

TU Chemnitz
Fakultät für Informatik
Fachgebiet Rechnerarchitektur
Studienarbeit

Betreuer: Torsten Höfler

Dozent: Prof. W. Rehm

Erweiterung eines existierenden Infiniband Benchmarks

Carsten Viertel
Diplomstudiengang Informatik
09. März 2006

Inhaltsverzeichnis

1	Einleitung	3
2	Infiniband	4
2.1	Einleitung	4
2.2	Komponenten und Topologie	4
2.3	Kommunikation	4
2.3.1	Queuing	4
2.3.2	Verbindungsarten	5
2.3.3	Keys	6
2.3.4	Speicherzugriff	7
2.3.5	Partitionen	7
2.3.6	Virtual Lanes	7
2.3.7	Adressierung	8
2.4	Übertragungsmodi im Detail	8
2.4.1	Send	8
2.4.2	RDMA Write	8
2.4.3	RDMA Read	9
3	Der existierende Benchmark	10
3.1	Überblick	10
3.2	mpi_liba_bench.h	10
3.3	mpi_liba_bench.c	11
3.4	scenario_1.sub.c	12
3.5	scenario_2.sub.c	12
3.6	Geplante Erweiterungen	13
4	Die Erweiterungen	14
4.1	Unreliable Datagram	14
4.1.1	Funktionsweise	14
4.1.2	Implementierung	14
4.2	Multicast	21
4.2.1	Funktionsweise	21
4.2.2	Implementierung	22
4.3	RDMA	25
4.3.1	Implementierung	25
5	Benchmark Ergebnisse und Auswertung	31
5.1	Messumgebung und gemessene Zeiten	31
5.2	Ergebnisse für SEND, RDMAW	31
5.2.1	Ergebnisse für kleine Nachrichten	31
5.2.2	Ergebnisse für große Nachrichten	34
5.3	Ergebnisse für RDMA	36
5.4	Auswertung	38
5.4.1	SEND mit Reliable und Unreliable Connection	38
5.4.2	RDMAW mit Reliable und Unreliable Connection	39
5.4.3	SEND mit Unreliable Datagram	39
5.4.4	SEND mit Multicast	40
5.4.5	RDMA mit Reliable Connection	40

5.4.6	Fazit	40
6	Schlussfolgerungen und Ausblick	41
A	Flussdiagramme	43
A.1	mpi_lba_bench.c	44
A.2	scenario_1.sub.c	48
A.3	scenario_2.sub.c	51

1 Einleitung

Infiniband [1] wird zunehmend als Verbindungsnetzwerk für Cluster eingesetzt. Dadurch wird es nötig existierende Bibliotheken für parallele Programmiersprachen an das neue Netzwerk bestmöglich anzupassen.

Ein wichtiger Bestandteil paralleler Programmiersprachen sind dabei kollektive Operationen, die es erfordern, eine Nachricht von einem Knoten zu vielen anderen oder auch von vielen Knoten an einen einzelnen zu senden.

Um herauszufinden, welche Verbindungsarten und Operationen am besten für diese kollektiven Operationen geeignet sind, wurde ein Benchmark entwickelt. Ziel dieser Studienarbeit ist es, dieses Programm zu erweitern, auf einem Cluster zu testen und die Ergebnisse auszuwerten.

In Abschnitt 2 wird zunächst ein kurzer Überblick über Infiniband gegeben, um danach näher auf die möglichen Übertragungsmodi eingehen zu können. In Abschnitt 3 wird die Funktionsweise und Realisierung des existierenden Benchmarks beschrieben. Der folgende Teil 4 widmet sich dann der Erweiterung des Programms. Im Anschluss folgen die Benchmarkergebnisse und eine Auswertung. Der letzte Abschnitt liefert Schlussfolgerungen aus den Ergebnissen und einen Ausblick auf weiterführende Entwicklungen.

2 Infiniband

2.1 Einleitung

Infiniband ist ein Verbindungsnetzwerk, das ursprünglich sowohl für die Verbindung eines Prozessors mit I/O Geräten als auch für die Verbindung zwischen mehreren Prozessoren entwickelt wurde. Um dieses Ziel zu erreichen, gründeten 1999 mehrere Firmen, unter anderem Intel, IBM, Sun Microsystems, HP und 3COM die Infiniband Trade Association. Diese definierte mit der Infiniband Architecture (IBA) den Standard, der die Grundlage für die Implementierungen von Infiniband darstellt.

2.2 Komponenten und Topologie

Bei Infiniband wird die Verbindung zwischen den Endknoten als Punkt zu Punkt Verbindung mittels Switches realisiert. Das gesamte Kommunikationsnetzwerk (Switches, Router und Leitungen) wird bei Infiniband 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 Host Channel Adapter (HCA) an Infiniband angeschlossen und bei der Peripherie übernimmt dies ein Target Channel Adapter (TCA). Der Übergang zu anderen Netzen (weitere Infiniband Subnetze, LAN, WAN, ...) wird mithilfe von Routern hergestellt.

2.3 Kommunikation

2.3.1 Queuing

Die Kommunikation zwischen den Nutzern eines Channel Adapters (Consumer) erfolgt mittels Queues (Schlangen). Diese existieren immer als Queue Pair (QP) mit einer Queue zum Senden und einer zum Empfangen (Abbildung 1).

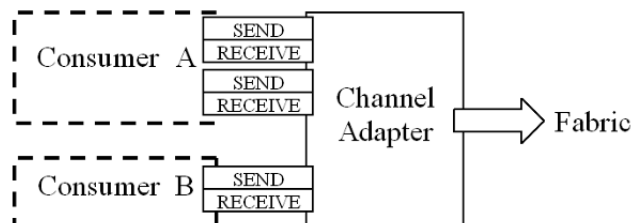


Abbildung 1: Zwei Consumer mit drei QPs an einem Channel Adapter

Die Steuerung der Queues und somit der Kommunikation erfolgt mittels Work Requests. Diese werden von einem Consumer gesendet und erzeugen ein Work Queue Element (WQE) in der entsprechenden Queue (Send oder Receive). Sobald der Channel Adapter ein WQE abgearbeitet hat, erstellt er ein Completion Queue Element (CQE) in der Completion Queue. Jedes CQE enthält alle nötigen Informationen, um eine Operation abzuschließen.

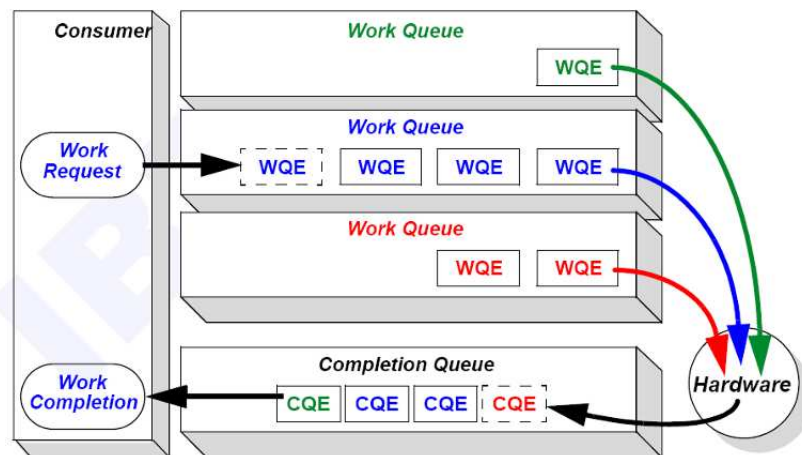


Abbildung 2: Work Queues und Completion Queue (Quelle IBA)

Jeder Consumer kann mehrere völlig voneinander unabhängige Queue Pairs und dazugehörige Completion Queues erstellen (Abbildung 2). Die Queues werden somit als virtuelles Kommunikationsinterface von der Hardware bereitgestellt und stellen eine private Ressource für den Consumer dar.

Die möglichen Operationen auf WQs sind Send, Remote Memory Access (RDMA) und Memory Binding für die Send Queue sowie ein Receive WQE für die Receive Queue.

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

RDMA Hier wird zusätzlich zu den Daten der SEND Operation noch eine Speicheradresse des Ziels mit angegeben. Es existieren READ, WRITE und ATOMIC Operationen. ATOMIC ist dabei ein READ eines 64bit Words des Ziels, das eventuell (swap if equal bzw. add) noch ein Update des Inhalts auslöst.

MB Memory Binding erlaubt es einem Consumer einen Speicherbereich festzulegen, den er mit anderen teilt, sowie die Lese- und Schreibrechte dafür zu vergeben. Es wird dabei ein Memory Key verschickt, den andere Consumer danach bei ihren RDMA Operationen verwenden.

Receive Ein Receive WQE legt fest, wo empfangene Daten von einem anderen Consumer, der eine Send Operation ausführt, abgelegt werden. Bei erfolgreichem Empfang werden ein Receive WQE verbraucht, die Daten an der entsprechenden Stelle abgelegt und ein CQE erzeugt.

2.3.2 Verbindungsarten

Indem man eine bestimmte Verbindungsart (Type of Service) wählt, legt man fest wie Quell und Ziel QP miteinander kommunizieren. Beide müssen dabei

gleich konfiguriert sein. Die Verbindung wird nach folgenden Kriterien klassifiziert:

connection oriented	datagram
Ein QP ist genau mit einem anderen QP verbunden. Ein Work Request resultiert in einer Nachricht an das Ziel QP.	Ein QP kann Nachrichten von beliebigen QPs empfangen bzw. Nachrichten an beliebige QPs senden, die entsprechend konfiguriert sind.
acknowledged (reliable)	unacknowledged (unreliable)
Ein QP schickt für jedes empfangene Paket eine Antwort zurück. Mögliche Antworten sind ACK, NAK oder Antwortdaten.	Es wird keine Antwort geschickt
IBA transport	other transport
Kommunikation über das in der IBA definierte Protokoll.	Erlaubt es IPv6 und Ethernet Pakete über den Channel Adapter zu verschicken.

Tabelle 1: Kriterien für die Verbindungsarten

In Tabelle 2 sind die möglichen Verbindungsarten und ihre Eigenschaften aufgelistet.

Service Type	Connection Oriented	Acknowledged	Transport
Reliable Connection	yes	yes	IBA
Unreliable Connection	yes	no	IBA
Reliable Datagram	no	yes	IBA
Unreliable Datagram	no	no	IBA
RAW Datagram	no	no	RAW

Tabelle 2: Verbindungsarten und ihre Eigenschaften

2.3.3 Keys

Keys (Schlüssel) werden in Nachrichten verwendet, um verschiedene Zugriffsrechte zu gewährleisten. Folgende Keys gibt es:

Management Key (M_Key) Wird vom Subnet Manager (SM) an Channel Adapter Ports vergeben. Sobald er aktiviert ist, sorgt er dafür, dass Managementpakete ohne den korrekten M_Key verworfen werden.

Baseboard Management Key (B_Key) Wird vom Subnet Baseboard Manager vergeben und hat die gleiche Aufgabe für bestimmte Management Datagramme (MAD) wie der M_Key für die Managementpakete vom SM.

Partition Key (P_Key) Wird vom Partition Manager über den Subnet Manager vergeben. Jeder Channel Adapter Port hat eine Tabelle mit 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 IBA transport Paket mitgesendet.

Queue Key (Q_Key) Wird vom Channel Adapter vergeben und regelt Zugriffsrechte bei Datagramm Verbindungen. Während des Verbindungsaufbaus werden die Q_Keys für bestimmte QPs ausgetauscht und im folgenden bei der Kommunikation zwischen den QPs immer mitgesendet.

Memory Keys (L_Key und R_Key) Wird vom Channel Adapter vergeben, ermöglicht die Verwendung von virtuellen Speicheradressen und regelt den Speicherzugriff. Wenn ein Consumer einen Teil seines Speichers für den RDMA Zugriff freigibt, erhält er einen L_Key sowie einen R_Key vom Channel Adapter. Der L_Key wird dafür verwendet dem QP lokalen Speicher bereitzustellen und der R_Key wird an entfernte Consumer gesendet, um ihnen den RDMA Zugriff zu ermöglichen.

2.3.4 Speicherzugriff

Mithilfe der R_Keys und der L_Keys ist es dem Channel Adapter möglich die virtuellen Adressen nach Bedarf in physikalische Adressen umzuwandeln. Mithilfe so genannter Protection Domains ist es auch möglich den Speicherzugriff für verschiedene QPs unterschiedlich zu regeln. Dafür wird zuerst eine Protection Domain angelegt, der im folgenden freigegebener Speicher und die gewünschten QPs zugeordnet werden. Mithilfe unterschiedlicher Protection Domains lässt sich somit eine differenzierte Speicherfreigabe erreichen.

2.3.5 Partitionen

Eine Partition definiert eine Gruppe von Endknoten die miteinander kommunizieren können. Jeder Port eines Endknotens ist mindestens Mitglied einer Partition und kann in mehreren Mitglied sein. Partitionen werden mithilfe von P_Keys gesteuert. Switches und Router können optional dafür genutzt werden, die Partitionierung durchzusetzen, indem sie eine Reihe P_Keys erhalten und im folgenden alle Pakete mit ungültigen P_Keys verwerfen.

2.3.6 Virtual Lanes

Virtual Lanes (VL) ermöglichen mehrere logische Verbindungen auf einer einzigen physikalischen Verbindung. Eine Virtual Lane repräsentiert dabei einen Sende- und einen Empfangsbuffer eines Ports. Alle Ports unterstützen VL15, welche für das Subnet Management reserviert ist.

Darüber hinaus wird mindestens eine Daten VL (VL0) unterstützt. Es können außerdem weitere 14 Daten VLs (VL1..VL14) bereitgestellt werden. Die tatsächliche Anzahl von Daten VLs, die ein Port nutzt, wird vom Subnet Manager so festgelegt, dass beide Endknoten einer Verbindung die gleiche Anzahl verwenden. Solange dies nicht geschehen ist, wird nur VL0 genutzt.

Der Port verhindert mit einer separaten Flusssteuerung auf jeder VL, dass hoher Traffic auf einer VL die Kommunikation auf einer anderen beeinträchtigt.

Welche VL von einem Paket tatsächlich benutzt wird, bestimmt der Service Level im Header des Pakets.

2.3.7 Adressierung

Jeder Endknoten kann einen oder mehrere Channel Adapter haben. Ein Channel Adapter kann wiederum einen oder mehrere Ports besitzen. Weiterhin hat ein Channel Adapter eine Reihe von QPs.

Jedes QP besitzt eine Queue Pair Nummer (QPN), die für die richtige Zuordnung von Paketen an die QPs benötigt wird. Die QPN ist in allen Paketen (außer RAW) enthalten.

Ein Port hat eine Local ID (LID), die vom Subnet Manager des lokalen Subnets vergeben wird, und mindestens eine Global ID (GID) im Format einer IPv6 Adresse. In einem Subnet sind die LIDs eindeutig und werden von den Switches zum richtigen Verteilen der Pakete innerhalb des Subnets verwendet. Ein Paket enthält dazu eine Source LID (SLID), die die Quelle angibt, und eine Destination LID (DLID), die das Ziel spezifiziert. GIDs sind global eindeutig und werden bei Paketen in einem optionalen Global Route Header (GRH) verwendet. Der GRH, in dem sich wieder die GIDs von Quelle und Ziel finden, wird von Switches ignoriert und nur von Routern ausgewertet. Jeder Channel Adapter sowie jeder seiner Ports hat einen Globally Unique Identifier (GUID), der vom Hersteller vergeben wird. Aus dem GUID und einem 64bit Subnetpräfix wird die global gültige GID zusammengesetzt.

2.4 Übertragungsmodi im Detail

2.4.1 Send

Die Operation SEND wird von allen Verbindungsarten unterstützt. Um den Empfang einer Nachricht vorzubereiten, muss der Empfänger dazu zuerst einen Receive Request für das benutzte QP posten. Darin wird festgelegt, wo die ankommende Nachricht im Speicher abgelegt werden soll und wieviel Platz dafür zur Verfügung steht.

Der Sender postet einen Send Request für das benutzte QP, welcher unter anderem die Speicheradresse, von der gelesen werden soll, sowie die Größe der zu sendenden Nachricht enthält. Bei Verwendung von Unreliable Datagram oder Multicast enthält der Send Request ausserdem die Adresse des Ziels bzw. der Ziele. Der CA kümmert sich nun darum, dass die Nachricht aus der im SR angegebenen Speicheradresse gelesen, in Pakete aufgeteilt und an den Empfänger geschickt wird.

Auf Empfängerseite legt das aktuelle WQE in der RQ fest, wo die ankommenden Daten abgelegt werden. Sobald die komplette Nachricht angekommen ist, wird ein CQE in der RQ CQ erstellt und das aktuelle WQE in der RQ gelöscht.

