Forschungszentrum Jülich

Zentralinstitut für Angewandte Mathematik

Programmierkonzepte und Lastverteilungsstrategien für heterogene Parallelrechnersysteme

Jörg Henrichs

Jül-3676

Programmierkonzepte und Lastverteilungsstrategien für heterogene Parallelrechnersysteme

Metacomputing, im Sinne von heterogenem Supercomputing, ermöglicht es, große und rechenaufwendige Programme bearbeiten zu lassen, da durch die Kopplung von Rechnern ein insgesamt deutlich größerer Hauptspeicher und eine größere Gesamtrechenkapazität zur Verfügung steht als bei einem einzelnen Rechner. In der Praxis aber erschweren die vergleichsweise geringe Bandbreite und hohe Latenz des externen Netzwerks, das die unterschiedlichen Rechner miteinander verbindet, die Vorteile von Metacomputing zu nutzen. Es ist oftmals eine umfangreiche Umstrukturierung einer Anwendung notwendig, um den Einsatz auf einem Metacomputer sinnvoll werden zu lassen. Faktoren wie Lastausgleich und Kommunikationsoptimierung müssen dabei berücksichtigt werden.

In dieser Arbeit wird ein Programmiermodell vorgestellt, das automatisch eine Optimierung der externen Kommunikation durch Zusammenfassung externer Nachrichten und Überlappung von Kommunikation und Berechnung ermöglicht. Weiterhin wird die unterschiedliche Rechenkapazität der genutzten Rechner in einem statischen Lastausgleichsverfahren berücksichtigt. Schließlich wird durch ein dynamisches Lastausgleichsverfahren die Last auf die einzelnen Rechner zur Laufzeit umverteilt, wobei auch die unterschiedliche Rechenkapazität der Maschinen berücksichtigt wird. Anhand einer prototypischen Implementierung werden die Vorteile, die sich durch den Einsatz des Modells ergeben, nachgewiesen.

Programming Concepts and Load-Balancing Strategies for Heterogeneous Parallel Computing Systems

With metacomputing, in the sense of heterogeneous supercomputing, large computational problems can be solved, utilizing main memory and computational power of several coupled computers. However, because of the comparatively low bandwidth and high latency of external networks, it is quite difficult to realize these theoretical advantages. Experience shows that extensive restructuring of applications is necessary to make reasonable use of a metacomputer, as factors like load balancing and optimization of the communication have to be considered.

In this thesis a programming model is developed which automatically optimizes external communication by combining messages and by overlapping computation and communication. A static load balancing scheme allows for different computational powers of the used machines. Furthermore, a dynamic load balancing module redistributes the load during runtime, considering the different speeds of the machines. A prototypical implementation shows the advantages which result from using the model in typical practical applications.

Danksagung

Die vorliegende Arbeit ist am Zentralinstitut für Angewandte Mathematik (ZAM) des Forschungszentrums Jülich entstanden. Ich möchte dem Institutsleiter Herrn Prof. Dr. rer.nat. F. Hoßfeld, Inhaber des Lehrstuhls für Technische Informatik und Computerwissenschaften der Rheinisch-Westfälischen Technischen Hochschule Aachen, herzlich für sein Interesse an der Arbeit und die mir gewährte Unterstützung danken. Herrn Prof. Dr. rer. nat. D. Haupt, ehemaliger Inhaber des Lehrstuhls für Betriebssysteme der Rheinisch-Westfälischen Technischen Hochschule Aachen, danke ich für die Übernahme des Korreferates und fruchtbare Diskussionen.

Mein besonderer Dank gilt Herrn Prof. Dr. W. E. Nagel, Leiter des Zentrums für Hochleistungsrechnen der Universität Dresden. Seine Bemerkungen und wertvollen Anregungen haben die Arbeit positiv beeinflußt.

Für konstruktive Diskussionen möchte ich mich bei den Herren Dr. M. Gerndt, Dr. R. Berrendorf und Dr. M. Bücker bedanken. Weiterhin gilt mein Dank den Mitarbeitern des Zentralinstituts für Angewandte Mathematik für die sehr gute Arbeitsatmosphäre und Hilfsbereitschaft. Für die technische Unterstützung bei der Nutzung der verschiedenen Maschinen und Netzwerke bin ich insbesondere Herrn U. Detert (ZAM), R. Niederberger (ZAM), R. Vogelsang (SGI) und H. Bast (Intel) dankbar.

Meinen Eltern danke ich, daß sie mir das Studium ermöglicht haben.

Mein ganz besonderer Dank gilt aber schließlich meiner Freundin Stefanie Kethers für ihr Verständnis und die Unterstützung während meiner Promotionszeit.

Inhaltsverzeichnis

1 Einleitung

Während früher wissenschaftliche Erkenntnis durch die beiden Disziplinen Theorie und Experiment gestützt wurde, hat sich mittlerweile die Simulation zu einer eigenständigen Kategorie entwickelt. Unter der Bezeichnung „Wissenschaftliches Rechnen“, eine etwas unzulängliche Übersetzung des Begriffes *Computational Science & Engineering* [66, 67], bildet sie das dritte Standbein wissenschaftlicher Erkenntnis. Simulationen werden dabei nicht nur eingesetzt, um Theorien zu verifizieren und Ergebnisse von Experimenten zu erklären, sondern insbesondere in Situationen, in denen ein Experiment nicht sinnvoll ist, weil es zu teuer, physikalisch nicht durchführbar oder zu gefährlich ist. Beispiele für derartige Einsatzgebiete sind die Verformung von Tragflächen im Luftstrom [86], die Kollision von Galaxien [105] oder die Simulation von Atomexplosionen, die im Rahmen der *Accelerated Strategic Computing Initiative* (ASCI) [82] durchgeführt werden soll. Insbesondere die Alterungsprozesse kerntechnischer Apparate, die eigentlich eine kontinuierliche Überprüfung notwendig machen, sollen im ASCI-Projekt simuliert werden.

Simulationen komplexer Systeme benötigen allerdings enorme Rechenkapazitäten, um in sinnvoller Zeit zu verwertbaren Ergebnissen zu kommen – Kapazitäten, die zur Zeit in Form von herkömmlichen sequentiellen Rechnern allein nicht zur Verfügung stehen. Durch die Entwicklung von Parallelrechnern wurde ein erster Schritt gemacht, höhere Rechenleistungen verfügbar zu machen. Parallelrechner haben beim momentanen Stand der Technik die größten Aussichten, die in Zukunft benötigten Rechenkapazitäten bereitzustellen. Als konsequente Fortsetzung dieses Gedankens wird im ASCI-Projekt die Steigerung der Leistung einzelner homogener Rechner angestrebt. So wurde Ende 1996 erstmalig die „magische“ Grenze von einem Teraflop von der Intel ASCI-Maschine durchbrochen [119]. Ziel von ASCI ist es, die verfügbare Rechenkapazität einzelner Systeme bis zum Jahre 2004 um zwei Größenordnungen zu erhöhen.

Aber auch diese neueren Systeme sind noch nicht in der Lage, die Rechenkapazität zur Verfügung zu stellen, die die Berechnung der „großen Herausforderungen“ (*grand challenges*) des wissenschaftlichen Rechnens [65] erlaubt: Kernfragen aus den Gebieten der atmosphärischen Chemie, der Astrophysik, der Materialforschung, der Molekularbiologie, der Elementarteilchenphysik und der Aerodynamik sind teilweise noch gar nicht, teilweise zumindest nicht in akzeptabler Zeit simulierbar. Neben der Rechenkapazität ist auch die Speicherkapazität ein entscheidender Engpaß. Aber selbst Anwendungen, die nicht zu dem Bereich dieser Herausforderungen zählen, wie z.B. die Wettervorhersage, brauchen für detaillierte Simulationen zu lange, um verwertbare Ergebnisse zu erhalten. Neben der Leistungssteigerung der Hochleistungsrechner müssen neue Wege gefunden werden, um eine größere Rechenkapazität zur Verfügung zu stellen.

Hier bieten die Fortschritte der Netzwerktechnik einen innovativen Ansatzpunkt. Mittlerweile stehen für WANs (*wide area networks*) Übertragungskapazitäten zur Verfügung, die noch vor wenigen Jahren leistungsstarken Parallelrechnern vorbehalten waren: Kapazitäten von 2.4 GBit/s sind bereits prototypisch in WANs realisiert [35]. Dies ist mit der internen Übertragungskapazität zwischen zwei Knoten einer Cray T3E vergleichbar. Dadurch liegt der Gedanken nahe, mehrere Supercomputer miteinander zu koppeln und so zu einer großen Leistungssteigerung zu kommen. *Metacomputing*, also die Kopplung von

zwei oder mehr Superrechnern, die über ein WAN miteinander verbunden sind, zu einem *Metacomputer* wird sinnvoll einsetzbar [123]. Hierbei müssen aber nicht unbedingt gleiche Computer eingesetzt werden - die Kopplung eines massiv-parallelen Systems mit einem Vektorrechner kann zu weiteren Vorteilen führen, wenn man die Heterogenität bewußt einsetzt. So können z.B. die Teile einer Anwendung, die sich gut parallelisieren lassen, von einem massiv-parallelen System berechnet werden, die Teile hingegen, die sich durch gute Vektorisierbarkeit auszeichnen, können von einem Vektorrechner bearbeitet werden. Man kann dann ein verteiltes Rechnersystem so zusammenstellen, daß es optimal auf die Bedürfnisse der Anwendung abgestimmt ist. Dadurch ist es möglich, eine deutliche Reduzierung der Bearbeitungszeit einer Anwendung zu erreichen. Auch die Nutzung von Spezial-Hardware, wie z.B. für die Visualisierung der berechneten Ergebnisse, verspricht eine weitere Beschleunigung.

Um diese theoretischen Vorteile in der Praxis nutzen zu können, müssen jedoch noch zahlreiche Probleme bewältigt werden. Zunächst ist hier die externe Kommunikation zu nennen. Obwohl die Leistungskapazität der externen Netzwerke stark gestiegen ist, bleibt sie in der Praxis immer noch weit hinter der Kapazität der internen Netzwerke von homogenen Hochleistungsrechnern zurück. Insbesondere die relativ hohe Latenz, die sich beim Einsatz eines WANs aufgrund der Signallaufzeit ergibt, ist ein Hindernis für den effizienten Einsatz eines Metacomputers. Dies kann dazu führen, daß der Zeitgewinn durch die höhere Rechenleistung durch den gleichfalls gestiegenen Kommunikationsaufwand wieder verlorengeht.

Ein weiterer wichtiger Aspekt beim Metacomputing ist der Lastausgleich. Da die zu einem Metacomputer gekoppelten einzelnen Rechner im allgemeinen über deutlich unterschiedliche Merkmale wie Rechen- und Speicherkapazität, *Cache*-Größe etc. verfügen, muß sichergestellt werden, daß alle Rechner gleichmäßig ausgelastet sind, damit keine Rechenleistung durch Warten auf langsamere Komponenten verloren geht.

Schließlich erhofft man auch, durch Metacomputing eine steigende Auslastung der einzelnen Superrechner zu erreichen. Aufgrund ungünstiger Zusammenstellungen der zu berechnenden Anwendungen auf einem System kann insbesondere ein paralleler Rechner nicht vollständig ausgelastet sein. Die somit vorhandene ungenutzte Rechenkapazität verschiedener Rechner könnte zusammengefaßt und weiteren Anwendungen zur Verfügung gestellt werden.

Ziel dieser Arbeit ist es, ein Programmierkonzept zu entwickeln und vorzustellen, das eine effiziente Ausführung von Anwendungen auf einem Metacomputer ermöglicht. Der Schwerpunkt liegt dabei nicht darauf, eine hohe Auslastung der einzelnen Systeme, sondern eine möglichst kurze Bearbeitungszeit von Anwendungen zu erreichen. Weiter sollen nicht einzelne Anwendungen oder Algorithmen optimiert werden, sondern es gilt, ein möglichst breites Anwendungsspektrum zu unterstützen. Insbesondere ist auch eine geeignete Lastverteilungsstrategie ein wichtiger Gesichtspunkt. Das Konzept wird anhand einer prototypischen Implementierung in der Praxis getestet, und soll langfristig für den Einsatz von großen Metacomputing-Anwendungen genutzt werden.

Die Arbeit gliedert sich wie folgt: Im Kapitel 2 wird der Begriff „Metacomputer“ definiert. Während zunächst die Vorteile von Metacomputing vorgestellt werden, folgt im

hinteren Teil des Kapitels eine Übersicht über die in der Praxis zu beobachtenden Probleme. Weiter werden die zur Zeit einsetzbaren Programmierkonzepte untersucht, die für die Programmentwicklung von Metacomputing-Anwendungen eingesetzt werden können. Der Schwerpunkt des Kapitels ist aber die Beschreibung der Vorteile und der Probleme, die der Einsatz eines Metacomputers mit sich bringt. Eine ausführliche Fallstudie motiviert in Kapitel 3 die Optimierungen, die als wichtig für den effizienten Einsatz eines Metacomputers identifiziert werden. In Kapitel 4 wird ein Programmiermodell namens CoFu entwickelt und formal spezifiziert, mit dem „typische“ Programme für Hochleistungsrechner modelliert werden können. Dieses Modell und die damit zusammenhängenden Eigenschaften von CoFu-Anwendungen werden im weiteren vorausgesetzt. Das eigentliche Konzept zur Optimierung von Metacomputing-Anwendungen wird in Kapitel 5 vorgestellt. Es umfaßt verschiedene Optimierungen der Kommunikation, aber auch einen statischen und einen dynamischen Lastausgleichsalgorithmus. Dabei werden gezielt die Eigenschaften von CoFu-Programmen eingesetzt. In Kapitel 6 werden die Ansätze dieser Arbeit mit existierenden Ansätzen in der Literatur verglichen. Hier werden nicht nur die Unterschiede verdeutlicht, sondern auch die Gemeinsamkeiten herausgearbeitet, die eine Kopplung verschiedener Ansätze ermöglichen. Die Beschreibung der prototypischen Implementierung wird im anschließenden Kapitel 7 vorgenommen. Dabei wird auch die Programmierschnittstelle beschrieben, die den Einsatz der hier vorgestellten Optimierungen für andere Anwendungen ermöglicht. Kapitel 8 validiert das Konzept und die vorgenommenen Optimierungen anhand ausführlicher Messungen. Eine Zusammenfassung mit einem Ausblick schließen die Arbeit ab.

2 Metacomputing

In diesem Kapitel werden der Ausdruck Metacomputing definiert und die Unterschiede zu verwandten Begriffen wie z.B. heterogenem Rechnen und *hypercomputing* verdeutlicht. Neben einer Zusammenfassung der potentiellen Vorteile der Nutzung eines Metacomputers werden auch auftretende Schwierigkeiten beschrieben.

2.1 Definition

In der Literatur werden die Begriffe „Metacomputer“ und „Metacomputing“ nicht immer einheitlich verwendet. So versteht Smarr [123] unter dem Begriff „Metacomputer“ diejenigen Computer-Ressourcen, die dem Nutzer transparent über ein Netzwerk zugänglich sind. Dies ist wohl die allgemeinste Definition, da weder genauer spezifiziert ist, welche Art von Computer-Ressourcen genutzt werden, noch nähere Angaben über das sie verbindende Netzwerk gemacht werden. Im Sinne dieser Definition kann nach Smarr sogar ein einzelner Personalcomputer (PC) als „Mini-Metacomputer“ bezeichnet werden. Das Netzwerk ist in diesem Fall der interne Bus des PCs, die Computer-Ressourcen der Mikroprozessor, Graphikprozessoren, Ein- und Ausgabegeräte etc. Andere Definitionen dagegen konkretisieren den Begriff der Computer-Ressource oder des genutzten Netzwerks. So bezeichnen Ramme und Kremer [112] mit „Metacomputer“ ein gekoppeltes System von Rechnern, von Personalcomputern und Arbeitsplatzrechnern bis hin zu parallelen Supercomputern. Während hier konkret die gekoppelten Recheneinheiten spezifiziert werden, wird keine weitere Angabe über das die Rechner verbindende Netz gemacht. In [55] wird hierfür der Begriff *metasystem* geprägt. Oftmals wird „Metacomputing“ auch benutzt, um die Nutzung eines Verbundes von Arbeitsplatzrechnern zu beschreiben, wofür allerdings auch der Begriff *hypercomputing* [37] gebräuchlich ist.

Die Begriffe „Metacomputer“ bzw. „Metacomputing“ werden hier in folgendem Sinne verwendet:

METACOMPUTER: Ein METACOMPUTER ist ein verteiltes Rechnersystem, das aus zwei oder mehr Hochleistungsrechnern besteht, die über ein *wide area network* (WAN) miteinander gekoppelt sind.

METACOMPUTING: METACOMPUTING ist die Nutzung eines Metacomputers zur Berechnung einer Anwendung.

Dabei bezeichnet der Begriff ANWENDUNG ein oder mehrere Programme, die in enger Abstimmung gemeinsam eine Aufgabe lösen. Ein PROGRAMM wiederum, welches ausgeführt wird, kann aus einem oder mehreren Prozessen bestehen, denen der gleiche Quell-Code zugrundeliegt. In der Literatur wird ein Metacomputer im hier definierten Sinne auch als *distributed heterogeneous supercomputing system* (DHSS) [54] bezeichnet, oder es wird von *global distributed heterogeneous supercomputing* (*global* DHS) [48] gesprochen.

Im Gegensatz zu den anderen oben aufgeführten Definitionen wird hier einerseits die ausschließliche Kopplung von Arbeitsplatzrechnern, sozusagen als einfacher Parallelrechner,

ausgeschlossen, andererseits wird die nicht-lokale Kopplung der Rechner betont. Um allgemein eine Kopplung unterschiedlicher Rechner zu bezeichnen, wird in dieser Arbeit der Begriff HETEROGENES RECHNEN (*heterogeneous computing*) benutzt. Dabei können sich zwei Rechner schon dadurch unterscheiden, daß sie über eine unterschiedliche Menge an Speicher verfügen oder eine andere Compiler-Version benutzen, da dies oftmals einen erheblichen Einfluß auf die Rechenleistung hat, die ein Programm nutzbar machen kann. Somit ist Metacomputing im Sinne der oben gegebenen Definition eine besondere Form des heterogenen Rechnens.

Die Definition des Begriffes Metacomputing spiegelt allerdings in erster Linie den momentanen Sprachgebrauch wieder. Aufgrund der stetig anwachsenden Rechenleistung von Prozessoren und der damit verbundenen Leistungssteigerung von Arbeitsplatzrechnern spielt auch die Kopplung von Arbeitsplatzrechnern als potentielle Hochleistungsrechner der Zukunft eine wichtige Rolle [126]. Die in dieser Arbeit entwickelten Konzepte zur effizienten Nutzung eines Metacomputers können durchaus auch in diesem Bereich eingesetzt werden, allerdings beschränken sich die hier vorgenommenen Untersuchung auf Metacomputer im obigen Sinne.

Im folgenden Abschnitt werden die Vorteile des Metacomputings vorgestellt.

2.2 Vorteile des Metacomputings

Der offensichtliche Vorteil eines Metacomputers ist der Zuwachs an Rechen- und Speicherkapazität. Dadurch ist es möglich, Anwendungen zu rechnen, die z.Zt. nicht auf einem einzelnen Hochleistungsrechner bearbeitet werden können, da entweder der Speicherplatz nicht ausreicht oder die benötigte Rechenzeit zu groß ist. Durch Metacomputing besteht Hoffnung, schon jetzt Anwendungen aus dem Bereich der *grand challenges* [65, 117] in akzeptabler Zeit bearbeiten zu können.

Derartige Anwendungen für Hochleistungsrechner bestehen im allgemeinen aus einzelnen Teilen, die jeweils völlig unterschiedliche Anforderungen an den Rechner stellen, von dem sie bearbeitet werden. So können diese Teile in völlig unterschiedlichem Maße

- parallelisierbar sein,
- vektorisierbar sein,
- Ein- und Ausgaben durchführen,
- Anforderungen an die Größe des Hauptspeichers stellen,
- Spezialhardware wie z.B. Graphikprozessoren nutzen.

Auch die einzelnen Rechner, die einen Metacomputer bilden, können nach ihrer Architektur in verschiedene Klassen wie z.B. SIMD-, MIMD-, Vektor- und VLIW (*very large instruction word*) - Maschinen [54] eingeteilt werden. Betrachtet man weiterhin die Speicherorganisation (z.B. *Cache*-Struktur, *stream-buffer* [97]) und das interne Netzwerk, so erschließt sich eine Vielzahl von Maschinen, die sich in ihren Leistungsmerkmalen signifikant unterscheiden und somit für die einzelnen Teile einer Anwendung mehr oder weniger gut geeignet sind. Dies bedeutet, daß bei der Nutzung eines homogenen Systems nur ein Teil der gesamten Anwendung auf einer adäquaten Plattform gerechnet wird; ein unter

Umständen großer Anteil hingegen wird von einem Rechner bearbeitet, der dafür keine optimale Leistung liefert. Im Gegensatz dazu erhofft man sich durch die Nutzung eines heterogenen Systems eine Reduktion der Anwendungslaufzeit, indem man die einzelnen Teile der Anwendung von gerade dem Rechner bearbeiten läßt, der eine möglichst gute Leistung dafür bietet. Als Beispiel für die mögliche Reduktion der Laufzeit betrachte man ein hypothetisches Programm, das aus drei Teilen bestehe, von denen der erste, der 40% der gesamten Rechenzeit ausmache, sehr gut parallelisierbar sei, der zweite mit einem Anteil von 30% am besten für einen Vektorrechner geeignet sei und der Rest auf einem Graphik-Rechner die beste Leistung liefere. Der Unterschied zwischen der Bearbeitung dieses Programms auf verschiedenen homogenen Systemen und der Bearbeitung auf einem Metacomputer ist in Abbildung 1 schematisch dargestellt. Auf der linken Sei-

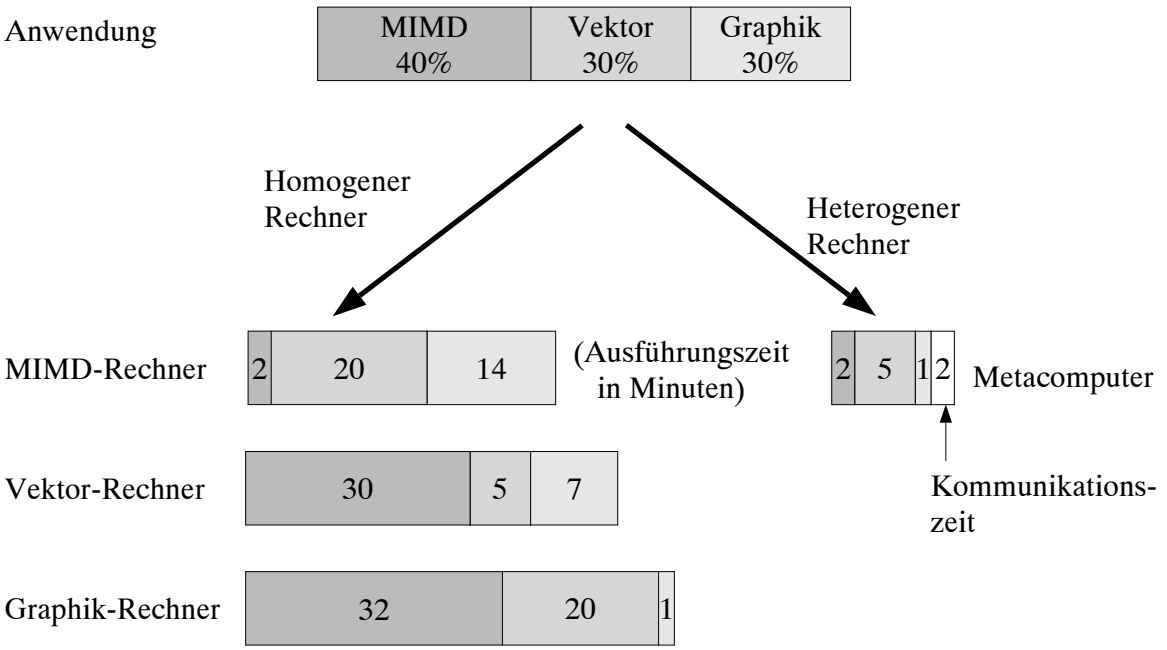

Abbildung 1: Vergleich der hypothetischen Bearbeitungszeit einer Anwendung auf verschiedenen homogenen Rechnern und einem Metacomputer.

te sind exemplarisch Rechenzeiten aufgeführt, wie sie sich mit einem einzelnen MIMD-, Vektor- oder Graphikrechner ergeben können[1]. Auf der rechten Seite hingegen steht die hypothetische Bearbeitungszeit auf einem Metacomputer, der aus allen drei Rechnertypen besteht. Hierbei muß allerdings zusätzlich die Zeit für die Datenübertragung zwischen den einzelnen Rechnern berücksichtigt werden. Auf dem Metacomputer ergibt sich dann z.B. eine Gesamtbearbeitungszeit von 10 Minuten, was einem einem superlinearen *speedup* entspricht, da die minimale Bearbeitungszeit auf einem einzelnen Rechner 36 Minuten beträgt und sich somit ein *speedup* von $36/10 = 3.6 > 3$ ergibt. Dieser Effekt eines superlinearen *speedups* ist nicht nur theoretisch möglich, sondern er konnte bereits bei realen

[1]Dabei wird davon ausgegangen, daß die Anwendung auf den einzelnen Rechner tatsächlich bearbeitet werden kann.

Anwendungen nachgewiesen werden [145].

In der Praxis ist aber häufig eine wie oben dargestellte Bearbeitung auf einem homogenen System gar nicht möglich. So kann es sein, daß der Graphikteil der Anwendung spezielle Hard- und Software voraussetzt und dementsprechend nur auf einem Graphikrechner ausgeführt werden kann, der wiederum nicht über ausreichend Speicher- und Rechenkapazität verfügt, um die Berechnung vorzunehmen. In der Regel werden deshalb Berechnung und Visualisierung unabhängig voneinander sequentiell durchgeführt. Dadurch ist aber ein interaktives Arbeiten, also die Visualisierung der Ergebnisse schon zur Rechenzeit und die Steuerung der Berechnung, nicht möglich. Dies wird erst durch die Kopplung von Hochleistungsrechnern mit entsprechender Graphikhardware wie z.B. einer *Cave* oder einer *PowerWall* [114], also durch Metacomputing, möglich. Ein wichtiger Einsatzbereich für die interaktive Steuerung sind Parameterstudien, bei denen eine Berechnung für einen großen Parameterraum durchgeführt werden muß, wozu oftmals ein enormer Rechenaufwand erforderlich ist. Die Visualisierung der Ergebnisse zur Laufzeit ermöglicht es, unwichtige Teile des Parameterraumes frühzeitig zu erkennen und die Berechnung auf relevante Bereiche zu beschränken. Durch diese Techniken werden unnötige Wartezeiten vermieden, wodurch im Produktionsbetrieb teure Rechenzeit eingespart und ein höherer Durchsatz erreicht werden kann. Darüberhinaus können durch die Visualisierung auch Fehler, seien es nun Programmierfehler oder Fehler bei der Modellierung, schneller erkannt werden [111].

Als Beispielszenario hierfür betrachte man die Verwaltung und Visualisierung großer Datenmengen, wie sie typischerweise bei Rechnersimulationen, Experimenten oder Messungen anfallen. Heute nimmt z.B. die Auswertung nur eines Bruchteils der Radardaten, die im Rahmen der Flüge des *Space Shuttle Endeavour* 1994 gemessen wurden, auf Arbeitsplatzrechnern fast zwei Jahre in Anspruch [91]. Durch die Kopplung von Hochleistungsrechnern mit Graphik-Arbeitsplätzen sind die Wissenschaftler in der Lage, mit nur einem Bruchteil der insgesamt notwendigen Rechen- und Arbeitszeit die für sie interessanten Daten auszuwählen und dann auszuwerten.

Ein weiterer nicht zu unterschätzender Vorteil des Einsatzes heterogener Rechner ist die Zeitersparnis bei der Programmentwicklung für ein heterogenes System. Es sind ein hohes Maß an Erfahrung und ein erheblicher Arbeitsaufwand notwendig, um ein sequentielles Programm effizient für einen massiv-parallelen Rechner mit verteiltem Speicher umzustellen. Hauptgrund sind hierfür gerade die Programmteile, die nicht für einen Parallelrechner geeignet sind, aber trotzdem aus Effizienzgründen oder wegen mangelnden Hauptspeichers parallelisiert werden müssen. Als Beispiel betrachte man die Lösung eines großen linearen Gleichungssystems. In vielen Programmen wird ein signifikanter Anteil der Bearbeitungszeit für die Aufstellung des Gleichungssystems benötigt. Da die Einträge in der Koeffizientenmatrix unabhängig voneinander berechnet werden können, ist dies ein inhärent paralleler Programmteil. Die Berechnung der Lösung des aufgestellten Gleichungssystems hingegen ist günstiger sequentiell vorzunehmen, muß nun aber auch parallelisiert werden, da (eine entsprechende Größe des Gesamtsystems vorausgesetzt) ein einzelner Prozessor eines Parallelrechners nicht über ausreichend Hauptspeicher verfügt oder zu langsam ist. Dies ist wiederum mit erheblichem Programmieraufwand verbunden, da die einzelnen Programmteile nicht mehr unabhängig voneinander arbeiten können. Würde hingegen das

Gleichungssystem auf einem Vektorrechner gelöst, der über einen größeren Hauptspeicher verfügt, entfiele dieser zusätzliche Aufwand der Parallelisierung. Diese Überlegung allein war für Morton und Tyler [96] ein ausreichendes Argument für die Nutzung eines Metacomputers. Außerdem kann sogar mit einer zusätzlichen Beschleunigung der Anwendung gerechnet werden, da dort die Lösung des Gleichungssystems ggf. schneller möglich ist als auf einem Parallelrechner. Ein ausführliches Beispiel für die Verteilung einer großen Anwendung auf einen Metacomputer findet sich in [141].

Schließlich ermöglicht die Kopplung heterogener Rechnersysteme auch eine größere Auslastung der einzelnen Rechner, da z.B. ungenutzte, kleinere Partitionen von verschiedenen Parallelrechnern zusammengefaßt werden können und dadurch eine weitere Anwendung berechnet werden kann. Auch entstehen wirtschaftliche Vorteile durch die Nutzung „entfernter" Spezialrechner (im Sinne von *remote computing*) und Softwarepakete, da beide nicht vor Ort zusätzlich angeschafft und gewartet werden müssen.

Um einen Metacomputer einsetzen zu können, muß es aber zunächst überhaupt möglich sein, ein Programm für ein verteiltes, heterogenes System zu entwickeln. Im nächsten Abschnitt werden Programmierkonzepte bzgl. ihrer Einsetzbarkeit im Metacomputing untersucht und bewertet.

2.3 Programmierung eines Metacomputers

Da ein Metacomputer aus mindestens zwei gekoppelten Rechnern besteht, können die Konzepte aus der parallelen Programmierung auch zur Programmentwicklung für Metacomputer eingesetzt werden. Gerade für parallele Algorithmen gibt es aber einen wichtigen Unterschied zwischen einem homogenen Parallelrechner und einem Metacomputer: Während ein Parallelrechner mittlerweile im wesentlichen unabhängig von der Topologie des Netzwerkes über die gleiche Kommunikationsleistung zwischen beliebigen Prozessoren verfügt[2], gibt es bei einem Metacomputer eine vergleichsweise langsame externe Verbindung. Hieraus ergibt sich zunächst das technische Problem, daß für die Kommunikation unterschiedliche Schnittstellen benutzt werden müssen, abhängig davon, ob intern, also innerhalb eines Rechners, oder extern, also zwischen verschiedenen Rechnern kommuniziert werden muß. Man beachte insbesondere, daß die Prozessoren eines einzelnen Rechners über einen gemeinsamen Speicher verfügen können, so daß die Nutzung des Netzwerks nicht explizit programmiert werden muß, wohingegen der externe Datenaustausch mit einem anderen Rechner zumindest auf einer systemnahen Programmschicht gerade dies erfordert. Neben diesem technischen Problem kann außerdem die Effizienz eines parallelen Algorithmus durch eine langsame Verbindung signifikant verschlechtert werden, so daß sich in Einzelfällen sogar eine längere Bearbeitungszeit bei der Nutzung eines heterogenen Systems ergeben kann. Deshalb werden im folgenden kurz die wichtigsten Konzepte der parallelen Programmierung vorgestellt und auf ihre Nutzbarkeit für die Programmierung eines Metacomputers untersucht. In Anlehnung an die Klassifikation von Kumar et al. [80] werden die beiden Klassen der expliziten und der impliziten parallelen Pro-

[2]Durch moderne Techniken wie das *wormhole-routing* und *virtual-cut-through* [80, 104] ergeben sich nur noch minimale Unterschiede in der Signallaufzeit, die in der Praxis vernachlässigt werden können.

grammierung untersucht. Bei der EXPLIZITEN PARALLELEN PROGRAMMIERUNG müssen Anweisungen für den Datenaustausch und für die Synchronisation der Prozessoren bei der Programmentwicklung eingefügt werden, wohingegen bei der IMPLIZITEN PARALLELEN PROGRAMMIERUNG sequentielle Programme durch den Compiler in ein paralleles Programm übersetzt werden, d.h. der Compiler fügt entsprechende Anweisungen ein, um das Programm parallel ausführen zu können.

2.3.1 Implizite parallele Programmierung

Bei der Programmentwicklung ist die implizite parallele Programmierung im Vergleich zur expliziten einfacher, da die Aufgabe der Parallelisierung dem Compiler überlassen wird. Für Rechner mit gemeinsamen Speicher gibt es vielversprechende Ansätze; hier sei nur OpenMP [22] erwähnt, das die Programmiersprachen Fortran, C und C++ um Compiler-Direktiven und zusätzliche Funktionen erweitert, so daß Parallelismus durch Nutzung des gemeinsamen Speichers (*shared-memory parallelism*) ausgedrückt werden kann. Für Systeme mit verteiltem Speicher hingegen ist die Situation schwieriger. Die für die Datenverteilung notwendige Analyse der Datenabhängigkeit, die durch Programmverzweigungen, Prozeduraufrufe, Schleifen etc. erschwert wird, führt bei der derzeitigen Compilertechnik im allgemeinen zu recht ineffizienten Programmen [84, 133]. Um die Aufgabe für den Compiler zu vereinfachen, kann bzw. muß bei der Programmierung oftmals zusätzlich spezifiziert werden, wie die Daten auf die Prozessoren verteilt werden sollen. Dies geschieht z.B. beim SVM Fortran [53] durch Direktiven, die ein eigener Präprozessor verarbeitet und so aus dem sequentiellen Fortran-Programm ein durch *message-passing* parallelisiertes Programm erzeugt. *High Performance Fortran* (HPF) [44, 78] und Fortran D [80] sind Fortran-Dialekte, die um zusätzliche Deklarationen und Befehle zur Parallelisierung erweitert wurden.

Während implizite parallele Programmierung für homogene Systeme, insbesondere für SSMP-Rechner (*scalable shared memory multiprocessor*), gut geeignet ist, ist sie für die Programmentwicklung für einen Metacomputer nicht geeignet, da es keine Implementationen dieser Sprachen gibt, die auf einem heterogenen System einsetzbar sind.

2.3.2 Explizite parallele Programmierung

Bei der expliziten parallelen Programmierung müssen Anweisungen für den Datenaustausch und für die Synchronisation der Prozessoren bei der Programmentwicklung eingefügt werden. Es gibt verschiedene Mechanismen, mit denen dies erfolgen kann:

***Message-Passing*:** Hierbei werden bei der Programmentwicklung Befehle zum Senden und Empfangen von Daten verwendet. Um Daten von einem Prozessor A zu einem anderen Prozessor B zu übermitteln, muß A eine Sende-Anweisung und B eine Empfangsanweisung ausführen. Während ältere Parallelrechner hierfür herstellerspezifische Kommunikationsbibliotheken wie z.B. NX [69] der Intel Paragon oder MPL der IBM SP2 [70] zur Verfügung stellten, hat sich mittlerweile die Nutzung von einheitlichen Kommunikationsbibliotheken durchgesetzt. Diese de-facto

Standards stehen mittlerweile für nahezu alle Parallelrechner in einer (oftmals vom Hersteller) optimierten Version zur Verfügung, so daß die Vorteile der Portabilität mit einer hohen Leistung gekoppelt sind. Hier ist in erster Linie das *Message-Passing Interface* MPI [89] zu nennen, das nicht nur auf neuen Rechnern, sondern auch auf älteren Systemen wie der Intel Paragon oder der IBM SP2 zur Verfügung steht. Auch die *Parallel Virtual Machine* PVM [52] bildet einen weiteren, älteren de-facto Standard. Beide Bibliotheken unterstützen die wichtigsten Konzepte wie globale Operationen und Punkt-zu-Punkt-Kommunikationen. Eine Diskussion der Unterschiede dieser beiden Bibliotheken findet sich in [57].

Damit eine effiziente parallele Programmierung möglich ist, müssen verschiedene Semantiken für Sende- und Empfangsanweisungen vorhanden sein, um zeitaufwendiges Kopieren von Daten zu verhindern:

- Beim GEPUFFERTEN Senden (*buffered send*) werden die zu übertragenden Daten beim Sender oder Empfänger zwischengespeichert. Dadurch steht der Speicherbereich der zu versendenden Daten direkt wieder zur Verfügung, doch kostet das Kopieren Zeit und zusätzlichen Speicherplatz.
- Beim SYNCHRONISIERENDEN Senden wird eine Sende-Operation erst beendet, wenn die Daten empfangen worden sind. Wenn auch das Warten auf den Empfänger ggf. Zeit kostet, wird hier ein unnötiges Kopieren der Daten verhindert.
- Bei einem NICHT-BLOCKIERENDEN Senden kann das Programm sofort mit der Bearbeitung fortfahren, auch wenn die Daten noch nicht aus dem Speicher geholt worden sind.

Natürlich sind auch Kombinationen dieser Operationen denkbar, siehe z.B. die Vielzahl von Sende-Operationen in MPI [89]. Durch diese verschiedenen Sende-Operationen können Optimierungen vorgenommen werden, die die Leistung eines Programms steigern. Nicht-blockierende Sende- und Empfangsanweisungen verhindern beispielsweise ein Kopieren der Daten und ermöglichen eine Überlappung von Kommunikation und Berechnung. Dies wird gerade bei Programmen für Hochleistungsrechner häufig eingesetzt.

Besonders wichtig ist es für das Metacomputing, daß eine *message-passing*-Bibliothek nicht nur die Kopplung verschiedener Rechner erlaubt, sondern daß sie auch unterschiedliche Kommunikationsprotokolle nutzen kann, also die sogenannte MULTIPROTOKOLLFUNKTIONALITÄT besitzt (auch als *multimethod-communication* [45] bezeichnet). Dadurch kann sowohl für die interne als auch für die externe Kommunikation das Protokoll gewählt werden, das die höchste Leistung ermöglicht.

Einseitige Kommunikation: Bei der einseitigen Kommunikation muß, im Gegensatz zur Kommunikation durch Nachrichtenaustausch, nur einer der an der Kommunikation beteiligten Prozessoren eine Anweisung ausführen - entweder ein Lese- oder ein Schreibzugriff auf den Speicher eines anderen Prozessors. Obwohl dieses Konzept des expliziten Datenaustausches einen gemeinsamen Speicher vorauszusetzen scheint, ist dies in der Praxis nicht so. Für die Cray-Rechnersysteme einschließlich der massiv-parallelen Systeme mit verteiltem Speicher wie die T3D und T3E wer-

den sie als sogenannte *shared memory*-Routinen (*shmem*) zur Verfügung gestellt. Auch sieht der neue MPI-2 Standard [90], der nicht nur für Rechner mit gemeinsamen Speicher gedacht ist, ebenfalls einseitige Kommunikationsroutinen vor.

Der große Nachteil dieses Ansatzes bzgl. heterogener Rechnersysteme ist jedoch die mangelnde Verfügbarkeit. Die Bibliotheken, die eine einseitige Kommunikation ermöglichen, liegen nur für einzelne, homogene Parallelrechner vor; vom neuen MPI-2 Standard existiert z.Zt. nur eine Implementation der Firma Pallas GmbH [101].

***Remote Procedure Call* (RPC):** Bei einem RPC wird auf einem Prozessor eine Prozedur aufgerufen, die auf einem anderen bearbeitet wird. Die zu sendenden Daten sind dabei die übergebenen Prozedurparameter. Bei einem Standard-RPC blockiert der aufrufende Prozeß so lange, bis das Funktionsergebnis zur Verfügung steht („synchroner RPC“) [120]. Dies widerspricht allerdings den Anforderungen für die Programmierung von Hochleistungsrechnern, da durch das Blockieren letztendlich eine sequentielle und keine parallele Bearbeitung erreicht wird. Erweiterungen des RPC wie z.B. asynchrone Aufrufe durch *futures* [139] oder der DFN-RPC [109] sind zwar vorgeschlagen worden, doch existieren bis heute noch keine Implementationen dieser Konzepte für heterogene Hochleistungsrechner [110]. Auch hat sich das Konzept der RPCs im Bereich des Hochleistungsrechnens nicht durchsetzen können, so daß schon aus Gründen der Portabilität dieses Konzept nicht geeignet ist.

Linda: Ein logisch gemeinsamer Speicherbereich, der sogenannte *tuple space* (Tupel-Raum), in dem Daten abgelegt und entfernt werden können, bildet den Kern von Linda [106]. In diesen Tupel-Raum können zwei Arten von Tupeln enthalten sein: einerseits Prozeß-Tupel, die die aktiven Prozesse der gesamten Anwendung bilden, andererseits passive Daten-Tupel. Die aktiven Prozeß-Tupel können Daten austauschen, indem sie Daten-Tupel erzeugen, lesen oder sie aus dem Tupel-Raum entfernen. Eine ausführlichere Darstellung von Linda findet sich in [106].

Aufgrund des hohen Aufwandes für den Zugriff auf diesen Tupel-Raum ist Linda nur für grob-granulare Anwendungen geeignet [38]. Zusammen mit der recht geringen Verbreitung spricht dies gegen den Einsatz von Linda in dem Bereich des Metacomputings.

Erweiterte Programmiersprachen: Häufig werden existierende sequentielle Programmiersprachen durch Bibliotheken oder Sprachkonstrukte erweitert, damit eine parallele Bearbeitung des Programms ermöglicht wird. Beispiele hierfür sind Klassenbibliotheken für C++ [122], die ein *message-passing* realisieren, oder Methodenaufrufe, die einen synchronen oder asynchronen RPC-Aufruf durchführen [74]. Auch gibt es Programmiersprachen, die für die Programmierung eines Parallelrechners oder eines Transputers entwickelt worden sind und dazu Befehle enthalten, die den Datenaustausch zwischen verschiedenen Prozessen ermöglichen. Ein Beispiel dafür ist OCCAM [106]. Da bei allen diesen Erweiterungen der Datenaustausch weiterhin explizit programmiert werden muß, zählen diese Programmierkonzepte zum Bereich der expliziten parallelen Programmierung.

Derartige Programmiersprachen existieren allerdings nur für homogene Systeme, so daß der Einsatz für die Entwicklung von Metacomputing Anwendungen nicht möglich ist.

2.3.3 Folgerungen für die Programmierung eines Metacomputers

Zusammengefaßt ist also nur *message-passing* als Programmierkonzept für einen Metacomputer geeignet, da von den Konzepten zur expliziten parallelen Programmierung Linda und RPC keine ausreichende Leistung bieten und erweiterte Programmiersprachen und einseitige Kommunikation für heterogene Systeme noch nicht unterstützt werden. Aus den gleichen Gründen sind auch die Ansätze zur impliziten parallelen Programmierung nicht geeignet. Ein zusätzlicher Vorteil ist, daß etablierte Programmiersprachen wie Fortran, C und C++ mit den für sie existierenden optimierenden Compilern weiter eingesetzt werden können. Weiterhin gibt es wegen der weiten Verbreitung von *message-passing* zahlreiche Anwendungen, die von den hier entwickelten Konzepten und Optimierungen profitieren können.

Während somit durch *message-passing* die Erstellung von Metacomputing Anwendungen und der Einsatz zumindest in Testumgebungen möglich sind, gibt es noch zahlreiche Probleme zu bewältigen, bevor Metacomputing in einer Produktionsumgebung eingesetzt werden kann.

2.4 Herausforderungen des Metacomputings

Die Schwierigkeiten beim Einsatz von Metacomputing liegen, wie Abbildung 2 zeigt, in den unterschiedlichsten Ebenen. Dies beginnt bei der Hardware und der Netzprogrammierung und geht über den praktischen Einsatz im Produktionsbetrieb mit dem notwendigen *job-scheduling* bis hin zur Entwicklung und Optimierung von Metacomputing-Anwendungen. Im folgenden werden zunächst diese verschiedenen Ebenen und die dort auftretenden Schwierigkeiten erläutert.

Auf der Hardware-Ebene ist zunächst das Problem der rein technischen Kopplung unterschiedlicher Rechner über ein schnelles Netzwerk zu lösen. Mittlerweile sind 622–MBit/s–Netze Stand der Technik, und schnellere Netze mit einer Bandbreite von beispielsweise 2.4 GBit/s sind bereits exemplarisch realisiert [35]. Andererseits gibt es nur für wenige Hochleistungsrechner entsprechende Schnittstellen, die die Nutzung derartiger Bandbreiten direkt ermöglichen. So stehen für eine ATM-Verbindung für eine IBM SP2 und Cray T3E nur 155–MBit/s ATM-Schnittstellen zur Verfügung. Damit trotzdem die schnellen ATM-Leitungen eingesetzt werden können, müssen andere, oftmals umständliche und aufwendige Wege genutzt werden, wie z.B. ein *multiplexing* mehrerer langsamer ATM-Schnittstellen in einem Rechner. Alternativ kann das WAN auch nicht direkt an die beteiligten Rechner angeschlossen werden. Stattdessen nutzt man ein anderes Protokoll, welches wiederum für jeden einzelnen Rechner zur Verfügung steht. So kann z.B. eine HiPPI-Schnittstelle (*High Performance Parallel Interface*) mit einer Datenübertragung von 800 MBit/s genutzt werden, um zwei Rechner über ein ATM–WAN zu koppeln [73].

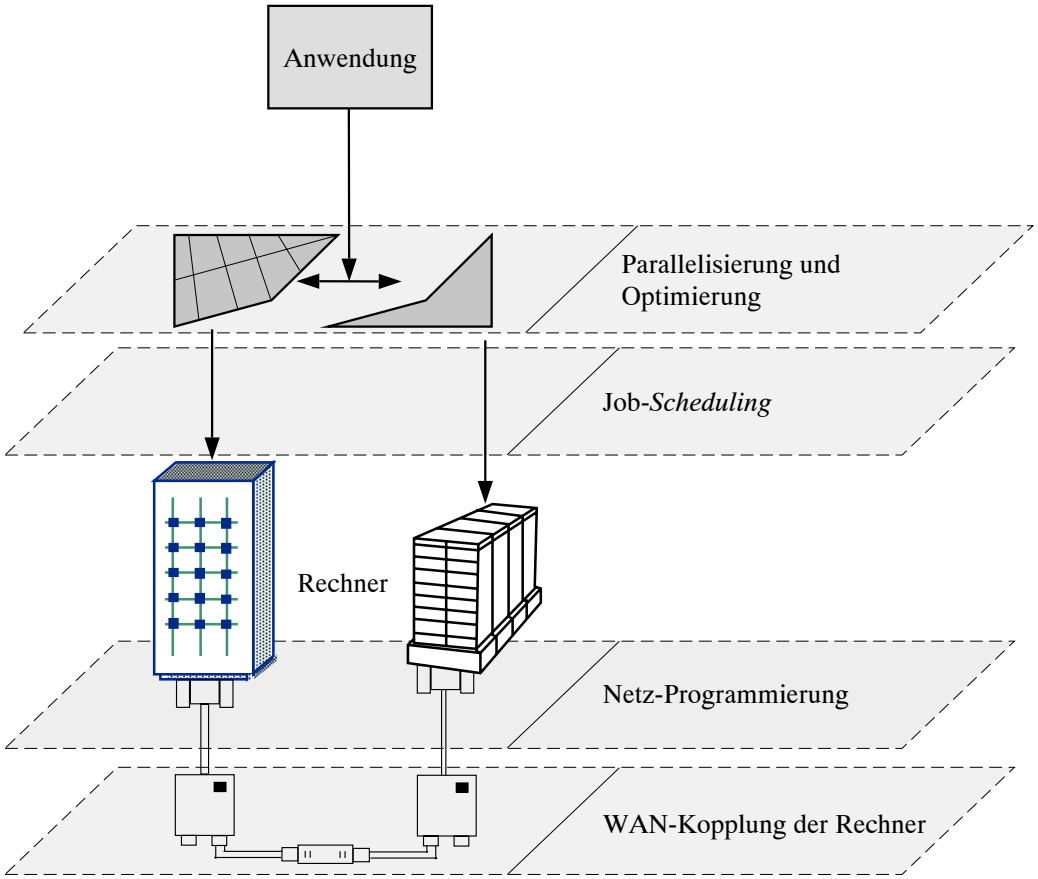

Abbildung 2: Schematische Darstellung der verschiedenen Problemebenen beim Einsatz eines Metacomputers.

Auf der nächst höheren Ebene, der Netz-Programmierung, muß eine Möglichkeit gefunden werden, den Datenaustausch zwischen den Rechnern über das schnelle WAN zu ermöglichen. Wie in Abschnitt 2.3 gezeigt wurde, ist z.Zt. nur *message-passing* geeignet, um ein derartiges verteiltes System effizient zu nutzen. Da die rechnerspezifische Programmierung der externen Schnittstelle für die Entwicklung von Anwendungen zu aufwendig ist, muß eine Kommunikationsbibliothek diese Funktionalität transparent zur Verfügung stellen. Diese Bibliothek muß nicht nur multiprotokollfähig sein, um eine hohe Leistung für die interne und externe Kommunikation zu ermöglichen, sondern sie muß auch die für heterogene Systeme im allgemeinen notwendige Datenkonvertierung vornehmen. Viele existierende Bibliotheken erfüllen diese Anforderungen nicht. So ist die Leistung von PVM [52] innerhalb einer einzelnen Maschine recht gering [76], *public-domain* MPI-Implementationen wie z.B. MPICH [58] bieten z.Zt. noch keine Multiprotokollkommunikation, herstellerspezifische MPI-Implementationen hingegen erlauben keine Kopplung heterogener Maschinen. Verschiedene Erweiterungen existierender Kommunikationsbibliotheken sind vorgenommen worden. Stellvertretend sei hier nur PVMPI [39] und PACX [5] genannt. PVMPI setzt, für den Benutzer der normalen MPI-Schnittstelle transparent, PVM für die externe und

MPI für die interne Kommunikation ein. PACX hingegen erweitert existierende, insbesondere herstellerspezifische MPI-Implementationen transparent um die Kopplungsfähigkeit mit anderen Maschinen; die Kopplung wird dabei durch eigene, systemnahe Funktionen realisiert.

Die in Abschnitt 2.2 aufgeführten Vorteile des Metacomputings können nur dann im Produktionsbetrieb genutzt werden, wenn eine Anwendung im *batch*-Betrieb auf mehreren Rechnern zeitgleich gestartet werden kann. In der Praxis ist das *job-scheduling* schon für einen einzelnen Rechner ein schwieriges Problem [131], da die verschiedenen, sich teilweise widersprechenden Ansprüche wie hohe Auslastung der Rechner, Fairneß, hoher Durchsatz und kurze Verweilzeit bis zum Start einzelner Anwendungen erfüllt werden müssen. Durch die notwendige Kopplung der verschiedenen *scheduling*-Systeme steigt die Komplexität der Berechnung eines *schedules* weiter an. Dieser wichtige Bereich wird z.Zt. im Projekt UNICORE [60, 42] bearbeitet, in dem versucht wird, dem Benutzer eine einheitliche Sicht eines verteilten Systems zu geben und auch eine gemeinsame Abrechnungsbasis für die benutzte Rechenzeit oder Netznutzung zu ermöglichen. Letzteres ist wichtig, da beim Einsatz eines Metacomputers im allgemeinen unterschiedliche Institutionen miteinander kooperieren müssen, die eigenständig die vorhandenen Ressourcen verwalten. Durch diese enge Zusammenarbeit wird aber auch die Datensicherheit bedeutsam: Daten der einzelnen Benutzer, insbesondere Paßwörter, die zur Authentifizierung benutzt werden, müssen über das WAN verschickt werden. Hier müssen geeignete Authentifizierungsverfahren eingesetzt werden, die die Sicherheit der einzelnen Rechnersysteme gewährleisten, gleichzeitig dem Benutzer aber weiterhin einen komfortablen Zugang zu den einzelnen Systemen ermöglichen. Diese Problematik wird von Sander [118] diskutiert, dessen Ergebnisse ebenfalls in UNICORE verwendet werden sollen.

Als letztes und entscheidendes Problem bleibt die Optimierung von Metacomputing-Anwendungen. Dies beginnt mit der unzureichenden Unterstützung durch Programmierwerkzeuge. Als Beispiel betrachte man das *debugging* einer Anwendung. *Public domain debugger*, wie z.B. GDB [124] von der *Free Software Foundation*, sind zwar für eine Vielzahl von Plattformen verfügbar, unterstützen aber Parallelrechner nur sehr rudimentär, da für jeden Prozeß ein eigener *debugger* gestartet werden muß, was bei mehreren hundert Prozessen nicht praktikabel ist. Kommerzielle Programme wie z.B. Totalview [19] sind oftmals nicht auf allen Plattformen vorhanden und unterstützen im allgemeinen auch nicht die Nutzung heterogener Plattformen[3]. Auch Werkzeuge zur Performance-Optimierung, wie z.B. PARAGRAPH [68], APPRENTICE [20] oder PERFVIEW [18] sind nicht für den Einsatz in Metacomputing-Umgebungen gedacht, da sie meist spezielle Eigenschaften eines Rechnersystems oder eines Prozessors wie z.B. Hardware-Operationszähler [103] nutzen. Eine Ausnahme ist das Werkzeug VAMPIR (*Visualization and Analysis of MPI Resources*) [100], das im Rahmen des Regionalen Testbed NRW (vgl. Kapitel 3) für heterogene Systeme erweitert wurde [62].

Neben der mangelnden Unterstützung durch Software sind die relativ niedrige Bandbreite und hohe Latenz eines WANs einschränkende Faktoren, die bei der Programment-

[3]TOTALVIEW unterstützt zwar verteiltes *debugging*, allerdings mit folgender Einschränkung: „*Distributed debugging currently requires that all machines have the same machine architecture and operating system.*“ [32], wodurch ein Einsatz auf einem Metacomputer nicht möglich ist.

wicklung berücksichtigt werden müssen. Die Kommunikationsgeschwindigkeit eines WANs liegt mittlerweile in der Größenordnung eines Parallelrechners: Die Intel Paragon XP/S 10 des Forschungszentrums Jülich hat eine maximale Bandbreite von ca. 90 MByte/s, bei einer 622–MBit/s–ATM Leitung ist eine maximale theoretische Bandbreite von von ca. 520 MBit/s (65 MByte/s) möglich, wenn man den Verlust von ca. 17% durch die Protokolle auf den verschiedenen Netzwerkschichten berücksichtigt [10]. Moderne Rechner wie die Cray T3E bieten eine Bandbreite von bis zu 308 MByte/s [116], was um den Faktor fünf schneller ist als die ATM-Verbindung. In der Praxis ist dieser Unterschied noch viel größer: Bei einem massiv-parallelen System kann bei einer geeigneten Zuordnung der Prozesse zu den Prozessoren jeder Prozessor gleichzeitig Daten mit dieser Rate zu jeweils einem anderen Prozessor senden, so daß man bei einer Cray T3E mit 512 Prozessoren eine Gesamtbandbreite von über 157 GByte/s erreicht[4]. Dies macht den Unterschied zwischen der externen und internen Kommunikation deutlich: Ein WAN ist nicht nur langsamer als das interne Verbindungsnetzwerk, sondern alle Daten, die zwischen den Rechnern ausgetauscht werden, müssen über diese eine externe Leitung übertragen werden. Weiter ist aufgrund der großen Entfernung zwischen den Rechnern in einem WAN die Latenz deutlich höher: Während die Latenz einer Cray T3E bei ca. 16 Mikrosekunden liegt, ist bei einer externen Verbindung eine Latenz im Bereich von einigen Millisekunden die Regel. Die Zeit für die Datenübertragung macht sich schon bei einem homogenen System nachteilig bemerkbar, da sie einen zusätzlichen Aufwand für ein paralleles Programm bedeutet. Der Anstieg der Latenz um mehr als zwei Zehnerpotenzen und die gleichzeitige Reduktion der Bandbreite resultiert in einem signifikanten Verlust an Rechenleistung. Hier müssen neue Konzepte gefunden werden, damit eine möglichst große Klasse von Anwendungen effizient auf einem Metacomputer einsetzbar ist.

