Scalable Computing Aachen

LEHRSTUHL FÜR BETRIEBSSYSTEME

Univ.-Prof. Dr. habil. Thomas Bemmerl

Optimierung des Versands von nicht-zusammenhängenden Daten in einer MPI-Bibliothek für SCI-Cluster

Andreas Gäer
Matrikelnummer 201291

19. November 2001

Studienarbeit
an der
Rheinisch-Westfälischen Technischen Hochschule Aachen
Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl für Betriebssysteme

Betreuer
Univ.-Prof. Dr. habil. Thomas Bemmerl
Dipl.-Ing. Joachim Worringen

Inhaltsverzeichnis

Kapitel 1

Einleitung

MPICH ist eine frei verfügbare und weit verbreitete Implementation des *Message Passing Interface (MPI)* Standards. Die am Lehrstuhl für Betriebssysteme entwickelte, darauf aufsetzende Implementation *SCI-MPICH* ist eine Anpassung für die Benutzung von *Scalable Coherent Interface (SCI)* Netzwerkadaptern unter verschiedenen Betriebssystemen. Diese Arbeit beschäftigt sich mit der Verbesserung der Kommunikationsleistung bei der Übertragung von nicht-kontinuierlichen Daten.

Der MPI Standard definiert Methoden, um über Nachrichten Daten zwischen Prozessen einer parallelen Applikation auszutauschen. Die Daten müssen dabei nicht in einer kontinuierlichen Form im Speicher abgelegt sein, sondern es besteht die Möglichkeit, beliebig strukturierte, nicht-kontinuierliche Datenverteilungen durch *benutzerdefinierte Datentypen (derived datatypes)* zu beschreiben und diese in Kommunikationsoperationen zu benutzen. Dadurch wird dem Programmierer der Applikation die Aufgabe abgenommen, die Daten in eine für den Versand optimale Form zu bringen. Stattdessen obliegt es der Implementation der MPI Bibliothek, für den Versand eine - den logischen und physikalischen Gegebenheiten des Übertragungsweges angepaßte - Darstellung der Daten zu wählen. Gerade für Netzwerkverbindungen, die eine hohe Übertragungsrate und große Paketlängen bereitstellen, bietet sich eine optimierte Implementierung des Versands benutzerdefinierter Datentypen an. In dieser Arbeit wird ein Mechanismus vorgestellt, mit dem durch ein *direktes Kopieren* in das entfernte Shared Memory die Übertragungsleistung bei Benutzung von nicht-kontinuierlichen Datentypen auf SCI Systemen stark verbessert wird.

In den folgenden Abschnitten dieses Kapitels wird kurz auf den MPI Standard, die Hardwareschnittstelle SCI, das Shared Memory Interface und die MPICH Implementation des LfBS, SCI-MPICH eingegangen. Danach folgt eine kurze Erläuterung der unterschiedlichen Übertragungsprotokolle in MPICH. Diese Einführung führt die wichtigsten Begrifflichkeiten ein, die in den folgenden Kapiteln benötigt werden.

Im zweiten Kapitel folgt eine Betrachtung der benutzerdefinierten Datentypen in MPI. Es werden grundsätzliche Überlegungen zum Versand und Empfang von nicht-kontinuierlichen Daten durchgeführt und erste Abschätzungen zur

theoretischen Leistungsfähigkeit des Übertragungsmechanismus aufgestellt.
Das dritte Kapitel enthält eine Beschreibung der Implementation eines effektiven Mechanismus zum Versenden und Empfangen nicht-kontinuierlicher Daten in SCI-MPICH. Die für diese Aufgabe benötigten Datenstrukturen und Algorithmen werden vorgestellt und erläutert.
Eine Betrachtung der Leistung der hier vorgestellten Verbesserungen wird in Kapitel vier präsentiert. Hier werden die Übertragungsleistungen für verschiedene Datenverteilungen untersucht und auf verschiedenen Systemen miteinander verglichen.

1.1 Message Passing Interface

Einzelne Prozesse einer parallelen Applikation auf einem parallelen Rechnersystem, wie etwa einem Cluster, müssen miteinander kommunizieren können, um eine Synchronisation und Datenaustausch zu gewährleisten. Hier existieren zwei Ansätze, um diese Kommunikation zu ermöglichen. Zum einen können alle vorhanden Verarbeitungseinheiten (Prozessoren) mit einem gemeinsamen Speicher (True Shared Memory) verbunden werden. Die Zugriffgeschwindigkeit auf Daten ist dann unabhängig von der Zugehörigkeit zu einem bestimmten Prozeß, Kommunikation mit einem anderen Prozeß ist in der gleichen Geschwindigkeit möglich wie der Zugriff auf lokale Daten. Allerdings steigen die Kosten für ein solches paralleles Rechnersystem nicht-linear mit der Anzahl der verwendeten Prozessoren.
Ein anderer Ansatz zum Aufbau eines parallelen Rechnersystems ist das Verbinden von einzelnen Recheneinheiten mit jeweils lokalem Speicher, über ein lokales Netzwerk. Die Kommunikation muß in diesem Fall durch den Austausch von *Nachrichten* über das Netz erfolgen und der Zugriff auf Daten eines entfernten Prozesses ist dadurch mit erheblichen Geschwindigkeitseinbußen verbunden. Die Kosten für ein solches System sind jedoch lediglich proportional zur Anzahl der eingesetzten Einheiten[1]
Da bei vielen Anwendungen der notwendige Datenaustausch zwischen einzelnen Prozessen auf ein Minimum reduziert werden kann, läßt sich mit einem solchen verteilten System eine kosteneffektive, effiziente Lösung für viele Probleme schaffen. Der Programmierer einer verteilten Anwendung kann für die Kommunikation auf Bibliotheken zurückgreifen, die die benötigte Funktionalität bereitstellen. Der Zugriff auf die in der Bibliothek gekapselten Dienste erfolgt dabei über eine *Schnittstelle (Application Programming Interface, API)*. Eine solche Schnittstelle ist mit *MPI* seit 1994 als Standard definiert [6]. MPI hat sich seitdem stark verbreitet und ist aufgrund der frei verfügbaren *MPICH Implementation* auf fast allen Rechnerplattformen und Betriebssystemen verfügbar.

[1]bei der Verbindung über das in dieser Arbeit betrachtete SCI Netzwerk, sonst fallen zusätzliche Kosten für zentrale Netzwerkhardware, wie z.B. Switches, an.

1.2 SCI-MPICH

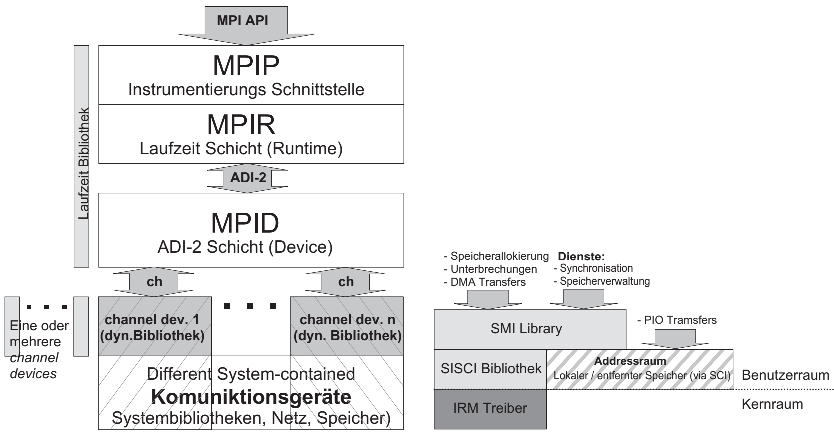

Abbildung 1.1: MPICH Schichtenmodell und ch_smi Gerätetreiber

Das am LfBS entwickelte *SCI-MPICH* [10] bindet einen neuen Gerätetreiber in in die frei verfügbare MPICH-Distribution ein und stellt somit einen weiteren Übertragungskanal (SCI) für MPI Anwendungen zur Verfügung. Das Shared Memory Interface ist hier als Gerätetreiber *ch_smi* in die unterste Ebene des MPICH-Schichtenmodell (Abb. 1.1(a)) eingegliedert. Der Aufbau von SCI-MPICH ist streng hierarchisch gegliedert, die Funktionen der einzelnen Schichten kommunizieren nur über festgelegte Schnittstellen mit Diensten einer anderen Schicht. Die MPI-Applikation bildet hier die oberste Schicht. Sie kann alle Funktionen des MPI-1 und eine Teilfunktionalität des MPI-2 Standards über die nächste Schicht, die MPI-API, anfordern. Die MPI-API setzt alle Anforderungen der Applikation in Aufrufe der eigentlichen Library um, die in der MPIR-Schicht implementiert ist. Die zwischengeschaltete MPIP dient zum Profiling, also zur Zeitmessung einzelner Teilbereiche und braucht hier nicht berücksichtigt werden. In der MPIR-Schicht ist die Funktionalität der Library implementiert. In der darunter liegenden MPID-Schicht, die über die ADI-2 Schnittstelle angesprochen wird, sind alle Kommunikationsfunktionen generisch, unabhängig vom physikalischen Übertragungsmedium implementiert. Für den physikalischen Austausch von Daten werden die Dienste eines Gerätetreibers in Anspruch genommen. Dieser Treiber bietet der MPID Funktionen zur Initialisierung des Netzwerkes und für den Datentransport verschiedene Transportfunktionen (Kapitel 1.5) an.

1.3 Shared Memory Interface und ch_smi

Das Shared Memory Interface *SMI* wurde am LfBS entwickelt, um auf verschiedensten Hardwareplattformen einen einheitlichen Zugriff auf *verteilen Speicher (Shared Memory)* zu ermöglichen und die physikalische Implementation vom

Benutzer abzukapseln. Das SMI wurde für die Benutzung als Gerätetreiber in MPICH angepaßt (ch_smi) und ermöglicht somit auch die Benutzung von SCI-Netzwerkhardware als Übertragungskanal unter MPI.
Das SMI ist weitgehend plattformunabhängig und steht unter Linux, Solaris und Windows Systemen zur Verfügung. Der Gerätetreiber ch_smi setzt darauf auf und ermöglicht über seine Schnittstelle die Schaffung eines gemeinsamen globalen Speicherraums zwischen Prozessen einer parallelen Applikation. Vor dem eigentlichen Datentransport erfolgt eine Initialisierung, bei der über das SMI die SCI-Karten der am Cluster teilnehmenden Systeme angewiesen werden, einen Teil des lokalen Systemspeichers öffentlich verfügbar zu machen. Dieser Speicher wird dann bei allen beteiligten Prozessen in den *user space* eingebunden und steht danach als Teil eines gemeinsamen globalen Speicherraums zur Verfügung.
Nach der Initialisierung ist eine Kommunikation der einzelnen Prozesse über diesen Speicherraum möglich. Dafür schreibt der sendende Prozeß seine Daten in einen freien Puffer des globalen Speichers, der lokal auf dem System des Empfängers liegt. Der Schreibvorgang einer Nachricht erfolgt also auf entfernten Speicher über das physikalische Medium. Der Empfänger kann die Nachricht aus seinem lokalen Speicher direkt auslesen. Die Kommunikation erfolgt hier also *Punkt-zu-Punkt* zwischen den beteiligten Prozessen.

1.4 Scalable Coherent Interface

Der *Scalable Coherent Interface Standard (SCI)* [1] wurde entwickelt, als abzusehen war, daß ein Bussystem die Anforderungen an Skalierbarkeit und Leistungsfähigkeit für ein Verbindungsnetzwerk nicht erfüllen kann. Der Standard spezifiziert die physikalischen und logischen Ebenen eines Netzwerks, das den Zugriff auf den entfernten Speicher anderer Knoten im Verbund möglich macht. Im Netzwerk wird mit Hilfe der SCI-Hardware ein gemeinsamer Speicherbereich geschaffen, auf den die einzelnen Knoten gemeinsam zugreifen können. Der Zugriff auf entfernte Speichersegmente ist dabei zwar deutlich langsamer als auf lokalen Speicher, doch er erfolgt mit den gleichen Prozessorbefehlen, wodurch er für Applikationen transparent wird. Es entsteht also ein NUMA[2]-System, bei dem die Zugriffszeiten nach der physikalischen Lage des Speichers variieren (siehe auch Abb. 2.4 in Kapitel 2.3).

[2] Abkürzung für *non uniform memory architecture* (dt. ungleichförmiger Speicherzugriff)

1.5 Übertragungsprotokolle in SCI-MPICH

Für die Kommunikation zwischen Prozessen im Cluster stellt SCI-MPICH drei verschiedene Übertragungsprotokolle zur Verfügung. Die Auswahl der Protokolle erfolgt nach der Größe der zu verschickenden Nachricht, um stets eine optimale Ausnutzung der zur Verfügung stehenden Ressourcen zu gewährleisten [10].

