Eine Systemumgebung zur Erstellung paralleler C++ Programme und deren Ausführung in heterogenen verteilten Systemen

Dissertation
der Fakultät für Informatik
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Wolfgang Blochinger
aus Reutlingen

Tübingen
2002

Tag der mündlichen Qualifikation: 22.05.2002
Dekan: Prof. Dr. A. Zell
1. Berichterstatter: Prof. Dr. W. Küchlin
2. Berichterstatter: Prof. Dr. M. Kaufmann

Diese Arbeit ist meinen Eltern gewidmet.

Vorwort

Diese Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter im Arbeitsbereich Symbolisches Rechnen des Wilhelm-Schickard-Instituts für Informatik an der Universität Tübingen.

Bei der Durchführung standen mir viele Personen hilfreich zur Seite, bei denen ich mich an dieser Stelle bedanken möchte.

Mein besonderer Dank gilt Herrn Prof. Dr. W. Küchlin für die konstruktive Betreuung der Arbeit und seine stets großzügige Unterstützung. Bei Herrn Prof. Dr. M. Kaufmann möchte ich mich bedanken, dass er sich als Zweitgutachter zur Verfügung gestellt hat.

Bei Herrn Prof. Dr. A. Weber bedanke ich mich herzlich für die Unterstützung und die vielen hilfreichen Ratschläge während der Anfangszeit meiner Arbeit.

Ich möchte auch meinen Kollegen am Wilhelm-Schickard-Institut meinen Dank aussprechen, die mir in einer Vielzahl von Gesprächen und Diskussionen hilfreich zur Seite standen.

Wolfgang Blochinger

Inhaltsverzeichnis

1

Kapitel 1

Einführung

1.1 Die Verwendung von Parallelität

Das Konzept der Parallelität tritt bei der Realisierung von Rechnersystemen auf unterschiedlichen Ebenen implizit oder auch explizit auf.

Auf der niedrigsten Ebene wird Parallelität zur Verarbeitung von Daten verwendet. Daten werden von Rechnern nicht in ihrer kleinsten Darstellungseinheit, dem Bit, sondern in Datenworten zusammengefasst parallel verarbeitet. Man spricht hier von der *Bitebenenparallelität*.

Bei der *Befehlsebenenparallelität (Instruction Level Parallelism, ILP)* [41] werden die einzelnen Phasen der Abarbeitung eines Maschinenbefehls (Befehl laden, decodieren, ausführen, auf Speicher zugreifen und Ergebnis schreiben) überlappt ausgeführt, so dass sich zu jedem Zeitpunkt mehrere Befehle parallel in Ausführung befinden. Die Ausführungseinheit ist demnach als so genannte *Pipeline* organisiert. In modernen *superskalaren Architekturen* werden mehrere derartiger Pipelines parallel betrieben, um einen noch höheren Grad an Befehlsebenenparallelität zu erreichen.

Auf der nächsthöheren Ebene werden mehrere Kontrollflüsse *(Threads)* innerhalb eines Programms nebenläufig ausgeführt. Moderne Betriebssysteme unterstützen diese Art der Programmierung und können die einzelnen Threads eines Programms unterschiedlichen Prozessoren zuweisen, so dass diese parallel ausgeführt werden. Man spricht hier von *interner Programmparallelität*. Alternativ, bzw. ergänzend kann diese Art der Programmierung auch durch spezielle Prozessorarchitekturen *(Multithreaded Architecture, MTA)* unterstützt werden. Derartige Prozessoren verfügen über mehrere Registersätze und ein komplexes Steuerwerk zur gleichzeitigen Verwaltung mehrerer Kontrollflüsse.

Die Parallelität auf der höchsten Ebene wird als *externe Programmparallelität* bezeichnet. Hier werden mehrere Anwendungen oder Prozesse parallel ausgeführt. Die Verwendung dieser Art der Parallelität erfordert das Vorhandensein von mehreren Prozessoren in einem Rechnersystem.

Während die Ausnutzung von Parallelität auf der Hardwareebene seit langem als Standardtechnik gilt, ist dies bei der Verwendung von Parallelität im Bereich der Software nicht der Fall. Zum Beispiel unterstützen die zurzeit gebräuchlichen Programmiersprachen die Formulierung von parallelen Vorgängen nur in rudimentärer Form. Als Gründe für die Schwierigkeit des Einsatzes von Parallelität auf der Softwareebene lassen sich die folgenden Punkte anführen:

- Das menschliche Gehirn ist an sequentiellen Vorgehensweisen orientiert. Die Formulierung von parallelen Abläufen in einem Programm ist somit für viele Programmierer ungewohnt und schwierig. Zudem macht die komplexe Zustandsmenge eines parallelen Programms die Fehlersuche sehr viel aufwändiger, als dies bei herkömmlichen sequentiellen Programmen der Fall ist.
- Die Entwicklung theoretischer Grundlagen für parallele Algorithmen und für die parallele Programmierung ist noch nicht weit fortgeschritten und wurde erst nach den technologischen Realisierungen begonnen.
- Die Hardwarearchitekturen von Parallelrechnern ändern sich sehr schnell. (Vgl. die Ausführungen in Abschnitt 1.3). Dies macht oftmals eine vollständige Neuerstellung der Software erforderlich. Der Einsatz von paralleler Software ist somit im kommerziellen Bereich nicht lukrativ, da die Wartungskosten zu hoch sind. Zudem wurden Parallelrechner in der Vergangenheit von deren Herstellern vor allem für den Einsatz im wissenschaftlich-technischen Bereich positioniert.

Trotz der oben aufgezeigten Schwierigkeiten ist die Verwendung von Parallelität jedoch von grundsätzlicher Signifikanz im Bereich der Programmierung. Diese Aussage wird durch die folgenden Beobachtungen untermauert [79]:

- Die meisten Objekte und Vorgänge in der realen Welt weisen eine inhärente Parallelität auf; sie setzen sich gewöhnlich aus einer Vielzahl von parallelen Vorgängen und Abläufen zusammen. Solche Vorgänge können in einer einfachen und eleganten Weise mittels paralleler Programmierung formalisiert werden. Sequentielle Programme führen hingegen häufig eine artifizielle Ordnung auf Vorgängen ein, die eigentlich parallel ablaufen könnten.

- Durch die Parallelisierung eines Problems und die Ausführung des resultierenden Programms auf einem Parallelrechner lässt sich seine Bearbeitungsdauer oftmals wesentlich reduzieren.
- Eine parallele Berechnung kann bei gleicher Ausführungszeit oftmals kostengünstiger ausgeführt werden, als dies bei der Ausführung der Berechnung auf einem sequentiellen Rechner möglich ist. Die Preise für Prozessoren der aktuellsten Generation sind meistens unverhältnismäßig hoch, so dass ein Parallelrechner, der aus Prozessoren einer vorhergehenden Prozessorgeneration aufgebaut ist, bei gleicher Leistung günstiger sein kann, als ein sequentieller Rechner mit aktuellem Prozessor.

Für die vorliegende Arbeit sind alle drei Aspekte relevant, sie befasst sich jedoch hauptsächlich mit dem Einsatz von Parallelität zur Beschleunigung von Berechnungen, dem so genannten *parallelen Rechnen*.

1.2 Wichtige Begriffe des parallelen Rechnens

Bevor in den nächsten Abschnitten die Entwicklung der Parallelrechner im letzten Jahrzehnt und aktuelle Zukunftstrends untersucht werden, sollen an dieser Stelle einige grundlegende Begriffe des parallelen Rechnens eingeführt werden, die im Nachfolgenden häufig vorkommen.

1.2.1 Parallele Hardware

Zur Klassifikation von Parallelrechnern hat sich das von *Flynn* vorgeschlagene Klassifikationsschema [27] durchgesetzt. Die Einordnung eines Parallelrechners basiert auf der Untersuchung der beiden folgenden Aspekte:

1. Wieviele Befehle werden gleichzeitig bearbeitet?
2. Auf wievielen Datenworten wird ein Befehl gleichzeitig angewendet?

Hieraus ergeben sich die folgenden 4 Klassen von Parallelrechnern:

- **SISD**: Single Instruction Stream – Single Data Stream
  (von Neumann Architektur)
- **SIMD**: Single Instruction Stream – Multiple Data Streams
  (Vektorrechner, Feldrechner)

- **MISD**: Multiple Instruction Streams – Single Data Stream
  (kein Vertreter bekannt)
- **MIMD**: Multiple Instruction Streams – Multiple Data Streams
  (Multiprozessorsysteme, Mehrrechnersysteme)

MIMD Parallelrechner bestehen aus mehreren Prozessoren, die unabhängig voneinander arbeiten. Um ein Problem parallel zu lösen, müssen die einzelnen Prozessoren miteinander kommunizieren können. Die Art und Weise, wie die Kommunikation zwischen den Prozessoren realisiert ist, dient als Kriterium zur weiteren Einteilung von MIMD Systemen. Man unterscheidet hierbei zwei Hauptklassen:

- **Systeme mit gemeinsamem Speicher** *(shared-memory)*
  Diese Parallelrechner erlauben den Austausch von Daten zwischen den Prozessoren über einen gemeinsamen, physisch realisierten Adressraum. Man spricht hier auch von *symmetrischen Multiprozessoren, SMP*.
- **Systeme mit verteiltem Speicher** *(distributed-memory)*
  Bei diesen Parallelrechnersystemen kann jeder Prozessor nur auf seinen privaten Adressraum zugreifen. Die Kommunikation zwischen den Prozessoren erfolgt über spezielle Verbindungsnetzwerke. Man bezeichnet ein Prozessor-Speicher Paar in diesen Systemen üblicherweise als *Knoten*.

Es können auch *hierarchische Multiprozessorsysteme* realisiert werden. Dabei handelt es sich um Parallelrechner mit verteiltem Speicher, deren Knoten aber aus Systemen mit gemeinsamem Speicher bestehen.

Mit dem Begriff *Cluster* werden verteilte Systeme bezeichnet, die aus *gleichartigen*, enger gekoppelten Rechnersystemen aufgebaut sind, wobei die verwendeten Rechnersysteme einen autarken Charakter haben, d.h. sie sind grundsätzlich auch außerhalb des Clusters funktionsfähig.

Ein *Rechnernetz (Network of Workstations, NOW)* ist ein Parallelrechner mit verteiltem Speicher, der aus unterschiedlichen Rechnersystemen aufgebaut ist, die ähnlich wie bei einem Cluster autark sind. Oftmals werden diese neben der Verwendung als Baustein eines Parallelrechners noch als Arbeitsplatzrechner eingesetzt.

Für die Leistungsbewertung von Verbindungsnetzwerken und -protokollen bei Systemen mit verteiltem Speicher sind die beiden folgenden Begriffe relevant:

- Bandbreite, Durchsatz
  Bandbreite bezeichnet die maximale Übertragungskapazität, Durchsatz ist

der tatsächliche erreichte Wert für eine bestimmte Anwendung. Die Einheit beider Größen ist Bits pro Sekunde (bps).

- Latenzzeit
  Als Latenzzeit wird diejenige Zeit bezeichnet, die die Übertragung einer leeren Nachricht (d.h. einer Nachricht ohne Nutzinformation) vom Sender bis zum Empfänger benötigt.

1.2.2 Parallele Programmierung

In Kapitel 2 wird eine ausführliche Darstellung von Programmiermodellen für das parallele Rechnen gegeben. An dieser Stelle sollen ein paar allgemeine Begriffe der parallelen Programmierung eingeführt werden.

Ein paralleles Programm für ein System mit verteiltem Speicher kann grundsätzlich auf zwei Arten realisiert sein. Die einzelnen Prozessoren können verschiedene (Teil-) Programme ausführen, oder jeder Prozessor führt dasselbe (Teil-) Programm aus. Im ersten Fall spricht man von einem MPMD *(Multiple Program Multiple Data)* Ansatz, während die zweite Vorgehensweise als SPMD *(Single Program Multiple Data)* Ansatz bezeichnet wird.

Durch ein paralleles Programm wird die Zerlegung eines Gesamtproblems in einzelne, parallel zu verarbeitende Teilprobleme vorgenommen. Als *Granularität* eines parallelen Programms wird das Verhältnis von Berechnungszeit und Kommunikationszeit, das sich bei der Bearbeitung der Teilprobleme ergibt, bezeichnet. Bei *fein-granularen* Berechnungen stellt die Kommunikationszeit einen wesentlichen Faktor dar, bei *grob-granularen* Berechnungen ist die Kommunikationszeit hingegen vernachlässigbar.

1.2.3 Leistungsbewertung paralleler Programme

Der Begriff *Beschleunigung (speedup)* erfasst den Zeitgewinn eines Programms, der durch die Ausführung auf einem Parallelrechner erreicht wird.

Sei t_p die Ausführungszeit eines parallelen Programms zur Lösung eines Problems unter Verwendung von p Prozessoren und t_s die Laufzeit des schnellsten sequentiellen Programms zur Lösung des Problems auf einem Prozessor. Die Zeiten stellen jeweils die tatsächlichen Ausführungszeiten für das zu untersuchende Programm dar. Im Englischen ist hierfür auch der Begriff *wall clock time* gebräuchlich.

Die Beschleunigung s_p ist definiert durch:

$$s_p = \frac{t_s}{t_p}$$

Der Begriff der *Effizienz* gibt die tatsächliche genutzte Prozessorleistung eines parallelen Programms wieder. Die Effizienz e_p eines parallelen Programms, bei Verwendung von p Prozessoren, ist definiert durch:

$$e_p = \frac{s_p}{p}$$

Ein *skalierbares* paralleles Programm liegt vor, wenn die Effizienz mit der Verwendung einer wachsenden Anzahl von Prozessoren konstant gehalten werden kann.

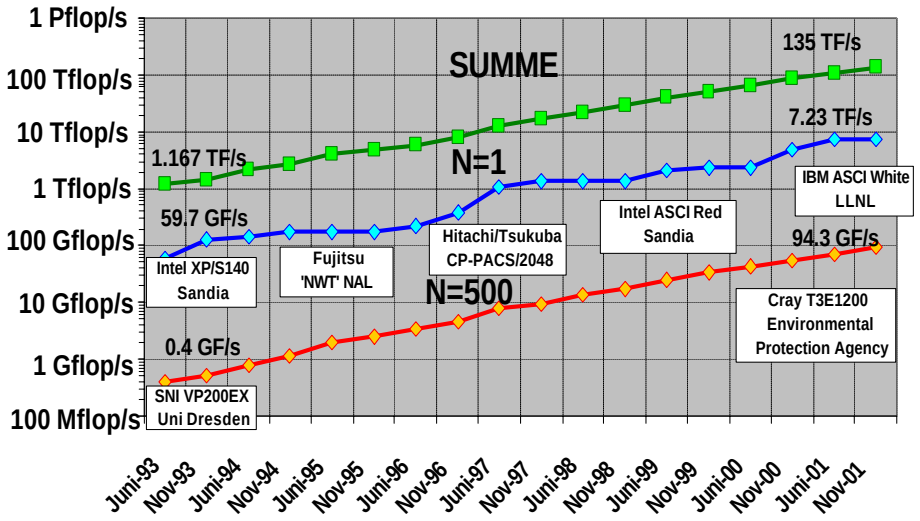
1.3 Entwicklungen im letzten Jahrzehnt

In diesem Abschnitt sollen die Entwicklungen im Bereich der parallelen Hardware im letzten Jahrzehnt anhand der Liste der 500 leistungsstärksten Rechner TOP500 [89] studiert werden. Diese Liste wird jährlich zweimal aktualisiert und basiert auf Leistungsmessungen, die mit dem *Linpack Benchmark* durchgeführt werden. Der Linpack Benchmark besteht im Wesentlichen aus der Berechnung der Lösung eines linearen Gleichungssystems. Das betrachtete Problem weist einen stark regulären Charakter auf, so dass die durch den Linpack Benchmark gelieferte Leistungsbewertung eine gute Approximation für die maximale Leistung eines Systems darstellt. Die Leistung eines Rechners wird in diesem Zusammenhang durch die Anzahl der Fließkommaoperationen pro Sekunde (flop/s), die bei der Ausführung des Benchmark Problems erreicht wurde, quantifiziert. Bis auf einige wenige Ausnahmen in den ersten Jahren der Auflistung sind unter den 500 schnellsten Rechnern ausschließlich Parallelrechner zu finden, so dass durch die Auswertung der TOP500 Liste eine zutreffende Auskunft über den jeweils aktuellen Stand der Technik im Bereich der parallelen Hardware gewonnen werden kann.

Der erste Aspekt, der genauer untersucht werden soll, ist die Entwicklung der Leistungsfähigkeit von Parallelrechnern während der letzten Dekade. In Abbildung 1.1 sind dazu die Leistungswerte für jeweils den schnellsten (N=1) und den langsamsten Rechner (N=500) einer Ausgabe der TOP500 Liste dargestellt. Zusätzlich zeigt die Abbildung die Summe der Leistungsdaten aller 500 Rechner der Liste. Es ist deutlich zu erkennen, dass sich die Leistungsfähigkeit der parallelen Hardware während der letzten 10 Jahre in einem rasanten Tempo entwickelt hat. Die Leistung des jeweils schnellsten Rechners ist innerhalb von 8 Jahren um mehr als einen Faktor von 120 gestiegen.

Ein weiterer wichtiger Trend der letzten Dekade, der durch die Auswertung der TOP500 Liste deutlich wird, ist die Veränderung der Architektur der schnellsten

Abbildung 1.1 Entwicklung der Leistungsfähigkeit von Parallelrechnern [89]

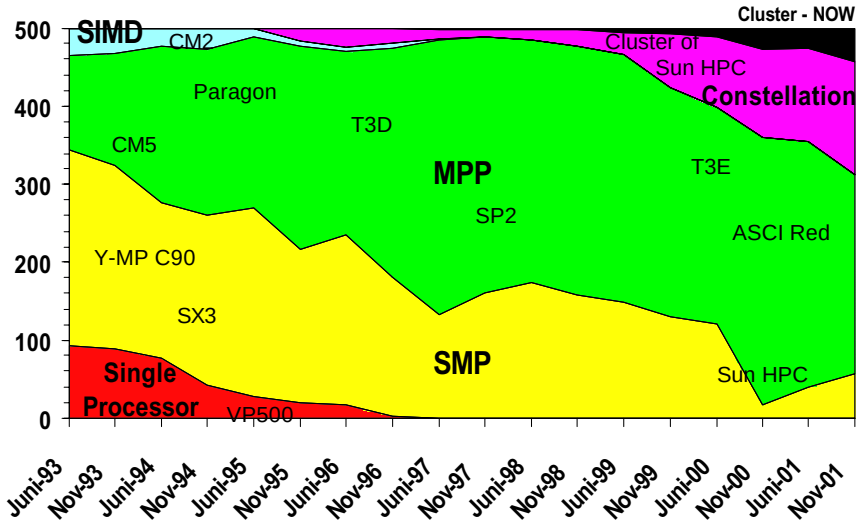


Parallelrechner. Abbildung 1.2 gibt einen Überblick über die in der Liste vorkommenden Architekturen. Auf numerische Berechnungen spezialisierte Vektorrechner (also Rechner mit SIMD Architektur) und auch reine SMP Parallelrechner verlieren an Bedeutung. Durch die spezielle Auslegung von Vektorrechnern auf reguläre numerische Berechnungen ist deren Einsatzbereich zu sehr eingeschränkt. Die SMP Architekturen geraten aufgrund ihrer mangelnden Skalierbarkeit ins Hintertreffen.

Diese beiden Architekturen werden in den letzten Jahren durch Rechner mit verteiltem Hauptspeicher verdrängt. Innerhalb dieser Klasse von Parallelrechnern ist ein beginnender Trend auszumachen, der weg von speziell gefertigten Hochleistungsparallelrechnern *(Massively Parallel Processing, MPP)* hin zu aus Standardrechnern aufgebauten Clustern oder nicht-dezidierten Rechnernetzwerken (Cluster-NOW) führt.

Bei der Chip Technologie, die für den Aufbau von Parallelrechnern verwendet wird, werden speziell gefertigte Prozessortypen von in Massenproduktion hergestellten und somit kostengünstigeren Standardprozessoren verdrängt. Diese werden im Englischen auch als *Commercial off-the-shelf (COTS)* Komponenten bezeichnet. Abbildung 1.3 macht diesen Trend deutlich.

Abbildung 1.2 Entwicklung der Architektur von Parallelrechnern [89]



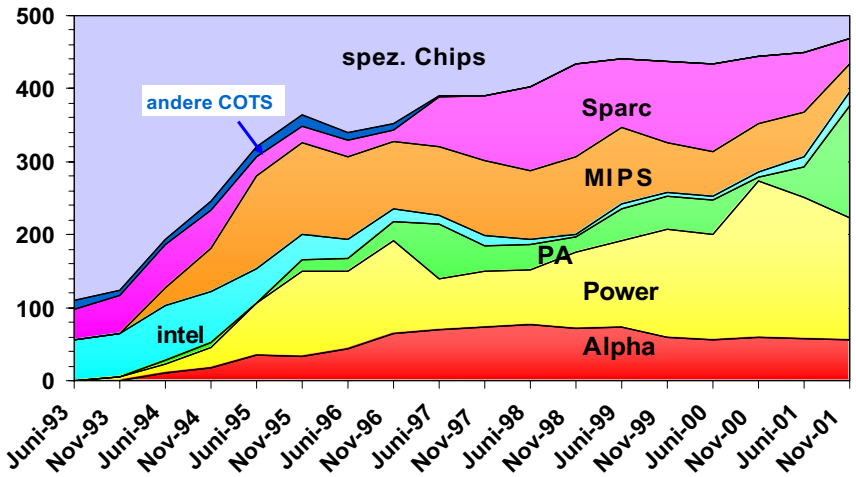
1.4 Zukunftstrend: Grid Computing

Der Begriff des *Grid Computing* [29] entstand durch die Assoziation mit dem elektrischen Stromnetz (Power Grid). Die ursprüngliche Idee von Grid Computing war es, dass Rechenleistung genauso universell verfügbar gemacht wird, wie dies bei elektrischer Energie der Fall ist. Für den Benutzer ist also vollkommen transparent, wo seine Berechnungen ausgeführt werden.

Neuerdings wird der Begriff des Grid Computing etwas umfassender definiert, als die flexible, sichere, koordinierte, gemeinsame Nutzung von Ressourcen [30]. Als Ressourcen sind in diesem Zusammenhang neben Rechenkapazität (“CPU Zyklen”) auch Daten und Geräte (z.B. Teilchenbeschleuniger oder Radioteleskope) zu verstehen. Zur Realisierung eines Grids ist daher eine Kommunikationsstruktur zu schaffen, in der transparent und einheitlich auf verschiedenste Ressourcen zugegriffen werden kann, ohne dass für jede Anwendung die zugehörigen Protokolle neu erfunden werden müssen.

Das Globus Toolkit [35] stellt den de-facto Standard zur Realisierung von Grid Infrastrukturen dar. Es ist nach dem so genannten *Sanduhr-Prinzip* schichtartig organisiert. Die mittlere Schicht *(Resource+Connectivity)* bildet ein schlanker Kern von Abstraktionen und Protokollen, die die grundlegende Funktionalität zum gemeinsamen Verwenden einzelner Ressourcen in verteilten Systemen rea-

Abbildung 1.3 Entwicklung der Chip Technologie für Parallelrechner [89]



lisieren. Oberhalb dieser Schicht findet die Koordination und Verwaltung einer Ansammlung von Ressourcen statt *(Collective)*. Unterhalb der Kernschicht befindet sich eine breite Schicht möglicher Ressourcen, die durch die Grid Infrastruktur zur Verfügung gestellt werden können *(Fabric)*.

Zur Verdeutlichung der Anforderungen im Bereich des Grid Computing sollen an dieser Stelle noch zwei Aussagen von Prof. Francine Berman, Direktorin des San Diego Supercomputer Center (SDSC), einem der zukünftigen Hauptknotenpunkte des *Teragrid* [87] Projekts, herangezogen werden.

- *"One should not be a hero to write programs for the Teragrid"*
- *"Uniformity is convenient, heterogeneity inescapable"*

Wie im nächsten Abschnitt ausgeführt wird, sind einfache Programmierung und Unterstützung von Heterogenität auch zentrale Zielsetzungen der in dieser Arbeit vorgestellten Systemumgebung.

1.5 Motivation und Zielsetzung der Arbeit

Die vorliegenden Arbeit beschreibt die Realisierung einer Systemumgebung zur einfachen Erstellung von parallelen C++ Programmen und deren Ausführung in heterogenen verteilten Systemen. In diesem Abschnitt sollen die Zielsetzungen und die Motivation für diese Arbeit diskutiert werden.

1.5.1 Programmiermodell

Für die Erstellung paralleler C++ Anwendungen ist ein Programmiermodell erforderlich, mittels dem Parallelität auf Programmebene formuliert werden kann. In Kapitel 2 wird ausführlich auf diese Thematik eingegangen. An dieser Stelle sollen vor allem die für diese Arbeit spezifischen Anforderungen an das Programmiermodell diskutiert werden. Es handelt sich dabei um die folgenden Aspekte:

- **Einfache und schnelle Programmerstellung**
  Algorithmen im Bereich des Symbolischen Rechnens weisen keine hohe temporale Stabilität auf. Oftmals werden durch Verbesserung des sequentiellen Verfahrens Beschleunigungen um mehrere Größenordnungen erzielt. Der Einsatz von paralleler Software macht in diesen Fällen nur Sinn, wenn sie stets auf den aktuellsten (sequentiellen) Algorithmen basiert ist. Für das zu wählende Programmiermodell bedeutet dieser Umstand, dass es eine möglichst einfache und schnelle Erstellung von parallelen Applikationen erlaubt. Insbesondere sollte die Transformation eines sequentiellen Programms in ein paralleles Programm leicht durchführbar sein. Außerdem müssen parallele Programme einfach an neue Entwicklungen angepasst werden können.

  Hierfür sind die folgenden Punkte wesentlich:

  - Zur Parallelisierung sollten keine großen Veränderungen an der Programmstruktur des sequentiellen Programms vorgenommen werden müssen.
  - Das parallele Programm sollte eine einfache und übersichtliche Struktur aufweisen.
  - Der Programmierer sollte sich nicht mit (plattformspezifischen) Systemdetails, wie etwa Kommunikation oder Synchronisation befassen müssen.

- **Unterstützung aktuell relevanter Architekturen**
  Wie in Abschnitt 1.3 gezeigt wurde, ist die Entwicklung der Parallelrechnerarchitekturen starken Veränderungen unterworfen. Zur Zeit sind hauptsächlich SMP und Parallelrechner mit verteiltem Hauptspeicher im Einsatz. Das Programmiermodell sollte möglichst unabhängig von der Architektur des zur Programmausführung verwendeten Parallelrechners sein. Insbesondere sollte ein Programm sowohl auf SMP Systemen, als auch verteilten Systemen (sowie auf hierarchischen Systemen) effizient ausgeführt

werden können und somit die einfache Migration zwischen diesen Plattformen ermöglichen.

- **Unterstützung irregulärer Probleme**
  Probleme aus dem Bereich des Symbolischen Rechnens weisen oftmals einen stark irregulären Charakter auf. Das bedeutet, dass sich die einzelnen Teilprobleme der parallelen Berechnung stark hinsichtlich ihrer Struktur und Berechnungszeit unterscheiden. Das Programmiermodell sollte geeignete Konstrukte zur Verfügung stellen, mit denen Irregularität behandelt werden kann.

1.5.2 Verteiltes paralleles Rechnen in heterogenen Rechnernetzen

Zur Realisierung eines parallelen Programmiermodells ist eine Systemumgebung erforderlich, die die Ausführung von entsprechenden Programmen ermöglicht. Für den praktischen Einsatz der Systemumgebung ist die Unterstützung von heterogenen Rechnernetzwerken, die aus einer Vielzahl unterschiedlicher Rechnern und Betriebssystemen bestehen, ein wichtiger Punkt. Gerade im zukunftsträchtigen Bereich des Grid Computing ist die Unterstützung von heterogenen Rechenumgebungen eine wesentliche Anforderung.

In den letzten Jahren erfuhr die plattformübergreifende Programmiersprache Java [4, 49] eine wachsende Bedeutung. Bei Java handelt es sich um eine interpretierte Programmiersprache. Ein Java Quellprogramm wird von einem Compiler in einen plattformunabhängigen Maschinencode, den *Bytecode*, umgewandelt. Der Bytecode wird durch einen Interpreter, der als *Java Virtual Machine (JVM)* bezeichnet wird, ausgeführt. Java Programme können also auf jeder Plattform ablaufen, für die eine JVM zur Verfügung steht.

Durch die interpretierte Verarbeitung können Java Programme grundsätzlich nicht mit derselben Effizienz ausgeführt werden, wie dies für C/C++Programme möglich ist, die direkt in die jeweilige Maschinensprache des verwendeten Prozessors übersetzt werden. Für einige, wenige Plattformen, wie etwa Microsoft Windows, stehen hochoptimierte JVMs zur Verfügung, die durch die Anwendung spezieller Just-in-Time Compilationstechniken die Ausführungszeiten von Java Programmen in bestimmten Fällen denen von nicht-interpretierten Sprachen annähern. Für viele Plattformen sind aber derartig hochoptimierte JVMs nicht verfügbar. Zusammenfassend kann gesagt werden, dass mit Java zwar plattformunabhängige, verteilte parallele Programme erstellt werden können, diese aber in stark heterogenen Umgebungen nicht mit der maximal möglichen Effizienz ausführbar sind.

Da heutzutage Mikroprozessorarchitekturen stets zusammen mit zugehörigen

Compilern entwickelt werden, ist sichergestellt, dass für jede Plattform hochoptimierte C/C++ Compiler zur Verfügung stehen.

Aus den angeführten Gründen ist C++ zur Erstellung von parallelen verteilten Applikationen (auch) für heterogene Umgebungen gegenüber Java (und anderen interpretierten Sprachen, wie etwa C#) heute noch vorzuziehen.

1.6 Aufbau der Arbeit

An dieser Stelle soll ein kurzer Überblick über den Aufbau der Arbeit und den Inhalt der einzelnen Kapitel gegeben werden.

- Das Kapitel 2 gibt eine Einführung in das Gebiet der Programmiermodelle für paralleles Rechnen. Es werden Anforderungen an Programmiermodelle formuliert, sowie verschiedene Klassifikationsstrategien vorgestellt. Abschließend wird ein Reihe ausgewählter Programmiermodelle ausführlich beschrieben.

- In Kapitel 3 wird das im Rahmen der vorliegenden Arbeit realisierte *Distributed Object-Oriented Threads System* (DOTS) vorgestellt. DOTS ist eine Systemumgebung zur Erstellung verteilter paralleler C++ Programme für heterogene Rechnernetzwerke. Es werden ausführlich die von DOTS realisierten Programmiermodelle beschrieben, und es wird auf verschiedene Aspekte der Systemarchitektur näher eingegangen.

- Kapitel 4 befasst sich mit einigen ausgewählten Implementierungsaspekten von DOTS. Dabei wird besonders auf den Aspekt der Realisierung von plattformunabhängigen Programmen und die Optimierung der objektorientierten Kommunikation für heterogene Netzwerke eingegangen.

- Eine mittels DOTS realisierte Applikation zur verteilten parallelen Erfüllbarkeitsprüfung von booleschen Formeln ist Gegenstand von Kapitel 5. Die vorgestellte Anwendung parallelisiert einen neuartigen Algorithmus, der ein dynamisches Lernverfahren in den klassischen Davis-Putnam Algorithmus zur Erfüllbarkeitsprüfung integriert. Neben den theoretischen Grundlagen der Parallelisierung wird ausführlich die Realisierung des parallelen Erfüllbarkeitsprüfers mit DOTS dargestellt. Das Kapitel schließt mit der Beschreibung einer industriellen Anwendung des vorgestellten Verfahrens.

- In Kapitel 6 werden zwei weitere parallele Anwendungen beschrieben, die mit DOTS realisiert wurden. Der erste Teil des Kapitels befasst sich mit ei-

ner verteilten parallelen Faktorisierungsapplikation, die nach der Methode der elliptischen Kurven arbeitet. Im zweiten Teil wird die Realisierung einer parallelen Anwendung aus dem Bereich der Computer Graphik dargestellt. Es handelt sich dabei um eine Applikation zur Volumenvisualisierung.

- Das Kapitel 7 gibt einen vergleichenden Überblick über andere Systemplattformen, die zu DOTS ähnliche Programmiermodelle unterstützen.
- In Kapitel 8 werden die wesentlichen Beiträge der Arbeit zusammengefasst.

2

Kapitel 2

Programmiermodelle für das parallele Rechnen

Ein Programm, das auf einem Parallelrechner ausgeführt wird, besteht in der Regel aus einer Vielzahl von nebenläufig ablaufenden Einheiten. Auf Implementierungsebene können diese Ablaufeinheiten unter anderem durch Prozesse oder auch durch leichtgewichtigere Threads realisiert sein. Alle Ablaufeinheiten können während der Programmausführung miteinander kommunizieren, wobei je nach betrachtetem Problem komplexe Kommunikationsmuster resultieren. Ein in Ausführung befindliches paralleles Programm stellt also ein Objekt mit einer extrem komplexen Abfolge von Zustandsübergängen und folglich einer praktisch unüberschaubar großen Zustandsmenge dar.

Die statische Beschreibung dieses komplexen Laufzeitobjektes, also der Programmtext, muss aber wesentlich einfacher strukturiert sein, damit sie für den Programmierer intellektuell handhabbar bleibt. Für die Vereinfachung der Erstellung von parallelen Programmen wurde deshalb bereits eine Vielzahl von *Programmiermodellen* vorgeschlagen.

Ein Programmiermodell für das parallele Rechnen stellt eine Schnittstelle zwischen der Programmierebene und der Implementierungsebene dar. Eine solche Schnittstelle soll sowohl möglichst mächtige und ausdrucksstarke Primitive zur Verfügung stellen, als auch eine möglichst effiziente Implementierung auf verschiedenen Klassen von paralleler Hardware erlauben. Alternativ kann man daher ein Programmiermodell auch als eine Art abstrakte parallele Maschine charakterisieren, welche die physische Realisierung und die Komplexität des tatsächlich verwendeten Parallelrechners verbirgt.

Programmiermodelle erlauben es außerdem, den Aspekt der Programmerstellung und den Aspekt der Programmausführung auf einer bestimmten Plattform getrennt zu betrachten. Somit können insbesondere diese beiden Aspekte von

unterschiedlichen Personen(-gruppen) bearbeitet werden. Entscheidungen, die die Implementierung eines Programmiermodells betreffen, müssen zudem nur einmal pro vorgesehener paralleler Hardwareplattform und nicht für jedes Programm erneut getroffen werden.

In diesem Kapitel wird der Stand der Technik im Bereich der Programmiermodelle für paralleles Rechnen dargestellt. Dazu werden zunächst allgemeine Anforderungen formuliert, die üblicherweise an Programmiermodelle für paralleles Rechnen zu stellen sind. Danach erfolgt die Darstellung verschiedener Klassifizierungmethoden für parallele Programmiermodelle. Anschließend werden anhand einer ausgewählten Klassifizierungsmethode mehrere unterschiedliche Programmiermodelle vorgestellt.

2.1 Anforderungen an Programmiermodelle

Ein Programmiermodell für das parallele Rechnen sollte möglichst viele der nachfolgend dargestellten Kriterien umfassend erfüllen [79].

2.1.1 Einfache Programmerstellung

Viele Anwendungsfälle weisen eine natürliche Parallelität auf. Die in diesen Fällen offensichtliche Struktur des parallelen Programms sollte möglichst einfach und ohne artifizielle Konstruktionen im betrachteten Programmiermodell verwirklicht werden können.

Ein Programmiermodell kann auf verschiedenen Ebenen angesiedelt sein. Die höheren Programmiermodelle zeichnen sich vor allem durch *Abstraktion* und die daraus resultierende *Stabilität* von Programmen aus.

- Abstraktion bedeutet, dass sich der Programmierer um möglichst wenig Details der Ausführung auf einer konkreten Hardwareplattform kümmern muss. Die Struktur der Programme wird somit vereinfacht und damit der Erstellungsprozess für korrekte Programme wesentlich beschleunigt. Im Extremfall tritt die Parallelität auf Programmtextebene gar nicht explizit zutage.

- Stabilität bedeutet, dass bei der Programmentwicklung davon ausgegangen werden kann, dass sich die Programmierschnittstelle über lange Zeiträume hinweg nicht ändert, ungeachtet der auftretenden Veränderungen bei der Architektur der parallelen Hardware.

Bei kommerziell eingesetzter paralleler Software ist insbesondere die Stabilität des Programmiermodells ein wichtiger Faktor, da sie die Investitionssicherheit für parallele Software wesentlich erhöht. Allgemein wird ein hohes Maß an Stabilität als wichtige Voraussetzung betrachtet, um parallele Programmiertechniken dauerhaft als Standardmethode sowohl innerhalb der Informatik, als auch in den verschiedenen Anwendungsdisziplinen etablieren zu können.

Programmiermodelle auf niedrigerer Abstraktionsebene haben hingegen den Vorteil, dass sie oftmals effizienter implementiert werden können, da der Programmierer schon viele Details der parallelen Ausführung in dem erstellten Programm explizit festlegt. Diese müssen somit nicht automatisch vom Compiler oder vom Laufzeitsystem bestimmt werden.

2.1.2 Softwareentwicklungsmethodik

Bei der sequentiellen Programmierung werden üblicherweise aufwändige Testmethoden angewendet, um die erstellten Programme von Fehlern zu bereinigen. Grundsätzlich kann durch Testen immer nur die Anwesenheit von Fehlern festgestellt, nicht aber deren Abwesenheit bewiesen werden. Dennoch hat sich Testen als zuverlässige Methode zur Qualitätssicherung von sequentieller Software erwiesen. Die Anwendung von Testverfahren, bestehend aus der Iteration von Test- und Debug-Zyklen, wird aus den folgenden Gründen im Falle von paralleler Software erheblich erschwert:

- Die Anzahl der möglichen Zustände eines parallelen Programms ist viel größer als die eines sequentiellen Programms, so dass die Zahl der durchzuführenden Tests extrem groß ist.
- Durch die Kommunikation zwischen den einzelnen Ablaufeinheiten wird die Programmausführung nichtdeterministisch. Die Reproduktion eines bestimmten, fehlerhaften Verhaltens des Programms wird damit erheblich erschwert.
- Parallele Programme können auf unterschiedlichen parallelen Architekturen zur Ausführung kommen. Je nach verwendeter Architektur kann sich ein und dasselbe Programm bei seiner Ausführung vollkommen unterschiedlich verhalten. In der Regel stehen aber bei der Softwareerstellung nicht alle potentiellen Ziel-Architekturen zur Durchführung aller notwendigen Tests zur Verfügung.

Ebenso wie bei der sequentiellen Programmierung sind formale Verifikationsmethoden für parallele Programme ab einer bestimmten Programmkomplexität

praktisch nicht durchführbar. Für viele Programmiermodelle für das parallele Rechnen existieren zudem noch keine ausreichenden formalen Modelle, so dass hier Verifikationsmethoden sogar grundsätzlich nicht angewendet werden können.

Aufgrund dieser Situation ist es für die parallele Programmierung sehr wichtig, dass ein Programmiermodell eine Software-Entwicklungsmethodik ermöglicht, die die Erstellung von korrekten Programmen in besonderem Maße fördert.

2.1.3 Plattformunabhängigkeit

Die rasanten Fortschritte bei der Prozessor-, Speicher- und Netzwerktechnologie führen zu immer neuen Generationen von Parallelrechnern, die sich zum Teil erheblich in der grundlegenden Architektur von der vorhergehenden Generation unterscheiden. Ein Programmiermodell sollte daher möglichst plattformunabhängig sein. Diese Anforderung umfasst sowohl statische als auch dynamische Aspekte. Die Portierung eines bestehenden parallelen Programms sollte kein grundsätzlich neues Design oder sonstige weitergehende Veränderungen am Programmtext erfordern. Ebenso sollte das portierte Programm auf der neuen Hardwareplattform effizient ausgeführt werden können, ohne dass größere Anpassungen vorzunehmen sind.

2.1.4 Einfache Erlernbarkeit

Neben den klassischen Anwendungsbereichen für das parallele Rechnen, wie etwa numerischen Berechnungen zur Simulationen komplexer physikalischer Vorgänge, treten in zunehmendem Maße Anwendungsmöglichkeiten aus anderen wissenschaftlichen Fachgebieten, zum Beispiel der Biologie oder den Wirtschaftswissenschaften zutage. Es muss also davon ausgegangen werden, dass die Programmierer von parallelen Anwendungen aus anderen wissenschaftlichen Disziplinen stammen und somit kein spezielles Fachwissen über parallele Programmierung vorausgesetzt werden kann. Programmiermodelle sollten somit möglichst einfach zu erlernen und intuitiv anzuwenden sein. Dieser Aspekt ist natürlich auch bei der Weiterbildung von Programmierern, die bisher ausschließlich mit der Erstellung von sequentiellen Programmen vertraut sind, von großer Bedeutung, um die allgemeine Akzeptanz der parallelen Programmierung innerhalb der Informatik zu steigern.

2.2 Klassifizierung von Programmiermodellen

In der Vergangenheit wurde eine Vielzahl von Programmiermodellen für das parallele Rechnen vorgeschlagen. Zum Beispiel werden im Übersichtsartikel von *Bal, Steiner und Tanenbaum* [5] Referenzen für fast 100 Programmiersprachen allein für verteilte Systeme angeführt. Diese Vielzahl an verschiedenen Sprachen und Programmiermodellen für das parallele Rechnen macht eine systematische Klassifizierung erforderlich. Für die Einteilung von Programmiermodellen in geeignete Klassen werden in der Literatur mehrere Schemata vorgeschlagen. Bislang hat sich aber noch keines dieser Schemata als universell anwendbare Klassifizierungsmethode durchsetzen können. Im Folgenden sollen drei der vorgeschlagenen Klassifizierungsansätze genauer betrachtet werden. Die dargestellten Klassifizierungsschemata zeigen gleichzeitig auch die vielfältigen Entwurfsmöglichkeiten für parallele Programmiermodelle auf.

2.2.1 Klassifizierung anhand allgemeiner Merkmale

Foster [28] betrachtet verschiedene allgemeine Merkmale von parallelen Programmiermodellen, die zu deren Klassifizierung herangezogen werden können. Im Einzelnen erfolgt die Einteilung entlang der nachfolgend dargestellten Achsen.

- **Spracherweiterung oder neuartige Sprache**
  Es kann entweder eine herkömmliche sequentielle Sprache um parallele Konstrukte erweitert oder eine völlig neue Sprache eingeführt werden. Bei Spracherweiterungen können vorhandene Programmierkenntnisse ebenso wie evtl. vorhandener Programmcode in der entsprechenden sequentiellen Programmiersprache für die parallele Programmierung weiter genutzt werden. Desweiteren können vorhandene Programmiertools, wie zum Beispiel Debugger oder auch ganze Entwicklungsumgebungen weiterhin eingesetzt werden. Eine Spracherweiterung kann durch Compiler, bzw. Präcompiler realisiert werden oder aber durch einen bibliotheksbasierten Ansatz, bei dem die zusätzliche Funktionalität in Form von Bibliotheksfunktionen zur Verfügung gestellt wird. Der bibliotheksbasierte Ansatz ist zwar in seiner Ausdrucksmächtigkeit beschränkt, hat aber den Vorteil, dass die gesamte Programmierumgebung (zum Beispiel Debugger) ohne Anpassungen weiterhin Verwendung finden kann.

  Gänzlich neue Sprachen können die semantischen Barrieren der herkömmlichen sequentiellen Sprachen bezüglich der Beschreibung von Parallelität durch vollkommen andersartige Ansätze überwinden.

- **Taskparallelität oder Datenparallelität**
  Bei Programmiermodellen, die für Taskparallelität konzipiert sind, führen verschiedene Ablaufeinheiten verschiedene Aufgaben parallel aus. Programmiermodelle, die primär für Datenparallelität entworfen sind, unterstützen die parallele Anwendung derselben Abfolge von Operationen auf verschiedene Bereiche einer Datenstruktur in besonderer Art und Weise. Taskparallelität ist der allgemeinere Ansatz, entsprechende Programmiermodelle sind somit für ein breiteres Spektrum von Anwendungen einsetzbar. Für reguläre Probleme können jedoch mit datenparallelen Programmiermodellen gut strukturierte und effiziente Programme erstellt werden.

- **Programmiersprache oder Koordinationsprache**
  Die Sprache kann als eigenständige Programmiersprache gedacht sein, oder aber lediglich als Koordinationssprache für die parallele Ausführung von Programmen, die in einer herkömmlichen sequentiellen Sprache verfasst sind.

- **Architekturspezifisch oder architekturunabhängig**
  Das Programmiermodell kann für eine bestimmte Architektur von Parallelrechnern vorgesehen sein (z.B. für shared-memory oder distributed-memory Rechner), oder als allgemeines Modell für das parallele Programmieren gedacht sein. Die erste Gruppe umfasst auch Programmiersprachen, die speziell für ganz bestimmte Parallelrechner entwickelt wurden (zum Beispiel die Sprache OCCAM für den Transputer [44]).

2.2.2 Klassifizierung anhand funktionaler Teilaspekte

Das in diesem Abschnitt vorgestellte Einteilungsschema verwendet eine Reihe von Kriterien, anhand derer bestimmte funktionale Teilaspekte des betrachteten Programmiermodells untersucht und in Kategorien eingeordnet werden.

Der zunächst betrachtete Teilaspekt ist die Art und Weise, wie die Parallelität in einem Programmiermodell ausgedrückt wird. Es werden also die Programmeinheiten untersucht, die parallel zur Ausführung kommen. Dazu werden die folgenden Kategorien für parallele Ablaufeinheiten aufgestellt:

- **Prozesse**
  Prozesse können als logischer Prozessor betrachtet werden, der sequentiellen Programmcode ausführt und dazu einen eigenen Zustand besitzt. Prozesse werden entweder implizit durch deren Deklarierung erzeugt, oder

es kann explizit ein entsprechendes Primitivum zur Prozesserzeugung vorgesehen sein. Bei der impliziten Prozesserzeugung muss die Anzahl der benötigten Prozesse zur Übersetzungszeit bekannt sein.

- **Objekte**
  Objekte in sequentiellen objektorientierten Programmiersprachen haben einen passiven Charakter. Ein Objekt wird erst dann aktiviert, wenn es eine Nachricht von einem anderen Objekt erhält. Solange der Empfänger der Nachricht aktiv ist, wartet der Sender der Nachricht auf das Ergebnis und ist somit also passiv. Nachdem das Ergebnis berechnet ist, wird der Empfänger wieder passiv und der Sender fährt mit der Ausführung fort. Zu jedem Zeitpunkt ist also nur ein Objekt im System aktiv. Parallelität kann durch Erweiterung dieses sequentiellen Modells auf folgende Arten erhalten werden:

  - Ein Objekt ist aktiv, ohne dass es eine Nachricht empfangen hat.
  - Der Sender wird nach dem Abschicken der Nachricht parallel zum Empfänger ausgeführt.
  - Nachdem das Resultat berechnet wurde, wird der Empfänger weiter ausgeführt.
  - Nachrichten werden gleichzeitig zu mehreren Objekten geschickt.

- **Anweisungen**
  Anweisungen, die parallel ausgeführt werden sollen, sind im Programm syntaktisch ausgewiesen. Dies kann zum Beispiel statisch durch parallele Blöcke oder dynamisch durch parallele Schleifenkonstrukte realisiert werden. Bei parallelen Schleifen werden die einzelnen Iterationen des Schleifenrumpfes parallel ausgeführt.

- **Funktionen**
  Wenn das Ergebnis von Funktionen nur von ihren Eingabeparametern abhängt, ist deren Ausführungsreihenfolge nicht relevant, sie können also prinzipiell auch parallel ausgeführt werden. Die einzige Einschränkung der Parallelität besteht darin, dass die Eingabeparameter einer Funktion vom Ergebnis einer anderen Funktion abhängen können und deshalb auf deren Berechnung gewartet werden muss.

- **Klauseln**
  Diese Programmeinheit der Parallelität ist für logik-basierte Programmiermodelle von Bedeutung.

Der zweite zu untersuchende Teilaspekt befasst sich mit der Art der Zuordnung von parallelen Ablaufeinheiten zu den vorhandenen Prozessoren des Parallelrechners. Zur Einordnung sind hierfür die folgenden Kategorien ausgewiesen:

- **Explizite oder implizite Zuordnung**
  Die Zuordnung einer parallelen Ablaufeinheit zu einem Prozessor kann explizit vom Programmierer festgelegt oder auch implizit vom Compiler oder Laufzeitsystem vorgenommen werden. Die explizite Zuordnung ist einerseits für den Programmierer weniger komfortabel, andererseits lassen sich aber bekannte Eigenschaften der betrachteten Anwendung vom Programmierer für eine effiziente Zuordnung ausnutzen.

- **Zeitpunkt der Zuordnung**
  Für den Zeitpunkt der Zuordnung einer parallelen Einheit zu einem der vorhandenen Prozessoren gibt es drei Möglichkeiten.

  - Feste Zuordnung zur Übersetzungszeit
    Diese Möglichkeit ist wenig flexibel, der Vorteil besteht aber darin, dass vom Programmierer Ablaufeinheiten, die häufig miteinander kommunizieren, benachbart platziert werden können.
  - Feste Zuordnung zur Laufzeit
    Hier wird bei der Erzeugung einer neuen Ablaufeinheit ein Prozessor fest für die gesamte Zeitdauer ihrer Ausführung zugeordnet.
  - Keine feste Zuordnung
    Diese Möglichkeit ist sehr flexibel, aber in vielen Fällen nur mit großem Aufwand zu realisieren, da Ablaufeinheiten zur Laufzeit zwischen den Prozessoren migriert werden müssen.

Der im Folgenden betrachtete Teilaspekt der Interprozesskommunikation und Synchronisation ist vor allem für Parallelrechnerarchitekturen mit verteiltem Speicher relevant. Zur Klassifizierung sind die folgenden Kriterien vorgesehen:

- **Nachrichtenaustausch**
  Der explizite Austausch von Nachrichten zwischen den Knoten eines Parallelrechners mit verteiltem Speicher wird häufig als die natürlichste Art der Interprozesskommunikation für derartige Architekturen betrachtet, da der verteilte Charakter hier explizit zutage tritt. Der explizite Austausch von Nachrichten kann nach den im Folgenden dargestellten Schemata ablaufen:

  - Explizite oder implizite Kommunikation
    Der Empfang einer Nachricht kann explizit oder implizit ablaufen. Beim expliziten Empfang ruft der Empfänger zuvor ein entsprechendes Primitiv auf, um die Empfangsbereitschaft herzustellen. Bei der impliziten Empfangsmethode wird, wenn die Nachricht beim Empfänger angekommen ist, automatisch Programmcode zu deren Verarbeitung ausgeführt.
  - Symmetrische oder asymmetrische Adressierung
    Bei der symmetrischen Adressierung benennen sowohl der Sender, als auch der Empfänger den jeweiligen Kommunikationspartner, während bei der asymmetrischen Adressierung der Empfänger keinen bestimmten Sender festlegt.
  - Direkte oder indirekte Adressierung
    Bei der direkten Adressierung wird für den Empfänger einer Nachricht eine bestimmte Ablaufeinheit benannt, während bei der indirekten Adressierung eine Zwischeninstanz, oft Mailbox genannt, adressiert wird, aus der der Empfänger die Nachricht liest.
  - Synchrone oder asynchrone Kommunikation
    Beim synchronen Nachrichtenaustausch wird der Sender einer Nachricht solange blockiert, bis der Empfänger die Nachricht akzeptiert hat. Im Gegensatz zum asynchronen Nachrichtenaustausch umfasst der synchrone Nachrichtenaustausch also noch zusätzlich die Synchronisation zwischen Sender und Empfänger.

- **Strukturierter Nachrichtenaustausch**
  Beim strukturierten Nachrichtenaustausch unterliegt die Kommunikation zwischen den Knoten vorgegebenen Mustern und ist meistens mit der Synchronisation der beteiligten Kommunikationspartner verbunden. Im Wesentlichen sind die folgenden Verfahren für diese Art der Interprozesskommunikation relevant:

  - Entfernter Funktionsaufruf (Remote Procedure Call, RPC)
    Ein entfernter Funktionsaufruf unterscheidet sich von einem normalen Funktionsaufruf im Wesentlichen dadurch, dass der aufrufende und der ausführende Prozess der Funktion auf verschiedenen Rechnern ausgeführt wird. Die Eingabeparameter und das Resultat werden transparent in jeweils eine Nachricht verpackt *(parameter marshalling)* und entsprechend verschickt. Der Aufrufer wird solange blockiert, bis das Funktionsresultat eintrifft.
  - Rendezvous
    Das Rendezvous-Konzept wird durch die Primitive *Entry-Deklaration*

und *Accept-Anweisung* beim einen und *Entry-Aufruf* beim anderen der beiden Kommunikationspartner realisiert. Die Entry-Deklaration ähnelt einer Prozedurdeklaration im herkömmlichen Sinne. Sie besteht aus einem Bezeichner und mehreren formalen Parametern. Der Entry-Aufruf ist vergleichbar mit einem gewöhnlichen Prozeduraufruf. Neben dem Bezeichner für das betreffende Entry werden aktuelle Parameter und der Ausführungsort angegeben. Mit der Accept-Anweisung für ein Entry wird ein Block von weiteren Anweisungen festgelegt, die ausgeführt werden, wenn das Entry aufgerufen wird.