Der entscheidende Vorteil von Metacomputing liegt, wie in Abschnitt 2.2 beschrieben, in der Heterogenität der beteiligten Systeme. Dadurch wird es aber noch wichtiger als in einem homogenen System, eine gleichmäßige Auslastung der einzelnen Prozessoren zu erreichen, da sonst die Rechenkapazität gerade der schnelleren Rechner verloren gehen kann, wenn diese auf die langsameren Rechner warten müssen. Dies kann sogar dazu führen, daß die Gesamtbearbeitungszeit einer Anwendung steigt, wenn langsamere Prozessoren hinzugefügt werden (vgl. Abschnitt 3.3). Während es bei homogenen Systemen ausreicht, daß jeder Prozessor den gleichen Anteil an den Berechnungen zugewiesen bekommt, gilt dies für heterogene Systeme nicht. Auch hängt die tatsächliche Leistung der einzelnen Rechner stark von der Struktur der Anwendung ab, so daß ein Vergleich weniger systemspezifischer Parameter wie z.B. Mflops (Millionen Fließkommaoperationen pro Sekunde), Hauptspeichergröße oder Geschwindigkeit des internen Netzwerkes nicht ausreicht. Dementsprechend ist es für einen Metacomputer komplizierter, eine gleichmäßige Verteilung der Rechenlast auf die einzelnen Prozessoren zu finden. Verhält sich die Anwendung adaptiv, d.h. ändert sich die Anzahl der Rechenoperationen in den einzelnen Programmteilen im Laufe der Berechnung, wird es noch schwieriger, eine „gute“ Verteilung der Aufgaben auf alle Prozessoren zu finden und beizubehalten. Hier sind Lastausgleichs-

[4]Die Cray T3E kann aufgrund des dreidimensionalen bidirektionalen Torusnetzwerkes gleichzeitig zu sechs verschiedenen Prozessoren Daten übertragen, wodurch sich eine Bandbreite von 1.8 GByte/s pro Prozessor und eine theoretische Gesamtbandbreite von über 940 GByte/s ergibt.

konzepte erforderlich, die auch die Heterogenität des Rechnersystems berücksichtigen.

2.5 Einordnung dieser Arbeit

Während in der Heterogenität einer der großen Vorteile des Metacomputings liegt, führt diese aber in fast allen Bereichen, von der Hardware bis hin zur Optimierung der Programme, zu neuen Problemen.

Diese Arbeit befaßt sich mit der Optimierung von Anwendungen für Metacomputer, also der obersten Ebene in Abbildung 2. Es wird ein paralleles Programmiermodell vorgestellt, das für eine Vielzahl von Anwendungen und für verschiedene Algorithmen nicht nur eine Optimierung der Kommunikation durchführt, sondern auch den statischen und dynamischen Lastausgleich unter Berücksichtigung der Heterogenität des zugrundeliegenden Metacomputers ermöglicht, ohne daß weitgehende Änderungen an der Anwendung vorgenommen werden müssen. Damit ist eine der entscheidenden Voraussetzungen für den praktischen Einsatz von Metacomputing erfüllt.

Als Motivation für die vorgenommenen Optimierungen wird im nächsten Kapitel zunächst exemplarisch untersucht, welche Faktoren für die Leistung einer Anwendung auf einem Metacomputer wichtig sind, und wo dementsprechend Handlungsbedarf besteht, um eine höhere Effizienz zu erzielen.

3 Fallstudien

Die bisherige Diskussion der Vorteile und Schwierigkeiten von Metacomputing waren nicht durch praktische Daten gestützt. In diesem Kapitel wird anhand von Netzleistungsmessungen und der Analyse zweier Anwendungen verdeutlicht, wie sich die Situation in einem realen Umfeld dargestellt hat. Dieses Umfeld bestand aus dem „Regionalen Testbed NRW“ [28], einem vom BMBF und DFN-Verein geförderten Projekt, in dem „der Zugriff auf verteilte Hochleistungs-Rechnerkapazitäten erprobt und innovative Anwendungen durchgeführt werden“ sollten [29].

Hierfür wurde zwischen April 1995 und August 1996 vom DFN-Verein eine dedizierte 34-MBit/s-ATM-Leitung zur Verfügung gestellt, die

- das Forschungszentrum Jülich (FZJ),
- das Forschungszentrum Informationstechnik (GMD),
- die Rheinisch-Westfälische Technische Hochschule Aachen,
- die Universität zu Köln,
- die Universität Bonn und
- die Deutsche Luft- und Raumfahrtgesellschaft in Bonn (DLR)

miteinander verband. Die Netztopologie ist in Abbildung 3 dargestellt. Die hier vorgestell-

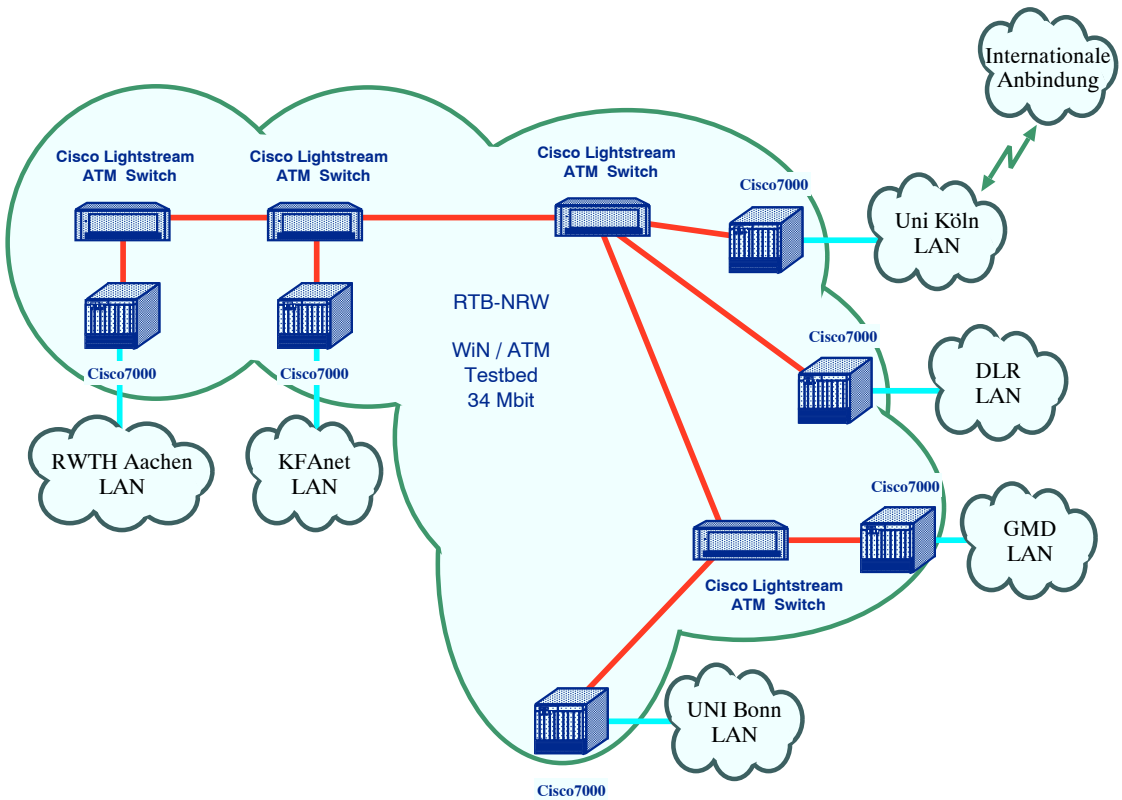

Abbildung 3: Die Netzwerktopologie des RTB NRW [102].

ten Untersuchungen und Messungen wurden im Rahmen des Teilprojektes Z2, „Höchstleistungsrechenzentrum: Verteilter massiv-paralleler Rechner“ [62, 142, 43] vorgenommen. Ziel des Teilprojektes war der Pilotbetrieb einer massiv-parallelen Anwendung über ein Hochleistungs-Datennetz. Dazu wurden die Intel Paragon XP/S 10 des FZJs und die IBM SP2 der GMD zu einem Metacomputer verbunden, auf dem eine Anwendung verteilt berechnet werden konnte. Tabelle 1 stellt die Intel Paragon in Jülich der IBM SP2 der GMD gegenüber. Obwohl beide Rechner zur Klasse der massiv-parallelen Systeme gehören, zei-

	XP/S 10	SP2
Anzahl CPUs	140	37
Hauptspeicher pro CPU [MBytes]	32	128
Rechenleistung pro CPU [Mflops]	75	250
Interne Bandbreite [MByte/s]	90	34
Interne Latenz [Mikrosekunden]	38	45
Topologie	2D-Mesh	Crossbar
Netzwerk-Schnittstellen	Ethernet, FDDI, HiPPI	Ethernet, HiPPI, ATM

Tabelle 1: Technische Daten der Intel Paragon XP/S 10 und der IBM SP2.

gen ihre technischen Daten sehr unterschiedliche Eigenschaften. Während sich die Paragon durch ihr gutes internes Kommunikationsverhalten, d.h. kleine Latenz und hohe Bandbreite auszeichnet, liegen die Vorteile der SP2 in ihrem großen Speicher und der relativ hohen Rechenleistung der einzelnen Prozessoren. Zur Kommunikation zwischen den beiden Systemen wurde die Bibliothek PARMACS (*Parallel Macros*) [13] der Firma Pallas GmbH benutzt. Damit PARMACS im RTB NRW eingesetzt werden konnte, erweiterte Pallas die Bibliothek um die Eigenschaften der Multiprotokollkommunikation und Datenkonvertierung. Hierdurch war eine für die Anwendung transparente Kopplung der beiden Rechner Paragon XP/S 10 und SP2 möglich.

Im folgenden werden drei wesentliche Bestandteile des Teilprojektes Z2 vorgestellt, nämlich die Leistungsmessungen über das WAN und zwei Anwendungen. Eine ausführliche Darstellung der Ergebnisse findet sich im Abschlußbericht des Projektes [62].

3.1 Leistungsmessung im RTB NRW

Die Intel Paragon war zum Zeitpunkt des Projektbeginns schon ein relativ alter Rechner, für den noch keine ATM-Schnittstelle zur Verfügung stand. Zusammen mit der Tatsache, daß Intel begonnen hatte, sich aus dem Bereich des Hochleistungsrechnens zurückzuziehen, war es aus ökonomischer Sicht für Intel nicht mehr lohnenswert, noch ATM-Schnittstellen für diesen Rechner zu entwickeln. Deshalb erfolgte die Anbindung mittels eines NSC-*routers* mit einer HiPPI-Verbindung zur Intel Paragon auf der einen und einem FDDI-Anschluß an das FZJ-interne LAN auf der anderen Seite. Das FZJ-LAN wurde über einen *router* Cisco 7000 und einen *switch* Cisco Lightstream 100 mit dem RTB NRW

verbunden. Abbildung 4 zeigt eine schematische Darstellung der Netzwerktopologie im Forschungszentrum Jülich.

Eine für das Metacomputing wichtige ATM-Eigenschaft, die in bisherigen LAN- und WAN-Kommunikationstechnologien in dieser Form nicht zur Verfügung stand, ist die Option *Quality of Service* (QoS), die es erlaubt, für eine Anwendung einen festen Datendurchsatz zu reservieren [81]. Störungen durch weitere Netzbenutzer sind aufgrund der Bandbreitenreservierung dann nicht mehr möglich, und die angeforderte Bandbreite kann voll von der Anwendung benutzt werden. Wie oben beschrieben, verfügte die verwendete Infrastruktur nicht über eine integrierte ATM-Schnittstelle, so daß weiterhin *classical IP* über ATM benutzt werden mußte und die Option *Quality of Service* nicht zur Verfügung stand. Da die Leitung zwischen der GMD und dem FZJ auch von der RWTH Aachen und der Universität zu Köln im Rahmen anderer Teilprojekte genutzt wurde, stand de facto keine dedizierte Verbindung zur Verfügung.

Wichtig für Metacomputing sind die Bandbreite und die Latenz, die für eine Anwendung unter Nutzung einer Kommunikationsbibliothek tatsächlich zur Verfügung steht bzw. berücksichtigt werden muß. Theoretisch war mit einer maximalen Bandbreite von 34 MBit/s und einer Latenz von 3.6 ms zu rechnen. Letztere wurde abgeschätzt durch die Signallaufzeit zwischen dem FZJ und der GMD und der Anzahl der zwischengeschalteten *router*, die jeweils eine Verzögerung von ca. 1 ms hinzufügen [102]. In der Praxis konnte für die Anwendung allerdings nur eine Bandbreite von ca. 2 MBit/s bei einer Latenz von 30 ms gemessen werden. Hier spielte eine Vielzahl von Faktoren eine Rolle:

- Da das TCP/IP-Protokoll über die ATM-Verbindung genutzt wurde, mußten zusätzlich zu den Nutzdaten, die die Anwendung sendet, und den Protokollinformationen für ATM auch noch die Kontrollinformationen des TCP/IP-Protokolls übertragen werden. Bei einer *maximum transmission unit* (MTU) von 9180 Bytes benötigen diese Kontrollinformationen ca. 17% [10]. Somit ergab sich eine maximale Bandbreite von 28.2 MBit/s. Auch hat sich herausgestellt, daß die Parameter für eine TCP/IP-Verbindung oftmals vom Hersteller nicht optimal für höchste Leistung ausgelegt sind[5] und eine manuelle Optimierung notwendig ist. Messungen zeigten, daß sich die maximale Bandbreite nicht notwendigerweise bei möglichst großen Blöcken ergibt, sondern daß schon kleinere Blockgrößen dafür ausreichen können [10]. Dies hängt stark von der Implementierung des TCP/IP-Protokolls und der zugrundeliegenden Hardware ab. Zum Beispiel kann eine Blockgröße, die genau einer Speicherseite entspricht, effizient gehandhabt werden und zu einer hohen Bandbreite führen.

- In der Praxis kam es auf dem Netz zu Paket-Kollisionen bzw. Pufferüberläufen, so daß Datenpakete mehrfach übertragen werden mußten. Dies hängt stark von der Auslastung der Leitung ab [130].

- Die in der Praxis notwendigen *router* (und in geringerem Umfang auch die *switches*)

[5] Für eine hohe Bandbreite wird ein großer interner Pufferplatz für die einzelnen *sockets* benötigt. Während dies problemlos möglich ist, wenn nur wenige *socket*-Verbindungen gleichzeitig benutzt werden, wird der verfügbare Hauptspeicher bei vielen gleichzeitigen Verbindungen schnell zu einem Engpaß.

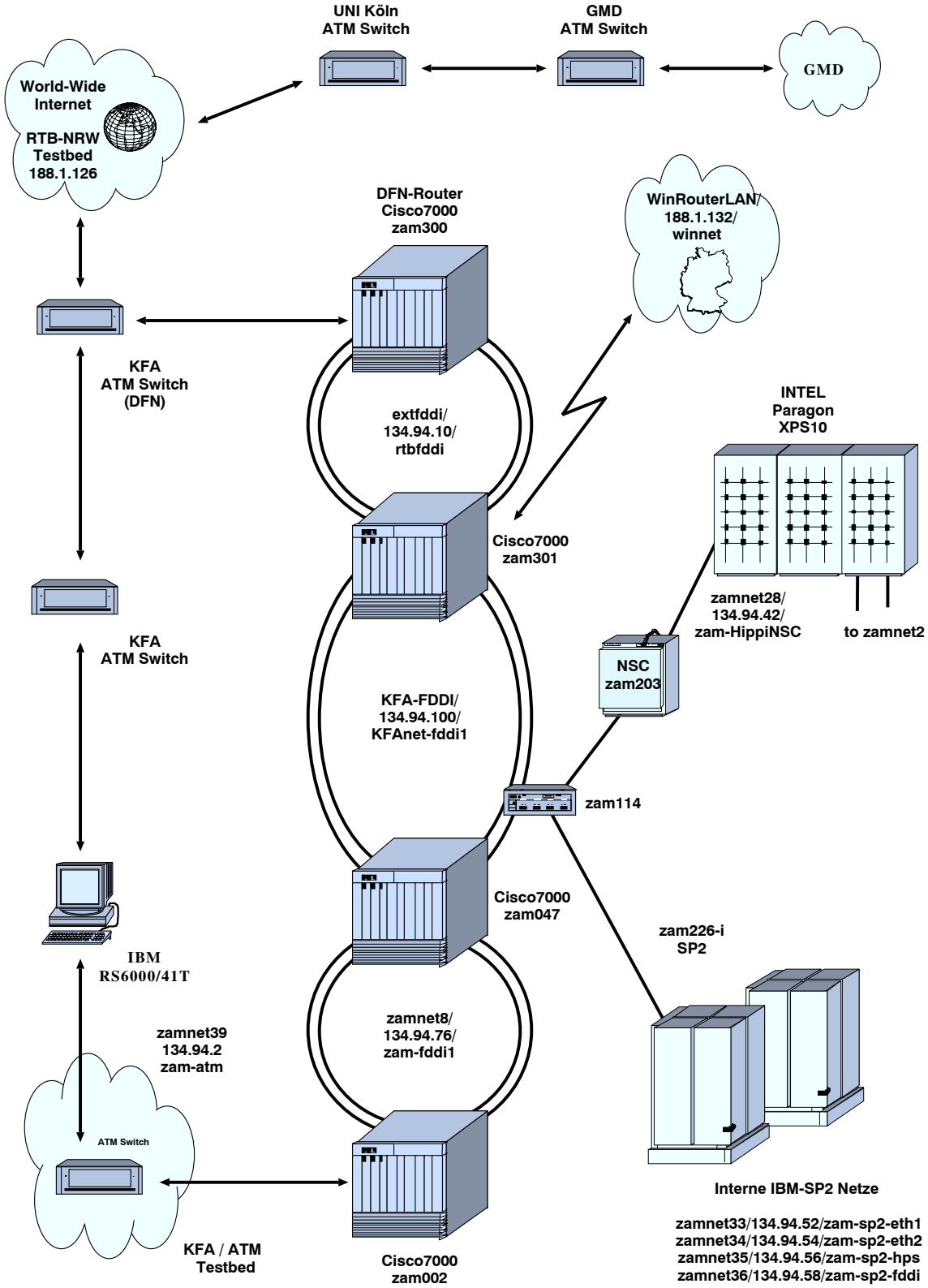

Abbildung 4: Schematische Darstellung der Netzwerktopologie im Forschungszentrum Jülich [62].

vergrößern die Latenz um ca. eine Millisekunde [102].

- Die TCP/IP-Implementierung der Intel Paragon bietet nur eine recht geringe Leistung. Schon bei einer TCP/IP-Verbindung innerhalb der Paragon ergaben sich bei Messungen eine Latenz von 7 ms und eine Bandbreite von 320 KByte/s.
- Im Gegensatz zu einer reinen Leistungsmessung, bei der nur die Menge der übertragenen Daten berücksichtigt wird, ist es für eine Anwendung wichtig, daß die Daten auch falls notwendig konvertiert werden, damit sie auf dem Zielsystem richtig interpretiert werden können. Zwar verwenden sowohl die Intel Paragon XP/S 10 als auch die IBM SP2 den IEEE-Standard, doch haben sie ein unterschiedliches Datenformat (*Little Endian* auf der SP2 bzw. *Big Endian* auf der Paragon [41]). Dies machte die Datenkonvertierung für jeden Datentyp notwendig.
- Die Nutzung von PARMACS, die die Kommunikation zwischen den Prozessen unabhängig davon ermöglicht, ob sie auf der gleichen Maschine bearbeitet werden oder nicht, vergrößert durch die zusätzliche Softwareschicht weiter die Latenz.

Der Einfluß der einzelnen Faktoren wurde in einer ausführlichen Testreihe ermittelt. Abbildung 5 zeigt die sich ergebende Bandbreite in Abhängigkeit von der zu übertragenden Datenmenge bei verschiedenen Szenarien; Tabelle 2 faßt die Latenz und maximale Bandbreite zusammen.

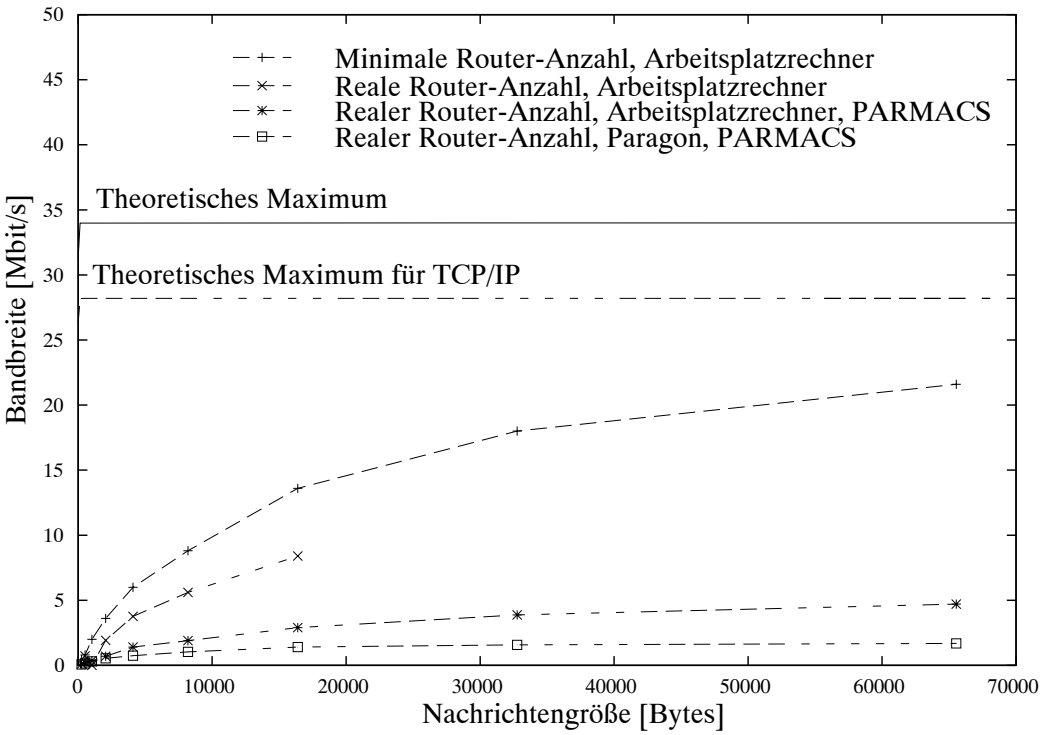

Abbildung 5: Maximale Bandbreite in Abhängigkeit von der Nachrichtengröße für die verschiedenen Szenarien.

Beschreibung	Latenz [ms]	maximale Bandbreite [MBit/s]
Theoretisch bester Wert bei minimaler *router*-Anzahl und Vernachlässigung des TCP/IP-Protokoll-*overheads*	$\leq$2.5	34
Theoretisch bester Wert bei minimaler *router*-Anzahl und Nutzung des TCP/IP-Protokolls über ATM	$\leq$ 2.5	28.2
Theoretisch bester Wert bei normaler *router*-Anzahl und Nutzung des TCP/IP-Protokolls über ATM	$\leq$3.5	28.2
Praktische Messung mit minimaler *router*-Anzahl zwischen dedizierten Arbeitsplatzrechnern	3.6	21.6
Praktische Messung mit normaler *router*-Anzahl zwischen dedizierten Arbeitsplatzrechnern	4.5	8
Praktische Messung mit PARMACS bei normaler *router*-Anzahl zwischen dedizierten Arbeitsplatzrechnern	14.9	4.7
Praktische Messung mit PARMACS bei normaler *router*-Anzahl zwischen der Intel Paragon und der IBM SP2	30	2.0

Tabelle 2: Latenz und maximale Bandbreite bei den verschiedenen Szenarien.

Die einzelnen Szenarien sind dabei:

Theoretisches Maximum: Diese Kurve resultiert aus dem vereinfachten Zusammenhang zwischen der Übertragungszeit t, der Datenmenge n, der Latenz l und der Bandbreite b, wie er z.B. in [44] dargestellt ist:

$$t(n) = l + n/b \tag{1}$$

Hierbei wird nur die Zeit für die reine Übertragung im Netz berücksichtigt, der Aufwand für die Implementierung der verschiedenen Protokolle, die ggf. vorgenommene Blockbildung auf den unterschiedlichen Netzwerkschichten etc. (vgl. [128]) gehen in Gleichung (1) nicht mit ein. Hieraus ermittelt sich die effektive Übertragungsbandbreite $b(n)$, also die Bandbreite unter Berücksichtigung der Latenz, in Abhängigkeit von der zu übermittelnden Datenmenge n zu

$$b(n) = \frac{n}{t(n)} = \frac{bn}{bl + n} \tag{2}$$

Die Latenz wurde durch die Verzögerung durch zwei *router* und durch die Signallaufzeit zwischen FZJ und GMD (ca. 70 km) zu $l \approx 2 * 1\mu s + 0.5\mu s = 2.5\mu s$ abgeschätzt, als Bandbreite wurden das Maximum, also $b = 34$ MBit/s angenommen. Zwei *router* ist hierbei, wie Abbildung 3 zeigt, die minimale Anzahl von *router*, die zwischen dem FZJ und der GMD im Rahmen des RTB NRWs eingesetzt werden mußten.

Theoretisches Maximum für TCP/IP: Hier wurden wieder die Gleichungen (1) und (2) genutzt. Es wurde allerdings berücksichtigt, daß TCP/IP über ATM genutzt

wurde, wodurch nur noch ca. 83% der Bandbreite für die Nutzdaten zur Verfügung standen [10], also $b = 28.2$ MBit/s

Minimale *router*-Anzahl, Arbeitsplatzrechner: Hier wurde eine Messung zwischen einem Arbeitsplatzrechner am FZJ und der IBM SP2 der GMD durchgeführt, wobei in einer Testkonfiguration ein Arbeitsplatzrechner in Jülich logisch direkt in das ATM-Netz der GMD geschaltet wurde [102]. Bei der normalen Konfiguration, also der Nutzung von drei *routern* wurde eine MTU von 512 Bytes zwischen den Rechnern genutzt, da die Rechner nicht im gleichen Subnetz lagen. Bei Rechnern innerhalb eines Subnetzes hingegen beträgt die MTU 9180 Bytes. Dies bedeutet, daß bei der normalen Konfiguration Bandbreite dadurch verloren geht, daß die einzelnen Blöcke (der maximalem Größe 9180 Bytes) in kleine Blöcke der Größe 512 Bytes zerlegt werden müssen, was zu einer Reduktion der Bandbreite führt. Durch diese Testkonfiguration mit nur zwei *routern* hingegen reduziert sich die Latenz und die Bandbreitenreduktion durch die Aufteilung der Blöcke entfällt [102]. Wie erwartet ergaben sich in diesem Testszenario mit 21.6 MByte/s die größte Bandbreite.

Reale *router*-Anzahl, Arbeitsplatzrechner: Auch hier wurde wiederum eine Messung zwischen einem Arbeitsplatzrechner am FZJ und der IBM SP2 der GMD durchgeführt, jetzt allerdings bei der normalen *router*-Anzahl. Im Vergleich zum vorherigen Szenario ergab sich nicht nur eine höhere Latenz (vgl. auch Tabelle 2), sondern durch die Aufspaltung der Blöcke auch eine geringere Bandbreite.

Reale *router*-Anzahl, Arbeitsplatzrechner, PARMACS : Hier wurde die Kommunikationsbibliothek PARMACS auf einem Arbeitsplatzrechner am FZJ zusammen mit der SP2 der GMD genutzt. Da PARMACS innerhalb eines einzelnen Rechners eine gute Leistung bietet [62], macht sich hier der Einfluß der Datenkonvertierung bemerkbar, der aber aufgrund des unterschiedlichen Datenformats auf der Paragon XP/S 10 und der SP2 nicht vermieden werden konnte.

Reale *router*-Anzahl, Paragon, PARMACS Hier wurde die Paragon XP/S 10 anstelle eines Arbeitsplatzrechners benutzt. Die Kommunikation fand wiederum mittels PARMACS statt. Dabei zeigte sich die relativ geringe Leistung der TCP/IP-Implementierung des Paragon, die sich in erster Linie durch einen deutlichen Anstieg der Latenz auswirkte.

Eine Übersicht über den Einfluß, den die unterschiedlichen Faktoren auf die Latenz bzw. die maximale Bandbreite haben, ist in Tabelle 3 dargestellt. Zusammenfassend ergibt sich, daß die geringe TCP/IP-Leistung der Intel Paragon und die Nutzung von PARMACS den größten Einfluß auf die Latenz hatten. Im Rahmen des Projektes war keine Leistungsverbesserung der Intel Paragon möglich, da hierfür ein erheblicher Hardware-Aufwand notwendig gewesen wäre (vgl. z.B. [73]). Auch mußte immer eine Datenkonvertierung zwischen der Paragon und der SP2 durchgeführt werden, da sich die Darstellung aller Datentypen unterschied. Der dadurch verursachte Leistungsverlust war also unumgänglich. Wichtig ist auch die geringe MTU bei der Nutzung eines WANs. Die Technik des *Path MTU Discovery* [93] hilft hier, die tatsächliche minimale MTU zu bestimmen, um den dadurch verursachten Leistungsverlust zu reduzieren.

Parameter	Anstieg der Latenz [ms]	Reduktion der Bandbreite [MBit/s]
TCP/IP über ATM statt *native* ATM, theoretisch	0.0	5.8 (17%)
Zusätzlicher *router*, theoretisch	1.0	0 (0%)
Verlust in der Praxis im Vergleich zur Theorie bei minimaler *router*-Anzahl und der Nutzung dedizierter Arbeitsplatzrechner	1.1	6.6 (23%)
Verlust in der Praxis im Vergleich zur Theorie bei normaler *router*-Anzahl	1.0	20.2 (72%)
Verlust in der Praxis durch zusätzlichen *router*	0.9	13.6 (63%)
PARMACS mit Datenkonvertierung statt TCP/IP	10.4	3.3 (41%)
Intel Paragon statt Arbeitsplatzrechner	15.1	2.7 (57%)

Tabelle 3: Einfluß der einzelnen Größen auf den Anstieg der Latenz bzw. den Verlust an Bandbreite.

Von den 34 MBit/s standen also maximal ca. 21.6 MBit/s zur Verfügung – und selbst das nur in einer Testumgebung, in der die Rechner in das gleiche Subnetz geschaltet wurden und Arbeitsplatzrechner mit einer hohen Eingabe- bzw. Ausgabeleistung eingesetzt wurden. Für die eigentliche Anwendung bei der Nutzung der Parallelrechner blieb nur eine Bandbreite von ca. 2.0 MBit/s bei einer Latenz von 30 ms. Selbst wenn keine Datenkonvertierung notwendig ist, geht ca. 75% der Bandbreite durch unterschiedliche Protokolle, zwischengeschaltete *router* etc. verloren. Diese Ergebnisse sind auch für aktuelle Metacomputing-Projekte relevant, denn das TCP/IP-Protokoll wird aufgrund der weiten Verbreitung immer noch häufig für Rechnerkopplungen eingesetzt, siehe z.B. [33, 92].

Der Einfluß, den nun die relativ große Latenz und geringe Bandbreite auf praxisrelevante Anwendungen haben, wird in den nächsten beiden Abschnitten untersucht.

3.2 Inhomogene Anwendung auf einem Metacomputer

Als Anwendung für einen Metacomputer wurde das Programm BD vom Institut für Festkörperforschung des Forschungszentrums Jülich untersucht [146], das die Greensche Funktion für periodische Kristalle berechnet. Das Programm lag bereits in einer parallelisierten Form für die Intel Paragon vor. Es berechnet die Werte der Greenschen Funktion für 16 Elektronenniveaus gleichzeitig, bevor in einer abschließenden globalen Operation ein Zwischenergebnis bestimmt wird. Dieses Vorgehen wird mehrfach iteriert, bis ein Ergebnis mit ausreichender Genauigkeit vorliegt. Charakteristisch für den Code ist, daß das letzte Elektronenniveau mit einer größeren Genauigkeit berechnet werden muß, so daß dieser Knoten deutlich mehr Rechenzeit als die anderen benötigte. Dies kann mit Hilfe des Visualisierungs-Tool VAMPIR [100] veranschaulicht werden. Der Anhang A enthält einige Bilder von VAMPIR, die die hier getroffenen Aussagen bestätigen.

Grundlage für die folgende Analyse war ein Programmlauf von BD, bei dem zwei Iterationen ausgeführt wurden. Schon bei dieser relativ kurzen Berechnung machte sich der Unterschied in der benötigten Rechenzeit bemerkbar. Abbildung 6 zeigt einen Gesamtüberblick über die Zeit, die die jeweiligen Prozessoren für die Kommunikation und für die eigentliche Berechnung brauchen. Der letzte Prozessor, der das Elektronenniveau mit der größeren Genauigkeit berechnet, benötigt fast die vollständige Bearbeitungszeit nur für die Berechnung. Alle anderen Prozessoren hingegen brauchen nur ca. ein Drittel der Bearbeitungszeit für die Berechnung, die restliche Zeit geht durch Kommunikation verloren. Eine detailliertere VAMPIR-Analyse, vgl. Abbildung 38 im Anhang (Seite 123), bestätigt dies. Durch die globale Operation am Ende eines Berechnungsschrittes sind alle Prozessoren gezwungen, auf den länger rechnenden Prozessor zu warten.

Um eine Leistungssteigerung für diese Anwendung zu erreichen, muß die Berechnung für das letzte Elektronenniveau schneller durchgeführt werden, da sich dadurch die Wartezeit aller anderen Prozessoren reduziert. Hierzu könnte einerseits eine weitergehende und aufwendige Parallelisierung des vorhandenen Programms vorgenommen werden. Alternativ bot sich aber der Einsatz eines Metacomputers an, da dann die Berechnung des letzten Elektronenniveaus auf einem schnelleren Prozessor durchgeführt werden konnten. Wie Tabelle 1 (Seite 20) zeigt, hat ein Knoten der IBM SP2 ungefähr die dreifache Rechenleistung eines Knotens der Paragon XP/S 10. Dementsprechend wurde ein Prozeß auf der SP2 gestartet, der das letzte Elektronenniveau berechnete, während alle anderen Berechnungen auf der Paragon stattfanden. Dabei ergab sich eine Reduktion der Rechenzeit von 181 Sekunden auf 106 Sekunden. Der Überblick über die Verteilung der Gesamtzeit in Abbildung 7 bestätigt die durchgeführte Analyse. Da die Berechnung auf der SP2 sogar schneller durchgeführt wurde als die weniger aufwendigen Berechnung auf den einzelnen Paragon Prozessoren, hat dort die Kommunikationszeit einen höheren Anteil an der Gesamtzeit. Dies ist in der detaillierten VAMPIR–Analyse in Abbildung 39, Seite 124 zu sehen. Zwar steigt durch die Nutzung des langsameren, externen Verbindungsnetzwerkes die Zeit für die Übertragung der Daten zu den Prozessoren, jedoch reduziert sich die Wartezeit, die in den Abbildungen 6 und 7 ebenfalls als Kommunikationszeit angezeigt wird. Abbildung 8 zeigt, wieviel Zeit die Prozessoren jeweils insgesamt für Kommunikation und für Berechnung aufgewendet haben. Auch hier ist die Einsparung, die man durch

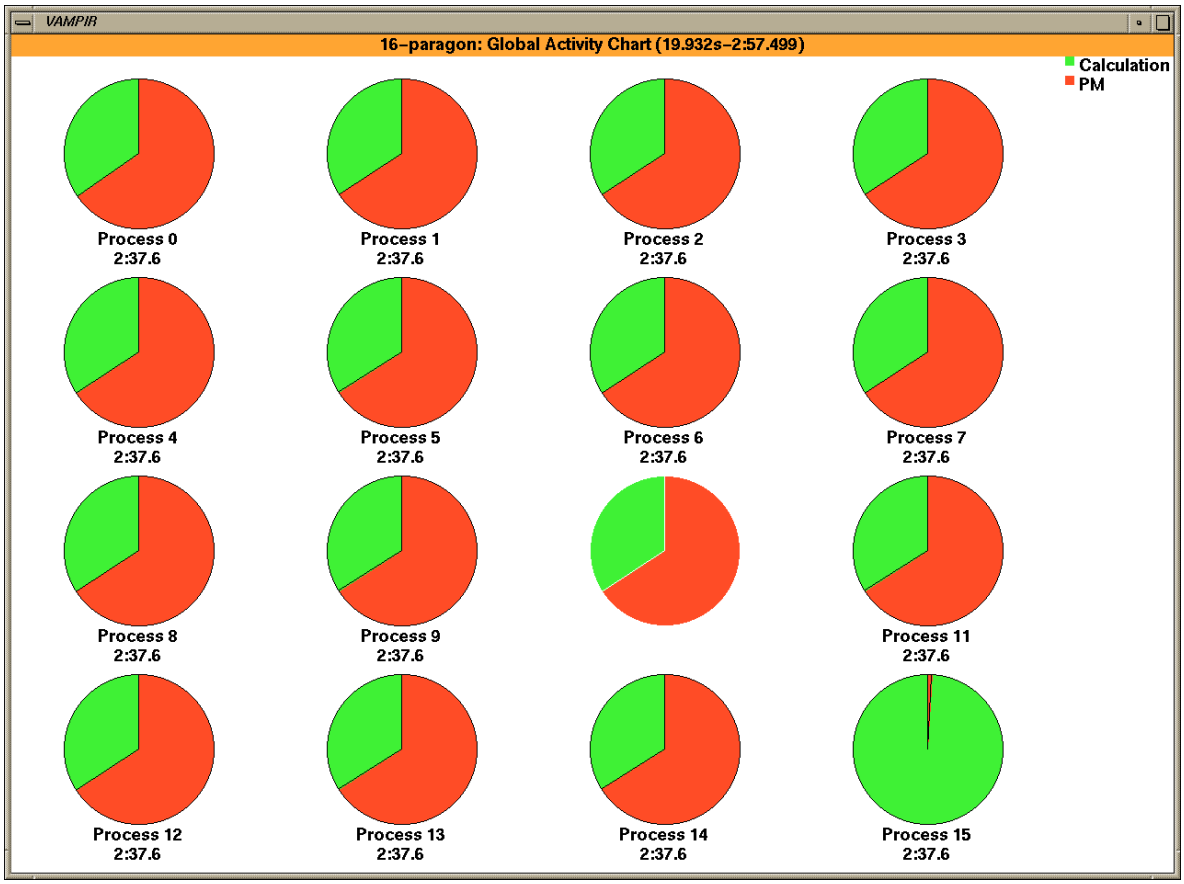

Abbildung 6: Aufteilung der Gesamtzeit auf Rechnung und Kommunikation für die Anwendung BD auf der Paragon.

die Nutzung eines heterogenen Systems erhält, gut zu sehen. Insgesamt ergab sich durch die Nutzung eines heterogenen Systems eine deutliche Reduktion der Bearbeitungszeit.

Zusammenfassend läßt sich sagen, daß ein Lastungleichgewicht von Anwendungen auf homogenen Systemen durch den Einsatz eines Metacomputers verschwinden kann, wenn der Metacomputer die richtige Rechenkapazität für die einzelnen Prozessoren zur Verfügung stellt.

3.3 Homogene Anwendung auf einem Metacomputer

Als zweite Anwendung wurde das Programm TRACE (*Transport of Contaminants in Environmental Systems*) [138] untersucht, eine Anwendung aus dem Bereich der Umweltforschung. Es berechnet den Wasserfluß in porösen heterogenen Medien und bildet somit die Grundlage für die Simulation des Stofftransportes im Boden und Grundwasser. Die Diskretisierung des zugrundeliegenden Systems partieller Differentialgleichungen fand mit Hilfe der Methode der finiten Elemente statt. Die ursprüngliche Version wurde für die Cray

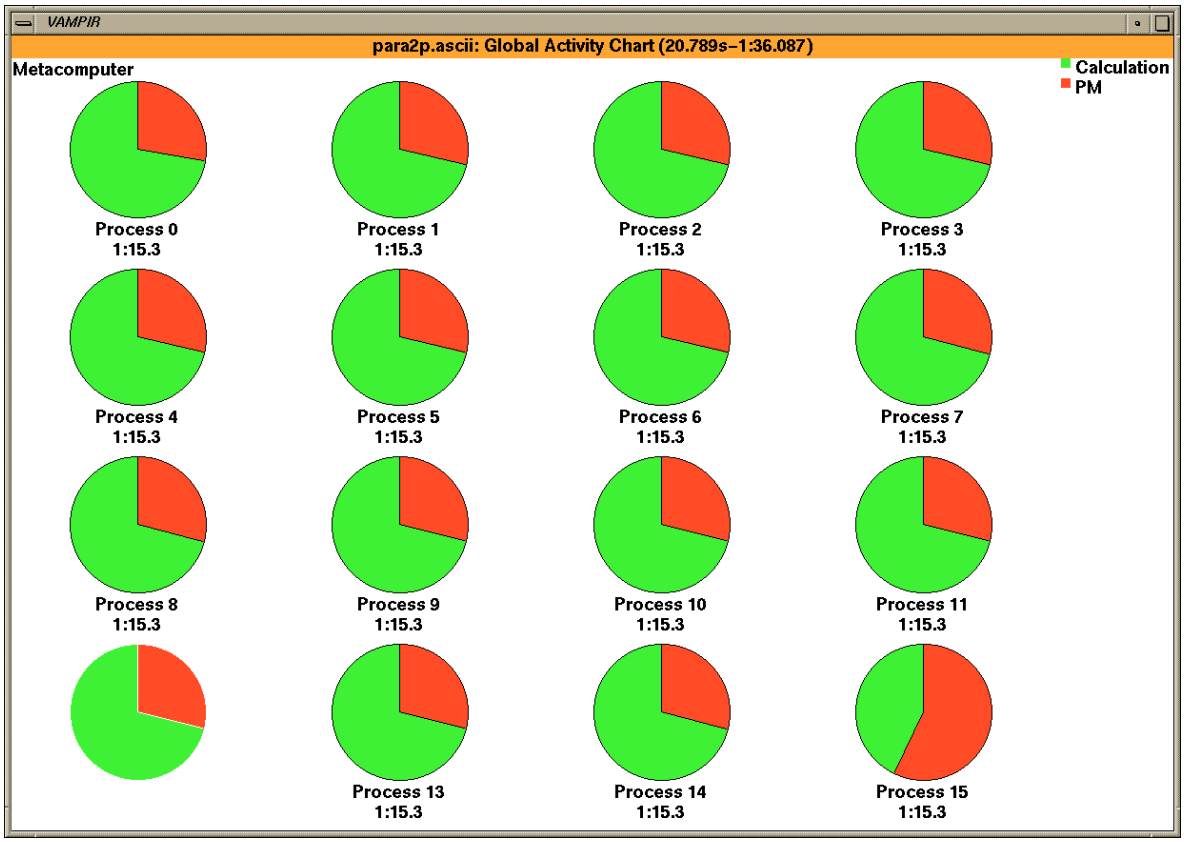

Abbildung 7: Aufteilung der Gesamtzeit auf Rechnung und Kommunikation für die Anwendung BD auf dem Metacomputer.

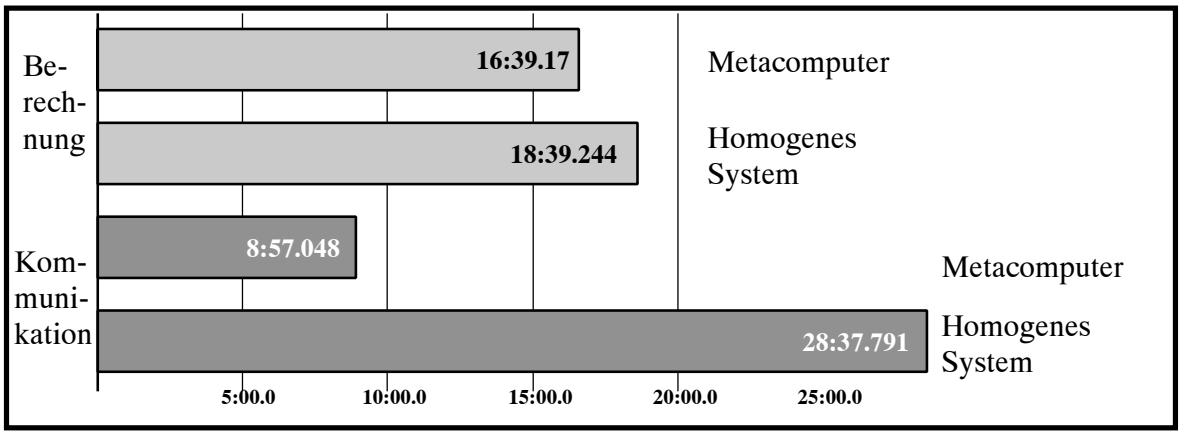

Abbildung 8: Vergleich der Gesamtzeiten für Kommunikation und Berechnung.

X-MP entwickelt. Da es sich jedoch schon bald zeigte, daß aufgrund des geringen Hauptspeichers nur kleine Gebietsgrößen (ca. 5×10^4 Knoten) berechnet werden konnte, wurde schon bald eine parallelisierte Version für die Intel Paragon XP/S 10 erstellt [144]. Die

Parallelisierung fand mittels einer Gebietsaufteilung statt. Das parallelisierte Programm zeigte eine sehr homogene Struktur: jeder Prozessor bekam die gleiche Gebietsgröße zugewiesen, so daß auch jeder Prozessor ungefähr die gleiche Arbeit ausführte. Es lag eine gute Lastbalancierung für dieses eng-gekoppelte System vor.

Im Rahmen des RTB NRWs wurde untersucht, inwiefern eine weitere Leistungssteigerung durch den Einsatz eines Metacomputers möglich war. Erste Ergebnisse wurden mit einem Beispieldatensatz durchgeführt, dem eine Diskretisierung mit insgesamt 31x31x31 Knoten zugrunde lag. Mit dieser Konfiguration wurden 15 Zeitschritte gerechnet. Diese Anzahl war zwar relativ gering, jedoch zeigt das Programm auch hier schon sein typisches Verhalten, so daß die Ergebnisse auch für reale, längere Rechnungen gültig sind. Abbildung 9 stellt die Laufzeiten für homogene Tests auf den einzelnen Rechnern und für das gekoppelte Gesamtsystem dar. Für den Metacomputer wurde jeweils die gleiche Zahl von Prozessoren auf beiden Maschinen benutzt, so daß dort die Angabe „acht Prozessoren“ bedeutet, daß vier Prozessoren der Paragon und vier Prozessoren der SP2 eingesetzt wurden.

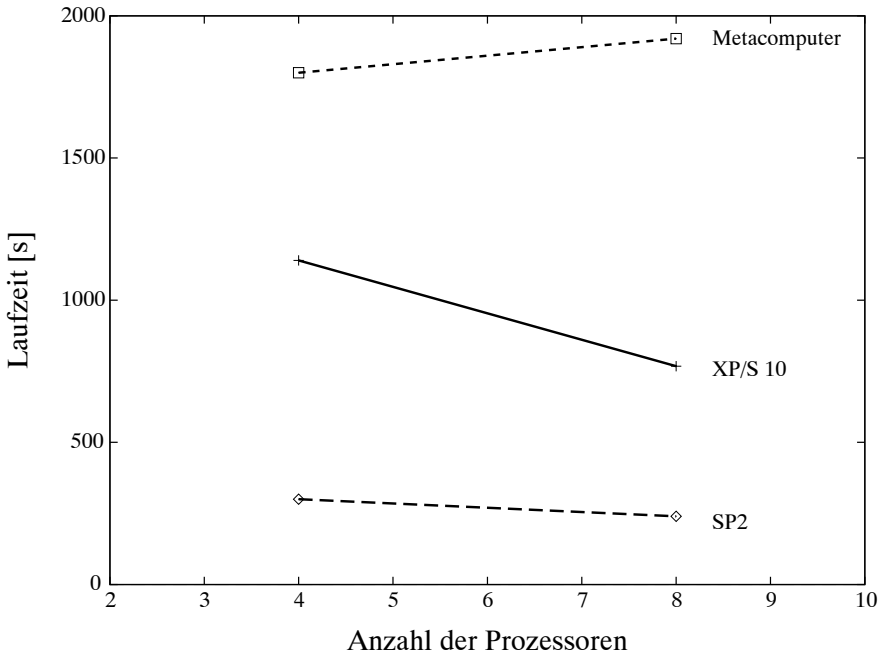

Abbildung 9: Laufzeiten der nicht-optimierten TRACE-Version.

Während die Laufzeiten auf den homogenen Systemen zufriedenstellend waren, ergaben sich für den Metacomputer zwei wichtige Fakten: Die Rechenzeit des verteilten Systems ist größer als die Rechenzeit des langsameren Parallelrechners und durch Hinzunahme weiterer Prozessoren stieg die Rechenzeit weiter an. Vier Faktoren waren hierfür verantwortlich:

- Die Zeit für die initiale Datenverteilung ist von ca. 3 Minuten auf über 20 Minuten angestiegen. Dies ist auf mehr als 60 000 kleinen Nachrichten zurückzuführen, die

während der Initialisierung verschickt werden. Während dies auf einem einzelnen Rechner bei einer Latenz von unter 50 μs einen Zeitaufwand von ca. 3 Sekunden für die gesamte *startup*-Zeit bedeutet, ergibt sich auf dem Metacomputer, bei dem ca. die Hälfte der Nachrichten über das WAN mit einer Latenz von 30 ms übertragen werden muß (vgl. Abschnitt 3.1), als reine *startup*-Zeit ca. 15 Minuten. Im Anhang A zeigt Abbildung 40 (Seite 125) mit Hilfe von VAMPIR einen Ausschnitt aus der Initialisierungsphase, in der Prozessor 0 die einzelnen Daten an alle anderen Prozessoren verteilt. Vor allem der deutliche Unterschied in der Nachrichtenlaufzeit wird hierbei deutlich.

- Da ein Prozessor der IBM SP2 ungefähr die drei- bis vierfache Rechenleistung eines Intel Paragon XP/S 10 Prozessors hat, was bei der Gebietsaufteilung nicht berücksichtigt wurde, bleibt die Rechenzeit erwartungsgemäß deutlich hinter der Rechenzeit für eine Berechnung auf der SP2 zurück, da jeder SP2 Prozessor genausoviel Rechenarbeit bewältigen muß wie ein Prozessor der XP/S 10. Folglich ist es mit dieser gleichmäßigen Datenverteilung nicht möglich, eine schnellere Berechnung als bei alleiniger Nutzung der Paragon zu erhalten. Abbildung 41 im Anhang (Seite 125) zeigt dieses ungleiche Auslastung. Um den Metacomputer effizient einsetzen zu können, muß bei der Gebietsaufteilung die Leistung der einzelnen Rechner berücksichtigt werden, um nicht die Leistung des schnelleren Systems zu verlieren.

- Die hohe Latenz und niedrige Bandbreite für externe Nachrichten spielt bei einer eng-gekoppelten Anwendung eine entscheidende Rolle, da die Anwendung auf eine möglichst kurze Nachrichtenübertragungszeit angewiesen ist. Da die externe effektive Bandbreite um den Faktor 50 kleiner ist als die interne Verbindung (wie in der Matrix *Message Statistics* in Abbildung 40 auf Seite 125 zu sehen ist), ist dies für die auftretende Verlangsamung im Vergleich zur homogenen Version verantwortlich.

- Der zu beobachtende Anstieg der Bearbeitungszeit bei der Hinzunahme weiterer Prozessoren erklärt sich durch die damit verbundene Zunahme der Anzahl der Nachrichten im Gesamtsystem. Während dies bei einem homogenen System keine große Rolle spielt, da alle Nachrichten praktisch gleichzeitig ausgetauscht werden können, wird hier die einzige externe Verbindung zu einem deutlichen Engpaß, da alle externen Nachrichten nacheinander über diese Leitung versendet werden müssen. So müssen im obigen Beispiel bei zwei Paragon und zwei SP2 Prozessoren in einem Zeitschritt acht Nachrichten verschickt werden, bei vier plus vier Prozessoren hingegen schon 20 Nachrichten[6]. Proportional dazu steigt die Zeit für die Kommunikation an.

Um TRACE trotzdem auf einem Metacomputer effizient rechnen zu können, wurden folgende Optimierungen vorgenommen: Zunächst wurden die einzelnen Initialisierungsnachrichten für jeden Prozessor gepuffert. Dadurch konnten anstelle von vielen kleinen Nachrichten wenige große Nachrichten verschickt werden, was gerade aufgrund der hohen

[6]Dies hängt stark von der vorgenommenen Gebietsaufteilung ab. Es wurde hier die in [144] entwickelte automatische Gebietsaufteilung benutzt, die eine Minimierung der Gesamtanzahl der Nachrichten vornimmt, nicht aber die unterschiedliche Kommunikationsleistung berücksichtigt.

Latenz von Bedeutung ist. Dadurch konnte die Initialisierungszeit von über 20 Minuten auf ca. 9 Minuten reduziert werden. Weiterhin wurde die Gebietsaufteilung so vorgenommen, daß die tatsächliche Rechenkapazität der einzelnen Knoten berücksichtigt wurde. Schließlich wurde die gesamte Kommunikationstruktur des Programms gemäß Abbildung 10 optimiert. Auf der linken Seite ist die bisherige Kommunikationsstruktur anhand eines zweidimensionalen Beispiels dargestellt[7]. Es ist deutlich zu sehen, daß die Anzahl der Nachrichten, die über das externe Netzwerk übertragen werden müssen, mit der Anzahl der Prozessoren ansteigt. Da aber nur eine externe Verbindung vorhanden ist, werden diese Nachrichten sequentiell übertragen, was durch die relativ hohe Latenz einen großen Zeitaufwand bedeutet. Deshalb wurde die Kommunikationsstruktur so geändert, daß jeweils ein Prozessor die Punkte zur Berechnung zugeordnet bekam, die an der Schnittstelle zu der anderen Maschine liegen, vgl. rechte Seite von Abbildung 10. Dadurch muß jeweils nur ein Prozessor Daten extern übertragen, und es wird nur eine große Nachricht statt vieler kleiner Nachrichten verschickt. Insbesondere diese letzte Änderung erforderte einen

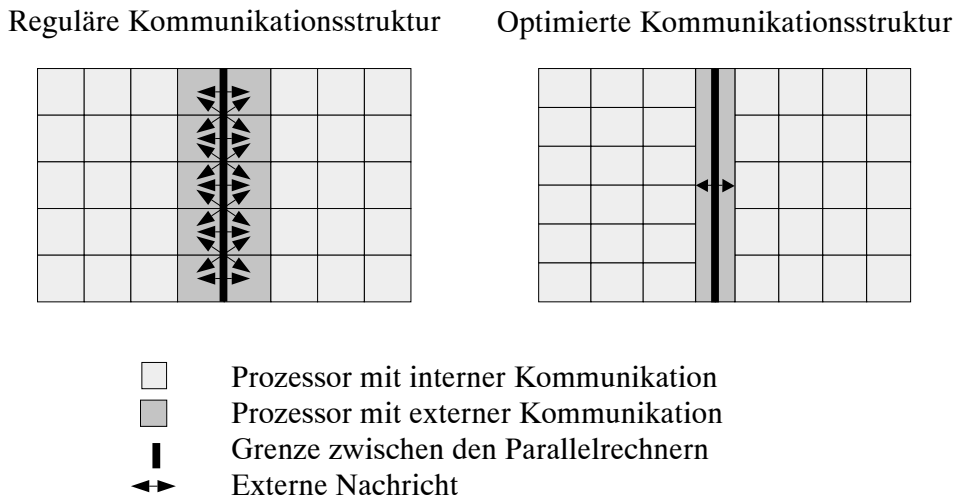

Abbildung 10: Angepaßte Kommunikationsstruktur von TRACE.

hohen Programmieraufwand, da die gesamte Kommunikationslogik geändert werden mußte. Abbildung 11 zeigt einen Vergleich der Laufzeiten auf den homogenen Systemen und dem Metacomputer, bei dem die optimierte Version von TRACE eingesetzt wurde. Die Laufzeit, die sich nun ergab, erfüllte die Erwartung, ungefähr dem Durchschnitt der Laufzeit auf den jeweiligen homogenen Maschinen zu entsprechen. Obwohl dies eine deutliche Verbesserung war und die Laufzeit bei der Nutzung einer größeren Prozessoranzahl sank, war das Ergebnis nicht befriedigend. Während sich zwar eine kürzere Bearbeitungszeit einstellte, wenn man zu der Paragon Prozessoren der SP2 hinzunahm, ergab sich noch eine weitere Verbesserung, wenn man die Paragon gar nicht mehr benutzte, sondern nur die SP2 Prozessoren einsetzte: Die Laufzeit auf dem Metacomputer mit acht Prozessoren

[7]Tatsächlich wird TRACE auf einem dreidimensionalen Gebiet berechnet. Die hier vorgestellten Überlegungen und Optimierungen sind vereinfacht dargestellt, sie lassen sich aber direkt auf eine dreidimensionale Kommunikationsstruktur übertragen.

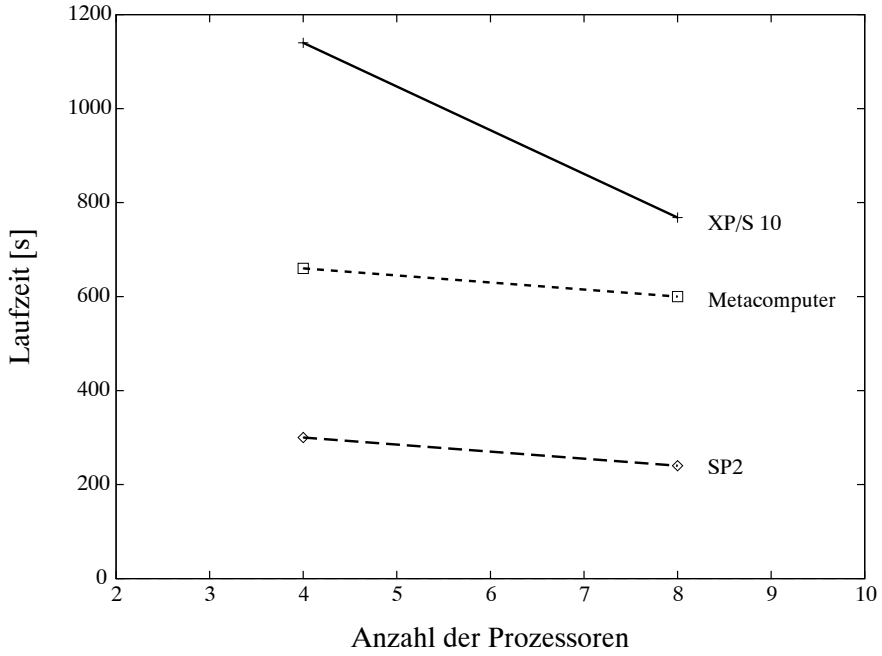

Abbildung 11: Laufzeiten der optimierten TRACE-Version.

ist höher als die Laufzeit auf vier SP2 Prozessoren allein – durch Hinzufügen der Paragon Prozessoren verschlechtert sich die Laufzeit.