Short

Das für geringe Datenmengen ausgelegte Short Protokoll wird auch von den anderen Protokollen zur Koordination benutzt. Es ist also grundlegend für jede Kommunikation. Die Größe eines Short-Paketes liegt minimal bei 64 Bytes[3], von denen dann 32 Bytes als Nutzlast zur Verfügung stehen. Da eine spezielle Behandlung nicht-kontinuierlicher Datentypen bei einer Datenmenge von 32 Bytes aufgrund der relativ niedrigen Leistung der SCI-Hardware bei kleinen Blockgrößen (Kapitel 2.3) wenig Sinn macht, ist dieses Protokoll für die weiteren Ausführungen nicht relevant.

Eager

Das Eager Protokoll ist für die Übertragung von Daten einer Größe von bis zu einigen dutzend Kilobyte gedacht. Für jeden Prozeß der Applikation werden im lokalen Shared Memory einige Puffer angelegt, die der Aufnahme von Eager Nachrichten dienen. Beim Senden sucht sich der sendende Prozeß einen freien Eager Puffer des Empfängers und schreibt die Daten in diesen Puffer. Über eine Kontrollnachricht wird der Empfänger über den abgeschlossenen Schreibvorgang in Kenntnis gesetzt. Die Verwaltung der Eager Puffer erfolgt intern über Ringpuffer, die Zeiger auf die eigentlichen Speicherbereiche enthalten.

Rendez-vous

Für alle Nachrichten die größer sind als die Eager Puffer oder für den Fall, daß keine Eager Puffer mehr frei sind, kommt das Rendez-vous (RNDV) Protokoll zum Einsatz. Eine obere Grenze für die Nachrichtenlänge existiert hier nicht, so daß Nachrichten, für die nicht genug Speicher im Shared Memory zur Verfügung steht, in kleinere Teilstücke aufgeteilt werden müssen. Der Ablauf einer RNDV Übertragung ist in 1.2 skizziert. Der folgende Text enthält Verweise auf die Stationen der Übertragung in Klammern.
Zu Beginn der Kommunikation signalisiert der Sender dem Empfänger durch eine Kontrollnachricht seinen Wunsch, eine RNDV Übertragung durchzuführen (1). Dieser antwortet auf diese Anfrage mit der Angabe eines Speicherbereichs in seinem lokalen Shared Memory, welcher im folgenden zum Datenaustausch benutzt wird (2). Der Sender schreibt nun seine Daten in diesen Puffer und signalisiert dem Empfänger wenn die Daten bereit zum Auslesen sind (3). Der Empfänger liest die Daten nun aus und signalisiert dem Empfänger danach,

[3] maximale Datenpaketgröße von SCI, je nach Hardware auch 128 Bytes

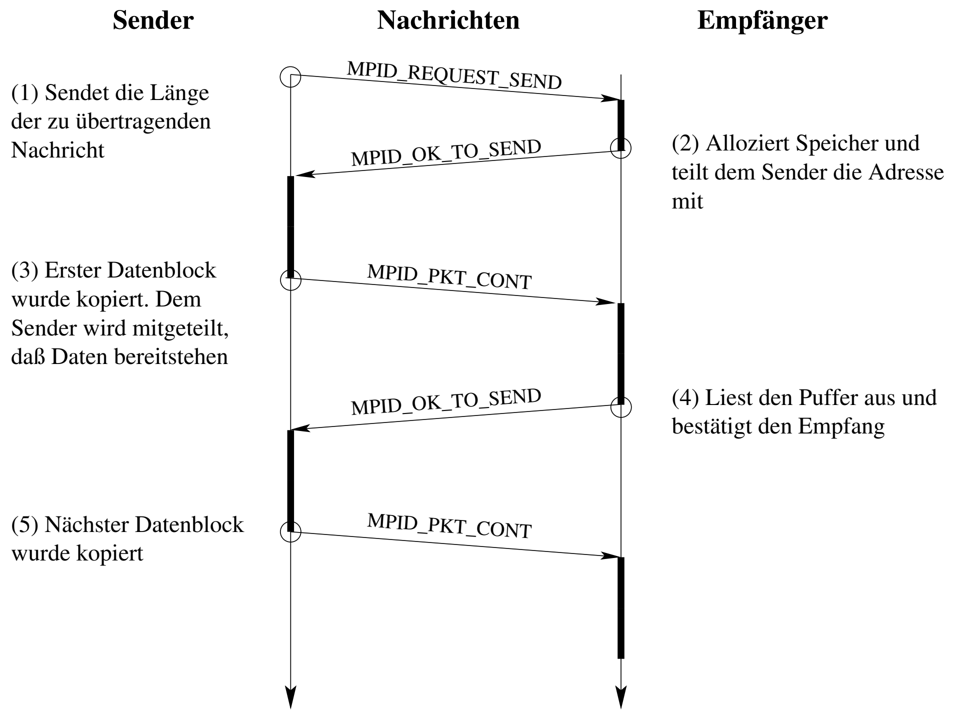

Abbildung 1.2: Ablauf des Rendez-vous Protokoll (aus [3])

daß er die Daten erhalten hat und für weiteren Empfang bereit ist (4). Die Schritte (3) und (4) wiederholen sich jetzt so lange, bis die komplette Nachricht vollständig übertragen ist. Die abschließende Empfangsbestätigung des Empfängers entfällt.
Für das RNDV Protokoll sind einige Verbesserungen implementiert worden. So ist es nicht nötig, daß der Empfänger wartet, bis der Sender alle Daten komplett geschrieben hat. Er kann bereits mit dem Lesen beginnen, wenn der Sender nur einen Teil der Daten geschrieben hat. Dies wird vom Sender mittels Protokollnachrichten während des Schreibvorgangs angezeigt.

Kapitel 2

Nicht-kontinuierliche Datentypen

Bei einigen Problemen, die auf einem verteilten Rechnersystem durch Parallelisierung gelöst werden sollen, ergibt sich der Fall, daß die auszutauschenden Daten nicht kontinuierlich im Speicher vorhanden sind. Dies ergibt sich zum Beispiel bei der Verteilung von Daten in einem dreidimensionalen Arrays, wie dies bei der *Ocean Decomposition* der Fall ist (Abb. 2.1). Dafür stellt der MPI Standard bereits einige Funktionen bereit, mit denen Strukturen vom Benutzer gebildet werden können, die diese nicht-kontinuierliche Datenverteilung beschreiben. Die so definierten Datentypen können direkt in allen Funktionen zur Datenmanipulation benutzt werden, ohne daß sich der Benutzer weitere Gedanken zur Behandlung der Daten machen müßte. Die MPI Bibliothek hat in diesem Fall die Aufgabe, einen optimalen Weg zur Übertragung der Daten zu finden.

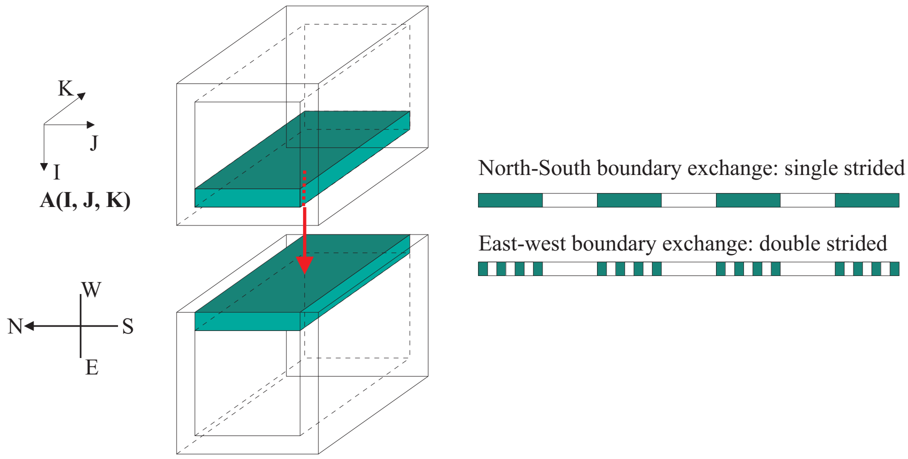

Abbildung 2.1: *Ocean Decomposition* und resultierende nicht-kontinuierliche Datenverteilung beim *boundary exchange* (aus [2])

Im folgenden wird die Zusammensetzung dieser Datentypen und die Verwendung bei Sende- und Empfangsoperationen beschrieben.

2.1 Benutzerdefinierte Datentypen

Um die Beschreibung komplexer Datenstrukturen im Speicher zu erleichtern, bietet der MPI Standard die Möglichkeit, ausgehend von den Standarddatentypen komplexere Strukturen zu bilden. In MPI 1 stehen dazu die Typen *vector*, *hvector*, *indexed*, *hindexed* und *struct* zur Verfügung. Für eine nähere Beschreibung sei hier auf [6] verwiesen, wichtig ist jedoch anzumerken, daß durch die Benutzung der *benutzerdefinierten Datentypen* Strukturen entstehen können, die nicht mehr einen durchgängigen Bereich im Speicher beschreiben, sondern durchaus Lücken enthalten können. Daten, die so beschrieben werden, können direkt mit den *MPI_Send()* und *MPI_Recv()* Library Aufrufen versandt werden, die Funktionen der MPIR Schicht sind dann für eine korrekte Behandlung der Daten verantwortlich.
Für das Zusammensetzen stehen verschiedene Methoden zur Verfügung, die jeweils eine andere Beschreibung der Datenverteilung erlauben. Im folgenden werden diese Möglichkeiten kurz beschrieben:

1. `MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)`

   Bildet einen neuen Datentyp, indem der ursprüngliche Typ ohne Lücken *count* mal direkt hintereinander gehängt wird. Ist der ursprüngliche Typ kontinuierlich, dann ist auch der neue Typ kontinuierlich.

2. `MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)`

   Wie bei (1) wird *blocklength* mal der ursprüngliche Typ ohne Lücken hintereinander gehangen. Von den so gewonnenen Blöcken werden *count* Stück mit einem Zwischenraum aneinander gereiht. Dabei gibt *stride* den Abstand zwischen den Anfängen der Blöcke an, bezogen auf die Länge des ursprünglichen Typs.

3. `MPI_TYPE_HVECTOR(count, blocklength, stride, oldtype, newtype)`

   Der neue Typ wird wie bei (2) gebildet, jedoch wird hier der Abstand *stride* in Bytes gemessen.

4. ```
   MPI_TYPE_INDEXED(count, array_of_blocklengths,
                    array_of_displacements, oldtype, newtype)
   ```

   Dieser Konstruktor bildet eine Struktur, bei der der ursprüngliche Typ zu Blöcken zusammengesetzt wird, deren Längen in *array_of_blocklengths* definiert werden. Diese Blöcke werden dann gemäß einer Liste der Offsets in *array_of_displacements* aneinander gereiht. Die Offsets beziehen sich auf die Länge des ursprünglichen Typs. Insgesamt werden *count* Blöcke gebildet.

5. ```
   MPI_TYPE_HINDEXED(count, array_of_blocklengths,
                     array_of_displacements, oldtype, newtype)
   ```

   Wie (4), jedoch werden die Offsets in *array_of_displacements* als Bytes angegeben.

6. `MPI_TYPE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)`

   Dieser Konstruktor bietet die größte Freiheit und erweitert den Konstruktor (5) so, daß jetzt für jeden Block ein Basistyp angegeben werden kann. Insgesamt können damit *count* Blöcke unterschiedlicher Länge und unterschiedlicher Basistypen an im *array_of_displacements* definierten Offsets gesetzt werden. Die Offsets werden als Bytes übergeben.

Wendet man diese Konstruktoren sukzessive an, um einen benutzerdefinierten Datentyp zu erhalten, so läßt sich die entstandene Struktur als ein hierarchisch gegliederter Baum betrachten, an dessen Wurzel der zuletzt definierte Typ steht und dessen Blätter durch MPI-Basistypen gegeben sind. In Abbildung 2.2 ist als Beispiel die Datenverteilung und der resultierende Typbaum eines Datentyps gegeben, der aus der Definition eines Vector- von einem Struct-Typen gebildet wurde. Die Parameter der Definition sind ebenfalls in der Abbildung gegeben. Auf diesen Typ werden sich alle folgenden Beispiele beziehen.

