Diplomarbeit

Optimierte Implementierung ausgewählter kollektiver Operationen unter Ausnutzung der Hardwareparallelität des InfiniBand™ Netzwerkes

Maik Franke

Betreuer: Dipl.-Inf. Torsten Höfler

Betreuender Hochschullehrer: Prof. Dr.-Ing. Wolfgang Rehm

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbstständig angefertigt und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Chemnitz, 30. April 2007

Maik Franke

Zusammenfassung der Arbeit

Ziel der Arbeit ist eine optimierte Implementierung der im MPI-1 Standard definierten Reduktionsoperationen MPI_Reduce(), MPI_Allreduce(), MPI_Scan(), MPI_Reduce_scatter() für das InfiniBand™ Netzwerk. Hierbei soll besonderer Wert auf spezielle InfiniBand™ Operationen und die Hardwareparallelität gelegt werden.

InfiniBand™ ermöglicht es Kommunikationsoperationen klar von Berechnungen zu trennen, was eine Überlappung beider Operationstypen in der Reduktion ermöglicht. Das Potential dieser Methode soll modelltheoretisch als auch praktisch in einer prototypischen Implementierung im Rahmen des Open MPI Frameworks erfolgen. Das Endresultat soll mit vorhandenen Implementierungen (z.B. MVAPICH) verglichen werden.

Inhaltsverzeichnis

1. Einleitung

Die hinter der Arbeit stehende Motivation soll im ersten Teil der Arbeit zum Ausdruck gebracht werden. Anschließend wird als Einführung in das Themengebiet eine kurzer Überblick über die MPI Implementierung OpenMPI, das der Arbeit zugrund e liegende Netzwerk InfiniBand™ und die Kommunikationsmodelle gegeben.

1.1 Motivation

Kollektive Kommunikation ist eine sehr wichtige und häufig genutzte Komponente in MPI. Eine fünfjährige Studie an der Universität von Stuttgart hat gezeigt, dass über 40% der Ausführungszeit von Routinen des Message Passing Interface (MPI) auf die kollektiven Kommunikationsroutinen MPI_Allreduce und MPI_Reduce entfallen [23]. Eine Optimierung dieser und generell aller kollektiven Operationen wäre im Bezug auf die Perf ormance von parallelen Programmen, die auf MPI aufbauen, von sehr großem Nutzen.

1.2 OpenMPI

MPI ist die Abkürzung für Message Passing Interface, welches einen Standard für das Schreiben paralleler Programme darstellt. Für diesen Standard gibt es versch iedene Implementierungen, unter anderem ist hier FT-MPI [41,42] von der Universität von Tennessee, LA-MPI [43,44] vom Los Alamos National Laboratory und LAM/MPI [45,46] von der Universität Indiana zu nennen. Um den Anforderungen der Forschungsgemeinschaft von MPI gerecht zu werden, wurde OpenMPI [47] entwickelt. Es ist eine MPI-2 Implementation, die auf den Forschungen und Erfahrungen mit den oben genannten Implementierungen beruht.

OpenMPI ermöglicht durch seine modulare Komponentenarchitektur (Modular Component Architecture – MCA) eine ideale Umgebung für die Optimierung kollektiver Operationen auf das InfiniBand™ Netzwerk. Open MPI besitzt mehrere Frameworks, welche bestimmte Aufgaben übernehmen. Eine Komponente kann implementiert werden, um diese Auf gabe auszuführen und eine Instanz der Komponente zur Laufzeit nennt sich Modul. Manche Frameworks können viele aktive Module zu einem Zeitpunkt besitzen. In Bild … sind die wichtigen Teil der Architektur von Open MPI zu finden.

Open MPI besteht aus drei Softwareschichten:

1. **MPI** - MPI Schicht (MPI Layer)
2. **RTE** - Laufzeitumgebung (Run Time Environment)
3. **MCA** - Modulare Komponenten Architektur (Modular Component A rchitecture)

Die MPI Schicht ist eine Anpassungsschicht, die den MPI Standard in die d arunter liegenden Funktionalitäten integriert (hauptsächlich RTE und MCA).

Die RTE Schicht stellt zur Laufzeit Dienste zur Verfügung. Beide Schichten müssen nicht verändert werden, um neue kollektive Operationen zu integrieren, somit wird nicht weiter auf diese Schichten eingegangen.

Die MCA Schicht in ein Komponenten Framework, welches andere Schichten managt, die darunter liegen. Dies geschieht durch Bereitstellung verschiedener Services. Jeder wichtige Funktionsbereich hat ein zugeordnetes Komponenten Framework, um verschiedene Komponenten zu managen, die verwandte oder identische Aufgaben behandeln. Jede Komponente wird durch ihre Schnittstelle beschrieben und bietet den höheren Schichten des Frameworks bestimmte funktionale Dienste an.

OpenMPI ermöglicht durch seine modulare Komponentenarchitektur eine ideale Umgebung für die Optimierung kollektiver Operationen auf das InfiniBand ™ Netzwerk. Open MPI besitzt mehrere Frameworks, welche bestimmte Aufgaben übernehmen. Eine Komponente kann implementiert werden, um diese Aufgabe auszuführen und eine Instanz der Komponente zur Laufzeit nennt sich Modul. Manche Frameworks können viele aktive Module zu einem Zeitpunkt besitzen.

Das Framework mit allen Schichten und mehrere Komponenten Frameworks mit ihren Modulen werden in Bild 1 gezeigt

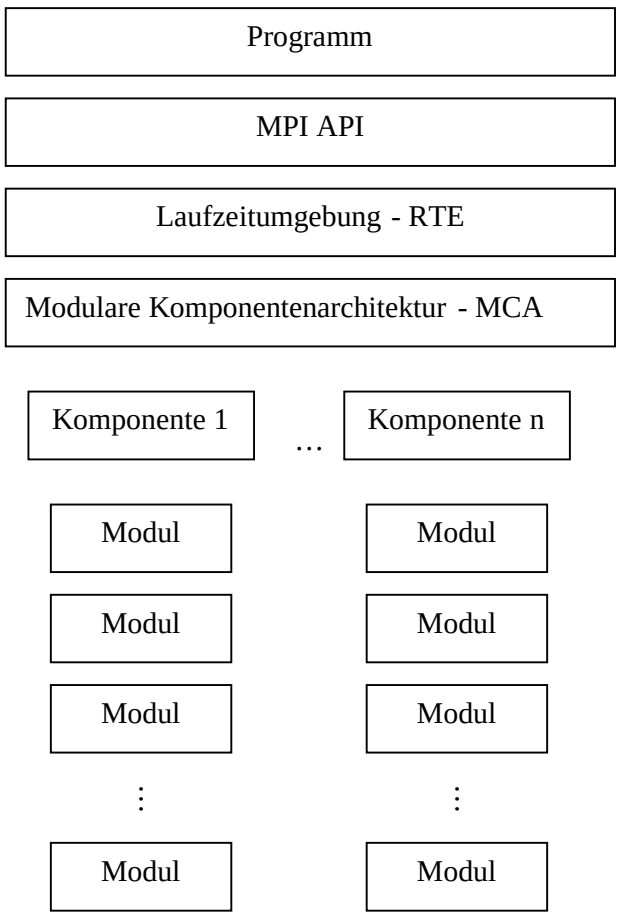


Bild 1. Architektur von Open MPI

In Open MPI sind unter anderem folgende Frameworks implementiert:

- **PTL** – Punkt-zu-Punkt Transport Schicht (Point-to-point Transport Layer), welche aus netzwerkspezifischen Modulen besteht, die für den untersten (low level) Datentransfer verantwortlich sind.
- **PML** – Punkt-zu-Punkt Management Schicht (Point-to-point Management Layer), welche mehrere Transportdienste für die MPI Schicht zur Verfügung stellt.
- **COLL** – Kollektives Framework, welches die Module für die kollektiven Operationen bereitstellt.

Für diese Arbeit ist das COLL Framework von großer Bedeutung. Deshalb wird im folgenden Abschnitt die generelle Struktur von Komponenten beschrieben, um die Struktur von COLL verstehen zu können.

1.2.1 Der Lebenszyklus einer Komponente

Eine Komponente durchläuft während ihrer Existenz innerhalb der MCA fünf Stadien: Selektion, Initialisierung, Checkpoint/Neustart, Normale Benutzung und Beendigung.

In Bild 2 wird der Lebenszyklus einer Komponente mit den verschiedenen Stadien dargestellt.

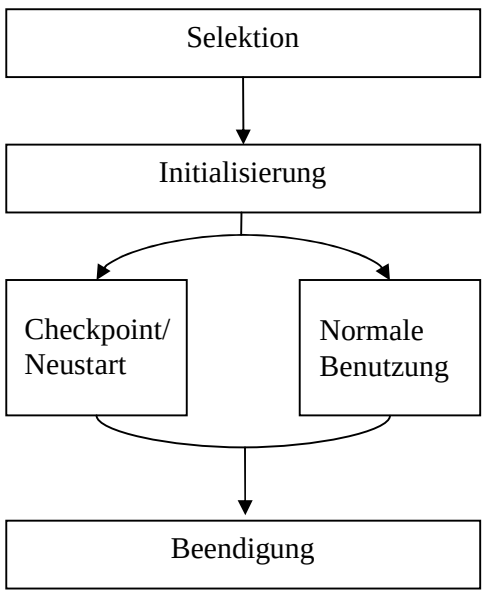


Bild 2. Lebenszyklus einer Komponente

Die COLL Komponente wird nach der Initialisierung als Modul bezeichnet, da jeder Kommunikator mit einem einzelnen COLL Modul verknüpft ist, wobei aber den gleichen Quellcode benutzen. Daraus ergibt sich, dass zu einer Zeit nur eine Instanz der COLL Komponente aktiv sein kann, aber jeder Kommunikator seinen eigenen Zustand seines Moduls hat.

Die Selektion wird beim Erzeugen eines neuen Kommunikators (einschließlich MPI_COMM_WORLD) durchlaufen, was in den MPI API Funktionen MPI_INIT, MPI_COMM_CREATE, MPI_COMM_COMMIT, MPI_COMM_SPLIT und MPI_COMM_DUP passiert.

Die **mca_coll_<name>_comm_query** Funktion jeder verfügbaren Komponente wird vom Framework abgefragt, um eine Liste von Funktionszeigern zu den angebotenen Funktionen und der Priorität (zwischen 0 und 100) zurückzugeben.

Die Initialisierungsfunktion wird auch zum Testen genutzt, ob bestimmte Eigenschaften innerhalb eines erzeugten Kommunikators zur Verfügung stehen. Sollten dabei bestimmte Eigenschaften, wie in unserem Fall eine InfiniBand ™ Verbindung zu allen Knoten, nicht zur Verfügung stehen, schaltet sich die Komponente selbst ab und gibt einen Nullzeig er zurück. Vom COLL Framework wird die Komponente ausgewählt, die die höchste Priorität zurückliefert.

Damit geht die Komponente mit der höchsten Priorität in den Zustand Initialisierung über und das COLL Framework ruft die dazu passende **mca_coll_<name>_module_init** Funktion. Damit kann die Komponente alle wichtigen Dinge wie interne Datenstrukturen initialisieren, die für die Benutzung kollektiver Funktionen notwendig sind.

Die initialisierten Daten müssen dann mit der Kommunikationsstruktur verknüpft we rden durch Umhängen des c_coll_basic_data Zeigers auf den Beginn der Datenstruktur, die Komponente gibt einen Zeiger an das Framework zurück, wo die nach der Initialisierung zur Verfügung stehenden Funktionen aufgelistet sind. Sollten bestimmte Funktionen nicht zur Verfügung stehen, müssen diese Funktionszeiger Nullzeiger sein, damit dem COLL Framework signalisiert wird, dass es für diese Funktionen Basisfunktionen nutzen soll, die von der Komponente namens basic angeboten werden.

Im Zustand der normalen B enutzung werden die angeforderten kollektiven Operationen (zum Beispiel MPI_Reduce()) ausgeführt. Dazu müssen vorher gespeicherte Daten aus dem Kommunikator extrahiert werden. Die Funktionen werden über ihren Funktionszeiger gerufen, der in der Initialisi erungsphase dem COLL Framework zur Verfügung gestellt wurde.

Der Schritt zur Beendigung erfolgt durch die Funktion **mca_coll_<name>_module_finalize**. Er fordert das Modul auf, alle benutzten Datenstrukturen zu beseitigen und das Netzwerk von Daten zu entleeren.

1.3 InfiniBand™

Die InfiniBand™ Architektur (IBA) ist ein neuer Industriestandard für Server I/O und Inter-Server Kommunikation. Er wurde von der InfiniBand Trade Association (IBTA) entwickelt, um Anforderungen wie Zuverlässigkeit, Verfügbarkeit, Performance und Skalierbarkeit besser erfüllen zu können, als dies mit herkömmlichen Bus-orientierten I/O Strukturen möglich ist. Dieser Standard bildet die Grundlage für die Implementierungen von InfiniBand.

InfiniBand ist eine neue Technologie, die höheren Datendurchsatz, geringere Latenz, neue Server/Storage-Architekturen, bessere Plug-And-Play-Funktionalität (z.B. automatische Erkennung) und auch Dienste wie Quality of Services (QoS) verspricht.

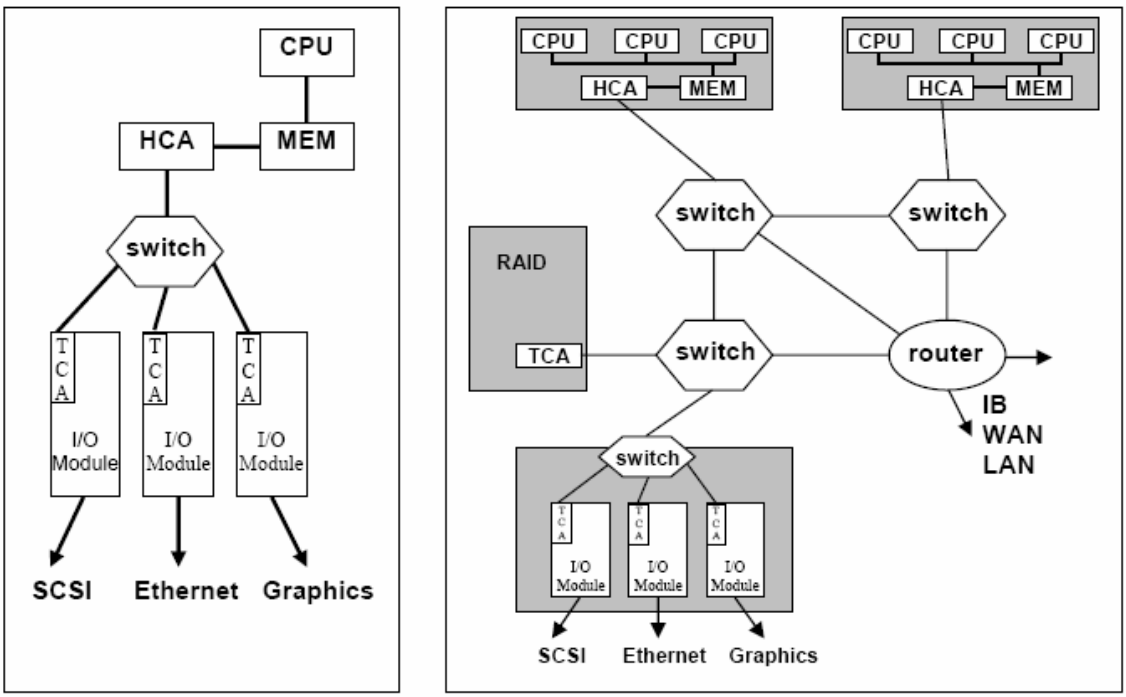


Bild 3. IBA-Topologien

1.3.1 Einleitung

Wie man im Bild 3 sehen kann, handelt es sich bei InfiniBand um ein geswitchtes Netzwerk, in dem alle Endknoten durch Punkt-zu-Punkt-Verbindungen mittels Switches verbunden sind. Das gesamte Kommunikationsnetzwerk (Switches, Router & Leitungen) wird bei Infin iband als Fabric (Gewebe) bezeichnet.

Dieses Fabric bietet eine Reihe von Vorteilen gegenüber dem klassischen Bus. So sind mehrere Verbindungen gleichzeitig möglich und falls eine Verbindung ausfallen sollte, kann die Kommunikation über vorhandene Ausweichrouten trotzdem fortgeführt werden.
Die Hosts sind mit einem HCA (Host Channel Adapter) an InfiniBand angeschlossen, bei der Peripherie übernimmt dies ein TCA (Target Channel Adapter).
Die kleinste komplette IBA-Einheit ist ein Subnetz (~ 64k Knoten pro Subnetz). Mehrere Subnetze werden durch Router zu einem großen IBA Netzwerk verbunden. Durch einen Router wird auch ein Übergang zu anderen Netzen (LAN, WAN,…) ermöglicht.

In den nächsten Abschnitten möchte ich einen kurzen Überblick über die Kommunikatio n und das Schichtenmodell der InfiniBand™ Architektur geben. Nähere Informationen zur Architektur finden sie unter anderem in [48, 49, 50].

1.3.2 Kommunikation

1.3.2.1 Verbindungsarten

IBA stellt mehrere Verbindungsarten zur Verfügung. Die Verbindungs art legt fest, wie zwei Knoten miteinander kommunizieren.

Verbindungsart	Verbindungsorientiert	Empfangsbestätigung	Protokoll
Reliable Connection	Ja	Ja	IBA
Unreliable Connection	Ja	Nein	IBA
Reliable Datagram	Nein	Ja	IBA
Unreliable Datagram	Nein	Nein	IBA
RAW Datagram	Nein	Nein	RAW

1.3.2.2 Queuing

Die Kommunikation zwischen Nutzern eines Channel Adapters (Consumer) erfolgt mittels Queues, wobei diese immer paarweise existieren (Queue Pair – QP); eine Queue zum Senden und eine zum Empfangen.

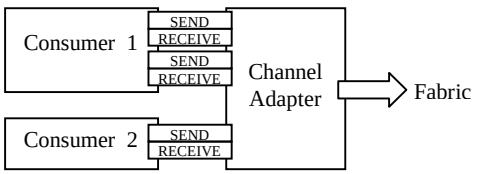


Bild 4. Channel Adapter mit zwei Nutzern und drei QPs

Mittels Work Requests werden die Queues und somit auch die Kommunikation gesteuert. Die Work Requests werden von einem Consumer gesendet und erzeugen ein Work Queue Element (WQE) in der entsprechenden Queue (Send oder Receive). Wenn der Channel Adapter ein WQE abgearbeitet hat, erzeugt er sofort ein Completion Queue Element (CQE) in der Completion Queue. Jedes CQE enthält alle Informationen, um eine Operation abschließen zu können.

