Ein Modell zur effizienten Parallelisierung von Algorithmen auf komplexen, dynamischen Datenstrukturen

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

Vorgelegt von
Dipl.-Inf. Klaus Birken
aus Sulzbach-Rosenberg

Hauptberichter:	Prof. Dr. Roland Rühle
Mitberichter:	Prof. Dr. Siegfried Wagner
Tag der mündlichen Prüfung:	30. Oktober 1998

Rechenzentrum der Universität Stuttgart

1998

Zusammenfassung

Moderne berechnungsintensive Algorithmen, beispielsweise adaptive numerische Lösungsverfahren für partielle Differentialgleichungen, arbeiten oftmals auf komplexen, dynamischen Datenstrukturen. Die Implementierung solcher Algorithmen auf Parallelrechnern mit verteiltem Speicher mittels Datenpartitionierung wirft zahlreiche Probleme auf (z.B. Lastverteilung). Im Rahmen der vorliegenden Arbeit wurde das neue parallele Programmiermodell *Dynamic Distributed Data* (DDD) entwickelt, durch das die Parallelisierungsarbeit vom Design der verteilten Datenstrukturen bis hin zur Erstellung des portablen, parallelen und effizienten Programmcodes (sowohl für bereits existierende Programme als auch für Neuentwicklungen) unterstützt wird. Als Motivation diente dabei die Einsicht, daß bestehende parallele Programmiermodelle (auf *message-passing*-Basis) nur ungenügende Abstraktionen zur Verfügung stellen, welche die Realisierung solcher Verfahren schwer bis unmöglich machen.

Dem DDD-Konzept liegt ein graphbasiertes formales Modell zugrunde. Dabei wird die Datenstruktur des jeweiligen Programms (z.B. unstrukturierte Gitter) formal auf einen verteilten Graphen abgebildet, der aus mehreren lokalen Graphen besteht. Das formale Modell dient als Spezifikation des Programmiermodells und gleichzeitig zur Definition der wichtigen in dieser Arbeit verwendeten Begriffe. Aus einer Vielzahl von bestehenden Datenkonsistenzmodellen wird für DDD das Prinzip der *lean consistency* abgeleitet, das dem Anwendungsprogramm erlaubt, ein beliebiges Konsistenzmodell je nach den Bedürfnissen des auszuführenden parallelen Algorithmus frei zu definieren.

Der Systemarchitektur von DDD-basierten Anwendungen liegt ein Schichtenmodell zugrunde, den Kern stellt dabei die DDD-Programmbibliothek dar. Diese bietet Funktionen zur dynamischen Definition verteilter Datentypen und zur Verwaltung lokaler Objekte. In den Überlappungsbereichen der lokalen Graphen stehen abstrakte Kommunikationsfunktionen in Form von sog. Interfaces zur Verfügung. Die wesentliche Neuerung gegenüber nahezu allen bestehenden Arbeiten ist jedoch die Möglichkeit zur dynamischen Veränderung des verteilten Graphen; dies ermöglicht es beispielsweise, dynamische Lastverteilung, Gitterverfeinerung/-vergröberung oder Gittergenerierungsverfahren einfach und effizient zu implementieren. Damit können beliebig komplexe Datentopologien dynamisch erzeugt, migriert und wieder entfernt werden.

In vorliegender Arbeit werden außerdem die Implementierung der DDD-Bibliothek und eine umfangreiche Untersuchung ihrer Leistungsmerkmale beschrieben. Erst das Zusammenspiel aus speziellen Datenstrukturen und (neuartigen) Algorithmen in der DDD-Implementierung erlaubt, ein portables, wartbares und gleichzeitig effizientes Softwareprodukt bereitzustellen. Aus dem weiten Bereich von Applikationen, den das DDD-Modell abdeckt, werden schließlich zwei numerische Anwendungsprojekte detailliert beschrieben.

Danksagung

Meinem Betreuer Herrn Prof. Dr. Rühle danke ich herzlich für die Bereitstellung einer außerordentlichen Arbeitsumgebung an Menschen, Maschinen und Motivation; die erstklassige Ausstattung eines international bekannten Rechenzentrums kam vorliegender Arbeit sehr zugute. Die darüberhinaus quasi unbegrenzt zur Verfügung gestellten Reisemittel und Industriekontakte haben mir einen umfassenden Erfahrungsaustausch und Wissensabgleich mit Interessenten und anderen Forschenden ermöglicht, auch dafür vielen Dank.

Die Grundlage zur Thematik dieser Arbeit danke ich Herrn Dr. Peter Bastian. Das im Rahmen seiner Dissertation entwickelte Programm *ugp* war der bestmögliche Ausgangspunkt für meine Arbeit. Weit über den bloßen Anfangsimpuls hinausgehend begleiteten mich seine wertvollen Vorarbeiten, Hinweise und Erfahrungen durch das gesamte Projekt, bis hin zum dezenten Zeitplanungshinweis auf der Korrekturfahne.

Ebenso danke ich Herrn Clemens Helf, der mir in Form seines Programms CEQ das Problem zu meiner Lösung bereitstellte. Als Schöpfer der ersten konkreten Anwendung zum vorliegenden abstrakten Modell hat er dieses objekt-orientiert und somit entscheidend mitgeprägt. Herrn Uwe Küster danke ich vielmals für die inhaltliche und menschliche Unterstützung in den Jahren des Projekts. In zahlreichen, stets erbaulichen Gesprächen hat er meine Arbeit in einen größeren Kontext eingebettet. Die dreijährige Zusammenarbeit bei RUS-NfH war stets interessant, lehrreich und – nicht zuletzt – vergnüglich, beiden herzlichen Dank!

Herrn Stefan Lang als Hauptanwender der Software verdanke ich nicht nur wertvolle Hinweise zu möglichen Weiter- und Warnungen zu möglichen Fehlentwicklungen, sondern auch förderliche Kommentare zum Text der Arbeit in der Endphase: mehr als einmal hat er die Geduld bewiesen, ein dickes und noch unfertiges Stück Prosa gründlich zu kommentieren.

Aufgrund der glücklichen Lage, während der Erstellung dieser Arbeit an zwei Instituten arbeiten zu können, konnte ich stets eine Vielzahl von Einflüssen aus *zwei* Kreisen von Kollegen genießen. Daher möchte ich nun noch Herrn Prof. Dr. Wittum und den Leuten seines *Instituts für Computeranwendungen III* danken, mit denen ich täglich erlebe, welche Durchschlagskraft ein interdisziplinäres Arbeitsumfeld entwickeln kann. Ebenso wichtig ist mir das Dankeschön an die Kollegen vom RUS, die mir vor und nach meinem Wechsel stets zur Seite standen; die „Baracke“ wird mir in lebhafter Erinnerung bleiben.

Meiner Frau Julia Birken danke ich schließlich für die Geduld, mit der sie mich auf der zurückliegenden vierjährigen Schwangerschaft begleitet hat. Nur ihrer Unterstützung in geistigen und lebenspraktischen Dingen ist es zu verdanken, daß ich nach langen, auch für sie leidvollen Wehen nun doch ein Baby zur Welt bringen konnte.

Stuttgart, im Juni 1997 Klaus Birken

FÜR JULIA

Inhaltsverzeichnis

Kapitel 1

Motivation und Einführung

1.1 Ausgangspunkt

Datenstrukturen kompliziertester Topologie bilden die Grundlage, auf der eine Vielzahl von teils neuartigen, teils wohlbekannten Algorithmen aus allen Bereichen von Technik und Naturwissenschaften berechnungsintensive Prozeduren durchführen. Dabei übersteigt der Aufwand an Rechenzeit- und Speicherplatzressourcen oftmals die Leistungsfähigkeit bestehender Monoprozessor-Architekturen; dies macht den Einsatz von skalierbaren, massiv parallelen Systemen unumgänglich (siehe dazu sog. *grand challenges*, [74, S. 118f]). Zwischen den Bedürfnissen der Anwender und den Angeboten der Rechnerarchitekten klafft jedoch eine unübersehbare Lücke: der Entwurf komplexer, verteilter Datenstrukturen samt zugehöriger Algorithmen und deren Implementierung ist derzeit nur mit unangemessen hohem Aufwand an Personal und Entwicklungszeit möglich. Ähnlich wie bei der Programmierung mittels Assemblercode bieten die zur Verfügung stehenden Programmiermodelle zwar die nötige Funktionalität, sind aber zu maschinennah.

Sobald jedoch die Portierungsdauer eines bislang nur sequentiell eingesetzten Programms auf eine neue Parallelrechnerarchitektur die Zeit zwischen zwei Rechnergenerationen übersteigt, steht der Einsatz dieser Technologie zumindest in industriellem Kontext ernsthaft in Frage. Weiterentwicklungen der anwendungsbezogenen, algorithmischen Aspekte eines bestehenden Codes einerseits sowie aller die parallele Implementierung betreffenden Aspekte andererseits sollten vom Standpunkt des *Software Engineering* aus weitgehend unabhängig voneinander sein. Beim heutigen Stand der Softwaretechnik sind diese beiden Richtungen jedoch oftmals eng verquickt; jede Fortentwicklung des intendierten Verfahrens erfordert Integration mit der parallelen Implementierung und umgekehrt. Daher benötigt man quadratischen Entwicklungsaufwand, um die notwendigen Qualitätsfaktoren einzuhalten (dies sind Korrektheit, Lesbarkeit, Erweiterbarkeit, Wiederverwendbarkeit, Portabilität, aber gerade im Kontext dieser Arbeit auch Effizienz [136]).

Als Ansatzpunkt zur Lösung obiger Problematik bietet die Informatik verschiedene Vorgehensweisen an, die sich auf einem Kontinuum einordnen lassen:

- *automatische Parallelisierung* des sequentiellen Programmcodes durch einen entsprechend mächtigen Compiler (bisher bestenfalls Ansätze, Teillösungen und unterstützende Werkzeuge [74, Kap. 10 und 11])
- spezielle, für die Formulierung paralleler Abarbeitung geeignete Programmiermodelle bzw. -sprachen (vgl. Abschnitt 1.6)
- herkömmliche, explizite Formulierung der Parallelität durch den Anwendungsentwickler (unter Verwendung von *message-passing* [91, S. 529ff])

Die Parallelisierung komplizierter Anwendungen auf reiner *message-passing*-Basis führt zu schwerwiegenden Problemen (vgl. Abschnitt 1.4). Andererseits ist jeder weitgehend automatisierte Parallelisierungsansatz sofort auf explizite Angaben des Anwendungsentwicklers angewiesen (z.B. über Datenlokalität im parallelen Algorithmus), sobald die Anforderungen zum einen an die Komplexität und Generalität der zugrundeliegenden (sequentiellen) Algorithmen, zum anderen an Effizienz und Portierbarkeit der resultierenden parallelen Programme nur genügend hoch sind.

Eine praktikable Lösung läßt sich durch einen Kompromiß erreichen: Schränkt man die Klasse der zu parallelisierenden Algorithmen geeignet ein und erlaubt die Angabe expliziter, anwendungsabhängiger Informationen durch den Entwickler, so läßt sich eine Vorgehensweise zur Parallelisierung angeben, die einen weiten Bereich der in der Praxis vorkommenden Anwendungsfälle abdeckt und gleichzeitig die angesprochenen Kriterien des Software Engineering erfüllt. Die vorliegende Arbeit schlägt dazu ein Konzept sowie dessen Realisierung vor, das auf eine Vielzahl von (gerade industriell) relevanten Fällen anwendbar ist und den Entwicklungsaufwand von parallelen *real-world*-Anwendungen drastisch verringert. Der Schwerpunkt kann dabei nicht auf der Bereitstellung eines vollautomatischen, allgemeingültigen Parallelisierungswerkzeugs liegen; stattdessen konzentriert sich das vorgestellte Konzept auf einfache, portable und effiziente Einsetzbarkeit im Sinne eines offenen Systems [75], das auch als Basis für künftige, semi-automatische Parallelisierungsmethoden dienen kann (vgl. Abschnitt 6.3.3). Zusätzliche Vorteile des Konzepts sind leichte Wartbarkeit und erhebliche Vereinfachung der Weiterentwicklung des parallelen Programmcodes. In solcher Hinsicht leistet diese Arbeit einen Beitrag zur Überwindung der oben beschriebenen Lücke zwischen ressourcenintensiven Anwendungen und leistungsfähigen Rechnerarchitekturen.