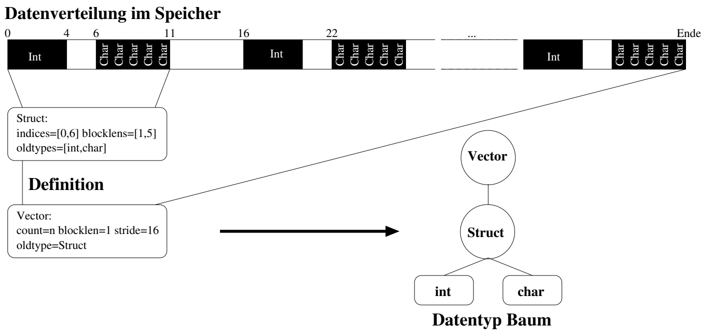

Abbildung 2.2: Verteilung der Daten im Speicher und resultierender Typbaum

2.2 Versenden nicht-kontinuierlicher Daten

Um Daten aus dem Benutzerspeicher zu versenden, die mit nicht-kontinuierlichen Datentypen beschrieben werden, ergeben sich für den Benutzer in MPI drei Möglichkeiten:

1. Für jeden kontinuierlichen Block wird eine Nachricht versandt.

2. Alle Blöcke werden in einem Puffer gesammelt - dieser Vorgang wird als *Packen* bezeichnet - und dieser wird dann mit einer Nachricht übermittelt. Der Empfänger muß diese Nachricht im Anschluß wieder *entpacken*.

3. Die Verteilung wird über einen *benutzerdefinierten Datentyp* beschrieben und mit diesem wird eine Nachricht versandt.

Möglichkeit 1 begrenzt die Übertragungsleistung stark, weil für jeden Block eine Initialisierung der Kommunikation nötig wird. Möglichkeit 2 erfordert neben

der Kopieroperation zwischen den beteiligten Prozessen noch mindestens einen lokalen Kopiervorgang beim Sender und dämpft daher ebenfalls die Übertragungsleistung. Bei Möglichkeit 3 entscheidet die MPI Bibliothek über die Art der Übertragung und somit entsteht die Möglichkeit, eine optimale Methode zu wählen.
Das Packen und Entpacken von nicht-kontinuierlichen Daten in einen bzw. aus einem kontinuierlichen Puffer ist über die MPI-Funktionen *MPI_Pack()* und *MPI_Unpack()* definiert. Von diesen Routinen machen auch die bisherigen Funktionen von SCI-MPICH zum Versenden und Empfangen Gebrauch, wenn ein nicht-kontinuierlicher Datentyp übergeben wird. In diesem Fall wird zuerst ein Zwischenpuffer erzeugt, in den dann die kompletten Daten aus dem Sendespeicher gepackt werden. Danach wird dieser Zwischenpuffer mit den Funktionen zum kontinuierlichen Senden je nach Größe des Puffers und der zur Verfügung stehenden Nachrichtenlänge entweder an einem Stück (Short, Eager) oder in Teilen (Rendez-vous) über das Netzwerk verschickt. Das Empfangen läuft analog dazu, auch hier wird zuerst ein Zwischenpuffer erzeugt, über den die Daten kontinuierlich empfangen werden. Nach beendeter Übertragung werden sie dann von diesem Puffer in den Empfangsspeicher entpackt. Insgesamt ergeben sich so vier Kopieroperationen (Abb. 2.3).

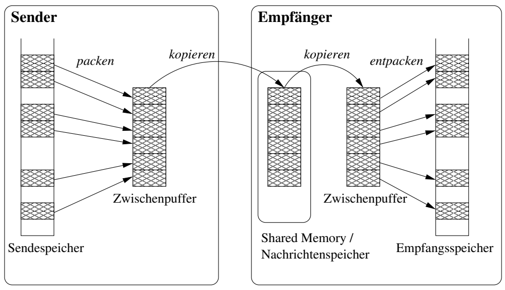

Abbildung 2.3: Kopieren über Zwischenpuffer

2.3 Optimierungspotential

Bei einem Nachrichtenaustausch über *Shared Memory* erscheinen zwei der vier genannten Kopiervorgänge auf den ersten Blick überflüssig. Es wäre möglich, direkt in den entfernten Speicher des Empfängers zu packen und von dort in den Empfangsspeicher zu entpacken. Es liegt nahe, an dieser Stelle eine Optimierung anzusetzen, allerdings ist zu beachten, daß für kleine Datenmengen eine Kopieroperation via SCI in entferntes Shared Memory relativ langsam ist. Daher stellt sich die Frage, ob für diese Blockgrößen nicht ein *Sammeln* im lokalen Speicher vorteilhaft wäre. Die Bandbreite B_z beim 4-Wege-Kopieren über Zwischenpuffer setzt sich zusammen aus der Bandbreite für ein lokales Kopieren der n_b Einzelblöcke der Blockgröße b_l in den Zwischenpuffer, der Bandbreite für ein entferntes Kopieren B_{sh} der gesamten Daten, dem Kopieren der gesamten Daten aus dem Nachrichtenpuffer in einen weiteren Zwischenpuffer und schließlich

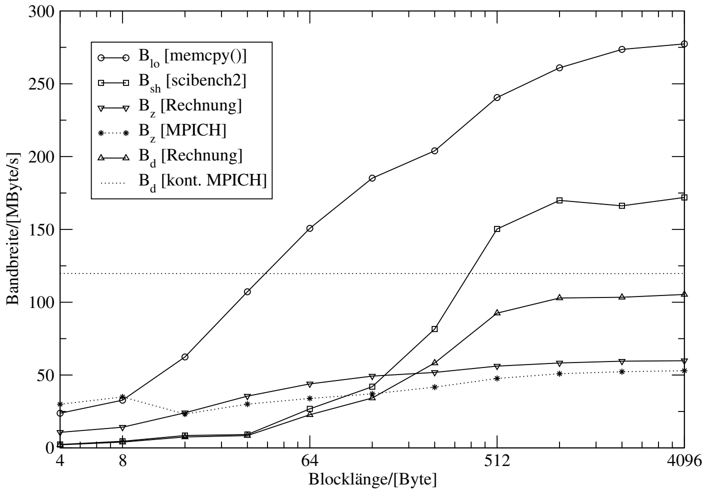

Abbildung 2.4: Bandbreite verschiedener Kopieroperationen

ein lokales Kopieren der Einzelblöcke in den Empfangspuffer:

$$\frac{1}{B_z(b_l)} = \underbrace{\frac{1}{B_{lo}(b_l)} + \frac{1}{B_{sh}(n_b \cdot b_l)}}_{\text{Sender}} + \underbrace{\frac{1}{B_{lo}(n_b \cdot b_l)} + \frac{1}{B_{lo}(b_l)}}_{\text{Empfänger}} \tag{2.1}$$

$$\Rightarrow B_z(b_l) = \frac{B_{lo}(b_l) \cdot B_{sh}(n_b b_l) \cdot B_{lo}(n_b b_l)}{2B_{sh}(n_b b_l) \cdot B_{lo}(n_b b_l) + B_{lo}(b_l)\left[B_{lo}(n_b b_l) + B_{sh}(n_b b_l)\right]} \tag{2.2}$$

Die Bandbreite B_d für einen direkten Kopiervorgang ohne Benutzung der Zwischenpuffer ergibt sich aus dem Kopieren der einzelnen Blöcke in das entfernte Shared Memory und dem lokalen Kopieren des Empfängers in den Empfangspuffer:

$$\frac{1}{B_d(b_l)} = \underbrace{\frac{1}{B_{sh}(b_l)}}_{\text{Sender}} + \underbrace{\frac{1}{B_{lo}(b_l)}}_{\text{Empfänger}} \Rightarrow B_{d,ges}(b_l) = \frac{B_{sh}(b_l) \cdot B_{lo}(b_l)}{B_{sh}(b_l) + B_{lo}(b_l)} \tag{2.3}$$

In Abbildung 2.4 ist die Bandbreite von kontinuierlichen Kopieroperation über das SCI Device (B_{sh} [scibench2]) für unterschiedliche Blockgrößen zusammen mit der Bandbreite für lokale memcpy() Aufrufe (B_{lo} [memcpy()]) aufgetragen. Die daraus resultierenden Werte für B_z und B_d gemäß (2.2) und (2.3) sind ebenso aufgeführt. Die Messungen der Werte erfolgten auf Pentium III-Systemen[1]. Die Puffergrößen für den memcpy() Test wurden groß genug gewählt, um reine Cache-Kopien auszuschließen. Zur Verifikation der berechneten Größen wurde noch eine Messung der Bandbreite bei nicht-kontinuierlichem Senden mit der

[1]Dual Prozessor SMP, 2x800 MHz Intel Pentium III Prozessor, ServerWorks ServerSet III-LE Chipsatz, 64 Bit/66 MHz PCI Bus und Dolphin D330 PCI-SCI Adapter

existierenden 4-Wege Implementation aufgeführt (B_z [MPICH]). Die Obergrenze für alle Optimierungen sollte bei dem Wert der Bandbreite für kontinuierliches Senden in SCI-MPICH liegen (B_d [kont. MPICH]).
Die Werte für das Kopieren mit Zwischenpuffer liegen bis zu einer Blockgröße von ca. 256 Byte deutlich über denen des direkten Kopierens, weil hier die SCI-Übertragung der kleinen Blöcke das direkte Verfahren stark ausbremst. Der Vergleich mit den realen Werten zeigt, daß die Rechnung eine gute Näherung darstellt. Die realen Werte liegen etwas unter den berechneten, was, bedingt durch den Overhead der Bibliothek, nicht weiter verwundert. Nur für Blockgrößen unter 8 Byte ist die Leistung der SCI-MPICH Implementation deutlich besser als die Rechnung. Dies liegt an einem optimierten Kopiervorgang für kleine Blöcke, während in der Rechnung stets von einem Standard-memcpy() ausgegangen wird.
Eine optimierte Implementation sollte daher für das Kopieren von nicht-kontinuierlichen Daten folgende Untergrenzen für die Bandbreite erfüllen:

1. $1 \leq b_l \leq 8$ (Bytes): vorhandene SCI-MPICH Implementation
2. $8 < b_l \leq 256$ (Bytes): berechnetes B_z
3. $256 < b_l$ (Bytes): berechnetes B_d

Parallelisierung

Bei diesen Überlegungen wurde allerdings stets von einer streng sequentiellen Abfolge der Kopieroperationen ausgegangen. Dies ergibt sich aus dem einfachen *Rendez-vous* Protokoll, wie es in Kapitel 1.5 beschrieben wurde. Bei der optimierten Version des Protokolls ist jedoch teilweise ein überlappendes Schreiben und Lesen bei Sender und Empfänger möglich und somit ein *Pipelining* der Kopieroperationen. Eine Obergrenze für Formel 2.2 ergibt sich dann zu:

$$\frac{1}{B_{z,max}(b_l)} = \frac{1}{B_{lo}(b_l)} + \frac{1}{B_{sh}(n_b \cdot b_l)} + \frac{1}{B_{lo}(b_l)} \tag{2.4}$$

$$\Rightarrow B_{z,max}(b_l) = \frac{B_{sh}(n_b b_l) \cdot B_{lo}(b_l)}{2B_{sh}(n_b b_l) + B_{lo}(b_l)} \tag{2.5}$$

Da das Packen und Entpacken stets auf die gesamten Daten angewandt wird, lassen sich die lokalen Kopien im 4-Wege Verfahren nicht parallelisieren.
Die Obergrenze beim direkten Kopieren für Formel 2.3 wird zu:

$$\frac{1}{B_{d,max}(b_l)} = \frac{1}{B_{sh}(b_l)} \Rightarrow B_{d,max}(b_l) = B_{sh}(b_l) \tag{2.6}$$

Hier bestimmt die Leistung der SCI-Übertragung die Bandbreite des gesamten Kopiervorgangs.

Kapitel 3

Direktes Senden/Empfangen

Wie im vorherigen Kapitel erläutert, werden zum Versenden von nicht-kontinuierlichen Daten im herkömmlichen vier Kopiervorgänge durchgeführt. Dies ist der Ausgangspunkt für die folgenden Überlegungen, durch das direkte Packen und Entpacken zwei Kopiervorgänge einzusparen und dadurch eine Steigerung der Leistung zu erlangen. Die Daten sollen also vom Versender direkt in das entfernte Shared Memory gepackt und vom Empfänger direkt in den Empfangspuffer entpackt werden (Abb. 3.1).