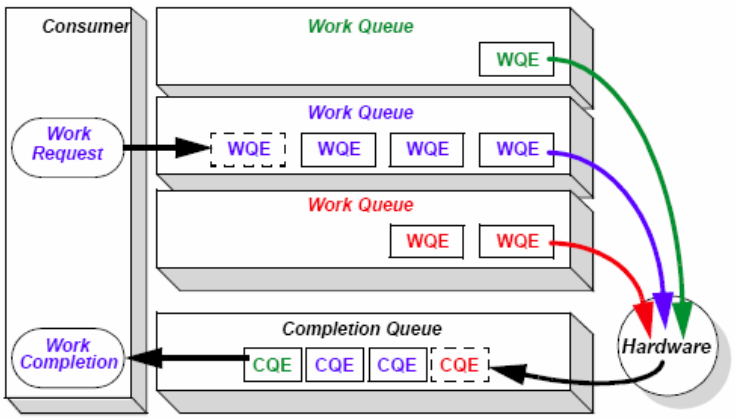


Bild 5 [48]. Consumer Queuing Modell

Jeder Consumer kann mehrere voneinander unabhängige Queue Pairs und dazugehörige Completion Queues erzeugen. Die Queues sind somit ein virtuelles Kommunikationsinterface, das von der Hardware bereitgestellt wird. Die Queues stellen eine private Ressource des Consumers dar.

Auf die Send Queue lassen sich folgende Operationen anwenden:

Send: Im WQE wird ein Speicherblock des Consumers angegeben, der von der Hardware an die Zieladresse geschickt wird. In der Receive Queue des Ziels ist angegeben, wo die Daten abgelegt werden sollen.

Remote Memory Access (RDMA):
Zusätzlich zu den Daten der Send Operation wird noch eine Speicheradresse des Ziels angegeben. Es gibt RDMA-Read, RDMA-Write und Atomic Operationen. RDMA-Write bedeutet, dass die Hardware Daten vom Consumer Speicher zum Remote Speicher überträgt. RDMA-Read bedeutet, dass die Hardware Daten vom Remote Speicher zum Consumer Speicher überträgt. Atomic ist ein Read eines 64 Bit Word des Ziels, das eventuell (swap if equal oder add) noch ein Update des Inhalts auslöst.

Memory Binding (MB):
MB erlaubt es dem Consumer einen Speicherblock anzugeben, den er mit anderen teilt, sowie die Lese- und Schreibrechte dafür zu festzulegen. Daraus resultiert ein Memory Key (R_Key), den der Consumer an Remote-Knoten verschickt, damit diese dann den Memory Key in ihren RDMA Operationen verwenden können.

Auf die Receive Queue lässt sich eine Operation anwenden, die den Receive Data Buffer spezifiziert:
Ein Receive WQE legt fest, wo die empfangenen Daten von einem anderen Consumer abgelegt werden sollen, der eine Send Operation ausführt. Beim erfolgreichen Empfang wird ein Receive WQE verbraucht, die Daten werden an der entsprechenden Stelle abgelegt und ein CQE wird erzeugt.

1.3.2.3 Partitionen

Eine Partition definiert eine Gruppe von Knoten, die miteinander kommunizieren können, wobei jeder Port eines Endknotens mindestens Mitglied einer Partition ist. Partitionen werden mittels Partition Keys gesteuert (siehe nächsten Ab schnitt). Sowohl Switches als auch Router können optional dafür genutzt werden, die Partitionierung durchzusetzen, indem sie P_Keys erhalten und daraufhin alle Pakete mit anderen P_Keys verwerfen.

1.3.2.4 Verschiedene Schlüssel

Die Schlüssel (Keys) werden in den Nachrichten verwendet, um verschiedene Zugriffsrechte zu ermöglichen:

- Memory Keys (L_Key und R_Key):
  Die Schlüssel werden vom Channel Adapter vergeben und ermöglichen die Verwendung von virtuellen Speicheradressen und die Regelung für den Speicherzugriff. Wenn ein Consumer einen Teil seines Speichers für den RDMA-Zugriff freigibt, erhält er einen L_Key und einen R_Key vom Channel Adapter. Der L_Key wird für die Bereitstellung von lokalem Speicher für die QP verwendet. Der R_Key wird an andere entfernte Consumer gesendet, um ihnen den RDMA-Zugriff zu ermöglichen.

- Management Key (M_Key):
  Der M_Key wird vom SM an die Ports der Channel Adapter vergeben. Der SM kann jedem Port einen anderen M_Key zuweisen. Sobald er aktiviert ist, werden Managementpakete ohne korrekten M_Key verworfen.

- Partition Key (P_Key):
  Der P_Key wird durch den Subnetz Manager vom Partition Manager vergeben. Jeder Port eines Channel Adapter hat eine Tabelle von P_Keys. QPs, die miteinander kommunizieren wollen, müssen der gleichen Partition angehören und somit einen gemeinsamen P_Key haben. Der P_Key wird in jedem Paket mitgeschickt. Jeder Switch besitzt eine P_Key Tabelle für Managementnachrichten und kann optional eine Filterung der Nachrichten nach P_Keys unterstützen.

- Baseboard Management Key (B_Key):
  Der B_Key wird vom Baseboard Manager vergeben und hat die gleiche Aufgabe für bestimmte MADs wie der M_Key für die Managementpakete vom SM.

- Queue Key (Q_Key):
  Der Q_Key wird vom Channel Adapter vergeben und regelt die Zugriffsr echte bei Übertragungen mittels Unreliable und Reliable Datagram. Während der Einrichtung der Kommunikation werden die Q_Keys für bestimmte QPs ausgetauscht und dann bei der Kommunikation zwischen den QPs immer mitgesendet.

1.3.2.5 Adressierung

Jeder Endknoten kann einen oder mehrere Channel Adapter haben. Ein Channel Adapter kann wiederum ebenfalls einen oder mehrere Ports besitzen. Außerdem hat der Channel Adapter eine Reihe von QPs.

Jede QP besitzt eine Queue Pair Nummer (QPN), die für die korrekte Z uordnung von Paketen an die QPs erforderlich ist. Die QPN ist deshalb in allen Paketen (außer RAW) enthalten.

Ein Port hat einen Local ID (LID), der vom Subnetz Manager des lokalen Subnetzes vergeben wird und mindestens einen Global ID (GID) im Format ein er IPv6 Adresse. In einem Subnetz sind die LIDs eindeutig und werden für die richtige Verteilung der Pakete im Subnetz von den Switches verwendet. Eine Paket enthält immer den Source LID (SLID), der die Quelle angibt, und einen Destination LID (DLID), der das Ziel angibt. GIDs sind auch global eindeutig und werden bei Paketen mit optionalem Global Route Header (GRH) benutzt. Der GRH, der ebenfalls wieder GIDs für Quelle und Ziel enthält, wird von den Switches ignoriert und nur von den Routern ausgewertet.

Jeder Channel Adapter sowie jeder seiner Ports hat einen Globally Unique Identifier (GUID), der vom Hersteller vergeben wird. Der GUID und ein 64bit Subnetzpräfix ergeben zusammen den global gültigen GID.

1.3.2.6. Virtual Lanes

Virtual Lanes (VL) ermöglichen mehrere logische Verbindungen auf einer einzigen physikalischen Verbindung. Dabei repräsentiert eine Virtual Lane einen Sende - und einen Empfangsbuffer eines Ports. Alle Ports müssen VL_{15} unterstützen, die für das Subnetz Management reserviert ist.

Außerdem muss mindestens eine Daten VL (VL_0) unterstützt werden. Außerdem können weitere 14 Daten VLs (VL_1 bis VL_{14}) bereitgestellt werden. Die Anzahl der von einem Port genutzten Daten VLs wird vom Subnetz Manager so festgelegt, dass beide Endknoten eine r Verbindung die gleiche Anzahl nutzen. Solange die Festlegung nicht geschehen ist, wird nur VL_0 genutzt.

Der Port verhindert mit einer separaten Flusssteuerung auf jeder VL, dass großes Trafficaufkommen auf einer VL die Kommunikation auf einer anderen VL beeinflusst.

Der Service Level im Header eines Paketes bestimmt, welche VL tatsächlich von einem Paket benutzt wird.

1.3.3. Schichtenmodell

Die Funktionsweise der InfiniBand™ Architektur kann als Schichtenmodell beschrieben werden, wobei die Protokolle der Schichten voneinander unabhängig sind und jede Schicht auf Leistungen der darunter liegenden Schicht angewiesen ist.

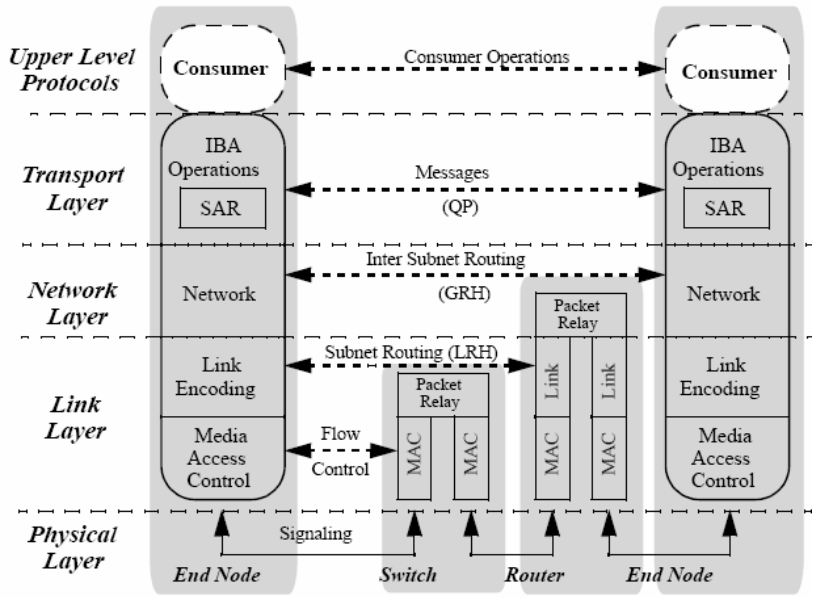


Bild 6 [48]: Schichtenmodell der IBA

1.3.3.1 Bitübertragungsschicht (Physical Layer)

Die Bitübertragungsschicht ist für die physische Übertragung verantwortlich. Sie ist unter anderem für die richtige Kodierung der Bits sowie das Einfügen von Start - und Endzeichen zur Einrahmung von Paketen verantwortlich.

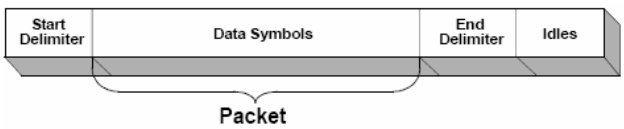


Bild 7 [48]: IBA Packet Framing

1.3.3.2 Subnetzschicht (Link Layer)

Die Subnetzschicht ist für das Senden und Empfangen von Daten auf Paketebene verantwortlich. Diese Schicht ist unter anderem für die Adressierung, die Verteilung der Pakete innerhalb des Subnetzes, die Flusssteuerung und die Fehlererkennung vera ntwortlich.

Auf der Subnetzschicht unterscheidet man zwei verschiedene Pakettypen:

Link Management Pakete:

Diese Pakete sind für die Flusssteuerung und dem Parameterabgleich zwischen den Ports an jedem Ende eines Links verantwortlich. Die LMPs werde n dafür in der Subnetzschicht erzeugt und auch wieder verbraucht.

Datenpakete:

Diese Pakete sind für die IBA Operationen verantwortlich. Im folgenden Bild ist der allgemeine Aufbau eines Datenpaketes dargestellt, wobei diverse Header optional sind.

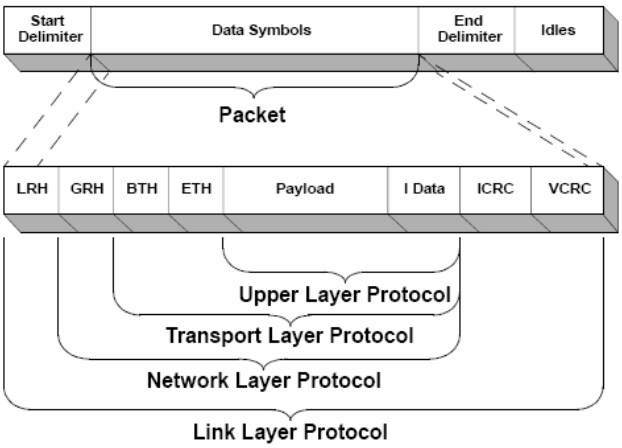


Bild 8 [48]: Allgemeiner Aufbau eines Datenpaketes

Ein Paket besitzt immer einen Local Route Header (LRH). In diesem sind der SLID und DLID (LIDs von Quelle und Ziel) enthalten. Falls das Paket in ein anderes Subnetz geroutet werden soll, enthält die DLID die LID eines Routers.

Der Invariant CRC (ICRC) ist für die Fehlererkennung aller Bereiche des Paketes verantwortlich, die sich während des Durchlaufs durch das Fabric nicht ändern. Der Variant CRC (VCRC) ist für alle Bereiche des Paketes verantwortlich, die sich beim Durchlauf ändern.

1.3.3.3 Vermittlungsschicht (Network Layer):

Die Vermittlungsschicht ist für das Routing zwischen Subnetzen verantwortlich. Dafür ist der Global Route Header (GRH) erforderlich. Der GRH enthält den GID von Quelle und Zie l. Wenn ein Paket über Subnetzgrenzen hinweg verschickt werden soll, enthält der LRH als DLID den LID eines Routers im aktuellen Subnetz. Der Router ersetzt den LRH und schickt das Paket somit an den nächsten Router. Dies wird solange wiederholt, bis das S ubnetz des Ziels erreicht ist. Der letzte Router trägt den LID des Ziel in den neuen LRH ein.

Router verändern auch den GRH, jedoch sind die GIDs von Quelle und Ziel durch den ICRC geschützt und bleiben unverändert.

1.3.3.4 Transportschicht (Transport Layer)

Die Transportschicht teilt eine Nachricht in Pakete auf und ist dafür verantwortlich, dass das Paket an die richtige QP geliefert wird.

Im Base Transport Header (BTH), der in allen Paketen außer bei RAW Datagram vorhanden ist, sind das Ziel QP, der Opcode, die Paketsequenznummer (PSN) und die Partition angegeben. Der Opcode gibt an, ob das Paket das erste, letzte, ein mittleres oder ein einziges Paket ist und welche Operation ausgeführt wird (Send, RDMA Read / Write / Atomic). Beim Kommunikationsaufbau wird die PSN initialisiert und immer erhöht, wenn das QP ein neues Paket erstellt. Das Ziel QP überwacht die PSN und überpüft damit, ob ein Paket verlorengegangen ist.

Zusätzlich zum BTH existieren noch Extended Transport Headers (ETH), die sich je nach Verbindungsart und Opcode unterscheiden oder ganz fehlen.

1.4. Kommunikations- und Berechnungsmodelle

Zur Analyse und Bewertung von verschiedenen Algorithmen zur Implementierung einer bestimmten MPI-Funktion wie z.B. MPI_Reduce werden so genannte parallele Kommunikationsmodelle verwendet. Diese sind für die Voraussage der theoretischen Laufzeit eines Algorithmus notwendig und bilden somit eine Basis für das Design und die Analyse der parallelen Algorithmen.

1.4.1 Einleitung

Ein gutes Modell zeichnet sich durch zwei Punkte aus, es beinhaltet die kleinstmögliche Anzahl von Parametern, ist aber dennoch in der Lage, die Komplexität des zu untersuchenden Systems ausreichend genau zu erfassen und zu beschreiben. Eine zu hohe Anzahl von Parametern erschwert den Umgang mit dem Modell, da dieses zu komplex wird und eventuell der Zugang zu den Parametern schwierig sein könnte.

Da bei den kollektiven Operationen wie MPI_Reduce, MPI_Allreduce, MPI_Reduce_scatter und MPI_Scan der Algorithmus aus einem Kommunikations- und Berechnungsteil besteht, müssen beide Aspekte in dem Modell berücksichtigt werden.

In diesem Abschnitt sollen zuerst verschiedene Modelle miteinander verglichen werden, um deren Vor- und Nachteile herausfiltern zu können. Diese Modelle basieren auf standardisierten Systemparametern, für deren Berechnung bestimmte Methoden erprobt sind [5][6]. Anschließend wird ein Modell namens LogfP vorgestellt, welches genauer evaluiert wird.

Es existiert auch eine Analyse über kollektive Kommunikationskosten im InfiniBand™ [3].

1.4.2 Hockney Modell

Das Hockney Modell [7] nimmt an, dass die Zeit für das Senden einer Nachricht der Größe m zwischen zwei Knoten $\alpha + \beta$ m beträgt. Hierbei steht α für die Latenz jeder Nachricht und β für die Übertragungszeit pro Byte oder das Reziproke der Netzwerkbandbreite. Netzwerk Congestion kann mit Hockney nicht modelliert werden.

1.4.3 PRAM Modell

PRAM Modell. Dieses Modell [14] wurde von Fortune et al. schon 1978 vorgeschlagen. Es ist das einfachste bekannte parallele Modell, enthält aber einige Fehler in der Genauigkeit. Es stammt vom RAM Modell ab, welches auf dem „Von Neumann“ Modell [15] basiert. Es wird durch P charakterisiert, der Anzahl Prozessoren, die sich einen gemeinsamen Speicher teilen. Es wird angenommen, dass alle Prozessoren synchron laufen und jeder Prozessor auf jeden Speicher in einem Schritt Zugriff hat. Alle Kosten für die Parallelisierung werden ignoriert, wodurch eine ideale parallele Komplexität des Algorithmus angenommen wird. Für die Einfachheit dieses Modells müssen mehrere Nachteile akzeptiert werden, zum Beispiel, dass die Kommunikation zwischen Prozessoren kostenlos ist. Deshalb ist dieses Modell für die Bewertung kollektiver Algorithmen ungeeignet.

