Höchstleistungsrechenzentrum HLRS
Universität Stuttgart
Prof. Dr.-Ing. M. Resch
Nobelstraße 19
70569 Stuttgart

Analyse und Optimierung der Softwareschichten von wissenschaftlichen Anwendungen für Metacomputing

Von der Fakultät Maschinenbau der Universität Stuttgart
zur Erlangung der Würde eines Doktor-Ingenieurs (Dr. - Ing.)
genehmigte Abhandlung

vorgelegt von

Dipl. -Inf. Rainer Keller
aus Konstanz/Deutschland

Hauptberichter:	Prof. Dr.-Ing. Michael Resch
Mitberichter:	Univ.-Prof. Dr.-Ing. Prof. E.h. Dr.-Ing. E.h. Dr. h.c. mult. Engelbert Westkämper

Tag der Einreichung: 29. September 2007
Tag der mündlichen Prüfung: 19. Dezember 2008

Höchstleistungsrechenzentrum Stuttgart

2008

Höchstleistungsrechenzentrum Stuttgart (HLRS)
Universität Stuttgart
Nobelstraße 19
D-70569 Stuttgart

Zusammenfassung

Für parallele Anwendungen ist das Message Passing Interface (MPI) das Programmierparadigma der Wahl für Höchstleistungsrechner mit verteiltem Speicher. Mittels des Konzeptes des MetaComputings wiederum können verschiedenste Rechenressourcen mit PACX-MPI gekoppelt werden. Dies ist einerseits von Interesse, weil Problemgrößen gelöst werden sollen, die nicht auf nur einem System ausgeführt werden könnten, andererseits, weil gekoppelte Simulationen gerechnet werden, die auf bestimmten Rechnerarchitekturen ausgeführt werden sollen oder weil Systeme mit bestimmten Eigenschaften wie Visualisierungs- mit parallelen Rechenressourcen verbunden werden müssen. Diese Koppelung stellt für die verteilten Anwendungen eine Barriere dar, da Kommunikation zu nicht-lokalen Prozessen weitaus langsamer ist, als über das rechnerinterne Netzwerk.

In dieser Arbeit werden Lösungen auf den Software-Ebenen ausgehend von der Netzwerkschicht, durch Verbesserungen innerhalb der verwendeten Middleware, bis hin zur Optimierung innerhalb der Anwendungsschicht erarbeitet.

In Bezug auf die unterste Softwareschicht wird für die Middleware PACX-MPI eine allgemeine Bibliothek zur Netzwerkkommunikation auf Basis von User Datagram Protocol (UDP) entwickelt. Somit können Limitierungen des Transport Control Protocols (TCP) umgangen werden, vor allem in Verbindung mit Netzwerken mit hoher Latenz und großer Bandbreite, so genannte Long Fat Pipes. Die hier implementierte Bibliothek ist portabel programmiert und durch die Verwendung von Threads effizient. Dieses Protokoll erreicht gute Werte für die Bandbreite im Local Area Network (LAN), aber auch im Wide Area Network (WAN). Getestet wird dieses Protokoll zur Veranschaulichung mittels einer Verbindung zwischen Rechnern in Stuttgart und Canberra, Australien.

Innerhalb der Middleware wird die Optimierung der kollektiven Kommunikationsroutinen behandelt und am Beispiel der Funktion `PACX_Alltoall` die Verbesserung anhand des `IMB` Benchmarks auf einem Metacomputer gezeigt. Zur Analyse der Kommunikationseigenschaften wird die Erweiterung einer Tracing-Bibliothek für PACX-MPI, sowie die Implementierung einer generischen Schnittstelle zur Messung der Kommunikationscharakteristik auf MPI-Schicht erläutert. Weiterhin wird eine allgemeine MPI-Testsuite vorgestellt, die beim Auffinden von Fehlern sowohl in PACX-MPI, als auch innerhalb der Open MPI Implementierung hilfreich war.

Auf der obersten Softwareschicht werden Optimierungsmöglichkeiten für Anwendungen für MetaComputing aufgezeigt. Beispielhaft wird die Analyse des Kommunikationsmusters einer Anwendung aus dem Bereich der Bioinformatik gezeigt. Weiterhin wird die Implementierung des Cachings und Prefetchings von vielfach kommunizierten Daten mit räumlicher und zeitlicher Lokalität vorgestellt. Erst die Methodik des Cachings und Prefetchings erlaubt die Ausführung der Anwendung in einem Metacomputer und ist exemplarisch für eine Klasse von Algorithmen mit ähnlichem Kommunikationsmuster.

Abstract

The Message Passing Interface (MPI)-Standard is the programming paradigm for parallel applications on high-performance computers with distributed memory. Using the MetaComputing-concept, a wide variety of computing resources may be coupled using PACX-MPI. This technique is of interest, either to solve problem sizes, that otherwise could not be run on a single system, or on the other hand, because coupled simulations may allow execution on specific computer architectures tailored for parts of the whole problem, or because the computer systems offer specific features, such as connecting visualization with parallel computing resources. This coupling however, poses an obstacle to the distributed application, as communication to non-local processes is considerably slower then over the high-performance network between local processes.

In this treatise, solutions on several software layers are being worked on, including the network software stack, improvements in the middleware, as well as optimizations done within the application layer.

With regard to the lowest software layer, a general network communication library is developed for the middleware PACX-MPI, based on the User Datagram Protocol (UDP). Thereby, limitations of Transport Control Protocol (TCP) are circumvented, especially with regard to networks with high latency and large bandwidth, the so-called long fat pipes. The implemented library is programmed portably and is efficient by using a multi-threaded approach. This protocol achieves good results with regard to bandwidth in a Local Area Network (LAN), as well as in a Wide Area Network (WAN). For exemplification, the protocol is tested with a connection between computers in Stuttgart and Canberra, Australia.

Within the middleware, the optimization of collective communication routines is being shown on the function `PACX_Alltoall` by means of the `IMB` benchmark on a Metacomputer. To enable analysis of communication behaviour, the extension of a tracing-library for PACX-MPI is exemplified, as well as the implementation of a generic interface for the measurement of communication characteristics of the MPI layer. Additionally a generic MPI-Testsuite is presented, that has been helpful in finding bugs within PACX-MPI, and moreover within the Open MPI implementation.

On the highest software layer, possibilities for optimization of applications for MetaComputing are being presented. For instance, the methodological analysis of the communication pattern of a bioinformatic application is shown. Moreover, the implementation of caching and prefetching of repeatedly communicated data with spatial and temporal locality is introduced. Only using this method of caching and prefetching allows the execution of this application in a Metacomputer and is exemplary for a class of algorithms with a similar communication patterns.

Danksagung

Mein Dank richtet sich an Prof. Michael Resch, der mit seinem großen Einsatz das Höchstleistungsrechenzentrum aufgebaut und so für ein einmaliges Arbeitsumfeld gesorgt hat. Weiterhin möchte ich Prof. Westkämper für seine gute Zusammenarbeit und Mithilfe als Zweitkorrektor danken.

Am HLRS möchte ich allen Mitarbeiter meiner Arbeitsgruppe und darüber hinaus dem gesamten netten Team danken. Es macht Spaß, im Doktorandenseminar zu diskutieren und Ideen weiterzuentwickeln. Ganz besonderer Dank geht hier an meinen ehemaligen Betreuer Edgar Gabriel, an Peggy Lindner sowie Uwe Küster für die hilfreichen Kommentare und konstruktiven Diskussionen. Weiterhin möchte ich Alexander Schulz, sowie Katharina Benkert für Ihre tolle Unterstützung danken. Für den Support bei den MetaComputing Tests möchte ich mich herzlich bei allen Administratoren am HLRS bedanken. Auch wenn man mehrfach Firewalls freischalten und Software nachinstallieren oder konfigurieren musste, konnte man sich immer auf die Mannschaft verlassen, namentlich Rolf Supper, Bernd Krischok und Thomas Beisel, sowie die NEC Systemgruppe, Holger Berger, Danny Sternkopf und Stefan Haberhauer. Ohne diesen Support wäre der Gewinn der HPC-Challenge auf der SuperComputing nicht möglich gewesen. Weiterhin möchte ich Prof. Jesus Labarta vom Barcelona Supercomputing Center (BSC) für seine freundliche Zusammenarbeit mit Paraver, Prof. Marc Garbey von der University of Houston (UH) für die tolle Unterstützung, sowie Dr. Markus Buchhorn von der Australian National University (ANU) für den Zugriff auf seine Rechner danken.

Ohne meine Eltern wäre dies alles überhaupt nicht möglich gewesen, deswegen möchte ich mich bei Ihnen an dieser Stelle noch mal ganz herzlich für ihre große Unterstützung bedanken. Mein größter Dank gilt meiner Freundin Elke Strauß, die mir bei meinem Tun moralisch und mit guten Ratschlägen zur Seite gestanden hat. Schon aufgrund des normalen Arbeitspensums war es manchmal auch am Wochenende nicht einfach, mich von der Arbeit loszueisen. Für Deine tolle Art möchte ich mich hier bei Dir herzlich bedanken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Wissenschaftliche Simulationen wie zum Beispiel im Ingenieurbereich die Berechnung von Strömungsvorgängen um einen Flugzeugflügel oder in der Bioinformatik die Berechnung des Faltungsvorganges einer RNA-Sequenz verlangen große Rechen- und Speicherressourcen. Diese Ressourcen können in vielen Anwendungsfällen nur von Höchstleistungsrechnern, so genannten Supercomputern erzielt werden.

1.1 Entwicklung des Höchstleistungsrechnens

In den vergangenen Jahren hat die wissenschaftliche Gemeinde den enormen Zuwachs der Prozessorperformance genutzt, immer komplexere und detailliertere Problemstellungen zu berechnen. Beispielsweise wird derzeit daran gearbeitet, nicht nur das Strömungsverhalten eines Flügels zu berechnen, sondern den gesamten Flügel, Klappen und Triebwerk mit detailliertem Einlass dreidimensional zu berechnen. So lässt sich durch die Optimierung des Luftwiderstandskoeffizienten um 1% der Verbrauch um 10% reduzieren. Insofern kann man das ökonomische Interesse der Fluggesellschaften ermessen, ganz abgesehen von den ökologischen Vorteilen. Eine weitere wichtige Entwicklung sind Strömungsberechnungen gekoppelt mit der Simulation der Schallausbreitung zur Geräuschreduktion, bspw. um zukünftige strengere gesetzliche Grenzwerte einzuhalten. Diese Simulationen allerdings erfordern die numerische Lösung sehr großer Gleichungssysteme, was wiederum eine große Leistung der verwendeten Rechner erfordert: die Klasse der Supercomputer, einem Begriff geprägt durch die Rechner von Seymour Cray [131]. Erst durch die gestiegene Geschwindigkeit und die durch Massenfertigung günstige Herstellung von Mikroprozessoren werden solche Simulationen rentabel.

Für die Geschwindigkeitssteigerung der Prozessoren der Vergangenheit gibt es mehrere Gründe. Der Ingenieur Gordon Moore von Intel sagte bereits in den 60er Jahren die Zunahme der Transistoren auf einem Chip anhand der bisherigen Entwicklung voraus. Seitdem folgt die Anzahl der Transistoren und damit die Leistung eines Prozessors näherungsweise dem so genannten Moore'schen Gesetz, wonach sich die Transistoren pro Chip etwa alle 18 Monate verdoppelt [99]. Dies wurde durch Miniaturisierung in der Prozessorfertigung auf Strukturgrößen von derzeit 65 nm erzielt, womit sich wiederum die Taktfrequenz von wenigen Megahertz auf Frequenzen im Gigahertzbereich steigern ließen.

Tatsächlich hat sich die Prozessorperformance in Fließkommazahlen seit 1996 vertausendfacht [34]. Abbildung 1.1 zeigt den zeitlichen Zusammenhang am Beispiel der Intel-Prozessoren über mehrere Generationen (aus [119, 131]). Einerseits wird die Fließkommaleistung durch verschiedene architektonische Verbesserungen deutlich (oberes Diagramm,

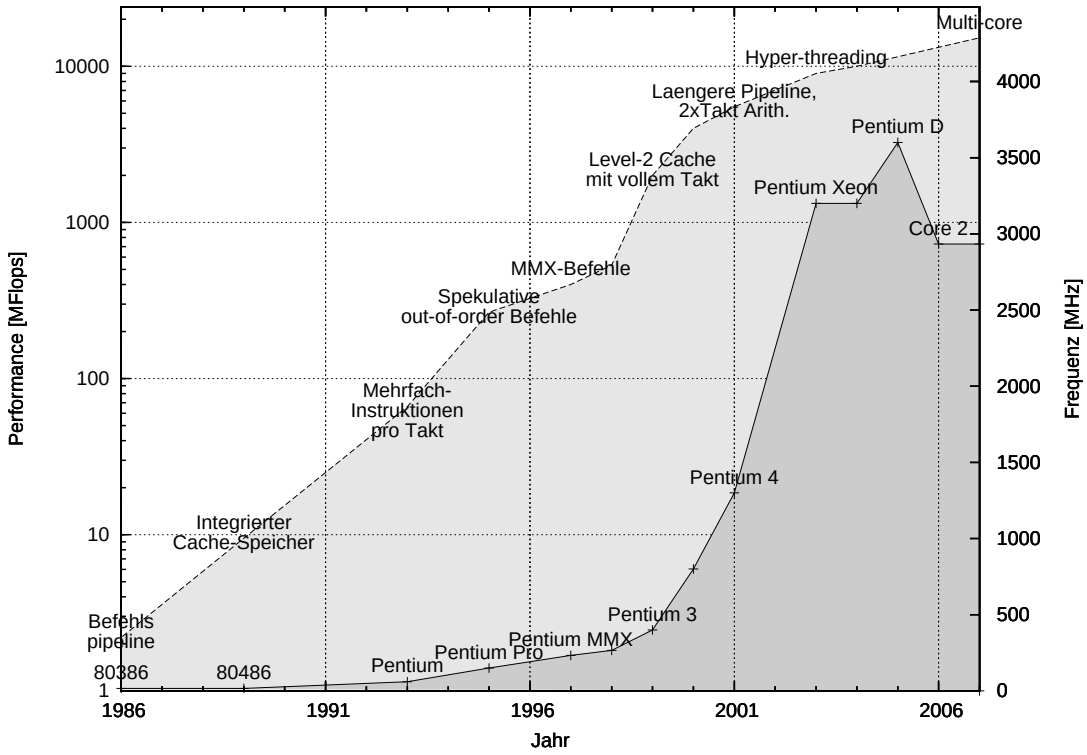

Abbildung 1.1: Entwicklung der Intel-Prozessoren in Bezug auf Fließkommageschwindigkeit (linke Skala; log) und Taktrate (rechte Skala, linear)

logarithmische Skala links), andererseits kann man die Steigerung der Taktfrequenz (unteres Diagramm, lineare Skala rechts) der Prozessoren bei Markteinführung ablesen. Diese immense Steigerung der Taktrate lässt sich auf eine Verkleinerung der Strukturen und damit der kürzeren Signallaufzeit, als auch mit der Verlängerung der Befehlspipeline[1] erklären. Es wird aber auch deutlich, dass der Steigerung der Taktfrequenz gewisse Grenzen gesetzt sind. Neben den wirtschaftlichen Gründen spielen hier vor allem thermische Probleme eine Rolle, da die Energiedichte auf der Chipoberfläche höher als bei einem Bügeleisen liegt. Die entstehende Wärme abzuführen ist durch Luftkühlung nicht mehr möglich.

Ebenso wie die Leistung eines einzelnen Prozessors, hat die akkumulierte Leistung eines parallelen Höchstleistungsrechners, d. h. Rechner mit hunderten bis mehreren tausend Prozessoren in dieser Zeit sogar um den Faktor 2800 zugenommen [34]. Besonders der Umstand, dass heute Mikroprozessoren mittels Hyperthreading [14, 101] mehrere Ausführungspfade und mehrere Prozessorkerne (engl. Multi-Core Prozessoren) und zukünftige Prozessoren mehrere dutzend Kerne auf einem Chip vereinen, erfordert zwingend eine parallele Programmierung, wenn durch eine Anwendung eine große Leistung erzielt werden soll.

[1] Instruktionen werden durch den Prozessor in kleinere Befehle (μops) zerlegt, den Einheiten zugeordnet und abgearbeitet. Beim Pentium4 sind dies bis zu 29 Schritte (engl. Stages).

Die Parallelisierung der vorhandenen Software ist damit der wichtigste Faktor, zukünftige Hardware zu nutzen, um aufwändige Simulationen überhaupt berechnen zu können, bzw. diese innerhalb einer vorgegebenen Zeit fertig stellen zu können. Eine genauere Wettervorhersage der nächsten Tage nützt nichts, wenn Sie mehrere Tage zur Berechnung braucht.

Mit den beiden hauptsächlich verwendeten Standards für parallele Programmierung, MPI [94, 95] und OpenMP [105, 106] haben sich Programmierschnittstellen für Systeme mit verteiltem Speicher, bzw. Systeme mit gemeinsamen Speicher etabliert. Dies vereinfacht die Programmierung und Portierung von parallelen Programmen auf neue hoch skalierende Systeme mit mehreren evtl. zehntausend Prozessoren, bzw. ermöglicht dies erst. Weitere Programmiermodelle werden in den Abschnitten 1.3.1, sowie 1.3.2 besprochen.

1.2 Klassifikation paralleler Rechnerarchitekturen

Um einen Überblick von Rechnerarchitekturen im Allgemeinen und parallele Rechner im Speziellen zu bekommen, kann man diese nach den Technologien einteilen [1]. Eine Klassifikation von parallelen Architekturen sollte durch die Charakteristika der Hardware erfolgen, einerseits anhand der Funktionalität der Prozessoren, andererseits anhand der Verknüpfung von Kommunikation und Synchronisation der rechnenden Einheiten.

Klassifikation anhand der Befehlssätze

Befehlssätze eines Prozessors wurden im Laufe der Entwicklung einer Prozessorlinie immer etwas verändert und den Bedürfnissen der ausgeführten Algorithmen angepasst. Als Beispiel sollen hier die Erweiterungen der ältesten Mikroprozessorlinie dienen: Ausgehend vom ursprünglichen Befehlssatz der Intel-Prozessoren der 8080-Baureihe (8086, 80186, ..., 80486 und Pentium-Reihe) wurden Befehle hauptsächlich für graphische Verarbeitung von Spielen hinzugefügt. Die Tabelle auf der nächsten Seite schlüsselt die Befehlssatzerweiterungen auf[2].

Es werden aber nicht nur Befehle hinzugefügt, sondern auch gelegentlich entfernt, wenn diese sich als nicht effizient bezüglich der verwendeten Chipfläche erweisen. Ein Beispiel hierfür ist der 80386-Prozessor, bei dem Befehle in einer späteren Version des Prozessors (B2-Stepping) wieder entfernt wurden [139].

Besonderes Potential aber bieten Spezialprozessoren, die für einen bestimmten Zweck im Hinblick auf Befehlssatz und Ausführungsgeschwindigkeit optimiert sind. Ein Beispiel für solche Spezialprozessoren sind Vektorprozessoren, die für numerische Anwendungen optimiert sind. Sie bieten Instruktionen um Operationen auf bis zu 256 Operanden gleichzeitig anzuwenden. Einerseits reduziert dies den Overhead der Instruktionen, andererseits laufen diese Operationen mit sehr hoher Geschwindigkeit, da auch die Speichersysteme auf den vektorartigen Zugriff optimiert sind. Weiterhin werden die Ausgaben der Operationen für nachfolgende Vektoroperationen geeignet miteinander kombiniert.

[2] Die Chipversion und unterstützten Befehle lassen sich mit der `cpuid`-Instruktion auslesen.

Jahr	CPU	Kürzel	Funktionalität der Befehle
1971	4004	–	Erster Mikroprozessor
1978	8086	–	Einführung der x86-Architektur
1982	80286	–	Erweiterung des Adressraumes auf 16 MB
1985	80386	DX	32-Bit Erweiterung (Register + Befehlssatz) für einen Adressraum von 4 GB
1989	80486	Floating Point	Integration der Fließkommaeinheit und Verbesserung der Anbindung an Hauptprozessor
1997	Pentium	Multimedia Extension (MMX)	56 Befehle zum gleichzeitigen Berechnen (Addition, Multiplikation) von bis zu acht Ganzzahlwerten
1999	Pentium3	Internet Streaming Extension (SSE)	70 Befehle zur Berechnung von zwei oder vier Fließkommazahlen mit einfacher Genauigkeit
2001	Pentium4	SSE2	144 Befehle zur Berechnung entweder zwei oder vier Fließkommazahlen mit doppelter Genauigkeit
2004	Core	SSE3	13 Befehle zur Synchronisation zwischen Threads und der Anbindung der Fließkommaeinheit
2007	Zukunft	SSE4.1/SSE4.2	Befehle für die Vektorregisterauswahl

Tabelle 1.1: Erweiterungen des Intel-Befehlssatzes

Somit werden Befehlsketten für besonders schnelle Berechnungen erreicht. Ein neues Ergebnis wird nach einer anfänglichen Aufbauphase der Kette zu jedem Takt zurückgeschrieben (so genanntes Pipelining). Der Overhead zum Dekodieren und Ausführen der Instruktion ist damit pro Operand minimal. Algorithmen, die eine große Vektorlänge bieten, wie bspw. Simulationen aus dem Ingenieursbereich oder Matrizen-Operationen auf großen dicht besetzten Matrizen, lassen sich dadurch mit hoher Leistung ausführen. Eine Bedingung für die Ausführung mit Vektorbefehlen ist aber die Unabhängigkeit der Operanden voneinander und die Erkennung der Unabhängigkeit durch den Compiler.

Neuere Beispiele für die Verwendung von spezieller Hardware sind die Programmierung anwendungsspezifischer Prozessoren wie MD-Grape, Grafikkarten und Field Programmable Gate Arrays (FPGAs). Der Nachweis, Gitter-Boltzmann-Verfahren, Partikel-Probleme und auch lineare Gleichungslöser mit der Hardware von Grafikkarten zu nutzen, ist bereits erbracht [53]. Versuche, eine für Konsumelektronik entwickelte Plattform für numerische Simulation zu nutzen, gibt es bei dem von IBM, Sony und Toshiba entwickelten Cellprozessor. Dieser verbindet hohe Leistung durch acht parallele Einheiten mit schneller Kommunikation. Die Abbildung auf der nächsten Seite zeigt das Bild des Prozessorkerns. Gut sichtbar ist der Power-Prozessor und die acht SPEs, die durch den EIB-Bus verbunden sind. Der Cellprozessor leistet bei 3,2 GHz theoretische 200 GFlops bei Verwendung der acht SPEs.

Die Konvergenz der Mikroprozessoren zu Spezialprozessoren oder solchen mit speziellen Funktionseinheiten zeigt sich besonders bei der Integration von Vektorbefehlen, wenn auch derzeit mit kurzer und starrer Vektorlänge. Zudem werden spezielle Befehle zur Berechnung von Cyclic Redundancy Checks (CRCs) oder numerische Befehle in den

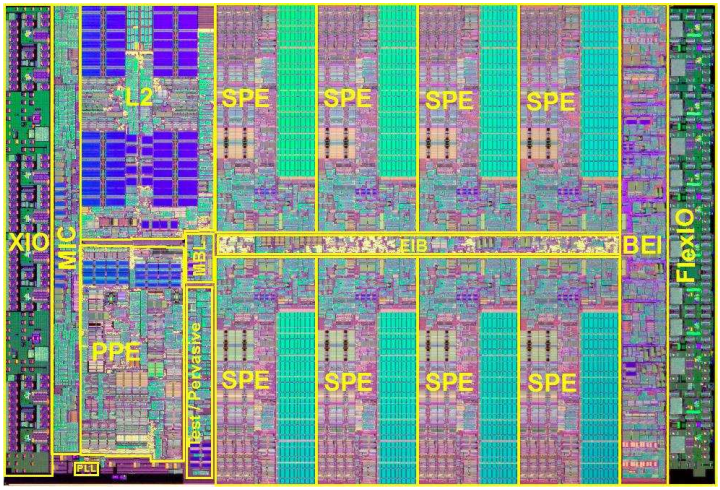

Abbildung 1.2: Abbild des Dies eines Cellprozessors von IBM

Prozessoren von Intel oder der Power-Architektur von IBM aufgenommen.

Klassifikation anhand der Prozessorkopplung

In jedem parallelen Rechner müssen die Prozessoren miteinander kommunizieren und synchronisieren können. Neben einigen anderen leistungsrelevanten Zahlen werden beim Vergleich unterschiedlicher Verbindungsnetzwerke meist nur die Bandbreite (bei sehr großen Datenpaketen), sowie die Latenz (bei 0-Byte Nachrichten) angegeben. Je nachdem, wie aufwändig die Hardware und das dabei benutzte Protokoll ist, können Daten unabhängig vom Prozessor, direkt aus dem Speicher mit großer Bandbreite übertragen werden.

Die günstigsten Lösungen mit Standardkomponenten, also Knoten aus Standard-PCs gekoppelt über ein Standardnetzwerk, sind so genannte Beowulf-Cluster [110, 131]. Hier wird als Netzwerk meist Fast-Ethernet oder Gigabit Ethernet verwendet.

Teurere Cluster mit höheren Anforderungen an die Kommunikationsgeschwindigkeit verwenden Spezialnetzwerke, wie Myrinet oder Infiniband, diese werden in den Abschnitten 2.2.2 und 2.2.3 näher vorgestellt. Während das derzeit verwendete Gigabit-Ethernet auf nur 70-80 $^{\mathrm{MB}}/_{\mathrm{s}}$, bei einer Latenz von 20-30 μs kommt, erreicht Myrinet Bandbreiten von 250-300 $^{\mathrm{MB}}/_{\mathrm{s}}$, bei 7-8 μs. Spezielle Implementierungen von Infiniband des Herstellers Qlogic erreichen Bandbreiten von 1400 $^{\mathrm{MB}}/_{\mathrm{s}}$ bei nur 1,29 μs. Für diese Art von Systemen muss das Programm explizit Daten kommunizieren, wenn ein Prozessor die Daten eines anderen Prozessors braucht. Abschnitt 1.3.2 auf Seite 8 erläutert die Art der Programmierung solcher Cluster.

Spezielle Hardware, um Speicher eines Knotens allen Knoten zur Verfügung zu stellen, wie in cache-coherent Non-Uniform Memory Architecture (ccNUMA)-Systemen ermöglicht die Parallelisierung über gemeinsamen Speicher, wie auf der nächsten Seite beschrieben. Die schnellste Kommunikation ist geschieht bei Mehrprozessorsystemen (engl. Symmetric Multiprocessing (SMP)) durch den Speicher. SMP-Systeme skalieren aber wegen

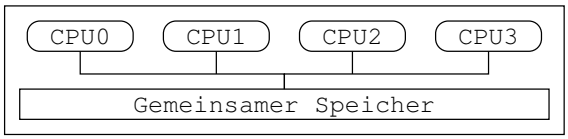

Abbildung 1.3: Parallele Rechnerarchitektur mit gemeinsamem Speicher

des Bus nur bis zu acht, bei der Verwendung von Crossbar-Chips bis zu 1024 Prozessoren. Ein Beispiel für ein solches großes System sind die Altix-Systeme von SGI.

1.3 Formen der Programmierung des Verteilten Rechnens

Über die Client/Server-Architektur als einfachste Form des verteilten Rechnens hinaus existieren viele Ansätze, verteilte Ressourcen für die Ausführung einer Anwendung und die Speicherung großer Datenmengen zu verwenden. Hier wird eine kurze Übersicht über verschiedene, vor allem für die Parallelverarbeitung wichtige Ansätze gegeben.

1.3.1 Übersicht Verteiltes Rechnen mit gemeinsamen Speicher

Bei dieser Form der Rechnerarchitekturen greifen mehrere Prozessoren auf einen gemeinsamen Hauptspeicher zu. Abbildung 1.3 stellt schematisch das Konzept einer solchen Architektur dar.

Die Parallelisierung für Rechnersysteme mit gemeinsamen Speicher ist für Programmierer mit relativ wenig Aufwand verbunden, da die Kommunikation und Synchronisation zwischen den ausführenden Einheiten über den gemeinsamen Speicher erfolgt, d. h. jeder Thread hat Zugriff auf den Speicher aller Threads. Diese ist die schnellste Form der Kommunikation und Synchronisation, welche durch den Prozessor oder auch durch Spezialhardware erfolgen kann, bspw. in so genannten ccNUMA-Maschinen wie die Origin- oder Altix-Serie von SGI [75]. Bei diesen Systemen verwaltet die Hardware mit Hilfe des Seitenschutzmechanismus des Prozessors (engl. Page mechanism/Paging) in Kooperation mit dem Betriebssystem den Ort eine Speicherseite in Bezug auf den physikalischen Knoten des Computers. Ist die Seite nicht lokal auf dem Knoten vorhanden, wird sie auf dem Knoten eingeladen. Schreibende Zugriffe werden durch ein Invalidierungsverfahren ebenfalls durch die Hardware erkannt und behandelt. Als paralleles Programmiermodell eignet sich hierfür besonders der Ansatz mit mehreren parallelen Ausführungspfaden innerhalb der Anwendung, den sogenannten Leichtgewichtigen Prozessen (LWP) oder engl. Threads. Im Folgenden werden Varianten dieses Programmiermodells vorgestellt.

Schon früh wurden von Computerherstellern Programmierschnittstellen zur Unterstützung mehrerer Ausführungspfade innerhalb einer Anwendung dem Programmierer zur Verfügung gestellt, bspw. DECthreads [32]. Erst Standards wie Posix Threads (PThreads) [20, 67] haben portable Programmierung ermöglicht. Der Programmierer muss den Zugriff auf den Speicher und andere Ressourcen explizit koordinieren. Das

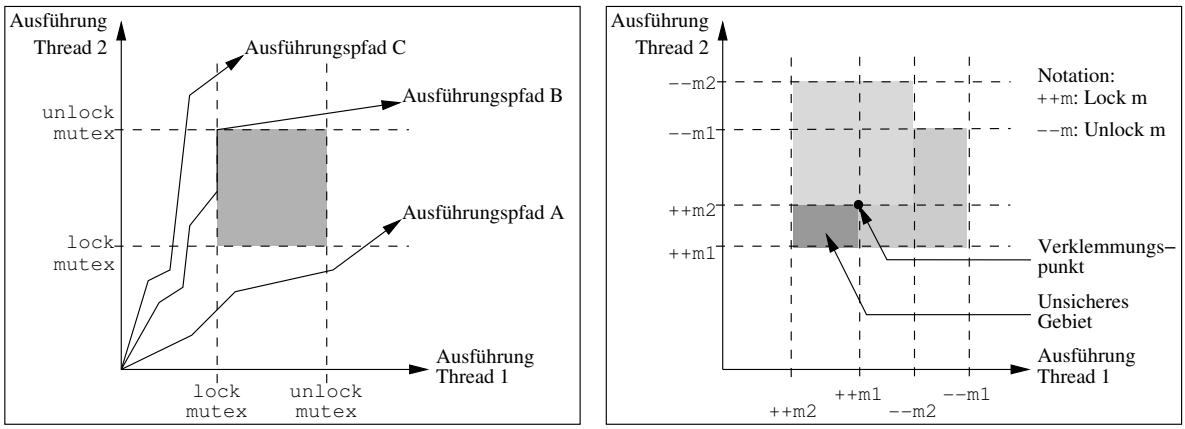

Abbildung 1.4: Mögliche Ausführungspfade und Deadlock bei Lock-Hierarchien

heisst, dass der schreibende Datenzugriff von den verschiedenen Threads gegeneinander geschützt und evtl. synchronisiert werden muss. Darin liegt auch das grösste Problem dieser Parallelisierungsmethode: Greift ein Thread des Programms auf eine Ressource, bspw. eine Speicherstelle zu, während ein anderer Thread ebenfalls schreibend auf diese Speicherstelle zugreift, verwendet ein Thread evtl. einen veralteten Wert oder überschreibt den Wert des anderen Threads, falls die Hardware dies nicht erkennt und umgeht – dies kommt auf die nicht-deterministische Ausführungsreihenfolge des in Frage kommenden Programmabschnitts an (engl. Race Condition). Der Zugriff muss hierbei durch geeignete Lockingmechanismen wie Mutexe durch den Programmierer geschützt werden, d. h. für den Abschnitt des Zugriffs auf die gemeinsame Ressource wird eine Mutex-Variable gesetzt und am Ende des Zugriffs wieder freigegeben. Die Implementierung der Funktion `pthread_mutex_lock` zum Setzen der Mutex-Variablen basiert hierbei auf dem `LOCK`-Präfix des Befehlssatzes, die anderen Prozessoren den Zugriff auf den Systembus verwehrt, zugleich wird der Wert der Mutex-Variablen mit einem Registerwert ausgetauscht: `lock cmpxchgl %edi %eax`. Ist der Wert aus dem Speicher gleich Null, ist der Mutex noch nicht belegt. Bei bereits gesetztem Mutex muss `pthread_mutex_lock` in einer Schleife auf die Freigabe durch einen entsprechenden `pthread_mutex_unlock` warten.

Bei komplexen Programmen mit mehrstufigen Lock-Hierarchien kann es aber zu einer besonderen Form einer Race Condition kommen: ein Programm setzt mehrere unterschiedliche Locks nacheinander, gibt diese aber in unterschiedlicher Reihenfolge wieder frei. Abbildung 1.4 links zeigt mögliche Ausführungspfade zweier Threads welche beide einen kritischen Abschnitt durch eine einzige Mutexvariable `mutex` synchronisieren; die Abbildung rechts offenbart mittels Abdeckung mögliche unsichere Gebiete, bei unterschiedlicher Lock/Unlock-Reihenfolge zweier Mutexe `m1` und `m2` in zwei Threads. Dies kann nicht-deterministisch, je nach unterschiedlicher Ausführungsreihenfolge zum Hängen des Programms führen.

Dennoch lassen sich bestehende, sequentielle Programme recht einfach mit Hilfe von Compilern und mit Standards wie OpenMP [105, 106] durch Direktiven parallelisie-

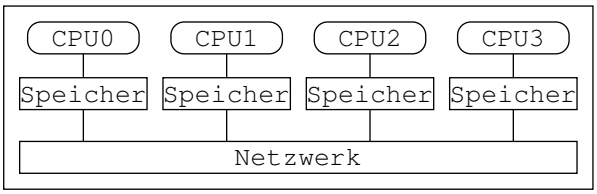

Abbildung 1.5: Parallele Rechnerarchitektur mit verteiltem Speicher

ren [19, 22, 23, 75]. OpenMP zielt auf wissenschaftliche Anwendungen, welche häufig aus Schleifen mit langer Ausführungszeit bestehen. Der Programmierer fügt hierzu vor dem parallel auszuführenden Code OpenMP-Direktiven ein. Meist basiert die Implementierung von OpenMP wiederum auf der PThread-Implementierung des Betriebssystems.

Der große Vorteil hierbei ist, dass sich die Anwendung ausgehend vom zeitaufwändigsten Programmabschnitt inkrementell parallelisieren lässt. Durch die Beschränkung auf bestimmte OpenMP Konstrukte lassen sich Race Conditions vermeiden. Da OpenMP nur die Arbeitsverteilung, nicht aber die Verteilung von Daten auf Prozessoren oder Knoten definiert, kann es hier beim Hochskalieren auf viele Threads zu Engpässen kommen, da der Zugriff auf entfernte Daten eben nicht gleich schnell wie der Zugriff auf lokale Daten ist. Ebenso hat der Programmierer durch das abstrakte Interface wenig Möglichkeiten, den Daten- und Programmfluss zu steuern.

Als Forschungsprojekt wird bei Intel derzeit an der Unterstützung von OpenMP-parallelen Programmen auf Systemen mit verteiltem Speicher gearbeitet, dem sogenannten Cluster-OpenMP [65, 69]. Dies erfordert aber zusätzlichen Aufwand durch den Programmierer: es müssen dem Compiler explizit die zu verteilenden Variablen genannt werden. Der Compiler sorgt dann dafür, dass die Daten auf den verteilten Threads immer den neuesten Stand haben, bevor darauf zugegriffen wird.

1.3.2 Übersicht Verteiltes Rechnen mit verteiltem Speicher

Bei dieser Form der Rechnerarchitektur haben die Prozessoren keinen unmittelbaren Zugriff auf den Speicher anderer Prozessoren. Der Datenaustausch zwischen unterschiedlichen Prozessoren muss explizit durch den Programmierer über ein schnelles, verbindendes Netzwerk angestoßen werden. Abbildung 1.5 zeigt schematisch das Konzept einer solchen Architektur. Als Netzwerk werden hier günstige Hardware wie Gigabit-Ethernet, aber auch Speziallösungen wie Scali, Myrinet oder Infiniband (IB), sowie andere proprietäre Herstellerlösungen eingesetzt. Ein Beispiel für letztgenannte proprietäre Lösungen ist das Netzwerk der NEC SX-Architekturen, dem sogenannten IXS Crossbar Switch, der bei der Earth-Simulator Installation in Japan 640 SX-Knoten direkt miteinander verknüpft. In der 72-Knoten NEC-SX8 Installation am HLRS können Anwendungen eine theoretische Bandbreite von 16 $^{\mathrm{GB}}/_{\mathrm{s}}$ duplex zwischen den Knoten erreichen [54].

Die Programmierung paralleler Anwendung für diese Klasse von Architekturen erfolgt meist über standardisierte Application Programming Interfaces (APIs). Neben dem in dieser Arbeit behandelten Standard MPI [58, 94–96] gibt es noch andere weniger weit

verbreitete Bibliotheken und Programmierparadigmen. Ein Beispiel für eine Kommunikationsbibliothek ist Global Arrays [102], die dem Programmierer die Arbeit der Verteilung und Koordination von verteilten Daten erleichtert. Global Arrays setzt u. a. auf Shared Memory, Cray Shmem und MPI, damit werden die verteilten Datenarrays und großen Matrizen effizient kommuniziert. Als weitere, wichtiger werdende Bibliothek ist GasNet [8] zu nennen. Diese erlaubt hoch performante, einseitige Kommunikation (engl. One-Sided Communication, d. h. Kommunikation wird durch einen Prozess initiiert). Diese Bibliothek hat entscheidende Vorteile gegenüber der One-sided Kommunikation im MPI-2-Standard [95], bspw. verhindert dessen komplexe Semantik die einfache Benutzung in Anwendungen, aber auch die Abbildung auf Hardware-basierten direkten Speicherzugriff (engl. Direct Memory Access (DMA)).

Auch im Compilerbau hat man die Problematik expliziter Kommunikation zwischen parallelen Prozessen erkannt. Besonders für hoch skalierende Anwendungen müsste der Programmierer die Verteilung der Daten und deren Austausch genau kennen. Hierfür gibt es neue Ansätze, entweder auf hoher Ebene, wie die graphische Beschreibung der Abhängigkeiten zur Angabe von Parallelität [21] oder auf niederer Ebene wie die Erweiterungen von Programmiersprachen seitens der so genannten Partitioned Global Address Space (PGAS)-Sprachen, wie bspw. Co-Array Fortran (CAF) [103], Unified Parallel C (UPC) [41] oder wie bei High-Performance Fortran (HPF) [2, 114]. Bei UPC und CAF kann der Programmierer die Verteilung der Daten mittels spezieller Zugriffsspezifikationen und Direktiven dem Compiler angeben. Die Kommunikation erfolgt dann automatisch, angestoßen durch den Compiler. Da dieser das lokale Zugriffsmuster auf die verteilten Variablen analysieren kann, werden der Kommunikationsbeginn und die Synchronisation am Ende des Austausches möglichst separiert und dazwischen von der Kommunikation unabhängige Daten berechnet. Für die Kommunikation zur Lösung eines Problems festgelegter Größe werden dann zwar weit mehr Kommunikationsschritte durchgeführt, dafür erlaubt die kleinteiligere Kommunikation eine bessere Überlappung mit der Berechnung [8]. Für die eigentliche Übertragung kann ebenfalls auf MPI aufgesetzt werden. Häufig aber werden optimierte Bibliotheken wie GasNet oder die Bibliotheken des Interconnects, wie bspw. `mx` für Myrinet oder `vapi` bei Infiniband, direkt benutzt.

1.3.3 Übersicht MetaComputing

Der Begriff MetaComputing wurde von Larry Smarr geprägt [126], dem damaligen Direktor des National Center for Supercomputing Applications (NCSA) der Universität von Illinois. MetaComputing beschreibt eine Technik zur Kopplung von Rechenressourcen, meist Höchstleistungsrechner wie in den Arbeiten von [40, 51, 112], zu einem virtuellen größeren Computer über ein verbindendes Netzwerk. Dabei kann man die Software zur Implementierung dieser Kopplung unterscheiden in:

- Software, die die Kopplung der Rechenressourcen nicht verbirgt. Hierbei muss die Anwendung explizit Rechnungen via Bibliotheksaufruf anstoßen. Ausführungsort, -parallelität sowie andere Modalitäten werden hierbei versteckt. Konzepte und Im-

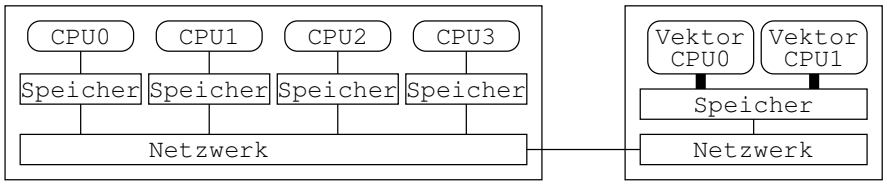

Abbildung 1.6: Kopplung verschiedener Rechnerarchitekturen im MetaComputing

plementierungen sind die Unix Remote Procedure Calls (RPC) [133], Java's Remote Method Invocation (RMI) [3] oder der Standard Common Object Request Broker (CORBA) [125]. Speziell für numerische Simulationen erlauben Bibliotheken wie das Mesh-based Code-Coupling Interface (MpCCI) [60] die automatische Abbildung unterschiedlicher Gitter, bspw. circulare Gitter auf recti-lineare Gitter, sowie die Interpolation der Werte auf den entsprechenden Gitterpunkten.

- Software, die diese Kopplung verbirgt. Diese ist meist als Softwarebibliothek implementiert, die zur Anwendung hinzugefügt wird. Meist versucht diese Bibliothek als mittlere Schicht (engl. Middleware) den Umstand der Kopplung und Eigenheiten des darunterliegenden Systems zu verbergen. Im Extremfall wird das System der Anwendung als ein einziges, gesamtes System (engl. Single System Image (SSI)) dargestellt. Dies erfordert von der Systemsoftware einigen Aufwand, um Netzwerkverbindungen weiterzureichen, Systemlast auf den Rechnern zu verteilen und ein gemeinsames Dateisystem anzubieten.

Die in dieser Arbeit verwendete Implementierung PACX-MPI gehört zum letzten Typ: Während die Anwendung die Kommunikation der ausführenden Prozesse explizit durch MPI-Kommunikationsaufrufe tätigt, versteckt PACX-MPI die Kopplung der Rechenressourcen. Wie später gezeigt werden wird, kann die Anwendung die Topologie der Rechenressourcen erfragen, um die Berechnungen besser verteilen oder die Kommunikation zwischen den Rechnern optimieren zu können. Anderenfalls bleiben unterschiedliche Rechnerarchitekturen, Ausführungsort sowie evtl. Unterschiede der darunterliegenden MPI-Implementierung verborgen.

Abbildung 1.6 zeigt, wie eine Anwendung einer gekoppelten Simulation auf geeignete Rechnerarchitekturen eines Metacomputers, hier ein paralleler Vektorrechner sowie ein Cluster, verteilt werden kann. Ersterer würde z. B. die gut vektorisierenden Schleifen des Computational Fluid Dynamics (CFD)-Programmteils ausführen; auf dem Cluster könnte der strukturmechanische Teil der Berechnung ausgeführt werden. Die Kommunikation würde für die Anwendung zwar transparent mit MPI-Aufrufen, intern aber mit den entsprechenden PACX-MPI Funktionen erfolgen.

1.3.4 Übersicht Peer-to-Peer Computing

Das Peer-to-Peer (P2P)-Konzept beschreibt die Kopplung unterschiedlicher Ressourcen über ein Netzwerk; laut der weit gefassten Definition von Clay Shirkey (Zitat aus dem Englischen, entnommen aus [38]):

> *Peer-to-peer beschreibt eine Klasse von Anwendungen, die Ressourcen der Endsysteme des Internets ausnutzen: Speicherplatz, Prozessorzeit, Dateninhalte und die involvierten Menschen. Weil der Zugriff auf diese dezentralisierten Ressourcen in einer stets veränderlichen Umgebung operiert, müssen Peer-to-Peer Systeme auf instabile Verbindungen und nicht-vorhersagbare IP-Adressen eingestellt sein, und damit auch ohne einen verlässlichen Domain Name Service (DNS) auskommen und eine erhebliche oder komplette Autonomie von zentralen Servern haben.*

In der Praxis haben sich zwei Klassen des P2P-Computing etabliert, so dass man unterschieden kann in:

- Einerseits P2P-Netze, bei denen die Endsysteme kleine Rechenaufgaben eines größeren Problembereichs abarbeiten. Beispiele hierfür sind trivial-parallele Rechenaufgaben, wie das Dechiffrieren von verschlüsselten Nachrichten oder die Faktorisierung großer Zahlen; hier ist eine möglichst große aggregierte Rechenleistung wichtig (engl. Number-Crunching).

- Eine weitere Anwendung ist das verteilte Speichern vieler und großer Dateien. Die Aufteilung in hunderte kleine Fragmente und die dezentrale, mehrfache Speicherung der Fragmente einer Datei sollte den Ausfall einzelner Teilnehmer des P2P-Netzwerkes verkraften.

Das prominenteste Projekt der erstgenannten Klasse ist sicherlich das Seti@Home-Projekt [146], welches seit 1999 mit ausgeklügelten Mechanismen nach intelligentem Leben im Weltall sucht. Radiosignale unterschiedlichster Frequenzen aus dem Weltraum werden gefiltert, einem PC aus dem Pool zugeordnet, auf welchem dann FFT-Filter ausgeführt werden, um Muster zu erkennen. Das riesige Spektrum des Frequenzbereichs kann damit trivial-parallel abgearbeitet werden.

Relevante frühe Projekte der zweitgenannten Klasse des P2P-Computing sind Napster und GNUtella [62]. Während Napster eine zentrale Datenbank für Dateien und damit einen Single-Point of Failure besitzt, ist das freie GNUtella-Projekt völlig dezentral organisiert. Dabei haben sich immense Skalierungsprobleme bei der Dateiensuche ergeben. Erfahrungen mit der GNUtella-Software wurden in neue, besser skalierbare Protokolle wie die Kademlia-Netze Azureus und BitTorrent [87] eingebracht.

Abbildung 1.7 zeigt beispielhaft einen P2P-Verbund mit vier Klienten und einem Dual-Prozessor Server. Als Anwendung für den Klienten wird meist eine plattformübergreifende Lösung für verschiedene Betriebssysteme im Hintergrund betrieben, bspw. als Bildschirmschoner. Sobald sich der Rechner im Leerlauf befindet, werden zu bearbeitende Abschnitte von einem Server heruntergeladen, im Hintergrund mit niedriger Priorität abgearbeitet und am Ende das Ergebnis zum Server wieder hochgeladen. Der Anwender bemerkt hiervon meist gar nichts. Sobald er wieder mit dem Rechner arbeitet wird dieser Klient unterbrochen.

Auf diese Weise lassen sich beliebig große, zeitunkritische Problemklassen berechnen. Die Probleme sollten bestenfalls trivial-parallel und gut aufteilbar für Rechner mit wenig Speicher sein. Das Programm, welches die eigentliche Berechnung durchführt, sollte

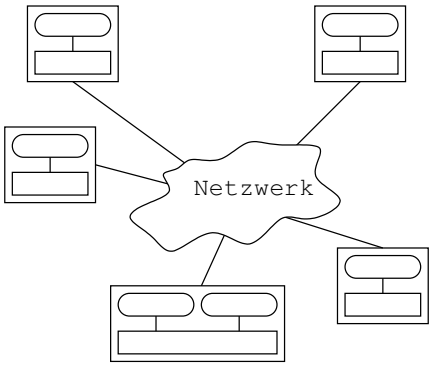

Abbildung 1.7: Peer-to-Peer Computing mit beliebig vielen PCs am Internet

plattform- und versionsunabhängig sein, um auf möglichst vielen Betriebssystemen und -versionen zu funktionieren. Das größte Problem hierbei ist die effiziente Speicherung und Suche innerhalb des P2P-Verbundes, vor allem die Skalierung auf eine große Anzahl von Klienten.

1.3.5 Übersicht Grid Computing

Der Begriff des Grid Computings wurde durch Ian Foster und Carl Kesselman in [46] geprägt. Es generalisiert die Benutzung von Rechen- und Datenressourcen [47, 56] soweit, sodass es theoretisch jedem Nutzer möglich sein sollte, diese Ressourcen so einfach für seine Anwendung oder Simulation zu verwenden, als würde er ein Gerät in das Stromnetz einstecken. Dies ist von der Idee her ähnlich dem P2P-Konzept aus Abschnitt 1.3.4, nur diesmal mit beliebigen und unter Umständen ohne Beschränkungen der Anwendungen im Vergleich zu vordefinierten Klienten-Programmen. Dabei ergeben sich eine Vielzahl von Fragen und Problemen, die es zu lösen gilt, unter anderem:

- Authentifizierung der Benutzer,
- Autorisierung der Benutzer für bestimmte (Rechen-) Ressourcen,
- Dynamische Ressourcen-Annoncierung und Registrierung der Rechner und installierten Software,
- Scheduling zur Verteilung der zu bearbeitenden Programme auf die Ressourcen,
- Dateitransfer von Eingabe- oder Ausgabedateien,
- Unterstützung interaktiver Programme,
- Fehlertoleranz und intelligente Fehlerbehandlung,
- Abrechnung der Kosten,
- Rechtevergabe und -einschränkung.

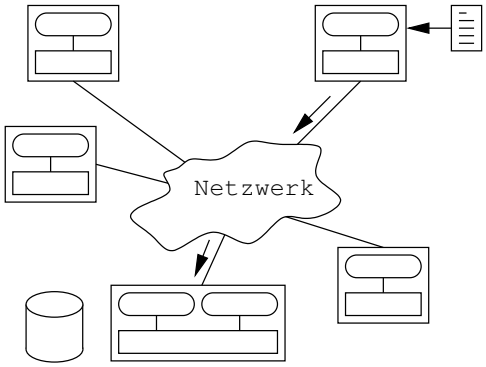

Abbildung 1.8: Submittierung eines Jobs auf Computerressourcen eines Grids

Es haben sich Gremien aus Forschung und Industrie zusammengefunden, um die oben genannten Probleme zu erörtern, plattformübergreifende Standards zu definieren und Prototypen als Referenz zu implementieren. Das wichtigste Forum, das Global Grid Forum (GGF) hat sich kürzlich umbenannt in das Open Grid Forum (OGF) [104]. In diesem Gremium finden regelmäßig Arbeitsgruppentreffen statt, um Standards für Gridlösungen zu erarbeiten. Erste Produkte von Grid-Implementierung von Avaki sind bereits am Markt [4].