Eine Reduktion der Nachrichtenanzahl ist für einen Metacomputer eine wichtige Optimierung für eng-gekoppelte Anwendungen. Aber dies allein reicht nicht aus, da sich weiterhin die relativ hohe Latenz negativ auswirkt. Hier müssen weitere Verbesserungen erreicht werden, um den praktischen Einsatz von gekoppelten Höchstleistungsrechnern zu ermöglichen. Ein Beispiel hierfür ist die Überlappung von Kommunikation mit Berechnung, so daß die Prozessoren die Zeit für die externe Datenkommunikation nutzen können.

3.4 Folgerungen

Für den erfolgreichen Einsatz eines Metacomputers auch für eng-gekoppelte Anwendungen, die mittels *message-passing* kommunizieren, sind drei Punkte von Bedeutung:

- Eine große Anzahl von externen Nachrichten führt wegen der hohen Latenz zu einer deutlichen Reduzierung der Gesamtleistung, da nur eine externe Leitung vorhanden ist und somit externe Nachrichten sequentiell übertragen werden. Auch ergeben sich erst mit großen Nachrichten hohe effektive Bandbreiten. Deshalb muß die externe Kommunikation auf möglichst wenige, dann aber möglichst große Nachrichten beschränkt werden.

- Der Lastausgleich ist gerade in einem heterogenen System von entscheidender Bedeutung, da sonst die Gefahr besteht, daß die Leistung der rechenstarken Prozes-

soren nicht genutzt wird, sondern durch Warten auf die langsameren Prozessoren verloren geht.

- Durch weitergehende Konzepte wie z.B. Überlappung von Kommunikation und Berechnung muß der Einfluß der hohen externen Latenz reduziert werden.

4 Programmiermodell für Metacomputing

Wie in Abschnitt 2.5 erwähnt, war das Ziel dieser Arbeit die Entwicklung eines Konzeptes, das den effizienten Einsatz eines Metacomputers ermöglicht. Eine entscheidende Randbedingung dabei war, daß die Lösung für ein breites Anwendungsspektrum und nicht nur für einzelne Algorithmen oder Anwendungen geeignet sein sollte. Die Vorteile dieses Konzeptes sollten anhand einer prototypischen Implementierung nachgewiesen werden. Hierfür konnte das Gigabit Testbed West [35, 34] genutzt werden, für das mit Förderung des DFN-Vereins und des BMBFs eine 622–MBit/s–ATM Leitung[8] zwischen dem Forschungszentrum Jülich und der GMD - Forschungszentrum Informationstechnik zur Verfügung stand. Es konnte so ein Metacomputer gebildet werden, der aus den beiden Cray Rechnern T3E-256 bzw. T3E-512, den Cray Systemen T90 und J90 des FZJs und der IBM SP2 der GMD bestehen konnte. Der Prototyp sollte aber nicht nur für diese Konfiguration, sondern auch für andere Metacomputer einsetzbar sein.

Wichtig ist aber auch die Unterstützung durch Software-Werkzeuge. So wurde z.B. im Kapitel 3 der Einsatz von Performance-Analysewerkzeugen gezeigt. Nur mit Hilfe derartiger Werkzeuge können Engpässe erkannt und beseitigt werden, damit eine effiziente Bearbeitung eines Programms auf einem verteilten System möglich ist [99]. Gerade hier besteht aber im Bereich des heterogenen Rechnens noch Handlungsbedarf, da die meisten Werkzeuge, wie z.B. APPRENTICE [20] oder PARAGRAPH [68], nur auf einzelnen homogenen Systemen lauffähig sind.

Das Problem ist komplex und in dieser Allgemeinheit sicher nicht zu lösen. Aus diesem Grund wird zunächst der Aufbau typischer Programme für Hochleistungsrechner untersucht, damit aus den daraus gewonnenen Erkenntnissen ein Modell für derartige Programme entwickelt werden kann. Für dieses im weiteren vorausgesetzte Programmiermodell wird dann in Kapitel 5 ein Konzept zur Optimierung der Kommunikation und für eine gleichmäßige Verteilung der Rechenlast auf die einzelnen Prozessoren entwickelt.

4.1 Modelle paralleler Programme für Hochleistungsrechner

Ein Ansatz für die Programmierung einer parallelen Anwendung ist das *farming*-Konzept, das auch unter dem Namen *master-slave* oder Manager-Arbeiter [44, 80] bekannt ist. Hierbei verwaltet ein zentraler Prozeß, meist Manager oder *master* genannt, eine Menge an Aufgaben, die er an die anderen Prozesse, die sogenannten Arbeiter oder *slaves* verteilt. Sobald ein solcher Arbeiter-Prozeß seine Teilaufgabe berechnet hat, schickt er seine Lösung an den Manager, der daraufhin eine weitere Teilaufgabe an den Arbeiter-Prozeß schickt. Die Arbeiter-Prozesse untereinander müssen nicht kommunizieren. Dieses Konzept ist besonders für Rechnersysteme geeignet, die eine langsame Verbindung haben und für Anwendungen, die sich so in Teile zerlegen lassen, daß jede dieser Teilaufgaben eine erhebliche Rechenzeit benötigt. Aufgrund der selbstorganisierenden Struktur ergibt sich der Lastausgleich automatisch, wenn es deutlich mehr Teilaufgaben als Rechner gibt, da schnellere Rechner eher ein Ergebnis zurücksenden und dann direkt die nächste Teilauf-

[8] Genaugenommen wird diese Bandbreite durch vier ATM-Leitungen mit jeweils 155 MBit/s realisiert.

gabe bearbeiten können. Das wohl bislang größte Beispiel für den Einsatz dieses Prinzips war die *RSA-Challenge* '97, bei der es gelungen ist, die Verschlüsselung des Algorithmus RC5-32/12/7 56-Bit durch eine Durchsuchung des Schlüsselraumes und den Vergleich mit einer bekannten Sequenz der verschlüsselten Nachricht zu dechiffrieren [31, 37]. Dazu wurden weltweit über 10 000 Arbeitsplatzrechner und PCs eingesetzt, die jeweils einen Teil des Schlüsselraumes nach dem *farming*-Prinzip zur Durchsuchung zugeordnet bekamen. Die Rechenzeit für einen Teil des Schlüsselraumes betrug mehr als 30 Minuten, so daß der langsame Zugriff mittels Internet auf den *master*-Rechner praktisch keine Rolle spielte. Es dauerte ca. 250 Tage, um den kodierten Testtext zu entschlüsseln [31]. Während *master-slave*-Anwendungen die Nutzbarkeit heterogener, verteilter Rechnersysteme zeigen, gehören diese Anwendungen wegen der Unabhängigkeit der einzelnen Teilaufgaben voneinander und der somit fehlenden Kommunikation zu der Klasse der *embarrassingly parallel problems* (trivialen parallelen Probleme) [44]. Da derartige Anwendungen eher selten sind und auch bei relativ langsamen Verbindungsnetzwerken noch eine gute Leistung zeigen, sind hier keine weiteren Optimierungen notwendig. Aus diesem Grund werden *master-slave*-Anwendungen hier nicht weiter betrachtet.

Viele Anwendungen können die funktionale Parallelisierung [44], auch *control parallelism* genannt [80], nutzen, bei der verschiedene Funktionen jeweils einen Datensatz transformieren. Die einfachste Form ist das *pipelining*, bei dem z.B. eine Auswertung der Form $f_3(f_2(f_1(x)))$ vorgenommen wird, aber auch komplizierte Abhängigkeiten sind möglich. Durch Berechnung der verschiedenen Funktionen f_i auf unterschiedlichen Prozessoren ergibt sich eine Parallelverarbeitung, wenn ein Datenstrom $x_1, x_2, \ldots$ transformiert werden soll. Im obigen Beispiel wird zunächst $f_1(x_1)$ berechnet, im nächsten Schritt dann gleichzeitig $f_2(f_1(x_1))$ und $f_1(x_2)$ etc. Ein Beispiel für die funktionale Parallelisierung ist die Bildverarbeitung, bei der unterschiedliche Filter auf ein Ursprungsbild angewendet werden. Während dies eine einfache Art der Parallelisierung ist, ergeben sich in der Praxis für Hochleistungsrechner zwei Nachteile: zunächst zerfallen viele Anwendungen nicht in derartige Teilfunktionen, die einzeln bearbeitet werden können, und weiterhin ist nur eine recht kleine Anzahl von Prozessoren nutzbar, da unabhängig von der Menge der Daten die Anzahl der Funktionen f_i und damit der Grad der Parallelität gleich bleibt [80].

Im Gegensatz zur funktionalen Parallelität steigt bei der Datenparallelität der Grad mit der Problemgröße an [80], weshalb dieses Konzept gerade für parallele Hochleistungsrechner mit einer großen Anzahl von Prozessoren genutzt wird. Hierbei werden die Daten auf die einzelnen Prozessoren verteilt, was auch Gebietsaufteilung (*domain decomposition*) genannt wird [44]. Auf jedem Prozessor wird die gleiche Berechnung durchgeführt, allerdings mit jeweils anderen Daten, weshalb man dieses Konzept mit *single program, multiple data* (SPMD) bezeichnet [44]. Da diese Berechnungen im allgemeinen Daten benötigen, die auf anderen Prozessoren gespeichert sind [56], muß ein Datenaustausch zwischen verschiedenen Prozessoren stattfinden. Bei dem eigentlichen numerischen Kern derartiger Anwendungen unterscheidet Gropp [56] zwei verschiedene Arten von Kommunikation:

Nachbarschaftskommunikation: Hierbei werden Daten mit einer Teilmenge aller Prozessoren, den sogenannten benachbarten Prozessoren, ausgetauscht. Diese Nachbarschaftsrelation spiegelt die vorgenommene Gebietsaufteilung wieder. Durch diese Kom-

munikation findet zwar eine lokale Synchronisation mit den Nachbarn, allerdings keine globale Synchronisation aller Prozessoren statt. Diese Nachbarschaftsrelation ist zudem konstant, d.h. die Nachbarn eines Prozessors ändern sich zur Laufzeit des Programms nicht, da die vorhandene Gebietsaufteilung unverändert bleibt.

Globale Kommunikation: Bei einer globalen Kommunikation sind alle Prozessoren, die die Anwendung nutzt, beteiligt. Hierbei wird ein globales Ergebnis berechnet, zu dem die einzelnen Prozessoren jeweils Daten beisteuern.

Neben diesen beiden Kommunikationsphasen gibt es die Phase der lokalen Berechnung, bei der keinerlei Kommunikation notwendig ist, sondern nur anhand der lokalen, d.h. im Speicher des jeweiligen Prozessors befindlichen Daten ein Ergebnis berechnet wird. Somit ergibt sich das in Abbildung 12 dargestellte Schema für den numerischen Kern eines „typischen“ parallelen Programms. Dieses Modell ist jedoch für die Modellierung

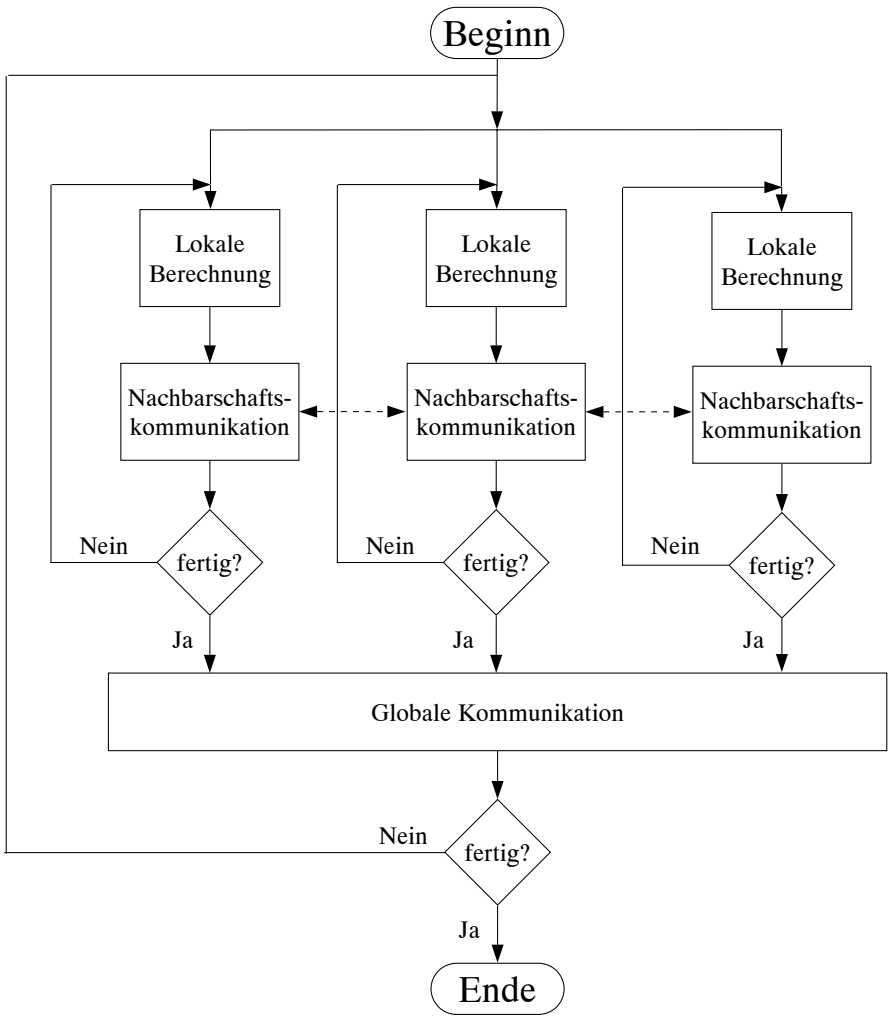

Abbildung 12: Typisches Schema des Kerns eines parallelen Programms

einer vollständigen Anwendung zu eingeschränkt, da z.B. die Initialisierungsphase, bei der Daten von einem Prozessor zu allen anderen Prozessoren verteilt werden, damit nicht modelliert werden kann. Auch haben Anwendungen für Höchstleistungsrechner häufig eine wechselnde Datenverteilung, so daß nach dem Abschluß einer Phase eine Umverteilung der Daten vorgenommen werden muß:

- Multigrid-Algorithmen [9] können bei der Parallelisierung der gröberen Gitter nicht mehr alle Prozessoren sinnvoll nutzen, weshalb dort nur noch eine Teilmenge der Prozessoren aktiv ist [134]. Dadurch ändert sich aber die Kommunikationsstruktur erheblich.
- Bei der in [7] vorgestellen makromolekularen Kristallberechnung wird zunächst eine zweidimensionale Fast-Fourier-Transformation (FFT) durchgeführt, gefolgt von einer eindimensionalen FFT orthogonal dazu. Hierfür ist eine Umverteilung der Daten sinnvoll, um für die eindimensionale FFT einen lokalen Zugriff auf die Daten zu ermöglichen.
- Eine Anwendung durchläuft verschiedene Phasen, die durch ein unterschiedliches Kommunikationsverhalten charakterisiert werden können. Bei der Initialisierungsphase wird z.B. häufig eine *broadcast*-Operation ($1 : n$ Kommunikation) durchgeführt, bei der Ermittlung eines Gesamtergebnisses hingegen eine $n : 1$ Kommunikation.
- Adaptive Algorithmen, bei denen sich zur Laufzeit die Diskretisierung der Problemstellung ändern kann, zeigen dadurch oftmals ebenfalls eine geänderte Kommunikationsstruktur.

Solche Programme haben oftmals einen enormen Rechenbedarf, weshalb sie für Metacomputing interessant sind. Deshalb muß ein allgemeineres Modell erstellt werden, um solche Anwendungen modellieren zu können.

Bei datenparallelen Anwendungen durchläuft jeder Prozeß i ($1 \leq i \leq n$) eine alternierende Folge von m Berechnungsschritten $b_{i,1}, \ldots, b_{i,m}$ und Kommunikationsschritten $k_{i,1}, \ldots, k_{i,m}$, also z.B. bei drei Schritten $b_{i,1}, k_{i,1}, b_{i,2}, k_{i,2}, b_{i,3}, k_{i,3}$. Wichtig ist hierbei, daß die Anzahl der Kommunikations- und Berechnungsoperationen bei allen Prozessoren gleich ist. Da jedem Berechnungsschritt ein Kommunikationsschritt folgt, kann man diese beiden Schritte gedanklich zusammenfassen und erhält so „kommunizierende Funktionen“, weshalb dieses Modell im weiteren CoFu-Modell, als Abkürzung für *communicating functions* genannt wird. Die einzelnen Funktionen, die mit einer Kommunikation gekoppelt sind, werden im weiteren als CoFu-Funktionen bezeichnet

Betrachtet man nun die Kommunikation, die im j-ten Schritt stattfindet, also die Menge $\{k_{1,j}, \ldots k_{n,j}\}$, erkennt man zwei unterschiedliche Arten von Kommunikation:

Synchronisierende Kommunikation: Es gibt einen Zeitpunkt, zu dem auf jeden Fall alle Prozessoren an dieser Kommunikation teilnehmen. Einfachstes Beispiel ist eine Barrieren-Operation (*barrier operation*), bei der die einzelnen Prozessoren erst weiterarbeiten können, wenn alle Prozessoren die Barriere ausgeführt haben [89]. Ein weiteres Beispiel ist die Berechnung einer globalen Summe, die anschließend auf allen Prozessoren vorhanden sein muß.

Nicht-synchronisierende Kommunikation: Hierzu gehören alle Kommunikationsmuster, bei denen es nicht notwendig ist, daß alle Prozessoren zu einem bestimmten Zeitpunkt diese Kommunikation ausführen. Es können sich dabei durchaus eine Teilmenge der Prozessoren miteinander synchronisieren, nicht aber zwangsweise alle. Eine Beispiel für eine nicht-synchronisierende Kommunikation ist eine *broadcast*-Operation, bei der ein Prozessor ein Datum an alle Prozessoren verteilt, da zum Zeitpunkt des Empfangens der Sender (und auch andere Prozessoren) diese Operation bereits beendet haben können.

Man beachte, daß die Menge der synchronisierenden Kommunikationsoperationen nicht identisch ist mit der Menge der globalen Operationen, bei denen alle Prozesse einer Anwendung beteiligt sind (vgl. [89]). Auch besteht kein Zusammenhang mit der synchronisierenden Sendeoperation (vgl. Abschnitt 2.3 und [89]), die sich auf einzelne Paare von Prozessoren bezieht, wohingegen sich die nicht-synchronisierende Kommunikation im obigen Sinn auf die Gesamtheit der Kommunikation in einem Kommunikationsschritt bezieht.

Da eine synchronisierende Kommunikation aus mehreren nicht-synchronisierenden Kommunikationen zusammengesetzt werden kann [80], ist theoretisch die Unterscheidung zwischen diesen beiden Klassen nicht notwendig. In der Praxis hingegen ist aber der Einsatz möglichst optimaler Algorithmen erforderlich, die ggf. auch die Struktur des Netzwerkes berücksichtigen [80]. Ein Beispiel für deren Wichtigkeit ist, daß sich an diesen Stellen besonders eine ungleiche Verteilung der Auslastung der Prozessoren zeigt: Prozessoren, die ihre Berechnungen schnell beendet haben, müssen in der synchronisierenden Kommunikation auf die länger rechnenden Prozessoren warten. Zeichnet man nun die synchronisierenden Operationen aufgrund ihres Einflusses in der Praxis besonders aus, so ergibt sich folgende Struktur: Jeder einzelne Prozessor i $(1 \leq i \leq n)$ durchläuft eine alternierende Folge von Berechnungsschritten $b_{i,1}, \ldots, b_{i,m}$ und nicht-synchronisierenden Kommunikationsschritten $k_{i,1}, \ldots, k_{i,m-1}$, bevor eine globale Kommunikation g durchgeführt wird, also z.B. $b_{i,1}, k_{i,1}, b_{i,2}, k_{i,2}, b_{i,3}, g$. Abbildung 13 zeigt schematisch das Modell, das im weiteren vorausgesetzt wird. Der entscheidende Unterschied zu Abbildung 12 ist, daß hierbei die einzelnen Kommunikationsschritte in jeder Iteration unterschiedlich sein können, also keine feste Nachbarschaftsstruktur vorausgesetzt wird. Eine lokale Berechnung kann sich somit durchaus auf eine Datenumverteilung beschränken.

Die formale Spezifikation des Verhaltens einer CoFu-Anwendung erfolgt im nächsten Abschnitt.

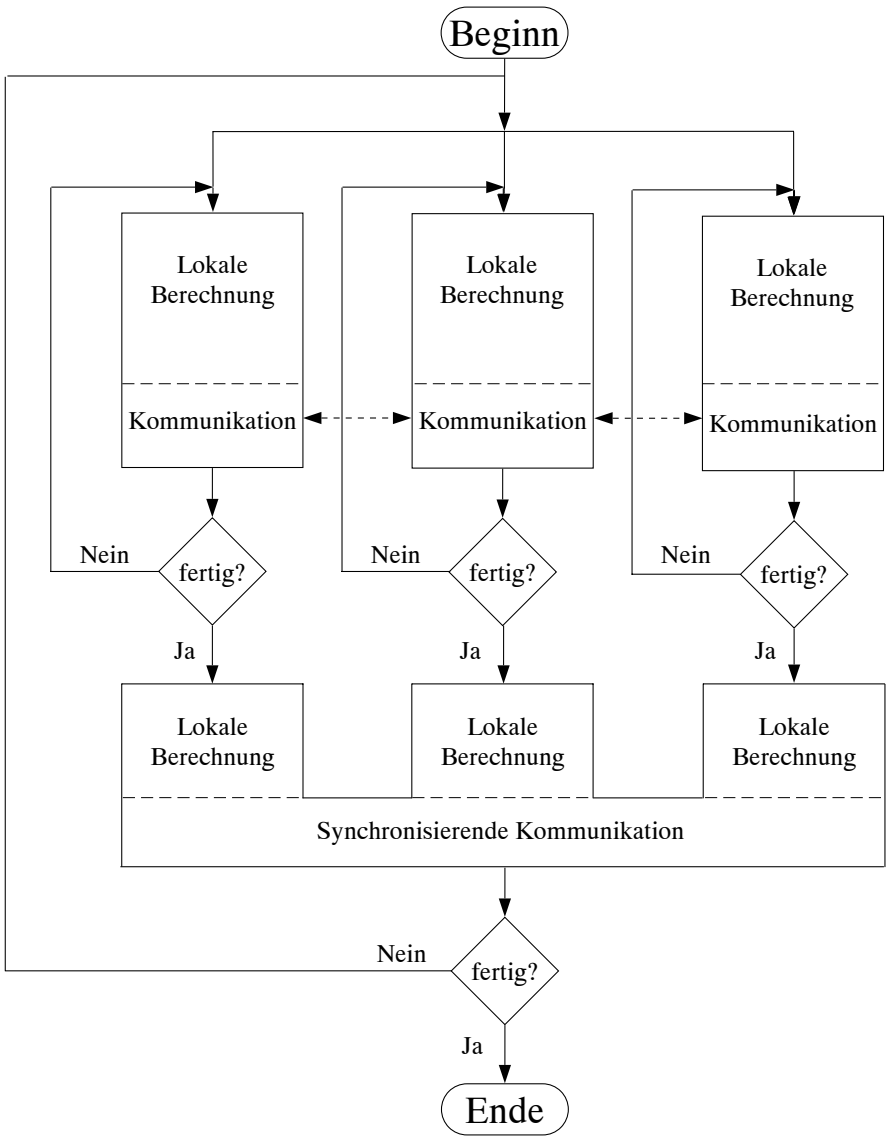

Abbildung 13: CoFu-Modell einer parallelen Anwendung

4.2 Formale Spezifikation des CoFu-Modells

Für die formale Spezifikation des CoFu-Modells wird zunächst ein einfaches Modell eines Prozessors angegeben. Mehrere derartiger Prozessoren zusammen mit einer Funktion zum Nachrichtenaustausch definieren dann einen Metacomputer. Anhand dieser Definition wird ein Modell eines heterogenen Systems spezifiziert, auf dem ein CoFu-Programm bearbeitet wird. Bei dieser Spezifikation werden die synchronisierenden Operationen nicht gesondert behandelt, da sich eine synchronisierende Operation aus einzelnen nicht-synchronisierenden Funktionen zusammensetzen läßt [80]. In der Praxis hingegen sollten die synchronisierenden Kommunikationen aus Effizienzgründen optimiert implementiert werden.

Definition eines Prozessors: Die Menge T definiert den Grunddatentyp des verwendeten Rechners, im allgemeinen ein Byte. Mit $M(n)$ wird der Speicher der Anwendung modelliert, der n Daten speichern kann. Dies umfaßt nicht nur den zur Verfügung stehenden Hauptspeicher, sondern auch Platz auf permanenten Speichermedien. Jeder Prozessor verfügt weiter über einen Eingabe- und ein Ausgabepuffer $B^{in}(n)$ und $B^{out}(n)$, die für die Kommunikation mit anderen Prozessoren geschrieben werden.

$$\begin{aligned} T &:= \{0, 1, \ldots, n_{number}\} \\ M(n) &:= T^n \\ B^{in}(n) &:= T^n \\ B^{out}(n) &:= T^n \\ P(n_{memory}, n_{in}, n_{out}) &:= M(n_{memory}) \times B^{in}(n_{in}) \times B^{out}(n_{out}) \end{aligned}$$

Im weiteren wird zur Vereinfachung der Schreibweise die explizite Angabe der Speicher- bzw. Puffergröße weggelassen und nur M, B^{in} bzw. B^{out} genutzt. Auf ein einzelnes Datum eines Puffers $b \in B^{in} \cup B^{out}$ wird mittels eines Index $i \in \mathbb{N}$ zugegriffen, geschrieben als $b[i]$. Ein $p \in P(n_{memory}, n_{in}, n_{out})$ ist ein Prozessor, im weiteren CoFu-Prozessor genannt, der über einen Hauptspeicher der Größe n_{memory}, einen Eingabepuffer der Größe n_{in} und einen Ausgabepuffer der Größe n_{out} verfügt. Im folgenden beschreibt P die Menge aller Prozessoren unabhängig von der Speicher- und Puffergröße.

Definition eines Metacomputers: Ein Metacomputer ist eine Menge von Prozessoren zusammen mit einer *message-passing*-Funktion $\mathcal{M}$, die die Ausgabepuffer eines Prozessors in die Eingabepuffer der anderen Prozessoren überträgt. Die genaue Angabe der Verteilung der Daten im Ausgabepuffer geschieht mit Hilfe von Kommunikationsfunktionen k.

$$\begin{aligned} K &:= \{k \mid k : B^{out} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N} \cup \{\perp\}\} \\ \mathcal{M} &: P \times K \to P \\ MC(n_{procs}) &:= ((p_1, \ldots, p_{n_{procs}}), \mathcal{M}) \text{ mit } p_i \in P \end{aligned}$$

Ein Metacomputer besteht aus einem geordneten Tupel von Prozessoren und der *message-passing*-Funktion $\mathcal{M}$, die den Übergang eines Empfangsprozessors bei einer Kommunikation darstellt. Dies wird modelliert, indem $\mathcal{M}$ den empfangenden Prozessor als Parameter

enthält und den Prozessor mit dem modifizierten Eingabepuffer als Funktionsergebnis zurückgibt. Dabei gibt eine Kommunikationsfunktion $k \in K$ an, welche Daten des aktuellen Prozessors in den Eingabepuffer der anderen Prozessoren übertragen werden sollen. Die Menge K ist dabei die Menge aller Kommunikationsfunktionen. Für ein $k \in K$, einem Ausgabepuffer $b \in B^{out}$, einem Index i und einem Prozessorindex r ist $k(b, i, r)$ die Position innerhalb des Eingabepuffers von Prozessor p_r, an der das zu versendende Datum $b[i]$ geschrieben werden soll. Falls als Funktionsergebnis $\perp$ zurückgegeben wird, soll das entsprechende Datum nicht zu diesem Prozessor übertragen werden. Mit Hilfe der Kommunikationsfunktionen kann eine beliebige Verteilung der Daten in dem Ausgabepuffer an eine Teilmenge der Prozessoren, die den Metacomputer bilden, vorgenommen werden. Ein $k \in K$ gibt dabei nur an, wie die Daten übertragen werden sollen, die eigentliche Übertragung wird noch nicht vorgenommen. Dies wird von der *message-passing*-Funktion $\mathcal{M}$ vorgenommen.

Definition eines CoFu-Programms: Mit Hilfe dieser Definition eines Metacomputers kann nun die Spezifikation des CoFu-Modells vorgenommen werden. Dazu wird eine Menge von Berechnungsfunktionen C definiert, die schließlich zusammen mit den oben definierten Kommunikationsfunktionen K die Menge der CoFu-Funktionen bildet.

$$\begin{aligned} F &:= \{(c,k) \mid c \in C, k \in K\} \cup \{null\} \\ C &:= \{c \mid c : B^{in} \times M \to B^{out} \times M \times F\} \end{aligned}$$

Eine CoFu-Funktion $f \in F$ ist dabei ein Paar (c, k), wobei $c \in C$ eine Berechnungsfunktion und $k \in K$ eine Kommunikationsfunktion ist. Zusätzlich ist die spezielle Funktion *null* eine CoFu-Funktion, die das Ende der Berechnung signalisiert. Eine Berechnungsfunktionen $c \in C$ führt mit Hilfe der empfangenen Daten $b^{in} \in B^{in}$ eine Transformation des Speicher M durch und schreibt die zu versendenden Daten in den Ausgabepuffer $b^{out} \in B^{out}$. Weiterhin wird als Funktionsergebnis die nächste zu bearbeitende CoFu-Funktion f zurückgegeben.

Ein CoFu-Programm ist dann eine Menge $\{f_i \in F \mid 0 \leq i < n_{functions}\}$. Die Funktion f_0, auch initiale Funktion genannt, beginnt die Berechnung.

Definition der Semantik eines CoFu-Programms: Ein CoFu-Programm wird durch die Funktion

$$CoFu^F_{n_{procs}} \quad : \quad B^{in} \times M \to B^{out} \times M \times F$$

ausgewertet. Diese Funktion übergibt die Übergabeparameter $(arg_1, \ldots, arg_{n_{args}})$ an die initiale Funktion. Die eigentliche Berechnung, die das CoFu-Modell vornimmt ist, durch das in Abbildung 14 dargestellte Flußdiagramm definiert. Die Bearbeitung eines Programms entspricht also dem Auswerten der Funktion

$$CoFu^F_{n_{procs}}((arg_1, \ldots, arg_{n_{args}}), m_{init}) \mapsto (b^{out}, m, null).$$

Das Ergebnis der Berechnung kann nun nach belieben entweder in den Ausgabedaten b^{out} oder in dem Speicher m enthalten sein.

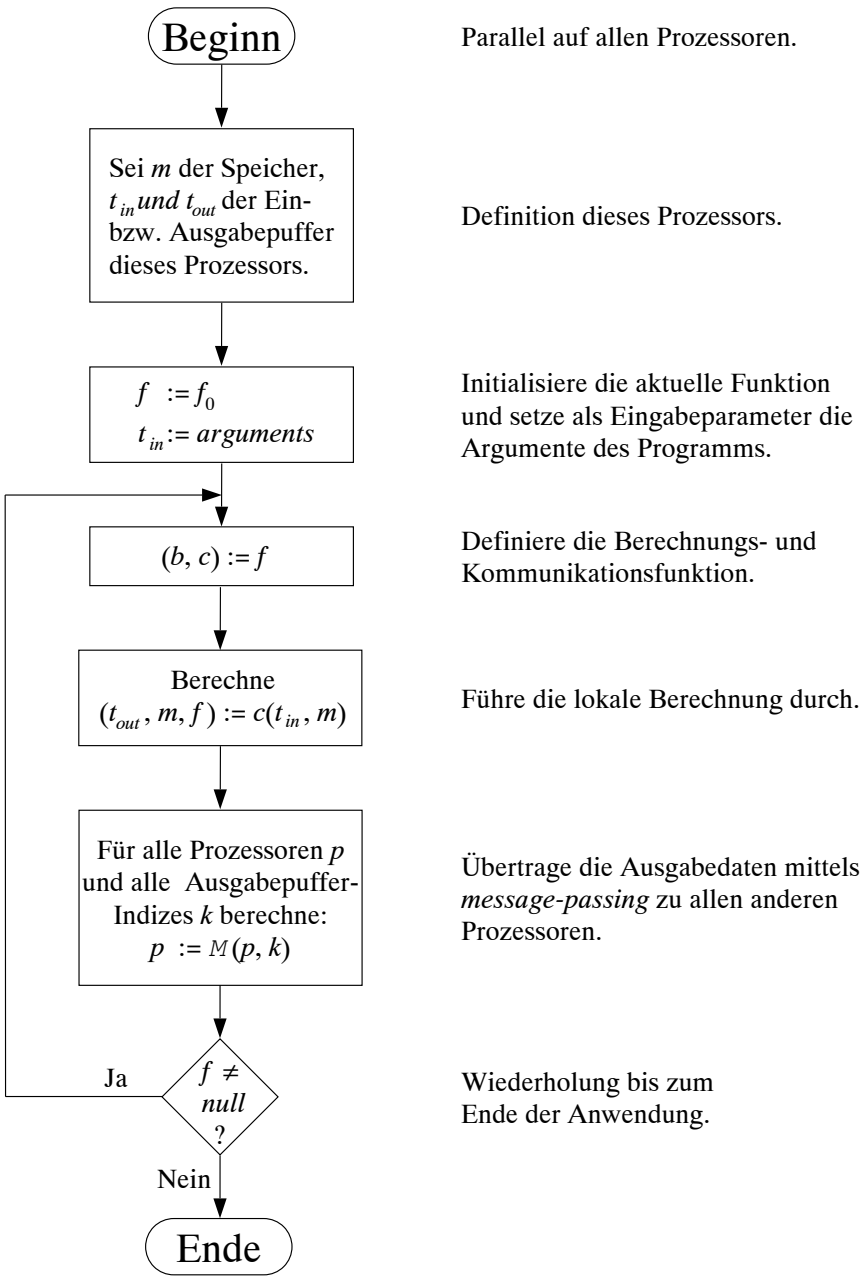

Abbildung 14: Flußdiagramm des CoFu-Modells

Während damit die formale Spezifikation abgeschlossen ist, erkennt man anhand dieses Modells noch eine Besonderheit: $CoFu^F_{n_{procs}}$ selbst ist wieder eine Berechnungsfunktion eines geeigneten übergeordneten CoFu-Modells, denn es gilt $CoFu^F_{n_{procs}} \in B$. Es ist also möglich, CoFu-Modelle ineinander zu schachteln. Die sich hierdurch ergebenden Möglichkeiten werden im nächsten Kapitel vorgestellt.

4.3 Hierarchisches CoFu-Modell

Wie schon in Abschnitt 2.2 erwähnt wurde, ist die Kopplung zweier oder mehrerer Anwendungen ein wichtiger Einsatzbereich von Metacomputing, da gerade hierbei die einzelnen Algorithmen völlig unterschiedliche Ansprüche an den Rechner stellen können. Mit dem einfachen CoFu-Modell sind derartige Anwendungen nicht modellierbar, da nicht davon ausgegangen werden kann, daß die einzelnen Teile der Anwendungen die gleiche Anzahl von Berechnungsschritten benötigen. Wie im vorherigen Abschnitt gezeigt wurde, ist es möglich, CoFu-Modelle ineinander zu schachteln – man erhält ein Hierarchisches-CoFu-Modell. Dies bedeutet, daß jede der CoFu-Funktionen selbst wieder ein vollständiges CoFu-Modell sein kann, wie in Abbildung 15 schematisch dargestellt. Dementsprechend kann jede der CoFu-Funktionen wiederum aus einer Gruppe von Prozessen bestehen, die selbst mittels eines CoFu-Modells beschrieben werden können. Eine derartige Gruppe wird im folgenden CoFu-Gruppe genannt. Ein Berechnungsschritt einer derartigen Funktion zerfällt dann in eine Folge von Berechnungs- und Kommunikationsschritten, die sich dann nur auf die Prozesse einer derartigen CoFu-Gruppe beziehen. Dies bedeutet insbesondere, daß eine synchronisierende Kommunikation nur die Prozesse synchronisiert, die in der gleichen CoFu-Gruppe liegen. Die einzelnen Prozesse einer solchen Gruppe müssen auch nicht notwendigerweise von dem gleichen Rechner bearbeitet werden, sondern die darunterliegende Hardware kann durchaus wieder ein Metacomputer sein. Die einzelnen CoFu-Funktionen, die jeweils durchaus wieder CoFu-Gruppen sein können, tauschen durch die entsprechenden synchronisierenden oder nichtsynchronisierenden Kommunikationen Daten aus.

Durch dieses hierarchische CoFu-Modell wird die funktionale Parallelisierung mit CoFu möglich. Damit können nun auch gekoppelte Anwendungen modelliert werden, wodurch eine große Anwendungsklasse potentieller Metacomputing-Anwendungen modellierbar ist.

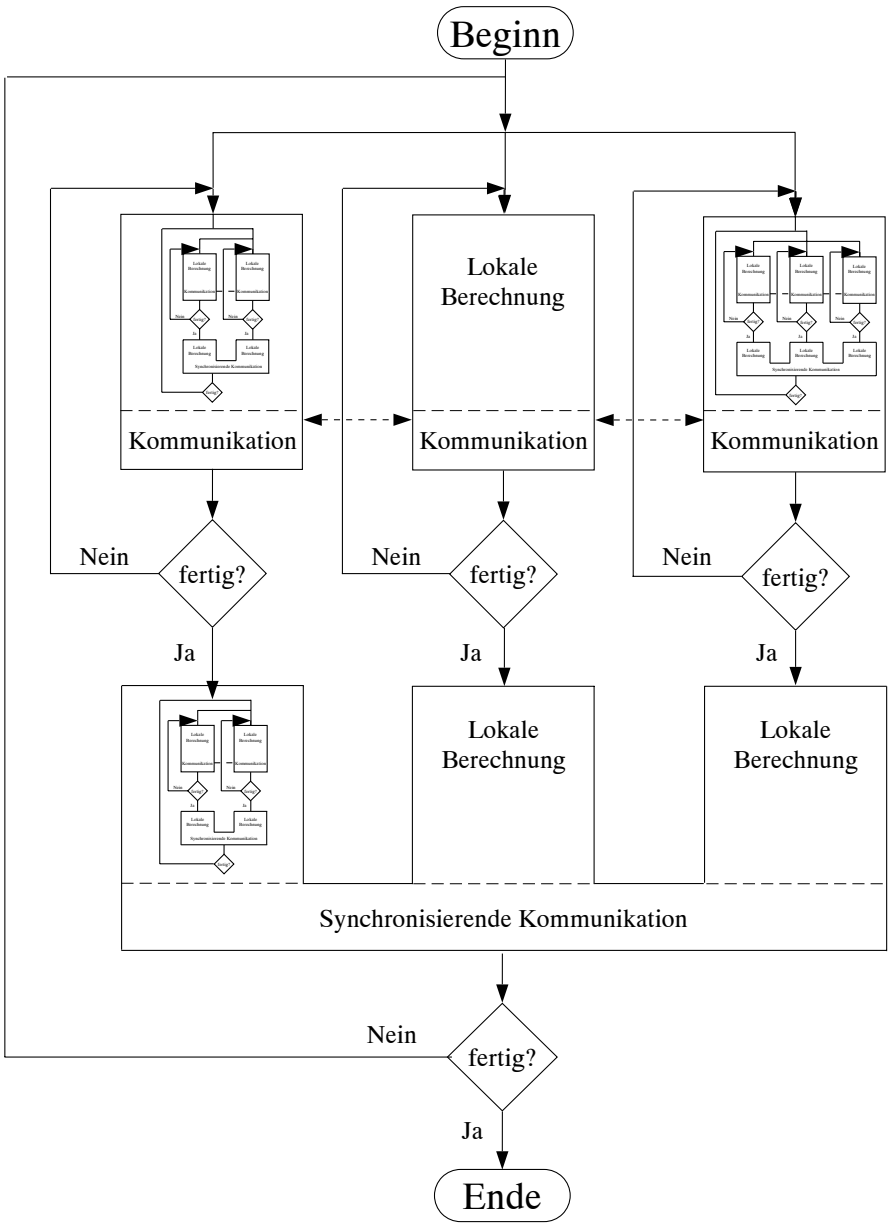

Abbildung 15: Hierarchisches CoFu-Modell

4.4 Zusammenfassung der Vorüberlegungen

Die in diesem Kapitel vorgenommenen Vorüberlegungen und die Analysen aus Kapitel 3 haben zu folgenden Anforderungen bzw. Voraussetzungen geführt:

- Es wird ein *message-passing*-Programmiermodell vorausgesetzt, das mittels des in Abbildung 15 dargestellten CoFu-Modells beschrieben werden kann.
- Die Anzahl der externen Nachrichten muß minimiert werden, um den sequentialisierenden Einfluß der einzelnen externen Verbindung und der hohen Latenz möglichst gering zu halten.
- Kommunikation muß mit Berechnung überlappt werden, damit die vergleichsweise hohe Latenz und geringe Bandbreite der externen Verbindung nicht zu einem Effizienzverlust führt.
- Eine möglichst gleichmäßige Verteilung der Rechenlast auf die einzelnen Prozessoren muß erreicht werden, da anderenfalls durch Wartezeit Rechenleistung verloren geht.
- Die Unterstützung durch Software-Werkzeuge ist wichtig, um weitere, anwendungsspezifische Optimierungen vornehmen zu können.
- Es dürfen nicht nur einzelne, spezielle Algorithmen optimiert werden, sondern die Lösung muß möglichst allgemein einsetzbar sein.
- Der Aufwand zur Nutzung der Lösung muß möglichst gering sein. Auch muß weiterhin eine effiziente Programm-Code-Erzeugung durch den Compiler möglich sein. Dies bedeutet, daß die gängigen Programmiersprachen aus dem Bereich des Hochleistungsrechnens, Fortran, C und C++, unterstützt werden müssen.
- Der Prototyp muß auf einem Metacomputer, der aus den Systemen Cray T3E-256, T3E-512, Cray T90 und IBM SP2 besteht, nutzbar sein. Eine weitergehende Verwendbarkeit für andere Systeme ist aber wünschenswert.
- Der Einsatz auf einem homogenen System sollte ohne signifikanten Effizienzverlust möglich sein.
- Eine möglichst einfache Programmierung des Metacomputers ist wünschenswert.

5 Optimierung von Metacomputing-Anwendungen

Dieses Kapitel beschreibt die einzelnen Konzepte, die für die Optimierung und den Lastausgleich von Metacomputing-Anwendungen entwickelt wurden. Dabei wird vorausgesetzt, daß die zu optimierende Anwendung mit dem in Abschnitt 4.1 vorgestellten CoFu-Modell programmiert wurde. Zunächst wird die Optimierung der externen Kommunikation vorgestellt, bevor die Unterstützung des CoFu-Programmiermodells erläutert wird. Anschließend wird beschrieben, wie die Überlappung von Kommunikation und Berechnung durch Unterstützung eines geeigneten *scheduling* erreicht wird. Schließlich werden ein statisches und ein dynamisches Lastausgleichsverfahren vorgestellt.

5.1 Optimierung der externen Kommunikation

Wie die in Kapitel 3 vorgestellten Untersuchungen gezeigt haben, ist die Optimierung der externen Kommunikation von großer Bedeutung für den effizienten Einsatz eines Metacomputers. Für *message-passing*-Programme bedeutet dies, daß externe Nachrichten in eine große Nachricht zusammengefaßt und verschickt werden müssen. Dies gilt zunächst für Nachrichten, die von einem Prozessor zu einem Prozessor einer anderen Maschine geschickt werden. Diese offenkundige Optimierung wird allerdings im allgemeinen schon bei der Programmentwicklung berücksichtigt, da eine Minimierung der Anzahl der Nachrichten auch für homogene Systeme eine Leistungssteigerung bewirkt (vgl. aber z.B. [91]: Dort wird diese Optimierung erst beim Übergang auf ein verteiltes Rechnernetzwerk explizit vorgenommen). Allerdings reicht dies allein nicht aus, um auch für einen Metacomputer eine gute externe Kommunikationsleistung zu erreichen, siehe Abschnitt 3.3.

Bei einem Metacomputer ist nur eine externe Leitung vorhanden. Wenn nun mehrere Nachrichten gleichzeitig (oder in kurzen Abständen hintereinander) übertragen werden, entsteht ein Konflikt beim Zugriff auf das externe Netzwerk. In der Praxis wird dies durch einen der folgenden Mechanismen behoben:

- Es existieren nur einer oder wenige Kommunikationsprozessoren[9], die für die Nutzung einer Schnittstelle zuständig sind. Wenn ein beliebiger Prozessor über eine Schnittstelle Daten übertragen muß, sendet er eine entsprechende Anforderung an einen dieser Kommunikationsprozessoren, der dann die entsprechende Übertragung der Daten über die Schnittstelle veranlaßt. Wenn mehrere Anforderungen eintreffen, werden diese nacheinander bearbeitet, die Daten werden somit sequentiell übertragen. Dies gilt z.B. für die Cray T3E, bei der spezielle *multi purpose nodes* (MPNs) [104] die externe Kommunikation vornehmen.
- Wenn jeder Prozessor eine eigene externe Schnittstelle hat, wird nahezu gleichzeitig mit der Übertragung begonnen. Da hierdurch aber die Kapazität der Leitung überschritten wird, müssen auf einer tieferen Netzwerksschicht Daten entweder gepuffert werden, bis die Leitung wieder zur Verfügung steht, oder die Daten gehen sogar verloren und müssen erneut übertragen werden. Dies geschieht transparent für

[9] Meist sind das bei einem Parallelrechner besondere Prozessoren, die nur für diese Aufgabe eingesetzt werden, also nicht für Berechnungen zur Verfügung stehen. Dies geschieht transparent für den Anwender.

die Anwendung [130]. Auch in diesem Fall werden die Daten schließlich sequentiell übertragen. Dies ist z.B. bei der IBM SP2 der Fall, bei der jeder Knoten eine eigene externe Schnittstelle hat.

Unabhängig davon, wie diese Konflikte beim Zugriff auf die externe Leitung gelöst werden, werden die einzelnen Nachrichten nacheinander übertragen. Abbildung 16 a) zeigt

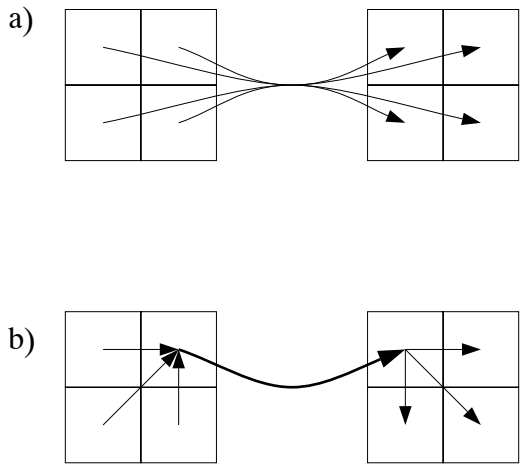

Abbildung 16: Zusammenfassen externer Nachrichten.

den nicht optimierten Vorgang: Hier sendet jeder Prozessor der ersten Maschine eine Nachricht an einen Prozessor der zweiten Maschine. Wenn alle vier Datenübertragungen gleichzeitig beginnen, muß die letzte Nachricht das vollständige Versenden von drei Nachrichten abwarten, bevor sie übertragen werden kann. In diesem Beispiel steigt die Wartezeit um mehr als den Faktor drei an. Angesichts der relativ hohen Latenz der externen Verbindung und der großen Anzahl von Prozessoren, die in Metacomputing-Anwendungen genutzt werden soll, ist diese Vorgehensweise nicht sinnvoll. Die Übertragung mehrerer Nachrichten von Prozessoren eines Parallelrechners zu einem anderen Rechner kann optimiert werden, indem diese Nachrichten auf einem Prozessor zusammengefaßt und als eine größere Nachricht übertragen werden, wie es in Abbildung 16 b) dargestellt ist. Auf der Empfängerseite wird diese Nachricht wieder in Einzelnachrichten zerlegt, die dann an die ursprünglichen Empfänger gesendet werden.

Für eine einzelne Nachricht hat dieses Verfahren keine Vorteile, da eine Nachricht dabei zusätzlich zweimal intern verschickt wird. Da aber die interne Übertragungszeit deutlich kürzer als die externe Übertragungszeit ist, ist eine Reduzierung der Gesamtübertragungszeit möglich, wenn eine ausreichende Anzahl von Nachrichten übertragen wird. Es hängt von den Randbedingungen wie der internen und externen Latenz bzw. Bandbreite ab, ob und wann sich dieses Zusammenfassen lohnt. Um dies abzuschätzen, betrachte man den Datenaustausch zwischen zwei Maschinen, von denen jede n Prozessoren hat. Jeder dieser Prozessoren schicke eine Nachricht der Länge k zu einem Prozessor der anderen Maschine. Wenn die Nachrichten nicht zusammengefaßt werden, müssen sie sequentiell übertragen werden, da nur eine externe Leitung vorhanden ist. Abbildung 17 zeigt detailliert den

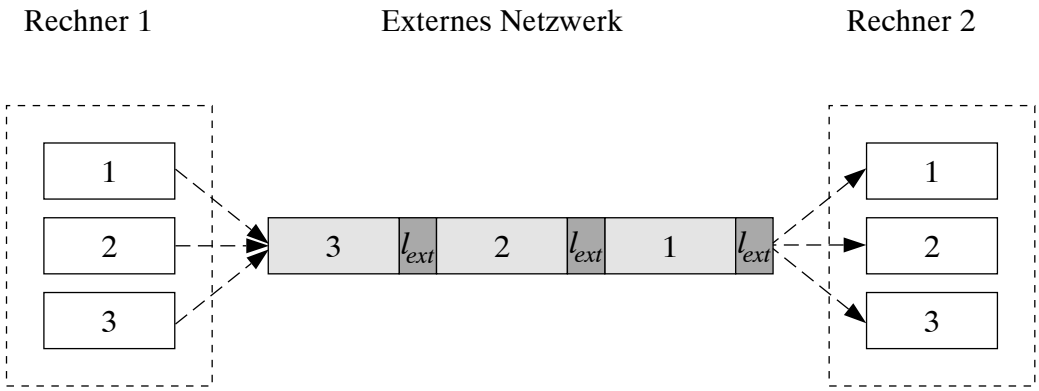

Abbildung 17: Detaillierte Darstellung des Übertragungsvorgangs im nicht-optimierten Fall.

Übertragungsvorgang im nicht-optimierten Fall. Für jede Nachricht fällt die relativ hohe externe Latenz an, die bei der Übertragungszeit mit berücksichtigt werden muß. Nach [44] (vgl. auch Gleichung (1) auf Seite 24) kann die Gesamtübertragungszeit für diese n Nachrichten der Länge k bei einer externen Latenz von l_{ext} und einer externen Bandbreite von b_{ext} durch

$$t_{nicht\_optimiert} = n(l_{ext} + k/b_{ext})$$

angenähert werden. Abbildung 18 zeigt schematisch den Übertragungsvorgang, wie er

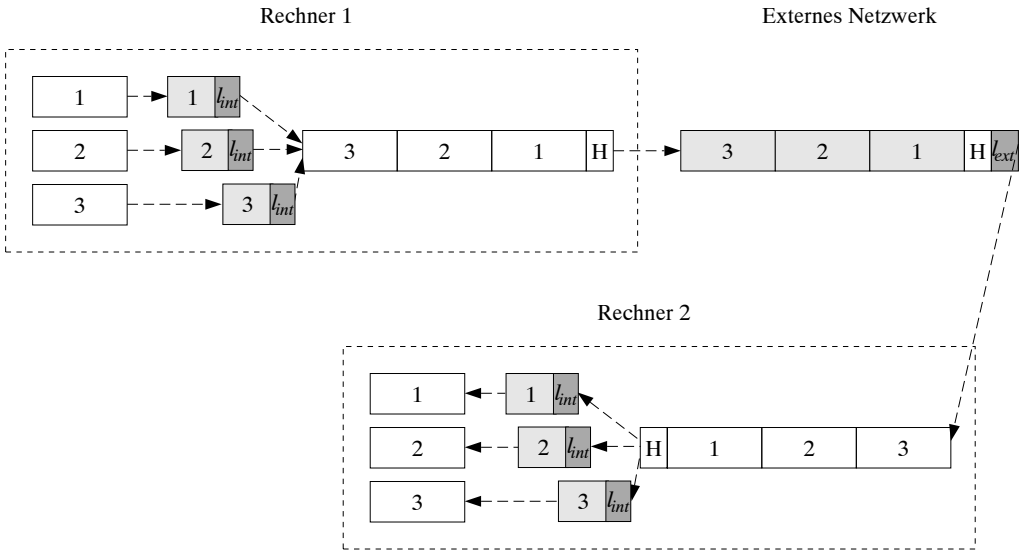

Abbildung 18: Detaillierte Darstellung der Übertragung im optimierten Fall.

sich beim Zusammenfassen der Nachrichten ergibt[10]. Es werden zuerst n Nachrichten mit

[10]Diese Zeichnung ist nicht maßstabsgetreu: Die externe Latenz ist mehr als hundertmal größer als die interne Latenz.

der internen Latenz l_{int} und der internen Bandbreite b_{int} zu einem Prozessor übertragen. Da in einem Parallelrechner im allgemeinen mehrere Nachrichten gleichzeitig übertragen werden können[11], ist die dafür notwendige Zeit höchstens so groß wie die Zeit für das sequentielle Übertragen von n Nachrichten innerhalb eines Rechners. Anschließend wird diese zusammengesetzte Nachricht extern übertragen. Diese externe Nachricht muß aber zusätzlich Daten über die Einzelnachrichten enthalten, damit das Zerlegen und Weiterleiten auf der Empfängerseite möglich ist. Hierfür werden weitere h Bytes pro Nachricht benötigt. Auf der Empfängerseite wird diese Nachricht dann wieder in die Einzelnachrichten zerlegt und an die jeweiligen Empfängerprozesse weitergeleitet. Auch hier können bei einem Parallelrechner wiederum mehrere Nachrichten gleichzeitig verschickt werden, so daß sich die Übertragungszeit nach oben durch das sequentielle Übertragen der n Nachrichten abschätzen läßt. Insgesamt läßt sich die optimierte Übertragungszeit $t_{optimiert}$ nach oben durch

$$t_{optimiert} \leq n \cdot t_{int}(k) + t_{ext}(nk + nh) + n \cdot t_{int}(k) \tag{3}$$

abschätzen[12], wobei $t_{int}(k) = l_{int} + k/b_{int}$ die Zeit für die Übertragung einer Nachricht der Länge k innerhalb eines Rechners ist. Damit die Optimierung durch Zusammenfassung der einzelnen Nachrichten tatsächlich einen Gewinn bringt, muß gelten:

$$t_{nicht\_optimiert} = n(l_{ext} + k/b_{ext}) > 2n(l_{int} + k/b_{int}) + l_{ext} + \frac{nk + nh}{b_{ext}} \geq t_{optimiert}.$$

Die mittlere Ungleichung ist äquivalent zu

$$n \left[l_{ext} - 2l_{int} - \frac{2k}{b_{int}} - \frac{h}{b_{ext}} \right] > l_{ext}.$$

Unter der Voraussetzung

$$\begin{aligned} & l_{ext} - 2l_{int} - \tfrac{2k}{b_{int}} - \tfrac{h}{b_{ext}} > 0 \\ \iff \quad & \tfrac{b_{int}}{2} \left[l_{ext} - 2l_{int} - \tfrac{h}{b_{ext}} \right] > k \end{aligned} \tag{4}$$

ergibt sich als hinreichende Bedingung für die Anzahl der Nachrichten, um einen Geschwindigkeitsgewinn durch Zusammenfassen der einzelnen externen Nachrichten zu erhalten:

$$n > \frac{l_{ext}}{l_{ext} - 2l_{int} - \frac{2k}{b_{int}} - \frac{h}{b_{ext}}} \tag{5}$$

Im Rahmen des Gigabit Testbed West wurden folgende Werte für die Werte der einzelnen Parameter gemessen [33]:

[11] Dies hängt natürlich stark von der Architektur des jeweiligen Rechners und der Zuordnung der Prozesse zu den Prozessoren ab.

[12] Wie in Abbildung 18 zu sehen ist, reichen eigentlich $n-1$ interne Nachrichten aus, wenn das Zusammenfassen auf einem Prozessor geschieht, der ebenfalls eine Nachricht extern verschickt. Die tatsächlich vorgenommene Implementierung hingegen (vgl. Abschnitt 7.3) nutzt einen eigenen Kommunikationsprozessor, der unabhängig von den Prozessoren der Anwendung ist, so daß doch n Nachrichten erforderlich sind.

$$\begin{array}{lcllcl} l_{int} & = & 16 \cdot 10^{-6}\text{s} & l_{ext} & = & 3.5 \cdot 10^{-3}\text{s} \\ b_{int} & = & 308 \cdot 10^{6}\ \text{Bytes/s} & b_{ext} & = & 5.8 \cdot 10^{6}\ \text{Bytes/s} \\ h & = & 12\ \text{Bytes} & & & \end{array}$$

Hierbei berechnet sich der Wert für h aus der Annahme, daß für jede Nachricht Länge, Empfänger und eine Typinformation angegeben werden muß, was jeweils durch einen Integer-Wert der Länge 4 Bytes geschieht. Aus Gleichung (4) erhält man mit Hilfe dieser Werte $k < 533\,753.37$ Bytes. Unter dieser Voraussetzung berechnet man aus Gleichung (5) die Bedingung $n > 1.93$. Sobald also mindestens zwei Nachrichten versendet werden, die jeweils kürzer als 533 753 Bytes sind, ist das Zusammenfassen von Nachrichten für obige Rechnerkonfiguration lohnenswert.