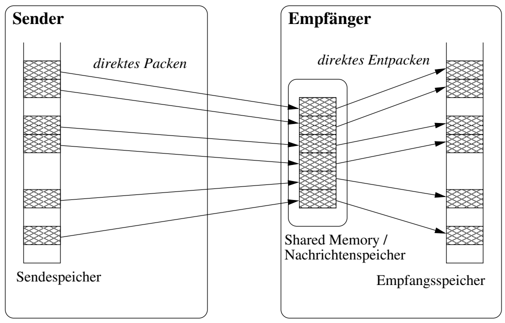

Abbildung 3.1: Direktes Kopieren in zwei Schritten

3.1 Anforderungen an ein Verfahren zum direkten Packen

Der Shared Memory Zugriff über SCI ist auf Blöcke einer maximalen Größe beschränkt. Eine Übertragung muß also mit Blöcken dieser Größe auskommen und die Nachricht evtl. auf mehrere Blöcke aufteilen. Daraus ergibt sich, daß ein Verfahren zum direkten Versenden/Empfangen in der Lage sein muß, nur einen Teil der Daten in einem Durchlauf zu packen und beim nächsten Durchlauf an der Stelle des Abbruchs effizient wieder anzusetzen.
Obwohl es möglich ist, diese Bedingungen auch mit den ursprünglichen rekursiven Packalgorithmen zu realisieren, wären dazu umfangreiche Änderungen nötig gewesen, wodurch die Leistung dieser Funktionen weiter eingeschränkt worden

wäre. Außerdem ist es in der rekursiven Implementierung lediglich möglich, Optimierungen auf einer Ebene des Datentypbaumes zu vollziehen, da Abhängigkeiten zwischen den Ebenen nicht betrachtet werden können. Zum Beispiel ist eine optimierte Bearbeitung des Vektor-Typs in einer Schleife mit festem Inkrement hier nur möglich, wenn der darunter liegende Typ ein Basistyp ist, ansonsten muß eine allgemeine rekursive Bearbeitung jedes Vektorblocks erfolgen.
Als weitere Anforderung kann man aus Abschnitt 2.3 entnehmen, daß eine Möglichkeit existieren sollte, für kleine Blockgrößen ein Sammeln im lokalen Zwischenspeicher vorzunehmen, damit die Leistung für diese Datentypen nicht unter die Leistung der herkömmlichen Methode fällt.
Zur Erfüllung dieser Anforderungen wurde mit dieser Studienarbeit ein vollkommen neuer Ansatz zum Packen und Entpacken realisiert, der als direkte Sende-/Empfangsroutine in SCI-MPICH erfolgreich eingefügt werden konnte.

3.2 Flattening on the fly

Für die Implementierung eines unterbrechbaren Pack- und Entpackverfahrens, wurde hier eine Methode gewählt, die in [8] beschrieben wird. Dieses *Flattening on the Fly (FF-Algorithmus)* genannte Verfahren wurde in dieser Arbeit so erweitert, daß es als Grundlage für ein Verfahren zum direkten Senden und Empfangen benutzt werden kann.
Die Funktionsweise des FF-Algorithmus ergibt sich aus der Erkenntnis, daß für jeden Basisdatentyp, im folgenden Blatt oder Basisblock genannt, im Datentypbaum ein spezifisches Wiederholungsmuster im Benutzerspeicher vorliegt, welches sich aus dem Weg von der Wurzel des Baums bis zum entsprechenden Blatt ergibt (Abb. 3.2). Ist dieses Wiederholungsmuster einmal bekannt, können alle Kopieroperationen auf dieser Grundlage durchgeführt werden.

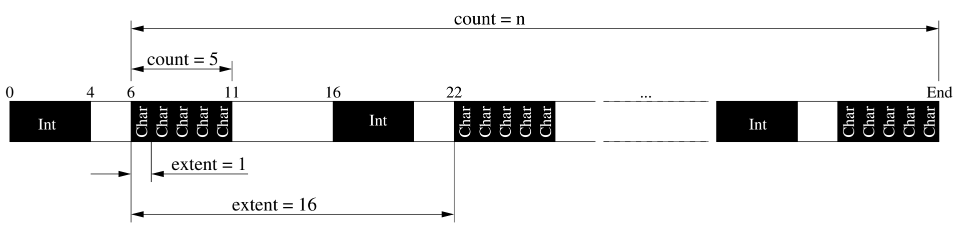

Abbildung 3.2: Wiederholungsmuster eines nicht kontinuierlichen Datentyps

Dafür sind für jede Ebene des Datentypbaums nur zwei Angaben signifikant, der Abstand zwischen den einzelnen Blöcken und ihre Anzahl. Außerdem müssen noch die Position des ersten Blocks im Speicher und die Länge eines einzelnen Blocks bekannt sein. Im gegebenen Beispiel (Abb. 3.2) ist die Länge des Basistyps *Char* 1 Byte und die erste Position 6. Es gibt *count = 5* Wiederholungen mit einem Abstand *extent = 1* zwischen den Anfängen der 1 Byte Blöcke und davon dann *count = n* Wiederholungen mit einem Abstand von *extent = 16.* Entsprechendes gilt für den Basistyp *Int.*

Der Stack

Das Wiederholungsmuster eines Datentyps ergibt sich sofort, wenn man den rekursiven Aufbau des Baumes betrachtet und eine komplette Rekursion von der Wurzel bis zum Blatt durchführt. Im internen Zustand der Rekursion ist dann auch das Wiederholungsmuster gespeichert [7]. Eine rekursive Implementation macht allerdings von dem vorhanden Muster und möglichen Abhängigkeiten der Blöcke keinen Gebrauch. Nachdem ein Basisblock kopiert wurde, kehrt der rekursive Algorithmus wieder zur nächst höheren Ebene zurück und prüft, ob weitere Basisblöcke kopiert werden müssen. Nur für die Vektortypen liegt in der MPICH auf unterster Ebene eine nicht-rekursive Optimierung vor.

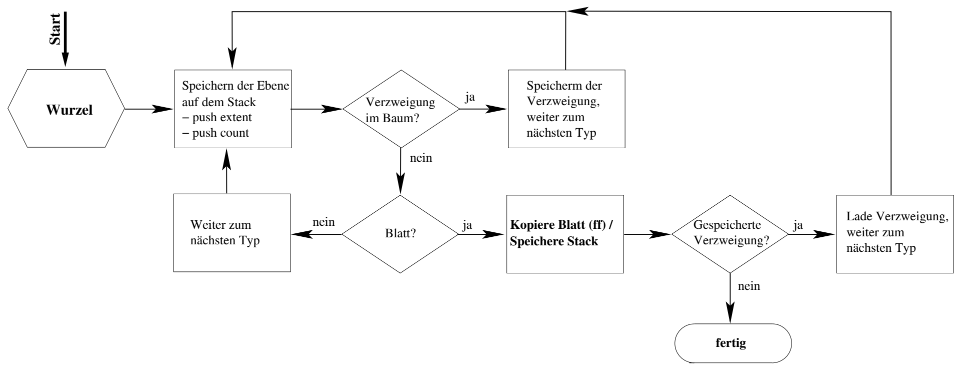

Abbildung 3.3: Iterativer Aufbau des Stacks

Im FF-Algorithmus wird der interne Zustand der Rekursion bei Erreichen eines Blattes explizit als Stack gespeichert und der Kopiervorgang erfolgt sodann iterativ auf Basis dieses Stacks. Der Stack enthält für jede Ebene des Typbaums wie schon vorher erwähnt die beiden Angaben Anzahl und Ausdehnung, im folgenden *count* und *extent* genannt. In Abbildung 3.3 ist der Ablauf des Stackaufbau in einer iterativen Form skizziert. Angefangen wird dabei an der Wurzel des Typbaums. Nun wird der Baum bis zu einem Blatt traversiert und für jeden durchlaufenen Typ werden die erforderlichen Informationen auf dem Stack gespeichert (Kapitel 3.3). Bei Verzweigungen im Typbaum (bei *indexed*, *hindexed* und *struct* Typen) wird der Ort der Verzweigung für einen späteren Rücksprung zwischengespeichert. Erreicht der Algorithmus ein Blatt, also einen Basisdatentyp, so ist der Stack vollständig und die Instanzen des Blattes können kopiert werden oder der Stack kann für ein späteres Kopieren gespeichert werden. Die Stacks für den betrachteten Beispieldatentyp sind in Abbildung 3.5 skizziert.

Das Packen

Das Packen der einzelnen Blätter erfolgt nun durch ein Abarbeiten des Stacks (Abb. 3.4). Für jede Ebene des Stacks wird ein Zähler (i) geführt, in dem die Anzahl der bereits kopierten Instanzen vermerkt werden. Zu Beginn ist der Zähler für jede Ebene zu Null initialisiert. Die Bearbeitung beginnt auf der obersten Ebene des Stacks. Hier werden alle *count* Instanzen des Basisblocks kopiert. Dabei wird die Position im Benutzerspeicher (*src*) um die Ausdehnung (*extent*) pro Kopie und die Position im Zielspeicher (*dest*) um die Länge der Kopie erhöht.

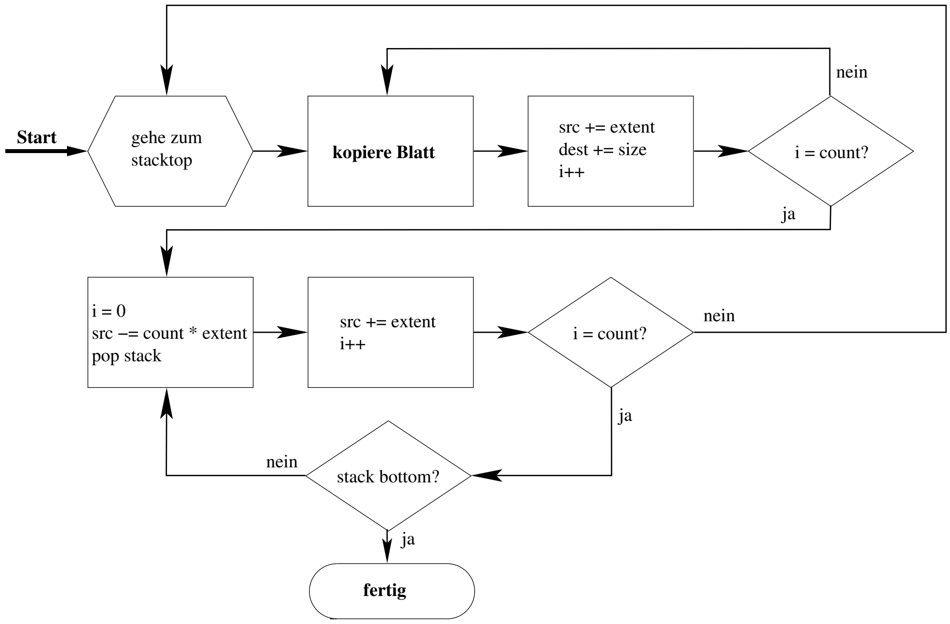

Abbildung 3.4: FF-Algorithmus, Durchlauf für ein Blatt

Danach wird der Zähler wieder zu Null gesetzt, die Position im Benutzerspeicher zurückgesetzt und die nächste Ebene vom Stack geholt *(pop stack)*. Der Zähler dieser Ebene wird erhöht und die Position im Benutzerspeicher angepaßt. Ist auch diese Ebene vollständig kopiert ($i = count$), wiederholen sich diese Schritte, bis eine Ebene gefunden wird, die noch nicht vollständig ist. In diesem Fall wird zum Anfang des Stacks zurückgesprungen und der Algorithmus kopiert wieder die Instanzen des Basisblocks. Wird keine Ebene mehr gefunden für die gilt $i \neq count$, sind alle Instanzen kopiert und der Algorithmus terminiert.
Im folgenden wird die genaue Implementation des Algorithmus in SCI-MPICH erläutert.

3.3 Implementation in SCI-MPICH

Das Konzept des *Flattening on the Fly* als Verfahren zum direkten Kopieren wurde in dieser Arbeit in drei Schritten in SCI-MPICH eingefügt.
In das MPI Runtime *MPIR* wurden zwei Funktionen eingefügt, die für den Aufbau der Stacks bei einem *MPI_Type_commit()* sorgen:

- `MPIR_build_ff (struct MPIR_DATATYPE *type, int *leaves)`

  Erstellt beim Aufruf die Stacks für den Datentyp *type* und gibt die Anzahl der gefundenen Blätter in *leaves* zurück. Die Stacks werden in einer verketteten Liste in der internen Datentypstruktur *struct MPIR_DATATYPE* gespeichert.

- `MPIR_free_ff (struct MPIR_DATATYPE *type)`

  Gibt den für die Stacks bereitgestellten Speicher des Typs *type* wieder frei.