Wenn Unreliable Datagram, Unreliable Connection oder Multicast verwendet werden, wird beim Sender ein CQE in der SQ CQ erstellt, nachdem die komplette Nachricht verschickt wurde. Bei Verwendung von Reliable Connection oder Reliable Datagram geschieht dies erst, nachdem der Empfänger den Empfang des letzten Pakets bestätigt hat.

2.4.2 RDMA Write

Die Operation RDMAW wird von den Verbindungsarten Reliable Connection, Unreliable Connection und Reliable Datagram unterstützt. Im Gegensatz zu

SEND ist kein Receive Request beim Empfänger nötig, da die Speicheradresse für die ankommende Nachricht vom Sender festgelegt wird. Um das zu ermöglichen, muss der Empfänger jedoch zuvor einen Speicherbereich festlegen, der für RDMA Operationen verwendet werden kann und dem Sender die Adresse und Größe dieses Speicherbereichs, sowie den `r_key`, der für den Zugriff darauf benötigt wird, mitteilen. Der `r_key` beinhaltet dabei auch die erlaubten Operationen auf dem Speicherbereich, wie RDMA, RDMAW oder Atomic. Falls die RDMAW Operation zusätzlich Immediate Daten enthält, so werden diese mit dem letzten (oder einzigen) Paket versendet und es wird ein RQ WQE auf Empfängerseite benötigt, das dann die Immediate Daten enthält.

Der Sender postet einen RDMAW Work Request, welcher die Speicheradresse, von der gelesen werden soll, sowie die Größe der zu sendenden Nachricht enthält. Außerdem enthält der Work Request die Adresse, an der die Daten beim Empfänger abgelegt werden sollen, die Länge der Nachricht, sowie den benötigten `r_key`. Der CA übernimmt wieder die Aufteilung der Nachricht in Pakete und das Versenden an den Empfänger.

Beim Empfänger sorgt die RQ dafür, dass die Daten an die angegebene Speicheradresse geschrieben werden, sobald der `r_key` überprüft wurde. Falls die Nachricht Immediate Daten enthält, wird nach Empfang des letzten Pakets ein CQE in der RQ CQ erstellt und das aktuelle RQ WQE entfernt.

Sobald die komplette Nachricht an den Empfänger verschickt wurde, wird beim Sender ein CQE in der SQ CQ erstellt und das aktuelle WQE in der SQ gelöscht.

2.4.3 RDMA Read

Die Operation RDMA wird von Reliable Connection und Reliable Datagram unterstützt. Der Empfänger eines RDMA Requests ist dabei derjenige, der die zu lesende Nachricht bereitstellt. Dabei muss analog zu RDMAW vorher ein Speicherbereich für RDMA samt `r_key` angelegt werden. Dessen Adresse und der `r_key` werden der Anwendung am entfernten CA, die den RDMA Requests durchführen will, mitgeteilt.

Danach kann von dieser Anwendung ein RDMA WR für die SQ gepostet werden. Enthalten sind dabei die Adresse, von der gelesen werden soll, der für den Zugriff benötigte `r_key`, die Länge der zu lesenden Daten sowie die Adresse eines oder mehrerer lokaler Empfangsbuffer.

Der Empfänger des RDMA Requests überprüft nun den `r_key`. Falls dieser korrekt ist, wird die Nachricht von der angegebenen Adresse gelesen, in Pakete aufgeteilt und zurückgesendet. Es wird dafür kein WQE in der RQ benötigt.

Nachdem der Sender des RDMA Requests das letzte Antwortpaket und damit die komplette zu lesende Nachricht erhalten hat, wird ein CQE in der SQ CQ erstellt und das aktuelle WQE in der SQ gelöscht.

Bei Verwendung von Reliable Connection können mehrere RDMA Requests abgesetzt werden, ohne dass die vorherigen schon abgeschlossen sind. Der Empfänger der Requests kann jedoch festlegen, wieviele ausstehende RDMA Requests er zulassen will.

3 Der existierende Benchmark

3.1 Überblick

Auf den folgenden Seiten findet sich eine Beschreibung zum schon existierenden Benchmark. Zusätzlich gibt es in Anhang A Flussdiagramme zu Teilen des Programms. `mpi_iba_bench.c` ist das Grundprogramm, in das die entsprechenden Testszenarien eingebunden werden. Welche Zeiten mit den einzelnen Szenarien gemessen werden, ist in Tabelle 3 aufgelistet. In `create_qp.c` ist außerdem eine Funktion zur Erstellung der QPs definiert.

scenario_1.sub.c	
MEA_POST_RR_CPU_OVERHEAD	misst die Zeit, die für das Absetzen eines einzelnen Receive Requests benötigt wird
MEA_POST_SR_CPU_OVERHEAD	misst die Zeit, die für das Absetzen eines einzelnen Send Requests benötigt wird
MEA_POLL_CQ_CPU_OVERHEAD	misst die Zeit, die für eine einzelne Abfrage der Completion Queue benötigt wird
MEA_SEND_TIME	misst die Zeit, die für das Abfragen der Completion Queue nach Absetzen eines Send Requests benötigt wird
scenario_2.sub.c	
MEA_RTT_TIME	misst die Round Trip Time für eine 1:n Kommunikation
MEA_SENDCOMPLETE_TIME	misst die Zeit, die für das Abfragen der Completion Queue nach Absetzen von Send Requests an alle n Knoten benötigt wird

Tabelle 3: Szenarien und gemessene Zeiten

3.2 `mpi_iba_bench.h`

In `mpi_iba_bench.h` lassen sich eine ganze Reihe Einstellungen vornehmen, die den Ablauf des Programms beeinflussen. Eine Übersicht über die wichtigsten Einstellungen gibt folgende Liste:

TRANSPORT legt die verwendete Verbindungsart fest. Möglich sind Reliable Connection, Unreliable Connection, Unreliable Datagram und Multicast.

MODE bestimmt, welcher Übertragungsmodus verwendet wird. SEND sowie RDMAW und RDMAV stehen zur Auswahl. Nicht alle Kombinationen von TRANSPORT und MODE sind möglich.

MEASURE legt fest, welche Zeit gemessen werden soll und somit auch welches Szenario verwendet wird (siehe Tabelle 3).

MEMSIZE ist die maximale Buffergröße für das Senden und Empfangen.

REPETITIONS gibt an, wie oft ein einzelner Benchmarkschritt wiederholt wird.

MAXPOSTS ist die maximale Anzahl in einem Schritt gesendeter Nachrichten für Szenario 1.

In `mpi_liba_bench.h` sind außerdem folgende Makros definiert:

MEMSTEP(memsize) verdoppelt *memsize* bis der Wert *MEMSIZE* erreicht ist, wonach *memsize* auf 0 gesetzt wird.

SET_MEM(i, addr, memsize) kopiert den Wert *i* in Adresse *addr+memsize*.

WAIT_MEM_RECV(i, addr, memsize, numhosts, size) wartet, bis die Speicheradressen [*addr + size + (0 .. (numhosts -1)) * MEMSIZE*] den Wert *i* enthalten.

VAPI_POLL_CQ wartet, bis mittels `VAPI_poll_cq` ein CQE erhalten wurde.

3.3 `mpi_liba_bench.c`

Da der Benchmark MPI verwendet, muss dieses zuerst initialisiert werden. Die Anzahl der beteiligten Prozesse wird dabei in *procs* und die Prozessnummer in *rank* gespeichert.

Danach werden Sende- und Empfangsbuffer angelegt und der HCA geöffnet. Im Anschluss werden Protection Domain und Completion Queues (CQs) für Send (SQ CQ) und Receive (RQ CQ) Queues angelegt, um die Erstellung der Queue Pairs (QPs) zu ermöglichen. Im Falle von Prozess 0 werden dabei *procs-1* QPs erstellt. Die anderen Prozesse erstellen immer ein QP. Es wird dabei jeweils ein QP von Prozess 0 mit einem QP eines anderen Prozesses verbunden.

Nachdem die Speicherbereiche von Sende- und Empfangsbuffer registriert wurden, tauschen die Prozesse 1 bis *procs-1* und Prozess 0 mittels MPI die *r_keys* und Speicheradressen ihrer Empfangsbuffer aus. Dies ist eine Voraussetzung für RDMAR und RDMAW.

Nun werden in Abhängigkeit vom gewählten Übertragungsmodus Sende- und Empfangslisten angelegt. Prozess 0 legt dabei für jedes QP eine eigene Liste an. Dies schliesst die Vorbereitungen für das Senden und Empfangen von Nachrichten ab.

Bevor der eigentliche Benchmark beginnt, werden einige Pakete gesendet, um zu verhindern, dass bei den ersten Paketen auftretende Verzögerungen die Ergebnisse verfälschen.

Nachdem *memsize* auf 1 gesetzt wurde, folgt der Code von `scenario_1.sub.c` oder `scenario_2.sub.c` (siehe Tabelle 3), in einer Schleife von *memsize* = 1 bis *memsize* = 0. Am Ende eines jeden Durchlaufs werden die Benchmarkergebnisse ausgegeben und `MEMSTEP(memsize)` ausgeführt. `MEMSTEP` setzt *memsize* dabei auf 0, sobald *MEMSIZE* erreicht ist.

Nachdem der Benchmarkteil beendet ist, werden die registrierten Speicherbereiche, Queue Pairs, Completion Queues und das HCA Handle freigegeben, der HCA geschlossen und MPI beendet.

3.4 scenario_1.sub.c

Szenario 1 wird für das Messen von CPU Overheads bei verschiedenen Operationen verwendet. Dazu werden 1 bis MAXPOSTS Nachrichten von Prozess 1 an Prozess 0 versendet.

Da Szenario 1 nur für 2 Prozesse geeignet ist, wird am Anfang überprüft, ob auch genau 2 Prozesse aktiv sind. Falls das nicht der Fall ist, wird das Programm beendet.

Der ganze Benchmarkteil wird von einer äußeren Schleife umschlossen, die REPETITIONS mal durchlaufen wird. Außerdem gibt es noch eine innere Schleife, die von $i = 1$ bis MAXPOSTS läuft.

Falls der Übertragungsmodus SEND ist, postet Prozess 1 i Receive Requests (RRs). Dies bereitet den Empfang von i Nachrichten vor. Danach werden beide Prozesse mittels MPIBarrier synchronisiert, um sicherzustellen, dass Prozess 0 erst sendet, nachdem die RRs von Prozess 1 gepostet wurden.

Jetzt setzt Prozess 0 i Nachrichten an Prozess 1 ab und wartet danach bis i CQEs in der SR CQ angekommen sind. Falls der Übertragungsmodus SEND ist, wartet Prozess 1 im Gegenzug auf i CQEs in der RR CQ. Dies schliesst einen Durchlauf der inneren Schleife ab.

3.5 scenario_2.sub.c

Szenario 2 wird hauptsächlich zum Messen der Round Trip Time einer 1:n und n:1 Kommunikation verwendet. Der Ablauf ist in Bild 3 illustriert.

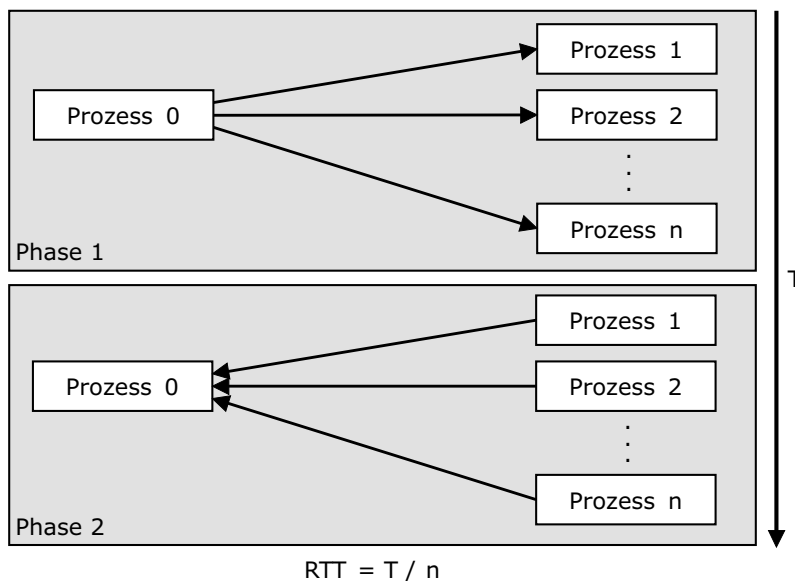


Abbildung 3: Ablauf Szenario 2

Wie auch bei Szenario 1 wird der Benchmarkteil von 2 Schleifen umschlossen. Die äußere wird REPETITIONS mal durchlaufen und die innere läuft von $i = 1$ bis $procs-1$. An der Kommunikation nehmen dabei immer die Prozesse 0 bis i teil.

Am Anfang eines jeden Durchlaufs werden Sende- und Empfangsbuffer gelöscht. Falls der Übertragungsmodus SEND ist, postet Prozess 0 jeweils einen Receive Request für die QPs 0 bis $i-1$. Die Prozesse 1 bis i posten genau einen Receive Request. Danach werden alle Prozesse mittels MPI_Barrier synchronisiert.

Jetzt wird die Variable *state*, deren Ausgangswert 0 ist, um 1 erhöht und das Makro SET_MEM(*state*, *s_addr*, *memsizesize*) ausgeführt. Damit wird der Wert von *state* an die Adresse *s_addr* + *memsizesize* kopiert, welche dem letzten Byte der zu sendenden Nachricht entspricht. Dies ermöglicht es später zu testen, ob ein entfernter Prozess eine RDMAW Operation abgeschlossen hat.

Prozess 0 postet jetzt jeweils einen Send Request für die QPs 0 bis $i-1$ und wartet danach, bis i CQEs in der SQ CQ angekommen sind. Die Prozesse 1 bis i warten währenddessen auf das Eintreffen der Nachricht. Falls der Übertragungsmodus SEND ist, wird auf ein CQE in der RQ CQ gewartet. Beim Modus RDMAW warten sie mit WAIT_MEM_RECV(*state*, *r_addr*, MEMSIZE, 1, *memsizesize*) darauf, dass das letzte Byte der aktuellen Nachricht mit Größe *memsizesize* im Empfangsbuffer ankommt.

In der zweiten Phase senden jetzt die Prozesse 1 bis i an Prozess 0 zurück, welcher auf die Nachrichten wartet.

3.6 Geplante Erweiterungen

Der Benchmark unterstützt im Moment Reliable und Unreliable Connection mit den Modi SEND und RDMAW. Als Erweiterungen geplant sind RDMAW für Reliable Connection sowie SEND für Unreliable Datagram und Multicast.

4 Die Erweiterungen

In den folgenden Abschnitten werden die Erweiterungen des Benchmarks und aufgetretene Probleme näher erläutert. Eine Übersicht über die Eigenschaften der verschiedenen Verbindungsarten findet sich in Abschnitt 2.3.2.

4.1 Unreliable Datagram

Als erstes wurde der Benchmark um die Verbindungsart Unreliable Datagram erweitert. Da bisher nur Reliable und Unreliable Connection unterstützt wurden, ergeben sich Änderungen bei der Queue Pair Erstellung, bei der Vorbereitung der Send Requests und auch im Benchmarkteil. Zuerst soll in Abschnitt 4.1.1 die Funktionsweise von Unreliable Datagram näher erläutert werden, um danach auf die tatsächliche Implementierung, sowie dabei aufgetretene Probleme eingehen zu können.

4.1.1 Funktionsweise

Ein Unreliable Datagram Queue Pair ist wie auch die Reliable und Unreliable Connection QPs fest mit einem Port verbunden. Im Unterschied zu RC und UC QPs ist es jedoch nicht mit einem anderen Queue Pair verbunden. So ist es möglich, von einem Queue Pair aus Nachrichten an mehrere andere QPs zu verschicken, wie in Abbildung 4 illustriert.

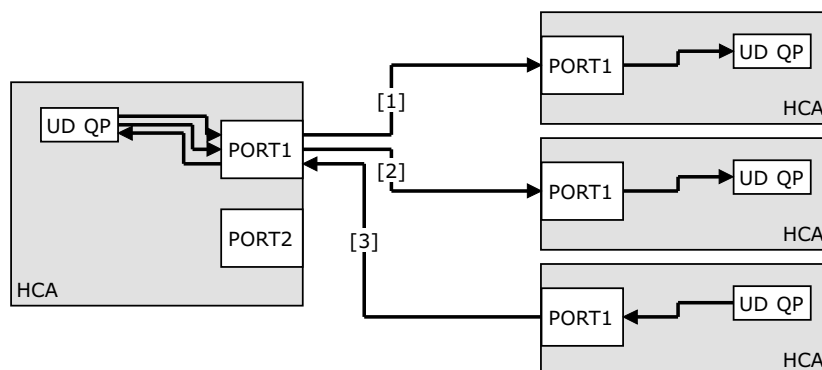


Abbildung 4: Kommunikation mittels Unreliable Datagram Queue Pairs

Bei einem UD QP wird kein ACK/NAK Protokoll verwendet, was zwar die Auslastung des Netzwerks verringert, aber dazu führt, dass die Übertragung einer Nachricht nicht garantiert werden kann. Eine weitere Einschränkung bei UD QPs ist die Nachrichtengröße. Diese darf die maximal mögliche Paketgröße auf dem gewählten Pfad durch das Netzwerk nicht überschreiten.