Um eine Abschätzung für die „typische“ Größe von Nachrichten zu erhalten, die Anwendungen austauschen, betrachte man eine reguläre dreidimensionale Struktur. Wenn eine solche Anwendung den kompletten Speicher der T3E nutzt, so entspricht dies einem Würfel mit der Seitenlänge $\sqrt[3]{128 \cdot 10^6} \approx 504$. Jede Nachricht besteht aus einer Ebene des Würfels, somit also aus ca. $k = 504^2 Bytes = 254\,016$ Bytes. Damit ist Gleichung (4) erfüllt. Demnach ist die Optimierung durch Zusammenfassen von Nachrichten für die oben erwähnte Rechnerkonfiguration und „typische“ Anwendungen sinnvoll. Für eine reguläre zweidimensionale Anwendung berechnet sich analog $k = 11\,313$ Bytes.

Nun ist allerdings noch wichtig, welche Nachrichten zusammengefaßt werden können, ohne daß sich das Programmverhalten ändert. Wartet man zu lange auf Nachrichten, kann der empfangende Rechner evtl. blockieren, da er die Daten benötigt und somit nicht weiterarbeiten kann. Es besteht sogar die Gefahr, daß eine Verklemmung (*deadlock*) erzeugt wird, wie die beiden folgenden Beispiele zeigen. In Abbildung 19 a) schickt Prozessor A_1

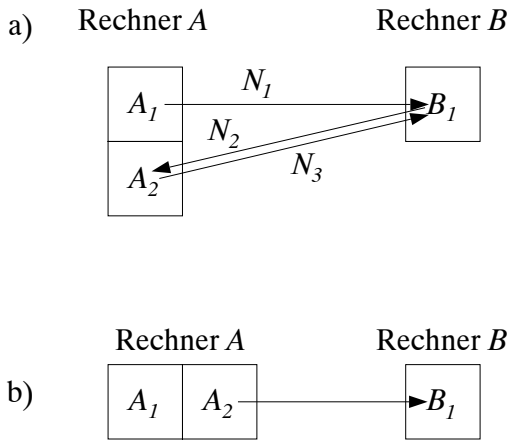

Abbildung 19: Verklemmung bzw. Leistungsverlust durch Zusammenfassung einzelner Nachrichten.

eine Nachricht N_1 an Prozessor B_1, der nach dem Empfangen von N_1 eine Nachricht N_2 an Prozessor A_2 schickt, der daraufhin eine Nachricht N_3 an Prozessor B_1 schickt. Aufgrund einer Datenabhängigkeit der Anwendung auf Prozessor B_1 kann N_2 erst nach dem Empfangen von N_1 geschickt werden. Wenn nun als Optimierung die Nachrichten N_1 und N_3

zusammengefaßt werden sollen, würde N_1 erst geschickt werden, wenn N_3 fertig ist. Dies kann allerdings nicht eintreten, da N_3 erst als Reaktion auf das Empfangen der Nachricht N_2 und damit von N_1 erzeugt wird. Eine Verklemmung tritt auf. Abbildung 19 b) zeigt einen ähnlich gelagerten Fall, in dem A_1 überhaupt keine Nachricht zu der anderen Maschine schickt. Würde in diesem Fall auf eine externe Nachricht von A_1 gewartet werden, entstünde ebenfalls eine Verklemmung.

Ohne Informationen über das Kommunikationsverhalten aller Prozessoren der Anwendung ist es also nicht allgemein möglich, die externe Kommunikation durch Zusammenfassen einzelner Nachrichten zu optimieren. Setzt man allerdings ein nicht-hierarchisches CoFu-Modell voraus, ergeben sich folgende Optimierungsmöglichkeiten:

Bei einem CoFu-Programm sind die Kommunikation und die Berechnung streng voneinander getrennt. Zuerst findet die Berechnung statt, und erst am Ende der Funktion werden die Daten übertragen. Auch sind die Berechnungen miteinander synchronisiert: Jeder Prozeß führt eine Berechnung durch, bevor ein Kommunikationsschritt durchgeführt wird. Deshalb können alle Nachrichten, die nach einem Rechenschritt gesendet werden, beliebig zusammengefaßt werden, ohne daß eine Verklemmung auftreten kann[13]. Für eine effiziente Bearbeitung muß allerdings eine ausgeglichene Lastverteilung vorhanden sein, da ansonsten Rechenzeit durch das Warten auf einzelne Nachrichten zum Zusammenfassen verloren geht. Dies wird durch die in den Abschnitten 5.4 und 5.5 vorgestellten Verfahren zum Lastausgleich erreicht. Bei einem CoFu-Programm ist nicht nur bekannt, daß alle Nachrichten eines Kommunikationsschrittes beliebig zusammengefaßt werden können, sondern auch, welche Prozesse am Ende der Berechnung Daten austauschen. Da diese Struktur für jede einzelne CoFu-Funktion konstant ist, kann einmal vorab berechnet werden, welche Nachrichten zusammengefaßt werden können, weil sie zu Prozessoren des gleichen externen Rechners geschickt werden.

Während somit das Zusammenfassen externer Nachrichten für CoFu-Programme ohne Verklemmung möglich ist, haben die Untersuchungen in Abschnitt 3.3 gezeigt, daß dies allein im allgemeinen nicht ausreicht, um eine gute Leistung eines Metacomputers zu erhalten. Es müssen weitere Optimierungen eingesetzt werden, um den großen Einfluß der hohen Latenz und geringen Bandbreite der externen Verbindung zu reduzieren. Ein Ansatz hierzu wird im nächsten Abschnitt vorgestellt.

5.2 Überlappung von Kommunikation und Berechnung

Wie in Abschnitt 3.1 beschrieben, hat die unterste Ebene der Netzhardware bzw. des Netzanschlusses einen entscheidenden Einfluß: Die Optimierung der externen Verbindung z.B. durch Reduktion der Anzahl der zwischengeschalteten *router* ist von großer Bedeutung. Auch die Datenkonvertierung sollte nicht bei jeder externen Verbindung vorgenommen werden, sondern nur, wenn sie wirklich notwendig ist. Der Einfluß der einzelnen Para-

[13]Genau diese Voraussetzung ist bei einem hierarchischen CoFu-Modell nicht erfüllt, da die einzelnen CoFu-Gruppen eines übergeordneten CoFu-Modells nicht die gleiche Anzahl von Berechnungsschritten ausführen müssen. Aus diesem Grund beschränken sich die möglichen Optimierungen bei der Nutzung eines hierarchischen Modells auf die einzelnen CoFu-Gruppen.

meter wurde in Tabelle 3 bzw. Abbildung 5 auf Seite 23 dargestellt. Obwohl diese Parameter bzw. Rahmenbedingungen im allgemeinen bereits optimiert vorliegen und die in Abschnitt 5.1 beschriebenen Verbesserungen durch Zusammenfassung externer Nachrichten vorgenommen wurden, reduziert die externe Verbindung weiterhin die Leistung einer Anwendung, da diese im Vergleich zu der internen Kommunikation zu langsam ist. Da eine weitere Reduzierung der externen Übertragungszeit (bei gegebener Hardware) nicht möglich ist, ist es wichtig ist, daß die Prozessoren währenddessen sinnvoll weiterarbeiten können, daß also Kommunikation und Berechnung überlappt werden [44]. Abbildung 20 a) zeigt schematisch den Datenaustausch ohne Überlappung von Kommunikation und Berechnung. Hier kann die Übertragungszeit der Nachrichten (dunkel markiert), nicht genutzt werden. Durch eine Überlappung stellt sich ein Verhalten wie in Abbildung 20 b) ein. Nach der recht geringen Zeit für das Initiieren der Datenübertragung rechnen die Prozessoren weiter, während die Nachrichten verschickt werden. Die ohne Überlappung anfallende Wartezeit kann genutzt werden. Diese Überlappung ist um so wichtiger, je länger die Datenübertragung dauert, weshalb diese Technik gerade für Metacomputer erfolgversprechend ist. Besonders gewinnbringend ist diese Überlappung für Hochleistungsrechner, bei denen oftmals eigene Prozessoren für die externe Kommunikation vorhanden sind (wie z.B. die *multi purpose nodes* einer Cray T3E), so daß die eigentlichen Rechenprozessoren von der Datenübertragung überhaupt nicht betroffen werden und weiterarbeiten können. Damit sich die Überlappung tatsächlich durch eine reduzierte Laufzeit bemerkbar macht, muß allerdings eine gleichmäßige Auslastung der Prozessoren gegeben sein, da sich ansonsten ein Verhalten wie in Abbildung 20 c) einstellt. In diesem Beispiel führt die Überlappung nur für den unten dargestellten Prozessor zu einer geringeren Bearbeitungszeit, da sich seine Wartezeit durch die früher losgeschickte Nachricht verkürzt. Für den oberen Prozessor hingegen bringt die Überlappung keinen Geschwindigkeitsvorteil, da die für ihn bestimmte Nachricht aufgrund seiner längeren Rechenzeit eintrifft, bevor er mit dem Empfangen beginnt – unabhängig davon, ob die Nachricht überlappt oder nicht-überlappt abgeschickt wird. Da die Gesamtrechenzeit durch den oberen Prozessor bestimmt wird, tritt in diesem Beispiel keine Reduzierung der Laufzeit ein.

Um diese Überlappung von Kommunikation und Berechnung zu ermöglichen, werden spezielle Algorithmen entwickelt, die nach dem Versenden der Daten, die andere Prozessoren benötigen, und vor dem Empfangen der Daten, die sie selbst brauchen, weitere sinnvolle Berechnungen durchführen können [27, 108]. Es ist allerdings für viele Anwendungen erstens fraglich, ob „überlappende“ Algorithmen überhaupt existieren, und zweitens ist meist ein erheblicher Programmieraufwand notwendig, um einen Algorithmus zu modifizieren bzw. zu ersetzen. Aus diesen Gründen wurde ein allgemeinerer Ansatz entwickelt.

Die Grundidee beruht hierbei darauf, mehrere CoFu-Funktionen von einem Prozessor auswerten zu lassen. Dadurch ist es möglich, daß eine weitere Funktion berechnet wird, während die Kommunikation einer anderen CoFu-Funktion durchgeführt wird. Bei der formalen Spezifikation des CoFu-Modells aus Abschnitt 4.2 wird jeweils eine CoFu-Funktion von einem Prozessor ausgeführt. Um diese formale Spezifikation weiter einzuhalten, werden mehrere CoFu-Prozessoren (vgl. Abschnitt 4.2) von einem physikalischen Prozessor simuliert. Man erhält das Konzept der VIRTUELLEN PROZESSOREN. Dieses Konzept ist schon aus dem Bereich der Betriebssysteme bekannt [72], analog wird auch bei datenpar-

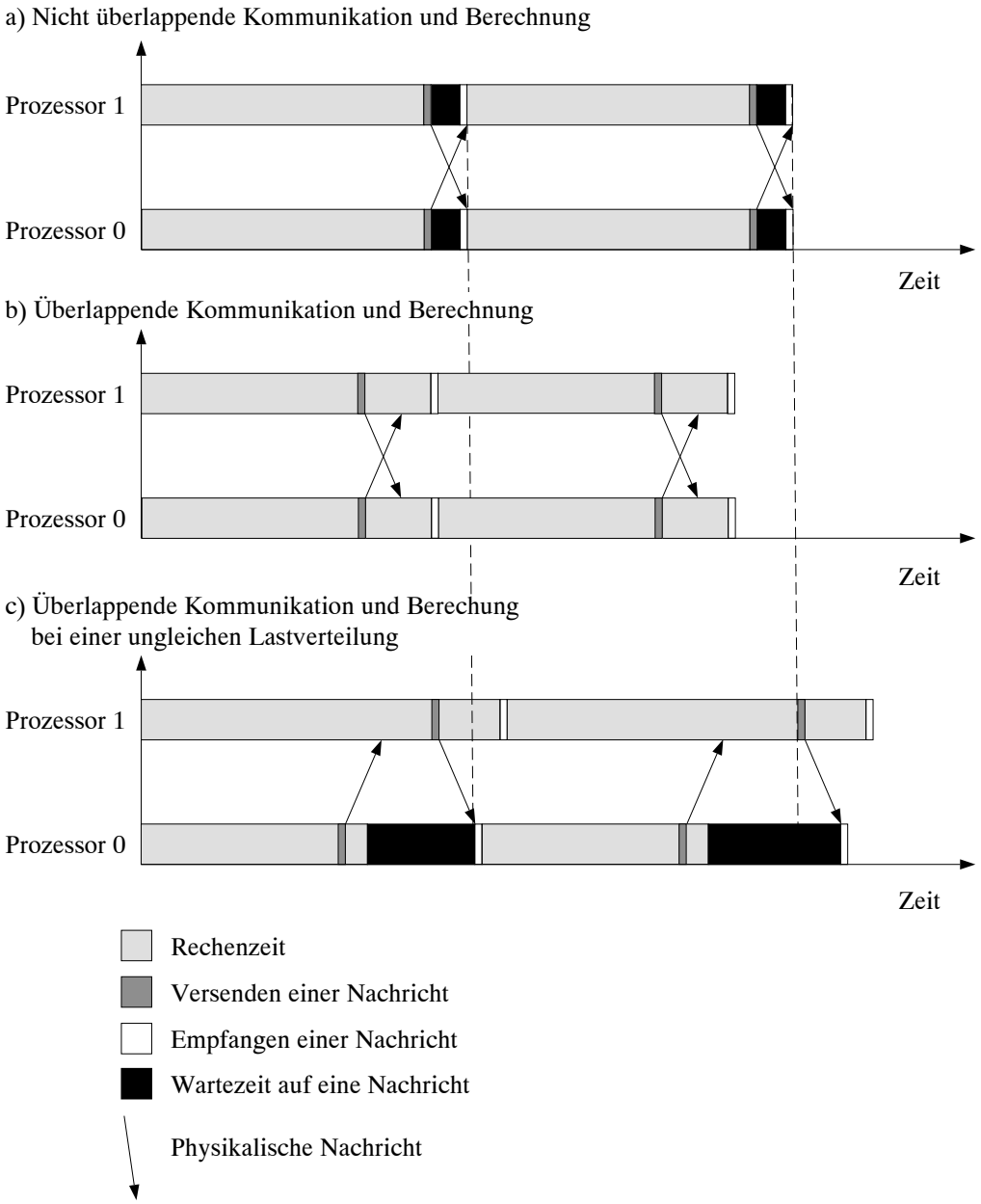

Abbildung 20: Reduktion der Kommunikationszeit durch Überlappung von Kommunikation und Berechnung.

allelen Programmiersprachen wie z.B. *High Performance Fortran* (HPF) von virtuellen Prozessoren gesprochen [107], die die Grundlage für die Datenverteilung bilden, indem die virtuellen Prozessoren den physikalischen Prozessoren zugeordnet werden[14]. Sobald einer dieser virtuellen Prozessoren kommunizieren muß, benötigt er die physikalische CPU nicht mehr und ein anderer virtueller Prozessor kann bearbeitet werden. Durch ein geeignetes *scheduling* der Bearbeitung der virtuellen Prozessoren durch den physikalischen Prozessor ist somit eine Überlappung von Kommunikation und Berechnung möglich. In einer Metacomputing-Umgebung ist es allerdings wichtig, daß wegen der längeren Übertragungszeit externer Nachrichten zuerst die virtuellen Prozessoren bearbeitet werden, welche extern Nachrichten schicken müssen. Während dies für allgemeine Anwendungen nicht bekannt ist, kann dies für CoFu-Programmen beim *scheduling* berücksichtigt werden, da die Kommunikationsstruktur bekannt ist. Hieraus resultieren erhöhte Anforderungen an das *scheduling*.

Während die theoretischen Vorteile dieses Konzeptes offenkundig sind, muß auch die Eignung dieses Konzeptes für eine Implementierung auf einem Metacomputer untersucht werden. Eine Implementierung muß nicht nur ein geeignetes *scheduling* der virtuellen Prozessoren ermöglichen, sondern auch auf einem heterogenen System einsetzbar sein und eine hohe Effizienz ermöglichen. Im folgenden werden deshalb verschiedene Implementierungen der virtuellen Prozessoren untersucht:

5.2.1 Prozesse simulieren virtuelle Prozessoren

Ein naheliegender Ansatz zur Einrichtung mehrerer virtueller Prozessoren auf einem physikalischen Prozessor ist der Einsatz mehrerer Prozesse, die jeweils einen virtuellen Prozessor simulieren. Dieser Ansatz wird z.B. auch in [44, 106] beschrieben, ohne daß allerdings die Abstraktionsebene der virtuellen Prozessoren eingeführt wird. Sobald ein Prozeß kommuniziert, kann ein anderer Prozeß aktiviert werden, so daß die Kommunikation mit Berechnung überlappt wird. Dieser Ansatz hat allerdings für die praktische Implementierung in einer Metacomputing-Umgebung einige Nachteile:

- Da jeder Prozeß ein eigenes Codesegment enthält, in dem der Programmcode abgelegt ist, befindet sich somit dieser Programmcode mehrfach im Speicher, wodurch Speicherplatz verloren geht.
- Die Zeit für den notwendigen Kontextwechsel bei der Aktivierung eines anderen Prozesses ist recht hoch. Es kann sich ggf. gar nicht lohnen, einen anderen Prozeß in der Zwischenzeit aufzuwecken, da die Zeit für einen Kontextwechsel größer ist als die Wartezeit [44].
- Obwohl ein Zugriff auf den Speicher der anderen Prozesse, die auf dem gleichen Prozessor bearbeitet werden, möglich ist, muß dazu doch ein Aufruf einer Funktion aus dem Betriebssystemkern erfolgen, da aus Sicherheitsgründen ein direkter Zugriff

[14] Obwohl das Konzept der virtuellen Prozessoren im Rahmen dieser Arbeit speziell für CoFu-Prozesse entwickelt wurde, kann es natürlich auch für Nicht-CoFu-Prozesse eingesetzt werden.

auf den Speicher anderer Prozesse nicht erlaubt ist. Ein solcher Betriebssystemaufruf kostet weitere Zeit.

- Der Prozeß-*scheduler* ist ein Teil des Betriebssystems, so daß praktisch kein direkter Einfluß auf das *scheduling* genommen werden kann, ohne daß eine Änderung des Betriebssystems durchgeführt werden müßte, was wegen der großen Anzahl von Zielplattformen nicht praktikabel ist. Bei fast allen Systemen ist ein einfaches *round-robin* Verfahren als *scheduler* implementiert, d.h. jeder Prozeß kann der Reihe nach eine bestimmte Zeit lang rechnen, bevor der nächste Prozeß aufgeweckt und der gerade aktive Prozeß an das Ende der Schlange der rechenbereiten Prozesse angehängt wird [131]. Für Metacomputing ist aber insbesondere die Reihenfolge der Auswertung der virtuellen Prozessoren wichtig.

- Teilweise lassen Systeme den Einsatz mehrerer Prozesse nicht zu – dies ist z.B. für die Cray T3E z.Zt. noch der Fall.

5.2.2 Durch *threads* implementierte virtuelle Prozessoren

Ein anderer verbreiteter Ansatz zur Implementierung virtueller Prozessoren bzw. zur Überlappung von Kommunikation und Berechnung ist die Nutzung von *threads*. Diese leichtgewichtigen Prozesse [131] ermöglichen verschiedene Kontrollflüsse in einem Prozeß und überlagern somit die Wartezeit einzelner *threads* mit Berechnungen eines anderen *threads* [107]. Die Kosten für den Wechsel zu einem anderen *thread* sind dabei deutlich geringer als bei einem vollständigen Prozeßwechsel, da hier der Austausch der Code- und Datensegments entfällt und nur der *stack* und die Register aktualisiert werden müssen [131]. Ebenso ist die Kommunikation der leichtgewichtigen Prozesse untereinander wegen des gemeinsamen Speichers, auf dem sie ohne Probleme zugreifen können, einfacher und schneller. Allerdings ist auch hier eine Synchronisation der *threads* notwendig, um sogenannte „zeitkritische Abläufe“ (*race conditions*) zu verhindern [131]. Der große Nachteil von *threads* im Bereich des Metacomputings ist die mangelnde Standardisierung und die geringe Verfügbarkeit. So ist z.B. für die Cray T3E noch kein *threading* möglich, und ein einheitlicher *thread*-Standard, der auf einer Vielzahl von Hochleistungsrechnern einsetzbar ist, existiert auch noch nicht. Weiterhin ist es bei *threads* wie auch schon bei der Realisierung von virtuellen Prozessoren durch verschiedene Prozesse schwierig, ein geeignetes *scheduling* zu implementieren, ohne Änderungen am Betriebssystem bzw. an den jeweiligen *thread*-Bibliotheken der einzelnen Systeme vorzunehmen.

5.2.3 Eigenständige Simulation der virtuellen Prozessoren

Die dritte Möglichkeit ist die Simulation der virtuellen Prozessoren in einem Prozeß (mit einem *thread*), also die sogenannten *user based threads*. Hierbei wird ein Prozeß (mit einem Kontrollfluß) genutzt, der die Datenstruktur für mehrere virtuelle Prozessoren enthält. Die Simulation der virtuellen Prozessoren geschieht durch den Aufruf einer Funktion, die die Daten des jeweiligen Prozessors als Parameter übergeben bekommt. Die einzelnen virtuellen Prozessoren können dann in einer Schleife der Reihe nach bearbeitet werden, wobei

die Reihenfolge der Aufruf in Abhängigkeit vom gewünschten *scheduling* vorgenommen wird.

Während dies als eigenständige Lösung einen höheren Programmieraufwand für die Implementierung des Konzeptes bedeutet, bietet diese Lösung zahlreiche Vorteile:

- Das *scheduling* der virtuellen Prozessoren wird explizit durch die Aufrufreihenfolge bestimmt. Dadurch kann einfach die gewünschte Überlappung von Kommunikation und Berechnung durchgeführt werden.
- Da die Implementierung von Anfang an für ein heterogenes System gedacht ist, kann dies berücksichtigt werden und die Anzahl der systemspezifischen Aufrufe minimiert bzw. gekapselt werden[15].
- Da die einzelnen virtuellen Prozessoren eines physikalischen Prozessors einen gemeinsamen Adreßbereich haben, können sie direkt auf den Speicher der anderen virtuellen Prozessoren zugreifen. Da nur ein Kontrollfluß vorhanden ist, ist keine weitere Synchronisation (wie bei *threads*) notwendig. Auch geschieht der Zugriff ohne den Aufruf von Betriebssystemfunktionen – je nach Implementierung muß entweder einmal der zu übertragende Datenbereich kopiert werden, oder es reicht die Weitergabe eines Zeigers (*pointer*) auf die Adresse, bei der die Daten beginnen.
- Die Kosten für einen Wechsel zu einem anderen virtuellen Prozessor sind niedrig, da nur ein Zeiger auf die Daten des jeweiligen virtuellen Prozessor auf den *stack* gelegt und ein Funktionsaufruf vorgenommen werden muß.

5.2.4 Implementierung virtueller Prozessoren

Wie zu sehen ist, sind die ersten beiden Ansätze, nämlich die Implementierung von virtuellen Prozessoren durch Prozesse bzw. durch *threads*, nicht für eine Metacomputing-Umgebung geeignet. Der letzte Ansatz hingegen, die sogenannten *user based threads*, ist auch für ein derartiges heterogenes Rechnersystem effektiv implementierbar.

Gemäß des CoFu-Modells (vgl. auch die formale Spezifikation in Abschnitt 4.2) besteht eine Anwendung aus einzelnen Funktionen. Dieser Programmierstil wird durch ein sogenanntes Skelett (*skeleton*, auch *programming frame* genannt [23, 94]) gut unterstützt. Der Begriff Skelett stammt aus dem Bereich der funktionalen Programmierung, wo ein *skeleton* eine polymorphe Funktion höherer Ordnung beschreibt, also eine Funktion, die beliebige andere Funktionen als Parameter enthalten kann [23]. Das CoFu-Skelett entspricht der Implementierung des in Abschnitt 4.2 vorgestellten CoFu-Modells. Aus praktischer Sicht ergibt sich durch die Nutzung eines Skelettes eine Verschiebung der Kontrolle: das Anwendungsprogramm übergibt nicht, wie bei einem Bibliotheksaufruf, die Kontrolle an die aufgerufene Bibliotheksfunktion, sondern das Skelett übergibt die Kontrolle an eine Funktion des Anwendungsprogramms. Um ein paralleles Programm zu entwickeln, müssen nur die anwendungsspezifischen Teile als Funktionen programmiert

[15] Die Erfahrungen bei der Implementierung (vgl. Abschnitt 7) haben sogar gezeigt, daß für die Realisierung keinerlei systemspezifische Befehle notwendig sind.

werden, die dann von dem CoFu-Skelett aufgerufen werden. Wie schon in Abschnitt 5.2.3 beschrieben, wird der eigentliche Kontrollfluß der Anwendung durch die Reihenfolge der Funktionsaufrufe realisiert.

Während dies nur ein unbedeutender Unterschied zu sein scheint, ergeben sich aber signifikante Vorteile:

- Die Programmentwicklung beschränkt sich auf die Entwicklung der einzelnen CoFu-Funktionen. Der Datenaustausch zwischen den virtuellen Prozessoren muß nicht mehr explizit programmiert werden, wodurch sich die Programmentwicklung vereinfacht. Die jeweiligen Funktionen können auf die einzelnen Eingabe- und Ausgabepuffer der virtuellen Prozessoren direkt zugreifen und so die empfangenen Daten verarbeiten bzw. die zu sendenden Daten bereitstellen.

- Bei der Nutzung von Fließkomma-Zahlen besteht die Gefahr, daß das Ergebnis einer Berechnung von der Reihenfolge abhängt, in der die einzelnen Rechenschritte durchgeführt werden. Dies ist besonders bei einer parallelen Bearbeitung möglich, wenn z.B. Nachrichten bei verschiedenen Programmläufen in einer anderen Reihenfolge empfangen werden. Das Ergebnis ist ggf. nicht mehr reproduzierbar, was die Fehlersuche erschwert. Dies ist bei der Nutzung des vorgestellten funktionalen Skelettes nicht mehr der Fall: Da die empfangenen Daten der jeweiligen Nachbarn gleichzeitig als Parameter übergeben werden, ist die Reproduzierbarkeit der Ergebnisse gewährleistet (solange das Anwendungsprogramm diese Daten in der gleichen Reihenfolge bearbeitet).

- Die Optimierung der externen Kommunikation kann durch das Skelett vorgenommen werden, ohne daß die Anwendung speziell geändert werden muß. Hierzu gehören insbesondere die in diesem Kapitel vorgestellten Optimierungen des Zusammenfassens von Nachrichten und der Überlappung von Kommunikation und Berechnung. Entsprechendes gilt für globale Operationen: auch diese können implementiert und optimiert werden, ohne daß die Anwendung davon betroffen wird. Dies ist möglich, da aufgrund der Nutzung des entwickelten Programmiermodells die Information über die Kommunikationsstruktur vorhanden ist, was bei allgemeinen *message-passing* Bibliotheken nicht der Fall ist.

- Es ist nicht möglich, eine Verklemmung zu erzeugen:
  Notwendig für eine Verklemmung ist, daß eine Teilmenge der Prozessoren zyklisch aufeinander warten [72]. Da aber bei der Nutzung des funktionalen Skelettes keine Kommunikationsanweisung in der Anwendung selbst eingefügt werden muß, kann eine solche Situation von der Anwendung allein nicht erzeugt werden. Aber auch das CoFu-Skelett selbst ist mit Sicherheit verklemmungsfrei[16]. Durch die Trennung von Kommunikations- und Berechnungsschritten, die in dem CoFu-Modell in Abschnitt 4.1 vorgenommen wurde, findet jegliche Kommunikation quasi gleichzeitig statt. Dabei werden zuerst alle Daten nicht blockierend gesendet, bevor mit dem Empfangen

[16]Hierbei wird allerdings eine korrekte Implementierung des Skelettes und ausreichende Ressourcen vorausgesetzt.

begonnen wird. Dadurch ist ein zirkuläres Warten auf Nachrichten ausgeschlossen, und eine häufige Fehlerursache der parallelen Programmierung beseitigt.

- Das CoFu-Skelett kann in einer homogenen und einer heterogenen Version zur Verfügung gestellt werden, so daß die Anwendung weiterhin ohne jegliche Änderung des Programmcodes auf einem einzelnen Hochleistungsrechner effizient berechnet werden kann.
- Die Unterstützung durch weitere Werkzeuge ist einfacher möglich. So kann z.B. das Laufzeitsystem für die Erzeugung von Performancedaten zur Laufzeit vollständig und transparent für die Anwendung in das CoFu-Skelett integriert werden.

Der Einsatz virtueller Prozessoren hat allerdings auch Nachteile. Durch deren Nutzung steigt die Anzahl der Prozesse, zwischen denen die Daten verteilt werden. Dadurch kann sich das Konvergenzverhalten des eingesetzten Algorithmus verschlechtern, so daß mehr Rechenschritte notwendig sind, damit die Genauigkeit des Ergebnisses nicht reduziert wird. Dies ist vor allem bei älteren Algorithmen der Fall, neue Verfahren hingegen werden im allgemeinen durch den Anstieg der Anzahl der Prozesse nicht betroffen. Weiter steigt mit der Anzahl der virtuellen Prozessoren auch der Aufwand für die Verwaltung der Daten der virtuellen Prozessoren und für die Berechnung des *schedulings*. Dieser Effekt muß durch eine effiziente Implementierung möglichst gering gehalten werden. In Abschnitt 8.2 wird dieser Aufwand für die prototypische Implementierung genauer untersucht. Schließlich steigt mit der Anzahl der virtuellen Prozessoren auch die Anzahl der Nachrichten an, die verschickt werden müssen. Durch eine geeignete Implementierung des CoFu-Skelettes kann diese Anzahl aber signifikant reduziert werden: alle Nachrichten, die von virtuellen Prozessoren eines physikalischen Prozessors im gleichen Kommunikationsschritt geschickt werden, können auf diesem Prozessor zusammengefaßt und als eine Nachricht verschickt werden. Der Empfangsprozessor kann daraus die einzelnen Nachrichten für die virtuellen Prozessoren wiederherstellen. Dadurch werden auch bei der Nutzung von virtuellen Prozessoren nicht mehr Nachrichten verschickt, als bei dem Einsatz von einem Prozeß pro physikalischem Prozessor. Auch hier wird, wie schon in Abschnitt 5.1, die dem CoFu-Modell inhärente Trennung von Kommunikation und Berechnung und die Kenntnis über die Kommunikationsstruktur genutzt, um dieses Zusammenfassen verklemmungsfrei zu ermöglichen. Somit müssen nur noch die lokalen Nachrichten, also Nachrichten zwischen virtuellen Prozessoren auf dem gleichen physikalischen Prozessor zusätzlich bearbeitet werden. Aufgrund der Implementierung der virtuellen Prozessoren in einem Prozeß mit einem *thread*, kann diese Kommunikation ohne zusätzliche Synchronisation durch einen direkten Speicherzugriff realisiert werden, so daß der hierfür notwendige Aufwand vernachlässigbar ist.

Bei einer sorgfältigen Implementierung der Simulation virtueller Prozessoren durch einen Prozeß ist dieses Konzept geeignet, um bei einem Metacomputer eine Überlappung von Kommunikation und Berechnung zu ermöglichen, ohne daß spezielle Algorithmen genutzt oder entwickelt werden müssen. Hierzu ist natürlich ein geeignetes *scheduling* erforderlich. Der folgende Abschnitt stellt ein prioritätsgesteuertes *scheduling*-Verfahren vor, daß im Rahmen dieser Arbeit speziell für den Einsatz in Metacomputing-Umgebungen entwickelt wurde.

5.3 *Scheduling* der virtuellen Prozessoren

Nach der Diskussion in Abschnitt 5.2.4 und den in Abschnitt 5.1 vorgestellten Kommunikationsoptimierungen stellt sich die Kommunikationsstruktur eines CoFu-Modells wie in Abbildung 21 dar. Dort ist schematisch ein Metacomputer abgebildet, der aus zwei

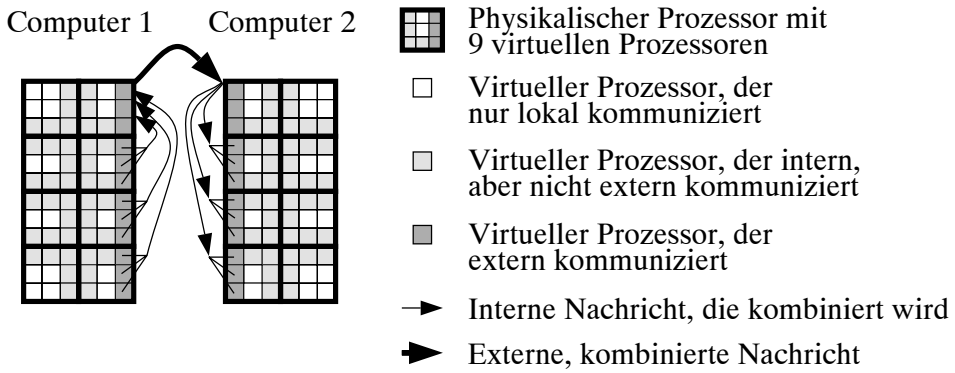

Abbildung 21: Kommunikationsstruktur bei der Nutzung virtueller Prozessoren.

Rechnern besteht, die jeweils acht Prozessoren zur Verfügung stellen. Auf jedem dieser physikalischen Prozessoren werden neun virtuelle Prozessoren simuliert. Die Anordnung der virtuellen Prozessoren entspricht dabei der Kommunikationsstruktur der Anwendung, d.h. es liegt eine zweidimensionale Gebietsaufteilung vor, wobei jeder virtuelle Prozessor mit den waagerecht und senkrecht benachbarten Prozessoren Daten austauscht. Interne Nachrichten, also Nachrichten, die zu einem virtuellen Prozessor auf einem anderen physikalischen Prozessor der gleichen Maschine geschickt werden, werden dabei beim Sender zusammengefaßt, übertragen und anschließend beim Empfänger wieder in die einzelne Nachrichten für die jeweiligen virtuellen Prozessoren zerlegt. Bei einer externen Nachricht, also einer Nachricht, die zu einem virtuellen Prozessor auf einer anderen Maschine gesendet wird, findet dieses Zusammenfassen hingegen auf zwei Ebenen statt: einerseits innerhalb eines jeden physikalischen Prozessors, der mehr als eine externe Nachricht zur gleichen Maschine verschicken muß, andererseits auf dem Prozessor, der für das Kombinieren zu einer externen Nachricht verantwortlich ist. Dementsprechend wird beim Empfangen der Nachricht vorgegangen: Zuerst wird die Nachricht in die Teile zerlegt, die für die einzelnen Prozessoren innerhalb der Maschine bestimmt sind, dort werden dann die jeweiligen Nachrichten für die virtuellen Prozessoren wiederhergestellt. Damit Kommunikation und Berechnung möglichst weitgehend überlappen, müssen zuerst die extern kommunizierenden virtuellen Prozessoren berechnet werden, also die in Abbildung 21 dunkelgrau dargestellten Prozessoren. Sobald alle diese Prozessoren ihre lokalen Daten berechnet haben, können die externen Nachrichten kombiniert und mit der Übertragung dieser Nachricht an den Prozessor, der für die externe Kommunikation verantwortlich ist, begonnen werden. Während die Nachricht übertragen wird, können diejenigen virtuelle Prozessoren rechnen, die interne Nachrichten verschicken. Hier ist es sinnvoll, zuerst die Prozessoren zu bearbeiten, die zu den meisten anderen Prozessoren Daten verschicken

müssen, damit möglichst frühzeitig die kombinierten internen Nachrichten bereitstehen. Zum Schluß werden die virtuellen Prozessoren berechnet, die ausschließlich lokal kommunizieren müssen. Solche Prozessoren sind wichtig, damit auch das Versenden der letzten Nachricht mit Berechnungen überlappt werden können.

Aufgrund der bei CoFu-Programmen vorhandenen Trennung von Kommunikation und Berechnung besteht keine Abhängigkeiten zwischen den virtuellen Prozessoren untereinander, so daß ein *scheduling* nur in Abhängigkeit der Kommunikationsstruktur berechnet werden kann. Die Reihenfolge der Bearbeitung der virtuellen Prozessoren innerhalb eines physikalischen Prozessors läßt sich durch die Festlegung einer Priorität für die einzelnen virtuellen Prozessoren steuern. Um die beschriebenen Kriterien für das *scheduling* zu berücksichtigen, wird die PRIORITÄT eines virtuellen Prozessors als das Tupel $(m, e, i) \in \mathbb{N}_0^3$ definiert, wobei gilt:

m: Anzahl der externen Maschinen, zu denen Daten geschickt werden müssen.
e: Anzahl der externen Nachrichten, also Nachrichten zu virtuellen Prozessoren auf anderen Maschinen.
i: Anzahl der internen Nachrichten, also Nachrichten zu virtuellen Prozessoren, die auf derselben Maschine bearbeitet werden.

Die Reihenfolge des *schedulings* der einzelnen virtuellen Prozessoren innerhalb eines physikalischen Prozessors ist dann durch die Relation $\succ$ festgelegt, die folgendermaßen definiert ist:

$$
\begin{aligned}
(m_1, e_1, i_1) \succ (m_2, e_2, i_2) \quad \Longleftrightarrow \quad & (m_1 > m_2) \vee && (6)\\
& ((m_1 = m_2) \wedge (e_1 > e_2)) \vee && (7)\\
& ((m_1 = m_2) \wedge (e_1 = e_2) \wedge (i_1 > i_2)) && (8)
\end{aligned}
$$

Von zwei virtuelle Prozessoren P_1, P_2 mit den Prioritäten $p_1, p_2 \in \mathbb{N}_0^3$ muß P_1 vor P_2 berechnet werden, wenn $p_1 \succ p_2$ gilt. Dadurch werden zuerst die virtuellen Prozessoren bearbeitet, die zu den meisten externen Maschinen Nachrichten versenden müssen (Gleichung (6)), danach diejenigen, die die meisten externen Nachrichten schicken (Gleichung (7)), und schließlich diejenigen, die die meisten internen Nachrichten senden müssen (Gleichung (8)). Zum Schluß werden dann die virtuellen Prozessoren bearbeitet, die nur noch mit anderen virtuellen Prozessoren auf dem gleichen Prozessor Daten austauschen müssen. Dadurch wird eine Überlappung auf zwei Stufen erreicht: Das Versenden externer Nachrichten wird mit der Bearbeitung von virtuellen Prozessoren mit interner und nur lokaler Kommunikation überlappt. Während die internen Nachrichten versendet werden, rechnen die virtuellen Prozessoren, die nur lokal kommunizieren müssen. Wenn zwei Prozessoren die gleiche Priorität haben, spielt die Reihenfolge der Bearbeitung keine Rolle und kann zufällig bestimmt werden. In Abbildung 22 ist ein Beispiel für die Berechnung der Prioritäten und ein sich daraus ergebendes *schedule* dargestellt. Es wird ein Metacomputer aus drei verschiedenen Rechnern benutzt. Jeder Prozessor bekommt neun virtuelle Prozessoren zugeordnet, von denen jeder mit den in der Abbildung waagerecht, senkrecht und diagonal benachbarten Prozessoren kommunizieren muß. Für den physikalischen Prozessor, dessen virtuelle Prozessoren mit Nummern versehen sind, berechnet sich die Priorität der einzelnen virtuellen Prozessoren zu:

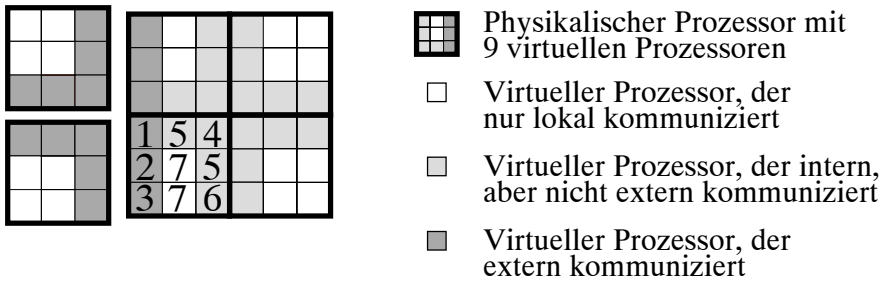

Abbildung 22: Verschiedene *scheduling*-Klassen.

$$p_1 = (2,3,2) \quad p_2 = (1,3,0) \quad p_3 = (1,2,0)$$
$$p_4 = (0,0,5) \quad p_5 = (0,0,3) \quad p_6 = (0,0,2)$$
$$p_7 = (0,0,0)$$

Virtuelle Prozessoren mit der gleichen Nummer haben die gleiche Priorität. Welcher von diesen zuerst berechnet wird, spielt somit keine Rolle – wichtig ist nur, daß vorher die Berechnungen aller Prozessoren mit einer höheren Priorität beendet sind und deren Kommunikation begonnen wurde.

Mit dem hier vorgestellten Konzept der virtuellen Prozessoren und dem entwickelten *scheduling* ist eine Überlappung vom Kommunikation und Berechnung möglich. Eine wichtige Voraussetzung sowohl für diese Überlappung als auch für die Optimierung der externen Kommunikation (vgl. Abschnitt 5.1) ist eine gleichmäßige Auslastung der einzelnen Prozessoren. Im weiteren wird deshalb zunächst ein statisches und anschließend ein dynamisches Lastausgleichsverfahren vorgestellt.

5.4 Statischer Lastausgleich

Wie schon in Kapitel 3 gezeigt wurde, ist der Lastausgleich bei der Nutzung heterogener Rechnersysteme von entscheidender Bedeutung. Gerade der Vorteil von Metacomputing, nämlich die Nutzung derjenigen Rechner, die für einzelne Programmteile die beste Performance bieten, geht verloren, wenn diese Rechner auf andere, langsamere Rechner warten müssen.

Das bisher vorgestellte Konzept der virtuellen Prozessoren für die Überlappung von Kommunikation und Berechnung läßt sich auch nutzen, um ein statisches Lastausgleichsschema zu implementieren. Statische Verfahren [95] haben im Gegensatz zu dynamischen Verfahren den Vorteil, daß sie zur Laufzeit keinen zusätzlichen Rechenaufwand benötigen, sondern nur bei der initialen Aufteilung der Daten aktiv sind [44]. Bei der Nutzung von virtuellen Prozessoren kann ein solches statisches Lastausgleichsschema implementiert werden, indem die Anzahl der virtuellen Prozessoren, die von einem physikalischen Prozessor simuliert werden, an die Rechenleistung der Prozessoren angepaßt wird. Dies erlaubt nur einen recht groben Ausgleich der unterschiedlichen Rechenkapazität, da die kleinste Einheit, die variiert werden kann, ein vollständiger virtueller Prozessor ist. Im

Gegensatz dazu kann bei einem in der Anwendung integrierten Verfahren durch Variation der Datenverteilung eine viel feinere Lastbalancierung vorgenommen werden. Dies geschieht allerdings auf Kosten der Programmentwicklung, da dies eine Erweiterung der Anwendungen bedeutet, wohingegen der Lastausgleich durch die virtuellen Prozessoren ohne jegliche Änderung auskommt.

Um die Granularität des somit möglichen Lastausgleiches zu bestimmen, betrachte man eine Anwendung, die für jeden einzelnen virtuellen Prozessor die gleiche Rechenleistung benötigt[17] und die auf zwei heterogenen Prozessoren bearbeitet werden soll, die jeweils die Rechenkapazität f_1 bzw. f_2 haben. Es werden jeweils n virtuelle Prozessoren auf einem Prozessor simuliert. Demgemäß ist die Rechenzeit proportional zu n/f_1 bzw. n/f_2. Durch Verschiebung von k virtuellen Prozessoren soll die Rechenzeit auf den beiden Maschinen möglichst gleich sein, also

$$\frac{n+k}{f_1} \approx \frac{n-k}{f_2} \Rightarrow \frac{f_1}{f_2} \approx \frac{n+k}{n-k} = 1 + \frac{2k}{n-k}$$

Bei einer Nutzung von 9, 16 und 27 virtuellen Prozessoren pro Prozessor[18] ergeben sich aus diesem Ausdruck die in Tabelle 4 dargestellten Verhältnisse. Bei 9 virtuellen Pro-

f_1/f_2	k=0	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8
n=9	1	1.25	1.57	2.0	2.6	3.5	5	8	17
n=16	1	1.13	1.29	1.46	1.67	1.91	2.2	2.56	3
n=27	1	1.08	1.16	1.25	1.35	1.45	1.57	1.7	1.84

Tabelle 4: Ausgleichbares Verhältnis der Rechengeschwindigkeiten in Abhängigkeit von den zu verschiebenden virtuellen Prozessoren.

zessoren kann also minimal ein Leistungsunterschied von 25% ausgeglichen werden, ein feinerer Unterschied kann nur durch den Einsatz einer größeren Anzahl von virtuellen Prozessoren berücksichtigt werden. Allerdings ist gerade bei Metacomputing der Unterschied zwischen den Prozessoren sehr groß. Tabelle 5 stellt die maximale Leistung in Mflops der

	Cray T90	Cray T3E-256	Cray T3E-512	IBM SP2
Maximale Leistung [Mflops]	1800	900	600	225
Verhältnis zur SP2	8	4	2.67	1

Tabelle 5: Unterschiedliche Rechenkapazität jeweils eines Prozessors der verschiedenen Zielmaschinen.

Maschinen, die in dieser Arbeit berücksichtigt werden sollten, gegenüber. Zum jeweiligen Verhältnis der Rechenzeiten kann man in Tabelle 4 eine Anzahl n von virtuellen Prozessoren und ein k so wählen, daß die ungleiche Rechenleistung gut ausgeglichen wird. Wie

[17] Diese Voraussetzung ist bei vielen Anwendungen aus dem Bereich des Hochleistungsrechnens erfüllt, da meist die Anzahl der Operationen proportional zu der Anzahl der Daten ist und die initiale Datenverteilung eine Gleichverteilung der Daten auf die Prozesse vornimmt.

[18] Vgl. die Hinweise zu der Anzahl der virtuellen Prozessoren in Abschnitt 5.6.

zu sehen ist, liegt das Verhältnis der Rechenkapazitäten in einem Bereich, der durch den Austausch von virtuellen Prozessoren bei der Nutzung von neun virtuellen Prozessoren pro physikalischem Prozessor ausgeglichen werden kann. Damit ist die Anwendbarkeit des Konzeptes zum statischen Lastausgleich für die angestrebte Metacomputing-Umgebung gezeigt. Durch Simulation einer größeren Anzahl von virtuellen Prozessoren auf einem Prozessor ist aber auch ein noch feinerer Lastausgleich möglich, ohne daß eine Änderung an der Anwendung durchgeführt werden muß. Man muß allerdings beachten, daß durch das Reduzieren der virtuellen Prozessoren auf einem Prozessor die Möglichkeiten zur Überlappung von Kommunikation und Berechnung evtl. genommen werden können. Abschnitt 8.5 faßt die Ergebnisse zusammen, die mit dem statischen Lastausgleichsverfahren erhalten wurden.

5.5 Dynamischer Lastausgleich

Während der vorgestellte statische Lastausgleich für viele Anwendungen ausreicht, zeigen sich doch in er Praxis deutliche Grenzen. Wenn die Rechenzeit, die ein virtueller Prozessor braucht, sich im Laufe der Zeit ändert, reicht ein statisches Verfahren nicht mehr aus, um eine gleichmäßige Lastverteilung zu ermöglichen. Dies ist z.B. bei adaptiven Verfahren der Fall, bei denen sich zur Laufzeit die Diskretisierung der Problemstellung und damit die notwendige Rechenzeit ändern kann, um eine gewisse Rechengenauigkeit einhalten zu können [59].

In derartigen Fällen ist ein dynamisches Lastausgleichsverfahren notwendig, bei dem zur Laufzeit des Programms kontrolliert wird, ob eine gleichmäßige Lastverteilung vorliegt. Wenn dies nicht der Fall ist, wird versucht, durch eine Verschiebung der Last eine bessere Lastverteilung zu erreichen [95]. Um ein dynamisches Lastausgleichsverfahren zu implementieren, müssen folgende Punkte berücksichtigt werden:

- Es muß festgelegt werden, in welcher Form Last umverteilt werden kann. Es können z.B. ganze virtuelle Prozessoren zwischen den Prozessoren migriert werden, oder nur Daten.

- Es muß ein Maß für die Last eines Prozessors definiert werden. Die dafür notwendigen Daten der einzelnen Prozessoren müssen zur Laufzeit gesammelt und verwaltet werden. Hierbei muß ein Kompromiß zwischen der Genauigkeit des Lastmaßes und dem Aufwand zur Ermittlung der Daten und Berechnung des Maßes gefunden werden, da ansonsten die Gefahr besteht, daß durch die Datenerfassung und Verwaltung mehr Zeit verloren geht, als durch den Lastausgleich gewonnen wird. Erfolgt die Datenerfassung hingegen zu selten, kann keine zuverlässige Aussage über den momentanen (und damit entscheidenden) Lastzustand eines Prozessors gemacht werden.

- Anhand der vorliegenden Lastdaten muß entschieden werden, zwischen welchen Prozessoren ein Lastungleichgewicht überprüft werden soll und zwischen welche Prozessoren Last umverteilt werden soll. Hier unterscheidet man grob die beiden Klassen zentrale und lokale Lastausgleichsverfahren.

- Dynamische Lastausgleichsverfahren setzen voraus, daß die Daten, die im Verlauf der bisherigen Rechnung gemessen wurden, Aussagen über das zukünftige Verhalten der Anwendung erlauben. Während in homogenen Systemen damit die Rechenzeit einzelner Prozesse auf anderen Prozessoren bekannt ist, gilt dies bei einem heterogenen System nicht. Hier ist eine Laufzeitvorhersage notwendig, um eine gute Umverteilung der Last vornehmen zu können.
- Schließlich muß anhand der vorliegenden Daten berechnet werden, wann eine Lastumverteilung durchgeführt werden soll. Dazu muß sowohl die Menge der umzuverteilenden Last als auch die Prozessoren, zwischen denen Last verschoben werden soll, ausgewählt werden.

Diese einzelnen Punkte werden in den folgenden Abschnitten für den dynamischen Lastausgleich eines CoFu-Programms definiert.

5.5.1 Form der Lastmigration

Analog zum Vorgehen beim statischen Lastausgleich (vgl. Abschnitt 5.4) wird auch hier ein virtueller Prozessor als Einheit für den Lastausgleich gewählt. Um eine Migration eines virtuellen Prozessors zu ermöglichen, müssen nur die Daten dieses Prozessors verschickt werden. Hier macht sich die Implementierung der virtuellen Prozessoren mittels *user based threads* positiv bemerkbar, da bei allen anderen Implementierungsmöglichkeiten (vgl. Abschnitt 5.2) noch der Stackbereich, Werte von Registern, ggf. sogar der Programmcode migriert werden müßte. Letzteres ist in einer heterogenen Umgebung praktisch nicht möglich. Bei der gewählten Implementierung ist also auch der Migrationsaufwand gering. Um die zu übertragende Datenmenge möglichst gering zu halten, und auch um eine Lösung für eine heterogene Umgebung zu erhalten, muß das Anwendungsprogramm zwei Funktionen für die Migration zur Verfügung stellen: Packen aller Daten in einen zusammenhängenden Speicherbereich und Festlegung der dort enthaltenen Datentypen, und Auspacken der Daten aus einem Speicherbereich. Die Angabe der Datentypen ist wichtig, damit eine ggf. notwendige Datenkonvertierung automatisch vorgenommen werden kann. Dieser Ansatz garantiert, daß wirklich nur die vom Entwickler als notwendig betrachteten Daten migriert werden und daß keine weitere Änderung der Anwendung erforderlich ist, um einen dynamischen Lastausgleich zu ermöglichen. Alle anderen Aufgaben können von dem CoFu-Skelett vorgenommen werden, eine Änderung der Anwendung ist nicht notwendig, um den dynamischen Lastausgleich zu ermöglichen.

5.5.2 Lastmaß

Für einen einzelnen Prozessor wird als Lastmaß die Summe der reinen Rechenzeit der von ihm simulierten virtuellen Prozessoren definiert. Diese Rechenzeit eines virtuellen Prozessors ist bei einem CoFu-Programm die Summe der Bearbeitungszeit der einzelnen CoFu-Funktionen. Kommunikationszeit wird dabei nicht mitgemessen, da aufgrund des CoFu-Modells Kommunikation erst nach dem Ende einer CoFu-Funktion durchgeführt wird. Aufgrund der gewählten Implementierung als funktionales Skelett kann diese Zeit

von dem CoFu-Skelett selbst gemessen werden, ohne daß Änderungen an der Anwendung dafür notwendig wären. Da nur ein Prozeß mit einem *thread* auf jedem Prozessor bearbeitet wird, kann hier die absolute Uhrzeit (*wallclock time*, im Gegensatz zu CPU-Sekunden) gemessen werden.

5.5.3 Zentrale und dezentrale Lastausgleichsverfahren

Für die Auswahl der Prozessoren, die Last abgeben bzw. zusätzliche Last entgegen nehmen müssen, existieren zwei grundsätzliche Konzepte [44]:

Dezentrale Lastausgleichsverfahren: Bei dezentralen Verfahren (oftmals auch lokalen Verfahren genannt) wird ein Lastausgleich nur zwischen benachbarten Prozessoren durchgeführt. Dies betrifft einerseits das Feststellen einer ungleichen Lastverteilung, andererseits aber auch die daraufhin durchgeführte Umverteilung der Last. Dezentrale Verfahren haben den Vorteil, daß sie sehr gut mit der Anzahl der Prozessoren skalieren, da unabhängig von der gesamten Prozessoranzahl nur Informationen der benachbarten Prozessoren, also einer konstanten Menge benötigt werden. Auf der anderen Seite kann es lange dauern, bis tatsächlich ein globales Lastgleichgewicht erreicht wird – insbesondere ist hierzu oftmals eine vergleichsweise große Anzahl von einzelnen Migrationsschritten notwendig [4]. In [7] bzw. [83] sind Beispiele aufgeführt, wie z.B. die Simulation eines Projektileinschlags, bei denen durch ein lokales Verfahren keine gleichmäßige Lastverteilung in einer ausreichend kurzen Zeit hergestellt wird.

Zentrale Lastausgleichsverfahren: Bei zentralen Verfahren (oftmals auch globale Verfahren genannt) existiert eine zentrale Instanz, die Informationen über die Last aller Prozessoren sammelt und zentral die Umverteilung der Last berechnet. Auf der einen Seite können zentrale Verfahren aufgrund der vorhandenen globalen Information sehr schnell eine Ungleichverteilung feststellen und eine gute Umverteilung der Last berechnen. Auf der anderen Seite kann diese zentrale Instanz gerade bei hohen Prozessorzahlen zu einem Engpaß werden, da das Sammeln der Lastinformation aller Rechner zu einer hohen Netzlast und die Berechnung der Migrationen zu einer hohen Rechenlast führen können. Während dieser Zeit sind aber alle anderen Prozessoren untätig, es geht also Rechenkapazität verloren.

Bei einem Metacomputer unterscheiden sich die einzelnen Rechner im allgemeinen deutlich in ihrer Rechenleistung voneinander. Dementsprechend muß auf ein Lastungleichgewicht möglichst schnell mit einer Lastmigration reagiert werden, damit nicht die Rechenkapazität des schnelleren Computers verloren geht. Da dies mit einem lokalen Lastausgleichsverfahren nicht erreicht werden kann, wird ein zentrales Lastausgleichsverfahren eingesetzt. Der bei zentralen Verfahren inhärente Aufwand für das Sammeln der Lastinformation aller Prozessoren bei einer zentralen Instanz kann bei einem CoFu-Modell implementiert werden, indem man den Lastausgleich mit synchronisierender Kommunikation koppelt. Die für den Lastausgleich benötigte Information wird zusammen mit den

Nachrichten für die synchronisierende Kommunikation übertragen. Der ansonsten erhebliche Aufwand für das Sammeln der Lastinformation der einzelnen Prozessoren wird somit erheblich reduziert, insbesondere weil keine eigenen Nachrichten notwendig sind. Der zusätzliche Vorteil ist, daß sich bei einer synchronisierenden Kommunikation alle virtuellen Prozessoren in einem definierten Ruhezustand befinden, nämlich dem Warten auf das Ende der synchronisierenden Kommunikation, so daß ein Wiederaufsetzen eines virtuellen Prozessors auf einem anderen Prozessor ohne Synchronisationsprobleme möglich ist. Dies ist insbesondere wichtig für die Verwaltung der internen Datenstrukturen des CoFu-Skelettes, insbesondere der *routing*-Tabellen. Durch den Lastausgleich zu diesem Synchronisationszeitpunkt ist sichergestellt, daß keine Nachrichten für einen virtuellen Prozessor unterwegs sind, die weitergeleitet werden müßten. Dadurch vereinfacht sich das Migrationsprotokoll.