1.4.4 BSP Modell

Das Bulk Synchronous Parallel (BSP) Modell [16] wurde 1990 von Valiant vorgeschlagen. Hierbei wird der Algorithmus in mehrere konsekutive Superschritte zerlegt, wobei jeder Superschritt aus einer Berechnungs- und einer Kommunikationsphase besteht. Alle Prozessoren starten zu Beginn jedes Superschrittes synchron. In der Berechnungsphase kann der Prozessor nur Berechnungen mit Daten in seinem lokalen Speicher vornehmen. Der Prozessor kann in der Kommunikationsphase Daten mit anderen Knoten austauschen, wobei er in jedem Superschritt höchstens h Nachrichten empfangen und h Nachrichten senden kann. Für die Kommunikation werden Kosten von $g * h$ (g entspricht der Bandbreite) veranschlagt.

Ein Nachteil dieses Modells ist die implizite Synchronisation. Außerdem muss jeder Superschritt lang genug sein, um h Nachrichten empfangen und senden zu können, wodurch manche Knoten am Ende des Schritts unbeschäftigt warten müssen. Aufgrund dieser Nachteile ist eine Verwendung zur Beurteilung kollektiver Operationen nicht geeignet.

1.4.5 C^3 Modell

Das C^3 Modell [17] wurde von Hambrusch et. al. vorgeschlagen. Es arbeitet ebenfalls wie das BSP Modell mit der Aufteilung des Algorithmus in mehrere Superschritte. Jeder dieser Superschritte besteht aus einer lokalen Berechnung gefolgt von der Kommunikation. Superschritte starten direkt nach Beendigung des vorangegangenen Superschrittes synchron. Das Modell erschließt die Komplexität von Kommunikation, und Berechnung, es wird auch der Unterschied zwischen blockenden und nicht blockenden Empfangen betrachtet, aber dennoch ist dieses Modell für die Bewertung kollektiver Algorithmen nicht geeignet, da es Nachrichten nur in Paketen bestimmter Länge austauschen kann und aufgrund fehlender Parameter für Clock Speed und Bandbreite dieses Modell nur für identische Prozessor - und Netzwerkbandbreite gültig ist.

1.4.6 LogP Modell

Das LogP Modell [8] wurde von Culler et al. 1993 vorgeschlagen. Es wurde als Erweiterung des PRAM Modells entwickelt, um den veränderten Bedingungen für paralleles Rechnen zu entsprechen. Es beschreibt verschiedene Aspekte von Maschinen, die als Sammlung von kompletten Computern verstanden werden, wobei jeder Computer aus einem oder mehreren Prozessoren, Cache, Hauptspeicher und Netzwerkverbindung besteht. Es beschreibt das Netzwerk mit den Parametern Latenz L, Overhead o, Gap pro Nachricht g und der Anzahl der in die Kommunikation involvierten Knoten P. Die Zeit für das Senden einer Nachricht zwischen zwei Knoten beträgt im LogP Modell L + 2o. Das Modell nimmt an, dass nur kleine Nachrichten konstanter Größe übertragen werden. Das Netzwerk erlaubt in diesem Modell eine simultane Übertragung von höchstens $\lfloor L/g \rfloor$ Nachrichten. Dem Sender ist es möglich, nach der Zeit g eine neue Nachricht zu initiieren.

Mehrere Studien [9,10,11] haben gezeigt, dass dieses Modell für kleine Nachrichten sehr genau ist. Es gibt alle wichtigen Aspekte im Kommunikationsverhalten von parallelen Systemen wieder.

Dieses Modell hat mehrere Vorteile gegenüber anderen Modellen, es wurde entwickelt für Prozessoren mit jeweils eigenem Speicher und der Tatsache, dass die Geschwindigkeit des Netzwerkes viel langsamer als die Geschwindigkeit der CPU ist.

Aufgrund seiner Genauigkeit und Einfachheit bildet es die Grundlage für weitere Entwicklungen, auf welche hier genauer eingegangen wird.

Für das LogP Modell gibt es noch eine Erweiterung für große Nachrichten, da diese mit LogP nicht modelliert werden können, das erweiterte Modell nennt sich LogGP.

1.4.6.1 LogGP

Das LogGP Modell [18] wurde 1995 von Alexandrov et al. vorgeschlagen. Es erweitert das LogP Modell um den Parameter G, Gap pro Byte. Dadurch können auch große Nachrichten modelliert werden. Die Zeit für das Senden einer Nachricht der Größe m zwischen zwei Knoten beträgt $L + 2o + (m-1)G$. Dem Sender ist es wie im LogP Modell möglich, nach der Zeit g eine neue Nachricht zu initiieren.

Aufgrund der Vorteile des LogP Modells und der Möglichkeit auch große Nachrichten korrekt zu modellieren, ist dieses Modell sehr gut für die Bewertung kollektiver Algorithmen geeignet.

1.4.6.2 LoGPC

Das LoGPC Modell [19] ist eine Erweiterung des LogP Modells, um Stauungen im Netzwerk (Contention) zu modellieren. Es wurde für k-äre n-Cube Netzwerke [20] entwickelt. Der Autor verspricht, dass dieses Modell auch leicht auf andere Netzwerktopologien durch Änderung eines einzelnen Parameters möglich ist. Dies scheint aber für eine zentrale Switch - basierte Architektur wie InfiniBand™ aber nicht sehr genau zu sein, wodurch diese sehr interessante Erweiterung für InfiniBand™ leider nicht praktikabel ist.

1.4.6.3 LogGPS

Das LogGPS Modell [21] schließt eine zusätzliche Verzögerung ein, die in mehreren MPI [22] Bibliotheken durch das Rendezvous Protokoll hervorgerufen wird, welches für das Senden langer Nachrichten genutzt wird. Der Synchronisationsoverhead bedingt durch das Rendezvous Protokoll wird durch den zusätzlichen Parameter S beschrieben.

1.4.6.4 PLogP Modell

Das PLogP Modell [6] ist ebenfalls eine Erweiterung des LogP Modells. Es wird mit den Parametern End-To-End Latenz L, dem Sender- und Empfänger-Overhead $o_s(m)$ und $o_r(m)$, dem Gap pro Nachricht g(m) und der Anzahl der in die Kommunikation involvierten Knoten P. Die Zeit für das Senden einer Nachricht der Größe m zwischen zwei Knoten beträgt $L + g(m)$. Im Unterschied zum LogP/LogGP Modell sind Sender-, Empfänger-Overhead und Gap pro Nachricht von der Nachrichtengröße abhängig. Hierbei unterscheiden sich die Begriffe Latenz und Gap etwas von denen im LogP/LogGP Modell. Latenz im PLogP Modell schließt alle anfallenden Faktoren wie das Kopieren zum und vom Netzwerkinterface ein, die zu der Zeit der Nachrichtenübertragung addiert werden. Der Gap Parameter ist das kleinstmögliche Zeitintervall zwischen aufeinander folgenden Nachrichten, wobei $g(m) \geq o_s(m)$ und $g(m) \geq o_r(m)$ ist. Ist $g(m)$ eine lineare Funktion von Nachrichtengröße m und L schließt den Sender Overhead aus, dann ist das Modell identisch mit LogGP.

1.4.7 LogfP Modell

In diesem Abschnitt soll ein Modell namens LogfP [12] genauer evaluiert werden, da es die Besonderheiten des Netzwerkes InfiniBand modellieren kann, welches dieser Arbeit zugrunde liegt.

Hierbei wird gezeigt, dass das LogP Modell für kleine Nachrichten mit InfiniBand keine korrekte Vorhersage treffen kann. Dafür wird der InfiniBand Benchmark IBA-Bench [13] mit der Verbindungsart Reliable Connection (RC) genutzt, um 1 : P – P : 1 Round Trip Times (RTT) für verschiedene Prozessoranzahlen und Nachrichtengrößen zu bestimmen.

In Bild ist das Ping Pong Schema der Kommunikation dargestellt. Da beim Benchmark die Verbindungsart Reliable Connection verwendet wird, fällt sowohl beim Sender als auch Empfänger Overhead beim Senden und Empfangen einer Nachricht an. Dies ist in Bild 10 dargestellt.

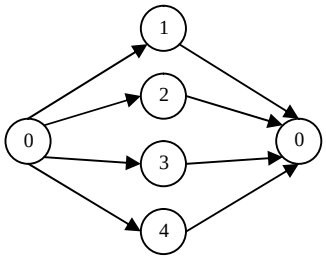


Bild 9. Ping Pong Benchmark Schema für 1 : 4 – 4: 1

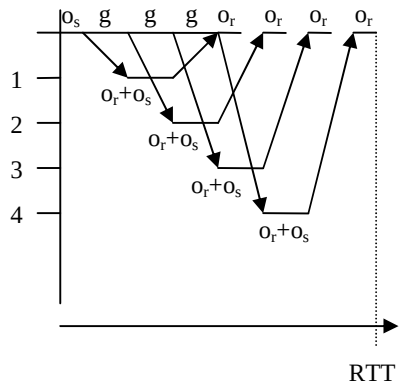


Bild 10. LogP Kommunikationsstruktur

Die Voraussagen des LogP Modells sind aber für das Senden und Empfangen kleiner Nachrichten im InfiniBand™ Netzwerk ungenau. Es scheint, dass bei einer 1:N Kommunikation kleine Nachrichten fast kostenlos mitg eschickt werden können. Diese Eigenschaft wurde auch schon in [51] beobachtet. Begründet liegt diese Eigenschaft in den parallelen Ausführungseinheiten, somit können bei Verwendung verschiedener Queue Pairs Nachrichten fast gleichzeitig versendet werden.

Die Messergebnisse und Voraussagen des LogfP Modells sind in den folgenden Bildern 11,12,13 zu sehen:

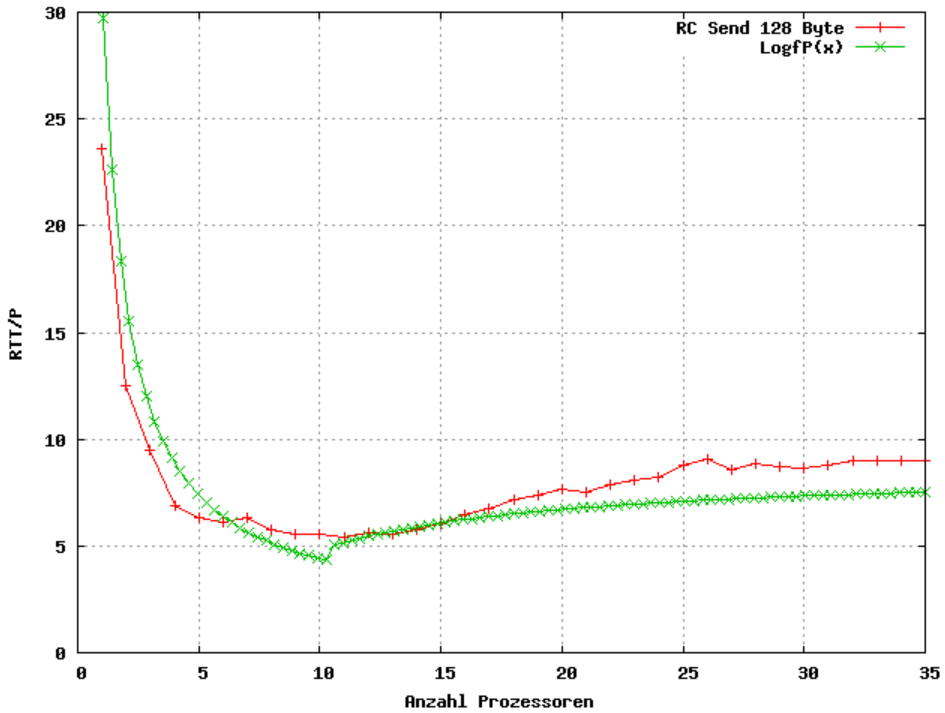


Bild 11. Messwerte und LogfP Modell für RC Send mit 128 Byte im 1:P P:1 Benchmark

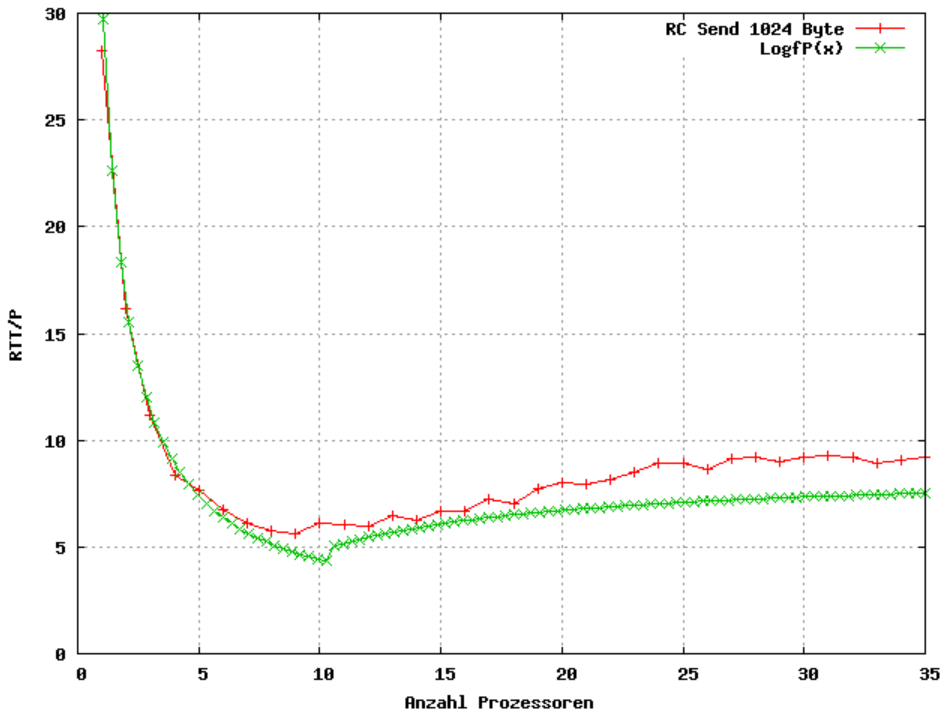


Bild 12. Messwerte und LogfP Modell für RC Send mit 1024 Byte im 1:P P:1 Benchmark

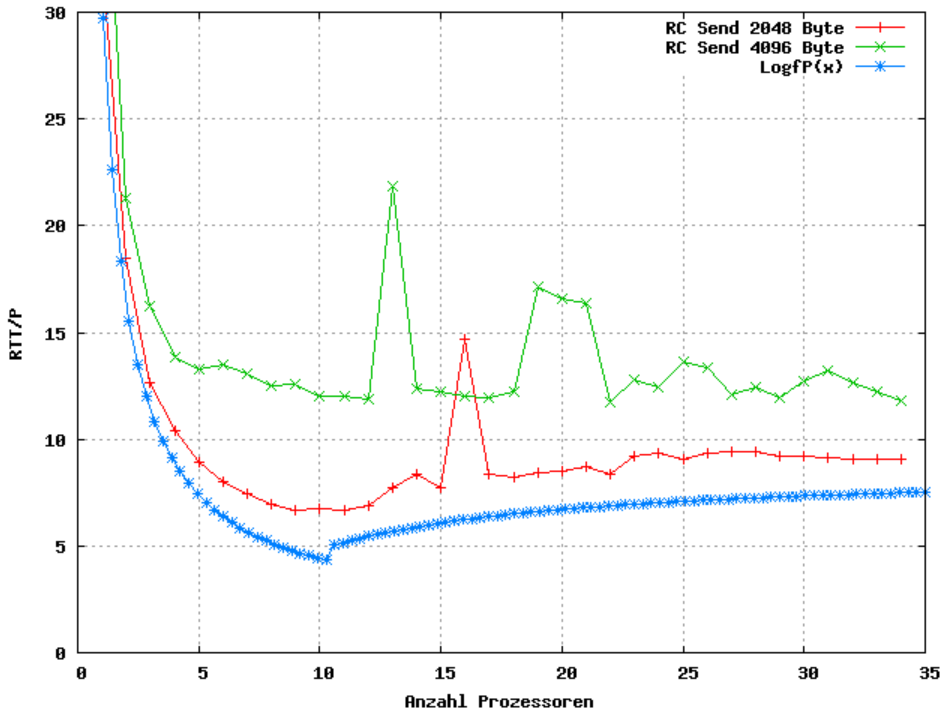


Bild 13. Messwerte und LogfP Modell für RC Send mit 2048/4096 Byte im 1:P P:1 Benchmark

An den Bildern 11,12,13 ist zu erkennen, dass das LogfP Modell nur für kleine Nachrichten in InfiniBand™ verwendbar ist, schon ab einer Nachrichtengröße von 2048 Byte werden die Abweichungen sehr groß. Die Messwerte bestätigen These 2 und 3.

Das LogfP Modell beruht auf folgenden Formeln:

$$\forall(P \le f)RTT(P) = 2*L + P*o_s(P) + o_r(1) + o_s(1) + o_r(P)$$

$$\forall(P > f)RTT(P) = 2*L + o_s(P) + o_r(P) + o_s(1) + o_r(1) + \max\{(P-1)*o_s(P) + (P-1)*o_r(P), (P-f)*g\}$$

Dabei sind die Funktionen für den Send- und Receive-Overhead folgender Form:

$$o_s(P) = o_{s\min} + \frac{o_s(1)}{P}$$

$$o_r(P) = o_{r\min} + \frac{o_r(1)}{P}$$

Aus den Messwerten und der Angleichung der Formel des Modells an die Messwerte ergeben sich folgende Werte für die Parameter:

$L = 6.5$
$o_s(1) = 3$
$o_r(1) = 1.8$
$o_{s\min} = 0.4$
$o_{r\min} = 0.3$
$g = 7$
$f = 9$

1.4.8 Berechnungsmodell

Da bei den in dieser Arbeit behandelten kollektiven Operationen Berechnung auftritt, muss diese ebenfalls modelliert werden.

Dies geschieht durch Einführung folgender Formel :

m	Länge der Nachricht
γ	Bandbreite der CPU

Berechnungszeit ergibt sich aus $m * \gamma$

2. Kollektive Operationen

2.1 Einleitung

Kollektive Kommunikation ist eine sehr wichtige und häufig gen utzte Komponente in MPI. Kollektive Kommunikationsoperationen sind Kommunikationen, an denen alle Prozesse aus der Gruppe des verwendeten Kommunikators beteiligt sind. Eine fünfjährige Studie an der Universität von Stuttgart hat gezeigt, dass über 40% der Ausführungszeit von Routinen des Message Passing Interface (MPI) auf die kollektiven Kommunikationsroutinen MPI_Allreduce und MPI_Reduce entfallen [23]. Eine Optimierung dieser und generell aller kollektiven Operationen wäre im Bezug auf die Performance v on parallelen Programmen, die auf MPI aufbauen, von sehr großem Nutzen.