4.1.2 Implementierung

Wie schon in Abschnitt 4.1.1 erläutert, ist ein UD QP nicht fest mit einem anderen Queue Pair verbunden. Dadurch war es nötig die Erstellung der QPs abzuändern. Die Funktion für Reliable und Unreliable Connection beinhaltete

einen Austausch der LID und QP Nummer zwischen Knoten 0 und allen anderen Knoten. Da bei Verwendung von Unreliable Datagram QPs nur ein QP für die Kommunikation zwischen Knoten 0 und allen anderen Knoten nötig ist, muss die Funktion zum Erstellen der QPs auch bei Knoten 0 nur einmal aufgerufen werden. Dadurch war es einfacher eine neue Funktion für UD QPs zu verwenden, anstatt die alte Funktion zu erweitern.

Funktion zur Queue Pair Erstellung

Durch die Änderung beim Aufruf der Funktion ergeben sich auch minimal andere Parameter für die neue Funktion `create_dg_qp` im Gegensatz zu `create_qp`:

```
//Funktion zum Erstellen von RC/UC QPs
VAPI_qp_hdl_t create_qp(    int rank,
                           int remote,
                           VAPI_hca_hdl_t hca_hdl_p,
                           VAPI_cq_hdl_t sr_cq_hdl_p,
                           VAPI_cq_hdl_t rr_cq_hdl_p,
                           VAPI_pd_hdl_t pd_hdl_p);

//Funktion zum Erstellen von UD QPs
VAPI_qp_hdl_t create_dg_qp( int rank,
                            VAPI_hca_hdl_t hca_hdl_p,
                            VAPI_cq_hdl_t sr_cq_hdl_p,
                            VAPI_cq_hdl_t rr_cq_hdl_p,
                            VAPI_pd_hdl_t pd_hdl_p);
```

Wie in der Funktionsdefinition zu sehen, werden der rank des Prozesses sowie Handles für den HCA, die Completion Queues und die Protection Domain übergeben. Da keine Kommunikation zwischen den Prozessen zur Erstellung des QPs nötig ist, fällt `remote` als Parameter weg.

Nun zum eigentlichen Inhalt der neuen Funktion. Die folgenden Initialisierungsparameter für das QP wurden von der alten Funktion übernommen:

```
qp_init_attr_p.sq_cq_hdl = sr_cq_hdl_p;
qp_init_attr_p.rq_cq_hdl = rr_cq_hdl_p;
qp_init_attr_p.cap.max_oust_wr_sq = 10000;
qp_init_attr_p.cap.max_oust_wr_rq = 10000;
qp_init_attr_p.cap.max_sg_size_sq = 10;
qp_init_attr_p.cap.max_sg_size_rq = 10;
qp_init_attr_p.cap.max_inline_data_sq = 10;
qp_init_attr_p.sq_sig_type = VAPI_SIGNAL_ALL_WR;
qp_init_attr_p.rq_sig_type = VAPI_SIGNAL_ALL_WR;
qp_init_attr_p.pd_hdl = pd_hdl_p;
```

Die ersten beiden Parameter bestimmen welche CQs für die Send und Receive Queue des QPs verwendet werden sollen. Danach wird festgelegt, wieviele Work Requests sich in der Send und Receive Queue befinden dürfen und wieviele verschiedene Buffer in den Scatter Gather Listen für Send und Receive Queue bestimmt werden können. Es folgt die maximale Anzahl Bytes, die als Inline Data in der Send Queue erlaubt sind, wobei dieser Parameter laut Mellanox Dokumentation [3] im Moment nicht ausgewertet wird. Per `VAPISIGNAL_ALL_WR`

wird festgelegt, dass alle fertiggestellten Work Requests ein Completion Queue Element generieren. Als letzter Initialisierungsparameter wird nun noch die verwendete Protection Domain zugewiesen.

Jetzt kann das Queue Pair erstellt werden:

```
ret = VAPI_create_qp( hca_hndp_p,
                    &qp_init_attr_p,
                    &qp_hndl_p,
                    &qp_prop_p      );
```

Nach dem Aufruf von `VAPI_create_qp` befindet sich das QP im *reset* Zustand. Um Nachrichten empfangen und senden zu können, muss das QP noch in den *ready to send* Zustand versetzt werden. Dies geschieht in mehreren Schritten mittels `VAPI_modify_qp`, wobei in jedem Schritt noch bestimmte Parameter gesetzt werden müssen. Mit `QP_ATTR_MASK_SET` kann dabei festgelegt werden, welche Parameter im aktuellen Schritt geändert werden sollen und `QP_ATTR_MASK_CLR_ALL` setzt die Auswahl für alle Parameter zurück.

Der erste Zustandswechsel erfolgt von *reset* zu *init*:

```
/* Transition RST to INIT */
QP_ATTR_MASK_CLR_ALL(qp_attr_mask_p);
// QP State
QP_ATTR_MASK_SET( qp_attr_mask_p, QP_ATTR_QP_STATE);
qp_attr_p.qp_state = VAPI_INIT;

// partition key index
QP_ATTR_MASK_SET( qp_attr_mask_p, QP_ATTR_PKEY_IX);
qp_attr_p.pkey_ix = 0; // first partition key

// queue key -> only for datagram (RD, UD)
QP_ATTR_MASK_SET( qp_attr_mask_p, QP_ATTR_QKEY);
qp_attr_p.qkey = QKEY;

// physical port
QP_ATTR_MASK_SET( qp_attr_mask_p, QP_ATTR_PORT);
qp_attr_p.port = 1;

ret = VAPI_modify_qp( hca_hndp_p,
                    qp_hndl_p,
                    &qp_attr_p,
                    &qp_attr_mask_p,
                    &qp_cap_p      );
IB_stat( ret, "VAPI_modify_qp() (RST->INIT)" );
```

Wie man sieht werden bei diesem Zustandswechsel Partition Key, Queue Key, sowie der vom QP benutzte Port festgelegt. Eine kurze Erläuterung zu den Keys findet sich in Abschnitt 2.3.3.

Beim Übergang von *init* zu *ready to receive* sind keine Änderungen an den Attributen des QPs nötig. Als letztes folgt nun noch der Übergang von *ready to receive* zu *ready to send*, wobei noch die Packet Sequence Number, also der Beginn der fortlaufenden Nummerierung der gesendeten Pakete, gesetzt wird. Jetzt kann das erstellte Queue Pair verwendet werden.

Funktion zum Erstellen der Send Requests

Da die UD QPs, wie in Abschnitt 4.1.1 beschrieben, nicht fest miteinander verbunden sind, ist es nötig, bei jedem Work Request für die Send Queue das Ziel mit anzugeben. Um diese Send Requests einfacher erstellen zu können, wurde die folgende Funktion hinzugefügt:

```
VAPI_sr_desc_t create_ud_sr ( int rank,
                             int remote,
                             VAPI_hca_hdl_t hca_hdl_p,
                             VAPI_pd_hdl_t pd_hdl_p,
                             VAPI_qp_hdl_t qp_hdl_p,
                             VAPI_ud_av_hdl_t *ud_av_hdl_p);
```

Die Parameter *rank* und *remote* bestimmen dabei, dass der erstellte Send Request eine Nachricht von Knoten *rank* an Knoten *remote* generieren wird. Außerdem werden der Funktion die Handles für den HCA, die Protection Domain, das Queue Pair und den zu erstellenden Unreliable Datagram Address Vector übergeben.

Nach Abfrage der eigenen QP Nummer und LID mit den entsprechenden VAPI Funktionen, tauschen die Prozesse *rank* und *remote* diese Informationen mittels MPI aus.

Nun können die beiden beteiligten Prozesse einen Send Request zum jeweils anderen erstellen:

```
//set remote QP Number
sr_desc_p.remote_qp = rem_qp_attr_p.qp_num;

//build address vector
ud_av_p.sl = 0;
ud_av_p.grh_flag = 0;
ud_av_p.dlid = rem_hca_port_p.lid;
ud_av_p.static_rate = 0;
ud_av_p.src_path_bits = 0;
ud_av_p.port = 1;

VAPI_create_addr_hdl( hca_hdl_p,
                    pd_hdl_p,
                    &ud_av_p,
                    ud_av_hdl_p);
IB_stat( ret, "VAPI_create_addr_hdl()" );

sr_desc_p.remote_ah = *ud_av_hdl_p;
```

Zuerst wird dem Send Request die QP Nummer des entfernten Prozesses zugewiesen. Danach werden die Werte für den UD Address Vector gesetzt. Da nur lokales Routing unterstützt wird (*grh_flag* nicht true), ist nur die DLID relevant. Außerdem wird noch der benutzte Port des HCAs festgelegt. Nachdem dies geschehen ist, kann das Address Handle erstellt und dem Send Request zugewiesen werden.

Änderungen im Hauptprogramm `mpi_iba_bench.c`

Nachdem die im folgenden benutzten Funktionen näher erläutert wurden, soll es nun um die Änderungen im Hauptprogramm gehen. Für eine genaue Übersicht über den ursprünglichen Ablauf siehe Abschnitt A.1.

Nach einigen Tests mit Unreliable Datagram wurde festgestellt, dass zusätzlich zum eigentlichen Inhalt der Nachricht, noch der GRH im Empfangsbuffer gespeichert werden muss. Dies ist leider nur sehr versteckt im IBA Standard [1] zu finden und im Gegensatz zur Definition darin verwerfen die verwendeten Mellanox IB Adapter bei Verwendung von UD QPs zu große Pakete, ohne ein CQE zu erstellen. Um UD zu ermöglichen, wurden in der Folge die Empfangsbuffer um 40 Bytes erweitert (`MEMSIZE + 40`).

Eine andere Änderung betrifft wie schon weiter oben beschrieben, die Erstellung der QPs:

```
#if TRANSPORT & (TRAN_RC | TRAN_UC)
    [QP Erstellung bei Verwendung von UC oder RC]
#else
if ( rank == 0){
    qp_h_arr = malloc(sizeof(VAPI_qp_hdl_t));
    if(qp_h_arr == 0) {
        printf("malloc() error");
        exit(1);
    }
    qp_h_arr_p = qp_h_arr;
    *qp_h_arr_p = create_dg_qp( rank,
                               hca_hndp_p,
                               sr_cq_hdl_p,
                               rr_cq_hdl_p,
                               pd_hndp_p );
}
}
#endif
```

Wie man sieht, wird die Funktion zum Erstellen der QPs sowohl von Prozess 0 als auch von allen anderen Prozessen nur einmal aufgerufen. Die Unterscheidung bei der Erstellung wird durch die verwendeten Compileranweisungen ermöglicht. Um die Änderungen im Benchmarkteil möglichst übersichtlich zu gestalten, wird bei Prozess 0 weiterhin die Variable für das bei RC und UC benutzte QP Handle Array verwendet.

Bei der nachfolgenden Registrierung der Speicherbereiche sind wiederum Änderungen an der Größe des Empfangsbuffers nötig und auch bei der Erstellung der Scatter Gather Liste für die Receive Requests muss darauf geachtet werden, dass 40 Bytes mehr zum Empfang der Nachricht nötig sind:

```
#if TRANSPORT & (TRAN_UD)
```

```

    rr_sg_lst->len = MEMSIZE+40;
#else
    rr_sg_lst->len = MEMSIZE;
#endif

```

Jetzt folgt die Erstellung der Send Requests. Hierbei wird einfach die oben beschriebene Funktion zusätzlich vor den schon vorhandenen Anweisungen zum Erstellen ausgeführt und der in `mpi_liba_bench.h` definierte Queue Key zugewiesen. Als Beispiel dazu, der Ablauf bei Prozess 0:

```

for(x = 0; x < (procs-1); x++) {
    rem_addr_p = rem_addr+x;
    r_key_p = r_key+x;

#ifdef TRANSPORT & (TRAN_UD)
    ud_av_hndl_p = ud_av_hndl_array + x;
    // create_ud_sr uses address vector to rank x+1
    sr_desc_p = create_ud_sr( rank, x+1,
                             hca_hndp_p,
                             pd_hndp_p,
                             *qp_h_arr_p,
                             ud_av_hndl_p);
    sr_desc_p.remote_qkey = QKEY;
#endif

#ifdef MODE & MODE_SEND
    PREPARE_SEND_LIST(VAPI_SEND, sr_desc_p, rem_addr_p, r_key_p);
#endif

[andere Modi]

    PREPARE_RECV_LIST(rr_desc_p);

    // copy receive descriptor in array structure ...
    rr_desc_array_p = rr_desc_array + x;
    memcpy(rr_desc_array_p, &rr_desc_p, sizeof(VAPI_rr_desc_t));
    sr_desc_array_p = sr_desc_array + x;
    memcpy(sr_desc_array_p, &sr_desc_p, sizeof(VAPI_sr_desc_t));
}

```

Die anderen Prozesse rufen die Funktion `create_ud_sr` jeweils nur einmal auf. Nachdem nun alle Vorbereitungen für das Senden einer Nachricht abgeschlossen sind, kommen wir zum Warmup Teil. Um zu vermeiden, dass bei vielen beteiligten Knoten das Limit für Work Requests in der Receive Queue überschritten wird, wurde der Ablauf etwas abgeändert. Probleme kann dies bei Unreliable Datagram deshalb bereiten, weil im Gegensatz zu Reliable und Unreliable Connection nur ein Queue Pair bei Prozess 0 verwendet wird, um Nachrichten an alle anderen Knoten zu senden und von allen zu empfangen. Um das Problem zu vermeiden, werden die Receive Requests jetzt nicht mehr vor der Schleife mit den Send Requests gepostet, sondern zu Beginn der Schleife. Außerdem waren auch einige Veränderungen beim Senden der Nachrichten

nötig, jedoch entsprechen diese den Änderungen im Benchmarkteil und werden im entsprechenden Abschnitt näher erläutert.

Nach dem Warmup folgt der Benchmarkteil in einer Schleife. Hier muss wieder auf die zusätzlichen 40 Bytes durch den GRH geachtet werden:

```
while (memsize != 0) {
    sr_sg_lst->len = memsize;
#if TRANSPORT & (TRAN_UD)
    rr_sg_lst->len = memsize+40;
#else
    rr_sg_lst->len = memsize;
#endif

    [Benchmarkteil]

    [Ausgabe der Ergebnisse]

    MEMSTEP(memsize); /* raise memstep or return 0 to stop */
} /* while memsize != 0*/
```

Nachdem alle Messungen durchgeführt wurden, werden nun alle Ressourcen wieder freigegeben. Dabei ist zu beachten, dass die neu hinzugefügten UD Address Handles auch explizit wieder freigegeben werden müssen:

```
#if TRANSPORT & (TRAN_UD)
    //destroy address handle
    if(rank == 0){
        for(x=0; x<(procs-1); x++){
            ud_av_hndl_p = ud_av_hndl_array + x;
            ret = VAPI_destroy_addr_hndl(hca_hndp_p, *ud_av_hndl_p);
            IB_stat(ret, "VAPI_destroy_addr_hndl");
        }
    }else{
        ret = VAPI_destroy_addr_hndl(hca_hndp_p, *ud_av_hndl_p);
        IB_stat(ret, "VAPI_destroy_addr_hndl");
    }
#endif
```

Anpassungen im Benchmarkteil scenario_2.sub.c

Nachdem alle Änderungen am Hauptprogramm und den darin benutzten Funktionen erläutert wurden, soll es nun um den Benchmarkteil gehen. Für ein Flussdiagramm zum ursprünglichen Ablauf siehe Anhang A.3.

Da bei Prozess 0 nur ein QP für das Senden und Empfangen von Nachrichten zuständig ist, muss kein QP Array beim Posten der Receive Requests durchlaufen werden. Da sich das UD QP Handle bei Prozess 0 als erster Eintrag im QP Array befindet, wird die Anweisung zum Durchlaufen des Arrays einfach per Compileranweisung entfernt. Somit werden *i* Receive Requests für ein QP abgesetzt. Für die anderen Prozesse ändert sich nichts. Da nur der Modus SEND zusammen mit UD QPs funktioniert, wird dieser Teil des Benchmarks bei Verwendung von UD immer durchlaufen:

```

#if MODE & MODE_SEND
    if(rank == 0) {
        for(j = 0; j < i; j++) {
#if TRANSPORT & (TRAN_RC | TRAN_UC)
            qp_h_arr_p = qp_h_arr + j;
#endif
            ret = VAPI_post_rr(hca_hndp_p, *qp_h_arr_p, &rr_desc_p);
            IB_stat(ret, "VAPI_post_rr()");
        }
    }
    if ((rank > 0) && (rank <= i)) { // only ranks < i post 1 rr
        ret = VAPI_post_rr(hca_hndp_p, qp_hndl_p, &rr_desc_p);
        IB_stat(ret, "VAPI_post_rr()");
    }

    // wait until all receive requests are posted
    MPI_Barrier(MPI_COMM_WORLD);
#endif

```

In der folgenden ersten Benchmarkphase sendet Prozess 0 nur von einem QP aus an i andere Prozesse. Dies wird analog zum Posten der Receive Requests realisiert. Es wird immer der erste Eintrag aus dem QP Array verwendet und das Array der Send Requests normal durchlaufen. Mehr Änderungen am Benchmarkteil waren nicht nötig.