5.5.4 Laufzeitvorhersage für CoFu-Funktionen mit Hilfe von Prozessorklassen

Um einen Anhaltspunkt über die Rechenzeit einzelner virtueller Prozessoren auf den verschiedenen Rechnern zu erhalten, wird im folgenden ausgenutzt, daß viele Prozesse ein gleichartiges Verhalten zeigen, weil sie (nahezu) gleiche Berechnungen durchführen und lediglich andere Daten benutzen. Derartige virtuelle Prozessoren werden im folgenden zu einer PROZESSORKLASSE zusammengefaßt. Dementsprechend kann davon ausgegangen werden, daß virtuelle Prozessoren in einer Prozessorklasse auf den gleichen Rechnern die gleiche Rechenzeit benötigen. Die Information über die Zugehörigkeit eines Prozessors p zu einer bestimmten virtuellen Prozessorklasse $\mathcal{A}$, abgekürzt $p \in \mathcal{A}$, muß allerdings von der Anwendung selbst zur Verfügung gestellt werden. Als Beispiel betrachte man einen adaptiven Programmcode, bei dem zur Laufzeit bestimmte Gebiete auf einem feineren Gitter berechnet werden müssen, was einen erhöhten Rechenbedarf bedeutet. Anfangs befinden sich alle Prozesse in der gleichen Prozessorklasse A_1. Prozessoren, die eine feinere Diskretisierung benötigen, verlassen die Klasse $\mathcal{A}_1$ und bilden eine neue Klasse $\mathcal{A}_2$. Findet eine erneute Verfeinerung des Gitters eines Prozessors aus der Klasse $\mathcal{A}_2$ statt, so gehört er einer dritten Klasse $\mathcal{A}_3$ an. Prozessoren in der gleichen Klasse werden aber, da sie die gleiche Datenmenge haben und die gleichen Operationen durchführen, ungefähr die gleiche Rechenzeit auf den gleichen Rechnern haben.

Sei nun $t(p, R)$ die Rechenzeit des virtuellen Prozessors p auf dem Rechner R (abgekürzt $p \in R$), die seit dem letzten Lastausgleich gemessen wurde, $t(\mathcal{A}, R)$ der Durchschnitt aller Rechenzeiten von virtuellen Prozessoren aus der Klasse $\mathcal{A}$ auf dem Rechner R. Der Ausdruck $t(\mathcal{A}, R)$ ist undefiniert, falls kein virtueller Prozessor $p \in \mathcal{A}$ mit $p \in R$ existiert, also kein virtueller Prozessor der Klasse $\mathcal{A}$ auf R berechnet wird. Sei weiterhin $t_{alt}(\mathcal{A}, R)$ der Wert von $t(\mathcal{A}, R)$ in dem letzten Lastausgleichschritt, in dem $t(\mathcal{A}, R)$ definiert war. Insbesondere ist $t_{alt}(\mathcal{A}, R)$ nur dann undefiniert, wenn noch niemals ein virtueller Prozessor der Klasse $\mathcal{A}$ auf dem Rechner R bearbeitet worden ist. Man betrachte nun einen Prozessor p, der zur Zeit auf dem Rechner R_1 bearbeitet wird und der zur Prozessorklasse $\mathcal{A}$ gehört. Für den Lastausgleich ist nun die Zeit wichtig, die dieser virtuelle Prozessor auf einem Rechner R_2 benötigen wird, also $t(p, R_2)$. Folgende Heuristik gibt eine Abschätzung für

diese prognostizierte Rechenzeit, wobei der erste zutreffende Punkt gewählt wird:

1. Gibt es einen virtuelle Prozessor $p_2 \in R_2$ mit $p_2 \in \mathcal{A}$, so kann $t(p, R_2) = t(\mathcal{A}, R_2)$ angenommen werden (Annahme: Virtuelle Prozessoren der gleichen Prozessorklasse benötigen die gleiche Rechenzeit).
2. Ist $t_{alt}(\mathcal{A}, R_2)$ definiert, so kann $t(p, R_2) = t_{alt}(\mathcal{A}, R_2)$ angenommen werden (Annahme: Rechenzeit ist unabhängig vom Zeitschritt).
3. Gibt es einen virtuellen Prozessor $p_1 \in R_1$, einen virtuellen Prozessor $p_2 \in R_2$ und eine Prozessorklasse $\mathcal{B}$ mit $p_1, p_2 \in \mathcal{B}$, so kann man die Rechenzeit von p auf R_1 abschätzen, indem man das Verhältnis der Rechenzeiten von p_1 und p_2 als Grundlage nimmt, also $t(p, R_2) = t(p, R_1)\frac{t(p_2, R_2)}{t(p_1, R_1)}$ (Annahme: Relative Rechenkapazität der eingesetzten Rechner ist unabhängig von der Prozessorklasse).
4. Gab es bislang keine zwei virtuelle Prozesse der gleichen Klasse auf den zu untersuchenden Rechnern R_1 und R_2, so ist kein Anhaltspunkt über die Rechenzeit von p auf R_2 vorhanden. Um wenigstens eine ungefähr Größenordnung zu haben, setze man $t(p, R_2) = \alpha \cdot t(p, R_1)$, wobei α ein vorgegebenes Verhältnis der Rechenleistung der beteiligten Rechner zueinander ist. Ein solches α kann z.B. aufgrund der Höchstleistung der Rechner oder durch eigene Messungen gewonnen werden.

Durch das Konzept der Prozessorklassen kann nun eine gute Laufzeitvorhersage auch für heterogene Systeme durchgeführt werden. Hierzu ist nur eine minimale Unterstützung bei der Anwendungsentwicklung notwendig, nämlich die Festlegung der Prozessorklasse. Die hierfür notwendigen Daten sind aber im allgemeinen leicht zu erhalten (vgl. obiges Beispiel).

5.5.5 Auswahl der zu migrierenden virtuellen Prozessoren

Da die Berechnung einer möglichst ausgeglichenen Verteilung der virtuellen Prozessoren auf die physikalischen Prozessoren ein NP-vollständiges Problem ist [95], muß eine schnelle Heuristik zur Bestimmung der auszutauschenden Last implementiert werden, da anderenfalls der Prozessor, der die Lastverteilung berechnet, zuviel Rechenzeit dadurch verliert. Diese Heuristik wird im weiteren beschrieben.

Der zentrale physikalische Prozessor, der die weitere Bearbeitung des Lastausgleiches vornimmt (im weiteren „Lastverwalter“ genannt), erhält bei einer synchronisierenden Kommunikation von allen physikalischen Prozessoren folgende Daten:

$R_{P,v}$: Für jeden virtuellen Prozessor v gibt der physikalische Prozessor P, auf dem v bearbeitet wird, die gesamte Rechenzeit $R_{P,v}$ an, die v seit dem letzten Lastausgleichsschritt (bzw. seit dem Programmstart) benötigte. Im allgemeinen wird ein virtueller Prozessor mehrfach lokale Berechnungen durchführen, bevor eine globale Operation ausgeführt wird. Die gesamte Rechenzeit eines virtuellen Prozessors ist dann die Summe der einzelnen Rechenzeiten.

G_P: Jeder physikalische Prozessor P gibt die Gesamtrechenzeit an, die er für die lokalen Berechnungen benötigt hat, also den Wert $G_P = \sum_v R_{P,v}$. Dieser Wert wird

zwar redundant übergeben, da er vom Lastverwalter berechnet werden könnte, dies verkürzt aber die notwendige Rechenzeit für den Lastverwalter.

K_v: Für jeden virtuellen Prozessor v gibt der physikalische Prozessor, auf dem v bearbeitet wird, die Prozessorklasse K_v an, zu der v gehört. Da es möglich ist, daß sich die Zugehörigkeit eines virtuellen Prozessors zu einer Prozessorklasse zur Laufzeit ändert, muß diese Information in jedem Lastausgleichsschritt erneut übertragen bzw. zumindest aktualisiert werden.

Da die Rechenzeit eines virtuellen Prozessors bis zum nächsten Lastausgleichsschritt nicht a priori bekannt ist, wird wie bei fast allen Lastausgleichsstrategien davon ausgegangen, daß die Rechenzeit $R_{P,v}$ seit dem letzten Lastausgleichsschritt einen Anhaltspunkt für das weitere Verhalten des virtuellen Prozessors in der Zukunft gibt. Es wird aber angenommen, daß eine Verteilung der Prozessoren, die eine gute Lastverteilung in der Vergangenheit garantiert hätte, auch für die Zukunft eine gute Lastverteilung herstellt.

Der hier vorgeschlagene einfache Algorithmus basiert auf einem *greedy*-Ansatz [80], also auf einer schrittweisen Reduzierung der Gesamtrechenzeit durch Migration einzelner virtueller Prozessoren. Als Maß für die Güte einer Verteilung der virtuellen Prozessoren auf die physikalischen Prozessoren wird die maximale Gesamtrechenzeit aller Prozessoren betrachtet. Im weiteren wird angenommen, daß eine Reduktion dieses Wertes auch zu einer Reduktion der Rechenzeit führt. Dies muß nicht zwangsläufig der Fall sein, da hier z.B. die Kommunikationszeit und Wartezeit auf Daten anderer Prozessoren nicht berücksichtigt wird. Dieser Ansatz wird aber durch die in Abschnitt 5.2 vorgestellten Optimierungen gerechtfertigt, da die Kommunikationsoperationen durch Berechnungen überlappt werden und somit nicht ins Gewicht fallen.

Durch eine Migration eines virtuellen Prozessors v vom physikalischen Prozessor P_{max} zu einem physikalischen Prozessor P_{Ziel} ergeben sich im $i+1$-ten Schritt voraussichtlich folgende neue Werte für die Parameter G und R:

$$\begin{aligned} G_{P_{max}}^{(i+1)} &= G_{P_{max}}^{(i)} - R_{P_{max},v}^{(i)} \\ R_{P_{Ziel},v}^{(i+1)} &= t(v, P_{Ziel}) \\ G_{P_{Ziel}}^{(i+1)} &= G_{P_{Ziel}}^{(i)} + R_{P_{Ziel},v}^{(i+1)} \end{aligned} \tag{9}$$

wobei $t(v, P)$ die oben definierte Abschätzung der Rechenzeit eines virtuellen Prozessors v auf einem physikalischen Prozessor P darstellt.

Der Algorithmus bestimmt zunächst vom physikalischen Prozessor mit der längsten Gesamtrechenzeit den virtuellen Prozessor, der dort die größten Rechenzeit benötigte. Anschließend wird versucht, für diesen virtuellen Prozessor einen physikalischen Zielprozessor zu finden, so daß die prognostizierte Gesamtrechenzeit durch die in Abschnitt 5.5.4 vorgestellte Funktion für den Zielprozessor kleiner ist als die jetzige Gesamtrechenzeit des Prozessors mit der längsten Rechenzeit. Hierbei werden die Prozessoren bevorzugt, auf denen der zu migrierende virtuelle Prozessor die kleinste prognostizierte Rechenzeit benötigt – die Heterogenität des Systems wird also berücksichtigt und genutzt.

Formal stellt sich der Algorithmus im i-ten Migrationsschritt wie folgt dar:

1. Bestimme den physikalischen Prozessor P_{max}, so daß dessen Gesamtrechenzeit $G^{(i)}_{P_{max}}$ das Maximum aller Gesamtrechenzeiten $G^{(i)}_P$ ist. Falls mehrere Prozessoren dieses Maximum annehmen, wähle einen zufällig aus.

2. Bestimme nun den virtuellen Prozessor v_{mig} von P_{max}, der die größte Rechenzeit auf P_{max} hatte und für den mindestens ein Prozessor P_{Ziel} existiert, so daß Gleichung

$$G^{(i+1)}_{P_{Ziel}} < G^{(i)}_{P_{max}} \tag{10}$$

   erfüllt ist. Falls kein solcher virtueller Prozessor existiert, terminiert der Algorithmus.

3. Von allen physikalischen Prozessoren P, die Gleichung (10) für den virtuellen Prozessor v_{mig} erfüllen, wird derjenige als Zielprozessor ausgewählt, für den $t(v_{mig}, P)$ minimal ist. Gibt es mehrere Prozessoren mit der gleichen voraussichtlichen Rechenzeit für v_{mig}, wird derjenigen ausgewählt, der bislang die geringste Gesamtrechenzeit hatte.

4. Setze $G^{(i+1)}_P = G^{(i)}_P$, $R^{(i+1)}_{P,v} = R^{(i)}_{P,v}$ für alle Prozessoren P und alle virtuellen Prozessoren v. Berechne die neuen voraussichtlichen Zeiten für die physikalischen Prozessoren P_{max} und P_{Ziel} und den virtuellen Prozessor v_{mig} auf dem neuen Prozessor P_{Ziel} gemäß Gleichung (9). Beginne wieder bei Schritt 1.

Dieser Algorithmus garantiert, daß bei jeder Migration die bisherige maximale Ausführungszeit auf einem Prozessor reduziert wird. Hierbei wird insbesondere die Heterogenität des zugrundeliegenden Metacomputers genutzt, indem der Prozessor als Ziel der Migration ausgewählt wird, der für den zu migrierenden virtuellen Prozessor die kürzeste Ausführungszeit verspricht. Virtuelle Prozessoren gelangen somit zu dem physikalischen Prozessor, auf dem sie möglichst schnell ausgeführt werden. Dieser Algorithmus garantiert nicht, daß in jedem Migrationsschritt die maximale Gesamtrechenzeit tatsächlich sinkt: Wenn mehr als ein Prozessor die gleiche maximale Gesamtrechenzeit benötigte, wird zunächst nur von einem dieser Prozessoren ein virtueller Prozessor wegbewegt, das Maximum der Gesamtrechenzeiten aller Prozessoren bleibt zunächst unverändert.

Es bleibt also noch zu zeigen, daß der beschriebene Algorithmus tatsächlich terminiert, wenn eine endliche Anzahl von virtuellen Prozessoren simuliert wird. Dazu sei n die Anzahl der Prozessoren, $G^{(i)}_j$ für $1 \leq j \leq n$ die Gesamtrechenzeit die Prozessor P_j im i-ten Migrationsschritt für die lokale Berechnungen benötigt. Weiter sei $R_{j,v} > 0$ die Rechenzeit des virtuellen Prozessors v auf dem physikalischen Prozessor P_j. O.B.d.A. seien die G_j absteigend sortiert, d.h. $G^{(i)}_1 \geq G^{(i)}_2 \geq \ldots \geq G^{(i)}_n$, so daß in Schritt 1 des Algorithmus Prozessor P_1 ausgewählt wird. Wenn in Schritt 2 kein passender virtueller Prozessor gefunden wird, terminiert der Algorithmus. Anderenfalls wird ein virtueller Prozessor, der auf P_1 die Rechenzeit t_0 benötigte, zu einem Prozessor P_{Ziel} migriert, wo er voraussichtlich die Rechenzeit t_{Ziel} hat. Wegen Gleichung 10 ist $P_{Ziel} \neq P_1$ und es gilt $G^{(i+1)}_1 < G^{(i)}_0$.

Lemma 1. *Nach spätestens $(n-1)$-maliger Wiederholung des Algorithmus wird die voraussichtliche maximale Gesamtrechenzeit reduziert.*

Beweis. **Sei zunächst $G_1^{(i)} > G_2^{(i)}$:**
Es gilt wegen $G_1^{(i)} > G_2^{(i)} \geq \ldots \geq G_n^{(i)}$ für das voraussichtliche Maximum aller Gesamtrechenzeiten:

$$\max_{1 \leq j \leq n} \{G_j^{(i+1)}\} = \max\{G_1^{(i+1)}, G_{Ziel}^{(i+1)}, G_2^{(i+1)}\}. \tag{11}$$

Damit folgt aus Gleichung (9):

$$\begin{aligned} \max_{1 \leq j \leq n} \{G_j^{(i+1)}\} &= \max\{G_1^{(i)} - R_{1,v}^{(i)}, G_{Ziel}^{(i+1)}, G_2^{(i+1)}\} \\ &< G_1^{(i)} = \max_{1 \leq j \leq n} \{G_j^{(i)}\}, \end{aligned}$$

da wegen Gleichung (10) die Abschätzung $G_{Ziel}^{(i+1)} < G_1^{(i)}$ gilt und für $Ziel \neq 2$ nach Voraussetzung gilt: $G_2^{(i+1)} = G_2^{(i)} < G_1^{(i)}$.

Sei nun $G_1^{(i)} = G_2^{(i)} \geq \ldots \geq G_n^{(i)}$.
Hier ergibt sich beim ersten Durchlaufen des Algorithmus keine Reduktion der Gesamtrechenzeit. Die neue Rechenzeit nach der Migration wird durch den Prozessor P_2 mit der Rechenzeit $G_2^{(i)} = G_1^{(i)}$ bestimmt. In diesem Fall wird der Algorithmus wiederholt. Nun haben zu Beginn des i-ten Migrationsschrittes k Prozessoren die gleiche Gesamtrechenzeit und es gilt $G_1^{(i)} = G_2^{(i)} = \ldots = G_k^{(i)} > G_{k+1}^{(i)} \geq \ldots \geq G_n^{(i)}$. Nach einem Migrationsschritt ist die Gesamtrechenzeit $G_1^{(i)}$ von Prozessor P_1 auf $G_1^{(i+1)} < G_1^{(i)}$ reduziert, so daß im nächsten Migrationsschritt $i+1$ nur noch maximal $k-1$ Prozessoren die gleiche Rechenzeit haben. Bei der $(k-1)$-ten Wiederholung ist wird das Maximum der Gesamtrechenzeiten von genau einem Prozessor angenommen. Folglich sind im $(i+k-1)$-ten Migrationsschritt die Voraussetzungen dieses Falles nicht mehr zutreffend und es ergibt sich, wie oben gezeigt, eine echte Reduzierung des Maximums alles Gesamtrechenzeiten. Da $k \leq n-1$ gilt (ansonsten läge bereits eine ideale Verteilung der Rechenzeiten vor, da dann $G_1^{(i)} = G_2^{(i)} = \ldots = G_n^{(i)}$ gilt), wird nach maximal $n-1$ Migrationsschritten eine Verkürzung der gesamten Rechenzeit eintreten. □

Wegen Lemma 1 muß die maximale Gesamtrechenzeit nach spätestens $(n-1)$-maliger Wiederholung sinken. Da die maximale Gesamtrechenzeit nach unten durch 0 beschränkt ist und sich dieses Maximum als Summe der Rechenzeiten endlich vieler virtueller Prozessoren ergibt, muß der Algorithmus terminieren, was zu zeigen war.

Man beachte, daß nicht bewiesen wurde, daß die minimale Gesamtrechenzeit erreicht wird, sondern nur, daß durch weitere einzelne Migrationen keine weitere Reduzierung mehr möglich ist. Die hier eingesetzte Heuristik kann ein lokales Minimum finden, das durch Migration eines einzelnen virtuellen Prozessors nicht verbessert werden kann. Als einfaches Beispiel betrachte man Abbildung 23 a). Dort bearbeiten zwei physikalische Prozessoren P_1 bzw. P_2 jeweils einen virtuellen Prozessor v_1 bzw. v_2. Dabei habe v_i ($i \in \{1,2\}$) eine Bearbeitungszeit von zwei Sekunden auf P_i und eine Bearbeitungszeit von einer Sekunde

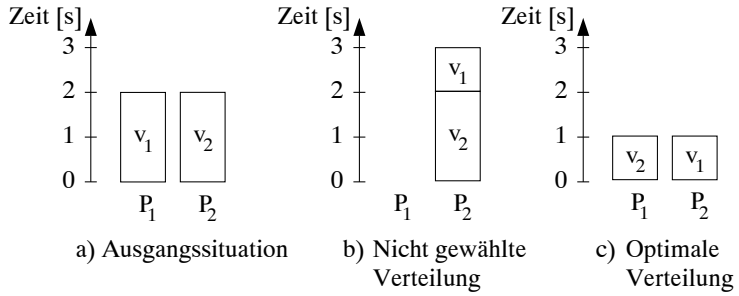

Abbildung 23: Beispiel eines Lastverteilung, die durch den vorgestellten Algorithmus nicht optimiert wird.

auf Prozessor P_{3-i}. In einem solchen Fall wird durch den vorgestellten Algorithmus keine Migration durchgeführt, da durch jede einzelne Migration die Gesamtbearbeitungszeit von zwei Sekunden auf drei Sekunden ansteigen würde, vgl. Abbildung 23 b). Durch einen Tausch der beiden virtuellen Prozessoren hingegen gibt sich die minimale Bearbeitungszeit von einer Sekunde, wie in Abbildung 23 c) dargestellt.

Der vorgestellte Algorithmus macht zahlreiche Vereinfachungen, damit nicht zuviel Zeit für die Berechnung der Lastumverteilung verloren geht. Er betrachtet nicht die einzelnen Rechenzeiten jeder lokalen Berechnung eines virtuellen Prozessors, sondern nur deren Summe. Zeigen virtuelle Prozessoren ein alternierendes Verhalten, also z.B. abwechselnd viel und wenig Rechenzeit, wird dies nicht berücksichtigt. Auch wird die Nachbarschaftskommunikation nicht beachtet - weder als verlorene Rechenzeit, noch wird bei der Migration eine Minimierung der Kommunikationskosten berücksichtigt. Dafür berücksichtigt der Algorithmus auf einfache Weise die Heterogenität des zugrundeliegenden Metacomputers und erlaubt aufgrund des Prinzips der Prozessorklassen eine Laufzeitvorhersage der jeweiligen Rechenschritte in einem heterogenen System, was für den Lastausgleich eine entscheidende Bedeutung hat.

5.6 Empfehlungen für die Anzahl der virtuellen Prozessoren

Die Anzahl der virtuellen Prozessoren ist eine wichtige Größe: zu viele virtuelle Prozessoren können ggf. die Konvergenzgeschwindigkeit eines Algorithmus negativ beeinflussen und einen zu hohen Verwaltungsaufwand verursachen, zu wenige virtuelle Prozessoren hingegen lassen keine ausreichende Überlappung zu. Für zweidimensionale, reguläre Probleme wie in Abbildung 22 auf Seite 62 sind 9 oder 16 virtuelle Prozessoren eine geeignete Anzahl, um die angestrebte Überlappung zu erreichen, da dann ein oder vier virtuelle Prozessoren vorhanden sind, die nur lokal kommunizieren müssen. Für dreidimensionale Probleme hingegen sind 27 virtuelle Prozessoren notwendig, damit tatsächlich ein virtueller Prozessor im Inneren eines $3 \times 3 \times 3$ Würfels vorhanden ist, der nur lokal kommuniziert. In Abschnitt 8.2 wird der Aufwand gemessen, der durch die virtuellen Prozessoren verursacht wird. Die dort ermittelten Ergebnisse sind ein Anhaltspunkt für die empfehlenswerte Anzahl der virtuellen Prozessoren.

6 Verwandte Arbeiten

Dieses Kapitel vergleicht diese Arbeit mit anderen Arbeiten, die in dem gleichen oder einem verwandten Gebiet veröffentlicht worden sind. Neben der Abgrenzung der hier vorgestellten Inhalte werden aber auch Gemeinsamkeiten herausgearbeitet, da sich oftmals eine Kopplung bzw. Nutzung von weiteren Ideen anbietet.

6.1 Erfahrungen mit Metacomputing

Die meisten Experimente werden im Rahmen sogenannter Testbeds durchgeführt. Testbeds sind dabei in erster Linie durch ein WAN gekennzeichnet, das verschiedene Rechnerstandorte miteinander verbindet. Oftmals ist die Leitung dediziert für einzelne Projekte vorhanden, so daß ideale Rahmenbedingungen vorliegen, da Einflüsse durch andere Benutzer nahezu ausgeschlossen sind[19]. Während eine solche Umgebung ideal für Testzwecke und Evaluierungen ist, so muß doch damit gerechnet werden, daß die Ergebnisse in der Praxis eher schlechter ausfallen werden.

Im folgenden werden zuerst die wichtigsten Testbeds kurz mit ihren technischen Rahmenbedingungen vorgestellt. Dabei werden nur kurz die beteiligten Institutionen[20], das genutzte WAN und die wichtigsten Projektziele erwähnt. Meist besteht ein Projekt, das ein solches Testbed zur Verfügung hat, aus mehreren Teilprojekten, von denen sich nur ein Teil mit Metacomputing im hier definierten Sinne beschäftigt. Die ausführlichere Diskussion der jeweiligen Projekte findet in den Abschnitten 6.1.1 bis 6.1.3 statt.

Das AURORA-Testbed: 1990 wurde in den USA das Projekt *Gigabit Testbed Initiative* zur Untersuchung von Gigabit-Netzwerken der *Corporation for National Research Initiatives* (CNRI) mit Förderung von der *Advanced Research Projects Agency* (ARPA) und der *National Science Foundation* (NSF) begonnen. In diesem Rahmen wurden insgesamt fünf Testbeds initiiert, die alle unterschiedliche Schwerpunkte hatten [17, 129].

Das AURORA-Testbed war eines dieser fünf Testbeds. Es umfaßte die Standorte

- *Bellcore's Morristown Research and Engineering Laboratory*,
- *IBM T.J. Watson Research Center*,
- *Massachusetts Institute of Technology (MIT), Laboratory for Computer Science* und
- *University of Pennsylvania's Distributed Systems Laboratory*,

die mittels drei paralleler OC-12 Leitungen, also ATM Verbindungen mit 622 MBit/s, verbunden waren. Der Schwerpunkt von AURORA lag dabei auf der Untersuchung der Netzwerktechnik, in erster Linie im Vergleich zwischen einer Netzverbindung mit fester und mit variabler Paketgröße anhand der beiden Protokolle *Asynchronous Transfer Mode*

[19] Allerdings werden oftmals noch Teile des lokalen LANs genutzt, bis der Übergang zum dedizierten WAN erreicht wird, so daß hierdurch eine Beeinflussung durch andere Anwendungen durchaus möglich ist.

[20] Bei allen Aufzählungen der beteiligten Institutionen wurden nur die genannt, die an der Nutzung der Netze beteiligt waren, nicht die Gesellschaften, die die Netze zur Verfügung stellen bzw. gestellt haben.

(ATM) und *Packet Transfer Mode* (PTM) [15]. Auch der Einsatz von verteiltem gemeinsamen Speicher (*distributed shared memory*) für den Einsatz in Gigabitnetzen wurde getestet, hierzu dienten Anwendungen, die aus dem Bereich der Multimediatechnik stammen. Eine echte verteilte Berechnung fand jedoch nicht statt [15].

Blanca Testbed: Das Blanca-Testbed war das zweite der fünf von der CNRI geförderten Gigabit-Projekte. Beteiligte Institutionen waren

- das *Lawrence Berkeley Laboratory* (LBL),
- das *National Center for Supercomputing Applications* (NCSA),
- die *University of California at Berkeley*,
- die *University of Illinois at Urbana-Champaign* und
- die *University of Wisconsin at Madison*.

Die jeweiligen Standorte wurden zunächst mittels einer 45–MBit/s–Leitung, die später auf eine Kapazität von 622 MBit/s erweitert wurde, miteinander verbunden. Neben Multimedia-Anwendungen wurde auch hier die Netzwerktechnik untersucht und optimiert. Hierzu gehört insbesondere das *Xunet OSI Management System* [1], das auch die Beobachtung- und Kontrolle des Netzwerkes auf der Ebene einzelner ATM-Zellen ermöglichte. Die Anwendungen stammten in erster Linie aus dem Bereich der Visualisierung und der „digitalen Multimedia-Bibliothek“, der „entfernten“ (*remote*) Kontrolle von Instrumenten und großen numerischen Anwendungen, z.B. aus dem Bereich der Atmosphärenforschung.

CASA Gigabit Network Testbed: Das CASA Gigabit Testbed mit einer Laufzeit von vier Jahren war das dritte der fünf Gigabit-Testbeds.

Beteiligte Institutionen waren das

- *California Institute of Technology* (Caltech),
- das *Jet Propulsion Laboratory* (JPL),
- das *Los Alamos National Laboratory* (LANL),
- das *San Diego Supercomputer Center* (SDSC) und
- das *Atmospheric Sciences Department* der *University of California at Los Angeles* (UCLA).

Diese Institutionen waren über jeweils acht parallele SONET OC-3 Kanäle (*Synchronous Optical Networks*) miteinander verbunden, von denen jeder eine Kapazität von 155.52 MBit/s hatte. Von diesen acht Kanälen wurden sechs für die eigentliche Datenübertragung, einer für die Fehlerkorrektur und der letzte als Ersatz für einen evtl. ausfallenden anderen Kanal eingesetzt. Die jeweiligen Rechner wurden über eine HiPPI-Schnittstelle (*High-Performance Parallel Interface*) an das SONET-WAN angeschlossen. Die Schnittstellen zum Übergang von HiPPI nach SONET wurden vom LANL entwickelt [73]. HiPPI erlaubt eine Bandbreite von ca. 800 MBit/s, die über die sechs SONET OC-3 Kanäle ohne Verlust übertragen werden konnten.

Neben der Hardware wurde in dem Testbed eine Netzwerk-Programmierumgebung ent-

wickelt, indem die Express-Umgebung [79] erweitert wurde, so daß'sie auch in heterogenen Systemen einsetzbar war. Insbesondere wurde für die externe Kommunikation nicht nur TCP/IP, sondern auch das HiPPI-Protokoll unterstützt. Wie schon im Regionalen Testbed NRW (vgl. Kapitel 3) zeigt sich im Vergleich die geringe Leistungsfähigkeit des TCP/IP-Protokolls: die Bandbreite sank von ca. 348 MBit/s über HiPPI auf 46 MBit/s bei TCP/IP über die HiPPI-Verbindung. Auch stieg die Latenz von 5.2 ms auf 13.6 ms an. Als Ursache hierfür wurde einerseits die Implementierung von TCP/IP auf den verwendeten Rechnern, die zu einem großen Aufwand und somit zu einer hohen Latenz führten, andererseits die geringe Paketgröße von TCP/IP festgestellt [79]. Hierdurch wird die Bedeutung des Versendens von großen Nachrichten und der Überlappung von Kommunikation und Berechnung, die durch die in Kapitel 5 vorgestellten Optimierungen erreicht werden, bestätigt.

Gigabit Nectar: Gigabit Nectar ist das vierte der CNRI geförderten amerikanischen Testbeds. Beteiligt waren

- die *Carnegie Mellon University* (CMU) und
- das *Pittsburgh Supercomputing Center* (PSC),

die knapp 30 km voneinander entfernt sind. Dementsprechend ist dieses Netzwerk eher ein MAN (*Metropolitan Area Network*) als ein WAN [16]. Die beiden Standorte waren durch eine 2.4–GBit/s–ATM Leitung miteinander verbunden. In dem Gigabit Nectar Projekt sollten in erster Linie die Entwicklung von effizienten Protokollen einschließlich Hardwareunterstützung und das Einbinden dieser Protokolle in das Betriebssystem vorgenommen werden [125]. Auch die Unterstützung durch entsprechende Programmierumgebungen war ein wichtiges Ziel. Schließlich wurden auch verteilte Berechnungen im Bereich der Chemie und der Visualisierung vorgenommen [16].

Vistanet: VISTAnet ist das letzte der fünf Gigabit-Testbeds, die 1990 von der CNRI gegründet wurde. VISTAnet ist zusammen mit dem Gigabit Nectar Projekt das kompakteste der Testbeds, da auch hier weniger als 30 km Glasfaserleitung benötigt wurden, um die beteiligten Institutionen,

- das *Microelectronics Center of North Carolina* (MCNC),
- die *North Carolina State University* und
- die *University of North Carolina at Chapel Hill*

miteinander zu verbinden [2]. Als Netzwerk-Protokoll wurde HiPPI über eine OC-12c SONET-ATM-Leitung benutzt, wodurch eine maximale Kapazität von 622 MBit/s zur Verfügung stand. Die Anwendungen stammten zum größten Teil aus dem Bereich der Medizin und befaßten sich mit der Planung von medizinischen Behandlungen im Bereich der Strahlentherapie.

BATMAN: Das *Boulder ATM Access Node*-Projekt (BATMAN) wurde im März 1994 von *USWest Advanced Technologies* ins Leben gerufen und umfaßte eine Vielzahl von

staatlichen, industriellen, öffentlichen und privaten Teilnehmern, vgl. [40]. Diese Institutionen wurden über eine OC-3 ATM-Leitung mit einer Kapazität von 155 MBit/s miteinander verbunden. Neben interaktiven Anwendungen wie der Zugang von Schulen zu staatlichen graphischen Datenbanken, Zugriff auf Daten von Bibliotheken und elektronischen Zeitschriften waren aber auch verteilte Berechnungen aus dem Bereich der *grand challenges* ein Ziel dieses Testbeds [85].

I-WAY: Im Rahmen des I-WAY-Projektes wurden 17 Supercomputing-Zentren, fünf *Virtual Reality Research Sites*, zwölf ATM Testbeds und mehr als 60 Anwendungsgruppen mittels ATM OC-3 Leitungen mit einer Kapazität von 155 MBit/s miteinander verbunden. Der I-WAY war eine Testumgebung für verteilte Berechnungen und für Visualisierung. Zum Bereich der Visualisierung gehörten insbesondere *virtual reality* Anwendungen [25]. Eine Übersicht über die dort behandelten Projekte findet sich in [24]. Ein Schwerpunkt war aber auch die Entwicklung von Programmierwerkzeugen und -bibliotheken, die die Nutzung eines solchen großen verteilten Systems ermöglichte [45].

National Technology Grid: Als Fortsetzung des I-WAYs ist das *National Technology Grid* gedacht [127]. Dies ist in erster Linie eine Kopplung der verschiedenen amerikanischen Netze zuerst mittels OC-12 ATM-Verbindungen (622 MBit/s), die bis zum Jahr 2002 auf 2.4 GBit/s aufgerüstet werden sollen. Dieses Netzwerk soll in zahlreichen Anwendungen genutzt werden, die von sogenannten *Application Technology Teams* entwickelt werden [87, 105].

E=MC^2: In dem Projekt *European Meta Computing Utilising Integrated Broadband Communications* ($E = MC^2$) waren die Institutionen

- *Queen's University of Belfast*,
- *Manchester HPC*,
- *Telmat Soultz*,
- GMD Bonn und
- IPVR Stuttgart

mittels einer ATM Leitung verbunden, auf der eine Bandbreitenreservierung von 2 MBit/s vorgenommen wurde [113]. Untersuchungsschwerpunkte waren das „entfernte“ Ausführen von Programmen auf anderen Höchstleistungsrechnern und verteilte Berechnungen, wobei hier viele industrielle Programme getestet wurden.

***Distributed High Performance Computing*:** In Australien wird im Rahmen des *Distributed High Performance Computing* (DHPC) Projektes zwischen der *Australian National University* in Canberra und der *University of Adelaide* eine OC-3-ATM-Leitung mit einer Bandbreite von 155 MBit/s genutzt. Mittels der Bandbreitenreservierung von ATM stehen davon für DHPC dediziert 12 MBit/s zur Verfügung [71]. Die untersuchten Anwendungen befassen sich mit der Auswertung von Satellitendaten und dem Nutzen dieser Informationen für die Wettervorhersage und Klimasimulationen [132].

Stuttgart-Pittsburgh: Die Universität Stuttgart und das *Pittsburgh Supercomputing Center* arbeiten in einem Metacomputing-Projekt zusammen, bei dem in erster Linie Anwendungen aus dem Bereich der Strömungsmechanik auf den beiden Cray T3Es eingesetzt werden soll. Aufgrund der großen Entfernung spielt hierbei die Latenz (in der Größenordnung von 40 ms) eine entscheidende Rolle [115].

Regionales Testbed NRW: Die genauen technischen Rahmenbedingungen des Projektes „Regionales Testbed Nordrhein-Westfalen" wurden bereits in Kapitel 3 dargestellt. Die vom DFN-Verein zur Verfügung gestellte 34-MBit/s-ATM-Leitung wurde in insgesamt vier Teilprojekten genutzt [29].

Gigabit-Testbed West: In diesem, im August 1997 begonnen Projekt wird zunächst eine 622-MBit/s-ATM-Leitung zwischen dem Forschungszentrum Jülich und der GMD genutzt, die im August 1998 auf 2.4 GBit/s aufgerüstet wurde. Die untersuchten Anwendungen fallen hauptsächlich in die beiden Bereiche verteiltes Rechnen und Visualisierung [33, 34, 35].

6.1.1 Bearbeitung und Visualisierung großer Datenmengen

Häufig müssen große Datenmengen, die z.B. Ergebnis einer Simulation oder eines Experimentes sind, ausgewertet und visualisiert werden. Solange der Rechenaufwand für die Auswertung bzw. Visualisierung von einem einzelnen Arbeitsplatzrechner oder Graphikrechner bewältigt werden kann, wird keine verteilte Berechnung, sondern nur ein ggf. nicht-lokaler Datenzugriff auf große Datenmengen vorgenommen. Beispiele dafür sind Videokonferenzen (Aurora [15]), Zugriffe auf entfernte Datenbanken (BATMAN [40]), etc. Da dieser Bereich mit dieser Arbeit keinen Zusammenhang hat, wird hier nicht weiter darauf eingegangen.

Häufig ist die Auswertung und Visualisierung der Daten jedoch selbst ein rechenaufwendiger Prozeß, der die Leistungsfähigkeiten eines einzelnen Rechners überschreitet. Ein Beispiel sind die Daten, die das *Spaceborn Imaging Radar-C/X-band Synthetic Aperture Radar* (SIR-C/S-SAR) auf seinen beiden Flügen an Bord des *Spaceshuttles Endeavor* 1994 gemessen hat. Die Auswertung dieser Daten auf einem Arbeitsplatzrechner würde beinahe 2 Jahre in Anspruch nehmen [91]. Dieser hohe Rechenbedarf führt dazu, daß bisher weniger als 10% der Daten, die von Satelliten wie SEASAT, Magellan oder Landsat aufgenommen wurden, analysiert werden konnten [6]. Derartige Problemstellungen sind Gegenstand zahlreicher Projekte im Bereich des verteilten Rechnens, für die der Begriff *visual supercomputing* [75] geprägt wurde. Weitere Beispiele, die bereits in den verschiedensten Testbeds untersucht wurden, sind:

- Visualisierung dreidimensionaler, zeitabhängiger Simulationen meteorologischer und luftchemischer Vorgänge in der Atmosphäre (RTB NRW [51])
- Analyse von Flugzeugtriebwerken (RTB NRW [135])
- Auswertung von elektronenmikroskopischer Tomographien (*electron microscope tomography*) [91]

- die Auswertung von Satelliten-Meßdaten der Erde (Casa-Testbed [6])
- Therapie-Planung (VISTAnet [2])
- Zahlreiche weitere Beispiele wurden auf der Supercomputing 95 im Rahmen der *Global Information Infrastructure* Demonstration gezeigt, die das I-WAY Testbed nutzte ([24]).

Die hier genannten Beispiele beschränken sich dabei nicht nur auf einen schnellen Zugriff auf „entfernte“ Daten oder experimentelle Geräte, sondern es sind aufwendige Berechnungen notwendig, die auf einzelnen Arbeitsplatzrechnern nicht mehr in akzeptabler Zeit durchgeführt werden können. Insbesonders die häufig gewünschte interaktive Auswertung erfordert den Einsatz von Höchstleistungsrechnern.

Als ausführlicheres Beispiel für eine derartige Verteilung sei das CALCRUST-System genannt, das im Rahmen des CASA-Testbeds bearbeitet wurde [6]. Hier wurden interaktiv Daten visualisiert, die von verschiedenen Satelliten gesammelt wurden. Eine entscheidende Anforderung an das Gesamtsystem war dabei das interaktive Arbeiten, das es den Wissenschaftlern ermöglicht, durch die visualisierten Daten zu „fliegen“ und Punkte auszuwählen, die detailierter ausgewertet werden müssen. Hierzu wurden zunächst eine Cray Y-MP am JPL bzw. eine Cray C-90 am SDSC mit einer Intel Delta am Caltech gekoppelt, wo die Anzeige auf dem SUN-*front-end* der Delta durchgeführt wurde. Der Cray Rechner war dabei für die zweidimensionale Bearbeitung der Daten zuständig, die Delta für die dreidimensionale. Die benötigte Rechenzeit wurde dabei aber stark von der Zeit für die zweidimensionale Auswertung bestimmt, so daß später die Cray C90 am SDSC bzw. die LANL CM-5 oder Y-MP zusätzlich benutzt wurden, um diesen Engpaß weiter zu reduzieren [6]. Diese Anwendung ist also dadurch charakterisiert, daß wie beim *pipelining* ein Datensatz aufeinanderfolgende Rechenstufen durchläuft, bis zum Schluß das darzustellende Bild berechnet worden ist.

Diese Struktur weisen viele Anwendungen aus dem Bereich der Visualisierung auf, da dort verschiedene Filter nacheinander auf die anzuzeigenden Daten angewendet werden. Parallelverarbeitung entsteht dadurch, daß jede Stufe mit dem Bearbeiten des nächsten Datensatzes anfängt, sobald der vorherige Datensatz bearbeitet und weitergeschickt wurde. Diese Art der Parallelität, bei der kein Datenaustausch zwischen den verschiedenen Rechnern, sondern nur ein Datenfluß von einem Rechner zum nächsten Rechner stattfindet, kann mit geringem Aufwand auf ein WAN-gekoppeltes System übertragen werden: Die Übertragungszeiten eines einzelnen Datensatzes kann mit der Berechnung für den nächsten Datensatz überlappt werden, solange die Rechenzeit jeweils größer ist als die Zeit für die Datenübertragung, was im allgemeinen gegeben ist. Hierbei wird allerdings die Bearbeitungszeit durch die langsamste Komponente dominiert, weshalb ein statischer Lastausgleich zwischen den einzelnen Komponenten wichtig ist.

Während das einfache CoFu-Modell in erster Linie datenparallele Programme unterstützt, ist das in Abschnitt 4.3 vorgestellte hierarchische Modell gerade für die Implementierung derartige Anwendungen gut geeignet. Abbildung 24 beschreibt schematisch eine Implementierungsmöglichkeit einer Visualisierungsanwendung. Die CoFu-Funktion F_1 liest den nächsten anzuzeigenden Datensatz von einem Sekundärspeichermedium ein, F_2 und F_3 bereiten die Daten auf und F_4 zeigt diese Daten schließlich auf dem Bildschirm an und

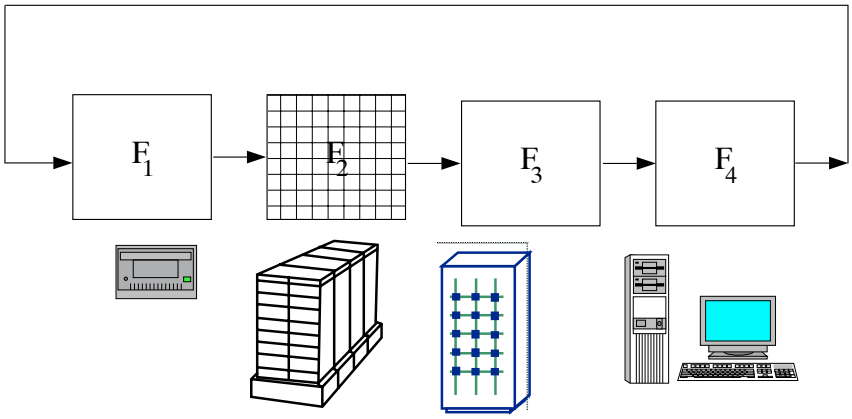

Abbildung 24: Schematisches Vorgehen bei der Parallelisierung eines Visualisierungsanwendung mittels CoFu.

überträgt die Befehle des Anwenders an F_1, die daraufhin den richtigen nächsten Datensatz auswählen kann. Die wichtige Voraussetzung (vgl. Abschnitt 4.1), daß jede der Funktionen F_1 bis F_4 synchron berechnen und Daten verschicken, ist erfüllt. Wenn nun, wie in der Abbildung dargestellt, F_2 selbst wieder ein paralleles Programm ist, kann dieses wiederum als CoFu-Funktion programmiert werden. Die Ausgabedaten von F_1 werden in die Eingabepuffer der einzelnen CoFu-Prozessoren geschrieben und werden dann von den entsprechenden CoFu-Funktionen gelesen und verarbeitet. Die Ergebnisdaten in den Ausgabepuffern wiederum werden zur CoFu-Funktion F_3 gesendet. Wichtig ist hierbei, daß die virtuellen Prozessoren in F_2 unabhängig von F_1 bis F_4 arbeiten können. Sie können mehrere Berechnungs- und Kommunikationsschritte durchführen, während F_1, F_3 und F_4 nur eine Berechnung durchführen.

Zusätzliche Vorteile bietet die Modellierung mit CoFu, wenn mehrere parallele Komponenten vorhanden sind, also z.B. in Abbildung 24 neben F_2 auch noch F_3. Der Lastausgleich kann zwischen den einzelnen Rechnern durchgeführt werden – durch den Einsatz des dynamischen Lastausgleiches sogar automatisch von dem CoFu-Skelett, ohne daß die Anwendung dies besonders vorsehen müßte[21].

Das hierarchische CoFu-Modell ist also sehr gut geeignet, um Anwendungen aus dem Bereich der Visualisierung zu optimieren. Aber auch schon das implementierte einfache CoFu-Skelett kann eingesetzt werden, um die eigentliche Berechnung weiter zu beschleunigen, indem einzelne parallele Teile mit CoFu implementiert werden.

6.1.2 *Multi-Physics*-Anwendungen

Dank der Leistungsstärke moderner Rechner ist es möglich geworden, sogenannte *multi-physics-* oder *coupled-fields*-Anwendungen [8, 86] zu berechnen, also Anwendungen, bei

[21] Die vorgenommene prototypische Implementierung des CoFu-Skelettes unterstützt allerdings noch kein hierarchisches CoFu-Modell, vgl. Abschnitt 5.1.

denen einzelne, ursprünglich eigenständige Programme zur Simulation eines größeren Gesamtsystems gekoppelt werden. Dadurch wird eine genauere Simulation des Gesamtsystems erreicht, da z.B. die Randbedingungen nicht mehr künstlich von außen vorgegeben werden müssen, sondern von den anderen Komponenten der Simulation zur Verfügung gestellt werden. Ein Beispiel für eine derartige Anwendung ist die Wettervorhersage, bei der eigenständige Programme die Meere, die Atmosphäre und die Eiszonen simulieren [35]. Erst durch die Kopplung dieser einzelnen Simulationen wird eine realistische Wettervorhersage möglich.

Bedeutsam ist hierbei, daß die einzelnen Programme, die gekoppelt werden, im allgemeinen völlig unterschiedliche Algorithmen nutzen. Diese Unterschiede können sich in einer feinen oder groben räumlichen und zeitlichen Diskretisierung, der Nutzung von adaptiven Verfahren, von expliziten oder impliziten Gleichungslösern etc. zeigen. Dies äußert sich wiederum in unterschiedlichen Ansprüchen an die Rechnerarchitektur, die für eine gute Leistung eingesetzt werden muß. Weiterhin ist zu beachten, daß die einzelnen Teile der Anwendung im allgemeinen nicht eng-gekoppelt sein müssen - der Informationsaustausch zwischen ihnen findet relativ selten statt. Dadurch wirkt sich die hohe Latenz und geringe Bandbreite nur marginal aus. Dies macht *multi-physics*-Anwendungen zu idealen Anwendungen für Metacomputing. Folgerichtig werden bzw. wurden *multi-physics*-Anwendungen in vielen Testbeds untersucht:

- Globale Klimamodellierung mittels einer gekoppelten Atmosphären - und Ozeansimulation (CASA [88], I-WAY [24], Gigabit Tested West [35])
- Simulation der Luftströmung um Tragflächen unter Berücksichtigung der Tragflächenverformung (BATMAN [86]),
- Gekoppelte biologische und physikalische Simulationen von Ökosystemen (I-WAY [24]) und
- Simulationen der Schadstoffausbreitung im Grundwasser (Gigabit-Testbed West [35], I-WAY [24]).

Ein Ansatz zur Unterstützung von *multi-physics*-Anwendungen wurde im Rahmen des CISPAR-Projektes [14] entwickelt. Der Schwerpunkt lag hierbei auf den mathematischen und technischen Problemen, die vor einer derartigen Kopplung gelöst werden müssen. Beispielsweise müssen unterschiedliche Diskretisierungen des Raumes und der Zeit umgerechnet bzw. interpoliert werden. Dies wird erschwert durch adaptive Verfeinerung der Diskretisierung und durch Verschiebung von Daten zu anderen Prozessoren (z.B. wegen einer verbesserten Lastbalancierung), da sich dadurch die Zuordnung der sich jeweils beeinflussenden Daten jederzeit ändern kann. Auch muß berücksichtigt werden, daß die jeweiligen Simulationen das Gitter der anderen Simulation verändern können (z.B. bei der Verformung von Tragflächen aufgrund des Luftstroms). Diese Funktionalität wird in der Kopplungsbibliothek COCOLIB (*Coupled Communications Library*) [8] implementiert, die genau diese Anforderungen erfüllt.

Im Gegensatz zur COCOLIB unterstützt das CoFu-Skelett nicht direkt die Kopplung einzelner Programme durch spezielle Routinen. Die Umrechnung der unterschiedlichen Diskretisierungen muß innerhalb der Anwendung selbst implementiert werden. Dafür bietet die COCOLIB kein eigenständiges Konzept zur Realisierung des Lastausgleiches, den die

Anwendung selbst durchführen muß. Ein Lastausgleich zwischen verschiedenen Maschinen ist dabei praktisch nicht möglich, da die einzelnen Anwendung möglichst unabhängig voneinander weiter einsetzbar sein sollen. Wünschenswert wäre die Kombination von COCOLIB und des CoFu-Modells. Dazu wäre entweder eine signifikante Erweiterung des CoFu-Modells notwendig, oder aber eine Kopplung dieser beiden Bibliotheken. Im letzteren Falle würde, wie schon bei der Visualisierung (vgl. Abbildung 24 auf Seite 79), mittels eines hierarchischen CoFu-Modells der Datenfluß zwischen den Komponenten realisiert, die COCOLIB hingegen würde die Umrechnung der Diskretisierungen etc. vornehmen.

Aber auch ohne die Funktionalität der COCOLIB bietet sich das hierarchische CoFu-Modell zur Programmierung von *multi-physics*-Anwendungen an. Jedes der einzelnen Programme, aus denen sich eine *multi-physics*-Anwendung zusammensetzt, bildet ein eigenständiges CoFu-Programm. Der Datenfluß zwischen diesen Komponenten wird dann über ein hierarchischesCoFu-Modell gesteuert, dessen einzelne Komponenten die eigenständigen CoFu-Programme sind (vgl. Abbildung 24). Besonders interessant ist hierbei, daß einzelnen Komponenten auch auf mehrere Computer verteilt werden können, was oftmals notwendig ist, um einen guten Lastausgleich zwischen den jeweiligen Teilen zu erreichen. Bei der herkömmlichen Verteilung jeweils eines Programms einer Anwendung auf einem Rechner sind die Anzahlen der sinnvoll einsetzbaren Prozessoren für die einzelnen Komponenten miteinander gekoppelt: Setzt man zu viele Prozessoren für eine Komponente ein, müssen diese Teile untätig auf einen anderen Programmteil warten. Verteilt man hingegen die Programme einer Anwendung auf alle Rechner, kann auch die eingesetzte Prozessoranzahl beliebig variiert werden. Diese Möglichkeit wird durch das CoFu-Skelett erstmalig geboten.

6.1.3 Verteilung einer einzelnen Anwendung

Der vielleicht schwierigste Versuch, einen Metacomputer zu nutzen, ist die Verteilung einer einzelnen, eng-gekoppelten Anwendung auf unterschiedliche Rechner. Da die Kommunikation zwischen den einzelnen Teilen hier deutlich häufiger erfolgt, als bei einer *multi-physics*-Anwendungen, wirkt sich die langsame Kommunikation stark aus. Dieser Anwendungsbereich ist jedoch in der Praxis von großem Interesse: Viele homogene Anwendungen haben einen großen Bedarf an Speicherplatz und Rechenkapazität, die den Einsatz von gekoppelten Rechnern wünschenswert machen. Praktische Beispiele zeigen zwar, daß der große Speicher eines Metacomputers genutzt werden kann, die theoretisch hohe Rechenstärke hingegen wirkt sich jedoch meist nicht in einer reduzierten Bearbeitungszeit aus. Oftmals beobachtet man sogar einen Anstieg der Gesamtbearbeitungszeit bei einer Erhöhung der Prozessorzahl in einem Metacomputer. Beispiele für den gewünschten Einsatz und dabei erzielte Ergebnisse sind:

- Die Simulation von einem *Space Shuttle*, wie sie zwischen der Universität von Stuttgart und dem *Pittsburgh Supercomputing Center* durchgeführt wurde [34, 115]. Hierbei wurde bei der Nutzung weiterer Prozessoren nicht nur keine Beschleunigung, sondern ab 16 Prozessoren (der minimal eingesetzten Prozessoranzahl) für den größten getesteten Datensatz sogar eine Verlangsamung der Anwendung beobachtet.

- Die Simulation der Entstehung des Universums [105]. Diese Anwendung soll im Rahmen des *National Technology Grid* bearbeitet werden, Ergebnisse liegen z.Zt. noch nicht vor.

- Berechnung des Wasserflusses im Grundwasser (RTB-NRW, [61]), vgl. auch Abschnitt 3. Hier ergab sich ebenfalls anfangs eine Verlangsamung des Systems bei der Nutzung eines Metacomputers, da sich die hohe Latenz und geringe Bandbreite sehr stark bemerkbar machten [62]. Durch Optimierung der Kommunikationsstruktur und Integration eines statischen Lastausgleichsverfahren konnte jedoch eine Beschleunigung erreicht werden [61], wenn man die Anzahl der Prozessoren im Metacomputer erhöhte. Trotzdem blieb die Gesamtleistung hinter den theoretischen Erwartungen zurück, da sich die Rechenzeit erhöhte, wenn man zu einem homogenen System weitere Prozessoren hinzufügt.

- Die Simulation des Multiphasenflusses von Flüssigkeiten, die von Morton und Tyler auf einer Cray Y-MP und einer Cray T3D des *Arctic Region Supercomputing Center* durchgeführt wurden [96]. Der eigentliche Hintergrund für die Nutzung eines heterogenen Systems lag darin begründet, daß eine vollständige Parallelisierung der Anwendung zu aufwendig war und nur der Teil parallelisiert werden sollte, in dem die Anwendung 90% der Zeit verbrauchte. Während im Vergleich zur seriellen Programmversion eine Beschleunigung erreicht wurde, ließ sich ein Anstieg der Rechenzeit bei Nutzung von mehr als acht Prozessoren beobachten.

- Fukumori et al [50] haben einen Metacomputer genutzt, der aus einer Fujitsu AP 1000 an der *Waseda University* in Tokio und eine NEC Cenju-3 im *NEC C&C Laboratory* in Kanagawa bestand. Diese ca. 370 km voneinander entfernten Rechner waren mittels einer ATM-Leitung mit einer Spitzenleistung von 155 MBit/s miteinander verbunden. Als Anwendung wurde ein Finite-Elemente-Löser genutzt. Die Autoren beobachteten eine signifikante Verlangsamung im Vergleich zur Nutzung einer homogenen Maschine. Sie schlossen daraus, daß weitere Arbeiten in den Bereichen Kommunikationsoptimierung und Lastausgleich notwendig sind [50].

- Simulation von quantenchemischen Reaktionsprozessen [145]. Das im Rahmen des CASA-Testbeds simulierte Modell kann in insgesamt drei Einzelteile zerlegt werden, die jeweils untereinander nur lose gekoppelt sind[22]. Durch die Überlappung von Kommunikation und Berechnung konnte die relativ hohe Übertragungszeit mit Berechnungen überdeckt werden. Bei den durchgeführten Experimenten ergab sich, begünstigt durch die lose Kopplung der Systeme, eine signifikante Beschleunigung. Es wurde sogar bei den großen Datensätzen ein superlinearer *speedup* beobachtet. Dieser wurde dadurch verursacht, daß die jeweiligen Teile völlig unterschiedliche Anforderungen stellten, die entweder durch einen Vektorrechner oder durch ein massivparalleles System am besten erfüllt werden konnten.

In diesen Bereich fallen aber auch viele Anwendungen, die bereits unter Visualisierung (Abschnitt 6.1.1) und *multi-physics*-Anwendungen (Abschnitt 6.1.2) genannt worden sind,

[22]Da die einzelnen Programmteile keine eigenständigen Simulationen sind, gehört diese Anwendung nicht zu den *multi-physics*-Anwendungen.

da die notwendigen Berechnungen oftmals in ihrer Größe durch die vorhandenen Rechen- oder Speicherkapazitäten beschränkt sind. Dies zeigt sich insbesondere bei interaktiven Anwendungen, bei denen eine schnelle Reaktion auf Befehle des Anwenders erfolgen muß. Im Rahmen das CASA-Testbeds mußte eine unerwünscht geringe Bildrate von vier Bildern pro Sekunde in Kauf genommen werden, da einerseits die Berechnung zu lange dauerte, andererseits die Ausgabe der Daten nicht schnell genug erfolgen konnte. Eine Verkürzung der Bearbeitungszeit ist erst durch eine Kopplung von Rechnern möglich. Die Geschwindigkeit der Ausgabe jedoch wurden durch Engpässe in der Eingabe- und Ausgabekapazität der beteiligten Rechner verursacht. Dies kann nur durch die Hersteller (bzw. durch spezielle Hardware, vgl. [73]) behoben werden.

Bei eng-gekoppelten Anwendungen zeigen sich schon die Stärken des einfachen CoFu-Modells: Während praktisch alle Ansätze manuell Optimierungen der Kommunikationsstruktur und den Lastausgleich vornehmen müssen, werden diese Aufgaben vom CoFu-Skelett automatisch gelöst, ohne daß dafür Erweiterungen an der Anwendung notwendig sind.