Ein Rendezvous zwischen zwei Prozessen P und Q kommt zu Stande, wenn P ein Entry in Q aufgerufen hat und Q die Accept-Anweisung für das Entry ausführt. Die Kommunikation läuft beim Rendezvous-Konzept vollkommen synchronisiert ab. P wird solange blockiert, bis Q alle zum Accept Block gehörenden Anweisungen ausgeführt und evtl. Ergebnisse an P zurückgeliefert hat. Nach dem Rendezvous werden P und Q weiter parallel ausgeführt.

Im Unterschied zum Remote Procedure Call sind beim Rendezvous-Konzept beide Kommunikationspartner gleichzeitig aktiv, sie synchronisieren sich nur für die Zeitdauer der Entry-Ausführung.

- **Gemeinsame Datenbereiche**
  Bei diesem Modell erfolgt die Kommunikation implizit über den Zugriff auf Variablen. Im Gegensatz zur Interprozesskommunikation durch (strukturierten) Nachrichtenaustausch erfolgt bei diesem Kommunikationskonzept der Datenaustausch anonym und asynchron. Ein Kommunikationsteilnehmer, der ein Datum schreibt, weiß nicht von wem es zu welchem Zeitpunkt gelesen wird. Entsprechend weiß ein Kommunikationsteilnehmer, der Daten liest nicht, wann und von wem diese geschrieben wurden.

  - Virtual Shared Memory
    Bei diesem Programmiermodell wird dem Programmierer ein gemeinsamer Adressraum zur Verfügung gestellt, der von der tatsächlichen Anordnung des physischen Speichers abstrahiert. Die Kommunikation zwischen den Prozessen geschieht somit implizit über den Zugriff auf Speicherstellen des gemeinsamen Adressraumes. Referenzen auf entfernte Speicherstellen werden je nach Implementierung entweder vom Compiler oder vom Laufzeitsystem ein einen Nachrichtenaustausch umgesetzt.
  - Variablen mit Single Assignment Eigenschaft
    Das Konzept der Variablen mit Single Assignment Eigenschaft ist hauptsächlich bei Logik basierten parallelen Programmiersprachen

anzutreffen. Es realisiert einen einheitlichen Mechanismus sowohl zur Kommunikation, als auch zur Synchronisation zwischen Kommunikationsteilnehmern. Variablen mit dieser Eigenschaft sind zunächst ungebunden. Nachdem ihnen durch Unifikation ein Wert zugewiesen wurde, kann dieser nicht mehr verändert werden. Wird auf eine Variable zugegriffen, bevor ihr ein Wert zugewiesen wurde, blockiert der lesende Prozess.

2.2.3 Klassifizierung nach dem Grad an Abstraktion

Skillicorn und Talia [79] wählen als Grundlage der Klassifizierung das Maß an Abstraktion, das das betrachtete Programmiermodell bietet. Als Kriterium zur Einordnung eines Programmiermodells wird dabei herangezogen, ob die folgenden Teilaspekte der Parallelisierung eines Problems explizit vom Programmierer betrachtet werden müssen oder implizit von der Implementierung des Programmiermodells behandelt werden.

1. Zerlegung in Teilprobleme *(Decomposition)*
   Das Gesamtproblem muss in mehrere Teilprobleme aufgeteilt werden, so dass diese parallel auf den vorhandenen Prozessoren bearbeitet werden können. Sowohl Programmcode, als auch Datenstrukturen können von diesem Zerlegungsvorgang erfasst werden.

2. Zuordnung von Teilproblemen zu Prozessoren *(Mapping)*
   Für jedes der erzeugten Teilprobleme muss festgelegt werden, auf welchem Prozessor es ausgeführt werden soll. Oftmals müssen bei dieser Zuordnung bestimmte Einschränkungen beachtet werden. Zum Beispiel kann die Ausführung eines Teilproblems spezielle Hardware erfordern, etwa Ein-/Ausgabebausteine. Außerdem sollten Teilprobleme, die bei ihrer Ausführung häufig miteinander kommunizieren auf möglichst dicht benachbarten Prozessoren (bezüglich der Topologie des Verbindungsnetzwerkes) ausgeführt werden.

3. Kommunikation *(Communication)*
   Soll während der Ausführung eines Teilproblems auf Daten zugegriffen werden, die nicht lokal verfügbar sind, müssen diese durch einen geeigneten Kommunikationsvorgang in den lokalen Speicher übertragen werden. Die genaue Realisierung der Kommunikation hängt stark von der verwendeten Hardware ab.

4. Synchronisation *(Synchronization)*
   Explizite Synchronisation ist dann erforderlich, wenn sichergestellt wer-

den muss, dass die Ausführung von zwei oder mehreren Teilproblemen einen bestimmten, gemeinsamen Zustand erreicht hat.

Bei den Programmiermodellen, die die höchste Abstraktion aufweisen, sind alle oben aufgeführten Teilaspekte der Parallelisierung implizit geregelt. Je weniger abstrakt ein Programmiermodell ist, desto mehr Teilaspekte müssen vom Programmierer explizit realisiert werden. Aus diesem Klassifikationsschema lassen sich 6 verschiedene Abstraktionsebenen für Programmiermodelle ableiten.

- Abstraktionsebene 5
  Bei Programmiermodellen auf dieser Abstraktionsebene tritt die Parallelität im Programm nicht explizit auf. Bei der Programmerstellung spielt es also keine Rolle, dass das Programm später auf einem Parallelrechner ausgeführt wird.

- Abstraktionsebene 4
  Zur Erstellung von Programmen in Modellen dieser Ebene muss vom Programmierer die mögliche Parallelität im Programm ausgezeichnet werden. Wieviel Parallelität zur Laufzeit des Programms tatsächlich ausgenutzt wird, ist hingegen für den Programmierer transparent.

- Abstraktionsebene 3
  Programmiermodelle dieser Ebene erfordern die explizite Zerlegung des Problems in Teilprobleme durch den Programmierer, die weitergehenden Fragestellungen der Abbildung der Teilprobleme auf Prozessoren, sowie deren Kommunikation und Synchronisation sind implizit geregelt.

- Abstraktionsebene 2
  Auf dieser Abstraktionsebene muss vom Programmierer sowohl eine Aufteilung in Teilprobleme, als auch die Zuordnung der Teilprobleme zu Prozessoren vorgenommen werden. Die Kommunikations- und Synchronisationsaspekte sind hingegen implizit.

- Abstraktionsebene 1
  Bei den Modellen auf dieser Ebene wird vom Programmierer die Zerlegung in Teilprobleme und deren Zuordnung zu den vorhandenen Prozessoren, sowie die Kommunikation der Teilprobleme explizit vorgenommen. Lediglich die fein-granulare Synchronisation geschieht implizit.

- Abstraktionsebene 0
  Auf dieser Abstraktionsebene werden alle Teilaspekte der Parallelisierung explizit vom Programmierer betrachtet und treten somit explizit im Programm auf.

Tabelle 2.1 Einordnung der Beispiele für Programmiermodelle

Ebene	Programmiermodell
5	Parallelisierende Compiler, funktionale Programmierung, Logik basierte Programmierung
4	Datenflusssprachen, Future Konstrukt in Multilisp, High Performance Fortran (HPF)
3	Bulk Synchronous Parallelism (BSP)
2	Tupelräume in Linda, Java Spaces
1	Aktive Nachrichten, Mentat Programming Language (MPL)
0	Message Passing Interface (MPI), Tübingen Parallel Objects (TPO++)

2.3 Beispiele für Programmiermodelle

In diesem Abschnitt sollen einige Beispiele für Programmiermodelle vorgestellt werden. Die Anordnung der Darstellung folgt der in Abschnitt 2.2.3 behandelten Klassifizierung nach dem Grad der Abstraktion. Sie beginnt mit Beispielen für Programmiermodelle, bei denen Parallelität nicht explizit auftaucht (Abstraktionsebene 5) und schließt mit der Behandlung von Programmiermodellen ab, bei denen alle Aspekte der parallelen Ausführung explizit vom Programmierer festgelegt werden (Abstraktionsebene 0).

Tabelle 2.1 gibt einen Überblick über die nachfolgend diskutierten Beispiele für Programmiermodelle und zeigt ihre Zuordnung zu einer Abstraktionsebene.

2.3.1 Vollständig implizite Parallelisierung (Abstraktionsebene 5)

Bei den Programmiermodellen in dieser Kategorie treten in den Programmen keinerlei parallele Konstrukte auf. Alle Aspekte der Parallelität sind auf Programmebene transparent und werden vom Compiler bzw. dem Laufzeitsystem behandelt.

Parallelisierende Compiler

Als parallelisierende Compiler werden Programmierwerkzeuge bezeichnet, die in einer herkömmlichen sequentiellen Programmiersprache geschriebene Programme automatisch in äquivalente parallele Programme umwandeln. Dieser Ansatz erweist sich aber bezüglich seiner allgemeinen Anwendbarkeit als problematisch, da zum Beispiel bei imperativen Programmiersprachen meist se-

quentielle Artefakte in Programmen vorkommen, die vom Compiler nicht automatisch behandelt werden können. Dieser Ansatz erfordert somit ein sorgfältig ausgewähltes sequentielles Programmiermodell um derartige Probleme zu umgehen.

Funktionale Programmierung

Funktionale Programmiersprachen sind gegenüber der oben angeführten Problematik resistent, da ihr Ausführungsmodell eine inhärente Parallelität aufweist. In diesen Programmiersprachen werden Funktionen als λ-Terme aufgefasst und ihre Werte mittels Reduktionen im λ-Kalkül berechnet [88]. Eine wesentliche Eigenschaft im Zusammenhang mit der Ausnutzung von Parallelität ist dabei, dass das Ergebnis der Auswertung einer Funktion grundsätzlich nur von deren Argumenten abhängig ist und keine Seiteneffekte hervorruft, wie dies bei Funktionen in imperativen Sprachen der Fall sein kann. Ein typischer Vertreter dieser rein funktionalen Programmiersprachen ist Haskell [43]. Funktionen werden durch Graphreduktion berechnet, wobei Funktionen als Bäume dargestellt werden, mit gemeinsamen Teilbäumen für gleiche Teilfunktionen (*hence graphs)*. Durch gegebene Berechnungsregeln werden Teilstrukturen des Graphen ausgewählt und reduziert. Die somit erhaltenen einfacheren Strukturen ersetzen die ausgewählte Struktur. Wenn keine Berechnungsregeln mehr angewendet werden können, stellt der restliche Graph das Ergebnis der Berechnung dar. Dieser Graphreduktionsprozess kann auf natürliche Weise parallelisiert werden, indem disjunkte Teilstrukturen parallel reduziert werden.

Logik basierte Programmierung

Bei der Ausführung von Programmen in einer Logik basierten Programmiersprache treten zweierlei Arten von inhärenter Parallelität auf, die mit *OR-Parallelität* und *AND-Parallelität* bezeichnet werden. Als Beispiel soll das folgende Programm betrachtet werden:

```
A :- B,C,D
A :- E,F
```

Für Logik basierte Programme gibt es mehrere äquivalente Lesearten. Das Potential für eine parallele Ausführung zeigt sich am Besten bei der prozeduralen Interpretation. Das Beispielprogramm kann prozedural folgendermaßen interpretiert werden: Um das Theorem A zu beweisen müssen entweder die Teiltheoreme *(Goals)* B, C und D oder die Teiltheoreme E und F bewiesen werden.

Es ergeben sich also die folgenden Möglichkeiten für eine parallele Bearbeitung des Programms:

- Bei der OR-Parallelität werden die beiden Klauseln für A parallel berechnet, bis entweder eine bewiesen wurde oder beide fehlschlagen.
- Unter AND-Parallelität versteht man die parallele Berechnung der Goals B, C und D bzw. E und F, bis alle Goals bewiesen sind oder eines fehlschlägt.

2.3.2 Explizite Parallelität (Abstraktionsebene 4)

Programmiermodelle dieser Art weisen explizite Konstrukte zur Auszeichnung von Parallelität auf. Der Programmierer kümmert sich aber nicht um weitergehende Details der Parallelisierung wie etwa die Aufteilung in Teilprobleme und deren Zuordnung zu den vorhandenen Prozessoren oder um Kommunikations- und Synchronisationsaspekte.

Datenflusssprachen

In Datenflusssprachen werden Berechnungen als einzelne, in der Regel kleine Operationen ausgedrückt. Die Berechnungsreihenfolge der Operationen einer Berechnung wird nur durch Datenabhängigkeiten zwischen den Operationen bestimmt. Operationen, die voneinander unabhängig sind, können somit parallel ausgeführt werden. Ein Beispiel für eine Datenflusssprache ist Sisal [78].

Future Konstrukt in Multilisp

In der Sprache Multilisp [38], einer Variante der Programmiersprache Lisp, wurde das *Future*-Konstrukt zur Kennzeichnung von Parallelität eingeführt. Der Ausdruck future X liefert unmittelbar ein so genanntes Future für den Wert des Ausdrucks X zurück und erzeugt einen neuen Task, der den Wert von X berechnet. Die Berechnung des Wertes von X und dessen weitere Verwendung laufen somit zunächst parallel ab. Wenn das Ergebnis der Berechnung von X vorliegt, wird das Future durch den berechneten Wert ersetzt, dieser Vorgang wird als *Auflösung* des Futures bezeichnet. Mit der Auflösung des Futures wird auch der Task terminiert, der die Berechnung des Wertes ausgeführt hat. Jeder Task, der den Wert des Futures benötigt, wird solange blockiert, bis das betreffende Future aufgelöst ist. Viele Operationen erfordern für ihre Ausführung zwingend die Auflösung eines Futures, zum Beispiel arithmetische Operationen oder Vergleiche. Es gibt jedoch auch Operationen, die ausgeführt werden können, ohne

dass das betreffende Future aufgelöst werden muss. Beispiele für solche Operationen sind unter anderem Zuweisung, Argumentübergabe an eine Funktion, oder auch das Einfügen in eine komplexere Datenstruktur. Unter Umständen kann also eine beträchtliche Anzahl von Operationen ausgeführt werden, ohne dass der betreffende Task auf die Auflösung eines Futures warten muss. Das Future-Konstrukt ist in Multilisp die einzige Möglichkeit, neue Tasks zu erzeugen; es besteht also eine eins-zu-eins Beziehung zwischen Tasks und Futures.

Das Konzept der Futures ähnelt in gewisser Weise der oben beschriebene Datenflussparallelität und ermöglicht somit deren Integration in die Klasse der funktionalen Programmiersprachen.

Datenparallele Programmierung in High Peformance Fortran

Das Grundprinzip der datenparallelen Programmierung besteht darin, dass dieselbe Operation parallel auf einzelne oder mehrere Elemente einer (üblicherweise umfangreichen) regulären Datenstruktur angewendet wird. High Performance Fortran (HPF) [51] ist eine Programmiersprache, die speziell für die datenparallele Programmierung entwickelt wurde. HPF basiert auf der Programmiersprache Fortran90.

Mit reinem Fortran90 kann bereits eine eingeschränkte Klasse von datenparallelen Programmen realisiert werden. Spezielle Compiler erzeugen aus diesen Programmen für die jeweilige Zielarchitektur ablauffähigen Programmcode. Für die Ausführung auf Parallelrechnerarchitekturen mit verteiltem Speicher wird typischerweise ein SPMD Programm generiert, das auf allen Prozessoren ausgeführt wird. Die einzelnen Programminstanzen bearbeiten jeweils eine Teilmenge der gesamten Datenstruktur. Dazu wird zunächst die Datenstruktur in disjunkte Teilmengen partitioniert und auf die einzelnen Programminstanzen verteilt. Zur Zuordnung der Operationen zu den Prozessoren wird die *owner rules* Regel angewendet. Das bedeutet, dass die Operation zur Berechnung eines Datenelements auf demjenigen Prozessor ausgeführt wird, dem das Datenelement zugeordnet wurde. Sind bei der Berechnung Datenelemente beteiligt, die anderen Prozessoren zugeordnet sind, müssen zusätzlich vom Compiler Kommunikationsaktionen generiert werden.

Die automatische Erzeugung dieser Programme durch den Compiler erfordert spezielle Analyse- und Optimierungstechniken. In [92] werden diese Verfahren genauer beschrieben.

In HPF sind gegenüber Fortran90 im Wesentlichen Programmierkonstrukte zur flexiblen Auszeichnung von parallelen Programmteilen und zur Steuerung der Datenverteilung auf die einzelnen Prozessoren hinzugefügt worden. Gegenüber

reinen Fortran90 Programmen sind Programme in HPF also auf der Abstraktionsebene 4 anzusiedeln. An den oben beschriebenen Implementierungstechniken ergeben sich aber keine grundlegenden Änderungen.

Die Erstellung eines datenparallelen Programms mit HPF wird durch zwei orthogonale Aspekte bestimmt. Dies ist zum einen der Aspekt der Auszeichnung der Parallelität und zum anderen der Aspekt der Datenverteilung mit dem Ziel einer möglichst hohen Datenlokalität.

Auszeichnung von Parallelität

Parallelität kann entweder implizit im Programm vorhanden sein oder explizit durch parallele Konstrukte ausgedrückt werden. Der Array-Zuweisungsoperator von Fortran90 ist ein Beispiel für ein explizites paralleles Konstrukt. Die Berechnung und Zuweisung der einzelnen Array-Elemente läuft parallel ab. Die folgenden Programmzeilen demonstrieren die Anwendung des Zuweisungsoperators.

```
real A(10,20)
real B(10,20)
logical L(10,20)
A = A + 1.0     ! Parallele Berechnung und Zuweisung
A = SQRT(A)
L = A .EQ. B
```

Implizite Parallelität tritt in Form von Programmschleifen zutage. In dem nachfolgenden Programmbeispiel kann vom Compiler die Unabhängigkeit der einzelnen Schleifeniterationen festgestellt und folglich automatisch paralleler Code erzeugt werden.

```
do i = 1,m
  do j = 1,n
    A(i,j) = B(i,j) * C(i,j)
  enddo
enddo
```

In manchen Fällen ist aber eine automatische Erkennung von impliziter Parallelität in Schleifen nicht möglich. In HPF wurde für diese Fälle die `INDEPENDENT` Direktive eingefügt, mittels der der Programmierer angeben kann, dass die einzelnen Iterationen einer Schleife unabhängig sind und somit parallel ausgeführt werden können. Das folgende Beispiel zeigt eine Verwendungsmöglichkeit der `INDEPENDENT` Direktive.

```
!HPF$ INDEPENDENT
```

```
do i = 1,n
  A(Index(i)) = B(i)
enddo
```

Datenverteilung

Ein anderer wesentlicher Aspekt der datenparallelen Programmierung ist die Verteilung der Daten auf die einzelnen Prozessoren. Die Zuordnung von Daten zu Prozessoren sollte eine möglichst hohe Lokalität aufweisen, so dass zur Berechnung der einzelnen Operationen möglichst wenig Kommunikationsaktionen durchzuführen sind. Die Optimierung der Lokalität der Daten kann die erreichbare Gesamtbeschleunigung eines datenparallelen Programms erheblich verbessern. Die Berechnung der besten Verteilung stellt jedoch ein globales Optimierungsproblem dar und ist somit im allgemeinen Fall nicht effizient automatisch durchführbar. In HPF werden deshalb Direktiven eingeführt, mit denen der Programmierer die Verteilung der Daten explizit steuern kann. HPF sieht dazu ein mehrstufiges Modell vor.

- In der ersten Stufe kann spezifiziert werden, welche Datenelemente auf denselben Prozessor abgebildet werden sollen. Dieser Vorgang wird auch *Kollokation* genannt. Hierzu dient die `ALIGN` Direktive. Im folgenden Programmfragment wird zum Beispiel festgelegt, dass die jeweils korrespondierenden Elemente zweier Arrays kolloziert werden sollen.

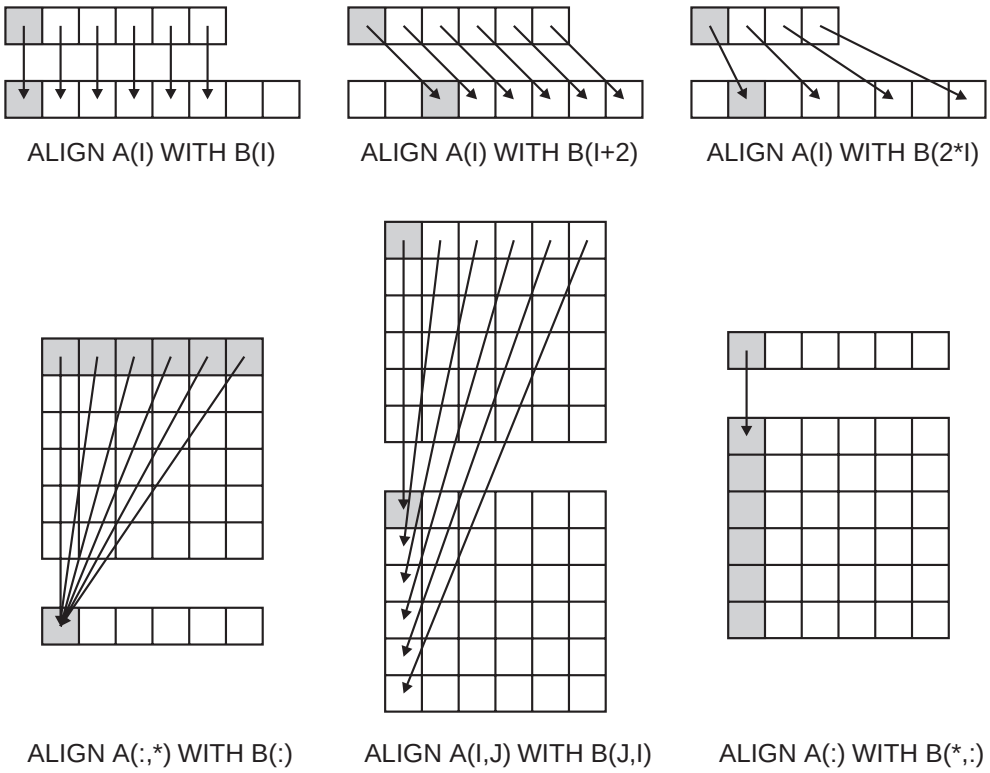
  ```
  real A(10)
  real B(10)

  !HPF$ ALIGN B(:) WITH A(:)
  ```

  Die `ALIGN` Direktive kann auch zur Realisierung komplexerer Kollokationsschemata verwendet werden. Abbildung 2.1 zeigt einige Beispiele hierfür.

- Auf der zweiten Stufe wird die Verteilung der Daten auf abstrakte Prozessoren durchgeführt. Dazu wird zunächst mit der `PROCESSORS` Direktive ein Array von virtuellen Prozessoren definiert. Zum Beispiel werden mit den folgenden Anweisungen zwei virtuelle Parallelrechner mit jeweils 64 Prozessoren festgelegt.

  ```
  !HPF$  PROCESSORS P(64)
  !HPF$  PROCESSORS Q(8,8)
  ```

Abbildung 2.1 Verwendung der `ALIGN` Direktive



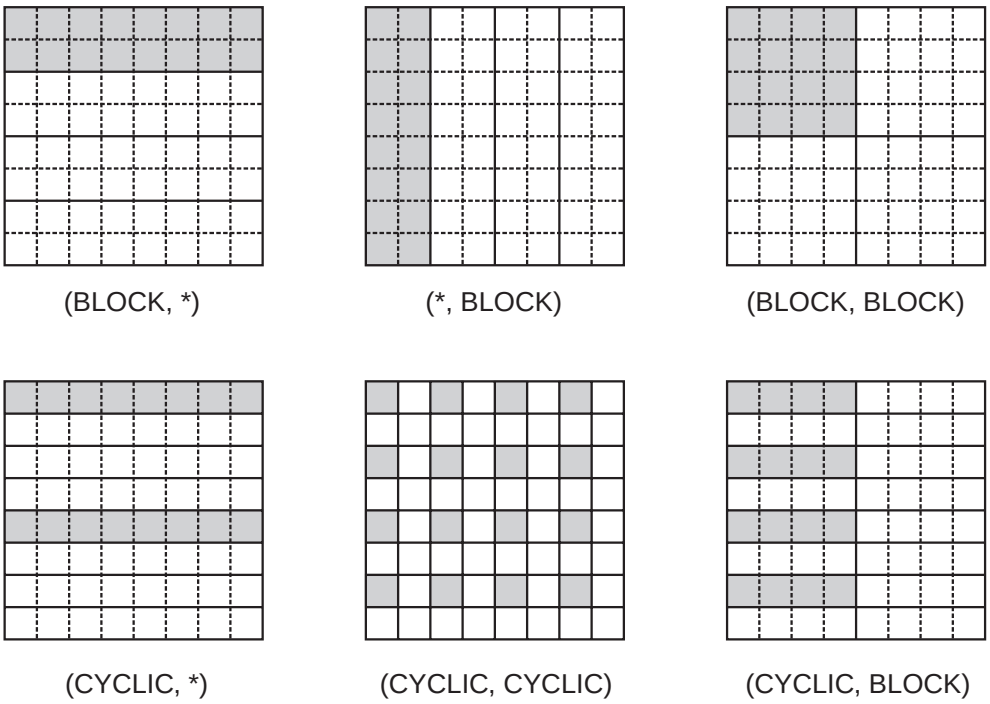
Mittels der `DISTRIBUTE` Direktive wird die eigentliche Partition der Datenstruktur und die Verteilung der Teilstrukturen auf die virtuellen Prozessoren vorgenommen. Für jede Dimension der betrachteten Array-Struktur werden die Elemente den virtuellen Prozessoren eines Prozessorarrays getrennt zugeordnet. Für die Zuordnung ist für jede Dimension des Arrays eines der folgenden Schemata anzugeben:

- `BLOCK(n)`: Blockweise Verteilung mit Blockgröße von n Elementen
- `CYCLIC(n)`: Zyklische Verteilung mit Verschiebung um n Elemente
- `*`: Keine Verteilung in dieser Dimension

In Abbildung 2.2 sind einige Kombinationen dieser Verteilungsschemata für ein Array der Größe 8x8 und 4 virtuellen Prozessoren gezeigt. Die dem ersten Prozessor zugeordneten Datenelemente sind grau dargestellt.

Die `DISTRIBUTE` Direktive beeinflusst nicht nur die Verteilung des explizit angegebenen Arrays, sondern auch aller anderen Arrays, die mittels der

Abbildung 2.2 Verwendung der `DISTRIBUTE` Direktive



`ALIGN` Direktive kolloziert wurden. Dies bedeutet, dass die `DISTRIBUTE` Direktive nicht auf Arrays angewendet werden kann, die bereits mit anderen Arrays kolloziert wurden. Das nachfolgende Programmfragment zeigt eine Beispiel für die Verwendung der `DISTRIBUTE` Direktive.

```
!HPF$  PROCESSORS P(64)
       real X(1024,1024)
!HPF$  DISTRIBUTE X(BLOCK,*) ONTO P
```

- Auf der dritten Stufe erfolgt die Zuordnung von virtuellen Prozessoren zu den real vorhandenen Prozessoren des verwendeten Parallelrechners. Die konkrete Abbildungsvorschrift ist implementierungsabhängig und für den Programmierer transparent.

HPF ist für die Bearbeitung von Arrays, also im Wesentlichen regulären Datenstrukturen ausgelegt. Die datenparallele Programmiersprache pC++ [15] erlaubt zusätzlich die Verwendung von komplexeren Datenstrukturen, wie etwa Mengen oder Bäume.

2.3.3 Explizite Zerlegung (Abstraktionsebene 3)

Das in diesem Abschnitt vorgestellte Programmiermodell erfordert die explizite Zerlegung eines Problems in parallel ausführbare Teilprobleme; die Zuordnung der Teilprobleme zu den vorhandenen Prozessoren und die Kommunikation zwischen ihnen ist hingegen implizit.

Das Bulk Synchronous Parallel (BSP) Programmiermodell

Das Bulk Synchronous Parallel (BSP) Programmiermodell [90] stellt eine Verallgemeinerung des in der Algorithmentheorie zur Beschreibung und Bewertung von parallelen Algorithmen verwendeten PRAM [46] Modells dar. Das BSP Programmiermodell wird mittels eines SPMD Ansatzes verwirklicht. Hierfür sind zurzeit Programmierbibliotheken für die Sprachen C und Fortran verfügbar. Eine besondere Eigenschaft von BSP ist, dass es über ein Kostenmaß für parallele Programme verfügt.

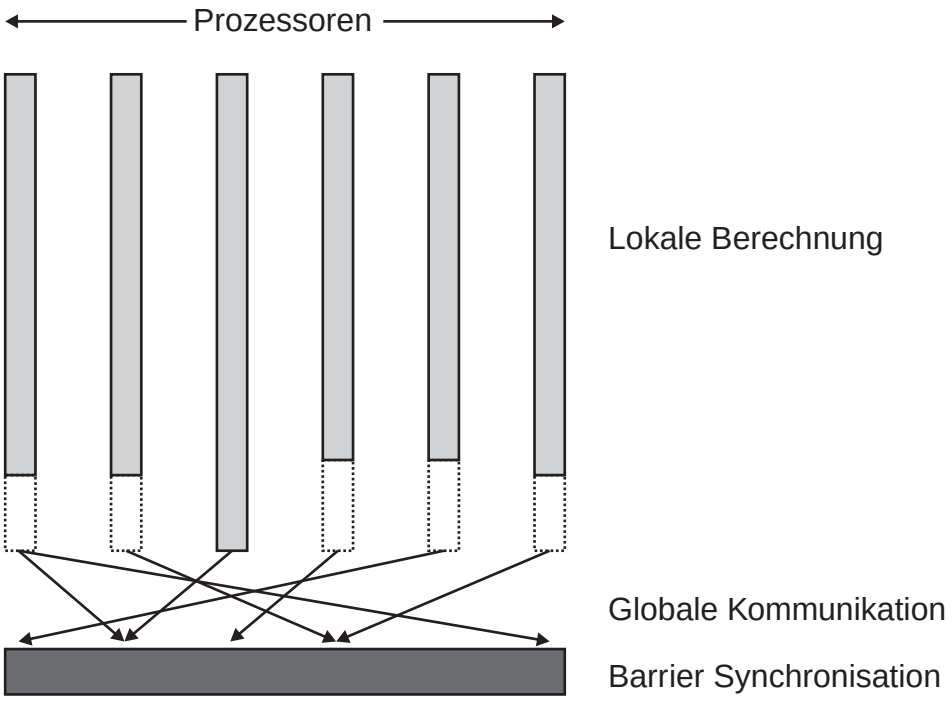
Als Grundlage des BSP Programmiermodells dient eine abstrakte Maschine, die aus p Prozessoren besteht, wobei jeder Prozessor über einen lokalen Speicher verfügt. Die Prozessor-Speicher Paare sind durch ein Netzwerk miteinander verbunden. Es handelt sich also um die Abstraktion eines Parallelrechners mit verteiltem Speicher und kann somit auf fast allen gängigen Parallelrechnerarchitekturen direkt umgesetzt oder einfach emuliert werden.

Eine elementare Rechenoperation wird im BSP Modell als *Step* bezeichnet, wobei ein Step nur auf Daten im lokalen Speicher zugreifen kann. Eine Berechnung im BSP Modell ist in mehrere so genannte *Supersteps* untergliedert. Abbildung 2.3 stellt den Ablauf eines Supersteps dar; dieser besteht aus den folgenden drei Phasen:

1. Ausführung einer Sequenz von Steps (auf lokalen Daten)
2. Nicht-blockierende, gepufferte Kommunikation zwischen den Prozessoren
3. Barrier-Synchronisation zwischen allen Prozessoren. Nach dieser Phase ist sichergestellt, dass alle Kommunikationsaktionen zwischen den Prozessoren abgeschlossen und die ausgetauschten Daten lokal verfügbar sind.

Zur Kommunikation zwischen den Prozessen wird das *Direct Remote Memory Access (DRMA)* Verfahren verwendet. Mittels der `put` und `get` Primitive können Datenworte zu und von einem entfernten Prozess kopiert werden. Die `put` und `get` Anweisungen können an jeder Stelle innerhalb der Phase 1 eines Supersteps verwendet werden. Die eigentlichen Transferaktionen werden aber erst

Abbildung 2.3 Ablauf eines BSP Supersteps



in der Phase 2 durchgeführt und deren Effekt ist mit Beendigung der Phase 3 wirksam. Durch diese Entkoppelung von Sender- und Empfängerprozess wird ermöglicht, dass die Zuordnung von Prozessen zu Prozessoren nicht explizit vom Programmierer festgelegt werden muss. Sie wird vom Laufzeitsystem nach einem randomisierten Verfahren vorgenommen. Durch das DRMA Verfahren sind Kommunikation und Synchronisation für den Programmierer transparent. BSP ist somit auf der Abstraktionsebene 3 anzusiedeln.

Eine besondere Eigenschaft des BSP Programmiermodells ist, dass es ein Kostenmodell ermöglicht, mit dem das Leistungverhalten eines BSP Programms für eine gegebene Architektur vorherbestimmt werden kann. Für das BSP Kostenmodell sind neben der Anzahl der Prozessoren p weitere Parameter relevant, mit denen die Leistung der verwendeten Prozessoren und des Kommunikationsnetzwerkes beschrieben wird. Im Einzelnen müssen die folgenden drei Parameter jeweils für den betrachteten Parallelrechner experimentell bestimmt werden:

- s: Prozessorgeschwindigkeit in Steps pro Sekunde
- l: Zeit für die Durchführung einer Barrier Synchronisation in Sekunden

- g: Transferrate des Netzwerkes in Datenworte pro Sekunde

Aus dem Programm müssen die folgenden beiden Werte für jeden Superstep i bestimmt werden:

- w_i: maximale Anzahl lokaler Steps, die von einem der Prozesse ausgeführt werden
- h_i: maximale Anzahl von Datenworten, die einer der Prozesse empfängt oder sendet

Die Zeit t_i, die der Superstep i maximal zur Ausführung benötigt, ergibt sich aus:

$$t_i = \frac{w_i}{s} + \frac{h_i}{g} + l$$

Die maximale Laufzeit t des Programms ist die Summe der Laufzeiten aller Supersteps:

$$t = \sum_i t_i$$

2.3.4 Explizite Zuordnung (Abstraktionsebene 2)

Bei den in diesem Abschnitt behandelten Programmiermodellen wird sowohl die Aufteilung des Problems in Teilprobleme, als auch deren Zuordnung zu den vorhandenen Prozessoren vom Programmierer explizit vorgenommen. Die Kommunikation läuft hingegen weitestgehend implizit ab, indem die Kommunikationspartner durch geeignete Konstrukte entkoppelt werden.

Tupelräume mit Linda

Bei Linda [1] handelt es sich um eine Koordinationssprache, mit deren Hilfe die Berechnungsaspekte von den Kommunikationsaspekten eines parallelen Programms getrennt werden können. Die Berechnung wird in einer herkömmlichen Programmiersprache, zum Beispiel C, beschrieben, während hierzu orthogonal die Kommunikation mit Linda-Primitiven realisiert wird. Das Programmiermodell von Linda stellt im Wesentlichen ein Speichermodell dar. In Linda sind somit Kommunikation und Synchronisation für den Programmierer transparent.

Alle weiteren Teilaspekte der Parallelisierung, wie etwa die Zuordnung von Teilproblemen zu den vorhandenen Prozessoren, sind von der verwendeten Programmiersprache und Systemumgebung abhängig. Das von Linda realisierte Programmiermodell befindet sich somit auf der Abstraktionsebene 2.

Das Programmiermodell von Linda besteht aus einem so genannten Tupelraum, der als Speicher und Kommunikationskanal dient, sowie einer Menge von Operationen zur Manipulation des Tupelraums. Daten können in Form von Tupeln in den Tupelraum eingefügt und Tupel können aus dem Tupelraum gelesen, bzw. entfernt werden. Neben den passiven Datentupeln sind in Linda zusätzlich aktive Prozesstupel vorgesehen, die sich in einem Auswertungsprozess befinden. Alle Prozesstupel werden parallel ausgeführt und kommunizieren über den Tupelraum durch Einfügen und Lesen von Datentupeln. Wenn die Ausführung eines Prozesstupels beendet ist, wird dieses zu einem gewöhnlichen Datentupel und kann aus dem Tupelraum von anderen Prozesstupeln gelesen oder entfernt werden. Ein Tupel besteht aus einer Sequenz von typbehafteten Werten, zum Beispiel

```
(‘‘sqrt’’, 16, 4).
```

Ein Muster ist ein spezielles Tupel, bei dem ein oder mehrere Einträge so genannte formale Parameter sind, zum Beispiel

```
(‘‘sqrt’’, 16, int ?X).
```

Ein Muster passt zu einem Tupel, wenn Tupel und Muster in den Werten punktweise übereinstimmen und der Typ aller formalen Parameter des Musters mit dem Typ der entsprechenden aktuellen Einträge des Tupels übereinstimmt. In diesem Fall werden die entsprechenden Einträge des Tupels den formalen Parameter des Musters zugewiesen.

Im Einzelnen stehen in Linda die folgenden Primitive zur Kommunikation mit dem Tupelraum zur Verfügung.

- `out(t)`
  Hiermit wird das Tupel t in den Tupelraum eingefügt. Der aufrufende Prozess kehrt unmittelbar nach dem Aufruf zurück und führt seine Berechnung weiter.

- `in(m)`
  Aus dem Tupelraum wird ein Tupel entfernt, das zum Muster m passt. Falls mehrere passende Tupel im Tupelraum vorhanden sind, wird zufällig eines ausgewählt. Wenn kein passendes Tupel im Tupelraum verfügbar ist,

blockiert der aufrufende Prozess. Die nicht-blockierende Variante dieser Operation ist `inp(m)`. Hier wird ein boolescher Wert zurückgeliefert, der angibt, ob ein passendes Tupel gefunden wurde.

- `rd(m)`
  Diese Operation unterscheidet sich von `in(m)` nur dadurch, dass das Tupel aus dem Tupelraum nicht entfernt wird. Das entsprechende nicht-blockierende Primitiv ist `rdp(m)`.

- `eval(t)`
  Diese Operation unterscheidet sich von `out(t)` dadurch, dass das Tupel t erst nach dem Einfügen in den Tupelraum ausgewertet wird. Die Auswertung erfolgt dabei in einem eigens erzeugten Prozess. Das Tupel t wird somit bis zum Ende des Auswertungsvorgangs zum Prozesstupel.

Mit der folgenden Anweisung wird das Tupel `(‘‘sqrt’’, 16, 4)` in den Tupelraum geschrieben

```
out(‘‘sqrt’’, 16, 4).
```

Es kann anschließend mit dem folgenden Aufruf wieder entfernt werden

```
in(‘‘sqrt’’, 16, int ?X)
```

In diesem Fall wird dem formalen Parameter X der Wert 4 zugewiesen.
Das folgende Beispiel zeigt die Verwendung von `eval`

```
eval(‘‘sqrt’’, 16, sqrt(16)).
```

Es wird als Prozesstupel in den Tupelraum eingefügt. Der erzeugte Prozess berechnet den Wert der Funktion `sqrt(16)`. Nach erfolgter Berechnung enthält der Tupelraum das Datentupel `(‘‘sqrt’’, 16, 4)`. Viele wichtige Datenstrukturen, Kommunikations- und Synchronisationskonstrukte lassen sich in dieser Tupel-Sprache codieren. In [20] sind hierzu einige Beispiele ausgeführt.

JavaSpaces

In jüngster Zeit wurde eine objektorientierte Erweiterung des Tupelraum-Programmiermodells von Linda für die Programmiersprache Java in Form der JavaSpaces Spezifikation [84] vorgestellt. Im Einzelnen sind gegenüber dem klassischen Linda Programmiermodell die folgenden Erweiterungen vorgenommen worden:

- Tupel werden in JavaSpaces als *Entries* bezeichnet. Zusätzlich zu ihren Einträgen sind Entries selbst typisiert, d.h. in JavaSpaces gibt es unterschiedliche Klassen von Tupeln.
- Entries und deren Einträge sind Objekte der Java Programmiersprache und können somit neben Daten auch Methoden enthalten.
- Als Muster können sowohl für Entries, als auch für deren Einträge Oberklassen eines Typs angegeben werden.
- JavaSpaces unterstützt die gleichzeitige Verwendung von mehreren Tupelräumen.
- Für die Entries wird das so genannte *Leasing-Konzept* eingeführt, d.h. nach Ablauf einer bestimmten Zeitdauer muss der Eintrag in den Tupelraum erneuert werden.

2.3.5 Explizite Kommunikation (Abstraktionsebene 1)

Die Programmiermodelle dieser Gruppe stellen einerseits explizite Kommunikationsprimitive zur Verfügung, vereinfachen andererseits aber die Synchronisation der Kommunikation mittels geeigneter implizit angewendeter Verfahren.

Aktive Nachrichten im Movie System

Beim Programmiermodell der aktiven Nachrichten besteht eine Nachricht sowohl aus einem Datenteil, als auch aus einem Codeteil, der ausgeführt wird, nachdem die Nachricht beim Empfänger angekommen ist. Eine aktive Nachricht verwandelt sich also auf Empfängerseite in einen Prozess. Dies bedeutet, dass beim Empfängerprozess keinerlei Synchronisation zum Empfang einer Nachricht notwendig ist.

Das Konzept der aktiven Nachrichten wurde zum Beispiel in Movie (Multitasking Object-Oriented Visual Environment) [33] verwirklicht. Ein Bestandteil von Movie ist die interpretierte Programmiersprache MovieScript, die im Wesentlichen eine Erweiterung von PostScript um C++ ähnliche, objektorientierte Sprachkonstrukte darstellt.

Die Knoten in einem Movie System kommunizieren miteinander, indem MovieScript Programme versendet werden. Auf diese Weise werden Berechnungsaktionen und Kommunikationsaktionen vereinheitlicht. Der Berechnungsaspekt wird realisiert, indem ein Computeserver MovieScript Programme ausführt. Der

Kommunikationsaspekt wird realisiert, wenn während der Ausführung ein MovieScript Programm einem anderen Computeserver zur entfernten Ausführung geschickt wird.

Mentat Programming Language (MPL)

MPL [36] ist eine Erweiterung der Programmiersprache C++ für paralleles Rechnen. Das Mentat System besteht aus zwei Komponenten: Ein Compiler, der MPL Programme in reguläre C++ Programme umwandelt und ein zugehöriges Laufzeitsystem, das in Form einer Bibliothek eine abstrakte Maschine für die Ausführung von MPL Programmen zur Verfügung stellt. Der Compiler fügt dem erzeugten C++ Code entsprechende Bibliotheksaufrufe hinzu.

MPL erweitert das Konzept der Kapselung von Daten und Methoden der Programmiersprache C++ um die für den Programmierer transparente Kapselung von Parallelität. Das Ausführungsmodell von MPL orientiert sich dabei an einem Datenflussmodell. Es wird ein Datenabhängigkeitsgraph aufgebaut, dessen Knoten Methodenaufrufe sind und dessen Kanten die Datenabhängigkeiten zwischen den Argument- und Resultatobjekten der Methodenaufrufe anzeigen.

Das folgenden Beispiel demonstriert die transparente Kapselung von Parallelität in MPL. Dabei sollen `A`, `B`, `C`, `D`, `E` und `R` Objekte einer Klasse zur Repräsentation von Vektoren darstellen und das Objekt `vector_operation` eine Instanz einer Klasse zur Ausführung von Operationen auf Vektoren, wie etwa Vektoraddition, sein.

```
A = vector_operation.add(B,C);
R = vector_operation.add(A, vector_operation.add(D,E))
```

Ob die Vektoradditionsmethode `add` sequentiell oder parallel implementiert wurde, ist für den Programmierer transparent. Diese Eigenschaft wird als *Kapselung von Intraobjekt-Parallelität* bezeichnet. Im Datenabhängigkeitsgraph kann also ein Knoten transparent durch einen Subgraph ersetzt werden. Da zwischen `vector_operation.add(B,C)` und `vector_operation.add(D,E)` keine Datenabhängigkeiten bestehen, kann auch die Ausführung dieser beiden Methodenaufrufe parallel erfolgen. In MPL wird die parallele Ausführung unabhängiger Methodenaufrufe ohne explizite Programmkonstrukte realisiert. Dies wird als *Kapselung von Interobjekt-Parallelität* bezeichnet. Im Folgenden wird genauer beschrieben, wie in MPL diese Kapselungseigenschaften erreicht werden. In Mentat wird der Aspekt der Kommunikation durch explizite Methodenaufrufe modelliert. Die Synchronisation auf der Empfängerseite (also die Ausführung

der aufgerufenen Methode) ist jedoch implizit. MPL ist somit auf Abstraktionsebene 1 einzuordnen.

Mentat-Klassen

Neben herkömmlichen C++ Klassen gibt es in MPL spezielle Mentat-Klassen. Instanzen von Mentat-Klassen werden entsprechend Mentat-Objekte genannt. Mentat-Objekte besitzen einen eigenen Adressraum, Kontrollfluss und einen systemweit gültigen Bezeichner. Das Schlüsselwort `mentat` kennzeichnet eine Mentat-Klasse. Durch Voranstellen eines der beiden Schlüsselworte `persistent` bzw. `regular` muss vom Programmierer spezifiziert werden, ob es sich um eine persistente oder reguläre Mentat-Klasse handelt. Persistente Mentat-Objekte erhalten ihren Zustand über mehrerer Methodenaufrufe hinweg. Reguläre Mentat-Objekte enthalten hingegen nur Methoden, die den Objektzustand nicht ändern. Somit können reguläre Objekte je nach der vorhandenen Parallelität vom System erzeugt und ein Objekt zur Ausführung eines Methodenaufrufs transparent ausgewählt werden.

Erzeugung von Mentat-Objekten

Die Instantiierung von Mentat-Objekten erfolgt in zwei Phasen. Zuerst wird ein Name für ein Mentat-Objekt erzeugt. Ein solcher Name kann sich in zwei verschiedenen Zuständen befinden. Im gebundenen Zustand ist mit dem Namen eine konkrete Instanz eines Mentat-Objekts assoziiert, während im ungebundenen Zustand kein konkretes Mentat-Objekt mit dem Namen verbunden ist. Bei der Erzeugung eines Namens ist dieser zunächst ungebunden. Bezeichnet der Name eine reguläre Mentat Klasse wird beim ersten Methodenaufruf eine entsprechende Instanz automatisch erzeugt. Zur Ausführung nachfolgender Methodenaufrufe können vom System zusätzlich neue Instanzen generiert werden. Instanzen persistenter Mentat Klassen müssen explizit mittels der `create` Methode eines Namensobjektes erzeugt werden. Dabei können zusätzlich Angaben zur Lokation des Objekts gemacht werden.

Methodenaufruf auf Mentat-Objekten

Methodenaufrufe auf gewöhnlichen C++-Objekten und auf Mentat-Objekten haben syntaktisch dieselbe Form. Sie unterscheiden sich aber hinsichtlich ihrer Semantik in zwei Aspekten:

- Da Mentat-Objekte über einen separaten Adressraum verfügen, erfolgt sowohl die Argumentübergabe als auch die Resultatrückgabe grundsätzlich mit Wertsemantik. Werden als Argumente, bzw. als Resultat Zeiger verwendet, so wird stets das durch den Zeiger referenzierte Objekt kopiert.
- Die Methodenaufrufe sind nicht-blockierend. Der Aufrufer setzt seine Berechnung parallel zur aufgerufenen Methode fort, bis der Wert des Methodenresultats explizit benötigt wird. Diese Eigenschaft wird durch die

spezielle return Anweisung `rtf` *(return to future)* realisiert. Das Resultat wird dann direkt zu den Mentat-Objekten geschickt, deren Ablauf von ihm abhängig sind. Dies bedeutet insbesondere, dass ein Methodenresultat nicht zwingend zum Aufrufer geschickt werden muss, wenn dieser zum Beispiel den Rückgabewert nur als Argument eines weiteren Methodenaufrufs auf einem anderen Mentat-Objekt verwendet.

2.3.6 Vollständig explizite Parallelisierung (Abstraktionsebene 0)

Die in diesem Abschnitt behandelten Programmiermodelle verlangen die explizite Kontrolle aller Aspekte der parallelen Programmierung durch den Programmierer.

Message Passing

Vor allem bei der Programmierung von Rechnern mit verteiltem Speicher hat das Message Passing Programmiermodell [59] eine wesentliche Bedeutung erlangt. Zum Beispiel ist PVM (Parallel Virtual Machine) [85] ein weitverbreitetes System zur Erstellung paralleler Anwendungen mit dem Message Passing Programmiermodell. Am Beispiel des aktuelleren Message Passing Interface (MPI) Standards [61, 66] soll die Grundfunktionalität von Message Passing Systemen behandelt werden. Der MPI Standard geht von einem bibliotheksbasierten Ansatz aus. Im MPI Standard werden Syntax und Semantik für eine Message Passing Programmierbibliothek definiert, die effizient auf einer Vielzahl unterschiedlicher Rechnertypen implementiert werden kann. Zu beachten ist aber, dass der MPI Standard keine vollständige Softwareinfrastruktur zur Erstellung von verteilten parallelen Programmen definiert. Insbesondere fehlen zum Beispiel Prozessmanagement und Unterstützung von Ein/Ausgabe Möglichkeiten.

Im Programmiermodell von MPI besteht eine Berechnung aus einer Menge von kooperierenden Prozessen, die in einer herkömmlichen sequentiellen Programmiersprache geschriebene Programme ausführen. Dies sind üblicherweise C/C++ oder Fortran Programme. Die Programme enthalten entsprechende Aufrufe von MPI Bibliotheksfunktionen. Grundsätzlich können die einzelnen Prozesse verschiedene Programme ausführen; MPI verwirklicht also das MPMD Programmiermodell. Gewöhnlich wird am Beginn einer MPI Berechnung eine feste Anzahl von Prozessen erzeugt, so dass auf jedem der vorhandenen Prozessoren ein Prozess zur Ausführung kommt. Jeder Prozess wird durch eine vom System bestimmte eindeutige ID bezeichnet, die durch einen Integer-Wert realisiert wird.

In MPI sind verschiedene Kommunikationsmodelle vorgesehen. Die einfachste Form der Kommunikation ist die *Punkt zu Punkt Kommunikation*. Sie wird verwendet um zwischen zwei bestimmten Prozessen Nachrichten auszutauschen. Zusätzlich kann auch *asynchrone Kommunikation* realisiert werden. Mehrere Prozesse können zu einer Gruppe zusammengefasst werden. Dabei können Prozesse einer Gruppe sich so genannter *Gruppenkommunikation* bedienen, um globale Operationen innerhalb einer Gruppe zu verwirklichen, wie etwa das Verschicken von Broadcast Nachrichten.

Das von MPI realisierte Message Passing Programmiermodell reflektiert direkt die Funktionalität von Parallelrechnerarchitekturen mit verteiltem Speicher. Alle Aspekte der parallelen Programmierung für nachrichtenorientierte Architekturen werden hier auf die Programmebene exponiert. Message Passing ist demnach auf der Abstraktionsebene 0 angesiedelt.

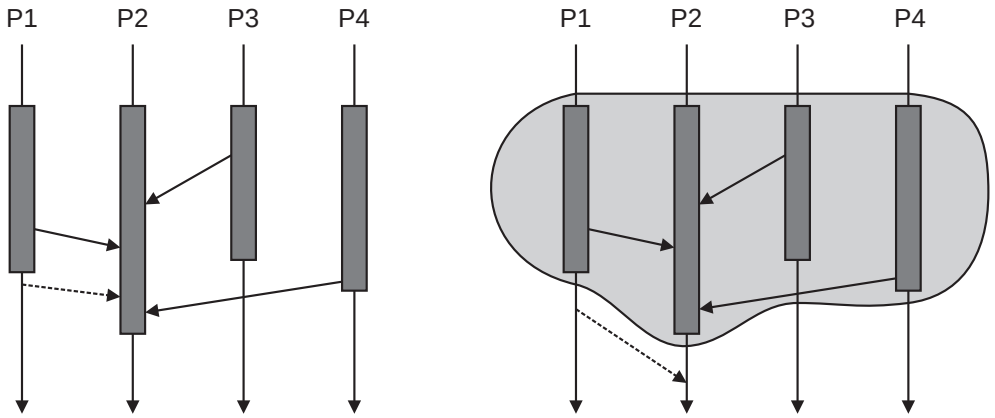
Insgesamt werden im MPI Standard weit über hundert Funktionen definiert. Für die Realisierung einer Vielzahl von Anwendungen reicht aber eine wesentlich kleinere Menge von MPI Funktionen aus. Diese werden in den nachfolgenden Unterabschnitten kurz beschrieben. Zuvor wird jedoch die Unterstützung von modularer Programmierung in MPI behandelt.