4.2 Multicast

Die zweite Erweiterung des Benchmarks ist Multicast. Da Multicast den selben Queue Pair Typ verwendet wie Unreliable Datagram, ergeben sich keine Änderungen bei deren Erstellung. Der Aufbau der Send Requests ändert sich zudem nur bei Knoten 0. Durch die Verwendung von Multicast ergeben sich jedoch Änderungen beim Versenden der Nachrichten und auch im prinzipiellen Ablauf des Benchmarkteils. Zuerst folgt wieder ein Überblick über die Funktionsweise von Multicast und in Abschnitt 4.2.2 dann die Implementierung.

4.2.1 Funktionsweise

Multicast Nachrichten sind wie auch UD Nachrichten auf ein Paket mit maximaler Paketgröße beschränkt. Wenn von einem QP aus eine Multicast Nachricht gesendet wird, so legen DGID und DLID fest, an welche Multicastgruppe die Nachricht weitergeleitet wird. Kommt die Nachricht bei einem Switch (oder Router) an, so schaut dieser in einer internen Tabelle nach und entscheidet anhand der DLID (Switch) oder DGID (Router), an welche Ports die Nachricht weitergeschickt werden soll. Die Nachricht wird dabei nicht nur an ein Ziel verschickt, sondern an alle Ziele, die Mitglieder der Multicast Gruppe sind, wie in Abbildung 5 dargestellt ist.

Beim Eintreffen einer Multicast Nachricht an einem Channel Adapter, schaut dieser in einer internen Tabelle nach, ob eines oder mehrere QPs momentan Mitglied einer Multicastgruppe mit der entsprechenden DGID sind. Sollte dies der Fall sein, so wird die Nachricht an alle QPs weitergeleitet, die Mitglied der Gruppe sind.

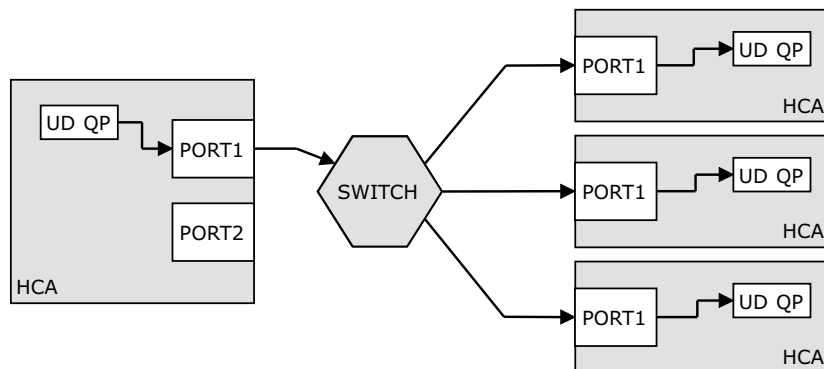


Abbildung 5: Kommunikation mittels Multicast

4.2.2 Implementierung

Wie schon erwähnt, verwendet Multicast UD QPs. Somit ändert sich bei der Erstellung der QPs im Vergleich zu UD nichts. Es bleiben auch alle Anpassungen bezüglich der Größe des Empfangsbuffers bestehen, da auch bei Multicast der GRH zusätzlich darin gespeichert werden muss.

Da Prozess 0 bei Verwendung von Multicast in Phase 1 des Benchmarks nur eine Nachricht verschicken muss, sind jedoch Änderungen bei der Erstellung der Send Requests nötig.

Funktion zum Erstellen der Send Requests für Multicast

Dadurch, dass die zweite Phase des Benchmarks im Vergleich zu UD unverändert bleibt, können alle Prozesse ausser Prozess 0 weiterhin die UD Funktion aus Abschnitt 4.1.2 verwenden. Prozess 0 hingegen verwendet eine neue Funktion:

```
VAPI_sr_desc_t create_mc_sr ( int rank,
                             int remote,
                             VAPI_hca_hdl_t hca_hdl_p,
                             VAPI_pd_hdl_t pd_hdl_p,
                             VAPI_qp_hdl_t qp_hdl_p,
                             VAPI_ud_av_hdl_t *ud_av_hdl_p );
```

Wie man sieht bleibt die Parameterliste unverändert. Um den anderen Prozessen die QP Nummer und LID von Prozess 0 zu senden, wird wieder MPI verwendet. Die QP Nummern und LIDs der entfernten Prozesse werden von der neuen Funktion nicht weiter verwendet. Stattdessen wird für Multicast eine feste QP Nummer und die erste Multicast LID verwendet. Beides ist in `mpi_iba_bench.h` definiert:

```
/* MC LID, QP number */
#ifdef MCLID
#define MCLID 0xC000
#endif
```

```

#ifndef MCQP
#define MCQP 0xFFFFF
#endif

```

Zusätzlich ist noch eine MC GID nötig. Da die Erstellung von eigenen Multicast Gruppen meines Wissens noch nicht möglich ist, wird die GID der *ALL CA Multicast Group* verwendet. Dies sorgt dafür, dass die Multicast Nachricht an alle CA des lokalen Subnets weitergeleitet wird, das Subnet dabei aber nicht verlässt. Definiert wird die GID am Anfang der Funktion zur Erstellung der MC Send Requests:

```

IB_gid_t mcgid={0xFF,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

```

Nun kommen wir zur eigentlichen Erstellung des Send Requests. Zuerst wird die MC QP Nummer als Ziel QP zugewiesen. Danach müssen die Parameter des UD AV festgelegt werden:

```

//set remote QP Number
sr_desc_p.remote_qp = MCQP;

//build address vector
ud_av_p.sl = 0;
ud_av_p.grh_flag = 1;
ud_av_p.dlid = MCLID;
ud_av_p.traffic_class = 0;
ud_av_p.flow_label = 0;
ud_av_p.hop_limit = 255;
ud_av_p.sgid_index = 0;
ud_av_p.dgid[0] = 0;
memcpy(ud_av_p.dgid, mcgid, sizeof(IB_gid_t));
ud_av_p.static_rate = 0;
ud_av_p.src_path_bits = 0;
ud_av_p.port = 1;

VAPI_create_addr_hdl( hca_hdl_p,
                    pd_hdl_p,
                    &ud_av_p,
                    ud_av_hdl_p);
IB_stat( ret, "VAPI_create_addr_hdl()" );

sr_desc_p.remote_ah = *ud_av_hdl_p;

```

Da eine Multicast GID verwendet werden muss, wird das *grh_flag* auf *true* gesetzt. Dies führt dazu, dass die DGID und einige andere Parameter ausgewertet werden. Nachdem alle Werte zugewiesen wurden, kann das Address Handle erstellt und dem Send Request zugewiesen werden.

Änderungen im Hauptprogramm

Wie bereits zu Beginn erwähnt, bleiben die Änderungen bezüglich der Größe des Empfangsbuffers bestehen und werden im folgenden nicht explizit aufgeführt.

Da die MC GID nicht nur zum Erstellen der Send Requests benötigt wird, sondern später auch zum Beitreten einer MC Gruppe, wird diese am Anfang des Hauptprogramms definiert. Nach Erstellung der QPs mit der in Abschnitt 4.1.2 definierten Funktion, werden die QPs zur durch die MC GID definierten Multicast Gruppe hinzugefügt:

```
//rank 1..n-1 attach QP to multicast group
#if TRANSPORT & TRAN_MC
    if (rank != 0)
        VAPI_attach_to_multicast( hca_hndp_p,
                                  mcgid,
                                  qp_hndl_p,
                                  0          );
#endif
```

Da man auch von einem UD QP, das nicht der MC Gruppe angehört, eine MC Nachricht versenden kann, muss das von Prozess 0 verwendete QP nicht zur Multicast Gruppe hinzugefügt werden.

Nachdem die Speicherbereiche registriert wurden, kommen wir nun zur Erstellung der Send Requests. Hier wird bei Prozess 0 die neu definierte Funktion `create_mc_sr()` anstatt `create_ud_sr()` aufgerufen. Um größere Änderungen im Programm zu vermeiden, erstellt Prozess 0 für jeden entfernten Prozess einen MC Send Request. Verwendet wird später nur der erste Send Request im erstellten Array.

Jetzt kommen wir zur Warmup Phase. Hier muss bei Prozess 0 beachtet werden, dass nur noch ein Send Request anstatt `procs-1` Send Requests nötig ist, um eine Nachricht an alle anderen Prozesse zu schicken. Die gleiche Änderung ist auch im Benchmarkteil nötig und wird dort näher erläutert.

Beim Einbinden des Benchmarkteils und der Ausgabe der Ergebnisse sind keine Änderungen nötig, jedoch muss bei der folgenden Freigabe der Ressourcen darauf geachtet werden, die QPs wieder aus der Multicast Gruppe zu entfernen:

```
#if TRANSPORT & TRAN_MC
    if (rank != 0){
        ret = VAPI_detach_from_multicast( hca_hndp_p,
                                          mcgid,
                                          qp_hndl_p,
                                          MCLID      );
        IB_stat(ret, "VAPI_detach_from_multicast");
    }
#endif
```

Anpassungen im Benchmarkteil `scenario_2.sub.c`

Wie auch bei Verwendung von UD QPs postet Prozess 0 zu Beginn des Benchmarks wieder `i` Receive Requests für das benutzte QP. Die anderen Prozesse posten jeweils einen Receive Request. Die einzige Änderung im Vergleich zu UD betrifft das Versenden von Nachrichten in Phase 1. So muss Prozess 0 nicht `i` Nachrichten versenden, sondern nur eine einzige:

```

if(0 == rank) {
    // post send requests to all peers!
#ifdef TRANSPORT & TRAN_MC /*just 1 SR for Multicast*/
#ifdef SEND & SEND_INLINE
    ret = EVAPI_post_inline_sr( hca_hndp_p,
                               *qp_h_arr_p,
                               sr_desc_array );
#else
    ret = VAPI_post_sr( hca_hndp_p,
                      *qp_h_arr_p,
                      sr_desc_array );
#endif
#endif
    IB_stat (ret, "VAPI_post_sr()");

#ifdef TRANSPORT & TRAN_MC /*
for (j = 0; j < i; j++) {
    sr_desc_array_p = sr_desc_array + j;
    [POST SR for UD, RC/UC]
}
#endif

```

Im Gegensatz zu UD sowie RC/UC wird bei Multicast nur der erste Eintrag aus dem Send Request Array benutzt.

Der Rest des Benchmarkteils läuft wie bei Verwendung von Unreliable Datagram ab, so dass keine weiteren Änderungen nötig waren.

4.3 RDMAR

Als letzte Erweiterung wurde noch RDMAR hinzugefügt. Da RDMAR von Reliable Connection unterstützt wird, ergeben sich keine Änderungen bei der Erstellung der Queue Pairs. Außerdem sind auch nur geringe Änderungen bei den Send Requests nötig, da RDMAR schon implementiert ist. Im Benchmarkteil wurde eine zusätzliche Synchronisation mittels Multicast hinzugefügt und im Hauptprogramm die dafür nötigen Queue Pairs und Work Requests.

In Abschnitt 2.4.3 findet sich ein Überblick über die Funktionsweise von RDMAR. Im folgenden wird daher nur auf die Implementierung eingegangen.

4.3.1 Implementierung

Ein Problem bei der Implementierung von RDMAR war, dass der Prozess, von dem gelesen werden soll, nicht weiß, wann die Operation abgeschlossen ist. Dadurch wurde es nötig, eine zusätzliche Synchronisation durch Benutzung von Multicast einzubauen. Welche Änderungen dies im Hauptprogramm nötig machte, ist im folgenden Abschnitt näher beschrieben.

Änderungen im Hauptprogramm

Als erstes erstellt jeder Prozess ein zusätzliches Queue Pair:

```

#ifdef MODE & MODE_RDMAR
    sync_qp_hdl_p = create_dg_qp( rank,

```

```

                                hca_hndp_p,
                                sr_cq_hndl_p,
                                rr_cq_hndl_p,
                                pd_hndp_p    );
if (rank != 0)
    VAPI_attach_to_multicast( hca_hndp_p,
                              mcgid,
                              sync_qp_hndl_p,
                              0          );
#endif

```

Alle Prozesse ausser Prozess 0 sorgen dann dafür, dass das Queue Pair der durch die *mcgid* festgelegten Multicast Gruppe zugeordnet wird. Nach der Erstellung der Queue Pairs müssen nun noch zusätzliche Work Requests erstellt werden, die folgende Scatter Gather Listen verwenden:

```

sr_sg_lst2 = malloc (sizeof(VAPI_sg_lst_entry_t));
sr_sg_lst2->addr = s_ptr;
sr_sg_lst2->len = 1;
sr_sg_lst2->lkey = s_rep_mrwr.p.l_key;

rr_sg_lst2 = malloc(sizeof(VAPI_sg_lst_entry_t));
rr_sg_lst2->addr = r_ptr;
rr_sg_lst2->lkey = r_rep_mrwr.p.l_key;
rr_sg_lst2->len = 41;

```

Wie man sieht wird immer nur das 1. Byte aus dem Send Speicherbereich an die entfernten Prozesse geschickt. Bei den Receive Requests ist wieder auf den zusätzlich benötigten Platz für den GRH zu achten.

Jeder Prozess erstellt nun jeweils einen Work Request für die RDMAR Operation. Prozess 0 erstellt zusätzlich den Send Request für Multicast und die anderen Prozesse erstellen den Send Request für Unreliable Datagram (momentan nicht im Benchmarkteil benutzt). Außerdem wird von jedem Prozess ein zusätzlicher Receive Request erstellt. Als Beispiel folgt der Ablauf bei Prozess 0:

```

for(x = 0; x < (procs-1); x++) {
    rem_addr_p = rem_addr+x;
    r_key_p = r_key+x;
    ud_av_hndl_p = ud_av_hndl_array + x;

#if MODE & MODE_RDMAR
    PREPARE_SEND_LIST( VAPI_RDMA_READ,
                      sr_desc_p,
                      rem_addr_p,
                      r_key_p          );
    // send requests for sync
    sr_desc2_p = create_mc_sr( rank,
                              x+1,
                              hca_hndp_p,
                              pd_hndp_p,

```

```

                                sync_qp_hndl_p,
                                ud_av_hndl_p    );
sr_desc2_p.remote_qkey = QKEY;
PREPARE_SEND_LIST( VAPI_SEND,
                  sr_desc2_p,
                  rem_addr_p,
                  r_key_p    );
sr_desc2_p.sg_lst_p = sr_sg_lst2;
PREPARE_RECV_LIST(rr_desc2_p);
rr_desc2_p.sg_lst_p = rr_sg_lst2;
#endif
PREPARE_RECV_LIST(rr_desc_p);

[andere Modi]
[kopieren der Work Requests in Arrays]
}

```

Hier ist zu beachten, dass die angegebene Adresse bei RDMA, die Quelladresse für die zu lesende Nachricht ist und nicht wie bei RDMAW die Zieladresse für die zu schreibende Nachricht.

Für die Erstellung des Send Requests für Multicast wird wieder die Funktion `create_mc_sr()` benutzt. Außerdem wird nach dem Aufruf der Makros `PREPARE_SEND_LIST` und `PREPARE_RECV_LIST` für Multicast beziehungsweise Unreliable Datagram noch die neue Scatter Gather Liste gesetzt.

Die umfangreichsten Änderungen waren im Benchmarkteil nötig, weshalb ein neues Szenario erstellt wurde. Dies wird im Hauptprogramm anstelle des alten Szenarios eingebunden.

