Multidimensional Torus and Mesh Networks.* IEEE Transaction on Parrallel and Distributed Systems, Vol.12 (1), pp.38-59, January 2001.

[131] Prasenjit Mitra, David G. Payne, Lance Shuler, Robert van de Geijn, Jerrell Watts: *Fast Collective Communication Libraries, Please.* Technischer Bericht, Department of Computer Science, Univ. of Texas, Austin, TR-95-22, June 1995.

[132] Jerrell Watts: *Efficient Collective Communication on Multidimensional Meshes with Wormhole Routing.* Computer Science Honors Thesis, Univ. of Texas, Austin, May 1994.

[133] Jerrell Watts and Robert van de Geijn: *A Pipelined Broadcast for Multidimensional Meshes.* Parallel Processing Letters, Heft 2, 5. Jg, World Scientific Publishing Company, 1995.

[134] Fabrizio Petrini, Salvador Coll, Eitan Frachtenberg and Adolfy Hoisie: *Hardware- and Software-Based Collective Communication on the Quadrics Network.* In Proc. IEEE International Symposium on Network Computing and Applications 2001 (NCA 2001), pp.24-35, Boston, February 2002.

[135] Jesper L. Traeff: *MPIR_intra_Scan().* MPICH Distribution, Datei mpich/src/coll/intra_scan.c, 1998.
`http://www.unix.mcs.anl.gov/mpi/mpich`

Abgeleitete Datentypen.

[136] Mike Ashworth: *OCCOM Benchmark Code.* In Proc. Workshop on Scalable Parallel Computing on Cray Systems / 8. RAPS Workshop, Berlin und Offenbach, November 1995.

[137] William Gropp, Ewing Lusk, and Deborah Swider: *Improving the Performance of MPI Derived Datatypes.* Message Passing Interface Developer's and User's Conference (MPIDC '99), Atlanta, March 1999.

[138] Jesper Larsson Träff, Rolf Hempel, Hubert Ritzdorf, Falk Zimmermann. *Flattening on the Fly: efficient handling of MPI derived datatypes.* In Proc. EuroPVM/MPI '99, LNCS 1697, pp.109-116, Springer-Verlag Berlin Heidelberg, September 1999.

[139] Andreas Gäer: *Optimierung der Kommunikation mit nicht-zusammenhängenden Datentypen.* Studienarbeit, Lehrstuhl für Betriebssysteme, RWTH Aachen, November 2001.

[140] Glenn R. Luecke, Silvia Spanoyannis, James Coyle: *The Performance of MPI Derived Types on a SGI Origin 2000, a Cray T3E-900, a Myrinet Linux Cluster and an Ethernet Linux Cluster.* Unpublished preprint, October 2001.
`http://www.public.iastate.edu/~grl/publications.html`

[141] Joachim Worringen, Andreas Gäer, and Frank Reker: *Exploiting Transparent Remote Memory Access for Non-Contiguous and One-Sided-Communication.* In Proc. 2nd Workshop on Communication Architecture in Clusters (CAC '02), in conjunction with IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS '02), Ft. Lauderdale, Florida, April 2002.

Leistungsuntersuchung

[142] John D. McCalpin: *Memory Bandwidth and Machine Balance in Current High Performance Computers.* IEEE Computer Society, Technical Committtee on Computer Architecture (TCCA) Newsletter, December 1995.
`http://www.cs.virginia.edu/stream`

[143] Larry McVoy and Carl Staelin: *Lmbench: Portable Tools for Performance Analysis.* In Proc. USENIX 1996 Annual Technical Conference, pp.279 - 294, January 1996.
`http://www.usenix.org/publications/library/proceedings/sd96/full_papers/mcvoy.pdf`

[144] William Gropp and Ewing Lusk: *Reproducible Measurements of MPI Performance Characteristics.* In Proc. EuroPVM/MPI '99, LNCS 1697, pp.11-18, Springer-Verlag Berlin Heidelberg, September 1999.
http://www-unix.mcs.anl.gov/~gropp

[145] Ralf Reussner, Jesper Larsson Träff, Gunnar Hunzelmann. *A Benchmark for MPI Derived Datatypes.* In Proc. EuroPVM/MPI 2000, LNCS 1908, pp.10-17, Springer-Verlag Berlin Heidelberg, September 2000.

[146] D.H. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo and M. Yarrow: *The NAS Parallel Benchmarks 2.0.* NASA Technical Report NAS-95-020, NASA Ames Research Center, December 1995.
http://www.nas.nasa.gov/Software/NPB

[147] Marcus Dormans: *Parallelisierung gitterbasierter Algorithmen auf NUMA-Verbundsystemen mit gemeinsamem Speicher.* Dissertationsschrift, Lehrstuhl für Betriebssysteme, RWTH Aachen, September 1999.