In dieser Arbeit sollen die kollektiven Operationen MPI_Reduce, MPI_Allreduce, MPI_Reduce_Scatter und MPI_Scan speziell auf das InfiniBand Netzwerk hin optimiert werden. Dazu werden verschiedene Algorithmen für diese Operationen betrachtet und falls möglich speziell auf InfiniBand hin optimiert. Es werden verschiedene Algorithmen in Bezug auf Nachrichten- und Kommunikatorgröße betrachtet, um dann situationsbedingt, den besten Algorithmus für eine Aufgabe auszuwählen.

Frühere Arbeiten mit kollektiven Operationen versuchten zum Beispiel MPI_Reduce als inversen Broadcast zu implementieren und versuchten nicht die Protokolle basierend auf verschiedene Puffergrößen hin zu optimieren [25]. A ndere Arbeiten versuchten MPI_Allreduce als Kombination von anderen Operationen zu implementieren, zum Beispiel [26] als Kombination aus verteiltem Kombinieren (Reduce_Scatter) und Sammeln (Allgather). Ebenso wurden kollektive Algorithmen für Wide -Area Cluster in [27,28,29] entwickelt, weiteres Protokoll Tuning findet man in [30,31,32,33], ebenso automatisches Tuning in [35,36].

Einen sehr guten Einstieg in das Thema der kollektiven Kommunikation bietet [4]. Bei der kollektiven Kommunikation mit Berechnung liegt das Hauptaugenmerk auf der Überlagerung (Overlap) von Kommunikation und Berechnung [58]. Es gibt verschiedene Möglichkeiten um dies zu erreichen. Ein sehr bekanntes Prinzip ist das Pipelining, welches in [56] beschrieben wird. Es ist besonders für lange Nachrichten geeignet. Man kann aber nicht nur Kommunikation mit Berechnung überlagern, sondern auch Kommunikation mit Kommunikation [57]. Eine Analyse über den Overlap in MPI findet sich in [59].

2.2 MPI_Reduce

2.2.1 Einleitung

Zuerst soll die Funktionsweise von MPI_Reduce erklärt werden, wozu der MPI -Standard [24] genutzt wird.

MPI REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

IN sendbuf - Adresse des Sendepuffers
OUT recvbuf - Adresse des Empfangspuffers (nur significant für Root Prozess)
IN count – Anzahl Elemente im Sendepuffer (Integer)
IN datatype - Datentyp der Elemente im Sendepuffer (Handle)
IN op - Reduktionsoperation (Erklärung siehe unten - Handle)
IN root – Rank des Rootprozesses (Integer)
IN comm - Kommunikator (Handle)

int MPI Reduce(void* sendbuf, void* recvbuf, int count,
MPI Datatype datatype, MPI Op op, int root, MPI Comm comm)

MPI_Reduce kombiniert die Elemente, die sich im Eingabepuffer jedes Prozesses der Gruppe befinden unter Nutzung einer Operation und gibt die kombinierten Werte in den Ausgabepuffer des Prozesses mit dem Rang Root (Wurzelprozess) zurück Der Eingabepuffer ist durch die Argumente sendbuf, count und datatype definiert, der Ausgabepuffer ist durch die Argumente recvbuf, count und datatype definiert. Beide Puffer haben die gleiche Anzahl von Elementen des gleichen Typs. Diese Routine wird durch alle Mitglieder einer Gruppe mit den gleichen Argumenten für count, datatype, op, root und comm gerufen. Jeder Prozess stellt ein Element oder eine Sequenz von Elementen zur Verfügung, wobei die Operation für jedes Element der Sequenz ausgeführt wird.

Beispiel: Die Operation ist MPI_MAX und der Sendepuffer enthält zwei Elemente, dann ergibt sich für den Emfpangspuffer:

Empfangspuffer(1) = Globales Maximum(Sendepuffer(1))
Empfangspuffer(2) = Globales Maximum(Sendepuffer(2))

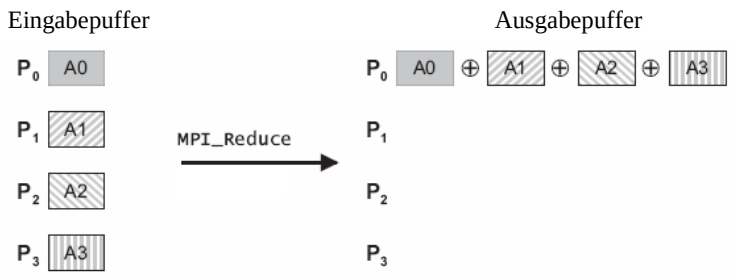


Bild 14. Verteilung der Daten bei MPI_Reduce

Es gibt eine Liste von vordefinierten Verknüpfungsoperationen, diese sind alle assoziativ und kommutativ:

MPI_MAX	- Maximum
MPI_MIN	- Minimum
MPI_SUM	- Summe
MPI_PROD	- Produkt
MPI_LAND	- Logisches UND
MPI_BAND	- Bit-weises UND
MPI_LOR	- Logisches ODER
MPI_BOR	- Bit-weises ODER
MPI_LXOR	- Logisches XOR
MPI_BXOR	- Bit-weises XOR
MPI_MAXLOC	- Maximaler Wert und Ort
MPI_MINLOC	- Minimaler Wert und Ort

Zusätzlich zu den vordefinierten Operationen können weitere Operationen vom Benutzer mittels MPI_OP_CREATE erzeugt werden. Weitere Informationen dazu finden sich im MPI - Standard [24].

MPI_Reduce kann mittels verschiedener Prinzipien implementiert werden. Diese sollen jetzt vorgestellt werden.

2.2.2 Implementierung mit Binomial -/k-nomialen Baum

Bei der Implementierung mit einem Binomialbaum werden die Knoten einer Kommunikatorgruppe in einem Binomialbaum angeordnet, wobei der Rootknoten der Operation auch der Wurzel dieses Baumes entspricht.

Ein Binomialbaum ist in der Graphentheorie ein spezieller Baum, dessen Struktur durch seine Ordnung eindeutig festgelegt ist.

Binomialbäume und ihre Ordnung sind wie folgt rekursiv definiert:

- Ein Binomialbaum der Ordnung 0 besteht aus einem einzelnen Knoten.
- Ein Binomialbaum der Ordnung l besitzt eine Wurzel mit Grad l deren Kinder genau die Ordnung l-1, l-2, ..., 0 (in dieser Reihenfolge) besitzen.

Ein Binomialbaum der Ordnung l lässt sich auch leicht aus zwei Binomialbäumen der Ordnung l-1 erstellen, indem einer der beiden Bäume zum am weitesten links stehenden Kind des anderen gemacht wird. Aus dieser Konstruktion lässt sich leicht per vollständiger Induktion die Eigenschaft ableiten, dass ein Binomialbaum der Ordnung l genau 2^l Knoten und die Höhe l besitzt.

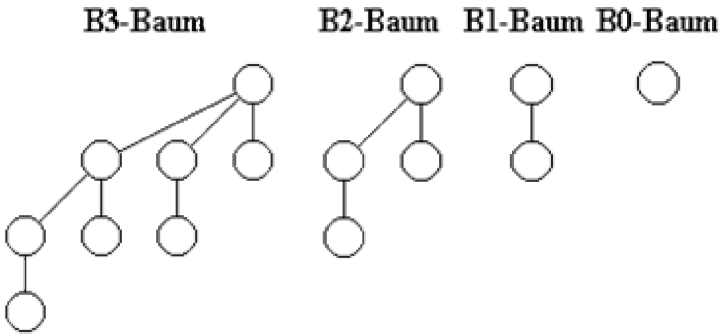


Bild 15. Darstellung von Binomialbäumen verschiedener Ordnung

Aufgrund der vorangegangenen Erkenntnisse über InfiniBand ™ soll es aber möglich sein, dass die parallelen Ausführungseinheiten im InfiniBand ™ beim Senden und Empfangen ausgenutzt werden können.

Dafür werden k-nomiale Bäume verwendet. K-nomiale Bäume sind eine Verallgemeinerung, ein Binomialbaum entspricht einem 2-nomialen Baum.

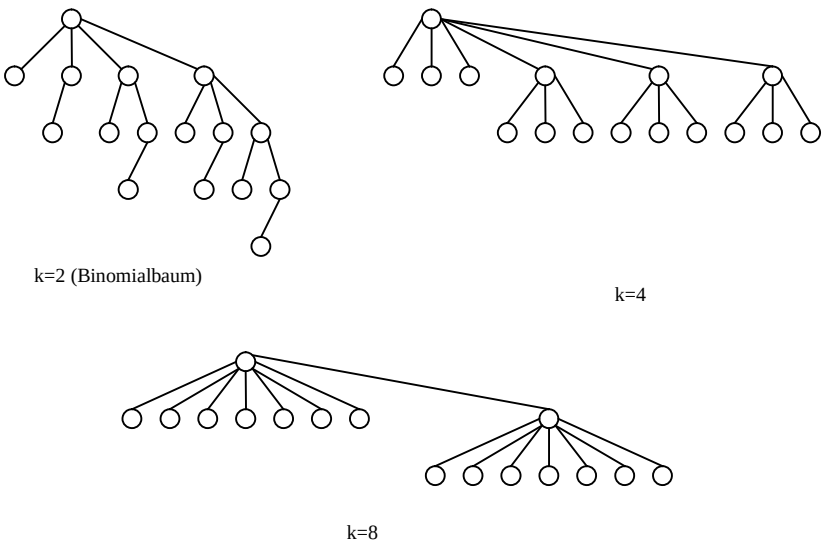


Bild 16. Beispiele für k-nomiale Bäume mit 16 Knoten

Die Kommunikationsstrukturen in effizienten Reduktionsalgorithmen müssen eine Balance zwischen der Zeit für die Berechnung und der Nachrichtenübertragung besitzen. Für einfache Operationen und kleine Datengrößen überwiegt die Zeit für die Kommunikation, deshalb ist es wichtig, diese zu beschleunigen. Dafür nutzen wir in diesem Fall k-nomiale Bäume, die eine Verallgemeinerung von Binomialbäumen sind, die eine bekannte Kommunikationsstruktur für die Reduktion darstellen.

Binomialbäume werden oft in Reduktionsalgorithmen genutzt, da sie reguläre Strukturen besitzen und deshalb einfach zu implementieren sind. Binomialbäume stellen eine optimale Kommunikationsstruktur in synchronen Netzwerken [4] dar.

Der Vorteil von k-nomialen Bäumen gegenüber Binomialbäumen besteht darin, dass diese eine größere Auswahlmöglichkeit von Kommunikationsstrukturen bieten, um eine bessere Ausgeglichenheit zwischen Übertragungs- und Berechnungskosten zu ermöglichen [39].

Zur Veranschaulichung wird hier ein Vergleich von Binomialbäumen und k-nomialen Bäumen anhand eines Broadcasts (Alle Knoten einer Gruppe erhalten die Daten des Wurzelknotens) erklärt. Ausgehened vom Wurzelknoten wird eine Broadcast Nachricht in einer Folge von mehreren Kommunikationsphasen verteilt. In jeder Phase sendet jeder Knoten, der eine Kopie der Nachricht besitzt zu einem anderen Knoten, der diese noch nicht besitzt. Damit verdoppelt sich die Anzahl der Knoten, die eine Kopie der Nachricht besitzen am Ende jeder Kommunikationsphase.

In einem k-nomialen Baum wird der Algorithmus so verändert, dass jeder Knoten mit einer Kopie der Nachricht, diese an (k-1) andere Knoten sendet, die diese Kopie noch nicht besitzen. Somit besitzen am Ende der ersten Phase der Wurzelknoten und seine (f-1) Kinder die Kopie der Nachricht. Somit wächst die Anzahl der Knoten, die die Kopie der Nachricht besitzen um ein Vielfaches von f.

Für die Benutzung dieser Kommunikationsstruktur wird vor der Ausführung der MPI_Reduce Funktion ein solcher k-nomialer Baum erstellt. Anschließend kann die Funktion auf diesem Baum ausgeführt werden.

Zur Erläuterung des Ablaufs in der Funktion verwenden wir ein Beispiel mit 16 Knoten und einen 4-nomialen Baum.

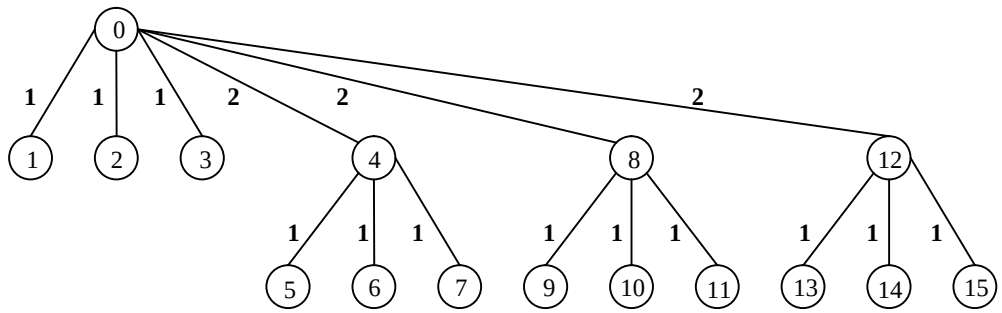


Bild 17. 4-nomiale Reduktion über 16 Knoten

Das Beispiel führt eine Reduktion mit 16 Knoten auf einem 4-nomialen Baum (Bild 17) durch. Ziel der Operation ist es das Ergebnis im Wurzelknoten 0 zu erhalten. Die Kanten des Graphen verbinden die Kommunikationspartner miteinander und sind mit Rundennummer gekennzeichnet, in der die Kommunikation stattfindet. Alle Nachrichten werden auf wärts von den Kindern zu den Eltern versandt.

In der ersten Phase empfängt und reduziert der Vaterknoten 0 genau (4-1) = 3 Nachrichten von den Knoten 1, 2 und 3; währenddessen empfangen und reduzieren simultan die Knoten 4, 8 und 12 von ihren drei Kindern. Am Ende der ersten Phase wurden die verteilten Daten teilweise reduziert und befinden sich in den vier Elternknoten 0, 4, 8 und 12. Der Algorithmus ist nach der zweiten Phase beendet, wenn der Knoten 0 von seinen drei Kindern, Knoten 4, 8 und 12 empfangen und die Teilergebnisse reduziert hat.

Der Parameter k ermöglicht also ein Abwägen zwischen Kommunikations- und Berechnungskosten. Aufgrund der beobachteten Effekte vom InfiniBand ™ beim Versenden und Empfangen kleiner Nachrichten in einer 1:N oder N:1 Beziehung können so die Kommunikationskosten der Operation gesenkt werden, dafür steigen natürlich die Berechnungskosten an, da die Berechnung nicht so gut parallelisiert ist.

2.2.3 Implementierung mit Verfahren von Rabenseifner

Beim Verfahren von Rabenseifner [23,38] wird MPI_Reduce durch eine Kombination aus Reduce_Scatter, implementiert mit rekursiver Vektorhalbierung (recursive vector halving) und Distanzverdopplung (recursive distance doubling) gefolgt von einem Gather, implementiert mit einem binären Baum, ausgeführt.

Dieses Verfahren funktioniert nur für eine Knotenanzahl, die einem Vielfachen von 2 entspricht. Ist dies nicht der Fall, wird in einem ersten Schritt die Anzahl der Prozesse auf den nächst niedrigeren Wert eines Vielfachen von 2 reduziert (p entspricht der Anzahl Prozesse in einer Gruppe): $p' = 2^{\lfloor \lg p \rfloor}$. Dazu müssen im ersten Schritt $r = p - p'$ Prozesse beseitigt werden.

Dazu senden die ersten $2r$ Prozesse paarweise von jedem geraden Rang zu einem ungeraden Rang (Rang+1) die zweite Hälfte ihres Eingabevektors und von jedem ungeraden Rang zu einem geraden Rang (Rang-1) die erste Hälfte des Eingabevektors. Alle $2r$ Prozesse berechnen die Reduktion auf ihrer Hälfte des Vektors. Anschließend werden die Prozesse neu durchnummeriert ($rank'$).

Jetzt ist die Voraussetzung (Prozessanzahl ist ein Vielfaches von 2) für dieses Verfahren gegeben und es kann mit dem ersten Schritt der rekursiven Vektorhalbierung und Distanzverdopplung begonnen werden. Dazu senden alle geraden Prozesse die zweite Hälfte ihres Puffers an $rank'+1$ und alle ungeraden Prozesse die erste Hälfte ihres Puffers an $rank'-1$. Anschließend wird die Reduktion zwischen dem lokalen Puffer und dem Empfangspuffer berechnet.

In den nächsten $\lg p' - 1$ Schritten werden die Puffer rekursiv halbiert und die Distanz verdoppelt. Danach besitzt jeder der p' Prozesse $\frac{1}{p'}$ des gesamten Ergebnisvektors der Reduktion. Damit ist der erste Teil des Verfahrens, das Reduce -Scatter beendet.

Jetzt wird das gegensätzliche Protokoll benötigt: Rekursive Vektorverdopplung und Distanzhalbierung. Jeder Prozess, dessen Bit mit dem Wert $p'/2$ in seinem neuen Rang identisch mit dem Bit im Wurzelprozess ist, muss ein Segment des Ergebnisvektors empfangen und die anderen Prozesse müssen ihr Segment schicken. Im nächsten Schritt arbeiten nur noch die Prozesse weiter, die im vorherigen Schritt e mpfangen haben, deren Bit wird um eins nach rechts geschoben (zum Beispiel $p'/4$), und so weiter bis die Gather Operation beendet ist.

Der Ablauf des Verfahrens lässt sich besser am Bild 18 erkennen. Es zeigt das Protokoll mit 13 Prozessen. Der Eingabevektor und alle Reduktionsergebnisse werden durch den Algorithmus in p' Teile (A, B, …, H) geteilt und werden deshalb mit A -H_{Rang} bezeichnet. Nach der ersten Reduktion hat der Prozess P0 A -D_{0-1} berechnet, was dem Reduktio nsergebnis der ersten Hälfte des Vektors (A -D) von den Prozessen 0 -1 entspricht. P1 hat E -H_{0-1}, P2 A -D_{2-3}, … berechnet. Der erste Schritt wird durch das Senden des Ergebnisses von jedem ungeraden Prozess (1 … 2r -1) zum Rang-1 in den zweiten Teil seines Puf fers beendet. Anschließend werden die ersten r geraden Prozesse und die $p - 2r$ letzten Prozesse neu von 0 bis $p' - 1$ durchnummeriert.