Neuer Benchmarkteil `scenario_2_rdma_mc.sub.c`

Da Multicast zur Synchronisation verwendet wird, müssen alle Prozesse außer Prozess 0 zu Beginn eines Benchmarkdurchlaufs Receive Requests für das Synchronisations-QP absetzen:

```

if (rank != 0) {
    ret = VAPI_post_rr( hca_hndp_p,
                      sync_qp_hndl_p,
                      &rr_desc2_p    );
    IB_stat( ret, "VAPI_post_rr()" );
}

```

Nachdem mittels `MPI_Barrier` sichergestellt wurde, dass alle Receive Requests abgesetzt wurden, fängt Prozess 0 an, von allen i entfernten Prozessen eine Nachricht der Größe `memsize` zu lesen:

```

if(0 == rank) {

#if MEASURE & MEA_RTT_TIME
    start_hr_timer(timer1);
#endif
    for (j = 0; j < i; j++) {

```

```

        qp_h_arr_p = qp_h_arr + j;
        sr_desc_array_p = sr_desc_array + j;
        sr_sg_lst->addr = (int) (s_addr + (j * MEMSIZE));

    #if SEND & SEND_INLINE
        ret = EVAPI_post_inline_sr( hca_hndp_p,
                                   *qp_h_arr_p,
                                   sr_desc_array_p );
    #else
        ret = VAPI_post_sr( hca_hndp_p,
                            *qp_h_arr_p,
                            sr_desc_array_p );
    #endif
    IB_stat( ret, "VAPI_post_sr()" );
}

if(0 == rank) {
    for (j = 0; j < i; j++)
        VAPI_POLL_CQ( hca_hndp_p,
                      sr_cq_hndl_p,
                      comp_desc_p,
                      cnt
                      );
    #if MEASURE & MEA_RTT_TIME
        stop_hr_timer(timer1);
    #endif
}

```

Wie man sieht, misst Prozess 0 nicht mehr die für den gesamten Benchmarkteil benötigte Zeit, sondern nur noch die Zeit, die er braucht, um von *i* Prozessen zu lesen. Außerdem werden die von den entfernten Prozessen gelesenen Nachrichten jeweils an eine eigene Speicheradresse für den jeweiligen Prozess geschrieben.

Prozess 0 wartet, bis alle Nachrichten gelesen wurden, und setzt dann den im Hauptprogramm vorbereiteten Send Request für das zur Synchronisation benutzte QP ab. Die anderen Prozesse warten darauf, dass durch ein CQE signalisiert wird, dass die Nachricht von Prozess 0 angekommen ist:

```

// use MC to sync with peers
if(0 == rank) {
    ret = VAPI_post_sr( hca_hndp_p,
                       sync_qp_hndl_p,
                       &sr_desc2_p
                       );
    IB_stat ( ret, "VAPI_post_sr()" );

    VAPI_POLL_CQ( hca_hndp_p,
                  sr_cq_hndl_p,
                  comp_desc_p,
                  cnt
                  );
}

```

```

// WAIT for MC from rank 0
else {
    VAPI_POLL_CQ( hca_hndp_p,
                  rr_cq_hndl_p,
                  comp_desc_p,
                  cnt
                  );
}

```

Danach beginnen die Prozesse 1 bis i eine Nachricht von Prozess 0 zu lesen:

```

if((rank > 0) && (rank <= i)) {

#if MEASURE & MEA_RTT_TIME //measure time for RDMAR from rank 0
    start_hr_timer(timer1);
#endif

#if SEND & SEND_INLINE
    ret = EVAPI_post_inline_sr( hca_hndp_p,
                               qp_hndl_p,
                               &sr_desc_p );
#else
    ret = VAPI_post_sr( hca_hndp_p,
                       qp_hndl_p,
                       &sr_desc_p );
#endif
    IB_stat( ret, "VAPI_post_sr()" );

    VAPI_POLL_CQ( hca_hndp_p,
                  sr_cq_hndl_p,
                  comp_desc_p,
                  cnt
                  );

#if MEASURE & MEA_RTT_TIME
    stop_hr_timer(timer1);
#endif
}

```

Jeder beteiligte Prozess misst dabei die Zeit, die er für das Lesen einer Nachricht von Prozess 0 benötigt. Um eine Auswertung dieser Zeiten möglich zu machen, müssen sie noch von Prozess 0 eingesammelt werden:

```

#if MEASURE & MEA_RTT_TIME
    timesend = 0.0;
    if((rank > 0) && (rank <= i)){
        timesend = microseconds_hr_timer(timer1);
    }

    MPI_Reduce( &timesend,
                &timerecv,
                1,
                MPI_DOUBLE,

```

```

        MPI_MAX,
        0,
        MPI_COMM_WORLD );

    if(rank == 0){
#ifdef RDMARTIME & RDMART_RTT
        timerecv = (microseconds_hr_timer(timer1) + timerecv) / i;
#endif
#ifdef RDMARTIME & RDMART_1N
        timerecv = microseconds_hr_timer(timer1) / i;
#endif
#ifdef RDMARTIME & RDMART_N1
        timerecv = timerecv / i;
#endif
        array[i] += timerecv;
        if(timerecv < minarray[i]) {
            minarray[i] = timerecv;
        }
    }
#ifdef RDMARTIME & RDMART_RTT
#endif

```

Prozess 0 erhält durch MPIReduce das Maximum der von den entfernten Prozessen gemessenen Zeit. Je nach Einstellung in `mpi_iba_bench.h` wird danach die gewünschte Zeit in `array` und `minarray` eingetragen und später ausgegeben. Mögliche Zeiten sind dabei:

RDMART_1N gibt die von Prozess 0 gemessene Zeit aus, die benötigt wird, um eine Nachricht von `emphi` entfernten Knoten zu lesen.

RDMART_N1 gibt die maximale von Prozess 1 bis `i` gemessene Zeit aus, die benötigt wird, um eine Nachricht von Knoten 0 zu lesen.

RDMART_RTT gibt die Summe von `RDMAR_1N` und `RDMAR_N1` aus.

Die Zeiten werden dabei immer durch `i` geteilt, geben also die durchschnittliche Zeit für eine einzige RDMAR Operation an.

5 Benchmark Ergebnisse und Auswertung

5.1 Messumgebung und gemessene Zeiten

Die Messungen wurden auf einem Cluster mit 64 Knoten durchgeführt. Die Knoten waren dabei mit 3GHz Xeon Prozessoren und MTPB 23108 HCAs ausgestattet und durch Mellanox MTS 9600 Switches verbunden.

Die folgenden Ergebnisse entsprechen der Round Trip Time (RTT). Diese setzt sich dabei wie folgt zusammen:

$$RTT = \frac{\text{Von Prozess 0 gemessene Zeit}}{\text{Anzahl der entfernten Prozesse}}$$

Prozess 0 misst dabei die Zeit, die benötigt wird, um eine Nachricht an alle beteiligten entfernten Prozesse zu senden und von diesen danach eine Nachricht zu erhalten.

Eine Besonderheit stellen dabei die Messungen für RDMA dar. Wie in Abschnitt 4.3.1 beschrieben, wird dabei von Prozess 0 nur die Zeit gemessen, die benötigt wird, um von den entfernten Prozessen eine Nachricht zu lesen. Die entfernten Prozesse messen wiederum nur die Zeit, die sie benötigen, um eine Nachricht von Prozess 0 zu lesen. Der gesamte gemessene Wert ist dabei die Summe aus der von Prozess 0 gemessenen Zeit und dem Maximum der von den entfernten Prozessen gemessenen Zeit. Aufgrund dieser Unterschiede werden die Messungen von RDMA gesondert aufgeführt.

5.2 Ergebnisse für SEND, RDMA

5.2.1 Ergebnisse für kleine Nachrichten

In Abbildung 6 auf Seite 32 sind die Ergebnisse für Nachrichten mit einer Größe von einem Byte aufgeführt. Das obere der beiden Diagramme zeigt dabei die minimale RTT für bis zu 15 entfernte Prozesse, das untere für bis zu 63.

Wie man sieht, ist RDMA, sowohl bei Verwendung von Reliable Connection als auch bei Unreliable Connection, für bis zu acht entfernte Knoten am schnellsten. Die Werte für SEND mit Reliable und Unreliable Connection liegen etwas darüber. Die Ergebnisse für SEND mit Unreliable Datagram sind bei einem entfernten Prozess noch nahezu identisch mit UC/RC SEND und liegen dann bis zum Wert bei sieben Knoten bis zu $1,5 \mu s$ darüber. SEND mittels Multicast ist bei wenigen entfernten Prozessen am langsamsten. So liegt die minimale RTT für einen entfernten Knoten bei $47 \mu s$ während bei Verwendung von RDMA mit Reliable Connection ein Wert von $20 \mu s$ erzielt wird und SEND mit Unreliable Datagram eine RTT von $22,5 \mu s$ liefert. Je mehr Knoten jedoch beteiligt sind, desto effektiver wird SEND mit Multicast.

So liegen bei neun entfernten Knoten die RTTs für alle verwendeten Übertragungsmodi und Verbindungsarten nahezu gleichauf zwischen $5 \mu s$ und $6 \mu s$, wobei die RTT für RDMA mit Unreliable und Reliable Connection in diesem Bereich schon ein Minimum erreicht hat und danach wieder ansteigt. Die Werte für SEND mit Unreliable und Reliable Connection bleiben bis zum Wert bei 12 Knoten relativ konstant und erreichen dort ihr Minimum mit rund $5,1 \mu s$ für Unreliable Connection und rund $5,6 \mu s$ für Reliable Connection.

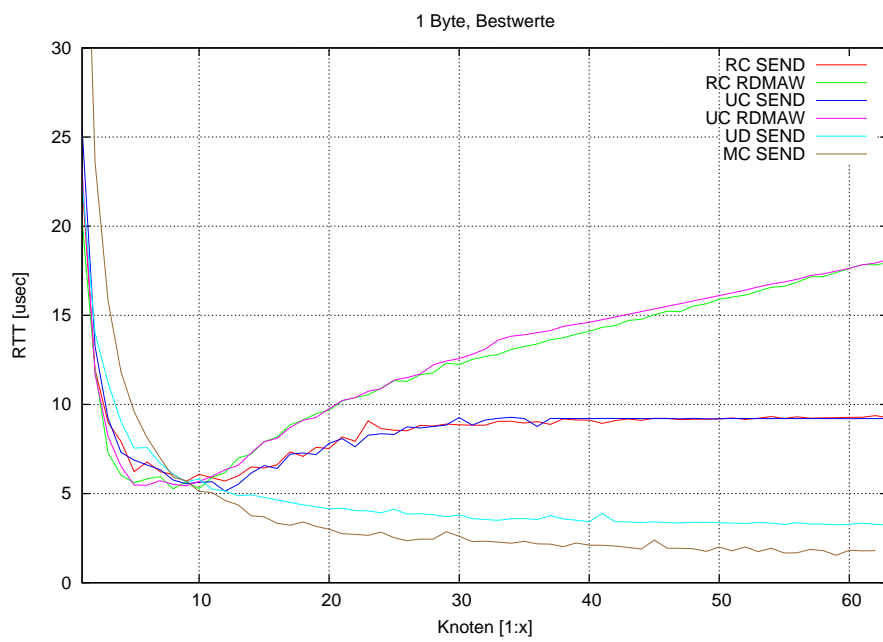
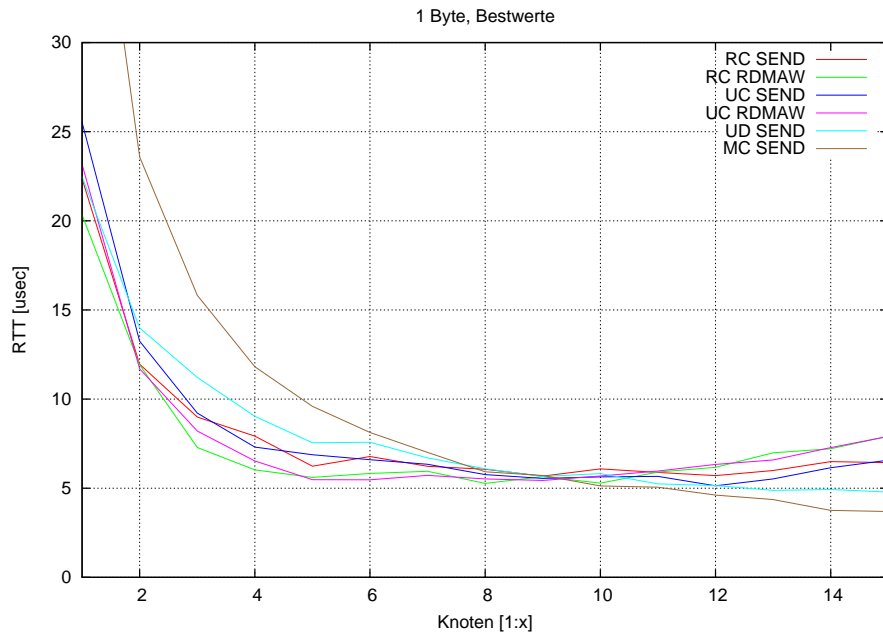


Abbildung 6: Bestwerte für Nachrichten mit einer Größe von 1 Byte

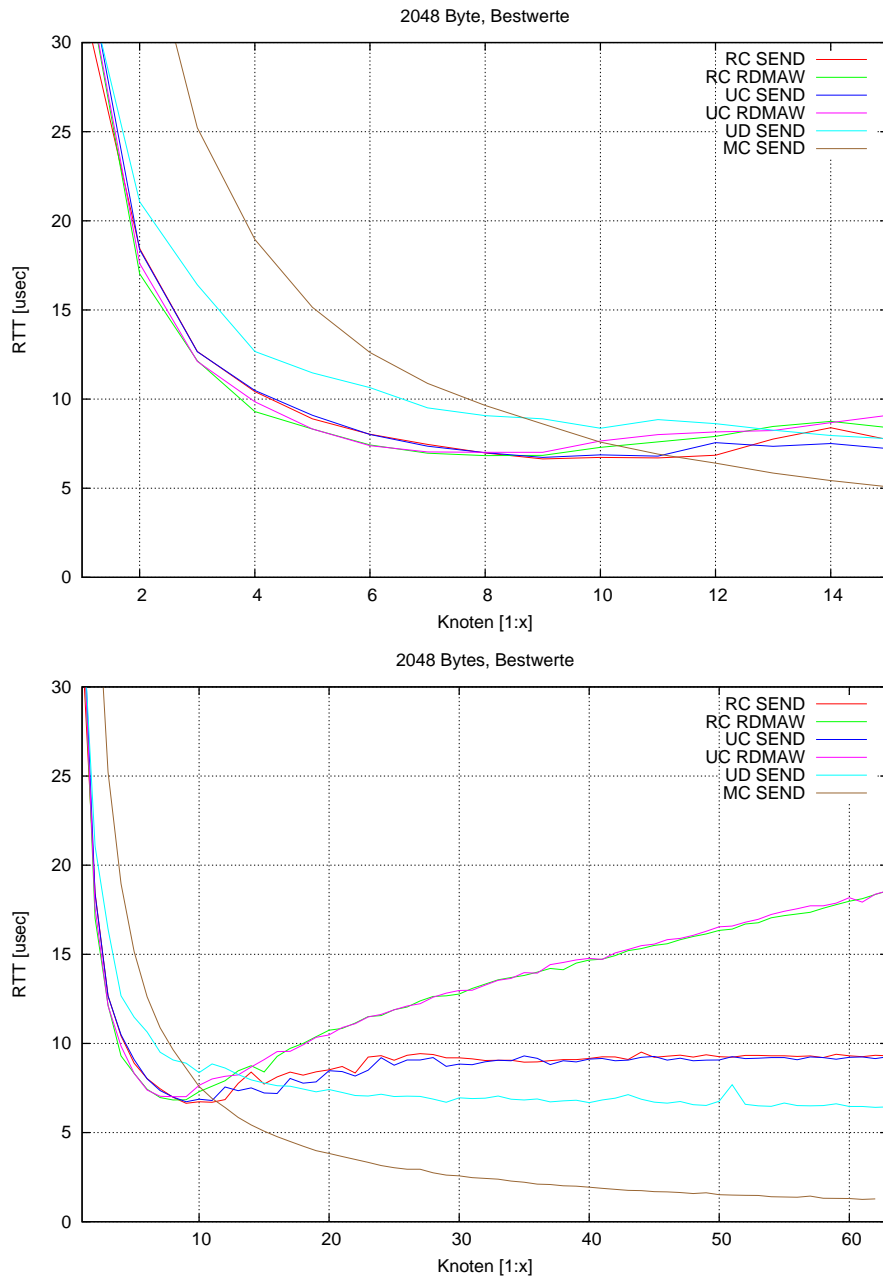


Abbildung 7: Bestwerte für Nachrichten mit einer Größe von 2048 Bytes

Die RTT für SEND mit Unreliable Datagram und Multicast fällt hingegen weiter, wobei Multicast bei zehn Knoten mit $5,1 \mu s$ schon schneller ist als Unreliable Datagram mit $5,8 \mu s$.

Die Werte für SEND mit Multicast und Unreliable Datagram fallen auch für hohe Knotenzahlen weiter, jedoch mit steigender Knotenzahl immer langsamer. So erreicht Multicast SEND bei 62 entfernten Knoten eine RTT von $1,8 \mu s$ und Unreliable Datagram SEND $3,3 \mu s$. Die Ergebnisse für SEND mit Reliable und Unreliable Connection steigen bis zum Wert bei ca. 30 Knoten an und bleiben danach konstant bei ca. $9,2 \mu s$. Bei Verwendung von RDMAW mit Unreliable und Reliable Connection steigen die Werte von 9 bis ca. 30 Knoten um rund $0,33 \mu s$ pro Knoten an. Von 31 bis 63 Knoten steigt die RTT danach etwas flacher um rund $0,17 \mu s$ pro Knoten weiter.