Unterstützung modularer Programmierung in MPI

Durch so genannte *Kommunikatoren* wird in MPI eine modulare Programmierweise ermöglicht. Die interne Kommunikation eines MPI basierten Programmmoduls (z.B. die Kommunikation innerhalb einer MPI basierten Programmbibliothek) wird mittels Kommunikatoren gekapselt. Somit kann die interne MPI Kommunikation des Moduls nicht mit der MPI Kommunikation des Programms, das das Modul verwendet, in Konflikt geraten. Jede MPI Funktion, die in irgendeiner Form Kommunikation zwischen Prozessen realisiert, verlangt zwingend als Argument die Angabe eines Kommunikators. Der Kommunikator beschreibt den *Kontext*, in dem Nachrichten ausgetauscht werden. Eine Empfangsfunktion kann nur eine Nachricht empfangen, wenn diese im selben Kontext verschickt wurde. Zur dynamischen Erzeugung eines neuen Kontextes stellt MPI die Funktion `MPI_COMM_DUP` zur Verfügung. Sie verlangt als Argument einen Kommunikator und liefert einen weiteren Kommunikator als Resultat zurück, der dieselbe Gruppe von Prozessen bezeichnet, aber einen neuen Kontext aufspannt.

In Abbildung 2.4 wird der oben beschriebene Sachverhalt nochmals verdeutlicht. Jede der vier vertikalen Linien stellt den zeitlichen Ablauf des Kontrollflusses eines Prozesses innerhalb einer Berechnung dar. Zur Vereinfachung wird hier angenommen, dass jeder Prozess dasselbe Programm ausführt; es handelt sich also um eine Berechnung im SPMD Modell. Der Austausch von Nachrichten wird durch Pfeile symbolisiert. Die graue Hinterlegung der Kontrollflusslinie deutet an, dass sich der Prozess gerade in einer Bibliotheksfunktion befindet,

Abbildung 2.4 Verwendung von Kommunikatoren zur Modularen Programmierung



die auch mit MPI realisiert sein soll. Die Ausführung der Bibliotheksfunktion soll in diesem Beispiel in den einzelnen Prozessen unterschiedlich lange dauern. In dieser Situation kann ohne die Verwendung von Kontexten Kommunikation außerhalb der Bibliothek mit der Kommunikation innerhalb der Bibliothek in Konflikt geraten, wie in der linken Abbildung dargestellt (gestrichelt dargestellter Nachrichtenaustausch). Das rechte Bild zeigt, wie dieses Problem durch die Erzeugung eines neuen Kontextes für die Kommunikation innerhalb der Bibliotheksfunktion gelöst wird.

Punkt zu Punkt Kommunikation

Zur Realisierung der grundlegenden Punkt zu Punkt Kommunikation werden in MPI die folgenden Primitive definiert:

- `MPI_Init`
  Initialisierung einer MPI Berechnung. Diese Funktion muss zu Beginn einer Berechnung (bevor andere MPI Funktionen verwendet werden) genau einmal aufgerufen werden.

- `MPI_Finalize`
  Beenden einer MPI Berechnung. Nach Aufruf dieser Funktion kann kein weiterer Aufruf einer MPI Funktion erfolgen.

- `MPI_Comm_Size`
  Bestimmung der Anzahl der an der Berechnung beteiligten Prozesse

- `MPI_Comm_Rank`
  Bestimmung der ID des eigenen Prozesses.

- `MPI_Send`
  Versenden einer Nachricht. Als Argumente sind neben den Daten, die verschickt werden sollen (Datenpuffer, Länge, und Datentyp), die ID des Empfängers und ein so genanntes *Message Tag* anzugeben.

- `MPI_Recv`
  Empfangen einer Nachricht. Hier ist neben dem Empfangspuffer die Senderadresse und ein Message Tag anzugeben. Es werden nur solche Nachrichten betrachtet, die zur angegebenen Absenderadresse und zum Message Tag passen. Ist noch keine entsprechende Nachricht vorhanden, blockiert der Aufruf.

- `MPI_IProbe`
  Test ob eine bestimmte Nachricht vorliegt. Als Argumente werden die ID des Senderprozesses und ein Message Tag angegeben. Der Aufruf liefert einen booleschen Wert zurück, der angibt ob eine passende Nachricht vorliegt. Dieser MPI Bibliotheksaufruf dient zur Verwirklichung asynchroner Kommunikation.

Globale Operationen

Die grundlegenden Funktionen für die globale Kommunikation innerhalb einer Gruppe von Prozessen lassen sich in Operationen zur globalen Synchronisation (`MPI_Barrier`), Operationen zum kollektiven Datentransfer (`MPI_Bcast`, `MPI_Gather`, `MPI_Scatter`) und Reduktionsoperationen (`MPI_Reduce`) einteilen.

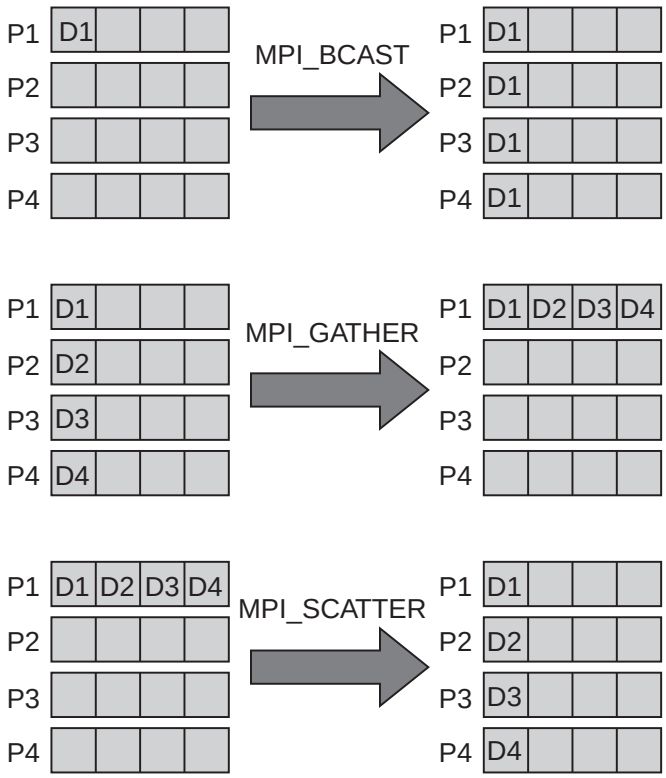
- `MPI_Barrier`
  Synchronisation der Ausführung aller Prozesse einer Gruppe. Kein Prozess der Gruppe kehrt aus dieser Funktion zurück, bevor nicht alle Prozesse der Gruppe sie aufgerufen haben.

- `MPI_Bcast`
  Verschicken von Broadcast Nachrichten. Ein ausgezeichneter Prozess verschickt eine Nachricht an alle Mitglieder der Gruppe.

- `MPI_Gather`
  Einsammeln von Daten. Alle beteiligten Prozesse senden Daten zu einem ausgezeichneten Prozess, bei dem diese konsekutiv in einem Pufferspeicher abgelegt werden.

- `MPI_Scatter`
  Verteilen von Daten. Von einem ausgezeichneten Prozess aus werden Daten aus einem Pufferspeicher zu allen Prozessen der Gruppe übertragen,

Abbildung 2.5 Die Operationen zum kollektivem Datentransfer von MPI



wobei der Prozess mit der ID i die Daten an der i-ten Position des Pufferspeichers erhält.

- `MPI_All_Reduce`
  Auf Daten, die im Eingabepuffer aller Prozesse abgelegt sind, wird punktweise eine anzugebende Operation angewendet und das Ergebnis in die angegebenen Ausgabepuffer aller Prozesse der Gruppe geschrieben.

In Abbildung 2.5 sind die oben beschriebenen Operationen zum kollektivem Datentransfer von MPI für eine Gruppe von 4 Prozessen skizziert. In der Abbildung entspricht eine Zeile jeweils einem der 4 Prozesse, wobei jeder Prozess 4 Speicherplätze für Daten enthält.

Objektorientiertes Message Passing mit TPO++

MPI bietet zwar grundsätzlich eine Anbindung an die Programmiersprache C++, hierbei wird aber der Aspekt der engen Integration der objektorientierten Kon-

zepte von C++ nicht erfasst. TPO++ (Tübingen Parallel Objects) [37] erweitert MPI um für den Programmierer einfache und typsichere Übertragungsmechanismen für Objekte, sowie um die effiziente Integration der C++ Standard Template Library.

Übertragung vordefinierter C++ Typen

Nach der Initialisierung mittels des Aufrufs `TPO::init` steht mit dem globalen Objekt `CommWorld` eine objektorientierte Abstraktion eines MPI Kommunikators zur Verfügung, der alle beteiligten Knoten umfasst. Einfache Datentypen können dann folgendermaßen übertragen werden:

```
double d
CommWorld.send(d, dest_rank)
```

Für die Verarbeitung von Containern werden zwei Iteratoren angegeben, die den zu übertragenden Datenbereich markieren.

```
vector<double> vd
CommWorld.send(vd.begin(), vd.end(), dest_rank)
```

Übertragung benutzerdefinierter C++ Typen

Bei der Übertragung benutzerdefinierter C++ Typen wird zwischen Typen mit trivialem Copy-Konstruktor und komplexen Typen (d.h. ohne trivialem Copy-Konstruktor) unterschieden. Typen mit trivialem Copy-Konstruktor müssen vor der Übertragung mit `TPO_TRIVIAL` registriert werden. Es wird dann automatisch der von Objekten dieses Typs belegte Speicherbereich übertragen. Bei komplexen Typen wird vom Programmierer zusätzlicher Programmcode angegeben, der die Konversion der Komplexen Struktur in eine lineare Struktur und umgekehrt vornimmt. Aus Effizienzgründen wird die Datenkonversion zwischen unterschiedlichen Datenformaten nicht unterstützt.

3

Kapitel 3

Die Systemumgebung DOTS

3.1 Überblick

In diesem Kapitel soll die Systemumgebung DOTS *(Distributed Object-Oriented Threads System)* vorgestellt werden. Mit DOTS steht eine Programmierumgebung zur Erstellung von verteilten parallelen Applikationen in C++ zur Verfügung. Das System bietet dem Softwareentwickler verschiedene Programmiermodelle zur Entwicklung paralleler Programme. Das zentrale Programmiermodell von DOTS ist das Modell der verteilten Kontrollflüsse (Threads). Das System stellt jedoch noch weitere Modelle zur Verfügung, die das Multithreading Modell ergänzen und erweitern.

Das Programmiermodell der verteilten Threads verwirklicht einen funktionalen Programmierstil zur Realisierung von taskparallelen C++ Programmen. Es eignet sich in besonderer Weise für die Erstellung von verteilten Anwendungen mit kompakten Datenstrukturen. Diese können in strukturierter Weise als Parameter und Resultate von Threads transparent kommuniziert werden.

DOTS bietet den Vorteil, dass ein Rechnernetzwerk aus vollkommen unterschiedlichen Rechnersystemen als homogener Pool von virtuellen Prozessoren für verteilte parallele Berechnungen genutzt werden kann. Die DOTS Systemplattform ist für eine breite Palette von Rechner- und Betriebssystemen – vom Echtzeitkern bis hin zum Mainframe Cluster – verfügbar. Im Einzelnen wurde DOTS bereits auf den folgenden Systemen erfolgreich eingesetzt:

- Microsoft Windows 95/98/NT/2000
- Linux, FreeBSD

- Solaris, AIX, IRIX
- IBM Parallel Sysplex Cluster unter OS/390 [1]
- QNX Realtime Platform

Wie in Abschnitt 1.5 dargestellt wird, ist die Unterstützung von Heterogenität eine zentrale Anforderung für die Verwendbarkeit einer Systemumgebung in zukünftigen Infrastrukturen für das parallele Rechnen, wie etwa den Computational Grids.

DOTS ist als rein bibliotheksbasiertes System realisiert, es sind somit keine speziellen Compiler oder weitere Tools erforderlich, um verteilte parallele DOTS Applikationen zu erstellen und auszuführen. Dies hat den Vorteil, dass vorhandene Entwicklungsumgebungen und -werkzeuge, wie etwa Debugger weiterhin sinnvoll eingesetzt werden können. Die Systemplattform besteht aus einer Reihe von Programmierschnittstellen und einem Laufzeitsystem die nachfolgend beschrieben werden.

Dieses Kapitel ist in zwei Teile gegliedert. Im ersten Teil werden die einzelnen Programmiermodelle von DOTS ausführlich beschrieben und diskutiert. Der zweite Teil umfasst die Beschreibung der grundlegenden Systemarchitektur von DOTS, sowie die Behandlung der Lastverteilungskomponente und die Diskussion eines XML basierten Ansatzes zur Beschreibung und Auswertung von parallelen verteilten Berechnungen.

3.2 Die Programmiermodelle von DOTS

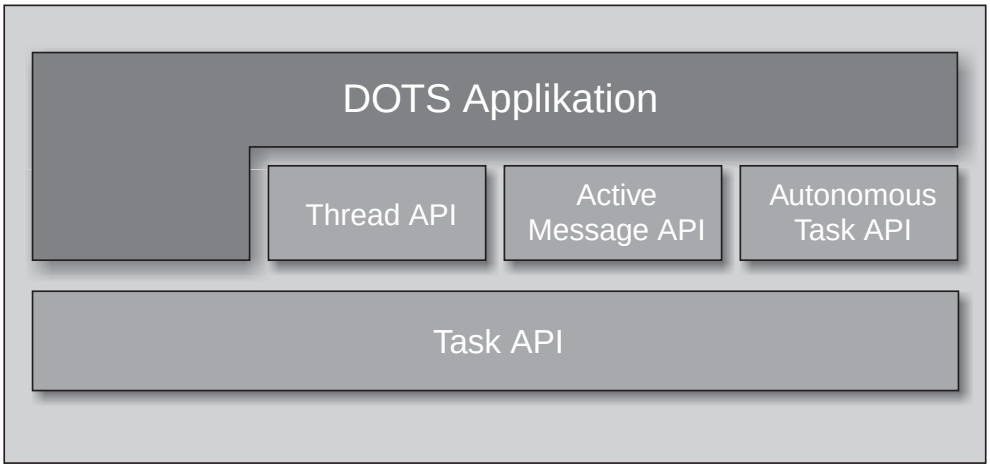
DOTS stellt dem Anwendungsentwickler für jedes unterstützte Programmiermodell eine separate Programmierschnittstelle (*Application Programming Interface, API)* zur Verfügung. Die einzelnen APIs sind in einer Schichtstruktur angeordnet (siehe Abbildung 3.1). Bei der Entwicklung von DOTS Applikationen kann gleichzeitig auf unterschiedliche APIs zurückgegriffen werden. In den folgenden Abschnitten werden die einzelnen APIs von DOTS und die durch sie realisierten Programmiermodelle detailliert beschrieben.

3.2.1 Task API

Das Task API stellt die allgemeinste und damit flexibelste Programmierschnittstelle von DOTS dar. Sie dient gleichzeitig auch als Grundlage der Implementie-

[1] Cluster bestehend aus IBM S/390 Mainframe Systemen

Abbildung 3.1 Die Programmierschnittstellen von DOTS



rung aller anderen APIs des Systems. Mit dem Task API wird ein rein objektorientiertes Programmiermodell realisiert.

Die grundlegende Ablaufeinheit dieser Programmierschnittstelle stellen DOTS Tasks dar. Hierbei handelt es sich um Objekte anwendungsspezifischer Klassen, die von der Klasse `DOTS_Task` abgeleitet sind und eine oder mehrere der folgenden virtuellen Methoden der Basisklasse redefinieren:

- `void source(void)`
- `void run(void)`
- `void drain(void)`

Mit der Methode `DOTS_Task::go(void)` wird ein Task-Objekt in ein aktives Objekt umgewandelt. Die oben aufgelisteten Methoden werden dann in der angegebenen Reihenfolge jeweils in einem eigenen Thread ausgeführt, sofern für sie anwendungsspezifischer Code angegeben wurde. Die Methode `source()` wird unmittelbar nachdem das Objekt aktiviert wurde ausgeführt. Die Ausführung der Methode `drain()` schließt sich unmittelbar der Ausführung von `run()` an.

Für diese drei Methoden kann mittels

- `DOTS_Task::set_source_node(DOTS_Node_ID)`
- `DOTS_Task::set_run_node(DOTS_Node_ID)`
- `DOTS_Task::set_drain_node(DOTS_Node_ID)`

explizit der Knoten angegeben werden, auf dem sie ausgeführt werden sollen. Durch diese Vorgehensweise kann ein Task-Objekt auf einem bestimmten Knoten des Systems initialisiert werden und nach der Ausführung der `run()` Methode das Ergebnis der Berechnung auf einem bestimmten Knoten abliefern. Bei der Methode `source()` handelt es sich somit um einen *lokationsbasierten Konstruktor*, entsprechend stellt `drain()` einen *lokationsbasierten Destruktor* für ein Task-Objekt dar. Innerhalb der `run()` Methode wird der eigentliche Berechnungsvorgang codiert.

Wurde für die Methode `run()` nicht explizit ein Ausführungsort angegeben, wird transparent von der Lastverteilungskomponente des Systems der Ausführungsort- und Zeitpunkt bestimmt (siehe Abschnitt 3.4). In Abbildung 3.2 ist das Ausführungsmodell für DOTS Task-Objekte nochmals grafisch dargestellt.

DOTS Tasks können zur einheitlichen Verwaltung von Berechnungsvorgängen in Gruppen zusammengefasst werden. Dazu wird für jede Gruppe mit der API Funktion `dots_new_gid(void)` zunächst eine Gruppen ID (GID) erzeugt, die zu deren systemweit eindeutigen Identifikation dient. Mit `DOTS_Task::set_gid()` kann für ein Task-Objekt die Zugehörigkeit zu einer bestimmten Gruppe durch Angabe einer GID als Argument festgelegt werden.

Bei Task-Objekten, die derselben Gruppe angehören, wird vom System sichergestellt, dass die lokationsbasierten Konstruktoren und Destruktoren auf einem Knoten immer sequentialisiert ablaufen. Wird zum Beispiel für zwei oder mehrere Task-Objekte einer Gruppe derselbe Ausführungsort für die `source()` Methode festgelegt, stellt das System sicher, dass sich dort zu keinem Zeitpunkt mehr als ein Task-Objekt der Gruppe in der Ausführung seiner `source()` Methode befindet. Eine weitere Anwendung des Gruppenkonzeptes wird weiter unten vorgestellt.

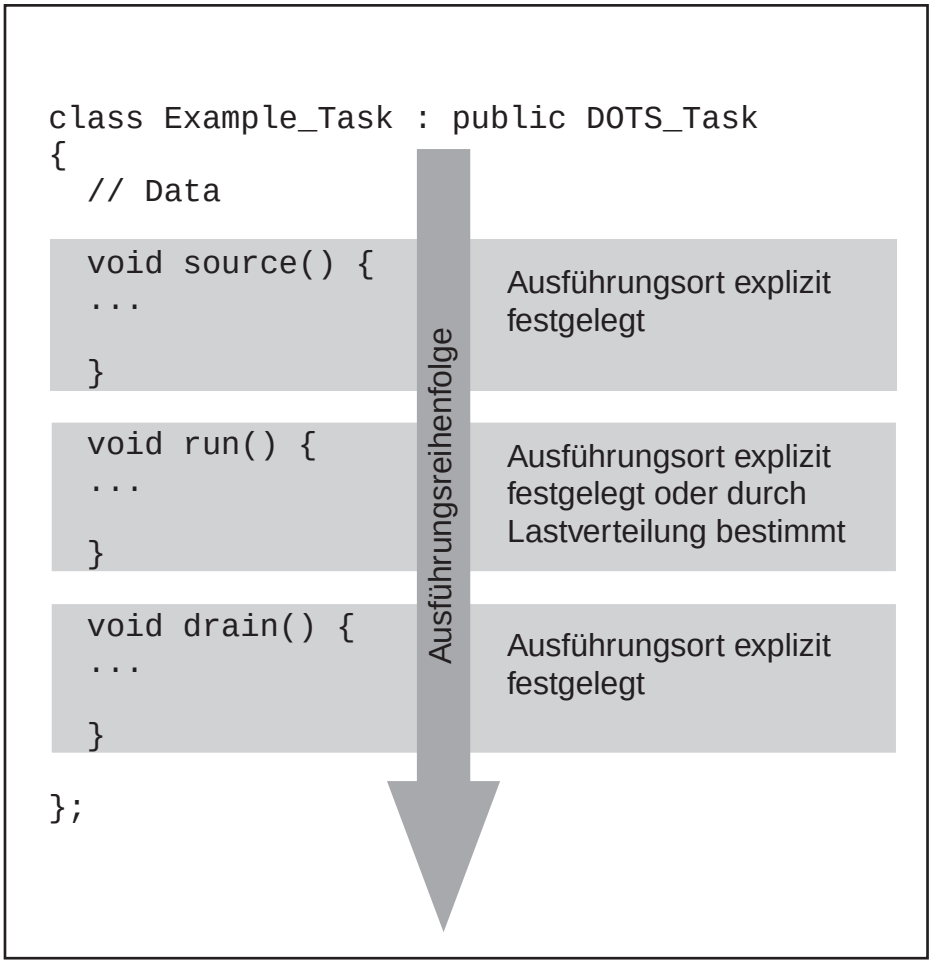
Migration von Task-Objekten

Die Basisklasse `DOTS_Task` stellt den von ihr abgeleiteten Klassen die Methode `migrate(DOTS_Node_ID)` zur Verfügung, mit der ein vom Programmierer kontrollierter Migrationsprozess für Task-Objekte realisiert werden kann. In der `run()` Methode eines Task-Objekts kann durch einen Aufruf von `migrate` die Migration zu einem anzugebenden Knoten veranlasst werden. Die Verwendung von `migrate()` ist nur möglich, wenn beim betreffenden Task-Objekt die Ausführung der `run()` Methode nicht vom Lastverteilungssystem kontrolliert wird.

Für die Realisierung von Objektmigration gibt es zwei unterschiedliche Ansätze, die sich bezüglich ihrer Transparenz und Effizienz unterscheiden.

- Bei der starken Migration *(Strong Migration)* ist es möglich, zu jedem

Abbildung 3.2 Die Ausführungsmodell für DOTS Tasks



beliebigen Zeitpunkt der Ausführung ein Objekt transparent auf einen anderen Knoten zu übertragen. Dieses wird dort an derjenigen Stelle im Programmcode weiter ausgeführt, die vor dem Migrationsprozess erreicht war. Dazu muss der vollständige Objektzustand, also inklusive des Programmstacks übertragen werden. Dieses Verfahren ist aufgrund seiner Transparenz für den Programmierer sehr komfortabel. Es müssen aber je nach dem angestrebten Grad an Transparenz umfangreiche Informationen über den Prozesszustand übertragen werden. Im Falle eines heterogenen Rechnernetzwerkes sind außerdem komplexe Konversionsaktionen auf den Zustandsdaten erforderlich.

- Beim schwachen Migrationskonzept *(Weak Migration)* kann der Migrati-

onsprozess zwar an einer beliebigen Stelle der Programmausführung gestartet werden, auf dem neuen Knoten wird jedoch die Ausführung an einer vordefinierten Stelle fortgesetzt. Bei diesem Verfahren muss also nur der Objektzustand bezüglich der Daten, nicht aber der Objektzustand bezüglich der Ausführung übertragen werden. Die Migration kann somit effizient realisiert werden, sie läuft aber für den Programmierer nicht transparent ab.

Um eine effiziente Migration von Objekten in heterogenen Rechnernetzen zu ermöglichen, wurde das schwache Migrationskonzept in DOTS integriert. Nachdem der Übertragungsprozess des Task-Objekts auf dem neuen Knoten abgeschlossen ist, wird die Methode `run()` erneut von Beginn an ausgeführt. Im Programmcode kann dann explizit eine Untersuchung des Objektzustandes vorgenommen werden, die den Ausführungsprozess der `run()` Methode wieder in den korrekten Zustand versetzt.

Abbruch der Ausführung von Task-Objekten

Mittels der Methode `DOTS_Task::abort()` kann die Ausführung der `run()` Methode eines Task-Objekts an einer beliebigen Stelle abgebrochen werden. Es wird dann unmittelbar mit der Ausführung der Methode `drain()` auf dem als Ausführungsort spezifizierten Knoten begonnen, falls diese in der entsprechenden Klasse des Task-Objekts redefiniert wurde.

Mit den beiden API Funktionen

- `dots_suspend_exec(DOTS_Task_Group_ID)`
- `dots_continue_exec(DOTS_Task_Group_ID)`

kann für eine Gruppe von Task-Objekten deren Ausführung zeitweise aufgehoben werden. Dies bedeutet, dass die `run()` Methode aller Task-Objekte der angegebenen Gruppe nicht mehr zur Ausführung kommt.

Um bei Task-Objekten, deren `run()` Methode gerade ausgeführt wird, festzustellen, ob die Ausführung der eigenen Task Gruppe zurzeit aufgehoben wurde, kann das Flag `DOTS_Task::is_canceled` abgefragt werden. Ist dies der Fall, wird mittels `DOTS_Task::abort()` die Ausführung von `run()` explizit beendet.

Diskussion des vom Task API realisierten Programmiermodells

Das Programmiermodell der DOTS Tasks stellt einen sehr allgemeinen, objektorientierten Ansatz zur taskparallelen Programmierung dar. Wie in den nachfolgenden Abschnitten gezeigt wird, kann das DOTS Task Modell als Grundlage

der effizienten Realisierung einer Reihe weiterer, vollkommen unterschiedlicher Programmiermodelle dienen.

Da der Ausführungsort der `run()` Methode sowohl transparent von der Lastverteilung, als auch explizit vom Programmierer bestimmt werden kann, ist in diesem Modell die Erstellung von Programmen auf unterschiedlichen Abstraktionsebenen möglich. Das Konzept der lokationsbasierten Konstruktoren erweitert die Grundidee von Konstruktoren und Destruktoren der sequentiellen objektorientierten Programmiersprachen für verteilte Systeme. Es ermöglicht die Initialisierung bzw. die Finalisierung eines Tasks auf bestimmten Knoten eines Systems, die zum Beispiel eine ausgezeichnete Funktionalität aufweisen. Dabei spielt keine Rolle, auf welchem Knoten das Task Objekt erzeugt wurde.

Ein Beispiel hierfür wäre eine Anwendung, die einen Web basierten Zugriff auf eine Berechnung (z.B. die Generierung von dreidimensionalen Landschaftsbildern aus einer Geodatenbank) in einem Cluster ermöglicht. Als Ausführungsort für die `source()` und die `drain()` Methode wird jeweils der Web-Server bestimmt. In der `source()` Methode wird der Auftrag vom Web-Server entgegen genommen, die `drain()` Methode liefert das berechnete Ergebnis dem Web-Server zurück. Die Berechnung des Auftrags erfolgt innerhalb der `run()` Methode, deren Ausführungsort transparent von der Lastverteilungsstrategie bestimmt wird. Diese Vorgehensweise erlaubt auch, dass mehrere Web-Server zur Entgegennahme von Aufträgen eingesetzt werden, die Berechnung der Aufträge aber auf einem gemeinsam genutzten Cluster erfolgt.

Das Ausführungsmodell von DOTS Tasks könnte um das folgende *Kolokationskonzept* erweitert werden. Dem Programmierer wird die Möglichkeit gegeben, den Ausführungsort einer Methode eines DOTS Task-Objekts mit dem vom Lastverteilungssystem bestimmten Ausführungsort der `run()` Methode eines anderen Task-Objekts zu kolozieren. Hiermit könnten zum Beispiel die auf den DOTS Tasks basierten Programmiermodelle enger miteinander verknüpft werden. Es wäre etwa möglich, einem DOTS Thread eine aktive Nachricht zu schicken.

3.2.2 Thread API

In diesem Abschnitt soll das zentrale Programmiermodell von DOTS vorgestellt werden. Dazu wird zunächst eine formale Beschreibung des Modells gegeben, eine ausführliche Diskussion der Primitive des Thread API schließt sich an.

Das Threads Programmiermodell

Zur Beschreibung von Berechnungen mit mehreren (verteilten) Kontrollflüssen (Threads) wird in [13, 14] das im Folgenden beschriebene graphentheoreti-

sche Modell vorgestellt. Das Modell wurde als Grundlage für die Diskussion von Scheduling Verfahren eingeführt. Es eignet sich aber darüber hinaus auch für die allgemeine Beschreibung und Klassifizierung von Berechnungen mit mehreren Kontrollflüssen.

In diesem Modell wird eine Berechnung mit mehreren (verteilten) Kontrollflüssen durch einen gerichteten Graphen, den so genannten *Ablaufgraphen* beschrieben. Eine Berechnung besteht dabei aus einer Menge von *Threads*, welche die grundlegende Strukturierungseinheit des Modells darstellen. Jeder Thread setzt sich wiederum aus einer endlichen Folge von elementaren Arbeitsschritten, den so genannten *Tasks*, zusammen. Die Tasks aller Threads bilden die Knotenmenge des Ablaufgraphen. Alle Tasks eines Threads sind durch so genannte *Fortsetzungskanten* verbunden und müssen in der durch diese Kanten induzierten Reihenfolge sequentiell abgearbeitet werden. Die Ausführung eines Threads endet mit der Ausführung seines letzten Tasks, also dem Task des Threads, von dem keine Fortsetzungskante ausgeht.

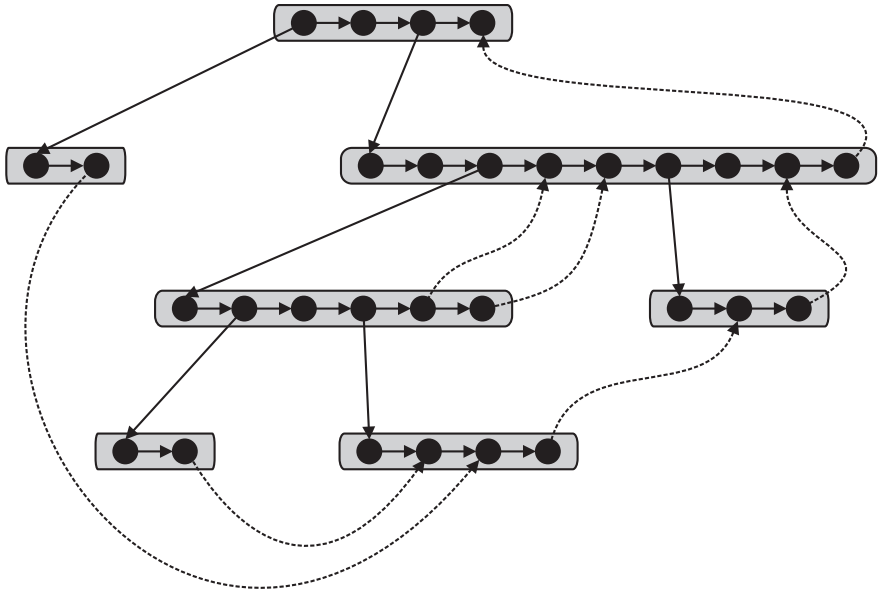
Ein Thread kann während seiner Ausführung neue Threads, so genannte *Kind*-Threads erzeugen. Der erzeugende Thread wird als *Vater*-Thread bezeichnet. Diese Erzeugungsbeziehung wird durch eine weitere Kantenart, den *Erzeugungskanten*, im Ablaufgraphen dargestellt. Eine Erzeugungskante führt von einem speziellen Erzeuger-Task des Vater-Threads zum ersten Task des Kind-Threads.

Mit diesen Definitionen lässt sich innerhalb des Ablaufgraphs eine zusätzliche beschreibende Baumstruktur definieren, die *Erzeugungsbaum* oder auch *Aktivierungsbaum* genannt wird. Der Erzeugungsbaum wird aus den Threads als Knotenmenge und den Erzeugungskanten des Ablaufgraphen gebildet. Die Wurzel des Erzeugungsbaumes wird als *Root*-Thread bezeichnet.

Zwischen Tasks verschiedener Threads kann eine Produzent-Konsument Beziehung bezüglich Daten auftreten. Diese Daten werden als *Resultate* eines Threads bezeichnet, wobei ein Thread im Laufe seiner Ausführung mehrere Resultate liefern kann. Eine derartige Beziehung zwischen Threads wird durch *Datenabhängigkeitskanten* angezeigt. Eine solche Kante geht von dem Task aus, der ein Resultat produziert und führt zu dem Task, der das Resultat konsumiert. Datenabhängigkeitskanten zeigen explizite Synchronisationsstellen der nebenläufigen Ausführung der Threads einer Berechnung an. Mit der Ausführung des konsumierenden Tasks (und damit des entsprechenden Threads) muss solange gewartet werden, bis der produzierende Task ausgeführt wurde und somit das entsprechende Resultat vorliegt.

Abbildung 3.3 zeigt den Ablaufgraphen für eine Berechnung. Die Tasks und Fortsetzungskanten eines Threads sind grau hinterlegt. Datenabhängigkeitskanten sind gestrichelt gezeichnet.

Abbildung 3.3 Ablaufgraph einer Berechnung mit mehreren Threads



Das oben dargestellte Schema ist ein sehr allgemeines Modell für Berechnungen mit mehreren Kontrollflüssen, da insbesondere Datenabhängigkeiten zwischen beliebigen Threads bestehen können. Dieser Umstand bewirkt, dass wie beim ähnlich allgemeinen Message Passing Programmiermodell unstrukturierte Programme entstehen können.

In diesem allgemeinen Modell lassen sich Teilklassen von Berechnungen definieren, bei denen die auftretenden Datenabhängigkeiten gewissen Einschränkungen unterliegen. Bei der Klasse der *strikten* Berechnungen führen die Datenabhängigkeitskanten eines Threads nur zu Vorfahren des Threads im Erzeugungsbaum. Abbildung 3.4 zeigt den Ablaufgraph einer strikten Berechnung.

Bei *vollständig strikten* Berechnungen können Datenabhängigkeitskanten eines Threads nur zu dessen Vater führen. In Abbildung 3.5 ist der Ablaufgraph einer vollständig strikten Berechnung abgebildet.

Produziert bei einer vollständig strikten Berechnung jeweils nur der letzte Task eines Threads ein Resultat, lässt sich diese Berechnung alternativ auch durch das Programmiermodell der asynchronen (entfernten) Prozeduraufrufe charakterisieren.

Durch diese Anforderungen wird die Komplexität der möglichen Abhängigkeitsstrukturen zwischen Threads beschränkt. Vollständig strikte Berechnungen werden deshalb auch als *wohlstrukturiert* bezeichnet.

Abbildung 3.4 Ablaufgraph einer strikten Berechnung

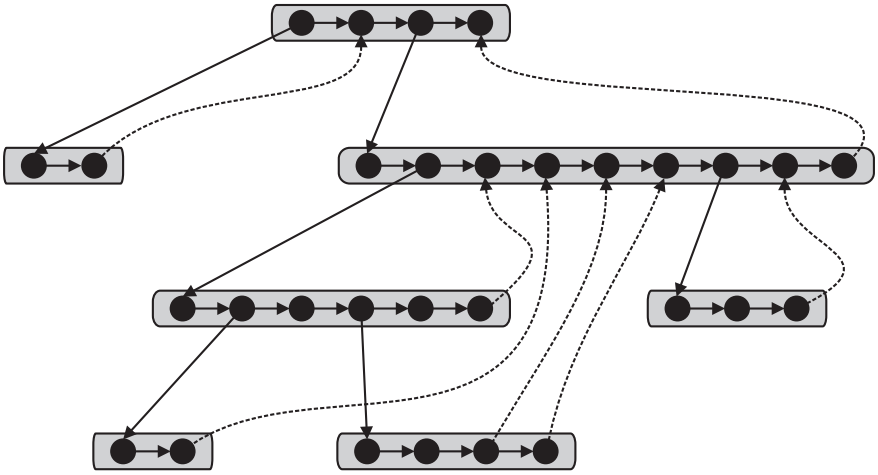
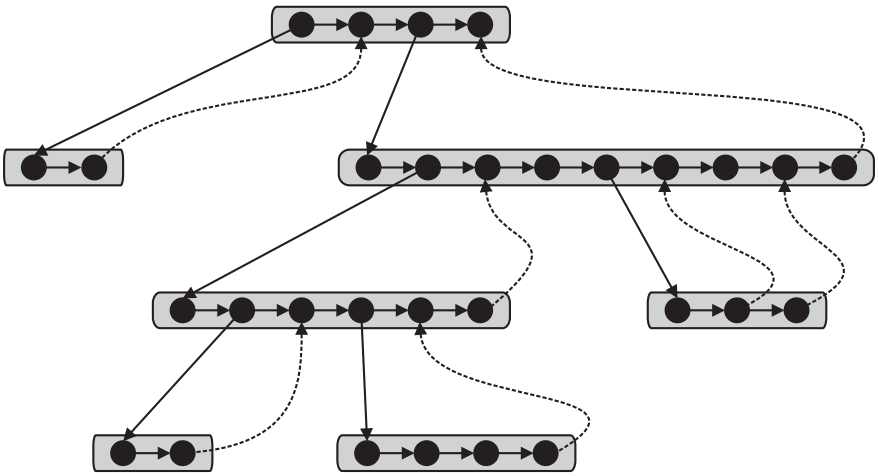


Abbildung 3.5 Ablaufgraph einer vollständig strikten Berechnung



Um Programmierern von parallelen verteilten Berechnungen das oben beschriebene Programmiermodell in Form einer Systemumgebung zur Verfügung zu stellen, muss ein entsprechendes API definiert und eine entsprechende Laufzeitumgebung implementiert werden. Hieraus ergeben sich weitere Anforderungen und Einschränkungen an das realisierte Programmiermodell.

Die Primitive des Thread API

Das Thread API stellt die zentrale Programmierschnittstelle von DOTS für die Erstellung von verteilten und parallelen Applikationen dar. Es baut unmittelbar auf dem Task API auf und erweitert dieses um spezielle Funktionalität zur Beschreibung von Berechnungen mit mehreren verteilten Threads. Beim Entwurf des Thread APIs standen die folgenden Designziele im Vordergrund:

- Unterstützung der Klasse der strikten Berechnungen
- Einfache und orthogonal verwendbare Primitive zur Erstellung von kompakten und wohlstrukturierten Programmen
- Unterstützung objektorientierter Programmierung
- Optimierung für die Programmierung von Anwendungen aus dem Bereich des Symbolischen Rechnens, zum Beispiel stark irregulärer Suchprobleme

Im Folgenden wird beschrieben, wie diese Anforderungen umgesetzt wurden.

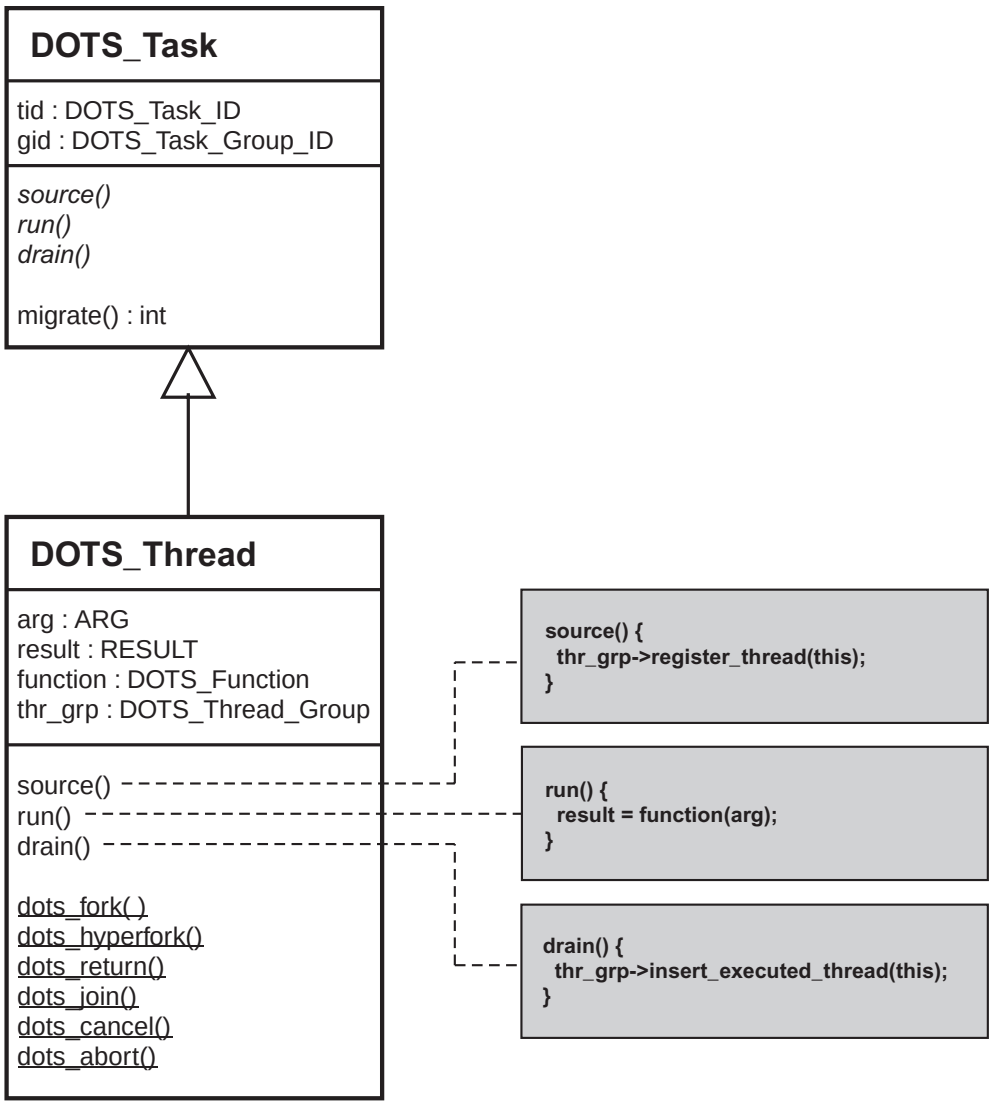
Die grundlegende Ausführungseinheit in diesem API sind so genannte DOTS Threads. Ein DOTS Thread ist ein DOTS Task-Objekt, das eine einstellige Funktion und ein ihrer Signatur entsprechendes Argument- und Resultat-Objekt kapselt. Bei seiner Erzeugung wird durch Primitive des DOTS Thread API das Funktions- und Argument-Feld initialisiert. Bei der Ausführung wird das Resultat-Objekt als Anwendung der Funktion auf das Argument-Objekt berechnet. Beim DOTS Thread API geschieht die Strukturierung der parallelen Ausführung also auf der Ebene von Funktionen, so dass die elementaren Tasks eines Threads die einzelnen Anweisungen der Funktion sind.

Es können beliebige Klassen von Argument und Resultat-Objekten verwendet werden, insbesondere müssen diese nicht von speziellen Basisklassen abgeleitet sein. Es werden grundsätzlich bei allen API Primitiven Zeiger auf Argument und Resultat-Objekte verwendet, sofern es sich nicht um primitive Datentypen handelt. Jeder DOTS Thread wird bei seiner Erzeugung einer Thread Gruppe zugeordnet. Nach seiner Ausführung kann das Resultat-Objekt durch ein API Primitiv unter Angabe der Thread Gruppe abgefragt werden.

Abbildung 3.6 zeigt ein UML Klassendiagramm, das die Beziehung zwischen DOTS Tasks und DOTS Threads veranschaulicht.

Zur Verwaltung von DOTS Threads auf Programmebene dienen Thread Gruppen. Mittels Thread Gruppen können DOTS Threads, die logisch miteinander in Beziehung stehen, in einheitlicher Weise behandelt werden. Thread Gruppen stellen eine Erweiterung der Task Gruppen des Task API dar.

Abbildung 3.6 Die Beziehung zwischen DOTS Tasks und DOTS Threads



Jeder DOTS Thread ist während seines gesamten Lebenszyklus Mitglied einer Thread Gruppe, die als eine Instanz der vom Thread API zur Verfügung gestellten Klasse `DOTS_Thread_Group` im Programm repräsentiert wird. In einer Thread Gruppe können solche DOTS Threads zusammengefasst werden, deren Funktionen dieselbe Signatur aufweisen, also gleiche Argument und Resultat Typen haben. Es können in einem Programm mehrere Thread Gruppen instantiiert werden, um DOTS Threads, die miteinander nicht in Beziehung stehen, getrennt verwalten zu können. Ein DOTS Thread kann entweder *explizit* oder *implizit*

Abbildung 3.7 Die Aufrufsyntax der Primitive des DOTS Thread API

```
#include <dots.h>

int dots_fork(DOTS_Thread_Group& thr_grp,
              RESULT(*func)(ARG),
              ARG arg,
              DOTS_Flags flags = (DOTS_Flags)0);
```
Rückgabewert: -1 bei Fehler, sonst 0

```
int dots_return(RESULT result);
```

Rückgabewert: -1 bei Fehler, sonst 0

```
int dots_hyperfork(RESULT(*func)(ARG),
                   ARG arg,
                   DOTS_Flags flags = (DOTS_Flags)0);
```

Rückgabewert: -1 bei Fehler, sonst 0

```
int dots_join(DOTS_Thread_Group& thr_grp,
              RESULT& result,
              DOTS_Flags flags = (DOTS_Flags)0);
```

Rückgabewert: -1 bei Fehler, 0 bei leerer Gruppe, sonst 1

```
void dots_cancel(DOTS_Thread_Group& thr_grp);
void dots_abort(void);
```

zu einer Thread Gruppe hinzugefügt werden. Beim expliziten Hinzufügen wird die entsprechende Gruppe explizit im Programm benannt, während sich beim impliziten Hinzufügen die Thread Gruppe aus dem Ausführungskontext ergibt.

Das DOTS Thread API stellt die im Folgenden beschriebenen grundlegenden Primitive zur Verfügung. In Abbildung 3.7 ist die genaue Aufrufsyntax für alle Primitive zusammengefasst.

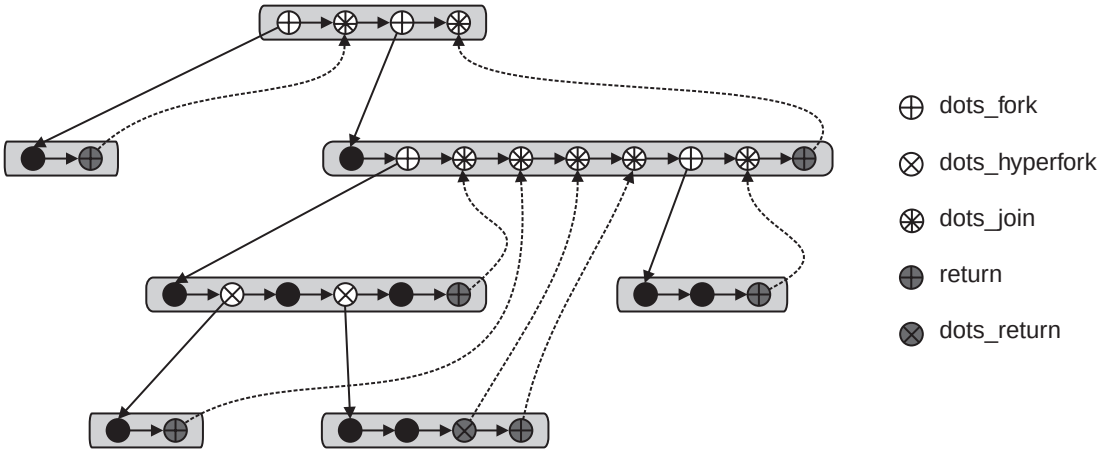
- `dots_fork`
  Dieser Aufruf wird verwendet, um einen neuen DOTS Thread zu erzeugen. Als Argumente sind eine Thread Gruppe anzugeben, zu der der neu erzeugte DOTS Thread hinzugefügt wird. Der DOTS Thread wird also in diesem Fall explizit zur Gruppe hinzugefügt. Außerdem ist die auszuführende Funktion und ein Argument-Objekt zu übergeben. Das Argu-

ment-Objekt darf im Programm erst dann modifiziert bzw. gelöscht werden, wenn der DOTS Thread ausgeführt und sein Ergebnis abgefragt wurde. Ist das Flag DELETE_ARG angegeben, so wird das Argument-Objekt automatisch nach der Ausführung gelöscht. Mittels des Flags IMMEDIATE_EXEC kann bestimmt werden, dass der erzeugte DOTS Thread unmittelbar lokal zur Ausführung kommt. Ist es nicht angegeben, wird der Ausführungszeitpunkt und der Ausführungsort für den Programmierer transparent vom Lastverteilungssystem von DOTS (siehe Abschnitt 3.4) bestimmt.

- `dots_return`
  Mit dieser Anweisung kann von einem DOTS Thread ein Resultat-Objekt zurückgegeben werden. Die Ausführung des DOTS Threads wird nach dem Aufruf weiter fortgesetzt und endet mit der Rückgabe des letzten Resultats mittels einer `return` Anweisung. Durch die Kombination von `dots_return` mit einer abschließenden `return` Anweisung kann ein DOTS Thread während seiner Ausführung eine beliebige Anzahl von Resultat-Objekten liefern. Der Typ aller zurückgegebenen Resultat-Objekte muss mit der Signatur der Funktion übereinstimmen.

- `dots_join`
  Dieses Primitiv wird verwendet, um den Nichtdeterminismus von (verteilten) parallelen Berechnungen zu kontrollieren und gleichzeitig um Resultate von DOTS Threads abzufragen. Als Argument ist eine Thread Gruppe anzugeben. Sind mehrere DOTS Threads in einer Gruppe zusammengefasst, so ist im allgemeinen Fall die Reihenfolge, in der die Resultate erzeugt werden, nicht vorherbestimmbar. Um blockierend (also effizient) auf Resultate von DOTS Threads einer Gruppe warten zu können, ist deshalb ein geeignetes Konstrukt notwendig, mit dem dieses nichtdeterministische Verhalten einer Berechnung explizit vom Programmierer kontrolliert werden kann. Der Aufruf von `dots_join` auf einer Thread Gruppe liefert ein Resultat-Objekt zurück. Sind keine Resultate von DOTS Threads der angegebenen Gruppe verfügbar, so blockiert der Aufruf, bis ein DOTS Thread der Gruppe ein Resultat liefert. Das Resultat wird von `dots_join` als Referenz auf das Resultat-Objekt zurückgegeben. Nachdem das letzte Resultat eines DOTS Threads abgefragt wurde, wird dieser automatisch aus der Gruppe entfernt. Ist kein DOTS Thread in der Gruppe enthalten, so gibt der Aufruf den Wert 0 zurück. Ist das Flag PROC_CALL_EXEC angegeben, so wird versucht, einen beliebigen DOTS Thread der Gruppe als lokalen Prozeduraufruf auszuführen, falls dieser nicht bereits zur Ausführung ausgewählt wurde. Der genaue Ablauf dieses Verfahrens wird im nächsten Abschnitt beschrieben.

Abbildung 3.8 Ablaufgraph einer strikten Berechnung mit DOTS Primitiven



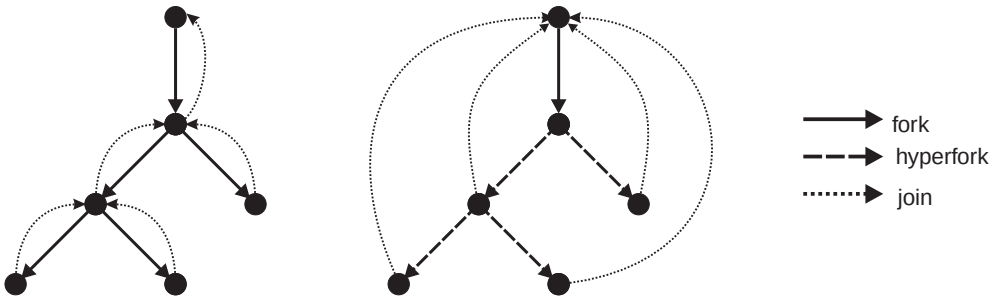
- `dots_hyperfork`
  Mittels `dots_hyperfork` wird ein neuer DOTS Thread erzeugt und implizit zur selben Thread Gruppe hinzugefügt, der der Vater Thread angehört. Die Resultate eines mit `dots_hyperfork` erzeugten DOTS Threads werden also nicht von seinem Vater, sondern von einem Vorfahren im Thread Erzeugungsbaum abgefragt und weiterverarbeitet. Als Parameter ist an `dots_hyperfork` ein Argument-Objekt und die auszuführende Funktion zu übergeben. Die Zugehörigkeit zu einer Thread Gruppe wird vom Vater Thread übernommen.

- `dots_cancel`
  Diese Anweisung dient zum Abbruch der Ausführung aller Threads einer angegebenen Thread Gruppe. Die DOTS Threads werden aus der Gruppe entfernt und alle eventuell bereits vorhanden Resultat-Objekte, die noch nicht abgefragt wurden, werden gelöscht.

- `dots_abort`
  Mit diesem Aufruf kann ein DOTS Thread seine Ausführung selbst beenden. Er gibt in diesem Fall kein (weiteres) Resultat-Objekt zurück.