6.2 Kommunikationsbibliotheken

Im Prinzip kann CoFu als eine Art spezielle Kommunikationsbibliothek betrachtet werden, da die Anwendung weiterhin explizit den Datenaustausch vornehmen muß, indem sie die Daten in entsprechenden Puffern bereitstellt.

Die meisten Kommunikationsbibliotheken wie z.B. MPI [89], MPI-2 [90], PVM [52], PARMACS [13] oder RPCs [120], stellen eine Vielzahl von Kommunikationsprimitiva mit jeweils verschiedenen Semantiken zur Verfügung, vgl. auch Abschnitt 2.3.2. Diese Bibliotheken ermöglichen zwar durch entsprechende Kommunikationsbefehle die Überlappung von Kommunikation und Berechnung, allerdings unterstützen sie die Programmentwicklung nicht: Die Entwickler sind selbst dafür verantwortlich, entsprechende Algorithmen zu wählen, den Speicherplatz für die ein- und ausgehenden Nachrichten zu verwalten und entsprechende Synchronisationspunkte in die Programme einzufügen. CoFu hingegen nimmt dies durch das Konzept der virtuellen Prozessoren automatisch vor und vereinfacht somit die Programmentwicklung erheblich.

Im Gegensatz zu den allgemeinen Kommunikationsbibliotheken kennt das CoFu-Skelett die Kommunikationsstruktur der Anwendung. Dieses Wissen wird zur Optimierung der Kommunikation genutzt. In Ansätzen wird dies auch von *message-passing*-Bibliotheken zur Verfügung gestellt. So kann man in MPI [89] n-dimensionale reguläre Gitter mittels `MPI_Cart...` und beliebige Graphen durch `MPI_Graph...` erzeugen und nutzen, allerdings dient diese Information nur der Vereinfachung der Programmierung, nicht jedoch der Optimierung der Kommunikation. PARMACS [13] erlaubt die Angabe der Kommunikationsstruktur beim Starten der Anwendung, um ein möglichst gutes *mapping* der Prozesse auf die Architektur des zugrundeliegenden Rechners zu ermöglichen. Für die eigentliche Kommunikation hat dies allerdings kaum Auswirkungen, da bei den heutigen Rechnern die Kommunikationsleistung in der Praxis unabhängig von der physikalischen Anordnung der Prozessoren bzw. der Art des zugrundeliegenden Netzwerkes ist. Das CoFu-Skelett

bestimmt anhand der Kommunikationsstruktur der Anwendung das *scheduling* der virtuellen Prozessoren so, daß sich eine Überlappung von Kommunikation und Berechnung ergibt. Weiterhin nutzt CoFu die Kenntnis über die Art der Verbindung (lokal, intern oder extern), um die externe Kommunikation durch Zusammenfassen einzelner Nachrichten zu optimieren. Trotz dieser Optimierungen gewährleistet das CoFu-Modell auch weiterhin das wichtige Prinzip der Ortstransparenz, d.h. die zu nutzende Kommunikationsoperation ist unabhängig von dem Ort, am dem sich der Kommunikationspartner befindet. Hierdurch wird die Programmentwicklung deutlich vereinfacht.

6.3 Programmiermodelle und Laufzeitvorhersage

Zur Analyse von parallele Algorithmen werden Modelle eingesetzt, die die wichtigsten Eigenschaften eines parallelen Rechners nachbilden. Diese Modelle stammen meist aus dem Bereich der theoretischen Informatik und dienen der Rechenzeit- und Kommunikationszeit–Abschätzung paralleler Programme. Die *Parallel Random Access Machine* (PRAM) [44] ist wohl das bekannteste dieser Modelle. Die PRAM-Maschine setzt einen gemeinsamen Speicher voraus, bei dem der Zugriff auf jede Speicheradresse die gleiche Zeit erfordert. Eine Modellierung des Verbindungsnetzwerkes und der damit zusammenhängenden veränderlichen Zugriffszeit erfolgt hingegen nicht, weshalb sich mit der PRAM zwar asymptotische Aussagen über das Laufzeitverhalten eines Algorithmus machen lassen; für Aussagen zum Verhalten auf realen Systemen ist das Modell jedoch nicht geeignet [21].

Zahlreiche andere Modelle haben das Ziel einer realistische Modellierung existierender paralleler Rechner:

BSP-Modell: Das von Valiant [137] entwickelte BSP-Modell (*bulk-synchronous parallel model*) soll die gemeinsame Schnittstelle von theoretischen Modellen und praktischen Anwendungen bilden. Es setzt, im Gegensatz zur PRAM, keinen gemeinsamen Speicher voraus, sondern benutzt *message-passing*, um Daten zwischen den verschiedenen Prozessoren auszutauschen. Ein BSP-Programm besteht aus einer Folge von einzelnen Schritten, sogenannten *supersteps*. Ein solcher *superstep* besteht jeweils aus einer beliebig komplexen lokalen Berechnung, einem Datenaustausch mit einer Teilmenge aller Prozessoren und anschließend aus einer globalen *barrier*–Synchronisation. Kennzeichnend für BSP-Algorithmen ist, daß Anwendungen nicht auf der Basis einzelner Prozesse analysiert werden, sondern auf einer abstrakteren Ebene die Menge aller Berechnungen bzw. die Gesamtkommunikation eines *supersteps* betrachtet wird [121]. Zur Modellierung eines Parallelrechners werden insgesamt drei Parameter benötigt:

- p: Anzahl der Prozessoren.
- g: Das Verhältnis von Rechenleistung zu Kommunikationsleistung.
- L: Minimale Zeit eines *supersteps*, also die Zeit für eine *barrier*-Synchronisation.

Die Zeit für einen *superstep* berechnet sich dann aus der Summe der Zeiten für die lokalen Berechnungen, für die Datenübertragung und für die Synchronisation.

Das BSP-Modell ermöglicht dadurch eine Laufzeitvorhersage, indem man das Verhalten einer BSP-Anwendung auf einem realen System anhand der jeweils gültigen Para-

meter p, g und L abschätzt. Durch Software-Werkzeuge wird diese Laufzeitvorhersage auch weitergehend unterstützt: Hill et al. beschreiben in [64] den Einsatz von BSP zur Parallelisierung eines Multigrid-Verfahrens aus dem Bereich der Strömungsmechanik. Dort wird ein Werkzeug vorgestellt, das neben der Visualisierung des Laufzeitverhaltens auch die Vorhersage der Laufzeit auf anderen Rechnersystemen ermöglicht.

CGM: Der von Dehne et al. vorgeschlagene *Coarse Grained Multicomputer* [26] ist eine Erweiterung des BSP-Modells. Bei CGM werden Annahmen über die Größe des lokalen Speichers und der Menge der übertragenen Daten getroffen. Wenn p die Anzahl der Prozessoren ist und s die Größe des gesamten Speichers aller Prozessoren, so verfügt jeder Prozessor über einen Speicher der Größe $O(s/p)$. Dabei muß die Menge des lokalen Speichers „deutlich" größer als $O(1)$ sein, z.B. $s/p \geq p$. Weiterhin wird angenommen, daß in einer einzelnen „Kommunikationsrunde", die dem *superstep* beim BSP-Modell entspricht, jeder Prozessor $O(s/p)$ Daten sendet und empfängt. Durch diese Annahmen entsteht ein etwas mächtigeres Modell, das sich für praktische Anwendungen aber nur minimal vom BSP-Modell unterscheidet.

LogP-Modell: Das LogP-Modell [21] von Culler et al. zeichnet sich durch eine detaillierte Modellierung der Kommunikation aus, bei der auch die Kapazität des Verbindungsnetzwerkes berücksichtigt wird. Der Name LogP ist ein Akronym aus den vier Parametern, die einen Rechner in LogP beschreiben:

- L: Eine obere Schranke für die Übertragungsdauer einer Nachricht. Beim LogP-Modell wird die Länge der Nachricht bei der Ermittlung der Übertragungszeit nicht weiter berücksichtigt, sondern durch L abgeschätzt.
- o: Der *overhead*, also die Zeit, die ein Prozessor braucht, um das Senden oder Empfangen einer Nachricht durchzuführen.
- g: Der *gap* ist das minimale Zeitintervall zwischen aufeinanderfolgenden Nachrichtenübertragungen eines Prozessors. Damit bildet $1/g$ die Bandbreite pro Prozessor.
- P: Die Anzahl der Prozessoren.

Zeitangaben werden immer in Vielfachen der Zykluszeit des Prozessors angegeben. Die Zeit für die Übertragung einer Nachricht ermittelt sich somit zu $o + L + o$, nämlich der Zeit für den Beginn des Versendens, der eigentlichen Nachrichtenlaufzeit und der Zeit für das Empfangen der Nachricht. Mit Hilfe des Parameters g kann abgeschätzt werden, wann die nächste Nachricht geschickt werden kann und somit, welche Zeit für Berechnung zur Verfügung steht [21]. Dadurch ist eine sehr detaillierte Modellierung eines Algorithmus möglich.

Alle hier vorgestellten Modelle dienen der Analyse von Algorithmen, entweder durch Angabe des asymptotischen Verhaltens oder sogar durch Abschätzungen für realistische Prozessorzahlen. Das BSP-Modell bietet zusätzlich eine höhere Abstraktionsebene durch Betrachtung der Gesamtkommunikation. Das in dieser Arbeit vorgestellte CoFu-Modell ist am nächsten mit dem BSP-Modell verwandt. Beide Modelle unterscheiden, im Gegensatz zum LogP- und PRAM-Modell, zwischen Phasen für die Kommunikation und für

die Berechnung. Während im BSP-Modell aber eine *barrier*-Operation eingesetzt werden muß, um die einzelnen Kommunikationsschritte voneinander zu trennen, ist dies beim CoFu-Modell nicht notwendig, da durch Betrachtung der gesamten Kommunikationsstruktur bekannt ist, wann ein einzelner Prozessor den nächsten Berechnungsschritt beginnen kann. Dies ist ein in der Praxis ein wichtiger Effizienzvorteil, da die Kosten für die Synchronisation gerade in einem heterogenen System relativ groß sind. Im Gegensatz zum BSP-Modell wird auch zwischen synchronisierender und nicht synchronisierender Kommunikation unterschieden. Dies erlaubt eine weitere Reduzierung der Synchronisationspunkte, da beim BSP-Modell bei einer synchronisierenden Kommunikation zusätzlich mindestens eine *barrier*-Operation durchgeführt werden muß. Beim CoFu-Modell ist allerdings keine Modellierung der Kommunikations- oder Rechenzeiten vorgesehen. Durch die fehlende globale Synchronisierung der einzelnen Prozesse untereinander ist dies auch nur mit deutlich höherem Aufwand erreichbar, da es von der Anwendung abhängt, welche Prozessoren Daten austauschen und dementsprechend aufeinander warten müssen. An dieser Stelle könnte das LogP-Modell zur Modellierung der Kommunikationskosten eingesetzt werden. Denkbar wäre hier insbesondere ein Software-Werkzeug, vergleichbar mit dem in [64] für BSP vorgestellten, das Laufzeitdaten einer Anwendung erfaßt, diese visualisiert und die Laufzeit auf anderen Systemen abschätzt.

In Fällen, in denen die Kommunikation vollständig mit Berechnungen überlappt wird, ist eine einfache Laufzeitvorhersage ohne Berücksichtigung der Kommunikationskosten möglich. In Abschnitt 8.8 wird exemplarisch die genaue Laufzeitvorhersage eines CoFu-Programms anhand eines Beispiels demonstriert.

6.4 Lastausgleich

Viele Lastausgleichsalgorithmen sind für den Einsatz in Systemen gedacht, die aus gekoppelten Arbeitsplatzrechnern bestehen [4, 36, 49, 63]. Im Gegensatz zum Metacomputing stehen Arbeitsplatzrechner nicht dediziert zur Verfügung – aufgrund des interaktiven Arbeitens ändert sich bei diesen Rechnern im Laufe der Tageszeit die Auslastung erheblich. Es gelten also völlig andere Rahmenbedingungen als beim Metacomputing, so daß derartige Verfahren nur begrenzt übertragbar sind.

Weitere Lastausgleichalgorithmen sind speziell für eine einzelne Anwendung oder für eine spezielle Anwendungsklasse geschrieben. Beispiele hierfür sind Lastausgleichsverfahren

- für ein Klima-Modell [46, 47],
- bei parallelen, adaptiven Mehrgitterverfahren [3],
- für partielle Differentialgleichungen auf adaptiven, unstrukturierten Gittern [140].

Diese Verfahren können spezielle Eigenschaften der Anwendung nutzen, um einen guten Lastausgleich zu erhalten. Bei der Klimamodellierung z.B. ist ein Teil des Lastungleichgewicht auf die unterschiedlichen Rechenanforderungen von Gebieten zurückzuführen, die auf der Tages- bzw. Nachthälfte liegen. Dies kann schon bei der Gebietsaufteilung berücksichtigt werden, indem jedem Prozessor zwei „gegenüberliegende“ Gebiete der Erdkugel zugewiesen werden. Auch die darüber hinausgehenden Optimierungen der Lastverteilung

nutzen derartiges Wissen über die Anwendung. Da es nicht Ziel dieser Arbeit war, einzelne Algorithmen oder Anwendungen zu optimieren, konnten derartige Lastverteilungsstrategien nicht genutzt werden.

Einen allgemeineren Ansatz, der mit den hier vorgestellten Methoden vergleichbar ist, wird in der Arbeit von Perez [107] vorgestellt. Dort wird die in HPF vorhandene Abstraktion der virtuellen Prozessoren ebenfalls zur Implementierung eines dynamischen Lastausgleich in HPF-Programmen mittels Migration von virtuellen Prozessoren genutzt. Die virtuellen Prozessoren werden dabei durch *threads* erzeugt. Gegen die hier gewählte Simulation der virtuellen Prozessoren spricht nach [107], daß die vorhandenen Datenabhängigkeiten ein kompliziertes *scheduling* nötig machen. Das *scheduling* der virtuellen Prozessoren in CoFu ist aber einfacher, da aufgrund der Eigenschaften des CoFu–Modells keine Datenabhängigkeiten in einer Phase der lokalen Berechnung existieren. Der Lastausgleich findet durch Migration von virtuellen Prozessoren statt. Somit muß bei einer Lastumverteilung ein *thread* migriert werden, wodurch sich der Aufwand für die Migration erhöht, da auch der *stack* verschickt werden muß. Weiterhin ist dieses Vorgehen aufgrund der Nutzung von *threads* und der damit zusammenhängenden Migrationsimplementierung nur für homogene Systeme einsetzbar. Es werden keine weiteren Angaben über den eigentlich Lastausgleichsalgorithmus gemacht.

Einen anderen interessanten Ansatz zeigen Boier Martin et al. in [7]. Hier werden sogenannte *Data Allocation Units* (DAUs) definiert, die jeweils ein Arbeitspaket darstellen. Diese Arbeitspakete werden den einzelnen Arbeiter-Prozessen zugeschickt. Im Gegensatz zu *farming*-Anwendungen (vgl. Abschnitt 4.1) können diese DAUs nicht unabhängig voneinander bearbeitet werden, sondern es werden Daten von anderen DAUs benötigt, wodurch Kommunikation zwischen den Arbeiter-Prozessen notwendig ist. Es werden drei Verfahren zur Verteilung der DAUs auf die Prozessoren untersucht:

Statische Verteilung: Hier bekommt jeder Prozessor einmal, zu Beginn des Programms, eine Menge von DAUs zur Bearbeitung zugeordnet.

Dynamische Zuordnung: Sobald ein Prozessor eine DAU bearbeitet hat, bekommt er aus der Menge der noch zu berechnenden DAUs eine neue zugeschickt.

Hybride Verteilung: Hierbei wird ein bestimmter Anteil η aller DAUs statisch verteilt, der Rest dynamisch.

Zusätzlich wird zur Laufzeit die Datenabhängigkeiten der einzelnen DAUs untereinander ermittelt, die die sogenannte Arbeitsmenge (*working set*) bestimmen. Der in [7] vorgestellte Algorithmus versucht nun, möglichst ganze Arbeitsbereiche einem Prozessor zuzuordnen, um die Kommunikation zu minimieren. Die Nutzung der DAUs zur Lastbalancierung ist somit ähnlich zu dem in dieser Arbeit eingesetzten Konzept der virtuellen Prozessoren: Jeder virtuelle Prozessor hat seine eigenen festen Daten, diese Daten entsprechen genau einer DAU. Im CoFu-Modell ist allerdings die Kommunikationsstruktur a priori bekannt, so daß eine einmalige Verteilung der virtuellen Prozessoren zu den physikalischen Prozessoren vorgenommen wird. In [7] werden weiterhin Anwendungen betrachtet, bei denen die Datenmenge der DAUs größer ist als der gesamte zur Verfügung stehende Hauptspeicher, so daß der sekundäre Speicher genutzt werden muß. Derartige *out-of-core*-Anwendungen lassen sich durch das Konzept der virtuellen Prozessoren völlig automatisch implemen-

tieren. Mit Hilfe der Funktionen, die für die Prozeßmigration zur Verfügung stehen, kann ein vollständiger virtueller Prozessor auf einem Sekundärspeicher ausgelagert werden.

Ein weiterer vielversprechender Ansatz, der sich mit dem Konzept der virtuellen Prozessoren sehr gut verbinden läßt, wird in [136] vorgestellt. Bei der *scattered decomposition* wird das Gebiet zunächst in eine Anzahl von Teilgebieten, sogenannte Boxen, aufgeteilt. Jede dieser Boxen wird dann wiederum in Teilgebiete aufgeteilt. Jedem Prozessor wird schließlich ein Teilgebiet jeder Box zugeordnet. Die somit erreichte Verteilung ist in gewissen Grenzen selbst-balancierend, da eine höhere Rechenlast in einer der Boxen automatisch auf alle Prozessoren verteilt ist. Dieses Verfahren läßt sich einfach durch das initiale *mapping* der virtuellen Prozessoren auf die physikalischen Prozessoren erreichen.

6.5 Zusammenfassung

In diesem Kapitel wurde das hier entwickelte CoFu-Modell für den effizienten Einsatz eines Metacomputers mit existierenden Ansätzen verglichen. Dabei wurde zunächst die Anwendbarkeit des Modell für typische Anwendungen aus dem Bereich des Metacomputings gezeigt und die zusätzlichen Vorteile herausgearbeitet, die das CoFu-Skelett gegenüber herkömmlichen Kommunikationsbibliotheken bietet, nämlich die Optimierung der Kommunikation anhand der Information über die Kommunikationsstruktur der Anwendung. Erwähnenswert ist die Tatsache, daß diese Optimierungen nicht nur für einzelne Anwendungen oder Algorithmen, sondern für ein breites Anwendungsspektrum einsetzbar sind. Dies gilt insbesondere für den statischen und dynamischen Lastausgleich. Das Kozept der virtuellen Prozessoren ermöglicht eine einfache Nutzung existierender Lastausgleichsalgorithmen, ohne daß weitere Änderungen an der Anwendung notwendig sind.

Obwohl das CoFu-Modell gewisse Ähnlichkeiten mit dem BSP-Modell [137] aufweist, ist es doch wesentlich stärker auf praxisorientierte Optimierungen ausgerichtet, indem es auf die bei BSP notwendigen Synchronisationen verzichtet. Auch dies wird durch die konsequente Nutzung der Information über die Kommunikationsstruktur ermöglicht.

Das nächste Kapitel beschreibt die prototypische Implementierung des CoFu-Modells.

7 Implementierung

In diesem Kapitel wird die vorgenommene Implementierung eines Prototypen beschrieben, der die wichtigsten in Kapitel 5 vorgestellten Konzepte realisiert. Dieser Prototyp wurde auf den Cray-Systemen des Forschungszentrum Jülich und auf der IBM SP2 der GMD im Rahmen des Gigabit Testbeds West [35] getestet.

7.1 Programmiersprache

Aufgrund der notwendigen Portabilität des Prototypen mußte eine Programmiersprache gewählt werden, die auf möglichst vielen Rechner vorhanden ist und eine Generierung von effizientem Laufzeitcode auf Hochleistungsrechnern ermöglicht. In Frage kommen hier die Sprachen C, C++, FORTRAN 77 und FORTRAN 90. Während eine Implementierung des CoFu-Skelettes in FORTRAN 77 aufgrund der nicht standardmäßig vorhandenen Möglichkeiten der dynamischen Speicherverwaltung kaum möglich ist, bieten alle anderen Sprachen den notwendigen Funktionsumfang. C bzw. C++ bieten Vorteile in der standardisierten Nutzung von Betriebssystemfunktionen, die bei FORTRAN 90 nicht im gleichen Umfang unterstützt wird.

In dieser Arbeit wurde die Sprache C++ gewählt, da sich das Konzept der virtuellen Prozessoren natürlich auf C++ Objekte abbilden läßt: Jeder virtuelle Prozessor wird durch eine Instanz eines Objektes erzeugt, wobei die anwendungsspezifischen Funktionen mittels virtueller C++ Methoden implementiert sind. Wichtig ist allerdings auch eine Schnittstelle für FORTRAN-Programme, da immer noch ein Großteil der Anwendungen auf Hochleistungsrechnern in FORTRAN 77 bzw. FORTRAN 90 geschrieben sind. Eine solche Schnittstelle läßt sich mit C++ auf eine standardisierte Art realisieren, indem die aufzurufenden FORTRAN-Funktionen besonders als Fortran-Unterprogramm deklariert werden.

7.2 Kommunikationsbibliothek

Wie in Abschnitt 2.3.3 beschrieben, wird das Modell der expliziten parallelen Programmierung mittels *message-passing* für den Metacomputer genutzt. Da die herstellerspezifischen Kommunikationsbibliotheken das Kriterium der Portabilität nicht erfüllen, mußte eine möglichst verbreitete *message-passing*-Bibliothek genutzt werden, die gleichzeitig noch eine hohe Leistungsfähigkeit (im Sinne von hoher Bandbreite und niedriger Latenz) garantierte. Folgende Bibliotheken standen auf den zu nutzenden Rechnersysteme zur Verfügung:

MPI: Der *Message-Passing Interface Standard* [89] ist die Bibliothek, die z.Zt. den wohl größten Funktionsumfang und das ausgereifteste Konzept anbietet. MPI wurde von Anfang an mit dem Ziel entwickelt, eine sehr hohe Leistungsfähigkeit, gleichzeitig aber auch eine komfortable Programmierung zu ermöglichen. Dies sorgte dafür, daß sich MPI in den letzten Jahren zu einem de-facto Standard entwickelt hat, der

mittlerweile von praktisch jedem Hersteller von Hochleistungsrechnern unterstützt wird. Auch für Arbeitsplatzrechner liegen qualitativ hochwertige Implementationen vor, wie z.B. MPICH [58], das als *public-domain*-Produkt zur Verfügung gestellt wird. Eine Weiterentwicklung des Standards zu MPI-2 ist mittlerweile abgeschlossen [90], allerdings liegt z.Zt. erst eine Implementation dieses erweiterten Standards vor [101]. Eine Kopplung verschiedener Hochleistungsrechner ist konzeptuell schon bei MPI vorgesehen, hier unterscheiden sich einzelne Implementationen zum Teil erheblich in den Möglichkeiten.

`PVM:` Die *Parallel Virtual Machine* [52] erlaubt die Nutzung eines heterogenen Rechnernetzwerkes als Parallelrechner. Mittlerweile liegen PVM-Implementationen für faktisch jeden Hochleistungsrechner vor, oft in einer vom Herstellern der jeweiligen Maschinen optimierten Form. Die Kopplung verschiedener PVM Implementierungen ist dabei kein Problem. Der Umfang der angebotenen Kommunikationsoperationen ist für alle praktischen Belange ausreichend, allerdings deutlich geringer als bei MPI [57]. Messungen auf den zur Verfügung stehenden Systemen haben jedoch gezeigt, daß die Leistungsfähigkeit von PVM zum Teil erheblich hinter der Leistung einer MPI-Implementierung zurückbleibt, vgl. hierzu auch Abschnitt 8.1 und [76].

Aufgrund der hohen Leistungsfähigkeit, der weiten Verbreitung und des hohen Funktionsumfang wurde MPI als Kommunikationsbibliothek gewählt.

Während MPI konzeptuell die Nutzung heterogener Rechnersysteme vorsieht, gibt es in der Praxis Probleme gerade mit der Kopplung von Hochleistungsrechnern. Dies ist in der notwendigen Multiprotokollfunktionalität (vgl. Abschnitt 2.3.2) begründet. Prozesse, die innerhalb eines Hochleistungsrechner miteinander kommunizieren, müssen die jeweils herstellerspezifischen Protokolle nutzen, um eine hohe Leistungsfähigkeit zu garantieren. Auf der anderen Seite muß ein völlig anderes Protokoll für die externe Kommunikation eingesetzt werden. Die von den Herstellern zur Verfügung gestellten Implementierungen sehen im allgemeinen keine Kopplung mit anderen Systemen vor – der MPI-Standard definiert nämlich keine externe Schnittstelle, die die jeweiligen Implementationen einhalten müssen, so daß hier jeweils eigene Protokolle eingesetzt werden und somit eine Kopplung verschiedener MPI-Implementationen nicht möglich ist. Andere Implementationen ermöglichen z.Zt. auch noch keine Nutzung mehrerer Kommunikationsprotokolle: entweder nutzt man die rechnerspezifischen Befehle, kann dann aber keine Kopplung mit anderen Systemen vornehmen, oder man ermöglicht die Kopplung unterschiedlicher Systeme, kommuniziert dann allerdings auch innerhalb einer Maschine mit der langsamen, externen Schnittstelle.

Einen Ausweg aus dieser Situation bieten sogenannte Kopplungsbibliotheken, also Bibliotheken, die eine Kopplung verschiedener MPI-Implementierungen erlauben. Zur Zeit ermöglichen die Bibliotheken PVMPI [39], PLUS [11] und PACX [5] diese Kopplung. Alle drei Bibliotheken ersetzen durch einen Macro-Mechanismus die MPI-Befehle durch eigene Funktionen, die dann entweder mit den optimierten MPI-Befehlen eine interne Nachricht schicken, oder mittels eines eigenen Protokolls eine Nachricht zu einer anderen Maschine übertragen. Dies geschieht im allgemeinen transparent für die Anwendung, die nur MPI Befehle einsetzt. PVMPI wird von G. Fagg, J. Dongarra und A. Geist an der *Universität*

of Tennessee, Knoxville bzw. dem *Oak Ridge National Laboratory, Oak Ridge* entwickelt [39] und nutzt als externes Protokoll PVM, wodurch alle Rechner, für die eine PVM-Implementierung vorliegt, mittels MPI gekoppelt werden können. Leider lag zu Beginn der Implementierungsphase noch keine funktionsfähige Implementierung von PVMPI vor, so daß PVMPI nicht getestet werden konnte. PLUS wurde am Paderborner Zentrum für Paralleles Rechnen von M. Brune, J. Gehring und A. Reinefeld entwickelt [11] und benutzt ein eigenes, auf UDP [128] basierendes Protokoll, um die externe Kommunikation zu ermöglichen. Insbesondere bietet es die Möglichkeit, unterschiedliche Kommunikationsbibliotheken miteinander zu koppeln, so daß z.B. eine Nachricht, die mittels MPI verschickt wird, durch einen PVM-Befehl empfangen werden kann. Allerdings liegen nur begrenzt Implementierungen dieser Bibliothek vor, insbesondere fehlten Implementationen für die Cray-Systeme und für die SP2. Die Bibliothek PACX wurde von T. Beisel, E. Gabriel und M. Resch am High Performance Center Stuttgart entwickelt [5], um die Kopplung zweier Cray T3E Systeme miteinander zu ermöglichen. Allerdings wird hier nur eine Untermenge aller MPI Befehle unterstützt. Im Rahmen des Gigabit Testbed West [35] wird PACX eingesetzt und erweitert, um auch Cray Systeme mit einer IBM SP2 koppeln zu können.Gleichzeitig wird PACX in Stuttgart weiterentwickelt, um einerseits die Kopplung von mehr als nur zwei Rechnern zu ermöglichen, andererseits um einen erweiterten Umfang von MPI zu unterstützen. Eine fertige Version dieser neuen Version liegt aber noch nicht vor. Da PACX die Kopplung der notwendigen Maschinen ermöglicht, eine gute Leistungsfähigkeit bietet und der Quellcode dankenswerterweise von Herr Resch zur Verfügung gestellt wurde, wurde PACX als Kopplungsbibliothek gewählt.

PACX enthält in der z.Zt. vorliegenden Form noch einige Einschränkungen:

- Es können nur zwei Maschinen miteinander gekoppelt werden.
- Es werden keine selbstdefinierten Datentypen unterstützt, so daß nur Felder der MPI-Grunddatentypen `float`, `double` und `int` eingesetzt werden können.
- Es werden nicht alle MPI-Operationen unterstützt.

Unter Berücksichtigung dieser Einschränkungen wurde ein Prototyp des CoFu-Skelettes sowie dreier Testanwendungen (vgl. Abschnitt 7.9) erstellt, die auf einer Cray T3E, der Cray T90 und der IBM SP2 verteilt lauffähig sind. Auf der Intel Paragon XP/S 10 ist eine homogene Version mit MPI oder PVM funktionsfähig, auf Arbeitsplatzrechner, wie z.B. Sun-, DEC-, IBM- und Linux-Systemen kann sowohl eine sequentielle Version als auch die MPI bzw. PVM Version benutzt werden. Der Einsatz auf anderen Systemen ist also möglich, solange entweder PVM oder MPI zur Verfügung steht. Die im Bereich von heterogenen Systemen wichtige Portabilität ist erreicht.

7.3 Optimierung der externen Kommunikation

PACX benötigt für die externe Kommunikation zwei zusätzliche Kommunikationsprozessoren auf jeder Maschine, so daß eine Anwendung, die auf n Prozessoren einer Maschine rechnen will, dafür $n+2$ Prozessoren benötigt, vgl. Abbildung 25 a). Der erste zusätzliche Prozessor, im weiteren Sendeprozessor genannt, leitet externe Nachrichten zur anderen

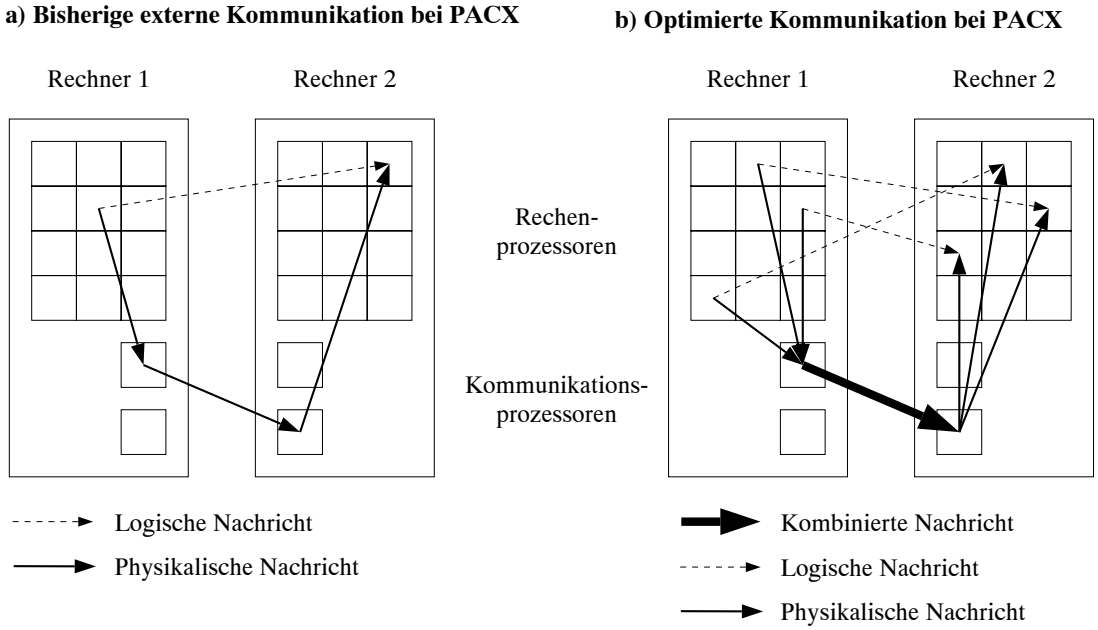

Abbildung 25: Schematische Darstellung einer externen Kommunikation mit PACX.

Maschine mittels eines eigenen TCP/IP-Protokolls weiter. Der zweite Prozessor, im weiteren Empfangsprozessor genannt, empfängt eingehende externe Nachrichten und schickt sie an den zuständigen Rechenprozessor, also einen der übrigen Prozessoren. Interne Nachrichten werden mittels der herstellerspezifischen MPI-Bibliothek übertragen. Eine externe Nachricht wird transparent für die Anwendung, wie in Abbildung 25 dargestellt, zuerst intern zum Sendeprozessor der eigenen Maschine geschickt, dort dann mittels TCP/IP zum Empfangsprozessor der anderen Maschine übertragen, wo wiederum eine interne Nachricht zum eigentlichen Empfangsprozessor generiert wird.

Wie in Abschnitt 5.2 gezeigt wurde, ist das Zusammenfassen mehrerer Nachrichten zu einer großen Nachricht wichtig, um die externe Kommunikation zu optimieren. Dies kann nun entweder auf der Ebene der Kommunikationsbibliothek geschehen, oder aber auf der Ebene des CoFu-Skelettes. Da PACX auf jedem Rechner einen eigenen Prozessor für die externe Kommunikation benötigt, bietet es sich an, das Zusammenfassen der Nachrichten auf diesem Prozessor durchzuführen. Erstens hat dieser Prozessor den größten freien Speicherplatz, da hier kein Anwendungsprogramm mit den zugehörigen Daten bearbeitet wird, und zweitens steht diese Optimierung auch Programmen zur Verfügung, die zwar PACX, nicht aber das CoFu-Skelett nutzen. Abbildung 25 b) zeigt das Vorgehen im optimierten Fall. Hierzu müssen aber, wie schon in Abschnitt 5.2 dargestellt, Informationen über die Kommunikationsstruktur des Programms vorhanden sein, um eine Verklemmung zu vermeiden.

Um dies zu ermöglichen, wurden zwei zusätzliche Befehle in PACX integriert: zunächst der Befehl `PACX_Get_ranks(...)`, der die Zuordnung von Prozessen zu den verschiedenen Maschinen zurückgibt. Mit Hilfe dieser Daten kann ermittelt werden, welche Kommuni-

kation extern und welche intern ist. Die eigentliche Optimierung wird mit Hilfe des neuen Befehls `PACX_Combine_nr(int nr_of_messages)` erreicht, mit dessen Hilfe jeder physikalische Prozessor festlegt, wieviele der nun folgenden externen Nachrichten kombiniert werden können, ohne daß eine Verklemmung auftritt. Das Speichern und Kombinieren der Nachrichten geschieht dann auf dem Sendeprozessor. Sobald von jedem Prozessor die angegebene Anzahl der externen Nachrichten eingetroffen ist, wird die kombinierte Nachricht zum Empfangsprozessor der Gegenseite geschickt, der die Nachricht wieder in die Einzelnachrichten zerlegt und diese dann weiteleitet.

7.4 Implementierung des CoFu-Modells

Wie in Abschnitt 5.2.4 beschrieben wurde, wird die Implementierung in der Form eines funktionalen Skelettes vorgenommen. Dies bedeutet, daß das Anwendungsprogramm einzelne C++ Methoden zur Verfügung stellt, die dann von dem CoFu-Skelett aufgerufen werden. Die einzelnen Module, die alle auf jedem Prozessor gleichzeitig beim Start des Programms angelegt werden, haben dabei folgende Funktionalität:

7.4.1 `CoFu_skelett`

Das Objekt `CoFu_skelett` liest eine Konfigurationsdatei (vgl. Abschnitt 7.8) ein, in der Angaben über die Zahl der virtuellen Prozessoren, den zu nutzenden *scheduling*-Algorithmus etc. enthalten sind. Diese Datei dient in erster Linie zu Testzwecken, damit für Leistungsvergleiche einzelne Optimierungen ein- oder ausgeschaltet werden können.

Anschließend wird ein Objekt der Klasse `CoFu_komm` angelegt, daß die einzelnen virtuellen Prozessoren und die dazugehörenden Objekte der Anwendung instanziiert, das *scheduling* der CoFu-Funktionen und deren Kommunikation durchführt.

7.4.2 `CoFu_komm`

Dieses Objekt legt zunächst in Abhängigkeit der von `CoFu_skelett` eingelesenen Daten die virtuellen Prozessoren an. Dabei ist ein virtueller Prozessor ein Objekt der Anwendung, das von von der Klasse `virtueller_prozessor` abgeleitet ist und das die Methoden und Daten der eigentlichen Anwendung enthält. Jedes dieser Objekt kann dabei eine Menge von CoFu-Funktionen enthalten und bekommt von `CoFu_komm` eine im gesamten Metacomputer eindeutige Identifikationsnummer (*id*) zugeordnet, über die dieser virtuelle Prozessor angesprochen werden kann.

Hauptaufgabe des Objektes `virtuell_komm` ist die Kommunikation, insbesondere das Kombinieren von Nachrichten, die zum gleichen physikalischen Prozessor geschickt werden müssen (vgl. Abschnitt 5.2.4). Bei der Implementierung dieser Routinen wurde insbesondere darauf geachtet, daß die Ausgabepuffer möglichst nicht kopiert werden. So wird eine lokale Nachricht nur durch Änderung entsprechender Zeiger und nicht durch Kopieren von Daten in den Ausgabepuffern implementiert. Dies ist wichtig, damit durch die Nutzung

der virtuellen Prozessoren und des damit verbundenen Anstiegs der Zahl der Nachrichten nicht zuviel Aufwand benötigt wird.

Weiterhin ist das Objekt `virtuell_komm` für das *scheduling* zuständig. Da das *scheduling* von der Kommunikation abhängt, um eine Überlappung von Kommunikation und Berechnung zu erreichen (vgl. Abschnitt 5.2), muß die Priorität eines jeden virtuellen Prozessors in Abhängigkeit von der auszuführenden CoFu-Funktion berechnet werden. Die Priorität p_i^f eines virtuellen Prozessors i und der CoFu-Funktion f ist ein ganzzahliger Wert, der so berechnet wird, daß sich das in Abschnitt 5.3 vorgestellte *scheduling* ergibt. Dazu wird angenommen, daß weniger als 100 000 Prozessoren in einem Metacomputer enthalten sind. Sei nun $recv_{i,j}^f$ der Empfänger der j-ten Nachricht der Funktion f vom virtuellen Prozessor i ($1 \leq j \leq m_i^f$, wobei m_i^f die Anzahl der Nachbarn der Funktion f vom virtuellen Prozessor i ist). Nun setzt man:

$$p_i^f = \sum_{j=1}^{m_i^f} p(i, recv_{i,j}^f), \quad \text{mit } p(i,k) = \begin{cases} 100\,000 & \text{falls die Nachricht} \\ & \text{von } i \text{ nach } k \text{ eine externe ist,} \\ 1 & \text{falls die Nachricht} \\ & \text{von } i \text{ nach } k \text{ eine interne ist,} \\ 0 & \text{sonst} \end{cases}$$

Durch die so definierte Gewichtungsfunktion $p(i,k)$ und der oben getroffenen Annahme über die Anzahl der physikalischen Prozessoren ist es nicht möglich, daß ein Prozessor durch zu viele interne Nachrichten eine höhere Priorität bekommt als ein Prozessor, der eine externe Nachricht versendet[23]. Das *scheduling* besteht nun einfach darin, die jeweilige CoFu-Funktion des virtuellen Prozessors in der durch die Priorität festgelegten Reihenfolge aufzurufen. Diese Reihenfolge wird für alle CoFu-Funktionen nur einmal bei der Initialisierung des Objektes `virtuell_komm` berechnet. Sobald nach dem Ende einer CoFu-Funktion alle Nachrichten für einen physikalischen Prozessor vorhanden sind, wird für diese Nachricht die Sende-Methode der `komm_lib`-Klasse aufgerufen, wo die Einzelnachrichten zusammengefaßt und als eine Nachricht verschickt werden. Wie zu sehen ist, ermöglicht die Implementierung der virtuellen Prozessoren durch die Simulation innerhalb des CoFu-Skelettes ein *scheduling*, ohne daß systemspezifische Funktionen genutzt werden. Die Portabilität der Lösung ist also gewährleistet (vgl. Abschnitt 5.2.3).

Weiterhin führt diese Klasse auch die Berechnungen der globalen Operation durch. Hierzu wird zuerst ein internes Zwischenergebnis unter Berücksichtigung aller Daten der virtuellen Prozessoren auf diesem Prozessor berechnet. Anschließend wird mit Hilfe dieser internen Zwischenergebnisse das globale Ergebnis mittels einer Funktion der Kommunikationsbibliothek berechnet. In dieser Prototypversion des CoFu-Skelettes wird somit die Reproduzierbarkeit zugunsten einer schnelleren Bearbeitung noch nicht implementiert: Für eine Reproduzierbarkeit des Ergebnisses muß garantiert werden, daß die Zwischenergebnisse in einer festdefinierten Reihenfolge ermittelt werden. Durch das Nutzen der

[23] Aufgrund der Beschränkung der eingesetzten PACX-Version können nur zwei Rechner miteinander gekoppelt werden (vgl. Abschnitt 7.2). Deshalb wird das in Abschnitt 5.3 vorgestellte erste Kriterium für das *scheduling*, der Vergleich der Anzahl der externen Maschinen, zu denen Daten geschickt werden müssen, nicht eingesetzt.

Kommunikationsbibliothek ist dies nicht mehr gewährleistet, da schon diese Funktion nicht notwendigerweise ein reproduzierbares Ergebnis liefert[24].

7.4.3 `komm_lib`

Diese Klasse abstrahiert von der zugrundeliegenden *message-passing*-Bibliothek, bietet aber auch komfortablere Funktionen an, wie z.B. das Zusammenfassen und Versenden von mehreren Nachrichten und das Gegenstück, also das Empfangen und Aufteilen einer kombinierten Nachricht. Auch speichert sie die Abbildung (*mapping*) der virtuellen Prozessoren auf die physikalischen Prozessoren. Dies ist besonders wichtig für den dynamischen Lastausgleich, da hier alle Prozessoren über eine Migration informiert werden müssen.

Von dieser Klasse gibt es vier unabhängige Implementierungen: `komm_pacx`, `komm_mpi`, `komm_pvm` und `komm_none`. Die Implementierung `komm_pacx` nutzt die in Abschnitt 7.2 beschriebene PACX Kommunikationsbibliothek und enthält alle vorgestellten Optimierungen. Man beachte, daß das Zusammenfassen der externen Nachrichten wie in Abschnitt 7.3 beschrieben transparent für das Skelett in der PACX-Bibliothek geschieht. Als Grundlage für `komm_pvm` dient PVM . Dies ist eine Version, die auch die Nutzung eines heterogenen Systems ermöglicht. Da sich allerdings bei ersten Messungen herausgestellt hat, daß die Leistung von PVM hinter der Leistung von PACX zurückbleibt, siehe auch Abschnitt 8.1, wurde diese Version nicht weiter optimiert, so daß das Zusammenfassen externer Nachrichten fehlt. Auf homogenen Systemen ist diese Version mit recht guten Leistungen einsetzbar. Auf MPI als Kommunikationsbibliothek setzt `komm_mpi` auf und enthält auch nicht die Optimierung des Zusammenfassens von Nachrichten. Sie ist in erster Linie für homogene Systeme geeignet und bietet dort eine gute Leistung. Besonders interessant ist noch `komm_none`. Hier wird keine Kommunikationsbibliothek genutzt, sondern es wird nur die Programmbearbeitung auf einem einzelnen Prozessor unterstützt, der dann virtuelle Prozessoren simuliert. Mit dieser Version kann ein paralleles Programm auf einem Arbeitsplatzrechner entwickelt und getestet werden. Dies hilft, teure Rechenzeit auf Parallelrechnern für die Programmentwicklung zu sparen. Zwar können mit dieser Version im allgemeinen keine großen Rechnungen durchgeführt werden, für die Programmentwicklung, zum Testen und zur Fehlersuche ist es aber eine wichtige Möglichkeit.

7.4.4 `virtueller_prozessor`

Ein Objekt vom Typ `virtueller_prozessor` verwaltet die Ein- und Ausgabepuffer eines virtuellen Prozessors und muß von Anwendungen als Basisklasse genutzt werden, vgl. Abbildung 26. Im Gegensatz zur formalen Spezifikation aus Abschnitt 4.2 wird hierbei aber nicht nur ein Ein- bzw. Ausgabepuffer zur Verfügung gestellt, sondern ein Feld von Ein- und Ausgabepuffern, so daß für jeden Kommunikationspartner ein eigener Puffer existiert. Die Anzahl der jeweiligen Puffer wird von dem CoFu-Skelett anhand der Informationen über das Kommunikationsverhalten eigenständig berechnet. Zusätzlich wird zu jedem

[24]Der MPI-Standard [89] empfiehlt zwar, daß die Ergebnisse reproduzierbar sein sollen, dies ist aber keine notwendige Bedingung an eine MPI Implementierung.

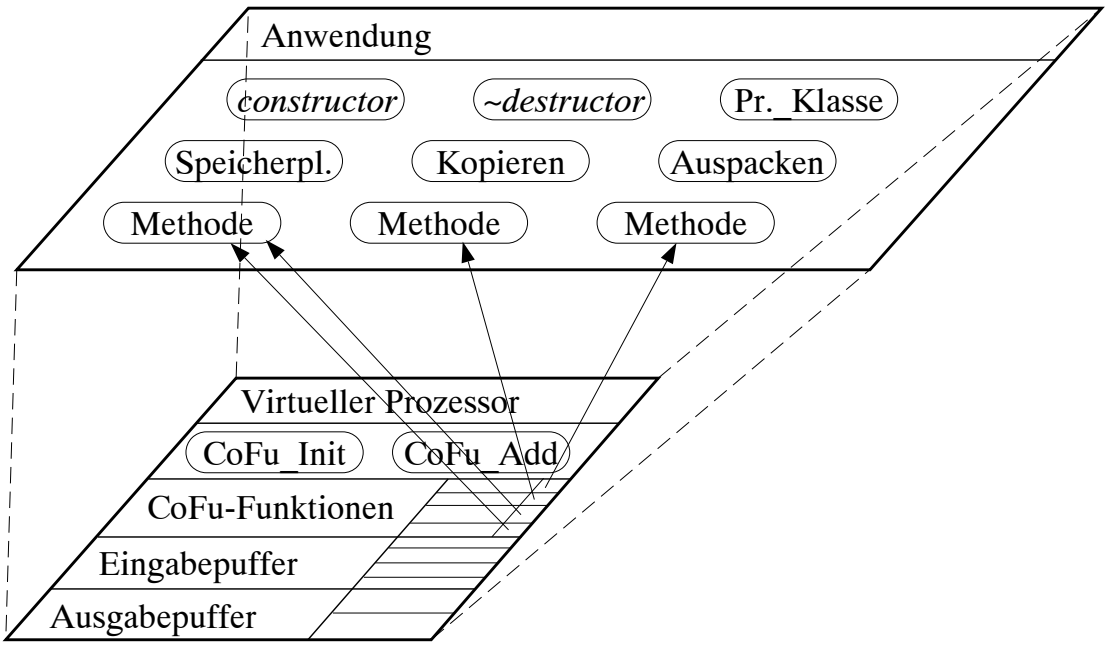

Abbildung 26: Architekturdiagramm des CoFu-Skelettes

Puffer seine Größe und eine Typangabe gespeichert. Die Typangabe ist dabei besonders für heterogene Systeme von Bedeutung, damit die ggf. notwendige Datenkonvertierung durchgeführt werden kann. Weiterhin definiert das Objekt `virtueller_prozessor` drei Methoden, die zur Definition einer CoFu-Funktion genutzt werden müssen. Dabei wird eine Methode des Anwendungsobjektes mit einer Kommunikation verknüpft, wodurch eine CoFu-Funktion entsteht.

`CoFu_init(n)`: Diese Funktion muß zuerst aufgerufen werden und legt die Anzahl `n` der Methoden fest, die als CoFu-Funktionen maximal genutzt werden können. Jeder CoFu-Methode wird eine Nummer von 0 bis `n-1` zugeordnet. Dadurch kann die gleiche Methode mit unterschiedlichen Kommunikationsmustern verbunden werden. Diese Methode ist für eine optimale Speicherallokierung notwendig.

`CoFu_add(int i, inr nr_sends, int *recvs, int *buffer, CoFu_fct fct)`: Diese Methode definiert, daß `fct` die `i`-te CoFu-Funktion der Anwendung ist. Diese Funktion wird im weiteren über ihre Nummer `i` angesprochen. Die am Ende der Funktion durchzuführende Kommunikation wird über die restlichen Parameter bestimmt: `nr_sends` ist die Anzahl der zu sendenden Nachrichten, `recv[j]` die Nummer des virtuellen Prozessors, der die Nachricht im `j`-ten Ausgabepuffer empfangen soll und `buffer[j]` der Eingabepuffer des virtuellen Prozessors `recv[j]`, in den die Daten geschrieben werden sollen. Die Anwendung ist dabei selbst dafür verantwortlich, daß die einzelnen Nachrichten in unterschiedliche Empfangspuffer geschrieben werden.

`CoFu_add(int i, CoFu_sync_op operation, CoFu_fct fct)`: Diese Funktion definiert eine synchronisierende Funktion mit der Nummer `i` und der Methode `fct`. In dem Prototyp sind die beiden Funktionen `COFU_SUM` und `COFU_MAX` definiert, die jeweils ei-

ne Summe der Daten bzw. das Maximum der Daten berechnen. Weitere Operationen, insbesondere benutzerdefinierte Funktionen sind auch denkbar. Hierbei wird der Ausgabepuffer 0 genutzt, um die Daten zur Verfügung zu stellen und der Eingabepuffer 0, um das globale Ergebnis der nächsten CoFu-Funktion zu übergeben.
Diese Funktionen lassen sich, wie schon in Abschnitt 4.2 beschrieben, auch mit normalen CoFu-Funktionen definieren, allerdings wird in dem Fall der Definition einer synchronisierenden Operation eine optimierte Funktion genutzt

Ein Anwendungsprogramm muß nun neben den eigentlichen Methoden der Anwendung folgende Methoden definieren bzw. überschreiben, damit ein CoFu-Programm entsteht:

- `anwendung::anwendung(...)`
  Der *constructor* einer Anwendung ist für die Initialisierung des Objektes und für die Festlegung der Kommunikationsstruktur verantwortlich. Hier müssen mittels der Methoden `CoFu_init` bzw. `CoFu_add` alle CoFu-Funktionen deklariert werden. Die CoFu-Funktion, die die Nummer 0 zugewiesen bekommt, wird dabei die initiale Funktion, d.h. sie ist die erste Funktion, die ausgeführt wird.

  Eingabeparameter

`int id`	Eine eindeutige Nummer, im weiteren Prozessornummer genannt, die jedem virtuellen Prozessor zugewiesen wird. Diese Nummer wird benutzt, um die Nachbarschaftsrelation festzulegen (vgl. die Methode `CoFu_add`). Diese Nummern werden fortlaufend, beginnend mit 0, vergeben.
`int *nr_processes_on`	`nr_processes_on[i]` gibt die Anzahl der virtuellen Prozessoren auf dem physikalischen Prozessor i an. Der erste physikalische Prozessor bekommt die virtuellen Prozessoren mit den Nummern 0 bis `nr_processes_on [0]`-1 zugewiesen etc. Diese Information kann benutzt werden, um eine kommunikationsminimale Festlegung der Nachbarschaftsrelation zu ermöglichen.

- `anwendung::~anwendung()`
  Wird bei Beendigung der Berechnung aufgerufen. Hier sollten notwendige Arbeiten für das Ende des Programms wie z.B. Speicherfreigabe, Schließen von Dateien etc. durchgeführt werden.

- `int anwendung::CoFu_fct()`[25]
  Eine CoFu-Funktion ist eine Methode, die keine Übergabeparameter enthält und als Funktionsergebnis die Nummer einer CoFu-Funktion zurückgibt. Diese Methoden nutzen die Ein- und Ausgabepuffer, die vom virtuellen Prozessor zur Verfügung gestellt werden. Auf diese Puffer kann über folgende Objektvariablen zugegriffen werden:

[25] Der Name dieser Methoden kann natürlich beliebig gewählt werden.

`void **input_buffer`	`*input_buffer[i]` zeigt auf den Speicherbereich, in dem die Daten vom `i`-ten virtuellen Nachbarprozessor enthalten sind.
`int *in_size`	`in_size[i]` enthält die Anzahl der Daten vom Typ `input_type[i]`, die in dem Puffer `*input_buffer[i]` stehen.
`datatype *input_type`	Dieses Feld enthält den Datentyp der Daten der jeweiligen Eingabepuffer. Aufgrund der Einschränkungen von PACX (vgl. Abschnitt 7.2) werden in der vorgenommenen Prototypimplementierung nur die Standarddatentypen `float`, `double` und `int` unterstützt. Eine Erweiterung auf beliebige Typen ist aber möglich, sobald dies von PACX unterstützt wird (vgl. hierzu die in MPI möglichen selbstdefinierten Datentypen [89]).
`void **output_buffer`	In diesem Feld werden die Daten für die virtuellen Nachbarprozessoren zurückgegeben, die in der Phase der nächsten Nachbarschaftskommunikation übertragen werden müssen.
`int *out_size`	Gibt die Anzahl der Daten für den jeweiligen virtuellen Nachbarprozessor an.
`datatype *output_type`	Gibt den Datentyp der Daten für den jeweiligen virtuellen Nachbarprozessor an.

- `int anwendung::prozessorklasse()`
  Diese Funktion gibt als Funktionsergebnis die momentane Prozessorklasse (vgl. Abschnitt 5.5.4) zurück, zu der dieser virtuelle Prozessor gehört. Dies ist ein ganzzahliger Wert, der von der Anwendung beliebig verwendet werden kann. Es muß nur sichergestellt sein, daß Prozessoren, die in der gleichen Prozessorklasse liegen, ein vergleichbares Rechenverhalten zeigen.

- `int anwendung::speicherplatz()`
  Diese Funktion gibt als Funktionsergebnis die Größe des Speicherplatzes zurück, der für die Speicherung aller Daten des virtuellen Prozessors notwendig ist. Diese Methode wird für die Migration eines virtuellen Prozessors benötigt.

- `int anwendung::migration_erlaubt()`
  Das Funktionsergebnis dieser Funktion bestimmt, ob ein virtueller Prozessor migriert werden darf, oder ob er auf dem physikalischen Prozessor bleiben muß, auf dem er gerade bearbeitet wird. Dies ist für virtuelle Prozessoren wichtig, die spezielle Eigenschaften des physikalischen Prozessors voraussetzen, z.B. Grafik-Hardware, lokale Platten etc.

- `anwendung::daten_kopieren(...)`
  Diese Methode muß eine Kopie aller benötigten Daten des virtuellen Prozessors anlegen.

  Ausgabeparameter

`void **buffer`	Ein Zeiger auf den Speicherbereich, in dem die Daten abgelegt werden sollen. Die Größe dieses Speicherbereiches wird mittels `int anwendung::speicherplatz()` ermittelt und vom CoFu-Skelett angelegt, so daß die Anwendung nur noch die Daten in den entsprechenden Speicherbereich kopieren muß.
`datatype *type`	Der Datentyp der abgelegten Daten.
`int *count`	Anzahl der Daten in `buffer`.

- `anwendung::daten_auspacken(...)`
  Diese Methode muß anhand der Daten, die mittels `anwendung::daten_kopieren()` angelegt worden sind, den virtuellen Prozessor wiederherstellen.

  Eingabeparameter

`void *buffer`	Ein Zeiger auf den Speicherbereich, in dem die Daten abgelegt sind.
`datatype type`	Der Datentyp der abgelegten Daten.
`int count`	Anzahl der Daten in `buffer`.

Um also ein paralleles Programm zu implementieren, daß von diesem CoFu-Skelett unterstützt wird, genügt es, diese sechs Methoden (plus der anwendungsspezifischen CoFu-Funktionen) in einer Objektklasse zu programmieren. Alle weiteren Aufgaben, also z.B. das *scheduling* der virtuellen Prozessoren, die vorgestellte, optimierte Kommunikation etc., wird von dem Skelett selbst ausgeführt.

7.5 Statischer Lastausgleich

Wie in Abschnitt 5.4 beschrieben wurde, wird der statische Lastausgleich implementiert, indem eine unterschiedliche Anzahl von virtuellen Prozessoren auf den jeweiligen physikalischen Prozessoren bearbeitet werden. Dies geschieht transparent für die Anwendung. In der Konfigurationsdatei (vgl. Abschnitt 7.8) kann entweder das Verhältnis der Rechenzeiten der einzelnen Prozessoren angegeben werden, oder sogar eine genaue Verteilung der einzelnen virtuellen Prozessoren auf die physikalischen Prozessoren.