Abbildung 1.8 demonstriert das Abschicken (engl. submitten) eines zu rechnenden Jobs auf einem der in diesem Grid verfügbaren Rechner. Die auszuführenden Befehle des Jobs werden in einer Jobbeschreibungssprache (engl. Job Description Language, kurz JDL) in einem Skript an den Rechner übergeben, der die zu verarbeitenden Befehle analysiert und an verfügbare Ressourcen verteilt. Hierbei müssen von dem Grid die oben aufgeführten Punkte abgearbeitet werden. Es gibt viel versprechende Ansätze, die die oben genannten Konzepte zumindest teilweise unterstützen. Die wichtigsten Vertreter sind das Unicore Projekt [115] und das Globus Toolkit [45–47] sowie das erwähnte Avaki [4].

Besonders die Koexistenz von bisherigen Zugangsmechanismen mit neuartigen institutionsübergreifenden Authentifizierungs- und Zertifizierungsstellen ist eine große rechtliche aber auch politische Hürde. Sind diese Hürden überwunden kann auch in allgemeinen Grids Metacomputing betrieben werden. Für Unicore wurde in der Dissertation von Lindner [89] erarbeitet, wie Anwendungen mit der hier bearbeiteten Bibliothek PACX-MPI verwaltet werden kann. Die Verteilung sowie Verwaltung der Anwendung kann der Benutzer von einem grafischen User Interface steuern.

Da aber die Problematik des Grid Computing besonders im Hinblick auf die spezielle Anforderungen paralleler Applikationen noch nicht gelöst sind, soll in dieser Arbeit nicht weiter auf generelle Probleme des Grids eingegangen werden. Vielmehr werden spezifische Lösungen des MetaComputing bearbeitet; wir beschränken uns also hier nur auf den Aspekt der verteilten Ausführung auf verteilten Systemen. Diese Systeme können unter Umständen heterogen sein, d. h. unterschiedliche Hardwarearchitektur, Systemsoftware und Kommunikationsbibliothek besitzen, was zusätzliche Anforderungen an die Middleware stellt.

1.4 Problemstellung und Ziele dieser Arbeit

Ob ein Höchstleistungsrechner für sich alleine oder als einzelne Ressource in einem Grid verwendet wird, für manche Anwendungen und Simulationen werden die Speicher- und/oder Rechenanforderungen mancher Simulationen die Kapazität eines heutigen Parallelrechners übersteigen. Oder die Anforderungen der Anwendung sind auf besondere Rechnerarchitekturen spezialisiert. Weiterhin kennt jeder Anwender das Problem, dass wenige Knoten auf mehreren Clustern zur Verfügung stehen, die gesamte Anzahl Prozessoren aber nicht auf einem einzigen Cluster. Die in dieser Abhandlung bearbeitete Bibliothek PACX-MPI (für **PA**rallel **C**omputer E**X**tension) kann die Probleme durch Kopplung zweier oder mehrerer Parallelrechner lösen helfen. Unter anderem sind die folgenden Szenarien denkbar:

1. Hierarchisch strukturierbare Anwendungen lassen sich auf gekoppelten Parallelrechnern effizient ausführen. Ein gutes Beispiel hierfür sind Client-Server-Anwendungen, die eine globale Operation aus mehreren lokalen Berechnungen zusammenführen. Die Anwendung `fastDNAml` welche in dem so genannten HPC-Challenge Wettbewerb der SuperComputing 2003 verwendet wurde, gehört zu dieser Klasse.

2. Besonders gekoppelte Simulationen, so genannte Multi-Physics Simulationen lassen sich durch PACX-MPI gut auf mehrere Rechner verteilen. Hierzu zählt bspw. die Berechnung der Umströmung eines Flugzeugflügels gekoppelt mit der Simulation der entsprechenden Auslenkung des Flügels durch die resultierenden Kräfte. Ersteres wird durch die so genannte CFD-Simulation berechnet, während letzteres mittels eines Strukturmechanikcodes, bspw. mit einer Finite Elemente Methode (FEM)-Simulation berechnet wird.

3. Sind spezielle Rechneranforderungen gegeben, wie bei der dezentralen Datenerfassung und der nachfolgenden -auswertung, kann die Verteilung auf Spezialrechner wie in der obigen gekoppelten Simulation durch PACX-MPI erfolgen.

4. Die performante Anbindung von HPC-Ressourcen an Visualisierungsrechner. Die Arbeit von Wenisch [145] hat sich mit der engen Kopplung von Visualisierung mit Simulation der Luftströmung in Operationssälen beschäftigt. Hierfür wurden Rechen- und Visualisierungsressourcen in München (Hitachi SR8k) und Erlangen (SGI) mittels PACX-MPI gekoppelt.

Das größte Problem bei der Koppelung der Rechenressourcen stellt jedoch der Aufwand der Kommunikation über das externe Netzwerk dar. Die rechner-interne Kommunikation ist um Faktoren schneller als die Kommunikation über das die Rechner verbindende Netzwerk, wie beispielsweise das Internet. Der Overhead externer Kommunikation wird auch in Zukunft mit größer werdender Diskrepanz zwischen Prozessor- und Netzwerkgeschwindigkeit immer dominanter. Um Anwendungen auf Metacomputern zu unterstützen, bzw. erst zu ermöglichen, muss der Overhead der externen Kommunikation drastisch reduziert werden.

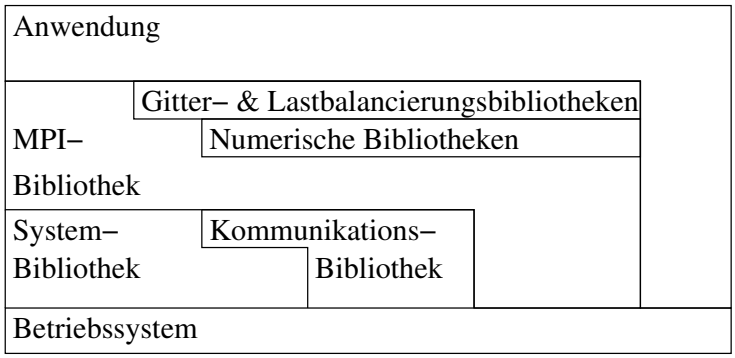

Abbildung 1.9: Softwareschichten von MPI-parallelen numerischen Anwendungen

In dieser Dissertation soll gezeigt werden, dass und unter welchen Vorraussetzungen MetaComputing Sinn macht. Es wird erläutert, mit welchen Methoden die Ausführung auf einem Metacomputer beschleunigt werden kann und wie diese Methoden für ähnlich Probleme angewendet werden können. Alle Softwareschichten, von der Netzwerkschicht, über die Algorithmen und Funktionalität der Middleware, bis hinauf zur wichtigsten Komponente, den eigentlichen Algorithmen in der Anwendung werden hierbei betrachtet. Die Vorgehensweise dieser Arbeit folgt dem Bottom-Up Approach, von der unteren zu den oberen Schichten der Softwarehierarchie, wie in Abb. 1.9 dargestellt. Einen Einblick in den Stand der Technik gibt Kapitel 2. In den folgenden Kapiteln werden die erarbeiteten Lösungen vorgestellt.
Im Einzelnen werden folgenden Probleme des MetaComputing behandelt:

1. Die Kommunikation zwischen den Rechnern des Metacomputers ist im Vergleich zur internen Netzwerkperformance sehr langsam. Im Damien Projekt wurden in MetaComputing-Tests die Höchstleistungsrechner des HLRS in Stuttgart und des Barcelona Supercomputing Centers (BSC) gekoppelt. Hierbei konnten auch über das europäische Hochgeschwindigkeitsnetzwerk Géant nur Latenzen zwischen 74 ms und 120 ms und über das Internet zwischen den IBM-RS6000 Rechnern in Spanien und an der Indiana University (IU), Bloomington nur Latenzen zwischen 166 ms und 180 ms erreicht werden [78]. Die interne Kommunikation erreicht Latenzen von 8 μs bis zu 1,2 μs. Für die Bandbreite gilt eine ähnliche Größenordnung. Während die Latenz durch die Entfernung, die Route und Signalausbreitungsgeschwindigkeit limitiert ist, kann die Bandbreite verbessert werden.

   In Kapitel 3 werden die Optimierungen der untersten Softwareschichten, der Kommunikationsbibliothek in Abb. 1.9, vorgestellt. Einerseits wird die Kommunikation in PACX-MPI durch parallele Übertragung durch mehrere Threads beschleunigt. Andererseits wird ein neuartiges Protokoll auf Basis von UDP-Paketen vorgestellt. Dieses bietet für die Kommunikation auf Weitverkehrsnetzen (WAN) hervorragende Bandbreite, und in lokalen Netzwerken (LANs) eine mit TCP vergleichbare Leistung.

2. Der MPI-Standard bietet eine Vielzahl von kollektiven Algorithmen zum Verteilen

und Sammeln von Daten, wie bspw. `MPI_Bcast` als $1 \times N$-Kommunikation[3]. Für die Ausführung in einem heterogenen Metacomputer muss auch die Middleware, hier die PACX-MPI Schicht, die Topologie optimal ausnutzen. Neben der Verbesserung der Übertragungsleistung des Netzwerks ist somit die Optimierung der kollektiven Algorithmen ein großes Problem.

In Abschnitt 4.1 werden die Optimierungen der Kommunikationsalgorithmen für kollektive Operationen am Beispiel der $N \times N$-Funktion `PACX_Alltoall` vorgestellt. Anhand des Benchmarks wird eine bis zu 100-fach schnellere Ausführung gegenüber dem alten Algorithmus erzielt.

3. Die besondere Kommunikationscharakteristik eines Metacomputers wurde bereits verdeutlicht. Für den Programmierer einer Anwendung ist somit das Kommunikationsmuster im Metacomputer von besonderem Interesse. Für parallele Anwendungen gibt es hierzu mehrere Tools zur Performanceanalyse.

   In Abschnitt 4.2 wird die Erweiterung einer Performance-Tracing Bibliothek für das Analysetool Paraver [5] beschrieben. Die `pacxtrace` Bibliothek erlaubt die Visualisierung der Kommunikation einer Anwendung im Metacomputer unter Ausblendung der beiden Dämonen von PACX-MPI. Damit wird ineffiziente Kommunikation über das externe Netzwerk ersichtlich.

4. Mittels der oben beschriebenen Performanceanalysetools lässt sich basierend auf Anfangs- und Endzeiten die Kommunikationsgeschwindigkeit einzelner Transfers ausrechnen. Die MPI-Bibliothek ist aber frei, wie der eigentliche Transfer durchgeführt wird, d. h. es gibt mehrere Implementierungsalternativen, wie auf Seite 25 dargestellt.

   In Abschnitt 4.3 wird erläutert, wie sich MPI-interne Zustände auslesen lassen. Hierfür wird die Spezifikation der Peruse-Schnittstelle [73] vorgestellt, an welcher der Autor beteiligt war. Ausserdem wird die neuartige Implementierung von Peruse in PACX-MPI, bzw. Open MPI vorgestellt [76], sowie beschrieben, wie Performance-Probleme der Anwendung aber auch Flaschenhälse der MPI-Implementierung aufgezeigt werden können. Dies wird anhand der Analyse der Anwendung `IMD` [116] gezeigt. Weiterhin wird erläutert, welche Informationen Peruse im Vergleich zum Standard-PMPI Interface bietet.

5. Die verzögerte Kommunikation bei verteilten Anwendungen kann zur Veränderung der Ausführungsreihenfolge führen. Fehler bei der Programmierung einer Anwendung, bspw. bei nicht-deterministischen MPI-Funktionalitäten wie `MPI_ANY_SOURCE` oder den Funktionen `MPI_Waitsome`, werden dabei schnell der Middleware zugeschrieben.

   Um Bugs in der Middleware weitestgehend auszumerzen wurde speziell für PACX-MPI eine neue Testsuite entwickelt [83, 123]. Diese zeichnet sich durch leichte

[3] Das heisst 1 Prozess sendet Daten zu N empfangenden Prozessen.

Erweiterbarkeit, flexible Selektion und durch die Kombination verschiedener Test-Parameter als besonders leistungsfähig aus. Das ermöglicht eine Kombination von derzeit 7616 verschiedener Tests innerhalb einer Anwendung.

6. Die wichtigste Komponente zur Reduktion der Laufzeit im MetaComputing ist der Algorithmus der Anwendung und die ihm inhärente Kommunikation. Wenn diese stark kommunizieren, werden die PACX Dämonen bzw. das verbindende Netzwerk zur Bottleneck.

   In Kapitel 5 werden Verbesserungen an Anwendungen vorgestellt, um MetaComputing zu ermöglichen. In Abschnitt 5.1.2 werden die Optimierungen innerhalb der Bioinformatik-Anwendung `RNAfold` vorgestellt. Für diese Anwendung wurde eine neuartige Kommunikationsoptimierung durch Caching und Prefetching implementiert. Erst diese Optimierungen erlauben die effiziente Ausführung im Metacomputer. Es lässt sich zeigen, wie ein bei Skalierung in schwacher Form (Weak Scaling) dominierender Algorithmus durch das Caching und Prefetching auch auf einem Cluster mit schnellem Interconnect (IB) um das 13-fache reduziert werden kann.

2 Stand der Technik

Dieses Kapitel gibt einen Überblick über den derzeitigen Stand der Technik von Hard- und Software für MetaComputing. Es werden die grundlegenden Begriffe und Konzepte der Netzwerktechnik eingeführt und darauf aufbauend für MetaComputing relevante Kommunikationshardware vorgestellt. Zudem werden die entwickelten Kommunikationsbibliotheken, wie Stampi, MPIch-G2, PACX-MPI und Open MPI aufgeführt und erläutert.

2.1 Grundlagen der Netzwerktechnik

Um auf die im MetaComputing verwendeten Begriffe näher eingehen zu können, werden zur Einführung die Grundlagen der Technik vorgestellt. Hierfür eignet sich das ISO/OSI Schichtenmodell besonders, da es die zur Kommunikation notwendigen Software- und Hardwareschichten klar strukturiert und damit, obwohl technisch nicht mehr relevant, gut zur anschaulichen Gliederung von den involvierten Schichten dienen kann. Darauf aufbauend werden Kommunikationsprotokolle im Allgemeinen betrachtet und Charakteristiken zusammengestellt, nach denen allgemeine Protokolle eingegliedert werden können. Diese Charakteristiken werden auf Transport Control Protocol (TCP) und User Datagram Protocol (UDP) angewandt und schließlich die Semantik aus Anwendersicht von TCP und der Punkt-zu-Punkt Kommunikation von MPI diskutiert.

2.1.1 Klassifikation von Protokollen

Nachdem erkannt wurde, dass mit den ersten verfügbaren, zueinander inkompatiblen Rechnernetzwerken ein immenser Aufwand für Interoperabilität getrieben werden musste, hat das Internationale Standardisierungsgremium ISO die Kommunikation zwischen Computern definiert, auch als OSI-Referenzmodell bekannt ist (engl. Open System Interconnect). Darin wird die Kommunikation über Netzwerke in sieben Schichten modelliert. Der Austausch der Daten auf gleicher Ebene wird durch ein Protokoll je Ebene definiert, während der Austausch zwischen den Schichten durch Schnittstellen definiert ist. Jede der Schichten hat eine spezifische Aufgabe, bspw. die Transportschicht zum Aufbau und Erhalt einer Ende-zu-Ende Kommunikation, siehe Abb. 2.1.

Obwohl die US-Regierung einmal verlangte, dass alle für staatlichen Behörden beschafften Rechner das ISO/OSI-Modell erfüllen sollten, gewann es doch nie außerhalb von Forschung und Lehre an Bedeutung, da es zu komplex und unpraktisch aufgebaut ist. Durch die Entwicklung und anschließende rasante Verbreitung der Protokolle TCP/IP

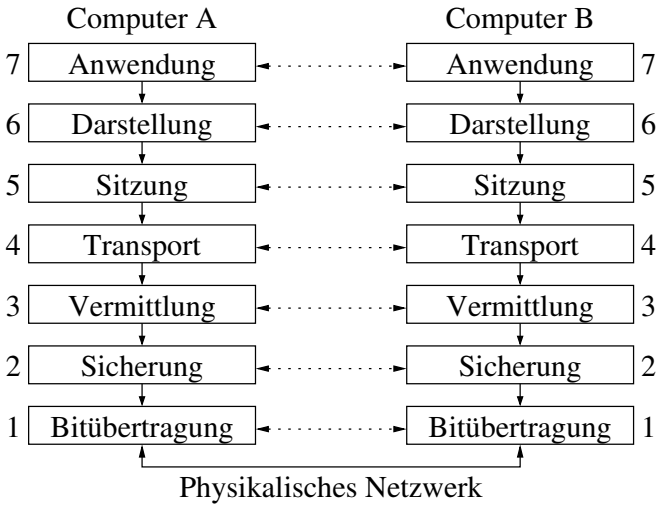

Abbildung 2.1: Das ISO/OSI 7-Schichtenmodell zur Rechnerkommunikation

wurde der ISO/OSI-Standard überholt. Dennoch bieten sich das darin entwickelte 7-Schichtenmodell zur Veranschaulichung von Netzwerken an, da Übertragungsprotokolle anhand des ISO/OSI-Modells eingegliedert werden können, bspw. HTTP zur Übertragung von Webinhalten in Schicht 5-7, sowie TCP in Schicht 4, oder das Internet Protocol (IP) in Schicht 3.

Will man Kommunikationsprotokolle beliebiger Netzwerke gliedern, gibt es mehrere Charakteristiken, nach denen man aufschlüsseln kann. Da in dieser Arbeit Kommunikation in Clustern und im MetaComputing behandelt wird, werden Beispiele für klassische Netzwerkprotokolle basierend auf dem Internet Protocol (IP) besprochen. Der MPI-Standard selbst enthält keine Protokollspezifikation für die Übertragung von Daten, alle Modalitäten der Übertragung sind Teil der MPI-Implementierung, dennoch werden die Klassifikationen besprochen, weil dies Gegenstand einer MPI-Implementierung werden muss.
Protokolle auf der Softwareebene können unterschieden werden in folgende Charakteristiken:

Art der Verbindungsroute und Verbindungsaufbau

- Verbindungsorientierte Protokolle, wie bspw. analoge Telefonleitungen: hier muss die Verbindung von den Partnern explizit aufgebaut werden, es wird eine Route auf allen Knotenpunkten zwischen den Endgeräten hergestellt. Aus dem Netzwerkbereich auf Softwareebene ist das Transport Control Protocol (TCP) ein gutes Beispiel: Die Verbindung wird in drei Schritten aufgebaut (engl. three-way Handshake), wie in Abschnitt 2.1.2 genauer erklärt.

- Verbindungslose Protokolle, wie bspw. die klassische Briefpost: hier werden an jeder Station der Route zum Empfänger Entscheidungen über die nächste Zwischenstation und zuletzt beim Briefträger getroffen. Aus dem Netzwerkbereich ist

UDP ein gutes Beispiel: das User Datagram Protocol (UDP) basiert wie TCP auf IP, setzt aber keinen Verbindungsaufbau voraus, sondern Server warten auf eingehende Verbindungspakete. Je nach Anwendungsprotokoll wird der Server mit einem Bestätigungspaket (Acknowledgement, oder kurz ACK) antworten, für die UDP-Schicht aber ist dies transparent.

Granularität der zu verschickenden Daten

- Byteorientierte Protokolle: es können eine beliebige Anzahl von Bytes vom Sender geschrieben und (in derselben Reihenfolge) vom Empfänger byteweise gelesen werden. Dies entspricht TCP.
- Paketorientierte Protokolle: die Daten können nur paketweise versandt und empfangen werden. Hierfür gibt es wiederum zwei Unterscheidungen:
  - Geordneter Empfang: die Daten kommen in der Reihenfolge beim Empfänger an, in der sie vom Sender abgeschickt wurden.
  - Ungeordneter Empfang: die Daten könnten evtl. auf dem Übertragungsweg vertauscht werden (bspw. weil sie eine andere Route im Netzwerk nehmen) und damit in einer anderen Reihenfolge beim Empfänger eintreffen. Ist die Reihenfolge für Nutzer des Protokolls notwendig, so müssen die Datenpakete durch eine eindeutige Sequenznummer durch die Applikation gekennzeichnet und beim Empfänger umsortiert werden. Auch hierfür ist UDP ein gutes Beispiel.

Erkennung und Abfangen von Datenverlusten

- Verlustfreie Protokolle: Die zu versenden Daten kommen ohne Verluste oder Änderungen von Daten beim Empfänger an. Beim TCP-Protokoll werden verloren gegangene IP-Pakete erkannt und spätestens nach einem Timeout und dem fehlenden Acknowledgement wieder versandt.
- Verlustbehaftete Protokolle: Es können Daten aufgrund von Überlastung des Netzwerkes, bspw. von Switches auf der Route, oder Ausfall der physikalischen Verbindung verloren gehen und werden auch nicht vom Protokoll abgefangen. Hierbei muss also die versendende Anwendung den Verlust oder die Veränderung von Daten bemerken und einen Neuversand anstoßen oder aber mit Verlusten zurecht kommen.

Integrierte Flusskontrolle

- Ein Beispiel für Protokolle mit Flusskontrolle ist wieder TCP. Ein Sender sollte Engpässe bei der Übertragung erkennen (engl. Congestion Control) und die Senderate entsprechend limitieren. Da bei TCP typischerweise mehrere Knoten auf der Route zum Empfänger sind, wurde hierfür kein separates Protokoll oder Erkennungspakete definiert, sondern der Engpass durch verlorene IP-Fragmente

erkannt. Bei den meisten MPI-Implementierungen herrscht ein einstufiges Netzwerk vor; hierfür werden auf tiefer liegenden Protokollen volle Pufferzustände an den Sender weitergereicht.

- Ein Beispiel für ein Protokoll ohne Flusskontrolle ist wieder UDP basierend auf IP. Häufig werden UDP-basierte Protokolle an der Fairness gemessen, wie sie konkurrierende TCP-Verbindungen bremsen. Theoretisch sollten alle bestehenden Verbindungen einen gleichen Anteil an der Übertragungskapazität einer Leitung erhalten.

Quality-of-Service und Echtzeitanforderungen

Eine immer wichtigere Eigenschaft von Übertragungsprotokollen ist die garantierte Einhaltung bestimmter Übertragungskriterien. Die Qualität von Multi-Media Anwendungen und hierbei die Audioübertragung leidet stark bei hohen Latenzen und besonders bei großen Varianzen der Latenz. Für High-Performance Computing (HPC) ist Quality-of-Service (QoS) eher nebensächlich – weiterhin wird es für Weitverkehrsnetze nicht durchgehend unterstützt und wenn, müssen die Eigenschaften manuell vom Netzadministrator an den Eingangsroutern (engl. Ingress-Routern) konfiguriert werden. Beispielsweise wurde in Géant die Unterstützung von QoS mittels eines dreistufigen Ansatzes in Service-Klassen „Best Effort“, “Premium IP“ und „Less than Best Effort“, alle ohne konkrete Performancezusagen unterteilt.

Derzeit wird allerdings in der IEEE 802.3ar-Arbeitsgruppe „Congestion Management Task Force“ ein Verfahren entwickelt, um zwischen Routern ein Protokoll zur Pufferüberlaufwarnung bereitzustellen. Dieses sollte bis September 2007 zur Standardisierung vorgelegt werden.

Darstellung der zu versendenden Daten

- Kein Wissen über die Semantik: Verbindungen basierend auf dem TCP Protokoll müssen die Datenkonvertierung durch den Aufrufer erledigen. TCP selbst überträgt nur einen Byte-granularen Datenstrom.

- Semantikbehaftete Protokolle: Für MPI müssen die Datentypen der zu versendenden Daten angegeben werden. Intern regelt die MPI-Implementierung die Konvertierung der Daten für den Zielrechner. Ob diese Konvertierung senderseitig oder beim Empfänger erfolgt, ist durch den MPI-Standard nicht definiert, sondern implementierungsabhängig.

Die wenigsten Protokolle haben Wissen über die Darstellung der zu versendenden Daten. Die Semantik und Übersetzung der Daten ist Aufgabe der aufrufenden Schicht. Eine häufig angeführte Kritik an MPI und dem vermeintlichen Overhead der Konvertierung soll hier diskutiert werden: Die in MPI definierte Typisierung von Puffern ist zwingend notwendig, um hohe Leistung und portable Anwendungen zu erlauben. Dafür ist es aber notwendig, Semantik einzuführen. Im homogenen Fall[1], der im HPC meist vorherrscht,

[1] Die meisten Programme laufen auf Clustern mit Prozessoren der gleichen Architektur.

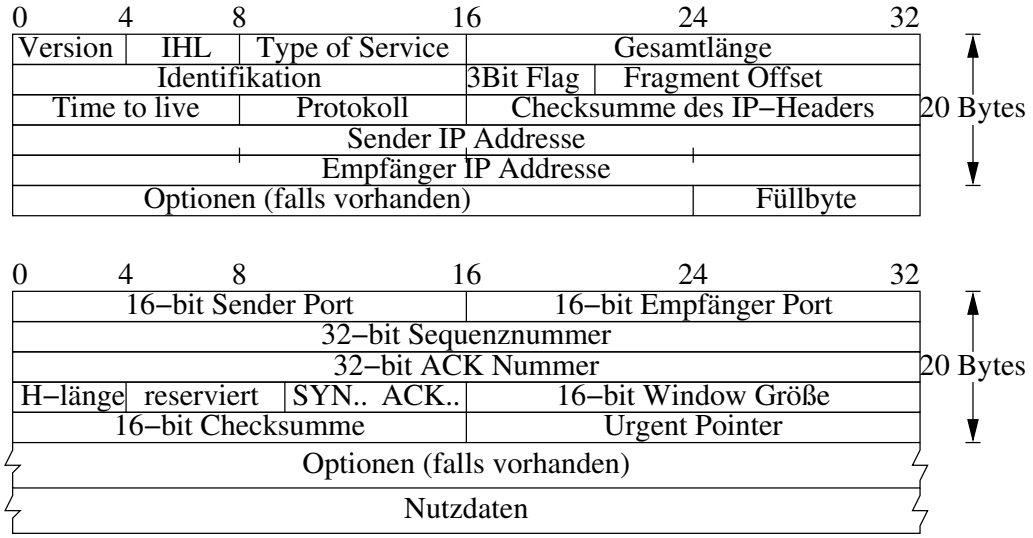

Abbildung 2.2: Header von IP (oben) und TCP (unten)

ist Konvertierung nicht notwendig und in optimierten MPI-Implementierungen eine Null-Operation. Im anderen Fall können für den Prozessor optimierte Algorithmen die (meist großen) Daten durch Software oder während der Übertragung (engl. On-the-Fly) durch die Hardware konvertieren.

Eine Applikation wiederum müsste für diese Konvertierungen im heterogenen Fall für verschiedenste Architekturen optimierte Konvertierungsroutinen anbieten.

2.1.2 Klassifikation der Kommunikationsmodelle

Semantik vom Transport Control Protocol (TCP) aus Anwendungssicht

Das Transport Control Protocol[2] basiert auf dem Internet Protocol (IP) und ist ein verbindungsorientiertes, fehlerfreies, zuverlässiges Protokoll für bidirektionale Byteströme. Abbildung 2.2 zeigt den TCP-Header, der das nachfolgende Datenpaket beschreibt. Jede Verbindung in TCP/IP wird über das 5-Tupel (Protokoll, IP-Adresse des Absender, Port des Absenders, IP-Adresse des Empfängers, Port des Empfängers) eindeutig bestimmt. Die Port-Nummer wird im TCP-Header eingetragen – es gibt eine Liste von Ports, die bestimmten Diensten zugewiesen sind, beispielsweise Port 80, welcher von Web-Servern belegt wird. Es ist zu beachten, dass Ports unter der Nummer 1024 nur von Programmen benutzt werden dürfen, die mit Superuser-Rechten laufen. Die nachfolgenden Daten im Paket werden durch die 32-Bit Sequenznummer identifiziert. Das Feld Headerlänge (in Abb. 2.2 als „H-Länge“ abgekürzt) beschreibt die gesamte Länge des TCP-Headers. Werden dem TCP-Header Optionen angehängt, muss dies in der Headerlänge angepasst werden.

Die Daten einer Webseite werden vom Webserver über eine bestehende Verbindung

[2]Standardisiert in Request for Comments (RFC) 793 „Transmission Control Protocol“.

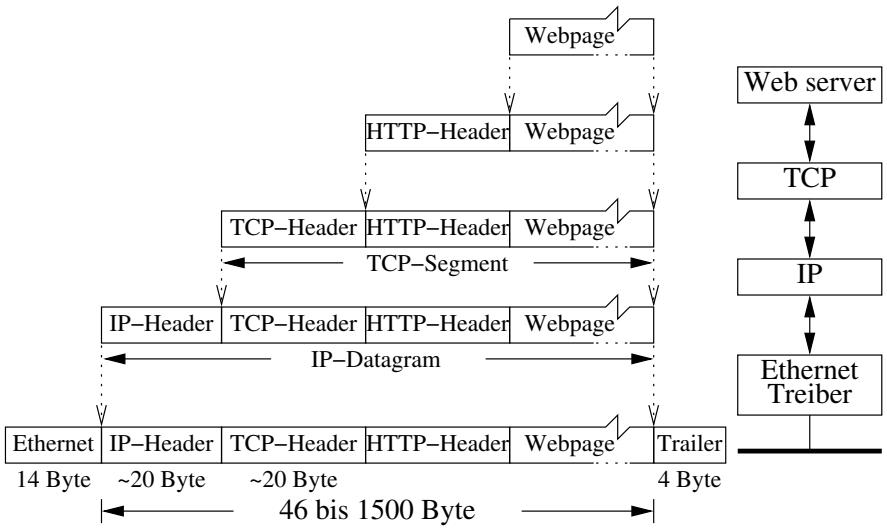

Abbildung 2.3: Kapselung der Daten einer Webseite in TCP/IP und Ethernet-Frames

mit TCP versandt. Damit werden die Nutzerdaten im TCP-Paket gekapselt und diese dann wiederum im IP-Paket verpackt. Bei Versendung über Ethernet mit Standard-Frames von 1500 Bytes für Daten werden die gesamten Daten ein weiteres Mal durch einen 14 Byte großen Header und einen 4 Byte großen Footer gekapselt, wie in Abb. 2.3 zu sehen.

Der große Erfolg von TCP/IP basiert auch auf der Tatsache, dass eine Anwendung plattformübergreifend Netzwerkverbindungen aufbauen kann und Fehler wie Netzwerkausfälle auf der Route, verloren gegangene, doppelte und überholende Pakete durch TCP erkannt und für die Anwendung transparent korrigiert werden. Weiterhin bietet TCP einen konservativen, aber fairen Algorithmus, um eine Überlastung des Netzwerkes zu erkennen (engl. Congestion Control) und fair die Bandbreite mit konkurrierenden TCP-Streams zu teilen. TCP hat sich besonders in diesen letztgenannten Aspekten über verschiedene Versionen (Tahoe, Reno, Vegas und neuerdings Westwood)[3] verbessert [98].

Eine TCP-Verbindung zwischen zwei Rechnern wird durch einen dreifachen Paketaustausch (engl. three-way Handshake) zwischen zwei Rechnern aufgebaut, wie in Abb. 2.4 dargestellt. Damit sich andere Rechner verbinden können, ruft der von hier an Server genannte Rechner die Funktionen `socket`, `bind`, `listen` und `accept` („passive open“) auf, während der sich an den Server verbindende Client die Funktionen `socket`, evtl. `bind` und schließlich `connect` aufruft („active open“). Hierbei ist `i` eine initiale Sequenznummer vom Client und `j` eine initiale Nummer vom Server zur eindeutigen Identifizierung der TCP-Verbindung. Pakete werden vom Empfänger nach einer gewissen Zeit als korrekt erhalten bestätigt (engl. Acknowledgement oder kurz Ack) – diese Zeit richtet sich danach, wie lange ein Paket für die Übertragung vom Sender zum Empfänger und zurück im Durchschnitt braucht (engl. Round Trip Time (RTT)). Es wird immer das zuletzt empfangene Paket bestätigt: damit sind implizit alle vorherigen Pakete ebenfalls

[3] Diese Städte liegen alle an der Westküste der USA angrenzend zur Wüste Nevadas.

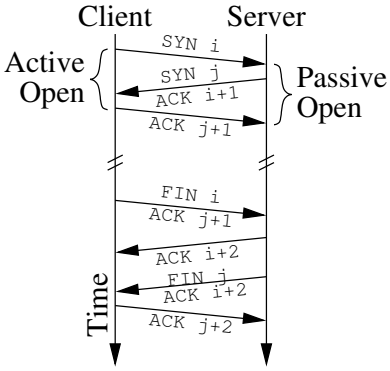

Abbildung 2.4: Auf- und Abbau von TCP-Verbindungen

bestätigt. Somit können unnötig häufige Acks vermieden werden und bei einer Kommunikation in beide Richtungen das Ack zusammen mit Daten des Empfängers zum Sender versandt werden (engl. Piggyback).

Sender und Empfänger handeln beim Verbindungsaufbau eine Puffergröße aus, die angibt, wie viele Daten der Sender maximal zu einem Zeitpunkt an den Empfänger schicken darf, um den Empfänger nicht zu überlasten (engl. Congestion Window). Es werden also nur eine bestimmte Anzahl an Paketen versandt und protokolliert. Die Menge der versandten, aber noch nicht bestätigten Pakete schiebt sich wie ein Fenster über alle zu versendenden Daten (engl. Sliding Window). Bei einer Verbindung mit entsprechend vielen Routern und sehr großer Latenz (engl. long, fat Pipes) muss dieses Sliding Window sehr groß werden. Da das ursprüngliche TCP für heutige Netzwerke nicht genügend Bits für die Kodierung dieser Größe bereitgestellt hat, wurde eine Erweiterung zur Vervielfachung der Fenstergröße[4] definiert.

Semantik der Punkt-zu-Punkt Operationen von MPI aus Anwendungssicht

Eine MPI-parallele Anwendung wird durch das Kommando `mpirun` gestartet. Dabei wird angegeben wie viele parallele Instanzen, Prozesse genannt, die Anwendung ausführen.

Kommunikation in MPI erfolgt immer innerhalb eines so genannten Kommunikators. Dieser ist in einer Anwendung global definiert und beschreibt den Kontext der Kommunikation: eine Gruppe der gestarteten Prozesse und weitere Attribute. Zu Beginn des Programms haben alle gestarteten Prozesse einen Rang im vordefinierten Kommunikator `MPI_COMM_WORLD`. Punkt-zu-Punkt-Kommunikation (P2P) wird in MPI durch den Nachrichtendeskriptor (engl. Message Envelope) identifiziert, einen 4-Tupel aus (Sender, Empfänger, Tag, Kommunikator). Sender und Empfänger sind eine positive Ganzzahl und stehen für den Rang des Prozesses an den gesandt, bzw. von dem empfangen wird, innerhalb des Kommunikators. Das Tag wiederum ist eine positive Ganzzahl mit dem Wertebereich $0 \leq$ Tag $\leq$ `MPI_TAG_UB`, wobei diese obere Schranke implementierungsabhängig ist (mindestens jedoch 32767).
Der Programmierer hat die Auswahl zwischen vier Sendemodi:

[4]Standardisiert in Request for Comments (RFC) 1323 „TCP Extensions for High Performance“.

- Dem so genannten „Normalen Sendemodus“ (`MPI_Send`), der je nach Implementierung und Nachrichtengröße entweder als „Buffered Send“ unter Verwendung eines Puffers der Implementierung, oder als “Synchronous Send“ erfolgen kann.

- Beim „Buffered Send“ (`MPI_Bsend`) wird die Nachricht in einen vom Programmierer zur Verfügung gestellten Speicherbereich kopiert. Dieser muss die größte zu sendende Nachricht[5] fassen können.

- Im „Synchronous Send“ (`MPI_Ssend`) Modus wartet der Sender auf den vom Empfänger zu startenden Aufruf. Ist dieser eingegangen, kann die Nachricht direkt in den Empfangspuffer ohne weitere Kopieroperationen transferiert werden.

- Beim „Ready Send“ (`MPI_Rsend`) kann die MPI-Implementierung davon ausgehen, dass ein passender Empfangsaufruf schon eingegangen ist. Die Nachricht kann somit von der MPI-Implementierung komplett in den Empfangspuffer gesandt werden, ohne zwischenzupuffern und ohne auf die Bestätigung des Empfangsaufrufes zu warten. Der Programmierer hat sicherzustellen, dass diese Annahme immer korrekt ist.

Bei Rücksprung aus dem `MPI_Send`, `MPI_Bsend`, `MPI_Ssend` oder `MPI_Rsend` kann der Programmierer sicher sein, dass die Nachricht korrekt versandt wurde. Verwendet der Programmierer den normalen Sendemodus, ist nicht definiert, ob die Implementierung für die gegebene Nachrichtengröße intern puffert oder den synchronen Sendemodus verwendet. Dies erfordert, dass der Entwickler für portable Programmierung keine Annahmen über die Implementierung macht und für korrekte Programme immer ein passender Empfangsaufruf abgesetzt werden muss.

Während der Sender bei Punkt-zu-Punkt Kommunikation (kurz P2P) den Rang des Empfängers explizit angeben muss, kann der Empfänger das Argument `MPI_ANY_SOURCE` und/oder `MPI_ANY_TAG` angeben, d. h. dass Nachrichten von beliebigen Sendern, bzw. mit beliebigen Tags akzeptiert werden. Von der Empfängerseite betrachtet, vergleicht die Implementierung den Nachrichtendeskriptor der eingegangenen Nachrichten auf Übereinstimmung zu den von der Anwendung abgesetzten Empfangsaufrufen (im Englischen kurz „Matching“ genannt). Bei Übereinstimmung werden die Daten in den Empfangspuffer der Anwendung geschrieben, anderenfalls ist implementierungsabhängig, wie weiter verfahren wird. Für diesen Fall ist das von der Implementierung verwendete Protokoll relevant:

- Beim Short- oder Eager-Message Protokoll wird die Nachricht an den Empfänger versandt, unabhängig davon, ob der passende Empfangsaufruf abgesetzt wurde. Insofern muss die Nachricht durch die MPI-Implementierung zwischengespeichert werden.

- Beim Long Message-Protokoll werden die Daten erst nachdem der passende Empfangsaufruf abgesetzt wurde, versandt. Meist wird eine initiale Nachricht wie im

[5] Plus einen implementierungsabhängigen Overhead der Größe `MPI_BSEND_OVERHEAD`.

Eager-Protokoll versandt. Bei Bestätigung des Empfangsaufrufes werden die restlichen Daten evtl. direkt in den Anwendungspuffer empfangen.

- Bei empfängergesteuerten Protokollen wird die Kommunikation durch den Empfänger initiiert. Dies erlaubt direktes Empfangen in den Anwendungsspeicher, erschwert aber die Implementierung für den `MPI_ANY_SOURCE`-Fall.

MPI definiert, dass zwei nacheinander versandte Nachrichten mit gleichem 4-Tupel, in korrekter Reihenfolge empfangen werden. Intern muss die MPI-Implementierung also durch Zähler im Nachrichtendeskriptor sicherstellen, dass die Daten korrekt, einmalig und in der richtigen Reihenfolge empfangen werden[6].

2.2 Netzwerktechnik für MetaComputing

In diesem Abschnitt werden Adapter für Cluster-Netzwerke im Hinblick auf MetaComputing vorgestellt. Wie in Abschnitt 1.2 erwähnt, sind die Übertragungscharakteristiken des die Cluster verbindenden Netzwerks entscheidend. Wichtig für High-Performance Computing (HPC) und ebenso für MetaComputing sind hierbei die Latenz, d. h. die Zeit für die Übertragung eines Null-Byte Datenpaketes und die Bandbreite, d. h. die Übertragungsrate angegeben in Gigabit pro Sekunde ($^{\mathrm{Gb}}/_{\mathrm{s}}$) oder Megabyte pro Sekunde ($^{\mathrm{MB}}/_{\mathrm{s}}$).

Die Laufzeit D einer Nachricht ist

$$D = d/v$$

mit d der Kabellänge und v der Ausbreitungsgeschwindigkeit auf dem Übertragungsmedium (Kupferkabel: 2,3*10^8 $^{\mathrm{m}}/_{\mathrm{s}}$, Glasfaserkabel: 2*$10^8$ $^{\mathrm{m}}/_{\mathrm{s}}$). Die Übertragungsdauer einer N-Byte Nachricht ergibt sich dann als

$$T_G = D + T_s + T_w$$

mit T_s der Sendedauer für das N-Byte Paket und T_w der Summe der Wartezeiten in den Routern. Die Sendedauer ist also

$$T_s = N/BW$$

mit BW als der nominellen Bandbreite in Byte/s für das Netzwerk. Die Übertragungsdauer pro Byte ist $T_B = 1/BW$. Die Anzahl Bytes, die sich in einem Netzwerk auf dem Weg vom Sender zum Empfänger befinden, ergibt sich durch:

$$\text{Bytes on Flight} = D/T_B = D\,BW$$

Anstatt als Anwendung (oder als MPI-Implementierung) eine Nachricht der Länge N-Bytes als eine Entität zu sehen, kann man das Pipelining-Konzept anwenden. Dabei

[6] Bspw. wenn UDP als darunterliegende Kommunikationsschicht verwendet wird.

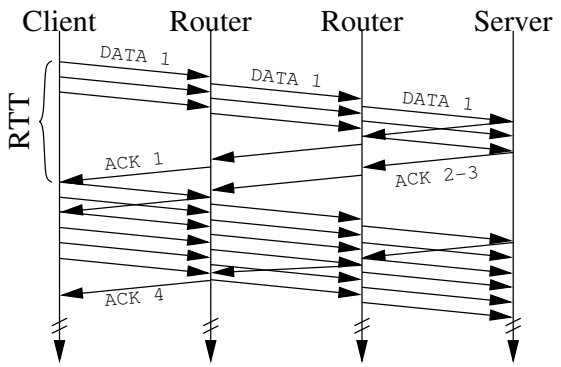

Abbildung 2.5: Pipelining bei der Netzwerkübertragung

wird die Nachricht in kleinere Fragmente aufgeteilt und diese versandt, bspw. entsprechend der Paketgröße der physikalischen Übertragungsschicht. Vereinfacht dargestellt zeigt Abbildung 2.5 wie in rascher Folge zuerst drei Datenpakete vom Sender an den Empfänger geschickt werden. Bei Empfang des ersten Acks als Bestätigung des ersten Paketes kann der Sender zwei weitere Datenpakete schicken; für das zweite Ack als Bestätigung der folgenden zwei Pakete schickt der Sender vier Pakete. So können die Puffer auf dem Weg zum Empfänger voll gehalten und die Wartezeit bis zum Eintreffen der Acknowledgments verkürzt werden. Zu bemerken ist, dass jeder Router auf dem Weg zusätzlich zur Übertragungszeit jedes Paketes die Weiterleitung verzögert (mit T_w der Summe der Wartezeiten). Da die Datenrate nicht beliebig wachsen kann, besitzt TCP eine integrierte Flusskontrolle (engl. Congestion Control) zur Erkennung von Engpässen auf Routern, siehe auf Seite 21.

Bei Netzwerkverbindungen mit großem Bandbreite×Latenz-Produkt spricht man von engl. Long-Fat Pipes [132, S. 344]. In Weitverkehrsnetzen und auf interkontinentalen Verbindungen hat TCP Probleme eine derartige Pipeline zu füllen. Eine Lösung für dieses Problem wird im Abschnitt 3.2 erarbeitet.

Wichtige Informationen können aus den Bandbreitenmessungen gewonnen werden, wenn man das Verhältnis aus tatsächlicher Bandbreite zu den aus Latenz T_{0-Byte} bei 0 Byte Nachrichtenlänge und Bandbreite $BW_{\infty-Bytes}$ mit sehr großer Nachrichtenlänge bildet. Theoretisch wäre damit für die Übertragung einer N-Byte großen Nachricht $T_{N-Bytes}$ Zeit notwendig:

$$T_{N-Bytes} = T_{0-Byte} + \frac{N}{BW_{\infty-Bytes}}$$

Tatsächlich wird aber bei allen Netzwerken eine langsamere Übertragungszeit gemessen. Der Hersteller Qlogic hat in [31] verschiedene Cluster-Netzwerke untersucht und das Verhältnis der realen zur theoretischen Übertragungszeit gesetzt. Es wurde festgestellt, dass auch die unten besprochenen Netzwerke Myrinet und Infiniband bis zu einem Faktor zwei langsamer als theoretisch berechnet sind. Abbildung 2.6 zeigt die Messungen von Qlogic [31]. Es wird deutlich, dass bei mittleren Nachrichtenlängen der Overhead der Hardware und des Protokolls die Übertragung bremst: teilweise reduziert sich die Band-

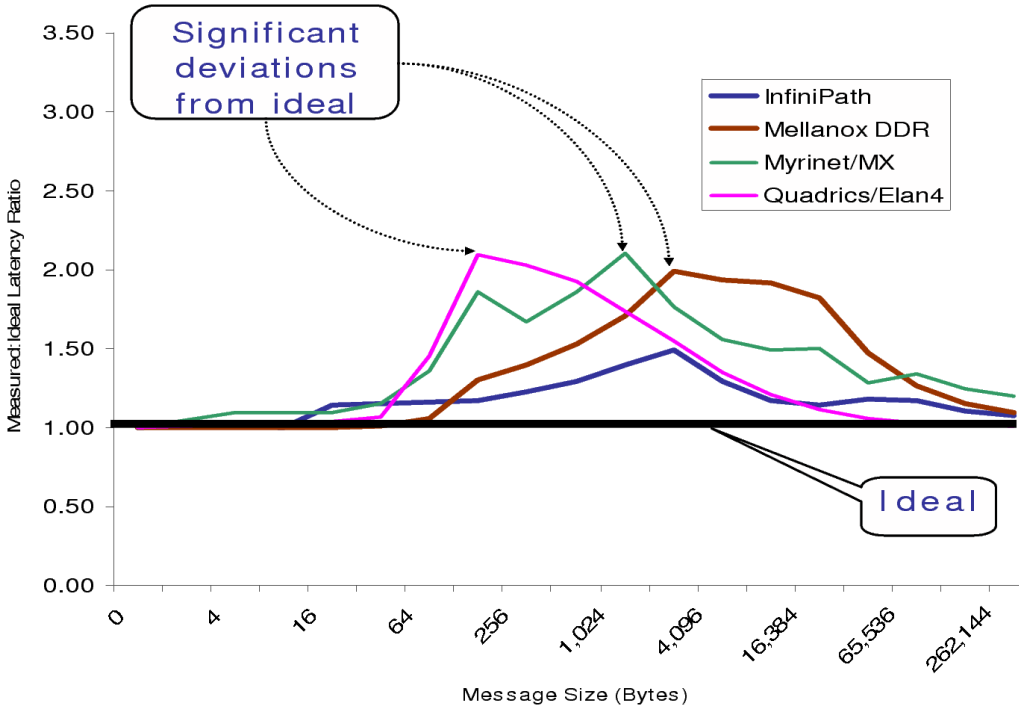

Abbildung 2.6: Verhältnis der realen zur theoretischen Übertragungszeit

breite um den Faktor zwei. Für sehr kleine Pakete wird meist das Piggybackverfahren verwendet, d. h. Benutzerdaten passen in das physikalisch kleinste Paket zum Paketkopf dazu und können in einem einzigen Schritt versandt werden. Es wird deutlich, dass manche Netzwerke intern ein Protokoll umschalten, daß die effektive Bandbreite stark reduziert. Je nach Netzwerktyp kann die Nachrichtengröße für die Umschaltung der intern verwendeten Methoden gesetzt werden.

Sehr wichtig im Bereich des HPC ist aber auch die so genannte $N_{\frac{1}{2}}$-Paketgröße, d. h. die Paketgröße, für die bereits die Hälfte der maximalen Bandbreite erreicht wird. Abbildung 2.7 zeigt exemplarisch die Bandbreite aufgetragen über die Paketgröße zweier fiktiver Netzwerke. Die Implementierung der gestrichelten Kurve würde die Hälfte der maximalen Bandbreite bereits bei einer geringeren Paketgröße erreichen. Vor allem diese $N_{\frac{1}{2}}$-Paketgröße ist bei der Skalierung in strenger Form (Strong Scaling) wichtig: Mit der Gebietszerlegungsmethode (engl. Domain Decomposition) parallelisierte Programme tauschen beim Strong Scaling kleinere und evtl. mehr Nachrichten auf. Somit sinkt die Effizienz der Kommunikation wegen der geringeren Paketgröße.

Diese Größen werden durch verschiedene Parameter beeinflusst, die sich in der Round Trip Time (RTT) widerspiegeln. Die RTT gibt die Zeit an, bis ein Signal oder eine Nachricht vom Sender zum Empfänger und wieder zurück gesandt wird. Das ist bspw. wichtig für Empfangsbestätigungen der Datenpakete (engl. Acknowledgement), um weitere Pakete versenden zu können und die Netzwerkleitung voll zu halten.

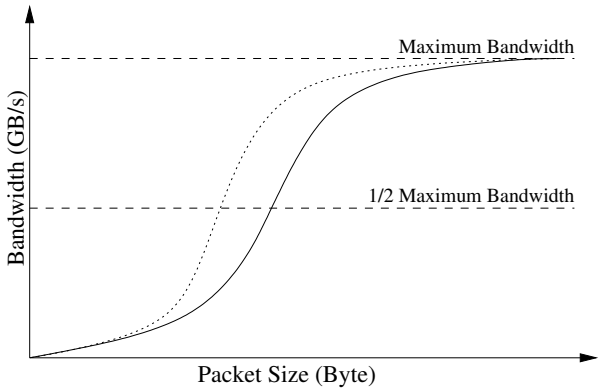

Abbildung 2.7: Bandbreite zweier fiktiver Netzwerke in Abhängigkeit der Paketgröße

2.2.1 Gigabit und 10-Gigabit Ethernet

Standardkomponenten mit Ethernet bieten die gängigste und kostengünstigste Möglichkeit, Knoten eines Clusters zu koppeln [97].

Theoretisch lassen sich mit Gigabit Ethernet (Gb) Bandbreiten von 125 $^{\mathrm{MB}}/_{\mathrm{s}}$ erreichen. Mit normalen Ethernet-Frames von 1518 Bytes kann man tatsächlich eine Bandbreite von 70-80 $^{\mathrm{MB}}/_{\mathrm{s}}$, bei einer Latenz von 20-30 μs mit Standardhardware über TCP erreichen. Die Gründe hierfür werden in Abschnitt 3.2.2 näher erläutert. Das schnellere 10-Gigabit (10 Gb) Netzwerk ist noch nicht weit verbreitet. Da die Infrastruktur und Topologie dem älteren Standard entspricht, dürfte der Wechsel kontinuierlich erfolgen. Die Probleme der 10 Gb-Technik sind:

- Die sehr hohe Anzahl von Paketen: bei 1500 Byte Paketgröße wird mit 10 Gb-Ethernet die Central Processing Unit (CPU) potentiell 833.333 $^{1}/_{\mathrm{s}}$ für die Verarbeitung von Paketen unterbrochen, was sich negativ auf den Cache und damit die ausführende Anwendung auswirkt[7]. Abhilfe schaffen hier so genannte Jumbo-Frames mit 9000 Byte anstatt der 1500 Byte-Paketen [39] und vor allem die Zusammenlegung von Interrupts (engl. Interrupt-Coalescing).