Mit diesen Primitiven lassen sich strikte Berechnungen formulieren. In Abbildung 3.8 ist der in Abbildung 3.4 bereits gezeigte Ablaufgraph einer strikten Berechnung nochmals wiedergegeben, wobei für einzelne elementare Tasks der Berechnung die entsprechenden Primitive des Thread API angegeben sind.

Abbildung 3.9 Vergleich: `dots_fork` vs. `dots_hyperfork`



Virtuelle DOTS Threads

Bei der Realisierung von `dots_join` wurde die von Küchlin und Ward [54] vorgestellte Optimierung integriert. Ist zum Zeitpunkt des Aufrufs von `dots_join` kein Resultat eines Threads der angegebenen Gruppe verfügbar, so wird geprüft, ob ein Thread der Gruppe sich noch nicht in Ausführung befindet und das betreffende Task Objekt lokal gespeichert ist. Ist dies der Fall, wird die Funktion des Threads als gewöhnlicher Prozeduraufruf im Kontrollfluss des Aufrufers von `dots_join` ausgeführt. Es wird hier also nur virtuell ein join von Kontrollflüssen vorgenommen. Ansonsten blockiert der Aufruf von `dots_join`, bis ein Resultat vorliegt.

Durch diese Vorgehensweise wird sichergestellt, dass ein Thread nur dann in einem Aufruf von `dots_join` blockiert ist, wenn sonst keine weiteren Threads ausgeführt werden können. Diese Eigenschaft ist für die Ausführung vieler Programme wesentlich, da das Laufzeitsystem von DOTS kein präemptives Scheduling von Tasks vornimmt.

Beispielprogramme

Im Programmbeispiel 3.1 wird eine mittels des DOTS Thread APIs realisierte Berechnung der n-ten Fibonacci Zahl gezeigt[2]. Das Programmbeispiel 3.2 stellt eine analoge Berechnung dar, die aber mit dem `dots_hyperfork` Primitiv realisiert wurde.

In Abbildung 3.9 sind für beide Berechnungen vereinfachte Ablaufgraphen dargestellt. Durch die Verwendung von `dots_hyperfork` kann in diesem Beispiel die Anzahl der Datenabhängigkeiten verringert werden.

[2] Dieses Berechnungsbeispiel soll nur zu Demonstrationszwecken dienen; für die angewendete naive Berechnungsmethode ist eine parallele Ausführung nicht sinnvoll.

Programmbeispiel 3.1 Naive Berechnung der n-ten Fibonacci Zahl mit DOTS

```
#include "dots.h"

int fib(int n)
{
  if (n<2)
    return n;
  else {
    int x,y;

    DOTS_Thread_Group thread_group;
    dots_fork(thread_group, fib, n-1);
    dots_fork(thread_group, fib, n-2);

    dots_join(thread_group, x, PROC_CALL_EXEC));
    dots_join(thread_group, y, PROC_CALL_EXEC));

    return (x+y);
  }
}

int main(int argc, char* argv[])
{
  dots_reg(fib);
  dots_init(argc, argv);

  int n, result;
  n = atoi(argv[1]);

  DOTS_Thread_Group thread_group;
  dots_fork(thread_group, fib, n);
  dots_join(thread_group, result, PROC_CALL_EXEC));

  printf("Result: %d\n", result);

  return 0;
}
```

Programmbeispiel 3.2 Naive Berechnung der n-ten Fibonacci Zahl mit dem `dots_hyperfork` Primitiv

```
int fib(int n)
{
  if (n<2)
    return n;
  else {
    dots_hyperfork(fib, n-1);
    dots_hyperfork(fib, n-2);

    dots_abort();
  }
}

int main(int argc, char* argv[])
{
  dots_reg(fib);
  dots_init(argc, argv);

  int n, res, result=0;
  n = atoi(argv[1]);

  DOTS_Thread_Group thread_group;
  dots_fork(thread_group, fib, n);

  while (dots_join(thread_group, res) > 0) {
    result = result + res;
  }

  printf("Result: %d\n", result);

  return 0;
}
```

Diskussion des vom Thread API realisierten Programmiermodells

Mit dem DOTS Thread Programmiermodell können strikte Berechnungen realisiert werden. Es ist somit wesentlich mächtiger, als das Modell der asynchronen entfernten Prozeduraufrufe, bei dem nur simple Kommunikationsmöglichkeiten zwischen einzelnen Threads bestehen. Im Vergleich zum allgemeineren Messa-

ge Passing Programmiermodell bleibt jedoch die Kommunikationsstruktur des Programms transparent, so dass DOTS Programme auf einer höheren Abstraktionsebene angesiedelt sind.

Aufgrund ihres funktionsbasierten Charakters weisen parallele Programme, die mit dem DOTS Thread Programmiermodell realisiert sind, nur geringe Unterschiede bezüglich ihrer Struktur gegenüber den entsprechenden sequentiellen Programmen auf. Dies ermöglicht eine leichte Transformation eines sequentiellen in ein entsprechendes paralleles C++ Programm. Die Integration objektorientierter Techniken unterstützt diesen Prozess zusätzlich.

Parallele Programme, die für SMP Systeme mit den Multithreading-Primitiven der Betriebssystemschnittstelle realisiert wurden, können unter Verwendung des DOTS Thread API in Programme für verteilte Systeme umgewandelt werden, ohne dass hierfür die Programmstruktur grundlegend verändert werden muss. Die Laufzeitumgebung von DOTS ermöglicht es sogar, dass eine Applikation, die mit dem DOTS Thread API erstellt wurde, ohne Neuübersetzung sowohl auf SMP Systemen, als auch auf Rechnernetzwerken effizient ausgeführt werden kann. Das von DOTS realisierte Threads Programmiermodell weist somit ein hohes Maß an Stabilität auf.

Durch die Anwendung von `dots_join()` auf eine Thread Gruppe wird eine *join-any* Semantik zur Kontrolle von Nichtdeterminismus realisiert. In Kombination mit dem `dots_cancel()` Primitiv steht dem Programmierer somit ein mächtiges Werkzeug zur effizienten Realisierung von stark irregulären parallelen Suchvorgängen zur Verfügung.

3.2.3 Ergänzung des Thread API um Primitive zur Erhöhung der Zuverlässigkeit der Programmausführung

In [39, 8] wurde eine Erweiterung der DOTS Thread Programmierschnittstelle um Primitive zur Erhöhung der Zuverlässigkeit *(Dependability)* der Ausführung von Berechnungen vorgestellt. Im Folgenden werden die wichtigsten Elemente dieser Erweiterung beschrieben.

Die Zuverlässigkeit der Ausführung einer Berechnung kann auf zwei verschiedene Arten erhöht werden [19]:

- Fehlervermeidung *(fault prevention)*
- Fehlertoleranz *(fault tolerance)*

Der Aspekt der Fehlervermeidung wird durch das einfache und klar strukturierte Programmiermodell von DOTS berücksichtigt. Bei der Erstellung von parallelen verteilten Applikationen mit DOTS muss sich der Programmierer nicht

um systemtechnische Details, wie etwa Kommunikation oder Synchronisation kümmern. Zudem weisen Programme, die mit DOTS erstellt wurden, eine kompakte und übersichtliche Struktur auf. Beide Eigenschaften von DOTS verringern die Wahrscheinlichkeit des Auftretens von Programmierfehlern.

Der Begriff Fehlertoleranz ist dadurch gekennzeichnet, dass aus dem Auftreten von Fehlern während der Programmausführung, wie etwa dem Ausfall eines Knotens, nicht ein vollständiges Fehlschlagen der Berechnung resultiert. Im Bereich des parallelen Rechnens ist insbesondere auch eine schlechte oder keine Beschleunigung der Berechnung als Fehlschlag zu betrachten.

Um für die Ausführung von Berechnungen mit DOTS Fehlertoleranz zu realisieren, werden Checkpointingverfahren in das System integriert. Beim Checkpointing wird in periodischen Abständen der Zustand der Berechnung auf einem stabilen Medium gespeichert. Beim Auftreten von Fehlern kann die Berechnung im zuletzt gespeicherten Zustand neu aufgesetzt werden.

Ähnlich wie bei der Objektmigration lassen sich Checkpointingverfahren nach dem Umfang des zu sichernden Berechnungszustands und der daraus resultierenden Transparenz und Effizienz des Verfahrens charakterisieren. Zusätzlich ist zu beachten, dass durch den Checkpointingprozess alle Programmläufe beeinflusst werden, auch solche, bei denen kein Fehler auftritt. Da diese mit einer viel höheren Wahrscheinlichkeit vorkommen, als Programmläufe, bei denen Knoten ausfallen, ist die Effizienz, mit dem der Checkpointingprozess durchgeführt werden kann, ein wichtiges Kriterium.

Um sowohl der Anforderung nach Transparenz, als auch der Forderung nach Effizienz gleichermaßen gerecht zu werden, ist das Checkpointing in DOTS durch zwei orthogonale Verfahren realisiert.

Beim *impliziten Checkpointing* wird das spezifische Ausführungsmuster von Berechnungen mit DOTS Threads berücksichtigt, um den Umfang und Zeitpunkt des Checkpointingprozesses für den Programmierer transparent festzulegen. Das *explizite Checkpointing* erlaubt hingegen dem Programmierer, den Zeitpunkt und Umfang des Checkpointingprozesses mittels spezieller Primitive explizit zu kontrollieren.

Eine weitere, in diesem Zusammenhang zu untersuchende Fragestellung ist, ob die Speicherung der Checkpointdaten zentral oder lokal auf den einzelnen Knoten erfolgen soll. Bei beiden Checkpointingverfahren wurde eine zentrale Lösung gewählt. Diese Vorgehensweise ist zwar mit einem erhöhten Kommunikationsaufwand verbunden, die Berechnung kann jedoch bei einem zentralen Ansatz auch bei nicht-transienten Fehlern eines Knotens im gespeicherten Zustand fortgesetzt werden.

Implizites Checkpointing-Verfahren

Beim impliziten Checkpointing handelt es sich um ein für den Programmierer transparent durchgeführtes Verfahren. Es basiert auf einer Methode, die als *Function Memoization* bezeichnet wird.

Ein Checkpoint besteht aus einem Paar, welches aus dem Argument-Objekt eines Threads und dem berechneten Resultat-Objekt besteht. Nachdem ein DOTS Thread ausgeführt wurde, wird ein entsprechender Checkpoint in einer Hash-Tabelle gespeichert. Bei einem Neustart der Berechnung nach einem Fehler wird vor der Ausführung eines Threads zunächst geprüft, ob dessen Argument-Objekt sich bereits in der Tabelle befindet. In diesem Fall wird das Resultat-Objekt nicht erneut berechnet, sondern aus der Tabelle entnommen. Es müssen also nur Berechnungen von Threads wiederholt werden, deren Ausführung beim Auftreten des Fehlers noch nicht beendet war. Das implizite Checkpointing-Verfahren ist somit besonders für Berechnungen mit fein-granularen Threads geeignet.

Explizites Checkpointing-Verfahren

Für die Realisierung des expliziten Checkpointing-Verfahrens wurden die folgenden Primitive dem DOTS Thread API hinzugefügt:

- `dots_reg_cpobj()`
- `dots_init_cpobj()`
- `dots_do_cp()`

Diese neuen Primitive werden verwendet, um die Granularität bezüglich des Umfangs und des zeitlichen Ablaufs des Checkpointing-Prozesses zu kontrollieren. Beim expliziten Checkpointing-Verfahren besteht ein Checkpoint aus den Daten anwendungsspezifischer Objekte. Mittels `dots_reg_cpobj()` können vom Programmierer alle für den Ausführungszustand relevanten Objekte für das Checkpointing registriert werden.

Durch den Aufruf von `dots_do_cp()` wird veranlasst, dass von jedem registrierten Objekt ein Checkpoint geschrieben wird. Der Programmierer muss also im Programm diejenigen Stellen identifizieren, an denen der Ausführungszustand sinnvollerweise gespeichert werden kann.

Um im Falle eines Neustarts die Programmausführung mit den gespeicherten Daten fortsetzen zu können, ist es erforderlich, dass alle für das Checkpointing registrierten Objekte vor ihrer ersten Verwendung mit einem Aufruf von

`dots_init_cpobj()` initialisiert werden. Abhängig davon, ob es sich um eine reguläre Programmausführung oder um einen Neustart handelt, wird dieser Initialisierungsvorgang auf unterschiedliche Art und Weise ausgeführt.

- Bei einem regulären Programmlauf wird das Objekt mit dem als Argument anzugebenden Wert initialisiert.
- Handelt es sich um einen Neustart des Programms, wird das Objekt mit den Daten aus dem zuletzt gespeicherten Checkpoint initialisiert.

Die Initialisierung mit den Daten aus einem Checkpoint geschieht im Falle eines Neustarts automatisch. Es muss aber anschließend eine explizite Inspektion des betreffenden Objekts vorgenommen werden, um die Programmausführung wieder in den entsprechenden Zustand versetzen zu können.

3.2.4 Autonomous Task API

Durch das Autonomous Task API wird eine spezielle Klasse von DOTS Tasks eingeführt, die so genannten *autonomen Tasks*. Ein wesentliches Charakteristikum von autonomen Tasks ist, dass ihr Ausführungsort wechselt und diese nicht der Kontrolle durch die Lastverteilung des Systems unterliegen. Das Autonomous Task API stellt Methoden zur komfortablen Steuerung der Migration von autonomen Tasks zur Verfügung.

Bei einem autonomen Task handelt es sich um ein Objekt einer Klasse, die von `DOTS_Autonomous_Task` abgeleitet ist. Da die Klasse `DOTS_Autonomous_Task` eine Unterklasse von `DOTS_Task` darstellt, steht die vollständige Funktionalität von DOTS Tasks zur Verfügung. Insbesondere muss ein autonomer Task die `run()` Methode reimplementieren.

Mit der API Funktion `start_autonomous_task(DOTS_Autonomous_Task*)` kann die Ausführung eines autonomen Tasks gestartet werden. Der Knoten des Rechnernetzwerks, auf dem diese Funktion aufgerufen wurde, wird als *Heimatknoten (home node)* des autonomen Tasks bezeichnet.

Zur Kontrolle von autonomen Tasks stehen die folgenden Methoden zur Verfügung:

- `travel_to_next_node()`
  Der autonome Task soll zum nächsten Knoten der Reiseroute migrieren.
- `travel_home()`
  Der autonome Task soll zu seinem Heimatknoten migrieren.

- `bool home_node()`
  Abfrage, ob sich der autonome Task auf seinem Heimatknoten befindet.
- `exclude_node()`
  Lokalen Knoten aus Reiseroute entfernen.
- `include_node(DOTS_Node_Id)`
  Knoten zu Reiseroute hinzufügen.

Mittels autonomer Tasks kann zum Beispiel eine rudimentäre Form von mobilen Agenten realisiert werden. In Kapitel 5 ist ein größeres Anwendungsbeispiel für das Autonomous Task API gegeben.

3.2.5 Active Message API

Mit den Primitiven des Active Message API lässt sich Kommunikation zwischen den Knoten eines Rechnernetzwerks mittels Austausch von aktiven Nachrichten realisieren. Wie in Abschnitt 2.3.5 beschrieben, besteht eine aktive Nachricht sowohl aus einem zu übertragenden Datenteil, als auch aus einem Codeteil, der ausgeführt wird, nachdem die Nachricht beim Empfänger angekommen ist.

Eine aktive Nachricht in DOTS stellt ein Objekt einer anwendungsspezifischen Klasse dar, die von der Klasse `DOTS_Active_Msg` abgeleitet ist und die `run()` Methode reimplementiert.

Mit der Methode `DOTS_Active_Msg::send(DOTS_Node_ID)` wird die Nachricht zum angegebenen Knoten verschickt. Nachdem diese beim Empfängerknoten angekommen ist, kommt die `run()` Methode der Nachricht zur Ausführung. Innerhalb von `run()` findet nun die anwendungsspezifische Verarbeitung der Nachricht auf Empfängerseite statt.

Bei der Klasse `DOTS_Active_Msg` handelt es sich um eine Unterklasse der Klasse `DOTS_Task`, die deren Funktionalität nur geringfügig erweitert. In Programmbeispiel 3.3 ist der vollständige Programmcode angegeben.

3.3 Systemarchitektur

In diesem Abschnitt soll die Architektur des Laufzeitsystems von DOTS beschrieben werden, mit dessen Hilfe die Funktionalität der DOTS APIs realisiert wird. Beim Entwurf der Systemarchitektur wurden die folgenden Ziele verfolgt:

- Trennung von Ausführungs- und Verteilungsaspekten.
  Die Ausführung von DOTS Tasks und deren Verteilung auf die vorhande-

Programmbeispiel 3.3 Die Implementierung der Klasse DOTS_Active_Msg

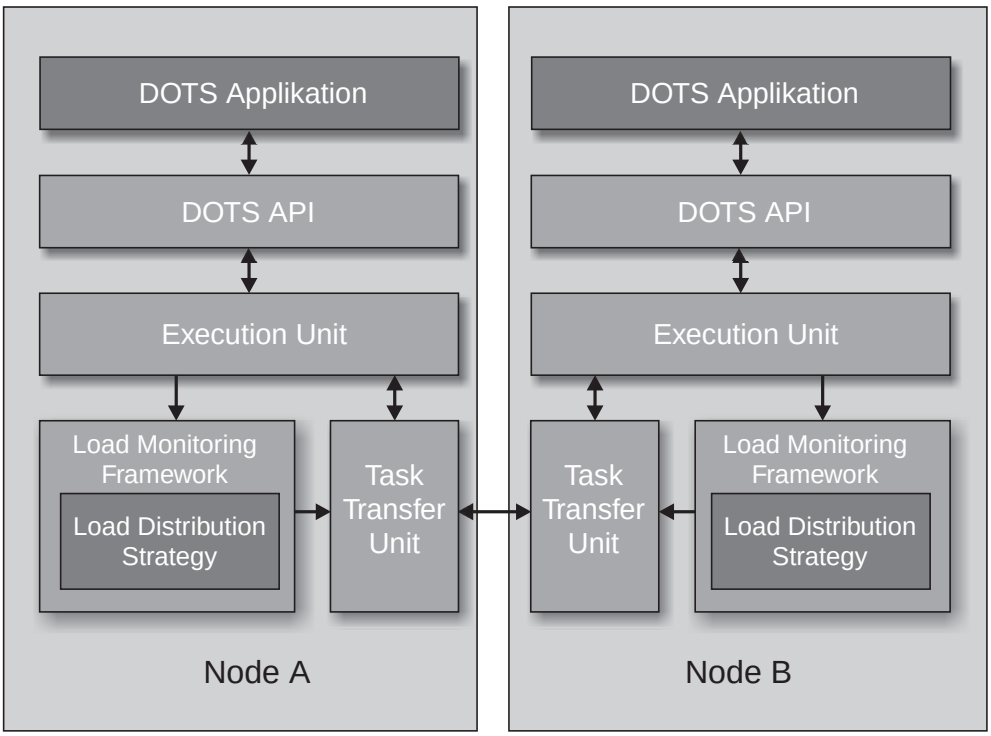
```
#include "DOTS_Task.h"
#include "DOTS_Node_Addr.h"

class DOTS_Active_Msg : public DOTS_Task
{
 public:
  void send(DOTS_Node_ID);
};

inline void DOTS_Active_Msg::send(DOTS_Node_ID node_id)
{
  set_run_node(node_id);
  go();
}
```

nen Prozessoren soll unabhängig voneinander ablaufen (sofern die Ausführungsorte der Methoden nicht explizit angegeben wurden). Dies bedeutet zum einen, dass eine DOTS Applikation ohne Modifikation (also auch ohne Neuübersetzung) sowohl auf Parallelrechnern mit gemeinsamem Speicher, als auch auf Parallelrechnern mit verteiltem Speicher *effizient* ablaufen kann. Zum anderen können für den Einsatz auf Parallelrechnerarchitekturen mit verteiltem Speicher verschiedene Lastverteilungsstrategien verwendet werden, ohne dass der Programmcode der Applikation angepasst werden muss.

- Einfache Erweiterbarkeit.
  Die Architektur soll modular aufgebaut sein, so dass sie einfach erweiterbar ist und somit gegebenenfalls neue Funktionalität leicht integriert werden kann.

Entsprechend diesen Vorgaben wurde die Systemarchitektur von DOTS aus mehreren unabhängigen Komponenten aufgebaut. In Abbildung 3.10 sind die einzelnen Komponenten dargestellt. Diese müssen bei einer verteilten Berechnung mit DOTS auf jedem Knoten vorhanden sein und werden von der Laufzeitbibliothek für jede DOTS Applikation zur Verfügung gestellt. Die Abbildung zeigt beispielhaft zwei Knoten mit den erforderlichen Komponenten der Systemarchitektur.

Abbildung 3.10 Die Systemarchitektur von DOTS

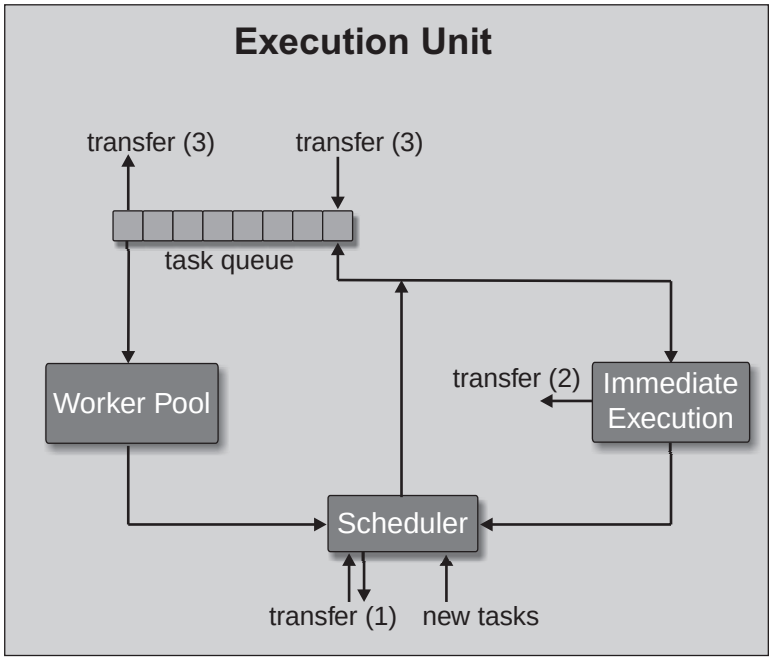


Die Systemarchitektur von DOTS umfasst im Einzelnen die folgenden Komponenten:

- **Execution Unit**
  Diese Komponente steuert den Ausführungsprozess von DOTS Tasks. Sie besteht im Wesentlichen aus einem *Scheduler*, einer *Task Queue* und einem *Worker Pool* (siehe Abbildung 3.11).

  Nach der Aktivierung eines DOTS Task-Objekts mittels `DOTS_Task::go()` wird dieses der lokalen Execution Unit übergeben. Der Scheduler prüft den Ausführungszustand des Task-Objekts und ermittelt daraus die als nächstes auszuführende Methode. Wurde für diese Methode explizit ein Ausführungsort festgelegt, der nicht dem lokalen Knoten entspricht, wird ein Transfer des Task-Objekts zum entsprechenden Knoten veranlasst; das Task-Objekt wird dazu der Task Transfer Unit übergeben. Nach dessen Transferierung wird das Task-Objekt der Execution Unit des Zielknotens übergeben. Sonst wird die nächste Methode in einem eigenen Betriebssystem-Thread lokal ausgeführt. Nach deren Ausführung wird das Task-Objekt wieder dem Scheduler übergeben.

Abbildung 3.11 Execution Unit



Handelt es sich bei der als nächstes auszuführenden Methode um `run()` und ist kein Ausführungsort angegeben, so wird das Task-Objekt in die Task Queue eingestellt. Zur Ausführung der in der Task Queue wartenden Tasks wird beim Programmstart ein Pool von Worker Threads (Worker Pool) erzeugt, der aus Betriebssystem-Threads besteht. Die Größe des Worker Pools kann zu Beginn der Berechnung festgelegt werden. Sie richtet sich gewöhnlich nach der Anzahl der lokal vorhandenen Prozessoren. Bei bestimmten Anwendungen kann aber auch eine größere Zahl von Worker Threads sinnvoll sein, um eine effizientere Ausnutzung der Prozessoren durch Überlappung von Berechnung und Kommunikation zu erreichen.

Jeder Worker Thread führt die folgenden Schritte aus:

1. Task-Objekt aus der Task Queue nehmen; ggf. blockieren bis ein Task-Objekt in die Task Queue eingestellt wird.
2. `run()` Methode des Task-Objekts ausführen.
3. Task-Objekt dem Scheduler zu weiteren Verarbeitung übergeben.
4. Weiter mit Schritt 1.

Wird vom Scheduler festgestellt, dass alle Methoden eines Task-Objekts

ausgeführt wurden, ist dessen Ausführungsprozess beendet.

Ist für den Ausführungsort der Methoden `source()` und `drain()` der lokale Knoten angegeben (oder sind diese Methoden nicht anwendungsspezifisch redefiniert worden) und wurde für die `run()` Methode nicht explizit ein Ausführungsort angegeben, so können DOTS Tasks vollständig lokal ausgeführt werden, ohne dass weitere Systemkomponenten beteiligt sind. Diese Situation liegt zum Beispiel bei der Ausführung von DOTS Threads vor.

Bei der Ausführung von DOTS Programmen auf Parallelrechnern mit verteiltem Speicher sind zusätzlich zur Execution Unit die folgenden Systemkomponenten beteiligt.

- **Task Transfer Unit**
  Diese Komponente realisiert den Transfer von DOTS Task-Objekten zwischen den Knoten eines Rechners mit verteiltem Speicher. Task-Objekte werden in serialisierter Form übertragen, wobei der Serialisierungsvorgang auf der Senderseite und Deserialisierungsvorgang auf der Empfängerseite von den jeweiligen Task Transfer Units kontrolliert wird. In Abschnitt 4.3 wird das Serialisierungsverfahren von DOTS ausführlich beschrieben. Eine Übertragung von Task-Objekten zwischen zwei Knoten kann aus den folgenden Gründen erforderlich werden (vgl. Abbildung 3.11):

  1. Für eine Methode ist ein entfernter Ausführungsort angegeben.
  2. Durch `DOTS_Task::migrate()` wurde eine Migration explizit im Programm ausgelöst.
  3. Zur Lastverteilung kann ein Transfer von Task-Objekten aus der lokalen Task Queue in eine entfernte Task Queue erfolgen.

- **Load Monitoring Framework**
  Das Load Monitoring Framework stellt die Grundfunktionalität zur Integration von Lastverteilungsstrategien *(Load Distribution Strategy)* zur Verfügung. Es überwacht alle lokalen Ausführungsereignisse und sammelt Statusinformationen über die lokale Execution Unit. Im folgenden Abschnitt wird detailliert beschrieben, wie mittels des Load Monitoring Framework benutzerdefinierte Lastverteilungsverfahren implementiert werden können, bzw. welche Verfahren bereits auf diese Weise in DOTS integriert sind.

3.4 Lastverteilung

Die Lastverteilungskomponente von DOTS steuert die Verteilung von DOTS Task-Objekten auf die vorhandenen Knoten des Systems zur Ausführung ihrer `run()` Methode. In diesem Abschnitt soll deren Realisierung vorgestellt werden. Es wird zunächst ein allgemeiner Überblick über die Realisierungsmöglichkeiten für die Lastverteilung gegeben. Im Anschluss werden die von DOTS bereitgestellten Lastverteilungsverfahren beschrieben. Abschließend wird die Integration neuer Lastverteilungsverfahren in DOTS behandelt.

3.4.1 Realisierungsmöglichkeiten der Lastverteilung

DOTS Applikationen können zu jedem Zeitpunkt ihrer Ausführung dynamisch neue Tasks erzeugen. Eine statische Ablaufplanung ist somit nicht möglich. Folglich werden nur dynamische Lastverteilungsverfahren in Betracht gezogen.

Bei der Diskussion von Lastverteilungsverfahren muss zunächst definiert werden, was unter dem Begriff der *Last* eines Knotens des Systems zu verstehen, bzw. wie diese zu messen ist.

Bei einfachen Lastmaßen wird zum Beispiel die Anzahl der sich momentan auf einem Prozessor in Ausführung befindlichen Ablaufeinheiten in Betracht gezogen. Kompliziertere Lastmaße beachten zusätzlich die Struktur der zu verteilenden Teilprobleme. Zum Beispiel könnte bei einer verteilten Suche in einer Baumstruktur die Anzahl der Knoten, die einem Teilproblem zur Bearbeitung zugewiesen wurde, zur Definition der Last herangezogen werden. Falls die einzelnen Ausführungseinheiten mit einem Prioritätswert versehen sind, kann dieser einen weiteren Parameter für die Berechnung der Last eines Knotens darstellen.

Zur Lastverteilung können verschiedene Strategien angewendet werden. Beim *Load Balancing* wird versucht, die Last gleichmäßig auf die Knoten des Systems zu verteilen. Dieses Verfahren ist vor allem bei komplexen Lastmaßen sinnvoll. Eine schwächere Form von Load Balancing ist die *Load Sharing* Strategie. Hier wird versucht sicherzustellen, dass zu jedem Zeitpunkt auf allen Knoten des Systems Last vorhanden ist.

Lastverteilungsverfahren können *zentral* oder *dezentral* realisiert werden. Bei zentralen Verfahren existiert eine ausgezeichnete Komponente im System, die Informationen über den aktuellen Lastzustand aller Knoten sammelt und die Lastverteilung mittels der gewählten Strategie anhand dieser Information vornimmt. Die Lastinformation, auf der bei dieser Vorgehensweise die Entscheidung zur Verteilung basiert, ist sehr genau, aber die zentrale Komponente stellt

einen sequentiellen Flaschenhals dar, der die Skalierbarkeit des Systems einschränkt. Bei dezentralen Lastverteilungsverfahren wird die Entscheidung nur aufgrund lokal vorhandener Lastinformation getroffen. Dieses Verfahren ist skalierbar, aber die Entscheidung zur Lastverteilung basiert auf global unvollständigen Informationen, so dass die gewählte Strategie nicht immer tatsächlich umgesetzt wird. Es lassen sich auch Verfahren realisieren, die zwischen diesen beiden Extremen angesiedelt sind. So können zum Beispiel die Knoten hierarchisch in Gruppen eingeteilt werden. Die einzelnen Knoten einer Gruppe tauschen genaue Lastinformationen aus, während die Gruppen zusammengefasste Informationen über die Last ihrer Mitglieder austauschen. Vergleiche hierzu die ausführlichen Untersuchungen in [17].

3.4.2 Integrierte Lastverteilungsverfahren

In DOTS wird die Lastverteilung auf zwei Ebenen realisiert. Auf der ersten Ebene wird eine einfache Load Balancing Strategie angewendet, während auf der zweiten Ebene eine (austauschbare) Load Sharing Strategie zur Anwendung kommt. Beide Ebenen sind vollkommen dezentral realisiert, es werden also nur lokal vorhandene Informationen zur Entscheidung der Verteilung berücksichtigt.

Das Lastmaß auf der ersten Ebene ist die Anzahl der aktiven Worker Threads auf einem Knoten. Durch die Vorgabe einer festen Anzahl von Betriebssystem-Threads im Worker Pool wird die Last auf jedem Knoten nach oben beschränkt. Sind auf einem Knoten mehr Task-Objekte zur Ausführung bereit, als Worker Threads zur Verfügung stehen, werden diese in die Task Queue eingestellt. Somit wird sichergestellt, dass sich einerseits die Lastunterschiede der einzelnen Knoten auf dieser Ebene in einem definierten Bereich befinden und andererseits diese Begrenzung für den Programmierer transparent erfolgt. Diese Vorgehensweise verhindert, dass durch eine zu große Anzahl von Worker Threads deren Verwaltungsaufwand zu hoch wird und sich dieser negativ auf die Berechnung der Anwendung auswirkt.

Als Lastmaß für die zweite Ebene wird die Länge der lokalen Task Queue betrachtet. Zur Realisierung der Load Sharing Strategie auf dieser Ebene kann eines der beiden folgenden Verfahren (oder beide in Kombination) gewählt werden:

- **Task Sharing**
  Bei diesem Verfahren werden bei Überschreitung einer vorgegebenen maximalen Länge der Task Queue Tasks zu anderen Knoten transferiert. Die Auswahl der Knoten erfolgt dabei nach dem Round Robin Prinzip, d.h. die Tasks werden reihum gleichmäßig auf die anderen Knoten verteilt. Dieses

Verfahren eignet sich zum Beispiel für Master-Slave Algorithmen, bei denen neue Tasks nur auf dem Master Knoten erzeugt werden.

- **Task Stealing**
  Dieses Verfahren zur Lastverteilung funktioniert nach einem zum Task Sharing komplementären Prinzip. Wenn die Länge der lokalen Task Queue einen vorgegebenen Wert unterschreitet, wird versucht, anderen Knoten Tasks wegzunehmen. Dazu wird ein Opfer ausgewählt und eine entsprechende Anfrage übermittelt. Die Auswahl eines Opfers kann fest vorgegeben werden oder randomisiert erfolgen. Ist nach einer bestimmten Wartezeit kein neuer Task eingetroffen, wird der Anfrageprozess (evtl. mit einem neuen Opfer) wiederholt.

  Falls der Anfrageprozess erst dann gestartet wird, wenn die lokale Task Queue keine Tasks mehr enthält, arbeitet diese Lastverteilungsstrategie besonders gut mit dem virtuellen Ausführungsmodus von `dots_join` zusammen, da nur Threads aus der lokalen Queue entnommen werden, wenn diese unmittelbar zur Ausführung kommen sollen. Somit ist sichergestellt, dass solange sich ein DOTS Thread nicht in Ausführung befindet, das entsprechende Task Objekt in der lokalen Task Queue bleibt und bei einem späteren Aufruf von `dots_join` gegebenenfalls virtuell ausgeführt werden kann.

3.4.3 Integration neuer Lastverteilungsverfahren

In bestimmten Fällen kann es vorkommen, dass die von DOTS standardmäßig zur Verfügung gestellten Lastverteilungsverfahren nicht ausreichend sind. Dies ist zum Beispiel der Fall, wenn bei der Zuordnung von Tasks spezielle Eigenschaften von Knoten berücksichtigt werden müssen oder wenn für die Verteilungsentscheidung bestimmte Eigenschaften der Tasks eine Rolle spielen. Ein weiteres Beispiel für diese Situation ist, wenn in der `source()` bzw. `drain()` Methode der Task-Objekte einer Anwendung umfangreicher Code ausgeführt wird und dies bei der Lastverteilung berücksichtigt werden soll.

Für die Integration derartiger anwendungsspezifischer Lastverteilungsverfahren in DOTS wird das Load Monitoring Framework verwendet. Man kann das Load Monitoring Framework als eine zusätzliche Programmierschnittstelle von DOTS betrachten, die es erlaubt, eine Applikation mit einem maßgeschneiderten Lastverteilungsverfahren auszustatten. Der Programmcode der Applikation und der Programmcode für das zu integrierende Lastverteilungsverfahren sind voneinander vollständig unabhängig, so dass beide zum Beispiel auch von getrennten Personen(-gruppen) erstellt und gewartet werden können.

Die Realisierung neuer Lastverteilungsverfahren mit dem Load Monitoring Framework basiert im Wesentlichen auf der Reimplementierung von Event Handlern in abgeleiteten Klassen. Dabei werden Ausführungsereignisse vom Load Monitoring Framework in Aufrufe von Event Handlern umgesetzt, die dann entsprechende anwendungsspezifische Aktionen veranlassen. Dazu überwacht das Load Monitoring Framework den gesamten Ausführungsablauf der lokalen Execution Unit. Ausführungsereignisse, die in irgendeiner Form eine Zustandsänderung der Execution Unit hervorrufen, werden vom Load Monitoring Framework in Aufrufe von entsprechenden Event Handlern umgesetzt. Es werden dabei die folgenden Ereignisse überwacht:

- Beginn/Ende der Ausführung der `source()` Methode eines DOTS Tasks
- DOTS Task wird in Task Queue eingestellt
- DOTS Task wird aus Task Queue entnommen
- Beginn/Ende der Ausführung der `run()` Methode eines DOTS Tasks durch einen Worker Thread
- Beginn/Ende der Ausführung der `drain()` Methode eines DOTS Tasks

Zur Vereinfachung können zusätzlich so genannte Schwellwert-Ereignisse definiert werden. Das Load Monitoring Framework überprüft regelmäßig in anzugebenden Zeitabständen bestimmte Parameter (zum Beispiel die Länge der Task Queue) und löst bei Über- bzw. Unterschreitung eines Schwellwerts ein entsprechendes Ereignis aus.

Die Implementierung anwendungsspezifischer Lastverteilungsverfahren besteht aus einer Hauptklasse, die von der Klasse `DOTS_Load_Monitoring_Framework` abgeleitet ist und Event Handler Methoden reimplementiert. Bei allen Event Handler Methoden wird vom Load Monitoring Framework die ID des DOTS Tasks übergeben, der das entsprechende Ausführungsereignis ausgelöst hat. Bei manchen Event Handler Methoden wird zusätzlich ein Zeiger auf das Task-Objekt als weiteres Argument übergeben. Eine anwendungsspezifische Lastverteilungsimplementierung kann mittels einer Downcastoperation auf das übergebene DOTS Task-Objekt spezifische Informationen über den Task gewinnen.

Die Implementierung eines Event Handlers realisiert dann anwendungsspezifische Reaktionen auf das entsprechende Ausführungsereignis. Zum Beispiel kann die Task Transfer Unit zu einer Übertragungsaktion veranlasst werden. Im Extremfall ist es möglich, mittels des Load Monitoring Framework die Execution Unit vollständig durch anwendungsspezifische Funktionalität zu ersetzen.

Die Komponente DOTS_NODE_DIRECTORY gibt Auskunft über die derzeit im System angemeldeten Knoten und kann zur Planung von Übertragungsaktionen verwendet werden.

In Programmbeispiel 3.4 ist die Verwendung des Load Monitoring Framework zur Realisierung einer einfachen Task Stealing Strategie mit randomisierter Opferauswahl dargestellt. Soll die im Beispiel gezeigte Lastverteilungsstrategie in einer Anwendung eingesetzt werden, ist die Klasse mit dem folgenden Aufruf zu Programmbeginn anzumelden:

```
dots_reg_lds(DOTS_Simple_Randomized_Workstealing_Strategy)
```

3.5 Beschreibung und Dokumentation von Berechnungen

DOTS wurde speziell für die Durchführung von verteilten parallelen Berechnungen in stark heterogenen Rechnernetzwerken konzipiert. Typischerweise werden in solchen Rechnernetzwerken eine Vielzahl unterschiedlicher Rechnerarchitekturen und Betriebssysteme verwendet. Zusätzlich können sich die eingesetzten Rechner auch stark hinsichtlich ihrer Leistungsfähigkeit unterscheiden. Eine weitere charakteristische Eigenschaft von solchen Rechenumgebungen ist, dass sie durch häufige Veränderungen gekennzeichnet sein können: Einzelne Rechner werden ausgetauscht oder unter einem anderen Betriebssystem betrieben. Bei der Durchführung von parallelen verteilten Berechnungen in solchen heterogenen Umgebungen sind neben dem Aspekt der Programmerstellung zusätzlich die folgenden Fragestellungen hinsichtlich der Programmausführung zu beachten:

- Die genaue Definition und Dokumentation der verwendeten Rechenumgebungen ist komplex, da jeder Knoten verschiedenartig sein kann und für die genaue Beschreibung eine Vielzahl von Parametern relevant ist.
- Die Laufzeiten und Beschleunigungen einer Berechnung können nicht nur von der Anzahl der verwendeten Prozessoren, sondern auch stark von der Rechenumgebung abhängen, in der sie erzielt wurden. Dieser Umstand macht es erforderlich, dass zu der Dokumentation der Ergebnisse noch eine genaue und umfangreiche Beschreibung des Rechnernetzwerkes gegeben werden muss.

In DOTS werden beide Aspekte in einem integrierten, XML [16] basierten Ansatz behandelt. Ziel bei der Realisierung war es, möglichst alle Aspekte, die zu

einer Berechnung gehören, in einem einzigen XML Dokument zu erfassen, um somit einerseits eine umfassende Auswertung der Berechnung zu ermöglichen und andererseits eine möglichst genaue Reproduzierbarkeit von Berechnungen zu erreichen. Die Verwendung von XML Dokumenten für diese Zwecke hat die folgenden Vorteile:

- Bei XML handelt es sich um ein standardisiertes und plattformunabhängiges Format für Dokumente. Es ist somit auf natürliche Weise zur Verwendung im heterogenen Umfeld geeignet.
- In einem XML Dokument können komplexe Sachverhalte strukturiert dargestellt werden.
- Es existiert ein große Anzahl von Tools zur Erstellung, Verarbeitung und Verwaltung von XML Dokumenten.
- XML Dokumente können einfach und plattformunabhängig mit XML verwandten Technologien wie etwa XPath [22] und XSLT [21] ausgewertet und in andere Darstellungsformate umgewandelt werden.

Um sicherzustellen, dass die verwendeten XML Dokumente ein einheitliches Format haben, wurde die *DOTS Markup Language (DOTSML)* mittels einer speziellen DTD (Document Type Definition) definiert.

Ein DOTSML Dokument sieht sowohl Elemente zur Definition der Rechenumgebung, als auch zur Dokumentation einer oder mehrerer darin durchgeführten Berechnungen vor. Es wird vom Laufzeitsystem von DOTS zu dessen Konfiguration ausgewertet und um Daten der jeweils ausgeführten Berechnung erweitert.

Die Konfiguration der Rechenumgebung umfasst dabei die folgenden Aspekte:

- DOTS Programm (Quellen, ausführbare Datei)
- Strategie für Lastverteilung
- Globale Konfiguration
- Definition und Konfiguration der einzelnen Knoten des verwendeten Rechnernetzwerkes

Zur Beschreibung einer Berechnung sind die folgenden Elemente vorgesehen:

- Name, Datum und Zeitpunkt der Programmausführung
- Programmeingaben und -ausgaben

- Auswertungen der Berechnung
- Logging Meldungen

Anhang A enthält die vollständige DTD für DOTSML und ein DOTSML Dokument.

Programmbeispiel 3.4 Realisierung einer einfachen Task-Stealing Lastverteilungsstrategie mit dem Load Monitoring Framework

```
class DOTS_Simple_Randomized_Workstealing_Strategy
  : public DOTS_Load_Monitoring_Framework
{
public:
  DOTS_Simple_Randomized_Workstealing_Strategy() {
    // subscribe for exec_queue_min_threshold_event
    // threshold: 0, check interval: 250 ms
    set_exec_queue_min_threshold(0, 250);
  }

  void exec_queue_min_threshold_event(void) {
    // Are there any other nodes?
    if (DOTS_NODE_DIRECTORY->get_size()<2)
      return;

    // create message containing own node ID
    DOTS_Archive arch;
    arch << DOTS_NODE_ID;
    DOTS_Msg msg(DOTS_MSG_STEAL_TASK, &arch);

    // Choose victim node randomly
    DOTS_Node_ID victim_id;
    do {
      victim_id = rand() % DOTS_NODE_DIRECTORY->get_size();
    } while (victim_id == DOTS_NODE_ID);

    // Send message to victim node
    msg.send(victim_id);
  }

  void message_handler(DOTS_Msg* msg) {
    // Unpack message
    DOTS_Archive* arch = msg->get_archive();
    DOTS_Node_ID dest_id;
    *arch >> dest_id;

    // Try to transfer task to destination node ID
    DOTS_TASK_TRANSFER_UNIT->transfer_task(dest_id);

    delete msg;
  }
};
```

4

Kapitel 4

Design- und Implementierungsaspekte

Im vorliegenden Kapitel werden einige Design- und Implementierungsaspekte von DOTS beschrieben, die insbesondere für dessen Einsatz in heterogenen Umgebungen relevant sind. Das Kapitel behandelt in einer Bottom-Up Vorgehensweise detailliert die Realisierung der plattformunabhängigen, objektorientierten Kommunikationsarchitektur von DOTS.

- Abschnitt 4.1 befasst sich mit der objektorientierten Homogenisierung von Betriebssystemschnittstellen zur Erstellung von plattformunabhängigen C++ Programmen.
- In Abschnitt 4.2 wird der auf der TCP/IP Protokollfamilie basierte Kommunikationskern von DOTS beschrieben.
- Abschnitt 4.3 beschäftigt sich mit der Realisierung der Objektserialisierung in DOTS.

4.1 Homogenisierung von Betriebssystemschnittstellen in C++

Betriebssysteme stellen für die Implementierung von Anwendungsprogrammen eine Programmierschnittstelle zum Zugriff auf Betriebssystemfunktionalität sowie zur Manipulation von Betriebsmitteln zur Verfügung.

Zum Schutz vor der unerlaubten bzw. undefinierten Manipulation des Betriebssystems durch (evtl. fehlerhafte) Programme werden Anwendungsprogramme

und Betriebssystemroutinen vom Prozessor in zwei unterschiedlich privilegierten Modi ausgeführt. Anwendungsprogramme laufen im *User Mode* ab, während das Betriebssystem im privilegierten *Kernel Mode* (auch *Supervisor Mode* genannt) ausgeführt wird. Der Übergang des Kontrollflusses vom User Mode in den Kernel Mode ist nicht durch einfache Programmsprünge oder Unterprogrammaufrufe möglich. Er wird je nach Prozessorarchitektur über spezielle Maschineninstruktionen (Trap Befehle) und/oder über spezielle Indirektionsmechanismen (Call Gates) realisiert. Zum Aufruf von Betriebssystemroutinen innerhalb C/C++ Programmen werden in C geschriebene Mantelfunktionen zur Verfügung gestellt, in die entsprechende Assemblerbefehle eingebettet sind, um die Parameter an das Betriebssystem zu übergeben und anschließend den Übergang in den Kernel Mode zu veranlassen.

Für den Anwendungsprogrammierer besteht also die Programmierschnittstelle bei den zurzeit gebräuchlichen Betriebssystemen aus einer Menge von C Funktionsdefinitionen, die im Folgenden auch als *Systemaufrufe* bezeichnet werden. Die Menge aller Systemaufrufe charakterisiert aus Sicht des Programmierers ein Betriebssystem vollständig; sie wird als *Betriebssystemschnittstelle* oder auch Betriebssystem API *(Application Programming Interface)* bezeichnet [86].

Zum Beispiel wird im POSIX Standard [45] eine Betriebssystemschnittstelle für UNIX ähnliche Betriebssysteme, wie etwa Solaris oder BSD, definiert. Das API der verschiedenen Windows Betriebssysteme der Firma Microsoft wird mit Win32 bezeichnet [62].

Während sich die Betriebssysteme der einzelnen Hersteller aus konzeptioneller Sicht sehr ähnlich sind (zum Beispiel verwenden sie meistens dieselben Abstraktionen, wie etwa Prozesse, Threads, Dateien, Sockets, usw.), unterscheiden sich die Betriebssystemschnittstellen der verschiedenen Hersteller zum Teil erheblich. Unterschiede bei den Systemaufrufen ergeben sich nicht nur durch verschiedene Bezeichner, sondern auch durch unterschiedliche Anzahl und Typen von Parametern. Oftmals ist auch die Semantik von Systemaufrufen grundlegend verschieden, so dass in einem API mehrere Systemaufrufe nötig sind, um die Funktionalität eines Systemaufrufs des anderen APIs nachzubilden.

Beispiele:

- Die Endpunkte einer Netzwerkverbindung (Sockets) werden unter UNIX durch Integer Werte im Programm repräsentiert, während das Win32 API zur Repräsentation von Verbindungsendpunkten eine komplexere Datenstruktur vorsieht.

- Zum Erzeugen eines neuen Prozesses wird im Win32 API der Systemaufruf `CreateProcess` verwendet. Dieser erwartet als Argument unter

anderem die Angabe des Dateinamens des auszuführenden Programms. Bei UNIX wird diese Funktionalität durch zwei aufeinanderfolgend anzuwendende Systemaufrufe erreicht. Mit `fork` wird ein neuer Prozess erzeugt, der zunächst eine Kopie des Vater-Prozesses darstellt. Durch einen anschließenden `exec` Systemaufruf kann der neu erzeugte Prozess unter Angabe des Namens einer ausführbaren Datei ein neues Programm ausführen.

Für die Anwendungsprogrammierung bedeutet dieser Umstand, dass ein erheblicher Anteil des Programmcodes einer Anwendung vom zugrundeliegenden Betriebssystem abhängig ist. Zur effizienten Erstellung und Wartung von Anwendungen in der Programmiersprache C++, die auf mehreren Plattformen ablaufen sollen, ist daher die Einführung einer *Abstraktionsschicht* erforderlich, die zwischen der Betriebssystemschnittstelle und den Anwendungsprogrammen angesiedelt ist.

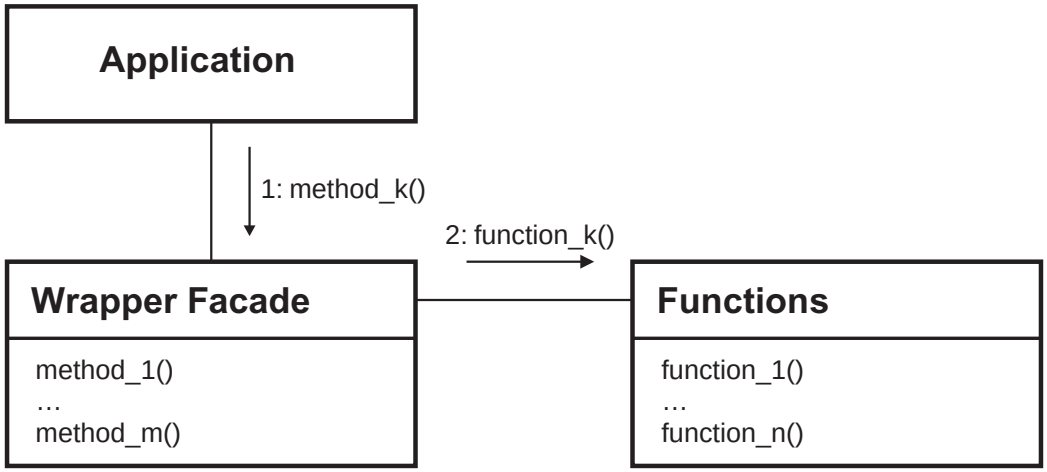
In diesem Abschnitt wird zunächst ein von Schmidt [73] beschriebenes Entwurfsmuster [34] vorgestellt, das eine Anleitung zur Erstellung einer derartigen Abstraktionsschicht gibt. Anschließend werden für die Programmiersprache C++ unterschiedliche Verfahren aufgezeigt, um eine nach dieser Art und Weise entworfene Abstraktionsschicht auf eine große Zahl von Zielplattformen abzubilden. Neben zwei klassischen Verfahren wird eine neue, auf C++ Template Techniken basierte Methode gezeigt. Im letzten Teil des Abschnitts werden zunächst verschiedene Bewertungkriterien aufgestellt, anhand derer die vorgestellen Verfahren im Anschluss verglichen werden.

Die in diesem Abschnitt diskutierten Techniken lassen sich nicht nur für Systemaufrufe, sondern auch allgemein für C basierte Bibliotheksaufrufe verwenden. Zum Beispiel finden zur Erstellung von Applikationen mit grafischen Benutzeroberflächen in dem unter UNIX gebräuchlichen X Window System eine Reihe unterschiedlicher Programmierschnittstellen, so genannte Toolkits (Motif, GTK, OpenWindows, ...), Verwendung. Auch hier könnte durch eine zusätzliche Abstraktionsschicht eine einheitliche Programmierumgebung geschaffen werden.

4.1.1 Das Wrapper Facade Entwurfsmuster

Alle nachfolgend beschriebenen Ansätze stellen Umsetzungen des von Schmidt [73] vorgestellten Wrapper Facade Entwurfsmusters dar. Die grundlegende Idee dieses Entwurfsmusters besteht darin, C basierte Systemaufrufe und Datenstrukturen durch objektorientierte C++ Klassenschnittstellen zu kapseln. Programme greifen nun nicht mehr direkt, sondern über die Indirektion einer abstrakteren, objektorientierten Schnittstelle auf Systemfunktionalität zu. Das Wrapper Fa-

Abbildung 4.1 Das Wrapper Facade Entwurfsmuster



cade Entwurfsmuster wird zum Beispiel von dem ACE (ADAPTIVE Communication Environment) Toolkit [72] oder von der Microsoft Foundation Classes (MFC) Klassenbibliothek [63] umgesetzt. Abbildung 4.1 zeigt die Struktur des Wrapper Facade Entwurfsmusters in einem UML Klassendiagramm.

Das Wrapper Facade Entwurfsmuster hat die folgenden Teilnehmer:

- **Functions**
  Die durch die jeweilige Zielplattform gegebene Menge an Systemaufrufen und zugehörigen Datenstrukturen.
- **Wrapper Facade**
  Eine oder mehrere Klassen, die die Elemente von Functions kapseln. Diese stellen den Anwendungsprogrammen Methoden zur Verfügung, die Aufrufe an jeweils eine oder mehrere Elemente von Functions weiterreichen.