In den ch_smi Gerätetreiber wurden die Funktionen zum direkten Packen/Entpacken eingefügt:

- ```
  MPID_SMI_Pack_ff (char *inbuf, struct MPIR_DATATYPE *dtype_ptr,
                    char *outbuf, int dest, int max, int *outlen)
  ```

  Packt einen Teil der Daten eines Datentyps aus dem Sendespeicher *inbuf*, dessen Struktur durch *dtype_ptr* gegeben ist, in den Nachrichtenpuffer *outbuf*. *max* gibt die Anzahl der zur Verfügung stehenden Bytes in *outbuf* an. *outlen* gibt an, wieviel Daten bereits kopiert wurden und wird von der Funktion aktualisiert.

- ```
  MPID_SMI_UnPack_ff (char *inbuf, struct MPIR_DATATYPE *dtype_ptr,
                      char *outbuf, int from, int max, int *outlen)
  ```

  Entpackt die Daten aus dem angegebenen Nachrichtenpuffer an die korrekten Positionen im Empfangspuffer. Die Parameter werden analog zu *MPID_SMI_Pack_ff()* übergeben. Hier ist *inbuf* der kontinuierliche Nachrichtenpuffer und *outbuf* ist der Empfangsspeicher in den die Daten, gemäß der durch den Typ *dtype_ptr* gegebenen Verteilung, geschrieben werden.

Vor dem Versenden erfolgt der Aufbau der benötigten Stacks innerhalb eines *MPI_Type_commit()* durch Aufruf von *MPIR_build_ff()*. Dies weicht etwas von dem in [8] vorgestellten Konzept ab, hat jedoch den Vorteil, daß beim Versenden bereits der Stack vorhanden ist und so etwas Zeit gespart werden kann. Bei der geringen Größe der verwendeten Datenstrukturen ist der zusätzliche Speicherbedarf vernachlässigbar. Allerdings macht dies eine gesonderte Behandlung der Kopierzähler nötig, diese können nicht mehr direkt im Stack gespeichert werden, da durchaus mehrere Sende-/Empfangsoperation gleichzeitig denselben Datentyp benutzen können.
Als zweiter Schritt erfolgt der Aufruf der direkten Packfunktion *MPID_SMI_Pack_ff()* innerhalb der normalen Sendefunktionen des ch_smi, wenn der Schalter *nc_enable* in der Gerätekonfiguration aktiviert ist *(nc_enable > 0)*[1]. Durch diesen Schalter ist es möglich, die Benutzung des direkten Packens komplett abzuschalten und das ursprüngliche Verhalten der SCI-MPICH zu erhalten.
Ist *nc_enable* gesetzt, wird das Erzeugen der Zwischenpuffer komplett unterdrückt und *MPID_SMI_Pack_ff()* ersetzt die *memcpy()* Aufrufe in den Sendeoperationen, die normalerweise Teile des bereits gepackten Speichers kopieren sollten. Das Empfangen verläuft analog dazu.

[1]siehe auch ≪Reihenfolge der gepackten Daten≫ auf Seite 24

Datenstrukturen der Implementation

Für den Algorithmus werden folgende Datenstrukturen benutzt, die alle nötigen Angaben für das Flattening on the Fly Verfahren bereitstellen. Eine Skizze der Datenstrukturen ist in Abbildung 3.5 zu finden.

- *Leaf list*

  Die einzelnen Blätter werden in einer einfach verketteten Liste gespeichert. Ein Zeiger auf diese Liste wird in der Struktur *struct MPIR_DATATYPE* für jeden Datentyp gespeichert.

```
typedef struct FF_STACK_T MPIR_FF_LIST_ITEM;
struct FF_STACK_T {
    MPIR_FF_STACK_ITEM *stack;
    int           contig_size;
    int                   top;
    int                   pos;
    MPIR_FF_LIST_ITEM   *next;
};
```

  Diese Struktur enthält einen Zeiger auf den dazugehörigen Stack **stack*, die Größe des Basisblocks *contig_size*, die Tiefe des Stacks *top* und die Startposition *pos* relativ zum Beginn des Datenpuffers.

- *Stack*

  Der Stack für ein einzelnes Blatt wird in einem Array gespeichert. Ein Eintrag dieses Arrays besteht aus folgender Struktur:

```
typedef struct {
    int   extent;
    int     size;
    int    count;
} MPIR_FF_STACK_ITEM;
```

  Auf dem Stack werden der Abstand *extent* und die Anzahl *count* der Basisblöcke gespeichert. Zusätzlich wird die Gesamtgröße der Daten innerhalb dieser Ebene, *size*, gespeichert, um den Wiedereintritt zu beschleunigen.

- *Counter Array*

  Die Zähler für die einzelnen Ebenen des Stacks werden in einem getrennten Array gespeichert, sie können nicht direkt in *struct MPIR_DATATYPE* gespeichert werden, da mehrere Instanzen einer Sende-/Empfangsfunktion denselben Datentyp benutzen können. Die Zähler werden von der Packroutine angelegt und verwaltet.

```
int *counter_array;
ALLOCATE (counter_array, int *, sizeof(int) * stacksize);
```

Aufbau des Stacks

Der Aufbau des Stacks für jedes Blatt erfolgt bei einem *MPI_Type_commit()* eines nicht-kontinuierlichen Datentyps. Der Typbaum wird von der Wurzel bis zu den Blättern durchgegangen und für jeden Knoten werden die Informationen auf dem Stack abgelegt.
Für jeden der verschiedenen Typen werden die Stackdaten gesondert berechnet. Der jeweils betrachtete Knoten enthält den Typ *type*.

- *CONTIGUOUS*
  - Stackitem
    ```
    extent = type->old_type->extent;
    count  = type->count;
    ```
- *VECTOR*
  1. Stackitem für Blocklänge
     ```
     extent = type->old_type->extent;
     count  = type->blocklen;
     ```
  2. Stackitem für Vektor
     ```
     extent = type->stride * type->old_type->extent;
     count  = type->count;
     ```
- *HVECTOR*
  1. Stackitem für Blocklänge
     ```
     extent = type->old_type->extent;
     count  = type->blocklen;
     ```
  2. Stackitem für Vektor
     ```
     extent = type->stride;
     count  = type->count;
     ```
- *INDEXED und HINDEXED*
  - Stackitem für einen Indexeintrag
    ```
    extent = type->old_type->extent;
    count  = type->blocklens[index];
    ```
- *STRUCT*
  - Stackitem für einen Struct-Eintrag
    ```
    extent = type->old_types[index]->extent;
    count  = type->blocklens[index];
    ```

Für die Vektortypen müssen zwei Einträge auf dem Stack abgelegt werden, in der aufgeführten Reihenfolge, zuerst der Eintrag für die Blocklänge dann der generelle Eintrag für den Vektor. Dies resultiert daraus, daß durch eine Blocklänge > 1 noch eine zusätzliche Wiederholung produziert wird.
Die Funktion zum Stackaufbau arbeitet iterativ, obwohl bei dem rekursiven Charakter des Typbaums eher eine rekursive Implementierung naheliegend wäre. Dies hat jedoch einzig und allein historische Gründe.
Ist die Funktion bei einem Blatt angekommen, wird der erstellte Stack noch vereinfacht (merging) und danach sortiert nach aufsteigender Basisblockgröße in die Liste der Blätter eingefügt.

Vereinfachung des Stacks (merging)

Es läßt sich zeigen, daß bei einigen Typdefinitionen unnötige Informationen auf dem Stack abgelegt werden. Alle Stackeinträge mit einem Zähler von eins sind überflüssig, wenn noch weitere Einträge vorhanden sind. Denn es gilt

$$count_1 \cdot \dots \cdot 1 \cdot \dots \cdot count_n \cdot contig\_size = count_1 \cdot \dots \cdot count_n \cdot contig\_size$$

Weiterhin können am Stacktop alle Einträge gelöscht werden, bei denen die Ausdehnung *extent* gleich der Größe *size* ist, wenn die Größe für den Basisblock entsprechend angepaßt wird. Solche Einträge weisen auf kontinuierliche Bereiche hin, die jedoch als nicht-kontinuierlich definiert wurden. Die Größe des Basisblocks berechnet sich dann zu:

$$contig\_size = contig\_size \cdot count_{stacktop}$$

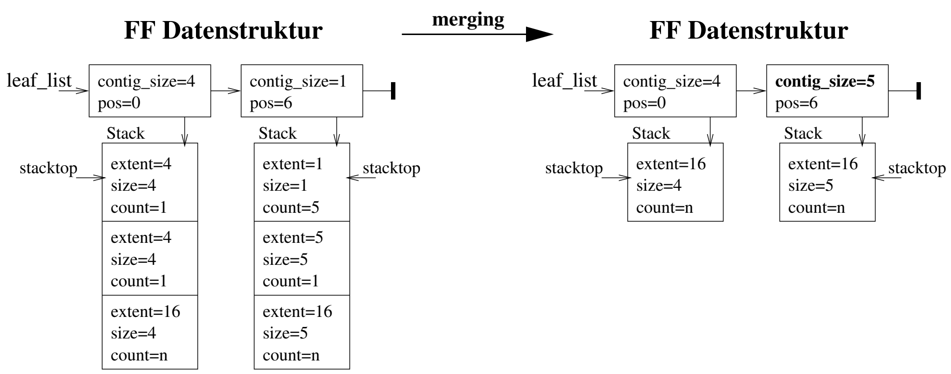

Abbildung 3.5: Stack vor und nach dem merging

In Abbildung 3.5 ist das Vereinfachen für den vorgestellten Beispieldatentyp skizziert. Zuerst wird der mittlere Eintrag mit einem *count = 1* gelöscht. Danach kann bei beiden Stacks der obere Eintrag gelöscht werden, da hier jeweils *size = extent* ist. Für den ersten Stack muß die Blockgröße *contig_size* nicht angepaßt werden, da der Zähler des gelöschten Eintrags 1 ist. Beim zweiten Stack wird *contig_size* mit dem *count* des gelöschten Eintrags multipliziert und wird zu 5.

Direktes Packen/Entpacken

Wenn der Stack für die einzelnen Blätter des Datentypbaumes aufgebaut ist, kann mit dem FF-Algorithmus das Kopieren der Daten erfolgen. Das Packen erfolgt hierbei in vier Schritten.

1. *Wiederfinden der Position im Benutzerspeicher*

   ```
   pos = find_pos (&start, &ff,counter_array);
   ```

   Die Anzahl der bereits kopierten Bytes wird in *start* übergeben, *ff* ist ein Zeiger auf das erste Blatt in der Liste und *counter_array* zeigt auf das Array der Zähler, das angepaßt wird.

2. *Kopieren der restlichen Daten eines Basisblocks*

   ```
   if (start > 0) copy_splitted_block(leaf_size - start);
   ```

3. *Kopieren der einzelnen Blätter (flattening on the fly)*

   ```
   do {
    copy_leaf_basic (pos, leaf);
    leaf = leaf->next;
   } while ((leaf != NULL) && (size_left > 0))
   ```

Die Schritte 1 und 2 sind nur notwendig, wenn die zu versendenden Daten nicht komplett in ein Nachrichtenpaket passen (Rendez-vous Protokoll). Schritt 1 sucht bei sukzessiven Aufrufen der Packfunktion die Position im Benutzerspeicher, liefert das nächste zu bearbeitende Blatt und paßt die verwendeten Zähler für den Stack korrekt an. Schritt 2 wird ausgeführt, wenn bei der vorhergehenden Kopieroperation nur ein Teil eines Basisblocks kopiert wurde. Dies ist immer dann der Fall, wenn die Länge einer Nachricht kein ganzzahliges Vielfaches der Basisblocklänge ist. In diesem Fall bricht Schritt 3 innerhalb eines Basisblocks ab. Das Kopieren der Blätter ist beendet, wenn entweder das letzte Blatt der Liste bearbeitet wurde, oder kein Platz mehr im Nachrichtenspeicher zur Verfügung steht *(size_left = 0)*.
Das Entpacken erfolgt hier analog, nur die Richtung des Kopierens muß umgedreht werden. Beim Packen wird, da hier ein Zugriff auf entferntes Shared Memory erfolgt, das Makro *REMOTE_COPY* benutzt, welches für das Schreiben über SCI optimiert ist. Beim Entpacken ist der Speicherzugriff lokal, daher wird das normale *memcpy()* benutzt.

Wiedereintritt *(find_pos)*

Bei sukzessiven Aufrufen der Packfunktion wird die Anzahl der bereits kopierten Bytes übergeben. Mit dieser Information ist es möglich die Position zu finden, bei der der Kopiervorgang erneut ansetzen muß. Dies erfolgt in zwei Schritten:

1. *Überspringen bereits kopierter Blätter*

```
/* Anfang der Blattliste */
leaf = first_leaf;

while (leaf->total_size < start_length) {
 start_length -= leaf->total_size;
 leaf = leaf->next;
}

pos = leaf->pos;
```

   Wenn die Gesamtgröße aller Instanzen eines Blattes *(total_size)* kleiner ist als die Anzahl der bereits kopierten Bytes *(start_length)*, dann ist dieses Blatt bereits komplett kopiert. Es wird zum nächsten Blatt gewechselt und *start_length* wird um die Größe aller Instanzen des Blattes dekrementiert. Danach wird der Positionszeiger auf die erste Position des resultierenden Blattes im Speicher gesetzt.