7.6 Dynamischer Lastausgleich

Der dynamische Lastausgleich findet, wie in Abschnitt 5.5 beschrieben, zusammen mit den synchronisierenden Operationen statt. Zu den Zwischenergebnissen der globalen Operationen, die auf jedem physikalischen Prozessor ermittelt werden (vgl. Abschnitt 7.4), wird zusätzlich für jeden virtuellen Prozessor dessen Prozessorklasse sowie die Summe der Rechenzeit übertragen, die er seit der letzten synchronisierenden Operation (bzw. seit dem Programmstart) benötigt hat. Die Rechenzeit wird dabei von dem CoFu-Skelett gemessen. Nach der Berechnung des globalen Ergebnisses wird anhand dieser Daten wie in Abschnitt

5.5 beschrieben zentral eine Umverteilung der virtuellen Prozessoren berechnet. Falls ein virtueller Prozessor mehrfach migrieren muß, also von einem Prozessor A zu B, und von dort dann zu einem Prozessor C, wird dies zu einem Migrationsschritt von A zu C zusammengefaßt. Zu dem Ergebnis der synchronisierenden Kommunikation bekommt jeder Prozessor mitgeteilt, welche virtuellen Prozessoren zu welchen physikalischen Prozessoren migrieren. Diese Information wird einerseits benötigt, um die Tabellen über den Ort jedes virtuellen Prozessors zu aktualisieren, andererseits erfährt so jeder physikalische Prozessor, welche Prozessoren er migrieren muß und wieviele neue Prozessoren er zugeschickt bekommt. Bei dieser prototypischen Implementierung findet noch keine Fehlerbehandlung statt, falls also nicht genügend Speicherplatz vorhanden ist, um die Migrationen durchzuführen, bricht das Programm ab. Dies kann durch ein zweistufiges Protokoll behoben werden, indem zunächst die vorzunehmenden Migrationen an alle Prozessoren verschickt werden, und nach Abschluß der Migrationen alle erfolgten Migrationen. Wenn eine Migration aufgrund von Speichermangel oder einer anderen Ursache nicht durchgeführt werden kann, wird im zweiten Schritt keine Bestätigung dieser Migration verschickt, so daß dieser virtuelle Prozessor auf dem ursprünglichen physikalischen Prozessor weiterverarbeitet wird.

Das Versenden eines virtuellen Prozessors wird mit Hilfe der beiden Methoden `anwendung::speicherplatz()` und `anwendung::daten_kopieren()` realisiert. Dazu wird zuerst mittels `anwendung::speicherplatz()` die Größe des notwendigen Speicherplatzes für die Daten des virtuellen Prozessors ermittelt. Dann wird der Speicherplatz für die Daten des virtuellen Prozessors und für die internen Verwaltungsdaten wie z.B. die Kommunikationsstrukturen der einzelnen CoFu-Funktionen angelegt. Nach dem Kopieren der internen Daten werden mittels eines Aufrufs von `anwendung::daten_kopieren()` die Daten des virtuellen Prozessors kopiert. Diese Migrationsdaten werden dann an den physikalischen Prozessor gesendet, der diesen virtuellen Prozessor ab jetzt bearbeiten muß. Beim Versenden der migrierenden Prozessoren wird wiederum sichergestellt, daß externe Nachrichten vom PACX-Sendeprozessor (vgl. Abschnitt 7.2) zusammengefaßt werden und als eine Nachricht übertragen werden. Der Empfangsprozessor verteilt die Einzelnachrichten dann wieder an die jeweiligen Empfängerprozessoren.

Das Empfangen eines virtuellen Prozessors wiederum benötigt nur die Methode `anwendung::daten_auspacken()`. Da jeder Prozessor aufgrund der ihm mitgeteilten Informationen des Lastverwalters weiß, wieviele virtuelle Prozessoren er neu zugeteilt bekommt, legt er zuerst eine entsprechende Anzahl neuer Objekt der Klasse `virtueller_prozessor` an und wartet auf die Migrationdaten. Sobald diese eintreffen, entnimmt er der Nachricht die Verwaltungsinformationen, die zur Vervollständigung der virtuellen Prozessoren notwendig sind, und ruft anschließend `anwendung::daten_auspacken(...)` auf, um den virtuellen Prozessor zu initialisieren. Nachdem alle virtuellen Prozessoren eingetroffen sind, werden die Prioritäten aller virtuellen Prozessoren neu bestimmt, damit das *scheduling* aktualisiert wird.

7.7 Unterstützung der VAMPIR-Tracefile-Erstellung

Als Beispiel für die Möglichkeiten der Unterstützung der Entwicklung und Optimierung von Anwendungen, die durch das genutzte Programmiermodell möglich sind, wurde eine Schnittstelle zum Visualisierungswerkzeug VAMPIR [100] implementiert. VAMPIR besteht aus zwei Komponenten: einem Laufzeitsystem, das während der Bearbeitung der Anwendung sogenannte Laufzeitspuren (*traces*) sammelt und in Dateien schreibt, und der eigentlichen graphischen Oberfläche, die diese Daten entsprechend auswertet und anzeigt. Eine Laufzeitspur besteht dabei aus einer Folge einzelner Ereignisse, wobei im wesentlichen folgende vier Ereignisse möglich sind: Beginn einer Funktion, Verlassen einer Funktion, Versenden einer Nachricht und Empfangen einer Nachricht. Um nun eine VAMPIR-Spur einer Anwendung zu erzeugen, wurden die wichtigsten Routinen des CoFu-Skelettes modifiziert, so daß die entsprechenden VAMPIR-Ereignisse mit Hilfe des Laufzeitsystems erzeugt werden. Weiterhin werden für die einzelnen oben aufgeführten Methoden des Anwendungsprogramms entsprechende Ereignisse automatisch mitaufgezeichnet. Mit dieser Schnittstelle sind alle in dieser Arbeit enthaltenen VAMPIR-Abbildungen von CoFu-Programmen entstanden. Falls der Anwender eine detailliertere Übersicht über einzelne Unterprogramme seiner Anwendung benötigt, muß er die Aufrufe zu der VAMPIR-Laufzeitbibliothek einfügen bzw. die entsprechenden zusätzlichen Werkzeuge von VAMPIR einsetzen.

7.8 Konfigurationsdatei

Zu Testzwecken kann das Verhalten des CoFu-Skelettes anhand einer Konfigurationsdatei leicht geändert werden – einzelne Optimierungen können ein- oder ausgeschaltet werden, verschiedene *scheduling*-Strategien können ausgewählt werden etc. Im einzelnen sind folgende Parameter in der Konfigurationsdatei definiert:

scheduling: Über diesen Parameter kann gesteuert werden, welches *scheduling* der virtuellen Prozessoren innerhalb der Prozessoren genutzt werden soll. Mögliche Werte sind:

- `best`: Das hier in Abschnitt 5.3 beschriebene *scheduling*, das eine möglichst gute Überlappung von Kommunikation und Berechnung realisiert.
- `worst`: Hier werden die virtuellen Prozessoren genau in der umgekehrten Reihenfolge zum *schedule* `best` berechnet. Dies bedeutet, daß eine möglichst geringe Überlappung von Kommunikation und Berechnung erreicht wird.
- `none`: Hier werden die virtuellen Prozessoren in der Reihenfolge gestartet, in der sie in den internen Tabellen stehen. Da diese Reihenfolge unabhängig von der Kommunikationsstruktur ist, wird eine quasi zufällige, allerdings gleichbleibende Reihenfolge der Bearbeitung erreicht.

Standardmäßig wird hier das *scheduling* `best` genommen.

combine: Dies legt fest, ob externe Nachrichten kombiniert werden sollen oder nicht. Hier sind die Werte 0 (nicht zusammenfassen) und 1 (zusammenfassen) erlaubt. Standardmäßig werden externe Nachrichten zusammengefaßt.

virtual_processes: Hier muß die Anzahl der zu simulierenden virtuellen Prozessoren angegeben werden.

placement: Hier kann angegeben werden, wie die virtuellen Prozessoren auf die physikalischen Prozessoren abgebildet werden. Mögliche Werte sind:

- `same:` Hier bekommt jeder Prozessor die gleiche Anzahl an virtuellen Prozessoren zugewiesen. Falls die Anzahl der virtuellen Prozessoren nicht ohne Rest durch die Anzahl der physikalischen Prozessoren teilbar ist, bekommt eine Anzahl von Prozessoren einen virtuellen Prozessor zusätzlich zugewiesen.
- `weighted:` Hier kann ein Maß für die Leistungsfähigkeit jedes einzelnen Prozessors angegeben werden. Die virtuellen Prozessoren werden im gleichen Verhältnis zugewiesen, wie die Maße der jeweiligen Prozessoren. Auch hier sind rundungsbedingte Abweichungen möglich, wenn die Idealverteilung mit der angegebenen Anzahl der virtuellen Prozessoren nicht erreichbar ist. Dies ist die bequemste Möglichkeit, eine statische Lastverteilung zu erreichen.
- `individual:` In diesem Fall kann für jeden einzelnen Prozessor angegeben werden, auf welchem physikalischen Prozessor er berechnet werden soll. Dies erlaubt nicht nur eine statische Lastverteilung, sondern hier kann auch detailiert die Kommunikationsstruktur der Anwendung berücksichtigt werden.

Standardmäßig wird hier die Einstellung `same` genutzt.

output_filename: Hier kann eine Datei spezifiziert werden, in die Meldungen des CoFu-Skelettes geschrieben werden sollen. Hierzu gehört die genaue Abbildung der virtuellen Prozessoren auf die physikalischen Prozessoren, die jeweilige Kommunikationszeit der physikalischen Prozessoren, die gesamte Rechenzeit, durchgeführte Migrationen etc.

7.9 Testbeispiele

Im Rahmen dieser Dissertation wurden drei verschiedene Testanwendungen genutzt, um die Leistungsfähigkeit des CoFu-Skelettes zu untersuchen. Das erste Beispiel war die

Lösung einer eindimensionalen Differentialgleichung mittels eines einfachen Jacobi-Verfahren. Das zweite und dritte Beispiel berechnet die Lösung einer zweidimensionalen partiellen Differentialgleichung mit einem Multigrid-Verfahren und mit dem Schwarz-Verfahren.

7.9.1 Eindimensionale DGL und Jacobi-Verfahren

Als erstes Beispiel wurde die gewöhnliche eindimensionale Differentialgleichung

$$f''(x) = 0, \;\; 0 \leq x \leq 1, \;\; f(0) = 0, f(1) = 1 \tag{12}$$

mit der Lösung $f(x) = x$ genutzt. Nach der Diskretisierung mit Hilfe der Methode der finiten Differenzen ergab sich die diskrete Gleichung

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} = 0, \;\; 1 \leq i \leq M-1, \; y_0 = 0; y_M = 1, \tag{13}$$

wobei M die Anzahl der Gitterpunkte auf dem diskreten Gitter ist. Mittels des Jacobi-Verfahrens [9] kann nun eine Lösung nach folgendem Algorithmus bestimmt werden:

1. $y_i = 0$, für $0 \leq i \leq M-1$,
   $y_M = 1$,
2. $y_i^{neu} = \frac{y_{i-1}+y_{i+1}}{2}$, $1 \leq i < M$,
3. $y_i = y_i^{neu}$, $1 \leq i < M$

wobei die Schritte 2 und 3 solange iteriert werden, bis eine Lösung mit ausreichender Genauigkeit gefunden worden ist. Dieser Algorithmus wurde durch eine Gebietsaufteilung parallelisiert, d.h. jeder Prozessor p bekam einen Teilbereich $y_{i_p}, y_{i_p+1}, \ldots, y_{i_p+n_p-1}$ von n_p Elementen zugewiesen, wobei für eine homogene Maschine jeder Prozessor die gleiche Anzahl an Elementen zugewiesen bekam. An den Rändern der jeweiligen Teilbereiche mußten allerdings die Werte von y (also y_{i_p} bzw. $y_{i_p+n_p-1}$) zu den jeweiligen benachbarten Prozessoren $p-1$ bzw. $p+1$ geschickt werden, da die neuen, vom Nachbarprozessor berechneten Werte dort jeweils benötigt werden.

Dieser Algorithmus wurde einerseits mittels MPI [89], andererseits mit Hilfe des vorgestellten Programmiermodells implementiert. Dadurch konnte gemessen werden, wie hoch der Aufwand für das CoFu-Skelett und für die Verwaltung der virtuellen Prozessoren ist. Auf einer homogenen Maschine kann wegen des sehr geringen Datenmenge, die pro Rechenschritt übertragen werden (zweimal acht Bytes bei der Nutzung von doppeltgenauen Fließkommazahlen), keine Zeitersparnis durch die Überlappung von Kommunikation und Berechnung erwartet werden. Abschnitt 8.2 enthält die genauen Ergebnisse.

7.9.2 Zweidimensionale DGL und Multigrid-Verfahren

Multigrid-Verfahren zählen zu den leistungsfähigsten Verfahren zur Lösung von linearen Gleichungssystemen, die im Moment bekannt sind. Während sie mit deutlich weniger

Rechenschritten eine Lösung bestimmen können, als es mit den einfachen Jacobi- oder Gauß-Seidel-Verfahren möglich ist, ist doch ein deutlicher Mehraufwand bei der Programmierung eines Multigrid-Verfahren notwendig.

Das Grundprinzip des Multigrid-Verfahrens ist, ein Gleichungssystem nicht nur auf einem diskreten Gitter zu lösen, sondern eine Hierarchie von immer gröber werdenden Gittern zu benutzen. Abbildung 27 zeigt das prinzipielle Vorgehen anhand eines einfachen eindimensionalen Beispiels. Auf jedem dieser Gitter wird ein einfaches iteratives Verfahren, wie

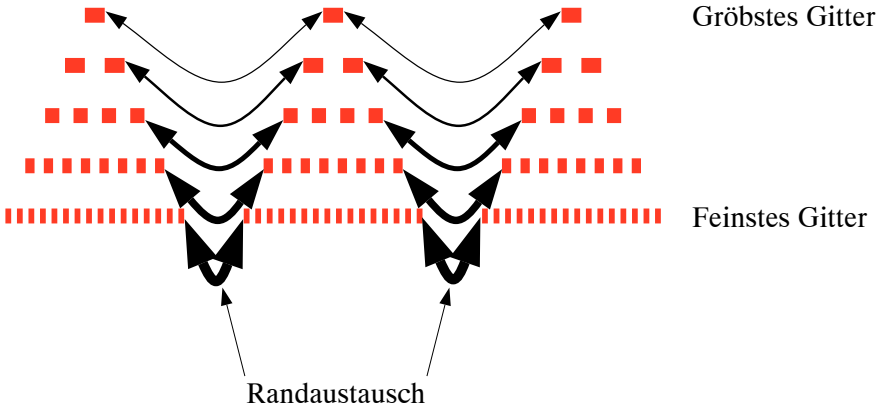

Abbildung 27: Schematische Darstellung eines eindimensionalen Multigrid-Verfahrens.

z.B. das Jacobi-Verfahren benutzt, um eine Approximation der Lösung auf dem jeweiligen Gitter zu berechnen. Dabei fließt die Information zwischen den verschiedenen Gitterebenen, indem eine Lösung von einem groben auf das nächst feinere Gitter extrapoliert bzw. von dem feineren Gitter auf das nächst gröbere Gitter projiziert wird. Da eine vollständige Einführung in die Technik der Multigrid-Verfahren den Rahmen dieser Arbeit sprengen würde, kann hier nur auf die entsprechende Einführungsliteratur, wie z.B. [9] verwiesen werden.

Wichtig ist hier, daß bei der Parallelisierung eines Multigrid-Verfahrens durch eine Gebietsaufteilung die Kommunikation von Randwerten der verschiedenen Gitter stattfinden muß. In einem zweidimensionalen Beispiel (mit einer dünn-besetzten Matrix) bedeutet dies für ein Gitter mit n^2 Punkten, daß die Menge der Rechenoperation und damit die benötigte Zeit in $O(n^2)$ liegt, und die Menge der auszutauschenden Daten und damit die Zeit für die Kommunikation in $O(n)$ (zur O-Notation siehe [77]). Dies bedeutet, daß für kleine n, also für grobe Gitter, für die Berechnung weniger Zeit benötigt wird, als für die Kommunikation. Für große n hingegen, also für die feineren Gitter, wird mehr Rechenarbeit benötigt. Das Programm enthält also Teile, bei denen die meiste Zeit für Kommunikation benötigt wird (die also *communication-bound* sind), und Teile, bei denen die meiste Zeit von Berechnungen gebraucht wird (die also *computation-bound* sind). Gerade bei einer relativ langsamen Verbindung sind Programme, die *communication-bound* sind, nur schwer effizient einsetzbar. Insofern stellt ein Multigrid-Verfahren eine echte Herausforderung an die vorgenommenen Optimierung dar, da auf einem Metacomputer bedingt durch die langsamere Kommunikation erst später der Übergang von *communication-bound*

zu *computation-bound* stattfindet.

Um den dynamischen Lastausgleich auch an einem adaptiven Programm testen zu können, können durch entsprechende Angaben in der Konfigurationsdatei in dieser Anwendung einzelne Teilgebiete weiter verfeinert werden, was zur Folge hat, daß diese Gebiete einen deutlich höheren Berechnungsaufwand benötigen. Analog ist auch wieder eine Vergröberung dieser verfeinerten Gebiete möglich, so daß ein adaptives Verhalten simuliert wird. Im Sinne des vorgestellten Verfahrens für den dynamischen Lastausgleich gehören verfeinerte Gebiete zu einer anderen Prozessorklasse. Als Prozessorklasse eines virtuellen Prozessors wird die Anzahl der Gitterebenen, die er berechnen muß, genommen. Dies bewirkt genau, daß virtuelle Prozessoren in der gleichen Prozessorklasse ungefähr die gleiche Rechendauer benötigen.

Als Beispiel wird die partielle Differentialgleichung

$$f_{xx}(x,y) + f_{yy}(x,y) = 32x(x-1) + 32y(y-1), x, y \in [0,1] \tag{14}$$

mit den Randbedingungen

$$f(0,\cdot) = f(1,\cdot) = f(\cdot,0) = f(\cdot,1) = 0, \tag{15}$$

benutzt, bei der analytisch die Lösung:

$$f(x,y) = 16x(x-1)y(y-1) \tag{16}$$

bestimmt werden kann. Eine Diskretisierung dieser Differentialgleichung (vgl. z.B. [9]) führt nun zu einem dünnbesetzten Gleichungssystem, das mit einem Multigridverfahren gelöst werden kann.

Dieses Testbeispiel stellt natürlich nur den rechenintensiven Kern einer typischen Anwendung dar. Im allgemeinen ist für die Aufstellung des Gleichungssystems ein deutlich höherer Rechenaufwand erforderlich, der aber ohne Kommunikation zwischen den Prozessoren parallel erfolgen kann. Bei einer vollständigen Anwendung wird also der Übergang von *communication-bound* zu *computation-bound* eher erfolgen, was eine bessere Leistungsfähigkeit zur Folge hat.

7.9.3 Zweidimensionale DGL und Schwarz-Verfahren

Hier wird die gleiche Differentialgleichung wie im vorherigen Abschnitt verwendet. Bei einem Schwarz-Verfahren wird nur ein Gitter benutzt, auf dem die vollständigen Berechnungen stattfinden. Dazu wird eine lokale Lösung desjenigen Teils der Differentialgleichung berechnet, die auf einem Prozessor bearbeitet wird. Anschließend werden die Randgebiete ausgetauscht, die einen sogenannten Überlappungsbereich bilden (vgl. auch [144]). Dieses Austauschen wird solange wiederholt, bis eine Lösung mit ausreichender Genauigkeit gefunden wurde.

Da das Lösen der lokalen Gleichungssysteme länger dauert als der Datenaustausch, ist ein Schwarz-Verfahren *computation bound*. Zwar zeigen Schwarz-Verfahren ein recht geringe Konvergenzgeschwindigkeit, trotzdem werden sie oftmals aufgrund ihrer einfachen Struktur in Programmen eingesetzt (vgl. z.B. [144]).

8 Ergebnisse

In diesem Kapitel werden die Ergebnisse vorgestellt, die mit dem in Kapitel 7 beschriebenen Prototypen auf verschiedenen Rechnerkonfigurationen erzielt wurden.

8.1 Vergleich von PACX und PVM

Der Prototyp unterstützt neben einer sequentiellen Version (vgl. Abschnitt 7.4.3) insgesamt drei verschiedene Kommunikationsbibliotheken, nämlich MPI, PVM und PACX. Von diesen Bibliotheken ermöglichen nur PACX und PVM die Kopplung verschiedener Rechner (vgl. Abschnitt 7.2). Die allgemeine Erfahrung ist, daß die Leistung von PVM hinter der Leistung von MPI zurückbleibt [76]. Ein Vergleich der Leistungsstärke von PVM und PACX stand aber noch aus. Deshalb wurde für das CoFu-Skelett gesondert untersucht, wie sich die Laufzeit einer Anwendung in Abhängigkeit von der eingesetzten Kommunikationsbibliothek verhält. Dabei sollte in erster Linie die Implementierung der externen Kommunikationsschnittstelle getestet werden. Es wurde der Multigrid-Löser (vgl. Abschnitt 7.9.2) einerseits mit der PACX-Implementierung des CoFu-Skelettes, andererseits mit der PVM-Implementierung gerechnet, wobei zwei verschiedene Szenarien getestet wurden. Zunächst wurde die Anwendung auf der T3E-256 bearbeitet. Bei diesem Test wurde das Programm zweimal eigenständig gestartet, so daß die Kommunikation zwischen diesen beiden Teilen nicht über das schnelle interne Netzwerk erfolgte, sondern über TCP/IP. Dabei wurde aber kein externes Netzwerk, sondern nur die eigentliche TCP/IP-Schnittstelle benutzt. Dadurch konnte die Implementierung der externen Kommunikation getestet werden, ohne daß Netzschwankungen das Ergebnis beeinflußten. Bei einer Konfiguration von zweimal vier Prozessoren ergab sich für die PACX-Version eine Laufzeit von 81.3 Sekunden, für die PVM-Version hingegen betrug die Laufzeit 196.2 Sekunden. In einem zweiten Test wurden die beiden Cray T3Es des FZJs eingesetzt. Die unterschiedliche Leistung der beiden Rechner wurde dabei nicht berücksichtigt. Die gesamte Bearbeitungszeit für die PACX-Version betrug dabei 160.2 Sekunden und für die PVM-Version 313.1 Sekunden.

Die PVM-Implementierung benötigt also ungefähr die doppelte Laufzeit im Vergleich zur PACX-Implementierung. Folgende Punkte sind dafür in erster Linie verantwortlich:

- PACX hat auf jedem Rechner zwei Prozessoren zur Verfügung, die nur für die externe Kommunikation benutzt werden. PVM hingegen benötigt keine reservierten Prozessoren. Dadurch geht auf jedem Prozessor Rechenleistung für die Verarbeitung der externen Kommunikation verloren.
- PVM übernimmt die vollständige Verwaltung der Ein- und Ausgabepuffer für die Anwendung, PACX als MPI-Erweiterung hingegen überläßt dies der Anwendung. Dadurch ist bei PVM ein Kopieren der Daten auf jeden Fall erforderlich. Dieses zeitaufwendige Kopieren wird bei PACX vermieden.

Da das Ziel dieser Arbeit eine möglichst effiziente Programmbearbeitung war, wurde PACX für alle weiteren Tests benutzt. Trotzdem wurde die PVM-Schnittstelle weiter-

entwickelt, so daß eine Grundversion auch auf Systemen, die kein MPI bzw. PACX zur Verfügung stellen, das CoFu-Skelett einsetzbar ist.

8.2 Zusätzlicher Aufwand durch virtuelle Prozessoren

Ein wichtiges Argument gegen den Einsatz von virtuellen Prozessoren ist, daß der zusätzliche Aufwand, der durch die Verwaltung der notwendigen Datenstruktur entsteht, die Zeitersparnis zunichte macht. Deshalb wurde ein Vergleich der Laufzeiten eines reinen MPI-Programms und einer Implementierung des gleichen Programms mittels des CoFu-Skelettes durchgeführt. Hierzu wurde das Testbeispiel der Lösung der eindimensionalen Differentialgleichung mittels des Jacobi-Verfahrens eingesetzt (vgl. Abschnitt 7.9.1). Bei dieser Anwendung war die zu übertragende Datenmenge gering; lediglich eine Fließkommazahl mußte zwischen den benachbarten Prozessoren ausgetauscht werden. Dadurch war der Einfluß der Kommunikation auf die Gesamtrechenzeit sehr gering. Weiterhin wurde das *scheduling* der virtuellen Prozessoren auf einem physikalischen Prozessor so eingestellt, daß keine Überlappung von Kommunikation und Berechnung stattfand. So konnte der durch das CoFu-Skelett verursachte Aufwand ermittelt werden. Abbildung 28 zeigt

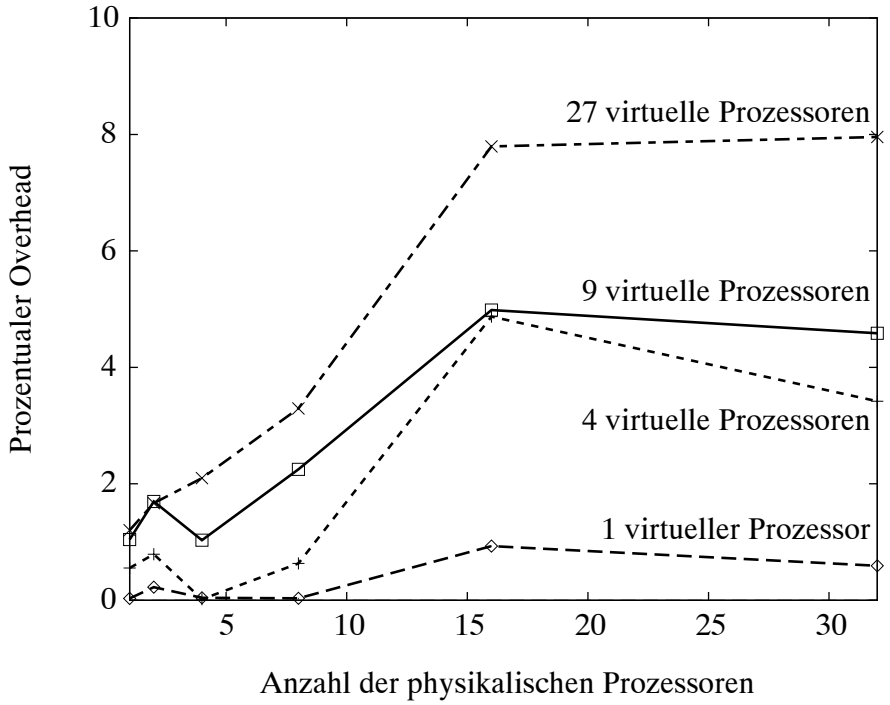

Abbildung 28: Prozentualer zusätzlicher Aufwand durch Benutzung des CoFu-Skelettes bei gleichbleibendem Gesamtberechnungsaufwand beim Einsatz von 1, 4, 9 und 27 virtuellen Prozessoren pro physikalischem Prozessor.

den prozentualen zusätzlichen Aufwand, der entsteht, wenn man statt MPI das CoFu-Skelett benutzt. Der Verwaltungsaufwand steigt zwar an, bleibt aber unter 8%. Setzt

man hingegen nur 9 virtuelle Prozessoren ein, ist der zusätzliche Aufwand geringer als 5%. Der Anstieg des prozentualen Overheads wird durch den sogenannten „Oberfläche-zu-Volumen-Effekt“ (*Surface-to-Volume Effect* [44]) verursacht: Der Aufwand für das CoFu-Skelett bleibt pro physikalischem Prozessor absolut gesehen gleich (bei gleichbleibender Anzahl von virtuellen Prozessoren), allerdings verringert sich die Berechnungsmenge pro physikalischem Prozessor und somit die Rechenzeit. Dadurch steigt der prozentualen Anteil des Verwaltungsaufwandes an. Dies wird deutlich, wenn man bei der Nutzung von

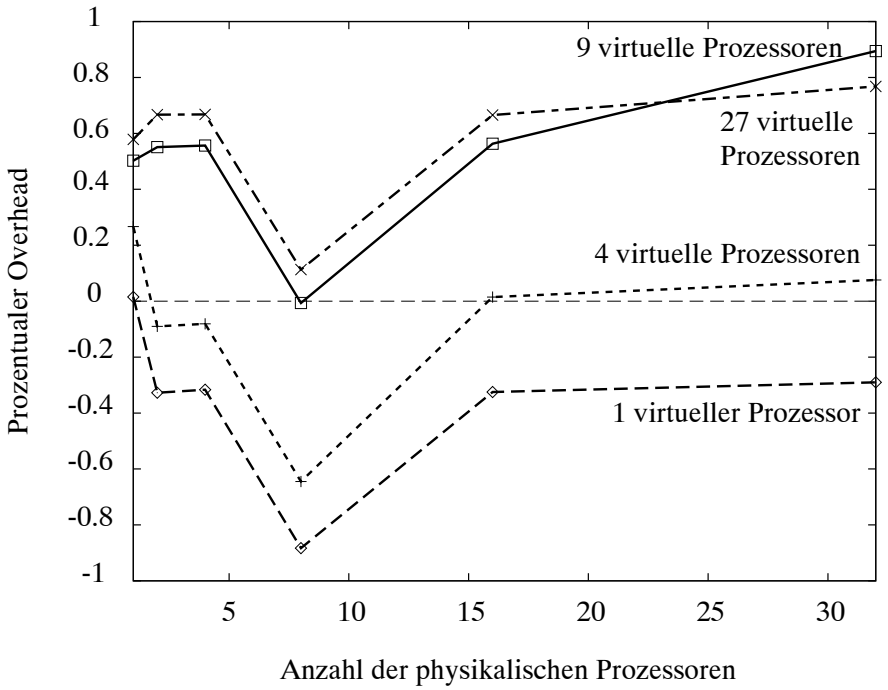

Abbildung 29: Prozentualer zusätzlicher Aufwand durch Benutzung des CoFu-Skelettes bei steigendem Gesamtberechnungsaufwand.

weiteren Prozessoren auch die Menge der Berechnungen im gleichen Verhältnis wachsen läßt. Abbildung 29 zeigt wiederum den prozentualen zusätzlichen Aufwand, allerdings wurde hier die Problemgröße jeweils an die Anzahl der physikalischen Prozessoren angepaßt, so daß immer die gleiche Menge an Rechenarbeit auf jedem einzelnen Prozessor notwendig war. Wie zu sehen ist, bleibt der Aufwand unter 1% im Vergleich zu einer MPI-Implementierung. Abbildung 30 zeigt den zusätzlichen Aufwand für unterschiedliche Anzahlen von virtuellen Prozessoren auf jedem der betrachteten 16 Prozessoren. Es ist zwar ein leichter Anstieg der Rechenzeit mit wachsender Anzahl von virtuellen Prozessoren zu beobachten, jedoch bleibt der Aufwand auch für große virtuelle Prozessorzahlen sehr gering.

Zusammengefaßt ist der zusätzliche Aufwand, der durch Benutzung des CoFu-Skelettes entsteht, gering – er liegt unter 8% für praxisrelevante Anwendungen. Setzt man einen hohen Berechnungsaufwand auf jedem Prozessor voraus, bleibt der Aufwand sogar unter 1%. Dabei wurden die Vorteile des CoFu-Skelettes, insbesondere die auch für homogene

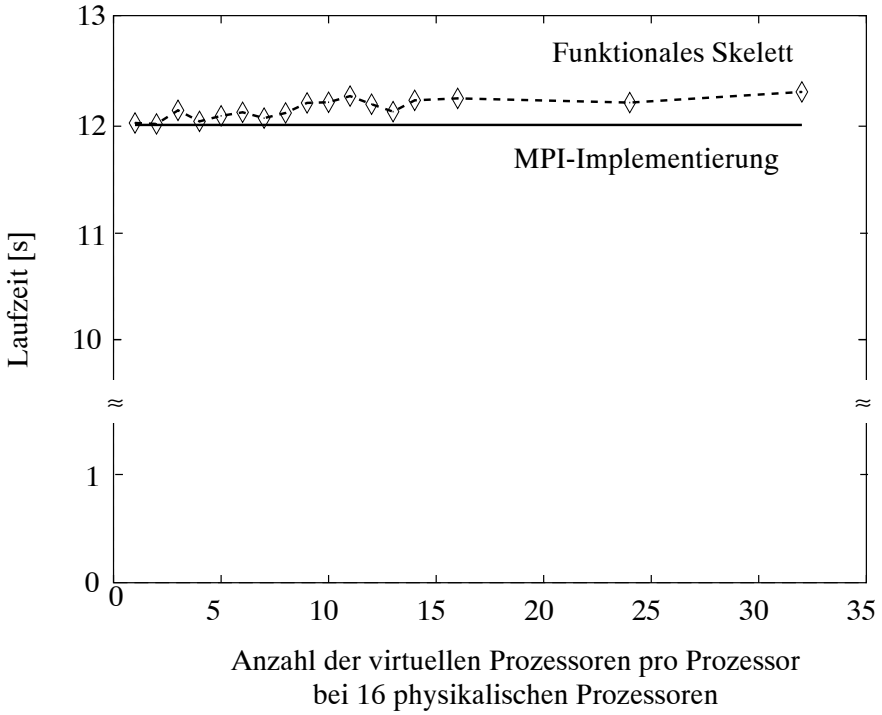

Abbildung 30: Auswirkung der Anzahl der virtuellen Prozessoren auf die Laufzeit.

Systeme wichtige Überlappung von Kommunikation und Berechnung nicht genutzt. Eine der wichtigen Anforderungen an diese Arbeit, nämlich daß eine Anwendung, die die hier vorgestellten Optimierungen nutzt, weiterhin auf einem homogenen System effizient einsetzbar ist, ist somit erfüllt.

8.3 Homogene Systeme und CoFu

Es ist nicht nur wünschenswert, daß eine Anwendung weiterhin auf einem homogenen System einsetzbar ist, sondern, daß sich in diesem Fall eine Leistungssteigerung ergibt. Dies wurde anhand des Lösers der zweidimensionalen Differentialgleichung mit Hilfe des Schwarzschen Verfahrens 7.9.3 auf der Cray T3E-256 des FZJs untersucht. Abbildung 31 vergleicht die Laufzeiten, die sich beim herkömmlichen Programm, also bei der Nutzung eines virtuellen Prozessors pro physikalischem Prozessor und bei der Überlappung von Kommunikation und Berechnung durch den Einsatz von 9 virtuellen Prozessoren ergeben haben[26]. Dabei wurde eine Diskretisierung von 1536 x 1536 Punkten verwendet. Dieses Beispiel kann erst auf mindestens vier Prozessoren bearbeitet werden, da ansonsten der

[26]Um den Implementierungsaufwand gering zu halten, wurde keine eigene Implementierung der Programme mit MPI vorgenommen. Als Vergleich wird deshalb immer die Berechnung mit einem virtuellen Prozessor eingesetzt. In Abschnitt 8.2 wurde der zusätzliche Aufwand im Vergleich zu einem reinen MPI-Programm bestimmt, der bei unter 1% liegt, solange ein großer Rechenanteil vorhanden ist. Daraus läßt sich in guter Näherung der Vergleich mit einer MPI-Implementierung ermitteln.

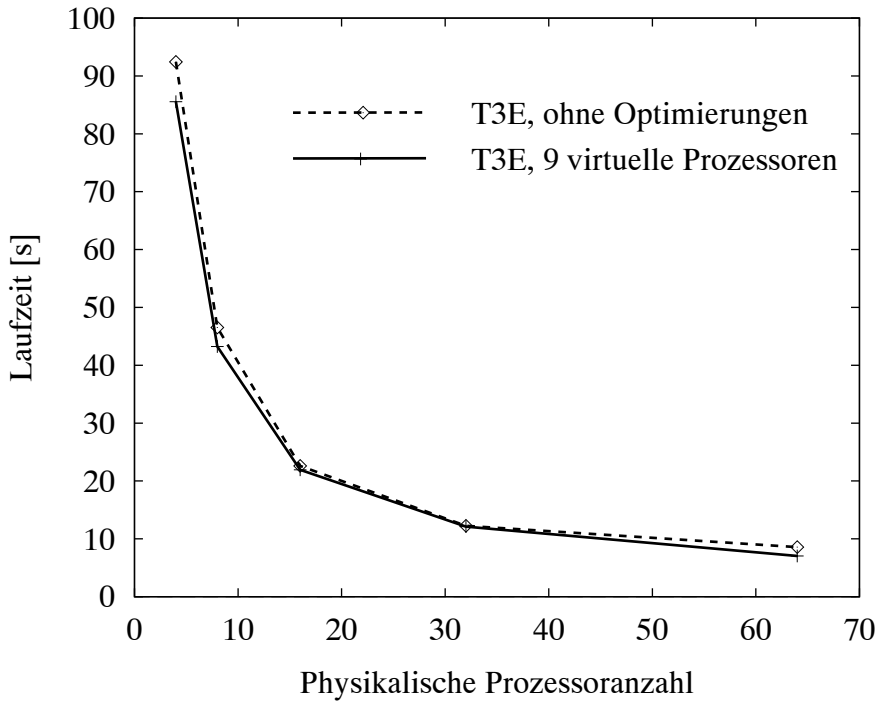

Abbildung 31: Laufzeiten für das Schwarzsche Verfahren auf einem homogenen System.

Hauptspeicher nicht ausreicht[27]. Durch die Überlappung von Kommunikation und Berechnung wird zwischen 1% und 17% der Rechenzeit eingespart.

Berücksichtigt man den Aufwand von ca. 1% (vgl. Abschnitt 8.2) durch den Einsatz des CoFu-Skelett, so ergibt sich immer ein Geschwindigkeitsgewinn. Durch den Einsatz des CoFu-Skelett ist also nicht nur weiterhin ein effizienter Einsatz auf einem homogenen System möglich, sondern es ergibt sich auch in diesen Fällen eine Reduktion der Gesamtbearbeitungszeit.

8.4 Auswirkungen der Kommunikationsoptimierung

Wie in Kapitel 3 und 5 gezeigt wurde, ist die Optimierung der externen Kommunikation durch Zusammenfassen externer Nachrichten von erheblicher Bedeutung beim Einsatz eines Metacomputers. In diesem Abschnitt werden die Auswirkungen dieses Zusammenfassens anhand des Multigrid-Lösers der zweidimensionalen Differentialgleichung (vgl. Abschnitt 7.9.2) auf einem Metacomputer, der aus der Cray T3E-256 des FZJs und der IBM SP2 der GMD besteht, untersucht. Bei diesem Experiment wurden noch keine weiteren Optimierungen wie der statische Lastausgleich oder die Überlappung von Kommunikation und Berechnung durchgeführt. Abbildung 32 vergleicht die Laufzeit dieser Anwendung einmal mit und einmal ohne Zusammenfassen. Dabei bedeutet die Angabe „4“ Prozesso-

[27]Die Cray T3E unterstützt z.Zt. noch keinen virtuellen Speicher, so daß man auf die maximal 128 MByte Hauptspeicher pro Prozessor beschränkt ist, über die die Cray T3E am FZJ verfügt.

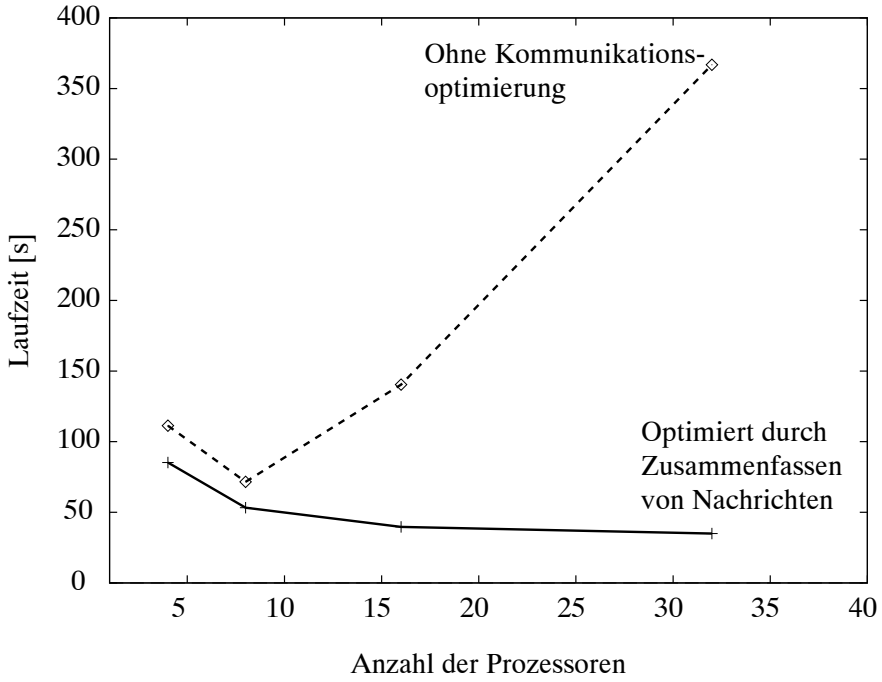

Abbildung 32: Laufzeiten des Multigrid-Lösers auf einem Metacomputer bestehend aus der Cray T3E und der IBM SP2.

ren, daß zwei Prozessoren der Cray T3E und zwei Prozessoren der IBM SP2 eingesetzt wurden. Ähnlich zu den Erfahrungen beim RTB NRW wurde bei mehr als 8 Prozessoren ein Anstieg der Rechenzeit beobachtet. Durch das Zusammenfassen der externen Nachrichten wurde eine signifikante Beschleunigung erreicht. Im Gegensatz zum Vorgehen im RTB NRW war hier aber keinerlei weitere Änderung an der Anwendung notwendig, sondern diese Optimierung wurde vom CoFu-Skelett transparent für die Anwendung vorgenommen – die zeitaufwendige manuelle Optimierung der Kommunikationsstruktur ist überflüssig.

8.5 Statischer Lastausgleich und Überlappung von Kommunikation und Berechnung

Der aus der Cray T3E-256 und IBM SP2 bestehende Metacomputer hat unterschiedlich leistungsfähige Komponenten – ein einzelner Cray T3E-256 Prozessor ist ungefähr doppelt so schnell wie ein IBM SP2 Prozessor. Aus Tabelle 4 auf Seite 63 entnimmt man, daß dieses Verhältnis durch den Einsatz von 9+3 virtuellen Prozessoren auf der T3E und 9-3 virtuellen Prozessoren auf der SP2 ausgeglichen wird. Auch ist es durch den Einsatz der virtuellen Prozessoren nun möglich, eine Überlappung von Kommunikation und Berechnung zu erreichen. Abbildung 33 stellt die so erhaltenen Werte gegenüber, die mit der gleichen Konfiguration wie im vorherigen Abschnitt erreicht wurden. Auch hier bedeutet die Angabe „4“ Prozessoren, daß zwei Prozessoren der Cray T3E und zwei Prozessoren

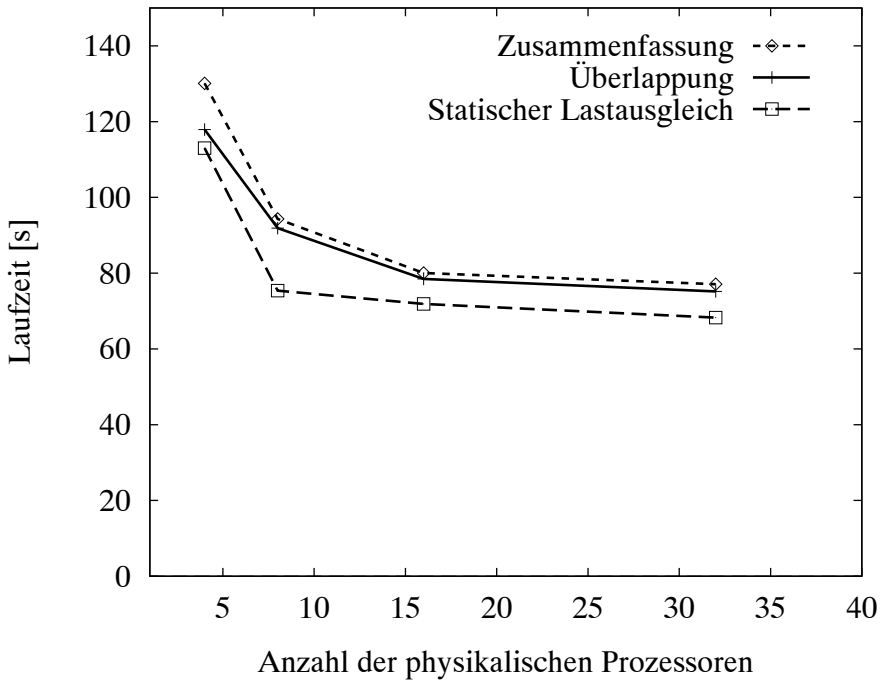

Abbildung 33: Statischer Lastausgleich und Überlappung von Kommunikation und Berechnung auf einem Metacomputer.

der IBM SP2 eingesetzt wurden. Sowohl der Lastausgleich als auch die Überlappung bringen jeweils eine weitere Reduzierung der Gesamtlaufzeit. Auch diese Optimierungen sind möglich, ohne daß eine Änderung an der Anwendung vorgenommen werden mußte.

Abbildung 34 zeigt einen Ausschnitt aus der VAMPIR-Analyse der Anwendung auf dem Metacomputer. In der oberen Hälfte sind die Prozessoren der IBM SP2, in der unteren Hälfte die Prozessoren der Cray T3E eingezeichnet. Die Aktion mit der Bezeichnung `region:compute` ist dabei eine CoFu-Funktion, hier also die Berechnung der Lösung des lokalen Gleichungssystems auf einem virtuellen Prozessor. Es ist zu sehen, daß auf einem SP2 Prozessor die Funktion `region:compute` sechsmal aufgerufen wird, auf einem T3E Prozessor hingegen zwölfmal (aus Platzgründen wird im unteren Teil des Bildes statt des Namens die interne Nummer „596“ genutzt). Dies entspricht genau der Simulation der virtuellen Prozessoren auf den einzelnen physikalischen Prozessoren. Durch die unterschiedliche Anzahl der virtuellen Prozessoren wird die unterschiedliche Rechenkapazität ausgeglichen – die Zeit für die Simulation der sechs virtuellen Prozessoren auf einem Knoten der SP2 benötigt ungefähr die gleiche Zeit wie die Simulation der zwölf virtuellen Prozessoren auf einem Knoten der T3E. Auch die Überlappung von Kommunikation und Berechnung wird deutlich.

In der VAMPIR-Analyse in Abbildung 34 ist zu sehen, wie nach der Berechnung der ersten CoFu-Funktion bereits eine externe Nachricht geschickt wird. Das Zusammenfassen der externen Nachrichten wird von VAMPIR nicht angezeigt, da es intern in der Kommunikationsbibliothek PACX implementiert ist. VAMPIR zeigt in diesen Fällen die einzelnen

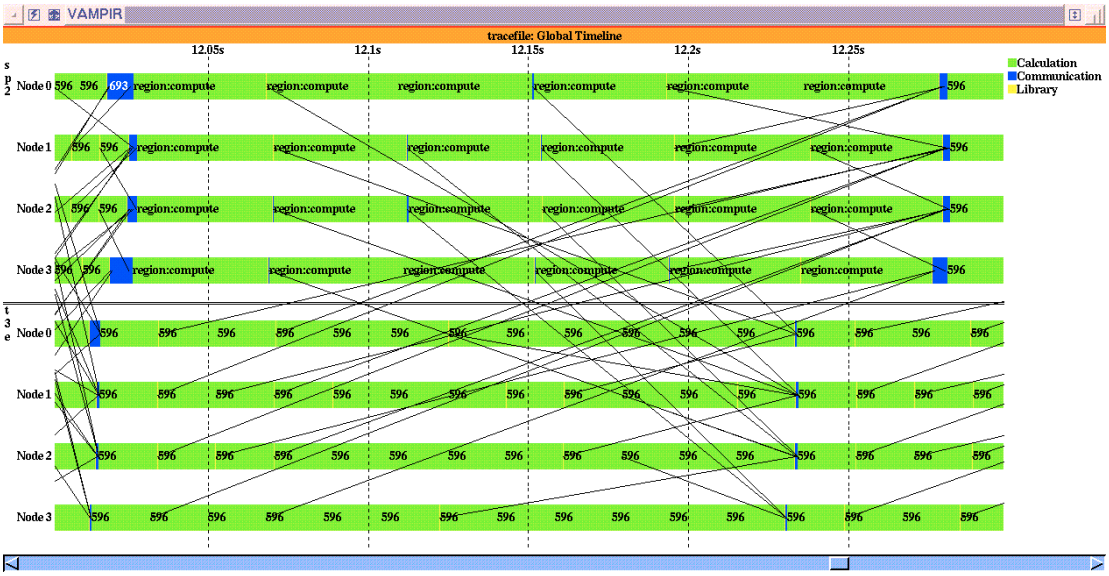

Abbildung 34: VAMPIR-Analyse der Bearbeitung auf dem Metacomputer

Nachrichten, wie sie aus der logischen Sicht der Anwendung geschickt werden und nicht die tatsächlich übertragenen Nachrichten.

Die Kombination von statischem Lastausgleich, dem Zusammenfassen von Nachrichten und der Überlappung von Kommunikation und Berechnung ergibt eine signifikante Beschleunigung einer eng-gekoppelten Anwendung auf einem Metacomputer.

8.6 Leistungsvergleich einer Anwendung auf einem Metacomputer und auf einem homogenen System.

Im RTB NRW (vgl. Abschnitt 3) ist es nicht gelungen, durch Hinzunahme weiterer, weniger leistungsstarker Prozessoren zu einem homogenen System die Bearbeitungszeit einer Anwendung zu reduzieren. Dadurch wird aber einer der Ansprüche des Metacomputings, nämlich die Nutzung der Rechenkapazität (und nicht nur der Speicherkapazität) eines verteilten Systems, nicht erfüllt. In diesem Experiment wurde das Schwarzsche Verfahren auf dem aus der T3E und SP2 bestehenden Metacomputer getestet. Abbildung 35 zeigt die Laufzeit der Anwendung auf den beiden homogenen Systemen und auf dem Metacomputer, wobei für die Metacomputer-Messungen immer die gleiche Anzahl von Prozessoren auf beiden Maschinen eingesetzt wurde. Durch Hinzunahme von Prozessoren der SP2 wurde die Gesamtrechenzeit weiter reduziert. Es ist also gelungen, die tatsächliche Rechenleistung des Gesamtsystems zu nutzen und in eine verkürzte Bearbeitungszeit umzusetzen.

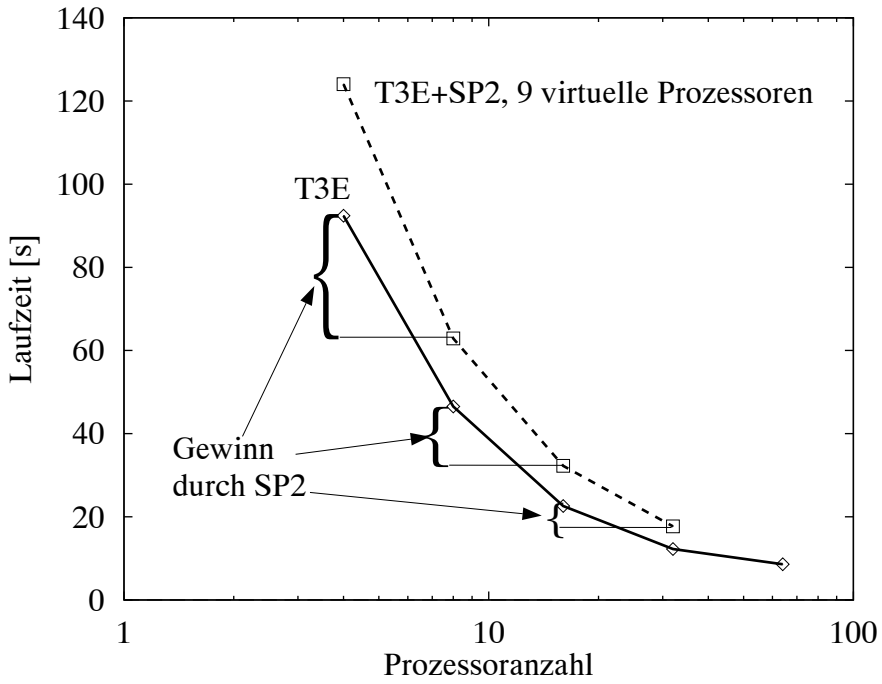

Abbildung 35: Vergleich des Schwarzschen Verfahrens auf einem Metacomputer mit der Berechnung auf einem homogenen System.

8.7 Dynamischer Lastausgleich mittels CoFu.

Der dynamische Lastausgleich wurde ebenfalls in einem Testszenario evaluiert. Dazu wurde eine Differentialgleichung eingesetzt, bei der einige Teilgebiete feiner diskretisiert sind bzw. zur Laufzeit feiner diskretisiert werden müssen. Die Anwendung wurde auf 4 Prozessoren einer T3E eingesetzt. Abbildung 36 zeigt die Rechenzeit der einzelnen Prozessoren für jeden der insgesamt 14 Rechenschritte. Es ist zu sehen, daß Prozessor 0 von Anfang an etwas mehr Rechenzeit benötigt, als die anderen Prozessoren. Dies wird durch feiner diskretisiertes Teilgebiet verursacht, das von diesem Prozessor bearbeitet wird. Im fünften Schritt schließlich werden weitere Teilgebiete feiner diskretisiert, so daß sich ein massives Lastungleichgewicht ergibt. Wie schon bei der Anwendung BD (vgl. Abschnitt 3.2) geht die Rechenzeit der Prozessoren verloren, da sie auf den langsameren Prozessor warten müssen. Während das initiale, leichte Ungleichverteilung durch einen statischen Lastausgleich ausgeglichen werden könnte, ist das adaptive Verhalten und das daraus resultierende erneute Ungleichgewicht durch statische Methoden nicht zu verhindern.

Abbildung 37 zeigt die Rechenzeiten der einzelnen Prozessoren für den gleichen Versuch, bei dem jetzt allerdings der dynamische Lastausgleich aktiviert wurde. Dabei wurden auf jedem physikalischen Prozessor anfänglich 36 virtuelle Prozessoren eingesetzt. Es ist zu sehen, daß bereits nach dem ersten Schritt die initiale Datenverteilung korrigiert wurde und zu einer besseren Verteilung der Rechenlast auf die Prozessoren führte. Im weiteren ist diese Verteilung stabil, bis wiederum im fünften Schritt durch die Verfeinerung einzelner

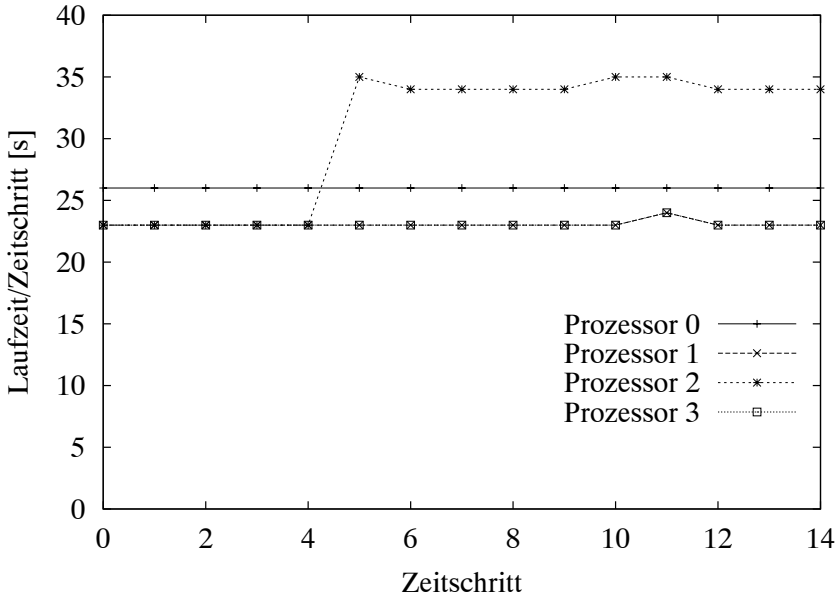

Abbildung 36: Rechenzeit der einzelnen Prozessoren pro Zeitschritt ohne dynamischen Lastausgleich.

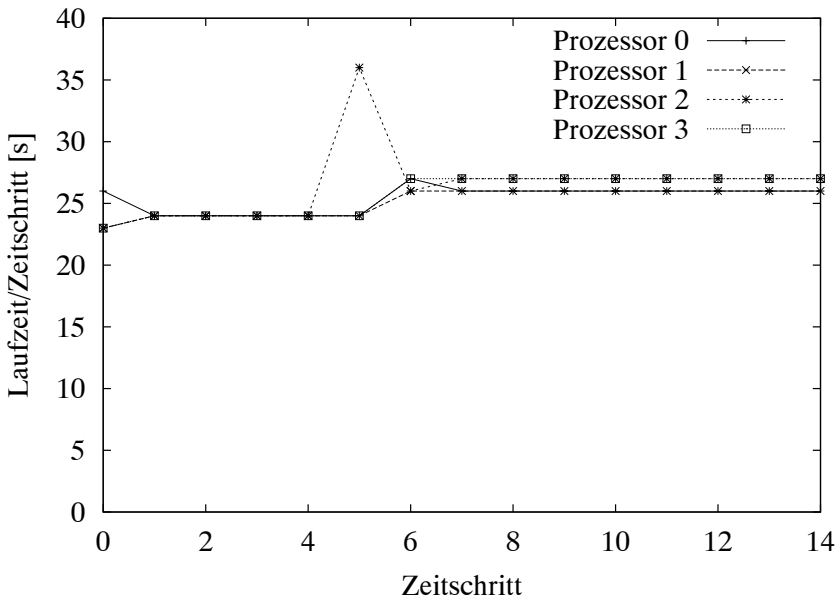

Abbildung 37: Rechenzeit der einzelnen Prozessoren pro Zeitschritt mit dynamischen Lastausgleich.

Gebiete das Gleichgewicht erneut gestört wurde. Im ersten Schritt des Lastausgleichsalgorithmus wird bereits eine deutlich bessere Lastverteilung erreicht. In einem weiteren Schritt schließlich wird diese Verteilung noch optimiert. Im weiteren ist der jetzt erreichte Zustand stabil, es tritt kein *thrashing*, also ein kontinuierliches Verschieben einzelner Prozesse, auf. Das weiter existiertende leichte Lastungleichgewicht von ca. 0.6 Sekunden kann durch Migration eines virtuellen Prozessors nicht weiter ausgeglichen werden. Die Laufzeit des Programms konnte durch die Anwendung des dynamischen Lastausgleichsalgorithmus von 484 auf 412 Sekunden reduziert werden.