Zu beachten ist, dass Functions für jede Zielplattform verschiedene Elemente enthält, während die Wrapper Facade immer gleich bleibt.

Zur Realisierung des Wrapper Facade Entwurfsmusters werden zunächst Systemaufrufe und Datenstrukturen anhand ihrer Funktionalität in verschiedene Gruppen eingeteilt. Dies können zum Beispiel Gruppen für Netzwerkkommunikation, Multithreading, Synchronisation, usw. sein. Für jede Gruppe werden dann zur Kapselung eine oder mehrere in Beziehung stehende Klassen eingeführt. Die Methoden dieser Klassen können entweder einfach eine Weiterleitung an entsprechende Systemaufrufe darstellen oder auch mehrere Systemaufrufe zu komplexeren Konstrukten kombinieren.

Aus der Verwendung des Wrapper Facade Entwurfsmusters ergeben sich die folgenden Vorteile für die Softwareentwicklung:

- Verbesserte Softwarequalität durch robuste und typsichere Programmierschnittstellen
- Kompakterer und besser strukturierter Programmcode
- Leichtere Wartung und bessere Wiederverwendbarkeit von Komponenten
- Einfachere Portierung von Applikationen auf andere Plattformen

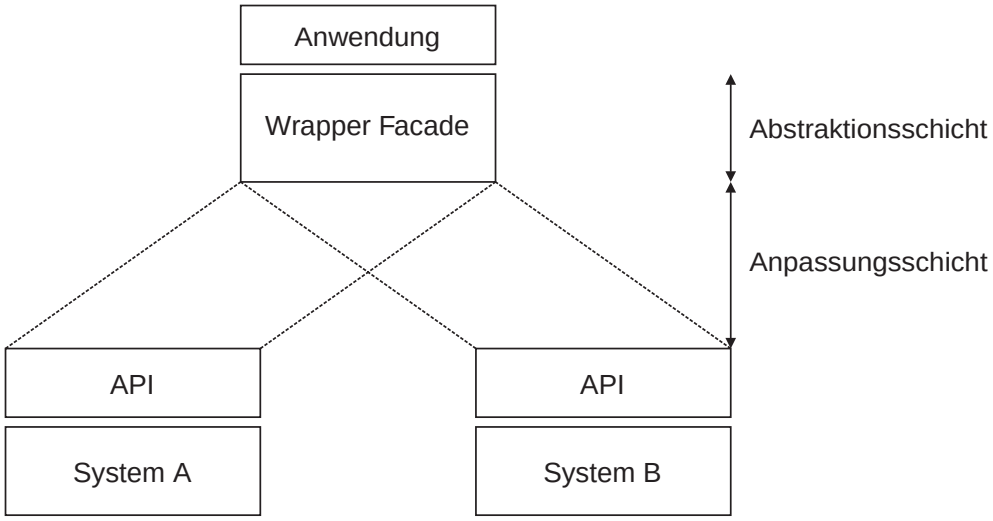
Für die in diesem Abschnitt zu untersuchende Fragestellung ist insbesondere der letztgenannte Punkt von Bedeutung. Jedoch gibt das Wrapper Facade Entwurfsmuster zunächst nur eine Anleitung, wie eine objektorientierte Kapselung von Systemaufrufen realisiert werden kann. Die Frage der Portierbarkeit, also mit welchen Mechanismen diese abstrakten Schnittstellen auf unterschiedliche Ziel-Plattformen abgebildet werden können, lässt dieses Entwurfsmuster jedoch offen. Das Wrapper Facade Entwurfsmuster befasst sich nur mit dem Aspekt der Definition einer einheitlichen, abstrakten Programmierschnittstelle, nicht aber mit dem Aspekt der (gleichzeitigen) Implementierung dieser Programmierschnittstelle auf vielen unterschiedlichen Plattformen.

Auf konzeptioneller Ebene betrachtet ist also neben einer *Abstraktionsschicht*, die mit dem Wrapper Facade Entwurfsmuster realisiert wird, noch eine weitere, zwischen Abstraktionsschicht und der jeweiligen Betriebssystemschnittstelle angesiedelte *Anpassungsschicht* erforderlich. In der Anpassungsschicht wird zum Beispiel die Anordnung von Implementierungsklassen und insbesondere die Frage der Einbindung und Organisation von plattformspezifischem Programmcode geregelt. Abstraktionsschicht und Anpassungsschicht können auf programmiertechnischer Ebene durchaus auch als Einheit realisiert werden. In Abbildung 4.2 ist die Beziehung zwischen Abstraktions- und Anpassungsschicht nochmals grafisch dargestellt.

Im Folgenden sollen verschiedene Möglichkeiten zur Realisierung dieser Anpassungsschicht aufgezeigt und miteinander verglichen werden. Bei der Beschreibung des Wrapper Facade Entwurfsmusters in [73] werden hierfür die beiden folgenden Verfahren vorgeschlagen:

- Präprozessormethode:
  Es werden Präprozessordirektiven zur bedingten Kompilation (`#ifdef`) zur Auswahl des plattformspezifischen Programmcodes verwendet. Mit

Abbildung 4.2 Das Verhältnis von Abstraktions- und Anpassungsschicht beim Wrapper Facade Entwurfsmuster



Autokonfigurationswerkzeugen (z.B. GNU autoconf) können vor der Übersetzung automatisch für jede Plattform die notwendigen Präprozessorkonstanten definiert werden.

- Codeorganisation mittels des Dateisystems:
  Für jede unterstützte Plattform wird ein separates Verzeichnis angelegt, das die Quelldateien der gesamten Implementierung für die jeweilige Plattform enthält. Die Programmierumgebung wird so konfiguriert, dass beim Übersetzen einer Anwendung die Quelldateien aus dem richtigen Verzeichnis hinzugefügt werden.

Das letztgenannte Verfahren ist bei größeren Projekten nur schwer zu realisieren, da die Programmtexte der Implementierungen für die einzelnen Plattformen in keinerlei Beziehung zueinander stehen. Es existiert keine gemeinsame Softwarebasis. In den nachfolgenden Abschnitten wird die oben angeführte Präprozessormethode mit einer klassischen objektorientierten und einer neuen, auf C++ Template-Techniken basierten Methode verglichen. Dabei wird davon ausgegangen, dass bereits mittels des Wrapper Facade Entwurfsmusters eine abstrakte Klassenschnittstelle definiert wurde.

4.1.2 Zusätzliche Anforderung an die Abstraktionsschicht

Bei der Beschreibung des Wrapper Facade Entwurfsmusters in [73] ist ein wesentliches Problem, das im Zusammenhang mit der gleichzeitigen Umsetzung einer abstrakten Klassenschnittstelle auf unterschiedliche Zielplattformen auftritt, nicht ausreichend berücksichtigt. Es handelt sich dabei um die Frage der Signatur der Methoden der Klassen der Wrapper Facade. Um eine vollständige Plattformunabhängigkeit zu erreichen, dürfen keine plattformspezifischen Typen bei den Argumenten einer Methode bzw. deren Resultat auftreten. Im Allgemeinen ist es jedoch nicht sinnvoll, die Klassenschnittstelle so zu entwerfen, dass bei den Signaturen der Methoden nur noch elementare C++ Typen, wie etwa `int` oder `char*`, vorkommen. Dies würde die Funktionalität einer einzelnen Methode zu stark beschränken und somit zu einer zu großen Anzahl von Methoden einer Schnittstelle für eine funktionale Gruppe führen. Die Klassenschnittstelle würde somit nur noch einen eingeschränkten Abstraktionscharakter aufweisen.

In diesem Zusammenhang sind zwei Arten von Typen zu unterscheiden:

- Typen für Objekte der Schnittstelle, die im Programm modifiziert werden. Für diese Fälle müssen, unabhängig vom gewählten Umsetzungsverfahren für das Wrapper Facade Entwurfsmuster, neue, für alle Zielplattformen einheitliche Typen eingeführt werden. In der Anpassungsschicht wird dann die Konvertierung zwischen diesen einheitlichen Typen und den entsprechenden Typen der einzelnen Betriebssystemschnittstellen vorgenommen.

- Typen für Objekte der Schnittstelle mit Transitcharakter.
  Transitcharakter bedeutet in diesem Zusammenhang, dass ein Objekt nicht vom Anwendungsprogramm modifiziert wird (zum Beispiel durch Methodenaufrufe). Solche Objekte werden also von einer Methode der abstrakten Schnittstelle als Rückgabewert geliefert und einer anderen Methode der abstrakten Schnittstelle unverändert als Argument übergeben. In diesem Fall sind einheitliche Typbezeichner, die mit der `typedef` Anweisung realisiert werden können, ausreichend.

Da der Umwandlungsprozess zwischen einheitlichen und plattformspezifischen Datenstrukturen oftmals einen erheblichen Zusatzaufwand darstellt, sollte, falls möglich und sinnvoll, mit Transitobjekten gearbeitet werden. Die nachfolgend beschriebenen Umsetzungsverfahren sollten also zusätzlich die Einführung von einheitlichen Typbezeichnern unterstützen und vereinfachen.

4.1.3 Realisierungsverfahren für die Anpassungsschicht

In diesem Abschnitt werden drei Verfahren für die Realisierung der Anpassungsschicht vorgestellt. Es handelt sich dabei um die auf bedingter Compilation beruhende Präprozessormethode, einer klassischen objektorientierten Methode, sowie einer neuen, auf Template Techniken basierten Methode. Zu jedem vorgestellten Realisierungsverfahren werden kurze Programmfragmente angegeben, die dessen wesentliche Charakteristika zeigen. Diese sind einem minimalen Anwendungsbeispiel entnommen, das aber alle wesentlichen Anforderungen an die Realisierungsverfahren beinhaltet. Bei dem betrachteten Beispiel wird davon ausgegangen, dass gemäß dem Wrapper Facade Entwurfsmuster eine abstrakte Schnittstelle `Abstract_Interface` entworfen wurde. Diese Schnittstelle kapselt die Methode `Y syscall(X)` und stellt zusätzlich die Typen für das Argument und Resultat von `syscall`, sowie eine Konstante `xconst` des Typs `X` zur Verwendung in Anwendungsprogrammen zur Verfügung.

Die kompletten Realisierungen der Umsetzungsverfahren für das Minimalbeispiel sind in Anhang B zu finden.

Zum besseren Vergleich werden die einzelnen Verfahren zur Realisierung der Anpassungsschicht ausschließlich in ihrer Reinform betrachtet. In der Praxis können natürlich auch beliebige Kombinationen der Verfahren eingesetzt werden, um deren jeweiligen spezifischen Vorteile gezielt ausnutzen zu können.

Präprozessormethode

In diesem Abschnitt soll zunächst die herkömmliche Realisierungsweise der Anpassungsschicht mittels Präprozessordirektiven zur bedingten Compilation diskutiert werden. Diese stellt eine direkte Umsetzung des Wrapper Facade Entwurfsmusters dar.

Das Design besteht im Wesentlichen aus den beiden folgenden Klassen:

- `Abstract_Interface`
  Diese Klasse realisiert die Abstraktionsschicht und stellt somit die Schnittstelle zur Anwendungsprogrammierung dar. Ihre Methoden kapseln Aufrufe der Methoden in der Anpassungsschicht.

- `Adaptation`
  Mittels dieser Klasse wird die Anpassungsschicht realisiert. In den Methoden befinden sich die jeweiligen systemabhängigen Implementierungen, die durch `#ifdef` Präprozessordirektiven voneinander separiert sind.

Beim Übersetzungsvorgang wird die für die jeweilige Ziel-Plattform passende Implementierung durch die Definition der entsprechenden Präprozessorkonstante ausgewählt.

Im Programmbeispiel 4.1 sind die oben beschriebenen Klassen dargestellt. Die Klasse `Adaptation` enthält keine Datenfelder, sie kapselt also keine Zustandsinformationen. Die Methoden sind deshalb alle als `static` deklariert um auf die Erzeugung einer Instanz der Klasse verzichten zu können.

Die beschriebene Präprozessormethode findet zum Beispiel bei der Implementierung von ACE Anwendung. Grundsätzlich könnten bei der Realisierung der Präprozessormethode die Abstraktionsschicht und die Anpassungsschicht auch innerhalb nur einer Klasse implementiert werden. Die gegebene Darstellung ist aber übersichtlicher, da sie die beiden funktionalen Aspekte auch auf der Implementierungsebene trennt. Sie folgt außerdem der bei der Implementierung von ACE gewählten Klassenanordnung.

Abstract Factory Entwurfsmuster

Der in diesem Abschnitt beschriebene Ansatz zur Realisierung der Anpassungsschicht verwendet das Abstract Factory Entwurfsmuster [34]. Die grundlegende Idee bei diesem Verfahren ist, die Abstraktionsschicht als Interface Klasse zu realisieren. Die Klasse `Abstract_Interface` enthält somit ausschließlich rein virtuelle Methoden. Die Anpassungsschicht wird gebildet, indem für jede zu unterstützende Plattform die Methoden des definierten Interface in einer eigenen, abgeleiteten Klasse (zum Beispiel `Adaptation_Sys_A`) implementiert werden. Die Erzeugung eines entsprechenden Objekts für eine konkrete Implementierung erfolgt im Anwendungsprogramm indirekt über die Methoden einer analog organisierten Hierarchie von Factory Klassen. Dabei wird die Erzeugung eines plattformspezifischen Implementierungsobjekts der Anpassungsschicht an eine abgeleitete Klasse delegiert. In Abbildung 4.3 ist die Struktur dieser Klassenhierarchien in UML Notation dargestellt.

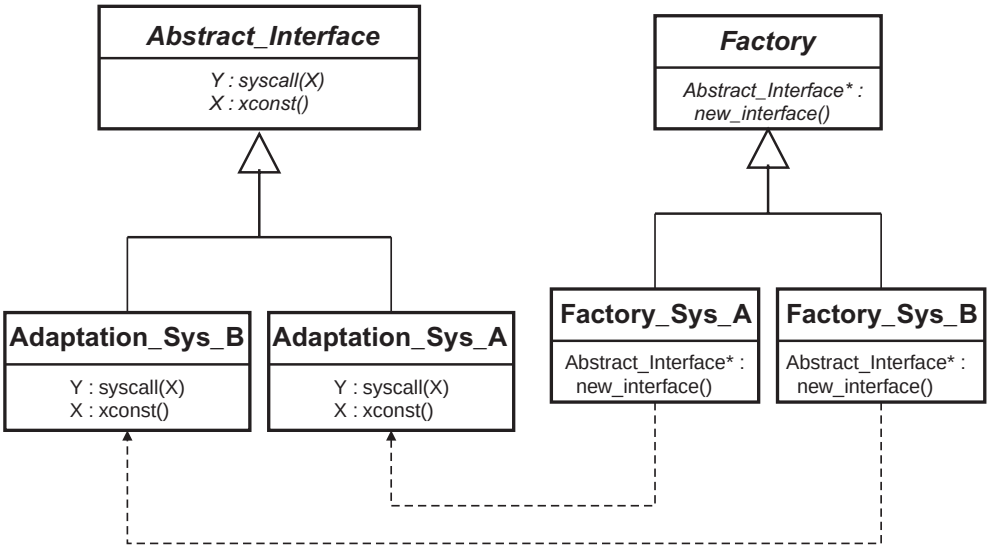
Im Programmbeispiel 4.2 wird die Verwendung der beschriebenen Klassen gezeigt.

Der in Anhang B dargestellte Programmcode weist auch Präprozessordirektiven auf. Im Gegensatz zur oben beschriebenen Präprozessormethode kommen diese jedoch nur an wenigen Stellen im Programmtext vor, im Wesentlichen um für die gewählte Plattform ein entsprechendes Factory-Objekt zu erzeugen. Dies ist unumgänglich, da grundsätzlich an einer Stelle im Programm die zu verwendende Plattform spezifiziert werden muss.

Programmbeispiel 4.1 Die Präprozessormethode zur Realisierung der Anpassungsschicht

```
#ifdef SYS_A
  typedef sysa_x_t X;
  typedef sysa_y_t Y;
#endif

#ifdef SYS_B
  typedef sysb_x_t X;
  typedef sysb_y_t Y;
#endif

class Adaptation {
public:
  static Y syscall(X arg) {

#ifdef SYS_A
  return ::syscall_A(arg);
#endif

#ifdef SYS_B
  return ::syscall_B(arg);
#endif

  }
};

class Abstract_Interface {
public:
  Y syscall(X arg) {
    return Adaptation::syscall(arg);
  }
};
```

Abbildung 4.3 Das Abstract Factory Entwurfsmuster



Template Methode

Der Template Mechanismus von C++ erlaubt es, Typen als Parameter für Klassen und Methoden anzugeben. Man spricht in diesem Zusammenhang auch von *generischem Programmieren*. Die Typparameter müssen dabei nicht aus einer gemeinsamen Vererbungshierarchie stammen.

Die grundlegende Idee bei diesem Ansatz besteht darin, die Abstraktionsschicht mit der Anpassungsschicht zu parametrisieren. Die Klasse `Abstract_Interface` wird somit als Template Klasse realisiert (siehe Programmbeispiel 4.3). Die Anpassungschicht bilden für jede zu unterstützende Plattform separat zu implementierende Klassen (zum Beispiel `Adaptation_Sys_A`), die als Template Parameter übergeben werden (siehe Programmbeispiel 4.4).

Die Methoden der Template Klasse verwenden zur Realisierung einer abstrakten Systemschnittstelle Methoden, Typen und Konstanten, die in der als Typparameter übergebenen Klasse definiert sind. Dieses Verfahren ähnelt stark der *Traits Technik* [67], die bei der Implementierung der C++ Standard Template Library (STL) in großem Umfang eingesetzt wurde.

Dieses Verfahren erlaubt neben der Homogenisierung von Methoden und Konstanten auch die konsistente und elegante Definition einheitlicher Typbezeichner. In der Klasse der abstrakten Schnittstelle können unter Verwendung des `typename` Schlüsselworts mittels einer Indirektion die in der jeweiligen Anpassungsschicht vorgenommene Vereinheitlichung der Typbezeichner auf die Ebe-

Programmbeispiel 4.2 Die Abstract Factory Methode zur Realisierung der Anpassungsschicht

```
class Abstract_Interface {
public:
  virtual X xconst() = 0;
  virtual Y syscall(X arg) = 0;
};

class Adaptation_Sys_A : public Abstract_Interface
{
public:
  X xconst() {
    return sysa_x_const;
  }

  Y syscall(X arg) {
    return ::syscall_A(arg);
  }
};
```

ne der Schnittstelle exponiert werden. Bei diesem Verfahren werden also Methoden, Konstanten und Typbezeichner auf einheitliche Weise behandelt.

4.1.4 Vergleichskriterien

Die im letzten Abschnitt vorgestellten Verfahren sollen nach den folgenden Kriterien verglichen werden.

Effizienz

Mit der Einführung einer zusätzlichen Abstraktions- und Anpassungsschicht sollten keine signifikanten Effizienzeinbußen verbunden sein, da sonst bei zeitkritischen Aufrufen die Kapselung umgangen werden muss und somit plattformspezifischer Code oberhalb der Abstraktionsschicht auftritt.

Portabilität

Bei dem Kriterium der Portabilität wird bewertet, wie einfach eine neue Ziel-Plattform integriert werden kann. Hierbei spielt vor allem der Umfang der

Programmbeispiel 4.3 Die Realisierung der Abstraktionsschicht bei der Template Methode

```
template<class Adaptation>
class Abstract_Interface {
public:
  // types
  typedef typename Adaptation::X X;
  typedef typename Adaptation::Y Y;

  // constants
  static const X xconst = Adaptation::xconst;

  // functions
  Y syscall(X arg) {
    return Adaptation::syscall(arg);
  }
};

#ifdef SYS_A
#include "Adaptation_Sys_A.h"
typedef Abstract_Interface<Adaptation_Sys_A> INTERFACE;
#endif

#ifdef SYS_B
#include "Adaptation_Sys_B.h"
typedef Abstract_Interface<Adaptation_Sys_B> INTERFACE;
#endif
```

neu hinzuzufügenden Programmteile und insbesondere auch die Lokalität und der Umfang der notwendigen Veränderungen an bestehendem Code der Anpassungschicht eine wesentliche Rolle. Plattformspezifische Unterscheidungen sollen in möglichst grob-granularer Form, d.h. in möglichst wenigen, zusammenhängenden Einheiten im Programmcode der Anpassungsschicht auftreten.

Einfache Überprüfung auf Vollständigkeit der Implementierung

Moderne Betriebssysteme bieten eine Vielzahl an Abstraktionen. Die entsprechenden Betriebssystemschnittstellen sind daher extrem umfangreich. Daselbe gilt folglich auch für die Abstraktions- und Anpassungsschicht.

Programmbeispiel 4.4 Die Realisierung der Anpassungsschicht bei der Template Methode

```
#include <sys/sysA.h>

class Adaptation_Sys_A {
public:
  // types
  typedef sysa_x_t X;
  typedef sysa_y_t Y;

  // constants
  static const X xconst = sysa_x_const;

  // functions
  static Y syscall(X arg) {
    return ::syscall_A(arg);
  }
};
```

Der Programmcode der Abstraktions- und der Anpassungsschicht wird üblicherweise für jede unterstützte Plattform als Programmbibliothek zur Verfügung gestellt, die später zum Anwendungscode hinzugelinkt wird. Hier soll bewertet werden, inwieweit es möglich ist, schon beim Übersetzen der Bibliothek Fehler zu finden, insbesondere, ob alle notwendigen Methoden der abstrakten Schnittstelle für eine bestimmte Plattform implementiert wurden.

4.1.5 Bewertung der Realisierungsmöglichkeiten

Vergleich der Verfahren bezüglich ihrer Effizienz

Bedingt durch die Einführung der Abstraktions- und Anpassungsschicht, kommt es beim Zugriff auf die Funktionalität des Betriebssystems zu mehreren Indirektionen bei Funktionsaufrufen, die prinzipbedingt die Hauptquelle für Effizienzverluste in allen drei vorgestellten Verfahren darstellen. Dieser Problematik kann in C++ durch die Verwendung der *Inlining* Technik begegnet werden. Hier wird vom Compiler für einen Funktionsaufruf kein entsprechender Unterprogrammaufruf auf Assemblerebene generiert, sondern es wird der betreffende Funktionsrumpf an der Stelle des Funktionsaufrufs direkt eingefügt. Bei der Präprozessormethode und der Template Methode lässt sich dieses Verfahren an-

wenden, und somit kann der Zusatzaufwand, der durch die Indirektionen bei Funktionsaufrufen entsteht, weitestgehend eliminiert werden.

Bei der Realisierung mit dem Abstract Factory Entwurfsmuster ist die Anwendung der Inlining Technik hingegen nicht möglich, da dieses Verfahren auf der Verwendung von virtuellen Methoden beruht. Da bei virtuellen Methoden erst zur Laufzeit der auszuführende Programmcode festgelegt wird *(late binding)* ist die oben beschriebene Einbettung des Programmcodes durch den Compiler prinzipiell nicht möglich.

Außerdem ist die Verwendung von virtuellen Methoden selbst mit zusätzlichen Effizienzeinbußen verbunden. Der Compiler setzt bei einem Methodenaufruf den Namen einer virtuellen Methode in einen Index in die so genannte *Virtual Function Table (vtbl)* des Betreffenden Objekts um [83]. Diese Tabelle enthält für jede virtuelle Methode des Objekts einen Funktionzeiger auf die tatsächlich auszuführenden Methode. Der Aufruf von virtuellen Methoden erfolgt also indirekt, es muss zunächst in einem zusätzlichen Schritt die Adresse der tatsächlich aufzurufenden Funktion ermittelt werden.

Beim Vergleich bezüglich der Effizienz ist also sowohl die Präprozessormethode, als auch die Template Methode gleichermaßen der Realisierung nach dem Abstract Factory Entwurfsmuster deutlich überlegen.

Vergleich der Verfahren bezüglich ihrer Portabilität

Die Präprozessor Methode weist keinerlei Lokalität von plattformspezifischem Programmcode auf. Wenn eine neue Ziel-Plattform hinzugefügt werden soll, sind Änderungen in verschiedensten Bereichen der Anpassungsschicht vorzunehmen, wobei die Stellen der Änderung durch keinerlei syntaktische Einheiten voneineinander abgetrennt sind.

Bei der Abstract Factory Methode kann die Implementierung der Methoden der Abstraktionsschicht in getrennten Unterklassen erfolgen. Das Konzept der Redefinition von Methoden in Unterklassen lässt sich jedoch nicht auf die Definition von Typen und Konstanten ausdehnen. Insbesondere dürfen sich die Argument- und Resultattypen einer Methode in einer abgeleiteten Klasse nicht von den Argument- und Resultattypen der entsprechenden virtuellen Methode in der Basisklasse unterscheiden. Das bedeutet, dass Typen und Konstanten, die ausschließlich in der Anpassungsschicht definiert wurden, nicht auf der Ebene der Abstraktionsschicht sichtbar sind. Für die Vereinheitlichung von Typbezeichnern ist somit ein anderes Verfahren anzuwenden, z.B. die Präprozessor Methode. Die geforderte Lokalität von plattformspezifischen Programmcode ist hierdurch aber verletzt (siehe oben). Einheitliche Konstanten können durch

zusätzlich eingeführte virtuelle Methoden realisiert werden, unter der Voraussetzung, dass der Typ der Konstante nach dem oben beschriebenen Verfahren zuvor homogenisiert wurde (siehe Programmbeispiel 4.2).

Die Template Methode bietet sowohl für Methoden, als auch für Typen und Konstanten eine einheitliche Definitionsmöglichkeit, die zudem die geforderte Lokalitätseigenschaft für die Integration von plattformspezifischem Programmcode vollständig erfüllt.

Hinsichtlich der Portabilität bietet die Template Methode gegenüber der Abstract Factory Methode und der Präprozessor Methode als einzige ein konsistentes Konzept zur Realisierung der modularen Erweiterung um weitere Ziel-Plattformen.

Vergleich bezüglich der einfachen Überprüfung auf Vollständigkeit

Bei der Präprozessor Methode kann vom Compiler das Fehlen der Implementierung einer Methode der Anpassungsschicht für eine bestimmte Zielplattform prinzipiell nicht in allen Fällen festgestellt werden. Dies ist lediglich dann der Fall, wenn die betreffende Methode einen Wert zurückgibt und die entsprechende `return` Anweisung Teil des plattformspezifischen Codes ist. Wenn bei der Implementierung einer Methode der Anpassungsschicht für eine Ziel-Plattform vergessen wurde entsprechenden plattformspezifischen Code anzugeben, fehlt auch die `return` Anweisung der Methode und es wird vom Compiler eine Fehlermeldung generiert.

Bei der Abstract Factory Methode werden nicht implementierte Methoden der Anpassungsschicht automatisch vom Compiler entdeckt und gemeldet. Die Abstraktionsschicht besteht aus einem Interface, in dem ausschließlich rein virtuelle Methoden deklariert sind. Wenn in einer abgeleiteten Klasse eine rein virtuelle Methode nicht implementiert wurde, bleibt diese rein virtuell. Von dieser Klasse kann somit keine Instanz erzeugt werden. Dieser Umstand wird vom Compiler erkannt und die für eine spezifische Plattform nicht implementierte Methode wird angezeigt. Als Vollständigkeitstest für Methoden reicht es also aus, für jede Zielplattform eine Instanz der entsprechenden Anpassungsschicht zu erzeugen. Die Überprüfung der Vollständigkeit der Typen kann vom Compiler nicht generell überprüft werden, ohne dass alle Typen explizit im Testprogramm referenziert werden.

Bei der Template Methode lässt sich durch einfaches Erzeugen einer mit der Anpassungsschicht der neuen Ziel-Plattform parametrisierten Instanz der Abstraktionsschicht überprüfen, ob alle plattformspezifischen Methoden, Typen und Konstanten für die neue Zielplattform vollständig angegeben wurden.

Hinsichtlich der einfachen Überprüfbarkeit auf Vollständigkeit der Implementierung bietet die Template Methode gegenüber den beiden anderen Verfahren ein einfaches und gleichzeitig alle wesentlichen Aspekte umfassendes Konzept.

Gesamtbewertung

Bei den drei vorgestellten Verfahren konnte nur die Template Methode allen aufgestellten Bewertungskriterien gerecht werden. Sie stellt somit hinsichtlich Effizienz, Portierbarkeit und Sicherstellung der Korrektheit ein geeignetes Verfahren zur Realisierung der Anpassungsschicht für das Wrapper Facade Entwurfsmuster dar. In [52] wird eine vollständige objektorientierte Kapselung des Socket API und deren Umsetzung mittels der Template Methode dargestellt.

Für die Implementierung von DOTS wurde aus pragmatischen Gründen jedoch auf ACE zurückgegriffen, das eine objektorientierte Schnittstelle für alle relevanten Betriebssystemkonzepte und -abstraktionen für eine Vielzahl von Plattformen bietet.

4.2 Realisierung der Kommunikationskomponente

In diesem Abschnitt soll die Realisierung der Kommunikationskomponente von DOTS diskutiert werden.

4.2.1 Anforderungen an die Kommunikationskomponente

Da die Kommunikationskomponente einen wesentlichen Kernteil des gesamten Systems darstellt, muss sie einer Vielzahl von Anforderungen gerecht werden. Für diese zentrale Komponente von DOTS lassen sich die folgenden Anforderungen identifizieren:

- **Effizienz der Kommunikation**
  Im Allgemeinen ist das Ziel beim parallelen verteilten Rechnen die möglichst optimale Beschleunigung der Gesamtberechnung unter Verwendung von mehreren Prozessoren, d.h. das Erreichen eines möglichst hohen Grades an paralleler Effizienz. Die Realisierung eines effizienten Kommunikationskanals zwischen den einzelnen Prozessoren kann je nach Kommunikationsintensität und -muster der betrachteten Anwendung wesentlich zur Optimierung der parallelen Effizienz der Gesamtberechnung beitragen.

Grundsätzlich wird die Effizienz der Kommunikation durch zwei unterschiedliche Faktoren bestimmt; einerseits durch eine möglichst hohe Bandbreite (bzw. Durchsatz) und andererseits durch eine möglichst kleine Latenzzeit. Welcher dieser beiden Aspekte eine größere Auswirkung auf die Optimierung der parallelen Effizienz hat, hängt stark von der Kommunikationscharakteristik der betrachteten Anwendung ab.

Zum Beispiel spielt bei extrem fein-granularen verteilten parallelen Berechnungen die Optimierung der Latenzzeit des Nachrichtenaustauschs eine wesentliche Rolle. Eine geringe Latenzzeit bewirkt, dass schneller parallele Vorgänge auf entfernten Knoten angestoßen werden können und somit die parallele Effizienz der Berechnung verbessert wird.

- **Zuverlässigkeit der Kommunikation**
  Anders als bei verteilten Multimedia-Anwendungen, wie etwa der Übertragung von Audio- oder Videodaten, ist beim verteilten parallelen Rechnen eine zuverlässige Kommunikation erforderlich. Je nach verwendeter Netzwerkhardware und eingesetzten Netzwerkprotokollen gibt es unterschiedliche Ursachen für das Verlorengehen von Nachrichten. Zum Beispiel können beim IP Protokoll durch Überlastung von Zwischenstationen (so genannte Router) oder durch Überlauf von Empfangspuffern Datenpakete verworfen werden und damit verloren gehen.

  Die Zuverlässigkeit des Nachrichtenaustauschs kann durch zusätzlichen Protokollaufwand sichergestellt werden, indem zum Beispiel dem Sender der Empfang von Daten durch Bestätigungsnachrichten angezeigt wird und der Sender bei Ausbleiben dieser Bestätigung nach einer bestimmten Zeit eine erneute Übertragung der Daten vornimmt.

  Bei DOTS soll der Nachrichtenaustausch zwischen den an der Berechnung beteiligten Knoten für den Programmierer transparent sein. In diesem Fall muss also die zugrundeliegende Kommunikationskomponente des Systems eine zuverlässige Kommunikation sicherstellen.

- **Kommunikation in heterogenen Umgebungen**
  Bei heterogenen Umgebungen treten weitere Anforderungen bezüglich der Effizienz an die Kommunikationskomponente des Systems auf. Unterschiedliche Datenformate müssen auf effiziente Art und Weise konvertiert werden. Zusätzlich müssen auch Rechner mit unterschiedlicher Leistungsfähigkeit effizient kommunizieren können.

4.2.2 Realisierungsmöglichkeiten

Zur Realisierung der oben aufgezeigten Anforderungen gibt es eine Vielzahl von Ansatzpunkten. Vor allem zur Verbesserung der Effizienz der Kommunikation wird häufig spezielle Hardware, zusammen mit eigens entwickelten Kommunikationsprotokollen, eingesetzt. Ein aktuelles Beispiel hierfür ist die SCI *(Scalable Coherent Interface)* Technologie [40]. Durch SCI wird ein cache-kohärenter gemeinsamer Speicherraum zwischen den Knoten eines Clusters zur Verfügung gestellt. Dieser gemeinsame Speicherbereich kann direkt im Programm adressiert werden oder es kann mit speziell angepassten Message Passing Bibliotheken eine effiziente Kommunikation über expliziten Nachrichtenaustausch realisiert werden.

Für den Einsatz in heterogenen Umgebungen sind derartige Verfahren nicht oder nur äußerst begrenzt geeignet, da die benötigte spezielle Hardware oft nur für eine bestimmte bzw. bestenfalls für wenige Plattformen verfügbar ist.

Im heterogenen Umfeld muss also auf standardisierte Hardware und insbesondere auf standardisierte Protokolle zurückgegriffen werden. Mit dem ISO/OSI Modell steht ein derartiger Kommunikationsstandard zur Verfügung. In der Praxis gibt es aber kaum Implementierungen des ISO/OSI Modells. Ihm gegenüber hat sich die mit TCP/IP bezeichnete Protokollfamilie zur Kommunikation in Rechnernetzwerken durchgesetzt.

Das TCP/IP Protokoll ist für nahezu alle Plattformen verfügbar und unterstützt viele verschiedene Typen von Netzwerkhardware. Zum Beispiel wird die Kommunikation der im Internet verbundenen Rechner mittels des TCP/IP Protokolls realisiert. Es hat sich somit ideal zur Kommunikation zwischen vielen verschiedenen Typen von Rechnern erwiesen. Da für DOTS die Verwendbarkeit in stark heterogenen Umgebungen eine wichtige Anforderung darstellt, wurde die Kommunikationskomponente auf dem TCP/IP Protokoll basiert.

Bei der Verwendung der TCP/IP Protokollfamilie besteht grundsätzlich die Auswahl, ob UDP *(User Datagram Protocol)* oder aber TCP *(Transmission Control Protocol)* zum Austausch von Daten verwendet werden soll. Diese beiden Protokolle unterscheiden sich in wesentlichen Punkten. Während das UDP Protokoll verbindungslos und unzuverlässig ist, realisiert das TCP Protokoll zusätzlich eine verbindungsorientierte und zuverlässige Kommunikation. Hierfür werden im TCP Protokoll Sequenznummern für Datenpakete eingeführt, die zusammen mit Bestätigungsnachrichten die Zuverlässigkeit der Kommunikation sicherstellen. Zusätzlich werden bei TCP zur Verbesserung der Effizienz Protokolle zur Flusskontrolle eingesetzt Als Beispiel ist hier das so genannte *Sliding Window Protocol* zu nennen. Mit diesem Protokoll wird verhindert, dass der Sender mehr Daten schickt als, im Puffer des Empfängers gehalten werden können. Somit

kommen erneute Datenübertragungen aufgrund von Pufferüberläufen bei Verwendung von TCP nicht vor.

Neben der Effizienz ist die Realisierung von zuverlässiger Kommunikation eine weitere zentrale Anforderung an die Kommunikationskomponente von DOTS. Um in TCP/IP eine zuverlässige Kommunikation zu verwirklichen ergeben sich somit zwei unterschiedliche Vorgehensweisen. Zum einen kann direkt das TCP Protokoll verwendet werden, das Zuverlässigkeit bereits zur Verfügung stellt, zum anderen kann auf dem UDP Protokoll aufbauend durch zusätzlichen Implementierungsaufwand eine zuverlässige Kommunikation realisiert werden.

Aus den folgenden Gründen wurde die Kommunikationskomponente von DOTS nach der erstgenannten Vorgehensweise, also unter Verwendung von TCP, verwirklicht:

- Um UDP so zu erweitern, dass eine zuverlässige Kommunikation möglich ist, müsste ein großer Anteil an Funktionalität, die in TCP bereits vorhanden ist, nachimplementiert werden. Da die aktuellen Implementierungen von TCP weitestgehend ausgereift und für die jeweilige Plattform hochoptimiert sind, ist aber eine solche teilweise Neuimplementierung von bereits vorhandener Funktionalität nicht sinnvoll [82].
- Der bei der Verwendung von TCP im Vergleich zu UDP zusätzlich auftretende Protokollaufwand entsteht einerseits durch zusätzliche Softwareschichten, andererseits durch zusätzlichen Nachrichtenaustausch. Der durch Software verursachte Zusatzaufwand kann in zunehmendem Maße vernachlässigt werden, da die Entwicklung der CPU Leistung viel schneller voranschreitet als die Entwicklung der Übertragungsleistung von Netzwerkhardware [57]. Effizienzeinbußen, die durch zusätzlichen Nachrichtenaustausch entstehen, können durch geeignete Maßnahmen stark vermindert werden. In den folgenden Abschnitten wird beschrieben, wie dies bei der Kommunikationskomponente von DOTS erreicht wird.

4.2.3 Connection Caching

Connection Caching ist zurzeit Gegenstand umfangreicher theoretischer Untersuchungen. Cohen et al. [23] führten ein graphentheoretisches Modell für das Connection Caching ein. Dieses dient als Grundlage weiterer theoretischer Untersuchungen zur effizienten Implementierung von Connection Caches unter bestimmten Rahmenbedingungen, wie etwa unterschiedlicher Kosten für den Verbindungsaufbau [2].

In diesem Abschnitt soll nun eine praktische Umsetzung dieses Verfahrens zur Optimierung verbindungsorientierter Kommunikation vorgestellt werden. Es

werden zunächst die relevanten protokoll- und systemtechnischen Fragestellungen ausführlich diskutiert, anschließend wird die Realisierung des Connection Caches des Kommunikationskerns von DOTS beschrieben.

Technische Motivation für Connection Caching

Die Ausführungen in diesem Abschnitt beschreiben den Zugriff auf die Funktionalität von TCP mittels den Primitiven des in der UNIX Welt inzwischen weit verbreiteten Stream Socket API. Wie in Abschnitt 4.1 ausgeführt ist, können dafür abstraktere, objektorientierte Programmierschnittstellen verwendet werden. Dieses Vorgehen ermöglicht auch die transparente Abbildung auf eine andere Systemschnittstelle zum Zugriff auf TCP, wie zum Beispiel dem WinSock API der Microsoft Windows Betriebssysteme [3] oder dem X/Open Transport Interface (XTI) [82], das in den Unix98 Standard neben der Socket-Schnittstelle aufgenommen wurde.

Üblicherweise wird zum Öffnen einer TCP Verbindung, der anschließenden Übertragung von Daten und dem Schließen der Verbindung die folgende Sequenz von Systemaufrufen benötigt:

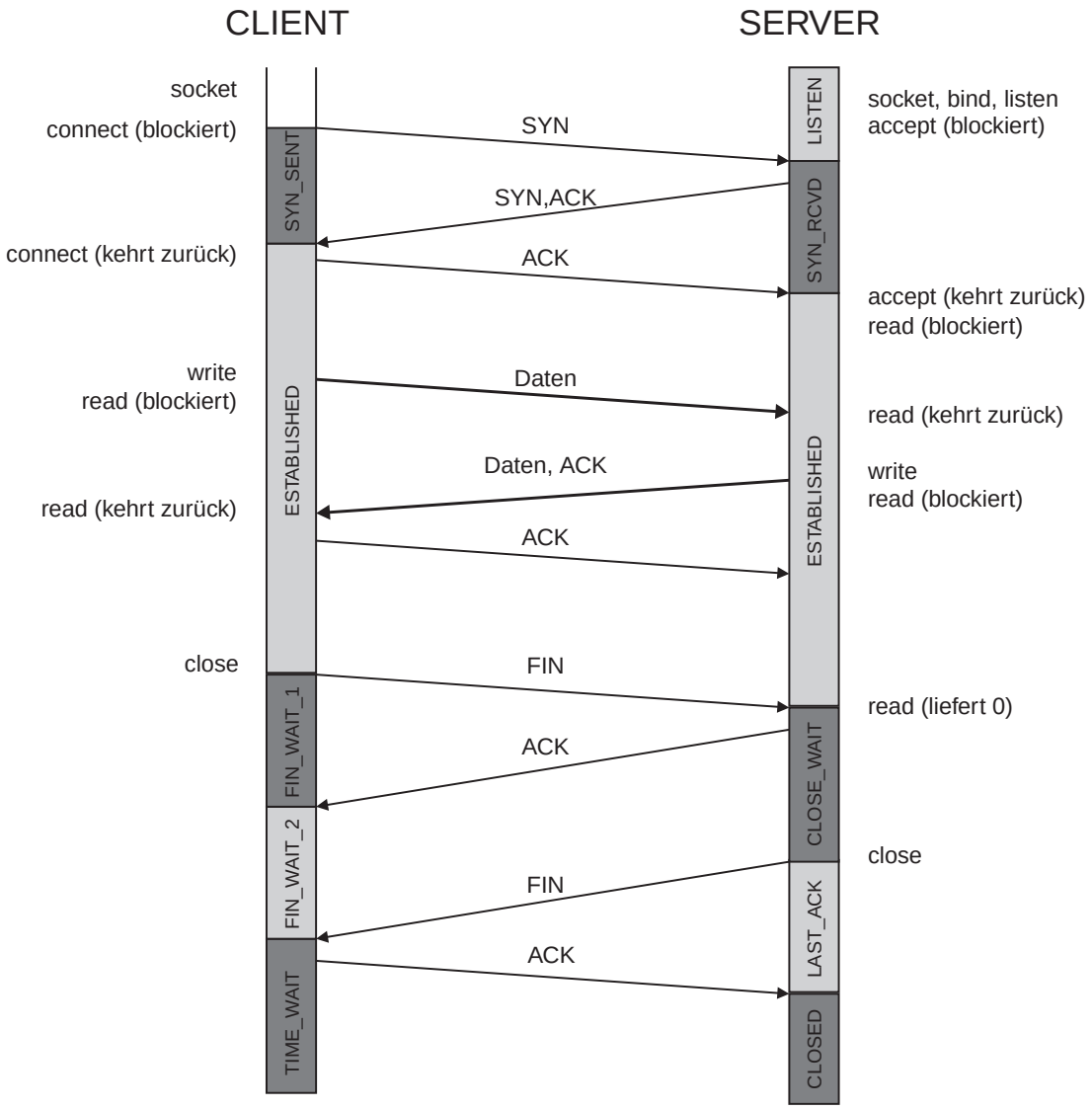
- Aktiver Kommunikationspartner (Client):
  1. Aktives Öffnen der Verbindung: `socket`, `connect`
  2. Datenaustausch: `write`, `read`
  3. Schließen der Verbindung: `close`
- Passiver Kommunikationspartner (Server):
  1. Passives Öffnen der Verbindung: `socket`, `bind`, `listen`, `accept`
  2. Datenaustausch: `read`, `write`
  3. Schließen der Verbindung: `close`

Im Zusammenhang mit der Kommunikationsstruktur bei DOTS sind die Begriffe Server und Client nicht als permanentes Charakteristikum eines Knotens, sondern als temporäre Rollen zu verstehen, die ein Knoten im Verlauf einer verteilten Berechnung einnimmt. In den meisten Fällen wechselt die Rolle eines Knotens ständig während einer Berechnung.

Wird bei jedem Nachrichtenaustausch nach dem oben dargestellten Schema eine entsprechende Verbindung neu aufgebaut und nach Beendigung des Datentransfers wieder geschlossen, treten die nachfolgend diskutierten Effekte auf, die sich in vielen Fällen negativ auf die Effizienz des Datenaustauschs auswirken können:

- Bei TCP sind die zu übertragenden Daten in Einheiten gruppiert, die als *Segmente* bezeichnet werden. In Abbildung 4.4 wird der Austausch von TCP Segmenten dargestellt, der sich aus der oben beschriebenen Abfolge von Systemaufrufen auf der Client- und Serverseite ergibt.

Abbildung 4.4 Der Austausch von Segmenten bei einer TCP Verbindung



Die Abbildung zeigt auch die im Verlauf eingenommenen Zustände im TCP Zustandsübergangsdiagramm. Eine detaillierte Beschreibung der ein-

zelnen Zustände einer Verbindung ist zum Beispiel in [81] zu finden.

Zum Öffnen der Verbindung werden 3 TCP Segmente ausgetauscht *(TCP 3-way handshake)*. Der Client schickt zum Server ein SYN *(synchronize)* Segment. Dieses enthält im Wesentlichen eine initiale Sequenznummer für die Datenübertragung. Der Server sendet daraufhin auch ein entsprechendes SYN Segment. Dieses Segment dient gleichzeitig auch als ACK *(acknowledge)* Segment zur Bestätigung des Empfangs des SYN Segments des Clients. Der Client antwortet nach dessen Empfang seinerseits mit einem ACK Segment.

Beim Schließen der Verbindung erfolgt ein Austausch von 4 weiteren Segmenten. Beide Knoten senden beim Aufruf des `close` Systemaufrufs ein FIN *(finish)* Segment zum anderen Kommunikationspartner, der den Empfang mit einem ACK Segment bestätigt. Eine Gruppierung eines FIN und eines ACK Segmentes in ein gemeinsames Segment (analog zum Öffnen der Verbindung) ist in diesem Fall nicht möglich, da die beiden `close` Systemaufrufe nicht zwingend zeitlich aufeinanderfolgen müssen.

Die maximale Länge der Daten, die in einem TCP Segment übertragen werden können, hängt vom Übertragungsmedium ab. Zum Beispiel muss bei Ethernet aufgrund physikalischer Gegebenheiten die Länge eines Rahmens *(Ethernet Frame)* zwischen 46 Bytes und 1500 Bytes betragen. Nach Abzug von jeweils 20 Byte für den IP bzw. TCP Header ergibt sich in diesem Fall für die maximale Länge der Daten 1460 Bytes. Diese Information wird beim Verbindungsaufbau in den SYN Segmenten zwischen den Kommunikationspartnern als so genannte Maximum Segment Size (MSS) ausgetauscht.

Kleinere Nachrichten können also in einem einzigen TCP Segment transportiert werden. Für den eigentlichen Datenaustausch sind dann nur 2 Segmente erforderlich. Im ersten Segment werden die Daten gesendet, das zweite ist ein ACK Segment zur Bestätigung des Empfangs. Durch das oben beschriebene Protokoll zum Öffnen und Schließen der Verbindung erhöht sich aber die Zahl der tatsächlich übertragenen Segmente um weitere 7 Segmente. Hieraus resultiert somit insbesondere bei Nachrichten mit kleiner Länge eine erhebliche Erhöhung des Übertragungsaufwands, die sich vor allem in einer größeren Latenz des Nachrichtenaustauschs bemerkbar macht.

- Ein Verbindungsendpunkt ist im TCP Protokoll durch das Paar *(IP-Adresse, Port Nummer)* gegeben. Gelegentlich wird dieses Paar – in Anlehnung an Steckdosen als End- bzw. Zugriffspunkte für das elektrische Stromnetz – auch als *Socket* bezeichnet. Eine Verbindung wird für die Dauer ihres

Bestehens daher eindeutig durch ein Paar von Sockets beschrieben. Für jedes Netzwerkinterface eines Rechners wird eine feste IP Adresse vergeben. Im Regelfall besitzen Rechnern nur ein Netzwerkinterface, so dass die IP Adressen der beiden Sockets implizit festgelegt sind. Die Portnummer der Serverseite wird vom Programmierer gewählt und muss auf der Clientseite zur Adressierung des Servers bekannt sein. Sie bleibt mindestens während der gesamten Programmausführung unverändert. Die Portnummer des Clients wird dagegen in der Regel vom System bestimmt, indem eine freie Portnummer ausgesucht wird. Da diese Portnummer nur für die Dauer des Bestehens der Verbindung genutzt wird, spricht man von einem kurzlebigen oder auch flüchtigen Port *(Ephemeral Port)*.

Der letzte Zustand, in dem sich der Client-Socket beim Schließen einer Verbindung befindet ist TIME_WAIT (siehe Abbildung 4.4). Dieser Zustand wird je nach Implementierung für eine Zeitdauer von etwa 60 – 480 Sekunden eingenommen. Dies ist notwendig, da das letzte vom Client gesendete ACK Segment verloren gehen kann. In diesem Fall wird vom Server erneut ein FIN Segment übertragen, das wiederum vom Client bestätigt werden muss.

Während der Zeitdauer, in der der Socket sich im TIME_WAIT Zustand befindet, bleibt jedoch die Nummer des flüchtigen Ports noch gebunden, da der Schließvorgang der Verbindung noch nicht vollständig abgeschlossen ist. Das bedeutet, dass in dieser Zeitspanne die Portnummer noch nicht wiederverwendet werden kann. Wenn in einem kurzem Zeitraum viele Verbindungen auf und wieder abgebaut werden, kann es vorkommen, dass vorübergehend keine freien Portnummern für flüchtige Ports mehr vorhanden sind und dadurch für eine verhältnismäßig lange Zeit die gesamte Kommunikation auf einem Knoten blockiert wird.

- Um eine Optimierung der Kommunikation über die Grenzen des lokalen Netzwerkes hinweg zu erreichen, wenden TCP Implementierungen den so genannten *Slow Start* Algorithmus an. Ziel bei diesem Verfahren ist es, eine Überlastung von Zwischenstationen (so genannte Router) zu verhindern. Die grundlegende Idee besteht darin, die Senderate von TCP Segmenten an die Empfangsrate der entsprechenden ACK Segmente zu koppeln. Vereinfacht dargestellt funktioniert der Slow Start Algorithmus wie folgt: Wenn eine Verbindung neu hergestellt wird, kann zunächst nur ein TCP Segment verschickt werden. Trifft das entsprechende ACK Segment ein, können zwei weitere TCP Segmente versendet werden. Wenn für diese zwei Segmente die Bestätigungssegmente eingetroffen sind, können 4 weitere Segmente verschickt werden, usw. Die Senderate für TCP Seg-

mente steigt also exponentiell mit dem Empfang der entsprechenden ACK Segmente. Wenn eine Überlastung auftritt, also Pakete verworfen werden und damit die entsprechenden ACK Segmente ausbleiben, wird die Senderate wieder reduziert.

Bei der ersten Verwendung einer Verbindung tritt durch den oben beschriebenen Slow Start Algorithmus beim Versenden einer Nachricht, die aus mehreren Segmenten besteht, eine erhöhte Latenzzeit auf, da auf das Eintreffen der ACK Segmente gewartet werden muss.

Die oben dargestellten Probleme machen es also erforderlich, dass zur Erzielung einer effizienten Kommunikation TCP Verbindungen nicht nach einmaligem Nachrichtenaustausch geschlossen, sondern mehrfach wiederverwendet werden sollten. Aus den folgenden Gründen ist es aber nicht möglich, gleichzeitig eine größere Zahl von Verbindungen geöffnet zu halten:

- Mit jedem Verbindungsendpunkt ist ein so genannter Socket Deskriptor verbunden, der auf Programmebene zur Identifizierung der Verbindung dient. Die maximale Anzahl von Deskriptoren ist bei vielen Systemen jedoch begrenzt. Zwar liegen diese Grenzen bei den aktuellen Betriebssystemen relativ hoch (1024 oder oftmals auch mehr), dieser Faktor muss aber trotzdem berücksichtigt werden, um die grundsätzliche Portierbarkeit von DOTS zu gewährleisten.
- Für jeden Verbindungsendpunkt wird vom Betriebssystemkern ein Sende- und Empfangspuffer für Daten verwaltet. Bei großen Puffergrößen und vielen Verbindungen kann ein beträchtlicher Bedarf an Kernspeicher entstehen, dessen Seiten nicht ausgelagert werden können.
- Wenn mehrere Verbindungen offen gehalten werden, ist es erforderlich, dass gleichzeitig an mehreren Endpunkten blockierend auf ankommende Daten gewartet wird. Zwar stellen die Betriebssysteme hierfür entsprechende Systemaufrufe zur Verfügung (unter UNIX zum Beispiel `select` bzw. `poll`), die Anzahl der Deskriptoren, die mit diesen Systemfunktionen gleichzeitig überwacht werden können, ist aber beschränkt. Bei den Microsoft Windows Betriebssystemen liegt diese Beschränkung zum Beispiel bei 64 Verbindungen.

Grundlegende Abläufe und Anforderungen beim Connection Caching

Das Ziel des Connection Caching ist es, eine möglichst große Zahl von TCP Verbindungen gleichzeitig geöffnet zu halten, um den Aufwand, der durch das

Öffnen und Schließen von Verbindungen entsteht, zu verringern. Die Größe des Connection Caches, also die Zahl der Verbindungen, die gleichzeitig auf einem Knoten offen gehalten werden, wird durch die oben beschriebenen Ressourcenbeschränkungen bestimmt und ist somit ein systemabhängiger Parameter.