2. *Suchen der Position im Stack und anpassen der Zähler*

```
for (sp = stackbottom; sp <= stacktop; sp++) {
  counter_array[sp] = start / stack[sp].size;
  start = start % stack[sp].size;
  pos += stack[sp].extent * counter_array[sp];
}
```

   In der Schleife werden alle Ebenen des Stacks durchgegangen, angefangen bei der Ebene mit der größten Ausdehnung, also dem *stackbottom*. Im Zähler für jede Ebene in *counter_array[sp]* wird vermerkt, wie oft eine Ebene bereits komplett abgearbeitet wurde. Dies läßt sich durch eine *ganzzahlige* Division der bereits kopierten Bytes durch die Gesamtgröße der Ebene erreichen. Die Anzahl der verbleibenden Bytes für die nachfolgenden Ebenen ergibt sich dann aus dem Rest der Division. Die Position ergibt sich aus der Ausdehnung der Ebene multipliziert mit der vorher berechnet Anzahl der bereits kopierten Instanzen.

Kopieren eines Blattes *(copy_leaf_basic)*

Der eigentlich Packalgorithmus für die einzelnen Blätter behandelt stets ein komplettes Blatt:

```
do {
 top = leaf->stacktop; /* Am Stacktop anfangen */

 /* Schleife für das Kopieren auf Ebene der Basisblöcke */
 for (top->i = 0; top->i < top->count; top->i++) {

  /* Reicht der Platz nicht mehr für ein komplettes Blatt,
     wird ein Teil kopiert und die Funktion verlassen */
  if (size_left < leaf->size) {
   copy_part(size_left);
   return;
  }

  /* Kopieren eines Basisblocks */
  REMOTE_COPY (dest,src,leaf->size);

  /* Anpassen der Positionen im Benutzerspeicher
     und Shared Memory */
  src += top->extent;
  dest += leaf->size;
 }

 /* Anpassen der Zähler */
 while (top->i == top->count) {
  src -= top->count*top->extent;
  top->i = 0;
  top = top->prev; /* pop stack */ if (top==NULL) break;
  src += top->extent;
  top->i++;
 }
} while (top!=NULL)
```

In der Implementation ergeben sich einige Unterschiede zu dem hier angegebenen Beispielcode. Zum einen werden statt der Zeigeroperationen für den Stack Array-Zugriffe benutzt, da der Stack aus Leistungsgründen als Array gespeichert wird. Die Zähler (i) sind nicht als Bestandteil des Stacks gespeichert, sondern werden für jede Instanz der Kopierfunktion in einem weiteren Array *(counter_array)* vorgehalten, da mehrere Packoperation gleichzeitig denselben Datentyp und somit denselben Stack benutzen könnten. Außerdem ist noch eine spezielle Behandlung kleiner Basisblöcke unter Benutzung eines lokalen Zwischenpuffers implementiert.

Lokaler Puffer

Da aus den theoretischen Überlegungen und nach ersten Tests klar war, daß ein direktes Kopieren für kleine Blockgrößen sehr ineffizient ist, wurde ein Mechanismus eingebaut, mit dem sich kleine Basisblöcke in einem lokalen Zwischenpuffer sammeln lassen. Hierzu wird vor dem Kopieren eines Blattes geprüft, ob die Größe kleiner als ein einstellbarer Schwellenwert ist. Dieser Schwellenwert kann in der Gerätekonfiguration mit dem Parameter *nc_min <Größe>* angepaßt werden. In einem solchen Fall wird ein lokaler Zwischenspeicher alloziert, der so groß ist wie der restliche im Nachrichtenspeicher zur Verfügung stehende Platz. Die Auswahl dieser Größe hat den Vorteil, daß keine weitere Überprüfung der Speichergrenzen notwendig wird, lediglich das Ziel der Kopiervorgänge wird angepaßt. Alle nachfolgenden Kopiervorgänge in diesem Durchlauf von *MPID_SMI_Pack_ff()* laufen dann über den Zwischenspeicher, ist dieser vollständig gefüllt, kann er mit dem für SCI-Operationen optimierten Makro *REMOTE_COPY()* in das entfernte Shared Memory kopiert werden. Dadurch ist der Nachrichtenspeicher komplett gefüllt und der Durchlauf wird beendet.

Kopieroperationen

Bei Tests hat sich eine starke Beschränkung der Kopieroperationen über SCI herausgestellt. Die Bandbreite vermindert sich um Größenordnungen, wenn das Ziel des Kopierens keine auf 32 Byte Wortgrenzen *(32 Byte alignment)* ausgerichtete Adresse ist. Da es besonders bei Datentypen mit mehreren Blättern vorkommen kann, daß diese Bedingung beim Kopieren nicht erfüllt wird, mußte ein Mechanismus eingebaut werden, der die negativen Auswirkungen abfängt. Zu diesem Zweck erfolgt bei allen Kopieroperationen eine Überprüfung auf das alignment. Ist dies nicht gegeben, wird zuerst ein Teilstück kopiert, um zur 32 Byte Grenze zu kommen, erst danach wird der komplette Rest des Blocks kopiert. Abbildung 4.3 im nächsten Kapitel zeigt die positiven Auswirkungen dieser Methode.

Reihenfolge der gepackten Daten

Einige Fragen wirft die Reihenfolge der Daten im gepackten Puffer auf. Diese ist bei Benutzung des FF-Algorithmus in der vorliegenden Form nicht mehr als sequentiell anzunehmen (Abb. 3.6), wenn der Typ aus mehr als einem Basistyp abgeleitet ist (bei *indexed*, *hindexed* und *struct* Typen). Dadurch verhalten sich Programme, die von einer sequentiellen Anordnung ausgehen, unter Umständen fehlerhaft. Das mit dieser Arbeit implementierte Verfahren zum direkten Kopieren verhält sich in dieser Beziehung nicht konform mit dem MPI Standard *(Type Matching Rules [6])*.
Eine fehlerhafte Kommunikation kann jedoch vermieden werden, wenn zum Versand und Empfang jeweils ein nicht-kontinuierlicher Datentyp benutzt wird, der aus identischen Basisdatentypen aufgebaut ist. Sollte ein zum Standard vollkommen konformes Verhalten notwendig sein, ist es möglich, den Schalter *nc_enable* in der Device-Konfiguration auf *1* zu stellen, was den Aufbau der FF-Stacks für Datentypen unterdrückt, die ein fehlerhaftes Verhalten verursachen könnten. Typen in denen nur die Konstruktoren *contiguous*, *vector* und *hvector* vorkommen, werden unabhängig davon stets korrekt mit dem direkten Verfahren behandelt. Ist klar, daß der Ablauf der Applikation nicht von der Anordnung der gepackten Daten abhängt, kann *nc_enable* auf *2* gestellt werden,

um die volle Leistung der Optimierung zu erhalten.

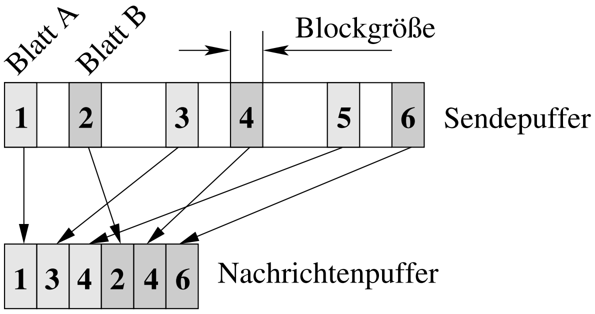

Abbildung 3.6: Reihenfolge der Daten nach dem Packen

Einbindung in die Sende-/Empfangsroutinen von SCI-MPICH

Wird eine der MPI Sende-/Empfangroutinen aufgerufen, wird zuerst überprüft, ob der benutzte Datentyp kontinuierlich ist oder nicht. Im letzteren Fall wird bei aktiviertem *nc_enable* das Bereitstellen der Zwischenpuffer unterbunden und der Datentyp an die Sende-/Empfangsfunktion des Device übergeben. Hier folgt dann der Aufruf der direkten Packfunktion. Der Ablauf innerhalb der Bibliothek für Versand und Empfang ist in Abbildung 3.7 skizziert.

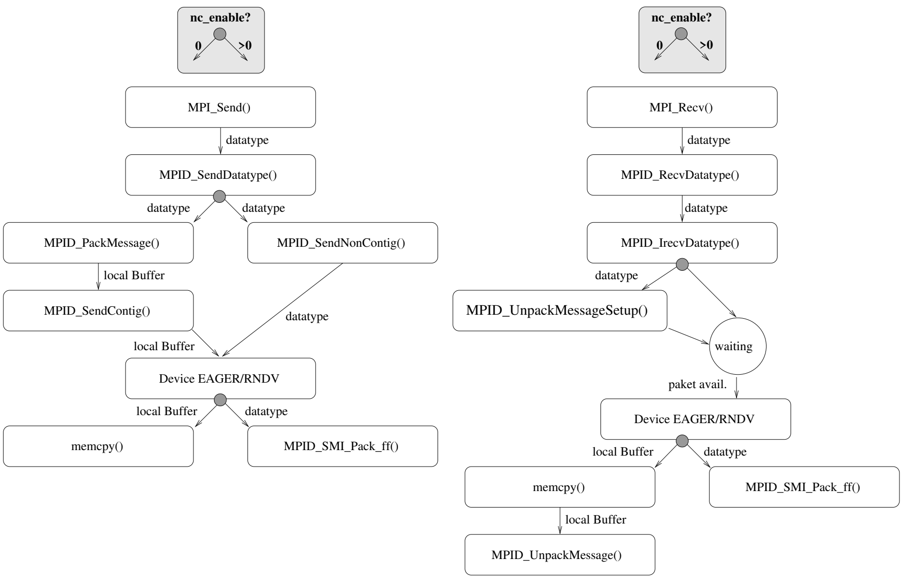

Abbildung 3.7: Ablauf für MPI_Send() und MPI_Recv() bei nicht-kontinuierlichen Datentypen

Kapitel 4

Leistungsevaluation

Zur Beurteilung der Leistungsfähigkeit des direkten Kopierens wurden Messungen mit verschiedenen Datentypen durchgeführt. Die Messungen wurden auf verschiedenen Hardwareplattformen durchgeführt und mit den Ergebnissen der alten Implementation und einiger anderer Implementationen verglichen.

4.1 Messungen auf Pentium III Systemen

Die Messungen für verschiedene Verteilungen wurden im *Ping-Pong*-Verfahren zwischen zwei Prozessen durchgeführt. Die verschickte Datenmenge wurde, um einen zu großen Einfluß des Prozessorcache auszuschalten, auf 10 Megabyte festgelegt, die Puffergröße der Sende- und Empfangspuffer ist daher je nach Test entsprechend größer. Es wurden jeweils 10 Iteration von Versand/Empfang durchlaufen, um einen stabilen Meßwert zu bekommen.

Die Durchführung erfolgte auf Dual Prozessor SMP Systemen, mit 800 MHz Intel Pentium III Prozessoren, auf Boards mit ServerWorks ServerSet III-LE Chipsatz, 448 MB RAM, 64 Bit/66 MHz PCI Bus und Dolphin D330 PCI-SCI Adaptern.

Vektor und *double-strided* Vektor

Es wurden zuerst zwei häufig anzutreffende Verteilungen untersucht, die bei der Aufteilung von Arrays benutzt werden. Eine *Vektor-Verteilung*, bei der Datenblöcke fester Länge von stets gleichlangen Lücken getrennt werden, und eine *Double-strided*-Verteilung, die aus einem Vektor wie im ersten Fall besteht, der nochmals mit einer weiteren Lückengröße aneinandergereiht wird.

```
total_size = 10485760;
count = total_size / blocklen;
stride = 2;
dstride = 2;

/* strided vector */
MPI_Type_vector (count,blocklen,stride*blocklen,MPI_CHAR,&vector);
```

```
/* double strided vector */
MPI_Type_vector (count/dstride,blocklen,stride*blocklen,
                 MPI_CHAR,&oldtype);
MPI_Type_vector (dstride, 1, stride, oldtype, &dsvector);
```

Die Vektortypen wurden für den ersten Test gewählt, da in der ursprünglichen Implementation speziell für Vektortypen eine optimierte Packroutine existiert und sich eine neue Packroutine vor allem daran messen lassen muß. Außerdem ist der FF-Algorithmus generisch aufgebaut, d.h. die Leistung hängt nicht von den verwendeten Datentypen ab, sondern nur von der Lage und Größe der Basisblöcke.