1.2 Architekturen von Höchstleistungsrechnern

Derzeitige Rechnerarchitekturen erzielen hohe und höchste Leistungen durch folgende Konzepte der parallelen Ausführung der Operationen: *Vektorisierung*, *shared-memory*-Parallelität und *distributed-memory*-Parallelität.

Vektorisierung, bereits seit Ende der 70er Jahre eingesetzt (z.B. Cray 1S), nutzt Parallelität feinster Granularität. Komplexe Operationen werden in einfachere Operationen aufgespalten, die in Form von Vektor-*Pipeline*s angeordnet werden. Damit können mehrere komplexe Operationen quasiparallel ausgeführt werden. Diese Idee liegt heute bereits allen gängigen Mikroprozessoren zugrunde, entweder als datenparallele Variante oder in Form von Superskalar-Architekturen. Der Vorteil

des Konzepts ist die direkte Implementierbarkeit in Hardware mit teilweiser Unterstützung durch vektorisierende Compiler, so daß auf der Ebene einer höheren Programmiersprache keine Anpassungsarbeit nötig ist. Dieser Ansatz ermöglicht sehr effiziente Programmcodes, ist jedoch ungeeignet für Verfahren, die komplexe indirekte Adressierungsschemata benutzen (z.B. adaptive numerische Verfahren mit Verfeinerung/Vergröberung des Rechengitters). Beispiele aktueller Maschinen, die auf Vektorisierungstechniken beruhen, sind Cray Triton, Cray J90, NEC SX4, Fujitsu VPP600 oder Hitachi SR2201.

Shared-Memory-Parallelrechner bieten als Programmiermodell einen einheitlichen, globalen Adreßraum für alle Prozessoren der Maschine. Zwischen den Prozessoren und dem physikalischen Speicher liegt ein Kommunikationsnetzwerk, über das Adressen und Daten zwischen Prozessor und Speicherchip versendet werden. Oft sind einige Speicherbereiche direkt einzelnen Prozessoren zugeordnet; auf diese kann daher schneller zugegriffen werden als auf solche, die anderen Prozessoren zugeordnet sind. Da sich die zu leistende Arbeit in den Schleifen eines Programms verbirgt, kann die Parallelisierung durch Aufteilung der Schleifen (gegebenenfalls nach entsprechenden Transformationen des Programmcodes, z.B. *loop splitting*) zumindest halbautomatisch geschehen (d.h. durch den Compiler, evtl. mit Annotationen durch den Programmentwickler). In Hwang [74, S. 567ff] sind dafür notwendige Techniken der Abhängigkeitsanalyse (*dependency analysis*) dargestellt. Probleme dieses Konzepts sind derzeit die mangelnde Skalierbarkeit (die Obergrenze liegt bei etwa 2^7 bis 2^8 Prozessoren), hohe Kosten (z.B. für das Kommunikationsnetzwerk) und unterschiedliche Speicher-Zugriffszeiten trotz konzeptionell globalem Adreßraum. Letztere Eigenschaft bewirkt, daß trotz (halb-)automatischer Parallelisierungsmöglichkeit auf Anwendungsebene die Datenlokalität berücksichtigt werden muß. Beispiele aktueller Maschinen des *shared-memory*-Typs sind Cray Triton, Cray J90, NEC SX4, BBN GP1000, SGI/Cray Origin oder auch Tera MTA.

Die beiden letztgenannten Architekturen erscheinen vom Standpunkt dieser Arbeit besonders bemerkenswert, da sie ähnliche Ziele verfolgen, allerdings durch Entwicklung von Hardware. Bei der *SGI/Cray Origin*-Architektur (eine Weiterentwicklung des Stanford DASH-Prototyps [100]) soll eine verteilte Speicherhierarchie ein skalierbares, effizientes *Distributed-Shared-Memory*-Konzept in Hardware umsetzen. Jeder Rechenknoten besteht aus zwei R10000-Prozessoren, dem lokalen Speicher und einem Controller (sog. *hub*), der für die verteilte Verwaltung des gemeinsamen Speichers zuständig ist. Die Daten der jeweils einem Prozessor zugeordneten Cache-Speicher werden über ein spezielles, *directory*-basiertes Protokoll konsistent gehalten (*cache-coherence protocol*). Sobald ein Prozessor eine Speicherseite häufig benutzt, wird diese in dessen lokalen Speicher migriert. Auf diese Art wird Datenlokalität automatisch erkannt und ausgenutzt [111]. Die Entwickler der *Tera MTA*-Architektur versuchen, Parallelität durch hardwareunterstützte leichtgewichtige Prozesse (*threads*) zu ermöglichen. Um Prozeßwechsel schnell (d.h. in einem Taktzyklus) durchführen zu können, enthält jeder Prozessor alle *thread*-bezogenen Daten vielfach (z.B. Register). Als zusätzliche Besonderheiten dieser Architektur sind das dreidimensionale, ausgedünnte Kommunikationsnetzwerk mit Prozessoren *oder* Speicherbausteinen als Knoten und zusätzliche Kontrollbits je Speicherwort zu nennen. Die zweite Eigenschaft (sog. *tagged memory*) ermöglicht z.B. transparente indirekte Zugriffe ohne Beteilung eines Prozessors und Synchronisation von Prozessen mit der Granularität eines Speicherworts [2]. Einige Software-Varianten mit ähnlichen Techniken wie diese Architekturen werden in Abschnitt 1.6 kurz diskutiert.

Bei *Distributed-Memory*-Parallelrechnern ist jedem Prozessor ein lokaler Speicher zugeordnet, auf den von den anderen Prozessoren des Rechners nicht zugegriffen werden kann. Die Kommunikation zwischen den Prozessoren findet hauptsächlich über den Austausch von Nachrichten (d.h. *message-passing*) statt. Demzufolge stehen an der Programmierschnittstelle nicht ein globaler Adreßraum, sondern viele lokale Adreßräume (einer für jeden Prozessor) zur Verfügung. Die Interprozessor-Kommunikation wird durch geeignete Kommunikationsnetzwerke unterstützt (z.B. Crossbars oder 2D/3D-Felder). Vorteile des *distributed-memory*-Konzepts sind dessen Skalierbarkeit und die Kosten der Hardware; der hauptsächliche Nachteil ist die Komplexität der parallelen Programme. Diese rührt vom Fehlen eines einheitlichen Adreßraums und der Notwendigkeit zur expliziten Kommunikation her; eine globale Sicht ist also nicht gegeben, aber für die meisten Applikationen notwendig. Aktuelle Parallelrechner dieses Typs sind beispielsweise Intel Paragon, Cray T3E, IBM SP2, Hitachi ST2201 oder Parsytec CC. Konzeptionell fallen Cluster von handelsüblichen Workstations oder PCs (z.B. unter Windows NT) ebenfalls in diese Kategorie.

In vorliegender Arbeit soll der Schwerpunkt auf massiver Parallelität und skalierbaren Systemen liegen, deshalb werden *distributed-memory*-Architekturen im Brennpunkt des Interesses stehen. Das im Verlauf der Arbeit entwickelte Konzept soll eine komfortable Parallelisierungsmöglichkeit bieten, ohne dafür einen einheitlichen Adreßraum vorauszusetzen.

1.3 Berechnungsintensive, graphbasierte und dynamische Algorithmen

In diesem Abschnitt wird eine Klasse von Algorithmen spezifiziert, für die die Anwendung des hier beschriebenen Konzepts sinnvoll erscheint; Abb. 1.1 gibt dazu einen Überblick. Zunächst sollen alle berechnungsintensiven Anwendungen in Betracht gezogen werden, deren Rechenaufwand so von der zugrundeliegenden Datenbasis abhängt, daß auf parallelen Rechnersystemen mit physikalischer *distributed-memory*-Architektur die Verfolgung einer *Datenpartitionierungs-Strategie* naheliegt, wobei alle beteiligten Prozessoren dasselbe Programm auf (größtenteils) disjunkten Teilmengen der gesamten Datenbasis ausführen. Dieses sogenannte SPMD-Modell (*single program over multiple data streams*) kann als Spezialfall des MIMD-Maschinenmodells (*multiple instructions over multiple data streams*) aus dem Klassifikationsschema nach Flynn [53] betrachtet werden, wobei es Parallelität mittlerer Granularität auf Prozedur- bzw. Unterprogrammebene mit moderatem Kommunikationsaufwand verbindet [74, S. 61ff]. Als prominente Vertreter dieser Anwendungsklasse sollen hier hauptsächlich numerische Lösungsverfahren für (partielle) Differentialgleichungen bzw. -gleichungssysteme (kurz: PDG) behandelt werden, wie sie z.B. in Form von Euler- und Navier-Stokes-Gleichungslösern im Bereich der *Computational Fluid Dynamics* (kurz: CFD) Verwendung finden.

Eine grundsätzliche Entwurfsentscheidung läßt sich bereits von der sequentiellen Implementierung solcher Anwendungen ablesen: Im einfacheren Fall basieren Algorithmen auf (ggf. mehrdimensionalen) *Feldern*, auf die mittels linearer Indexrechnung (direkt) zugegriffen werden kann; im zweiten, weitaus komplizierteren Fall verwenden Algorithmen graphbasierte Strukturen, die nur durch rekursive Zugriffe erschlossen werden können [91, Kap. 11]. Für die Klasse der Gleichungslöser

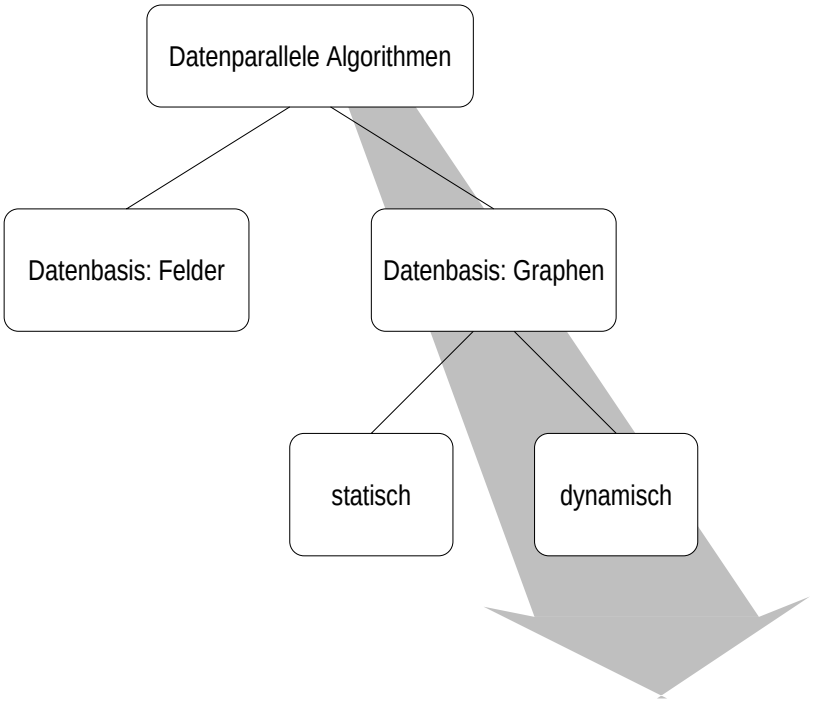


Abbildung 1.1: *Eigenschaften von Algorithmen, bei denen das vorgestellte Parallelisierungskonzept anwendbar ist.*

ergibt sich diese Entscheidung aus der Wahl der zur Diskretisierung des Problemraums verwendeten Gitter: regelmäßige, strukturierte Gitter werden durch Felder implementiert, unstrukturierte Gitter durch spezialisierte Graphen (z.B. UG [10, 13], CEQ [67], KASKADE [43, 49, 97] oder PLTMG [8]). Natürlich sind auch Hybridformen denkbar: Verfahren auf blockstrukturierten Gittern verwenden Kombinationen von strukturierten Gittern (als Felder) und deren Nachbarschaftsbeziehungen (als Graphen) [98, 122].

Die Umsetzung *graphbasierter Strukturen* im Rahmen von bestimmten Implementierungen geschieht abstrakt mittels eines *Referenz*-Konzeptes; Objekte (Knoten des Graphen), welche die notwendigen Daten enthalten, verweisen dabei über *Referenzen* (gerichtete Kanten des Graphen) aufeinander, was eine potentiell beliebige Datentopologie erlaubt. Im Hinblick auf gebräuchliche Programmiersprachen wird dies z.B. durch hardwarenahe *pointer* (in der Programmiersprache ANSI C), Indexberechnungen (bei Verwendung von Fortran) oder durch vom Benutzer definierte Referenz-Klassen (in objektorientierten Sprachen) verwirklicht.