- Die Kommunikation erfordert den Aufruf von Kernelfunktionen: möchte eine Applikation eine Verbindung aufbauen oder Daten kommunizieren, werden diese Daten in den Kernelspeicherbereich kopiert, die Übertragung vorbereitet (CRC berechnen) und die Netzwerkkarte angesteuert. Abgesehen von den Kopieroperationen sind Sprünge in den Kernelkontext relativ teuer. Durch so genannte Null-Kopiermethoden (engl. Zero-Copy Networking Treiber), Hardwareunterstützung für die schnelle Berechnung von Checksummen, sowie TCP-Implementierungen in Hardware (engl. TCP-Offload Engine) werde die Kosten für diese Operationen vermindert.

[7] Das heißt der Prozessor wechselt vom Programm zur Interrupt-Service Routine (ISR) des Betriebssystems. Diese fragt die Interrupt-Leitungen ab und stößt die Übertragung des Netzwerkpaketes in einen Empfangspuffer im Betriebssystems an. Die tatsächliche Übertragung in den Speicher der Anwendung erfolgt evtl. erst später, um die ISR so kurz wie möglich zu halten.

Da der Gb-Netzwerkadapter mittlerweile in den Chipsatz auf jedem Knoten eines Clusters integriert ist, muss nicht in separate Netzwerkkarten investiert werden. Die Performance dieser Onboard-Karten ist allerdings nicht besonders hoch. Ein 48-Port Switch kostet 800 €, d. h. nur 16 € pro Port (alle Preise Stand September 2007).

2.2.2 Myricom Myrinet 2000 und Myri-10G

Für die schnelle Kommunikation innerhalb von Clustern bietet die Firma Myricom mehrere Netzwerkkarten, sowie dafür passende Switches an: Myrinet 2000 sowie Myri-10G. Während das ursprüngliche Myrinet [12] auf Signalebene ein spezielles Protokoll und Kabel verwendet, kann Myri-10G neben den proprietären Switches von Myricom mit 10 Gb-Ethernetswitches diverser Hersteller verwendet werden. Myrinet 2000 bietet eine Bandbreite von 235 MB/s bei einer Zero-Byte Latenz von 7,16 μs, Myri-10G wiederum erreicht eine Bandbreite von 1053 MB/s und eine Latenz von 3,32 μs (am HLRS gemessen mit `netpipe-3.6.2` [127] auf dem Cluster des DGrid-Projektes)[8].
Für die Kommunikation liefert Myricom drei Softwarestacks [100] mit:

1. Die Softwarebibliothek `gm` bietet eine garantierte, byte-orientierte Übertragung mit bis zu 16 MB-Paketen.

2. Im Vergleich hierzu bietet die Nachfolgerbibliothek `mx` eine API für die vollständige Punkt-zu-Punkt Kommunikation, inklusive durch Hardware unterstützte Warteschlangen für garantierten, paket-sequentiellen Empfang (engl. in-order packet receive). Dies erlaubt MPI-Implementierungen wie Open MPI [49] eine recht einfache Abbildung der Message Passing Interface-Semantik auf `mx`. Weiterhin bietet die Bibliothek `mx` die Möglichkeit, Pakete zum Empfänger über verschiedene Routen zu schicken, bspw. bei Switch-Hierarchien oder bei mehreren NICs pro Knoten. Auch für NIC-, sowie Kabel-Fehler ist diese Funktion interessant, sodass bei Ausfall einer Komponente die Kommunikation über die redundante Komponente erfolgt.

3. Auch auf Standard Unix-Sockets aufsetzende Programme können von der Performance von Myri-10G profitieren, indem die Bibliothek `libmyriexpresstcp` bei Programmstart hinzu geladen wird, die die Unix-Socket Aufrufe durch interne `open`, `close`, `read`, `write` und `select`-Funktionen der Bibliothek `mx` ersetzt.

Myrinet bietet den direkten Transfer aus dem und in den Speicherbereich der Anwendung an, d. h. es wird „am Betriebssystem vorbei“ via DMA kommuniziert. Damit das Betriebssystem nicht währenddessen die Adresse der physikalischen Speicherseite verschiebt oder gar auf Platte auslagert, muss die Speicherseite als nicht-relokierbar markiert werden (engl. Page pinning). Verschiedene Protokolle und Bibliotheken für Message Passing wurden für Myrinet programmiert [9, 17], die meisten MPI-Implementierungen setzen aber auf den oben erwähnten drei Bibliotheken des Herstellers Myricom auf.

[8] Laut dem Hersteller Myricom erreicht Myri-10G eine Bandbreite von 1204 MB/s und eine Latenz von 2,3 μs.

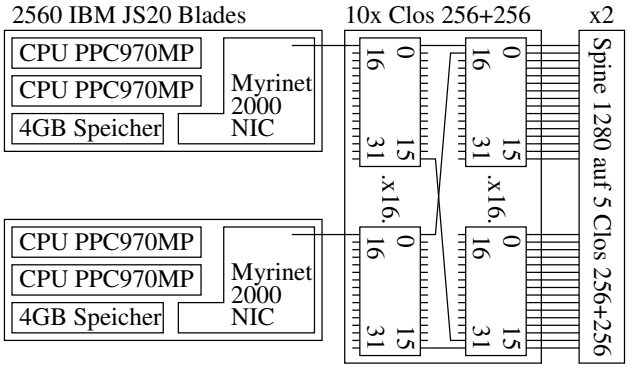

Abbildung 2.8: Myrinet-Netzwerk von Marenostrum mit 2560 Knoten

Mit Myrinet 2000 konnten Erfahrungen mit großen Installationen wie dem Marenostrum Cluster von IBM mit 2560 Knoten und insgesamt 10240 Prozessorcores gesammelt werden. In der Netzwerkkonfiguration von Marenostrum sind zehn CLOS 256+256 Switche [25], sowie als deren Backbone zwei Spine 1280-Switche verbaut. Eine Nachricht kann damit im schlimmsten Fall bis zu fünf der 32-Port Switche[9] auf dem Weg zum Empfänger durchlaufen [86]. Abbildung 2.8 zeigt einen Ausschnitt des Netzwerks.

Eine Installation der Größe von MareNostrum zeigt immer auch Limitierungen der verwendeten Software- und Hardwaretechnologien auf. Bei der Konfiguration von Marenostrum traten in Bezug auf das Netzwerk folgende Probleme auf:

- Die Bibliothek `gm` unterstüzt nur statisches Routing[10]. Diese Limitierung wurde in der Bibliothek `mx` durch einen dynamischen Algorithmus aufgehoben, welcher nicht nur bei einer Überlastung (engl. Congestion) eines Switches, sondern auch bei Ausfall einer Route, bspw. wegen eines defekten Kabels, eine neue Route berechnet.

- Die Bibliothek `gm` bietet keine nebenläufige Kommunikation mittels Threads. Das kann auch gut programmierten Anwendungen beim Hochskalieren Probleme bereiten. Zum Beispiel, wenn die CPU des empfangenden Prozesses nicht mehr innerhalb der MPI-Bibliothek läuft, sondern wieder die Anwendung ausführt, wird innerhalb von MPI kein Fortschritt der Kommunikation (engl. Progress) stattfinden. Damit ist es dem Sender nicht möglich, den Versand langer Nachrichten fortzusetzen, bis der empfangende Prozess wieder einen MPI-Aufruf tätigt.

Im Vergleich zu den Infiniband-Komponenten sind die Kosten für den Myrinet-NIC und Switch relativ hoch, da nur ein Hersteller die Hardware anbietet. Bei einem 16-Port Myrinet 2000 Switch einschließlich des NIC ergeben sich Preise von ca. 580 € pro Port. Laut Herstellerangaben würden die Kosten für die Switche von Marenostrum $505600 betragen (Preisliste Januar 2007), was für den Gesamtumfang und -leistung des Systems relativ gering ist.

[9] Via two-stage CLOS 256+256 Switch und single-stage Spine 1280 Backbone.

[10] Die Route eines Paketes wird beim Start des so genannten Mapper-Hintergrundprozesses einmalig festgelegt. Die Pakete folgen damit immer dem gleichen Pfad entlang der beteiligten Switches.

2.2.3 Infiniband

Der Infiniband (IB)-Interconnect hat sich aus zwei konkurrierenden Systembussen Future I/O von IBM, Compaq und HP, sowie Next Generation I/O von Intel, Microsoft und Sun entwickelt. Der Standard wird durch ein Industriekonsortium definiert, dem mittlerweile insgesamt 26 Firmen beigetreten sind.

Kommuniziert wird bei Infiniband (IB) über einen bidirektionalen, seriellen Bus. Dies ist kostengünstig zu implementieren und bietet eine ausgesprochen niedrige Latenz. Die Übertragungsrate liegt bei theoretischen 2,5 Gb/s duplex, bei der Double Data Rate (DDR)-Variante 5 Gb/s. Es können hierbei mehrere Kanäle gebündelt werden, bei vier Kanälen und DDR sind also theoretisch 20 Gb/s möglich – Verbindungen zwischen Switches können auch 12x ausgelegt sein, d. h. mit bis zu 60 Gb/s.

Die Programmierung erfolgt entweder über die (veraltete) Verbs-API des Herstellers Mellanox (kurz `mvapi`). Durch die Standardisierung wurde eine herstellerunabhängige API definiert, die so genannten IB-Verbs API. Diese API bietet Funktionen zur Verwaltung des Infiniband (IB)-NICs und Routinen zum Datentransfer. Die Kommunikation erfolgt über so genannte Queue-Pairs (QP), die in einem von vier Modi betrieben werden können (hier aufgeführt mit den oben beschriebenen Charakteristiken):

- Reliable Connection (RC): ein verbindungsorientiertes, Byte-granulares Protokoll mit Fehler-Erkennung und Flusskontrolle.
- Reliable Datagram (RD): ein verbindungsloses, Paket-granulares Protokoll mit Fehler-Erkennung ohne Flusskontrolle.
- Unreliable Connection (UC): ein verbindungsorientiertes, Byte-granulares Protokoll mit Flusskontrolle ohne Fehler-Erkennung.
- Unreliable Datagram (UD): ein verbindungsloses, Paket-granulares Protokoll mit Flusskontrolle ohne Fehlererkennung.

Die Initialisierung der QPs erfolgt via (langsamer) Aufrufe in den Kernel. Funktionalitäten im kritischen Pfad (engl. Fast Path), wie Kommunikation und Steuerung der QPs werden mittels der in den Anwendungsprozess gemappten Register gesteuert. Die Kommunikation erfolgt somit am Kernel vorbei, mittels Page pinning auch mit Remote Direct Memory Access (DMA), d. h. ein Prozess greift direkt auf Daten im Speicher eines entfernten Prozesses zu.

Um einen fairen Vergleich zwischen Myri-10G und Infiniband (IB) mit DDR zu ziehen, wurde der NetPipe Benchmark auf einem neuen Cluster mit Dual Intel-Woodcrest Prozessoren ausgeführt: Acht Knoten des Clusters wurden mit Myri-10G NICs[11] ausgestattet, die restlichen Knoten waren mit Mellanox IB-NICs[12] ausgestattet.

Die Abbildung 2.9 zeigt die Bandbreite von Myri-10G und IB-DDR mit verschiedenen MPI-Implementierungen (sortiert nach der Übertragungsbandbreite):

[11] LANai-Chipsatz mit 300 Mhz Taktfrequenz und 2 MB SRAM lokalen Speicher, angebunden über PCI-Express 8x.

[12] Mellanox HT25204 Chipsatz, Memfree, PCI-Express 8x, Firmware Version 1.0.8, DDR: IB-4x, 5 Gb/s.

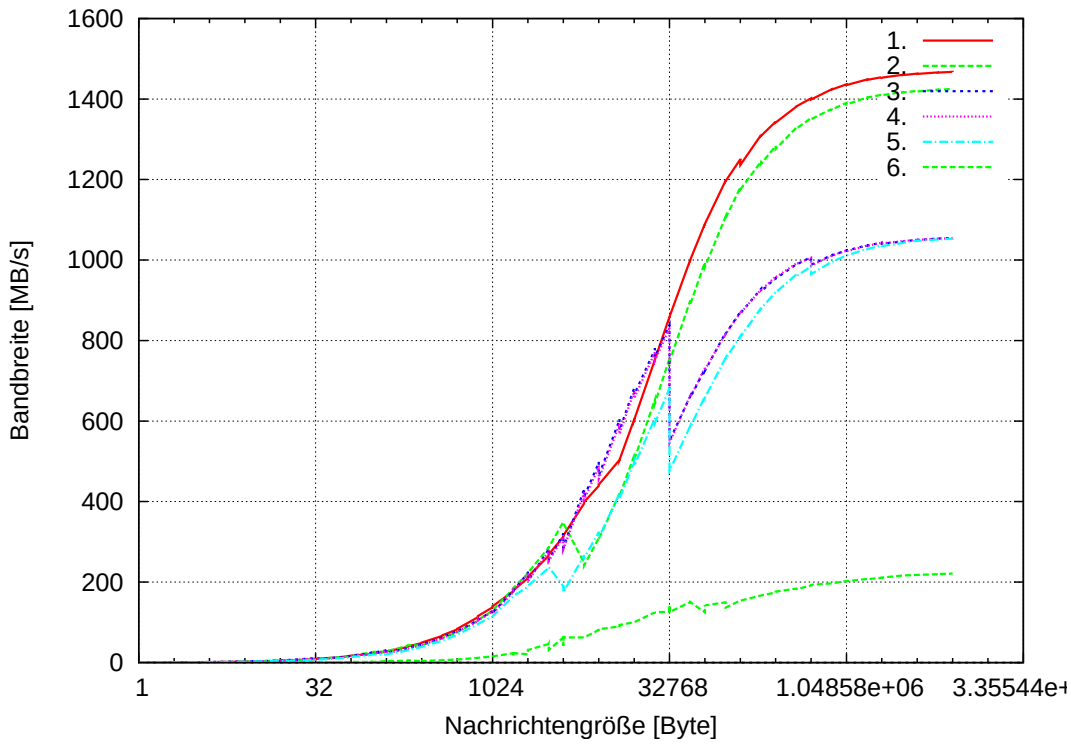

Abbildung 2.9: Vergleich der Bandbreite von Myri-10G und Infiniband DDR

1. *Open MPI (BTL/openib)*: Open MPI auf IB, hier basierend auf der Bit Transfer Layer (BTL)-Schicht[13].

2. *MVApich-0.9.7 (IB)*: Die vom IB-Hersteller mitgelieferte MPI-Implementierung, hier MVApich, basierend auf `MPIch2-0.9.5`.

3. *MPIch-mx*: Die von Myricom mitgelieferte Implementierung `MPIch-1.2.7.1`. Diese verwendet die Punkt-zu-Punkt Semantik der `mx`-Bibliothek.

4. *Open MPI (MTL/mx)*: Open MPI auf Myrinet unter Verwendung von `mx` in der Message Transfer Layer (MTL)-Schicht.

5. *Open MPI (BTL/mx)*: Open MPI auf Myrinet mit BTL/`mx`.

6. *Open MPI (tcp over mx)*: Open MPI auf Myrinet aber über das TCP-Interface.

Zu diesen Messungen ist Folgendes anzumerken. Die kleinen „Zacken“ in den Linien rühren von der Art der Messung von `netpipe` [127] her: Die Nachrichtenlängen werden in 2-er Potenzen erhöht, um Vorteile von Nachrichtenlängen mit 2-er Potenz auszuschalten,

[13] Während die BTL-Schicht nur die korrekte Übertragung von Fragmenten garantiert, verlässt sich die (dünnere) MTL-Schicht von Open MPI auf die korrekte Semantik von Punkt-zu-Punkt Kommunikation der darunter liegenden Schicht – dies wird derzeit von `mx` für Myrinet, `portals` auf Cray Maschinen, sowie von `PSM` des Infinipath-NICs von Qlogic garantiert.

werden zusätzlich Messungen mit ±3 Bytes Nachrichtenlänge durchgeführt, bspw. nicht nur mit 16 Byte, sondern auch mit 13 Byte und 19 Byte Nachrichtenlänge. Die Messungen werden bis zu einem verstellbaren Wert (hier 10000) Mal wiederholt, bis sich die Messzeit stabilisiert hat.

Während Myri-10G Bandbreiten bis zu 1053 MB/s erreicht, kommt IB in dieser Konfiguration mit Open MPI[14] auf 1467 MB/s. Für MVApich wurden keine Optionen, wie Page Pinning angewandt; evtl. hätte damit der starke Einbruch der Bandbreite bei der Protokollumschaltung bei 4096 Byte Nachrichtengröße vermieden werden können. Bei näherer Betrachtung fallen die Protokollumschaltungen, besonders bei Myrinet auf; das Limit für den Versand mit dem „Eager Message"-Protokoll kann durch eine Umgebungsvariable von 32 kB auf 64 kB hochgesetzt werden Unter Open MPI mit BTL/mx wird ein Einbruch der Bandbreite sichtbar, der mit MTL/mx und MPIch-mx nicht auftritt: hier wird BTL/mx ein anderes Protokoll benutzt als unter MTL/mx, d. h. wenn die P2P-Funktionalität von `mx` in Open MPI ausgenutzt wird, tritt diese Umschaltung nicht auf. Damit kann das BTL/mx die Bandbreite von MPIch-mx im Bereich der mittleren Nachrichtengröße nicht erreichen. Erst bei großen Nachrichtenlängen gleicht sich die Bandbreite wieder an. Nachdem sowohl MPIch-mx als auch Open MPI mit MTL/mx die `mx`-Bibliothek nutzen, erreichen beide die gleichen Resultate. Diese liegen im Bereich von 4 kB bis 32 kB sogar leicht höher als bei Infiniband (IB). Offensichtlich ist die Kombination mit TCP über Myrinet die langsamste Möglichkeit der Kommunikation, da hier die Daten an den Kernel kommuniziert werden.

Vergleicht man die Ausgereiftheit von Infiniband (IB) mit anderen Interconnects, hat in der Vergangenheit die Kompatibilität und Stabilität der Software von IB-Karten für Probleme gesorgt. Dies lag an der fehlenden Standardisierung der Schnittstellen und daran, dass mehrere Firmen aus den unterschiedlichsten Bereichen an Infiniband (IB) mitarbeiten. Diese Firmen haben unterschiedliche Ansprüche in Bezug auf Funktionalität und Leistungsfähigkeit, je nachdem ob Lösungen für High-Performance Cluster- oder Storage Area Network-Interconnects entwickelt werden. Durch die Integration in eine gemeinsame Softwaredistribution, die so genannte Open Fabrics Enterprise Distribution (OFED), wurde die Qualität besser. Die Software ist mittlerweile in einem stabilen Zustand, die Protokolle und Softwareschichten klar strukturiert und verschlankt, sodass die Software den unterschiedlichen Ansprüchen gerecht wird.

Die Kosten für die IB-Karten und Switches ist in den letzten zwei Jahren (seit 2004) stark gesunken – vor allem weil man die Auswahl aus verschiedenen Herstellern hat. Dies gilt leider nicht für die Chipsätze, die intern in den Karten und Switches verwendet werden: diese stammen zumeist von Mellanox. Umgerechnet auf den Port kostet Infiniband derzeit 400 €.

Infiniband ist dafür ausgelegt, als Interconnect für verschiedene Cluster zu dienen. Das macht es für MetaComputing besonders interessant, bspw. können mehrere kleinere Cluster in Firmen, die meist von unterschiedlichen Cost-Centern beschafft werden, zu einem großen Höchstleistungsrechner gekoppelt werden.

[14] Folgende Optionen wurden für Open MPI verwendet: `leave-pinned`, `btl_openib_rd_num=128`, `btl_openib_max_eager_rdma=20`.

2.2.4 InfiniPath

InfiniPath ist ein schneller Interconnect des Herstellers Qlogic (ehemals PathScale). Der Interconnect kann einerseits über PCI-X als auch direkt am Hypertransport (HTX)-Kanal des AMD Opteron Prozessors angebunden sein und ist als Single-Chip ASIC implementiert. Die Kommunikation von InfiniPath basiert auf den Infiniband (IB)-Verbs. Die Verwendung des Infiniband-Standards erlaubt die Verwendung von IB-Switches unterschiedlicher Hersteller, nicht aber die heterogene Ausführung auf einer Mischung von Knoten mit Standard Infiniband- sowie InfiniPath-Network Interconnect Cards (NICs). Der NIC, sowie die Kommunikation mit dem Prozessor des NIC ist bei InfiniPath anders aufgebaut:

Durch die direkte Ankopplung an den Prozessorbus erreicht InfiniPath Übertragungsraten von 953 $^{\mathrm{MB}}/_{\mathrm{s}}$ und eine sehr geringe Latenz von 1,29 μs [31]. Das Besondere von InfiniPath ist hierbei die sehr geringe $N_{\frac{1}{2}}$-Paketgröße von 88 Byte, d. h. die halbe mögliche Bandbreite wird schon für Pakete der Größe 88 Byte erreicht und damit für ein sehr großes Spektrum von Kommunikationsmustern von typischen Anwendungen [33]. Das wird durch den geringen Overhead des Netzwerkadapters pro Paket erreicht. Laut dem genannten Whitepaper gehen von den 1,29 μs nur 0,6 μs zu Lasten des NICs, nur 0,2 μs liegen beim Switch, die restlichen 0,5 μs werden durch den Prozessor verbraucht[15]. Besonders hier kann InfiniPath von hohen Prozessortaktraten profitieren.

Da InfiniPath auf Infiniband Infrastruktur aufsetzt, entsprechen die Kosten für Switches und Verkabelung dem von Infiniband (IB). Die Karten aber sind im Vergleich mit 1200 € relativ teuer.
Eine Unterstützung von heterogenen Systemen im InfiniPath-Softwarestack `psm` ist nicht bekannt.

2.3 Message Passing Interface Implementierungen für MetaComputing

Die Kopplung von Clustern setzt eine weitere Softwareschicht und die Abstraktion der eigentlichen MPI-Prozesse voraus. Für den Spezialfall des MetaComputing gibt es demnach wenige Implementierungen. Diese decken meist nur einen Teilbereich des Funktionsumfang des MPI-Standards ab. In diesem Abschnitt werden die Historie, Architektur und erkennbare Vor- und Nachteile erläutert.

2.3.1 MPIch-G2

Diese Bibliothek basiert auf MPIch der Version 1. Sie wurde speziell für den Grid-Bereich an der *Northern Illinois University* (NIU) in Zusammenarbeit mit den *Argonne National Labs* (ANL) entwickelt [43, 44].

MPIch-1 abstrahiert Kommunikationsschnittstellen durch so genannte Devices. Für MPIch-G2 wurde ein spezielles Device, *globus2* genannt, implementiert [74], das auf den

[15] Dies entspricht bei dem verwendeten Prozessor mit 2,8 GHz noch 1400 Taktzyklen.

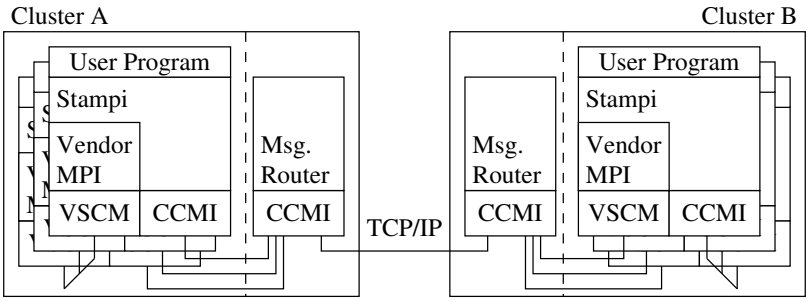

Abbildung 2.10: Schematischer Aufbau der Kopplung durch Stampi

Funktionalitäten des Globus Toolkit (Globus TK) basiert. Da diese Bibliothek Konnektivität zwischen allen Prozessen erfordert, ist der Betrieb zwischen Clustern mit lokalen IP-Adressen (bspw. 192.168.x.y oder 10.x.y.z) nicht möglich[16]. Weiterhin müssen einige Ports für Cluster, die durch Firewalls geschützt sind, freigeschaltet werden. Diese Ports sind dann wiederum bei Ausführung mit anzugeben, was zu Problemen führt, wenn diese Angabe nicht auf allen Prozessen gesetzt ist. MPIch-G2 verwendet für die Kommunikation TCP-Verbindungen und kann (bei nicht-gethreadetem Globus) mehrere Netzwerkverbindungen zur Erhöhung der Bandbreite verwenden.

MPIch-G2 benutzt Funktionalitäten des Globus Toolkit [45] der Version 2.2, bspw. um Benutzer zu authentifizieren, für den Programmstart oder um Daten zu konvertieren. Laut der Dokumentation von MPIch-G2 funktioniert das IO-Konzept nicht mehr mit der Version 3.2.1 von Globus, auch ist eine Reimplementierung für neuere Versionen wie das Globus TK-4 nicht geplant.

Eine Weiterentwicklung von MPIch-G2 ist derzeit nicht absehbar, zumindest gibt es keine weiteren Publikationen auf den Webseiten des Projektes. Dennoch wird es in einigen Anwendungsszenarien verwendet, bspw. beschreibt [13] eine transatlantische Verbindung von Rechnern in England und dem TeraGrid.

2.3.2 Stampi

Die Stampi-Bibliothek (**S**eamless **T**hinking **A**id **MPI**) wird von dem japanischen Forschungszentrum für Atomenergie (JAEA, ehemals JAERI) entwickelt.

Die Architektur von Stampi ist in Abbildung 2.10 dargestellt (aus [68]). Diese Lösung ist der von PACX-MPI relativ ähnlich, wie auf der nächsten Seite aufgezeigt wird. Auch Stampi verwendet für die externe Kommunikation TCP, ansonsten keine weiteren Protokolle für externe Kommunikation. Es können Cluster mit lokaler IP-Adresse mittels des so genannten Message Routers gekoppelt werden.

Ein wichtiger Unterschied zu PACX-MPI ist, dass der Routerprozess nicht Teil der MPI-Applikation selbst ist, d. h. die Kommunikation muss über TCP erfolgen. Anderer-

[16] IP-Pakete mit lokalen Adressen können nicht in das Internet geroutet werden.

seits können in Stampi beliebig viele Routerprozesse erzeugt werden, um Bottlenecks der externen Kommunikation zu reduzieren. Weiterhin wurde in Stampi eine ausgeklügelte Integration in die am JAERI entwickelte Grid-Umgebung vorgenommen.

Es sind derzeit keine weiteren Entwicklungen an der Grundfunktionalität von Stampi bekannt.

2.3.3 MetaMPICH

Diese Bibliothek basiert auf der MPIch-1.1-Bibliothek und wurde ursprünglich unter dem Namen MetaMPI vom Forschungszentrum Jülich und der Pallas GmbH uentwickelt. Es wird nun in einer Kooperation der RWTH Aachen und dem Forschungszentrum Jülich weiter gepflegt.

Bei dieser Lösung wird nicht auf einem Hersteller-MPI aufgesetzt, stattdessen wird der Netzwerktreiber (in diesem Fall der Scali Interconnect (SCI)) direkt angesprochen. In neueren Ansätzen wird der direkte Anschluss an Hochgeschwindigkeitsnetzwerke beschrieben [11]. Kommunikation mit anderen Teilen des Metacomputers erfolgen mit TCP. Bei dieser Lösung können Cluster mit lokalen IP-Adressen gekoppelt werden, auch wenn dies zu Performanceverlusten durch den weiteren Routerprozesse führt. Um den Start der Applikation innerhalb des Metacomputers zu erleichtern, wurde MetaMPICH direkt in die Unicore-Umgebung [10, 115] integriert. Das erleichtert die Probleme mit Konfigurations- und Input-Dateien.

Die Problematik bei diesem Ansatz ist die Notwendigkeit, die interne Kommunikation optimieren zu können, d. h. es müssen Algorithmen für Punkt-zu-Punkt und kollektive Kommunikation zur Verfügung gestellt werden.

Laut [24] wird MetaMPIch in Weitverkehrsnetzen, wie dem VIOLA Projekt eingesetzt. Für den Support neuerer Cluster wird vor allem die Unterstützung weiterer Netzwerke wie Infiniband wichtig werden. Weiterhin wird an der Unterstützung von IMPI gearbeitet.

2.3.4 PACX-MPI

Die Bibliothek PACX-MPI (**PA**rallel **C**omputer E**X**tension) wurde an der Universität Stuttgart zur Kopplung heterogener Rechnerarchitekturen entworfen und in verschiedenen internationalen Projekten [50, 78] weiterentwickelt. In den Dissertationen von M. Resch [111] und E. Gabriel [48] wird das Konzept des MetaComputing im Allgemeinen und bei Gabriel die Software PACX-MPI im speziellen, inklusive der Implementierung behandelt. Insofern soll hier nur kurz auf die Grundlagen eingegangen werden und die Arbeiten erläutert werden, auf denen aufgebaut wird.

PACX-MPI koppelt zwei oder mehr Höchstleistungsrechner über ein verbindendes LAN oder WAN, bspw. das Internet. Dies geschieht über zwei Prozesse der Applikation, die die zu kommunizierenden Pakete an den empfangenden Rechner weiterleiten, wie in Abb. 2.11 dargestellt. In der Initialisierungsphase der Anwendung werden diese beiden Prozesse separiert. Alle weiteren Prozesse führen den Anwendungscode aus, während die beiden Prozesse Daten nach außen (Outdaemon) schicken, bzw. von außen für Prozesse

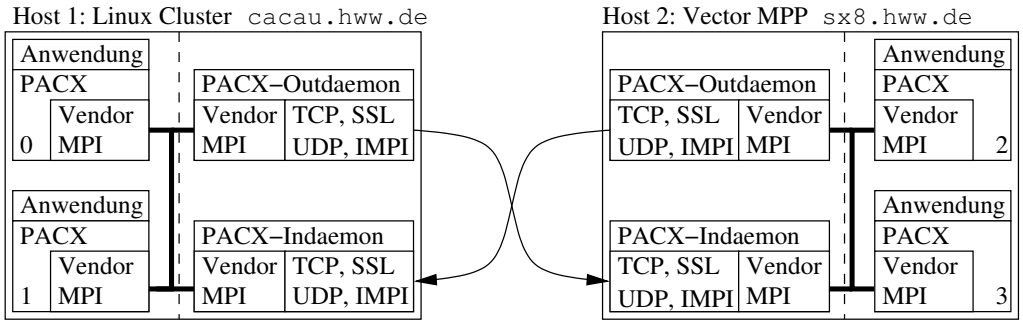

Abbildung 2.11: Architektur von PACX-MPI

im Parallelrechner (Indaemon) empfangen. Die Prozesse, die die Anwendung ausführen, bekommen einen neuen MPI-Rank innerhalb des globalen Kommunikators. In Abb. 2.11 sind zwei Hosts mit je zwei Anwendungsprozessen gekoppelt, hier ein Linux-Cluster und der Vektor-MPP NEC-SX8. Den Anwendungsprozessen des Clusters sind die Ranks 0 und 1, den Prozessen des zweiten Hosts sind die Ranks 2 und 3 zugewiesen.

Die gesamte Kommunikation innerhalb einer Anwendung erfolgt über das vom Hersteller optimierte MPI (engl. Vendor-MPI), während Kommunikation mit Prozessen außerhalb des Rechners über die Dämonen versandt wird. Dies erfordert dementsprechend das Senden von Headerdaten vor den Nutzdaten sowie Kopierroutinen auf den Dämon. Als Kommunikationsprotokoll über das WAN standen TCP, SSL, ATM und die vom Autor implementierte Thread-TCP, sowie Interoperables MPI (IMPI) [28, 130] zur Verfügung.

In dieser Arbeit wird erörtert, welche Möglichkeiten bestehen, diesen Overhead zu reduzieren, einerseits auf algorithmischer Ebene und andererseits sehr nah an der Hardware innerhalb und unterhalb der PACX-MPI Schicht.

2.3.5 Open MPI

Auf der SuperComputing 2003 in Phönix, USA, haben sich die Entwickler der Implementierungen LAM/MPI [129], FT-MPI [42] und LA-MPI [57] getroffen, um über eine komplette Neuentwicklung einer MPI-Bibliothek zu diskutieren. Der Grund hierfür war, dass alle Bibliotheken die gleichen Probleme bezüglich der Unterstützung von MPI-2 Funktionalitäten haben. Bei diesem Neudesign sollten Möglichkeiten des MetaComputing von vornherein berücksichtigt werden, weswegen das HLRS mit PACX-MPI an dem Konsortium als einer der ersten vier Partner beteiligt ist.

Ziel von Open MPI ist die Unterstützung des vollen MPI-2 Standards, inklusive kompletter Thread-Sicherheit. Dies soll in einer einzigen stabilen und offenen Bibliothek integriert sein. Dennoch soll die Bibliothek als Basis für Forschungsprojekte dienen und Implementierungen von neuartigen Konzepten erlauben. Durch das offene BSD-Lizenzmodell soll die Integration in bestehende Softwarepakete erleichtert, aber auch explizit die kommerzielle Verwendung und Weiterentwicklung durch Hersteller von Netzwerkhardware und Softwarefirmen (engl. Independant Software Vendors (ISV)) ermöglicht werden. Beispielsweise liefern einige ISVs für eine zertifizierte Version ihrer Simu-

lationspakete den gesamten Softwarestack inklusive der MPI-Bibliothek aus.

Seit der Version 1.0 auf der Supercomputing 2005 sind drei weitere wichtige Releases der Bibliothek veröffentlicht worden. Mittlerweile ist das Konsortium auf 20 Partner angewachsen, u. a. die amerikanischen Forschungslabs Los Alamos National Labs (LANL), Sandia National Labs (SNL), Oakridge National Labs (ORNL), fünf Universitäten, namentlich die Indiana University (IU), die Universität Stuttgart, vertreten durch das HLRS, die Universität Houston, die Technische Universität Dresden und die Universität Chemnitz.

Besonders wichtig für eine derartige MPI-Implementierung ist das Interesse und die Unterstützung durch Hersteller. Derzeit sind IBM, Sun Microsystems und die Netzwerkhersteller Qlogic, Myricom, Mellanox, Voltaire und Cisco am Konsortium beteiligt. Außerdem bietet Coverity als Projektpartner einen Service für statische Source-Code Analyse dem Open-MPI Projekt an.

Das HLRS bringt seine Erfahrung und Kompetenz in Open MPI im Rahmen von EU-geförderten Projekten sowie einem Sonderforschungsbereich mit ein und wird damit für eine weitere Entwicklung der PACX-Funktionalitäten sorgen.

3 Verbesserungen innerhalb der Kommunikations- und Netzwerkschicht

In diesem Abschnitt werden die Arbeiten auf der untersten Softwareschicht oberhalb des Betriebssystems erläutert, vergleiche Abb. 1.9 auf Seite 15. Die Kommunikationsgeschwindigkeit des externen Netzwerkes kann beim MetaComputing nicht mit der des internen Netzwerkes mithalten. Dies liegt einerseits konzeptionell bedingt an der unterdimensionierten Bandbreite in Relation zur Anzahl der möglichen Kommunikationspartner, andererseits an der verwendeten Technologie für externe Kommunikation, sowie selbstverständlich an den zu überbrückenden Entfernungen zwischen den Hosts. Abbildung 2.11 zeigt dieses Missverhältnis der Kommunikationsperformance deutlich durch die Stärke der Verbindungen der Prozesse untereinander im Vergleich zur externen Verbindung.

Der Erste in dieser Arbeit vorgestellte Ansatz, die Ausführung von Applikationen im MetaComputing zu beschleunigen, ist die unterste Softwareschicht des Netzwerkes zu betrachten und zu optimieren. Um die Übertragung über allgemeine Netzwerke wie das Internet zu beschleunigen, müssen mehrere Faktoren in Betracht gezogen werden:

- die Netzwerkinfrastruktur,
- die Netzwerkkarte,
- das Betriebssystem,
- andere Nutzer des Netzwerkes.

Auf die Hardware, die Netzwerkinfrastruktur sowie andere Benutzer hat der Programmierer und Anwender keinen Einfluss, außer dass der Netzwerkadministrator die Routen über andere Provider schalten kann[1], um anderen konkurrierenden Nutzern zu entgehen. Die Einschränkungen des Betriebssystems kann man teilweise durch die Wahl anderer Kommunikationsprotokolle und durch das Setzen von Parametern verändern.

Ordnet man die Verzögerung innerhalb der Netzwerkkommunikation in verschiedene Kategorien ein, ergibt sich eine Aufteilung, wie in Abb. 3.1 dargestellt. Neben der

[1] Die Universität Stuttgart ist an das Baden-Württembergische Wissenschaftsnetz (BelWue) angebunden. Aus dem BelWue kommen die Pakete in das Deutsche Forschungsnetz (DFN); von dort gelangen sie in das Europäische Hochgeschwindigkeitsnetzwerk Géant.

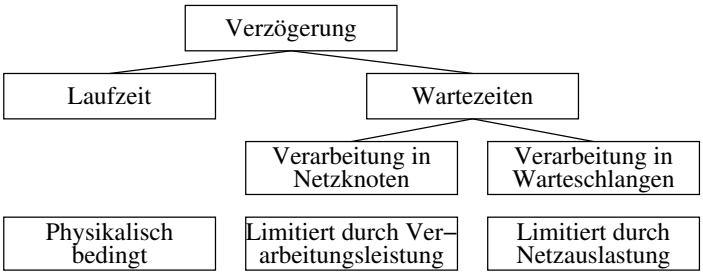

Abbildung 3.1: Grobe Aufteilung der Wartezeiten bei Netzwerkkommunikation

eigentlichen Signallaufzeit tragen die Verarbeitungszeiten in den Netzknoten und Warteschlangen erheblich zur Verzögerung bei. Die Unterscheidung der Wartezeiten ist deswegen interessant, weil nicht nur die konkurrierenden Datenströme die Verarbeitung eines Datenpaketes ausbremsen können, sondern auch Router für bestimmte IP-Pakete optimiert sein können. Werden außergewöhnliche IP- oder TCP-Optionen, wie bspw. Window-Scaling-Optionen verwendet, werden die Datenpakete im Slow-Path des Routers weiterverarbeitet und damit unnötigerweise verzögert.

Die im MetaComputing verwendeten Rechner verfügen im Frontend-System meist über Ethernet-Netzwerkkarten und durch Gigabit-Ethernet mittlerweile mit annehmbarer Geschwindigkeit für interaktive Arbeit. Anwendungen setzen meist IP-basierte Protokolle[2]. Für die Übertragung wird daher auch in dieser Arbeit TCP und UDP verwendet und erweitert.

3.1 Multiple Netzwerkverbindungen

Die in diesem Abschnitt beschriebene Funktionalität wurde im Rahmen der Diplomarbeit des Autors [79] implementiert. Deswegen soll hier nur kurz auf die Methoden und Ergebnisse eingegangen werden. Das Aufgabengebiet und die Lösung passen jedoch genau in die Thematik dieser Dissertation. Zu Details der Implementierung und zu Messungen zwischen den Systemen Cray T3E und sei auf [79] verwiesen.

TCP besitzt einige Einschränkungen, die den Anforderungen an das Protokoll geschuldet sind, siehe Abschnitt 2.1.2 auf Seite 25:

- Fairness mit anderen TCP-Streams verhindert die Auslastung der Bandbreite.
- Die Window-Size reicht evtl. bei Netzwerken mit großer Bandbreite und sehr hoher Latenz nicht aus, um die Pipeline des Netzwerkes zu füllen.
- Die Verwaltung des TCP-Streams führt zu Overhead im Betriebssystem.

Zur Verbesserung der ersten beiden Punkte, d. h. um die vorhandene Bandbreite effektiver zu füllen, hat der Autor in [79] eine Technik für PACX-MPI entwickelt, um mit

[2]Mit Administratorrechten können auch nicht-privilegierte Anwendungen Ethernet-Frames senden – dies widerspricht aber den Sicherheitsrichtlinien aller Rechenzentren.

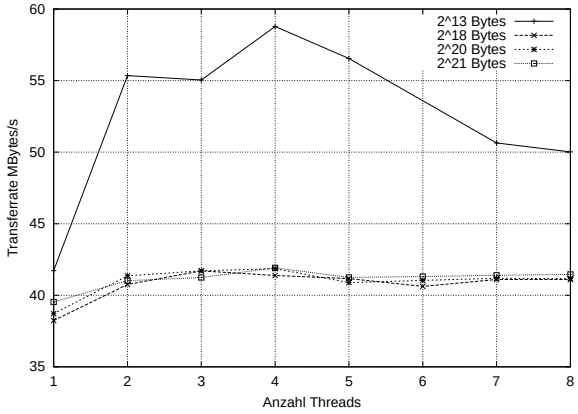

Abbildung 3.2: Bandbreite zwischen zwei Linux-SMPs über HIPPI mit konfigurierten Sockets, *un*synchronisierte Version

PThreads mehrere parallele TCP-Netzwerkverbindungen zwischen dem Out- und Indaemon aufzubauen. Es muss die vom MPI-Standard geforderte Semantik[3] erfüllt werden, beschrieben auf Seite 25, d. h. bei mehreren parallelen Verbindungen Nachrichten muss berücksichtigt werden, dass Nachrichten einander überholen können. Hierfür wurde eine Pufferung und Sortierung von Paketen in den Dämonen implementiert. Für die Koordination der Empfangsthreads wurden mehrere Netzwerkverbindungen zusammengefasst behandelt, um den Overhead des Lockings gering zu halten.

Ein Großteil der Arbeit in [79] wurde für die Tests und portable Programmierung aufgewendet, da einige der geforderten Systeme[4] den PThread-Standard [67] nicht komplett unterstützten. So basiert die Implementierung der Hitachi SR8k nicht auf dem finalen PThread-Standard, sondern auf einem Thread, in dem einige Variablennamen, sowie statische Initialisierung noch nicht definiert war. Dies musste in der Implementierung berücksichtigt werden.

Für die hier vorgestellte Konfiguration wird mit mehreren parallelen TCP-Streams eine Steigerung der Bandbreite um über 35% erreicht. Diese Konfiguration besteht aus der Koppelung von Linux-SMPs über das so genannte HIgh Performance Parallel Interface (HIPPI) bei vier sendenden Threads, wie in Abb. 3.2 zu sehen.

3.2 Übertragung mittels eines UDP-basierten Protokolls

Eine weitere Möglichkeit, die externe Kommunikation zu beschleunigen, ist die Benutzung eines alternativen Protokolls. Um die beschriebenen Probleme mit TCP zu umgehen und die Bandbreite und evtl. Latenz zu verbessern, wurde eine allgemein verwendbare

[3] Garantierter Fortschritt (engl. Progress), korrekter Empfang, keine überholenden Nachrichten.

[4] Geforderte Betriebssysteme: SuperUX auf NEC-SX4/SX5, Unicos auf der Cray T3E, HiUX auf Hitachi SR8k, sowie Linux.

Bibliothek `RUDP` [81] vom Autor entwickelt. Diese setzt zur Übertragung das User Datagram Protocol (UDP) ein und verwendet Threads zur nebenläufigen Kommunikation. Da das Ziel für die Kopplung von Clustern über ein verbindendes Netzwerk wie das Internet vorgegeben war, reduziert sich die Auswahl auf IP-basierte Protokolle, neben TCP eben UDP, das Transparent Inter Process Communication (TIPC) oder auch das Datagram Congestion Control Protocol (DCCP). Hierbei fiel die Wahl auf UDP, da es auf allen Zielplattformen und Betriebssystemen (Linux, Solaris, Unicos und Super-UX) zur Verfügung steht.

Diese Bibliothek stellt einer Anwendung die Funktionen `udp_open`, `udp_close`, sowie `udp_read`, `udp_write` zur Verfügung. Für die Übertragung von großen Nachrichten genügen diese vier Funktionsaufrufe. Routinen für nichtblockierende Semantik hätten die Implementierung verkompliziert[5]. Die Funktionen haben die gleiche Aufrufsyntax wie Unix IO-Routinen. Auch in `RUDP` wird ein Integer als Handle verwendet, der den Stream identifiziert, d. h. nach einem `udp_open` kann man mit `udp_write`(`fd`, `buffer`, `buffer_len`) einen Puffer mit `buffer_len` Bytes über das Netzwerk übertragen. Insofern lässt sich die Bibliothek leicht in beliebige Anwendungen und eben auch PACX-MPI integrieren.

3.2.1 Implementierung der UDP/IP Netzwerkübertragung

Zuerst wird UDP gemäß den Charakteristika, beschrieben auf den Seiten 20–22, eingestuft:

Verbindungsaufbau:	Es gibt keinen three-way-handshake für den Verbindungsaufbau.
Granularität:	Paket-orientierter Versand.
Datenverlust:	Pakete können verloren gehen, Daten innerhalb des Paketes können fakultativ durch einen CRC verifiziert werden, d. h. Pakete werden vom Betriebssystem und für die Anwendung transparent verworfen.
Flusskontrolle:	Keine Flusskontrolle, eine Anwendung kann beliebig Daten senden und den Empfänger oder evtl. das Netzwerk komplett auslasten.
Quality of Service:	Ein QoS-Merkmal kann nur über IP gesetzt werden.
Darstellung:	Keine Semantik der versendeten Daten.

Für die Implementierung wurde die Programmiersprache C gewählt [36]. Diese Sprache eignet sich besonders, da die geforderte Funktionalität hardwarenah ist und damit performant im Maschinencode sein muss [61]. Weiterhin können die Funktionen aus der Bibliothek aus anderen Programmiersprachen wie Fortran oder C^{++} aufgerufen werden. Um anhand dieser Charakteristika und der geforderten Programmiersprache ein effizientes Protokoll zu implementieren, wurden die folgenden Vorgaben definiert:

- Keine unnötigen Kopieroperationen der zu versendenden/empfangenden Daten,

[5] Die Übertragung erfolgt mittels Threads – für Abfragen hätte ein teurer Locking-Mechanismus implementiert werden müssen.

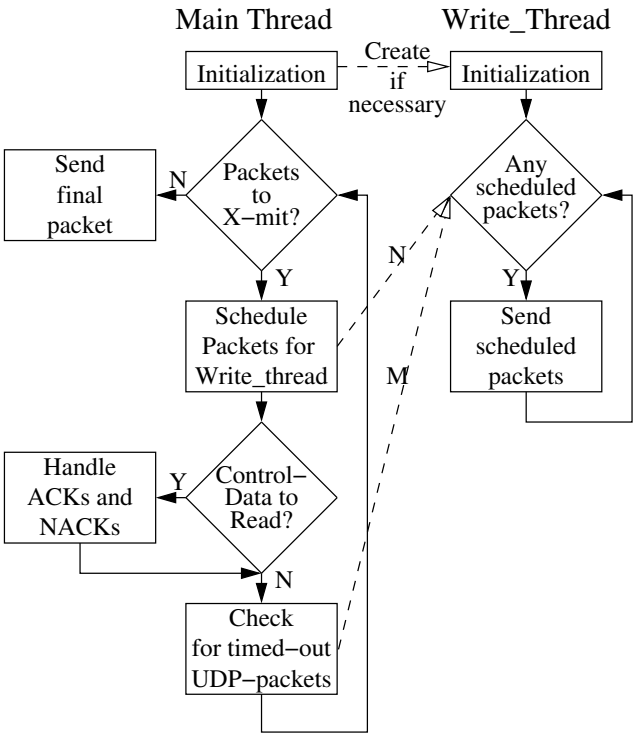

Abbildung 3.3: Programmflussdiagramm des Schreibaufrufs und Schreibthreads

- frühzeitige Erkennung verloren gegangener Daten,
- feingranulares, „schlaues“ Protokoll für das Acknowledgement von empfangenen und verloren gegangenen Daten,
- nebenläufiger Versand und Empfang mit Threads,
- geringer Overhead in Bezug auf Speicher- und Prozessorbelastung.

Zur Erklärung der Implementierung ist das Ablaufdiagramm in Abb. 3.3 hilfreich.

Die Kommunikation der UDP-Pakete erfolgt nebenläufig zum Versand/Empfang der Acknowledgements, hierfür wird je ein Schreib-, bzw. Lese-Thread für den Versand bzw. Empfang verwendet. In der Initialisierungsphase wird beim ersten Aufruf der Schreibthread gestartet. Zwischen Server und Client wird eine TCP-Verbindung für Initialisierungsdaten sowie Acknowledgements aufgebaut. Es wird die Option `TCP_NODELAY` für die TCP-Verbindung gesetzt, um ACK-Daten unverzüglich auszutauschen. Weiterhin wird `IP_MTU_DISCOVER` verwendet, um die Maximum Transfer Unit (MTU) der Route herauszufinden [132, S. 344]. Alle Daten, ob über TCP oder UDP werden von `RUDP` mit dieser MTU versandt, um den Overhead gering zu halten, d. h. es werden die Nutzdaten fragmentiert in einzelnen UDP-Paketen versandt. Eine weitere Fragmentierung der IP-Pakete wird durch das Setzen des `DF`-Bit (engl. für „Don’t Fragment“) für ausgehende Pakete verhindert[6].

[6] Änderungen der MTU, bspw. durch wechselnde Routen, werden damit nicht abgefangen.

Der einmalig gestartete `write_thread` arbeitet in einer sehr kompakten, optimierten Schleife die angewiesenen Pakete ab (editiert, gekürzt):

```
... /* Loop over the currently assigned r->write_ids_num packets          */
for (p = 0; r->write_ids_num > 0; p++, r->write_ids_num--) {
  msg.msg_iov[0].iov_base = &(r->write_ids[p]);     /* ID within Buffer    */
  msg.msg_iov[0].iov_len  = sizeof (int32_t);       /* ID is 32bit value   */
  msg.msg_iov[1].iov_base =
    r->buffer +                                     /* Start of Buffer     */
    packet_size*(r->write_ids[p]-r->buffer_low_id); /* Offset from Window  */
  msg.msg_iov[1].iov_len = packet_size;             /* Length of packet    */
  /* Busy loop for RUDP_INITIAL_TIME_SENDMSG of > 30                       */
  if (write_ids_wait > 30) {
      now = gettimenow();
      while (now - time_last_send < write_ids_wait) /* Wait to send packet */
          now = gettimenow();
  }
  ret = sendmsg (r->fd_udp, &msg, 0);               /* Send next packet    */
  time_last_send = now;                             /* Store the send time */
}
...
```

Der Empfangsthread hat eine ähnliche Struktur wie der `write_thread`. Hier werden Leseaufrufe für Pakete direkt an der erwarteten Stelle im Puffer des Aufrufers abgesetzt. Nur wenn Pakete außerhalb der Reihenfolge empfangen werden, müssen diese kopiert werden, ansonsten werden keine Pakete kopiert. Wird ein Paket außerhalb des erwarteten Fensters empfangen, wird es verworfen. Im Normalfall werden Pakete immer in der korrekten Reihenfolge empfangen. Anschließend wird die Nummer des korrekt empfangenen Paketes an die Acknowledgement-Informationen zum späteren Versand an den Sender angehängt.