[148] E. Lindahl, B. Hess und D. van der Spoel: *GROMACS 3.0: A package for molecular simulation and trajectory analysis.* Journal ol Molecular Modeling 7 (2001), pp.306-317, 2001.
http://www.gromacs.org

[149] Laxmikant Kalé, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten: *NAMD2: Greater Scalability for parallel molecular dynamics.* Journal of Computational Physics 151, pp.283 - 312, 1999.
http://www.ks.uiuc.edu/Research/namd

[150] Th. Matthey and J.P. Hansen: *Evaluation of MPI's One-Sided Communication Mechanisms for Short-Range Molecular Dynamics on the Origin2000.* In Proc. PARA 2000, Bergen, 2000.

[151] Th. Matthey, J.A. Izaguirre: *ProtoMol: A Molecular Dynamics Framework with Incremental Parallelization.*
http://www.nd.edu/~lcls/Protomol.html

[152] Persönliche Kommunikation mit Thierry Matthey, Universität Bergen, Norwegen, Oktober 2001.

[153] Pallas MPI Benchmark (PMB), Pallas GmbH, 2001.
http://www.pallas.de

[154] L. Ruby Leung, John G. Michalakes, and Xindi Bian: *Parallelization of a Subgrid Orographic Precipation Scheme in a MM5-based Regional Climate Model.* LNCS 2073, pp.195-204, Springer-Verlag Berlin Heidelberg, 2001.
http://citeseer.nj.nec.com/449184.html
http://www.mmm.ucar.edu/mm5/mm5-home.html

[155] Hong Ong and Paul A. Farrell: *Performance Comparison of LAM/MPI, MPICH and MVICH on a Linux Cluster connected by a Gigabit Ethernet Network.* In Proc. 4th Annual Linux Showcase & Conference, pp.353-362, USENIX Association, Atlanta, October 2000.

Sonstige

[156] Peter Ottmann und Peter Widmeyer: *Algorithmen und Datenstrukturen.* Spektrum Akademischer Verlag, Heidelberg, 1996.

[157] David A. Patterson, John L. Hennessy: *Computer Architecture: A Quantitative Approach.* 2. Auflage, Morgan Kaufmann Publishing, San Francisco, 1996.

[158] Kai Hwang: *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, New York, December 1992.

[159] David E. Culler, Jaswinder Pal Singh, Anoop Gupta: *Parallel Computer Architecture: A Hardware/Software Approach.*1st edition, Morgan Kaufmann Publishers, San Franciso, August 1998.

[160] U. Vahalia: *UNIX Internals: The New Frontiers.* Prentice-Hall Publishing, Upper Saddle River, New Jersey, 1996.

[161] B. Goodheart, J. Cox: *UNIX System V Release 4 - Reise durch den Zaubergarten.* Prentice-Hall Verlag, München, 1994.

[162] Andrew S. Tannenbaum: *Computer Networks.* 3rd edition, Prentice-Hall Publishing, New York, March 1996.

[163] W. Richard Stevens: *Network Programming.* 2nd edition, Prentice-Hall Publishing, Upper Saddle River, New Jersey, August 1998.

[164] *Linux Kernel - Algorithmen und Datenstrukturen der Version 2.4.* Addison-Wesley Verlag, München, 2001.

[165] Richard Mc Dougall and Jim Mauro: *Solaris Internals.* Sun Press, Palo Alto, 2001.

[166] I.N. Bronstein, K.A. Semendjajew: *Taschenbuch der Mathematik.* B.G. Teubner Verlagsgesellschaft, Stuttgart/Leipzig, 1991.

[167] N.N.: *Portland Group Compiler, Workstation Suite.* Version 3.2.4, 2001.
`http://www.pgroup.com`

[168] N.N.: Intel C Compiler for Linux. Version 5.0.1, 2001.
`http://www.intel.com/software/products/compilers/`

DANKSAGUNG

Für die Möglichkeit zur Anfertigung dieser Dissertationsschrift im Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Betriebssysteme (LFBS) möchte ich dem Leiter des Lehrstuhls, Prof. Dr. habil. Thomas Bemmerl, herzlich danken. Er hat durch die Gewährung großer Freiheit bei der wissenschaftlichen Betätigung bei gleichzeitiger Fokussierung auf das Ziel dieser Betätigung zum Gelingen der Arbeit wesentlich beigetragen. Dies schließt die Bereitstellung der notwendigen Ressourcen am Lehrstuhl ein.