Was bedeutet diese Entwurfsentscheidung nun im wesentlichen für die parallele Implementierung? Im einfacheren Fall werden die (mehrdimensionalen) Felder der Datenbasis entlang einer oder mehrerer Dimensionen (ggf. zyklisch) aufgespalten und die entstehenden Teilfelder den einzelnen Prozessoren[1] des Parallelrechners zugeordnet (z.B. implizit durch den Compiler bei Verwendung

[1] Selbstverständlich kann in einem allgemeineren (feingranularen) Kontext sowohl *Prozeß* als auch *Prozessor* gemeint sein; im Rest der Arbeit wird jedoch als Sprechweise *Prozessor* beibehalten.

der *array notation* in Fortran 90 [46, 105, S. 244ff] oder auch durch DISTRIBUTE-Direktiven und FORALL-Konstrukt in HPF [54]). Da eine so geartete Aufteilung als einfache arithmetische Beziehung von Prozessortopologie und Feldindizes dargestellt werden kann, liegt die komplette Struktur der gesamten Datenbasis jedem Prozessor als globale Information vor. Dies vereinfacht alle in Abschnitt 1.4 beschriebenen Probleme und wurde bereits in vielen vorausgehenden Arbeiten behandelt. Deshalb soll dieser Fall nicht (bzw. höchstens indirekt als Spezialisierung des allgemeinen, graphbasierten Falls) Gegenstand dieser Arbeit sein.

Basiert der zu parallelisierende Algorithmus dagegen auf graphartigen Strukturen, muß der gegebene Graph auf geeignete Weise in Subgraphen aufgespalten werden; die entstehenden Teilgraphen werden wieder den einzelnen Prozessoren des Parallelrechners zugeordnet. Bei dieser Art der Aufteilung existiert jedoch im allgemeinen keine arithmetische Abbildungsfunktion, die aus der Sicht eines einzelnen Prozessors die globale Datenbasis beschreiben würde: die einzige vollständige Information über die Aufteilung des globalen Graphen ist eben dieser Graph selbst. Die daraus resultierenden Probleme werden im nächsten Abschnitt detaillierter behandelt.

Das hier behandelte Parallelisierungskonzept soll gerade für allgemeine, graphbasierte Algorithmen anwendbar sein, indem es ein Datenmodell zur Verfügung stellt, das die Aufteilung in Subgraphen und deren Prozessorzuordnung unterstützt. Im Beispiel der PDG-Lösungsverfahren bedeutet dies: Das unstrukturierte Gitter (Graph) als Diskretisierung des Problemgebiets wird in Teilgitter (Subgraphen) aufgespalten, die den Prozessoren geeignet zugeordnet werden.

Betrachtet man nun die Klasse der graphbasierten Anwendungen, so lassen sich wiederum zwei grundsätzliche Fälle unterscheiden:

1. Der Graph wird in einer Initialisierungsphase (zu Laufzeitbeginn) erzeugt und nachträglich nicht mehr verändert.
2. Der Graph und damit die Topologie der Datenbasis verändern sich dynamisch zur Laufzeit.

Das anzuwendende Parallelisierungskonzept muß zur Unterstützung des zweiten, weitaus komplizierteren Falles Möglichkeiten vorsehen, die Topologie der bereits verteilten Datenbasis (d.h. die Struktur und Verteilung der Subgraphen) konsistent zu manipulieren. Diese dynamische Komponente erfordert die vollständige Unterstützung des jeweiligen Referenzkonzepts durch das parallele Programmiermodell.

Typische PDG-Lösungsverfahren, die dynamische Veränderungen des Gitters zur Laufzeit erfordern, sind die *adaptiven* Verfahren. Dabei wird während des iterativen Lösungsprozesses das Diskretisierungsgitter durch Verfeinerung bzw. Vergröberung dergestalt verändert, daß es Besonderheiten der Lösungsfunktion widerspiegeln kann (z.B. [8, 10, 43, 67]). Dies ermöglicht Einsparungen in der Rechenzeit bei gleichzeitiger Erhöhung der Genauigkeit des Verfahrens. Als Repräsentant adaptiver numerischer Verfahren auf unstruktierten Gittern und gleichzeitig als Hauptbeispiel der Anwendung des vorgestellten Parallelisierungskonzepts dient das Finite-Volumen-Verfahren CEQ (siehe Abschnitt 5.2).

Neben adaptiven numerischen Verfahren basieren viele andere Algorithmen aus verschiedensten Bereichen auf dynamischer Konstruktion bzw. Veränderung der jeweiligen Datenbasis. Als kleiner

Ausschnitt seien hier lediglich einige Beispiele willkürlich herausgegriffen: Generierung von Gittern für Probleme aus dem Ingenieurbereich [140], Konstruktion von adaptiven Energiehyperflächen aus der Computer-Chemie [28], Simulation von Vorgängen aus der Astrophysik [117].

Zum Abschluß dieses Abschnitts sollen noch einmal die Eigenschaften der Anwendungsklasse zusammengefaßt werden, die sich für die Parallelisierung durch das im folgenden beschriebene Konzept anbietet:

Notwendig:	der Algorithmus legt eine SPMD-Strategie nahe	(Datenpartitionierung)
Vorteilhaft:	die Datenbasis hat die Struktur eines Graphen	(Referenzkonzept)
Optional:	die Datenbasis unterliegt topologischen Änderungen	(Dynamik)

Dabei ist es gerade die letzte, optionale Eigenschaft, welche die Neuartigkeit des Konzepts ausmacht: die Möglichkeit zu dynamischen Veränderungen der Datentopologie zur Laufzeit unter Aufrechterhaltung von Effizienz und Portierbarkeit als integraler Bestandteil des Parallelisierungskonzepts.

1.4 Probleme der Parallelisierung

Die grobe Vorgehensweise zur Parallelisierung von Anwendungen mit allen drei zuvor erläuterten Eigenschaften ist sofort klar: Der gesamte (globale) Graph muß zerlegt und auf die lokalen Speicher[2] der beteiligten Prozessoren verteilt werden. Bei höherem Detaillierungsgrad treten jedoch schwerwiegende Probleme auf, deren Spektrum von grundsätzlichen, algorithmischen Anforderungen (z.B. Notwendigkeit zur Lastbalancierung) bis hin zu technischen Implementierungsfragen (z.B. Verwaltung von redundanten Daten und deren Konsistenz) reicht. Im einzelnen sind dies [18, 22, 126]:

Lastverteilung: Die Verteilung des Graphen auf die lokalen Speicher und damit die Zuordnung von Arbeitsaufwand zu den vorhandenen Prozessoren muß ausgewogen erfolgen (*Lastbalancierung*), um die effiziente Ausführung des parallelen Programms zu gewährleisten. Sobald die Topologie des verteilten Graphen zur Laufzeit maßgeblich verändert wird, ist die Last pro Prozessor im allgemeinen nicht mehr gleichverteilt, was zu einer Neuverteilung der Datenbasis führen muß. Aus Skalierungsgründen muß das dynamische Lastbalancierungsverfahren dabei selbst als paralleler Algorithmus ablaufen; die jeweils vorhergehende Verteilung muß in Betracht gezogen werden, um den Umverteilungsaufwand gering zu halten [10, 70]. Da die Struktur des Graphen und damit die beste Lastverteilung stark von der jeweiligen Anwendung abhängt, kann das Problem der Lastverteilung vom Betriebssystem oder von der Rechnerhardware nur sehr eingeschränkt und ineffizient gelöst werden. Es bleibt somit ein Problem der Anwendungsebene (oder geeigneter Programmbibliotheken).

[2] Selbst bei skalierbaren Parallelrechnern mit einem gemeinsamen Adreßraum sind von jedem einzelnen Prozessor aus verschiedene physikalische Speicherbereiche auch unterschiedlich effizient zugreifbar (siehe Abschnitt 1.2). Es gibt also in bezug auf Speicherzugriffe durchaus einen Lokalitätsbegriff, auch wenn dieser an der Programmierschnittstelle nicht immer sichtbar ist (vgl. NUMA-Architekturen [74, S. 22f]).

Globale Referenzen: Für die Implementierung der Datenbasis wird ein Referenzkonzept benötigt (S. 15). Die im sequentiellen Fall verwendeten Referenzen basieren jedoch auf einem einheitlichen Adreßraum und werden beim Vorhandensein von mehreren lokalen Adreßräumen sofort unbrauchbar.

Um diesen Konflikt aufzulösen, müssen entweder globale Referenzen implementiert oder Wege zur Vermeidung der Notwendigkeit von globalen Referenzen gefunden werden. Die erste Lösung impliziert einen konzeptuellen Unterschied der sequentiellen bzw. parallelen Implementierung und kann deshalb höchstens auf Compiler-Ebene (wenn nicht bereits in der Rechnerhardware) sinnvoll eingesetzt werden. Zwar können *shared-memory*-Architekturen wie z.B. SGI Origin oder Tera MTA Hardwareunterstützung in Form eines globalen Adreßraums bieten, unzureichende Performance von entfernten Speicherzugriffen und hohe Hardwarekosten sind jedoch derzeit noch Nachteile dieser Technologie (vgl. Abschnitt 1.2).

Mit der zweiten Lösung ergeben sich Folgeprobleme: Beispielsweise werden während der Datenumverteilung aufgrund der dynamischen Lastbalancierung komplexe Strukturen, die mittels des Referenzkonzepts konstruiert wurden, aus dem lokalen Speicher eines Prozessors in den eines anderen übertragen. Alle Referenzen, die dieser Prozedur unterworfen sind, werden beim Transfer über die Grenze zweier lokaler Adreßräume ungültig und müssen während der Übertragung umgerechnet werden.

Datenredundanz: Durch die Aufteilung des globalen Graphen entstehen an den Grenzen zwischen benachbarten Subgraphen (sog. *inter-processor*-Grenzen) inkonsistente Situationen, in welchen bestimmte Annahmen über Nachbarschaften von Objekten bzw. Knoten des Graphen ungültig werden. Bei Zugriffen über die Nachbarschaftsrelation kann das benötigte Datum im Speicher eines fremden Prozessors liegen.

Um diese Problemfälle auszuschließen, ohne die Übersichtlichkeit des Programms und die Gesamteffizienz (durch oftmalige Zugriffe auf nicht-lokale Speicherbereiche) zu beeinträchtigen, müssen an den *inter-processor*-Grenzen redundante Kopien von einzelnen Objekten erzeugt werden. Diese *Lokalisierung* der Nachbarschaftsrelation erzwingt jedoch ein Protokoll, welches die Daten der redundanten Kopien konsistent hält. Die Eigenschaften solcher Protokolle haben wesentlichen Einfluß auf die Effizienz des parallelen Programms und beinhalten somit kritische Designentscheidungen (vgl. Abschnitt 2.3.8). In einem entsprechenden *shared-memory*-System würde diese Redundanz und die notwendigen Konsistenzprotokolle in der verteilten Speicherhierarchie repräsentiert werden, z.B. in verteilten Caches der SGI Origin-Architektur (Abschnitt 1.2).

Aus Sicht des parallelen Programms muß zur Einhaltung des Konsistenzprotokolls ein Informationsaustausch zwischen den Prozessoren auf der bestehenden, statischen Datentopologie stattfinden. Die dabei entstehenden Kommunikationsschemata auf komplexen Prozessor- und Datentopologien können bei unzureichender Organisation zu Effizienzverlusten führen, die von Latenzzeiten in der Kommunikation oder übermäßiger Zwischenspeicherung von Daten herrühren.

Aufgrund dieser Probleme scheint eine *layer*-basierte Strategie zur Implementierung von Programmen mit dynamischen Datenstrukturen auf *distributed-memory*-Rechnern unvermeidbar. Der An-

wendungsteil eines parallelen Programmes sollte dem sequentiellen Code möglichst entsprechen oder sogar identisch sein; nur dadurch kann eine einfache Wartung und Weiterentwicklung des Programmcodes garantiert werden. Dazu muß eine Abstraktionsebene spezifiziert werden, die es erlaubt, die Funktionalität zur Bewältigung obiger Probleme in eine tiefere Schicht des Programmiermodells zu verlagern.