Der meiste Aufwand liegt in der Verwaltung des gleitenden Fensters (engl. Sliding Window) durch den Hauptthread. Dies ist mit mehreren Bitfeldern effizient implementiert, je eines für die versendeten, bzw. korrekt empfangenen Pakete, sowie für die Pakete die durch einen Timeout wieder versandt wurden. Abbildung 3.4 zeigt das Sliding Window für einen Ausschnitt des Puffers. Zu Demonstrationszwecken hat das Fenster hier eine Größe von nur vier Paketen, wovon drei „auf dem Kabel liegen“: Während die Pakete 200 bis 202 und 204 bereits als bestätigt markiert sind („geackt“), stehen die Pakete 203, sowie 205 bis 209 noch aus. Sobald Paket 203 geackt wird, kann das Fenster um zwei Pakete nach rechts weiterrutschen und Pakete 207 und 208 dem Schreibthread zuweisen. Dementsprechend durchwandern die Pakete sieben diskrete Zustände, wie in Abb. 3.4 (unten) anhand des Sliding Windows dargelegt.

Die Acknowledgement-Informationen, die vom Empfänger zum Versender geschickt werden, sollten eine möglichst große Anzahl von korrekt versandten, aber auch verlorener Pakete aufzeigen. In TCP wird das zuletzt korrekt empfangene Byte bestätigt, verloren gegangene Daten werden über den Timeout, bzw. durch den Fast Retransmit

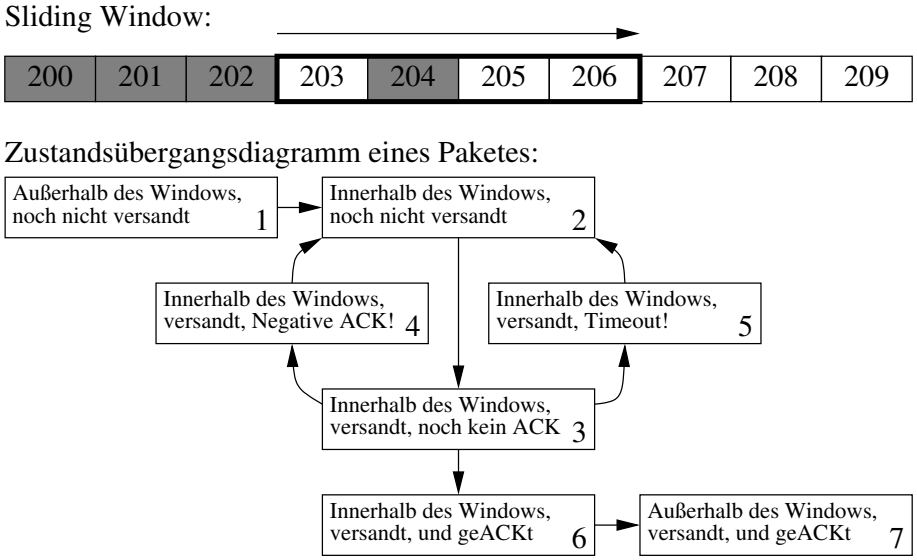

Abbildung 3.4: Übergangsdiagramm der Pakete beim Sliding-Window Protokoll

Algorithmus mittels eines dreifach-versandten duplizierten ACKs (engl. Triple Duplicate ACK) für das fehlende Segment erkannt.

In dem hier implementierten Protokoll können die ACK-Informationen als Paket-Bereiche beschrieben werden. Genauer gesagt durch eine festgelegte Anzahl von korrekt empfangenen und nicht korrekt empfangenen Bereichen, sowie eine festgelegte Anzahl von einzelnen Paketnummern, auch hier wieder für korrekt empfangene und nicht korrekt empfangene Pakete. Da fehlerbehaftete Pakete mittels des CRC erkannt und durch das Betriebssystem verworfen werden, wird in diesem Protokoll ein fehlendes Paket durch einen Negative ACK (NACK) als verloren markiert.

In Abb. 3.5 ist ein Acknowledgement-Paket für das obige Beispiel dargestellt; die Pakete 205 und 206 werden in einem ACK-Range, das Paket 203 als Single-ACK codiert. In der Regel werden ACK-Pakete bis zur MTU-Größe aufgefüllt, spätestens aber zu einem nach einem per Parameter wählbaren Zeitpunkt versandt, um das Fenster weiterzutreiben. Bei Empfang der ACK-Information kann der Sender vier weitere Pakete schicken.

Die Paketübertragung regeln verschiedene Parameter, unter anderem die:

- Größe des Sliding Windows in Anzahl Paketen (default: 4000).
- Anzahl der Pakete, die an den Schreib-Thread übergeben werden (default: 512).
- Anzahl der ACK-Paketbereiche (default: 4).
- Zeit für den Timeout in Sekunden, wann Pakete als verloren markiert werden (default: 1 s).

Diese Parameter sollten je nach Netzwerkgeschwindigkeit angepasst werden. Erfahrungswerte für Gigabit-Ethernet sind in den Default-Werten oben vermerkt.

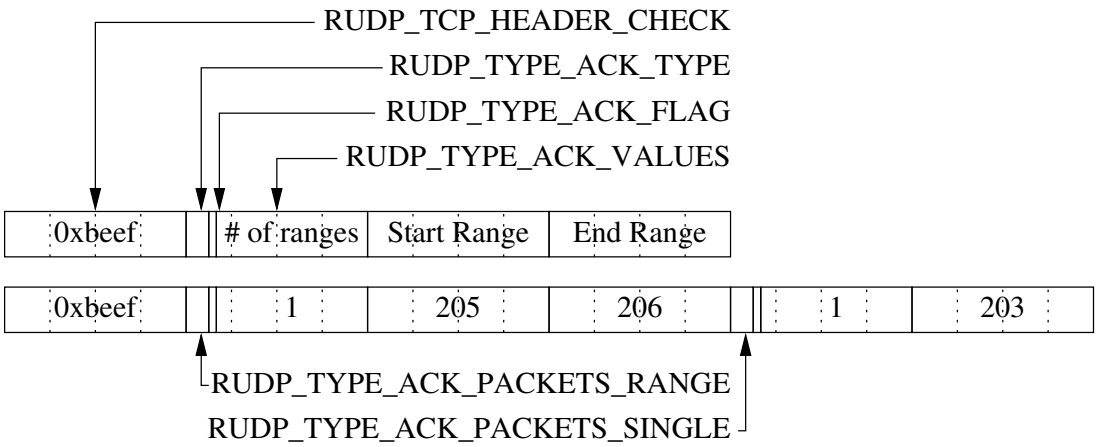

Abbildung 3.5: Acknowledgement Pakete am Beispiel des Sliding-Window Protokolls

3.2.2 Messungen der UDP/IP Netzwerkübertragung

Um gute Anfangsparameter herauszufinden, wurden Messungen über Netzwerke mit verschiedenen Parametern gemacht. Diese reichen von 10 MBit und 100 MBit zu 1 Gbit-Ethernet, Messungen über das europäische Hochgeschwindigkeitsnetz Géant und Weitverkehrsnetzen von Stuttgart nach Australien.

In diesem Abschnitt sollen beispielhaft die Ergebnisse des schnellen LANs und die Messungen nach Australien vorgestellt werden.

Messungen im LAN über Gigabit-Ethernet

Das Gigabit-Ethernet-LAN zwischen zwei Standard Linux-PCs wurde als Referenzmessung genommen, da hier eine klar definierte Topologie mit festen Parametern (NIC, Switch, Betriebssystem und Last) vorliegt.

Vor den eigentlichen Messergebnissen soll die theoretisch mögliche Übertragungsbandbreite analysiert werden: Geht man optimistisch von der theoretisch möglichen Übertragungsrate von Gigabit Ethernet mit 125 $^{\mathrm{MB}}/_{\mathrm{s}}$ aus und von 1518 Byte Ethernet-Frame, bleiben 1460 Byte für Nutzdaten[7]. Hieraus lässt sich die theoretische Bandbreite für Nutzdaten berechnen. Es wären also 81380 Frames pro Sekunde mit UDP zu versenden, also alle 0,012288 ms ein Paket. Dies erfordert eine sehr genaue Taktung. In Abschnitt 6 im Anhang wird die Problematik hochgenauer Zeitmessung und die in dieser Arbeit verwendeten Methoden vorgestellt. Um die geforderte Taktung zu erreichen und den Overhead der Zeitmessung gering zu halten, wird hierbei die Messung auf Basis des Prozessortaktes gemacht, wenn es das System erlaubt (siehe Routine `gettimenow` im Programmausschnitt des Schreibthreads).

Der Vollständigkeit halber werden die Voreinstellungs- und Maximalwerte für einige performancerelevanter Parameter aufgeführt. Hierzu zählen die Ringpuffer für Pakete

[7] 1518 Byte pro Ethernet Frame, davon 1500 Byte für das IP-Paket, hiervon 1480 Byte für das UDP-Paket, hiervon wiederum 1460 Byte für Nutzdaten.

Rechner	**Pcglap12**	**James**
Prozessor	Intel PentiumM, 1,7 GHz	Intel Pentium4, 3 GHz
NIC	Gb-Ethernet, Intel 82540EP	Gb-Ethernet, Intel 82540
Betriebssystem	Suse 10.1, Kernel-2.6.16.27	Suse 9.3, Kernel-2.6.11
Libc	glibc-2.4	glibc-2.3.4
Compiler	gcc-4.1.0	gcc-4.1.0

Tabelle 3.1: Konfiguration für die Messung von `RUDP` zwischen zwei Linux-PCs

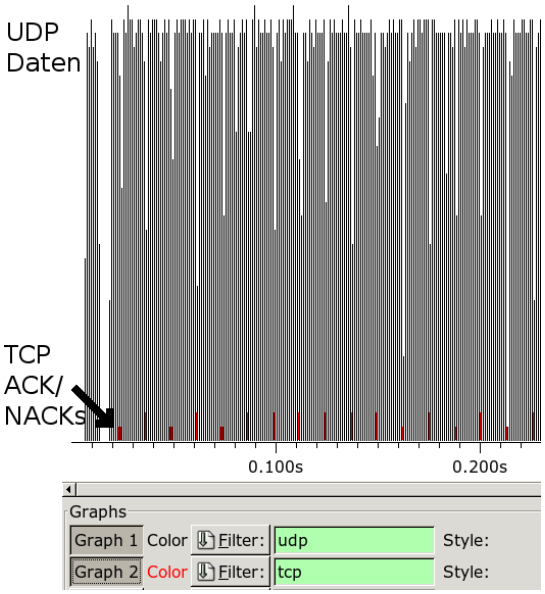

Abbildung 3.6: Netzwerktrace der TCP- und UDP-Pakete mit Ethereal

im Receive-Buffer (Default: 256, Maximal:4096), Duplexkommunikation (Default: An), Offloading von Checksummenberechnung auf die Hardware (Default: für RX, TX, an). Die Konfiguration der für die Messungen verwendeten PCs ist in Tab. 3.1 aufgeführt, als Switch wurde ein handelsüblicher Büroswitch D-Link Gigabit Switch DGS-1008D verwendet. Das Interesse galt hier den detaillierten Netzwerkmessungen mit Tools wie Wireshark (früher Ethereal) [27], welches alle versandten und empfangenen Pakete des NICs protokolliert.

Diese Analysen zeigen deutlich die Sensitivität der Übertragungsgeschwindigkeit auf die TCP-Acknowledgements. Abbildung 3.6 zeigt einen Netzwerktrace des unten vorgestellten ersten Szenarios mit zwei Linux-PCs. Hier wurde die Anzahl der ACKs pro TCP-Paket auf Single-Acks begrenzt (entsprechend 1452 Bytes / 4 Bytes = 363 ACKs pro TCP-Paket)[8], um den notwendigen TCP-Verkehr zu erhöhen. Ethereal zeigt in dem Trace die versandten Bytes gefiltert nach Protokoll UDP (schwarze dünne Linie) und TCP (rote dicke Linie) an. Da für die TCP-Verbindung die Option `TCP_NODELAY` ge-

[8] 1460 Bytes für Nutzdaten laut MTU, minus 4 Byte für den Headercheck und 4 Byte Typdeklaration/Anzahl Single-ACKs.

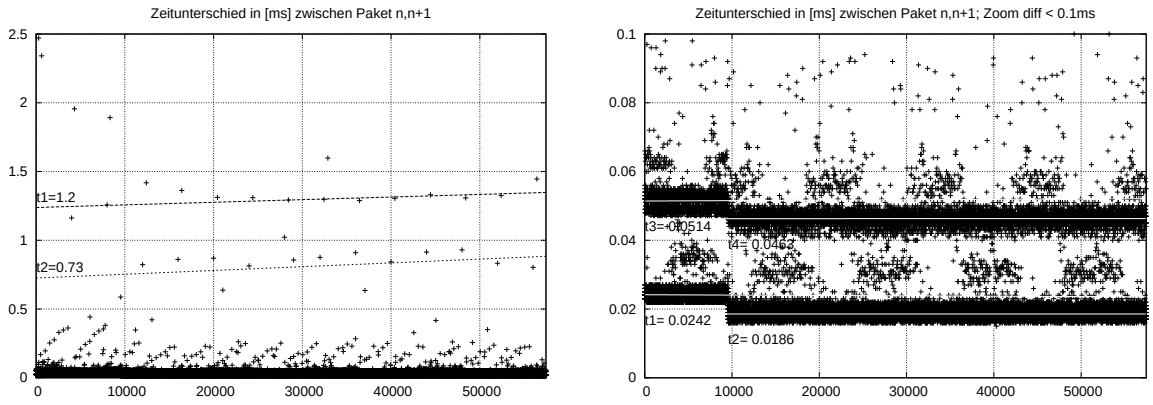

Abbildung 3.7: Zeit zwischen dem Versand zweier aufeinanderfolgender UDP-Pakete

setzt ist, werden TCP-Pakete sofort versandt. Werden ACK-Pakete via TCP versandt, bricht die Übertragung von UDP-Paketen ein. Ob dieses Verhalten dem Switch, den Netzwerkkarten oder dem Betriebssystem geschuldet ist oder ob es daran liegt, dass ein Einprozessorsystem die Abarbeitung der in Threads implementierten Empfangsroutinen durchführt, kann nicht ohne Weiteres gesagt werden.

In Abb. 3.7 ist die Zeitdifferenz zwischen zwei aufeinanderfolgenden UDP-Paketen beim Schreibthread protokolliert und am Ende des Testlaufs gespeichert. Es zeigt sich, dass von 57455 Paketen 57256 innerhalb von 0,1 ms versandt werden, dass es aber auch Ausreißer mit bis zu 2,47 ms Zeitdifferenz beim Versand gibt. Diese Ausreißer treten periodisch nach 512 Paketen auf und resultieren aus dem notwendigen `pthread_mutex_lock` und `pthread_mutex_unlock` um einen neuen Abschnitt an zu übertragenden Paketen abzuholen.

Um die Bänder in den Messdaten weiter zu analysieren, wurden mehrere lineare Funktionen $f(x) = a * x + b$ in die Punkte gelegt und mit `gnuplot` mit einer nicht-linearen Methode der kleinsten Fehlerquadraten die Parameter a und b gefittet. Während a für die Steigung von f steht, bezeichnet b die gewichtete Zeitdifferenz in den Messdaten. Deutlich sind in Abb. 3.7 (links) die wegen der beschriebenen Synchronisation verzögerten Pakete zu erkennen. Diese Ausreißer liegen in einer Stufe von 1,2 ms[9] und einer Stufe von 0,73 ms[10]. Bei der Vergrößerung in der Abb. rechts erkennt man, dass nach einer kurzen Phase des Einpendelns (nach ca. 9550 Paketen) mit 66% die meisten der Pakete in einem Band mit einer Zeitdifferenz von ca. 0,0186 ms[11], bzw. mit 15% der Paketen von ca. 0,0463 ms[12] versandt werden. Eine Erklärung für das Einpendeln nach 14 MB übertragenen Daten kann nicht eindeutig gegeben werden.

Mit der im Protokoll gewählten Sendrate von 0,0186 ms pro Paket liegt der ermittelte

[9] Band 1-1,5 ms: 13 Punkte, Differenz $b \equiv \varnothing\, 1,24, \sigma \pm 0,055$, Steigung $a \equiv \varnothing\, 1,90 * 10^{-6}, \sigma \pm 1,62 * 10^{-6}$.

[10] Band 0,5-1 ms: 15 Punkte, Differenz $b \equiv \varnothing\, 0,73, \sigma \pm 0,066$, Steigung $a \equiv \varnothing\, 2,75 * 10^{-6}, \sigma \pm 1,88 * 10^{-6}$.

[11] Band 0,014-0,025 ms: 38146 Punkte, Differenz $b \equiv \varnothing\, 0,0186, \sigma \pm 1,852 * 10^{-5}$, Steigung $a \equiv \varnothing\, 2,12 * 10^{-11}, \sigma \pm 5,11 * 10^{-10}$.

[12] Band 0,0398-0,052 ms: 8455 Punkte, Differenz $b \equiv \varnothing\, 0,0463, \sigma \pm 5,391 * 10^{-5}$, Steigung $a \equiv \varnothing\, 4,38 * 10^{-10}, \sigma \pm 1,49 * 10^{-9}$.

Defaultwert der obigen Übertragung 50% über dem errechneten theoretischen Wert von 0,012288 ms. Hierbei muss man bedenken, dass kein anderes Protokoll die theoretische Bandbreite von 118 MB/s erreicht[13].

Insgesamt überträgt RUDP in dieser Konfiguration zwischen den zwei Linux-PCs (siehe Tab. 3.1) über Gigabit-Ethernet immerhin 60-62 MB/s, während TCP mit großen Sendepuffern von 1024 kB 68 MB/s und mit Sabul/UDT 58 MB/s erreicht. Reduzierend auf die Übertragungsrate wirkt sich der relativ kleine Puffer von 9 kB auf dem NIC aus, sowie mit 32 kB auf dem D-Link Switch. TCP hat hierbei den Vorteil, dass die Daten als Ganzes dem Kernel zur Übertragung angewiesen werden, d. h. innerhalb des Kernels wird die TCP-Schicht direkt dem Treiber der Gigabit-Ethernet Karte transferieren, evtl. ohne Daten zu kopieren (engl. zero-copy Networking).

Messungen mit Australien

Um die Kapazitäten von RUDP auch in Weitverkehrsnetzen zu testen, wurde eine Bandbreitenmessung zwischen zwei PCs, einem am HLRS und einem in Australien gemacht und in [81] vom Autor beschrieben. Für echte gekoppelte Anwendungen machen diese Weitverkehrsmessungen wegen der hohen Latenz keinen wirklichen Sinn, dennoch kann damit die Leistung von RUDP demonstriert werden. Wie schon für die Messung im LAN auf Seite 48 soll vor den eigentlichen Messungen eine Analyse der theoretisch möglichen Performance gegeben werden. Da keine verlässliche Information über die genaue Bandbreite vorliegt, wird die RTT und die Latenz analysiert.

Die direkte Entfernung zwischen dem Rechner am HLRS und dem PC an der Australian National University in Canberra beträgt 16429 km Luftlinie[14]. Signale wären mit der Lichtgeschwindigkeit (c) von 300.000 km/s nur 54 ms unterwegs, damit wäre die Leitung für Pakete bis zu der theoretischen RTT von 108 ms zu füllen (vorausgesetzt, es wird mit Lichtgeschwindigkeit übertragen, und die Route für rückgesendete IP-Pakete ist die gleiche wie für hinwärtsgesendete). In einer durchgehenden Glasfaserleitung würden die Photonen auf 66% der Lichtgeschwindigkeit abgebremst, bei einer direkten Strecke dieser Länge läge die RTT immer noch bei 164 ms.

Tatsächlich aber verläuft die Netzwerkleitung von Frankfurt über Singapur, dann Perth und Adelaide bis Canberra mit insgesamt 16 Switches, die zur Latenz beitragen. Im Folgenden ist die mit `mtr` gemessene Route aufgezeigt. Es werden die Anzahl der ICMP Pakete, die Verlustrate, die letzte gemessene RTT sowie Durchschnitts, Worst- und Best-case und Standardabweichung angegeben. Es wird offensichtlich, daß der Übergang vom Gigabit-Ethernet-Router in Frankfurt (`fra`) nach Singapur (`sin`) einen Sprung von 8,5 ms auf 219,5 ms und von Singapur nach Perth (`per`) einen Sprung auf 273,3 ms ausmacht. Weiterhin zeigen einige Router eine hohe Standardabweichung auf. Dies deutet auf eine Überlastung einzelner Routen oder die niedrige Priorität von ICMP-Antworten in einzelnen Routern hin. Der Router in Canberra (`cbr`) hat bspw. höhere Latenzen als der Router in Melbourne (`mel`) oder der Zielrechner.

[13] Von 125 MB/s sind bei TCP/IP nur 95% für Nutzdaten übrig. Eine (betriebssystemabhängige) Lösung für Benutzer mit Administratorrechten wäre Ethernet direkt zu verwenden.

[14] Laut http://www.mapcrow.info, basierend auf http://maps.google.com.

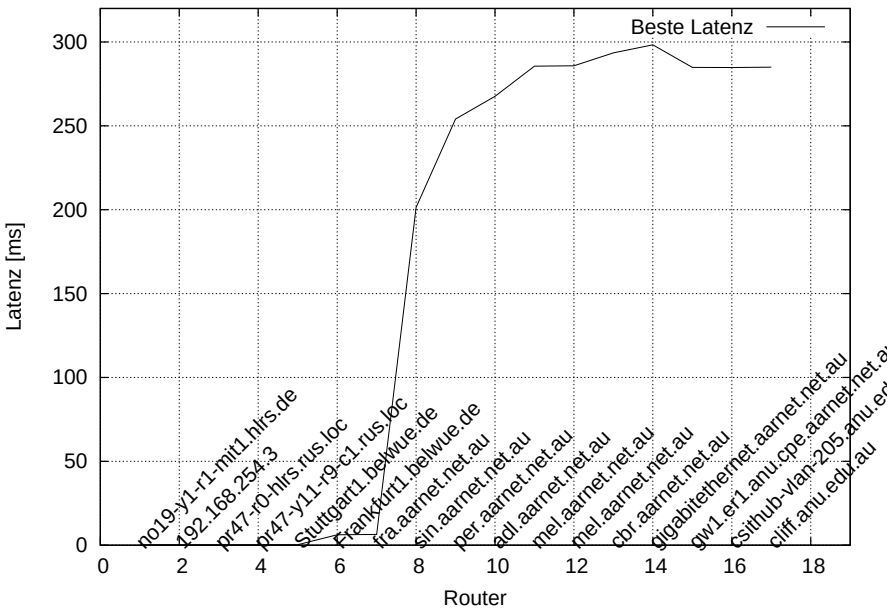

Abbildung 3.8: Latenzen auf der Strecke nach Australien

Die RTT zum Endsystem `cliff.anu.edu.au` beträgt schlussendlich bestenfalls 285 ms.

		Packets		Pings				
	Host	Loss%	Snt	Last	Avg	Best	Wrst	StDev
1.	no19-y1-r1-mit1.hlrs.de	0.0%	70	0.7	0.8	0.3	5.1	0.7
2.	192.168.254.3	0.0%	70	0.5	0.6	0.3	5.7	0.7
3.	pr47-r0-hlrs.rus.loc	0.0%	70	1.2	1.1	0.3	25.6	3.1
4.	pr47-y11-r9-c1.rus.loc	0.0%	70	0.7	6.6	0.3	200.2	26.9
5.	Stuttgart1.belwue.de	0.0%	69	0.5	0.5	0.3	2.3	0.3
6.	Frankfurt1.belwue.de	2.9%	69	6.5	14.0	6.2	201.8	31.2
7.	ge-3-2-0.bb1.a.fra.aarnet.net.au	0.0%	69	6.9	8.5	6.4	79.4	9.7
8.	so-0-1-0.bb1.a.sin.aarnet.net.au	0.0%	69	201.5	219.5	201.4	1152.	114.0
9.	so-3-3-0.bb1.a.per.aarnet.net.au	0.0%	69	268.1	273.3	254.1	1092.	100.2
10.	so-0-1-0.bb1.a.adl.aarnet.net.au	0.0%	69	268.1	274.3	267.6	282.5	5.2
11.	so-0-1-0.bb1.a.mel.aarnet.net.au	0.0%	69	299.7	293.3	285.6	300.3	5.5
12.	ge-0-0-0.bb1.b.mel.aarnet.net.au	0.0%	69	299.6	294.0	285.8	300.5	5.3
13.	so-0-1-0.bb1.b.cbr.aarnet.net.au	0.0%	69	298.4	301.8	293.5	311.0	5.0
14.	gigabitethernet....aarnet.net.au	0.0%	69	312.2	308.6	298.3	424.1	17.6
15.	gw1.er1.anu.cpe.aarnet.net.au	0.0%	69	298.8	293.3	284.9	327.4	6.9
16.	csithub-vlan-205.anu.edu.au	0.0%	69	291.3	452.3	284.8	11370	1333.
17.	cliff.anu.edu.au	0.0%	69	312.3	459.3	285.0	11310	1325.

Die Abb. 3.8 stellt die beste Latenz der 69 `ICMP`-Paketen mit Messungen auf der Route dar. Der Anstieg von 6,4 ms auf 201,4 ms beim Übergang vom Router in Frankfurt nach Singapur ist nicht nur mit der Entfernung zu erklären, vielmehr rührt dies von mehreren auf der Strecke liegenden Repeater her. Da diese auf einer niederen ISO-Schicht liegen,

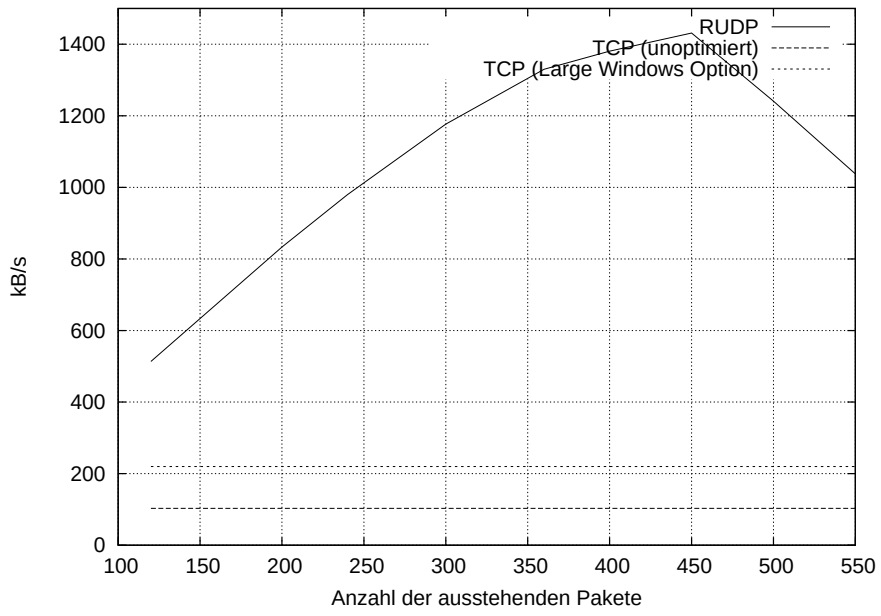

Abbildung 3.9: Bandbreitenmessung zwischen PCs in Deutschland und Australien

wird die gesetzte Time-to-Live (TTL) des Paketes nicht reduziert und damit keine `ICMP`-Antwort generiert. Deswegen sind diese Repeater für `mtr` nicht sichtbar. Nimmt man die Latenz der Verbindung Singapur-Perth (Latenzdifferenz von 52,7 ms bei 3911 Kilometer Luftlinie, d. h. 13,5 $^{\mu s}/_{km}$) als theoretischen Massstab, könnte man für die Verbindung Frankfurt-Singapur (9829 Kilometer) eine theoretische Latenz von 132 ms annehmen. Die gemessene Latenzdifferenz von 195 ms liegt somit 50% höher.

Die mit dem Tool `mrt` zum Zeitpunkt der Bandbreitentests gemessene Round Trip Time (RTT) von 326 ms, entsprechend einer Latenz von 163 ms. Für die Bandbreitenmessungen wurde ein 14 MB großer Puffer versandt. Mit einer nicht-optimierten TCP-Verbindung wurde zwischen den PCs 103 $^{kB}/_{s}$, mit einer optimierten TCP-Verbindung[15] immerhin 220 $^{kB}/_{s}$ gemessen. Um den Zusammenhang zwischen der Window-Size des Fensterprotokolls zu demonstrieren, wurde für `RUDP` die Anzahl der ausstehenden Pakete variiert. Abbildung 3.9 zeigt die erzielten Übertragungsraten, mit 450 Paketen auf dem Kabel ergibt sich ein Maximum für das UDP-Protokoll von 1,43 $^{MB}/_{s}$. Dies ist im Vergleich zur Standard TCP-Verbindung eine 14-fache Verbesserung der Übertragungsrate.

3.2.3 Diskussion der Implementierung

Prinzipiell sind UDP-basierte Protokolle zu kritisieren, die nicht die Senderate anpassen oder nach beliebigen Vorgaben des Aufrufers/Benutzers Pakete versenden (Paket-Storms). Die Flow Control von TCP ist nicht umsonst sehr konservativ angelegt. Nur damit ist der Transfer von vielen TCP-Streams mit niedriger Verlustrate und damit das

[15] Mit einer maximalen Window-Size von 1024 kB.

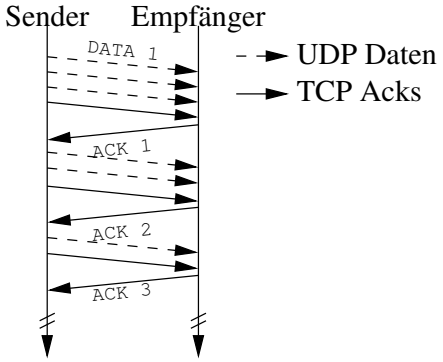

Abbildung 3.10: Kommunikation im Reliable Blast UDP-Protokoll

Funktionieren des Gesamtsystems gewährleistet. Schon jetzt werden Peer-to-Peer (P2P)-Protokollströme bei großen Providern gefiltert und limitiert. Eine starke Verbreitung von UDP-Protokollen ohne Flow Control könnten das Gesamtsystem belasten und würden ebenfalls gefiltert werden.

Bezüglich des Überholens von Datenpaketen (engl. Reordering) gibt es unterschiedliche Meinungen in der Wissenschaft: in [136, auf Seite 47] und [137] erläutern die Autoren, dass das Problem von überholenden Datenpaketen nicht betrachtet werden muss, da die Wahrscheinlichkeit gering und die Kosten für Kopieraktionen relativ hoch seien. Für die dort untersuchten drahtlosen Netzwerke (Wireless LANs) mag dies gelten – für Weitverkehrsnetze tritt aber in der Tat Reordering durch unterschiedliches Routing auf. Durch optimierte Kopieroperationen für Pakete weniger Kilobyte lassen sich bereits übertragene Daten schnell an die korrekte Adresse im Benutzerpuffer kopieren. Es wäre unsinnig, vertauschte Pakete erneut anzufordern oder durch einen Timeout beim Sender erneut verschicken zu lassen, nur um die Reihenfolge zu garantieren.

Weiterhin wird in [135] die tatsächliche Datenfehlerrate in TCP-Streams untersucht. Dabei wird festgestellt, dass bis zu eines von 1000 Paketen einen Fehler enthält (meist durch fehlerhafte Implementierungen in der Netzwerksoftware der Switche), diese aber durch die 32-Bit Checksumme erkannt werden. Dennoch werden einige fehlerhafte Pakete vor allem bei großen Paketen nicht durch diese Checksumme erkannt. Es wird in [135] empfohlen, die Nutzdaten durch eine weitere Checksumme zu sichern und zu korrigieren.

Reliable Blast UDP

Das Realiable Blast UDP-Protokoll [63] hat das Ziel, große Nachrichten am Stück an den Empfänger zu übertragen. RBUDB ist eine in C++ implementierte Bibliothek. Hierfür wird die Kommunikation in zwei Phasen eingeteilt. In der ersten werden die Daten mit einer vom Benutzer angegebenen Senderate zum Empfänger geschickt, in der zweiten Phase wird mittels TCP übermittelt, welche Daten aus den diversen Gründen verloren gegangen sind und wiederholt werden müssen. Diese zweite Phase des Abgleichs wiederholt sich so lange, bis schließlich alle Daten komplett angekommen sind.

Der Nachteil dieses einfachen Protokolls ist, dass der Benutzer durch ungeschickte

Wahl der Parameter das Netz oder den Empfänger überlasten kann. Weiterhin geht die Zeit zwischen den Phasen für Kommunikation verloren, wie in Abb. 3.10 zu sehen ist. Optional kann in nachfolgenden Phasen die Senderate in Bezug auf die derzeitige Verlustrate (engl. lossrate) angepasst werden. Dies geschieht gemäß der Formel:

$$if(lossRate > 0)\ R_{new} = R_{old} * (0.95 - lossRate);$$

Es fällt auf, dass die Senderate sobald einmal reduziert nicht wieder erhöht wird. Durch eine temporär hohe Auslastung des Switches, bei dem Paketverlust auftritt, kann also den gesamten nachfolgendne RBUDP Verkehr ausbremsen.

Tsunami

Mit Tsunami [93] steht eine Bibliothek für den Transfer von großen Dateien zur Verfügung. Hier werden separate Threads auf der Empfänger-Seite verwendet, um die Daten zu empfangen und entkoppelt auf der Festplatte zu speichern. Der Sender ist durch einen Thread implementiert.

Um die Senderate zu limitieren, kann der Benutzer einen Schwellenwert für die Verlustrate der UDP-Pakete angeben. Liegt die Verlustrate höher, passt das Protokoll in regelmäßigen Abständen senderseitig die Paketrate an. Allerdings steht für Tsunami keine Programmierschnittstelle für Applikationen zur Verfügung. Somit kann es leider nicht in PACX-MPI eingebaut und getestet werden.

Sabul/UDT

Sabul [59] ist das ausgereifteste der Protokolle und am ehesten mit der hier entwickelten Bibliothek vergleichbar. Ähnlich wie `RUDP` öffnet auch die in C++ implementierte Bibliothek Sabul eine TCP-Verbindung zur Übertragung der Acknowledgements. In dem Nachfolger von Sabul namens UDT werden auch die Acknowledgements via UDP versandt. In diesem Protokoll werden Acknowledgement-Informationen per Piggybackverfahren in den Daten versandt. Damit sind die Möglichkeiten begrenzt, ACK-Daten zu speichern. Dennoch werden selektive Acknowledgements wie in `RUDP` versandt. Auch in UDT verwirft der Empfänger Pakete außerhalb der Reihenfolge nicht, sondern puffert und ordnet sie. Die Senderate wird bei Sabul/UDT dynamisch durch die Messung der RTT mittels spezifischer Acknowledgements angepasst. Es wird auch versucht, die Varianz der RTT für die Vorhersage der zukünftigen Senderate hinzuzunehmen. Wie in TCP wird die Senderate mittels eines AIMD-Algorithmus (Additive Increase Multiplicative Decrease) langsam gesteigert und bei Paketverlust in UDT mit einem Faktor von $0,9$ stark reduziert. In den Messungen war der Congestion Control Mechanismus aber sehr sprunghaft.

Leider erlaubt eine in C++ implementierte Bibliothek keine plattformübergreifende Möglichkeit für den Aufruf aus C-Bibliotheken, wie bspw. PACX-MPI.

4 Middleware – PACX-MPI

Damit eine parallele Anwendung performant läuft, muss auch die Middleware, hier PACX-MPI, optimiert werden. Im Folgenden werden Verbesserungen der vorhandenen Funktionalität von PACX-MPI auf der algorithmischen Ebene erläutert. Der Funktionsumfang von PACX-MPI ist in der Dissertation von Gabriel [48] beschrieben. Ausgehend von diesem Stand der Software wurden die hier beschriebenen Techniken implementiert.

Wie in Abschnitt 2.3.4 beschrieben, benutzt PACX-MPI für die Kommunikation zwischen lokalen Prozessen die native MPI-Implementierung. Meist ist dies die für die Hardware optimierte Bibliothek des Herstellers des Rechners oder des Netzwerkes (hier engl. Vendor-MPI genannt).

Kommunikation zu externen Prozessen wird wie beschrieben über die beiden so genannten Dämonen, PACX-Outdaemon und PACX-Indaemon, durchgeführt. Diese beiden als MPI-Prozesse implementierten Dämonen sind damit für die Anwendung nicht sichtbar[1]. Dies erfordert, dass in einer dünnen Schicht bspw. die Prozessorranks in der Ordnung von `PACX_COMM_WORLD` auf die Ordnung von `MPI_COMM_WORLD` umgerechnet werden. Darüberhinaus besitzt jede von MPI definierte Struktur ein Äquivalent in PACX-MPI. Strukturen wie Kommunikatoren, Gruppen, Datentypen, Operatoren, Requests und Stati müssen in PACX-MPI gekapselt und verwaltet werden. Diese Umsetzung der Prozessorranks und Verwaltung der Strukturen ist nicht algorithmisch nicht besonders aufwändig. In dieser Arbeit soll PACX-MPI vielmehr auf algorithmischer und funktionaler Ebene verbessert werden. Neben der hier nicht vorgestelten IMPI-Implementierung, wird PACX-MPI in Bezug auf kollektive Kommunikation verbessert, sowie auf funktionaler Ebene mit neuartigen Performanceanalysemöglichkeiten verbessert. Um praktische Software-technische Probleme bei der Bibliotheksentwicklung zu entdecken, wird eine neu entwickelte MPI-Testsuite vorgestellt, die Probleme in PACX-MPI, Open MPI und anderen Implementiernug aufdecken konnte.

4.1 Optimierung von kollektiven Kommunikationsoperationen

Punkt-zu-Punkt (engl. Point-to-point, kurz P2P) Kommunikation in PACX-MPI innerhalb des lokalen Hosts wird direkt mittels das optimierte Vendor-MPI durchgeführt. Deswegen gibt es wenig Möglichkeiten die interne Kommunikation zu verbessern. Liegt der Empfänger der Nachricht hingegen nicht innerhalb dieses Hosts, wird ein die Kommunikation beschreibender Header an den PACX-Outdaemon, gefolgt von den Nutzda-

[1] Da die Dämonen nicht aus der Funktion `MPI_Init` in die Anwendung zurück springen.

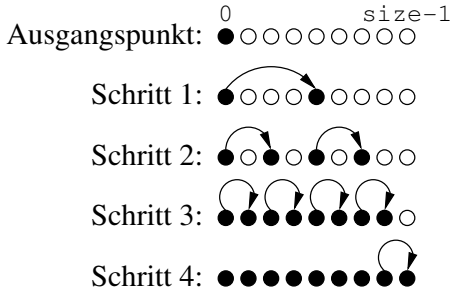

Abbildung 4.1: Kommunikationsschritte der Funktion `MPI_Bcast` als Binärbaum

ten an den PACX-Outdaemon gesandt. Hierfür müssen die Nutzdaten evtl. konvertiert werden, wenn die Datenrepräsentation auf dem Zielsystem eine andere ist, wie in dem Beispiel von Abb. 2.11 (hier ist die Kopplung eines Linux-Clusters mit der Vektormaschine NEC-SX8 gezeigt). Der Transfer von und zu den Dämonen sollte durch MPI mit der größtmöglichen Performance ablaufen[2]. Optimierungen der Kommunikation über das verbindende Netzwerk wurden im vorherigen Kapitel besprochen.

Für die Implementierung der kollektiven Kommunikation jedoch sollte PACX-MPI die hierarchische Struktur des Metacomputers in Betracht ziehen. Mehrfache Kommunikation und Synchronisation über externe Netzwerkleitungen sind zu vermeiden. Auch hier gilt, dass kollektive Routinen soweit wie möglich auf den optimierten kollektiven Funktionen des Vendor-MPIs basieren sollten.

4.1.1 Implementierung von `MPI_Bcast`

Für kollektive Funktionen, wie bspw. `MPI_Bcast` gibt es mehrere Kommunikationsalgorithmen, die je nach Netzwerktopologie, sowie -hardware und Größe des zu versendenden Puffers unterschiedliche Performance liefern [71, 108]. So wären für `MPI_Bcast` die folgenden Kommunikationsmuster denkbar (zur Vereinfachung nehmen wir an, der *root*-Prozess sei 0 innerhalb des Kommunikators mit *size* Prozessen):

- Insgesamt n Sendeoperationen in linearer Reihenfolge von Prozess 0 zu 1, dann von 1 zu 2, und so weiter bis schließlich zu $size - 1$.
- Eine baumartige Kommunikationsstruktur, bspw. im Binärbaum: bei Schritt eins eine Nachricht von Prozess 0 zu $\lfloor (size - 1)/2 \rfloor$, im Schritt zwei insgesamt zwei Nachrichten, je eine von Prozess 0 zu $\lfloor (size - 1)/4 \rfloor$ und von $\lfloor (size - 1)/2 \rfloor$ zu $\lfloor 3(size - 1)/4 \rfloor$ und so fort, wie in Abb. 4.1 dargestellt.
- Eine Kombination aus den beiden vorherigen Algorithmen, engl. Fractional Tree genannt [117].
- Falls vom Verbindungsnetzwerk unterstützt, durch hardwarebasierte Broadcasts (bspw. bei Ethernet oder Infiniband).

[2] Anderenfalls liegt das Problem in der Implementierung des Vendor-MPIs [138].

Es ist auch eine Kombination aus den obigen Algorithmen denkbar, sowie evtl. Pipelining einzelner Nachrichten, um das Netzwerk besser auszulasten [118]. Für die Untersuchung und Performancemessung der verschiedenen Kommunikationsmuster von kollektiven Routinen existieren eine Reihe von Mikrobenchmarks. Wie bei diesen Mikrobenchmarks üblich liefern sie in Abhängigkeit der Kommunikationsreihenfolge stark variierende Ergebnisse. Je nach Netzwerktopologie kann das eine oder andere Kommunikationsmuster bessere Performance liefern, siehe Abschnitt 6 über Netzwerktopologien im Anhang. Beispielsweise ist der Test für `MPI_Bcast` im Intel MPI-Benchmark (kurz IMB, ehemals Pallas MPI Benchmark) als Schleife mit dem Root über 0,..., $size - 1$ implementiert, d. h. die Messung erfolgt über eine Schleife von 10000 `MPI_Bcast`s. Diese `MPI_Bcast`s laufen jeweils mit einem andere Root-Prozess ab. Somit werden triviale Implementierungen mit einem einfachen linearen Send bevorzugt, da der Benchmark in sich schon Pipelining betreibt.

4.1.2 Optimierung von `PACX_Alltoall`

Durch seine zweistufige Hierarchie bietet PACX-MPI besonders gute Möglichkeiten zur Optimierung der kollektiven Routinen. Basierend auf der Dissertation von Gabriel [48] wurde die fehlende Optimierung von kollektiven Kommunikationsroutinen, wie `PACX_Alltoall` implementiert. Die `PACX_Alltoall`-Funktion ist für Fast-Fourier Transformationen (FFT) essentiell wichtig. Die ursprüngliche Implementierung setzte für `PACX_Alltoall` die Punkt-zu-Punkt Kommunikation zwischen allen beteiligten Prozessen ein, d. h. es werden $O(n^2)$ Kommunikationsschritte über das verbindende Netzwerk durchgeführt.

Zur Einführung der Begrifflichkeiten wird die optimierte Funktion von `PACX_Bcast` erläutert. Wenn die externe Kommunikationsperformance zwischen den einzelnen Hosts als gleichwertig angenommen und somit keine Topologie im Metacomputer vorausgesetzt wird (siehe auch [84, 85]), kann für `PACX_Bcast` ein offensichtliches Kommunikationsmuster verwendet werden: Ein lineares Send des Root-Prozesses zu jedem der beteiligten so genannten `local_root` Prozesse innerhalb des Kommunikators. Daraufhin führen der globale Root-Prozess, sowie die `local_root`-Prozesse einen lokalen `MPI_Bcast` aus. Am Ende des Austausches hat jeder Prozess im Kommunikator die gleichen Daten erhalten.

Die effiziente Umsetzung von `PACX_Alltoall` unter Benutzung der darunter liegenden MPI-Funktionen ist hingegen weniger trivial. Sie verwendet $O(hosts)$ Kommunikationsschritte über das verbindende Netzwerk sowie auf jedem Host die gleiche Anzahl von `MPI_Gather`- und `MPI_Scatter`-Aufrufen und ein `MPI_Alltoall` für die Host-internen Daten.

Abbildung 4.2 stellt die allgemeine Funktionsweise (siehe MPI-Standard in [94]) und das hier verwendete Kommunikationsschema grafisch für zwei Hosts in einem Metacomputer dar. In diesem Beispiel nehmen fünf Prozesse in dem Kommunikator auf zwei Hosts teil, jeder Prozess übergibt dem `MPI_Alltoall` Daten für jeden anderen Prozess, bspw. liefert der Prozess mit Rang 0 die Werte $(A_1, A_2, ..., A_5)$. Wie in einer transponierten Matrix enthält der Prozess mit Rang 0 nach dem Funktionsaufruf die Werte $(A_1, B_1, ..., E_1)$. Proc_0 und Proc_3 sind die Prozesse mit dem Rang 0 in dem lokalen MPI-

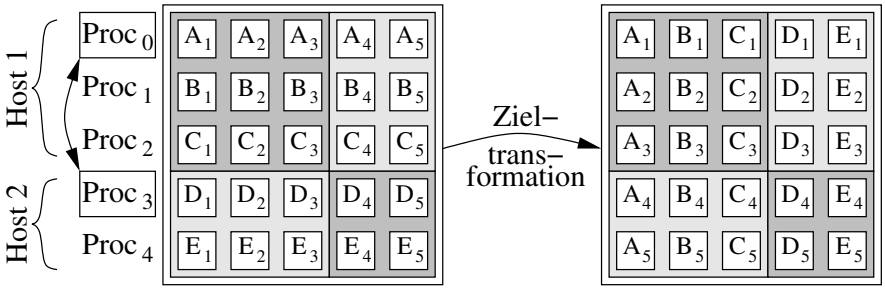

Abbildung 4.2: Kommunikationsschema für die optimierte `PACX_Alltoall`-Funktion

Cluster	cacau	DGrid
Prozessor	Dual Intel Nocona, 3,2 GHz	Dual Intel Woodcrest, 2,66 GHz
Netzwerk	Infiniband SDR, 10 Gbs	Infiniband DDR, 20 Gbs
Schnittstelle	`mvapi-4.1.0`	`ofed-1.1`
Compiler	GNU Compiler gcc-3.4.3	GNU Compiler gcc-3.4.6
MPI	Open MPI-1.2.3	Open MPI-trunk (rev. 16070M)

Tabelle 4.1: Konfiguration der Cluster für den `PACX_Alltoall` Benchmark

Kommunikator. Sie werden wie beschrieben als `local_root` bezeichnet und übernehmen als lokaler Root-Prozess das Sammeln (`MPI_Gather`) und Verteilen (`MPI_Scatter`) der Daten. Zur übersichtlicheren Darstellung sind die `local_root`-Prozesse in Abb. 4.2 umrandet.

Um dies mit möglichst wenig Kommunikationsschritten in PACX-MPI auszuführen, wurde wie folgt vorgegangen:

- Lokal wird je Host ein `MPI_Alltoall` der lokalen Daten abgesetzt, siehe das dunkelgraue Gebiet in Abb. 4.2.
- Jeder `local_root`-Prozess sammelt die entsprechenden lokalen Daten mit `MPI_Gather` (hellgraues Gebiet).
- Die `local_root`-Prozesse aller Hosts tauschen paarweise in einer einzigen `MPI_Sendrecv`-artigen Funktion die Daten aus.
- Die `local_root`-Prozesse ordnet die Daten anhand der Anzahl der beteiligten Prozesse korrekt um.
- Jeder `local_root`-Prozess verteilt die Daten mit `MPI_Scatter` an die lokal beteiligten Prozesse.

Die Performance der Implementierung des Algorithmus wurde mit der alten Version verglichen. Dies wurde mit dem Intel MPI-Benchmark (IMB), ehemals Pallas MPI Benchmark, gemessen. Gemäß der Defaulteinstellung von `IMB` werden alle Prozesse vor der eigentlichen Schleife über 200 `PACX_Alltoall` mit zwei `PACX_Barrier` synchronisiert, außerdem werden zwei Warmup-Schleifen ausgeführt, d. h. die Kommunikationsroutine

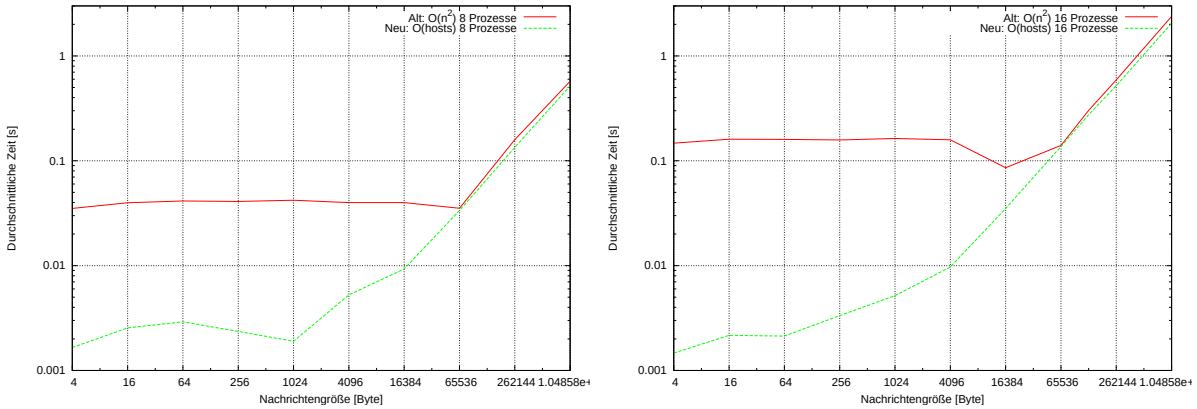

Abbildung 4.3: Vergleich zwischen altem und neuem Algorithmus für `PACX_Alltoall`

wird vor der eigentlichen Messunge zwei Mal aufgerufen. Tabelle 4.1 stellt die Daten der für den Test verwendeten Cluster dar. Um die Skalierung der beiden Algorithmen zu zeigen, wurden auf beiden Hosts vier und acht Knoten mit je einem Prozess gestartet, d. h. insgesamt 8 und 16 Anwendungsprozesse. Die Dämonen wurden auf den Frontend-Knoten des Clusters gestartet, um die Messung nicht zu beeinträchtigen. Die für die externe Kommunikation verwendete Verbindung wird über zwei Switche zwischen den beiden Frontend-Knoten ausgeführt, die Latenz zwischen den Frontend Knoten beträgt im Durchschnitt 0,5 ms (gemessen mit `mtr`) unabhängig von der Auslastung des Frontend-Knoten.