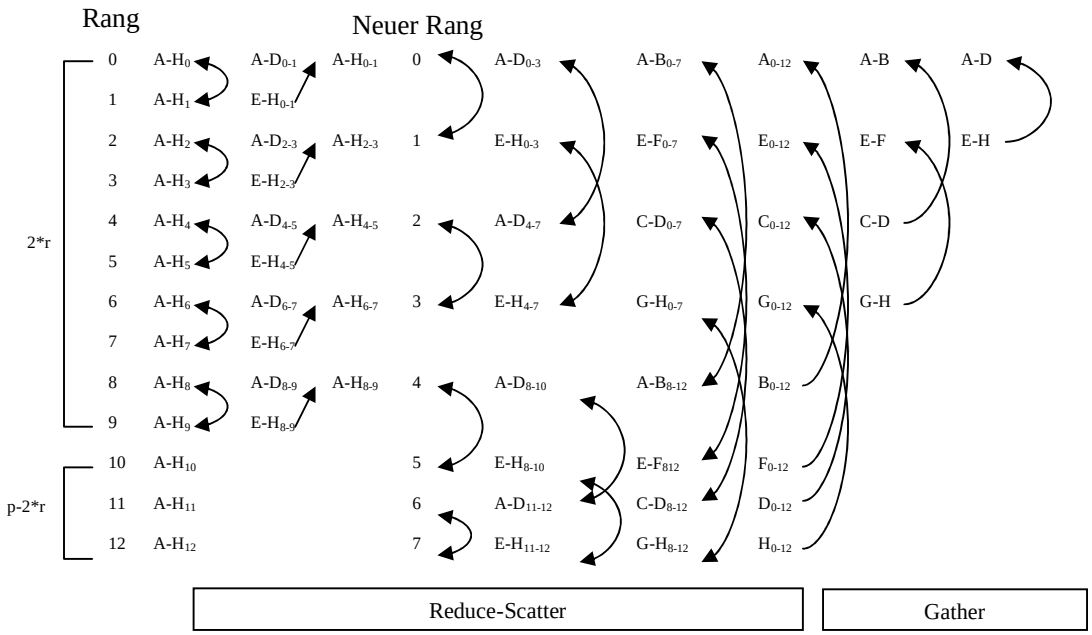


Bild 18. Rekursives Halbieren und Ver doppeln

Das Bild 18 zeigt die Zwischenergebnisse nach jedem Pufferaustausch (gefolgt von einer Reduktionsoperation im Reduce-Scatter Teil).

Beim Rabenseifner Verfahren findet eine optimale Parallelisierung der Berechnung statt, aber es ist hier keine Überlappung von Kommunikation und Berechnung möglich.

2.2.4 Implementierung mit Pipeline-Verfahren

Bei der Reduktion großer Vektoren spielt die Zeit für die Berechnung der Reduktion der einzelnen Elemente eine große Rolle. Eine Kommunikationsstruktur wie beim f-nomialen Baum bietet sich hier nicht an. Es muss versucht werden, die Reduktionsoperation so gleichmäßig wie möglich auf alle Prozesse zu verteilen, um so die Zeit für die Berechnung zu minimieren. Eine sehr gute Parallelisierung findet auch beim Rabenseifner Verfahren statt. Problematisch ist aber hier, dass keine Überlappung von Kommunikation und Berechnung stattfinden. Gerade bei großen Vektoren spielt aber die Überlappung (Overlap) eine entscheidende Rolle.

Eine fast optimale Parallelisierung von Berechnung und Kommunikation kann durch Pipelining erreicht werden. Dazu werden ausgehende vom Prozess mit dem höchsten Rang Segmente seines Puffers an den Prozess mit dem nächst niedrigerem Rang gesendet. Der nächst niedrigere Rang empfängt das erste Segment, führt die Verknüpfung mit seinem lokalen Segment durch und sendet es an den nächst niedrigeren Prozess

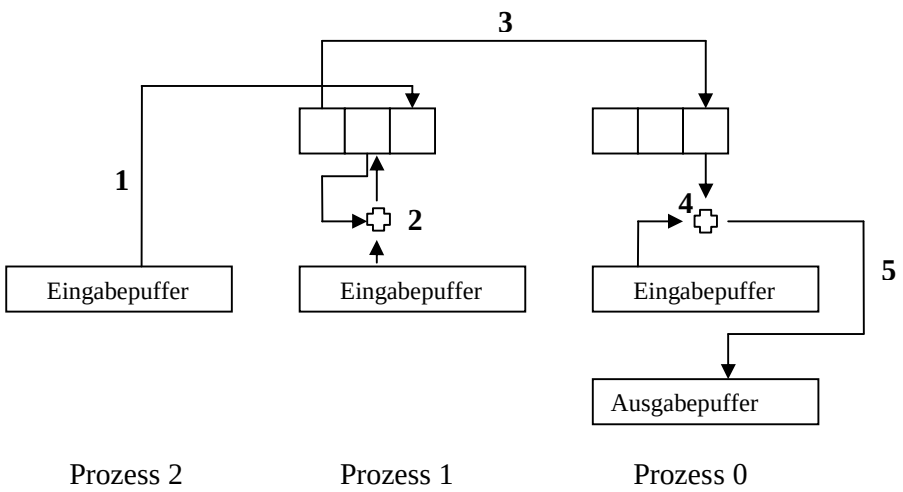


Bild 19. Ablauf des Pipeline-Prinzips am Beispiel von 3 Prozessen

Durch das Pipeline-Prinzip kommt es zur Überlappung von Kommunikation, bei gefüllter Pipeline findet eine Berechnung statt, während das nächste Segment gerade empfangen wird und das vorher berechnete Segment versendet wird.

2.3 MPI_Allreduce

2.3.1 Einleitung

Zuerst soll die Funktionsweise von MPI_Allreduce erklärt werden, wozu der MPI-Standard [24] genutzt wird.

MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)

IN sendbuf - Adresse des Sendepuffers
OUT recvbuf - Adresse des Empfangspuffers
IN count – Anzahl Elemente im Sendepuffer (Integer)
IN datatype - Datentyp der Elemente im Sendepuffer (Handle)
IN op - Reduktionsoperation (Erklärung siehe unten - Handle)
IN comm - Kommunikator (Handle)

int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm

MPI_Allreduce kombiniert die Elemente, die sich im Eingabepuffer jedes Prozesses der Gruppe befinden unter Nutzung einer Operation und gibt die kombinierten Werte in den Ausgabepuffer jedes Prozesses zurück Der Eingabepuffer ist durch die Argumente sendbuf , count und datatype definiert, der Ausgabepuffer ist durch die Argumente recvbuf, count und datatype definiert. Beide Puffer haben die gleiche Anzahl von Elementen des gleichen Typs. Diese Routine wird durch alle Mitglieder einer Gruppe mit den gleichen Argumenten für count, datatype, op und comm gerufen. Jeder Prozess stellt ein Element oder eine Sequenz von Elementen zur Verfügung, wobei die Operation für jedes Element der Sequenz ausgeführt wird.

Die Globalreduktion (MPI-Funktion MPI_Allreduce) arbeitet genau wie die Reduktion, jedoch liegt dann der Ergebnisvektor nach Abschluß der Reduktion in den Empfangspuffern aller Prozesse.

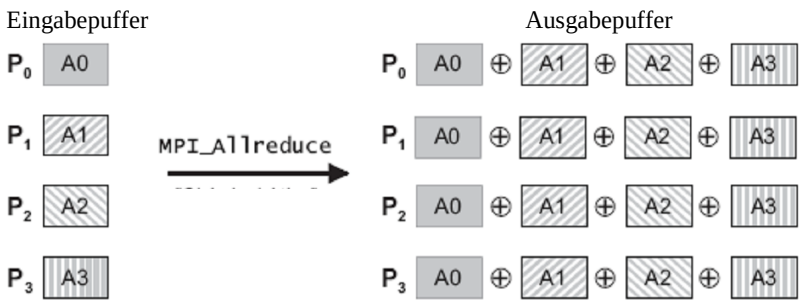


Bild 20. Verteilung der Daten bei MPI_Allreduce

Es gibt eine Liste von vordefinierten Verknüpfungsoperationen, diese sind alle assoziativ und kommutativ:

MPI_MAX	- Maximum
MPI_MIN	- Minimum
MPI_SUM	- Summe
MPI_PROD	- Produkt
MPI_LAND	- Logisches UND
MPI_BAND	- Bit-weises UND
MPI_LOR	- Logisches ODER
MPI_BOR	- Bit-weises ODER
MPI_LXOR	- Logisches XOR
MPI_BXOR	- Bit-weises XOR
MPI_MAXLOC	- Maximaler Wert und Ort
MPI_MINLOC	- Minimaler Wert und Ort

Zusätzlich zu den vordefinierten Operationen können weitere Operationen vom Benutzer mittels MPI_OP_CREATE erzeugt werden. Weitere Informationen dazu find en sich im MPI-Standard [24].

2.3.2 Lineare Implementierung

Bei der linearen Globalreduktion wird diese in zwei aufeinanderfolgende lineare Operationen zerlegt. Zuerst erfolgt ein lineare Reduktion zu einem bestimmten Prozess (Beispiel Prozess mit Rang 0), anschließend erfolgt ein lineares Broadcast vom Prozess mit Rang 0 zu allen anderen Prozessen in der Gruppe.

Wurzelknoten – Rank 0: Kopiere Eingabepuffer in Ausgabepuffer

Für alle Knoten der Gruppe außer Wurzelknoten
- Empfange alle Elemente aus Eingabepuffer vom Knoten in temporären Puffer
- Führe Verknüpfungsoperation für Elemente im temporären Puffer mit Elementen im Ausgabepuffer aus, schreibe Ergebnis in den Ausgabepuffer

Für alle Knoten der Gruppe außer Wurzelknoten
- Sende alle Elemente aus Ausgabepuffer zum Knoten

Alle Knoten der Gruppe außer Wurzelknoten: Sende Elemente des Eingabepuffers zum Wurzelknoten

Empfange alle Elemente vom Wurzelknoten in den Ausgabepuffer

2.3.3 Implementierung mit rekursiver Distanz-Verdopplung (Recursive Doubling)

Bei diesem Verfahren wird die Distanz zwischen den Kommunikationspartnern ausgehend von eins in jedem Schritt verdoppelt. In jedem Schritt werden zwischen den Kommunikationspartnern die Daten ausgetauscht und anschließend die Berechnung mit den schon vorhandenen Daten durchgeführt.

Im ersten Schritt tauschen also die Prozesse mit dem Abstand eins ihre Daten aus und verknüpfen die lokalen mit den empfangenen Daten. Im zweiten Schritt erfolgt die Kommunikation mit Prozessen vom Abstand zwei mit anschließender Verknüpfung, und so weiter.

Somit ist es möglich, eine Globalreduktion für eine Prozessoranzahl, die ein Vielfaches von zwei beträgt, in $\lg p$ Schritten durchzuführen.

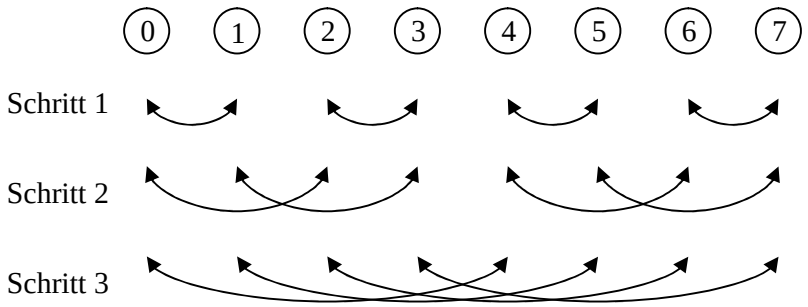


Bild 21. Kommunikation bei Implementierung von Allreduce mit Recursive Doubling

In Bild 21 ist die Kommunikation der Knoten an einem Beispiel mit 8 Knoten zu sehen. Im Bild fehlt aber die Darstellung der Berechnung, diese findet jeweils mit den lokalen und den empfangenen Daten statt.

2.3.4 Implementierung mit Rabenseifner Verfahren

Beim Verfahren von Rabenseifner [23,38] wird MPI_Allreduce durch eine Kombination aus Reduce_Scatter, implementiert mit rekursiver Vektorhalbierung (recursive vector halving) und Distanzverdopplung (recursive distance doubling) gefolgt von einem Allgather mit rekursiver Verdopplung ausgeführt.

Dieses Verfahren funktioniert nur für eine Knotenanzahl, die einem Vielfachen von 2 entspricht. Ist dies nicht der Fall, wird in einem ersten Schritt die Anzahl der Prozesse auf den nächst niedrigeren Wert eines Vielfachen von 2 reduziert (p entspricht der Anzahl Prozesse in einer Gruppe): $p' = 2^{\lfloor \lg p \rfloor}$. Dazu müssen im ersten Schritt $r = p - p'$ Prozesse beseitigt werden.

Dazu senden die ersten $2r$ Prozesse paarweise von jedem geraden Rang zu einem ungeraden Rang (Rang+1) die zweite Hälfte ihres Eingabevektors und von jedem ungeraden Rang zu einem geraden Rang (Rang-1) die erste Hälfte des Eingabevektors. Alle $2r$ Prozesse berechnen die Reduktion auf ihrer Hälfte des Vektors. Anschließend werden die Prozesse neu durchnummeriert ($rank'$).

Jetzt ist die Voraussetzung (Prozessanzahl ist ein Vielfaches von 2) für dieses Verfahren gegeben und es kann mit dem ersten Schritt der rekursiven Vektorhalbierung und Distanzverdopplung begonnen werden. Dazu senden alle geraden Prozesse die zweite Hälfte ihres Puffers an $rank'+1$ und alle ungeraden Prozesse die erste Hälfte ihres Puffers an $rank'-1$. Anschließend wird die Reduktion zwischen dem lokalen Puffer und dem Empfangspuffer berechnet.

In den nächsten $\lg p'-1$ Schritten werden die Puffer rekursiv halbiert und die Distanz verdoppelt. Danach besitzt jeder der p' Prozesse $\frac{1}{p'}$ des gesamten Ergebnisvektors der Reduktion. Damit ist der erste Teil des Verfahrens, das Reduce -Scatter beendet.

Jetzt wird das gegensätzliche Protokoll benötigt: Rekursive Vektorverdopplung (re cursive vector doubling) und Distanzhalbierung (recursive distance halving). Im ersten Schritt tauscht der Prozess $\frac{1}{p'}$ seines Puffers aus, um dann $\frac{2}{p'}$ des Ergebnisvektors zu bekommen. Im nächsten Schritt werden $\frac{2}{p'}$ ausgetauscht, um $\frac{4}{p'}$ zu erhalten und so weiter. A-B, A-D… in Bild 22 bezeichnen die zur Zeit gespeicherten Teile des Ergebnisvektors. Nach jedem Kommunikationsaustauschschritt, verdoppelt sich der Ergebnispuffer und nac h $\lg p'$ Schritten besitzen die p' Prozesse das gesamte Reduktionsergebnis.

Sollte die Anzahl Prozesse am Anfang nicht Vielfaches von zwei gewesen sein, muss der Ergebnisvektor noch an die r beseitigten Prozesse gesandt werden.

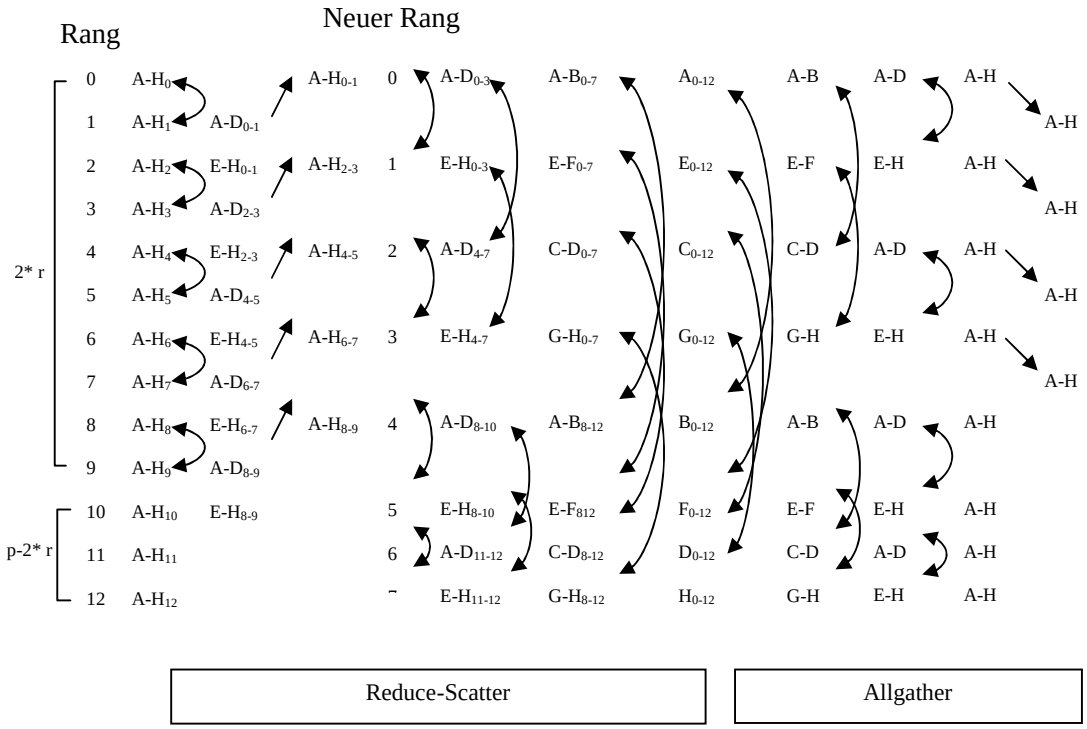


Bild 22. Rekursives Halbieren und Verdoppeln

Im Bild 22 sieht man den kompletten Ablauf dieses Verfahrens am Beispiel von 13 Prozessen. Es zeigt auch die Zwischenresultate nach jedem Pufferau stausch (gefolgt von einer Reduktionsoperation im Reduce -Scatter Teil).