1.5 Entwurfsziele und Lösungsansatz

Aus den bisher diskutierten Aspekten läßt sich eine Reihe von *Entwurfszielen* für die Entwicklung des Parallelisierungskonzepts ableiten:

- *Vorhandene sequentielle Programme*, welche die drei Eigenschaften *Datenpartitionierbarkeit*, *Graphstruktur* und *Dynamik* (vgl. S. 17) aufweisen, müssen mit vergleichsweise geringem Aufwand auf Parallelrechner übertragen werden können. Der Großteil des bestehenden Programmcodes muß dabei unangetastet bleiben, um spätere, simultane Weiterentwicklung und Wartung von sequentiellem und parallelem Programm zu ermöglichen.
- Ebenso muß die komfortable *Neuentwicklung paralleler Programme* möglich sein, ohne jedoch auf Einzelheiten der zugrundeliegenden Rechnerarchitektur Bezug nehmen zu müssen.
- Einmal parallelisiert, sollen Anwendungsprogramme mit minimalem Aufwand auf *alle gängigen Rechnerplattformen portiert* werden können.
- Die *Laufzeit- und Speichereffizienz* der resultierenden parallelen Programme soll nicht wesentlich hinter parallelen Programmen zurückbleiben, die durch den Anwendungsentwickler von Hand auf die Eigenschaften und Bedürfnisse eines gegebenen Rechnersystems eingestellt wurden (manuelles *tuning*).
- Eine *klare Abstraktionsebene* soll die anwendungsbezogenen Teile des parallelen Programms von den auf das parallele Programiermodell bezogenen Teilen abgrenzen. Dies erfolgt aufgrund der Problemstellung am zweckmäßigsten an der Datenschnittstelle, also bei der Verteilung des dynamischen Graphen.
- Trotz der Datenabstraktion soll der Entwickler der Anwendung an klar definierten Schnittstellen explizites Wissen, z.B. über die Lokalität der Daten, einbringen können. Dies soll in einer Weise geschehen, die unabhängig vom jeweils aktuellen Stand der Parallelrechnerarchitektur ist (z.B. *distributed-* oder *shared-memory*-Architekturen).

Aufgrund obiger Anforderungen ist das hier vorgestellte Parallelisierungskonzept aus folgenden Bestandteilen zusammengesetzt: ein formales Datenmodell, die Spezifikation der Abstraktionsebene, ein zugehöriger Konsistenzbegriff und eine portable, effiziente Implementierung des Konzepts in Form einer Programmbibliothek.

Das *formale Datenmodell* beschreibt die globale Datenbasis und ihr verteiltes Gegenstück in Form von Graphen sowie den Begriff *konsistente Verteilung*. Es bildet somit den Kern des gesamten

Konzepts. Die konzeptuell wichtige Abstraktionsebene wird anschließend durch zusätzliche Beschreibung einer funktionalen Schnittstelle (auf obigem Datenmodell) spezifiziert. Dies schließt die schon angesprochene Datenredundanz an den *inter-processor*-Grenzen ein. Für die redundanten Daten wird ist ein Konsistenzprotokoll benötigt, das sich an den Bedürfnissen der Anwendung orientiert (z.B. Konsistenz der Überlappungsbereiche bei Gittern numerischer Anwendungen) und daher durch den Entwickler der Anwendung spezifizierbar sein muß.

Schließlich beinhaltet das Konzept eine *Programmbibliothek*, die eine Implementierung der funktionalen Abstraktionsebene darstellt. Als wichtigste Eigenschaften der Funktionalität dieser Bibliothek sind *Portabilität*, verwirklicht durch Abstützung auf ein einfaches, allgemein verfügbares (oder entsprechend leicht implementierbares) Programmiermodell (vom Typ *message-passing*), und *Effizienz* zu nennen. Dabei kann letztere durch Ausnützung von Einschränkungen der Ziel-Anwendungsklasse (z.B. Graphen, nur lokale Referenzen, geeignete Datenredundanz) und Berücksichtigung von Eigenschaften des zugrundeliegenden Programmiermodells (z.B. nichtblockierende Kommunikation, Minimierung von Kommunikationszeiten und Nachrichtenanzahl) erreicht werden.

Zur Parallelisierung einer bestehenden Anwendung wird deren Datenbasis zunächst in das parallele Datenmodell eingebettet, wodurch bereits die Funktionalität an der Abstraktionsebene erschlossen wird. Zur vollständigen Anbindung muß noch die gewünschte Verteilung der Daten sichergestellt werden und das Konsistenzprotokoll implementiert werden. Beide Vorgänge sind anwendungsabhängig und damit für den Anwendungsentwickler explizit, werden aber durch geeignete Konstrukte des Parallelisierungskonzepts unterstützt.

1.6 Verwandte Arbeiten

Die Problematik der Verwaltung von verteilten, dynamischen Graphstrukturen auf lose gekoppelten Parallelrechnern war und ist Gegenstand einer Vielzahl von Forschungs- und Entwicklungsarbeiten. In diesem Abschnitt sollen die wichtigsten Ansätze dargestellt und klassifiziert werden; diese lassen sich dabei nach dem jeweils zugrundeliegenden bzw. bereitgestellten Programmiermodell in folgende Klassen einteilen:

- *user-level*-Modelle
- verteilte Objekte (auf *compiler/runtime-Ebene*)
- leichtgewichtige Prozesse (*Threads*)
- datenparallele und funktionale Sprachen, algorithmische Skelette u.ä.
- *Distributed Shared Memory*

Das in dieser Arbeit beschriebene Konzept ist ein typisches *user-level*-Modell: die gesamte Funktionalität steht in Form einer Programmbibliothek zur Verfügung und benötigt keine Unterstützung durch einen speziellen Compiler oder besondere Betriebssystemfunktionen. Ähnliche Ansätze werden daher besonders ausführlich behandelt. Ansätze auf Basis von *verteilten Objekten* und *leichtgewichtigen Prozessen* können für ähnliche Anwendungen wie das hier beschriebene Konzept eingesetzt werden und werden deshalb ebenfalls genauer analysiert.

Bei der Verwendung von *datenparallelen Sprachen* (z.B. Fortran 90 [105], HPF [54], oder auch [63]) gibt der Anwender dem Compiler über parallele Sprachkonstrukte Hinweise auf Art und Ort der Parallelität des implementierten Verfahrens; der Compiler übernimmt (evtl. in Verbindung mit einem geeigneten Laufzeitsystem [95]) die eigentliche Datenverteilung und Konsistenzerhaltung. Funktionale Sprachen (z.B. Haskell [62] oder Sisal [135] mit parallelem Laufzeitsystem) erlauben es, aus einer funktionsorientierten Formulierung des Programms (frei von Seiteneffekten) einen Datenflußgraph zu erstellen, aus dem die vorhandene Parallelität abgelesen werden kann. Der Programmierer wird so von expliziter Parallelisierungsarbeit isoliert. Andere Ansätze versuchen, über die Bereitstellung von algorithmischen Skeletten (vgl. [39, 42], Anwendung z.B. in *Skil* [26, 27]) die Grundeigenschaften paralleler Programme zu kapseln, um sie einem Anwender transparent verfügbar zu machen. Alle drei Ansätzen haben gemein, daß ihr Einsatz nur bei Software-Neuentwicklungen möglich ist; für das enorme Reservoir an bestehender Software können diese Techniken nicht oder nur in kleinen, manuell bestimmten Teilen verwendet werden (z.B. Sisal *Foreign Language Interface*). Das in dieser Arbeit vorgestellte Konzept dagegen läßt die Wiederverwendung bestehender Programme (*software reuse*) problemlos zu.

Im Konzept des *Distributed Shared Memory* (DSM, siehe [32, 112]) wird die Parallelität auf die Ebene des Betriebssystems verlagert, indem ein allen Prozessoren gemeinsamer, globaler Adreßraum bereitgestellt wird. Alle Zugriffe auf den gemeinsamen Adreßraum müssen effizient auf die

tatsächlichen, physikalisch verteilten Speicher abgebildet werden. Dies kann durch einen speziellen Betriebssystemkern, Bibliotheksroutinen mit compilergenerierten Aufrufen oder Sourcecode-Transformationen geschehen. Oftmals werden DSM-Systeme in das Betriebssystemkonzept der virtuellen Speicherverwaltung integriert, wodurch die Größe der von mehreren Prozessoren sichtbaren Speicherblöcke auf die Seitengröße der Speicherverwaltung festgelegt wird. Je nach Art des Konsistenzprotokolls von verteilten, an der Programmiermodell-Schnittstelle jedoch gemeinsamen Daten kann es zu unerwünschten Effekten kommen: hohe Aufsetzzeiten durch fehlende Minimierung der Nachrichtenanzahl, zyklische Migration von Objekten (sog. *thrashing*) oder disjunkte Speicherzugriffe auf dieselbe Seite, was unnötige Konsistenzoperationen erfordert (sog. *false sharing*). Bei manchen *shared-memory*-Architekturen wird der globale Adreßraum bereits von der Hardware unterstützt bzw. vollständig implementiert und nicht erst auf Betriebssystemebene nachträglich virtuell erzeugt (z.B. SGI Origin, Abschnitt 1.2). In der hier vorgestellten Arbeit wird die Lokalität explizit formuliert; dadurch ergibt sich zwar ein komplizierteres Programmiermodell, das aber deutlich effizientere, vor allem aber skalierbare parallele Applikationen ermöglicht.

Die obige grobe Klassifikation erlaubt nicht für jedes Konzept eine eindeutige Zuordnung, oftmals läßt sich ein Ansatz mehreren Bereichen zuordnen (z.B. kann *Distributed Shared Memory* auf Basis von *message-passing* implementiert werden oder können algorithmische Skelette als Grundlage einer datenparallelen Sprache dienen). Trotzdem wurden die bestehenden Ansätze grob aufgeteilt, um einen Vergleich des hier vorgestellten Konzepts mit verwandten Arbeiten zu ermöglichen. Im Zweifelsfall werden einzelne Ansätze in mehreren Kategorien aufgeführt.

1.6.1 Modelle auf *user-level*

In den letzten Jahren wurde eine Vielzahl von Programmbibliotheken bzw. Spracherweiterungen zur Unterstützung von verteilten Gitter- bzw. Graphstrukturen entwickelt. Die folgende Tabelle gibt eine alphabetische Übersicht der bestehenden Ansätze, aufgeschlüsselt u.a. nach den Kriterien *Dynamik*, *Gitterart-* und *-dimension*, *Plattformen* und *Datentypen*. Alle Ansätze haben gemeinsam, daß sie ein mächtigeres Programmiermodell auf das bestehende, nachrichtenbasierte Modell aufsetzen. Der Programmierer der parallelen Anwendung sieht nach Möglichkeit nur noch verteilte Datenstrukturen, die vom System auf verteilte Speicher und *message-passing*-Kommunikation abgebildet werden. Für den Einsatz dieser Werkzeuge ist allerdings keine Unterstützung durch den Compiler notwendig, weshalb sie auch als *user-level*-Modelle bezeichnet werden können.

Eine ausführliche Fassung der nachfolgenden Tabelle und eine kurze Erläuterung der verwendeten Abkürzungen finden sich in Anhang E am Ende dieser Arbeit. Informationen zu einigen der enthaltenen Konzepte und eine Vorform des zugrundegelegten Klassifikationsschemas wurden einer Studie von Parashar [114] entnommen. Es wurden jedem Konzept hier wie in Anhang E die hauptsächlich an der Entwicklung beteiligte Institution sowie Literaturverweise hinzugefügt, um eine schnelle Zuordnung zu ermöglichen.