In Abb. 4.3 ist die durchschnittliche Zeit pro `PACX_Alltoall`-Aufruf für den neuen Algorithmus im Vergleich zum alten Algorithmus für insgesamt 8 und 16 Anwendungsprozesse über verschiedene Nachrichtengrößen dargestellt. Um den großen Unterschied zwischen dem alten und dem neuen Algorithmus zu zeigen wird eine logarithmische Zeitachse gewählt. Die Achsen beider Diagramme sind normiert, sodass die Zeit für 8 und 16 Anwendungsprozesse miteinander verglichen werden kann. Durch die Optimierung ist `PACX_Alltoall` mit dem neuen Algorithmus mit 16 Prozessen bis zu 100 Mal schneller bei 4 Byte-Nachrichten (von 147 ms auf 1,5 ms). Das ist darauf zurückzuführen, dass nicht auf jedem Prozess mit allen anderen Prozessen ausgetauscht und damit synchronisiert wird. Für Nachrichten der Länge 256 kB ist der Algorithmus immerhin noch 14%, bzw. für 1 MB Nachrichten 16% schneller. Hier werden die Dämonen bei großen Nachrichten zum Flaschenhals.

Für kleine Nachrichten erfolgt die externe Kommunikation nebenläufig zur internen. Die durchschnittliche Zeit je `PACX_Alltoall` ist für den neuen Algorithmus ca. konstant und unabhängig von der Anzahl der Prozesse. Für 16 Prozesse steigt die Zeit bei 1024 Byte Nachrichtenlänge auf Grund der limitierten Bandbreite zum PACX-Outdaemon an. Außerdem erfolgt der Transfer bei großen Nachrichten zum PACX-Outdaemon nicht mehr als „Eager Send“. Es wurde am Beispiel von `PACX_Alltoall` deutlich gemacht, dass algorithmische Verbesserungen auch in der Middleware die besten Performanceverbesserungen für höher skalierende Anwendungen bringt.

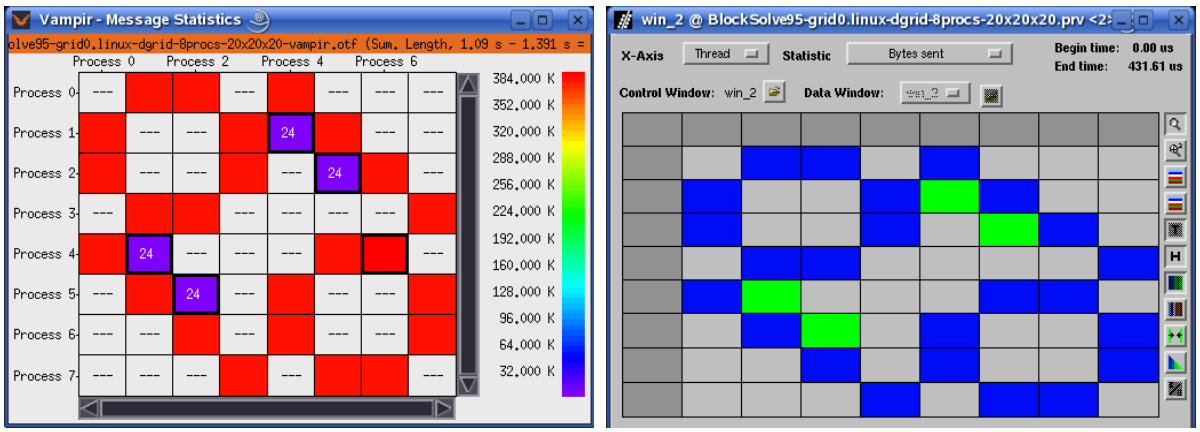

Abbildung 4.4: Kommunikationsmatrix in Vampir (links) und Paraver (rechts)

4.2 Performance-Analyse mittels einer Tracing-Bibliothek

Damit der Anwendungsprogrammierer einen Einblick in die serielle und parallele Performance seines Codes bekommt, stehen mehrere Tools für die Performance-Analyse wie Vampir [18], Jumpshot [148] oder Paraver [5] zur Verfügung.

Mit diesen Tools kann man das Verhalten des Programmes in verschiedenen Detailstufen untersuchen. Zu bestimmten Zeitpunkten werden Zeitstempel genommen und als so genannter Event abgespeichert. Die MPI-Funktionsaufrufe sowie manuell oder automatisiert vom Compiler eingefügte Messstellen usw. werden in den Trace-daten gesammelt. Diese Daten umfassen meist nicht nur den Zeitpunkt des Aufrufs, sondern auch die Adresse des Aufrufs im Anwendungsprogramm, manchmal sogar den gesamten Stacktrace. Zusammen mit den Hardware Performance Countern des Prozessors können zu jedem gemessenen Zeitstempel Informationen zur Auslastung und Verwendung des Prozessors abgeleitet werden. Beispielsweise lassen sich mit diesen Countern Aussagen zur Qualität des Codes treffen, bspw. aus dem Verhältnis der falsch vorhergesagten Sprungbefehle zur Anzahl aller Sprünge im betreffenden Codeabschnitt oder aus dem Verhältnis der arithmetischen Befehle zu den Lade- und Speicherbefehlen. Alle dem Autor bekannten Tracing Bibliotheken für MPI setzen auf dem so genannten Profiling MPI-Interface (PMPI) auf. Der MPI-Standard definiert, dass jede MPI-Funktion auch durch eine Funktion gleichen Namens, durch den Präfix `PMPI` aufrufbar ist, bspw. für die Funktion `PMPI_Init` statt `MPI_Init`. Zur Messung der Kommunikationszeit werden die MPI-Funktionen durch Routinen der Trace-Bibliothek ersetzt (engl. wrappen), in denen dann die eigentlichen `PMPI`-Funktionen aufgerufen werden[3].

Für die Darstellung der parallelen Performance verfügen die Analyseprogramme über eine Zeitachsenansicht. Hier repräsentieren verschiedene Farben unterschiedliche MPI-Aufrufe. Die Punkt-zu-Punkt-Kommunikation, sowie die kollektiven Aufrufe werden se-

[3]Die einzige Funktion, die sich damit nicht austauschen lässt, ist `MPI_Pcontrol`. Mit dieser lässt sich das Verhalten der PMPI-Schnittstelle steuern.

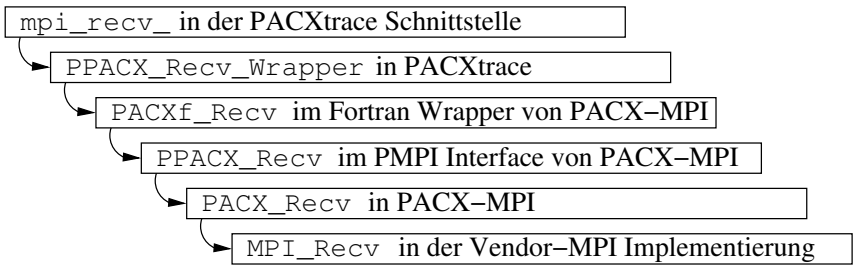

Abbildung 4.5: Link-Reihenfolge mit der Bibliothek `pacxtracef` im Falle von Fortran

parat meist durch Linien zwischen den beteiligten Prozessen dargestellt. Damit lassen sich die Kommunikationsmuster der Prozesse analysieren, Bandbreiten der Kommunikation grob errechnen und Lastbalancierungsprobleme visualisieren. Beispielsweise zeigt eine Situation mit einem unverhältnismäßig langem Empfangsaufruf (`MPI_Recv` oder `MPI_Wait`) ein so genanntes Late-Sender Ereignis. Ist die Anzahl der Instruktionen zwischen dem letzten gemessenen Zeitstempel der beiden Prozesse unterschiedlich, liegt evtl. ein Ungleichgewicht der den Prozessen zugewiesenen Arbeit vor (engl. Load-Imbalance).

Zur weiteren Analyse dieser feingranularen Daten bieten die oben genannten Programme mehrere Funktionen zur Aggregation der Informationen an. Beispielsweise kann man das Kommunikationsmuster leicht aus der Matrix der versandten Nachrichten ersehen, wie in Abbildung 4.4 für ein Programm mit der Bibliothek `BlockSolve95` [72] mit acht Prozessen dargestellt. Wie in beiden Varianten für Vampir links und Paraver rechts gezeigt, lässt sich sehen, dass bspw. Prozess 4 mit den Prozessen 0, 1, 5 und 6 kommuniziert[4]. Hierbei ist der minimale Wert mit 24 Bytes bei allen Prozessen gleich und der maximale Wert bis max. 398900 Bytes ähnlich. Durch den Farbverlauf ist die Verteilung visuell leicht erfassbar.

Im Rahmen des HPC-Europa Projektes wurde die am Barcelona Supercomputing Center entwickelte Tracingbibliothek `mpitrace` zur Generierung der Traces für Paraver auf die NEC-SX8 portiert. Die SX-Architektur erforderte Umstellungen im Build-System, Anpassungen von nicht unterstützten Systemaufrufen[5], Annahmen über die in der MPI-Implementierung verwendeten Datentypen, sowie die Unterstützung der Hardware Performance Counter der Prozessoren. Mit dieser Bibliothek und dem Paraver-Tool lassen sich die Vektorperformance, sowie allgemeine Qualität des Codes viel genauer analysieren, als dies mit grobgranularen Analyseprogrammen wie `ftrace` möglich ist.

Weiterhin wurde die Tracing-Bibliothek `mpitrace` um die Messung der Kommunikation von PACX-MPI erweitert, um Performance-Analyse von Anwendungen auf einem Metacomputer mit verteilten Rechnern zu erlauben. Diese Tracing-Bibliothek `pacxtrace` kann vom Anwendungsprogrammierer gegen die Anwendung gelinkt werden; hierfür muss auf die richtige Link-Reihenfolge geachtet werden[6].

[4] Vampir: Aktivieren im Untermenü „Global Displays → Message Statistics“.
Paraver: Aktivieren durch Menü „Global Controller → Analyzer 2D → X-Axis=Thread“.

[5] Beispielsweise gibt es keinen `mmap`-Aufruf für die Einblendung von Dateien in den Virtuellen Speicher.

[6] Mit `-L/PARAVER/lib -lpacxtrace -L/PACX/lib -lppacx -lpacx -L/ompi/lib -lmpi`.

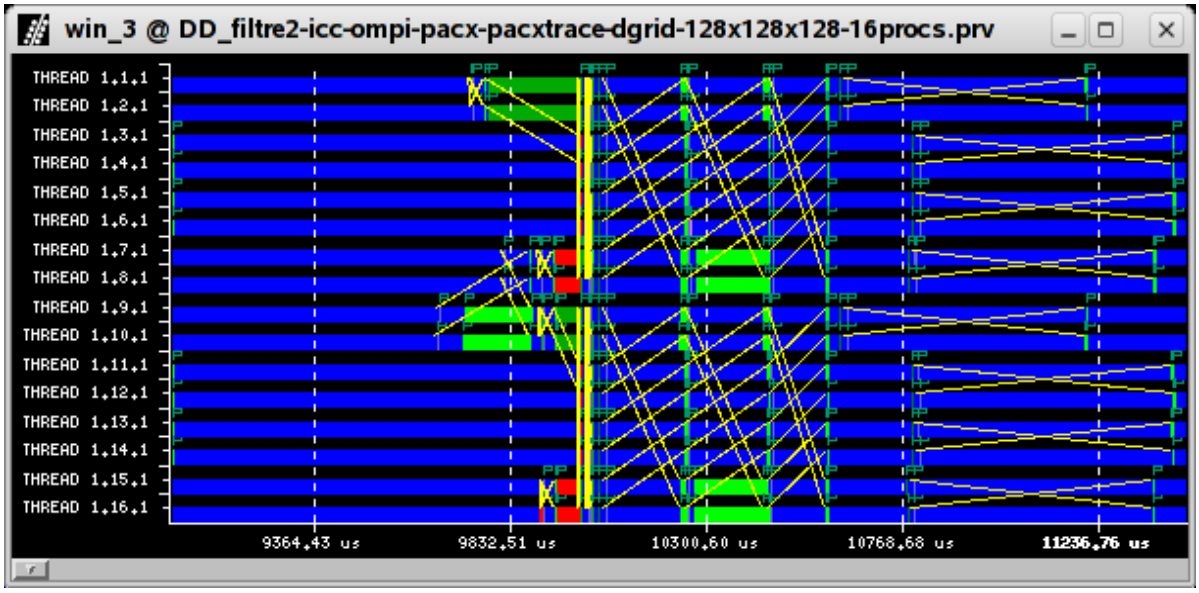

Abbildung 4.6: Ausschnitt eines Traces der Anwendung `DD_Filtre2`, gemessen mit PACX-MPI und `pacxtracef`

Um auch in Fortran programmierte Anwendungen zu unterstützen, musste das Fortran Interface von PACX-MPI um die so genannte schwache Namensauflösung (engl. Weak Symbols) erweitert werden. Dadurch konnten die auf PMPI basierenden Bibliotheksaufrufe korrekt an die Fortranfunktionen von PACX-MPI weitergegeben werden können. Die damit erweiterte Hierarchie von aufzurufenden Funktionen stellt Abb. 4.5 dar.

Mit der Bibliothek `pacxtrace` wird nur die Kommunikation zwischen den Anwendungsprozessen dargestellt, die schnelle Kommunikation mit den Dämonen wird ausgeblendet. Damit wird die langsamere Kommunikation ersichtlich, die durch die längere Transferzeit herrührt, wenn Prozesse Daten über die externe Verbindung austauschen. Weiterhin lassen sich die erreichten Bandbreiten in der von Paraver integrierten 2D-Analyse anzeigen, wie in Abb. 4.4 dargestellt.

In Abb. 4.6 ist ein Trace der mit PACX-MPI verteilt laufenden Anwendung `DD_filtre2` mit Hilfe des Performanceanalyseprogramms Paraver dargestellt. Die Zeitachsenachsicht zeigt 16 Prozesse[7], aufgeteilt auf zwei Hosts mit je acht Prozessen. Die Kommunikationen zwischen den Prozessen werden als gelbe Linien angezeigt. Die Hosts haben jeweils acht Prozesse, d. h. der Austausch zwischen den Hosts findet in diesem Abschnitt zwischen den Prozessen neun und sieben, bzw. zehn und acht statt. Kommunikation findet hier blockierend statt. Dies ist daran zu sehen, dass die Empfangsoperation, erkennbar am grünen Event, von Prozess neun und zehn nicht nebenläufig erfolgt, d. h. es erfolgt keine Unterbrechung durch die Anwendung (blaue Events). Wie man aus der Trace-Analyse sieht, ist dieser Austausch verzögert. Weiterhin zeigt sich eine Last-Imbalanz, die Prozesse eins und zwei sind früher mit der Arbeit fertig als die Prozesse neun und zehn.

[7] Achse links: angezeigt als `Thread Anwendungs-#.Prozess-#.Thread-#`.

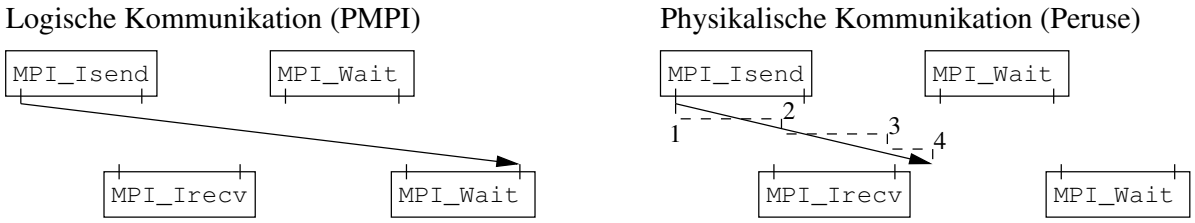

Abbildung 4.7: Kommunikationsanalyse mit der PMPI und Peruse Schnittstelle

4.3 Implementierung der MPI-Peruse Schnittstelle

4.3.1 Nachteile der PMPI-basierten Messungen

In den Tracing-Bibliotheken werden Zeitstempelinformationen vor, bzw. nach dem Aufruf der `PMPI`-Funktion ausgelesen. Anhand der Zeitmessung am Beginn und am Ende bspw. eines `MPI_Send`-Aufrufs, kann man erfassen wie lange ca. der Versand gedauert hat. Aus verschiedenen Gründen ist dies aber nur eine ungefähre Zeit, da die interne Implementierung in der MPI-Bibliothek nicht definiert ist[8]. Weiterhin wird in Tools zur Performance-Analyse die Zeit für nicht-blockierende Kommunikation als logische Zeit für die gesamte Übertragung angenommen, wie in Abb. 4.7 (links) dargestellt. Der Kommunikationsbeginn wird mit der Messung vor dem eigentlichen Aufruf von `MPI_Send` gleichgesetzt, während das Ende der Übertragung nach der korrekten Rückkehr aus dem `MPI_Wait` auf der Empfängerseite angenommen wird.

Tatsächlich aber kann die eigentliche Kommunikation in der MPI-Implementierung beliebig aufgeteilt werden, bspw. könnte eine Implementierung den gesamten Transfer im ersten `MPI_Isend` erledigen. Oder aber, wenn die Implementierung Threads für die eigentliche Übertragung verwendet, könnte die Kommunikation echt nebenläufig zur Ausführung der Anwendung passieren. Dies zeigt, dass bei der Übertragung auch nur einer Nachricht die Performance stark variieren kann. Dies ist in Abb. 4.7 (rechts) dadurch erkennbar, dass der physikalische Send in diskreten Schritten (mehreren Fragmenten) zeitlich verzögert ankommt, aber schneller fertig ist, als durch das `PMPI`-Interface sichtbar wird.

Um einen genaueren Einblick in die Implementierung, die Kommunikationseigenschaften über die physikalischen Netzwerke und evtl. vorhandene Performanceprobleme, wie in [16] beschrieben, zu bekommen, hat ein internationales Konsortium die Peruse-Spezifikation [73] herausgegeben, an welcher der Autor beteiligt war. In dieser Spezifikation wird eine MPI-implementierungsunabhängige Schnittstelle für Anwendungen und Bibliotheken definiert, mit der bestimmte Zustände der Kommunikation aus der MPI-Implementierung nach außen weitergegeben werden. Diese werden in der Peruse Version 2.0 als so genannte Events definiert, für die ein vom Anwender, bzw. der Tracing-Bibliothek definierter Funktionsaufruf (engl. Callback) aufgerufen wird, wie unten näher erklärt.

[8] Im Falle von `MPI_Send` könnte die Implementierung die gesamte Nachricht in den MPI-internen Puffer kopieren.

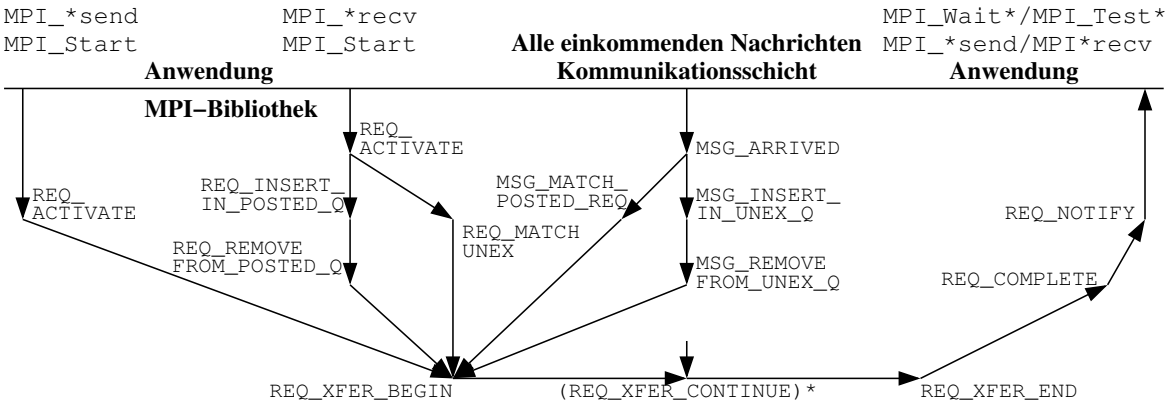

Abbildung 4.8: Abfolge von Peruse-Events für P2P-Kommunikation

4.3.2 Peruse-Implementierung in PACX-MPI und Open MPI

Die erste Implementierung in PACX-MPI auf Basis der Peruse-Version 1.10 brachte allerdings keine Vorteile für die Erkennung von Performanceproblemen, solange die eigentlich interessante Kommunikation innerhalb des Vendor-MPI nicht durch Peruse instrumentiert ist. Im Rahmen des Open MPI Projektes wurde Peruse zusammen mit Projektpartnern der University of Tennessee, Knoxville (UTK) auf Basis von der Spezifikation Version 2.0 in Open MPI integriert. Weiterhin wurde die Bibliothek `mpitrace` um Tracing-Funktionalitäten für Peruse erweitert und in [76] dokumentiert.

Peruse definiert 17 Events für Point-to-Point Kommunikation, von denen 13 in Abb. 4.8 darstellt sind. Das Diagramm zeigt die zeitliche Abfolge der Events, wenn eine Applikation mit den P2P-Funktionen kommuniziert. Dies entspricht dem Zustandsübergangsdiagramm von Nachrichten, bzw. Fragmenten innerhalb der MPI-Bibliothek: Wird ein `MPI_Send`-Aufruf abgesetzt oder ein so genannter Persistent Request mit `MPI_Start` gestartet, wird der Event `PERUSE_COMM_REQ_ACTIVATE` abgesetzt, d. h. der hierfür registrierte Callback der Tracing-Bibliothek wird aufgerufen. Sobald die eigentliche Übertragung beginnt, wird der Event `PERUSE_COMM_REQ_XFER_BEGIN` abgesetzt, am Ende der physikalischen Übertragung der Nutzdaten `PERUSE_COMM_REQ_XFER_END`. Damit lässt sich die eigentliche physikalische Transferzeit, wie in Abb. 4.7 (rechts) dargestellt, ableiten. Weiterhin kann man damit errechnen, bis zu welcher Latenz des Netzwerkes diese Konstellation aus nicht-blockierender Kommunikation und Berechnung skalieren würde.

4.3.3 Minderung der Performance durch Peruse

Nachdem die Peruse-Funktionalität für P2P-Kommunikation immerhin 17 Events definiert, die teilweise im Ausführungspfad jeder Nachricht und schlimmer im Pfad für Fragmente liegt, muss evaluiert werden, wie groß die Degradation der Übertragungsgeschwindigkeit ist[9]. Um einen Vergleich des Einbruchs der Leistung mit der Peruse-

[9] Der Benutzer/Administrator muss die Peruse-Funktionalität explizit beim Konfigurieren von Open MPI mit `--enable-peruse` hinzufügen.

Cluster	**cacau**	**strider**
Prozessor	Dual Intel Nocona, 3.2 GHz	Dual AMD Opteron 246, 2GHz
Netzwerk	Infiniband SDR, 10 Gbs	Myrinet 2000
Schnittstelle	`mvapi-4.1.0`	`gm-2.0.8`
Compiler	Intel compiler 9.0	PGI compiler 6.1.3
Open MPI	no debug, static build	no debug, dynamic build
Natives MPI	Voltaire MPIch-1.2.6	MPIch-1.2.6

Tabelle 4.2: Konfiguration der Cluster für den Test der Peruse-Funktionalität

	cacau			**strider**		
	Native	mvapi	sm	Native	gm	sm
1. Ohne Peruse	4,13	4,69	1,02	7,16	7,16	1,33
2. Peruse, Keine Callbacks		4,67	1,06		7,26	1,71
3. Peruse, No-Op Callbacks		4,77	1,19		7,49	1,84

Tabelle 4.3: Latenz (in μs) für 0-Byte Nachrichten ohne/mit Peruse

Funktionalität auf verschiedenen Interconnects zu bekommen, wurden Messungen mit dem Intel MPI-Benchmark (IMB) auf den Clustern `strider` und `cacau` durchgeführt. Wie in Tab. 4.2 ersichtlich, bietet `cacau` einen Infiniband-NIC (mit `mvapi`-Schnittstelle), während `strider` mit Myrinet (hier `gm`-Schnittstelle) ausgestattet ist.

Für die Messungen wurden die Standardwerte von IMB auf maximal 10000 Iterationen und 10 Aufwärmphasen hoch gesetzt. Die 0-Byte Latenz wird für beide Cluster in Tab. 4.3 vorgestellt. Einerseits wird Open MPI mit dem nativen MPI des Herstellers verglichen, andererseits werden verschiedene Konstellationen berücksichtigt:

1. Ein Lauf ohne Peruse-Funktionalität,
2. mit Peruse wenngleich auch keine Callbacks für Events definiert sind und
3. mit Peruse und einem Callback pro Event, welcher allerdings sofort zurückkehrt.

Open MPI kann ähnliche Performanceresultate wie das Hersteller-MPI liefern. Von Interesse ist aber die zusätzliche Funktionalität durch Peruse. Im Vergleich zur Version ohne Peruse-Unterstützung, zeigt Open MPI nur Infiniband (IB) (`mvapi`) und Myrinet (`gm`) nur marginale Unterschiede im Bereich von 1,7%, bzw. 4,6% in der Latenz. Daraufhin wurde ein weiterer Test mit dem Shared Memory Schnittstelle (Spalte `sm` in Tab. 4.3) auf beiden Clustern gemacht, da dieses eine höhere Sensitivität auf die Anzahl der ausgeführten Instruktionen hat. Hierfür werden die zwei Prozesse des PingPong-Tests auf dem gleichen Knoten gestartet. Hier erfährt man eine Verschlechterung der Latenz um 16%, bzw. bis zu 38%, wenn Callbacks für alle Events registriert sind.

Für den Test der Bandbreite wird das Verhältnis der durchschnittlichen Werte in Relation zur Bandbreite ohne Peruse-Funktionalität gesetzt. Wie in Abb. 4.9 gezeigt, beträgt der maximale Einbruch mit dem Infiniband nur bis zu 4% auf `cacau`, bzw. mit Myrinet max. 7% auf `strider`. Hier muss hinzugefügt werden, dass für Performanceanalyse

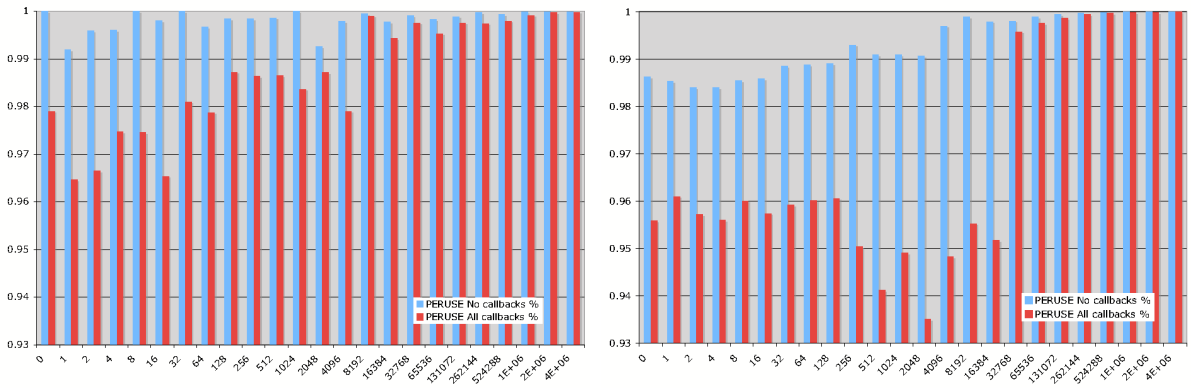

Abbildung 4.9: Verhältnis der Bandbreite mit Peruse zur Bandbreite ohne Peruse auf `cacau`(links) und `strider`(rechts)

nicht notwendig ist, alle 17 Peruse-Events zu speichern. Dies würde zu unnötig großen Tracefiles führen, die nicht viel aussagekräftige Informationen liefern. Nur durch die Abfragen innerhalb der Peruse-Funktionalität (die linken Angaben) in beiden Diagrammen, ergeben minimale Einbußen bis max. 2% auf beiden Systemen.

4.3.4 Analysemöglichkeiten mit der Peruse-Funktionalität

Die Behandlung unerwarteter Nachrichten (so genannte „Late Receiver“-Situationen) geschieht durch die Events `PERUSE_COMM_MSG_INSERT_IN_UNEX_Q` und `PERUSE_COMM_MSG_REMOVE_FROM_UNEX_Q`. Aus diesen kann man ableiten, wie viele Nachrichten und wie lange eine spezifische Nachricht in der Warteschlange der unerwarteten Nachrichten verweilen musste. Damit lassen sich nicht nur Last-Imbalance und Many-to-One Probleme der Applikation sondern auch eine ineffiziente Implementierung der Warteschlange der MPI-Bibliothek erkennen.

Durch den (nicht in der Peruse-Spezifikation definierten) Event `PERUSE_COMM_REQ_XFER_CONTINUE` lassen sich Fragmentraten ableiten, die durch den Event `PERUSE_COMM_MSG_ARRIVED` nicht von regulären Verwaltungsnachrichten der MPI-Implementierung zu unterscheiden wären. In Abb. 4.10 ist die Kommunikation der Molecular Dynamik Anwendung `IMD` [116] mit Paraver dargestellt. Dieser Programmlauf wurde hier auf dem Cacau-Cluster mit 32 Prozessen ausgeführt. Im Testfall `bench_cu3au_1048k.param` wurde mit 1048576 Atomen im Einheitskubus mit einer Aufteilung in Z-Richtung, d. h. $1 \times 1 \times 32$ Prozessen gerechnet. Abbildung 4.10 (oben) zeigt den logischen Datenaustausch mit dem linken und rechten Nachbarn am Ende einer Iteration, jeweils mit nichtblockierenden Funktionen, d. h. zwei Sende- und zwei Empfangsaufrufe je Nachbar. Diese Information lässt sich mit der in `mpitrace` vorhandenen Funktionalität mittels PMPI darstellen. Die Kommunikation überlappt in der logischen Ansicht mit einem anderen Kommunikationsmuster vor dem Austausch. Durch die so genannte semantische Interpretation erlaubt `Paraver` zusammen mit den Peruse-Events die grafische Darstellung der Anzahl der gleichzeitigen physikalischen Übertragungen. Nachdem vier Kommuni-

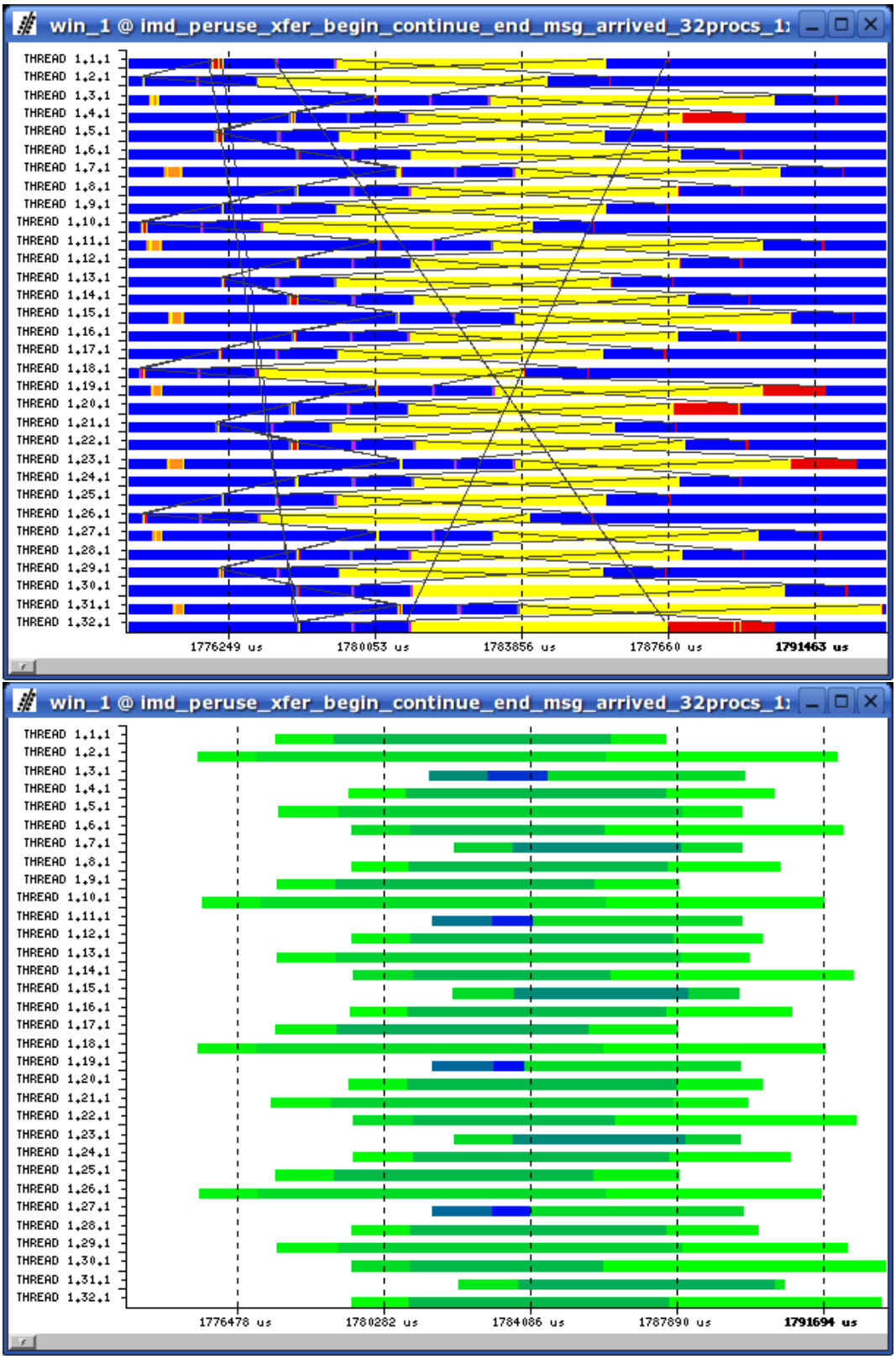

Abbildung 4.10: Logische (oben) und physikalische Kommunikation (unten) von IMD

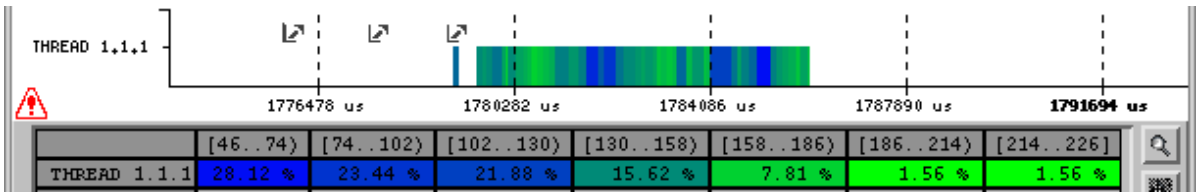

Abbildung 4.11: Analyse der Fragmentraten für Prozess eins, siehe Abb. 4.10

kationen je Prozess parallel ablaufen, sind in Abb. 4.10 (unten) vier Abstufungen von hell-grün bis dunkel-blau zu sehen.

Filtert man weiterhin nach dem Event `PERUSE_COMM_REQ_XFER_CONTINUE` und analysiert die Zeit zwischen den gleich großen Fragmenten, sieht man, dass die Differenz stark variiert (hier für MPI-Prozess eins in Abb. 4.11 oben, mit angezeigt ist der logische Start der Senderoutinen mittels des kleinen Pfeils). Korrespondierend hierzu liefert die 2D-Analyse in Abb. 4.11 unten die Verteilungen: immerhin 28,12% der Fragmente kommen in einem Zeitraum von 46..74 μs, weitere 23,44% kommen zwischen 74..102 μs, 21,88% innerhalb 102..130 μs an. Die restlichen 26,55% kommen allerdings zwischen 130..226 μs an. Das ist immerhin dreimal so lange wie das kürzeste Fragment gleicher Größe gebraucht hat und ist damit nicht mehr durch statistische Streuung zu erklären.

Mit diesen Informationen kann der Anwendungsentwickler seine Kommunikationsmuster optimieren, bspw. in mehrstufigen Netzwerken Netzauslastung erkennen oder analysieren ob die MPI-Implementierung für die Nachrichtengröße nebenläufige Kommunikation zulässt. Dies lässt sich mit einem Testprogramm relativ leicht bewerkstelligen, verschiedene mehrere Nachrichten mit immer größerer Pufferlängen verschickt werden. Beim Empfänger wird ein nicht-blockierender `MPI_Irecv` abgeschickt, den Prozess schlafen gelegt und auf `PERUSE_COMM_MSG_ARRIVED` gewartet. Entsprechend kann es für manche Anwendungen sinnvoll sein, das „Eager Send Limit“ durch einen Parameter der MPI-Implementierung zu erhöhen.

Weiterhin kann mit Peruse-Informationen ausgerechnet werden, bis zu welcher Prozessorzahl ein Algorithmus mit nebenläufiger Kommunikation skaliert. Ist die Rechengeschwindigkeit eines Algorithmus in Abhängigkeit der Datengröße bekannt, kann die Zeit errechnet werden, die zum Verstecken der physikalischen Kommunikation benötigt wird. Daraus ergibt sich, wieviel Arbeit pro Prozessor durchgeführt werden muss, bis die Prozessoren in den `MPI_Wait` eintreten (siehe Strong Scaling).

Noch mehr helfen die Daten von Peruse allerdings dem Entwickler der MPI-Bibliothek. Wie in Abschnitt 4.1 erläutert, gibt es für kollektive Kommunikationsroutinen eine große Anzahl von Implementierungsalternativen in Bezug auf Algorithmus und Parameter. Mittels Peruse lässt sich die Auslastung des Netzwerks durch kollektive Kommunikationsroutinen überprüfen und besser balanciert werden. Dies ist vor allem für mehrstufige Netzwerktopologien wichtig (beschrieben auf Seite 104 im Anhang). Außerdem können Vergleiche der Effizienz der Behandlung von unerwarteten Nachrichten verschiedener MPI-Implementierungen gezogen werden oder Fragmentraten für verschiedene Netzwerktypen ausgewertet werden, wie oben beschrieben.

4.4 Implementierung einer MPI Testsuite

4.4.1 Nachteile bestehender Testsuites

Die Fehlersuche in einer parallelen Anwendung ist sehr aufwändig. Die zusätzliche Komplexität von PACX-MPI erschwert die Fehlersuche nochmals. Vor allem die langsamere Kommunikation über die externen Verbindungen führt zu unerwarteten Effekten, besonders in Verbindung mit nicht-deterministischen MPI-Funktionalitäten wie `MPI_ANY_SOURCE`, `MPI_ANY_TAG` oder den Funktionen `MPI_Waitany`, `MPI_Waitsome` oder `MPI_Cancel`. Auch durch die Koppelung unterschiedlicher MPI-Implementierungen mit unterschiedlichen Kommunikationscharakteristiken und Interpretationen, werden Fehler in Anwendungen offenbar[10]. Weiterhin kann beim heterogenen MetaComputing die Datenkonvertierung in externe Datenformate (8 Byte zu 4 Byte Ganzzahlen, Abschneiden von Fließkommazahlen) zu unerwarteten Effekten führen. Wie in jeder umfangreichen Bibliothek, existieren auch in PACX-MPI Fehler.

Es gibt einige existierende Testsuites der MPI-Implementierungen, namentlich von IBM, MPIch oder Intel. Diese haben aber Nachteile. Zum einen ist nicht eindeutig, welche Funktionalitäten getestet werden. Zum anderen führen viele Suites den Großteil der Programme mit fester Anzahl von Prozessen aus. Weiterhin ist es bei diesen auch nicht möglich, beliebige Kombinationen der Testfälle anzugeben oder Parameter zu variieren. Dies ist notwendig, da manche Fehler erst mit einer bestimmten Nachrichtengröße, nach einer Vielzahl von Tests (bspw. erst ein Send mit Persistent Requests, dann Buffered Sendemodus) oder nach einer Kombination von Parametern (erst mit `MPI_COMM_SELF`, dann `MPI_COMM_WORLD`) auftreten. Der größte Nachteil für den Test mit PACX-MPI aber ist die Vielzahl von Einzeltests, die mit PACX-MPI jeweils separat auf dem Metacomputer gestartet werden müssen.

Um diesen Nachteil zu umgehen und vor allem einen bestimmten Funktionsaufruf mit unterschiedlichen Parametern wie Nachrichtenlänge, Kommunikatoren, Datentypen und Anzahl der beteiligten Prozesse zu testen, wurde vom Autor eine neue Testsuite entwickelt [83] und in [123] um MPI-2 Tests der Funktionalität von Parallel File-IO erweitert.

4.4.2 Aufbau und Konzept

Für die Testsuite galten die folgenden Designkriterien:

- Ausführung aller Tests innerhalb eines parallelen Programmes,
- einfache Erweiterbarkeit durch beliebige Tests,
- mehrmalige Ausführung mit einer beliebigen Anzahl von Prozessen,

[10] Beispielsweise bei einer Applikation die auf einem Teil des Metacomputers mit C-Aufrufen zu einem Empfänger auf einem anderen Teil des Metacomputers mit Fortran-Aufrufen aber durch den Fortran MPI-Type `MPI_INTEGER` kommuniziert hat. Dies führte bei Ausführung mit PACX-MPI zu Fehlern.

- die Ausführung jedes Tests mit einer möglichst großen Kombination von Datentypen, Kommunikatortypen, Nachrichtenlängen,
- die mehrfache Ausführung jedes Tests (für Benchmarkingzwecke),
- einfache Auswahl der Tests und Steuerung der Testläufe und -kombinationen, bspw. für strikte oder laxere Einhaltung des MPI-Standards.

Um unabhängig von der tatsächlichen MPI-Bibliothek zu bleiben, werden keine Annahmen über die Implementierung gemacht, bei der Konfiguration werden die Funktionalitäten besonders im Hinblick auf Unterstützung von MPI-2 [95] überprüft. Mittels eines `configure`-Skripts werden Betriebssystemabhängigkeiten, sowie Details der MPI-Implementierung, wie Größe von MPI-Datentypen, etc. erkannt. Die Testsuite wird dann mit allen Tests in ein einzelnes Programm compiliert. Durch Parameter kann die Ausführung gesteuert werden, bspw. können Ausgabe, Gruppen- sowie Einzeltests oder Datentypen, sowie ein restriktiver Modus selektiert werden. Bei letzterem Modus werden auf strikte Vorgaben des Standards überprüft. Ohne Angabe werden alle Tests in allen Kombinationen von Datentypen ausgeführt und erst am Ende die fehlgeschlagenen Testkombinationen ausgegeben.
Für jeden zu durchlaufenden Testfall müssen drei Funktionen programmiert werden:

```
int tst_CLASS_TEST_init (struct tst_env * env);
int tst_CLASS_TEST_run (struct tst_env * env);
int tst_CLASS_TEST_cleanup (struct tst_env * env);
```

`CLASS` steht für eine zu testende Klasse. Klassen fassen einen zu testenden Funktionsumfang zusammen, die folgenden Klassennamen sind bereits vordefiniert:

- `env` steht für Environment,
- `p2p` für Point-to-Point,
- `coll` für kollektive Tests,
- `dynamic` für den dynamischen Prozessstart von MPI-2,
- `io` für Parallelen IO,
- `one-sided` für einseitige Kommunikation und
- `threaded` für Tests mit mehreren Threads.

Durch `TEST` wird ein eindeutiger Name innerhalb der Testsuite festgelegt. Der übergebene Zeiger auf die Struktur `tst_env` beinhaltet alle notwendigen Informationen über die verwendeten MPI-Objekte, Prozesszahl, Nachrichtenlänge, etc. Für jeden Test muss eine Struktur ausgefüllt werden, die den Test auch für den Benutzer beschreibt. Hier werden die unterstützten MPI-Objekte angezeigt, eine Beschreibung des Tests gespeichert, sowie die Funktionen `init`, `run` und `cleanup` registriert.

MPI_COMM_WORLD	MPI_COMM_NULL	MPI_COMM_SELF
Duplicated MPI_COMM_WORLD	Reversed MPI_COMM_WORLD	Halved MPI_COMM_WORLD
Odd/Even split MPI_COMM_WORLD	Zero-and-Rest Intercomm	Two dimensional Cartesian
Three dimensional Cartesian	Full-connected Topology	Halved Intercomm
Intracomm merged of the Halved Intercomm		

Tabelle 4.4: Liste der implementierten Kommunikatoren

Im folgenden Programmcode wird der „Many-to-one“ Test gezeigt, welcher intern MPI_Iprobe und MPI_ANY_SOURCE verwendet. Der Test gibt an mit Intra- und Interkommunikatoren arbeiten zu können, weiterhin unterstützt er alle C-Datentypen und kann mit dem strikten und dem laxen Testmodus ausgeführt werden. Nachdem der Test MPI_ANY_SOURCE verwendet, sollte er gegen andere Tests durch eine MPI_Barrier-Synchronisation geschützt werden. Dies geschieht durch Angabe von TST_SYNC, damit nicht bereits versandte Nachrichten des nachfolgenden Tests empfangen werden.

```
{TST_CLASS_P2P, "Many-to-one MPI_Iprobe",
 TST_MPI_INTRA_COMM | TST_MPI_INTER_COMM,
 TST_MPI_ALL_C_TYPES, TST_MODE_RELAXED,
 /* Recv. w/ MPI_ANY_SOURCE & MPI_ANY_TAG */
 TST_SYNC,
 &tst_p2p_many_to_one_iprobe_init,
 &tst_p2p_many_to_one_iprobe_run,
 &tst_p2p_many_to_one_iprobe_cleanup}
```

Das Hauptprogramm steuert die Ausführung in einer Schleife über alle selektierten Tests, alle selektierten Kommunikatoren, alle selektierten Datentypen für alle gewünschten Nachrichtenlängen. In dieser Schleife wird die init-Funktion einmal je Testfall aufgerufen, damit Initialisierungen für die bestimmte Konfiguration aus tst_env ausgeführt werden. Sodann wird die run-Funktion mindestens einmal aufgerufen, kann aber auch, bspw. für Benchmark-Zwecke mehrfach aufgerufen werden. In der cleanup-Funktion werden dann alle alloziierten Speicherbereiche wieder freigegeben. Jeder Test muss mit beliebig vielen Prozesse ausführbar sein und möglichst viele Varianten von Daten- und Kommunikatortypen unterstützen.

Tabelle 4.4 führt die derzeit unterstützten Kommunikatoren auf. Abgeleitete Kommunikatoren (engl. derived communicator) werden nur bei der entsprechenden Anzahl von ausgeführten Prozessen erzeugt.

In Tabelle 4.5 werden die derzeit verwendeten Basis- und abgeleiteten Datentypen angegeben. Auch hier werden manche der Datentypen nur verwendet bzw. erzeugt, wenn die Implementierung dies unterstützt. Bei einer MPI-2 Implementierung sind dies z. B. der Typ MPI_UNSIGNED_CHAR und mit der Funktion MPI_Dup duplizierte Typen.

Tabelle 4.6 führt die derzeit unterstützten Tests auf, hierbei sind für Point-to-Point Funktionen verschiedene Kommunikationsmuster, wie ein Ring-Austausch, eine Alltoall-Kommunikation oder auch Many-to-One Situationen programmiert. Diese Kommunikationsmuster sind dann in den verschiedenen Kombinationen der Sende- und Empfangsaufrufe aufgeschlüsselt. Für kollektive Aufrufe variieren Parameter wie die Operation

MPI_CHAR	MPI_UNSIGNED_CHAR	MPI_SIGNED_CHAR	MPI_BYTE
MPI_SHORT	MPI_UNSIGNED_SHORT	MPI_INT	MPI_UNSIGNED
MPI_LONG	MPI_UNSIGNED_LONG	MPI_FLOAT	MPI_DOUBLE
MPI_LONG_DOUBLE	MPI_LONG_LONG	MPI_FLOAT_INT	MPI_DOUBLE_INT
MPI_LONG_INT	MPI_SHORT_INT	MPI_2INT	MPI_LONG_DOUBLE_INT
MPI_CONTIGUOUS_INT	MPI_VECTOR_INT	MPI_HVECTOR_INT	MPI_INDEXED_INT
MPI_HINDEXED_INT	MPI_STRUCT_INT	MPI_TYPE_MIX	MPI_TYPE_MIX_ARRAY
MPI_TYPE_MIX_LB_UB	Dup MPI_CHAR	Dup MPI_TYPE_MIX_LB_UB	

Tabelle 4.5: Liste der implementierten Datentypen

in MPI_Op, auch durch eine selbst mit MPI_Op_create definierte Operation (Summe der Quadrate) oder bei MPI-2 Implementierungen durch Verwendung des Parameters MPI_IN_PLACE.

Nachdem die internen Funktionen der Testsuite, sowie einzelne Tests thread-sicher ausgelegt sind, kann die Testsuite durch Parameter so gesteuert werden, dass mehrere Threads bestimmte Tests nebenläufig ausführen. Deswegen reduziert sich die Anzahl der explizit als Thread-Tests angegebenen Fälle in Tab. 4.6 auf derzeit fünf. Durch die Kombination aller Optionen werden damit 7616 verschiedene Tests durchgeführt.

4.4.3 Ergebnisse mit der Testsuite

Mit Hilfe der Testsuite wurden auch in schon bewährten MPI-Implementierungen Fehler gefunden. In dieser Arbeit sollen diese nicht weiter aufgeführt werden. Wir analysieren nur Fehler der Implementierungen PACX-MPI, Open MPI und den Bibliotheken mpitrace/pacxtrace. Für detailliertere Informationen sei auf [83] und für Parallel IO sei auf die Diplomarbeit von Sheng [123] verwiesen. In letztgenannter Arbeit wurde auch der Abdeckungsgrad der Open MPI-Implementierung untersucht. Es wird gezeigt, dass mit bis zu 90% des Codes von Open MPI alle relevanten Codezeilen abgedeckt werden. Der nicht abgedeckte Code besteht aus Zeilen für die Fehlerbehandlung, die in dieser Testsuite aus den Designentscheidungen nicht getestet werden können.

In PACX-MPI wurden zusammen mit dem Memory Debugger valgrind [121] Fehler bei der Konvertierung des Datentyps MPI_TYPE_MIX_LB_UB festgestellt. Hier sind ausserdem einige Überläufe bei den Versenderoutinen der Dämonen aufgetreten. Weiterhin wurde ein Initialisierungsfehler des so genannten Command Paketes bei kollektiven Routinen entdeckt.

In der Tracing Bibliothek mpitrace fanden wir Fehler in Funktionswrappern. Diese Wrapper waren nicht auf die Parameter MPI_PROC_NULL oder MPI_TYPE_NULL vorbereitet. Zudem traten Probleme in der Implementierung der Wrapper für MPI_Waitsome und MPI_Testsome auf, da hier nicht korrekt auf die Beendigung von Requests gewartet wurde. Diese Fehler scheinen nicht gravierend, da nur wenige Benutzer die Funktionalität nutzen, dennoch sollte eine Bibliothek diese Fälle korrekt handhaben.