Der grundlegende Ablauf beim Connection Caching ist analog zur Funktionsweise von Cache Speichern bei Speicherhierarchien [41]. Bei einem Speicher Cache wird ein Datenwort aus dem Cache angefordert; dieses kann entweder direkt aus dem Cache gelesen werden, oder es wird erst aus dem Hauptspeicher transparent in den Cache geladen und weitergereicht. Analog dazu wird eine Verbindung vom Connection Cache angefordert. Dabei ist transparent, ob diese bereits im Cache vorhanden ist, oder erst (zeitaufwändig) neu aufgebaut und in den Cache eingetragen werden muss.

In beiden Fällen sind die Plätze für Cache Einträge beschränkt, so dass Einträge nach einem bestimmten Verfahren aus dem Cache gelöscht werden müssen, um für neue Einträge Platz zu schaffen. Da aber die beiden Endpunkte einer Verbindung auf unterschiedlichen Knoten lokalisiert sind, ist eine Verbindung in zwei Cache Speichern auf verschiedenen Knoten einzutragen und zu einem späteren Zeitpunkt ggf. wieder zu löschen. Dieser Umstand macht eine komplexere Verwaltung eines Connection Caches erforderlich, als dies bei Speicher Caches der Fall ist.

Für die Realisierung des Connection Caches der Kommunikationskomponente von DOTS stand neben der Effizienz vor allem die Transparenz als Entwurfsziel im Vordergrund. Die Anforderung der Transparenz umfasst die beiden folgenden Aspekte:

- Zur Verwaltung des Caches soll keinerlei Kenntnis über eine Kommunikationsstruktur vorausgesetzt werden. Das bedeutet zum Beispiel, dass kein Wissen vorhanden ist, ob und zu welchen Zeitpunkten über ein Verbindung, die sich im Cache befindet, Daten übertragen werden.
- Bei der Verwendung des Connection Caches soll dessen interne Realisierung nicht berücksichtigt werden müssen, zum Beispiel sollte eine Verbindung während ihrer Verwendung nicht geschlossen werden.

Schnittstelle zum Connection Cache

Der Connection Cache der Kommunikationskomponente von DOTS stellt die folgende Schnittstelle zur Verwaltung von Verbindungen zur Verfügung:

- `Stream request_send_stream(Node_ID ID)`
  Hiermit wird eine Stream-Verbindung vom Connection Cache zum Knoten

mit der angegebenen ID angefordert und zur Datenübertragung für den Anforderer reserviert. Eine Reservierung ist notwendig, damit eine Verbindung nicht gleichzeitig von mehreren internen Threads des Systems zur Übertragung verwendet wird wodurch es zu einer Vermischung der gesendeten Daten käme.

- `release_send_stream(Stream s)`
  Die Reservierung einer Verbindung wird aufgehoben, so dass sie von anderen Threads angefordert oder ggf. aus dem Cache entfernt werden kann.

- `Stream request_receive_stream()`
  Eine eingehende Verbindung anfordern und ggf. blockieren bis eine Verbindung Daten zum Lesen bereithält. Wenn Daten an einer Verbindung ankommen, wird diese zum Lesen für den Anforderer reserviert.

- `release_receive_stream(Stream s)`
  Aufheben der Reservierung einer Verbindung, nachdem alle benötigten Daten aus ihr gelesen wurden. Die betreffende Verbindung wird dann wieder auf das Eintreffen weiterer Daten überwacht.

Interner Aufbau und Realisierung des Connection Caches

Zur Vereinfachung der Verwaltung des Connection Caches und insbesondere zur Verwirklichung der Transparenzanforderung werden Verbindungen nur zur unidirektionalen Kommunikation verwendet. Hierdurch kann zum Löschen von Verbindungen ein wesentlich einfacheres Protokoll angewendet werden (siehe weiter unten).

Der Connection Cache wird somit durch zwei getrennt verwaltete Cache Speicher mit unterschiedlicher Funktionalität realisiert: Ein Cache für ausgehende Verbindungen *(Send Cache)*, sowie ein Cache für eingehenden Verbindungen *(Receive Cache)*. Ein Endpunkt einer Verbindung ist im Send Cache eingetragen, während der andere Endpunkt der Verbindung im Receive Cache des Kommunikationspartners gespeichert ist.

Realisierung der Schnittstelle zum Connection Cache

Wenn mittels einem Aufruf von `request_send_stream` eine ausgehende Verbindung angefordert wird, wird diese Anfrage an den Send Cache weitergeleitet. Ist eine Verbindung zum gewünschten Knoten vorhanden *(Cache Hit)*, so wird diese zurückgegeben, ansonsten wird zunächst eine entsprechende Verbindung aufgebaut *(Cache Miss)*.

Auf allen Verbindungen des Receive Caches wird gleichzeitig auf eingehende Daten blockierend gewartet, wenn eine `request_receive_stream()` Anforderung

vorliegt. Es wird diejenige Verbindung zurückgegeben, bei der als erste Daten ankommen. Sind mehrere Verbindungen zum Lesen bereit, wird diejenige ausgewählt, von der am Längsten nicht mehr gelesen wurde. Durch dieses Round Robin Verfahren wird verhindert, dass Daten aus einer Verbindung überhaupt nicht verarbeitet werden *(Connection Starvation)*.

Aufbau der Einträge im Send und Receive Cache

Ein Eintrag im Receive Cache besteht aus den folgenden Feldern:

- Descriptor
  Hier ist der Deskriptor eingetragen, der den Verbindungsendpunkt auf Betriebssystemebene repräsentiert.
- Lock
  Dieses Feld enthält eine Lock Variable, mit der der exklusive Zugriff auf eine Verbindung realisiert wird.
- Valid Flag
  Mittels dieses Feldes kann festgestellt werden, ob der Eintrag gültig ist.
- Time Stamp
  In diesem Feld wird der Zeitpunkt gespeichert, zu dem der Eintrag zuletzt verwendet wurde.

Einträge im Send Cache enthalten zusätzlich noch das folgende Feld:

- Address
  Dieses Feld gibt die IP Adresse des Knotens an, zu dem die Verbindung des Eintrags aufgebaut ist.

Entfernen von Verbindungen

Falls vom Send Cache eine Verbindung zu einem Knoten angefordert wird, die nicht im Cache vorhanden ist, wird diese zunächst aufgebaut und in beiden beteiligten Caches eingetragen. Dabei kann der Fall eintreten, dass in einem (oder auch in beiden) der beteiligten Cache Speicher kein Platz mehr für einen weiteren Eintrag vorhanden ist. In diesem Fall wird zunächst aus dem betroffenen Cache Speicher eine Verbindung zum Schließen ausgewählt und anschließend in jeweils beiden beteiligten Cache Speichern gelöscht. Es können also bis zu 4 Knoten bei dieser Aktion beteiligt sein.

Zur Auswahl einer Verbindung, die aus dem Cache gelöscht werden soll, wird die *Least Recently Used (LRU)* Strategie angewendet, d.h. es wird diejenige Verbindung entfernt, die am längsten unbenutzt ist. Dazu wird das Time Stamp Feld der einzelnen Cache Einträge ausgewertet.

Für das Löschen von Einträgen aus dem Send Cache und dem Receive Cache sind unterschiedliche Verfahren erforderlich.

- **Aktives Schließen**
  Muss in einem Send Cache eine Verbindung gelöscht werden, wird diese einfach mittels `close()` geschlossen und das Valid Flag des entsprechenden Eintrags auf FALSE gesetzt. Im beteiligten Receive Cache kann beim nächsten Aufruf von `request_receive_stream()` erkannt werden, dass die betreffende Verbindung geschlossen ist. Dazu wird, wenn eine zum Lesen bereite Verbindung erkannt wurde, zunächst versucht mittels `read()` ein Byte aus der Verbindung zu lesen. Ist dies nicht möglich (d.h. `read()` liefert den Wert 0 zurück), kann die Verbindung mit `close()` geschlossen werden und der entsprechende Eintrag im Receive Cache durch Setzen des Valid Flags auf FALSE freigegeben werden. Ansonsten wird das gelesene Byte wieder zurückgestellt und die Verbindung dem Anforderer ausgeliefert.

- **Passives Schließen**
  Falls aus dem Receive Cache eine Verbindung gelöscht werden muss, ist zunächst sicherzustellen, dass auf dieser keine Daten mehr gesendet werden. Dazu wird entgegen der eigentlichen Kommunikationsrichtung ein Byte übertragen. Dies kann beim beteiligten Send Cache leicht festgestellt werden, indem auf allen Verbindungen gleichzeitig auf ankommende Daten gewartet wird. Da Verbindungen nur zur unidirektionalen Kommunikation verwendet werden, ist implizit klar, dass es sich um eine Anforderung zum Schließen der Verbindung handelt. In diesem Fall wird einfach das im ersten Punkt beschriebene Protokoll zum aktiven Schließen der Verbindung gestartet.

4.2.4 Verbesserung des Durchsatzes mittels Pufferanpassung

In TCP wird durch das *Sliding Window Protokoll* eine Flusskontrolle zwischen dem Sender und Empfänger von Daten realisiert. Der Sender schickt nicht mehr Daten, als auf Empfängerseite im Empfangspuffer aufgenommen werden können. Dieses Vorgehen verhindert, dass der Empfangspuffer überläuft und somit ankommende TCP Segmente verworfen und aufwändig neu übertragen werden müssten.

Um in heterogenen Netzwerken eine effiziente Übertragung von großen Datenmengen zu gewährleisten, ist eine Anpassung der Größen für die Sende- und Empfangspuffer wichtig. Schnelle Rechner sollten über einen größeren Sendepuffer und langsamere Rechner über einen hinreichend großen Empfangspuffer

verfügen, um Verzögerungen, die durch das Sliding Window Protokoll von TCP hervorgerufen werden, zu minimieren.

4.2.5 Effiziente Datenkonversion

Heterogene Rechnernetzwerke sind auch dadurch gekennzeichnet, dass die beteiligten Rechner unterschiedliche Formate zur Datenrepräsentation verwenden. Zum Beispiel finden für Zeichensätze unterschiedliche Codierungen Verwendung. Neben der weit verbreiteten ASCII Codierung ist vor allem im Großrechnerbereich die EBCDIC Codierung anzutreffen. Auch unterscheiden sich Prozessorarchitekturen durch die Reihenfolge, in der die einzelnen Bytes eines Datenwortes im Speicher abgelegt sind. Man unterscheidet hier das Big Endian und das Little Endian Format. Schließlich verwenden unterschiedliche Rechnerarchitekturen verschiedene Bitbreiten für die interne Darstellung von Ganzzahldatentypen.

Die Übertragung von Daten über das Netzwerk erfolgt grundsätzlich byteweise, so dass Typinformationen verlorengehen. Ein möglicher Ansatz, um dieses Problem zu lösen. besteht darin, alle Daten vor der Übertragung in ein einheitliches Format zu überführen. Dies wird zum Beispiel durch den XDR *(External Data Representation)* [80] Standard realisiert. Der Nachteil dieser Vorgehensweise ist, dass zwei unnötige Datenkonversionen vorgenommen werden, wenn zwei Rechner mit der gleichen Datenrepräsentation kommunizieren, die aber nicht der definierten Standardrepräsentation entspricht.

In DOTS werden Datenkonversionen nur dann durchgeführt, wenn diese tatsächlich notwendig sind. Der Sender schickt die Daten in seinem nativen Format zusammen mit einer kurzen Präambel, die das Format beschreibt. Diese wird vom Sender ausgewertet und gegebenenfalls wird eine Konvertierung der Daten in sein eigenes Format vorgenommen. Das bedeutet also, dass nicht für die Daten selbst, sondern für die Beschreibung der Datencodierung ein einheitliches Übertragungsformat definiert wird.

4.2.6 Laufzeitmessungen

Um die oben diskutierten Optimierungen der Kommunikation mit TCP quantitativ nachzuprüfen, wurden Laufzeitmessungen durchgeführt. Dabei kam das in Tabelle 4.1 beschriebene heterogene Rechnernetzwerk zum Einsatz. Es bestand aus 11 Knoten, die sich zum Teil stark hinsichtlich ihrer Rechenleistung, aber auch hinsichtlich der Prozessorarchitekturen und der verwendeten Betriebssysteme unterschieden haben. Alle nachfolgend aufgeführten Angaben für Laufzeiten basieren auf dem arithmetischen Mittel der gemessenen gesamten

Tabelle 4.1 Heterogenes Rechnernetzwerk für Laufzeitmessungen

Node	CPU (MHz)	RAM (MByte)	NIC (MBit/s)	Betriebssystem
1	UltraSparcII (400)	2048	100	Solaris 7
2	PentiumII (300)	128	100	Solaris 7
3	PentiumII (400)	128	100	FreeBSD 4.1
4	PentiumII (400)	128	100	Linux (Kernel 2.4)
5	PowerPC (60)	32	10	AIX 4.3
6	UltraSparcI (140)	64	10	Solaris 7
7	PentiumIII (500)	256	100	Windows 2000
8	PentiumII (400)	128	100	Windows NT4
9	PentiumIII (600)	256	100	QNX 4
10	Pentium (233)	64	100	Windows 98
11	MIPS R8000 (100)	1024	100	IRIX 6.2

Ausführungszeit (wall clock time) von 3 Programmläufen in dem angegebenen Rechnernetzwerk.

Um den Einfluss der vorgestellten Optimierungen auf das Gesamtsystem zu quantifizieren, wurde der Ausführungsoverhead von DOTS Threads gemessen. Der Ausführungsoverhead ist definiert durch die Ausführungszeit eines DOTS Threads, der keine Berechnungen ausführt, sondern lediglich seine Argument-Daten als Resultat zurückliefert. Der Ausführungsoverhead ist somit im Wesentlichen durch die Übertragungszeit der Daten über das Netzwerk bestimmt.

Die Messungen wurden mit einem DOTS Programm durchgeführt, bei dem auf einem Master Knoten für verschiedene Größen von Argument- und Resultat-Daten jeweils 1000 DOTS Threads erzeugt werden, die auf den restlichen 10 Slave Knoten zur Ausführung kommen. Der Master ist selbst nicht an der Ausführung der DOTS Threads beteiligt, so dass die Argument- und Resultat-Daten immer über das Netzwerk übertragen werden müssen. Aus den gemessenen Zeiten für die Ausführung von insgesamt 1000 DOTS Threads wird dann der Ausführungsoverhead für einen DOTS Thread berechnet.

In einer ersten Messreihe sollte die Auswirkung des Connection Caching auf den Ausführungsoverhead bestimmt werden. In Tabelle 4.2 sind die Ergebnisse dieser Laufzeitmessungen abgebildet. Wie oben diskutiert, sind beim Connection Caching vor allem bei kleineren Größen der Argument- und Resultat-Daten Verbesserungen zu erzielen. Bei einer Gesamtgröße von 512 Bytes von Argument und Resultat-Daten konnte durch Connection Caching eine maximale Beschleunigung um den Faktor 6.3 erzielt werden.

Bei der zweiten Messreihe wurden die Auswirkungen von angepassten Puffer-

Tabelle 4.2 Durchschnittlicher Ausführungsoverhead für DOTS Threads mit unterschiedlichen Argument- und Resultat-Größen, gemessen mit und ohne Connection Caching

Argument + Resultat Größe (Byte)	Ausführungsoverhead (ms) ohne Caching	mit Caching	Beschleunigung
32	5.5	1.2	4.6
64	6.2	1.2	5.2
128	6.6	1.1	6.0
256	7.4	1.2	6.2
512	7.6	1.2	6.3
1024	6.9	1.2	5.8
2048	6.5	1.2	5.5
4096	6.1	1.2	5.1
8192	5.0	1.4	3.6
16384	4.6	2.2	2.1
32767	4.9	4.7	1.0

Tabelle 4.3 Durchschnittlicher Ausführungsoverhead für DOTS Threads mit unterschiedlichen Argument- und Resultat-Größen, gemessen mit standard und angepassten Puffergrößen

Argument + Resultat Größe (KByte)	Ausführungsoverhead (ms) ohne Anpassung	mit Anpassung	Beschleunigung
64	10.9	8.9	1.2
128	23.3	17.7	1.3
256	50.7	33.9	1.5
512	112.4	87.7	1.3
1024	225.1	182.6	1.2

größen auf den Ausführungsoverhead von DOTS Threads untersucht. In Tabelle 4.3 sind die gemessenen Werte zusammengefasst. Die maximale Beschleunigung konnte hier bei einer zu übertragenden Datenmenge von 256 KByte pro DOTS Thread erzielt werden.

4.3 Objektkommunikation mittels Serialisierung

Bei der verteilten Ausführung von DOTS Programmen müssen Objekte über eine Netzwerkverbindung zwischen den Adressräumen der beteiligten Knoten übertragen werden. Dabei kann es sich einerseits um Objekte von systemintern definierten Klassen handeln, die zur Kontrolle und Steuerung der verteilten Berechnung dienen. Andererseits müssen auch Objekte von anwendungsspezifischen Klassen transferiert werden, zum Beispiel Task-Objekte oder auch Argument- und Resultat-Objekte von DOTS Threads. Diese Unterteilung ist für die Realisierung der Objektkommunikation von Bedeutung, da die systemintern definierten Klassen zur Übersetzungszeit von DOTS bekannt sind, während dies für die Klassen von anwendungsspezifischen Objekten nicht der Fall ist.

Oftmals werden von Objekten komplexe, verzeigerte Datenstrukturen, zum Beispiel Listen oder Bäume, gekapselt. In diesen Fällen ist ein direkter Austausch eines Objekts durch simples Kopieren der entsprechenden Speicherstellen des Objekts über eine Netzwerkverbindung nicht sinnvoll, da bei diesem Verfahren nur die im Objekt aggregierten Zeiger kopiert würden *(shallow copy)*, nicht aber die von den Zeigern bzw. Zeigerketten referenzierten Objekte *(deep copy)*.

Vor der Übertragung müssen daher Objekte zunächst nach einem definierten Verfahren in einen linear angeordneten Bytestrom umgewandelt werden. Dieser Vorgang wird *Serialisierung* genannt. Nach der Übertragung des Bytestroms über das Netzwerk wird aus ihm eine Kopie des ursprünglichen Objekts im entfernten Adressraum aufgebaut. Dieser Vorgang wird entsprechend als *Deserialisierung* bezeichnet. Neben der Übertragung von Objekten über Netzwerkverbindungen sind Verfahren zur Objektserialisierung auch eine Voraussetzung zur Realisierung von Persistenzmechanismen (permanente Speicherung von Objekten auf nicht-flüchtigen Datenspeichern).

Bei der Diskussion in diesem Abschnitt werden nur die Datenfelder von Objekten berücksichtigt. Die zusätzliche Übertragung von Programmcode der Methoden eines Objekts zusammen mit dessen Daten wird nicht betrachtet, da sie in C++, insbesondere in heterogenen Umgebungen, nur mit unverhältnismäßigem Aufwand realisierbar ist. Vereinfachend wird stattdessen davon ausgegangen, dass in jedem Adressraum der Code aller Methoden einer Klasse für die jeweilige ausführende Prozessorarchitektur zur Verfügung steht.

4.3.1 Anforderungen an den Objektserialisierungsmechanismus

Zur Realisierung der Objektkommunikation mittels Serialisierung müssen neben allgemeinen Anforderungen, wie zum Beispiel Effizienz, im Wesentlichen

zwei entwurfs- bzw. implementierungstechnische Fragestellungen betrachtet werden.

- Wie oben dargestellt, wird bei der Serialisierung eine Umwandlung von einer komplexen Datenstruktur in eine flache Datenstruktur, bei der Deserialisierung die Umwandlung einer flachen Datenstruktur in eine komplexe Struktur vorgenommen. Um diese Umwandlungen durchzuführen, muss entsprechender Programmcode ausgeführt werden. Je nach Mächtigkeit der verwendeten Programmiersprache bzw. deren Laufzeitsystem kann hierfür generischer Code zur Verfügung gestellt werden, der die Umwandlung von Objekten beliebiger Klassen durchführt *(implizite Serialisierung)*. Bei der *expliziten Serialisierung* muss für jede zu serialisierende Klasse hingegen vom Programmierer explizit Umwandlungscode angegeben werden. Um dem Anwendungsprogrammierer im Fall der expliziten Serialisierung die Arbeit zu erleichtern, kann ein Präcompiler eingesetzt werden, der aus den Klassendefinitionen automatisch entsprechenden Umwandlungscode generiert.

- Bei der Objektkommunikation sollen im allgemeinen Fall beliebige, d.h. anwendungsspezifische Objekte (im Fall von DOTS sind das zum Beispiel Task-Objekte) serialisiert übertragen werden können. Das bedeutet, dass die Klassen dieser Objekte im Anwendungsprogramm definiert sind. Zum Zeitpunkt der Erstellung der (De-)Serialisierungskomponenten (die in der Laufzeitbibliothek des Systems angesiedelt sind), können diese Klassen natürlich noch nicht bekannt sein. Hieraus ergibt sich das folgende grundlegende Problem:

  Wenn aus dem serialisierten Bytestrom wieder ein Objekt hergestellt werden soll, muss die Klasse des zu erzeugenden Objekts bekannt sein, damit

  1. eine Instanz des Objekts alloziert werden kann (zum Beispiel verlangt in C++ der `new` Operator zwingend die Angabe eines Klassennamens) und
  2. anschließend der entsprechende Umwandlungscode für die Klasse ausgeführt werden kann, um mit den im Bytestrom enthaltenen Daten die Felder des zuvor erzeugten Objekts zu initialisieren.

  Eine etwas abstraktere Betrachtungsweise dieses Problems ist, dass beim Prozess der Serialisierung eines Objekts Typinformation verlorengeht, die aber bei der Deserialisierung zwingend erforderlich ist. Es muss also ein Mechanismus bereitgestellt werden, mit dem der (De-)Serialisierungskomponente anwendungsspezifische Typen mitgeteilt werden können.

Bevor die Realisierung der Objektkommunikation mittels Serialisierung in DOTS ausführlich diskutiert wird, soll zum Vergleich zunächst der Serialisierungsmechanismus der Programmiersprache Java betrachtet werden.

4.3.2 Realisierung der Objektserialisierung in Java

Spezielle Eigenschaften von Java machen es möglich, ein für den Programmierer besonders elegantes Verfahren zur Objektserialisierung zu verwirklichen. In Java wird eine implizite Objektserialisierung realisiert, sie läuft also für den Programmierer weitestgehend transparent ab.

Die einzige Anforderung an ein serialisierbares Objekt ist, dass es das Interface `Serializable` implementiert. Da dieses Interface aber leer ist, kommt ihm nur eine semantische Bedeutung zu; mit ihm kann der Programmierer eine Klasse kennzeichnen, für deren Instanzen er die Serialisierung erlauben möchte. Die für den Programmierer komfortable implizite Objektserialisierung ist realisierbar, da in Java zur Laufzeit detaillierte Information über die Struktur von beliebigen Objekten erhalten werden kann. Dies wird ermöglicht, da Java Programme innerhalb einer speziellen Laufzeitumgebung, der *Java Virtual Machine (JVM)*, ausgeführt werden. Die entsprechende Funktionalität wird im Java Reflection API zur Verfügung gestellt. In den Objekten der Klasse `Class`, die für jedes Objekt vom Typ `Object` mittels der Methode `getClass` geliefert werden, ist ein genauer Bauplan der zum Objekt gehörigen Klasse zur Laufzeit verfügbar. Es werden Informationen über den Aufbau einer Klasse, also zum Beispiel die enthaltenen Felder samt deren Typen und Bezeichner, sowie deren Werte im betrachteten Objekt zur Verfügung gestellt. Mit diesen Informationen kann nun generischer Code erstellt werden, der die Umwandlung des Objekts in einen konsekutiven Bytestrom für beliebige Klassen vornimmt.

Neben dem Bytestrom werden noch weitere Daten gespeichert, zum Beispiel die zum Objekt gehörige Klasse. Eine detaillierte Beschreibung des Aufbaus des Java Serialisierungsprotokolls ist etwa in [75] zu finden.

Beim Prozess der Deserialisierung wird erneut die Eigenschaft von Java ausgenutzt, dass Programme innerhalb der Java Virtual Machine ausgeführt werden und somit Code dynamisch geladen werden kann. Innerhalb der JVM kann die mit dem Bytestrom mitgelieferte Klassenbeschreibung geladen, ein entsprechendes Objekt angelegt und anschließend mit den Daten aus dem Bytestrom initialisiert werden.

4.3.3 Realisierung der Objektserialisierung in DOTS

Auch in C++ stehen mit den unter dem Begriff *Run Time Type Information (RTTI)* [83] zusammengefassten Konstrukten Sprachmittel zur Untersuchung von Objekten zur Laufzeit hinsichtlich ihres genauen Typs zur Verfügung. Im Wesentlichen umfasst RTTI zwei Sprachkonstrukte:

- Der `dynamic_cast` Operator
  Mit diesem Operator kann zur Laufzeit abgefragt werden, ob ein Objekt von einer bestimmten Klasse ist. Da der Klassenname explizit als Argument angegeben werden muss, ist dieses Konstrukt für die hier diskutierten Zwecke nicht geeignet.

- Der `typeid` Operator
  Mittels des `typeid` Operators kann für ein beliebiges Objekt zur Laufzeit ein `type_id` Objekt erzeugt werden. Dieses enthält im Wesentlichen den Namen der Klasse des Objekts als C String. Stroustrup [83] schlägt in diesem Zusammenhang unter dem Begriff *Extended Type Information* eine mögliche Erweiterung der Sprachimplementierung bzw. ein zusätzliches externes Werkzeug vor. Aus den in einem C++ Programm vorkommenden Klassen sollen zur Übersetzungszeit Beschreibungen ihres Layouts erzeugt und diese in Form einer mit den Klassennamen indizierten Tabelle innerhalb des Programms zur Verfügung gestellt werden. Zurzeit gibt es aber keine entsprechende Sprachimplementierung und auch kein plattformübergreifend verfügbares Tool, das diesen Vorschlag verwirklicht.

Im Gegensatz zum weitaus mächtigeren Java Reflection API, können mit RTTI also zurzeit noch nicht umfassende Informationen über die Struktur der Klasse eines beliebigen Objekts zur Laufzeit gewonnen werden. In C++ steht außerdem auch keine Laufzeitumgebung zur Verfügung, mit der Klassen, die zur Übersetzungszeit nicht bekannt waren, in das Programm zur Laufzeit eingeführt werden können.

Für die Realisierung eines Verfahrens zur Objektserialisierung in C++ sind daher folgende sprachbedingte Randbedingungen zu beachten:

- Ein rein implizites Serialisierungsverfahren ist in C++ wegen der begrenzten Funktionalität von RTTI nicht realisierbar. Dem Programmierer muss also eine Möglichkeit gegeben werden, für serialisierbare Klassen explizit entsprechenden Umwandlungscode anzugeben.

- Da in C++ keine Laufzeitumgebung verwendet wird, müssen anwendungsspezifische Klassen, die serialisierbar sein sollen, zusätzlich mit einem speziellen Registrierungsmechanismus explizit im System bekannt gemacht werden.

Im Folgenden wird die Umsetzung dieser Randbedingungen für die Objektserialisierung in DOTS diskutiert.

Objektserialisierung aus der Sicht des Programmierers

Die Formulierung von (De-)Serialisierungscode wird in DOTS durch die Definition des <<-Operators für die Serialisierung und die Definition des >>-Operators für den Prozess der Deserialisierung von Objekten der jeweiligen Klasse bewerkstelligt. Dazu wird vom System die Hilfsklasse `DOTS_Archive` zur Verfügung gestellt. Diese kapselt im Wesentlichen ein dynamisch erweiterbares, lineares Bytearray. Mit dem <<-Operator können Daten in das Bytearray gepackt werden, der >>-Operator dient zum Auslesen von Daten aus dem Bytearray. Für alle elementaren Datentypen sind die Varianten der Operatoren vom System vordefiniert. Sie dienen als Basis zur Erstellung der entsprechenden Operatoren für zusammengesetzte Strukturen. Durch dieses Verfahren können rekursiv Operatoren für beliebig komplexe Klassen definiert werden. Im Programmbeispiel 4.5 wird diese Vorgehensweise dargestellt.

Mit diesen Mitteln können auch für die in der C++ Standardbibliothek definierten Container-Klassen, wie zum Beispiel `vector`, die (De-)Serialisierungsoperatoren vordefiniert werden. Zur einfachen Verarbeitung von C++ Arrays wird der Manipulator [83] `array` definiert.

Das vorgestellte API zur Erstellung von Code zur Objektserialisierung bietet die folgenden Vorteile:

- **Lokalität**
  Der Code zur Serialisierung bildet mit der entsprechenden Klasse eine syntaktische Einheit.
- **Wiederverwendbarkeit**
  Es können rekursiv aus den Operatoren für elementare Datentypen Operatoren für komplexere Strukturen definiert werden, die ihrerseits wieder in weiteren Definitionen wiederverwendet werden können.
- **Intuitive Verwendbarkeit**
  Durch die lineare Anordnung der Operatoren in einer Programmzeile wird der Vorgang der (De-)Serialisierung intuitiv wiedergegeben.

Programmbeispiel 4.5 Verwendung des (De-)Serialisierungsoperators

```
class X {
public:
  int a;
  double b;

  friend DOTS_Archive& operator<<(DOTS_Archive& arch, X& x) {
    return arch << x.a << x.b;
  }

  friend DOTS_Archive& operator>>(DOTS_Archive& arch, X& x) {
    return arch >> x.a >> x.b;
  }
};

class Y {
public:
  short c;
  X x;

  friend DOTS_Archive& operator<<(DOTS_Archive& arch, Y& x) {
    return arch << y.c << y.x;
  }

  friend DOTS_Archive& operator>>(DOTS_Archive& arch, Y& x) {
    return arch >> y.c >> y.x;
};
```

Serialisierung anwendungsspezifischer Klassen

Zur Registrierung von anwendungsspezifischen serialisierbaren Klassen wird von DOTS der Aufruf `dots_reg_class` zur Verfügung gestellt, dessen Implementierung in Programmbeispiel 4.6 gezeigt wird. Ziel der Registrierung ist, dass anwendungsspezifische Klassen im System bekannt gemacht werden.

Die Makrodefinition dient nur zur Vereinfachung der Verwendung, sie erfüllt sonst keine weitere Funktion. Die wesentliche Funktionalität wird durch die Template-Funktion `dots_register_class` realisiert. Bei der Registrierung einer Klasse im Anwendungsprogramm wird diese Funktion mit der betreffenden Klasse als Template Argument vom Compiler instantiiert. Innerhalb der Funk-

Programmbeispiel 4.6 Die Implementierung des `dots_reg_class` Aufrufs

```
#define dots_reg_class(X) dots_register_class<X>()

template <class TYPE>
void dots_register_class(void)
{
  DOTS_Object_Handler_Base* obj_handler =
    new DOTS_Object_Handler<TYPE>();

  const char* type_name = typeid(TYPE).name();

  DOTS_REGISTRY->register_object_handler(obj_handler, type_name);
}
```

tion wird für diese spezielle Klasse ein Objekt vom Typ `DOTS_Object_Handler` erzeugt und zusammen mit dem Klassennamen, der mit dem `typeid` Operator ermittelt wird, in einem globalen Verzeichnis registriert.

Die Klasse `DOTS_Object_Handler` stellt die generische Objektserialisierungskomponente des Systems dar. Bei der Erzeugung einer Instanz wird als Template Argument die zu serialisierende Klasse übergeben. Sie verfügt über Methoden zum Erzeugen und Löschen von Objekten registrierter Klassen, sowie Methoden zu deren (De-)Serialisierung. Programmbeispiel 4.7 zeigt ihre Implementierung.

Da es in C++ keine allgemeine Basisklasse (wie etwa in Java die Klasse `Object`) gibt, verwendet das Interface zur Objektübergabe bzw. -rückgabe den Typ `void*`. Der Typverlust ist in diesem Fall nicht von Bedeutung. Bei Argumenten kann mittels des Template Arguments eine entsprechende Downcast Operation ausgeführt werden. Bei den Rückgabewerten ist ein solcher Downcast ebenfalls möglich, da an der Stelle der Verwendung des Objekts, zum Beispiel im Anwendungsprogramm, sein genauer Typ ebenfalls bekannt ist.

Im Grunde wird mit dem beschriebenen Verfahren eine dynamische Variante des oben vorgestellten Extended Type Information Konzepts realisiert. Die relevanten Layoutbeschreibungen der Klasse werden jedoch nicht durch den Compiler generiert, sondern sind in Form der vom Programmierer definierten Serialisierungsoperatoren gegeben. Diese Information ist für jede registrierte Klasse über ein entsprechend spezialisiertes `DOTS_Object_Handler` Objekt verfügbar. Mittels des Klassennamens (der zur Laufzeit durch den `typeid` Operator geliefert wird) kann auf diese Objekte in der globalen Tabelle zugegriffen werden.

Programmbeispiel 4.7 Die Klasse DOTS_Objekt_Handler

```
#include "DOTS_Archive.h"

class DOTS_Object_Handler_Base
{
public:
  virtual void* create_object(void) = 0;
  virtual void delete_object(void*) = 0;
  virtual void serialize_object(DOTS_Archive&, void*) = 0;
  virtual void deserialize_object(DOTS_Archive&, void*) = 0;
};

template <class TYPE>
class DOTS_Object_Handler : public DOTS_Object_Handler_Base
{
  void* create_object(void) {
    return new TYPE;
  }

  void delete_object(void* obj) {
    delete (TYPE*) obj;
  }

  void serialize_object(DOTS_Archive& arch, void* obj) {
    arch << *((TYPE*) obj);
  }

  void deserialize_object(DOTS_Archive& arch, void* obj) {
    arch >> *((TYPE*) obj);
  }
};
```

Bei der Übertragung serialisierter Objekte über das Netzwerk reicht es zur Wiederherstellung eines Objekts aus, wenn der Klassenname zusammen mit dem Bytestrom übermittelt wird und die entsprechende Klasse auf allen Knoten registriert wurde. Mittels des Klassennamens kann auf der Empfängerseite in der Tabelle ein entsprechendes `DOTS_Object_Handler` Objekt erhalten werden, mit dem der typspezifische Deserialisierungsprozess realisiert wird. Es stellt Methoden zur Erzeugung eines entsprechenden Objekts und zum Aufruf des passenden Deserialisierungscodes zur anschließenden Initialisierung des erzeugten Objekts mittels der übertragenen Daten zur Verfügung.

5

Kapitel 5

Parallele Erfüllbarkeitsprüfung von booleschen Formeln

In diesem Kapitel wird die Parallelisierung eines Algorithmus zur Behandlung des Erfüllbarkeitsproblems der Aussagenlogik vorgestellt. Dieses wird im allgemeinen auch kurz als SAT Problem bezeichnet.

SAT ist das erste Problem, für das die Zugehörigkeit zu der Klasse der NP vollständigen Probleme nachgewiesen wurde (Satz von Cook [24]). Das bedeutet zunächst, dass für alle bekannten SAT Algorithmen Eingaben existieren, die exponentielle Laufzeiten aufweisen und somit praktisch nicht lösbar sind. Durch ausgeklügelte Algorithmen und Heuristiken kann aber die Laufzeit für eine große Klasse von Eingaben dramatisch reduziert werden.

Neben seiner theoretischen Relevanz ist das SAT Problem auch von praktischer Bedeutung. Viele anwendungsbezogene Probleme lassen sich als SAT Probleme formulieren. In den folgenden Abschnitten werden zum Beispiel Probleme aus den Bereichen Kryptoanalyse, Hardwareverifikation und Konsistenzprüfung von Produktdokumentation beschrieben. Gerade für diese Fälle kann dann auch die weitere Beschleunigung der Berechnung durch die Parallelisierung durchaus gewinnbringend sein.

5.1 Problembeschreibung

In diesem Abschnitt soll zunächst eine formale Beschreibung des SAT Problems gegeben werden. Eine umfangreichere Darstellung ist zum Beispiel in [50] zu finden.

Eine *boolesche Formel* in konjunktiver Normalform mit n booleschen Variablen (im Folgenden auch kurz mit *Formel* bezeichnet) besteht aus einer Konjunkti-

on ($\wedge$) von *Klauseln*. Die Klauseln werden wiederum als Disjunktion ($\vee$) von einem oder mehreren *Literalen* gebildet, wobei ein Literal eine boolesche Variable oder die Negation einer booleschen Variable ist. Besteht eine Klausel aus genau einem Literal, spricht man von einer *Unit Klausel*.

Eine *Variablenbelegung* weist den booleschen Variablen einer Formel je einen der Werte TRUE oder FALSE zu. Eine Variablenbelegung, für die eine gegebene Formel F den Wert TRUE annimmt, wird als *erfüllende Belegung* oder auch *Modell* von F bezeichnet. Besitzt eine Formel wenigstens eine erfüllende Belegung, ist sie *erfüllbar*. Existiert für eine Formel keine erfüllende Belegung, so wird diese als *nicht erfüllbar* bzw. *unerfüllbar* bezeichnet.

Beispiel: Für die boolesche Formel:

$$F = (x_2 \vee \overline{x_3}) \wedge (x_1 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \wedge x_3,$$

ist die Variablenbelegung

$$x_1 \leftarrow \text{FALSE}, x_2 \leftarrow \text{TRUE}, x_3 \leftarrow \text{TRUE}$$

eine erfüllende Belegung. Somit ist F erfüllbar.

Die Menge aller Formeln, die mindestens eine erfüllende Belegung besitzen, wird mit SAT bezeichnet. Das Erfüllbarkeitsproblem der Aussagenlogik besteht nun darin, für eine gegebene boolesche Formel F zu entscheiden, ob sie Element der Menge SAT ist.

Eine naive Vorgehensweise um für eine Formel F mit n Variablen zu prüfen, ob sie erfüllbar ist, wäre alle 2^n möglichen Variablenbelegungen zu untersuchen. Diese Suche nach einer erfüllenden Variablenbelegung könnte zum Beispiel mit einem Backtracking-Suchverfahren realisiert werden. Für größere Werte von n ist dieses Verfahren jedoch nicht mehr in akzeptabler Zeit durchführbar. Im nächsten Abschnitt wird ein Verfahren beschrieben, welches den Suchvorgang nach einer erfüllenden Variablenbelegung für viele Formeln wesentlich beschleunigen kann.

5.2 Das Verfahren von Davis und Putnam

Im Jahr 1960 stellten M. Davis und H. Putnam einen Algorithmus zur Lösung des Erfüllbarkeitsproblems [25] vor, der wegen seiner grundlegenden Bedeutung allgemein als Davis-Putnam (DP) Algorithmus bezeichnet wird. Er dient bis heute als Grundlage für viele erweiterte SAT Algorithmen.

Im Wesentlichen wird beim Davis-Putnam Algorithmus eine optimierte kombinatorische Suche mittels eines Backtracking-Verfahrens durchgeführt. Der Davis-Putnam Algorithmus erweitert dabei das oben erwähnte naive Suchverfahren

um einen Problemvereinfachungsschritt *(Constraint Propagation)*, der fortlaufend während der Suche nach einer erfüllenden Belegung ausgeführt wird. Die Problemvereinfachung geschieht durch schrittweise Variablenbelegung in einem Backtracking Suchprozess und einer nach jeder Variablenbelegung ausgeführter Anwendung zweier Operationen zur Vereinfachung des Problems. Die Operationen werden als *Unit Subsumption* und *Unit Resolution* bezeichnet.

Bevor diese Operationen und der gesamte Davis-Putnam Algorithmus im Einzelnen beschrieben werden, soll eine weitere Darstellungsweise für boolesche Formeln eingeführt werden. Zur einfacheren Beschreibung des Davis-Putnam Algorithmus werden üblicherweise Formeln als Mengen von Klauseln und Klauseln als Mengen von Literalen dargestellt. Wegen der Kommutativität und Assoziativität, sowie Idempotenz der Operatoren $\vee$ und $\wedge$ ist diese Mengendarstellung äquivalent zur sonst gebräuchlichen Formelschreibweise.

Beispiel: Die Formel:

$$(x_2 \vee \overline{x_3}) \wedge (x_1 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \wedge x_3$$

besitzt die folgende äquivalente Mengendarstellung:

$$\{\{x_2, \overline{x_3}\}, \{x_1, x_3\}, \{\overline{x_1}, \overline{x_2}, \overline{x_3}\}, \{x_3\}\}.$$

Abbildung 5.1 zeigt den grundlegenden Davis-Putnam Algorithmus in Pseudo-Code. Er besteht im Wesentlichen aus der rekursiven Prozedur davis_putnam. Diese erhält als Argument die im Vereinfachungsprozess jeweils aktuelle Formel in Form einer Klauselmenge. Dem erstem Aufruf von davis_putnam wird die Eingabeklauselmenge als Argument übergeben.

Im ersten Schritt werden zunächst iterativ alle Unit-Klauseln aus der Klauselmenge entfernt. Eine in der Klauselmenge enthaltene Unit-Klausel erzwingt die Zuweisung der entsprechenden Variablen derart, dass die Unit-Klausel (bzw. das entsprechende Literal) den Wert TRUE erhält. Diese Variablenzuweisung wird auch als *implizite* oder *erzwungene Variablenzuweisung* bezeichnet. Die aus dieser erzwungenen Variablenzuweisung resultierenden Konsequenzen werden anschließend von dem Unit-Subsumptions- und dem Unit-Resolutionsschritt berechnet.

- Bei der Unit-Subsumption werden diejenigen Klauseln aus der Klauselmenge entfernt, die durch die implizite Variablenzuweisung erfüllt sind. Alle weiteren in der Klausel vorkommenden Variablen, denen noch kein Wert zugewiesen wurde, können nun belegt werden, ohne dass diese Klausel noch weiter betrachtet werden muss.

- Im Unit-Resolutionsschritt werden alle Klauseln verkürzt, die die Unit Klausel negiert enthalten. Das entsprechende Literal kann aus der Klausel entfernt werden, da es den Wert FALSE (Neutralelement von $\vee$) hat.

Durch die Unit-Resolution können weitere Unit-Klauseln entstehen. Die Anwendung der Unit-Subsumption und Unit-Resolution wird daher solange iteriert, bis in der Klauselmenge keine Unit-Klauseln mehr vorkommen.

Enthält nach der Durchführung der Unit-Subsumptions und -Resolutionsschritte die Klauselmenge eine leere Klausel, so kann unter der aktuellen Variablenbelegung keine erfüllende Belegung erreicht werden. Dieser Zustand wird auch als *Konflikt* bezeichnet. Ist hingegen die Klauselmenge selbst leer, wurde eine erfüllende Variablenbelegung gefunden.

Ansonsten ist das Resultat dieser Schritte eine (reduzierte) Klauselmenge, die als Argument bei den beiden folgenden rekursiven Aufrufen von davis_putnam angegeben wird.

Im nächsten Schritt wird ein Literal aus der Klauselmenge ausgewählt, dessen Belegung mit TRUE bzw. FALSE in zwei nachfolgenden rekursiven Aufrufen der Prozedur davis_putnam untersucht werden soll. Für diese Literalauswahl gibt es eine Vielzahl von Heuristiken [42], die zum Teil wesentlichen Einfluss auf die mögliche Problemvereinfachung haben können. Die Zuweisung der entsprechenden Variable (die so genannte *explizite Variablenzuweisung*), wird technisch durch Hinzufügen des ausgewählten Literals beim ersten, bzw. des negierten Literals beim zweiten rekursiven Aufruf als Unit Klausel zur Klauselmenge erreicht. Beim nächsten Problemvereinfachungsschritt wird dann die entsprechende Variable durch eine implizite Variablenzuweisung belegt. Eine Variablenzuweisung kann somit einfach durch ein Literal dargestellt werden.

Durch diese rekursive Vorgehensweise werden schrittweise ungebundene Variablen belegt, bis entweder eine erfüllende Belegung gefunden wurde oder ein Konflikt auftritt. Wenn eine erfüllende Belegung gefunden wurde, kann die Berechnung abgebrochen werden, da hier davon ausgegangen wird, dass man nur an einem Erfüllbarkeitstest interessiert ist, nicht aber alle Modelle einer Formel finden will. Tritt jedoch ein Konflikt auf, wird ein Backtrackingschritt ausgeführt.

Die doppelte Rekursion des DP Algorithmus induziert einen binären Suchbaum. Die Knoten des Baumes sind die (vereinfachten) Klauselmengen in den einzelnen Berechnungsebenen. Die zwei ausgehenden Kanten eines Knotens werden entsprechend den beiden rekursiven Aufrufen mit L und $\overline{L}$ markiert. Ein Blatt des Baumes repräsentiert entweder einen Konflikt, also eine Klauselmenge, die ein leere Klausel enthält, oder eine Lösung, d.h. die leere Klauselmenge. In diesem Fall wurden alle Variablen belegt, ohne dass ein Konflikt aufgetreten ist.

Algorithmus 5.1 Davis-Putnam Algorithmus

```
boolean davis_putnam( ClauseSet S,  Level d)
  while (S contains a unit clause {U})
     delete clauses containing U from S     /* Unit-Subsumption */
     delete ¬U from all clauses in S     /* Unit-Resolution */

  if (∅ ∈ S)     /* Konflikt? */
    return FALSE
  if (S = ∅ )     /* Lösung? */
    return TRUE

  Literal L
  assign L to a literal occurring in S     /* Literalauswahl */

  if (davis_putnam(S ∪ {L}), d+1)     /* 1. rek. Aufruf */
    return TRUE
  if (davis_putnam(S ∪ {¬L}), d+1)     /* 2. rek. Aufruf */
    return TRUE
  else
    return FALSE
```

5.3 Erweiterung des DP Algorithmus durch dynamisches Lernen

Silva und Sakallah haben eine Erweiterung des DP Algorithmus durch einen dynamischen Lernprozess vorgestellt [74]. Die grundlegende Idee besteht darin, dass beim Auftreten von Konflikten zusätzliches Wissen über das Problem generiert wird, welches ein späteres Vorkommen von Konflikten der selben Art verhindert und somit die Suche in anderen Teilbäumen verkürzt werden kann. Diese Vorgehensweise wird als *Konflikt-Analyse* bezeichnet.

Im Folgenden wird eine kurze Einführung in die Funktionsweise der Konflikt-Analyse gegeben, eine ausführliche Darstellung ist in [74] zu finden. Tritt nach dem Unit-Subsumptions- und -Resolutionsschritt ein Konflikt auf, werden zunächst aus der Menge aller aktuellen expliziten Variablenzuweisungen diejenigen bestimmt, deren gleichzeitiges Auftreten bereits eine hinreichende Bedingung für den Konflikt darstellt. Dazu wird während des gesamten Berechnungs-

ablaufs ein so genannter Implikationsgraph aufgebaut. Die Knotenmenge dieses gerichteten Graphen wird von der Menge aller aktuellen Variablenzuweisungen gebildet. Die Kanten zeigen an, durch welche Variablenzuweisungen eine Klausel zu einer Unit-Klausel wurde und somit eine neue Zuweisung erzwungen wurde. Tritt ein Konflikt auf, werden alle Kanten beginnend bei der letzten Zuweisung zurückverfolgt, bis eine explizite Variablenzuweisung erreicht wird. Explizite Variablenzuweisungen haben keine eingehende Kanten, da sie nicht durch die Belegung von anderen Variablen erzwungen wurden. Die derartig bestimmte Menge von expliziten Variablenzuweisungen kann auch als Menge von entsprechenden Literalen

$$\{L_{d_0}, \ldots, L_{d_k}\}$$

repräsentiert werden. Aus dieser Teilmenge wird eine neue Klausel der Form

$$C_C = \overline{L}_{d_0} \vee \cdots \vee \overline{L}_{d_k}$$

generiert und zur ursprünglichen Klauselmenge hinzugefügt. Die neu generierte Klausel wird *Konflikt-Klausel* oder auch *Lemma* genannt. Dabei ist die Konflikt-Klausel so beschaffen, dass in jedem Fall mindestens eine Variable anders belegt wird, als dies beim Auftreten des betrachteten Konflikts der Fall war. Somit kann eine hinreichende Bedingung für den Konflikt nicht mehr zustande kommen. Es kann gezeigt werden, dass durch das Hinzufügen von Lemmas zur Klauselmenge das Ergebnis der Berechnung nicht verändert wird.

Da Konflikte während der Berechnung sehr oft auftreten (jedes Blatt des Suchbaumes, das keine Lösung darstellt repräsentiert einen Konflikt), werden sehr viele Lemmas erzeugt, was zu einem starken Anwachsen der Klauselmenge führen kann. Im schlimmsten Fall kann ein exponentielles Wachstum der Klauselmenge resultieren. Hierdurch kann der Aufwand für die Durchführung des Unit-Subsumptions- und Unit-Resolutionsschritts so stark erhöht werden, dass der Effekt der Suchraumverkleinerung durch Lemmas nicht mehr zum Tragen kommt und die Berechnung durch die Konfliktanalyse insgesamt verlangsamt wird. Aus diesem Grund werden üblicherweise nur Lemmas bis zu einer maximalen Länge (d.h. maximalen Anzahl der enthaltenen Variablen) in die Klauselmenge eingefügt, um ein zu starkes Anwachsen der Klauselmenge zu verhindern.

5.4 Grundlegende Parallelisierungstechniken

Um eine parallele Ausführung des ursprünglichen Davis-Putnam Algorithmus zu ermöglichen, muss zunächst das gesamte zu betrachtende Problem in einzelne, möglichst voneinander unabhängige Teilprobleme aufgeteilt werden. Dazu

bedarf es einer Erweiterung des Davis Putnam Algorithmus, so dass zusätzlich auch Teilprobleme verarbeitet und die erhaltenen Teillösungen zu einer Gesamtlösung kombiniert werden können.

Zur Aufspaltung des Gesamtproblems in Teilprobleme wird in [91] ein Ansatz beschrieben, der auf der Aufteilung des Suchbaumes in einzelne Teilbäume basiert. Als Grundlage der Realisierung dieser Aufteilung wird der Begriff *Leitpfad (guiding path)* eingeführt. Ein Leitpfad dient zur Zustandsbeschreibung des Suchprozesses, der vom Davis-Putnam Algorithmus ausgeführt wird. Er gibt darüber Auskunft, welche Teile des Suchraumes schon bearbeitet wurden, bzw. noch bearbeitet werden müssen.

Ein Leitpfad ist im Wesentlichen ein Pfad von der Wurzel zur aktuellen Position des Suchprozesses im Suchbaum. Der Pfad wird als geordnete Liste von Paaren dargestellt, wobei für jede Ebene des Suchbaumes ein Paar in der Liste enthalten ist. Das Paar an der Stelle d im Pfad enthält die folgenden Informationen:

- Das Literal L_d, das auf der Rekursionsebene d ausgewählt wurde.
- Ein boolescher Wert B_d, der anzeigt, ob auf der Ebene d der Backtrackingschritt noch ausgeführt werden muss.

Abbildung 5.1 zeigt einen Suchbaum, in dem der Leitpfad

$$P =< x, \text{TRUE} >, < y, \text{FALSE} >, < \overline{z}, \text{TRUE} >$$

markiert ist.

Der Davis-Putnam Algorithmus kann so erweitert werden, dass er als weiteren Eingabeparameter einen Leitpfad erhält. Wie in Abbildung 5.2 gezeigt wird, kann der Suchprozess mittels des übergebenen Leitpfades effizient in den von ihm beschriebenen Zustand versetzt werden.