Bezeichnung		*Institution*
dynamisch/statisch	*Dimension*	*Modell*
Datentypen	*Gitterart*	
MP-Modell	*Plattformen*	
Applikationsbereich	*Zielsprache*	
A++/P++ [98, 99]	Univ. of Colorado & Los Alamos National Lab.	
dynamisch (gitterweise)	1-8 Dim	SPMD
integer, double	blockstrukturierte Gitter	
PVM, MPI	WsC	
Anwendungen auf strukturierten Gittern	C++	
BlockComm [60]	Argonne National Laboratory	
statisch	1-5 Dim	SPMD
F77-Datentypen	strukturierte Gitter	
Chameleon [61]	Cray, IBM, Intel u.a., WsC	
Anwendungen auf strukturierten Gittern	F77	
CHAOS (⊃ PARTI) [107, 119, 133]	University of Maryland	
dynamisch	beliebig	SPMD
C-/F77-Datentypen	unstrukturierte Gitter	
PVM, MPI, NX u.a.	Intel, IBM SP1, CM-5	
adaptive unstrukturierte Anwendungen	C, F77	
CHAOS++ (⊃ CHAOS) [34, 35]	University of Maryland	
dynamisch	beliebig	SPMD
C++ Klassen und Typen	allgemeine Graphen	
PVM, MPI, NX u.a.	Intel, IBM, CM-5	
OO-Anwendungen mit dynamischen Datenstrukturen	C++	
CLIC (auch: GMD Communications Library) [122]	GMD, St. Augustin	
dynamisch auf Blockebene	2-3 Dim	SPMD
F77-Datentypen	blockstrukturierte Gitter	
PARMACS 6.0 [69]	Cray, IBM, Intel u.a., WsC	
blockstrukturierte Anwendungen	F77	
Canopy [52]	Fermi National Accelerator Laboratory	
statisch	beliebig	SPMD
C-/F77-Datentypen	strukturierte Gitter	
eigene KS *CHIP*	Cray T3D, Sequent	
Anwendungen auf strukturierten Gittern	C, F77	
Concurrent Graph (oder: SCPlib) [139, 145]	California Institute of Technology	
dynamisch auf Threadebene	beliebig	taskparallel
beliebig	allgemeine Graphen	
eigene KS	Cray, IBM, Intel u.a., WsC	
CFD, Partikelmethoden, Monte Carlo	?	

Tabelle 1.1: *Übersicht existierender Konzepte zur Verwaltung verteilter dynamischer Datenbasen (dreiteilig).*

Bezeichnung		*Institution*
dynamisch/statisch	*Dimension*	*Modell*
Datentypen	*Gitterart*	
MP-Modell	*Plattformen*	
Applikationsbereich	*Zielsprache*	
DAGH [113, 115]		University of Texas at Austin, TX
dynamisch	beliebig	SPMD
Fortran Datentypen	blockstrukturierte Gitter	
MPI	Cray, IBM, WsC	
adaptive Mehrgitterverfahren (AMR)	C++ (und F77/F90)	
DIME [148, 150]		California Institute of Technology
dynamisch	2 Dim	SPMD
C Datentypen	Dreiecksgitter	
Express OS	NCube, Transputer, Intel	
FE (für Navier-Stokes)	C	
Dome [4]		Carnegie Mellon University
dynamisch	beliebig (Einbettung)	SPMD
VuM von C++-Typen	strukturierte Gitter	
PVM	heterogene WsC	
Anwendungen auf strukturierten Gittern	C++	
Global Arrays (GA) [110]		Pacific Northwest Laboratory
dyn. *array*-Verwaltung	2 Dim	MIMD
integer, double	strukturierte Gitter	
interrupt-driven, TCGMSG	Intel, IBM, KSR, WsC	
Computer-Chemie	C, F77	
GRIDS [56–58]		Universität Stuttgart
statisch	beliebig	SPMD
F77 Datentypen	unstrukturierte Gitter	
eigene KS *SPPL*	IBM, Intel Paragon	
gitterbasierte Anwendungen (z.B. CFD)	F77	
LOCO [84–86]		K. U. Leuven, Belgien
dynamisch	beliebig	SPMD
C Typen, Strukturen	beliebig	
PVM, NX	Intel, WsC	
adaptive numerische Verfahren, auch MG	C	
LPARX (vorher: GenMP [5]) [87, 88]		University of California, San Diego
dyn. *array*-Verwaltung	1-4 Dim	SPMD
C++ Typen/Klassen	blockstrukturierte Gitter	
eigene KS *MP++*	PVM, u.a.	
ML, adaptive FD, Partikelmethoden	C, C++ und F77	

Tabelle 1.1: *Übersicht existierender Konzepte zur Verwaltung verteilter dynamischer Datenbasen (dreiteilig).*

Bezeichnung		*Institution*
dynamisch/statisch	*Dimension*	*Modell*
Datentypen	*Gitterart*	
MP-Modell	*Plattformen*	
Applikationsbereich	*Zielsprache*	
Nearest Neighbor Tool (NNT) [123]	Forecast Systems Laboratory	
statisch	1-3 Dim	SPMD
F77-Datentypen	strukturierte Gitter	
PCL (Forecast Systems Lab)	Intel u.a., WsC	
Wettervorhersagemodelle (mittels FD)	F77	
OPlus [31]	Oxford University	
statisch	beliebig	SPMD
F77-Datentypen	unstrukturierte Gitter	
PVM, MPI	PVM, MPI	
gitterbasierte Anwendungen (z.B. CFD)	F77	
PARTI [38, 138]	ICASE, VA; Syracuse Univ., NY; Yale Univ., CT	
statisch	beliebig	SPMD
C-/F77-Datentypen	unstrukturierte Gitter	
NX	Intel, IBM SP1, CM-5	
unstrukturierte (Mehrgitter-)Verfahren (z.B. CFD)	C, F77	
PetSc [6, 7]	Argonne National Laboratory	
statisch	beliebig	SPMD
verteilte VuM, Datenfelder	strukturiert/unstrukturiert	
MPI	MPI	
numerische Anwendungen (z.B. CFD)	C, C++, F77	
Runtime System Library (RSL) [106]	Argonne National Laboratory	
dyn. Gitterschachtelung	2-3 Dim	Host/Node
F77-Datentypen	strukturierte Gitter	
Chameleon [61], PICL	IBM SP-1, Intel Delta	
Modelle zur Wettervorhersage und Geophysik	F77	
SUMAA3d [77, 78]	Argonne National Laboratory	
dynamisch	2 Dim	SPMD
integer, double	Dreiecksgitter	
MPI	MPI	
Anwendungen auf unstrukturierten Gittern	C	

Tabelle 1.1: *Übersicht existierender Konzepte zur Verwaltung verteilter dynamischer Datenbasen (dreiteilig).*

Die meisten dieser Werkzeuge sind auf bestimmte verteilte Gitterstrukturen beschränkt; dabei erlauben einige nur die Verteilung und Verwaltung *strukturierter Gitter* bzw. die Verteilung von Vektoren und vollbesetzten Matrizen (BlockComm, Canopy, Dome, NNT, RSL). Die Leistungsfähigkeit dieser Konzepte ist aus Sicht der Ziele dieser Arbeit sehr eingeschränkt; sie stellen jedoch die zeitlich ersten schichtenorientierten Ansätze dar, die versuchen, von reinen *message-passing*-Programmen zu abstrahieren.

Andere Konzepte sehen die Verwaltung *blockstrukturierter Gitter* vor (A++/P++, CLIC, LOCO, LPARX); dies bedeutet Dynamik auf einer Ebene mittlerer bis grober Granularität. Jeder Block ist ein strukturiertes Gitter, die Nachbarschaftsbeziehungen zwischen den Blöcken bilden einen allgemeinen Graphen. Diese Datenstruktur bot früh die Möglichkeit, Dynamik und Adaptivität auch auf Parallelrechnern bereitstellen zu können; daher sind ausgereifte Werkzeuge verfügbar (z.B. A++/P++, CLIC). Probleme bereiten jedoch die Erzeugung blockstrukturierter Gitter für komplexe Problemgeometrien sowie die Lastverteilung dieser gröbergranularen Datenstrukturen. Ein interessantes blockstrukturiertes Konzept neueren Datums ist die DAGH-Programmbibliothek, bei der beliebigdimensionale Gitter mit Hilfe von selbstähnlichen Abbildungen linearisiert werden (*space-filling curves* [128, 129]); das so entstandene eindimensionale Feld hat im wesentlichen die gleichen Lokalitätseigenschaften wie das ursprüngliche Gitter. Diese interessante Technik ist aber zugleich die Schwäche des Konzepts: komplexe Geometrien können nicht behandelt werden.

Einige weitere Ansätze bieten verteilte *unstrukturierte Gitter* (CHAOS, DIME, GRIDS, OPlus, PARTI, PetSc, SUMAA3d). Bei der Mehrzahl dieser Konzepte ist nur eine einmalige statische Verteilung des unstrukturierten Gitters möglich; eine spätere, evtl. mehrmalige Umverteilung ist nicht vorgesehen. Dies schließt z.B. die Parallelisierung adaptiver numerischer Verfahren aus. Die Verbindung von unstrukturierten Gittern *und* dynamischer Umverteilung bieten nur die Konzepte CHAOS, DIME und SUMAA3d.

Das bereits in **PARTI** implementierte *Inspector/Executor*-Schema wurde im CHAOS-Projekt auf das Konzept der *Lightweight Communication Schedules* erweitert. Die Grundidee, einen „Fahrplan“ für die oft verwendeten Kommunikationsschemata von iterativen Verfahren einmal vorab zu berechnen (*Inspector*) und dann oft zu verwenden (*Executor*), ist ein in vielen Arbeiten angewendetes Prinzip. Das in vorliegender Arbeit beschriebene Modell sieht eine ähnliche, aber sehr viel mächtigere und flexiblere Abstraktion vor (*DDD Interfaces*, Abschnitt 2.4.2).

Die **CHAOS-Bibliothek** bietet Zugriffe auf verteilte Array-Elemente über globale Indizes; dies wurde mit Hilfe einer verteilt speicherbaren *translation table* realisiert, in der für jeden globalen Index der Prozessor verzeichnet ist, der das zugehörige Array-Element speichert. Bei jedem Zugriff auf nichtlokale Daten muß daher zunächst mit dem Prozessor kommuniziert werden, der den Eintrag aus der *translation table* besitzt. Bei einer dynamischen Umverteilung der Daten muß die *translation table* aufwendig neu erstellt werden. Sowohl in PARTI als auch im Nachfolgeprojekt CHAOS muß die Irregularität der Daten auf indirekte Zugriffe über Indexfelder abgebildet werden; eine effiziente Implementierung von Referenzen zwischen Datenobjekten (z.B. über *pointer* in ANSI C) wie in dieser Arbeit beschrieben ist nicht möglich.

Die **PetSc-Programmbibliothek** bietet neben verteilten Vektoren und Matrizen (mit entsprechenden Konsistenz-Operationen) ebenfalls die aus PARTI/CHAOS bekannten Inspector/Executor-Strukturen, eingebettet in ein klar definiertes, objekt-orientiertes Konzept [7].

Die Ziele des **SUMAA3d-Projekts** beinhalten alle Schlüsselprobleme zur Entwicklung von parallelen Verfahren auf unstrukturierten, adaptiven Gittern. Dazu gehören neben Aspekten der Gittergenerierung, -glättung und -verfeinerung natürlich Methoden zur effizienten, skalierbaren Lösung von linearen Gleichungssystemen, aber auch modernste *virtual-reality*-Visualisierungsmethoden (z.B. interaktive adaptive Gitterfeinerung im *CAVE* [55]). Derzeit (Stand 1997) werden in *SUMAA3d* jedoch nur zweidimensionale Dreiecksgitter unterstützt; dies gilt ebenfalls für das **DIME-Projekt**. Eine Generalisierung auf drei Dimensionen, andere Gittertypen oder gar allgemeine Graphen kann eine komplette Neuentwicklung der existierenden Programmsysteme erfordern.

Sehr wenige bestehende Software-Werkzeuge behandeln das Problem verteilter allgemeiner Graphen (CHAOS++, Concurrent Graph). Es gibt zwar eine Vielzahl von Ansätzen für die Bereitstellung verteilter Objekte (siehe nächsten Abschnitt), die jedoch den Effizienzanforderungen aus dem Bereich der numerischen Anwendungen oftmals nicht genügen. Das objektorientierte Konzept **CHAOS++** kommt dem in dieser Arbeit vorgestellten Modell am nächsten. Es bietet zwei wesentliche Abstraktionen: *Mobjects* sind bewegliche Objekte, die ein *pack/unpack*-Funktionenpaar besitzen müssen, um in Nachrichtenpuffer kopiert werden bzw. wieder im Speicher etabliert werden zu können. *Gobjects* sind global adressierbare Objekte, die Kopien auf anderen Prozessoren besitzen können und auf die mittels globaler Zeiger (sog. *foreign pointer* [84] oder auch *global pointer* [35]) verwiesen werden kann. Durch die Anwendung objektorientierter Konzepte (Überladen von Operatoren, mehrfache Vererbung) können so elegant durch Zeiger verkettete, verteilte Graphstrukturen implementiert werden; einige Detailprobleme machen sich jedoch mit mangelnder Effizienz bemerkbar: jede globale Referenz auf ein Objekt eines anderen Prozessors wird mittels dessen Prozessornummer und der lokalen Adresse auf diesem Prozessor repräsentiert; somit verdoppelt sich der Speicheraufwand für alle Zeiger (vgl. Abschnitt 3.4.7). Zudem wird nicht nur die Kommunikation für die Migration von kompletten Objektkopien mittels des jeweiligen *pack/unpack*-Funktionenpaars abgewickelt, sondern auch der Datentransfer zwischen lokalen Objekten. Daher werden beim Konsistentmachen von Daten an einer Prozessorgrenze für jedes Objekt an dieser Grenze zwei virtuelle Funktionen aufgerufen, die im schlechtesten Fall nur einen *double*-Wert ein- bzw. auspacken. Dies kann einen erheblichen zusätzlichen Laufzeitaufwand bedeuten. Darüberhinaus vergrößert sich jedes Objekt einer Klasse mit virtuellen Funktionen mindestens um die Größe eines Zeigertyps (Implementierungsdetail der Sprache C++), was ebenfalls Speicherprobleme bewirken kann.