Der aus vordefinierten Datentypen abgeleitete Datentyp MPI_TYPE_MIX wird durch die

Environment Tests:

Status	`MPI_REQUEST_NULL`	

Point-to-Point Tests:

Ring	Ring Send Bottom	Ring Send Pack
Ring Isend	Ring Ibsend	Ring Irsend
Ring Issend	Ring Bsend	Ring Rsend
Ring Ssend	Ring Sendrecv	Ring same value
Direct Partner Intercomm	Many-to-one	Many-to-one `MPI_Probe`
Many-to-one `MPI_Iprobe`	Many-to-one Isend&Cancel	Alltoall
Alltoall-Persistent	Alltoall-xIsend	Alltoall-Irsend
Alltoall-Issend	Alltoall w/ `MPI_ANY_SOURCE`	Ring Send w/ Cartesian
Alltoall w/ Topology		

Collective Tests:

Bcast	Gather	Allgather
Allgather w/ `MPI_IN_PLACE`	Scan sum	Scatter
Scatterv	Scatterv stride	Reduce Min
Reduce Max	Allreduce Min	Allreduce Max
Allreduce Quadsum	Allreduce Sum	Alltoall
Reduce Min/Max w/ `MPI_IN_PLACE`		
Allreduce Min/Max w/ `MPI_IN_PLACE`		

One-sided Tests:

Ring w/ Get&Post	Ring w/ Get&Lock	Ring w/ Get&Fence
Ring w/ Put&Post	Ring w/ Put&Lock	Ring w/ Put&Fence
Accumulate w/ Post,Min	Accumulate w/ Lock,Max	Accumulate w/ Fence,Sum

Threaded Tests:

Ring	Ring Isend	Ring Bsend
Ring Persistent	Bcast w/ duplicated comms	

Parallel IO Tests:

Read	Write	Iread
Iwrite	Read at	Write at
Iread at	Iwrite at	Read at all
Write at all	Read at all begin	Write at all begin
Read all	Write all	Read all begin
Write all begin	Read shared	Write shared
Iread shared	Iwrite shared	Read ordered
Write ordered	Read ordered begin	Write ordered begin
Read/Write w/ `MPI_Type_create_darray`		
Read/Write w/ `MPI_Type_create_subarray`		
IO w/ hole	IO w/ Arrange	IO w/ Info
Read Convert	IO Atomar	IO Sync
IO Commself	Asyncio Atomar	Append Modus
Sequential Modus	Set Size	Preallocate

Tabelle 4.6: Liste der implementierten Tests

Datatype Engine von Open MPI[11] nicht korrekt behandelt. Dieser enthält eine Struktur mit elf vordefinierten Datentypen, u. a. Festkomma- und Fließkommazahlen sowie den vordefinierten Typen für `MPI_Reduce` Operationen, wie `MPI_FLOAT_INT`. Bei vollständiger Ausgabe (Parameter `-r FULL`) erhält man einen Fehler an den Bytes 58 und 59, da an diesen Stellen andere Werte als erwartet stehen (Byte `0xa5` wird bei der Initialisierung als Markierung verwendet).

```
Error at i:6 expected [68 Bytes]
[ 0-15]: 0x07 0xa5 0x07   0x00 0x07 0x00 0x00 0x00 0x07 0x00 0x00 0x00 0x00 0x00 0xe0 0x40
[16-31]: 0x00 0x00 0x00   0x00 0x00 0x00 0x1c 0x40 0x00 0x00 0xe0 0x40 0x07 0x00 0x00 0x00
[32-47]: 0x00 0x00 0x00   0x00 0x00 0x00 0x1c 0x40 0x07 0x00 0x00 0x00 0x07 0x00 0x00 0x00
[48-63]: 0x07 0x00 0x00   0x00 0x07 0x00 0xa5 0xa5 0x07 0x00 0x00 0x00 0x07 0x00 0x00 0x00
[64-67]: 0x07 0x00 0x00   0x00
Error at i:6 but received [68 Bytes]
[ 0-15]: 0x07 0xa5 0x07   0x00 0x07 0x00 0x00 0x00 0x07 0x00 0x00 0x00 0x00 0x00 0xe0 0x40
[16-31]: 0x00 0x00 0x00   0x00 0x00 0x00 0x1c 0x40 0x00 0x00 0xe0 0x40 0x07 0x00 0x00 0x00
[32-47]: 0x00 0x00 0x00   0x00 0x00 0x00 0x1c 0x40 0x07 0x00 0x00 0x00 0x07 0x00 0x00 0x00
[48-63]: 0x07 0x00 0x00   0x00 0x07 0x00 0xa5 0xa5 0x07 0x00 0xa5 0xa5 0x07 0x00 0x00 0x00
[64-67]: 0x07 0x00 0x00   0x00
```

[11] Entwicklerversion svn-r16159, Stand September 2007

5 Anwendungen und Ergebnisse

Der wichtigste Faktor bei der Ausführung von Anwendungen auf verteilten Rechnern ist die Kommunikation. Die langsame Verbindung vergrößert den inherenten Kommunikationsaufwand eines Algorithmus. Deswegen muss der Algorithmus dahingehend angepasst werden, dass die Kommunikation über diese Verbindung optimal verteilt wird. Bei der Optimierung paralleler Algorithmen müssen also nicht nur die lokalen Voraussetzungen (Prozessor- und Speichercharakteristiken), sondern vielmehr die Kommunikationscharakteristiken berücksichtigt und dementsprechend angepasst werden. Dies gilt vor allem für hierarchische Netzwerkmodelle, wie sie beim MetaComputing auftreten, aber sehr wohl auch für Clustersysteme mit mehrstufigen Switches. Man kann die Kommunikation in eine Topologie einordnen:

- Kommunikation zwischen Cores eines Prozessors
- Kommunikation zwischen Prozessoren innerhalb einer Zelle
- Kommunikation zwischen Zellen eines SMP-Knotens
- Kommunikation zwischen SMP-Knoten in einem Cluster
- Kommunikation zwischen Clustern im Metacomputer.

In diesem Abschnitt werden Verbesserungen in der Anwendung hinsichtlich des Kommunikationsmusters beschrieben. Es werden nicht nur Anzahl der Nachrichten, Nachrichtengröße und -verteilung berücksichtigt, sondern es wird eine ganzheitliche Sicht eingenommen: Was macht die Anwendung mit den kommunizierten Daten. Hier zeigt sich, dass existierende Anwendungen z. B. aus der Bioinformatik Potential für Optimierung haben. Zur Ausnutzung zukünftiger paralleler Prozessorarchitekturen wird es notwendig sein, die hier für MetaComputing vorgestellten Methoden auch für die Programmierung für Multicore-systeme zu verwenden. Es werden Anwendungen der Bioinformatik im Bereich RNA und DNA Analyse sowie der Differentialgleichungslöser `DD_filtre2`, entwickelt an der Universität von Houston, besprochen.

5.1 Anwendungen der Bioinformatik

Ein relativ neues Gebiet der medizinischen Forschung ist die Nutzung von Computern für die Simulation von biologischen Vorgängen. Unter anderem gibt es die folgenden Anwendungsgebiete:

- Numerische Simulation des Blutflusses innerhalb von Arterien, bspw. zur Berechnung des Verhaltens von so genannten Stents (engl. für endoluminale Gefäßprothesen), bei abdominalen Aneurysmen,
- Simulation zur Weiterverarbeitung von Computer Tomographie (CT) oder Scans mit Magnetresonanz Tomographie (NMR) zur Vorbereitung von Operationen und Knochentransplantationen,
- Simulation der chemischen Vorgänge innerhalb der Zelle,
- Simulation neuronaler Prozesse im Gehirn,
- Simulation zur Entwicklung und Vorhersage der Wirkungsweise von Medikamenten (meist mittels heuristischer Methoden),
- vergleichende Analysen der DNA-Sequenz.

Aus dem Bereich der Bioinformatik wurden mit PACX-MPI Anwendungen aus zweien der oben aufgeführten Gebiete optimiert. Für die Teilnahme beim HPC-Challenge Wettbewerb auf der SuperComputing 2002 und 2003 wurden die Anwendungen `fastDNAml` und `RNAfold` bearbeitet. Erstere berechnet den phylogenetischen Abstammungsbaum aus den DNA-Sequenzen unterschiedlicher Spezies, d. h. durch vergleichende Methoden werden anhand der DNA iterativ Relationen der Spezies hergeleitet und die Abstammungswahrscheinlichkeiten bestimmt. Mit diesem Code wurde auf der Supercomputing 2003 der HPC-Challenge Wettbewerb für die am weitesten verteilt laufende Anwendung gewonnen (mit insgesamt 641 Prozessoren in fünf Metacomputer verteilt über fünf Kontinente) [134]. Letztere Anwendung berechnet die zwei-dimensionale Faltung langer RNA-Sequenzen. Da die für diesen Algorithmus notwendigen Optimierungen für PACX-MPI interessanter sind, wird in dieser Arbeit auf `RNAfold` näher eingegangen.

5.1.1 Einführung in die Simulation der Faltung von RNA

Der genetische Code jedes Lebewesens wird in der Desoxyribonukleinsäure (DNA) gespeichert [144]. Die DNA befindet sich im Zellkern jeder Zelle des Organismus. Sie setzt sich zusammen aus einer Polymerverkettung der Proteine Adenin, Cytosin, Guanin und Thymin (Abk. A, C, G, bzw. T). Die Basen der DNA binden sich jeweils mit einer komplementären Base, A mit T und C mit G, weswegen die Konzentration der vier Basen innerhalb der DNA gleich sind.

Das Human Genome-Projekt [30] hatte das erklärte Ziel einerseits die Sequenzierung der menschlichen DNA, aber auch die Erforschung der Bedeutung jedes Genes. Durch das öffentliche und vor allem das ökonomische Interesse hat die Technik der Sequenzierung (engl. Gun-Shot Sequencing) sehr große Fortschritte erfahren. Die schiere Menge an Information ist das größte Problem der Genomforscher: Einerseits die angenommenen 50000 Gene innerhalb der ca. 3 Mrd. Basen zu identifizieren, andererseits noch die Funktion eines einzelnen Genes herauszufinden, wobei Gene auch wechselseitig mit anderen Genen interagieren können [30].

```
315L GTCCGGGCCG TATAGCTGGT CCAGTGGGTG TCGAAGTCGA          0x2080 6465 7461 0061 5f5f 3669 3638 672e 7465
319L TATCGTCCGA GACGTAGAAG GCGCCCACGA TCCAGTCCAG          0x2090 705f 5f63 6874 6e75 2e6b 7862 5f00 4c47
323L ACGACCGTCG TTGTCGTCGC TCAAACGCAT TTCCAGCGAG          0x20a0 424f 4c41 4f5f 4646 4553 5f54 4154 4c42
327L GTCTGGTTCA GCGTCTCATC GGCATCGAAG CGCGAGGCCG          0x20b0 5f45 5f00 6e65 0064 6f66 6570 406e 4740
```

Abbildung 5.1: Auszug eines beliebigen Ausschnitts einer Gensequenz (links) und Speicherabbild eines Programmcodes (rechts)

In [15] wurde der Vergleich dieser Aufgabe mit der Fehleranalyse eines Computerprogrammes anhand des Speicherabbildes des Programmes verglichen. Abbildung 5.1 zeigt die Analogie dieser beiden Probleme. Die Informatik hat mit Tools wie Disassemblern den Code in Prozessorbefehle lesbar und mit leistungsfähigen Debuggern jeden Programmabschnitt zu einer Zeile zuweisbar gemacht. Den Genomforschern steht bisher kein solches Mittel zur Verfügung. Es werden große Anstrengung unternommen, Tools zu entwickeln, mit denen sich Wirkungsweisen von Gensequenzen interaktiv über das Web annotieren lassen.

Während DNA für die Speicherung der reproduzierbare Geninformation zuständig ist, wird die durch die Polymerase aus der DNA der Zelle gebildete Ribonucleinsäure (RNA) vielseitig eingesetzt [109, 122]. Ribonucleinsäure wirkt unter anderem als Regulator für den Stoffwechsel innerhalb der Zelle und als Produzent von Botenstoffen zur Kommunikation mit anderen Zellen. Die RNA ist eine einsträngige Kopie der DNA (die Base Thymin wird ersetzt durch Uracil), die durch die so genannte Transkription im Polymeraseprozess aus der DNA innerhalb der Zelle abgelesen wird. Während DNA durch die Bindung zur Doppelhelix sehr stabil ist, unterliegt die RNA nach der Polymerase im Translationsprozess zur einkettigen Molekülkette einer Faltung, d. h. Basen verbinden sich unter Abgabe von frei werdender Energie. Die verschiedenen Kombinationen der möglichen lokalen Faltungen ergeben eine minimale freie Energie.

Diese dreidimensionale Faltung der Molekülkette legt die spätere Funktion in der Zelle fest; durch die Energiestruktur der entstandenen Oberfläche können die Moleküle an der passenden Stelle an Rezeptoren andocken. Mittels Röntgen- oder Magnetresonanz Tomographie (NMR) lässt sich die Struktur für kurze RNA-Sequenzen ableiten. Diese Verfahren sind aber sehr aufwändig und teuer und auch nur für kurze Sequenzen anwendbar. Numerische Simulation kann weitergehende Aussagen über die Faltung langer Sequenzen machen.

Der komplexe Vorgang der drei-dimensionalen Faltung ist *ab-initio* nicht in vernünftiger Zeit berechenbar. Heutige Molekular-Dynamik (MD) Programme können realistischerweise Moleküle mit wenigen hundert Atomen mit vertretbarem Aufwand rechnen. Für die Berechnung einer Faltungsreaktion langkettiger RNA-Moleküle ist dies allerdings zu aufwändig.

Für den zwei-dimensionalen Fall wurden aber schon sehr früh heuristische Methoden entwickelt [150], die den drei-dimensionalen Prozess annähern. Den Nachweis, dass die Annäherung der drei-dimensionalen eigentlichen Faltung entspricht, wurde in [6] gezeigt. Insofern lassen sich Anwendungen wie `RNAfold` für die Vorhersage von Faltungen von RNA-Sequenzen berechnen und damit evtl. Aussagen über den medizienischen Nutzen machen.

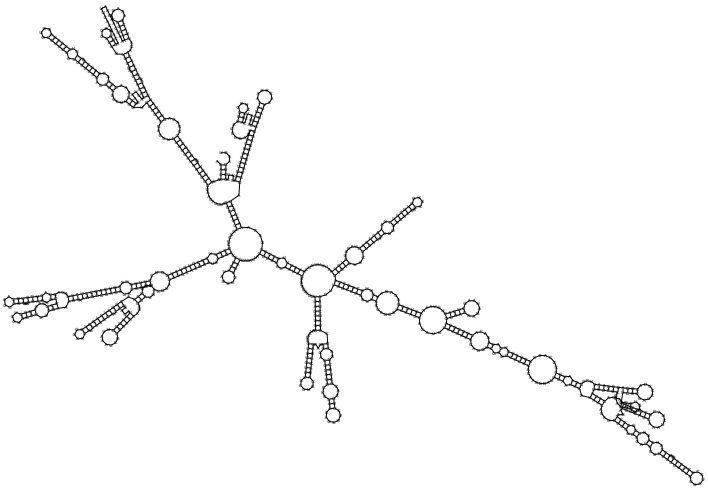

Abbildung 5.2: Beispiel der Ausgabe einer Faltung von `RNAfold`

5.1.2 Die Anwendung `RNAfold`

Die Anwendung `RNAfold` wurde im Rahmen des Vienna RNA-Paketes von Ivo Hofäcker an der Universität Wien entwickelt [66]. Es berechnet die Sekundärstruktur einer beliebigen RNA-Sequenz bei einer bestimmten Temperatur. `RNAfold` ist bereits MPI-parallelisiert. Die Berechnung hat für eine Sequenz der Länge n einen Aufwand von $O(n^3)$, außerdem sind $O(n^2)$ Kommunikationsschritte zur Lösung nötig.

In Zusammenarbeit mit Wissenschaftlern der Sandia National Labs wurde das Programm `RNAfold` zur Berechnung der zweidimensionalen Sekundärstruktur von RNA-Sequenzen erweitert. Um die Produktivität zu erhöhen und lange Sequenzen zu ermöglichen, waren einige Erweiterungen der Anwendung notwendig. Einerseits wurden die Heuristiken der Energieparameter verbessert, das Programm vom Autor in die kollaborative Visualisierungsumgebung Covise [147] integriert und für den Einsatz im MetaComputing die Kommunikation optimiert. Diese Entwicklungen und die Ausführung in einem Computational Grid wurden in [15] beschrieben. Die Ausführung auf einem verteilten Metacomputer machten mehrere Optimierungsschritte notwendig, diese verbessern aber auch die lokale Ausführung auf einem Cluster erheblich, wie in den folgenden Abschnitten erläutert wird.

Integration in die kollaborative Visualisierungsumgebung Covise

Die bisherige Visualisierungsmethode des Programmes entspricht Abb. 5.2 oder aber einer Abbildung mit einer kreisförmigen Repräsentation in Postscript-Dateien. Man erkennt sich gegenüberliegende Basenpaare, die sich in Strängen fortsetzen, bei nicht-kompatiblen Basenkombinationen ergeben sich Lücken, Ausbuchtungen oder Schleifen. Beide Darstellungen sind jedoch auf RNA-Ketten mit 3000 Basen limitiert sind [91] und können nur in Postscript-Darstellungsprogrammen angezeigt oder direkt gedruckt werden. Diese Art der Ausgabe ist aber für die Zusammenarbeit von Wissenschaftlern

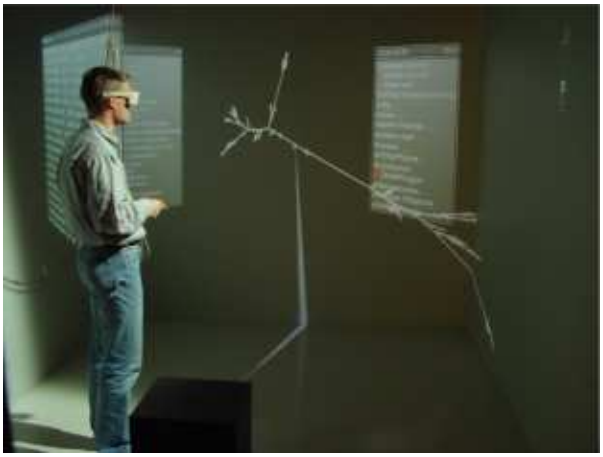

Abbildung 5.3: Visualisierung einer Faltung in einer Virtuellen Umgebung mit Covise

und vor allem für den Vergleich der Faltungen von unterschiedlichen Sequenzen nicht sehr hilfreich. Hierfür wurde ein neues Ausgabenformat mit den folgenden Informationen entworfen:

- Kommentar des Lokus (Ort innerhalb der RNA), Beschreibung der Sequenz,
- die minimale freie Energie und die Länge der Sequenz,
- die RNA-Sequenz,
- die berechnete Sekundärstruktur in Klammer-/Punkt-Notation, bspw. ((.(..).).),
- die Koordinaten jeder Base in der X,Y-Ebene,
- die berechnete Paarung der Basen.

In der Arbeit des Autors wurde `RNAfold` in die kollaborative Visualisierungsumgebung Covise [147] integriert und die Ausgabe der Faltung visualisiert. Der Vorteil für den Wissenschaftler ist, dass mehrere Faltungen unterschiedlicher Sequenzen oder bei unterschiedlichen Temperaturen visuell verglichen, übereinander gelegt oder im Raum vor sich verschoben und gedreht werden können. Gleichzeitig erlaubt Covise die Zusammenarbeit mehrerer Experten, die, obwohl physikalisch an unterschiedlichen Orten, in der Virtuellen Realität (VR)-Umgebung das gleiche Bild der Ausgabedaten von `RNAfold` sehen und miteinander interagieren können, wie Abb. 5.3 zeigt.

Mit der Integration von `RNAfold` in Covise wurde eine Steuerung der Simulation implementiert, d. h. es können neue RNA-Sequenzen geladen, die Temperatur gesetzt, Guanin-Uracil-Bindung erlaubt und dann die Simulation auf einem Höchstleistungsrechner dynamisch gestartet werden. Eine automatische Anzeige meldet den prozentualen Rechenfortschritt, bei zu langer Rechenzeit kann die Rechnung durch den Benutzer abgebrochen und evtl. auf einem anderen der zur Verfügung stehenden Rechner wieder gestartet werden. In Tests wurden damit mehrere Simulationen parallel auf unterschiedlichen Rechnern ausgeführt.

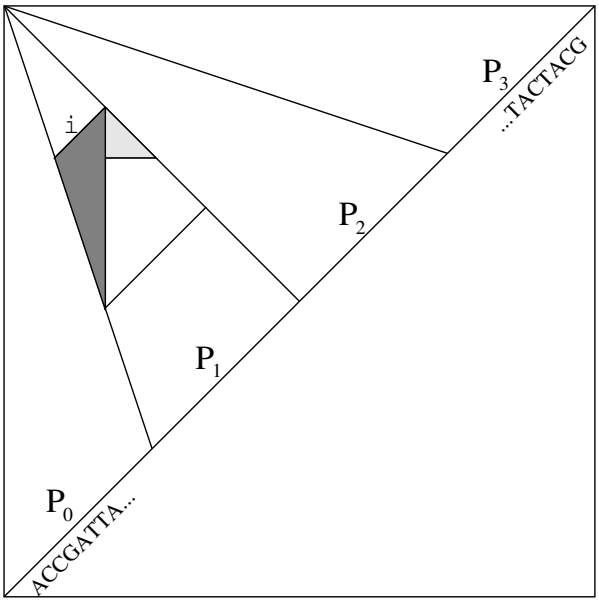

Abbildung 5.4: Aufteilung der RNA-Sequenz auf die Prozessoren

Reduktion des Speicherverbrauchs

Für die Berechnung der Minimalen Freien Energie (MFE) der Sequenz werden alle möglichen Kombinationen der Basen einer Sequenz verglichen. Hierfür werden anhand der Enthalpien der Kombinationen von bis zu sieben Basenpaaren (also 49 Kombinationen) die Verkettung und dadurch frei-werdenden Energien berechnet [64, 70, 107, 140]. Die Enthalpiematrizen von `RNAfold` wurden hierfür um weitere Basenpaarkombinationen [143] erweitert [82]. Die Basen der Eingabesequenz, sowie die finale Struktur sind jeweils als Zeichenketten gespeichert. Durch Umstrukturierung und Annotierung des Codes mit `inline`-Attributen konnten kleinere, häufig aufgerufene Routinen in den Ausführungspfad optimiert werden. Damit wurden Funktionsaufrufe für diese Routinen eingespart.

Für die Speicherung der freien Energie dieser Kombinationen dienen zwei Dreiecksmatrizen F^B und F^M der Größe $(n \times n)$, mit n als Länge der RNA-Sequenz. Der Vergleich der verschiedenen Kombinationsmöglichkeiten hängt auch von bis zu zwei Nachbarn der Base links und rechts innerhalb der Sequenz ab. Der Rechenaufwand besitzt damit die Komplexität $O(n^3)$. Die Rechenarbeit und damit die beiden Matrizen werden gleichmäßig entlang der Diagonalen zwischen den Prozessoren aufgeteilt, wie Abb. 5.4 darstellt. In Schritt i, von insgesamt n Schritten, berechnet jeder der m Prozessoren $(\frac{n}{m} - i)$ Werte. Jeder Prozessor muss also für die Berechnung am Rand Werte der Nachbarprozessoren verwenden. Daraufhin werden die berechneten Werte entlang der Linie zu Nachbarprozessoren kommuniziert.

Am Ende der Berechnung der lokalen freien Energie setzt Prozess 0 die minimale Struktur zusammen, hierfür werden von jedem Prozess die lokalen freie Energie aus den Matrizen F^B und F^M abgefragt (Backtracking). Dieser Schritt erweist sich als besonders kommunikationsintensiv, wie in den folgenden Abschnitten dargestellt.

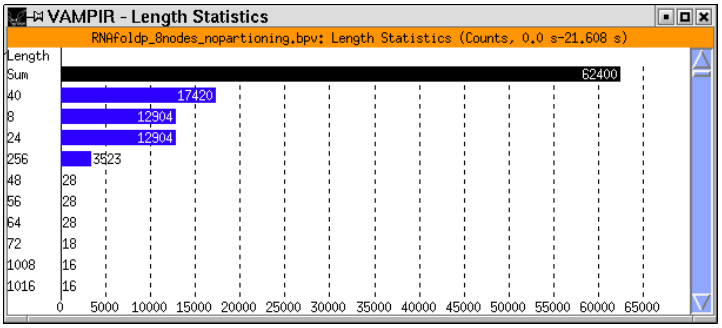

Abbildung 5.5: Anzahl der Nachrichten je Nachrichtenlänge

Verbesserungen der Kommunikation in `RNAfold`

Bei der Analyse des Programms fallen mehrere Punkte auf, die das Programm schlecht skalieren lassen. Einerseits ist der Speicherverbrauch für einfache Verwaltungsdaten hoch, wodurch auf der Entwicklungs- und Testplattform keine Tests auf starke Skalierung mit einer Startgröße von einem Knoten mit mehr als 2000 Basen durchgeführt werden konnten. Dies wurde durch komprimierte Speicherung der RNA-Sequenz sowie Bitsets für Verwaltungsdaten behoben, d. h. die Information, ob Reihen und Spalten der oben beschriebenen Matrizen belegt sind, werden mittels eines einzelnen Bits beschrieben.

Die Verbesserung der Kommunikation löste das zweite, hauptsächliche Problem. Die Analyse mittels Vampir [18] zeigte, dass die Kommunikation entlang der Hauptdiagonalen sehr hoch ist. Messungen mit einer 1000-basigen Testsequenz mit 8 Prozessen ergaben bspw. bis zu 7 Mio. Nachrichten von Prozess 0 zu Prozess 1, aber auch, dass der Kommunikationsbedarf von Prozess 0 zu anderen Prozessen und zurück zu hoch ist. Letzteres kommt vom Backtracking-Schritt am Ende der Berechnung der lokalen minimalen freien Energie. Vorerst soll die Kommunikation der Routine `fold` aus der Datei `foldpnew.c` betrachtet werden.

Vor der Berechnung der freien Energie entlang der Diagonalen des inneren Bereichs der Sequenz, werden die Randelemente gerechnet. Jeder Prozess kommuniziert $O(n)$ Elemente der bereits fertig berechneten Reihe/Spalte mit den Nachbarprozessen. Die Anzahl der zu kommunizierenden Elemente werden mittels eines Bresenham-Algorithmus bestimmt. Der Austausch findet nicht-blockierend statt, d. h. er wird mit der Berechnung der inneren Werten entlang der Diagonalen überlagert. Ursprünglich wurden hierfür die `fMLrow`, `crow`, sowie drei Zeichenketten aus `DMLrow` mit dem direkten Nachbar kommuniziert.

Die Länge der kommunizierten Nachrichten ist sehr kurz, wie die Vampir-Analyse in Abb. 5.5 zeigt: Von 62400 Nachrichten des oben genannten Falls sind 28% der Länge 40 Byte, 21% der Länge 8 Byte[1] und die gleiche Anzahl Nachrichten von 24 Byte Länge. Die nachfolgenden Nachrichtenlängen steigern sich jeweils um 8 Byte. Die jeweils um 8 Byte größeren Nachrichten entsprechen den Zeilen-, bzw. Spalten die jeweils zwischen zwei Nachbarprozessen ausgetauscht werden.

[1] 17420 Nachrichten der Länge 40 Byte, 12904 Nachrichten der Länge 8 Byte.

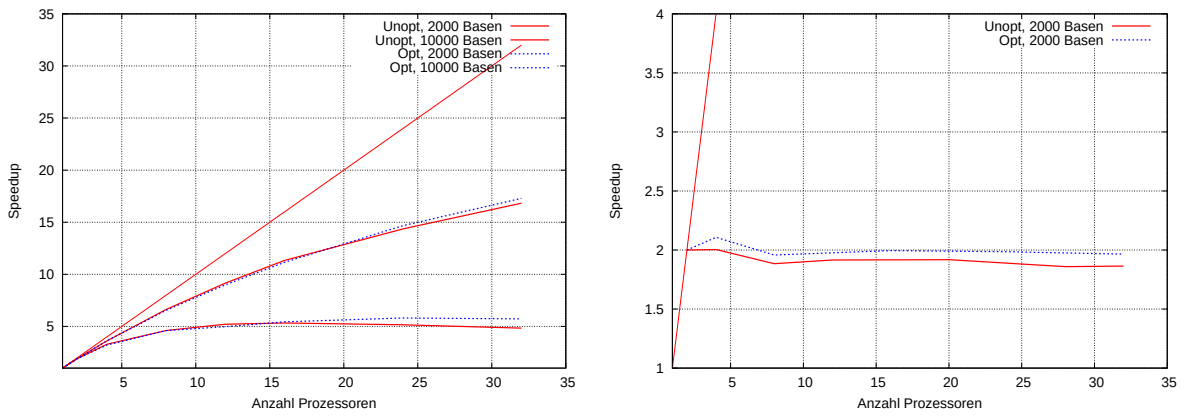

Abbildung 5.6: Ergebnisse der ersten Kommunikationsoptimierungen von `RNAfold`

Hier stellt sich heraus, dass sechs Kommunikationen, teilweise mit einer Länge von nur 20 Bytes, mit Nachbarprozessen entlang der Diagonalen zu einer einzigen Nachricht zusammengefasst werden können. Dadurch sinkt die Anzahl der Nachrichten entlang der Diagonalen um ein Sechstel, die Nachrichten verlängern sich entsprechend. Die Nachrichten können mittels `MPI_Pack` zusammen-, bzw. mit `MPI_Unpack` ausgepackt werden.

Analyse des Speedup der ursprünglichen und verbesserten Version

Um die parallele Effizienz zu analysieren, wurde `RNAfold` in mehreren Konfigurationen auf dem `dgrid`-Cluster und in Kombination mit dem `cacau`-Cluster ausgeführt. Für Informationen zu beiden Rechnern sei auf Tab. 4.1 verwiesen (siehe Abschnitt 4.1.2). Als Erstes sollen die oben aufgeführten Kommunikationsoptimierungen für die Anwendung auf einem Cluster, sowie in einem Computational Grid auf zwei Clustern analysiert werden.

In Abb. 5.6 wird der Speedup der nicht-optimierten (Unopt), sowie der ersten optimierten Version (Opt) dargestellt. Das linke Teilbild stellt den Speedup für das `dgrid`-Cluster dar, während das rechte Teilbild den Speedup für die Ausführung mit PACX-MPI zeigt. Da das Programm lokal mit nur einem Prozess ausgeführt werden kann, in der gekoppelten Variante mit PACX-MPI allerdings mindestens zwei Prozesse braucht, wird in letzterem Fall ein perfekter Speedup bei zwei Prozessen angenommen.

Um die unterschiedliche Skalierungen dieser beiden Sequenzen zu zeigen, werden für den Fall auf nur einem Cluster zwei Sequenzen, eine kurze der Länge 2000 und eine mittlerer Länge mit 10000 Basen, herangezogen. Man sieht, dass kurze Sequenzen wegen des geringen Arbeitsaufwandes nicht auf mehr als 16 Prozesse skalieren. Dennoch erlaubt die nebenläufige Kommunikation das Verstecken der Kommunikation bei großen Sequenzen. Es zeigt sich aber, dass die Optimierungen für das schnelle Infiniband (IB)-Netzwerk in der lokalen Ausführung selbst für große Sequenzen den Speedup um 10% nur marginal verbessern. Immerhin erlaubt die Optimierung das Skalieren auf bis zu 24 Prozesse für kleine Sequenzen.

Im rechten Teilbild ist der Speedup für eine kurze Sequenz auf einem Metacompu-

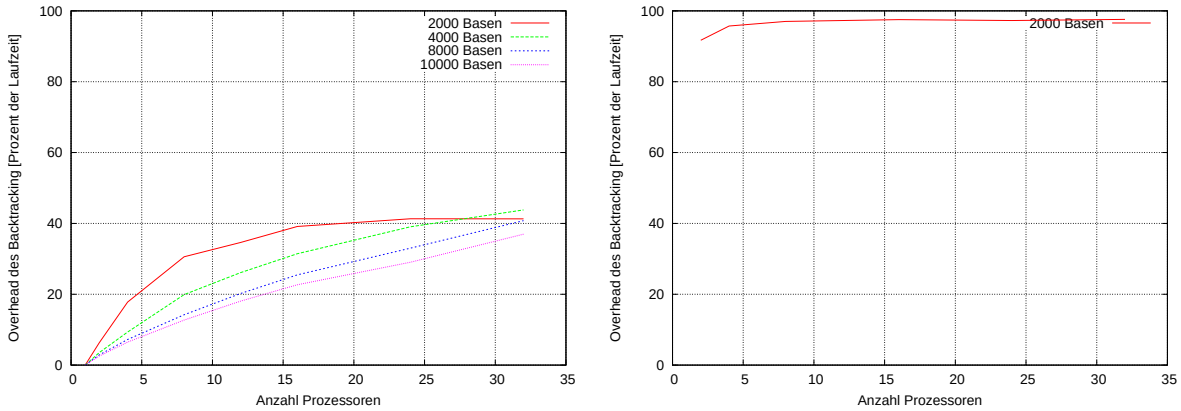

Abbildung 5.7: Prozentualer Overhead des Backtrackings

ter dargestellt. Für diesen mit PACX-MPI verteilten Fall ergibt das Zusammenfassen der bisher beschriebenen Optimierung eine Verbesserung um 10%. Dennoch skaliert die Anwendung für PACX-MPI überhaupt nicht[2].

Betrachtet man nun den prozentualen Zeitanteil des Backtracking-Schritts, fällt auf, dass dieser mit höherer Prozesszahl zu dominieren beginnt. In Abb. 5.7 ist dieser Anteil an der Gesamtlaufzeit dargestellt, links für den lokalen Fall auf einem Cluster, rechts für den verteilten Fall. Der Backtracking-Schritt belegt bei einer Sequenz mit 10000 Basen selbst im lokalen Fall 37% der Laufzeit (13,91 s bei bei einer Gesamtlaufzeit von 38,05 s). Für noch kürzere Sequenzen ist er entsprechend höher.

Für den verteilten Fall stellt diese Kommunikation den grössten Flaschenhals dar. Schon bei der kurzen Sequenz mit 2000 Basen wird mit 92% die Zeit fast nur in der Rekombination der Faltung verbracht, wie in Abb. 5.7 rechts zu sehen. Um diese Sequentialisierung zu eliminieren, wird das Kommunikationsmuster analysiert und eine Optimierung in Form von Caching und Prefetching in `RNAfold` eingebaut.

Elimination von Kommunikation durch Caching und Prefetching

Die Nachrichten der Länge 8 und 24 Byte werden für das Sammeln der Faltung mit der Minimalen Freien Energie (MFE) auf Prozessor 0 kommuniziert. Mittels einer 12 Byte Anfrage (ein 4 Byte Integer jeweils für Abfrage zur Matrix, X- und Y-Koordinate innerhalb der Matrix), erhält Prozess 0 eine 8 Byte Antwort (Datentyp `double` für die freiwerdende Energie).

Für diese 1000-Basen Sequenz ergeben sich insgesamt 12904 Abfragen zum Zusammensetzen der Faltung. Hierfür werden 9819 Elemente mehrfach, ein spezifisches Matrizenelement sogar bis zu 700 Mal abgefragt. Der Zugriff auf die Elemente der F^M- und der F^B-Matrix ist in Abb. 5.8 dargestellt (Farben repräsentieren die Reihenfolge der Abfragen; von blau über grün nach gelb zu rot). Man sieht, dass die Abfragen auf die (dicht besetzte) Matrix einem relativ geringen Teilbereich betreffen. Diese hat die

[2]Der Speedup wird anhand von 2 Prozessen normiert.

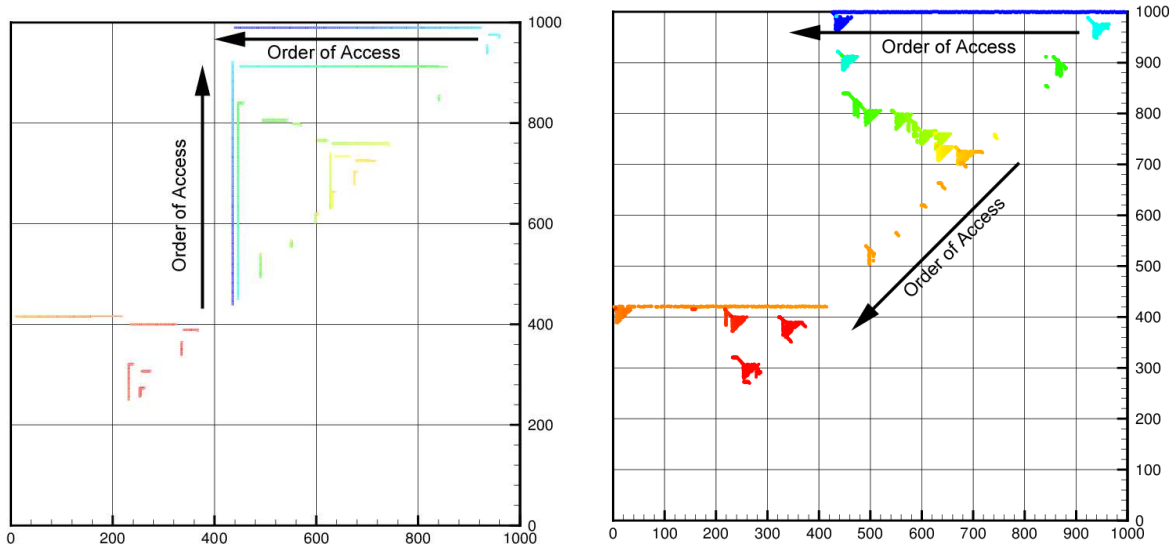

Abbildung 5.8: Zugriffsmuster auf die F^M-Matrix (links) und die F^B-Matrix (rechts)

Qualität einer dünn besetzten Matrix (engl. Sparse Matrix). Es fallen weiterhin mehrere Eigenheiten des Abfragealgorithmus auf: Einerseits sind die Abfragen sehr begrenzt, es werden nur bestimmte Matrixelemente abgefragt. Andererseits hat der Zugriff auf beide Matrizen eine sehr regelmäßige Struktur. Aus der F^M-Matrix werden die Zeilen von links nach rechts (oder in umgekehrter Reihenfolge von rechts nach links) oder Spalten von unten nach oben (und wieder umgekehrt) abgefragt. Für die F^B-Matriz ergibt sich eine Abfragecharakteristik der Form einer oberen Dreiecks-Matrix der Größe 50 bei der hier verwendeten Sequenz mit 1000 Basen. Weiterhin ist zu beachten dass die Abfragen je nach Eingabefall[3] immer anders und dennoch von der Qualität der Abfrage immer ähnlich ist.

Um diese temporale und räumliche Lokalität bei der Abfrage durch Prozessor 0 auszunutzen, wurde ein Algorithmus implementiert, der einerseits benutzerdefinierte Caches und Prefetches- basierend auf einer Heuristik auf Anwendungsebene implementiert. Dies vermeidet Kommunikation, da die Abfragen erst im Cache gesucht werden, der Prefetch-Mechanismus bewirkt, dass (vermutlich) später abgefragte Werte bereits in dem Zwischenspeicher für die nachfolgenden Zugriffe bereitstehen. Weiterhin werden bei einem Prefetch größere Datenmengen und damit effizienter kommuniziert.

Die Größe des Caches, sowie die Größe der abgefragten Werte sind je F^M und F^B einstellbar. Der Algorithmus hat allerdings heuristischen Charakter, da nicht deterministisch bestimmt werden kann, welche Größen und abgefragten Werte der optimalen Abfragenreihenfolge passen. Die Abfrage nicht mehr nur eines Wertes, sondern einer ganzen Zeile, bzw. Spalte für F^M und eines ganzen Blocks in einer Umgebung um den abzufragenden Wert für F^B vergrößert somit die Länge der Kommunikation und reduziert damit den prozentualen Overhead. Die Kommunikationszeit verlängert sich für die

[3] Je nach Sequenz, Temperatur und anderer Parameter und damit der lokalen Minimalen freien Energie.

zusätzlichen Werte nur geringfügig, ebenso sind die notwendigen Kopieroperationen in den Sendepuffer und in den Cache, sowie deren Verwendung im Vergleich zu den Kosten der Kommunikation minimal.

Die Heuristik der Erkennung der Ausrichtung der nachfolgenden Abfragen innerhalb der F^M-Matrix ist einfach: basierend auf der letzten Abfrage kann zwischen den vier Fällen (Zeile ab-/aufsteigend, Spalte ab-/aufsteigend) unterschieden werden. Für die F^B-Matrix zeigte sich eine gedachte Aufteilung der Gesamtmatrix in Blöcke der Prefetch-Blockgröße als ideal, um die Kommunikation bereits gecachter Daten zu vermeiden. Der linke, obere Punkt der kommunizierten Matrix wird modulo der Blockgröße gesetzt, um eine Kommunikation von bereits gecachten Werten zu verhindern.

In diesem Zusammenhang muss der Cache-Algorithmus kritisch auf seine Effizienz analysiert werden. Die folgenden unerwünschten Effekte können bei Caching und Prefetching auftreten:

1. Es werden Daten kommuniziert und gecached, die nie oder nur einmal abgefragt werden. Dies hat zur Folge, dass unnötigerweise Platz im Cache verschwendet wird, der für andere, mehrfach abgefragte Werte fehlt.

2. Es werden kommuniziert, die bereits im Cache enhalten sind. Durch die Heuristik können Werte, bspw. innerhalb einer Reihe oder Spalte transferiert werden, die bereits im Cache vorhanden sind.

3. Es werden Daten, die noch benötigt werden, aus dem Cache durch andere Daten verdrängt. Der Anteil der unabsichtlicht verdrängten Werte kann nur ermittelt werden, wenn der Cache-Algorithmus, sowie die Abfragenreihenfolge des gesamten Backtracking-Schritts bekannt ist.

Diese Probleme werden im folgenden Abschnitt anhand der Implementierung in `RNAfold` analysiert. Da die Abfragereihenfolge dynamisch von der Sequenz und anderen Parametern wie Temperatur abhängt, lässt sich das erste Problem nicht zufriedenstellend lösen: der Cache- und Prefetch-Algorithmus kann auf keine Weise vorhersagen, welche der vorher transferierten Daten doch nicht gebraucht werden. Auch eine Verbesserung der Heuristik ist nicht möglich da die abzufragenden Werte von den Parametern der Simulation abhängen.

Das zweite Problem wurde dadurch gelöst, dass Daten immer ausgerichtet an der Prefetch-Größe abgefragt werden. Für die lineare Abfrage der F^M-Matrix wird der Beginn modulo der Cache-Länge normiert, Abb. 5.9 zeigt die kommunizierten Blöcke anhand des Beispiels einer $3\,x\,3$-Aufteilung bei der Abfrage der F^B-Matrix. Es werden in diesem Beispiel also neun Werte in einer Nachricht anstatt nur eines Wertes versandt und gecached. Während die weissen Blöcke komplett kommuniziert werden, werden an den Grenzen der gespeicherten Teilmatrix die Werte, die nicht auf dem abgefragten Prozessor gespeichert werden ausmaskiert und nicht kommuniziert. Da in diesem Beispiel der abgefragte Block im Verhältnis zur Größe der Daten pro Prozessor groß ist, erscheint die Abfrage ineffizient, für größere Matrizen jedoch wird der Randblöcke kleiner.

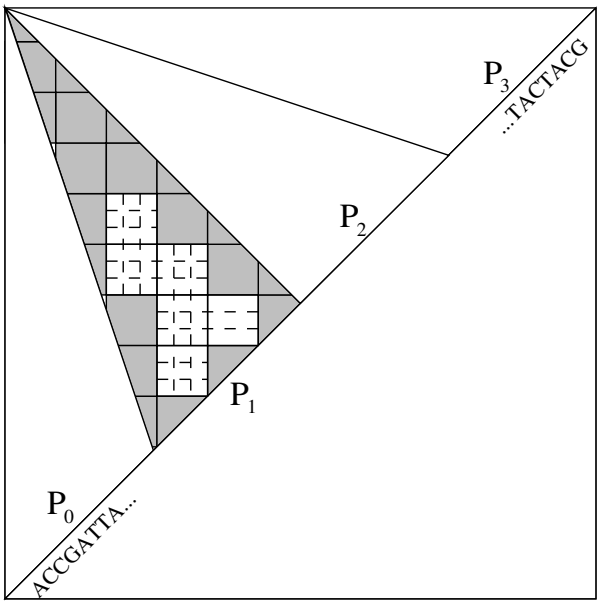

Abbildung 5.9: Aufteilung in $3\,x\,3$-Blöcke beim Cache/Prefetch-Verfahren

Basen	Cache-Hits	Cache-Misses	Prozent
2000	60592	4209	6,5%
4000	162079	9993	5,8%
8000	694509	34205	4,7%
10000	1124849	52863	4,5%

Tabelle 5.1: Cache-Verhalten verschiedener Sequenzen mit PACX-MPI

Für einen verteilten Lauf mit einer Eingabesequenz von 2000 Basen verbessert sich die Ausführung von Anfangs 130,9 s auf 13,67 s bei 32 Prozessen, eine Beschleunigung um den Faktor 10!

Dennoch sollen die Werte für das Cache-Verhalten in diesem Lauf mit 2000 Basen mit PACX-MPI betrachtet werden. In Tab. 5.1 wird das Cache-Verhalten für verschiedene Sequenzen bei 8 Prozessen aufgeführt. Mit maximal 6,5% der Zugriffe auf nicht vorhandene Elemente (engl. Cache-Misses) liegt das Cacheverhalten in einem sehr guten Bereich. Es zeigt sich, dass für längere Sequenzen bis 10000 Basen die Cache-Misses prozentual auf 4,5% der gesamten Zugriffe abnehmen. Dennoch ist die Gesamtausführungszeit immer noch zu langsam, da die Abfragen der bis zu 52863 Cache-Misses immer noch die Laufzeit dominieren.

Speicherung auf dem Root

Die Speicherung der Matrizen mit einer Länge von 10000 Basen belegt ca. 1,2 GB Speicher. Besitzt der Prozess mit dem MPI-Rang genügend Hauptspeicher, lassen sich Daten der F^M und F^B Matrizen auf dem Root sammeln. Dies basiert auf der Struktur des Algorithmus, der für die Berechnung einer Zelle auf die Daten des Nachbarprozessors

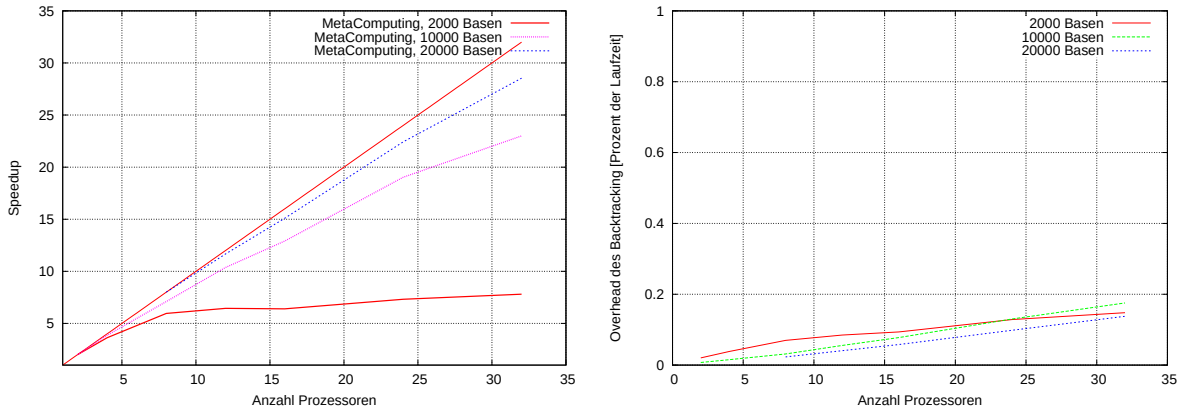

Abbildung 5.10: Speedup und Overhead mit Caching und Speicherung auf dem Root

angewiesen ist. Somit kann Prozess 0 sukzessive alle Daten sammeln, was wiederum die Speicherung und Abfrage mittels des Caching-Algorithmus im Backtracking-Schritt unnötig macht.

In Abb. 5.10 (links) ist der Speedup für diesen MetaComputing-getauften Algorithmus dargestellt. Da der Root als einziger Prozess auf einem Knoten des `dgrid`-Clusters mit 8 GB Hauptspeicher ausgeführt wird, lassen sich nicht nur Sequenzen mit 2000, 10000, sondern mit bis zu 20000 Basen simulieren. Einerseits liefert `RNAfold` auch für mittlere Sequenzlängen mit 10000 Basen einen akzeptablen Speedup von 23 bei 32 Prozessen. Andererseits ist nur mit dieser Version erstmals eine Ausführung mit 20000 Basen auf einem Metacomputer möglich. Bei diesem großen Fall wird ein Speedup von 28,5 bei 32 Prozessen erreicht. Da die Kommunikation des Backtracking-Schrittes nun inherent in der Zeilen-/Spaltenaustausch der Funktion `fold` versteckt ist, beträgt der Anteil des Backtrackings im MetaComputing Fall weniger als 1% der Gesamtlaufzeit, wie in Abb. 5.10 (rechts) zu sehen.

Lokale Ausführung mit Caching und Prefetching

Nicht jeder Knoten eines Clusters wird den notwendigen Hauptspeicher aufweisen, die gesamten Matrizen auf dem Prozess 0 zu halten. Deswegen soll aufgezeigt werden, wie der Caching und Prefetching die lokale Ausführung auf einem Cluster beschleunigt.

In Abb. 5.11 (links) ist der Speedup der Version mit Caches (Caching) und Prefetching im Vergleich zur ersten, optimierten Version dargestellt. Diese Caching erreicht auch für Sequenzen mit 10000 Basen einen akzeptablen Speedup von 26 bei 32 Prozessen im Vergleich zur ersten optimierten Version, welche nur einen Speedup von 17 bei 32 Prozessen erreicht.

Weiterhin wird mit dem Caching-Mechanismus erreicht, dass der Anteil des Backtracking Schritts bei 32 Prozessen im Vergleich zur ersten, optimierten Version von 36,9% auf nun 4,5% der Gesamtausführungszeit sinkt. Das zeigt die Bedeutung dieses Mechanismus auch für die lokale Ausführung auf einem Cluster mit schnellem Infiniband (IB)-DDR Netzwerk.