8.8 Laufzeitvorhersagen für große CoFu-Programme

Ein wichtiges Einsatzgebiet für Metacomputing ist die Bearbeitung von Anwendungen, die schon aufgrund des mangelnden Hauptspeicherplatzes nicht auf einer einzelnen Maschine bearbeitet werden können. Um dies exemplarisch zu zeigen, wurden die beiden Cray T3E Systeme zusammen benutzt, um ein großes Gleichungssystem zu lösen. Insgesamt standen somit 243+510 Prozessoren zur Verfügung[28], wobei ein Prozessor der T3E-256 ca. 2.5-mal so leistungsfähig ist wie ein Prozessor der T3E-512[29]. Für die Praxis ist es wichtig, die Rechenzeit gerade für solche großen Anwendungen im voraus abschätzen zu können, um die Prozessoren auf allen beteiligten Maschinen ausreichend lange zur Verfügung zu haben[30]. Das formale CoFu-Modell berücksichtigt die Kommunikationszeit nicht mit. Wenn aber eine vollständige Überlappung der Kommunikation mit Berechnungen möglich ist, kann eine einfache Heuristik eingesetzt werden, indem man die Kommunikationszeit vernachlässigt und nur die reine Rechenzeit berücksichtigt.

Zunächst wurde dazu eine zwei-dimensionale Differentialgleichung mit Hilfe des Schwarzschen Verfahrens auf einem kleinen Gebiet auf vier Prozessoren der T3E-256 allein gelöst. Anschließend wurde das Gebiet so vergrößert, daß beim Einsatz von 256 Prozessoren jeder Prozessor die gleiche Datenmenge zu bearbeiten hatte, wie die vier Prozessoren im kleineren Beispiel. Es war also zu erwarten, daß bei einer Bearbeitung des großen Datensatzes auf 256 Prozessoren die gleiche Rechenzeit wir für die Bearbeitung des kleinen Datensatzes auf 4 Prozessoren notwendig ist, und daß sich diese Zeit halbiert, wenn man den großen Datensatz auf 512 Prozessoren berechnet. Tatsächlich ergaben sich die in Tabelle 6 dargestellten Laufzeiten. Die Abweichung von der tatsächlichen Laufzeit ist also nur mini-

[28] Es muß berücksichtigt werden, daß auf jeder Maschine zwei Prozessoren für die Kommunikation reserviert werden müssen, und daß die T3E-256 nur 247 Prozessoren für parallele Anwendungen zur Verfügung stellt.

[29] Zunächst unterscheiden sich die beiden Systeme durch die Taktfrequenz: die T3E-256 wird mit 450 MHZ getaktet, die T3E-512 nur mit 300 MHZ. Weiter können bei der T3E-512 aufgrund des dort verwendeten älteren Chip-Satzes die *stream-buffer* [97] nicht eingesetzt werden. Die hier benutzten Anwendungen setzen aber die *stream-buffer* sehr effektiv ein, so daß ein überdurchschnittlich großer Leistungsunterschied zustande kommt.

[30] Wie in Abschnitt 2.4 beschrieben wurde, wird das *job-scheduling* für verteilte Systeme noch nicht unterstützt, so daß derart große Berechnungen z.Zt. noch manuell mit den Verantwortlichen der einzelnen Maschinen abgestimmt werden müssen und nur zu speziellen Testzeiten durchführbar sind. Da derartige Tests nur begrenzt möglich sind, ist eine vorherige Absprache der benötigten Rechenzeit von entscheidender Bedeutung. Aber auch im zukünftigen Einsatz von Metacomputing in Produktionsumgebungen ist die Angabe der insgesamt benötigten Rechenzeit notwendig.

	Kleiner Datensatz	Großer Datensatz	
	4 Prozessoren	256 Prozessoren	512 Prozessoren
Gemessene Laufzeit [s]	220.9	219.9	111.4
Vorausgesagte Laufzeit [s]		220.9	110.4
Prozentuale Abweichung		0.4%	0.9%

Tabelle 6: Vorausgesagte und tatsächliche Laufzeit einer CoFu-Anwendung.

mal. Dies ist nur möglich, wenn die Kommunikation nur einen minimalen Einfluß auf die Gesamtbearbeitungszeit hat, daß also die Kommunikation vollständig mit Berechnungen überlappt war.

Als nächstes Experiment wurden nun beide T3E-Rechner zusammen eingesetzt, um den oben erwähnten großen Datensatz zu bearbeiten. Dazu wurde der kleine Testdatensatz auf vier Prozessoren der T3E-256 berechnet. Es ergaben sich dabei die in den linken Spalten von Tabelle 7 enthaltenen Ergebnisse. Demnach bietet, wie anfangs bereits erwähnt, ein Prozessor der T3E-512 nur ca. 40% der Leistungsfähigkeit eines Prozessors der T3E-256. Bei einer Verteilung des großen Testdatensatzes auf insgesamt 243 Prozessoren der T3E-256 und 510 Prozessoren der T3E-512 steht damit insgesamt theoretisch eine Rechenkapazität von $243 + \frac{87.3}{220.9} \cdot 510 \approx 444.6$ Prozessoren der T3E-256 zur Verfügung. Unter Vernachlässigung der Kommunikationszeit erwartet man somit eine Rechenzeit von $87.3 \cdot 256/444.6 \approx 50.3$ Sekunden. Die gemessene Laufzeit von 53.2 Sekunden (vgl. Tabelle

	4 Prozessoren T3E-256	4 Prozessoren T3E-512	243 Prozessoren T3E-256 510 Prozessoren T3E-512
Gemessene Laufzeit [s] für Datensatz 1	87.3	220.9	53.2
Vorausgesagte Laufzeit [s] (Prozentuale Abweichung)			50.3 (5.4%)

Tabelle 7: Vorausgesagte und tatsächliche Laufzeit einer CoFu-Anwendung auf einem heterogenen System.

7) weicht nur um ca. 5.4% von dieser Vorhersage ab.

Als letzter Versuch wurde ein großer Datensatz berechnet, der die volle Speicherkapazität beider Cray-Systeme voll nutzte. Da auf beiden Systemen jeder Prozessor über 128 MByte Speicher verfügt, wird dadurch allerdings eine ungleiche Lastverteilung erreicht. Dadurch sollte die Rechenzeit genauso groß sein wie die Zeit für die Bearbeitung des kleinen Datensatzes auf der T3E-512. Tabelle 8 faßt die erhaltenen Ergebnisse zusammen. In allen hier durchgeführten Experimenten zeigte sich, daß trotz der verwendeten langsamen Verbindung zwischen den T3Es die Laufzeitvorhersage sehr exakt nur mit Hilfe der Laufzeiten für einen kleinen Datensatz möglich war und daß die Kommunikationszeit durch die Überlappung von Kommunikation und Berechnung vernachlässigt werden kann.

	4 Prozessoren T3E-256	4 Prozessoren T3E-512	243 Prozessoren T3E-256 510 Prozessoren T3E-512
Gemessene Laufzeit [s] für Datensatz 2	87.3	220.9	221.5
Vorausgesagte Laufzeit [s] (Prozentuale Abweichung)			220.9 (0.3%)

Tabelle 8: Vorausgesagte und tatsächliche Laufzeit einer CoFu-Anwendung auf einem heterogenen System.

8.9 Bewertung

In diesem Kapitel wurde der Prototyp des CoFu-Skelettes in zahlreichen Experimenten bzgl. seiner Leistungsfähigkeit untersucht. Es hat sich gezeigt, daß die Erwartungen an diesen Prototyp erfüllt werden: Neben den Optimierungen für Anwendungen auf Metacomputer zeigen sich die Vorteile sogar auch bei homogenen Anwendungen. Damit ist ein entscheidender Beitrag für das Metacomputing erbracht, nämlich der Nachweis, daß sich die theoretischen Vorteile tatsächlich in der Praxis erzielen lassen.

9 Zusammenfassung und Ausblick

Die Simulation immer komplexerer Systeme stellt ständig wachsende Anforderungen an Rechenleistung und Hauptspeichergröße. Metacomputer, d.h. Hochleistungsrechner, die über ein WAN (*wide area network*) gekoppelt sind, bieten theoretisch eine ausreichende Rechenkapazität und Hauptspeichergröße. In der Praxis verhindern bislang jedoch zwei Probleme den effizienten Einsatz von Metacomputern:

- Zunächst ist die Leistung des externen Netzwerkes geringer als die Leistung des internen Verbindungsnetzwerkes eines Hochleistungsrechners. Da es nur eine externe Leitung gibt, reduziert sich die nutzbare Netzleistung im Gegensatz zu einem Parallelrechner, bei dem im allgemeinen viele Leitungen vorhanden sind und auch gleichzeitig benutzt werden können, deutlich. Praktische Erfahrungen zeigen, daß diese geringe Netzleistung die nutzbare Rechenleistung drastisch verringert.
- In einem heterogenen Rechnersystem ist der Lastausgleich von entscheidender Bedeutung. Hier droht der Verlust der Rechenkapazität gerade der leistungsstärkeren Rechner, wenn sie auf andere Rechner warten müssen. Aber durch die Heterogenität des zugrundeliegenden Gesamtsystems wird es auch schwieriger, eine ausgeglichene Lastverteilung zu erreichen. Diese Heterogenität muß bislang auf der Ebene der Anwendung berücksichtigt werden, wodurch die Erstellung von Programmen für Metacomputer noch aufwendiger wird.

Die praktische Relevanz dieser beiden Punkte wurde im Rahmen des Regionalen Testbed NRW anhand einer umfangreichen Fallstudie bestätigt.

Zur Lösung dieser Probleme wurde in dieser Arbeit ein Modell „typischer“ Supercomputer-Anwendungen, das sogenannte CoFu-Modell, entwickelt und formal spezifiziert. Beim Einsatz dieses Modells zur Entwicklung von Anwendungen für Hochleistungsrechner stehen zur Laufzeit zusätzliche Informationen über das Kommunikations- und Rechenverhalten einer Anwendung zur Verfügung. Mit Hilfe dieser Informationen wird die Kommunikation optimiert, indem externe Nachrichten zusammengefaßt werden. Durch den Einsatz des Konzeptes der virtuellen Prozessoren wird durch ein geeignetes *scheduling* gleichzeitig eine Überlappung von Kommunikation und Berechnung erreicht. Diese beiden Optimierungen sind möglich, ohne daß Änderungen an einer CoFu-Anwendung vorgenommen werden müssen.

Die Anzahl der virtuellen Prozessoren, die ein physikalischer Prozessor berechnet, kann der Rechenleistung des Rechners entsprechend gewählt werden. Dadurch wird ein statischer Lastausgleich erreicht, der bereits für zahlreiche Anwendungen ausreicht und den effizienten Einsatz eines Metacomputers ermöglicht.

Als Ergänzung zu diesen Optimierungen wurde ein dynamischer Lastausgleichsalgorithmus entwickelt, der eng in das CoFu-Modell integriert ist und auf der Migration ganzer virtueller Prozessoren beruht. Das hier entwickelte Konzept der Prozessorklassen wird eingesetzt, um die Laufzeit der unterschiedlichen Prozesse auf den verschiedenen Rechnern eines Metacomputers möglichst gut vorhersagen zu können. Mit Hilfe dieser Laufzeitvorhersage wird die Heterogenität des zugrundeliegenden Rechnersystems gezielt eingesetzt,

um virtuelle Prozessoren von den physikalischen Prozessoren bearbeiten zu lassen, die eine möglichst kurze Bearbeitungszeit ermöglichen.

Eine prototypische Implementierung des CoFu-Konzeptes wurde zur Validierung der vorgenommenen Optimierungen eingesetzt. Umfangreiche Messungen haben die Leistungssteigerung nachgewiesen, die sich beim Einsatz des CoFu-Skelettes für Anwendungen ergibt.

Die Implementierung liefert somit einen Nachweis der Vorteile des Konzeptes. Aber Erweiterungen sind denkbar, die das Konzept allgemeiner einsetzbar und komfortabler machen können:

- Das hierarchische CoFu-Modell, das in Abschnitt 4.3 entwickelt wurde, ist noch nicht implementiert. Diese Erweiterung ermöglicht den einfachen Einsatz des CoFu-Programmiermodells für lose gekoppelte Anwendungen, wie z.B. *multi-physics*-Simulationen.
- Der Algorithmus zum dynamischen Lastausgleich kann weiter optimiert werden. Durch den gewählten *greedy*-Ansatz wird nur ein lokales Minimum gefunden. Hier wurde ein Kompromiß zwischen der Berechnungszeit der Lastumverteilung und der Güte der erreichten Verteilung geschlossen. Auch könnte man die Kommunikationsstruktur, die aufgrund des CoFu-Modells bekannt ist, bei der Umverteilung berücksichtigen, wodurch aber der Berechnungsaufwand für die Umverteilung weiter steigt.
- Der dynamische Lastausgleichsalgorithmus kann modifiziert werden, so daß er abschätzen kann, daß ein Lastungleichgewicht eintreten wird: anstatt erst a posteriori beim Eintreten eines Lastungleichgewichtes, das durch die ungleiche Rechenzeiten bemerkt wird, zu reagieren, kann eine a priori Abschätzung über die zukünftige Rechenzeit erstellt werden, die die aktuelle Prozessorklassen zugrunde legt. Somit können virtuelle Prozessoren umverteilt werden, bevor ein Lastungleichgewicht eintritt, was zu einer weiteren Leistungssteigerung von Anwendungen führen kann.
- Die Programmierstruktur ist zunächst ungewohnt. Ein eigener Präprozessor, der speziell strukturierte und um zusätzliche Anweisungen erweiterte Programme in die CoFu-Struktur übersetzt, kann die Akzeptanz des Modells erhöhen. Auch der Einsatz einer graphischen Benutzeroberfläche zur Programmentwicklung kann hier helfen.
- Die automatische Unterstützung von *out-of-core*-Anwendungen, also von Programmen, die Daten auf Sekundärspeichern auslagern, ist denkbar: Einzelne virtuelle Prozesse können, ähnlich zum Migrationsmechanismus, automatisch ausgelagert und bei Bedarf nachgeladen werden. Auch hier ermöglicht das implementierte *scheduling* eine Überlappung von Ein- und Ausgabe mit Berechnungen.

A Vampir-Analyse

In diesem Anhang sind einige Ausschnitte aus der Vampir-Analyse der in Kapitel 3.2 und 3.3 vorgestellten Anwendungen enthalten.

Bei den Abbildungen, die den Zeitverlauf des Programms darstellen (z.B. Abbildung 38), ist auf der horizontalen Achse die Zeit, auf der vertikalen Achse die Prozessornummer aufgetragen. Die farbliche Markierung besagt, welche Art von Aktivität der Prozessor zu welchem Zeitpunkt ausführt: hellgrau stellt Berechnungen dar, dunkelgrau Kommunikation. Zur Kommunikationszeit wird aber auch die Wartezeit auf Nachrichten gerechnet, also ungenutzte Rechenzeit (*idle time*). Linien zwischen zwei Prozessorreihen symbolisieren jeweils Nachrichten, die zwischen verschiedenen Prozessoren verschickt werden.

A.1 Inhomogene Anwendung auf einem Metacomputer

Abbildung 38 zeigt einen Programmlauf von BD, bei dem zwei Iterationen ausgeführt wurden (vgl. Abschnitt 3.2). Während sich die ersten 15 Prozessoren bereits in der dun-

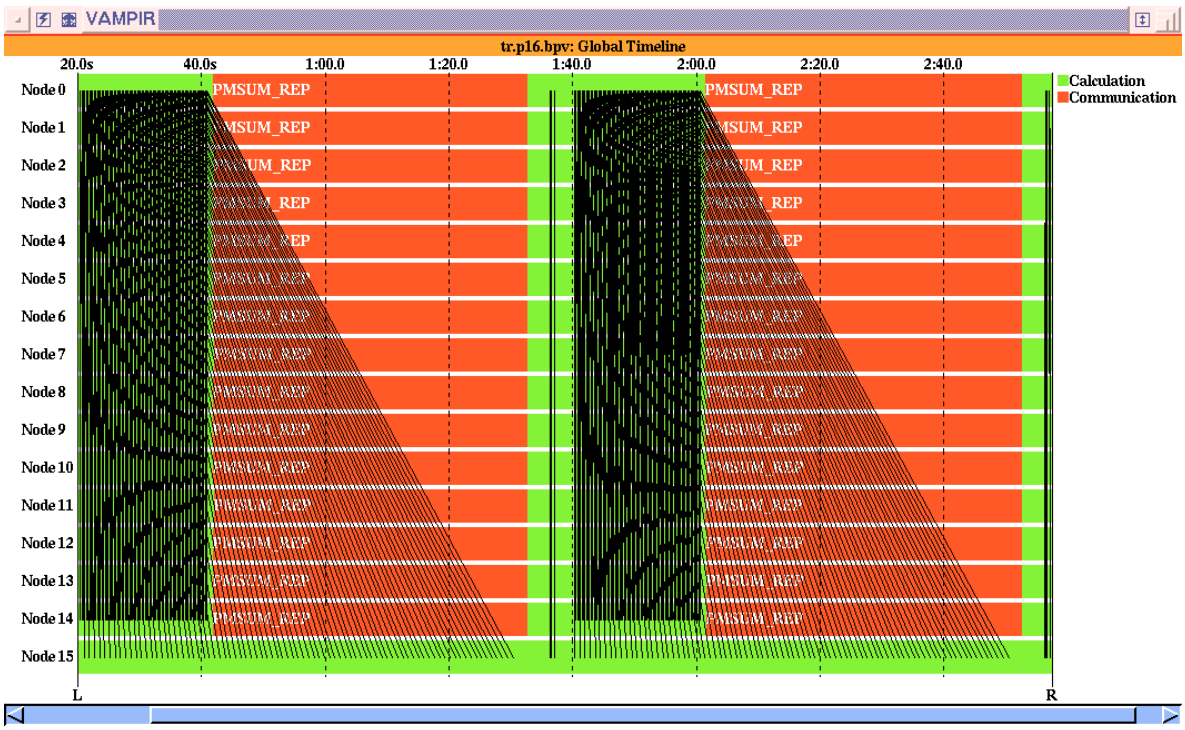

Abbildung 38: Heterogenes Verhalten der Anwendung BD.

kelgrau dargestellten, globalen Operation `PMSUM_REP` befinden, hat der letzte Knoten, der das Elektronenniveau mit einer größeren Genauigkeit bearbeitet, seine hellgrau markierten, lokalen Berechnungen noch nicht beendet, sondern muß noch ca. 50 Sekunden weiterrechnen. In dieser Zeit sind alle anderen Prozessoren untätig; ihre Rechenkapazität

ist verloren.

Abbildung 39 zeigt das Verhalten dieser Anwendung auf dem Metacomputer (vgl. Abschnitt 3.2), bei der das Elektronenniveau, das mehr Rechenzeit benötigt, auf einem anderen, schnelleren Computer berechnet wurde. Es ist sehr gut zu sehen, daß auf dem

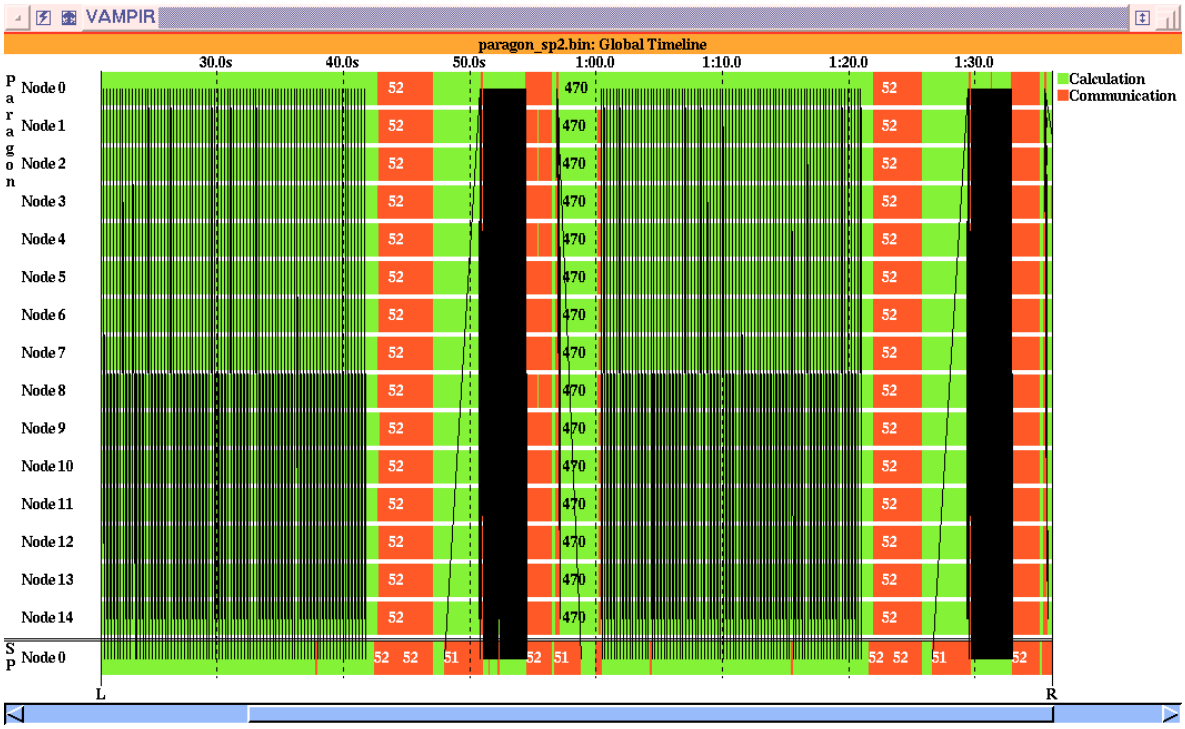

Abbildung 39: Anwendung BD auf dem Metacomputer.

Metacomputer viel weniger Zeit für die Kommunikation aufgewendet werden muß, da jetzt keine Zeit mehr dadurch verloren geht, daß auf den einzelnen, länger rechnenden Prozessor gewartet werden muß.

A.2 Homogene Anwendung auf einem Metacomputer

In Abschnitt 3.3 wird als zweite Fallstudie die Anwendung TRACE untersucht. Ein Ausschnitt aus der Initialisierungsphase von TRACE ist in Abbildung 40 dargestellt, in der Prozessor 0 die einzelnen Daten an alle anderen Prozessoren verteilt. Vor allem der deutliche Unterschied in der Nachrichtenlaufzeit zwischen internen und externen Nachrichten wird hierbei sichtbar. Dies zeigt sich auch in der maximalen Übertragungsrate, deren Wert in dem kleinen Fenster in Abbildung 40 zu sehen ist. Während intern eine maximale Bandbreite von 32 KByte/s erreicht wird, liegt die externe maximale Bandbreite zwischen 360 und 680 Bytes/s.

Bei der in Abschnitt 3.3 durchgenommenen Analyse von TRACE wurde die unterschiedliche Rechenleistung der SP2 und Paragon-Prozessoren nicht berücksichtigt. Dadurch kann

die Rechenleistung des schnelleren Rechners, also der SP2, nicht adäquat genutzt werden. Abbildung 41 verdeutlicht dieses Laufzeitverhalten anhand eines Ausschnitts aus einer

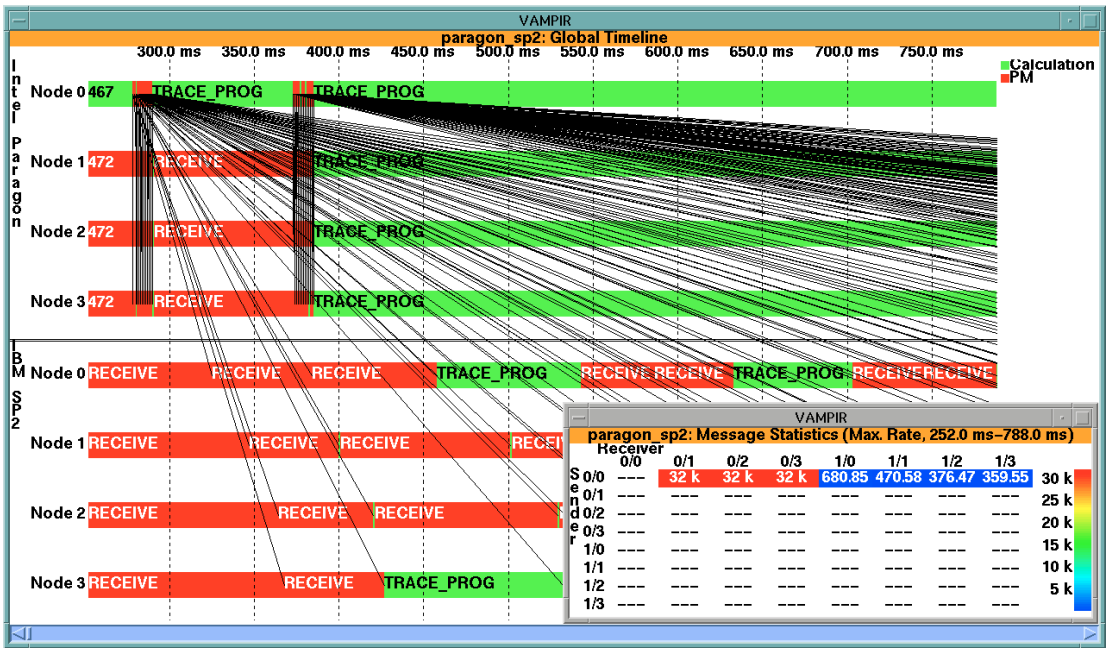

Abbildung 40: Unterschiedliche Nachrichtenlaufzeit bei der initialen Datenverteilung.

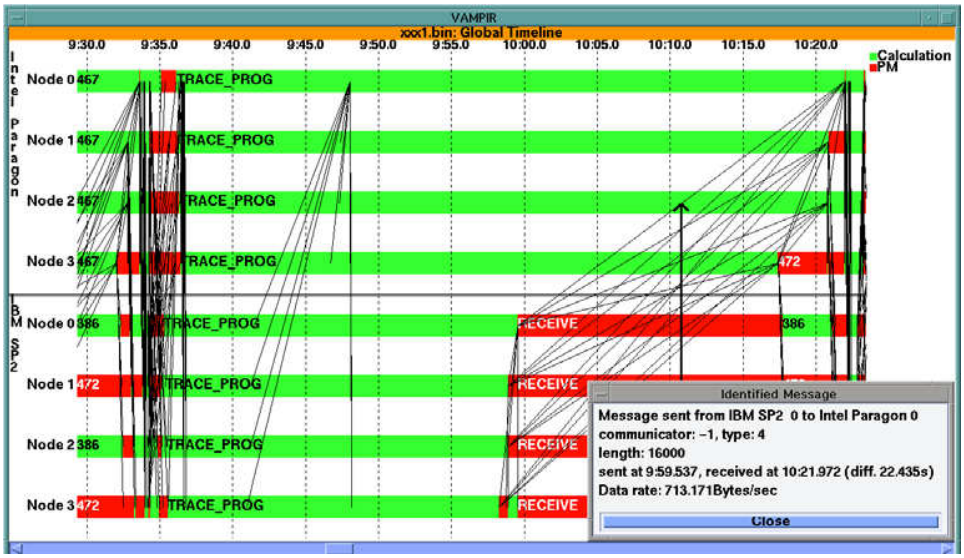

Abbildung 41: Die schematische Darstellung einer Iteration des Programms TRACE verdeutlicht die ungleiche Lastverteilung.

einzelnen Iteration. Die in der oberen Hälfte dargestellten Paragon Prozessoren rechnen noch, während die SP2 bereits mit der Kommunikation beginnt, dort aber auf die Paragon warten muß.

Literatur

[1] N.G. Aneroussis, A.A. Lazar, and D.E. Pendarakis. Taming Xunet III. *ACM Computer Communications Review*, 25(3):44–65, October 1995. ftp://ftp.ctr.columbia.edu/CTR-Research/comet/public/papers/95/ANE95b.ps.Z (5.6.1998).

[2] B. Basch, D. Becker, R. K. Singh, S. J. Bharrat, J. D. Loop, J. R. Symon, and D. Winkelstein. VISTAnet Deployment and System Integration Experiences. *IEEE Journal on Selected Areas in Communications*, 12(6):1097–1109, August 1994.

[3] P. Bastian. *Parallele adaptive Mehrgitterverfahren*. B. G. Teubner, Stuttgart, 1996.

[4] W. Becker. *Dynamische adaptive Lastbalancierung für große, heterogen konkurrierende Anwendungen*. PhD thesis, Institut für Parallele und Verteilte Höchstleistungsrechner (IPVR) der Universität Stuttgart, 1995.

[5] T. Beisel, E. Gabriel, and M. Resch. An Extension to MPI for Distributed Computing on MPPs. In Bubak et al. [12], pages 75–82.

[6] L. Bergman, P. Stolorz, R. Blom, D. Stanfill B. Kwan, B. Crippen, P. Lyster, P. Li, and D. Okaya. CALCRUST: Interactive Three-Dimensional Rendering of Multiple Earth-Science Datasets. In Messina and Mihaly-Pauna [92], pages 34–64.

[7] I.M. Boier, D.C. Marinescu, and J.R. Rice. Adaptive Load Balancing Strategies for Solving Irregular Problems on Distributed Memory MIMD Systems. In *Proc. 9th Int. Parallel Processing Symposium*. IEEE Press, 57-64, Los Alamitos, CA, April 1995. http://www.cs.purdue.edu/homes/boier/papers/ieee_ipps95.ps (8.5.1998).

[8] E. Brakkee, K. Wolf, D.P. Ho, and Toni Schüller. The COupling COmmunications LIBrary. In *Proc. Fifth Euromicro Workshop on Parallel and Distributed Processing, London, UK, January 22–24 1997*, pages 155–162. IEEE Computer Society Press, Los Alamitos, California, 1997.

[9] W.L. Briggs. *A Multigrid Tutorial*. Lancaster Press, Lancaster, Pennsylvania, April 1991.

[10] F. Brockners, C. Cseh, and M. Horneffer. IP über ATM Performance am Beispiel des RTB-NRW. *Offene Systeme*, 5(3), August 1996. http://www.uni-koeln.de/RRZK/uklan/rtb-nrw.html (8.6.1998).

[11] M. Brune, J. Gehring, and A. Reinefeld. A Lightweight Communication Interface for Parallel Programming Environments. *Lecture Notes in Computer Science*, 1225:503–513, 1997.

[12] M. Bubak, J. Dongarra, and J. Waśniewski, editors. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1332 of *Lecture Notes in Computer Science*. Springer-Verlag Berlin Heidelberg, 1997.

[13] R. Calkin, R. Hempel, H.C. Hoppe, and P. Wypior. Portable Programming with the PARMACS Message-Passing Library. *Parallel Computing*, 20(4):615–632, April 1994.

[14] CISPAR – Open Interface for Coupling of Industrial Simulation Codes on Parallel Computers. Technical report, ESPRIT Project 20161, 1996.
http://www.pallas.de/outpage/out-cis.htm (5.6.1998).

[15] D.D. Clark, B.S. Davie, D.J. Farber, I.S. Gopal, B.K Kadaba, W.D. Sincoskie, J.M. Smith, and D.L. Tennenhouse. An Overview of the AURORA Gigabit Testbed. Technical report, University of Pennsylvania, School of Engineering and Applied Science, Distributed Systems Laboratory, August 1992.
http://www.cis.upenn.edu/~dsl/read_reports/aurora-overview-20.ps.Z (8.6.1998).

[16] R. Clay and P. Steenkiste. Distributing a Chemical Process Optimization Application over a Gigabit Network. In *Supercomputing '95, ACM/IEEE, San Diego*, December 1995.

[17] Corporation for National Research Initiatives. *Gigabit Testbed Initiative Overview*, June 1990.
http://www0.cnri.reston.va.us/overview.html (4.6.1998).

[18] Cray Research, Inc. *Guide to Parallel Vector Applications*, sg-2182 2.0 Edition, November 1995.
http://www.kfa-juelich.de/zam/docs/crayman/postscript/2182-2.0.ps (18.5.1998).

[19] Cray Research, Inc. *Introducing the Cray TotalView Debugger*, in-2502 3.0 Edition, May 1997.
http://www.kfa-juelich.de/zam/docs/crayman/postscript/2502_3.0/ (18.5.1998).

[20] Cray Research, Inc. *Introducing the MPP Apprentice Tool*, in-2511 3.0 Edition, May 1997.
http://www.kfa-juelich.de/zam/docs/crayman/postscript/2511_3.0/ (18.5.1998).

[21] D.E. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proc. of Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, San Diego, CA*, May 1993.

[22] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, pages 46–55, January-March 1998.

[23] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel Programming Using Skeleton Functions. In *PARLE '93*, volume 694 of *Lecture Notes in Computer Science*, pages 146–160. Springer-Verlag Berlin Heidelberg, 1993.

[24] T. A. DeFanti and M. D. Brown. GII Testbed. In *Proc. Supercomputing '95 Conf.*, 1995.
http://www.supercomp.org/sc95/proceedings/GII_HPC/GII_AP.HTM (8.6.1998).

[25] T. A. DeFanti, I. Foster, M. E. Papka, R. Stevens, and T. Kuhfuss. Overview ofd the I-WAY: Wide Area Visual Supercomputing. *International Journal of Supercomputer Applications*, 1996.

[26] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable Parallel Computational Geometry for Coarse Grained Multicomputers. In *Proc. ACM Symposium on Computational Geometry*, pages 298–307, 1993.
http://www.scs.carleton.ca/~dehne/papers/papers/2-36.html (8.6.1998).

[27] F. Desprez, S. Domas, and B. Tourancheau. Optimization of an LU Factorization Routine Using Communication/Computation Overlap. Technical report, Unité de recherche INRIA Rhône-Alpes, February 1997.
ftp://ftp.inria.fr/INRIA/publication/RR/RR-3094.ps.gz (27.5.1998).

[28] Deutsches Zentrum für Luft- und Raumfahrt, e.V. (DLR). *Regionales Testbed Nordrhein-Westfalen (RTB-NRW)*, August 1994.
http://www.dlr.de/RTB-NRW/ (29.4.1998).

[29] *DFN-Mitteilungen*, volume 42. DFN-Verein, November 1996.

[30] E. D'Hollander, K. De Bosschere, and J. Van Campenhout, editors. *Parallel Computing: The State-of-the-art and Perspective: Proc. Int. Conf. ParCo95, Gent, Belgium, 19–22 September 1995*, volume 11 of *Advances in Parallel Computing*, Amsterdam, The Netherlands, 1996. North-Holland Publishing Co.

[31] distributed.net. *distributed.net - Project RC5*, April 1998.
http://www.distributed.net/rc5/ (8.5.1998).

[32] Dolphin Interconnect Solutions, Inc. *TotalView Features*, October 1997.
http://www.dolphinics.com/tw/tvfeatur.htm (28.4.1998).

[33] T. Eickermann. Gigabit Testbed West. Zwischenbericht an den DFN-Verein, February 1998.

[34] T. Eickermann, J. Henrichs, M. Resch, and R. Stoy. Metacomputing in Gigabit Environments: Networks, Tools and Applications. *Parallel Computing*, to appear.

[35] T. Eickermann, P. Wunderling, R. Niederberger, and R. Völpel. Aufbruch ins Jahr 2000. *DFN-Mitteilungen*, 45:13–15, November 1997.

[36] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A Worldwide Flock of Condors: Load Sharing Among Workstation Clusters. *Future Generations Computer Systems, Special Issue: Managment in Distributed Systems*, 12(1):53–66, May 1996.
http://pds.twi.tudelft.nl/ condor/reports/DUT-TWI-95-130.ps (5.6.1998).

[37] P. Eschholz, M. Koch, S. Preuß, and D. Tavangarian. Hypercomputing - erste Erfahrungen. In *1. Workshop Cluster-Computing, Chemnitz, 6.-7. November 1997*, 1997.

[38] M. M. Eshagian and Ying Chieh Wu. A Portable Programming Model for Network Heterogeneous Computing. In M. M. Eshagian, editor, *Heterogeneous Computing*, pages 155–195. Artech House, Inc., 1996.

[39] G. E. Fagg, J. J. Dongarra, and A. Geist. Heterogeneous MPI Application Interoperation and Process Management under PVMPI. In Bubak et al. [12], pages 91–98.

[40] R. G. Foldvik and O. A. McBryan. Experiences with ATM: The Batman Perspective. Technical report, University of Colorado, October 1994.
http://www.cs.colorado.edu/home/mcbryan/mypapers/icc95.ps (5.6.1998).

[41] Forschungszentrum Jülich, KFA-ZAM-BHB-0139. *The Cray Systems at Research Centre Jülich, Vol. 2: Programming Environment and Application Software*, 3rd Edition, January 1998.
http://www.kfa-juelich.de/zam/docs/bhb/bhb-cray.html (8.6.1998).

[42] Forschungszentrum Jülich, ZAM. *Verbundproject UNICORE (UNiform Interface to COmputing REsources)*, March 1998.
http://www.kfa-juelich.de/zam/RD/coop/unicore/unicore.html (29.4.1998).

[43] Forschungszentrums Jülich, ZAM. *ZAM Project RTB-NRW Metacomputer*, August 1994.
http://www.dlr.de/RTB-NRW/rtb-nrw_hlrz_d.html (29.4.1998).

[44] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Publising Company, Inc., 1995.
http://www.mcs.anl.gov/dbpp (19.5.1998).

[45] I. Foster, J. Geisler, and S. Tuecke. MPI on the I-WAY: A Wide-Area, Multimethod Implementation of the Message Passing Interface. In *Proc. MPI Developers Conf.*, University of Notre Dame, 1996.
ftp://ftp.mcs.anl.gov/pub/nexus/reports/mdc96_iway.ps.Z (18.6.1998).

[46] I. Foster and B. Toonen. Load Balancing Algorithms for Climate Models. In *Proc. 1994 Scalable High-Performance Computing Conf., IEEE, 1994*, pages 674–681, June 1994.
ftp://info.mcs.anl.gov/pub/tech_reports/reports/P439.ps.Z (4.6.1998).

[47] I. Foster and B. R. Toonen. Load-Balancing Algorithms for the Parallel Community Climate Model. Technical Report ANL/MCS-TM-190, Mathematics and Computer Science Division, Argonne National Laboratory,, January 1995.
ftp://info.mcs.anl.gov/pub/tech_reports/reports/TM190.ps.Z (4.6.1998).

[48] R. F. Freund and D. Sunny Conwell. Superconcurrency - A Form of Distributed Heterogeneous Supercomputing. *Supercomputing Review*, pages 47–49, October 1990.

[49] U. Fröhlings and J. Henrichs. Dynamic Load Balancing in a Time Warp Simulation Using Heterogeneous Workstation Cluster. In D'Hollander et al. [30], pages 447–454.

[50] H. Fukumori, Y. Kono, K. Nishimatsu, and Y. Muraoka. Finite Element Analysis with Heterogeneous Parallel Computer Environment over ATM Network. In *Proc. I-SPAN'96: Int. Symposium on Parallel Architectures, Algorithms, and Networks*, pages 124–130, 1996.

[51] H. Geiß, S. Jansen, and A. Oberreuter. Meteorologie online – Schnelle Datenanalyse im B-WiN. In DFN-Verein [29], pages 13–16.

[52] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, 1994.

[53] M. Gerndt. Sprachunterstützung zur Programmierung von Multiprozessorsystemen mit Shared Virtual Memory. Technical Report FZJ-ZAM-IB-9712, Forschungszentrum Jülich, ZAM, July 1997.

[54] A. Ghafoor and Jaehyung Yang. A Distributed Heterogeneous Supercomputing Management System. *IEEE Computer*, 26(6):78–86, June 1993.

[55] A. S. Grimshaw, J. B. Weissman, E. A. West, and Jr. E. C. Loyot. Metasystems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, 21:257–270, 1994.

[56] W. Gropp. Parallel Computing and Domain Decomposition. Technical Report MCS-P257-0891, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[57] W. Gropp and E. Lusk. Why are PVM and MPI so Different? In Bubak et al. [12], pages 3–10.

[58] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[59] D. Hänel, C. Thill, U. Uphoff, and R. Vilsmeier. Adaptive Grid Methods in Computational Fluid Dynamics. In Nagel [98], pages 53–67.

[60] U. Harms. Zentralorgan: Einheitliche Schnittstelle für Höchstleistungsrechner – UNICORE. *iX - Magazin für professionelle Informationstechnik*, 1:18, January 1998.

[61] J. Henrichs, W.E. Nagel, T. Eickermann, R. Niederberger, and R. Völpel. Schnelle Kopplung von Rechnern in einem ATM-basierten WAN – Erste Erfahrungen und Perspektiven. In *PARS-Workshop Rostock (Warnemünde)*, number 16 in Mitteilungen - Gesellschaft für Informatik e.V., Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware, pages 100–109, December 1997.

[62] J. Henrichs, W.E. Nagel, M. Weber, R. Völpel, and H. Grund. Höchstleistungsrechenzentrum: Verteilter massiv-paralleler Rechner. Technical Report FZJ-ZAM-IB-9711, Forschungszentrum Jülich, ZAM, July 1997.

[63] D.A. Hensgen, L. Moor, T. Kidd, R.F. Freund, E. Keeith, M. Kussow, J. Lima, and M. Campbell. Adding Rescheduling to and Integrating Condor with SmartNet. In *Proc. the Heterogeneous Computing Workshop*, pages 4–11. IEEE Computer Society Press, Los Alamitos, California, April 1995.

[64] J.M.D. Hill, P.I. Crumpton, and D.A. Burgess. The theory, Practice, and a Tool for BSP Performance Prediction Applied to a CFD Application. Technical Report PRG-TR-4-1996, Oxford University Computing Laboratory, Programming Research Group, 1996.
ftp://ftp.comlab.ox.ac.uk/pub/Documents/techreports/TR-4-96.ps.gz (8.6.1998).

[65] F. Hoßfeld. „Grand Challenges“ – wie weit tragen die Antworten des Supercomputing? In *Proc. des Arbeitsgesprächs „Physik und Informatik – Informatik und Physik“, München, 21./22.11.1991*, Informatik - Fachberichte Bd. 306, pages 241–251. Springer-Verlag, 1992.

[66] F. Hoßfeld. Partielle Differentialgleichungen: Die permanente Herausforderung. In Nagel [98], pages 1–9.

[67] F. Hoßfeld and W.E. Nagel. Per Aspera ad Astra: On the Way to Parallel Processing. In *Proc. Seminars Supercomputing '95 in Mannheim*. Saur Verlag, München, June 1995. (auch: interner Bericht des Forschungszentrums Jülich, FZJ-ZAM-IB-9507).

[68] Intel Corporation, Scalable Systems Division. *Paragon System Application Tools User's Guide*, May 1995.

[69] Intel Supercomputer Systems Division. *Paragon User's Guide*, no. 312489-002 Edition, 1993.

[70] International Business Machines Corporation. *IBM AIX Parallel Environment – Parallel Programming Reference*, first Edition, September 1993.

[71] J.A.Mathew and K.A.Hawick. Applications Characteristics and ATM Performance on LAN and WAN. Technical Report DHPC-016, University of Adelaide, August 1997.
http://www.dhpc.adelaide.edu.au/reports/016/abs-016.html (8.6.1998).

[72] P.A. Janson. *Operating Systems — Structures and Mechanisms.* Academic Press, Inc. London, 1985.

[73] W.S. John, S. Tenbrink, and D. DuBois. Gigabit Network Development and Deployment. In Messina and Mihaly-Pauna [92], pages 98–119.

[74] L.V. Kalé and S. Krishnan. CHARM++. In Wilson and Lu [143], pages 175–213.

[75] K. Kennedy, C.F. Bender, J.W.D. Connolly, J.L. Hennessy, M.K. Vernon, and L. Smarr. A Nationwide Parallel Computing Environment. *Communications of the ACM*, 40(11):63–72, November 1997.

[76] J. Kitowski, K. Boryczko, and J. Moóciński. Comparison of PVM and MPI Performance in Short-Range Molecular Dynamics Simulation. In Bubak et al. [12], pages 11–16.

[77] D.E. Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms.* Addison-Wesley Publishing Company, second Edition, 1973.

[78] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L.Steele Jr., and M.E. Zosel. *The High Performance Fortran Handbook.* Scientific and Engineering Computation Series. MIT Press, Cambridge, MA, 1994.

[79] A. Kolawa and K. Nagaya. The Express Program Development and Execution Environment. In Messina and Mihaly-Pauna [92], pages 90–97.

[80] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms.* The Benjamin/Cummings Publishing Company, Inc., California, 1994.

[81] O. Kyas. *ATM-Netzwerke – Aufbau, Funktion, Performance.* DATACOM Buchverlag GmbH, 2. Edition, 1995.

[82] Lawrence Livermore National Laboratory. *ASCI*, September 1996. http://www.llnl.gov/asci/ (9.6.1998).

[83] J.M. Lemme and J.R. Rice. Speedup in Parallel Algorithms for Adaptive Quadrature. *Journal of the ACM*, 26(1):65–71, January 1979.

[84] G.R. Luecke and J.J. Coyle. High Performance Fortran Versus Explicit Message Passing on the IBM SP-2 for the Parallel LU, QR, and Cholesky Factorizations. *Supercomputer 68*, XIII(2):4–14, 1997.

[85] O.A. McBryan. Limiting Factors in High Performance Computing. In J. Waśniewski, editor, *Proc. PARA94 Conf.*, volume 879 of *Lecture Notes in Computer Science.* Springer-Verlag Berlin Heidelberg, 1994.

[86] O.A. McBryan. HPCC: The Interrelationship of Computing and Communication. In D'Hollander et al. [30], pages 31–55.

[87] G. J. McRae. How Application Domains Define Requirements for the Grid. *Communications of the ACM*, 40(11):75–83, November 1997.

[88] C. R. Mechoso, J. Farrara, C. Ma, J. Spahr, B. Chinoy, G. Hanyzewski, R. Moore, C. Scarbnick, and H. Werner-Braun. Global Climate Modeling. In Messina and Mihaly-Pauna [92], pages 65–89.

[89] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* University of Tennessee, Knoxville, USA, 1995.
http://www.mcs.anl.gov/mpi/index.html (8.6.1998).

[90] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface.* University of Tennessee, Knoxville, USA, July 1997.
http://www.mcs.anl.gov/mpi/index.html (8.6.1998).

[91] P. Messina. Metacomputing and Data-Intensive Applications. Technical Report CACR-141, Center for Advanced Computing Research, California Institute of Technology, April 1997.

[92] P. Messina and T. Mihaly-Pauna, editors. *CASA Gigabit Network Testbed Final Report.* Center for Advanced Computing Research, California Institute of Technology. Technical Report CACR-123, July 1996.

[93] J. Mogul and S. Deering. *RFC 1191: Path MTU Discovery.* Network Working Group, November 1990.
ftp://sunsite.informatik.rwth-aachen.de/pub/mirror/nic.ddn.mil/rfc/rfc1191.txt.Z (18.5.1998).

[94] B. Monien, R. Diekmann, R. Feldmann, R. Klasing, R. Lüling, K. Menzel, T. Römke, and U. P. Schroeder. Efficient Use of Parallel & Distributed Systems: From Theory to Practice. *Lecture Notes in Computer Science*, 1000:62–77, 1995.

[95] B. Monien, R. Diekmann, and R. Preis. Lastverteilungsverfahren für Parallelrechner mit verteiltem Speicher. In Nagel [98], pages 205–225.

[96] D. J. Morton and J. M. Tyler. An Integrated Scheme for the Distribution of Adaptive Finite Element Code in the Cray Y-MP/T3D Computing Environment. Technical Report arsc-sp-117-95, Arctic Region Supercomputing Center, University of Alaska, Fairbanks, August 1995.
http://www.arsc.edu/ftp/publications/papers/arsc-sp-117-95.ps (8.6.1998).

[97] W. E. Nagel. Der neue massiv-parallele Rechner CRAY T3E im Frühling 1996 – Erfahrung mit der Jungfräulichkeit –. In H. W. Meurer, editor, *Supercomputer 1996, Anwendungen, Architekturen, Trends*, pages 92–107. K. G. Saur, 1996.

[98] W. E. Nagel, editor. *Partielle Diffentialgleichungen, Numerik und Anwendungen*, number 18 in Konferenzen des Forschungszentrums Jülich. Forschungszentrum Jülich GmbH, Zentralbibliothek, 1996.

[99] W.E. Nagel and A. Arnold. Performance-Optimierung von parallelen Programmen mit VAMPIR – Sehen und Verstehen –. *GI/ITG MMB-Mitteilungen*, 29:21–30, 1996.

[100] W.E. Nagel, A. Arnold, M. Weber, H.C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer 63*, XII(1):69–80, 1996.

[101] National Science Foundation Engineering Research Center for Computational Field Simulation, Mississippi State University. *Freely Available MPI Implementations*, March 1998.
http://WWW.ERC.MsState.Edu/mpi/implementations.html (15.5.1998).

[102] R. Niederberger. Schnelle Netze für das Metacomputing. Anspruch und Wirklichkeit im RTB-NRW. Technical Report FZJ-ZAM-IB-9625, Forschungszentrum Jülich, ZAM, 1996.

[103] W. Oed. *Das Parallel-Vektor-Prozessorsystem CRAY T90.* Cray Research, München, February 1995.
http://www.kfa-juelich.de/zam/docs/craydoc/t90/t90.ps (10.6.1998).

[104] W. Oed. *Massiv-paralleles Prozessorsystem CRAY T3E.* Cray Research, München, March 1996.
http://www.kfa-juelich.de/zam/docs/craydoc/t3e/t3e.ps (14.5.1998).

[105] J.P. Ostriker and M.L. Norman. Cosmology of the Early Universe Viewed Through the New Infrastructure. *Communications of the ACM*, 40(11):85–94, November 1997.

[106] V. Penner. *Parallelität und Transputer: Von den Grundlagen zur Anwendung: Occam und Transputer, Concurrent Prolog, Linda.* Vieweg Verlag, Braunschweig, Wiesbaden, 1992.

[107] C. Perez. Load Balancing HPF programs by migrating virtual processors. Technical Report RR-3037, Unité de recherche INRIA Rhône-Alpes, November 1996.

[108] M. Pourzandi and B. Tourancheau. A Parallel Performance Study of Jacobi-like Eigenvalue Solution. Technical Report CS-94-226, University of Tennessee, March 1994.
http://netlib2.cs.utk.edu/tennessee/ut-cs-94-226.ps (4.6.1998).

[109] R. Rabenseifner. The DFN Remote Procedure Call Tool for Parallel and Distributed Applications. In K. Franke, U. Huebner, and W. Kalfa, editors, *'Kommunikation in Verteilten Systemen - KiVS 95' Proceedings, Chemnitz-Zwickau*, pages 415–419, February 1995.

[110] R. Rabenseifner, J. Bönisch, C. Dodge, B. Hesse, M. Karimian, J. Kayser, W. Kollak, and H.D. Reimann. *The DFN Remote Procedure Call Tool, Reference Manual, Vol.*

I + II, Release 1.0.60. Rechenzentrum der Universität Stuttgart, December 1994. ftp://ftp.uni-stuttgart.de/pub/rus/dfn_rpc/dfnrpc_e_1.0.60beta67.ps.Z (7.5.1998).

[111] H. Rafii-Tabar. Visualization Reveals Model Defects. *Scientific Computing World*, (35):32–34, February 1998.

[112] F. Ramme and K. Kremer. Scheduling a Metacomputer by an Implicit Voting System. In *Proc. 3rd IEEE Int. Symp. on High-Performance Distributed Computing*, pages 106–113, 1994.

[113] R. Rankin and D. Horne et al. European Meta Computing Utilising Integrated Broadband Communications, Final Report. Technical report, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, 1997. http://www.informatik.uni-stuttgart.de/ipvr/as/projekte/emc2/emc2.html (8.6.1998).

[114] D. A. Reed, R. C. Giles, and C. E. Catlett. Distributed Data and Immersive Collaboration. *Communications of the ACM*, 40(11):39–48, November 1997.

[115] M. Resch, T. Beisel, T. Boenisch, B. Loftis, and R. Reddy. Performance Issues of Intercontinental Computing. *Cray User Group Conf.*, 1997.

[116] M. Resch, H. Berger, and T. Boenisch. A Comparison of MPI Performance on Different MPPS. In Bubak et al. [12], pages 25–32.

[117] C. E. Rhoades, editor. *Future Generations Computer Systems, Special Double Issue: Grand Challenges to Computational Science*, volume 5(2&3), 1989.

[118] V. Sander. Eine Metacomputing-Architektur auf der Basis einer kooperativen Ressourcenverwaltung. Technical Report Jül–3304, Forschungszentrums Jülich, ZAM, November 1996.

[119] Sandia National Laboratories. *ASCI Red UltraComputer*, August 1997. http://www.sandia.gov/ASCI/Red/ (9.6.1998).

[120] A. Schill. Remote Procedure Call: Fortgeschrittene Konzepte und Systeme – ein Überblick, Teil 1. *Informatik Spektrum*, 15(2), April 1992.

[121] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers About BSP. Technical Report PRG-TR-15-96, Oxford University Computing Laboratory, Programming Research Group, November 1996. ftp://ftp.comlab.ox.ac.uk/pub/Documents/techreports/TR-15-96.ps.Z (4.6.1998).

[122] A. Skjellum and Z. Lu. MPI++. In Wilson and Lu [143], pages 465–506.

[123] L. Smarr and C.E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–53, June 1992.

[124] R. M. Stallman and Cygnus Support. Debugging with GDB. Distributed with GDB, January 1994.

[125] P. Steenkiste. A High-Speed Network Interface for Distributed-Memory Systems: Architecture and Applications. *ACM Transactions on Computer Systems*, 15(1):75–109, February 1997.

[126] T. Sterling, T. Cwik, D. Becker, J. Salmon, M. Warren, and B. Nitzberg. An assessment of Beowulf-class computing for NASA requirements: Initial findings from the first NASA workshop on Beowulf-class clustered computing. In *Proc. IEEE Aerospace Conference, Aspen CO, March 21-28*, 1998. http://loki-www.lanl.gov/papers/ieee_aero98/p312.ps.

[127] R. Stevens, P. Woodward, T. DeFanti, and C. Catlett. From the I-WAY to the National Technology Grid. *Communications of the ACM*, 40(11):51–60, November 1997.

[128] W.R. Stevens. *UNIX Network Programming*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, first Edition, 1990.

[129] G. Stix. Gigabit Connection. *Scientific American*, pages 118–120, October 1990.

[130] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1981.

[131] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. Studienbücher der Informatik. Carl Hanser Verlag München Wien, 1995.

[132] DHPC Project Team. Distributed Geographic Information Systems Project Concepts Discussion Document. Technical Report DHPC-001, University of Adelaide, November 1996. http://www.dhpc.adelaide.edu.au/reports/001/abs-001.html (8.6.1998).

[133] C.A. Thole. Programmiersprachen für Höchstleistungsrechner: MPI and HPF. In Nagel [98], pages 69–81.

[134] U. Trottenberg and K. Oosterlee. Parallel Adaptive Multigrid – an Elementary Introduction. In Nagel [98], pages 159–193.

[135] F. Hasenbrink und S. Pokorny. Das Virtuelle Triebwerk – Verteilte Simulation und Echtzeit-Darstellung der Strömung durch Flugzeugtriebwerke. In DFN-Verein [29], pages 10–12.

[136] R. v. Hanxleden and L. Ridgway Scott. Load Balancing on Message Passing Architectures. *Journal of Parallel and Distributed Computing*, 13:312–324, 1991.

[137] L.G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[138] H. Vereecken, G. Lindenmayr, O. Neuendorf, U. Döring, and R. Seidemann. TRACE: A Mathematical Model for Reactive Transport in 3D Variably Saturated Porous Media. Technical Report 501494, Forschungszentrum Jülich, ICG-4, 1994.

[139] E.F. Walker, R. Floyd, and P. Neves. Asynchronous Remote Operation Execution in Distributed Systems. In *10th Int. Conf. on Distributed Computing Systems, Paris*, pages 253–259, 1989.

[140] C. Walshaw and M. Berzins. Dynamic Load Balancing for PDE Solvers on Adaptvie Uunstructured Meshes. Computer Studies Tech. Rep. 92.32, School of Computer Studies, University of Leeds, December 1992.
ftp://agora.leeds.ac.uk/scs/doc/reports/1993/92_32.ps.Z (8.6.1998).

[141] A. Watermann. *Parallelisierung des Finite-Element-Codes SMART-Konvektion.* PhD thesis, Forschungszentrum Jülich, Jül-3122, September 1995.

[142] M. Weber, J. Henrichs, W.E. Nagel, R. Niederberger, R. Völpel, and H. Grund. Schneller, höher, weiter – Breitbandnetze als Weg ins Metacomputing. In DFN-Verein [29], pages 7–9.

[143] G.V. Wilson and P. Lu, editors. *Parallel Programming Using C++.* MIT Press, Cambridge, MA, 1996.

[144] R. Wimmershoff. Entwicklung und Implementierung einer dreidimensionalen Partitionierungsstrategie für das Programm TRACE auf einem massiv-parallelen Rechner. Technical Report Jül-3157, Forschungszentrum Jülich, ZAM, 1995.

[145] M. Wu, A. Kuppermann, and P. Messina. Quantum Chemical Reaction Dynamics. In Messina and Mihaly-Pauna [92], pages 9–33.

[146] R. Zeller. Green-Function Method for Electronic Structure of Periodic Crystals. *International Journal of Modern Physics C*, 4(6):51–58, 1993.