1.6.2 Verteilte Objekte

Ein beliebter Ansatz zur Einführung von Parallelität ist die Erweiterung bestehender objektorientierter Sprachen um parallele Konstrukte. Ein Beispiel, die Programmbibliothek CHAOS++ [34, 35], wurde im vorherigen Abschnitt bereits genannt; dort wurde die Sprache C++ um global adressierbare und migrierbare Objekte erweitert. In diesem und ähnlichen Ansätzen wird der Schwerpunkt meistens auf ein leicht zu bedienendes, mit dem objektorientierten Konzept verschränktes Programmiermodell gelegt. Zusätzlich streben manche Ansätze noch an, die Datenverteilung einer Kombina-

tion aus Compiler und Laufzeitsystem zu überlassen; der Anwendungsprogrammierer muß lediglich die Gleichzeitigkeit von Objektausführungen (*concurrency*) und evtl. die Kommunikation zwischen den einzelnen Objekten handhaben.

In vorliegender Arbeit wird ein Ansatz vorgestellt, der zwar auf verteilten Objekten aufsetzt, aber stets Mengen von diesen Objekten bearbeitet und auf diesen Mengen (in der OO-Literatur als *collections* bezeichnet) Kommunikationsprimitive zur Verfügung stellt (vgl. Kapitel 2). Dies kann effizient implementiert und doch mit klar definierten Schnittstellen versehen werden, wie im Rest der Arbeit gezeigt wird: jedes einzelne verteilte Objekt kann dabei sehr feingranulare Arbeit erledigen; durch die Zusammenfassung in Objektmengen können die feingranularen Objekte effizient gebündelt werden.

Stellt man jedoch Gleichzeitigkeit auf Ebene einzelner Objekte bereit, so dürfen die von diesen Objekten ausgeführten Aufgaben nicht mehr so feine Granularität haben, um gleiche Effizienz und Skalierbarkeit wie im vorliegenden Ansatz zu erreichen. Dennoch sollen nun die wichtigsten Konzepte bzw. Projekte verteilter Objekte vorgestellt werden; derzeit sind diese zwar für die hier angestrebte Algorithmen-Klasse noch nicht konkurrenzfähig zum Konzept dieser Arbeit, dies kann sich jedoch mit zunehmend schnelleren Kommunikationsnetzwerken, neuen Rechnerarchitekturen und immer ausgereifteren Compilertechnologien innerhalb der nächsten Dekade ändern.

Da der Schwerpunkt hier auf Anwendungen des wissenschaftlichen Rechnens liegen soll, werden in diesem Abschnitt nur Konzepte berücksichtigt, die Varianten der Sprache C++ darstellen (z.B. Ansätze für *Smalltalk* werden ausgeklammert). Bei den meisten Ansätzen wird die Sprache um parallele Erweiterungen und Konzepte ergänzt; einige wenige schränken die Funktionalität ein (z.B. bei Adreßoperationen und Zeigern), um dem Compiler die Arbeit zu erleichtern (z.B. im *Concert*-Projekt [37]). Ein Klassifikationsmerkmal für parallele objektorientierte Ansätze ist die bevorzugte Art der Parallelität; deshalb wird im folgenden zwischen *taskparallelen* und *datenparallelen* Ansätzen unterschieden.

Taskparallele Ansätze. Taskparallele Ansätze erweitern die gegebene Programmiersprache um Konstrukte, die es dem Anwendungsprogrammierer ermöglichen Nebenläufigkeit, Lokalität, Kommunikation und Task/Prozessor-Zuordnung zu spezifizieren. Damit werden beliebige, irreguläre Kommunikations- und Parallelitätsstrukturen unterstützt. Kernkonzepte sind der *global pointer* zur Bereitstellung eines globalen Namensraumes und der Aufruf von Methoden nichtlokaler Objekte über RPC-ähnliche Mechanismen (*Remote Procedure Call*); jeder Zugriff auf nichtlokale Daten löst also eine Kommunikation aus. Die Kontrolle der Granularität der Objekte obliegt dem Anwender, durch die aufwendigen Kommunikationsprimitive ist diese jedoch nach unten beschränkt. Für viele Anwendungen sind solche Ansätze daher ungeeignet, z.B. bei iterativen numerischen Verfahren. Die inhärente Datenparallelität dieser Verfahren ist orthogonal zum taskparallelen Konzept. Beispiele: *Compositional C++* [33], *Charm++* [80–82], *Mentat* [59].

Datenparallele Ansätze. Basierend auf der Annahme, daß sich die zu parallelisierende Arbeit eines Programms auf viele *gleichartige* Objekte verteilt, werden als datenparallele Spracherweiterungen Mengen von diesen Objekten gebildet, die partitioniert und den Prozessoren

zugeordnet werden können (ähnlich wie in nicht-objektorientierten, datenparallelen Sprachen). Ausgehend von einer objektorientierten Sprache (z.B. C++), bieten diese Ansätze Konstrukte an, um zu einer gegebenen Klasse eine *collection*-Klasse (z.B. *in pC++ [103, 104]*) oder *aggregate*-Klasse (z.B. *in C** [96]*) zu erzeugen. Der Compiler muß Schleifen über den Elementen einer solchen übergeordneten Klasse in lokale Schleifen über den Teilmengen der einzelnen Prozessoren umformen und die Methodenaufrufe entsprechend anpassen. Neben dieser aus dem SIMD-Prinzip abgeleiteten Kernfunktionalität bieten die einzelnen Ansätze noch zusätzliche Eigenschaften: *pC++* bietet z.B. einen globalen Namensraum für Objekte; *C*** sog. *slices*, um Teile von *aggregate*-Klassen einzeln anzusprechen (z.B. eine Zeile einer Matrix).

Gemischte Ansätze. Gemischte Ansätze bieten sowohl datenparallele Konstrukte (z.B. Objekt-*collections*) als auch taskparallele Konstrukte an. Im *Concert*-System [37] z.B. werden einer eingeschränkten Variante von C++ Sprachmittel zur Spezifikation von Nebenläufigkeit hinzugefügt (*conc*-Schlüsselwort). Gleichzeitig wird durch aggressive Optimierungsverfahren auf Compilerebene und mit Hilfe eines Laufzeitsystems mögliche Datenparallelität erkannt und durch spezielle Techniken in effiziente parallele Programme umgesetzt (z.B. *object inlining, method inlining*).

Zur Implementierung der Laufzeitsysteme zu den oben beschriebenen Ansätzen wird meistens *thread*-basierte Parallelität verwendet; selbst im *Concert*-System, dessen Compiler und Laufzeitsystem mit großem Aufwand auf die Generierung effizienter Programme getrimmt wurden[3], ist so die Granularität der Objekte in den existierenden Anwendungen noch eine Größenordnung von der Feinheit entfernt, die in unstrukturierten, dynamischen Anwendungen nötig ist (siehe Eigenschaften auf S. 17).

1.6.3 Leichtgewichtige Prozesse (*Threads*)

Gegenüber den *user-level*-SPMD-Modellen, bei denen der Zugang zur Parallelität über Datenaufteilung und explizite Kommunikation erfolgt, ermöglichen Ansätze auf der Basis von *leichtgewichtigen Prozessen* (*Threads*) implizite Parallelität ohne Zutun des Anwenders. Sind die einzelnen Durchläufe von Programmschleifen frei von Datenabhängigkeiten (evtl. signalisiert durch Annotationen durch den Programmentwickler), so können diese in Form von *Threads* gleichzeitig ausgeführt werden. Sobald ein solcher Prozeß auf nicht-lokale Daten zugreifen muß, wird eine Kommunikation angestoßen und der Prozeß solange zurückgestellt, bis die Kommunikationsanforderung erfüllt ist. Die Datenaufteilung bestimmt also implizit die Struktur der Parallelität. Typischerweise gibt es also sehr viele Threads im System, so daß eine effiziente Ausführungskontrolle (das sog. *scheduling*) eine Herausforderung darstellt.

Dieses Konzept wurde bereits in verschiedenen Architekturen in Hardware umgesetzt; dazu muß der Prozessor schnelle Kontextwechsel unterstützen (z.B. bei Tera MTA durch vielfache Registersätze, Abschnitt 1.2). Um jedoch portable Programme auf der Basis von Threads zu entwickeln,

[3] Der Compiler benötigt zur Übersetzung von 5000-10000 Programmzeilen mehrere Stunden! [36]

muß auf allen Plattformen die gleiche, geeignete Programmierschnittstelle angeboten werden, deren Implementierung hauptsächlich durch ein (Software-)Laufzeitsystem unterstützt wird. Dieses muß auf alle nötigen Rechner-Plattformen portiert werden. Einige Beispiele aus der Vielzahl der Softwaresysteme, denen das Thread-Modell zugrundeliegt, sind Cilk [25], Concert [37], Elite [16], Olden [124, 125], Multipol [146, 151] und TAM [41].

Cilk [25] bietet Erzeugung und Kontrolle von Threads samt zugehöriger Synchronisationsmechanismen als Erweiterung der Sprache C. Damit das Laufzeitsystem leicht implementierbar und unabhängig vom C-Laufzeitsystem bleibt, muß die Nebenläufigkeit in Cilk explizit formuliert werden. Die Synchronisation zwischen einzelnen Threads geschieht über sog. *continuations*; dies sind Daten bzw. Variablen, die von einem Thread beschrieben und von einem anderen Thread erwartet werden (übliche Technik, Syntax in Form von Argumenten einer Funktion). Neben einer Implementierung für verschiedene Systeme (z.B. Intel Paragon, SGI Power Challenge) wurden theoretische Modelle zur Performance-Modellierung von Cilk-Scheduler und parallelen Anwendungen erarbeitet.

Das *Concert*-Projekt [37] wurde im vorigen Abschnitt bereits angeführt. Die Nebenläufigkeit wird hier ebenfalls über Threads erschlossen. Diese werden allerdings direkt vom Laufzeitsystem erzeugt und verwaltet, auf Ebene des Anwendungsprogramms sind keine Threads sichtbar (implizite Threadkontrolle). Als Besonderheit ist hier das hybride Stack/Heap-Ausführungsmodell hervorzuheben: Threads können zunächst effizient auf dem lokalen Stack aufgesetzt werden (entsprechend einem normalen Funktionsaufruf); sofern nötig, kann der Thread auf den dynamisch allokierten Heap ausgelagert werden. Diese Technik ermöglicht im allgemeinen schnelle Kontextwechsel zwischen logischen Threads.

Im Erlangener *Elite*-Projekt [16] werden leichtgewichtige Threads auf Ebene des Anwendungsprogramms untersucht. Der Elite-Scheduler verwendet Lokalitätsinformation eines Hardware-Monitors (z.B. *cache-miss*-Raten) zur Entscheidung über Aktivierung bzw. Suspendierung von Threads. Über den speziell entwickelten *Sleeping Threads*-Mechanismus wird (mittels weniger Änderungen im Betriebssystemkern) die Implementierung von effizienten *user-level*-Threads auf Basis von Betriebssystem-Threads gröberer Granularität ermöglicht [89].