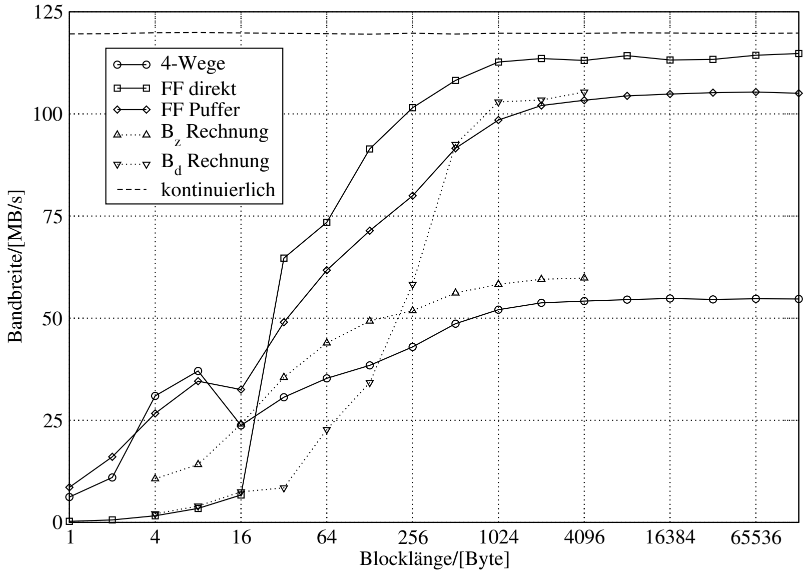

Abbildung 4.1: Durchsatz für Vektortypen im Vergleich

Abbildung 4.1 zeigt den resultierenden Durchsatz für die Vektortypen in Abhängigkeit von der Blockgröße der kontinuierlichen Basisblöcke. Zwischen dem einfachen und dem double-strided Fall gab es, wie zu erwarten, keine Unterschiede, daher ist nur der Vektor-Typ aufgetragen. Die Gesamtmenge der zu übertragenden Daten blieb bei der Messung konstant bei 10MB. Die Obergrenze für den Durchsatz liegt bei dem Wert für eine kontinuierliche Übertragung der Gesamtmenge. Zum Vergleich wurden nochmals die berechneten Werte aus Kapitel 2.3 (B_z und B_d) mit aufgetragen.
Das Verfahren zum direkten Kopieren *(FF direkt)* ist dem alten 4-Wege Verfahren ab einer Blockgröße von 32 Byte deutlich überlegen. Ab 1024 Byte liegt es nur gering unter der maximalen Leistung, die mit einem Kopiervorgang eines kontinuierlichen Typs auf diesem System erreicht werden kann. Die Messung des FF-Verfahrens unter Benutzung des lokalen Zwischenpuffers ist als *FF Puffer* aufgetragen. Bei dieser Messung wurden alle Blöcke gepuffert. Bei diesem Datentyp schneiden sich die Kurven für *FF direkt* und *FF Puffer* bei unter 32 Byte. Für Vektortypen ist auf dem Testsystem ein *nc_min* von *32* optimal.

Unterschiedliche Blockgrößen

Um den Einfluß von unterschiedlichen Basisblockgrößen auf die Übertragungsleistung zu untersuchen wurde ein zusammengesetzter Typ konstruiert, der aus einem Vektor von einem Indexed-Typ von einem Struct-Typ besteht. Der Struct Typ enthält dabei zwei unterschiedliche Basisblöcke. Die Messung erfolgte dann bei einer konstanten Größe des einen Basisblocks von 1 Byte, während die Größe des anderen Blocks verändert wurde.

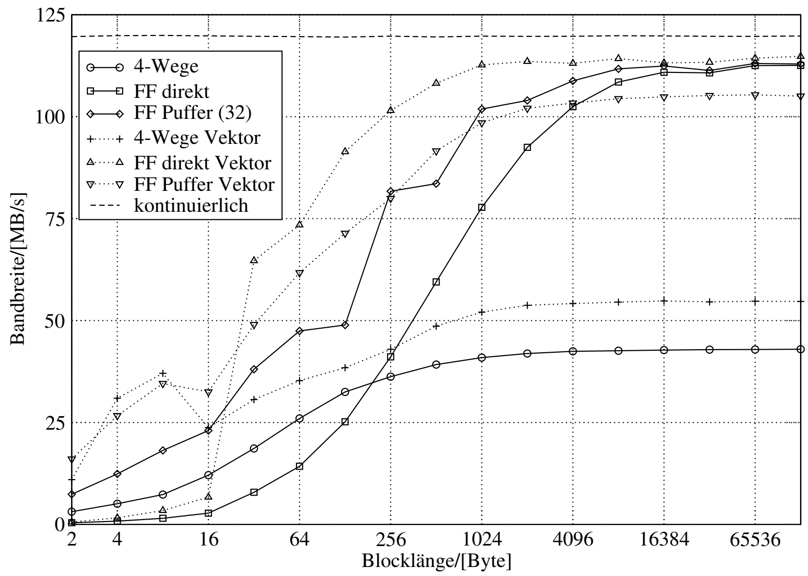

Abbildung 4.2: Durchsatz für einen gemischten Typ

```
blen1 = 1; /* Größe des einen Blocks konstant*/
blen2 = blocklength; /* Zweite Blocklänge variabel */

/* Für den Struct-Typ */
int          st_blens[2] = { blen1, blen2 };
MPI_Aint     st_disps[2] = { 0, blen1 * (stride + 1) };
MPI_Datatype st_types[2] = { MPI_CHAR, MPI_CHAR };

/* Für den Indexed-Typ */
int           i_blens[2] = { 3, 1 };
MPI_Aint      i_disps[2] = { 0, 5 };

MPI_Type_struct (2, st_blens, st_disps, st_types, &tmp);
MPI_Type_indexed (2, i_blens, i_disps, tmp, &tmp2);
MPI_Type_vector (count/4, 1, stride, tmp2, type);
```

Die Bandbreite geht durch das zusätzliche Kopieren der kleinen Blöcke im Vergleich zum einfachen Vektor etwas zurück. Allerdings kann dieser Effekt durch das Verwenden des Zwischenpuffers abgemildert werden und die Leistung liegt deutlich über dem 4-Wege Verfahren. In der Device-Konfiguration wurde für die Messung *FF Puffer (32) nc_min* auf *32* eingestellt. Diese Einstellung ist für den hier betrachteten Datentyp optimal. Zum Vergleich wurden nochmals die Meßwerte eines Vektors punktiert aufgetragen.

Ungünstige Blockgrößen

Um das Verhalten bei ungünstigen Blockgrößen zu untersuchen, die keinem Vielfachen einer 2-er Potenz entsprechen, wurden einfache Vektoren benutzt, deren Blocklängen auf 3^n Byte und $7 \cdot 3^n$ Byte gesetzt wurden.

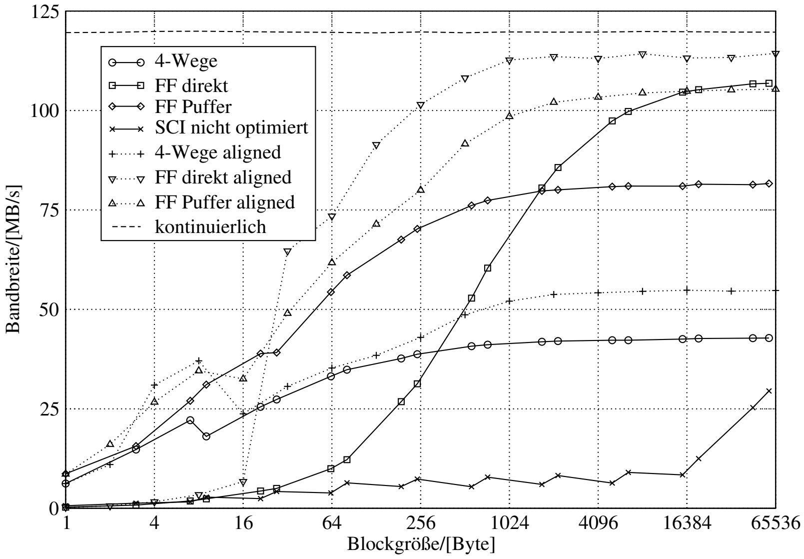

Abbildung 4.3: Durchsatz für ungünstige Blockgrößen

Ohne besondere Behandlung der nicht ausgerichteten Kopieroperationen beim direkten Kopieren *(SCI nicht optimiert)* liegt der Durchsatz weit unter dem 4-Wege Verfahren, unter 10 MB/s für Blöcke bis 16384 Byte Größe. Erst durch die Optimierung wird eine bessere Bandbreite erreicht *(FF direct). Diese Optimierung wird standardmäßig benutzt und wurde hier nur zur Verdeutlichung der in Kapitel 3.3 angestellten Überlegungen temporär ausgeschaltet.* Durch Einschalten des Zwischenpuffers *(FF Puffer)* wird die Bandbreite bei Blöcken unter 1024 Byte nochmals angehoben. Zum Vergleich sind die Meßwerte für „günstige“ Blockgrößen (2-er Potenzen) punktiert aufgetragen.

4.2 Vergleich mit anderen Plattformen

In Tabelle 4.2 sind die verschiedenen Hardwareplattformen aufgelistet, mit denen die Leistung des direkten Verfahrens verglichen wurde [11]. Aufgeführt sind jeweils der Rechnertyp, die benutzte MPI Implementation und der Verbindungstyp. In der letzten Spalte sind die Codes angegeben, mit denen sich die zugehörigen Messungen in den Diagrammen auffinden lassen. Auf allen Plattformen lief der einfache Vektor Benchmark aus Kapitel 4.1, die Resultate sind in

Abbildung 4.4 aufgeführt. Der Übersicht halber sind die bekannten Ergebnisse aus Abbildung 4.1 nicht noch einmal eingetragen.

System	MPI	Verbindung	ID
Cray T3E-1200	Cray	custom	C
Sun Fire 6800 24-way SMP	Sun HPC 3.1	Gigabit Ethernet	F-G
750 MHz, 64 bit PCI		shared memory	F-s
Pentium III Dual SMP	MP-MPICH	SCI	M-S
800 MHz, 64 bit PCI	1.2.1 beta	shared memory	M-s
Pentium III Xeon	LAM 6.5.4	fast ethernet	X-f
Quad SMP, 550 MHz		shared memory	X-s
Pentium II Dual SMP	SCore 2.4.1	Myrinet 1280	S-M
400 MHz, 32 bit PCI		shared memory	S-s

Tabelle 4.1: Hardwareplattformen für die Evaluierung

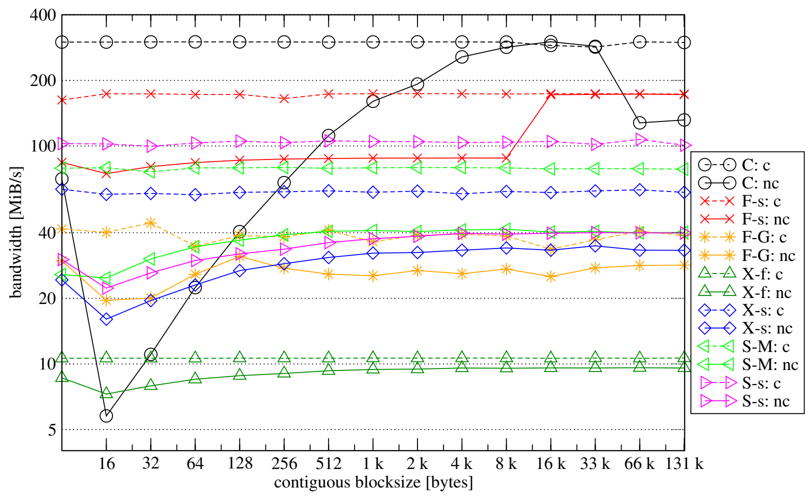

Abbildung 4.4: Leistung der nicht-kontinuierlichen Übertragung im Vergleich

In Abbildung 4.5 ist die Relation zwischen kontinuierlicher und nicht-kontinuierlicher Übertragungsleistung als *Effizienz* aufgeführt, die bekannten Werte des vorher betrachteten Pentium III Systems sind als M-S (SCI) und M-s (shared memory) eingetragen. Die Werte zeigen, daß keine der getesteten MPI Implementationen eine durchgängige Optimierung für nicht-kontinuierliche Datentypen besitzt. Die in dieser Arbeit vorgestellte Optimierung erreicht ab einer Blockgröße von 256 Byte eine Effizienz von über 0.9 und bietet damit eine optimale Leistung für nicht-kontinuierliche Übertragungen. Die Cray T3E erreicht eine Effizienz von fast 1 nur für Blockgrößen zwischen 8 und 32 Kbyte, außerhalb dieses Bereichs fällt die Leistung stark ab. Die hohe Effizienz der Fast Ethernet und Gigabit Ethernet benutzenden Implementationen wird dadurch relativiert, daß die niedrige Bandbreite dieser Verbindungen den größten Teil der Übertragungszeit hervorruft und weitere Optimierungen daher kaum ins Gewicht fallen.