Wie schon bei MPI_Reduce nach dem Rabenseifner Verfahren ist eine gute Parallelisierung der Berechnung gewährleistet. Sehr nachteilig wirkt sich aus, dass keine Überlagerung von Kommunikation und Berechnung möglich ist.

2.4 MPI_Reduce_scatter

2.4.1 Einleitung

Zuerst soll die Funktionsweise von MPI_Reduce_Scatter erklärt werden, wozu der MPI - Standard [24] genutzt wird.

MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcoun ts, datatype, op, comm)

IN sendbuf - Adresse des Sende puffers
OUT recvbuf - Adresse des Empfangs puffers
IN recvcounts – Feld von Integern, die die Anzahl von Elementen im Ergebnis enthält, die an jeden Prozess verteilt werden. Das Feld muss bei jedem ruf enden Prozess identisch sein.
IN datatype - Datentyp der Elemente im Sende puffer (Handle)
IN op - Reduktionsoperation (Erklärung siehe unten - Handle)
IN comm - Kommunikator (Handle)

int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts, M PI_Datatype datatype, MPI_Op op, MPI_Comm comm)

MPI_REDUCE_SCATTER führt zuerst eine elementweise Reduktion auf dem Vektor von $count = \sum_i recvcounts[i]$ Elementen im Sendepuffer durch, der durch sendbuf, count und datatype definiert ist. Anschließend wird der Ergebnisvektor in n disjunkte Segmente geteilt, wobei n die Anzahl der Mitglieder der Gruppe ist. Segment i enthält $recvcounts[i]$ Elemente. Das i-te Element wird an Prozess i gesendet und dort im Empfangsbuffer gespeichert, der durch recvbuf, recvcounts [i] und datatype charakterisiert wird.

Bei der Verteilungsreduktion (MPI_Reduce_scatter) hat im Gegensatz zu Allreduce nicht jeder Prozeß den gleichen und vollständigen Vektor, der sich aus der Verknüpfung der Vektoren aller beteiligten Prozesse ergibt, als Ergebnis, sondern nur einen bestimmten Tei l dieses Vektors, was in Bild 24 zu sehen ist.

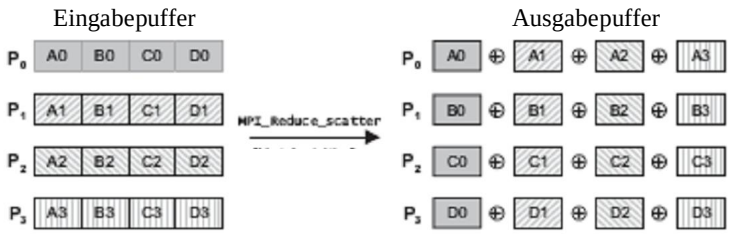


Bild 24. Verteilung der Daten bei der Verteilungsreduktion (MPI_Reduce_scatter)

Es gibt eine Liste von vordefinierten Verknüpfungsoperatio nen, diese sind alle assoziativ und kommutativ:

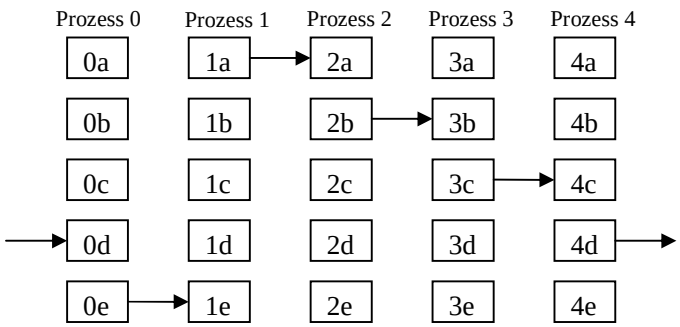
MPI_MAX	- Maximum
MPI_MIN	- Minimum
MPI_SUM	- Summe
MPI_PROD	- Produkt
MPI_LAND	- Logisches UND
MPI_BAND	- Bit-weises UND
MPI_LOR	- Logisches ODER
MPI_BOR	- Bit-weises ODER
MPI_LXOR	- Logisches XOR
MPI_BXOR	- Bit-weises XOR
MPI_MAXLOC	- Maximaler Wert und Ort
MPI_MINLOC	- Minimaler Wert und Ort

Zusätzlich zu den vordefinierten Operationen können weitere Operationen vom Benutzer mittels MPI_OP_CREATE erzeugt werden. Weitere Informationen dazu finden sich im MPI-Standard [24].

2.4.2 Implementierung mit Ring-Algorithmus

Beim Ring-Algorithmus werden die Blöcke, die in recvcounts[] definiert sind, solange im Ring geschickt und berechnet, bis sie an der benötigten Stelle angelangt sind.

Der Algorithmus wird in Bild 25 dargestellt.



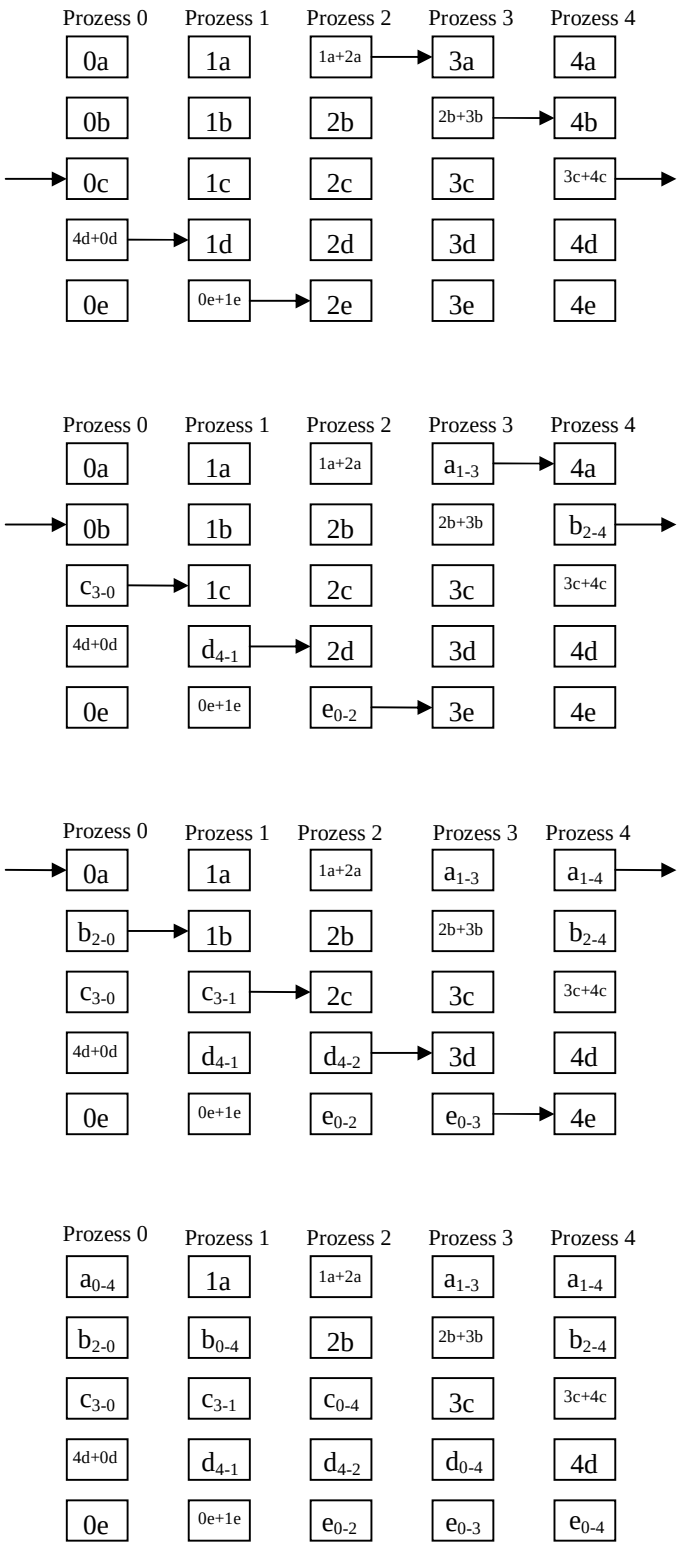


Bild 25. Ablauf des Ring-Algorithmus am Beispiel von 5 Prozessen

Es ist also ersichtlich, dass der Algorithmus linear mit der Anzahl Prozessen skaliert.

2.4.3 Implementierung mit rekursivem Halbieren (Recursive Halving)

Dieses Verfahren wird hauptsächlich bei kurzen und mittleren Vektoren eingesetzt. Dabei sendet die erste Hälfte der Prozesse die zweite Hälfte ihrer Daten an ihre Gegenstelle in der zweiten Prozesshälfte und empfängt von ihr die erste Hälfte ihrer Daten. Dies wird rekursiv wiederholt, bis jeder Prozess die von ihm benötigten Daten besitzt. Hier wird also rekursive Vektor- und Distanzhalbierung angewandt.

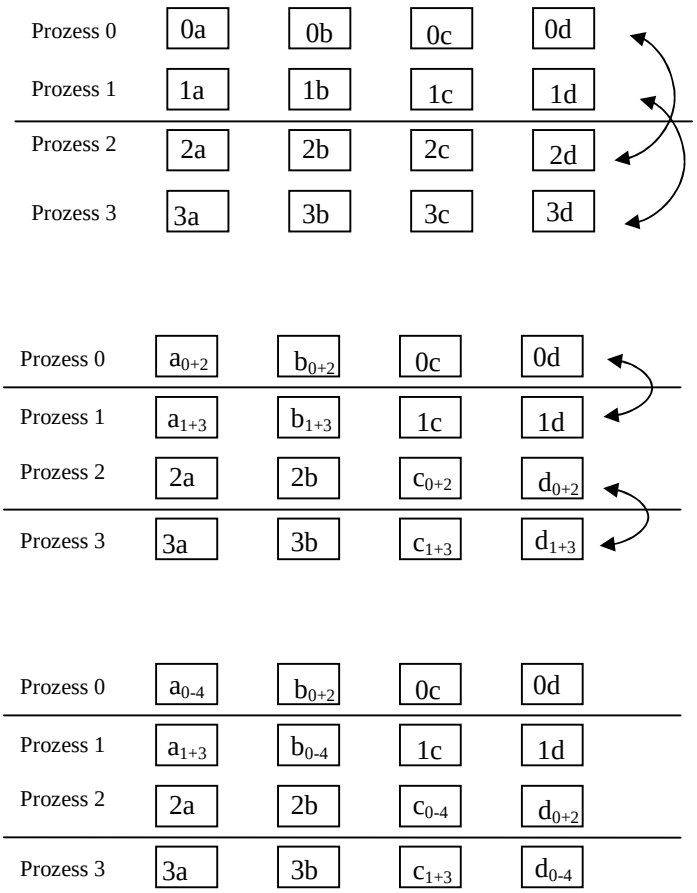


Bild 26. Darstellung des rekursiven Halbierens am Beispiel von 4 Prozessen

2.5 MPI_Scan

2.5.1 Einleitung

Zuerst soll die Funktionsweise von MPI_Scan erklärt werden, wozu der MPI -Standard [24] genutzt wird.

MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)
IN sendbuf - Adresse des Sendepuffers
OUT recvbuf - Adresse des Empfangspuffers
IN count – Anzahl Elemente im Sendepuffer (Integer)
IN datatype - Datentyp der Elemente im Sendepuffer (Handle)
IN op - Reduktionsoperation (Erklärung siehe unten - Handle)
IN comm - Kommunikator (Handle)

int MPI_Scan(void* sendbuf, void* recvbuf, int count, MPI_ Datatype datatype, MPI_Op op, MPI_Comm comm)

MPI_Scan ist eine Präfixreduktion, es ist eine Variante der normalen Reduktion, bei der immer der Prozess mit dem höchsten Rang in der Kommunikatorgruppe als Wurzelprozess festgelegt ist. Nach Abschluss der Präfixreduktion hat jeder Prozess im Unterschied zur Reduktion die Verknüpfung seines Vektors mit den Vektoren aller Prozesse mit niedrigerem Rang in Ausgabepuffer. Dies ist in Bild 27 zu erkennen.

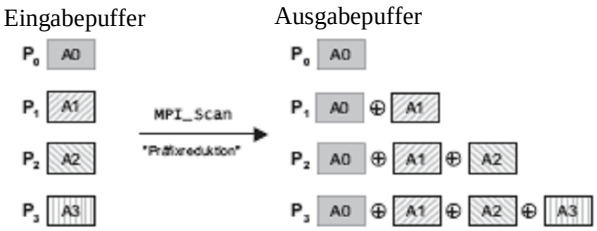


Bild 27. Verteilung der Daten bei der Präfixreduktion (MPI_Scan)

Es gibt eine Liste von vordefinierten Verknüpfungsoperationen, diese sind alle assoziativ und kommutativ:

MPI_MAX	- Maximum
MPI_MIN	- Minimum
MPI_SUM	- Summe
MPI_PROD	- Produkt
MPI_LAND	- Logisches UND
MPI_BAND	- Bit-weises UND
MPI_LOR	- Logisches ODER

MPI_BOR - Bit-weises ODER
MPI_LXOR - Logisches XOR
MPI_BXOR - Bit-weises XOR
MPI_MAXLOC - Maximaler Wert und Ort
MPI_MINLOC - Minimaler Wert und Ort

Zusätzlich zu den vordefinierten Operationen können weitere Operationen vom Be nutzer mittels MPI_OP_CREATE erzeugt werden. Weitere Informationen dazu finden sich im MPI - Standard [24].

2.5.2 Implementierung mit Pipeline-Verfahren

Ähnlich wie bei MPI_Reduce mit dem Pipeline Verfahren ist diese Möglichkeit auch für die Präfixreduktion möglich. Günstig ist dies wie bei MPI_Reduce für lange Vektoren, da der Algorithmus linear mit der Prozessoranzahl skaliert und nur für große Vektoren sich die Füllzeit der Pipeline rentiert.

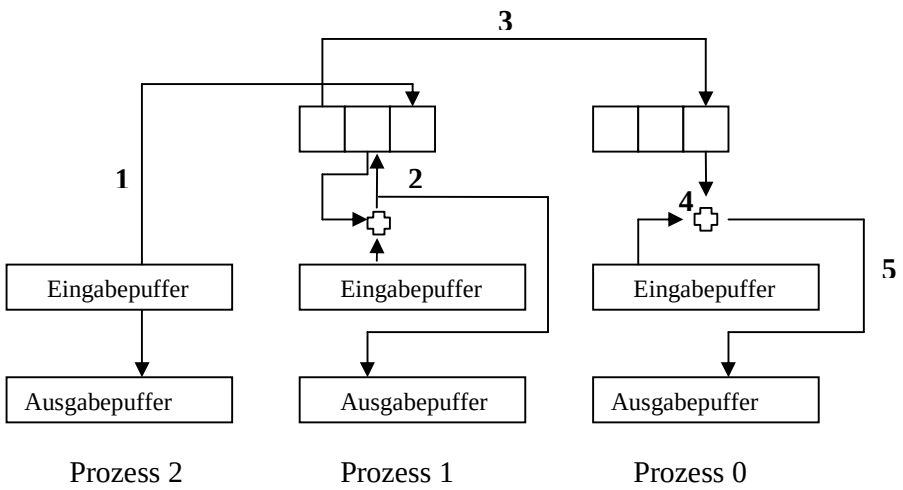


Bild 28. Ablauf des Pipeline-Prinzips bei MPI_Scan am Beispiel von 3 Prozessen

2.5.3 Implementierung mit rekursivem Verdoppeln (Recursive Doubling)

Für MPI_Scan gibt es auch einen Algorithmus, der logarithmisch mit der Anzahl der Prozesse skaliert.

Der Ablauf des Algorithmus im Pseudocode:

```
recvbuf = sendbuf;
partial_scan = sendbuf;
mask = 0x1;
while (mask<size) {
        dst = rank^mask;
        if (dst < size) {
                Sende partial_scan an dst;
                Empfange von dst in tmp_buf;
                if (rank > dst) {
                        partial_scan = tmp_buf verknüpft mit partial_scan;
                        recvbuf  = tmp_buf verknüpft mit recvbuf;
                }
                else {
                        tmp_buf = partial_scan + tmp_buf;
                        partial_scan = tmp_buf;
                }
        }
        mask <<= 1;
}
```

2.6 Verifikation der MPI Implementierung

Problematisch bei der Implementierung neuer Algorithmen in MPI ist die allgemeine Verifikation dieser Algorithmen. Bei der Implementierung werden meist nur bestimmte Situationen überprüft, es erfolgt aber keine vollständige Prüfung, ob die Implementierung korrekt ist.

Dafür sind so genannte Test Suiten erforderlich. Diese müssen systematisch alle möglichen Fälle (zum Beispiel überlappende Kommunikatoren, usw.) durchlaufen und die Implementierung auf Korrektheit prüfen. Eine Übersicht ist hier [60] zu finden. Diese Suiten lassen aber eine systematische Fehlersuche vermissen und sind mehr oder weniger eine Sammlung von Fehlerquellen.

Es wurde aber begonnen eine neue Test Suite zu entwickeln, die in [51] beschrieben wird, was sehr Erfolg versprechend erscheint.

3. Praktische Ergebnisse und Schlussfolgerungen

Die implementierten MPI Funktionen wurden mit dem nbcbench [51] auf dem CHIC, einem neuen InfiniBand Cluster der TU Chemnitz, getestet. Die Benchmarkergebnisse der Funktionen MPI_Reduce, MPI_Allreduce, MPI_Reduce_scatter und MPI_Scan werden hierbei mit den Ergebnissen der Implementierung aus der TUNED Komponente des COLL Frameworks von OpenMPI und MVAPICH in der Version 0.9 .7 verglichen.