Nachdem ein Überblick über die Zeiten bei Nachrichten mit einer Größe von einem Byte gegeben wurde, soll es nun um Abbildung 7 auf Seite 33 gehen. Diese enthält wieder zwei Diagramme, welche die RTT für 2048 Bytes große Nachrichten zeigen.

Wie man sieht, ergibt sich ein ähnliches Bild, wie bei einer Nachrichtengröße von einem Byte. SEND mit Multicast ist für wenige Prozesse wieder am langsamsten. Danach folgt mit einigem Abstand SEND mit Unreliable Datagram. RDMAW und SEND mit Reliable und Unreliable Connection sind am schnellsten und liegen dicht zusammen. Die RTTs für RDMAW und SEND mit Reliable und Unreliable Connection erreichen wieder im gleichen Bereich wie bei einer Nachrichtengröße von einem Byte ihr Minimum. Die Werte liegen jedoch um einiges höher als bei Nachrichten mit einer Größe von einem Byte.

Je mehr Knoten beteiligt sind, desto schneller wird auch SEND mit Multicast wieder. Es ist jedoch zu beobachten, dass die RTT für SEND mit Multicast bei 9 Knoten nur niedriger als die RTT für SEND mit Unreliable Datagram ist. Bei 11 Knoten liegt Multicast SEND dann gleichauf mit Reliable und Unreliable Connection SEND, sowie unter RDMAW mit Reliable und Unreliable Connection. Ab 12 entfernten Knoten hat SEND mit Multicast dann wieder die niedrigste RTT.

Auch SEND mit Unreliable Datagram wird mit zunehmender Knotenzahl schneller, jedoch fällt die RTT weitaus langsamer als bei einer Nachrichtengröße von einem Byte. So ist Unreliable Datagram SEND erst bei 14 entfernten Knoten schneller als RDMAW mit Reliable und Unreliable Connection und SEND mit Unreliable und Reliable Connection wird erst bei 18 entfernten Knoten überholt. Bei hoher Knotenanzahl ist der Abstand von Unreliable Datagram SEND zu Multicast SEND auch deutlich größer. War der Abstand zwischen beiden bei 62 entfernten Knoten und einer Nachrichtengröße von einem Byte noch $1,5 \mu s$, so ist er bei einer Nachrichtengröße von 2048 Bytes mit $5,1 \mu s$ schon deutlich größer. Die RTT von Multicast liegt dabei mit $1,3 \mu s$ deutlich unterhalb der RTT von Unreliable Datagram mit $6,4 \mu s$.

5.2.2 Ergebnisse für große Nachrichten

Da SEND mit Multicast und Unreliable Datagram auf eine Nachrichtengröße, die die zulässige Paketgröße nicht überschreitet, limitiert ist, enthält Abbildung 8 auf Seite 35 nur die RTTs für die anderen Übertragungsmodi und Verbindungsarten.

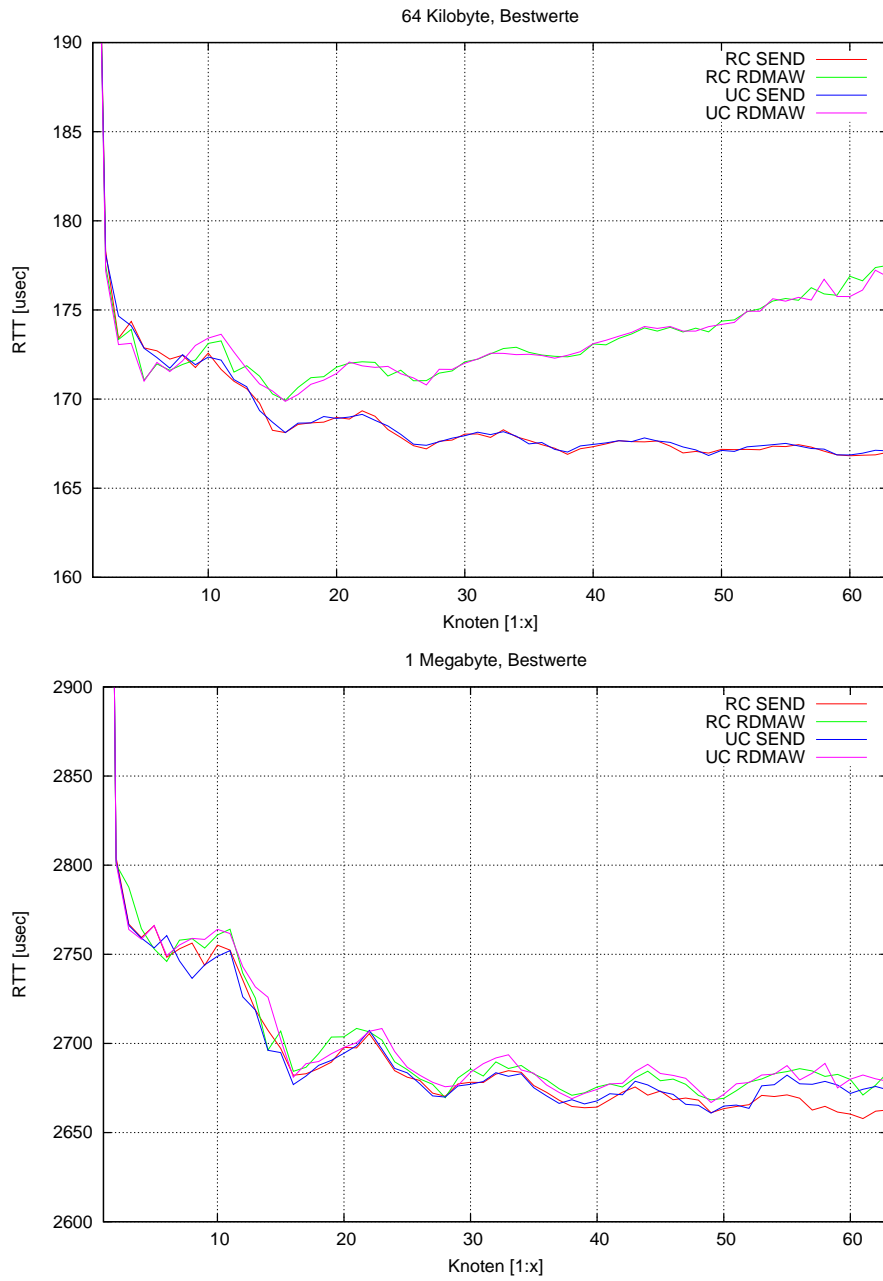


Abbildung 8: Bestwerte für Nachrichten mit einer Größe von 64 Kilobytes und einem Megabyte

Das obere Diagramm zeigt dabei die Werte für eine Nachrichtengröße von 64 Kilobyte und das untere die Ergebnisse für Nachrichten mit einer Größe von einem Megabyte.

Da die Werte von Unreliable und Reliable Connection sowohl bei SEND als auch bei RDMAW nahezu identisch sind, wird für die Auswertung von Abbildung 8 keine Unterscheidung von Reliable Connection und Unreliable Connection vorgenommen.

Bei einer Nachrichtengröße von 64 Kilobyte liegen RDMAW und SEND bei wenigen entfernten Knoten wieder dicht beisammen. So ist die RTT für SEND bei 2 entfernten Knoten mit $178 \mu s$ rund $1 \mu s$ höher als die für RDMAW. RDMAW bleibt bei bis zu 8 entfernten Knoten minimal schneller. Danach ist die RTT von SEND niedriger. Die Werte für RDMAW fallen dabei bis zu einer RTT von $170 \mu s$ bei 16 entfernten Knoten noch etwas, um danach auf bis zu $177 \mu s$ bei 63 entfernten Knoten anzusteigen. Die RTT von SEND fällt bis zum Wert bei 16 entfernten Knoten etwas stärker auf $168 \mu s$ und bleibt danach mit Schwankungen zwischen $166,8 \mu s$ und $168 \mu s$ konstant.

Wird die Nachrichtengröße nun noch weiter auf 1 Megabyte erhöht, lässt sich kaum noch ein Unterschied zwischen RDMAW und SEND feststellen. So fallen sowohl die Werte für RDMAW als auch die Werte für SEND bis 16 Knoten wieder etwas ab. So beträgt die RTT bei 2 Knoten rund $2800 \mu s$ und bei 16 Knoten rund $2680 \mu s$. Bei weiter steigenden Knotenzahlen bleibt die RTT mit Schwankungen relativ konstant im Bereich von $2660 \mu s$ bis $2690 \mu s$.

5.3 Ergebnisse für RDMAR

Nachdem die Ergebnisse für SEND und RDMAW erläutert wurden, soll es nun um RDMAR mit Reliable Connection gehen. Wie in Abschnitt 5.1 beschrieben, lassen sich die erhaltenen Werte nicht direkt miteinander vergleichen. Am Ende von Abschnitt 4.3.1 findet sich eine Erläuterung zu den verschiedenen Zeiten. Zu Beachten ist außerdem, dass die verschiedenen Zeiten nicht in einem einzigen Durchlauf gemessen wurden.

Abbildung 9 auf Seite 37 zeigt die Bestwerte für Nachrichten der Größe ein Byte und 2048 Bytes. Wie man sieht, beträgt die RTT bei einem entfernten Prozess und einer Nachrichtengröße von einem Byte $42 \mu s$ und fällt dann bis zum Wert bei 16 entfernten Knoten auf ein Minimum von $11,2 \mu s$. Danach bleibt die RTT mit kleinen Schwankungen konstant. Den Hauptanteil der RTT bildet dabei die 1:n read time, also die Zeit, die Prozess 0 benötigt, um eine Nachricht von n entfernten Knoten zu lesen. Diese liegt ab 5 Knoten relativ konstant bei $8,5 \mu s$. Die n:1 read time fällt hingegen auf rund $3 \mu s$ bei 9 entfernten Knoten und bleibt danach ebenfalls konstant.

Die Werte für eine Nachrichtengröße von 2048 Bytes verhalten sich ähnlich wie bei Nachrichten der Größe ein Byte. So erreicht die RTT bei 24 entfernten Knoten ein Minimum von rund $12 \mu s$ und bleibt danach mit einigen Schwankungen konstant. Der Anteil der 1:n read time ist ab 20 Knoten dabei mit rund $8,7 \mu s$ wieder deutlich größer als der Anteil der n:1 read time mit rund $2,5 \mu s$.

Abbildung 10 auf Seite 38 zeigt die Bestwerte für Nachrichten der Größe 32 Kilobytes. Hier ergibt sich ein anderer Verlauf der Zeiten als bei den kleineren Nachrichten. So fällt der Wert für die RTT zuerst zwar wieder bis auf rund $71 \mu s$, steigt danach jedoch wieder an. Ein Effekt der auch bei RDMAW zu beobachten war.

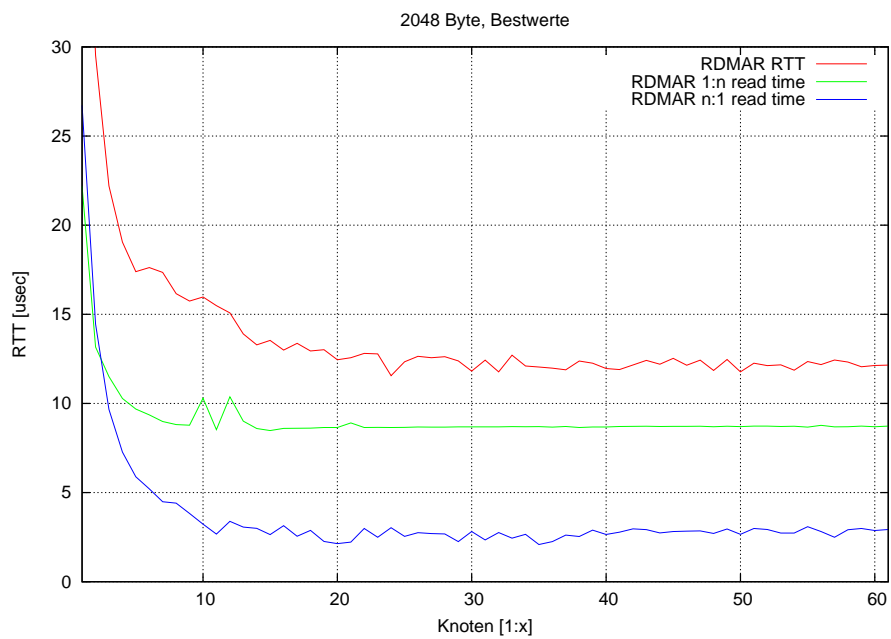
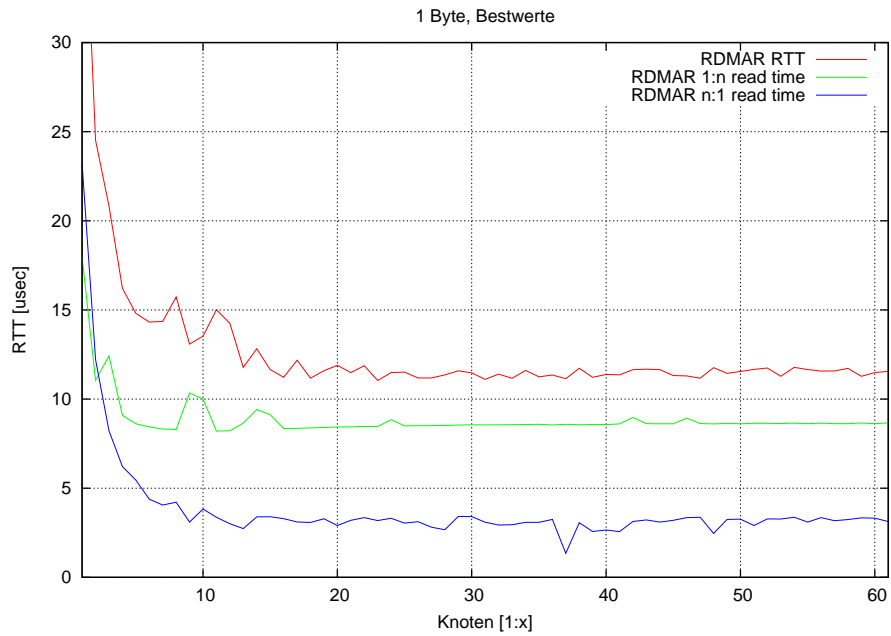


Abbildung 9: Bestwerte für RDMA mit Nachrichten der Größe 1 Byte und 2048 Byte

Weiterhin liegt die n:1 read time zwar bei weniger als 10 entfernten Knoten wieder unter dem Wert der 1:n read time, steigt danach jedoch auf einen Wert von $47 \mu s$ bei 20 entfernten Knoten und bleibt dann konstant. Die 1:n read time beträgt ab 20 Knoten rund $20,5 \mu s$.

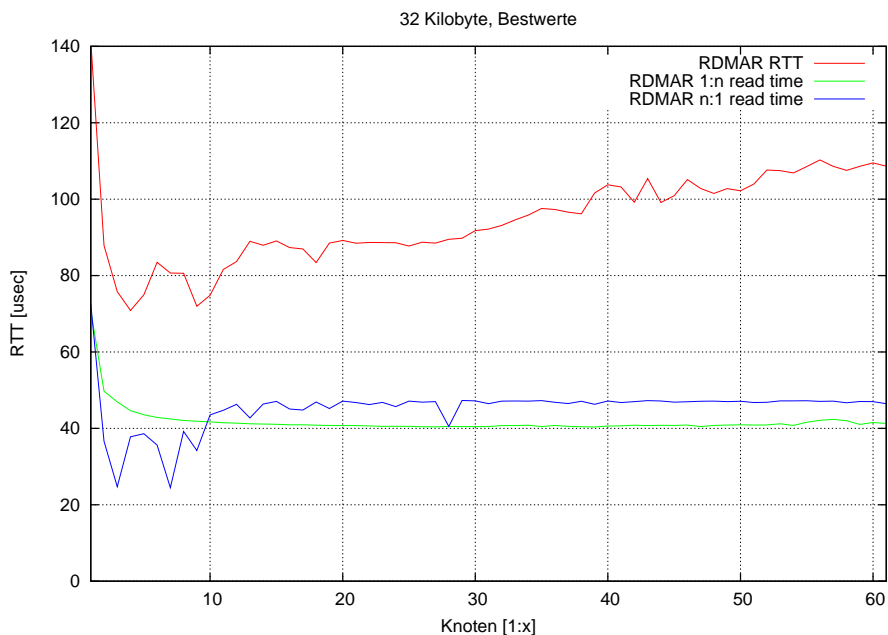


Abbildung 10: Bestwerte für RDMAR mit Nachrichten der Größe 32 Kilobyte

5.4 Auswertung

Im folgenden sollen die Benchmarkergebnisse der verschiedenen Übertragungsmodi und Verbindungsarten näher miteinander verglichen werden. Außerdem werden mögliche Erklärungen für den Verlauf der RTT erläutert.