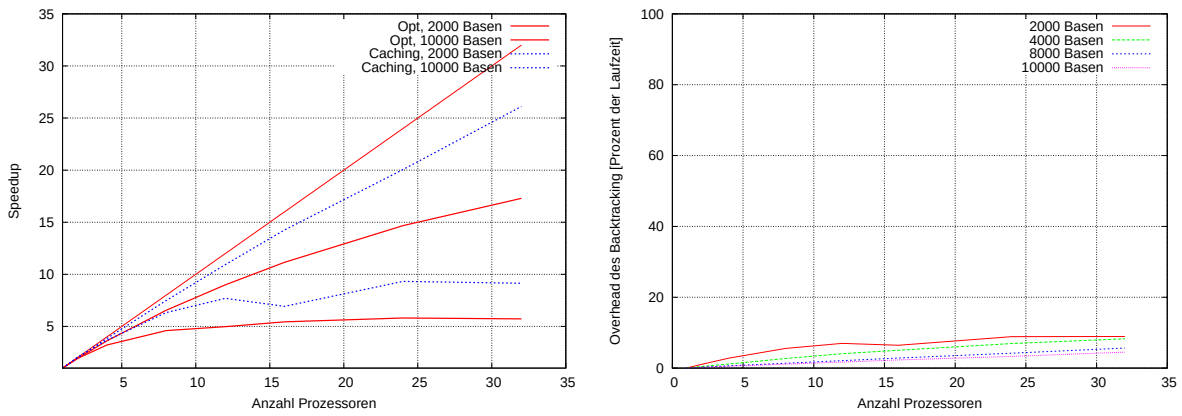

Abbildung 5.11: Ergebnisse der Optimierungen mittels Caching und Prefetching

	2000 Basen		10000 Basen	
Prozesse	**Opt**	**PACX-MPI**	**Opt**	**PACX-MPI**
1	10,15 s	—	644,37 s	—
2	5,28 s	10,53 s	335,91 s	516,06 s
4	3,15 s	5,81 s	178,20 s	267,57 s
8	2,21 s	3,53 s	98,40 s	145,41 s
12	2,04 s	3,26 s	71,63 s	99,56 s
16	1,86 s	3,29 s	57,75 s	79,99 s
24	1,75 s	2,88 s	43,91 s	54,24 s
32	1,77 s	2,69 s	37,26 s	44,88 s

Tabelle 5.2: Absolute Zeiten mit Optimierung, sowie MetaComputing (PACX-MPI)

Als Fazit ist zu bemerken, dass diese Anwendung auf Grund des Kommunikationsaufkommens auf einem Metacomputer bis zu zwei Mal so lange braucht, wie lokal auf nur einem Cluster. Tabelle 5.2 stellt die absoluten Zahlen für die Sequenzen mit 2000 und 10000 Basen auf dem lokalen, sowie dem Metacomputer dar. Für längere Sequenzen ist der Overhead der Kommunikation durch PACX-MPI nicht mehr groß, da die externe Kommunikation durch die Dämonen entkoppelt wird. Je weiter die Lösung in Richtung der oberen Ecke fortschreitet, werden die zu versteckenden Transfers kürzer.

5.2 Anwendungen von Differentialgleichungslöser

Physikalische Probleme können durch Differentialgleichungen (DGL) in Abhängigkeit von Grundgrößen wie Ort, Zeit oder Masse beschrieben werden [55]. Hängt die unbekannte DGL von nur einer unabhängigen Variablen ab, handelt es sich um eine gewöhnliche Differentialgleichung, anderfalls spricht man von einer partiellen Differentialgleichung. Die meisten physikalischen Probleme basieren auf partiellen DGL mit Ableitungen zweiter Ordnung von mehreren Variablen. Unter anderem basieren elektromagnetische Feld- und die Navier-Stokes-Gleichungen für Strömungssimulation auf partiellen Differential-

gleichungen höherer Ordnung. Die allgemeine Form dieser Differentialgleichungen zweiter Ordnung werden beschrieben durch:

$$\left(\sum_{i,j} \alpha_{ij} \frac{\partial^2}{\partial x_i \, \partial x_j} + \sum_k \beta_k \frac{\partial}{\partial x_k} + \frac{\partial}{\partial t}\right) \phi = 0 \tag{5.1}$$

mit α_{ij} und β_k als Polynom (oder Skalaren) als Gewichtung des Partialquotienten. Da davon ausgegangen werden kann, dass ∂x_i der Ableitung ∂x_j entspricht, kann angenommen werden, dass $\alpha_{ij} = \alpha_{ji}$, d. h. die Matrix α ist symmetrisch. In Abhängigkeit der Eigenwerte der Matrix α können DGL in das folgende Schema eingegliedert werden, wobei sich die Eigenwerte λ der Matrix α aus den Nullstellen des Polynoms n-ter Ordnung $det(A - \lambda I)$ ergeben (mit I der $n \times n$-Einheitsmatrix):

- Parabolische DGL: Die Matrix α hat mindestens einen Eigenwert 0. Beispiele für DGLs zweier Variablen, wobei die erste einmal und die zweite zweimal abgeleitet wird, sind die Diffusionsgleichung und die zeitabhängige Schrödingergleichung. Beide sind von der Form:
$$\frac{\partial}{\partial t} + \frac{\partial^2}{\partial x^2} = 0$$

- Elliptische DGL: Die Matrix besitzt nur positive oder nur negative Eigenwerte λ, d. h. die Matrix ist symmetrisch, positiv definit (SPD). Beispiele hierfür sind die Poissongleichung, sowie die zeitunabhängige Schrödingergleichung, beide mit zwei oder mehreren Ortsvariablen. Die stationäre Lösung einer parabolischen Differentialgleichung (DGL) entspricht einer elliptischen DGL.
$$\pm\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right) = 0$$

- Hyperbolische DGL: Die Matrix besitzt keine Eigenwerte mit 0 und entweder einen positiven oder einen negativen Eigenwert. Alle anderen Eigenwerte haben ein entgegengesetztes Vorzeichen. Das heißt bei hyperbolischen DGL ist die Koeffizientenmatrix nicht positiv definit. Ein Beispiel hierfür ist die Wellengleichung von Instrumentensaiten. Mit dem Wissen über die Zusammensetzung der Wellen aus Grundschwingungen wird zur Lösung von Hyperbolischen DGL meist eine Eigenmodenanalyse angewandt.
$$\pm\frac{\partial^2}{\partial x^2} \pm \frac{\partial^2}{\partial y^2} = 0$$

Da DGLen potentiell unendliche viele Lösungen besitzen, müssen weitere Bedingungen vorgegeben werden, die die Lösung festlegen:

- Randbedingungen mit festen Werten am Definitionsgebiet.

- Anfangsbedingungen der Startwerte der Variablen und deren Ableitungen.

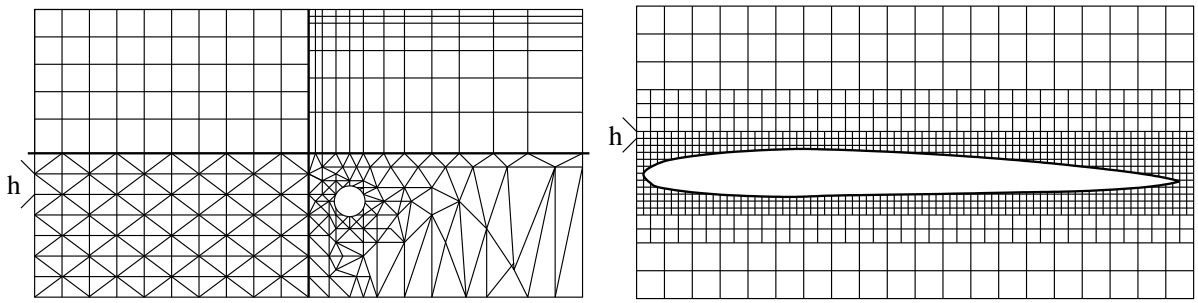

Abbildung 5.12: Verschiedene Gittertypen und verfeinerte Gitter um ein Flügelprofil

Methode	Konvergenzrate	Asymptotische Konvergenzrate
Jacobi	$-log(cos\ h)$	$h^2/2$
Gauss-Seidel (GS)	$-log(cos^2\ h)$	h^2
SOR	$-log\left(\frac{1-sin\ h}{1+sin\ h}\right)$	$2h$
Zeilen Jacobi	$-log\left(\frac{cos\ h}{2-cos\ h}\right)$	h^2
Zeilen GS	$-log\left(\frac{cos\ h}{2-cos\ h}\right)^2$	$2h^2$
Zeilen SOR	$-log\left(\frac{1-\sqrt{2}sin(h/2)}{1+\sqrt{2}sin(h/2)}\right)^2$	$2\sqrt{2}h$

Tabelle 5.3: Konvergenzraten iterativer Gleichungslöser mit elliptischen Differentialgleichungen, diskretisiert auf einem rektilinearen, äquidistanten Gitter

Differentialgleichungen höherer Ordnung lassen sich rekursiv auf Differentialgleichungen niedrigerer Ordnung reduzieren, bis ein Differentialgleichungssystem (DGLS) erster Ordnung entsteht. Nur die einfachsten DGL lassen sich analytisch lösen, die meisten DGLS werden deswegen iterativ gelöst. Für diese existieren Differentialgleichungslöser mit verschiedenen Eigenschaften, teilweise sind sie speziell auf das zu lösende Problem angepasst. Vielen iterativen Methoden ist gemein, dass das zu lösende physikalische Problem auf einem Gitter diskretisiert werden muss. Hierfür wird das Lösungsgebiet in diskrete Punkte unterteilt, auf dem die physikalischen Größen bestimmt werden. Da die Genauigkeit der Lösung sehr stark von der Genauigkeit der Gitter abhängt, wurden sehr unterschiedliche Diskretisierungsmethoden entwickelt. Hier kann man unterscheiden in die Diskretisierung im physikalischen Raum, sowie im Phasenraum. Da schlussendlich eine Lösung im physikalischen Raum angestrebt wird, muss somit in und aus dem Phasenraum transformiert werden.

Abbildung 5.12 (links) zeigt von links oben die Diskretisierungen eines Gebietes anhand eines orthogonalen, äquidistanten Gitters, eines in beiden Dimensionen rectilinearen Gitters, eines regulären Dreieckgitters und schließlich ein unreguläres Gitter um einen Kreissausschnittes, wie es in Strömungssimulationen um einen Zylinder verwendet wird. Auf der rechten Seite von Abb. 5.12 ist ein mehrfach verfeinertes Gitter um ein fiktives Flügelprofil dargestellt. Die Auflösung des Gitters (hier beispielhaft gezeigt anhand von h) bestimmt maßgeblich die Genauigkeit der Lösung.

Ist das Gebiet geeignet diskretisiert, werden die Differenzenquotienten in einer Matrix gespeichert und das entstehende Gleichungssystem mit Hilfe der linearen Algebra gelöst werden. Es existieren verschiedene iterative Verfahren zum Lösen von DGLS. Tabelle 5.3 gibt für gebräuchliche Lösungsverfahren die Konvergenzrate in Abhängigkeit der Gitterbreite h auf einem orthogonalen, äquidistanten Gitter auf einem quadratischen Gebiet der Kantenlänge π mit einer elliptischen Differentialgleichung.

5.2.1 Die Anwendung `DD_Filtre2`

Das Programm `DD_filtre2` [7, 52] löst ein System von Reaktions-Diffusions-Konvektionsgleichungen. Diese Arten von Gleichungen finden bspw. Anwendung in Simulationen zur Berechnung der Luftqualität. Der Gleichungslöser arbeitet auf Formeln der folgenden Art:

$$\frac{\partial C}{\partial t} = \Delta C + (\vec{a}(x,y,z,t).\nabla)C + F(x,y,z,t,C). \tag{5.2}$$

Hierbei ist $C \equiv C(x,y,z,t) \in R^m$ die chemische Zusammensetzung der Luft mit m der Anzahl der Spezies, sowie $(x,y,z) \in \Omega \subset \mathbf{R}^3$ die Koordinaten. Das Vektorfeld $\vec{a}$ im Raum $\mathbf{R}^3$ beschreibt die Windrichtung, während F die Reaktionsfunktion über alle chemischen Spezies in einem Punkt im Raum zum Zeitpunkt t definiert. Die Anzahl m der zu rechnenden Spezies kann potentiell sehr groß werden (wegen chemischer Zwischenprodukte mehrere 100) und damit einhergehend das zu lösende gewöhnliche Differentialgleichungssystem:

$$\frac{\partial C}{\partial t} = F(x,y,z,t,C). \tag{5.3}$$

Dieses System ist steif [29, 141]. Mit der Notation des totalen Differentials $\frac{D}{Dt}$ kann die Formel 5.2 umgeschrieben werden in:

$$\frac{DC}{Dt} = \Delta C + F(x,y,z,t,C), \tag{5.4}$$

Zur Lösung wurden verschiedene Zeit-Integrationsmethoden vorgeschlagen [141]. Für Reaktions-Diffusions-Gleichungen wurde in [37] ein numerisch effizienter, skalierender Löser vorgeschlagen. Die Diskretisierung der Gleichung 5.2 erfolgt auf einem kartesischen 3D-Gitter nach dem Euler-Verfahren erster Ordnung:

$$\frac{C^{n+1} - C^{n,*}}{dt} = \Delta C^n + F(x,y,z,t,C^{n+1}), \tag{5.5}$$

oder alternativ mittels dem Verfahren zweiter Ordnung:

$$\frac{3C^{n+1} - 4C^{n,*} + C^{n-1,*}}{2\,dt} = 2\Delta C^n - \Delta C^{n-1} + F(x,y,z,t,C^{n+1}), \tag{5.6}$$

wobei $C^{n,*}$ die Näherung erster Ordnung aus der folgenden Gleichung geschlossen werden kann:

$$C(x,y,z,t^{n+1}) = C(x - \vec{a}_x^n\,dt, y - \vec{a}_y^n\,dt, z - \vec{a}_z^n\,dt, t^n) = C^{n,*}. \tag{5.7}$$

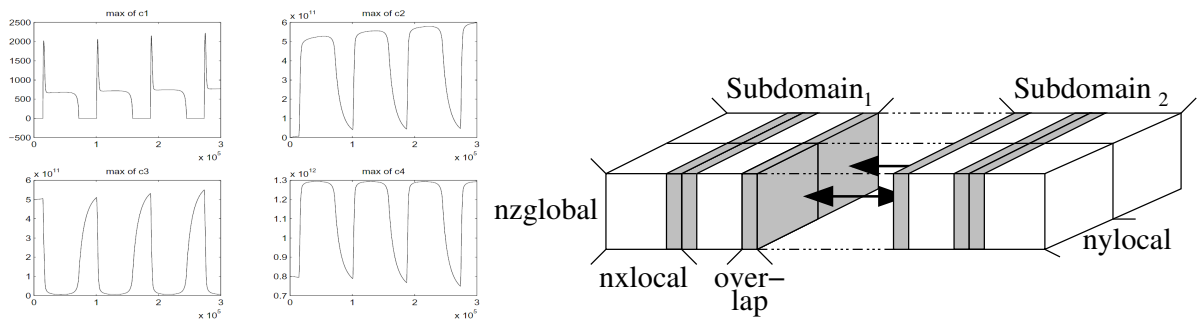

Abbildung 5.13: L_∞-Norm der vier von `DD_Filtre2` berechneten Spezies (links) und Domain Decomposition für 2 Subdomains (rechts)

Der explizite Diffusionsterm ΔC wird drei-dimensional über einem Standard Sieben-Punkt-Stern aufsummiert. Der Algorithmus von Dupros et al ist ein schneller Löser für Reaktions-Konvektionsgleichungen, mit guten Stabilitätseigenschaften in Bezug auf den Zeitschritt, ohne die komplette Jacobi-Matrix des gesamten Domains bei jedem Zeitschritt berechnen zu müssen. Üblicherweise wird die Methode des Operator-Splittings mit einem schnellen nicht-linearen Löser für gewöhnliche DGL verwendet. Allerdings verhindert die Steifigkeit des Reaktions-Teils der Gleichung einen Operator-Splitting mit höherer Ordnung. Der vorgeschlagene Algorithmus sieht die Domain Decomposition (DD)-Methode mit Filterung von hohen Frequenzen zwischen Domains vor [37].

Aus der Untersuchung folgt, dass nur die hohen Frequenzen den kleinen Zeitschritt mit $dt = O(h^2)$ aus der Courant-Friedrich-Lewy (CFL)-Bedingung [88] erzwingen. Die Idee dieses Verfahrens ist, durch Filterung diese Frequenzen herauszufiltern, um die Zeitschrittbedingung abzuschwächen, aber die Genauigkeit zweiter Ordnung im Raum beizubehalten. Dieses Verfahren zur Stabilisierung ist allerdings auf Gitter limitiert, die auf reguläre Gitter abgebildet werden können. Um die Filterung zu vereinfachen, wird keine Fast-Fourier-Transformation (FFT) mit der inherent globalen Kommunikation durchgeführt, sondern die Filterung durch eine Matrizenmultiplikation angenähert.
In jeder Iteration folgt das Verfahren dem Schema:

$$C^n \xrightarrow{\mathcal{Q}} C^{n+1} \xrightarrow{\mathcal{S}} \tilde{C}^{*,n+1} \xrightarrow{\mathcal{P}_{\sin}} \hat{C}^{n+1} \xrightarrow{\mathcal{D}_\sigma} \hat{C}_\sigma^{n+1} \xrightarrow{\mathcal{P}_{\sin}^{-1}} \tilde{C}^{n+1} \xrightarrow{\mathcal{S}^{-1}} C^{n+1},$$

wobei $\mathcal{Q}$ das Zeitschrittverfahren erster oder zweiter Ordnung aus Gleichung 5.5 bzw. 5.6 ist und $\mathcal{S}$ die homogenen Randbedingungen durch eine Punkt-affine Transformation herstellt. Die Frequenzen werden durch die drei-dimensionale trigonometrische Umrechnung $\mathcal{P}_{sin}$, die anschließende Filterung $\mathcal{D}_\sigma$ und die Rücktransformation $\mathcal{P}_{sin}^{-1}$ als Matrizenoperationen durchgeführt und werden deswegen in der Implementierung als eine Multiplikation $\mathcal{P}_{\sin}^{-1} \times \mathcal{D}_\sigma \times \mathcal{P}_{\sin}$ ausgeführt.

Der derzeit implementierte Fall verwendet vier Spezies in der Berechnung der Ausbreitung von Ozon. In Abbildung 5.13 (links) sind die Konzentrationen der vier Spezies über einen Zeitraum von 96 Stunden (4 Tage) dargestellt. Mit c_1 der Konzentration von hoch-reaktivem (atomaren) Sauerstoff ($O(3P)$), c_2 entspricht Stickstoffmonoxid (NO), c_3 entspricht Stickstoffperoxid (NO_2) und c_4 entspricht Ozon (O_3).

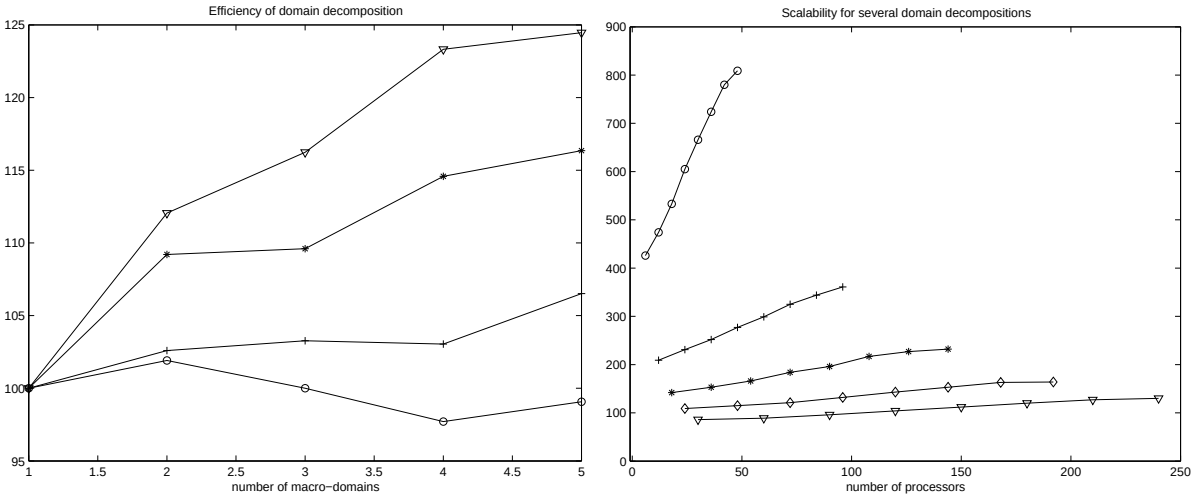

Abbildung 5.14: Effizienz von `DD_Filtre2` in Abhängigkeit der Subdomains

Die Aufteilung der Domains auf Prozessoren erfolgt wie in Abbildung 5.13 (rechts) angedeutet, hier mit vier Prozessen je Subdomain, getrennt entlang der x-Achse. Die Gebietszerlegung innerhalb eines Clusters erfolgt zwei-dimensional. Am Ende jedes Zeitschritts tauschen die beiden Systeme innerhalb des Metacomputers die Randgebiete, wie im Additive Schwarz Algorithmus [7, 120]. Es wurde gezeigt, dass mit einem hinreichend großen Überlapp (graue Gebiete in Abb. 5.13), die hohen Frequenzen ausgefiltert werden können und damit ein stabiles Zeitschrittverfahren verwendet werden kann. Für eine allgemeine Analyse von expliziten Gebietszerlegungsmethoden basierend auf impliziten Subdomain-Lösern für parabolische DGL sei auf [149] verwiesen. Die Umsetzung der Subdomain-lokalen Filtertechniken werden in [90] vorgestellt.

Lokale Ausführung zur Untersuchung der Subdomains

Erste Ergebnisse wurden auf der Cray T3E gesammelt, um die Auswirkung der Anzahl von Subdomains zu erörtern [52]. Diese Plattform wurde gewählt, da sie eine sehr gutes Verhältnis von Kommunikations- zu Rechengeschwindigkeit erzielt und damit die Kommunikation im Vergleich zu MetaComputing-Experimenten zu vernachlässigen ist. Vielmehr interessiert die numerische Stabilität des Filterverfahrens auf die Konvergenz und die dafür notwendige Anzahl von Subdomains.

Um eine aussagekräftige Rechenzeit und dennoch ein konvergierendes Ergebnis zu erhalten, wurden $ktmax = 100$ Iterationen berechnet. Die Prozessoranzahl wurde, einem Weak Scaling-Test entsprechend, der Problemgröße ungefähr angepasst. Da die Subdomains wieder exakt durch die Anzahl der Prozessoren in x-Richtung teilbar sein müssen, ergibt sich ein Fehler von bis zu 5%. Die Skalierungstests wurden mit folgenden Problemgrößen ausgeführt:

- $\circ$: Domaingröße $(124 \times 258 \times 32)$
- $+$: Domaingröße $(244 \times 258 \times 32)$

Cluster	Atlantis	Cacau	Cluster150
Prozessor	2x Itanium2, 1.3 GHz	2x Intel Nocona, 3,2 GHz	2x Itaniuam2, 1,6 GHz
Speicher	4 GB	2 GB	4 GB
Netzwerk	Myrinet-2k	Infiniband DDR, 20 Gbs	Myrinet-2k
MPI	MPIch-1.2.4	MPIch-1.2.1	MPIch-1.2

Tabelle 5.4: Konfiguration der Cluster für die Ausführung mit `DD_Filtre2`

Kopplung	Bandbreite	Latenz
Atlantis-Cacau	304 – 373 kB/s	62,5 – 66,4 ms
Cacau-Cluster150	546 – 808 kB/s	27.3 – 29.3 ms
Cluster150-Atlantis	71 – 127 kB/s	83 – 84 ms

Tabelle 5.5: Kommunikationsgeschwindigkeit zwischen den drei Clustern

$\star$: Domaingröße $(484 \times 258 \times 32)$

$\triangledown$: Domaingröße $(960 \times 258 \times 32)$

Abb. 5.14 (links) stellt die Effizienz des Verfahrens für verschiedene Anzahlen von Subdomains dar. Während für den kleinen Fall (markiert durch „o“) keine Verbesserung zu erkennen ist, erreichen die größeren Probleme bei mehreren Domains eine höhere Effizienz bis zu 25%. In Abb. 5.14 (rechts) sind die Absolutzeiten für 1 ($\circ$), 2 ($+$), 3 ($\star$), 4 ($\diamond$) und 5 ($\triangledown$) Subdomains.

Ausführung auf einem interkontinentalen Grid

Um die Stärke des Verfahrens zu demonstrieren, wurde `DD_filtre2` auf einem verteilten, heterogenen MetaComputer mit drei Clustern ausgeführt. Das IA64-Cluster `atlantis.tlc2.uh.edu` an der Universität Houston, das EM64t-Cluster `cacau.hww.de` am HLRS, sowie das IA64-Cluster `cluster150.inm.ras.ru` am Institut für Numerische Mathematik der Russischen Akademie der Wissenschaften in Moskau. Tabelle 5.4 zeigt die Konfiguration zum Zeitpunkt der Tests an.

Nachdem die Systeme einen großen Leistungsunterschied aufweisen, wurde für die (statische) Lastbalancierung von `DD_fitre2` der Benchmark `stream` [92], sowie der Benchmark High-Performance Linpack (HPL) [35] zur Evalierung der Systemperformance ausgeführt. Für die Evaluierung der Kommunikationsgeschwindigkeit wurden die Netzwerkbenchmarks aus Abschnitt 3.2.2 verwendet. Tabelle 5.5 stellt die Ergebnisse für den größten Test mit einer Kombinationen der drei Cluster dar. Für weitere Resultate sei auf [90] verwiesen.

Da der Cluster `cacau` am HLRS zu beiden anderen Systemen wegen seiner geographischen Position die besten Kommunikationsergebnisse lieferte, wurde er in die Konfiguration für die mittlere Subdomain vorgesehen, wie in Abb. 5.15 dargestellt. Das gesamte System hat eine Problemgröße von $(420 \times 96 \times 96)$, mit einem Überlapp von $d = 2$. Damit besitzt `atlantis` 516k Gridpunkte, `cacau` 839k Gridpunkte und `cluster150` 608k Gridpunkte. Die Anzahl der zu kommunizierenden Werte beträgt $d \times N_y \times N_z$

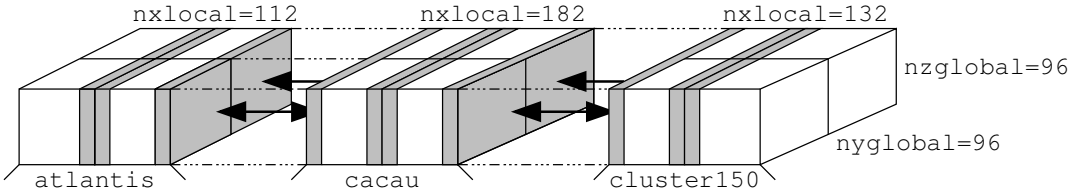

Abbildung 5.15: Aufteilung der Domain auf drei Cluster

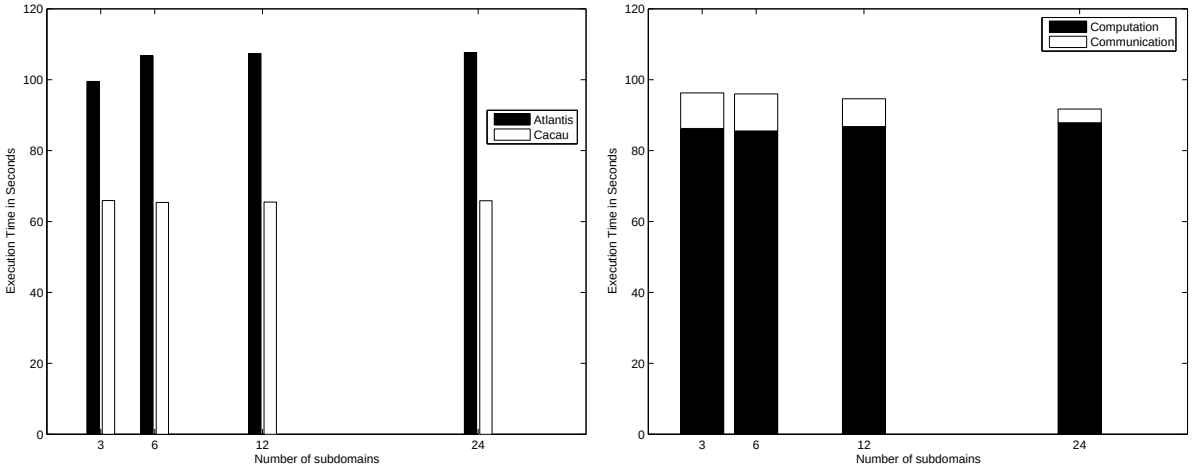

Abbildung 5.16: Laufzeit auf dem MetaComputer im Vergleich zu einem System

Gridpunkte pro Iteration, hier also 18432 doppeltgenaue Fließkommazahlen á 8 Byte bei den Clustern `atlantis` und `cluster150` und 36864 Zahlen bei `cacau`. Im Verlauf der gesamten Berechnung kommunizieren die beiden Cluster am Rand 2,46 MB und `cacau` 4,92 MB.

Um die Performance wieder in Abhängigkeit der Anzahl der Subdomains zu testen, wurden 3 Subdomains, 6, 12 und 24 Subdomains auf die drei Rechner verteilt. In Abb. 5.16 wird die Ausführungszeit gezeigt. Auf der linken Seite ist die Ausführungszeit des großen Problemfalls auf den Systemen `atlantis` und `cacau` gezeigt, während auf der rechten Seite die Gesamtlaufzeit der verteilte Fall, separiert in Rechen- und Kommunikationszeit zwischen den Teilen des MetaComputers gezeigt wird. Es zeigt sich, dass die Zeit für MetaComputing relativ zur Rechenzeit und damit die Gesamtlaufzeit abnimmt, je mehr Subdomains für die Filterung zwischen den Rechnern verwendet werden.

Als wichtigstes Ergebnis zählt aber, dass sich das Gesamtproblem trotz MetaComputing über transatlantische Leitungen schneller als nur auf einem Cluster lösen lässt.

6 Zusammenfassung und Ausblick

Das Konzept des MetaComputing beinhaltet die Kopplung von Resourcen zu einem größeren, virtuellen Computer. Mit der MPI-Implementierung PACX-MPI wurde am HLRS eine umfassende Bibliothek zur Kopplung von Höchstleistungsrechenressourcen geschaffen. Die Mächtigkeit dieser Software wurde in verschiedenen Projekten und Demonstrationen bewiesen, der Gewinn der so genannten HPC-Challenge auf der SuperComputing 2003 zeigt den Erfolg des Konzeptes. Um jedoch Anwendungen verteilt über mehrere Höchstleistungsrechner gewinnbringend zu koppeln, müssen alle Softwareschichten vom Netzwerktransport bis zum Algorithmus der Anwendung berücksichtigt und optimiert werden.

In dieser Arbeit wurde die praktische Umsetzung der Analyse und Optimierungen auf allen Ebenen der Softwareschichten für MetaComputing vorgestellt. Es wird gezeigt, dass die im Vergleich zur internen Rechnerkopplung immer langsamer werdende Kommunikation das größte Hindernis darstellt. Somit werden alle relevanten Softwareschichten von dem Netzwerkprotokoll oberhalb des Kernels, über Optimierungen der Middleware PACX-MPI bis zu den Algorithmen und der darin inherenten Kommunikation der Anwendung betrachtet.

Bezüglich der Netzwerkschicht wird durch parallele und durch nebenläufige Übertragung mit dem UDP-Protokoll die Kommunikation über das externe Netzwerk vor allem für Weitverkehrsnetze beschleunigt. Die Leistung dieses neuen UDP-basierten Protokolls wird in einem lokalen Netz im Vergleich zu Standard-TCP, sowie im Weitverkehrsnetz gezeigt und mit Sabul/UDT verglichen. Die erzielten Messungen mit `ethereal` zeigen, dass Limitierungen der Senderate durch den Overhead bei der Übergabe jedes einzelnen Paketes an den Kernel entstehen. Da das Protokoll portabel als Bibliothek implementiert ist, kann es leicht in neue Projekte eingebunden werden.

Innerhalb der PACX-MPI Bibliothek werden qualitative und funktionale Verbesserungen vorgestellt. Einerseits wird gezeigt, wie sich durch die weitere Optimierung von kollektiven Algorithmen, hier am Beispiel von `PACX_Alltoall` die Kommunikationszeit drastisch reduzieren lässt. Andererseits werden mehrere funktionale Änderungen innerhalb PACX-MPI beschrieben. Hierzu gehört die Möglichkeit, PACX-MPI parallele Anwendungen mit dem Performancetool Paraver zu analysieren, sowie mittels einer neuartigen Implementierung der Peruse-Spezifikation MPI-interne Zustände zu visualisieren. Letzteres ermöglicht einen detaillierten Einblick in das Zeitverhalten der Kommunikation, bspw. die Kommunikation einer Anwendung besser auszubalancieren oder um ineffiziente Implementierung der MPI-Implementierung herauszufinden. Als letztes wird eine MPI-Testsuite für PACX-MPI vorgestellt. Durch den flexiblen Aufbau konnten hiermit nicht nur Fehler in PACX-MPI, sondern auch in etablierten MPI-Implementierungen gefunden werden. Diese Testsuite wird für die ständige Qualitätssicherung im Open

MPI-Projekt weiter entwickelt werden.

Im letzten Kapitel werden die Optimierungen in zwei Anwendungen `DD_filtre2`, sowie `RNAfold` vorgestellt. Hierbei werden bisher Optimierungen implementiert, die so bisher nicht in MPI-Anwendungen verwendet wurden: Caching und Prefetching. Kommunizierte Daten, die eine spatiale und temporale Lokalität vorweisen, werden bei Bedarf blockweise kommuniziert und in einem Cache für die mehrmalige Abfrage zwischengespeichert. Diese Optimierung ermöglicht nicht nur die Ausführung in einem Metacomputer, sondern reduziert auch die Kommunikationszeit auf einem Cluster mit schnellem Infiniband (IB)-Netzwerk. Dennoch ist die Ausführungszeit auf dem Metacomputer bis zu doppelt so lang, wie auf nur einem Cluster. Nichtsdestotrotz lassen sich Anwendungen, wie `fastDNAml`, die je nach Eingabefall lose gekoppelt sind, gewinnbringend auf einem Metacomputer einsetzen. Mit dieser wurde auf der SuperComputing 2003 mit der Genanalyse von insgesamt 67 Tierspezies der HPC-Challenge Wettbewerb für die am weitesten verteilte Anwendung gewonnen.

Wie die Geschwindigkeit von gekoppelten Simulationen mit speziellen Anforderungen an die Hardware, bspw. CFD-Anwendungen angebunden an Strukturmechaniklöser, sich entwickelt, wird sich in zukünftigen Arbeiten zeigen. Die ersten Ergebnisse mit einer gekoppelten Anwendung des Institut für Aero- und Gasdynamik sehen sehr vielversprechend aus.

Die erarbeiteten Resultate mit dem UDP-Protokoll werden in die Open MPI Implementierung einfliessen. Ausserdem werden im Rahmen des Int.EU.Grid Funktionalitäten von PACX-MPI in diese Implementierung eingebaut. Da Open MPI allerdings bisher nicht auf Systeme wie die NEC-SX8 portiert worden ist, wird eine auf dem Vendor-MPI basierende MPI-Bibliothek, wie PACX-MPI immer von wissenschaftlichem Interesse sein. In dem bereits beantragten und genehmigten EU-Projekt DORII wird PACX-MPI somit weiter entwickelt werden.

Anhang

Zeitmessung

Für Zeitmessungen aus Anwendungen und Bibliotheken stehen auf Unix-Systemen verschiedene Funktionen zur Verfügung. Hochgenaue Uhren, auf denen die Systemuhr basieren könnte, sind noch immer ein technisches Problem, da die Taktgeber aufgrund von Temperatur-, Spannungs- oder Lastschwankungen nicht konstant den gleichen Takt vorgeben. Schon in den ersten Unix-Systemen wurde versucht, die Systemuhr mittels kontinuierlicher Netzwerksynchronisation interpolierend einer genaueren Uhr anzupassen. Um zeitabhängige Programme[1] nicht zu stören, wurde die Systemzeit nicht abrupt geändert, sondern inkrementell nachgestellt. Im Folgenden werden die in dieser Arbeit verwendeten Methoden zur Zeitmessung mit Vor- und Nachteilen aufgeführt.

Plattformunabhängige Messung

Mit `gettimeofday` steht eine Posix-kompatible[2] Funktion zur Verfügung, mit der plattformunabhängig auf unterschiedlichen Unix-Systemen gemessen werden kann. Damit liegt der Vorteil von `gettimeofday` auf der Hand. Einerseits bietet diese Methode Mikrosekundenauflösung in der zurückgelieferten Struktur `tv`, andererseits ist die Genauigkeit der Systemuhr nicht definiert. Auch ist der Overhead mit 0,77 μs dieser Routine relativ groß[3]. Dennoch ist dies die meist-verwendete Zeitmessungsfunktion, da unabhängig von der Anzahl der Prozessoren eine Zeit je Task ausgelesen werden kann. Andere Tasks haben keine Auswirkung auf die gemessenen Zeitwerte.
Der folgende Abschnitt zeigt eine Verwendung von `gettimeofday`.

```
static inline int64_t gettimenow (void) {
    struct timeval now_tv;
    gettimeofday (&now_tv, NULL);
    return (now_tv.tv_sec * 1000*1000 + now_tv.tv_usec);
}
...
    start = gettimenow();
    /* Long term computation and IO operations */
    end = gettimenow();
    printf ("Time taken :%f [sec]\n", (end-start)/(1000.0*1000.0));
```

[1] Beispielsweise `make`.
[2] Standard IEEE Std 1003.1-2001
[3] Gemessen auf einem 1,7 GHz Intel Pentium M Prozessor – bei 800000 Iterationen, mit Warmup.

Die Funktion `gettimeofday` ist für Linux als Kernelaufruf in der Systembibliothek `glibc` implementiert. Daraus ergibt sich auch der große Overhead von 0,77 μs.

Prozessortaktgenaue Zeitmessung

Eine weitaus genauere Möglichkeit bietet die Operation `rdtsc` (für „Read Time-Stamp Counter“) auf Intel ia32 (Intel 386, Pentium), x86-64 (Intel Xeon, AMD Opteron), sowie Intel IA64 Architekturen. Dies ist ein Prozessorregister, das bei jedem Prozessortakt inkrementiert wird. Damit sind sehr genaue Zeitmessungen von kleinen Codeabschnitten mit sehr geringem Overhead möglich. Dieses Register wird in die 32-Bit Register `EAX` und `EDX` eingelesen, zusammen bildet dies einen 64-Bit Zähler. Der Overhead beträgt bei dem eingesetzten Laptop mit 1,7 GHz Prozessor zu einem Overhead von nur 0,0276 μs je Aufruf (entspricht bei voller Prozessorfrequenz 47 Taktzyklen), d. h. dieser Aufruf hat ca. $1/30$ des Overheads von `gettimeofday`.

Die folgende kleine Funktion liest für alle obigen Prozessoren den Zähler mittels einer Assembler-Sequenz aus (hier auch für die Sparc-Architektur):

```
static inline uint64_t getrdtsc(void)
{
    uint64_t x;
#if defined(i386)
    __asm__ __volatile__ ("rdtsc" : "=A" (x));
#elif defined(__x86_64__)
    unsigned int _hi, _lo;
    __asm__ __volatile__ ("rdtsc" : "=a" (_lo), "=d" (_hi));
    x = ((unsigned long long int) _hi << 32) | _lo;
#elif defined(__ia64__)
    __asm__ __volatile__ ("mov %0=ar.itc" : "=r" (x) : : "memory");
#elif defined (__sparc__)
    __asm__ __volatile__ ("rd %%tick, %%0" : "=r" (x));
#endif
    return x;
}
```

Der Programmierer sollte beachten, dass für die Zeit bei einem Prozesswechsel (engl. Taskswitch) oder bei Prozessoren mit dynamischer Taktfrequenz falsche Zahlen zurückgeliefert werden. Prozessoren in Laptops sind für den Stromsparmodus optimiert. Hierfür passt der Prozessor dynamisch die Taktfrequenz an, bspw. auf Intel Pentium M Prozessoren von 1,7 GHz auf 600 MHz; weiterhin kann der Prozessor in stromsparende Modi wechseln, indem er sich schlafen legt (ACPI C1–C3 Zustand). In diesen wird der Time-Stamp Counter nicht korrekt hochgezählt. Auf neueren Intel Core-Prozessoren wird der Counter dennoch mit normalem Takt hochgezählt (Prozessorangabe unter Linux in `/proc/cpuinfo`: „`constant_tsc`“).

Das Register kann theoretisch überlaufen (engl. wrap-around), ist aber bei echter 64-Bit Registerbreite unwahrscheinlich[4], außer das Betriebssystem könnte den Registerwert

[4] Dies würde auf 3 Ghz-System erst nach 194 Jahren geschehen.

(absichtlich) herauf setzen. Dies ist aber mit dem Befehlssatz heutiger Prozessoren nicht möglich.

Bezüglich des ersten Punktes muss beachtet werden, dass auch bei Kernelaufrufen, wie dem Lesen von der Festplatte, das ausführende Programm wechseln kann. Hierbei kann der messende Task auf einem anderen Prozessor wieder zum Laufen kommen, auf dem ursprünglich die Referenzmessung gemacht wurde. Die folgende Code-Sequenz wird auf einem Mehrprozessorsystem mit vielen parallelen Anwendungen relativ ungenaue Daten liefern:

```
start = getrdtsc();
infd = open ("infile.dat", O_RDONLY);
size_read = size_count = 1024*1024;
while (size_read > 0) {
    size_read = read (infd, buffer, size_count);
    size_count -= size_read;
    buffer += size_read;
}
stop = getrdtsc();
close (infd);
printf ("Taktzyklen zum Lesen der Input-Daten:%Ld\n", stop - start);
```

Hochgenaue Zeitnehmer

Auf neueren Intel-basierten Systemen gibt es einen Ersatz für den alt-ehrwürdigen Timer des 8254-Chips, der einen Interrupt nach einer programmierbaren Zeit in der Auflösung von 1 ms liefert. Weitere Entwicklungen waren der PIT- und PM-Timer des APIC-Chips sowie der Real-Time Clock (RTC). Diese Systeme mit einem Northbridge-Chip ab i82801 stellen die so genannten High-Precision Event Timer (HPET)[5] zur Verfügung.

Prozessorinterrupts lassen sich mit dieser Technik periodisch wiederkehrend mit einer Frequenz von mindestens 10 MHz programmieren. Diese Technik ist für das in Abschnitt 3.2 erarbeitete UDP-Protokoll von Interesse. Beispielsweise bietet die Northbridge im vom Autor benutzten IBM T41p Laptop drei 64-bit Zähler mit 14.318 MHz. Somit könnten die Pakete periodisch alle 69 μs versandt werden. Es stehen 3 bis 32 frei programmierbare Register mit mindestens 32-Bit zur Verfügung. Ein Zähler inkrementiert die Register, sobald der gespeicherte Wert erreicht wird, wird ein Interrupt generiert und das Betriebssystem kann entsprechend reagieren (und evtl. einer Anwendung eine Benachrichtigung schicken).

Diese Zähler erlauben dem Linux-Kernel mit unterbrechungsfreiem Task-Switching zu arbeiten: sind viele Tasks in der Running-Queue, wird ein Interrupt häufig generiert, um dem Kernel-Scheduler die Möglichkeit zu geben, die Prozessorzeit fair zu verteilen. Läuft kein oder nur ein Task, muss der Scheduler nur selten laufen; die Zeit bis zur nächsten Unterbrechung kann lang sein, d. h. es muss kein (konstant wiederkehrender) Interrupt

[5] Gemeinsam von Intel und Microsoft mit dem Namen Multimedia Timers entwickelt, jetzt unter High-Precision Event Timers auf http://www.intel.com/hardwaredesign/hpetspec_1.pdf verfügbar.

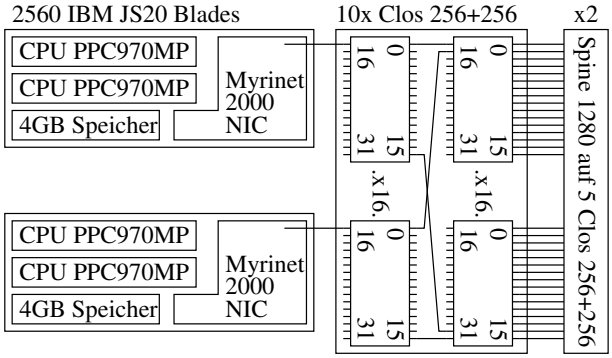

Abbildung 6.1: Das Netzwerk von MareNostrum als CLOS-Netzwerk

generiert werden. Damit kann der Prozessor des Laptops besonders lange (bis zu 80 ms anstatt 20 ms) im stromsparenden C3-Modus verharren.

Zeitmessung in parallelen Anwendungen

Die Bibliotheken für parallele Anwendungen definieren Funktionen, die die einheitliche Messung plattformübergreifend vereinfachen sollen. Bei MPI ist die Funktionen `MPI_Wtime` definiert, die die so genannte Wall-clock Zeit in Sekunden als Fließkommazahl zurück gibt, bezogen auf einen nicht näher definierten Startpunkt. Mit der Funktion `MPI_Wtick` kann die Genauigkeit der Implementierung erfragt werden.

Die parallele Programmierung wird in OpenMP eher durch Direktiven durchgeführt. Unter den wenigen Bibliotheksaufrufen, die im Standard definiert sind, gibt es die zu MPI äquivalenten Funktionen `ompi_get_wtime` und `ompi_get_wtick`, hier in einem Programm:

```
  start_mpi = MPI_Wtime();
#pragma omp parallel
  {
    start_omp = omp_get_wtime();
    /* Very long parallel section */
    stop_omp = omp_get_wtime();
  }
  MPI_Bcast (buffer, sizeof(buffer), MPI_CHAR, 0, MPI_COMM_WORLD);
  stop_mpi = MPI_Wtime();
  printf ("Zeit des gesamten Abschnitts:%lf [s]; Zeit in OpenMP:%lf [s]\n",
          stop_mpi - start_mpi, stop_omp - start_omp);
```

Basiswissen Netzwerktechnik

Um Netzwerktechnik auch für das MetaComputing diskutieren zu können, werden einige Basisbegriffe festgelegt und anhand des Netzwerkes von MareNostrum in Abb. 6.1 dargestellt (siehe [142]):

Netzwerkdurchmesser

Der Netzwerkdurchmesser ist der maximale kürzeste Pfad zwischen zwei beliebigen Knoten des Netzwerks. Dies ist ein Indikator für die höchste zu erwartende Latenz im Gesamtnetzwerk. Somit sollte bei der Auslegung des Netzwerkes der Durchmesser minimiert werden oder durch die Zuweisung von „nahen“ Ressourcen, bspw. in einem Batch-System, der Pfad minimal gehalten werden.

Bei Marenostrum beträgt der maximale kürzeste Pfad fünf, da bis zu fünf Switche auf der Route zwischen zwei Knoten involviert sein könnten.

Bisektionsbreite

Anzahl der zu entfernenden Kommunikationsleitungen, bis die Knoten eines Netzwerkes in zwei Netzwerkhälften getrennt werden. Dies entspricht der Anzahl der (evtl. redundanten) Kommunikationskanäle, die zwischen Knoten die Kommunikation aufrechterhalten können, bzw. parallele Kommunikation erlauben. Dies wird bspw. für hochskalierende Programme wichtig, wenn die Kommunikation zum gleichen Zeitpunkt angestoßen wird und damit Switche und Router zum limitierenden Faktor werden.

Wie Abb. 6.1 deutlich macht, müssten 15 Leitungen an den mittleren Switches oder 15 Leitungen zum Backbone-Switch unterbrochen werden.

Anzahl der Leitungen

Die Anzahl der Leitungen in einem Netzwerk ist proportional zur maximalen, im Netzwerk möglichen Bandbreite. Idealerweise sollten alle Leitungen gleichzeitig Kommunikation erlauben. Dies hängt primär aber mit den Kosten des Netzwerkes zusammen.

Nachdem die NEC-SX8 einen einstufigen Aufbau von Knoten zum Crossbar-Switch (IXS) besitzt, ist dieser Wert eher nebensächlich. Die Anzahl der Leitungen entspricht der Knotenzahl multipliziert mit dem Grad der Knoten, also 576 Leitungen von den Knoten in den IXS.

Für Marenostrum mit seinem dreistufigen Netzwerk enthält das CLOS-Netzwerk [25] mit 2560 Knoten, plus 2560 innerhalb der CLOS 256+256) plus 2560 in die zwei Spine-1280 Switchen insgesamt 7680 Leitungen.

Grad eines Knotens

Der Grad eines Knotens entspricht der Anzahl der physikalischen Kommunikationsports, die ein Knoten besitzt. Diese ist aus Skalierungsgründen meist limitiert, da sonst das Netzwerk zu teuer wird. Moderne Infiniband-NICs besitzen meist zwei Ports, ein Knoten der NEC-SX8 besitzt für jeden der acht Prozessoren ein Glasfaserkabel zum Crossbar-Switch.

Marenostrum ist diesbezüglich stark beschränkt. Jeder der 2560 Knoten von Marenostrum mit seinen vier Cores besitzt nur einen Myri-2k NIC (und einen Ethernet NIC für Administrationszwecke). Damit ist die Bandbreite (sowie durch die Technologie auch die Latenz) stark beschränkt.

Symmetrie

Ein Netz gilt dann als symmetrisch, wenn das Netzwerk von jedem Knoten des Systems aus gesehen gleiche Eigenschaften aufweist. Das heißt, der Grad jedes Knotens, sowie die Art der Kommunikationskanäle ist in jedem Knoten der gleiche. Dies erlaubt Applikationen und Betriebssystemen eine einfachere Programmierung. Ansonsten müssten spezielle Vorkehrungen zur Lastbalancierung bei ungleicher Performance angewendet werden.

Während das Netzwerk der NEC-SX8 in Bezug auf die Knoten als symmetrisch angesehen werden kann, ist es für die einzelnen Prozessoren nicht. Die NEC-SX8 hat als klassischer SMP-Cluster Optimierungen für die Kommunikation innerhalb eines Knotens.
Da Marenostrum ein mehrstufiges CLOS-Netzwerk besitzt, ist er nicht symmetrisch.

Routing

Verschiedene Topologien erfordern unterschiedliche Routingverfahren, in Abhängigkeit von Adressierung, Kommunikationskanälen und Protokollimplementierungen.

Für die NEC-SX8 ist das Routing in Bezug auf die Kommunikation zwischen Knoten einfach, der Crossbar-Switch stellt eine direkte Verbindung zwischen zwei Knoten her.

Das Routing in Marenostrum ist komplizierter, da es durch das CLOS-Netzwerk bspw. innerhalb der ersten Stufe innerhalb des CLOS 256+256 bis zu 15 Routen zwischen zwei Knoten geben kann. Hierbei bietet Myricom zwei verschiedene Kommunikationsbibliotheken `gm` und `mx`. Das Erste stellt die statische Routingtabellen je Prozess beim Starten der Maschine, während `mx` eine dynamische Änderung der Routingtabelle bei Überlast oder Ausfall von Leitungen unterstützt.

Netzwerktechnologie

Die physikalische Umsetzung einer Netzwerktopologie im Sinne der verwendeten Technik, der physikalischen Eigenschaften des Transportmediums, wie Kanalbandbreite und Kabellänge sind entscheidend für die Signalverzögerung, Taktfrequenz und schließlich für die Kommunikationsgeschwindigkeit.