Ebenso möchte ich dem ehemaligen Leiter des Zentralinstituts für angewandte Mathematik (ZAM) am Forschungszentrum Jülich, Prof. Dr. Friedel Hoßfeld, für die Übernahme des Koreferats danken. Durch seine Vorlesung und die an seinem Institut angefertigte Diplomarbeit bin ich zur Beschäftigung mit der Parallelverarbeitung und dem Hochleistungsrechnen motiviert worden. Sein sorgfältiges Koreferat war ein unverzichtbarer Beitrag zu dieser Arbeit.

Unter den Mitarbeitern des LFBS, die durch die freundschaftliche und kollegiale Atmosphäre alle dazu beigetragen haben, meine Zeit dort in bester Erinnerung zu halten, möchte ich einzelne Beiträge zum Gelingen dieser Arbeit besonders hervorheben. Mein Vorgänger Dr. Marcus Dormans hat SCI-gekoppelte Rechnerverbundsysteme am Lehrstuhl etabliert und die erste Version der SMI-Bibliothek entwickelt. Gemeinsam mit Martin Pöppe habe ich die ersten Gehversuche in MPI unternommen und auch weiterhin intensive Auseinandersetzungen gepflegt. Nicolas Berr hat durch seine Implementationsarbeiten und Anregungen wertvolle Unterstützung für die Entwicklung der SMI-Bibliothek geleistet. Die Diskussionen mit Karsten Scholtyssik bei der gemeinsamen Arbeit an MP-MPICH waren immer wieder hilfreich bei der Lösung von Problemen.

Mehrere Studenten haben als studentische Hilfskraft oder durch die Anfertigung von Studien- oder Diplomarbeiten zur Entwicklung von SCI-MPICH beigetragen. Für ihre produktive und motivierte Mitarbeit danke ich Boris Bierbaum, Stefan Meyer, Andreas Gäer, Frank Reker, Stefan Erben und Hendrik Meyer.

Die gute Zusammenarbeit mit dem Hersteller der verwendeten PCI-SCI-Adapter, Dolphin Interconnect Solutions (ICS) in Oslo, war zum Gelingen der Arbeit unabdingbar. Hugo Kohmann war in technischen Fragen nie um eine Antwort verlegen. Der CEO von Dolphin ICS, Kare Lochsen, hat durch finanzielle Unterstützung in Form von Reisekosten und Hardwarebeschaffung die Arbeiten an SCI-MPICH unterstützt.

Friedrich Seiffert vom Lehrstuhl für Rechnerarchitektur der TU Chemnitz hat mit seinen Arbeiten im Linux-Kernel und der gemeinsamen Erweiterung des SCI-Treibers die Nutzung von benutzer-allokiertem Speicher für DMA-Übertragungen in SCI-MPICH ermöglicht.

Meinem jetzigen Arbeitgeber NEC C&C Research Laboratories danke ich für die Unterstützung bei der Fertigstellung der Arbeit durch die Möglichkeit, die Ressourcen des Labors dazu zu nutzen.

Mein besonderer Dank gilt Ute und Ansgar: Durch ihren starken Rückhalt und Ansporn haben sie großen Anteil an dieser Arbeit. Meinen Eltern danke ich für all das, was sie mir auf dem langen Weg bis zum Abschluß dieser Arbeit mitgegeben haben.

LEBENSLAUF

Name	Joachim Worringen
Geburt	5. Oktober 1969 in Oberhausen
Familienstand	verheiratet, 1 Kind
Staatsangehörigkeit	deutsch

Schulausbildung

08/1975 - 06/1979	Ludgerusgrundschule Essen-Werden
08/1979 - 06/1988	Gymnasium Essen-Werden

Grundwehrdienst

10/1988 - 12/1989	Hilfsausbilder bei der Luftwaffe (21./V LAR 1, Pinneberg)

Studium

10/1989 - 06/1997	Studium der Elektrotechnik an der RWTH Aachen Studienrichtung Technische Informatik
	Diplomarbeit am ZAM, Prof. Dr. F. Hoßfeld: *Einbindung von Multithreading und effizienter Synchronisation in den SVM-Fortran-Compiler*

Tätigkeiten

12/1991 - 04/1994	Studentischer Mitarbeiter bei der ESCOM AG, Filliale Aachen
05/1994 - 09/1994	
und 05/1995	Praktikant bei Arianespace S.A., Korou, Französisch Guyana
07/1995 - 04/1996	Studentische Hilfskraft am Institut für Bergwerks- und Hüttenmaschinenkunde, RWTH Aachen (Prof. Dr. A. Seeliger)
07/1997 - 06/2002	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Betriebssysteme der RWTH Aachen (Prof. Dr. habil. Th. Bemmerl)
seit 07/2002	Research Scientist bei NEC Computer & Communications Research Laboatories, St. Augustin