Bei *Olden* [124, 125] wird das Hauptgewicht auf die Unterstützung dynamischer, rekursiver Datenstrukturen gelegt. Jedem Aufruf zur dynamischen Allokierung von Speicher wird die logische Nummer des Prozessors als Argument mitgegeben, dessen lokaler Speicher benutzt werden soll. Greift später ein Thread auf diesen Speicher zu, so wird ein *Trap*-Ereignis ausgelöst, das die Migration des Threads auf den Prozessor durchführt, der den Speicher besitzt. Nebenläufigkeit wird durch die explizite Einführung von sog. *futures* (d.h. *continuations*) in den Programmcode eingefügt.

Das Laufzeitsystem von *Multipol* [146, 151] setzt implizit Threads ein, um einen Satz von verteilten, irregulären Datenstrukturen zur Verfügung zu stellen. Beispiele solcher Datenstrukturen sind Hashtabelle, Ereignisgraph oder Task-Warteschlangen. Neben eingebauten Scheduling-Strategien kann der Anwendungsprogrammierer zusätzliche Strategien implementieren. Um auf *distributed-memory*-Systemen hohen Kommunikationsaufwand zu vermeiden, kann das Laufzeitsystem Nachrichten aggregieren, d.h. viele kleine zu wenigen großen Nachrichten zusammenfassen.

Eine frühe Arbeit beschreibt die *Threaded Abstract Machine* (*TAM* [41]), deren Programmiermodell als Grundlage für Compiler anderer, höherer Sprachen benutzt werden kann. Die Threads werden

hier vom Compiler erzeugt; Kommunikation zwischen Threads erfolgt über das Paradigma der *Active Messages* [141]. Active Messages ist eine Form von *message-passing*, bei der beim Empfang einer Nachricht ein Interrupt ausgelöst wird, der eine Handler-Funktion aufruft. Diese nimmt die Nachricht entgegen und arbeitet sie in die laufende Berechnung ein. Im Kontext der TAM bedeutet dies, Entscheidungen zur Aktivierung von Threads aufgrund ankommender Nachrichten anzustossen.

Threads erlauben meist eine elegante Formulierung von Nebenläufigkeit bzw. Parallelität, sei es in impliziter Form als Basis für höhere Programmiermodelle oder in expliziter Form auf Anwendungsebene. Leider ist die Erzeugung und Verwaltung von solchen leichtgewichtigen Prozessen ohne Hardware-Unterstützung oftmals teuer im Vergleich zur Rechenzeit eines feingranularen Objekts; je nach Implementierung und Architektur kann die Erzeugungszeit mehreren hundert Fließkommaoperationen entsprechen. Für iterative numerische Verfahren im besonderen ist daher eine derartige, feingranulare Parallelisierung nach heutigem Stand nicht praktikabel (siehe auch Schlußbemerkung in Abschnitt 1.6.2).

Kapitel 2

Dynamic Distributed Data: Konzept

Im folgenden Kapitel wird, ausgehend vom Lösungsansatz aus Abschnitt 1.5, ein Konzept zur Parallelisierung von Algorithmen auf dynamischen, graphartigen Datenstrukturen vorgestellt. Die daraus folgende Architektur paralleler Programme, die zugrundeliegende Spezifikation und die Funktionalität bzw. die Implementierung der resultierenden Programmbibliothek wurden, inspiriert durch die Hauptaspekte der Problematik, unter dem Begriff *Dynamic Distributed Data* (kurz: *DDD*) zusammengefaßt.

Das Kapitel gliedert sich in vier Abschnitte. Zunächst wird die Struktur paralleler Anwendungsprogramme bei Verwendung des DDD-Konzepts übersichtshaft dargelegt; nach einem Einschub über statische und dynamische Lastverteilung und deren Einbettung ins DDD-Konzept schließt sich die formale Beschreibung des zugrundeliegenden Daten- und Konsistenzmodells an. Zum Schluß wird die Funktionalität der DDD-Programmbibliothek und ihre Gliederung aus der Sicht eines Anwendungsprogramms beschrieben.

Darüberhinaus sei an dieser Stelle bereits auf Abschnitt 5.1 hingewiesen, in welchem viele praktische Gesichtspunkte zum Einsatz des hier vorgestellten DDD-Konzepts anwendungsbezogen aufgelistet sind. In Zusammenhang mit der Funktionalitätsbeschreibung weiter unten in diesem Kapitel wird damit dem potentiellen DDD-Benutzer ein schneller Einstieg ermöglicht.

2.1 Struktur paralleler Anwendungsprogramme

Wie bereits im letzten Kapitel dargelegt, wird die Basis des DDD-Parallelisierungskonzepts durch die Partitionierung und anschließende Verteilung des globalen Datengraphen gebildet. Ein vorhandenes, sequentielles Anwendungsprogramm wird somit zunächst über seine *Datenstrukturen* an das parallele Modell gekoppelt. Dabei wird die bisherige Datenbasis auf das DDD-Modell des verteilten Graphen abgebildet. Dadurch bleibt die bisherige Funktionalität der Anwendung unberührt, das Programm ist also nach der Ankopplung noch immer sequentiell (auf einem Prozessor) lauffähig. Eine formale Spezifikation der Graph-Schnittstelle wird weiter unten in Abschnitt 2.3 gegeben.

Nach der Einbettung der sequentiellen Datenbasis in das parallele Modell können die verteilte Konstruktion sowie die dynamische Verteilung des Graphen und ebenfalls die anwendungsgesteuerte

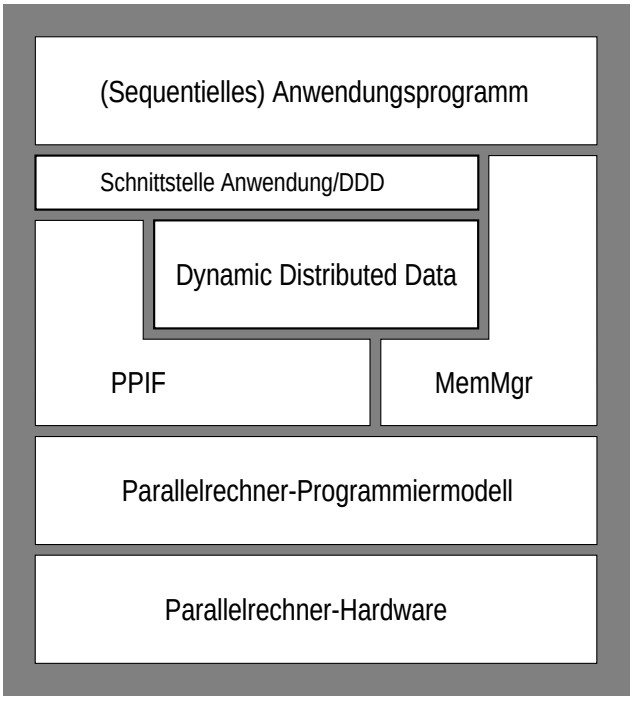


Abbildung 2.1: *Schichtenmodell eines parallelen Anwendungsprogramms beim Einsatz von DDD.*

Kontrolle der Datenkonsistenz durchgeführt werden. Zu diesen Zwecken benutzt das Anwendungsprogramm die Funktionen und Routinen der DDD-Bibliothek (beschrieben in Abschnitt 2.4); diese arbeiten auf der verteilten Datenbasis, die natürlich der formal definierten Graphschnittstelle entspricht. Abb. 2.1 zeigt den Zusammenhang: der Programmcode der Anwendung bleibt gegenüber der rein sequentiellen Version größtenteils unverändert; durch das formale Graphmodell ist eine klare Abstraktionsebene definiert, für welche die DDD-Bibliothek eine genau spezifizierte Funktionalität anbietet. Die Implementierung dieser Funktionalität ist von der Anwendung aus nicht sichtbar.

Auf der Hardwareseite wird o.B.d.A. ein Parallelrechner mit verteiltem Speicher angenommen, dessen *message-passing*-Funktionalität über eines von vielen gängigen Programmiermodellen zugänglich ist. Um die Portabilität des DDD-Modells und damit der Anwendungsprogramme zu gewährleisten, setzen die Funktionen der DDD-Bibliothek nicht direkt auf einem bestimmten dieser Programmiermodelle auf, sondern verwenden stattdessen grundlegende Funktionen der eingeschobenen PPIF-Zwischenschicht (*parallel processing interface*). Diese faßt blockierende und nichtblockierende Kommunikationsroutinen unter einem *virtual-channel*-Konzept zusammen (siehe [12], Anhang A.1).

Diese PPIF-Zwischenschicht steht für alle gängigen *message-passing*-Programmiermodelle zur Verfügung: MPI, PVM, NX (z.B. Intel Paragon), Cray-SHMEM sind einige Beispiele. Damit sind parallele Anwendungen auf der Basis von DDD auf den meisten Supercomputer-Plattformen lauffähig.

Da sequentielle Anwendungen auf dynamischen Datenstrukturen oftmals die (dynamische) Speicherverwaltung nicht direkt dem Betriebssystem überlassen, sondern ein eigens entwickeltes Speicherverwaltungsmodul einsetzen, muß ein Parallelisierungskonzept für solche Anwendungen eine Integrationsmöglichkeit mit fremden Speicherverwaltungen vorsehen. Dazu wird im Rahmen des DDD-Konzepts eine *MemMgr-Schnittstelle* (*memory manager*, siehe Abb. 2.1) definiert, wobei die DDD-Bibliothek selbst zur dynamischen Speicherverwaltung nur Funktionen benutzt, die an der MemMgr-Schnittstelle bereitgestellt werden müssen. Dies ermöglicht die Integration der Speicherverwaltung von DDD-Bibliothek und Anwendungsprogramm: Sollte die Anwendung ein eigenes Speicherverwaltungsmodul verwenden, so kann dieses ebenfalls die MemMgr-Funktionen bereitstellen; falls die Anwendung bereits direkt auf Betriebssystemfunktionen (z.B. *malloc()*, *free()*) aufsetzt, so wird dies auch die DDD-Bibliothek so handhaben.

Alle zur Parallelisierung nötigen Funktionen, die weder bereits im PPIF-Modul noch im DDD-Modul enthalten sind, werden entweder direkt in das Anwendungsprogramm integriert oder in einem eigenen Modul *Schnittstelle Anwendung/DDD* zusammengefaßt (vgl. Abb. 2.1). Auf diese Weise ist es möglich, durch Einsatz eines Präprozessors alle Bezüge auf die parallelisierte Programmversion abzuschalten und somit sequentielle und parallele Version des Programms zu integrieren. Die eigentliche Parallelisierungsarbeit beschränkt sich also auf die Erstellung dieses Schnittstellenmoduls, wozu in erster Linie folgende Aufgaben gezählt werden:

- Durchführung und Initialisierung der Datenankopplung (siehe dazu Abschnitt 2.4.1)
- Bereitstellung von Operationen aus der Anwendung (d.h. *callback*-Funktionen), die vom DDD-Laufzeitsystem aufgerufen werden können (sog. *Handler*, vgl. Abschnitt 2.4.5)
- Lastbalancierung, d.h. gleichmäßige Verteilung der Datenbasis bei gleichzeitiger Minimierung von redundanten Daten (Abschnitt 2.2)
- Lastverteilung, d.h. Strategie zur Migration der Daten, um die von der Lastbalancierungskomponente geforderte Verteilung samt Überlappung herzustellen (vgl. ebenfalls Abschnitt 2.2)

Um den Arbeitsaufwand noch über den Einsatz von DDD hinaus zu reduzieren, können einzelne Teilaufgaben von spezielleren Werkzeugen unterstützt bzw. automatisiert werden. Dies wird für die Teilaufgabe der Lastbalancierung im nächsten Abschnitt angedeutet; für die anderen Teilaufgaben wird an dieser Stelle lediglich auf Abschnitt 6.3.3 verwiesen.

2.2 Lastbalancierung und DDD

Die Aufgabe der Verteilung der Datenbasis läßt sich in zwei Phasen untergliedern [10]: zunächst muß die neue Partitionierung bestimmt werden, dazu dienen Verfahren der sog. *Lastbalancierung*. Dieser Vorgang verändert die Datentopologie noch nicht, sondern berechnet lediglich eine gleichmäßige Verteilung der Datenbasis unter verschiedenen Zusatzbedingungen (z.B. Minimierung der redundanten Daten und damit des Kommunikations- bzw. Speicheraufwands, Vermeidung mehrfach zusammenhängender Teilbereiche). Als zweite Phase folgt der Vorgang der *Lastverteilung*; dies

beinhaltet die eigentliche Migration der Datenbasis gemäß den Resultaten aus der ersten Phase sowie die Aufrechterhaltung von redundanten Teilen der Datenbasis (z.B. Überlappungsbereiche eines Gitters).