Leitpfade können außerdem dazu verwendet werden, den Suchraum in disjunkte Teilbereiche zu zerlegen. Auf Leitpfaden wird wie im Folgenden beschrieben eine *Split-Operation* definiert, die zwei neue Leitpfade liefert. Aus dem Leitpfad

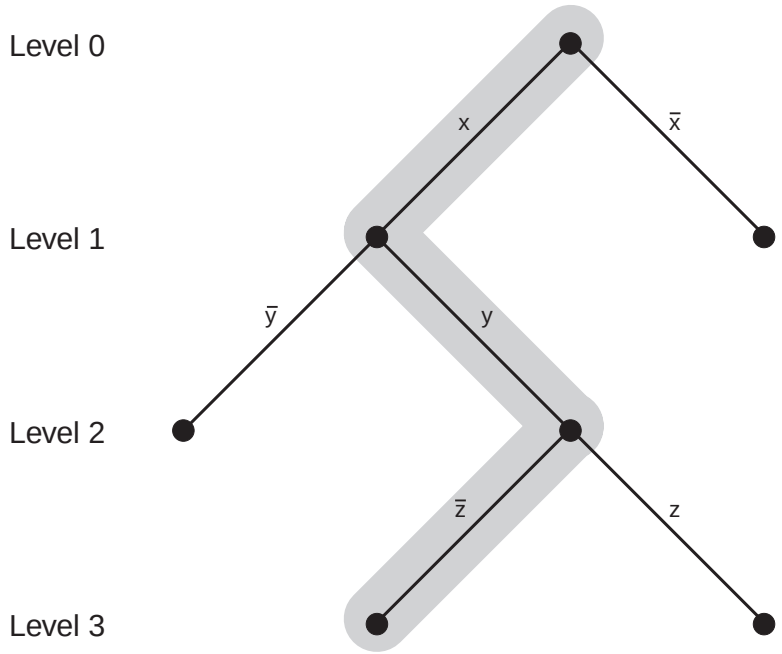
$$P =< L_1, B_1 >, \ldots, < L_{i-1}, B_{i-1} >, < L_i, \text{TRUE} >, < L_{i+1}, B_{i+1} >, \ldots, < L_k, B_k >$$

werden durch die Anwendung der Split-Operation die zwei folgenden neuen Leitpfade P_1 und P_2 erzeugt:

Algorithmus 5.2 Davis-Putnam Algorithmus mit Leitpfadverarbeitung

```
boolean guided := TRUE    /* Flag für Leitpfadmodus */

boolean davis_putnam( ClauseSet S,  Level d,
                      List gp := <L_1,B_1>, ···, <L_n,B_n>)

  if (guided  and  (d >= n))    /* Leitpfadmodus ausschalten? */
    guided := FALSE;

  while (S contains a unit clause {U})
     delete clauses containing U from S    /* Unit-Subsumption */
     delete ¬U from all clauses in S    /* Unit-Resolution */

  if (∅ ∈ S)    /* Konflikt? */
    return FALSE
  if (S = ∅ )    /* Lösung? */
    return TRUE

  Literal L

  if (guided)
     L := L_d
  else
     assign L to a literal occurring in S    /* Literalauswahl */
     L_d := L
     B_d := TRUE

  if (davis_putnam(S ∪ {L}), d+1, gp )    /* 1. rek. Aufruf */
    return TRUE

  if (B_d = FALSE)
    return FALSE

  L_d := ¬L
  B_d := FALSE

  if (davis_putnam(S ∪ {¬L}), d+1, gp )    /* 2. rek. Aufruf */
    return TRUE
  else
    return FALSE
```

Abbildung 5.1 Suchbaum mit Leitpfad



$$P_1 =< L_1, B_1 >, \ldots, < L_{i-1}, B_{i-1} >, < \overline{L_i}, \text{FALSE} >$$
$$P_2 =< L_1, B_1 >, \ldots, < L_{i-1}, B_{i-1} >, < L_i, \text{FALSE} >, < L_{i+1}, B_{i+1} >, \ldots, < L_k, B_k > \quad .$$

Der Leitpfad P_2 unterscheidet sich nur dadurch vom ursprünglichen Leitpfad, dass der Wert von B_i von TRUE auf FALSE geändert wurde. Der Index i wird als *Split-Punkt* bezeichnet. Durch dieses Verfahren erhält man zwei Leitpfade, die den gesamten noch zu bearbeitenden Suchraum (der durch den Leitpfad P beschrieben wird) in zwei disjunkte Suchräume aufteilen, die dann von Beweiserinstanzen parallel bearbeitet werden können. Um eine Aufteilung zu erreichen, die möglichst dicht an der Wurzel des Suchbaumes angesiedelt ist, muss der Split-Punkt i so gewählt werden, dass $B_j = \text{FALSE}$ für alle $j < i$ gilt.

In Abbildung 5.3 wird der Pseudo-Code für die Split-Operation wiedergegeben. Bei dieser Operation wird der Leitpfad P_1 aus dem Argument berechnet und als Resultat zurückgeliefert, P_2 ergibt sich durch Modifikation des Arguments am Split-Punkt.

Da das Ausmaß der Problemvereinfachung durch den Davis-Putnam Algorithmus und den dynamischen Lernprozess grundsätzlich nicht vorherbestimmt

Algorithmus 5.3 Split Operation für Leitpfade

```
List split_guiding_path( List gp := <L_1,B_1>, ···, <L_n,B_n>)
begin
  List< Literal, boolean> split_path
  i:=1
  while (B_i = FALSE)
    split_path.append_element(<L_i, B_i>)
    i:=i+1

  B_i := FALSE
  split_path.append_element(<¬L_i, FALSE>)

  return split_path
end
```

werden kann, ist aber eine statische Einteilung in balancierte Teilprobleme zur parallelen Bearbeitung nicht ausreichend. Der beschriebene Ansatz zur Teilproblemerzeugung muss deshalb in ein dynamisches Verfahren integriert werden, dessen Realisierung im nächsten Abschnitt detailliert behandelt wird.

5.5 Parallele Realisierung mit DOTS

An die Realisierung des parallelen SAT Beweisers mit DOTS wurden die folgenden Anforderungen gestellt:

- **Integration des dynamischen Lernprozesses**
  Die in Abschnitt 5.3 beschriebene Erweiterung des Davis-Putnam Algorithmus durch dynamisches Lernen vermag die Laufzeiten von Beweisen in vielen Fällen drastisch zu vermindern [74]. Um eine weitere Verkürzung der Laufzeit durch Parallelisierung erzielen zu können, ist es daher wesentlich, den dynamischen Lernprozess in die parallele Realisierung des SAT Beweisers zu integrieren.
- **Skalierbare und effiziente Ausführung irregulärer Probleminstanzen**
  Grundsätzlich kann das Ausmaß der vom Davis-Putnam Verfahren und dem dynamischen Lernprozess bewirkten Reduktion des Suchraumes und damit der Zeit, die für die Bearbeitung eines Teilproblemes nötig ist, nicht

vorherbestimmt werden. Je nach Probleminstanz kann eine beträchtliche Irregularität bezüglich der Rechenzeit der einzelnen Teilprobleme auftreten. Die parallele Realisierung des SAT Beweisers muss daher dezentral und dynamisch sein, um auch bei Probleminstanzen, die eine starke Irregularität aufweisen, eine effiziente und skalierbare Ausführung sowohl in Architekturen mit gemeinsamem als auch in Architekturen mit verteiltem Speicher zu ermöglichen.

Um beide Anforderungen gleichermaßen zu erfüllen, wird die verteilte parallele Ausführung in zwei unterschiedlichen logischen Schichten organisiert.

- **Suchschicht**
  Auf dieser Schicht wird mittels nebenläufiger, verteilter Threads eine parallele Suche in dynamisch erzeugten, disjunkten Teilbäumen ausgeführt.

- **Wissensaustauschschicht**
  Auf der zweiten Schicht wird Wissen in Form von Lemmas zwischen den einzelnen Knoten durch autonome Tasks gezielt ausgetauscht.

Die Abläufe auf diesen Schichten werden in den beiden folgenden Abschnitten ausführlich dargestellt.

5.5.1 Parallele Suche mit Threads

An der parallelen Ausführung auf dieser logischen Ebene sind so genannte Such-Threads beteiligt, die im Wesentlichen jeweils den in Abbildung 5.2 dargestellten modifizierten Davis-Putnam Algorithmus ausführen. Such-Threads sind als DOTS Threads realisiert, die als Argument einen Leitpfad erhalten und als Resultat einen booleschen Wert zurückliefern. Durch den als Argument übergebenen Leitpfad wird der Suchraum definiert, der von dem Such-Thread bearbeitet werden soll. Der Leitpfad wird während der Suche ständig aktualisiert, so dass dieser zu jedem Zeitpunkt den aktuellen Zustand des Suchprozesses wiedergibt. Das Resultat eines Such-Threads gibt an, ob im zugewiesenen Suchraum eine Lösung gefunden wurde.

Ein Haupt-Thread startet die Suche und wartet auf die Ergebnisse der einzelnen Such-Threads. Der parallele Suchprozess wird gestartet, indem der Haupt-Thread einen Such-Thread erzeugt, dem zunächst der gesamte Suchraum zugewiesen ist. Der initiale Such-Thread bekommt also als Argument einen leeren Leitpfad übergeben.

Um eine dynamische Aufteilung des Suchraumes je nach verfügbaren Prozessoren zu erreichen, wird wie folgt verfahren. Während der Berechnung überwachen alle beteiligten Such-Threads die Länge der lokalen Task Queue. Falls diese leer ist, wird eine Split Operation auf dem aktuellen Leitpfad ausgeführt. Der betreffende Such-Thread erzeugt dazu, wie in Abschnitt 5.4 gezeigt, aus seinem aktuellen Leitpfad einen neuen Leitpfad und ändert den eigenen Leitpfad entsprechend ab. Anschließend wird ein neuer Such-Thread erzeugt, der den neu erzeugten Leitpfad als Argument erhält. Der entsprechende Task wird vom DOTS Laufzeitsystem in die lokale Task Queue eingestellt und kann später lokal ausgeführt, oder aber vom Lastverteilungssystem zu einem anderen Knoten transferiert werden. Das beschriebene Verfahren realisiert somit eine dynamische Erzeugung von Teilproblemen. Um ein unkontrolliertes Abspalten von Such-Threads zu verhindern, wird eine festgelegte Zeitspanne gewartet, bevor ein neuer Abspaltungs-Vorgang gestartet wird.

Nachdem der Haupt-Thread den initialen Such-Thread erzeugt hat, wartet er mittels eines dots_join Aufrufs auf die Ergebnisse der einzelnen Such-Threads. Alle Such-Threads (bis auf den initialen Thread) werden mit dem dots_hyperfork Primitiv erzeugt, so dass deren Ergebnisse vom Haupt-Thread abgefragt werden können. Das Ergebnis eines Such-Threads zeigt an, ob in dem zugewiesenen Suchraum eine Lösung gefunden wurde. Die Berechnung ist fertig, wenn entweder alle erzeugten Such-Threads die Suche in ihrem zugewiesenem Teilbaum beendet haben ohne eine Lösung zu finden, oder wenn ein Such-Thread eine Lösung gefunden hat. Falls eine Lösung gefunden wurde, werden alle anderen Such-Threads mit einem dots_cancel Aufruf terminiert.

In den Abbildungen 5.4 und 5.5 ist der Pseudo Code zum vorangehend beschriebenen Ablauf angegeben.

Als Lastverteilungsstrategie wird das Task-Stealing Verfahren mit randomisierter Opferauswahl eingesetzt. Karp und Zhang haben gezeigt, dass diese Lastverteilungsstrategie für parallele Backtracking-Suchalgorithmen mit hoher Wahrscheinlichkeit zu einer Beschleunigung der Berechnung führt, die sich nur um einen konstanten Faktor vom optimalen Wert (also linearer Beschleunigung) unterscheidet [48].

5.5.2 Austausch von Wissen durch autonome Tasks

Beim Davis-Putnam Verfahren mit Konflikt-Analyse wird bei jedem Konflikt-Blatt im Suchbaum neues Wissen in Form eines Lemmas erzeugt, das zur Klauselmenge hinzugefügt werden kann. Bei der verteilten Ausführung ist dieses neue Wissen zunächst nur lokal vorhanden. Da jede Beweiserinstanz sehr viele neue Lemmas produziert, ist ein Austausch des gesamten erzeugten Wissens

Algorithmus 5.4 Paralleler Davis-Putnam Algorithmus

```
boolean guided := TRUE    /* Flag für Leitpfadmodus */

boolean davis_putnam( ClauseSet S,  Level d,
                      List gp := <L_1,B_1>, ··· , <L_n,B_n>)

  if (guided  and (d >= n))    /* Leitpfadmodus ausschalten? */
    guided := FALSE;

  while (S contains a unit clause {U})
     delete clauses containing U from S    /* Unit-Subsumption */
     delete ¬U from all clauses in S    /* Unit-Resolution */

  if (∅ ∈ S)    /* Konflikt? */
    return FALSE
  if (S = ∅ )    /* Lösung? */
    return TRUE

  Literal L

  if (guided)
     L := L_d
  else
     assign L to a literal occurring in S    /* Literalauswahl */
     L_d := L
     B_d := TRUE
     if (dots_queue_length() = 0)
        List split_gp := split_guiding_path(gp)
        dots_hyperfork(davis_putnam({S, 0, split_gp})

if (davis_putnam(S ∪ {L}), d+1,  gp )    /* 1. rek. Aufruf */
    return TRUE

 if (B_d = FALSE)
    return FALSE

 L_d := ¬L
 B_d := FALSE

 if (davis_putnam(S ∪ {¬L}), d+1,  gp )    /* 2. rek. Aufruf */
    return TRUE
 else
    return FALSE
```

Algorithmus 5.5 Hauptprogramm zum parallelen Davis-Putnam Algorithmus

```
main()
   List empty_gpath := <>;
  DOTS_Thread_Group sat_group;

  dots_fork(sat_group, davis_putnam, {S,0,empty_path});

   boolean result;

   while (dots_join(sat_group, result) != -1)
     if (result = TRUE)
       break;

   if (result = TRUE)
     print(SAT)
   else
     print(UNSAT)
```

in einem verteilten System wegen der beschränkten Bandbreite des Netzwerkes nicht sinnvoll. Deshalb wurde ein Ansatz gewählt, der autonome Tasks als mobile Agenten einsetzt, um in dem verteilten System gezielt geeignetes neues Wissen zu suchen und einzusammeln.

Sind auf einem Knoten mehrere Prozessoren verfügbar, können mehrere Beweiserinstanzen erzeugt werden, die parallel arbeiten. Jede Beweiserinstanz verfügt über einen Klauselspeicher, der die Eingabe-Klauselmenge und die im Laufe der Berechnung erzeugten Lemmas speichert. Lemmas können zwischen den Klauselspeichern der Beweiserinstanzen eines Knotens einfach durch Zugriff auf gemeinsame Speicherbereiche ausgetauscht werden.

Für den Austausch von Lemmas zwischen Klauselspeichern auf verschiedenen Knoten werden autonome Tasks eingesetzt, die als Agenten zum Suchen und Sammeln von geeignetem neuem Wissen in Form von Lemmas arbeiten. Dazu wird zu Beginn der Berechnung für jeden Klauselspeicher ein autonomer Task erzeugt, der als Agent neue Lemmas von Klauselspeichern auf anderen Knoten sammelt. Er besucht während der gesamten Berechnung der Reihe nach alle Knoten und transportiert die gesammelten Lemmas zu seinem Heimatknoten. Die Such-Agenten wählen Lemmas nach den folgenden Kriterien aus.

- Die Länge eines Lemmas darf eine vorgegebene Maximallänge nicht überschreiten.
- Es werden keine Lemmas gesammelt, die im aktuellen Kontext des Beweisers bereits subsumiert wären, d.h. kein verwertbares Wissen liefern können.

Durch diese Auswahlkriterien wird einerseits die Anzahl der ausgetauschten Lemmas verkleinert, gleichzeitig kann auch bei der verteilten Ausführung vom globalen Lernprozess profitiert werden. In Abbildung 5.6 ist der Pseudo Code für einen Such-Agenten dargestellt.

Algorithmus 5.6 Agent zur Suche neuer Lemmas

```
class LemmaAgent : public DOTS_Autonomous_Task
private:
  List< Clause> lemmas;
  List< Literal> closed_literals;

public:
 run()
    if (home_node())
      CLAUSE_STORE.store_new_lemmas(lemmas);
      closed_literals := CLAUSE_STORE.get_closed_literals();
      travel_to_next_node();
    else
      lemmas := CLAUSE_STORE.read_new_lemmas(closed_literals);
      travel_to_next_node();
```

5.6 Laufzeitmessungen

Im Folgenden werden Laufzeitmessungen vorgestellt, die mit der Implementierung des vorangehend beschriebenen verteilten, parallelen Beweisers durchgeführt wurden. Für die Messungen wurde der im Folgenden angegebene heterogene Pool von Rechnern mit insgesamt 14 Prozessoren verwendet. Die Rechner waren durch ein 100 Mbps Ethernet Netzwerk verbunden.

- 2 Sun Ultra E450, jeweils mit 4 UltraSparcII Prozessoren (@400 MHz), 1 GB bzw. 2 GB Hauptspeicher, Betriebssystem: Solaris 7.
- 1 PC, mit 2 Intel Pentium III Prozessoren (@500MHz) und 256 MB Hauptspeicher, Betriebssystem: Solaris 7.
- 4 PCs, jeweils mit 1 Intel Pentium II Prozessor (@400MHz) und 128 MB Hauptspeicher, Betriebssystem: Solaris 7.

Für die Laufzeitmessungen wurden Probleme aus unterschiedlichen, praktisch relevanten Bereichen ausgewählt: Quasigruppenprobleme, Kryptographie und Hardwareverifikation. Aus diesen Klassen wurden 3 Beispiele ausgesucht, bei deren paralleler Bearbeitung verschiedene Effekte beobachtet werden konnten. Die angegebenen sequentiellen Laufzeiten $\bar{t}_{\text{seq}}$ wurden folgendermaßen berechnet:

$$\bar{t}_{\text{seq}} = \frac{4 \cdot \bar{t}_{\text{PII}} + 2 \cdot \bar{t}_{\text{PIII}} + 8 \cdot \bar{t}_{\text{UltraSparcII}}}{14}.$$

Da die parallelen Berechnungen aufgrund der dynamischen Suchraumaufteilung und dem Austausch der Lemmas nichtdeterministisch sind, werden zusätzlich zum gemittelten Wert der parallelen Laufzeit jeweils die gemessenen Werte der 10 zugrundeliegenden parallelen Programmläufe angegeben. Um den Effekt des Austausches der Lemmas durch Agenten zu ermitteln wurden sowohl Programmläufe ohne Lemmaaustausch, als auch Programmläufe mit Lemmaaustausch durchgeführt. Neben den jeweils gemessenen Laufzeiten und den sich daraus ergebenden Speedup-Werten, ist als zusätzliches Maß für die erzielte Suchraumverringerung die Anzahl der Blätter des Suchbaumes angegeben.

5.6.1 Quasigruppenprobleme

Als Eingabe wurde ein Quasigruppenexistenzproblem aus der Klasse QG7 (siehe [32]) gewählt. Eine Quasigruppe ist ein endliches, kürzbares Gruppoid, das aus einer Grundmenge S und einer binären Multiplikation * besteht. Die Multiplikationstabelle ist auch unter dem Namen “Lateinisches Quadrat” bekannt, d.h. jede ihrer Zeilen und Spalten ist eine Permutation von S . Das gewählte Problem QG7-12 kodiert die Suche nach der Existenz einer Quasigruppe der Ordnung 12 mit der zusätzlichen Eigenschaft $((x*y)*x)*y = x$ für alle $x, y \in S$. Da keine derartige Quasigruppe existiert ist QG7-12 unerfüllbar.

Die Konflikt-Analyse vermag bei dieser Problemklasse keine signifikante Beschleunigung der Berechnung bewirken. Die sequentiellen Laufzeiten mit und

Tabelle 5.1 Messergebnisse für QG7-12

Sequentielle Laufzeit: 1504.4 s					
Kein Lemmaaustausch			Mit Lemmaaustausch		
Laufzeit (s)	Speedup	Blätteranzahl	Laufzeit (s)	Speedup	Blätteranzahl
138	10.9	247,325	143	10.5	193,488
145	10.4	247,526	139	10.8	198,639
141	10.7	254,319	135	11.1	196,322
137	11.0	248,393	143	10.5	196,360
145	10.4	256,027	141	10.7	202,314
138	10.9	243,492	148	10.2	203,966
147	10.2	260,576	140	10.7	195,374
142	10.6	254,730	139	10.8	198,975
145	10.4	254,417	139	10.8	198,693
143	10.5	258,499	145	10.4	207,590
142.1	10.6	252,530	141.2	10.7	199,172

ohne Durchführung der Konfliktanalyse sind praktisch identisch. Wie Tabelle 5.1 zeigt, ist damit natürlich auch der Austausch von Lemmas für Quasigruppenprobleme nicht gewinnbringend. Trotzdem konnte aber eine Beschleunigung der Berechnung um einen durchschnittlichen Faktor von 10.7 erzielt werden.

5.6.2 Kryptoanalyse

Die bei diesen Messungen verwendete SAT Formel codiert die Suche nach einem Schlüssel für das DES Verfahren [58] mit 3 Runden, wobei 3 Klartext/Schlüsseltext Block-Paare geben sind. Die Ergebnisse der Messungen sind in Tabelle 5.2 zusammengefasst. Es gibt bei diesen Anwendungsbeispielen immer einen Schlüssel, die zugehörige Formel ist also erfüllbar. Bei erfüllbaren Problemen können superlineare Beschleunigungen auftreten, da bei paralleler Bearbeitung die Wahrscheinlichkeit erhöht wird, dass ein Thread bei der Suche näher bei der Lösung startet, als dies in der sequentiellen Version der Falls wäre. Die Messungen ergaben, dass durch Austausch von Lemmas dieser Effekt deutlicher in Erscheinung tritt.

5.6.3 Hardwareverifikation

Die ausgewählte Formel codiert den Vergleich zweier äquivalenter 16-Bit Multiplizierer [6]. Bei der angewendeten Codierungstechnik wird Äquivalenz durch

Tabelle 5.2 Messergebnisse für DES

Sequentielle Laufzeit: 1595.0 s					
Kein Lemmaaustausch			Mit Lemmaaustausch		
Laufzeit (s)	Speedup	Blätteranzahl	Laufzeit (s)	Speedup	Blätteranzahl
312	5.1	1,217,581	56	28.5	119,377
308	5.2	1,287,493	63	25.3	132,456
312	5.1	1,250,425	64	24.9	122,319
310	5.1	1,074,611	61	26.1	140,352
312	5.1	1,109,904	74	21.6	187,450
314	5.1	1,119,698	94	17.0	264,180
308	5.2	1,189,128	89	17.9	244,282
312	5.1	1,194,960	87	18.3	227,919
312	5.1	1,327,524	89	17.9	285,094
309	5.2	1,000,395	103	15.5	298,306
310.9	5.1	1,117,172	78	21.3	192,174

Unerfüllbarkeit angezeigt. Trotzdem konnten auch hier superlineare Beschleunigungen der Berechnung beobachtet werden (siehe Tabelle 5.3). Durch den Austausch von Lemmas stehen schon früher Informationen zur Verfügung, die den Suchraum beschränken, als dies bei der sequentiellen Ausführung der Fall ist.

Tabelle 5.3 Messergebnisse für Longmult

Sequentielle Laufzeit: 2212.0 s					
Kein Lemmaaustausch			Mit Lemmaaustausch		
Laufzeit (s)	Speedup	Blätteranzahl	Laufzeit (s)	Speedup	Blätteranzahl
149	14.8	376,184	109	20.3	252,183
154	14.4	405,902	98	22.6	228,835
174	12.7	434,409	101	21.9	236,340
160	13.8	404,967	99	22.3	231,383
145	15.3	365,113	114	19.4	264,961
224	9.9	583,491	97	22.8	222,615
145	15.3	362,368	91	24.3	209,383
189	11.7	462,577	87	25.4	200,894
146	15.2	356,542	111	19.9	238,787
149	14.8	373,029	101	21.9	235,955
163.5	13.8	412,458	100.8	22.1	232,134

5.7 Industrielle Anwendung

In diesem Abschnitt soll eine Anwendung aus der Automobilindustrie beschrieben werden. Die in diesem Kapitel behandelten Verfahren für paralleles SAT Checking können eingesetzt werden, um die Benutzbarkeit eines existierenden Tools zur Verifikation von Produktdokumentationen zu verbessern [53].

5.7.1 Problembeschreibung

Fahrzeuge können heutzutage vom Kunden durch die Kombination einer Vielzahl von Optionen bei der Bestellung individuell konfiguriert werden. Die Mercedes C Klasse bietet beispielsweise über 1000 Konfigurationsoptionen. Im Nutzfahrzeugbereich sind die Konfigurationsmöglichkeiten sogar noch wesentlich vielfältiger. Dies bedeutet, dass im Nutzfahrzeugbereich eine bestimmte Fahrzeugkonfiguration nur wenige Male während der gesamten Produktlebenszeit gefertigt wird.

Da einerseits die Anzahl der möglichen Kombinationen von Optionen extrem hoch ist, andererseits aber nicht jede Kombination sinnvoll bzw. überhaupt baubar ist, muss die Gültigkeit jeder Bestellung rechnergestützt verifiziert werden. Dies geschieht mittels einer Produktdokumentationsdatenbank, die logische Regeln enthält, welche alle gültige Konfigurationen beschreiben [31].

Die manuelle Pflege dieser Produktdokumentationsdatenbank ist extrem fehleranfällig, da zum einen das Regelsystem sehr umfangreich und komplex ist und zum anderen häufig und von verschiedenen Personen Änderungen vorgenommen werden müssen. Ein Fehler in dem Regelsystem kann im schlimmsten Fall bewirken, dass eine baubare Bestellung zurückgewiesen wird oder aber eine nicht-baubare Bestellung angenommen wird. Beide Fälle sind natürlich nicht wünschenswert, die Annahme einer nicht-baubaren Bestellung könnte zum Beispiel bewirken, dass das Fließband während des Produktionsprozesses angehalten werden müsste. Die Korrektheit der Produktdokumentationsdatenbank ist daher von größter Bedeutung für den gesamten Bestell- und Produktionsprozess und somit unternehmenskritisch.

Um systematisch Fehler im Regelsystem der Produktdokumentationsdatenbank zu finden, wurde ein auf SAT basiertes, formales Modell des gesamten Bestellprozesses eingeführt [53]. Mit dem auf einem SAT Beweiser-Kern basierten System BIS (Bauberkeitsinformationssystem) [77] ist es möglich, interaktiv die folgenden Konsistenzüberprüfungen auszuführen:

- Notwendige und unzulässige Codes Check
  Bei diesem Test wird festgestellt, ob es Bestellcodes gibt, die in jeder bau-

baren Bestellung vorkommen müssen, bzw. ob es Bestellcodes gibt, die in keiner baubaren Bestellung vorkommen dürfen.

- Überflüssige Teile Check
  Hier wird überprüft, ob in der Stückliste Bauteile vorgesehen sind, die aber in keiner gültigen Bestellung vorkommen können.

Um diese Checks auf den gesamten Daten einer Modellreihe durchzuführen, sind bis zu 10000 SAT Beweise erforderlich. Die meisten Beweise können jeweils in wenigen Sekunden berechnet werden. Es hat sich aber gezeigt, dass in einer solchen Beweisreihe eine geringe Anzahl von schweren Beweisen vorkommt, die eine beträchtlich längere Zeitdauer zur Bearbeitung erfordern. In einigen Fällen kann die Laufzeit für schwere Beweise mehrere Stunden betragen. Bei der sequentiellen Ausführung einer Beweisreihe ergeben sich daher die folgenden Probleme bei der Verwendung des BIS Systems im Produktionsprozess:

- Die Gesamtlaufzeit für alle Konsistenzchecks ist sehr hoch.
- Es ist keine interaktive Benutzung des Systems möglich. Wenn zum Beispiel bereits zu Beginn der Beweisreihe ein rechenintensiver Beweis auftritt, wird die Ausgabe der Ergebnisse aller nachfolgenden Checks stark verzögert.

Eine zunächst einfach durchführbar erscheinende Lösungsmöglichkeit für das zweite aufgelistete Problem wäre die Sortierung der Beweise nach zunehmender Laufzeit (also nach dem Prinzip “shortest job first”). Diese Vorgehensweise ist aber in dieser einfachen Form nicht realisierbar, da keine allgemeine Methode existiert, um die Laufzeit eines Beweises vorherzubestimmen. Wie im nachfolgenden Abschnitt gezeigt wird, lässt sich dieser Ansatz jedoch durch Parallelisierung approximieren; gleichzeitig wird damit auch das Problem der langen Gesamtlaufzeit angegangen.

5.7.2 Parallelisierungsansatz

Bei der Realisierung der parallelen Ausführung der Konsistenzchecks wurden zwei Ziele verfolgt.

- Beschleunigung der Gesamtlaufzeit der Berechnung.
- Verkürzung der durchschnittlichen Wartezeit für die Ausgabe des Ergebnisses eines Konsistenzchecks.

Die durchschnittliche Wartezeit $\bar{t}_w$ ist definiert durch:

$$\bar{t}_w = \sum_{i=1}^{n} \frac{t_w(i)}{n} \ ,$$

wobei n die Gesamtzahl der Beweise der Beweisreihe und $t_w(i)$ diejenige Zeit ist, die vom Start der Berechnung bis zur Ausgabe des Ergebnisses des $i-ten$ Beweises der Beweisreihe gewartet werden muss.

Um die oben dargestellten Ziele zu verwirklichen wird die parallele Bearbeitung der Beweisreihe mit DOTS in zwei Phasen aufgeteilt.

- **Phase 1**: Erkennung von schweren Beweisen
  Die Beweise der Beweisreihe werden nebenläufig ausgeführt. Für die Ausführung eines jeden Beweises wird ein Zeitlimit gesetzt. Läuft das Zeitlimit ab, wird der Beweis als schwerer Beweis betrachtet. Seine Bearbeitung wird zurückgestellt, wobei der aktuelle Zustand der Berechnung gespeichert wird.

  Für jeden Beweis der Beweisreihe wird vom Haupt-Thread ein eigener DOTS Thread erzeugt. Das Resultat des Threads kann entweder des Ergebnis des Beweises (erfüllbar, bzw. nicht-erfüllbar) oder ein Leitpfad sein, wenn die Berechnung aufgrund einer Überschreitung des Zeitlimits unterbrochen wurde. Die Resultate werden vom Haupt-Thread abgefragt. Wenn ein Leitpfad zurückgegeben wird, wird dieser in eine Warteschlange eingestellt, ansonsten erfolgt die Ausgabe des Ergebnisses.

  Als Lastverteilungsstrategie für die 1. Phase der parallelen Berechnung der Beweisreihe wird ein Round Robin Task Sharing Verfahren verwendet.

- **Phase 2**: Parallele Ausführung der schweren Beweise
  Die in Phase 1 zurückgestellten schweren Beweise werden nun parallel berechnet, wobei die Berechnung im zuvor gespeicherten Zustand wieder aufgesetzt wird.

  Für die Realisierung der 2. Phase der parallelen Berechnung der Beweisreihe werden die in Abschnitt 5.5 beschriebenen Verfahren eingesetzt. Die Lastverteilungsstrategie wird in Phase 2 auf Task Stealing mit randomisierter Opferauswahl umgestellt.

5.7.3 Messergebnisse

Zur Durchführung von Laufzeitmessungen wurde der im Folgenden beschriebene heterogene Pool von Rechnern verwendet. Alle Rechner waren durch ein 100 Mbps Ethernet Netzwerk verbunden.

- 2 Sun Ultra E450, jeweils mit 4 UltraSparcII Prozessoren (@400 MHz), 1 GB bzw. 2 GB Hauptspeicher, Betriebssystem: Solaris 7.
- 4 PCs, jeweils mit 1 Intel Pentium II Prozessor (@400MHz) und 128 MB Hauptspeicher, Betriebssystem: Windows NT 4.0.

Für die Laufzeitmessungen wurden die “notwendige und unzulässige Codes” Checks sowie “Überflüssige Teile” Checks auf einen Teil der Produktdokumentation der Mercedes C und E Klasse angewendet. In den Tabellen 5.4 sind die Ergebnisse der Messungen zusammengefasst. Als Zeitlimit zur Erkennung von schweren Beweisen wurde 20 Sekunden gewählt. Die angegebenen Werte für die sequentielle Laufzeit und die durchschnittliche Wartezeit sind das nach der Anzahl der unterschiedlichen Prozessoren gewichtete Mittel aus dem arithmetischen Mittel von jeweils 3 auf den unterschiedlichen Rechnertypen gemessenen Werten. Außerdem sind die Werte für die Gesamtlaufzeit und die durchschnittliche Wartezeit für jeweils drei verschiedene parallele Programmläufe angegeben.

Tabelle 5.4 Messergebnisse für Daten der Mercedes C und E Klasse.

C-Klasse (Codes Checks)		
Beweise: 520	Schwere Beweise: 4	
	Laufzeit (s)	Wartezeit (s)
Seq. (E450)	1,417.0	654.6
Seq. (PC)	1,738.0	800.3
Seq. (gemittelt)	1,524.0	703.2
Parallel (Speedup)	158.0 (9.6)	66.8 (10.5)
	162.0 (9.4)	66.6 (10.6)
	164.0 (9.3)	67.1 (10.5)

C-Klasse (Teile Checks)		
Beweise: 512	Schwere Beweise: 6	
	Laufzeit (s)	Wartezeit (s)
Seq. (E450)	10,447.0	5,459.5
Seq. (PC)	13,667.0	7,167.1
Seq. (gemittelt)	11,520.3	6,028.7
Parallel (Speedup)	1,026.0 (11.2)	100.6 (59.9)
	1,043.0 (11.0)	100.8 (59.8)
	1,039.0 (11.1)	100.8 (59.8)

E-Klasse (Code Checks)		
Beweise: 525	schwere Beweise: 6	
	Laufzeit (s)	Wartezeit (s)
Seq. (E450)	1,935.0	956.1
Seq. (PC)	2,281.0	1,123.9
Seq. (gemittelt)	2,050.3	1,012.0
Parallel (Speedup)	253.0 (8.1)	111.4 (9.1)
	245.0 (8.4)	110.6 (9.2)
	252.0 (8.1)	111.5 (9.1)

E-Klasse (Teile Checks)		
Beweise: 500	schwere Beweise: 6	
	Laufzeit (s)	Wartezeit (s)
Seq. (E450)	40,186.0	16,370.5
Seq. (PC)	51,769.0	21,500.4
Seq. (gemittelt)	44,047.0	18,080.5
Parallel (Speedup)	2,198.0 (20.0)	110.4 (163.8)
	2,174.0 (20.3)	109.3 (165.4)
	2,192.0 (20.1)	108.8 (166.2)

6

Kapitel 6

Weitere Anwendungen

In diesem Kapitel sollen weitere parallele Applikationen vorgestellt werden, die mit DOTS realisiert wurden. Es handelt sich dabei um Anwendungen aus dem Bereich des Symbolischen Rechnens und der Computer Graphik.

6.1 Parallele Faktorisierung mit elliptischen Kurven

Dieser Abschnitt befasst sich mit einem parallelen Verfahren zur Faktorisierung von ganzen Zahlen. Das Faktorisierungsproblem besteht darin, eine positive ganze Zahl in Primfaktoren zu zerlegen. Nach dem Hauptsatz der elementaren Zahlentheorie ist jede positive ganze Zahl N ein Produkt von Primzahlen, die bis auf die Reihenfolge eindeutig bestimmbar sind [70].

Neben seiner theoretischen Relevanz, ist die Faktorisierung von Zahlen auch innerhalb der praktischen Informatik von Bedeutung. Viele Verfahren der Codierungstheorie und der Kryptologie basieren auf der Kenntnis der Primfaktordarstellung einer Zahl. Die Zahl wird bei diesen Verfahren so groß gewählt, dass nur mit enormen Rechenaufwand eine Primfaktorzerlegung durchgeführt werden kann und somit ein Dritter nur sehr schwer in Kenntnis der Zerlegung kommt.

6.1.1 Die elliptische Kurven Methode zur Faktorisierung

Zur Faktorisierung von großen Zahlen wurden bereits zahlreiche Verfahren vorgeschlagen. Das Faktorisierungsverfahren nach der elliptischen Kurven Methode [55] zeichnet sich vor allem dadurch aus, dass die zu erwartende Laufzeit

von der Größe der Primfaktoren und weniger von der Größe der zu faktorisierenden Zahl selbst abhängt. Deshalb ist es vor allem geeignet, wenn man einen kleinen Faktor vermutet, oder falls nur die Kenntnis kleiner Faktoren von Bedeutung ist. Zum Beispiel möchte man im Zusammenhang mit dem RSA Verschlüsselungsverfahren ausschließen, dass die Anzahl der zu N teilerfremden Zahlen $\varphi(N)$ aus kleinen Faktoren besteht.

Im Folgenden wird die Funktionsweise des Faktorisierungsverfahrens nach der elliptischen Kurven Methode kurz angedeutet; eine ausführliche Darstellung kann [55] entnommen werden.

Sei K ein Körper mit Charakteristik $q \notin \{2, 3\}$.

Eine elliptische Kurve über K ist durch die Punktemenge

$$E_{A,B} = \{(x, y) \in K | y^2 = x^3 + Ax + B\} \cup \{\infty\}$$

mit

$$4A^3 + 27B^2 \neq 0, (A, B \in K)$$

gegeben.

Auf der Punktemenge einer elliptischen Kurve kann eine Addition $\oplus$ definiert werden, so dass $(E_{A,B}, \oplus)$ eine abelsche Gruppe mit dem Neutralelement ∞ ist. Wenn man als Körper K die Menge der reellen Zahlen $\mathbb{R}$ wählt, so kann die Addition geometrisch erklärt werden: Zwei Punkte werden addiert, indem eine Gerade durch die beiden Punkte gezogen wird. Das Ergebnis der Addition ist der an der x-Achse gespiegelte dritte Schnittpunkt der Geraden mit der elliptischen Kurve. Für die Realisierung des Faktorisierungsverfahrens mit elliptischen Kurven ist lediglich relevant, dass die Additionsvorschrift für die Punkte $(x_1, y_1), (x_2, y_2) \in E_{A,B}$ die Berechnung der Terme $(x_2 - x_1)^{-1}$ bzw. $(2y_1)^{-1}$ erfordert.

Bei der Faktorisierung mit der elliptischen Kurven Methode wird die folgende Beobachtung ausgenutzt: Sei N die zu faktorisierende Zahl. Ersetzt man den Körper K durch den Ring $\mathbb{Z}/N\mathbb{Z}$, so sind die Additionsregeln weiterhin gültig, falls die Inversen $(x_2 - x_1)^{-1}$ bzw. $(2y_1)^{-1}$ existieren. In $\mathbb{Z}/N\mathbb{Z}$ existieren diese Terme genau dann, wenn sie zu N teilerfremd sind. Die Berechnung der Addition scheitert also genau dann, wenn $ggT((x_2 - x_1), N) > 1$ bzw. $ggT(2y_1, N) > 1$ gilt. Ist dies der Fall, so hat man aber in den meisten Fällen einen Faktor von N gefunden; nur wenn $ggT((x_2 - x_1), N) = N$ bzw. $ggT(2y_1, N) = N$ gilt, wurde keine verwertbare Information geliefert.

Sei nun P_0 ein zufällig gewählter Punkt auf einer beliebigen elliptischen Kurve. Unterstellt man einen Primfaktor p in N derart, dass die Gruppe der Punkte der Kurve modulo p eine Ordnung besitzt, deren Primteiler in höchster Potenz

allesamt kleiner als eine vorgegebene Schranke B_1 sind, so gilt dies auch für die Ordnung von P_0, d.h. $ord(P_0)|R := kgV(1, \ldots, B_1)$.

Beim Faktorisierungsverfahren nach der elliptschen Kurven Methode wird nun $R \oplus P_0$ berechnet. Gelingt dies, waren also alle auftretenden Nenner invertierbar, so ist der Versuch gescheitert und wird mit einer anderen Kurve und mit einem neuen Anfangspunkt erneut gestartet. Die Details des oben beschriebenen Ansatzes können [65] entnommen werden.

6.1.2 Parallele Realisierung

Die im Folgenden vorgestellte Realisierung einer parallelen verteilten Applikation zur Faktorisierung nach der elliptischen Kurven Methode mit DOTS ist eng an der Implementierung einer sequentiellen Programmversion angelehnt, die der Computeralgebra-Programmbibliothek LiDIA [56] entnommen ist.

In einem ersten Schritt werden durch Probedivision kleinere Primfaktoren eliminiert. Der zweite Schritt des Verfahrens wird in einem Master-Slave Ansatz parallel ausgeführt, wobei der Master und die Slaves als DOTS Threads realisiert sind. Der Master erzeugt zunächst eine Warteschlange von Jobgruppen, wobei jede Gruppe ein gewisse Zahl von zu berechnenden Kurven repräsentiert, mit denen nach Faktoren gesucht wird. Die Jobgruppen sind in der Warteschlange nach steigendem Berechnungsaufwand angeordnet. Der Master startet für jede Kurve der aktuellen Jobgruppe einen Slave-Thread, der deren Berechnung durchführt und fragt anschließend die Ergebnisse der Berechnung ab. Wurde ein Faktor p gefunden, werden alle Threads der Gruppe abgebrochen und anschließend neu gestartet, wobei aus der Eingabe p in maximaler Potenz herausgekürzt wurde. Falls durch die Berechnung aller Kurven einer Jobgruppe kein weiterer Faktor gefunden werden konnte, wird das oben beschriebene Verfahren mit der nächsten Jobgruppe der Warteschlange fortgesetzt.

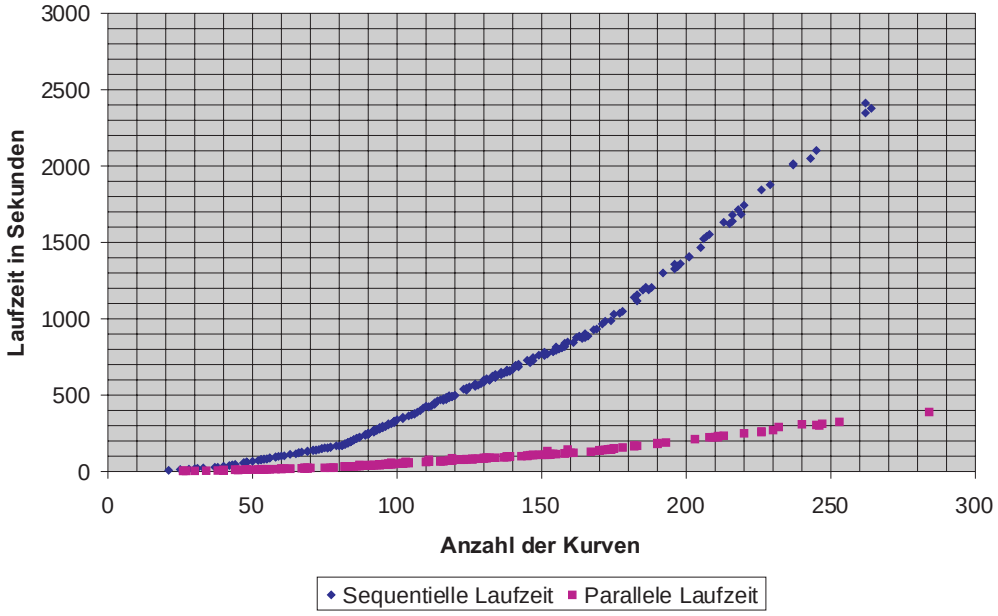
6.1.3 Messergebnisse

Zur Durchführung von Laufzeitmessungen wurden 2 Sun Ultra E450, jeweils mit 4 UltraSparcII Prozessoren (@400 MHz) und 1 GB bzw. 2 GB Hauptspeicher verwendet. Die beiden Rechner waren mit einem 100 Mbps Ethernet Netzwerk verbunden. In 250 sequentiellen und 250 parallelen Programmläufen wurde jeweils die Zahl

$$\begin{aligned} N &= 95761251540994803594523624829179262048455595 57 \\ &= 10000123327864826609730616 7^1 \cdot 9576007055249111517 1^1 \end{aligned}$$

faktorisiert.

Abbildung 6.1 Laufzeiten der sequentiellen und parallelen Faktorisierung



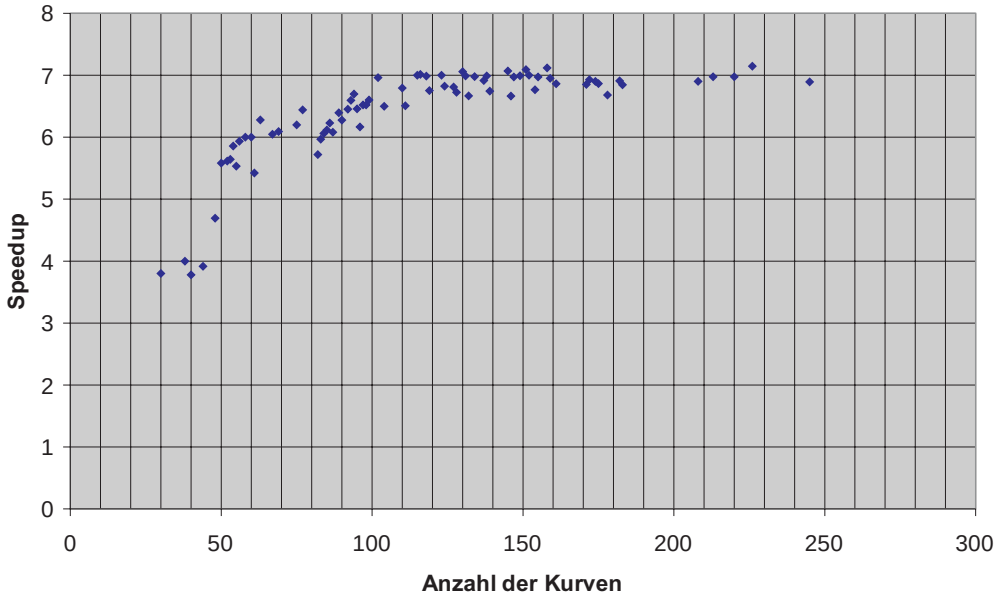
Die Ergebnisse der Laufzeitmessungen sind in Abbildung 6.1 zusammengefasst. Die gemessenen Laufzeiten sind dabei über der Anzahl der Kurven aufgetragen, die berechnet werden mussten, bis ein Faktor von N gefunden wurde. Es werden sowohl die Laufzeiten für die sequentielle Programmversion, als auch für die parallele Programmversion unter Verwendung von 8 Prozessoren angegeben.

Die Abbildungen 6.2 und 6.3 zeigen die aus den gemessenen Laufzeiten bestimmten Werte für die Beschleunigung und die Effizienz in Abhängigkeit von der Anzahl der generierten Kurven. Es sind hier nur solche Werte berücksichtigt, für die sowohl die sequentielle, als auch die parallele Laufzeit vorlag. Wie in Abbildung 6.3 zu sehen ist, konnten Effizienzwerte von bis zu 0.9 erzielt werden.

6.2 Parallele Volumenvisualisierung

Das Ziel der Volumenvisualisierung ist die grafische Darstellung von Volumendaten, die im allgemeinen als *Voxelwürfel* repräsentiert sind. Voxelwürfel werden verwendet, um Volumendaten zu speichern, bei denen keine unmittelbare Information über die Oberfläche eines Objekts vorhanden ist. Durch einen Voxelwürfel ist für jeden diskreten Punkt im Raumgitter ein Dichtewert gegeben.

Abbildung 6.2 Beschleunigung durch parallele Faktorisierung



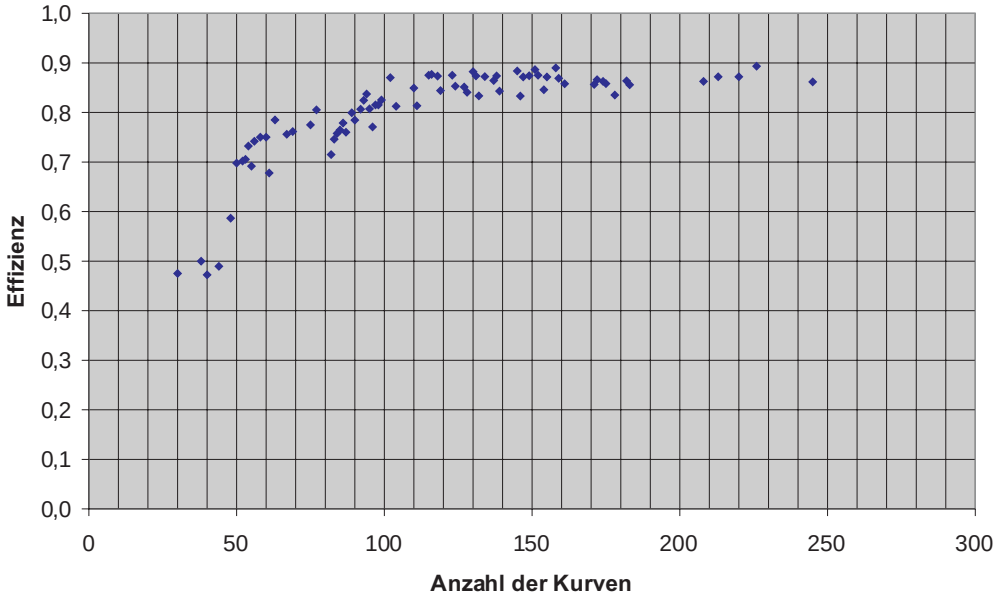
Solche Voxeldatensätze werden zum Beispiel von 3D Scannern geliefert. Mittels einer Transferfunktion wird einem Voxelwürfel anhand des Dichtewerts ein Farb- und Durchsichtigkeitswert zugeordnet. Die typische Größe dieser 3D Datensätze liegt zwischen 128^3 und 512^3 Voxelwürfel, wobei ein Voxelwürfel aus bis zu vier Bytes besteht. Die Gesamtgröße derartiger Datensätze kann somit bis zu 512 MByte betragen. Falls Bilder von hoher Qualität erzeugt werden sollen, ist es erforderlich, dass alle vorhandenen Daten berücksichtigt werden. Für die Generierung von Bildfolgen in Realzeit ist also eine beträchtliche Rechenkapazität notwendig.

6.2.1 Volumenvisualisierungsalgorithmus

Der parallelen Applikation, die in diesem Abschnitt betrachtet werden soll, liegt ein direktes Verfahren zur Volumenvisualisierung zu Grunde. Bei diesem Ansatz wird keine geometrische Zwischenstruktur zur Darstellung der Volumenelemente generiert. In [68] wird die Verwendung von DOTS zur Parallelisierung eines komplementären Verfahrens beschrieben, das den *Marching Cubes Algorithmus* zur Approximation von Isoflächen zur Volumendarstellung verwendet.

Der betrachtete Algorithmus gehört zur Klasse der bildraumorientierten Verfah-

Abbildung 6.3 Effizienz der parallelen Faktorisierung



ren, d.h. es wird für jeden Punkt der Betrachtungsebene festgelegt, welches Element dort sichtbar ist. Nachdem die Betrachtungsebene anhand gegebener Parameter im Raum fixiert wurde, werden vom Blickpunkt aus durch jeden Pixel der Betrachtungsebene Blickstrahlen in den Objektraum geworfen. Bei diesem Ray Casting Verfahren werden Farbwerte entlang eines Blickstrahls zusammengefasst und mittels der gegebenen Durchsichtigkeitswerte miteinander kombiniert. Um festzustellen, ob ein Strahl auf ein Volumenelement trifft, wird ein dreidimensionaler *Cohen-Sutherland Algorithmus* verwendet. Strahlen, die vorbeigehen, können ausgesondert werden.

6.2.2 Parallelisierungsansatz

Die parallele Ausführung folgt einem Master-Slave Ansatz. Der Master und die Slaves sind jeweils als DOTS Thread realisiert. Die Betrachtungsebene wird vom Master-Thread in einzelne Rechtecke (so genannte Kacheln) eingeteilt, die jeweils unabhängig voneinander durch einen Slave-Thread berechnet werden. Jeder Slave-Thread erhält als Argument die zur Berechnung der jeweiligen Kachel notwendigen Parameter, z.B. Betrachtungspunkt, Blickrichtung, Position von Lichtquellen, usw. und wendet den in Abschnitt 6.2.1 beschriebenen Vo-

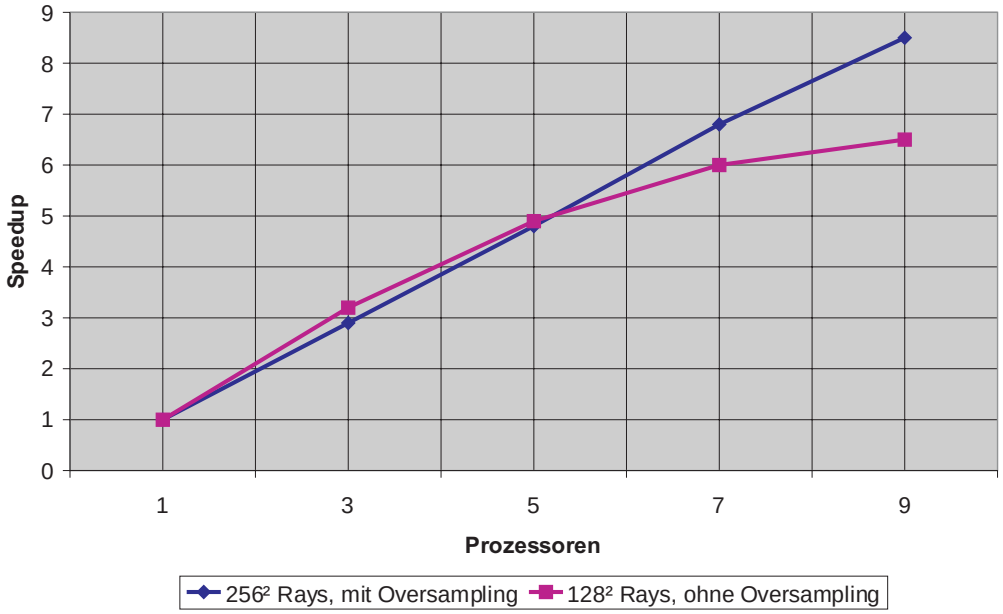
lumenvisualisierungsalgorithmus an. Der zugrundeliegende 3D-Datensatz wird auf allen Rechnern repliziert im Speicher gehalten. Die berechneten Teilbilder werden vom Master-Thread in ein Gesamtbild integriert. Da diese Integration eine Operation in 2D ist, stellt sie keinen sequentiellen Flaschenhals innerhalb der verteilten Berechnung dar.