In Abschnitt 2.2 wird die für MetaComputing relevante Netzwerktechnologie eingeführt. Mit dieser Netzwerktechnik werden für größere Systeme ein- oder mehrstufige Netzwerktopologien aufgebaut, wie im folgenden Abschnitt beschrieben. In Abb. 6.2 werden die verschiedenen Topologien graphisch dargestellt.

Übersicht der Netzwerktopologien

Linearer Array und Ring

Ein Beispiel für die Topologien eines Busses (Lineare Kette) ist das alte 10BaseT-Ethernet [97]; die Netzwerkteilnehmer kommunizieren nach dem Carrier Sense Multiple Access with Collision Detection (CSMA/CD)-Verfahren. Ist die Leitung belegt, muss der

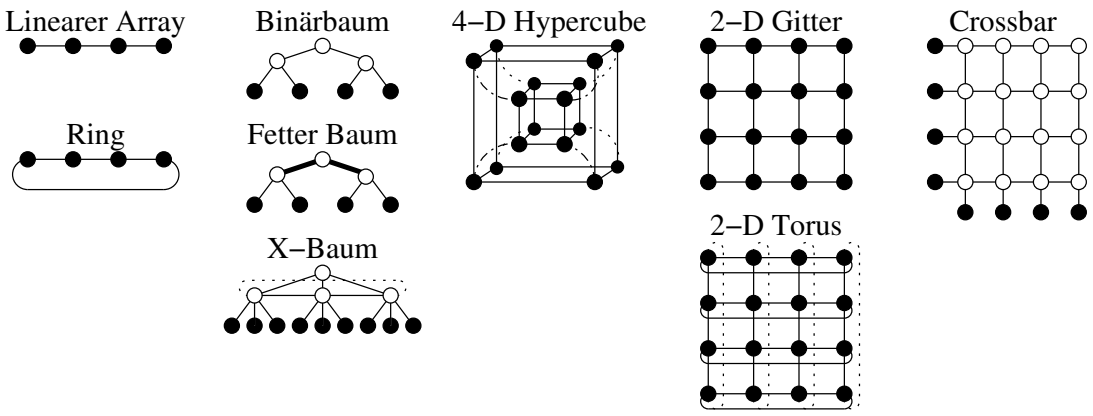

Abbildung 6.2: Vorgestellte Netzwerktopologien

Netzwerkteilnehmer eine zufällige Zeit warten, bis er erneut versuchen kann, die Leitung zu belegen.

Vorteil: Einfach und dadurch günstig.
Nachteil: Hoher Durchmesser, eine Bisektionsbreite von eins.

Binär-, Fette und X-Bäume

Ein Binärbaum besitzt eine Wurzel, mit je zwei Knoten als Söhne. Diese wiederum haben wieder zwei Söhne, dies setzt sich fort bis zu den Blättern. Um das Problem der möglichen Überlast in der Wurzel zu umgehen, werden bei so genannten fetten Bäumen die Bandbreite in den Knoten nahe der Wurzel erhöht. Letzteres hilft aber nicht, die Bisektionsbreite von eins zu erhöhen, d. h. der Ausfall einer Leitung teilt das Netzwerk immer in zwei Hälften. Bei den X-Bäumen sollen Ring-Netze in jeder der mittleren Ebene die höheren Wurzeln entlasten und erhöhen damit die Bisektionsbreite zu Lasten eines um eins erhöhten Grades in jedem Knoten. Auch sind diese Art der Bäume nicht auf zwei Kindknoten limitiert.

Vorteil: Einfache Topologie, einfaches Routing (in Binärbäumen).
Nachteil: Geringe Bisektionsbreite bei Binärbäumen, geringe Bandbreite in der Wurzel.

2-D Gitter und Torus

Diese Art der Gitter erlauben einfache Verbindung mit konstantem Grad in jedem Knoten und einfach gestaltetem Netzwerk. Der Torus reduziert den Netzwerkdurchmesser. 2-D Tori werden wegen ihres einfachen Layouts in der Prozessortechnologie verwendet, um mehrere Cores auf einem Chip zu koppeln.

Vorteil: Einfache Topologie.
Nachteil: Bei statischem Routing häufig Überlast einzelner Kommunikationsleitungen.

Hypercube und Cube-Connected Cycle

Ein Hypercube ist eine Graph mit $N = 2^n$ Knoten mit n Dimensionen. Ein Hypercube n-ter Ordnung kann aus zwei Hypercubes $n-1$-ter Ordnung erstellt werden – damit erhöht sich dann der Grad jedes Knotens von $n-1$ auf n.

In einem Cube-Connected Cycle (CCC) n-ter Dimension, wird jeder Eckpunkt durch n-Knoten Ringe ersetzt. Damit hat ein n-CCC $N = n2^n$ Knoten und damit einen höheren Netzwerkdurchmesser aber auch einen Grad von drei in jeden Knoten.

Vorteil: Für Parallelisierung von Domain Decomposition passende Topologie.
Nachteil: Der Batch-Scheduler und die MPI-Implementierung müssen die Topologie kennen und berücksichtigen.

Crossbar

Ein Crossbar-Switch ist ein Netzwerk, bei dem jeder Input- mit jedem Output-Port durch einen direkten Pfad über einen einzigen Netzwerkknoten verbunden werden kann. Diese Art der Netzwerktopologie hat die geringste Netzwerküberlastung, aber auch die größte Komplexität. Aufgrund der hohen Kosten ist ein Crossbar Switch nicht ohne Weiteres auf über 1000 Knoten skalierbar.

Vorteil: Sehr performant.
Nachteil: Sehr aufwändig und teuer, skaliert nicht auf große Knotenzahlen.

Zusammenfassend eine Übersicht der Eigenschaften der Netzwerktopologien in Tab. 6.1.

Topologie	Anzahl Knoten	Durchmesser	Bisektionsbreite	Anzahl Links	Grad d. Knoten	Symmetrie
Lin. Array	N	$N-1$	1	$N-1$	2	Nein
Ring	N	$N/2$	1	N	2	Ja
Binärbaum	$N = 2^n - 1$	$2(n-1)$	1	$N-1$	3	Nein
2-D Gitter	$N = k^2$	$2(k-1)$	k	$2(N-k)$	4	Nein
2-D Torus	$N = k^2$	$2(k/2)$	$2k$	$2N$	4	Ja
Hypercube	$N = 2^n$	n	2^{n-1}	$nN/2$	n	Ja
CCC	$N = n2^n$	$2n - 1 + n/2$	2^{n-1}	$3N/2$	3	Ja
Crossbar	N proc. N^2 sw.	2	N	N	2	Ja

Tabelle 6.1: Zusammenfassung der Eigenschaften von Netzwerktopologien

Glossar

In diesem Abschnitt werden die verwendeten Abkürzungen erläutert. Für eine möglichst einheitliche Form und für einen höheren Wiedererkennungswert werden nur die englischen Varianten der Abkürzungen ausgeschrieben.

API Application Programming Interface:
Programmierschnittstelle, definiert die Funktionsnamen, -parameter und -typen eines Programms oder einer Bibliothek.

Cache Schneller Zwischenspeicher zur Entkopplung des schnellen Prozessors vom langsamen Hauptspeicher (oder anderen Systemkomponenten). Dies setzt voraus, dass Anwendungen auf nahe liegende Daten (in Bezug auf Zeit und Raum) mehrfach zugreifen.

CAF Co-Array Fortran:
Erweiterung von Fortran zur Angabe der Verteilung von Arrays auf parallelen Rechnern mit verteiltem Speicher (PGAS-Sprache).

ccNUMA cache-coherent Non-Uniform Memory Architecture:
Bezeichnet eine Klasse von Parallelrechnern mit gemeinsamen Speicher, der durch die Hardware auf allen Prozessorcaches konsistent gehalten wird.

CFD Computational Fluid Dynamics:
Berechnung fluider Strömungen, evtl. mit verschiedenen Unterproblemen, wie Überschallströmungen, Viskositäten und anderen Einflüssen.

Cluster Parallelrechner mit verteiltem Speicher, meist gekoppelt durch einen schnellen Interconnect wie IB oder Myrinet.

CORBA Common Object Request Broker:
Ein Standard zur Beschreibung auszutauschender Objekte und Daten in heterogenen Netzwerken.

CPU Central Processing Unit:
Zentrale Steuereinheit, Hauptprozessor.

CRC Cyclic Redundancy Check:
Ein meist in der Netzwerktechnik verwendetes Prüfsummenverfahren über Daten, um Fehler bei der Übertragung zu erkennen.

CSMA/CD Carrier Sense Multiple Access with Collision Detection:
Methode der auf Bussen-basierenden Netzwerktechnik. Ist die Netzwerkleitung belegt, wird für einen zufällig gewählten Zeitraum gewartet.

DGL Differentialgleichung:
Mathematische Gleichung in Abhängigkeit mehrerer physikalischer Grundgrößen, wie Ort, Zeit und Masse.

DGLS Differentialgleichungssystem:
Mehrere, voneinander abhängige DGLen werden als DGLS zusammengefasst.

DMA Direct Memory Access:
Direkter Speicherzugriff einer Hardwarekomponente, bspw. Netzwerkkarte ohne unnötige Kopieroperationen durch die CPU und vor allem ohne Einsprung in das Betriebssystem.

DNA Desoxyribonukleinsäure:
Polymerverkettung aus abwechselnd Zuckermolekül und Phosphorsäure zur Speicherung des genetischen Codes.

FEM Finite Elemente Methode:
Numerisches Verfahren zur näherungsweisen Lösung von partiellen Differentialgleichungen. Der Begriff FEM geht auf [26] zurück.

FPGA Field Programmable Gate Array:
Ein Chip mit mehreren hundert frei programmierbaren Einheiten, so genannte Zellen, welche jeweils vier bis sechs Eingabesignale auswerten und an eine oder zwei Zellen weitergeben.

GGF Global Grid Forum:
Zusammenschluss von Institutionen aus Forschung und Industrie zur Definition von Standards im Grid-Computing.

HPC High-Performance Computing:
Höchstleistungsrechner zur Verarbeitung sehr großer Datenmengen oder Simulation komplexer Vorgänge; Rechner dieser Klasse werden je nach bearbeiteten Anwendungen unterschieden in Capability- oder Capacity-Systeme.

HPF High-Performance Fortran:
Erweiterung der Programmiersprache Fortran zur Angabe der Verteilung der Daten, meist große Matrizen; evtl. notwendige Kommunikation wird dabei automatisch durch den Compiler durchgeführt.

HTX Hypertransport:
Offene Spezifikation von AMD über einen Bus zwischen Prozessoren und anderen Systemkomponenten.

IB Infiniband:
Mittlerweile offene Netzwerkspezifikation des OpenFabrics-Konsortiums für sehr schnelle lokale Netze (siehe Local Area Network).

IMPI Interoperables MPI:
Eine Erweiterung des MPI, um unterschiedliche Implementierungen zu koppeln.

IP Internet Protocol:
Paketspezifikation zum Versand innerhalb des Internets; derzeitige Protokollversionen vier und sechs (IPv4 bzw. IPv6).

JDL Job Description Language:
Eine für Grid-Umgebungen standardisierte Beschreibungssprache für auszuführende Befehle und Anwendungen.

LAN Local Area Network:
Lokales Netzwerk, innerhalb eines Clusters, eines Rechenzentrums oder einer Firma.

MD Molekular-Dynamik:
Gebiet der theoretischen Physik zur Simulation auf molekularer und atomarer Ebene.

MMX Multimedia Extension:
Erweiterung von Intel-Prozessoren zur parallelen Verarbeitung mehrerer Ganzzahlen.

MpCCI Mesh-based Code-Coupling Interface:
Bibliothek zum Koppeln von bestehenden, auf Gitter basierenden parallelen Anwendungen.

MPI Message Passing Interface:
Standard einer API für die Programmierung paralleler Anwendungen mit verteiltem Speicher.

MTU Maximum Transfer Unit:
Größtes Datenfragment in einem IP-Datenpaket für eine bestimmte Route, bspw. 576 (SLIP) oder 1500 Bytes (Ethernet).

NIC Network Interconnect Card:
Allgemeiner Begriff für Netzwerkkarten, wie Ethernet, Infiniband (IB) oder Myrinet.

OpenMP Modell zur parallelen Programmierung für Hardware mit gemeinsamen Speicher, basierend auf Threads.

P2P Peer-to-Peer:
Rechner-Rechner-Verbindung zur verteilten Berechnung oder verteilten Speicherung großer Datenmengen.

PGAS Partitioned Global Address Space:
Meist im Zusammenhang mit Programmiersprachen, ein Programmiermodell, welches den Zugriff auf verteilten Speicher als Zugriff auf gemeinsamen Speicher erscheinen lässt.

POSIX Portable Open System Interface:
Standardisierungsgremium für UNIX-Programmierschnittstellen.

Prefetch Das nebenläufige, auch spekulative Laden von Daten aus einem langsamen Speicher (z. B. Hauptspeicher) in einen schnelleren Speicher, z. B. den Cache (oder Hauptspeicher), um dann zeitnah zugreifen zu können.

Prozess Ein Ausführungspfad zur evtl. parallelen Abarbeitung einer Anwendung; im Unterschied zu Threads haben Prozesse einer Anwendung getrennte Speicher.

PThread Posix Thread:
Standardisierte API zur portablen Programmierung von Threads (Posix 1003.1j).

QoS Quality-of-Service:
Bezeichnet in Bezug auf Netzwerktechnik die Zusicherung einer bestimmten Charakteristik, wie Bandbreite oder Latenz.

RFC Request for Comments:
Offenes Standardisierungsverfahren ohne festes Gremium; verwendet für die Definition von TCP/IP und anderen mit dem Internet verwandten Techniken.

RMI Remote Method Invocation:
Konzept in Java zum Aufruf von Methoden auf anderen Rechner über ein Netzwerk, ähnlich dem RPC.

RNA Ribonucleinsäure:
Polymerverkettung gebildet aus der DNA.

RPC Remote Procedure Calls:
Konzept unter Unix zum Aufruf von Funktionen auf einem anderen Rechner über ein Netzwerk, ähnlich den RMI.

RTT Round Trip Time:
Zeit vom Versenden bis zum Wiederempfang eines Paketes.

SAN Storage Area Network:
Lokales Netzwerk als Interconnect zu großen Speichersubsystemen.

SMP Symmetric Multiprocessing:
Eng-gekoppeltes Mehrprozessorsystem mit gemeinsamen Speicher.

SSE Internet Streaming Extension:
Erweiterung von Intel-Prozessoren zur parallelen Verarbeitung mehrerer Fließkommazahlen.

SSI Single System Image:
Meist Clustersysteme mit einer Erweiterung des Betriebssystems, die dem Benutzer und Anwendungen ein einziges, großes System vortäuscht.

Strong Scaling Skalierung in strenger Form:
Bezeichnet das Verhalten einer Applikation, wenn die Anzahl der rechnenden Prozesse/Threads erhöht wird, die insgesamt zu verarbeitende Problemgröße gleich gehalten werden.

TCP Transport Control Protocol:
Ein auf IP basierendes, byte-orientiertes Netzwerkprotokoll, das Empfang und Reihenfolge der geschriebenen Daten garantiert.

Thread Ein Ausführungspfad zur evtl. parallelen Abarbeitung einer Anwendung; im Unterschied zu Prozessen teilen sich die Threads den gemeinsamen Speicher.

TOE TCP-Offload Engine:
Implementierung teilweiser/vollständiger TCP-Kommunikationsstacks in Hardware.

UDP User Datagram Protocol:
Ein auf IP basierendes, verbindungsloses, paketorientiertes Netzwerkprotokoll, das weder Empfang noch die Reihenfolge der Datenpakete garantiert.

UPC Unified Parallel C:
Neue Erweiterung der Programmiersprache C um Konstrukte zur Angabe der inhärenten Parallelität (PGAS-Sprache).

WAN Wide Area Network:
Nationales, auch interkontinentales Netzwerk im Vergleich zu SAN und LAN.

Weak Scaling Skalierung in schwacher Form:
Bezeichnet das Verhalten einer Applikation, wenn die Anzahl der rechnenden Prozesse/Threads erhöht wird und gleichzeitig die Größe der zu verarbeitenden Daten pro Prozess/Thread äquivalent erhöht wird.

Literaturverzeichnis

[1] Amdahl, Gene M.: *The validity of the single processor approach to achieving large scale computing capabilities.* In: *Proceedings of the Spring Joint Computing Conference (AFIPS)*, Band 30, Seiten 483–485, 1967.

[2] American National Standards Institute: *Parallel Extension for Fortran*, April 1994. Technical Report X3H5/93-SD1-Revision M.

[3] Arnold, Ken und James Gosling: *The Java Programming Language.* Addison-Wesley, 1997.

[4] Avaki: *AVAKI Grid Software: Concepts and Architecture.* Technischer Bericht, Avaki Corporation, März 2002.

[5] Badia, Rosa M., Francesc Escalé, Edgar Gabriel, Judit Gimenez, Rainer Keller, Jesús Labarta und Matthias S. Müller: *Performance Prediction in a Grid Environment.* Lecture Notes in Computer Science (LNCS), 2970:257–264, Februar 2003.

[6] Baldwin, Robert L. und George D. Rose: *Is protein folding hierarchic? II – Local structure and peptide folding.* Tibs 24, Seiten 77–83, Februar 1999.

[7] Barberou, Nicolas, Marc Garbey, Matthias Hess, Michael M. Resch, Tuomo Rossi, Jari Toivanen und Damien Tromeur-Dervout: *Aitken-Schwarz method for efficient metacomputing of elliptic equations.* In: *Proceedings of the* 14th *Domain Decomposition meeting*, Seiten 349–356, Januar 2002.

[8] Bell, Christian, Dan Bonachea, Rajesh Nishtala und Katherine Yelick: *Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap.* In: 20th *International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, April 2006.

[9] Bhoedjang, Raoul A.F., Tim Ruhl und Henri E. Bal: *LFC: A Communication Substrate for Myrinet.* In: *Proceedings of the Fourth Annual Conference of the Advanced School for Computing and Imaging*, Lommel, Belgium, Juni 1998.

[10] Bierbaum, Boris, Carsten Clauss, Thomas Eickermann, Lidia Kirtchakova, Arnold Krechel, Stephan Springstubbe, Oliver Wäldrich und Wolfgang Ziegler: *Reliable Orchestration of Distributed MPI-Applications in a UNICORE-Based Grid with MetaMPICH and MetaScheduling.* In: Mohr, B.,

J. Larsson Träff, J. Worringen und J.J. Dongarra (Herausgeber): *Proceedings of the* 13th *European PVM/MPI Users' Group Meeting*, Band 4192 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 174–183, Bonn, Germany, September 2006. Springer.

[11] Bierbaum, Boris, Carsten Clauss, Martin Pöppe, Stefan Lankes und Thomas Bemmerl: *The New Multidevice Architecture of MetaMPICH in the Context of other Approaches to Grid-enabled MPI.* In: Mohr, B., J. Larsson Träff, J. Worringen und J.J. Dongarra (Herausgeber): *Proceedings of the* 13th *European PVM/MPI Users' Group Meeting*, Band 4192 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 184–193, Bonn, Germany, September 2006. Springer.

[12] Boden, Nanette J., Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic und Wen-King Su: *Myrinet: A Gigabit-per-second Local Area Network.* IEEE Micro, 15(1):29–36, Februar 1995.

[13] Boghosian, Bruce, Peter Coveney, Suchuan Dong, Lucas Finn, Shantenu Jha, George Karniadakis und Nicolas Karonis: *Nektar, SPICE, and Vortonics: Using Federated Grids for Large Scale Scientific Applications.* In: *Proceedings of the Conference on Challenges of Large Applications in Distributed Environments*, Paris, France, Juni 2006.

[14] Borkenhagen, John M., Richard J. Eickemeyer, Ronald N. Kalla und Steven R. Kunkel: *A multithreaded PowerPC processor for commercial servers.* Technischer Bericht, IBM Server Group, 3605 Highway 52N, Rochester, Minnesota 55901, November 2000.

[15] Breckenridge, Arthurine, Lyndon Pierson, Sergiu Sanielevici, Joel Welling, Rainer Keller, Uwe Wössner und Jürgen Schulze: *Distributed, On-Demand, Data-Intensive and Collaborative Simulation Analysis.* Future Generation of Computer Systems (FGCS), 19(6):849–859, 2002.

[16] Brightwell, Ron, Sue Goudy und Keith Underwood: *A Preliminary Analysis of the MPI Queue Characteristics of Several Applications.* In: *Proceedings of the International Conference on Parallel Processing (ICPP)*, Seiten 175–183, Juni 2005.

[17] Brightwell, Ron, William Lawry, Mike Levenhagen, Arthur B. Maccabe und Rolf Riesen: *A performance comparison of myrinet protocol stacks.* In: *Proceedings of the International Conference on High-Performance Clustered Computing*, St. Petersburg, Florida, Oktober 2002.

[18] Brunst, Holger, Manuela Winkler, Wolfgang E. Nagel und Hans-Christian Hoppe: *Performance Optimization for Large Scale Computing: The scalable VAMPIR approach.* In: Alexandrov, V.N. et al. (Herausgeber): *Proceedings of the International Conference on Computational Science (ICCS'01)*,

Band 2074 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 751–760. Springer, Mai 2001.

[19] Bull, Mark: *OpenMP 2.5 and 3.0.* In: *Proceedings of the Workshop on OpenMP Applications and Tools (WOMPAT)*, Houston, TX, USA, Mai 2004. (Invited Talk).

[20] Butenhof, David: *Programming with POSIX Threads.* Addison-Wesley, 1997.

[21] Chan, Fan, Jiannong Cao und Yudong Sun: *High-level abstractions for message-passing parallel programming.* Parallel Computing, 29:1589–1621, November 2003.

[22] Chandra, Rohit, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald und Ramesh Menon: *Parallel Programming in OpenMP.* Morgan Kaufmann Publisher, 2001.

[23] Clark, David: *OpenMP: A parallel standard for the masses.* IEEE Concurrency, 6(1):10–12, Januar 1998.

[24] Clauss, Carsten, Stephan Gsell, Stefan Lankes und Thomas Bemmerl: *A Fair Benchmark Tool for Evaluating the Latent Potential of Heterogeneous Coupled Clusters.* In: *In Proceedings of the 6th International Symposium on Parallel and Distributed Computing*, Seite 38, Hagenberg, Austria, Juli 2007.

[25] Clos, Charles: *A Study of Non-blocking Switching Networks.* Technischer Bericht 2, Bell Corporation, 1953.

[26] Clough, Ray W.: *The finite element method in plane stress analysis.* In: *Proceedings of the* 2nd *Conference on Electronic Computation*, Pittsburgh, Pennsylvania, September 1960.

[27] Combs, Gerald: *Network Data Extraction using Ethereal.* Eingeladener Vortrag auf der Konferenz FOSDEM, Februar 2005.

[28] Committee, IMPI Steering: *IMPI – Interoperable Message-Passing Interface*, 1998. http://impi.nist.gov.

[29] Dabdub, Donald und John H. Steinfeld: *Parallel Computation in Atmospheric Chemical Modeling.* Parallel Computing, 22(1):111–130, Januar 1996.

[30] Davies, Kevin: *The Sequence – Inside the Race for the Human Genome.* Phoenix, 2002.

[31] Dickman, Lloyd: *Beyond Hero Numbers: Factors Affecting Interconnect Performance.* Technischer Bericht, QLogic Corporation, Mai 2007.

[32] Digital: *A Guide to DECthreads.* Internet, 1994.

[33] Doerfler, Douglas w.: *An Analysis of the Pathscale Inc. Infiniband Host Channel Adapter InfiniPath.* Technischer Bericht, Sandia National Laboratories, Albuquerque, NM, USA, August 2005.

[34] Dongarra, Jack: *An Overview of Supercomputers, Clusters and Grids*, Juni 2005. Eingeladener Vortrag am HLRS.

[35] Dongarra, Jack J.: *The Linpack benchmark: An explanation.* In: *Proceedings of the* 1st *International Conference of Supercomputing*, Seiten 456–474, London, UK, 1988. Springer.

[36] Dubois, Paul F.: *Ten Good Practices in Scientific Programming.* Computing in Science & Engineering, 1(1):7–11, Januar 1999.

[37] Dupros, Fabrice, Marc Garbey und Walter E. Fitzgibbon: *A Filtering technique for System of Reaction Diffusion equations.* Int. J. for Numerical Methods in Fluids, 52:1–29, 2006.

[38] Dustdar, Schahram, Harald Gall und Manfred Hauswirth: *Software-Architekturen für verteilte Systeme.* Springer, Mai 2003.

[39] Dykstra, Phil: *Gigabit Ethernet Jumbo Frames.* Technischer Bericht, WareOnEarth Communications, Inc., Dezember 1999. http://sd.wareonearth.com/~phil/jumbo.html.

[40] Eickermann, Thomas, Jörg Henrichs, Michael M. Resch, Robert Stoy und Roland Völpel: *Metacomputing in Gigabit environments: Networks, tools, and applications.* Parallel Computing, 24:1847–1872, 1998.

[41] El-Ghazawi, Tarek, William Carlson, Thomas Sterling und Katherine Yelick: *UPC: Distributed Shared Memory Programming.* John Wiley and Sons, Mai 2005.

[42] Fagg, Graham E., Edgar Gabriel, Zizhong Chen, Thara Angskun, George Bosilca, Antonin Bukovski und Jack J. Dongarra: *Fault Tolerant Communication Library and Applications for High Performance.* In: *Los Alamos Computer Science Institute Symposium*, Santa Fe, NM, Oktober 2003.

[43] Foster, Ian, Jonathan Geisler, William Gropp, Nicholas T. Karonis, Ewing Lusk, George Thiruvathukal und Steven Tuecke: *Wide-Area Implementation of the Message Passing Standard.* Parallel Computing, 24:1735–1749, 1998.

[44] Foster, Ian und Nicholas T. Karonis: *A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems.* In: *Proceedings of SC'98*, Seiten 1–11, Washington, DC, USA, November 1999. IEEE Computer Society.

[45] Foster, Ian und Carl Kesselman: *The Globus Project: A Status Report.* In: *Proceedings of IPPS/SPDP '98 Heterogeneous Computing Workshop*, Seiten 4–18, 1998.

[46] Foster, Ian und Carl Kesselmann: *The Grid: A Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, November 1998.

[47] Foster, Ian, Carl Kesselmann und Steve Tuecke: *The Anatomy of the Grid: Enabling Scalable Virtual Organizations.* International Journal of High Performance Computing Applications, 15(3):200–222, 2001.

[48] Gabriel, Edgar: *Einsatz und Optimierung einer Kommunikationsbibliothek für Metacomputing.* Doktorarbeit, Universität Stuttgart, Juni 2002.

[49] Gabriel, Edgar et al.: *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation.* In: Kranzlmüller, D., P. Kacsuk und J.J. Dongarra (Herausgeber): *Proceedings of the* 11th *European PVM/MPI Users' Group Meeting*, Band 3241 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 97–104, Budapest, Hungary, September 2004. Springer.

[50] Gabriel, Edgar, Rainer Keller, Peggy Lindner, Matthias S. Müller und Michael M. Resch: *Software Development in the Grid: The DAMIEN toolset.* Lecture Notes in Computer Science (LNCS), 2659:235–244, 2003.

[51] Gabriel, Edgar, Michael M. Resch, Thomas Beisel und Rainer Keller: *Distributed Computing in a Heterogeneous Computing Environment.* In: Bubak, M., J.J. Dongarra und J. Wasniewski (Herausgeber): *Proceedings of the* 5th *European PVM/MPI Users' Group Meeting*, Band 1 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 180–187. Springer, September 1998.

[52] Garbey, Marc, Rainer Keller und Michael M. Resch: *Toward a Scalable Algorithm for Distributed Computing of Air-Quality Problems.* In: Dongarra, J.J., Domenico Laforenza und Salvatore Orlando (Herausgeber): *Proceedings of the* 10th *European PVM/MPI Users' Group Meeting*, Band 2840 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 667–671, Venice, Italy, September 2003. Springer.

[53] Göddeke, Dominik, Robert Strzodka und Stefan Turek: *Accelerating Double Precision FEM Simulations with GPUs.* In: Hülsemann, F., M. Kowarschik und U. Rüde (Herausgeber): *Frontiers in Simulation*, Seiten 139–144, September 2005.

[54] Golebiewski, Maciej, Hubert Ritzdorf, Jesper L. Träff und Falk Zimmermann: *The MPI/SX Implementation of MPI for NEC's SX-6 and other NEC Platforms.* NEC Research & Development, 44(1):69–74, 2003.

[55] Gould, Harvey, Jan Tobochnik und Wolfgang Christian: *An Introduction to Computer Simulation Methods*. Addison-Wesley, 3. Auflage, 2006.

[56] Graefen, Rainer: *Theoriebildung bei Data-Grids*. project 57 – Journal für Business Computing und Technologie, special 01/05:29–30, 2005.

[57] Graham, Rich L., Sung-Eun Choi, David J. Daniel, Nehal N. Desai, Ronald G. Minnich, Craig E. Rasmussen, L. Dean Risinger und Mitchel W. Sukalksi: *A network-failure-tolerant message-passing system for terascale clusters*. International Journal of Parallel Programming, 31(4):285–303, August 2003.

[58] Gropp, William, Ewin Lusk, Nathan Doss und Anthony Skjellum: *A high-performance, portable implementation of the MPI message passing interface standard*. Parallel Computing, 22(6):789–828, September 1996.

[59] Gu, Yunhong und Robert L. Grossman: *SABUL: A Transport Protocol for Grid Computing*. Grid Computing, 1:377–386, 2004.

[60] Hackenberg, Matthias G., Peter Post, René Redler und Barbara Steckel: *MpCCI, Multidisciplinary Applications, and Multigrid*. In: *Proceedings in European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS'00)*, CIMNE, Barcelona, Spain, September 2000.

[61] Haney, Scott: *Is C++ Fast Enough For Scientific Computing?* Computers in Physics, 8(6):690–694, November 1994.

[62] Haring, Bruce: *MP3, die digitale Revolution in der Musikindustrie*. Orange Press, 2002.

[63] He, Eric, Jason Leigh, Oliver Yu und Thomas A. DeFanti: *Reliable Blast UDP: Predictable High Performance Bulk Data Transfer*. In: *Proceedings of IEEE Cluster Computing*, Chicago, IL, USA, September 2002.

[64] He, Liyan, Ryszard Kierzek, John SantaLucia, Amy E. Walter und Douglas H. Turner: *Nearest-Neighbor Parameters for GU Mismatches*. Biochemistry, 30:11124–11132, 1991.

[65] Hoeflinger, Jay P.: *Extending OpenMP to Clusters*. Technischer Bericht, Intel, 2005.

[66] Hofacker, Ivo L., Walter Fontana, Sebastian L. Bonhoeffer, Manfred Tacker und Peter Schuster: *Vienna RNA Package*. Internet, Oktober 2002. http://www.tbi.univie.ac.at/~ivo/RNA.

[67] IEEE: *Posix Threads: Posix 1003.1c-Standard*, 1995.

[68] Imamura, Toshiyuki, Yuichi Tsujita, Hiroshi Koide und Hiroshi Takemiya: *An Architecture of Stampi: MPI Library on a Cluster of Parallel Computers*. In: Dongarra, J.J., P. Kacsuk und N. Podhorszki (Herausgeber): *Proceedings of the 7th European PVM/MPI Users' Group Meeting*, Band 1908 der Reihe *Lecture Notes In Computer Science (LNCS)*, Seiten 200–207. Springer, September 2000.

[69] Intel Corporation: *Cluster OpenMP*, Juni 2007. http://www.intel.com/cd/software/products/asmo-na/eng/329023.htm.

[70] Jaeger, John A., Douglas H. Turner und Michael Zuker: *Improved predictions of secondary structures for RNA*. PNAS, 86:7706–7710, Oktober 1989.

[71] Jia, Bin: *Process Cooperation in Multiple Message Broadcast*. In: Capallo, F., T. Herault und J.J. Dongarra (Herausgeber): *Proceedings of the 14th European PVM/MPI Users' Group Meeting*, Band 4757 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 27–35. Springer, September 2007.

[72] Jones, Mark T. und Paul E. Plassmann: *BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems*. Technischer Bericht, Argonne National Laboratory Report ANL-95/48, 1997.

[73] Jones, Terry und et al.: *MPI Peruse – An MPI Extension for Revealing Unexposed Implementation Information*. Internet, Mai 2006. http://www.mpi-peruse.org.

[74] Karonis, Nicholas T., Brian Toonen und Ian Foster: *MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface*. Journal of Parallel and Distributed Computing, 63(5):551–563, Mai 2003.

[75] Keller, Rainer, Michael Bane, Mike Pettipher, Michael Schanz und Paul Levi: *Analysis and parallelisation of a sequential Fortran 90 code based on OpenMP*. Studienarbeit, Universität Stuttgart, Fakultät Informatik, Oktober 1999.

[76] Keller, Rainer, George Bosilca, Graham Fagg, Michael M. Resch und Jack J. Dongarra: *Implementation and Usage of the PERUSE-Interface in Open MPI*. In: Mohr, B., J. Larsson Träff, J. Worringen und J.J. Dongarra (Herausgeber): *Proceedings of the 13th European PVM/MPI Users' Group Meeting*, Band 4192 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 347–355, Bonn, Germany, September 2006. Springer.

[77] Keller, Rainer, Shiqing Fan und Michael Resch: *Memory debugging of MPI-parallel Applications in Open MPI*. In: Joubert, G.R., C. Bischof, F. Peters, T. Lippert, M. Bücker, P. Gibbon und B. Mohr (Herausgeber): *Proceedings of ParCo'07*, Band 38 der Reihe *NIC series*, Seiten 517–527, Jülich, Germany, September 2007.

[78] Keller, Rainer, Edgar Gabriel, Bettina Krammer, Matthias S. Müller und Michael M. Resch: *Towards efficient execution of MPI applications on the Grid: Porting and Optimization issues.* Journal of Grid Computing, 1(2):133–149, Juni 2003.

[79] Keller, Rainer, Edgar Gabriel und Roland Rühle: *Leichtgewichtige Prozesse zur Beschleunigung der Kommunikation in einer Metacomputing Bibliothek.* Diplomarbeit, Universität Stuttgart, Rechenzentrum der Universität Stuttgart, Oktober 2001.

[80] Keller, Rainer, Bettina Krammer, Matthias S. Müller, Edgar Gabriel und Michael M. Resch: *MPI-Development Tools and Applications for the Grid.* In: *Proceedings of the Workshop on Grid Applications and Programming Tools, at Global Grid Forum (GGF8)*, Seattle, WA, USA, Juni 2003.

[81] Keller, Rainer und Markus Liebing: *Using PACX-MPI in MetaComputing applications.* In: *Proceedings of the* 18th *Symposium Simulationstechnique*, Erlangen, Germany, September 2005.

[82] Keller, Rainer, Matthias S. Müller und Michael M. Resch: *HPC tools and methods applied to an RNA folding application.* In: *Proceedings of ParCo'03*, Dresden, September 2003.

[83] Keller, Rainer und Michael Resch: *Testing the correctness of MPI implementations.* In: *Proceedings of the* 5th *Int. Symp. on Parallel and Distributed Computing (ISPDC)*, Seiten 291–295, Timisoara, Romania, Juli 2006.

[84] Kielmann, Thilo, Henri E. Bal, S. Gorlatch, K. Verstoep und Rutger F. H. Hofman: *Network Performance-aware Collective Communication for Clustered Wide Area Systems.* Parallel Computing, 27:1431–1456, 2001.

[85] Kielmann, Thilo, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat und Rutger A. F. Bhoedjang: *MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems.* In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Ppopp'99)*, Seiten 131–140. ACM, Mai 1999.

[86] Labarta, Jesús, Germán Rodríguez und Rosa M. Badia: *An Evaluation of MareNostrum Performance.* Technischer Bericht, BSC, 2006.

[87] Legout, Arnaud, Guillaume Urvoy-Keller und Pietro Michiardi: *Rarest First and Choke Algorithms Are Enough.* In: *Proceedings of the* 6th *ACM SIGCOMM on Internet Measurement (IMC'06)*, Seiten 203–216, Rio de Janeiro, Brazil, Oktober 2006. ACM Press.

[88] LeVeque, Randall J.: *Numerical methods for conservation laws.* Birkhäuser Verlag, 2. Auflage, 1992.

[89] Lindner, Peggy: *Management von verteilten ingenieurwissenschaftlichen Anwendungen in heterogenen Grid Umgebungen*. Doktorarbeit, Universität Stuttgart, Juni 2007.

[90] Ltaief, Hatem, Rainer Keller, Marc Garbey und Michael Resch: *A Grid Solver for Reaction-Convection-Diffusion Operators*. Parallel Computing, to be submitted, 2007.

[91] Mathews, David H., Jeffrey Sabina, Michael Zuker und Douglas H. Turner: *Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure*. Journal of Molecular Biology, 288(5):911–940, Mai 1999.

[92] McCalpin, John D.: *Memory Bandwidth and Machine Balance in Current High Performance Computers*. IEEE TCCA, Dezember 1995. http://www.cs.virginia.edu/stream.

[93] Meiss, Mark R.: *TSUNAMI: A High-Speed Rate-Controlled Protocol for File Transfer*. Technischer Bericht, Indiana University, Dezember 2002.

[94] Message Passing Interface Forum: *MPI: A Message Passing Interface Standard*, Juni 1995. http://www.mpi-forum.org.

[95] Message Passing Interface Forum: *MPI-2: Extensions to the Message-Passing Interface*, Juli 1997. http://www.mpi-forum.org.

[96] Message Passing Interface Forum: *MPI-2 Journal of Development*, Juli 1997. http://www.mpi-forum.org.

[97] Metcalfe, Robert M. und David R. Boggs: *Ethernet: Distributed Packet Switching for Local Computer Networks*. Communications of the ACM, 19(5):395–405, Juli 1976.

[98] Mo, Jeonghoon, Richard J. La, Venkat Anantharam und Jean Walrand: *Analysis and Comparison of TCP Reno and Vegas*. In: *INFOCOM '99. 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, Band 3, Seiten 1556–1563, März 1999.

[99] Moore, Gordon E.: *Semiconductor Technology Reaches Middle Age*. IEEE Computer, 24(9):105–106, September 1991.

[100] Myrinet Corporation: *Myrinet Networks Overview*, Juni 2007. http://www.myri.com/myrinet/overview/.

[101] *NEC On-Chip Multithreading Project*. Internet, August 2001. http://www.labs.nec.co.jp/MP98.

[102] Nieplocha, Jaroslaw, Robert J. Harrison und Richard J. Littlefield: *Global Arrays: A Portable „Shared-Memory“ Programming Model for Distributed Memory Computers*. In: IEEE (Herausgeber): *Proceedings of SC'94*, Supercomputing, Seiten 340–349, 1994.

[103] Numrich, Robert W. und John Reid: *Co-Array Fortran for Parallel Programming*. ACM Fortran Forum, 17(2):1–31, 1998.

[104] *Open Grid Forum – Overview*. Internet, Juni 2007.

[105] OpenMP committee: *OpenMP Fortran Application Program Interface*, Version 1.0 Auflage, Oktober 1997. http://www.openmp.org.

[106] OpenMP committee: *OpenMP Fortran Application Program Interface*, Version 2.0 Auflage, November 2000. http://www.openmp.org.

[107] Peritz, Adam E., Ryszard Kierzek, Naoki Sugimoto und Douglas H. Turner: *Thermodynamic Study of Internal Loops in Oligoribonucleotides*. Biochemistry, 30:6428–6435, 1991.

[108] Pješivac-Grbović, Jelena, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel und Jack J. Dongarra: *Performance analysis of MPI collective operations*. Cluster Computing, 10(2):127–143, Juni 2007.

[109] Polanski, Andrzej und Marek Kimmel: *Bioinformatics*. Springer, 2007.

[110] Pöppe, Christoph: *Aufbruch in neue Dimensionen*. Spektrum der Wissenschaft – Dossier, 2:25–27, 2007.

[111] Resch, Michael M.: *Metacomputing in Simulationsanwendungen*. Doktorarbeit, Universität Stuttgart, Juli 2001.

[112] Resch, Michael M., Thomas Beisel, Holger Berger, Katrin Bidmon, Edgar Gabriel, Rainer Keller und Dirk Rantzau: *Clustering T3Es for Metacomputing Applications*. In: *Cray User Group Conference*, 1998.

[113] Resch, Michael M., Dirk Rantzau, Holger Berger, Katrin Bidmon, Rainer Keller und Edgar Gabriel: *A Metacomputing Environment for Computational Fluid Dynamics*. Seiten 135–144. North Holland, 1999.

[114] Richardson, Harvey: *High Performance Fortran: history, overview and current developments*. Technischer Bericht TMC-261, Thinking Machines Corporation, April 1996.

[115] Romberg, Mathilde: *The UNICORE Grid Infrastructure*. Special Issue on Grid Computing of Scientifc Programming Journal, 10:149–157, 2002.

[116] Roth, Johannes, Jörg Stadler, Marco Brunelli, Dietmar Bunz und Franz Gähler et al: *IMD - A Massively Parallel Molecular Dynamics Package for Classical Simulations in Condensed Matter Physics*. In: Krause, Egon und Wolfgang Jäger (Herausgeber): *High Performance Computing in Science and Engineering '99*, Seiten 72–81. Springer, 2000.

[117] Sanders, Peter und Jop F. Sibeyn: *A Bandwidth Latency Tradeoff for Broadcast and Reduction*. In: Alexandrov, V. und J.J. Dongarra (Herausgeber): *Proceedings of the* 6th *EuroPar Conference*, Band 1900 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 918–926, Munich, Germany, August 2000. Springer.

[118] Sanders, Peter, Jochen Speck und Jesper Larsson Träff: *Full Bandwidth Broadcast, Reduction and Scan with Only Two Trees*. In: Capallo, F., T. Herault und J.J. Dongarra (Herausgeber): *Proceedings of the* 14th *European PVM/MPI Users' Group Meeting*, Band 4757 der Reihe *Lecture Notes In Computer Science (LNCS)*, Seiten 17–26, Paris, France, September 2007. Springer.

[119] *Sandpile.org Homepage*, August 2007. http://www.sandpile.org.

[120] Sandu, Adrian, Jan G. Verwer, Maarten van Loon, Gregory R. Carmichael, Florian A. Potra, Donald Dabdub und John H. Seinfeld: *Benchmarking Stiff ODE Solvers for Atmospheric Chemistry Problems I: Implicit versus Explicit*. CWI Report NM-R 9603/1996, Department of Numerical Mathematics, CWI, Amsterdam, Netherlands, Januar 27 1996.

[121] Seward, Julian und Nicholas Nethercote: *Using Valgrind to detect undefined value errors with bit-precision*. In: *Proceedings of the USENIX'05 Annual Technical Conference*, Anaheim, CA, USA, April 2005.

[122] Seyffert, Wilhelm (Herausgeber): *Lehrbuch der Genetik*. Spektrum Akademischer Verlag, 2. Auflage, 2003.

[123] Sheng, Feng: *Erweiterung einer MPI-Testsuite um MPI-2 Funktionalitäten*. Diplomarbeit, Universität Stuttgart, Höchstleistungsrechenzentrum Stuttgart, Februar 2007.

[124] Shoch, John F. und Jon A. Hupp: *Measured performance of an Ethernet local network*. Communications of the ACM, 23(12):711–721, Dezember 1980.

[125] Siegel, Jon: *CORBA 3 – Fundamentals and Programming*. John Wiley & Sons, 2. Auflage, 2000.

[126] Smarr, Larry und Charles E. Catlett: *Metacomputing*. Communications of the ACM, 35(6):44–52, Juni 1992.

[127] Snell, Quinn O., Armin R. Mikler und John L. Gustafson: *NetPIPE: A Network Protocol Independent Performance Evaluator*. In: *IASTED International Conference on Intelligent Information Management and Systems*, Juni 1996.

[128] Spingola, Marc, Leslie Grate, David Haussler und Manuel Ares: *Genome-wide bioinformatic and molecular analysis of introns in Saccharomyces cerevisiae*. RNA, 5:221–234, 1999.

[129] Squyres, Jeffrey M. und Andrew Lumsdaine: *A Component Architecture for LAM/MPI*. In: Dongarra, J.J., Domenico Laforenza und Salvatore Orlando (Herausgeber): *Proceedings of the* 10th *European PVM/MPI Users' Group Meeting*, Nummer 2840 in *Lecture Notes in Computer Science (LNCS)*, Seiten 379–387, Venice, Italy, September 2003. Springer.

[130] Squyres, Jeffrey M., Andrew Lumsdaine, William L. George, John G. Hagedorn und Judith E. Devaney: *The Interoperable Message Passing Interface (IMPI) Extensions to LAM/MPI*. In: *Proceedings of the MPI Developer's Conference (MPIDC'2000)*, Ithica, NY, USA, März 2000.

[131] Sterling, Thomas: *Supercomputer – eine Zwischenbilanz*. Spektrum der Wissenschaft – Dossier, 2:20–25, 2007.

[132] Stevens, W. Richard: *TCP/IP Illustrated I. The Protocols*, Band 1. Addison-Wesley, 2004.

[133] Stevens, W. Richard, Bill Fenner und Andrew M. Rudoff: *UNIX Network Programming: Networking APIs: Sockets and XTI*, Band 1. Prentice-Hall, 2. Auflage, 1997.

[134] Stewart, Craig, Rainer Keller, Richard Repasky, Matthias Hess, David Heart, Matthias Müller, Ray Sheppard, Uwe Wössner, Martin Aumüller, Hulan Li, Donald K. Berry und John Colbourne: *A Global Grid for Analysis of Arthropod Evolution*. In: *Proceedings of the* 5th *IEEE/ACM International Workshop on Grid Computing (Grid'04)*, Band 0, Seiten 328–337, November 2004.

[135] Stone, Jonathan und Craig Partridge: *When the CRC and TCP Check-Sum Disagree*. In: *SIGCOMM '00: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Seiten 309–319, Stockholm, Sweden, 2000.

[136] Todorović, Milan: *Comparative study of the end-to-end compliant TCP protocols for wireless networks*. Master Thesis, Texas Tech University, Dezember 2005.

[137] Todorović, Milan und Noé Lopez-Benitez: *Efficiency Study of TCP Protocols in Infrastructured Wireless Networks*. In: *Proceedings of the International conference on Networking and Services (ICNS)*, Seite 103, Washington, DC, USA, 2006.

[138] Träff, Jesper Larsson, William Gropp und Rajeev Thakur: *Self-consistent MPI Performance Requirements.* In: Capallo, F., T. Herault und J.J. Dongarra (Herausgeber): *Proceedings of the* 14th *European PVM/MPI Users' Group Meeting*, Band 4757 der Reihe *Lecture Notes In Computer Science (LNCS)*, Seiten 36–45, Paris, France, September 2007. Springer.

[139] Turley, James L.: *Advanced 80386 Programming Techniques.* Osborne McGraw-Hill, 1988.

[140] Turner, Douglas H., Naoki Sugimoto und Susan M. Freier: *RNA Structure Prediction.* Ann. Rev. Biophys., Biophys. Chem., 17:167–192, 1988.

[141] Verwer, Jan G., Willem H. Hundsdorfer und Joke G. Blom: *Numerical Time Integration for Air Pollution Models.* Technischer Bericht, November 27 1998.

[142] Walrand, Jean, Kallol Bagchi und George W. Zobrist (Herausgeber): *Wormhole Data Routing in Multiprocessors: Networks and Algorithms*, Kapitel 1. Gordon and Breach Science Publishers, März 1999.

[143] Walter, Amy E., Douglas H. Turner, James Kim, Matthew H. Lyttle, Peter Müller, David H. Mathews und Michael Zuker: *Coaxial stacking of helixes enhances binding of oligoribonucleotides and improves predictions of RNA folding.* PNAS, 91:9218–9222, 1994.

[144] Watson, James D. und Francis H. C. Crick: *A Structure for Deoxyribonucleic Acid.* Nature, 171:737–738, 1953.

[145] Wenisch, Petra, Oliver Wenisch und Ernst Rank: *Harnessing High-Performance Computers for Computational Steering.* In: *Proceedings of the* 12th *European PVM/MPI Users' Group Meeting*, Band 3666 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 536–543, Sorrento, Italy, September 2005. Springer.

[146] Werthimer, Dan, Jeff Cobb, Matt Lebofsky, David Anderson und Eric Korpela: *SETI@HOME—massively distributed computing for SETI.* Comput. Sci. Eng., 3(1):78–83, Januar 2001.

[147] Wössner, Uwe, Jürgen P. Schulze, Steffen P. Walz und Uli Lang: *Evaluation of a Collaborative Volume Rendering Application in a Distributed Virtual Environment.* In: *Proceedings of the* 8th *Eurographics Workshop on Virtual Environments (EGVE)*, Seiten 113–122. ACM, Mai 2002.

[148] Zaki, Omer, Ewing Lusk, William Gropp und Deborah Swider: *Toward Scalable Performance Visualization with Jumpshot.* High Performance Computing Applications, 13(2):277–288, September 1999.

[149] Zhuang, Yu und Xian-He Sun: *Stable, globally non-iterative, non-overlapping domain decomposition parallel solvers for parabolic problems.* In: *Proceedings of SC'01*, Seite 40, Denver, USA, 2001.

[150] Zuker, Michael und Patrick Stiegler: *Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information.* Nucleic Acids Research, 9:133–148, 1981.

Index

Curriculum Vitae

Persönliche Daten

Rainer Keller
Kernerweg 22
73773 Aichwald

Tel.: (0711) 389 0700
E-Mail: keller@hlrs.de

Geb. am 06. 12. 1974 in Konstanz
Ledig, deutsch

Schulische Ausbildung

1981 – 1985	Grundschule Konstanz/Dettingen
1985 – 1992	Alexander-von-Humboldt Gymnasium, Konstanz
1992 – 1993	Senior-Year, Bradwell-Institute, Hinesville, GA, USA
1993 – 1995	Alexander-von-Humboldt Gymnasium, Konstanz (Leistungskurse Mathematik und Englisch)

Grundwehrdienst

07/1995–05/1996	Kommandeursfahrer, Jägerbataillon Pfullendorf

Studium

09/1996–11/1999	Studium der Informatik an der Universität Stuttgart
07/1999–10/1999	Studienarbeit am Rechenzentrum CSAR in Manchester, UK
11/1999–10/2001	Hauptdiplom an der Universität Stuttgart

Beruflicher Werdegang

05/1996–04/1997	Administrator im Rechenzentrum Debis / Siemens AG, Konstan
10/1997–10/2001	Wissenschaftliche Hilfskraft am Rechenzentrum der Universität Stuttgart (RUS), bzw. Höchstleistungsrechenzentrum (HLRS)
10/2001–heute	Wissenschaftlicher Angestellter am HLRS
10/2005–heute	Leiter der Arbeitsgruppe „Application, Models and Tools“

Stuttgart, 29. September 2007