5.4.1 SEND mit Reliable und Unreliable Connection

Bei SEND liegen Reliable und Unreliable Connection unerwarteterweise sehr dicht beisammen. So würde man zumindest bei Nachrichten mit einer Größe von einem Byte davon ausgehen, dass die bei Unreliable Connection fehlende Bestätigungsmeldung eines erhaltenen Pakets für eine niedrigere RTT sorgt. Eine mögliche Erklärung dafür ist der Ablauf des Benchmarks. So sendet Prozess 0 zuerst an alle entfernten Prozesse eine Nachricht und wartet bei Verwendung von Reliable Connection, bis der Empfang bestätigt wurde. Die entfernten Prozesse senden jedoch auch sofort nach Erhalt der Nachricht von Prozess 0 ihrerseits eine Nachricht zurück und Prozess 0 stoppt die Zeitmessung, sobald er eine Nachricht von allen entfernten Prozessen erhalten hat. Der zusätzliche Overhead der Bestätigungsmeldung wird somit von anderen Operationen verdeckt.

Eine weitere Beobachtung bei SEND mit kleinen Nachrichten ist die zuerst fallende RTT. So ist die RTT bei einem entfernten Knoten fast doppelt so hoch, wie bei zwei entfernten Knoten. Die RTT fällt dabei bei kleinen Nachrichten weiter bis zu einer von der Nachrichtengröße abhängigen Knotenzahl, steigt danach wieder an und bleibt nach diesem Anstieg nahezu konstant. Eine Erklärung für die sinkende RTT liefert das LogP Modell [5]. So überlagert sich, vereinfacht gesehen, die Latenz des Netzwerks mit dem Overhead zum Senden und Empfangen einer Nachricht. Während Prozess 0 also noch Nachrichten versendet, sind die ersten Nachrichten schon auf dem Weg zu den entfernten Prozessen, welche sofort nach Erhalt der Nachricht ihrerseits eine Nachricht zurücksenden. Die dadurch überlagerten Overheads und Latenzen führen zu einer geringeren RTT. Der zwischenzeitliche Anstieg der RTT bis zu einem konstanten Wert liegt möglicherweise an den zusätzlichen QPs und deren interner Verwaltung im HCA. So bedeutet jeder zusätzlicher Knoten ein QP mehr. Um herauszufinden, ob dies oder ein anderer Grund für den Anstieg sorgt, müsste man sich näher mit der QP Verwaltung im HCA auseinandersetzen.

5.4.2 RDMAW mit Reliable und Unreliable Connection

Wie bei SEND liegen die RTTs für Reliable und Unreliable Connection auch mit RDMAW bei allen Nachrichtengrößen sehr dicht beisammen und ähnlich wie bei SEND fällt die RTT zu Anfang bei steigenden Knotenzahlen. Das Minimum wird jedoch schon bei weniger Knoten erreicht. Außerdem stabilisiert sich die RTT bei hohen Knotenzahlen nicht, sondern steigt immer weiter an.

Eine mögliche Ursache dafür könnte die Art und Weise sein, wie Prozess 0 testet, ob die Nachrichten von den entfernten Knoten schon angekommen sind. So wird pro Knoten das letzte Byte im jeweiligen Bereich des Empfangsbuffers getestet. Dies hat eventuell negative Auswirkungen auf das Cache Verhalten. Außerdem muss der HCA zur gleichen Zeit die Daten in den Empfangsbuffer schreiben, so dass zusätzliche Speicherlast entsteht. Um herauszufinden, ob dies der Grund für die steigende RTT ist, könnte man bei einem weiteren Test RDMAW mit Immediate Daten verwenden. So kann man auf den Speichertest verzichten und erhält wie bei SEND ein CQE nachdem eine Nachricht angekommen ist.

5.4.3 SEND mit Unreliable Datagram

SEND mit Unreliable Datagram ist bei wenigen Knoten zunächst langsamer als SEND mit Reliable und Unreliable Connection. Zurückzuführen ist dies eventuell auf die Tatsache, dass bei Unreliable Datagram ein QP nicht fest mit einem andern QP verbunden ist. So muss bei jedem Send Request ein Ziel mit angegeben werden, was beim HCA möglicherweise einen höheren Overhead verursacht.

Bei höheren Knotenzahlen fällt die RTT im Gegensatz zu SEND mit Reliable und Unreliable Connection jedoch weiter, so dass Unreliable Datagram ab einem bestimmten Punkt schneller ist. Da der größte Unterschied zu den Connected Services darin besteht, dass die QPs bei Unreliable Datagram nicht fest miteinander verbunden sind und Prozess 0 somit zur Kommunikation mit allen anderen Prozessen nur ein QP benötigt, verstärkt dies die Vermutung, dass der nötige QP Wechsel bei Reliable und Unreliable Connection dazu führt, dass die RTT ab einem bestimmten Punkt zuerst steigt und dann konstant bleibt.

5.4.4 SEND mit Multicast

SEND mit Multicast ist bei wenigen Prozessen zuerst weitaus langsamer als Unreliable Datagram SEND, obwohl beide Verbindungsarten den gleichen QP Typ verwenden. Eine mögliche Erklärung dafür sind die zusätzlichen Parameter für das globale Routing, die für Multicast nötig sind und der daraus resultierende Global Route Header. Außerdem ist anzunehmen, dass die Weiterverteilung des Multicast Pakets durch den oder die Switches auch zusätzliche Zeit kostet.

Je mehr Knoten beteiligt sind, desto effizienter wird Multicast jedoch, da Prozess 0 immer bloß eine Nachricht versenden muss, egal wieviele entfernte Knoten erreicht werden sollen. Die Verteilung übernehmen dabei die Switches beziehungsweise die Router. So fällt die RTT für Multicast weitaus stärker als die für Unreliable Datagram, was zu dem in Abschnitt 5.2.1 beschriebenen Geschwindigkeitsvorteil führt.

5.4.5 RDMAR mit Reliable Connection

Die Aussagekraft der gemessenen Zeiten hängt stark von der Gleichzeitigkeit von Multicast ab. So ist die $n:1$ read time nur genau zu bestimmen, wenn alle Knoten die Multicast Nachricht von Prozess 0 möglichst zeitgleich erhalten. Außerdem werden bei den entfernten Prozessen Verzögerungen, die zwischen der Synchronisation und dem Start der Zeitmessung liegen könnten, nicht mit berücksichtigt.

Der in Abschnitt 4.3 beschriebene veränderte Ablauf des Benchmarks führt dazu, dass sich die RDMAR Operationen von Prozess 0 und den entfernten Prozessen nicht überlagern können. Die $1:n$ read time fällt dabei bei kleinen Nachrichten weniger stark als die $n:1$ read time, was dazu führt, dass die $n:1$ read time bei hohen Knotenzahlen weitaus geringer ist. Zurückzuführen ist dies auf die mögliche Überlagerung der RDMAR Operationen der entfernten Prozesse. Die $1:n$ read time fällt anfangs zwar auch, jedoch in weitaus geringerem Umfang als bei RDMAR oder SEND. Die Funktionsweise von RDMAR liefert dafür eine mögliche Erklärung. So kann Prozess 0 an mehrere entfernte Prozesse einen RDMAR Request absenden, ohne dass er auf einen RDMAR Reply warten muss. Die Anzahl der ausstehenden RDMAR Requests ist jedoch begrenzt.

Bei größeren Nachrichten ist die $n:1$ read time nur bei sehr wenigen Knoten geringer und steigt danach über den Wert der $1:n$ read time an. Eine mögliche Ursache dafür ist die Abarbeitung der RDMAR Requests bei Prozess 0. So sammeln sich die Requests an und die RDMAR Replies müssen sequentiell abgearbeitet werden.

Wieso der Wert für die RTT ansteigt, obwohl $1:n$ und $n:1$ read time ab einem bestimmten Punkt konstant bleiben, ist momentan nicht geklärt. Die Messungen für die $1:n$ und $n:1$ read time wurden jedoch einzeln durchgeführt, also nur durch Benchmarkdurchläufe mit einer Nachrichtengröße. Die Messungen für die RTT hingegen wurden für alle Nachrichtengrößen in Folge gemacht.

5.4.6 Fazit

Für kleine Nachrichten und wenige entfernte Knoten ist RDMAR mit Reliable Connection eine gute Wahl. Zwar liegen die Bestwerte von SEND mit Reliable Connection nicht weit über den Werten von RDMAR, aber die Durchschnittswerte von RDMAR sind niedriger. Bei der Verwendung von Unreliable

Connection erhält man dabei keine Vorteile und verzichtet auf die gesicherte Übertragung, was den Einsatz von Unreliable Connection nicht sinnvoll macht.

Je nach Nachrichtengröße lohnt sich schon ab 9 bis 12 Knoten der Einsatz von Multicast. Je mehr entfernte Knoten dabei an der Kommunikation beteiligt sind, desto effektiver wird Multicast. Die Durchschnittswerte liegen jedoch weitaus höher als die Bestwerte. Ein weiteres Problem ist die mangelnde Unterstützung. So ist die Implementierung von Multicast nur optional bei HCAs.

Für größere Nachrichten muss entweder die Anwendung die Nachricht in kleinere Teile aufteilen oder man ist auf Reliable und Unreliable Connection angewiesen. Reliable Connection SEND ist dabei die beste Alternative, da man damit im Gegensatz zu Unreliable Connection eine sichere Übertragung und im Vergleich zu RDMAW die niedrigere RTT erhält.

6 Schlussfolgerungen und Ausblick

Welcher Infiniband Übertragungsmodus zusammen mit welcher Verbindungsart am besten zur Umsetzung von kollektiven Operationen paralleler Programmiersprachen geeignet ist, lässt sich nicht pauschal sagen. So sollte eigentlich anhand der Nachrichtengröße und der Anzahl beteiligter Knoten entschieden werden, wie eine Nachricht übertragen werden soll. Außerdem wird für die meisten Operationen gefordert, dass versendete Nachrichten auch wirklich ankommen. Deshalb muss bei Verwendung von Unreliable Connection, Unreliable Datagram oder Multicast ein zusätzlicher Mechanismus zur Sicherung der Übertragung hinzugefügt werden. Falls dies effizient möglich ist, so legen die ermittelten Bestwerte für die RTT nahe, dass Multicast für hohe Knotenzahlen im besonderen dazu geeignet ist, eine solche Operation umzusetzen.

Um verlässliche Durchschnittswerte für die RTT zu erhalten und herauszufinden, wie wahrscheinlich der Verlust von Nachrichten bei Unreliable Services ist, sind sicher weitere ausführliche Tests notwendig. Der aktuelle Benchmark kann dabei als Basis dienen.

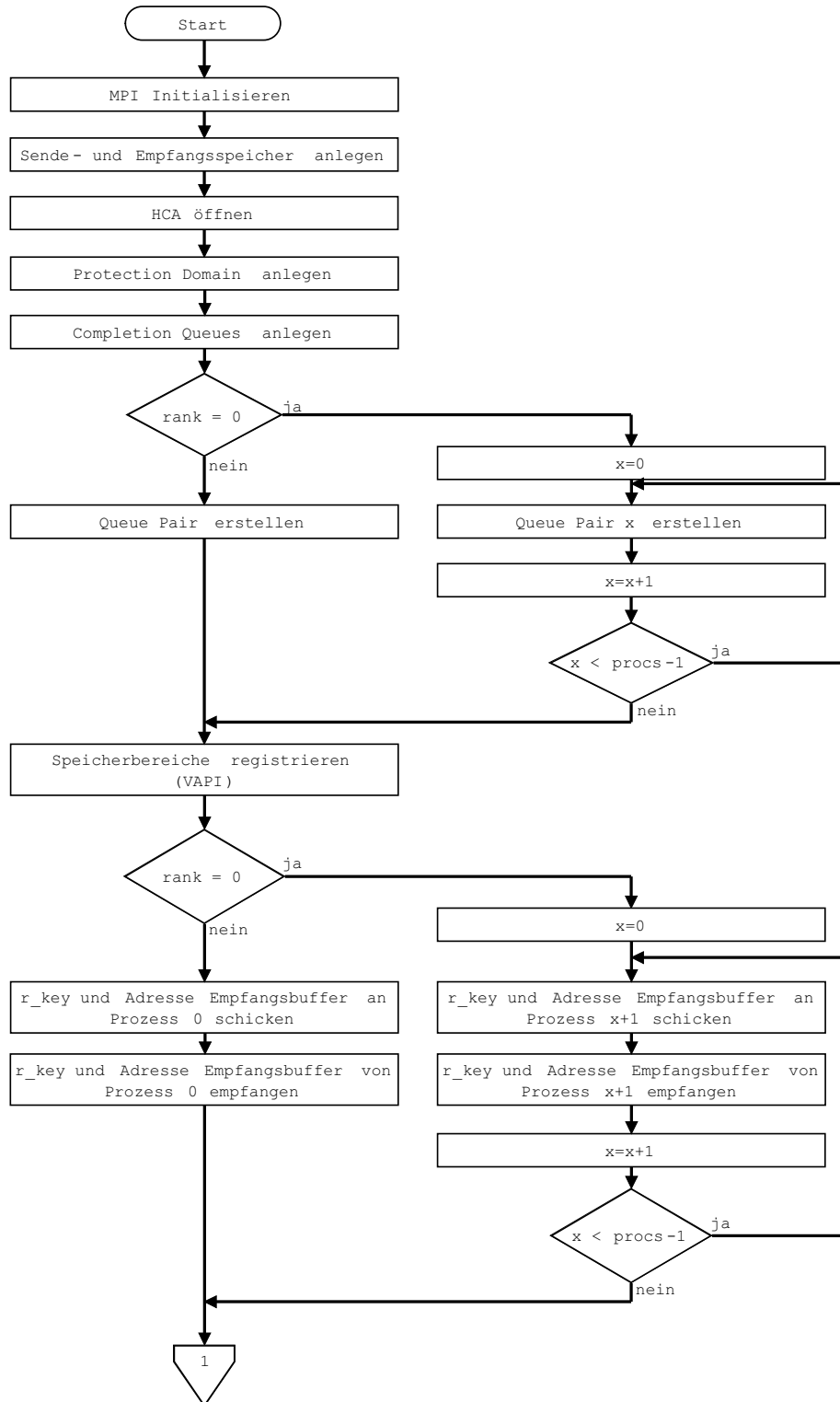
Literatur

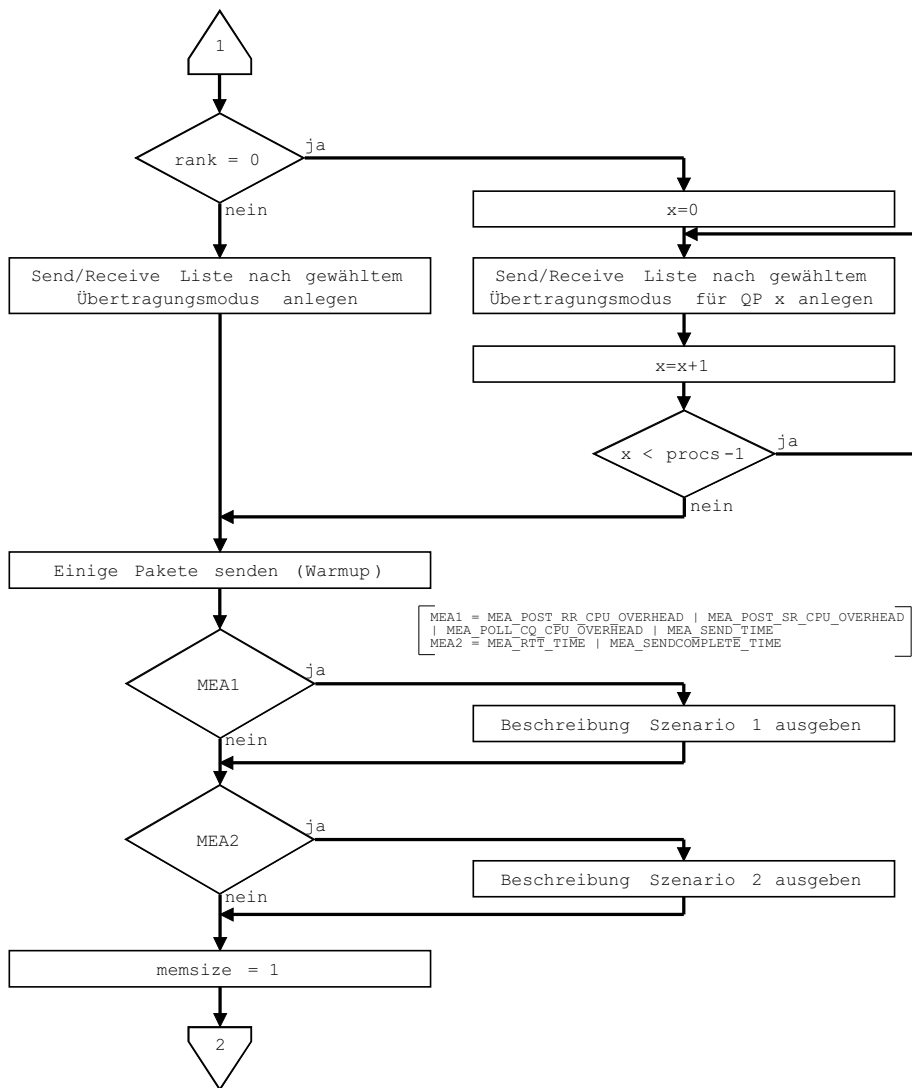
- [1] Infiniband Trade Association. InfiniBand Architecture Specification Volume 1, Release 1.2 (2004)
- [2] Tom Stanley. InfiniBand Network Architecture (2003)
- [3] Mellanox Technologies. Mellanox IB-Verbs API (VAPI) Rev. 1.00 (2004)
- [4] Mellanox Technologies. InfiniHost SDK User Guide Rev. 1.20 (2002)
- [5] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauer, E. Santos, R. Subramonian, T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation (1993)
- [6] Infiniband Trade Association. <http://www.infinibandta.org/>
- [7] Mellanox Technologies. <http://www.mellanox.com/>