3.1 MPI_Reduce

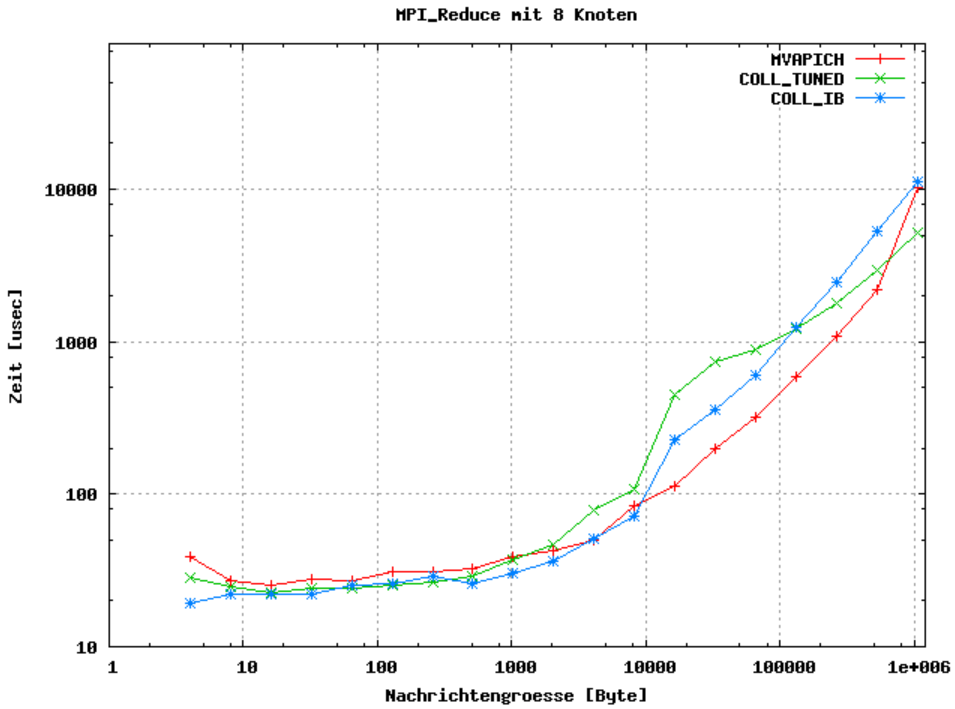


Bild 29. MPI_Reduce() mit 8 Knoten

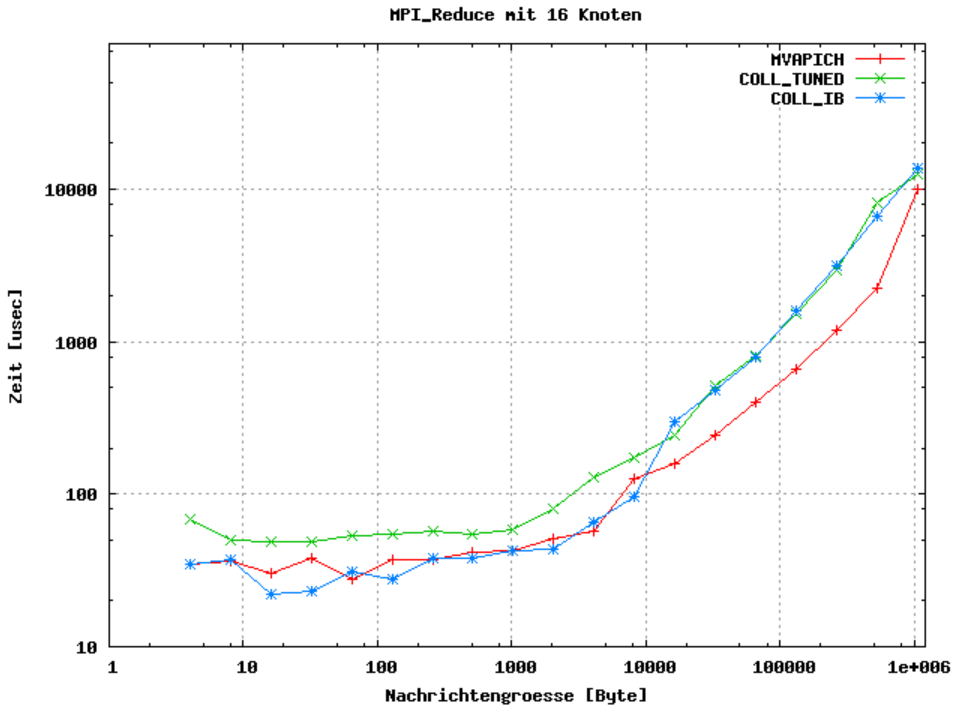


Bild 30. MPI_Reduce() mit 16 Knoten

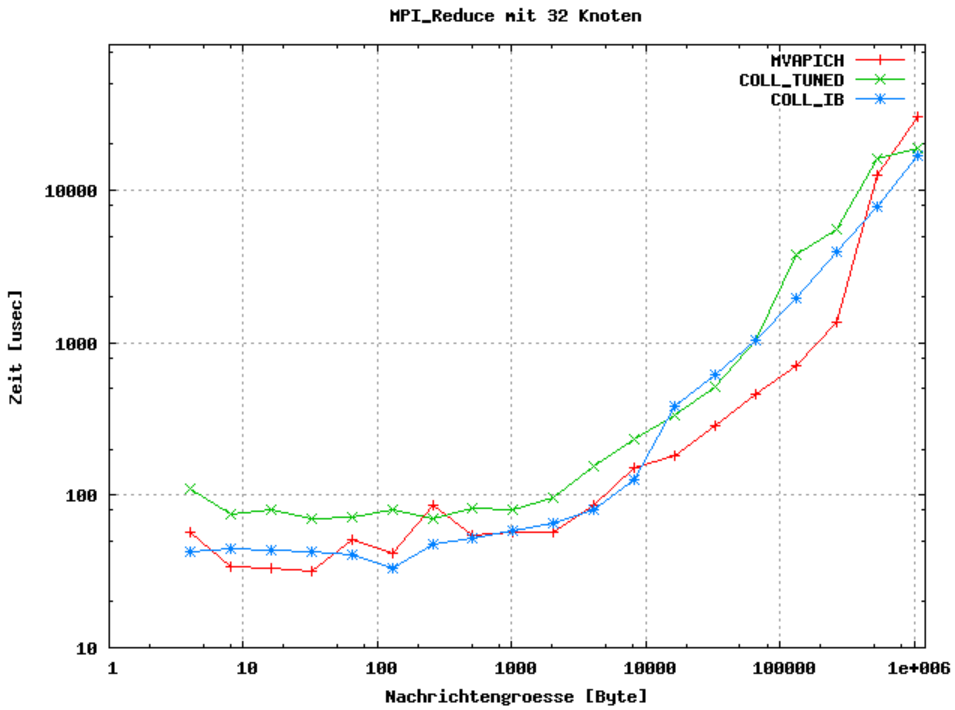


Bild 31. MPI_Reduce() mit 32 Knoten

An den Bildern 29,30,31 ist zu sehen, dass durch die IB Komponente die Funktion MPI_Reduce in OpenMPI eine deutliche Performanceverbesserung erfahren hat. Durch die Ausnutzung spezieller Eigenschaften des zugrunde liegenden InfiniBand ™ Netzwerkes (z.B. Möglichkeit des nahezu parallelen Sendens kleiner Nachrichten an verschiedene Knoten) und der Verbesserung der Überlappung von Kommunikation und Berechnung ist es gelungen, OpenMPI mit der Funktion MPI_Reduce auf InfiniBand ™ Netzwerken zu beschleunigen.

Bei manchen Nachrichten- und Kommunikatorgrößen ist es sogar schneller als MVAPICH, welches als gesamte MPI-Implementierung auf InfiniBand ™ hin optimiert ist.

3.2 MPI_Allreduce

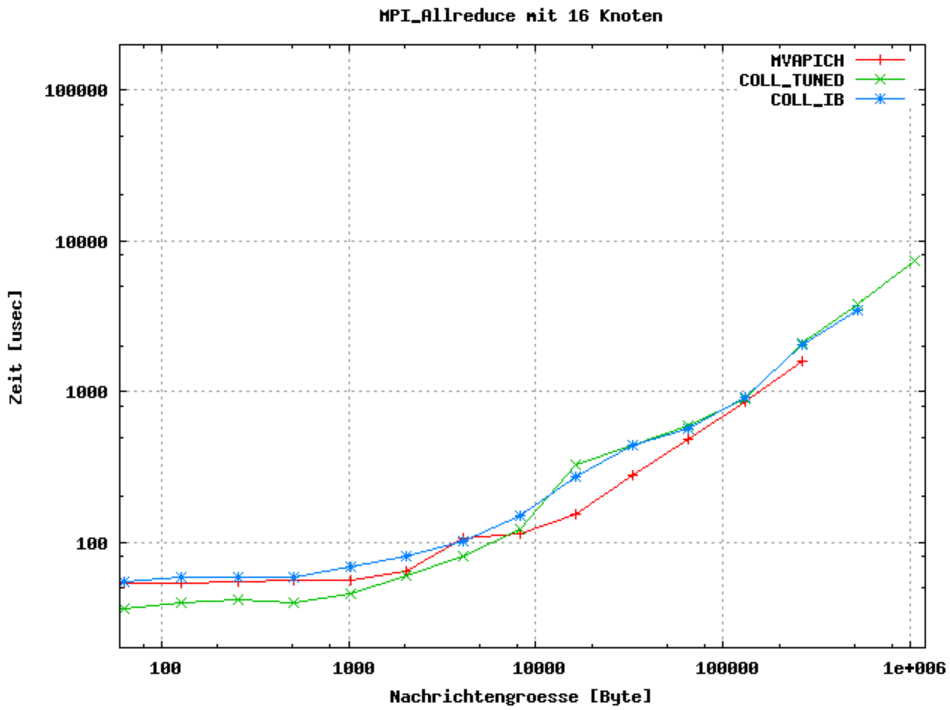


Bild 32. MPI_Allreduce() mit 16 Knoten

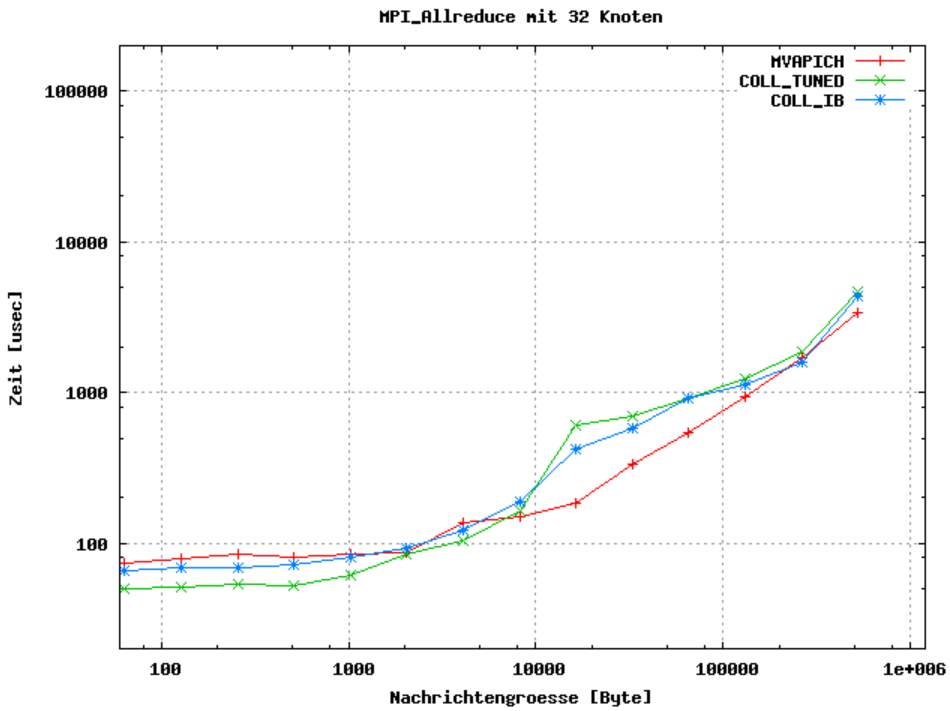


Bild 33. MPI_Allreduce() mit 32 Knoten

Bei MPI_Allreduce konnte mittels der jetzigen Version von IB keine so deutliche Verbesserung der Performance wie bei MPI_Reduce erzielt werden. Im Bereich der mittleren Nachrichtengröße offenbart sich eine schlechte Performance gegenüber MVAPICH.

3.3 MPI_Reduce_scatter

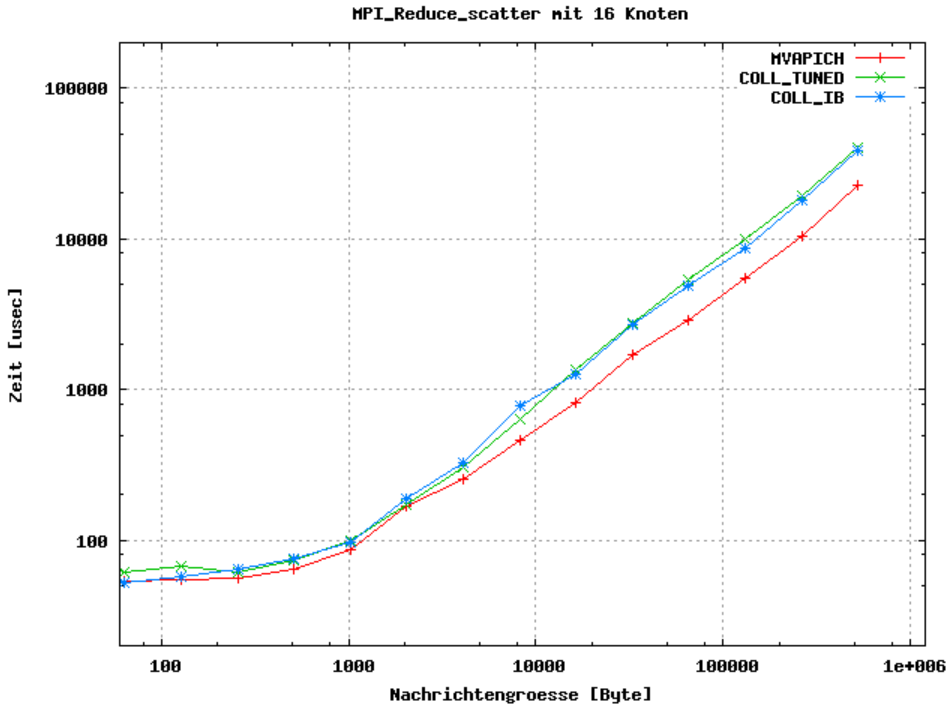


Bild 34. MPI_Reduce_scatter() mit 16 Knoten

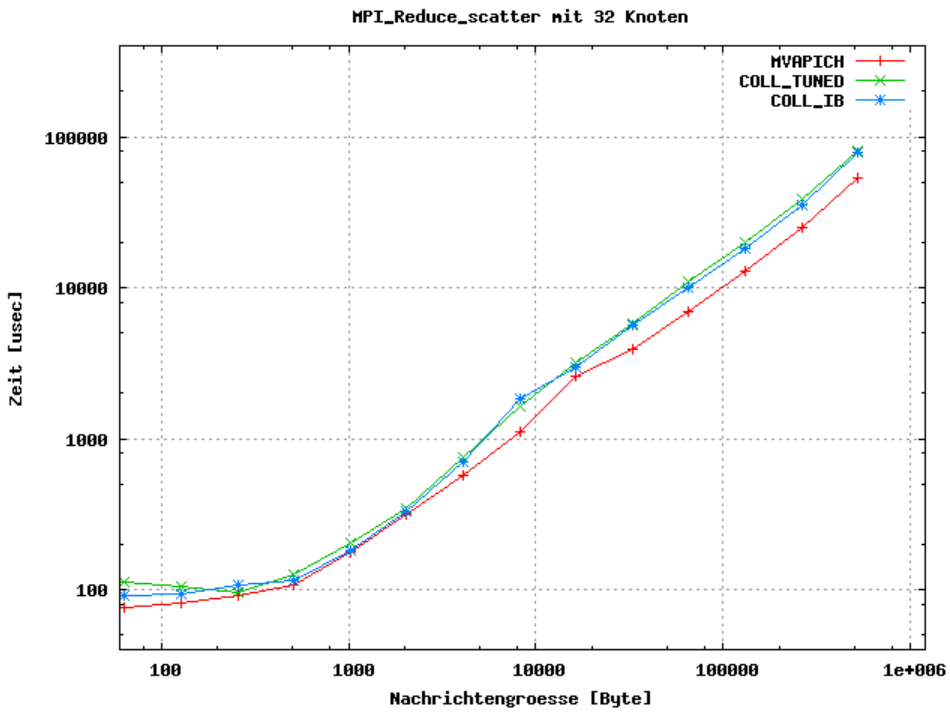


Bild 35. MPI_Reduce_scatter() mit 32 Knoten

Bei MPI_Reduce_Scatter ist der Performancegewinn gegenüber der TUNED Komponente von OpenMPI marginal, es ist aber eine kleine Verbesserung zu erkennen. Auch bei dieser Funktion zeigt MVAPICH eine bess ere Performance, da die Implementierung im Ganzen auf InfiniBand hin optimiert ist.