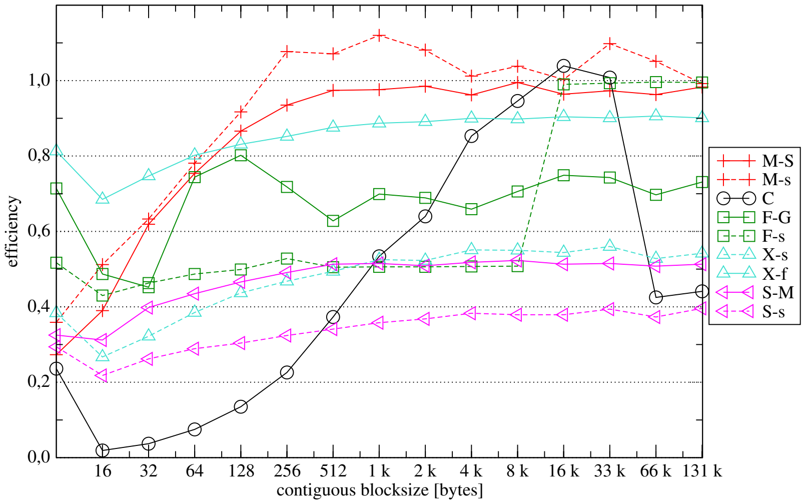

Abbildung 4.5: Effizienz der Übertragung

4.3 Verhalten bei sehr großen Datenmengen

Das Anlegen der Zwischenpuffer im 4-Wege Verfahren bringt für große Datenmengen eine unangenehme Grenze mit sich. Zusätzlich zu dem vom User verwalteten nicht-kontinuierlichen Sendespeicher, der aus den eigentlichen Daten (S_{Data}) und den Lücken (S_{Stride}) besteht, muß die Bibliothek hier noch einen weiteren Puffer allozieren, der die gesamte Datenmenge aufnehmen kann (S_{Data}). Da eine Auslagerung von Speicher aus dem physikalischen RAM stets zu starken Leistungseinbußen führt, ist zu fordern, daß die gesamte benutzte Speichermenge komplett in das RAM paßt. Bei dem 4-Wege Verfahren ergibt sich dann folgende Grenze für die benutzbare Datenmenge:

$$\underbrace{S_{Data} + S_{Stride}}_{Sendespeicher} \quad \underbrace{+S_{Data}}_{Zwischenpuffer} \quad \leq S_{RAM}$$

Bei direktem Versenden/Empfangen entfällt der Zwischenspeicher und die nutzbare Speichergröße erhöht sich um S_{Data} auf:

$$\underbrace{S_{Data} + S_{Stride}}_{Sendespeicher} \leq S_{RAM}$$

Ein Test mit einem einfachen Vektortypen auf den beschriebenen Testsystemen belegt dies. Der nutzbare physikalische Speicher war 448 MByte groß, die Datentypen wurden so definiert, daß der Sende- und Empfangsspeicher mit 400 MByte vollständig in diesen Speicher paßt:

	Blocklänge	Datenmenge	Sendespeicher	Bandbreite
4-Wege	4096	200 MByte	400 MByte	2.703 MB/s
FF direkt	4096	200 MByte	400 MByte	93.720 MB/s

Die Übertragungsleistung bricht, bedingt durch starke Auslagerungsaktivität des Systems, beim 4-Wege Verfahren um Größenordnungen ein.

4.4 Cache Verdrängung

Das Anlegen der zusätzlichen Zwischenpuffer beim 4-Wege Verfahren hat auch Auswirkungen auf die Verfügbarkeit der Daten im Cache. Wird im Programm ein Speicherbereich bearbeitet, der komplett in den Cache des Prozessors paßt, profitieren Speicherzugriff von der Lagerung der Daten im Cache. Ein Aufruf der 4-Wege Sendefunktion kann durch das Anlegen der neuen Puffer diesen Speicherbereich komplett verdrängen und damit einen späteren Zugriff auf den Bereich verlangsamen. Um dies zu testen, wurde ein einfaches Programm benutzt, welches zuerst einen allozierten Speicherbereich beschreibt und dann die MPI_Send() Funktion aufruft, um Daten aus dem Bereich zu versenden. Nach abgeschlossener Übertragung wurde die Bandbreite des anschließenden Zugriffs auf diesen Bereich gemessen. In Abbildung 4.6 ist die Bandbreite dieser Messung über der Größe des Sendespeichers aufgetragen. Die verschickte Datenmenge lag bei ca. der Hälfte des Sendespeichers, als Datentyp wurde ein Vektor benutzt. Man sieht deutlich, daß beim direkten Kopieren die Cache-Verdrängung und damit der Sprung in der Bandbreite später einsetzt. Das FF-Verfahren mit Zwischenpuffer liegt zwischen dem 4-Wege und dem direkten Verfahren. Auch hier wird zusätzlich Speicher alloziert, doch nicht in der Größe der gesamten Datenmenge, sondern des benutzten Rendez-vous Puffers.

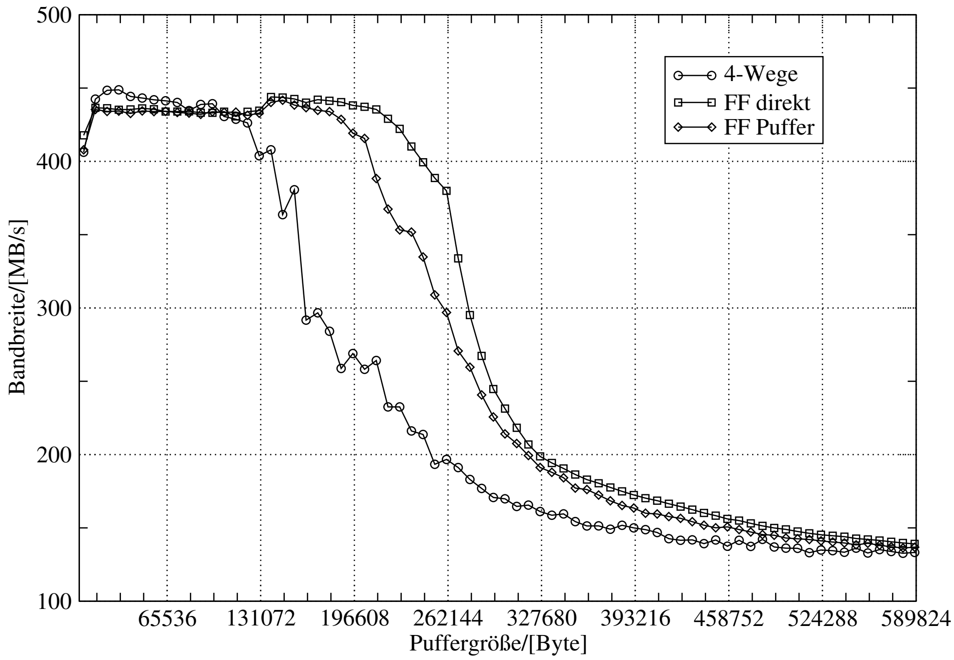

Abbildung 4.6: Einsetzen der Cache-Verdrängung beim 4-Wege Kopieren

Kapitel 5

Zusammenfassung und Ausblick

In dieser Arbeit wurde ein neues Verfahren zum Packen/Entpacken von nicht-kontinuierlichen Daten vorgestellt und die Einbindung als direktes Sende-/Empfangsverfahren in SCI-MPICH dargestellt. Das Versenden von nicht-kontinuierlichen Daten konnte mit den vorgestellen Änderungen stark beschleunigt werden. Ab einer minimalen Blockgröße von ca. 512 Bytes auf den betrachteten Testsystemen nutzt das Verfahren nahezu die volle Bandbreite des physikalischen Kanals aus. Zwischen dieser Größe und einer minimalen Größe von 32 Bytes ist es dem alten 4-Wege Verfahren immer noch deutlich überlegen. Darunter bricht das Verfahren jedoch als Folge der physikalischen Eigenschaften der Netzwerkhardware stark ein. Dieser Effekt konnte durch das Verwenden eines lokalen Zwischenpuffers soweit abgemildert werden, daß es auf dem Niveau des alten Verfahrens liegt.

Weitere Entwicklungen

Der benutzte Packalgorithmus könnte auch als genereller Packalgorithmus in MPICH eingefügt werden. Die Ergebnisse des Verfahrens mit Zwischenpuffer weisen darauf hin, daß er dem alten Algorithmus auch bei lokalen Operation überlegen ist.
Um eine optimale Übertragungsleistung zu erzielen, kann der Wert *nc_min* in der Device Konfiguration passend zum benutzten Datentyp angepaßt werden. Je nach Aufbau des Datentyps kann dieser Wert stark variieren. Um eine optimale Leistung zu erhalten, ist es zur Zeit notwendig, die Übertragung zu testen und von Hand die richtige Grenze einzustellen. Auch ist nur eine generelle Einstellung für die laufende Applikation möglich, werden mehrere verschiedene Datentypen parallel benutzt, kann es unmöglich sein, eine für alle Typen passende Einstellung zu finden. Hier wäre eine automatische Einstellung, oder wenigstens eine explizite Einstellung bezogen auf den Datentyp wünschenswert.
Das Fehlen der Konformität zum MPI Standard bei bestimmten komplexen Datentypen schränkt die Verwendbarkeit des Verfahrens praktisch nicht ein. Es wird in den meisten Fällen möglich sein, zum Senden und Empfangen denselben oder einen für diesen Zweck äquivalenten Datentyp zu benutzen. Optional

läßt sich jetzt schon die Konformität durch ein Ausschalten der Optimierung für problematische Datentypen erreichen. Insgesamt wäre es jedoch wünschenswert, durch eine Umstrukturierung der Kopierroutinen eine MPI Konformität für alle Fälle zu erreichen, um stets den vollen Leistungsgewinn des direkten Verfahrens nutzen zu können. Dies wird noch Gegenstand für weitere Überlegungen sein.

Abschließend kann gesagt werden, daß die hier vorgestellte Optimierung von SCI-MPICH Anwendern die Möglichkeit gibt, die im MPI Standard definierten Optionen für benutzerdefinierte Datentypen effizient zu nutzen. Unter Beachtung der oben erwähnten Einschränkungen, ist es jetzt möglich auch mit nichtkontinuierlichen Datenverteilungen eine Übertragungsbandbreite in der Größenordnung einer kontinuierlichen Übertragung zu erhalten. Ein solches generelles Verhalten ist in keiner anderen bekannten MPI Implementation verfügbar und stellt somit eine echte Innovation dar.

Literaturverzeichnis

[1] IEEE Std 1596-1992. *IEEE Standard for Scalable Coherent Interface (SCI).* The Institute of Electrical and Electronics Engineers, Inc., 1993.

[2] Mike Ashworth. OCCOMM Benchmark Code. In *Workshop on Scalable Parallel Computing on Cray Systems (Berlin) and 8th RAPS Workshop (Offenbach)*, November 1995.

[3] Stefan Erben. *DMA-gestützes asynchrones Übertragungsprotokoll.* Lehrstuhl für Betriebssystem der RWTH-Aachen, 2000.

[4] William D. Gropp, Ewing Lusk, and Debbie Swider. Improving the Performance of MPI Derived Datatypes. In *Message Passing Interface Developer's and User's Conference (MPIDC '99)*, March 1999.

[5] Hermann Hellwagner and Alexander Reinefeld (Eds.). *Architecture and Software for High-Performance Clusters*, volume 1734 of *LNCS–State-of-the-Art Surveys*, chapter 1. Springer Verlag, 1999.

[6] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Knoxville, Tenesse, 1995.

[7] Robert Sedgewick. *Algorithmen in C.* Addison-Wesley, 1992.

[8] Jesper Larsson Träff, Rolf Hempel, Hubert Ritzdorf, and Falk Zimmermann. Flattening on the Fly: efficient handling of MPI derived datatypes. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 1697:109–116, 1999.

[9] Joachim Worringen. SCI-MPICH: The Second Generation. In *Proc. "SCI Europe 2000" (conference stream of EuroPar 2000)*, pages 10–20. Technische Universität München, August 2000.

[10] Joachim Worringen and Thomas Bemmerl. MPICH for SCI-connected Clusters. In *Proc. "SCI Europe '99" (in conjunction with EuroPar '99)*, pages 3–11, September 1999.

[11] Joachim Worringen, Andreas Gäer, Frank Reker, and Thomas Bemmerl. Exploiting Transparent Remote Memory Access for Non-Contiguous- and One-Sided-Communication (to be published). 2001.