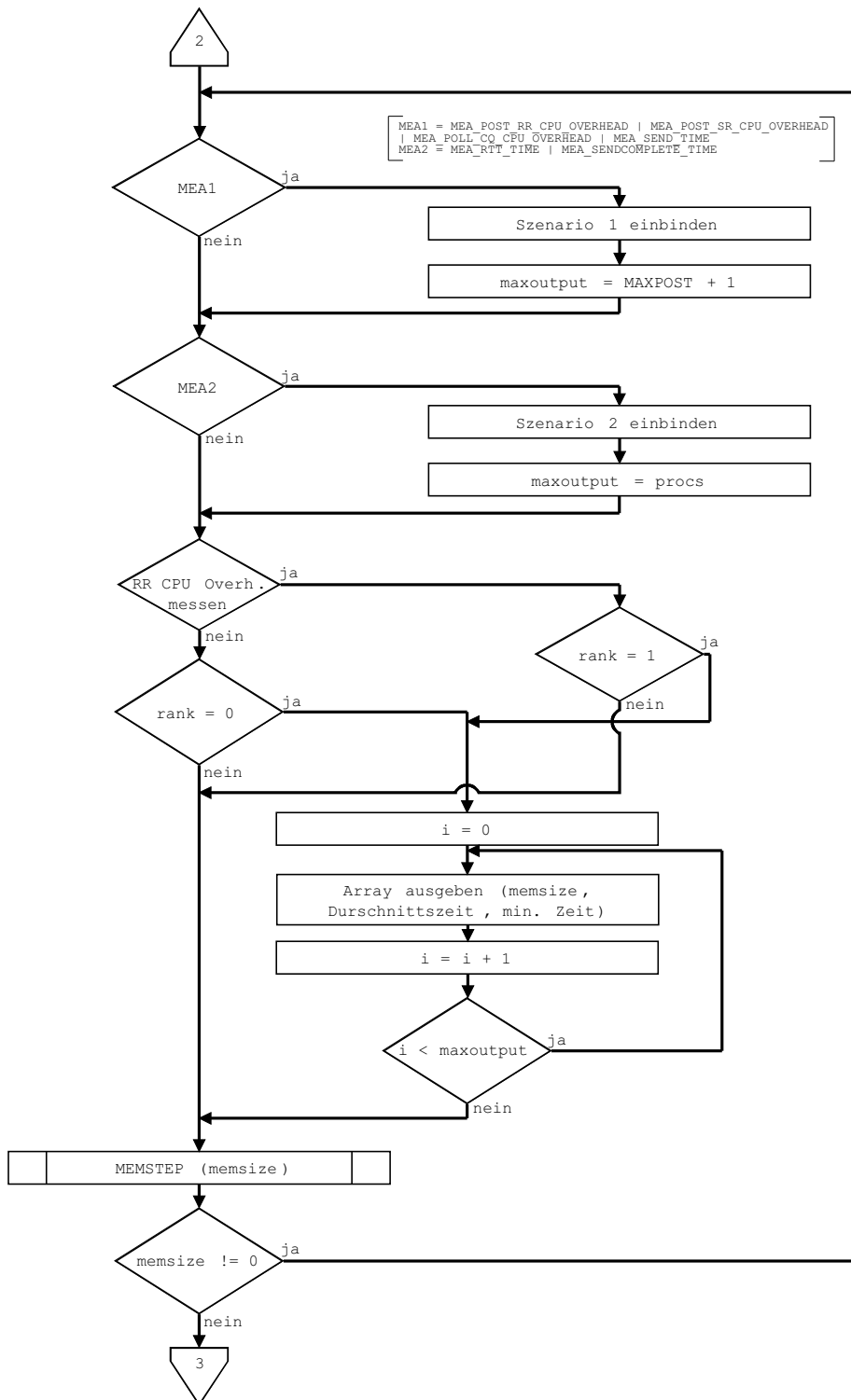
A Flussdiagramme

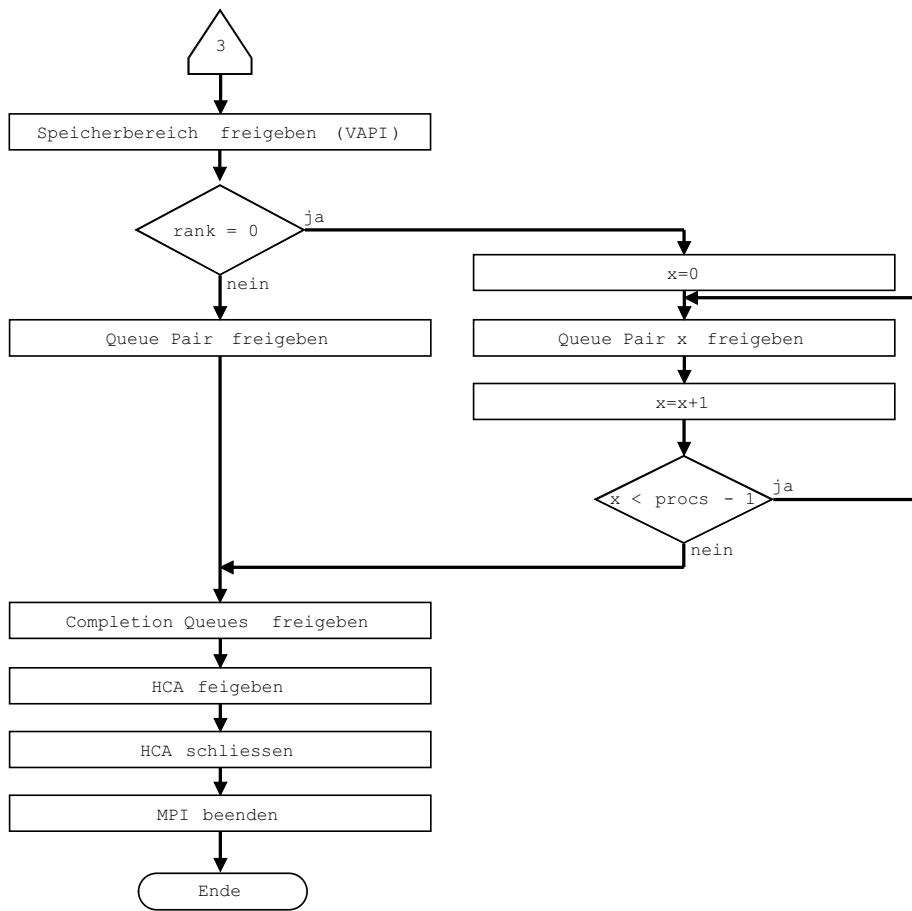
Auf den folgenden Seiten finden sich Flussdiagramme zu Teilen des schon vorhandenen Benchmarks.

A.1 mpi_iba_bench.c

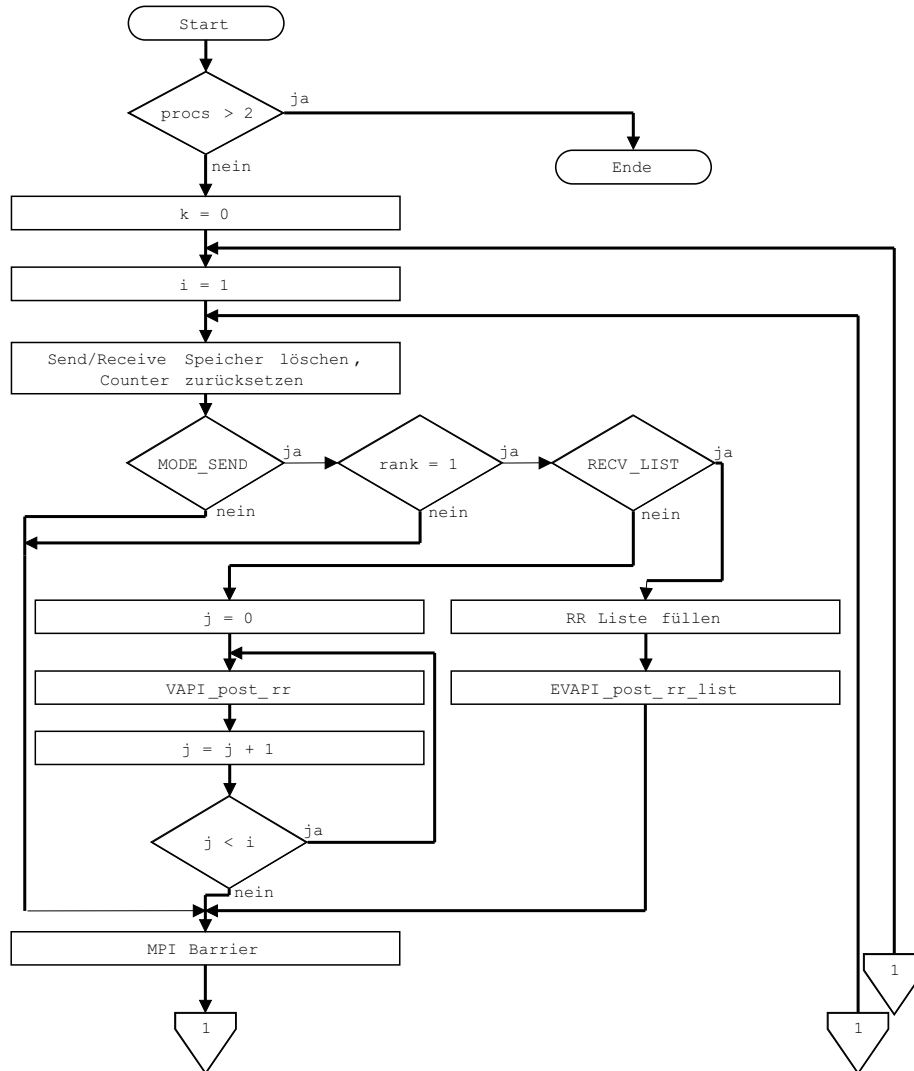


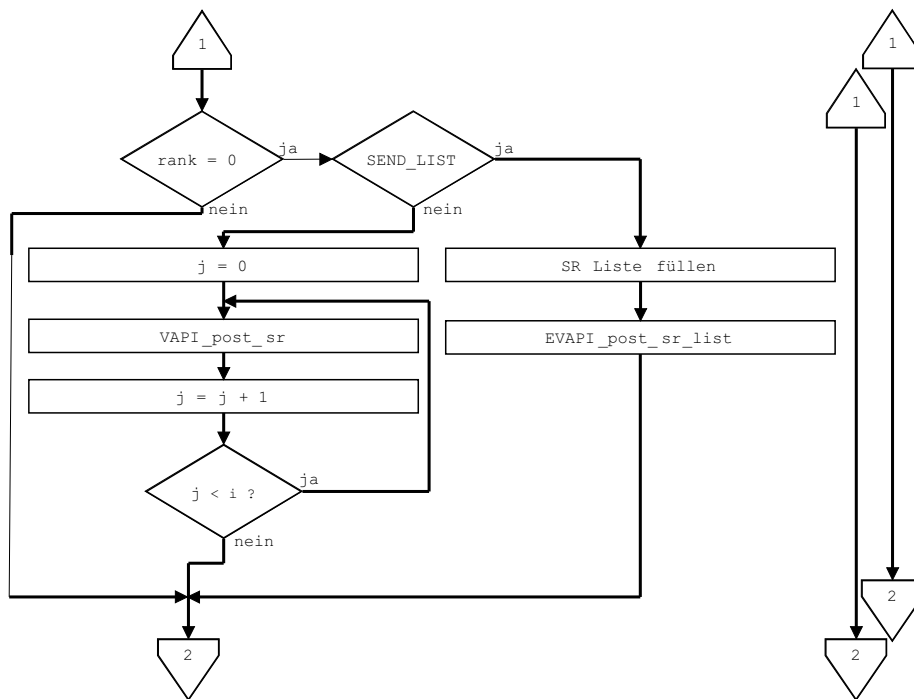


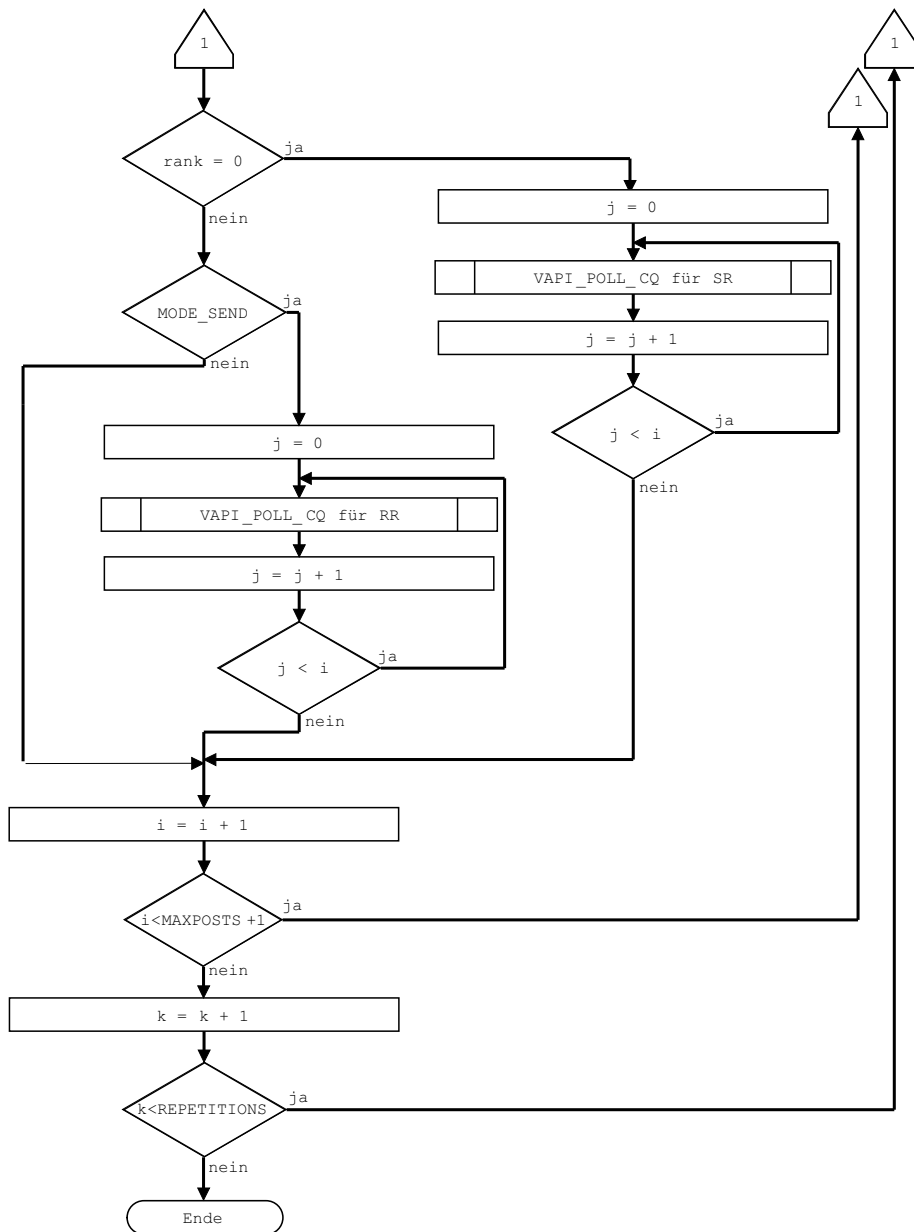




A.2 scenario_1.sub.c







A.3 scenario_2.sub.c