3.4 MPI_Scan

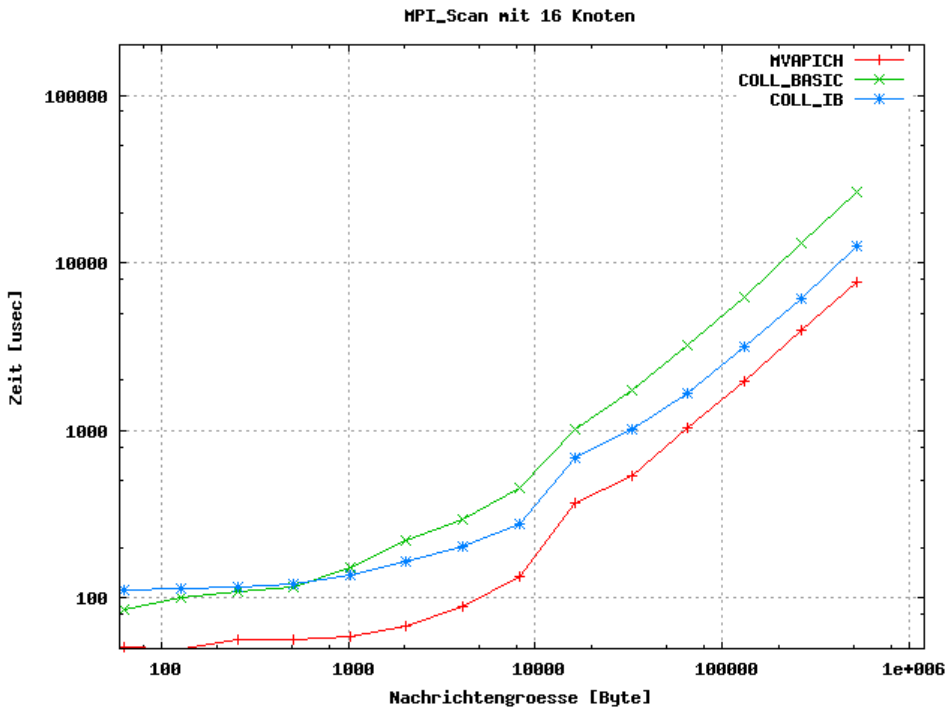


Bild 36. MPI_Scan() mit 16 Knoten

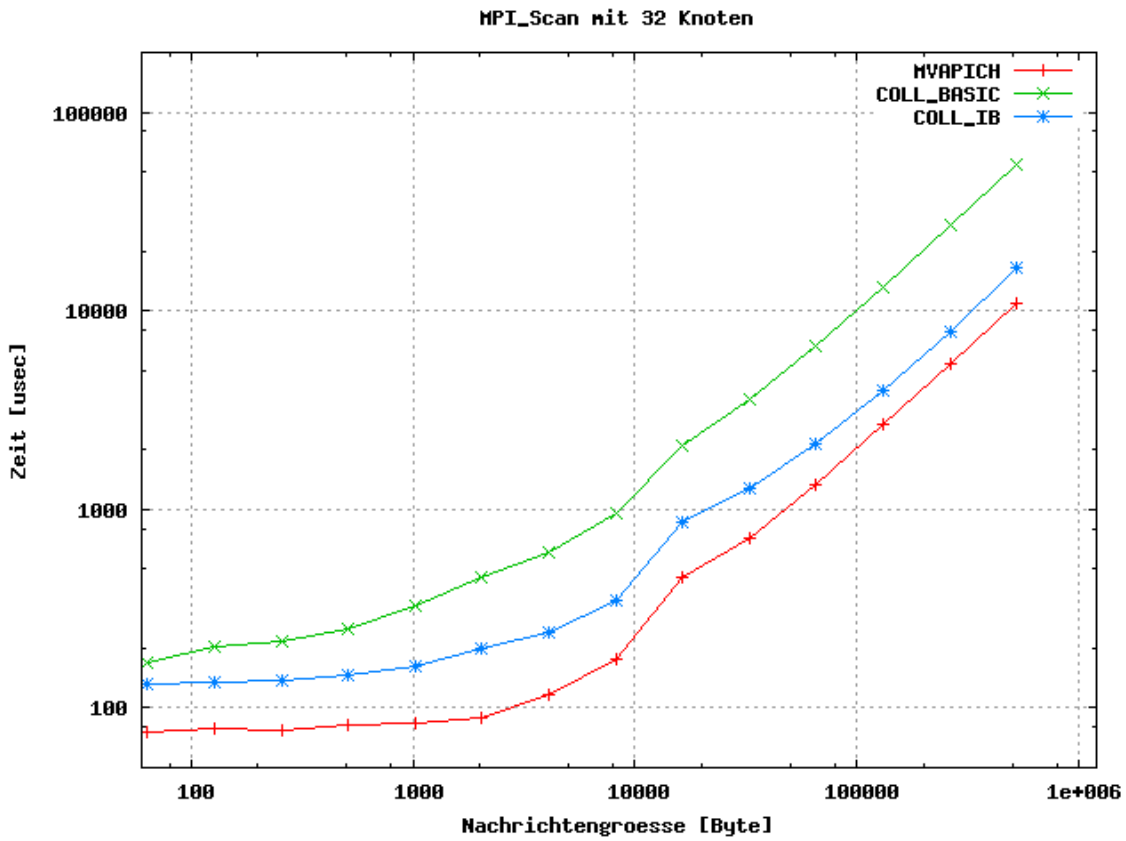


Bild 37. MPI_Scan() mit 32 Knoten

Bei MPI_Scan bringt die IB Komponente für OpenMPI einen sehr deutlichen Performancegewinn. Aber auch hier bietet MVAPICH eine bessere Performance.

3.5 Schlussfolgerung und zukünftige Aufgaben

Mit den Benchmarkergebnissen wurde gezeigt, dass die Optimierung kollektiver Operationen auf das darunter liegende Netzwerk teilweise einen deutlichen Performancegewinn ermöglicht, was These 5 bestätigt. Der Bereich der Optimierung kollektiver Algorithmen bietet noch viele Entwicklungsmöglichkeiten. In dieser Arbeit wurde Reliable Connection als Verbindungsart benutzt. Neue Arbeiten [53] im Bereich InfiniBand™ zeigen andere Möglichkeiten auf, welche weiter erörtert werden müssen. Ebenso ist die Nutzung von Hardware Multicast ein sehr interessanter Ansatz, auf dem in [54] eingegangen wird. Auf einen ganz anderen Ansatz für die Berechnung in kollektiven Operationen wird in [55] eingegangen. Eine viel versprechende Entwicklung zeigt sich auch in [61]. Aufgrund der starken Nutzung kollektiver Operationen ist jede mögliche Verbesserung hilfreich und muss erörtert werden.

Anhang

A1 Liste der Bilder

A2 Referenzen

[1] Sathis S. Vadhiyar, Graham E. Fagg, Jack J. Dongarra: *Towards an Accurate Model for Collective Communications*, International Journal of High Performance Computing Applications, Vol. 18, Issue 1, Feburar 2004, Seiten 159-167

[2] Torsten Hoefler : *Evaluation of publicly available Barrier–Algorithms and Improvement of the Barrier–Operation for large–scale Cluster–Systems with special Attention on InfiniBand™ Networks*, Diplomarbeit

[3] Hyacinthe Nzigou Mamadou, Takeshi Nanri, Kazuaki Murakami: *Collective Communication Costs Analysis over Gigabit Ethernet and InfiniBand* , LECTURE NOTES IN COMPUTER SCIENCE, NUMB 4297, Seiten 547-559, 2006

[4] Robert van de Geijn, David Payne, Lance Shuler, Jerell Watts: *A Streetguide to Collective Communication and its Application*, 1996

[5] D. Culler, L. T. Liu, R. P. Martin and C. Yoshikawa: *Assessing fast network interfaces*, IEEE Micro, 16:35-43, 1996

[6] T. Kielmann, H. E. Bal and K. Verstoep. *Fast measurement of LogP parameters for message passing platforms*. In José D. P. Rolim, editor, IPDPS Workshops, volume 1800 of Lecture Notes in Computer Science, Seiten 1176-1183, Cancun, Mexico, Mai 2000, Springer-Verlag

[7] R. W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. Parallel Computing, 20(3):389-398, März 1994

[8] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, Thorsten von Eicken. *LogP: Towards a realistic model of parallel computation*. In Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming, Seiten 1-12, ACM Press, 1993.

[9] T. Hoefler, L. Cerquetti, T. Mehlan, F. Mietke, and W. Rehm. *A Survey of Barrier Algorithms for Coarse Grained Supercomputers*. Chemnitzer Informatik Berichte – CSR-04-03-2004. url: http://archiv.tu-chemnitz.de/pub/2005/0074/data/CSR-04-03.pdf

[10] T. Hoefler, T. Mehlan, F. Mietke, W. Rehm: *A Communication Model for Small Messages with InfiniBand*, PARS Proceedings, 2005

[11] G. Bilardi, K.T. Herley, and A. Pietracaprina: *BSP vs LogP Model*, Journal of Parallel and Distributed Computing, 44[1]:71-79, 1995.

[12] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm: *LogfP – A Model for small Messages in InfiniBand,*. Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, 25-29. April 2006

[13] C. Viertel: *Erweiterung eines existierenden InfiniBand Benchmarks, Studienarbeit*

[14] S. Fortune and J. Wyllie: *Parallelism in random access machines*. In STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing, Seiten 114–118. ACM Press,1978.

[15] J. von Neumann: *First draft of a report on the edvac*. Technical report, University of Pennsylvania, 1945

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

[17] Susanne E. Hambrusch and Asfaq A. Khokhar: *An architecture-independent model for coarse grained parallel machines*. In Proceedings of the 6-th IEEE Symposium on Parallel and Distributed Processing, 1994

[18] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman: *LogGP: Incorporating Long Messages into the LogP Model*. Journal of Parallel and Distributed Computing, 44(1):71–79, 1995.

[19] C. A. Moritz and M. I. Frank: *LoGPC: Modelling Network Contention in Message-Passing Programs*. IEEE Transactions on Parallel and Distributed Systems, 12(4):404, 2001.

[20] A. Agarwal: *Limits on Interconnection Network Performance*. IEEE Transactions on Parallel and Distributed Systems, 2(4):398–412, 1991.

[21] Fumihiko Ino, Noriyuki Fujimoto, and Kenichi Hagihara. LogGPS: *A Parallel Computational Model for Synchronization Analysis*. In PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming, Seiten 133–142.ACM Press, 2001.

[22] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. www.mpiforum.org, 1995

[23] R. Rabenseifner: *Optimization of Collective Reduction Operations.* International Conference on Computational Science 2004: 1-9

[24] MPI. *A Message-Passing Interface Standard*. http://www.mpi-forum.org/docs/mpi1-report.PDF

[25] V. Bala, J. Bruck, R.Cypher, P. Elustondo, A - Ho, C.-T. Ho, S. Kipnis, M. Snir: *CCL: A portable and tunable collective communication library fpr scalable parallel computers*, in IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 2, Feb. 1995, Seiten 154-164

[26] M. Barnett, S. Gupta, D. Payne, L. Shuler, R. van de Geijn, J. Watts: *Interprocessor collective communication library (InterCom)*, in Proceedings of Supercomputing ´94, Nov. 1994.

[27] E. Gabriel, M. Resch, R. Rühle: Implementing *MPI with optimized algorithms for metacomputing*, in Proceedings of the MPIDC´99, Atlanta, USA, März 1999, Seiten 31-41.

[28] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, J. Bresnahan: *Exploiting hierarchy in parallel computer networks to optimize collective operation performance*, in Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS ´00), 2000, Seiten 377-384.

[29] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal., Aske Plaat, Raoul A. F. Bhoedjang: *MPI´s reduction operations in clustered wide area systems*, in Proceedings of the Message Passing Interface Developer´s and User´s Conference 1999 (MPIDC´99), Atlanta, USA, März 1999, Seiten 43-52

[30] Edward K. Blum, Xin Wang, Patrick Leung: *Architectures and message-passing algorithms for cluster computing: Design and performance*, in Parallel Computing 26 (2000) 313-332.

[31] J. Bruck, C.-T. Ho, S.Kipnis, E. Upfal , D. Weathersby: *Efficient algorithms for all-to-all-communications in multiport message-passing systems*, in IEEE Transactions on Parallel and Distributed Systems, Vol. 8, No. 11, Nov. 1997, Seiten 1143-1156.

[32] Man D. Knies, F. Ray Barriuso, William J. Harrod, George B. Adams III: *SLICCC: A low latency interface for collective communications*, in Proceedings of the 1994 conference on Supercomputing, Washington, D.C, Nov.14-18, 1994, Seiten 89-96.

[33] Rajeev Thakur, William D. Gropp: *Improving the performance of collective operations in MPICH*, in Recent Advances in Parallel Virtual Machine and Message Passing Interface, proceedings of the 10th European PVM/MPI Users´Group Meeting, LNCS 2840, J. Dongarra, D. Laforenza, S. Orlando (Eds.), 2003, 257-267

[34] Rajeev Thakur, Rolf Rabenseifner, William Gropp: *Optimization of Collective Communication Operations in MPICH*. International Journal of High Performance Computing Applications, 2005

[35] Sathish S. Vadhiyar, Graham E. Fagg, Jack Dongarra: *Automatically tuned collective communications*, in Proceedings of SC2000, Nov. 2000.

[36] Ahmad Faraj, Xin Yuan: *Automatic Generation and Tuning of MPI Collective Communication Routines*, International Conference on Supercomputing, Proceedings of the 19th annual international conference on Supercomputing, Session 11, Seiten 393-402, 2005

[37] A. Bar-Noy, S. Kipnis, B. Schieber: *Optimal Computations of Census Functions in the Postal Model*, Discrete Applied Mathematics, vol. 58, no. 3, Seiten 213-222, April 1995

[38] R. Rabenseifner, P. Adamidis: *Collective Reduction Operations on Cray X1 and Other Platforms*, Proceedings of the CUG 2004 Conference, Knoxville, Tennessee, USA, 17.-21. Mai 2004

[39] F. Petrini, A. Moody, J. Fernandez, E. Frachtenberg, D.K. Panda: *NIC-based Reduction Algorithms for Large-scale Clusters*, International Journal of High Performance Computing and Networking 2006 - Vol. 4, No.3/4 Seiten 122 - 136

[40] Jehoshua Bruck, Ching-Tien Ho, Schlomo Kipnis, Eli Upfal, Derrick Weathersby: Efficient *algorithms for all-to-all communications in multiport message-passing systems*. IEEE Transactions on Parallel and Distributed Systems, 8(11):1143-1156, November 1997

[41] G. E. Fagg, A. Bukovsky, J. J. Dongarra: *HARNESS and fault tolerant MP,*. Parallel Computing, 27:1479-1496, 2001.

[42] Graham E. Fagg, Edgar Gabriel, Zizhong Chen, Thara Angskun, George Bosilca, Antonin Bukovski, J.J. Dongarra: *Fault tolerant communication library and applications for high performance*. In Los Alamos Computer Science Institute Symposium, Santa Fee, NM, 27.-29. Oktober 2003.

[43] Rob T. Aulwes, David J. Daniel, Nehal N. Desai, Richard L. Graham, L. Dean Risinger, Mitchel W. Sukalski, Mark A. Taylor, Timothy S. Woodall: *Architecture of LA-MPI, a network-fault-tolerant mpi*. In Los Alamos report LA-UR-03-0939, Proceedings of IPDPS, 2004.

[44] R. L. Graham, S.-E- Choi, D. J. Damiel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, M. W. Sukalksi: *A network-failure-tolerant message-passing system for terascale clusters* - Intnl. Journal of Parallel Programming, 31(4):285-303, August 2003.

[45] Greg Burns, Raja Daoud, James Vaigl: *LAM: An Open Cluster Environment for MPI*. In Proceedings of Supercomputing Symposium, Seiten 379-386, 1994

[46] Jeffrey M. Squyres, Andrew Lumsdaine: *A Component Architecture for LAM/MPI*. In Proceedings, 10th European PVM/MPI Users´s Group Meeting, Nummer 2840 in Lecture Notes in Computer Science, Venice, Italy, September/ Oktober 2003. Springer-Verlag.

[47] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, J. J. Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, Timothy S. Woodall: *Open MPI: Goals, concept, and design of a next generation MPI implementation*. In Proceedings, Euro PVM/MPI, Budapest, Hungary, Septem ber 2004

[48] InfiniBand Architecture Specification 1.0

[49] Tom Shanley: *InfiniBand Network Architecture*, Addison Wesley

[50] William T. Futral: *InfiniBand Architecture – Development and Deployment*, Intel Press

[51] Rainer Keller, Michael Resch: *Testing the correctness of MPI implementations*, Proceedings of The Fifth International Symposium on Parallel and Distri buted Computing (ISPDC'06) Seiten. 291-295

[52] Torsten Hoefler: *nbcbench*

http://www.unixer.de/research/nbcoll/perf/nbcbench -0.9.2.tgz

[53] Amith R. Mamidala, Sundeep Narravula, Abhinav Vishnu, Gopal Santhanaraman, Dhabaleswar K. Panda: *On using Connection-Oriented vs. Connection-Less Transport for Performance and Scalability of Collective and One-sided Operations: Trade-offs and Impact*, Principles and Practice of Parallel Programming, Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, Seiten 46-54

[54] Amith R. Mamidala, Jiuxing Liu, Dhabaleswar K. Panda: *Efficient Barrier and Allreduce on IBA clusters using hardware multicast and adaptive algorithms*, IEEE Cluster Computing 2004, Sept. 20-23 2004, San Diego, California

[55] Jiesheng Wu, Amith R. Mamidala, Dhabaleswar K. Panda: *Can NIC Memory in InfiniBand Benefit Communication Performance? – A Study with Mellanox Adapter*, OSU-CISRC-4/04-TR20.

[56] Joachim Worringen: *Pipelining and Overlapping for MPI Collective Operations*, Proceedings of the 28th Annual IEEE International Conference on Local Computer Networks, 2003, Seite 548

[57] Christian Bell, Dan Bonachea, Yannick Cote, Jason Duell, Paul Hargrove, Parry Husbands, Costin Iancu, Michael, Welcome, Katherine Yelick: *An Evaluation of Current High-Performance Networks*, Proceedings of the 17th International Symposium on Parallel and Distributed Processing, 2003, Seite 28.1

[58] Costin Iancu, Parry Husbands, Paul Hargrove: *HUNTing the Overlap*, Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, Pages: 279 - 290, 2005, IEEE Computer Society

[59] Ron Brightwell, Keith D. Underwood: *An Analysis of the Impact of MPI Overlap and Independent Progress*, Proceedings of the 18th annual international conference on Supercomputing, 2004, Seiten 298-305

[60] http://www-unix.mcs.anl.gov/mpi/mpi-test/tsuite.html

[61] T. Hoefler, J. Squyres, W. Rehm, A. Lumsdaine: *A Case for Non-Blocking Collective Operations*, Vol 4331/2006, In Frontiers of High Performance Computing and Networking - ISPA 2006 Workshops, Seiten 155-164, Springer Berlin / Heidelberg

A3 Thesen

1.) Kommunikationsmodelle wie BSP, PRAM, C^3 sind nicht für die Herleitung optimaler kollektiver Algorithmen in MPI geeignet.
2.) Das LogP Modell ist nicht für kleine Nachrichten im InfiniBand ™ Netzwerk geeignet.
3.) Das LogfP Modell als Vereinfachung des LoP Modells ist für kleine Nachrichten im InfiniBand™ Netzwerk geeignet.
4.) Open MPI bietet ein erweiterbares Framework, welches ideal für die Implementierung neuer kollektiver Algorithmen ist.
5.) Kollektive Operationen können durch Ausnutzung von speziellen Eigenschaften des InfiniBand™ Netzwerkes beschleunigt werden.
6.) Das LogfP Modell als Vereinfachung des LoP Modells kann genutzt werden, um kollektive Operationen zu beschleunigen, die kleine Nachrichten für die Kommunikation nutzen.
7.) Überlappung von Kommunikation und Berechnung ist bedeutend für optimale kollektive Operationen mit Berechnung.