6.2.3 Messergebnisse

Zur Durchführung von Laufzeitmessungen wurde von einem in Harz gegossenen Hummer ein 3D Datensatz mit 128^3 Voxelwürfeln erstellt. Die parallelen Berechnungen wurden in einem Rechnernetzwerk von 9 PCs unter dem Betriebssystem Windows NT 4.0 ausgeführt. Es bestand aus 4 PCs mit jeweils einem 300 MHz Pentium II Prozessor und 5 PCs mit jeweils einem 200 MHz Pentium II Prozessor. Alle Rechner waren über ein 100 Mbps Ethernet Netzwerk verbunden.

Es wurden zwei Messreihen durchgeführt, die sich hinsichtlich des Rechenaufwands pro Bild unterschieden haben. Bei der ersten Messreihe wurde mit 256^2 Rays gearbeitet und zusätzlich ein aufwändiges Oversampling-Verfahren angewendet. Die sequentielle Erstellung eines Bildes benötigte auf einem der 200 MHz PCs dabei etwa 15 Minuten. Um in den Realzeitbereich vorzustoßen, wurde eine zweite Messreihe durchgeführt, bei der nur noch 128^2 Rays und kein Oversampling verwendet wurden. In Abbildung 6.4 sind die erreichten Beschleunigungen der beiden Messreihen abgebildet. Aufgrund der feinen Granularität bei der zweiten Messreihe nimmt die Effizienz der Berechnung mit zunehmender Zahl an Prozessoren ab.

Abbildung 6.4 Beschleunigung der Volumenvisualisierung



7

Kapitel 7

Systemvergleich

In diesem Kapitel soll DOTS mit anderen Systemumgebungen zum verteilten parallelen Rechnen verglichen werden. Dazu wurden drei Systeme ausgewählt, die alle das Programmiermodell der Threads unterstützen. Sie werden in den folgenden Abschnitten vorgestellt. Das Kapitel schließt mit dem Vergleich der Plattform mit DOTS ab.

7.1 Virtual Data Space

Virtual Data Space (VDS) [26] ist ein System, das speziell zur Erstellung von verteilten parallelen Applikationen, die die gleichzeitige Anwendung mehrerer Parallelisierungsparadigmen erfordern, konzipiert wurde. Für jedes verwendete Paradigma kann dabei eine separate Lastverteilungsstrategie ausgewählt werden. Ein Anwendungsgebiet ist zum Beispiel die Teilklasse der Divide-and-Conquer Algorithmen, die einen schlanken Aufrufbaum aufweisen und der Divide-Schritt in sich parallelisiert werden kann, um den kritischen Pfad zu verkürzen. In [26] wird ein Beispiel gegeben, bei dem es grundsätzlich nicht möglich ist, für die Parallelisierung des Divide-Schritts und der Verteilung der Knoten des Aufrufbaums dasselbe Paradigma zu verwenden.

VDS ist in der Programmiersprache C implementiert und wird in Form einer Programmierbibliothek in das Anwendungsprogramm integriert. Es verwendet zur Realisierung der Kommunikation wahlweise PVM oder eine MPI Implementierung.

VDS unterstützt auch das fork/join Programmiermodell zur verteilten parallelen Programmierung. Programmbeispiel 7.1 zeigt ein VDS Programm zur Be-

rechnung der n-ten Fibonacci Zahl mit dem fork/join Programmiermodell.

Die Programmierschnittstelle für das fork/join Modell besteht im Wesentlichen aus zwei Klassen von Primitiven: Funktionen zum Erzeugen neuer Threads und Funktionen zur Handhabung der Ergebnisse. Die wichtigsten Primitive und Ausführungsprinzipien sollen anhand des Beispielprogramms dargestellt werden. In `main()` wird zunächst durch den Aufruf von `VDS_Define_Class` eine neue Klasse von Threads (`fib_thread`) erzeugt. Mittels `VDS_Declare_Handler` wird dieser Klasse die Funktion `fibonacci` als so genannter *Object-Handler* zugewiesen. Der Object-Handler wird immer dann aufgerufen, wenn vom System ein Thread der Klasse `fib_thread` zur Ausführung ausgewählt wird. Threads sind im System als Instanzen einer Datenstruktur repräsentiert, die beim Aufruf des Object-Handlers als Argument übergeben werden. Im Programmbeispiel wird dazu die Struktur `fib_arg` definiert. Die parallele Ausführung wird durch den Aufruf von VDS_Fork_Join gestartet, wobei als Argument eine Instanz der Datenstruktur übergeben wird. Nach erfolgter Berechnung gibt der Prozess mit der Nummer 0 das Ergebnis aus.

Im Object-Handler `fibonacci` des Programmbeispiels werden nun weitere Kind-Threads zur rekursiven Berechnung der Fibonacci Zahl erzeugt. Das Ergebnis eines Kind-Threads wird durch den speziellen Resultat-Typ `VDS_Result` zum Vater-Thread transferiert. Dazu kann die Datenstruktur eines Threads als Komponenten Variablen des Typs `VDS_Result` enthalten. Wenn ein Kind-Thread ein Resultat erzeugt, wird es in die entsprechende Variable des Vater-Threads geschrieben. Falls aufgrund des Lastverteilungsprozesses Vater- und Kind-Thread auf unterschiedlichen Knoten ausgeführt werden, wird das Resultat transparent zum Knoten des Vater-Threads übertragen. Zum Beispiel wird durch den ersten Aufruf von `VDS_Fork` ein Kind-Thread erzeugt, dessen Resultat mit der Variable `f1` des Vater-Threads verbunden ist. Nach der Ausführung von `VDS_Fork` werden sowohl der Vater-, als auch der Kind-Thread wieder VDS übergeben und können zu einem späteren Zeitpunkt erneut zur Ausführung ausgewählt werden.

Bevor ein Thread auf Ergebnisse von Kind-Threads zugreifen kann, muss er `VDS_Join` aufrufen. Kehrt dieser Aufruf zurück, ist sichergestellt, dass die Ergebnisse des Kind-Threads berechnet und in die Datenstruktur eingetragen sind. Auf die Resultate kann mittels `VDS_Access` zugegriffen werden.

Mittels `VDS_Return` kann ein Thread ein Resultat seinem Vater-Thread zurückgeben. Die Ausführung des Threads ist nach dem Aufruf von `VDS_Return` beendet.

Programmbeispiel 7.1 Naive Berechnung der n-ten Fibonacci Zahl in VDS (zitiert aus [26]).

```
typedef struct {
  int n,step;
  VDS_Result f1, f2; } fib_arg;

void fibonacci (fib_arg *arg) {
  int n = arg->n;
  fib_arg child;
  child.step=0;

  if (n <= 1)
    VDS_Return(sizeof (int), &n);
  else
    switch (++arg->step) {
    case 1:
      {child.n = n-1; VDS_Fork(sizeof (fib_arg), &child, &(arg->f1)); break;}
    case 2:
      {child.n = n-2; VDS_Fork(sizeof (fib_arg), &child, &(arg->f2)); break;}
    case 3:
      {VDS_Join(2, &(arg->f1), &(arg->f2)); break;}
    case 4:
      {int fib =*(int *) VDS_Access (&(arg->f1)) +
         *(int *)VDS_Access (&(arg->f2));
      VDS_Return(sizeof (int), &fib);}
    }
}

int main (int argc, char **argv) {
  int *fib;
  VDS_ClassId fib_thread;
  fib_arg arg;

  VDS_Init (&argc, &argv);
  fib_thread = VDS_Define_class("fib", VDS_Thread, NULL, NULL, NULL);
  VDS_Declare_handler (fib_thread, fibonacci);
  VDS_Configure();

  arg.step = 0;
  arg.n = atoi (argv[1]);
  fib = VDS_ForkJoin (fib_thread, &arg, sizeof (arg));

  if (VDS_id == 0)
    printf (" fib(%d) = %d\n", arg.n, *fib);

  VDS_Finalize ();
}
```

7.2 Cilk

Die Programmiersprache Cilk [69] erweitert ANSI C um einige Schlüsselwörter für die parallele Programmierung mit dem Threads Programmiermodell. Eine besondere Eigenschaft von Cilk ist, dass ein Cilk Programm, das auf einem Prozessor ausgeführt wird, genau dieselbe Semantik wie ein C Programm hat, das aus dem Cilk Programm durch Streichen aller Cilk Schlüsselwörter entsteht. Dieses C Programm wird auch als *Serial Elision* des Cilk Programms bezeichnet. Das Laufzeitsystem von Cilk ermöglicht es außerdem, dass ein Cilk Programm bei Ausführung auf einem Prozessor nur unwesentlich langsamer abläuft als die Serial Elision des Programms.

Anhand der in Programmbeispiel 7.2 gezeigten Berechnung von Fibonacci Zahlen sollen nun die wesentlichen Eigenschaften von Cilk Programmen erläutert werden. Die grundlegenden Schlüsselwörter von Cilk sind `cilk`, `spawn` und `sync`. Das Schlüsselwort `cilk` bezeichnet eine *Cilk Prozedur*. Eine Cilk Prozedur kann durch `spawn` in einem eigenen Thread gestartet werden. Das Schlüsselwort `spawn` darf nur innerhalb einer Cilk Prozedur auftreten. Auf die Resultate der Kind-Threads einer Cilk Prozedur darf erst zugegriffen werden, nachdem eine `sync` Anweisung ausgeführt wurde. Ein Aufruf von `sync` blockiert solange, bis alle Kind-Threads der umgebenden Cilk Prozedur ihre Berechnungen beendet haben. Durch `sync` wird somit ein zu einer Cilk Prozedur lokales Barrier-Konstrukt realisiert.

Im Nachfolgenden soll kurz auf das Ausführungsmodell von Cilk eingegangen werden, eine ausführliche Beschreibung ist in [64] zu finden. Das Cilk System wird durch eine Kombination von Präcompiler und Laufzeitbibliothek realisiert. Ein Präcompiler erzeugt aus Cilk Programmen regulären C Programmcode, der Funktionsaufrufe der Laufzeitbibliothek einbindet. Für jede Cilk Prozedur werden vom Compiler zwei Versionen einer entsprechenden C Prozedur generiert. Diese Versionen werden als *nanoscheduled* und *macroscheduled* bezeichnet. Die nanoscheduled Version kommt bei der lokalen Ausführung der Cilk Prozedur auf einem Prozessor zum Einsatz. Hier wird eine `spawn` Anweisung in einen gewöhnlichen Funktionsaufruf umgesetzt. Der generierte Code führt zusätzlich eine Datenstruktur nach, die dem Aktivierungsblock der Cilk Prozedur ähnlich ist. Auf diese Weise wird bei der Ausführung eines Cilk Programms ein Laufzeitstapel realisiert, auf dessen Elemente aus dem generierten Code zugegriffen werden kann. Die Aktivierungsblöcke des nachgebildeten Laufzeitstapels können zur Lastverteilung auf andere Prozessoren transferiert und die entsprechende Funktion (nach der `spawn` Anweisung) weiter ausgeführt werden. Wenn eine Cilk Prozedur in Form eines transferierten Aktivierungsblocks ausgeführt wird, kommt die macroscheduled Version der Prozedur zum Einsatz. Diese

Programmbeispiel 7.2 Naive Berechnung der n-ten Fibonacci Zahl in Cilk (zitiert aus [69])

```
cilk int fib(int n)
{
  if (n<2) return n;
  else {
    int x, y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}

cilk int main(int argc, char * argv[])
{
  int n, result;
  n = atoi(argv[1]);
  result = spawn fib(n);
  sync;
  printf("Result: %d\n", result);
  return 0;
}
```

enthält zusätzlich `goto` Anweisungen, mit denen die Ausführung der transferierten Cilk Prozedur an der richtigen Stelle wieder aufgenommen wird.

7.3 DTS

Das Distributed Threads System (DTS) [17, 18] ist eine Systemplattform zur Parallelisierung von Anwendungen mittels des Programmiermodells der asynchronen entfernten Prozeduraufrufe. Es unterstützt die Erstellung von parallelen Programmen in den Programmiersprachen C und Fortran. Das System besteht aus einigen Compilern und einem Laufzeitsystem. Die Compiler werden im Wesentlichen eingesetzt, um Programmcode für das Parameter Marshalling zu generieren. Für die Durchführung von verteilten Berechnungen wird ein zentrales und ein dezentrales Lastverteilungsverfahren zur Verfügung gestellt.

Das Programmbeispiel 7.3 zeigt die Berechnung von Fibonacci Zahlen mit DTS.

Mittels `dts_thread_fork` werden Threads zur lokalen Ausführung erzeugt. Die Funktion `fork_fib` dient zur Erzeugung von Threads, die als asynchrone entfernte Prozeduraufrufe ausgeführt werden. Ein Präcompiler erzeugt für jede mit `/* pure */` gekennzeichnete Funktion eine entsprechende DTS Systemfunktion. Im Programm wird aufgrund des Berechnungsaufwands explizit unterschieden, ob eine Funktion als gewöhnlicher Funktionsaufruf, als lokaler Thread oder als asynchroner entfernter Prozeduraufruf ausgeführt werden soll.

7.4 Vergleich mit DOTS

Programmiermodell
DOTS ist zur Erstellung von parallelen Programmen in C++ entworfen und unterstützt im Gegensatz zu den anderen Systemen objektorientierte Techniken. Während DTS und Cilk ausschließlich das Programmiermodell der asynchronen Prozeduraufrufe zur Verfügung stellen, unterstützen DOTS und VDS das weitaus mächtigere Modell der strikten Berechnungen. Gegenüber VDS weisen DOTS Programme eine wesentlich kompaktere und übersichtlichere Struktur auf. VDS und DOTS unterstützen zusätzlich weitere Programmiermodelle, wie etwa Message Passing.

Unterstützung heterogener Umgebungen
Der Einsatz der drei anderen Systeme ist zurzeit auf UNIX basierte Plattformen beschränkt, DOTS kann hingegen auf einer Vielzahl unterschiedlichster Plattformen eingesetzt werden. Da die Implementierung von DOTS vollständig auf der ACE Klassenbibliothek aufbaut, ist die Portierung auf weitere Plattformen problemlos möglich. DTS und VDS realisieren die Kommunikation mittels PVM bzw. MPI, während DOTS einen eigenen, hochoptimierten Kommunikationskernel verwendet, der direkt auf dem TCP/IP Protokoll basiert. Da TCP auf fast jeder Plattform verfügbar ist, wird die umfassende Portabilität von DOTS sichergestellt. Außerdem werden durch diesen Ansatz auch parallele Berechnungen in geografisch ausgedehnten Rechnernetzwerken ermöglicht. Die verteilte Version von Cilk liegt zurzeit nur als Prototyp vor. Dieser verwendet UDP zur Kommunikation.

Systemrealisierung
Während in Cilk und DTS keine anwendungsspezifischen Lastverteilungsverfahren integriert werden können, ist dies in VDS und DOTS möglich. Mittels des Load Monitoring Framework von DOTS können Programmcode und Lastverteilungscode vollkommen getrennt voneinander entwickelt und orthogonal verwendet werden.

Cilk wird mittels eines Präcompilers realisiert. Gegenüber den anderen, im

Programmbeispiel 7.3 Naive Berechnung der n-ten Fibonacci Zahl in DTS (Dieses Programm ist der Distribution von DTS 2.1.5 entnommen; die Funktion `main()` wurde leicht angepasst.)

```
/* pure */ int fib( /*in*/ int n)
{
  int sum1,sum2;
  dts_thread   dts_thread_id1,dts_thread_id2;
  dts_t        dts_id1,dts_id2;
  int n1,n2;

  n1=n-1;
  n2=n-2;

  if (n<=1) return 1;                          /* trivial case */
  else if (n<20) return fib(n2)+fib(n1);       /* "hard" case */
  else if (n<25) {                             /* "harder" case */
    dts_thread_id1=dts_thread_fork(fib,n1,0);
    dts_thread_id2=dts_thread_fork(fib,n2,0);
    sum1=(int)dts_thread_join(dts_thread_id1);
    sum2=(int)dts_thread_join(dts_thread_id2);
  } else {                                     /* "complex" case */
    dts_id1=fork_fib(n1);
    dts_id2=fork_fib(n2);
    sum1=(int)join_fib(dts_id1);
    sum2=(int)join_fib(dts_id2);
  }
  return sum1+sum2;
}

int main(int argc, char *argv[])
{
  int n, result;
  n = atoi(argv[1]);
  dts_init(argc,argv,NULL);

  result=fib(n);
  printf("fib(%d)=%d\n",n,result); }

  dts_leave();
}
```

Tabelle 7.1 Zusammenfassung des Systemvergleichs

Eigenschaft	VDS	Cilk	DTS	DOTS
Objektorientierte Programmierung	Nein	Nein	Nein	Ja
Einsatz in heterog. Umgebungen	nur UNIX	nur UNIX	nur UNIX	Ja
Unterstützung strikter Berechnungen	Ja	Nein	Nein	Ja
Weitere Programmiermodelle	Ja	Nein	Nein	Ja
Präcompiler erforderlich	Nein	Ja	Ja	Nein

Wesentlichen bibliotheksbasierten Systemen, kann eine kompaktere Programmierschnittstelle zur Verfügung gestellt werden. Der eigentlich ausgeführte Programmcode ist bei Cilk jedoch wesentlich umfangreicher und komplexer als der vom Programmierer erstellte Programmcode. Dies erschwert die Fehlersuche erheblich, da zum Beispiel Debugging Werkzeuge nur den generierten Programmcode anzeigen.

Zusammenfassung

In Tabelle 7.1 sind nochmals die wichtigsten Eigenschaften der betrachteten Systeme zusammengefasst.

8

Kapitel 8

Zusammenfassung

8.1 Ergebnisse

Diese Arbeit beschreibt verschiedene Aspekte der Realisierung und der Anwendung der Systemumgebung DOTS. Mittels DOTS können parallele C++ Programme erstellt und in stark heterogenen Umgebungen ausgeführt werden. Das System wurde auf einer Vielzahl von Plattformen – vom Realzeit Mikrokernel bis hin zum Mainframe Cluster – erfolgreich eingesetzt. Die vorliegende Arbeit liefert im Einzelnen die folgenden Beiträge in den angeführten Gebieten:

Programmiermodell
DOTS sieht zur Erstellung von parallelen Programmen mehrere Programmiermodelle vor. Das zentrale Programmiermodell stellt das Modell der verteilten Threads dar. Mit DOTS können Programme erstellt werden, die zur Klasse der strikten Berechnungen gehören. Zur Beschreibung derartiger Berechnungen wird ein einfacheres und gleichzeitig mächtigeres API zur Verfügung gestellt, als dies bei vergleichbaren Systemplattformen der Fall ist. Durch die Integration zweier Checkpointingverfahren, die das spezifische Ausführungsmuster von Berechnungen mit verteilten Threads berücksichtigen, kann die Zuverlässigkeit der Ausführung von Berechnungen wesentlich erhöht werden. Zusätzlich stellt DOTS ein allgemeines, objektorientiertes, verteiltes Task Modell, sowie darauf aufbauend, aktive Nachrichten und eine einfache Form von mobilen Agenten als weitere Programmiermodelle zur Verfügung.

Systemarchitektur
Eine wesentliche Eigenschaft der Systemarchitektur von DOTS ist die vollständige Trennung von Ausführungs- und Verteilungsaspekten. Somit kann ein mit dem DOTS Thread API erstelltes Programm ohne Änderung, d.h. auch ohne

Neuübersetzung, auf Parallelrechnern mit gemeinsamem, sowie auch auf Parallelrechnern mit verteiltem Speicher effizient ausgeführt werden. Mittels des Load Monitoring Framework lassen sich anwendungsspezifische Lastverteilungsverfahren transparent in das System integrieren. Ein auf XML basiertes Verfahren ermöglicht es, dass im selben Dokument sowohl das verwendete (heterogene) Rechnernetzwerk definiert, als auch die darin durchgeführten Berechnungen umfassend dokumentiert werden können.

Implementierung

Ziel der Implementierung von DOTS war die Unterstützung stark heterogener Umgebungen. Diese Problemstellung umfasst im Wesentlichen den Aspekt der Portabilität und den Aspekt der Interoperabilität. Hierzu wurden die folgenden Beiträge geleistet: Zur Homogenisierung von Betriebsystemschnittstellen in C++ wird ein neuartiges Verfahren, das an der Traits Technik angelehnt ist, untersucht. Das vorgestellte Verfahren ist hinsichtlich der Effizienz, der Portierbarkeit und der Überprüfbarkeit auf Vollständigkeit der Implementierung anderen bekannten Verfahren überlegen. Es wird ein verteilter Cache Speicher für TCP Verbindungen vorgestellt, der die Effizienz der Kommunikation wesentlich verbessert. Mittels Laufzeitmessungen in einem stark heterogenen Rechnernetzwerk wurde die Reduzierung des Ausführungsoverheads von DOTS Threads um bis zu einem Faktor von 6.3 nachgewiesen.

Anwendung

Unter Verwendung von DOTS wurde die parallele Version eines dynamisch lernenden Algorithmus zur booleschen Erfüllbarkeitsprüfung realisiert. Die Parallelisierung läuft auf zwei Ebenen ab. Mittels DOTS Threads wird eine parallele Suche durchgeführt. Dazu wird ein dynamisches Verfahren zur Suchraumaufteilung verwendet. Zum Austausch des durch den Lernprozess generierten Wissens wird ein auf mobilen Agenten basierter Ansatz vorgestellt. Für eine Reihe praktisch relevanter Probleme durchgeführte Laufzeitmessungen zeigen, dass durch dieses neuartige Parallelisierungsverfahren signifikante Beschleunigungen der Berechnung erzielt werden können.

8.2 Publikationen

Aus der vorliegenden Arbeit gingen die nachfolgend aufgeführten Publikationen hervor.

- M. Meißner, T. Hüttner, W. Blochinger, and A. Weber. Parallel direct volume rendering on PC networks. In H. R. Arabnia, editor, *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA '98)*, Las Vegas, NV, U.S.A., July 1998. CSREA Press.

Gegenstand dieser Publikation ist die in Abschnitt 6.2 dargestellte Parallelisierung eines direkten Volumenvisualisierungsverfahrens mittels DOTS.

- W. Blochinger, W. Küchlin, and A. Weber. The distributed object-oriented threads system DOTS. In A. Ferreira, J. Rolim, H. Simon, and S.-H. Teng, editors, *Fifth Intl. Symp. on Solving Irregularly Structured Problems in Parallel (IRREGULAR '98)*, number 1457 in LNCS, pages 206–217, Berkeley, CA, U.S.A., August 1998. Springer-Verlag.

  In diesem Artikel werden die grundlegenden Primitive des DOTS Thread API vorgestellt. Außerdem wird auf das in Abschnitt 4.3 gezeigte Verfahren zur Objektkommunikation in C++ eingegangen.

- W. Blochinger, W. Küchlin, C. Ludwig, and A. Weber. An object-oriented platform for distributed high-performance Symbolic Computation. *Mathematics and Computers in Simulation*, 49:161–178, 1999.

  Dieser Artikel gibt einen Gesamtüberblick über das DOTS System. Zusätzlich wird die in Abschnitt 6.1 diskutierte parallele Faktorisierungsapplikation auf Basis der Methode der elliptischen Kurven vorgestellt.

- R.-D. Schimkat, W. Blochinger, C. Sinz, M. Friedrich, and W. Küchlin. A service-based agent framework for distributed Symbolic Computation. In M. Bubak, R. Williams, H. Afsarmanesh, and B. Hertzberger, editors, *Proc. 8th Intl. Conf. on High Performance Computing and Networking Europe, HPCN 2000*, number 1823 in LNCS, pages 644–656, Amsterdam, Netherlands, May 2000. Springer-Verlag.

  Diese Publikation setzt sich mit der Realisierung eines parallelen SAT Beweisers mittels des Agentensystems OKEANOS auseinander. Ein auf den in Abschnitt 5.4 dargestellten grundlegenden Parallelisierungstechniken für den klassischen Davis-Putnam Algorithmus basierter SAT Beweiser dient als Berechnungskern. Die Lastverteilung wird durch Java basierte mobile Agenten vorgenommen.

- W. Blochinger, R. Bündgen, and A. Heinemann. Dependable high performance computing on a Parallel Sysplex cluster. In H. R. Arabnia, editor, *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and*

Applications (PDPTA 2000), volume 3, pages 1627–1633, Las Vegas, NV, U.S.A., June 2000. CSREA Press.

Die Erweiterung von DOTS um verschiedene Checkpointing-Verfahren ist Gegenstand dieses Artikels. Weiterhin wird auf die Portierung von DOTS auf die IBM Parallel Sysplex Plattform eingegangen.

- W. Blochinger. Distributed high performance computing in heterogeneous environments with DOTS. In *Proc. of Intl. Parallel and Distributed Processing Symp. (IPDPS 2001)*, page 90, San Francisco, CA, U.S.A., April 2001. IEEE Computer Society Press.

  In dieser Publikation werden verschiedene Aspekte der Verwendung von DOTS in heterogenen Rechnernetzwerken dargestellt. Es werden die wesentlichen Eigenschaften der in Abschnitt 3.3 beschriebenen Systemarchitektur von DOTS diskutiert. Desweiteren wird auf den XML basierten Ansatz zur Beschreibung und Dokumentation von Berechnungen (siehe Abschnitt 3.5), sowie auf die Optimierung verbindungsorientierter Kommunikation in heterogenen verteilten Systemen mittels Connection Caching eingegangen (siehe Abschnitt 4.2).

- C. Sinz, W. Blochinger, and W. Küchlin. PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications. In H. Kautz and B. Selman, editors, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*, Boston, MA, June 2001. Elsevier Science Publishers.

  Dieser Artikel beschreibt die Anwendung eines mittels DOTS realisierten parallelen SAT Beweisers mit Lemma-Austausch für Parallelrechner mit gemeinsamem Speicher auf mehrere praktisch relevante Probleminstanzen.

- W. Blochinger, C. Sinz, and W. Küchlin. Distributed parallel SAT checking with dynamic learning using DOTS. In T. Gonzales, editor, *Proc. of the IASTED Intl. Conference Parallel and Distributed Computing and Systems (PDCS 2001)*, pages 396–401, Anaheim, CA, August 2001. ACTA Press.

  In diesem Artikel wird die Erweiterung der Programmierschnittstelle von DOTS um das Task API und das Autonomous Task

API, sowie die damit einhergehende Verfeinerung der Systemarchitektur gezeigt. Im zweiten Teil des Artikels wird beschrieben, wie diese neuen Eigenschaften von DOTS zur Realisierung eines verteilten parallelen SAT Beweisers mit Lemma-Austausch durch mobile Agenten eingesetzt wurden (vgl. Kapitel 5).

- W. Blochinger, C. Sinz, and W. Küchlin. Parallel consistency checking of automotive product data. In *Proc. of Intl. Conference Parallel Computing ParCo 2001*, Naples, Italy, September 2001.

  Gegenstand dieses Artikels ist die Beschreibung der in Abschnitt 5.7 dargestellten Anwendung eines auf DOTS basierten verteilten parallelen SAT Beweisers zur Konsistenzprüfung von Produktdokumentation in der Automobilindustrie.

A

Anhang A

DOTSML

A.1 Die DOTSML DTD

File: dotsml.dtd

```
<!ELEMENT dots (configuration, computation*)>

<!ELEMENT configuration (program, global, node+)>

<!ELEMENT program (source+, binary+)>

<!ELEMENT source (#PCDATA)>
<!ATTLIST source
   filename  CDATA   #IMPLIED
   comment   CDATA   #IMPLIED
>

<!ELEMENT binary (#PCDATA)>
<!ATTLIST binary
   arch     CDATA   #IMPLIED
   os       CDATA   #IMPLIED
   compiler CDATA   #IMPLIED
   comment  CDATA   #IMPLIED
>

<!ELEMENT global EMPTY>
<!ATTLIST global
```

```
   diststrategy    CDATA   "1"
   archsize        CDATA   "1024"
   archdelta       CDATA   "1024"
   lifetime        CDATA   "24"
>

<!ELEMENT node (startup)>
<!ATTLIST node
   name      CDATA   #REQUIRED
   arch      CDATA   #IMPLIED
   speed     CDATA   #IMPLIED
   numcpu    CDATA   #IMPLIED
   ram       CDATA   #IMPLIED
   nic       CDATA   #IMPLIED
   os        CDATA   #IMPLIED
   workers   CDATA   "1"
   rbufsize  CDATA   "-1"
   sbufsize  CDATA   "-1"
   backlog   CDATA   "16"
>

<!ELEMENT startup (#PCDATA)>

<!ELEMENT computation (input?, output?, comment?, logging?)>

<!ATTLIST computation
   name      CDATA   #IMPLIED
   date      CDATA   #IMPLIED
   time      CDATA   #IMPLIED
>

<!ELEMENT input (#PCDATA)>
<!ELEMENT output (#PCDATA)>
<!ELEMENT result (#PCDATA)>
<!ELEMENT comment (#PCDATA)>

<!ELEMENT logging (log)*>

<!ELEMENT log (#PCDATA)>
<!ATTLIST log
   node      CDATA   #IMPLIED
   level     CDATA   #IMPLIED
>
```

A.2 Beispieldokument

File: fibtest.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dots SYSTEM "dotsml.dtd">
<dots>
  <configuration>
    <program>
      <source filename="fibonacci.cpp"> <![CDATA[
        int fib(int n) {
          if (n<2)
            return n;
          else {
            dots_hyperfork(fib, n-1);
            dots_hyperfork(fib, n-2);

            dots_abort();
          }
        }

        int main(int argc, char* argv[])
        {
          dots_reg(fib);
          dots_init(argc, argv);

          int n, res, result=0;
          n = atoi(argv[1]);

          DOTS_Thread_Group thread_group;
          dots_fork(thread_group, fib, n);

          while (dots_join(thread_group, res) != -1)
            result = result + res;

          printf("Result: %d\n", result);

          return 0;
        }
      ]]> </source>

      <binary arch="x86" os="Windows2000">
        Z:\dots\fibonacci\fibonacci.exe
```

```
      </binary>

      <binary arch="UltraSparc" os="Solaris7">
        /home/bloching/dots/fibonacci/fibonacci
      </binary>

    </program>

    <global archsize="8192" archdelta="512"
            lifetime="2" diststrategy="1"/>

    <node name="aquarius" arch="x86" speed="500Mhz" numcpu="2"
          ram="256MB" nic="100Mbps" os="Windows2000" workers="2">
      <startup>rsh aquarius</startup>
    </node>
    <node name="luna" arch="UltraSparc" speed="400Mhz" numcpu="4"
          ram="2GB" nic="100Mbps" os="Solaris7" workers="4">
      <startup>ssh -l bloching luna</startup>
    </node>
  </configuration>

  <computation name="test1" date="21.11.2001" time="22:27">
    <input> 5 </input>
    <output> 5 </output>

    <logging>
      <log node="0" level="INFO">start time : 22.27.19.227000</log>
      <log node="0" level="INFO">ready time : 22.27.20.018000</log>
    </logging>
  </computation>

  <computation name="test2" date="24.11.2001" time="00:30">
    <input> 6 </input>
    <output> 8 </output>

    <logging>
      <log node="0" level="INFO">start time : 00.30.41.789000</log>
      <log node="0" level="INFO">ready time : 00.30.42.223000</log>
    </logging>
  </computation>

</dots>
```

B

Anhang B

Beispielimplementierungen der Anpassungsschicht einer abstrakten Klassenschnittstelle

In diesem Anhang wird für die in Abschnitt 4.1 diskutierten Verfahren zur Realisierung der Anpassungsschicht einer abstrakten Klassenschnittstelle jeweils eine vollständige Beispielimplementierung angegeben. Bei allen betrachteten Beispielen wird davon ausgegangen, dass gemäß dem Wrapper Facade Entwurfsmuster eine abstrakte Schnittstelle `Abstract_Interface` entworfen wurde. Diese Schnittstelle kapselt die Methode `Y syscall(X)` und stellt zusätzlich die Typen für das Argument und Resultat von `syscall`, sowie eine Konstante `xconst` des Typs `X` zur Verwendung in Anwendungsprogrammen zur Verfügung.

B.1 Präprozessor Methode

File: Abstract_Interface.h

```
#ifndef ABSTRACT_INTERFACE_H
#define ABSTRACT_INTERFACE_H

#include "Adaptation.h"

class Abstract_Interface{
public:
  Y syscall(X arg) {
    return Adaptation::syscall(arg);
  }
};
```

```
typedef Abstract_Interface INTERFACE;

#endif /* ABSTRACT_INTERFACE_H */
```

File: Adaptation.h

```
#ifndef ADAPTATION_H
#define ADAPTATION_H

#ifdef SYS_A
#include "sys/sysA.h"
#endif

#ifdef SYS_B
#include "sys/sysB.h"
#endif

// types

#ifdef SYS_A
  typedef sysa_x_t X;
  typedef sysa_y_t Y;
#endif

#ifdef SYS_B
  typedef sysb_x_t X;
  typedef sysb_y_t Y;
#endif

// constants

#ifdef SYS_A
  const X xconst = sysa_x_const;
#endif

#ifdef SYS_B
  const X xconst = sysb_x_const;
#endif


class Adaptation {
public:
  static Y syscall(X arg) {
```

```
#ifdef SYS_A
  return ::syscall_A(arg);
#endif

#ifdef SYS_B
  return ::syscall_B(arg);
#endif

  }
};

#endif /* ADAPTATION_H */
```

File: main.cpp

```
#include <iostream>
#include "Abstract_Interface.h"

int main(int argc, char* argv[])
{
  INTERFACE* interface = new INTERFACE;

  // types
  Y res;

  // constants
  const X arg = xconst;

  // functions
  res = interface->syscall(arg);

  std::cout << "res = " << res << std::endl;

  return 0;
}
```

B.2 Abstract Factory Methode

File: Abstract_Interface.h

```
#ifndef ABSTRACT_INTERFACE_H
```

```
#define ABSTRACT_INTERFACE_H

#ifdef SYS_A
#include "Types_Sys_A.h"
#endif

#ifdef SYS_B
#include "Types_Sys_B.h"
#endif

#include "Factory.h"


class Abstract_Interface {
public:
  virtual X xconst() = 0;
  virtual Y syscall(X arg) = 0;
};

typedef Abstract_Interface INTERFACE;

extern Factory* IFACTORY;

#endif /* ABSTRACT_INTERFACE_H */
```

File: Abstract_Interface.cpp

```
#include "Abstract_Interface.h"

#ifdef SYS_A
#include "Factory_Sys_A.h"
Factory* IFACTORY = new Factory_Sys_A;
#endif

#ifdef SYS_B
#include "Factory_Sys_B.h"
Factory* IFACTORY = new Factory_Sys_B;
#endif
```

File: Adaptation_Sys_A.h

```
#ifndef ADAPTATION_SYS_A_H
#define ADAPTATION_SYS_A_H
```

```
#include "Abstract_Interface.h"
#include "Types_Sys_A.h"

#include <sys/sysA.h>

class Adaptation_Sys_A : public Abstract_Interface
{
public:
  X xconst() {
    return sysa_x_const;
  }

  Y syscall(X arg) {
    return ::syscall_A(arg);
  }
};

#endif /* ADAPTATION_SYS_A_H */
```

File: Adaptation_Sys_B.h

```
#ifndef ADAPTATION_SYS_B_H
#define ADAPTATION_SYS_B_H

#include "Abstract_Interface.h"
#include "Types_Sys_B.h"

#include <sys/sysB.h>

class Adaptation_Sys_B : public Abstract_Interface
{
public:
  X xconst() {
    return sysb_x_const;
  }

  Y syscall(X arg) {
    return ::syscall_B(arg);
  }
};

#endif /* ADAPTATION_SYS_B_H */
```

File: Factory.h

```
#ifndef FACTORY_H
#define FACTORY_H

class Abstract_Interface;

class Factory {
public:
  virtual Abstract_Interface* new_interface() = 0;
};

#endif /* FACTORY_H */
```

File: Factory_Sys_A.h

```
#ifndef FACTORY_SYS_A_H
#define FACTORY_SYS_A_H

#include "Factory.h"
#include "Adaptation_Sys_A.h"

class Factory_Sys_A : public Factory {
public:
  Abstract_Interface* new_interface() {
    return new Adaptation_Sys_A;
  }
};

#endif /* FACTORY_SYS_A_H */
```

File: Factory_Sys_B.h

```
#ifndef FACTORY_SYS_B_H
#define FACTORY_SYS_B_H

#include "Factory.h"
#include "Adaptation_Sys_B.h"

class Factory_Sys_B : public Factory {
public:
  Abstract_Interface* new_interface() {
    return new Adaptation_Sys_B;
```

```
  }
};

#endif /* FACTORY_SYS_B_H */
```

File: Types_Sys_A.h

```
#ifndef TYPES_SYS_A_H
#define TYPES_SYS_A_H

#include <sys/sysA.h>

typedef sysa_x_t X;
typedef sysa_y_t Y;

#endif /* TYPES_SYS_A_H */
```

File: Types_Sys_B.h

```
#ifndef TYPES_SYS_B_H
#define TYPES_SYS_B_H

#include <sys/sysB.h>

typedef sysb_x_t X;
typedef sysb_y_t Y;

#endif /* TYPES_SYS_B_H */
```

File: main.cpp

```
#include <iostream>
#include "Abstract_Interface.h"

int main(int argc, char* argv[])
{
  INTERFACE* sysinterface = IFACTORY->new_interface();

  // types
  Y res;

  // constants
```

```
  const X arg = sysinterface->xconst();

  // functions
  res = sysinterface->syscall(arg);

  std::cout << "res = " << res << std::endl;

  return 0;
}
```

B.3 Template Methode

File: Abstract_Interface.h

```
#ifndef ABSTRACT_INTERFACE_H
#define ABSTRACT_INTERFACE_H

template<class Adaptation>
class Abstract_Interface {
public:

  // types
  typedef typename Adaptation::X X;
  typedef typename Adaptation::Y Y;

  // constants
  static const X xconst = Adaptation::xconst;

  // functions
  Y syscall(X arg) {
    return Adaptation::syscall(arg);
  }
};

#ifdef SYS_A
#include "Adaptation_Sys_A.h"
typedef Abstract_Interface<Adaptation_Sys_A> INTERFACE;
#endif

#ifdef SYS_B
#include "Adaptation_Sys_B.h"
typedef Abstract_Interface<Adaptation_Sys_B> INTERFACE;
```

```
#endif

typedef INTERFACE::X X;
typedef INTERFACE::Y Y;

#endif /* ABSTRACT_INTERFACE_H */
```

File: Adaptation_Sys_A.h

```
#ifndef ADAPTATION_SYS_A_H
#define ADAPTATION_SYS_A_H

#include <sys/sysA.h>

class Adaptation_Sys_A {
public:

  // types
  typedef sysa_x_t X;
  typedef sysa_y_t Y;

  // constants
  static const X xconst = sysa_x_const;

  // functions
  static Y syscall(X arg) {
    return ::syscall_A(arg);
  }
};

#endif /* ADAPTATION_SYS_A_H */
```

File: Adaptation_Sys_B.h

```
#ifndef ADAPTATION_SYS_B_H
#define ADAPTATION_SYS_B_H

#include <sys/sysB.h>

class Adaptation_Sys_B {
public:

  // types
```

```
  typedef sysb_x_t X;
  typedef sysb_y_t Y;

  // constants
  static const X xconst = sysb_x_const;

  // functions
  static Y syscall(X arg) {
    return ::syscall_B(arg);
  }
};

#endif /* ADAPTATION_SYS_B_H */
```

File: main.cpp

```
#include <iostream>
#include "Abstract_Interface.h"

int main(int argc, char* argv[])
{
  INTERFACE* interface = new INTERFACE;

  // types
  Y res;

  // constants
  const X arg = INTERFACE::xconst;

  // functions
  res = interface->syscall(arg);

  std::cout << "res = " << res << std::endl;

  return 0;
}
```

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Verzeichnis der Programmbeispiele

Literaturverzeichnis

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[2] S. Albers. Generalized connection caching. In *Proc. 12th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 70–78, 2000.

[3] J. Allard. Plug into serious network programming with the Windows Sockets API. *Microsoft Systems Journal*, 8(7), July 1993.

[4] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 3rd edition, 2000.

[5] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[6] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, number 1579 in LNCS. Springer-Verlag, 1999.

[7] W. Blochinger. Distributed high performance computing in heterogeneous environments with DOTS. In *Proc. of Intl. Parallel and Distributed Processing Symp. (IPDPS 2001)*, page 90, San Francisco, CA, U.S.A., April 2001. IEEE Computer Society Press.

[8] W. Blochinger, R. Bündgen, and A. Heinemann. Dependable high performance computing on a Parallel Sysplex cluster. In H. R. Arabnia, editor,

Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000), volume 3, pages 1627–1633, Las Vegas, NV, U.S.A., June 2000. CSREA Press.

[9] W. Blochinger, W. Küchlin, C. Ludwig, and A. Weber. An object-oriented platform for distributed high-performance Symbolic Computation. *Mathematics and Computers in Simulation*, 49:161–178, 1999.

[10] W. Blochinger, W. Küchlin, and A. Weber. The distributed object-oriented threads system DOTS. In A. Ferreira, J. Rolim, H. Simon, and S.-H. Teng, editors, *Fifth Intl. Symp. on Solving Irregularly Structured Problems in Parallel (IRREGULAR '98)*, number 1457 in LNCS, pages 206–217, Berkeley, CA, U.S.A., August 1998. Springer-Verlag.

[11] W. Blochinger, C. Sinz, and W. Küchlin. Distributed parallel SAT checking with dynamic learning using DOTS. In T. Gonzales, editor, *Proc. of the IASTED Intl. Conference Parallel and Distributed Computing and Systems (PDCS 2001)*, pages 396–401, Anaheim, CA, August 2001. ACTA Press.

[12] W. Blochinger, C. Sinz, and W. Küchlin. Parallel consistency checking of automotive product data. In *Proc. of Intl. Conference Parallel Computing (ParCo 2001)*, Naples, Italy, September 2001.

[13] R. D. Blumofe and C. E. Leiserson. Space efficient scheduling of multithreaded computations. In *Proc. of the Twenty Fifth Annual ACM Symp. on Theory of Computing*, pages 362–371, San Diego, CA, May 1993.

[14] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *35th Annual Symp. on Foundations of Computer Science (FOCS '94)*, pages 356–368, Mexico, November 1994.

[15] F. Bodin, P. Beckman, D. B. Gannon, S. Narayana, and S. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), 1993.

[16] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. *Extensible Markup Language (XML) 1.0*. World Wide Web Consortium (W3C), 1998. *http://www.w3.org/TR/REC-xml*.

[17] T. Bubeck. *Untersuchungen zur Parallelisierung und Ausführung von Programmen in verschiedenen parallelen Speicherarchitekturen*. PhD thesis, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, June 1998.

[18] T. Bubeck, M. Hiller, W. Küchlin, and W. Rosenstiel. Distributed symbolic computation with DTS. In A. Ferreira and J. Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems, 2nd Intl. Workshop, IRREGULAR'95*, volume 980 of *LNCS*, pages 231–248, Lyon, France, Sep 1995. Springer-Verlag.

[19] J. Carreira and J. Silva. Dependable clustered computing. In R. Buyya, editor, *High Performance Cluster Computing*, volume 1. Prentice Hall, 1999.

[20] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, 1989. September.

[21] J. Clark. *XSL Transformations (XSLT) 1.0*. World Wide Web Consortium (W3C), 1999. *http://www.w3.org/TR/xslt*.

[22] J. Clark and S. DeRose. *XML Path Language (XPath) 1.0*. World Wide Web Consortium (W3C), 1999. *http://www.w3.org/TR/xpath*.

[23] E. Cohen, H. Kaplan, and U. Zwick. Connection caching. In *Proc. of the 31st Annual ACM Symp. on Theory of Computing*, pages 612–621, 1999.

[24] S. A. Cook. The complexity of theorem proving procedures. In *3rd Symp. on Theory of Computing*, pages 151–158. ACM press, 1971.

[25] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[26] T. Decker. Virtual data space - load balancing for irregular applications. *Parallel Computing*, 26:1825–1860, 2000.

[27] M. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21:948–960, 1972.

[28] I. Foster. Languages for parallel processing. In J. Blazewicz, K. Ecker, B. Plateau, and D. Trystram, editors, *Handbook on Parallel and Distributed Processing*, chapter 3. Springer-Verlag, 2000.

[29] I. Foster and C. Kesselman. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[30] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *The International Journal of High Performance Computing Applications*, 15(3):200–222, Fall 2001.

[31] E. Freuder. The role of configuration knowledge in the business process. *IEEE Intelligent Systems*, 13(4):29–31, July/August 1998.

[32] M. Fujita, J. Slaney, and F. Bennett. Automatic generation of some results in finite algebra. In *Proc. of the International Joint Conference on Artificial Intelligence IJCAI*, pages 52–59, Chambéry, France, 1993. Morgan Kaufmann.

[33] W. Furmanski, C. Faigle, T. Haupt, J. Niemiec, M. Podgorny, and D. Simoni. MOVIE model for open systems based high performance distributed computing. *Concurrency: Practice and Experience*, 5(4):287–308, June 1993.

[34] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[35] The Globus Project. *http://www.globus.org*.

[36] A. S. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. *IEEE Computer*, 26(5):39–51, May 1993.

[37] T. Grundmann, M. Ritt, and W. Rosenstiel. TPO++: An object-oriented message-passing library in C++. In D. J. Lilja, editor, *Proc. of the 2000 International Conference on Parallel Processing*, pages 43–50, Toronto, Canada, 2000. IEEE Computer Society.

[38] R. H. Halstead. Parallel symbolic computation. *IEEE Computer*, 19(8):35–43, August 1986.

[39] A. Heinemann. Fault tolerance in parallel computing – a checkpoint and recovery extension to the distributed object-oriented threads system. Master's thesis, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, November 1999.

[40] H. Hellwagner and A. Reinefeld, editors. *SCI: Scalable Coherent Interface – Architecture and Software for High-Performance Compute Clusters*. Springer-Verlag, 1999.

[41] J. L. Hennessy and D. A. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1995.

[42] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, 1995.

[43] P. Hudack and J. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.

[44] Inmos Ltd.˙ *Occam Programming Manual*, 1984.

[45] Institute of Electrical and Electronic Engineers, Inc., IEEE, New York, N.Y. *Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C language] (ANSI)*, 1996. IEEE Standard 1003.1, (Also ISO/IEC 9945-1:1996).

[46] J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[47] D. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–491, September 1990.

[48] R. M. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM*, 40(3):765–789, July 1993.

[49] W. Küchlin and A. Weber. *Einführung in die Informatik – Objektorientiert mit Java*. Springer-Verlag, 2000.

[50] H. Kleine-Büning and T. Lettmann. *Aussagenlogik – Deduktion und Algorithmen*. B.G. Teubner Verlag, 1994.

[51] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[52] T. Krätschmer. Entwurf und Implementierung einer plattformunabhängigen C++ Klassenbibliothek. Studienarbeit, Juli 1999.

[53] W. Küchlin and C. Sinz. Proving consistency assertions for automotive product data management. *Journal of Automated Reasoning*, 24(1-2):145–163, February 2000.

[54] W. W. Küchlin and J. A. Ward. Experiments with Virtual C Threads. In *Proc. Fourth IEEE Symp. on Parallel and Distributed Processing*, pages 50–55, Dallas, TX, Dec 1992. IEEE Press.

[55] H. W. Lenstra. Factoring integers with elliptic curves. *Annals of Mathematics*, 126:649–673, 1987.

[56] LiDIA – a C++ library for computational number theory. *http://www.informatik.tu-darmstadt.de/TI/LiDIA*.

[57] D. Litaize, A. Mzoughi, C. Rochange, and P. Sainrat. Architecture of parallel and distributed systems. In J. Blazewicz, K. Ecker, B. Plateau, and D. Trystram, editors, *Handbook on Parallel and Distributed Processing*, chapter 4. Springer-Verlag, 2000.

[58] F. Massacci and L. Marraro. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning*, 24(1-2):165–203, Feb 2000.

[59] O. McBryan. An overview of message passing environments. *Parallel Computing*, 20:417–444, 1994.

[60] M. Meißner, T. Hüttner, W. Blochinger, and A. Weber. Parallel direct volume rendering on PC networks. In H. R. Arabnia, editor, *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA '98)*, Las Vegas, NV, U.S.A., July 1998. CSREA Press.

[61] Message Passing Interface Forum, University of Tennessee, Knoxville, Tenn. *Document for a Standard Message-Passing Interface*, 1993.

[62] Microsoft Corporation, Redmond, WA. *Microsoft Win32 API Programmer's Reference*, 1993.

[63] Microsoft Corporation, Redmond, WA. *Programming Microsoft Visual C++*, 5th edition, 1998.

[64] MIT Laboratory for Computer Science, Cambridge, MA. *Cilk 5.3 Reference Manual*, June 2000.

[65] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computations*, 48(177):243–264, January 1987.

[66] MPI: A message passing interface. In *Proc. of Supercomputing*, pages 878–883. IEEE Computer Society Press, 1993.

[67] N. C. Myers. Traits: A new and useful template technique. *C++ Report*, June 1995.

[68] M. Neumann. Implementierung einer auf Volumendaten basierten Integration und Auswertung von Tiefenbildern sowie deren Umrechnung in polygonale Netze mittels eines nebenläufigen Marching Cubes-Verfahrens. Master's thesis, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 2000.

[69] K. H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, June 1998.

[70] G. Scheja and U. Storch. *Lehrbuch der Algebra – Teil 1*. B.G. Teubner, Stuttgart, 1980.

[71] R.-D. Schimkat, W. Blochinger, C. Sinz, M. Friedrich, and W. Küchlin. A service-based agent framework for distributed Symbolic Computation. In M. Bubak, R. Williams, H. Afsarmanesh, and B. Hertzberger, editors, *Proc. 8th Intl. Conf. on High Performance Computing and Networking Europe, HPCN 2000*, number 1823 in LNCS, pages 644–656, Amsterdam, Netherlands, May 2000. Springer-Verlag.

[72] D. C. Schmidt. ACE: An object-oriented framework for developing distributed applications. In *Proc. of the 6th USENIX C++ Technical Conference*, Cambridge, MA, April 1994. USENIX Association.

[73] D. C. Schmidt. Wrapper-facade – a structural pattern for encapsulating functions within classes. *C++ Report*, Feb 1999.

[74] J. P. M. Silva and K. A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. In *Proc. of the IEEE International Conference on Tools with Artificial Intelligence*, Nov 1996.

[75] V. Simonis. Objektorientierter Entwurf eines Applikationsservers. Master's thesis, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, August 1998.

[76] C. Sinz, W. Blochinger, and W. Küchlin. PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications. In H. Kautz and B. Selman, editors, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*, Boston, MA, June 2001. Elsevier Science Publishers.

[77] C. Sinz, A. Kaiser, and W. Küchlin. Detection of inconsistencies in complex product model data using extended propositional SAT-checking. In I. Russell and J. Kolen, editors, *Proc. of the 14th International FLAIRS Conference*, pages 645–649, Key West, FL, May 2001. AAAI Press.

[78] S. K. Skedzielewski. Sisal. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 105–168, New York, 1991. ACM Press Frontier Series.

[79] D. B. Skillicorn and D. Talia. Models and languages for parallel computing. *ACM Computing Surveys*, 30:123–169, 1998.

[80] R. Srinivasan. XDR: External data representation standard.

[81] W. R. Stevens. *TCP/IP Illustrated: The Protocols*. Addison-Wesley, 1994.

[82] W. R. Stevens. *Unix Network Programming*. Prentice-Hall, 2nd edition, 1998.

[83] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[84] SUN Microsystems, Palo Alto, CA. *JavaSpaces Service Specification*, October 2000.

[85] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2:315–339, 1990.

[86] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 2nd edition, 2001.

[87] Teragrid project. *http://www.teragrid.org*.

[88] P. Thiemann. *Grundlagen der funktionalen Programmierung*. Teubner Verlag, Stuttgart, 1994.

[89] Top500 Supercomputers. *http://www.top500.org*.

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

[91] H. Zhang, M. P. Bonacina, and J. Hsiang. PSATO: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21:543–560, 1996.

[92] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley, 1991.

Lebenslauf

Persönliche Daten

Name	Wolfgang Blochinger
geboren am	10. August 1970
in	Reutlingen
Staatsangehörigkeit	deutsch
Konfession	evangelisch
Familienstand	ledig

Schulausbildung

Sep. 1977 – Juni 1981	Grundschule Reutlingen-Ohmenhausen
Aug. 1981 – Juni 1990	Isolde-Kurz-Gymnasium Reutlingen

Wehrdienst

Juli 1990 – Juni 1992	Res. Offizierausbildung Panzertruppe in Münsingen, Stetten a.k.M. und Munster (Örtze)

Hochschulausbildung

Okt. 1992 – Feb. 1998	Studium der Informatik mit Nebenfach Physik an der Universität Tübingen

Berufliche Tätigkeit

ab März 1998	Wissenschaftlicher Mitarbeiter im Arbeitsbereich Symbolisches Rechnen am Wilhelm-Schickard-Institut für Informatik der Universität Tübingen