Die letztere Phase, also die Migration von Teilen der Datenbasis, ist in der DDD-Konzeption berücksichtigt. DDD unterstützt das Versenden von verteilten Objekten und die Erzeugung von redundanten Daten, also Objektkopien auf verschiedenen Prozessoren. Der Anwendungsprogrammierer muß den Vorgang der Datenverteilung auf die Befehle abbilden, die DDD zur Verfügung stellt; dies sind im wesentlichen Kommandos zur Erzeugung von Objektkopien auf entfernten Prozessoren bzw. zur Löschung von lokalen Objektkopien (vgl. Abschnitt 2.4.3). Diese Abbildung kann für kompliziert strukturierte Datenbasen aufwendig sein, kann aber durch spezielle Werkzeuge unterstützt werden (Abschnitt 6.3.3).

Die erste Phase, d.h. die Balancierung der Datenbasis und damit der Prozessorlast, ist stark anwendungsabhängig und wurde deshalb nicht ins DDD-Konzept integriert. Je nach Anwendungsbereich muß der Lastbalancierer auf die speziellen Eigenschaften und Anforderungen dieses Bereichs abgestimmt werden; oftmals sind jedoch speziell entwickelte Werkzeuge zur Lastverteilung verfügbar, die unkompliziert in das parallele Anwendungsprogramm eingebaut werden können. Benötigt das zu parallelisierende Verfahren keine dynamische Veränderung der Datentopologie, so reicht eine einmalige Verteilung der Daten zu Beginn der Laufzeit des Programms, also ein *statisches* Lastverteilungsverfahren. Bei ständiger Veränderung der Datentopologie dagegen kann eine ständige Nachführung des Lastgleichgewichts notwendig sein, dies muß ein *dynamisches* Lastverteilungsverfahren leisten [149]. Bei den hier angestrebten SPMD-Programmen wird die dynamische Veränderung der Datenbasis phasenweise auftreten, der dadurch nötige *quasistatische* oder auch *iterativ statische* Lastverteilungsalgorithmus ist ein Spezialfall der *dynamischen* Strategien. Aus der Sicht der drei Komponenten Anwendung, DDD und Lastbalancierungskomponente ergibt sich folgendes Bild:

- Das *Anwendungsprogramm* benutzt DDD und dessen Funktionalität als komfortable, verteilte Datenbasis. Es ruft den Lastbalancierer auf, wann immer die Datenbasis derart verändert wurde, daß ein Ungleichgewicht in der Prozessorlast auftritt, um das Gleichgewicht wiederherzustellen. Danach benutzt das Anwendungsprogramm die DDD-Bibliothek, um das Ergebnis des Lastbalancierungswerkzeugs in die Tat umzusetzen.
- Die *DDD-Bibliothek* stellt der Anwendung die verteilte Datenbasis zur Verfügung und führt die Umverteilung durch, sobald der Lastbalancierer eine neue Verteilung errechnet hat.
- Die *Lastbalancierungskomponente* hat Zugriff auf die durch DDD verteilte Datenbasis und errechnet neue Verteilungen auf der Basis der alten Verteilung und zusätzlicher Informationen (z.B. Prozessorlast pro zu verteilendem Objekt), sobald das Anwendungsprogramm die Notwendigkeit zur Umverteilung feststellt. Da der Lastbalancierer keine Umverteilung der Daten vornimmt, benötigt er die Funktionen der DDD-Bibliothek im allgemeinen nicht (es sei denn, er arbeitet selbst parallel).

Da die Lastbalancierungskomponente klare Schnittstellen sowohl zum Anwendungsprogramm als auch zum DDD-Laufzeitsystem aufweist, können bereits hinreichend bekannte Lastbalancierungs-

verfahren leicht und unabhängig vom restlichen Programmsystem in parallele Anwendungen integriert werden. Zum schnellen Einstieg steht eine Reihe von (zumindest zu Forschungszwecken) frei verfügbaren Werkzeugen zur Verfügung, die eine quasistatische Lastverteilung erlauben, als Beispiele seien hier *Chaco* [71], *Metis* [83], *Jostle* [143], *TOP/DOMDEC* [51], *Scotch* [118] und *PARTY* [121] genannt.

Die verfügbaren Lastbalancierungsstrategien, die unter anderem in den genannten Graphpartitionierungswerkzeugen zur Anwendung kommen, werden nach [149] bzw. [11, Anhang A] in drei Klassen eingeteilt:

- *Verfahren der diskreten Optimierung* versuchen, das Minimum eines gegebenen Kostenfunktionals zu finden. Da exakte Verfahren für realistisch große Probleme wegen ihrer exponentiellen Komplexität nicht verwendet werden können, wurden nichtdeterministische Heuristiken (z.B. *Simulated Annealing* oder *Genetische Algorithmen*) oder deterministische Heuristiken (z.B. *Kernighan-Lin*-Verfahren oder *Greedy*-Verfahren) entwickelt. In der Praxis werden aus Rechenzeitgründen jedoch meist nur deterministische Heuristiken eingesetzt.

- *Verfahren zur Graphpartitionierung* bestimmen (teilweise rekursiv) Aufteilungen des Graphen in zwei oder mehr Teilgraphen. Dazu gehören geometrieorientierte Verfahren (z.B. RCB und RIB) sowie graphorientierte Verfahren (z.B. Spektrale Bisektion [126, 134]).

- *Diffusionsverfahren* können als dezentrale Verfahren verteilt und damit skalierbar eingesetzt werden. Jeder Prozessor gibt Teile seines lokalen Graphen an seine Nachbarprozessoren ab oder erhält von diesen neue Teile (z.B. [64, 72]).

Die algorithmischen Problemfelder, bei denen eine Parallelisierung mittels statischer Lastbalancierung nicht ausreicht, sind breit gestreut: adaptive Gitter (z.B. für instationäre Berechnungen), Gittergenerierung, Partikelmethoden (z.B. in der Astrophysik), *branch-and-bound*-Verfahren, parallele Datenbanken, oder parallele ereignisgesteuerte Simulation sind nur einige Bereiche, bei denen eine dynamische Umverteilung stattfinden muß. Die fünf wichtigsten *Forderungen* an ein dynamisches Lastbalancierungsverfahren sind [70]:

1. Die Lastbalancierung muß selbst *verteilt* ablaufen, um mit der Problemgröße zu skalieren.

2. Die Lastbalancierung muß *schnell* im Vergleich zur eigentlichen Berechnungsaufgabe sein (im statischen Fall genügt ein Präprozessor nahezu ohne Laufzeitvorgaben).

3. Die Lastbalancierung muß *speichersparend* ablaufen (vgl. Nachbemerkung zum iterativen Lasttransfer in Abschnitt 3.4.7).

4. Der Lastbalancierer muß auf *anwendungs- und problemabhängige Informationen* zurückgreifen; die dynamische Strategie kann nicht ebenso leicht hinter einer abstrakten Schnittstelle verborgen werden wie im statischen Fall.

5. Der Lastbalancierer muß die *vorherige Verteilung* in die Entscheidung einbeziehen; das Datenvolumen bei der Umverteilung muß minimiert werden.

Bei der Entwicklung eines dynamischen Lastbalancierungsverfahrens muß ein Kompromiß zwischen folgenden Faktoren gefunden werden: Lastungleichheit, Kommunikationsaufwand, Häufigkeit der Umverteilung, Kosten der Umverteilung, Datenvolumen bei der Umverteilung, Komplexität des Programmcodes. Diese Faktoren sind abhängig von der Art des gestellten Anwendungsproblems, vom Algorithmus zu dessen Lösung und von der zugrundeliegenden Hardware. Ein interessantes Beispiel für den Einfluß der verschiedenen Faktoren auf die Güte der Lastbalancierung bietet die Klasse der adaptiven Mehrgitterverfahren. Dabei muß sowohl die horizontale Kommunikation zwischen den Prozessorpartitionen des gleichen Gitters als auch die vertikale Kommunikation zwischen Gittern verschiedenen Verfeinerungsgrades berücksichtigt werden. Neben der geforderten Gleichverteilung der Last muß also ein Kostenfunktional minimiert werden, das die verschiedenen Einflüsse von zusätzlich nötiger Kommunikation zusammenfaßt. In [93] wird dazu ein *Clustering*-Verfahren vorgestellt, bei dem der verteilte Graph durch Clusterbildung vergröbert wird, um so den möglichen Kommunikationsaufwand zu reduzieren und gleichzeitig das Lastverteilungsproblem an sich zu vereinfachen.

(Quasi-)dynamische Lastbalancierung ist ein aktuelles Forschungsthema [48, 143]. Das DDD-Konzept kann zu diesem Thema einen Beitrag leisten, indem es die Erstellung von komplexen, parallelen Programmen mit dynamischen Veränderungen der Datenbasis ermöglicht. Im Sinn von Forderung 1-3 bietet DDD die notwendigen Effizienzeigenschaften, im Hinblick auf Forderung 4 die nötige Abstraktion durch das zugrundeliegende formale Graphenmodell (Abschnitt 2.3). Bislang muß durchaus noch als Problem für die Entwicklung dynamischer Lastbalancierungsstrategien die fehlende Verfügbarkeit von parallelen Codes gesehen werden, die adäquate Testumgebungen bieten könnten.

2.3 Formales Modell

Im folgenden Abschnitt wird ein formales Modell einer verteilten Datenstruktur und deren Erzeugung beschrieben. Abb. 2.2 zeigt die Vorgehensweise in vereinfachter Form.

Der angestrebten verteilten Datenstruktur im parallelen Algorithmus (Abb. 2.2, rechts) liegt eine korrespondierende Datenstruktur des (existierenden) sequentiellen Algorithmus zugrunde. Diese *globale* Datenstruktur (Abb. 2.2, links) soll dazu dienen, die Anforderungen an Funktionalität und Konsistenz definieren. Sie bildet somit die Basis für die korrekte und effizienzorientierte Verteilung der Daten.

Das abstrakte Datenmodell benutzt *Graphen*, um die Eigenschaften der verschiedenen Datenstrukturen zu definieren. Dabei besteht die verteilte Struktur aus mehreren *lokalen Graphen* (Abb. 2.2), die jeweils einem der beteiligten Prozessoren zugeordnet sind bzw. in dessen Speicher abgebildet werden. Der Schritt vom globalen zum verteilten Graph entspricht der eigentlichen Parallelisierung des Programms bzw. der Verteilung (Partitionierung) der Datenstruktur und kann durch das formale Modell einfach und konstruktiv beschrieben werden. Ein einfaches Beispiel begleitet dabei die Darlegung des formalen Modells.

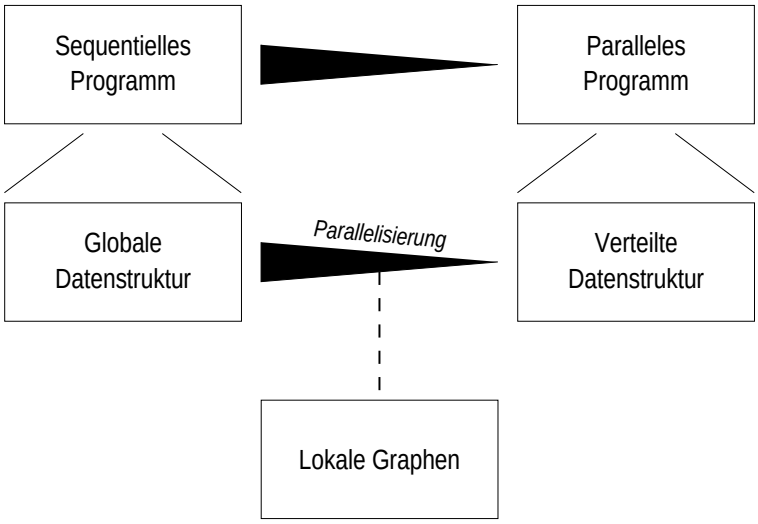


Abbildung 2.2: *Parallelisierung und Datenverteilung im DDD-Modell*

2.3.1 Globale Sicht der Datenstruktur

Die globale Datenstruktur wird in Analogie zu ihrer Implementierung auf einem sequentiellen Rechner definiert; auf einem Parallelrechner dient diese Struktur als Grundlage zur genauen Spezifikation der verteilten Daten. Dabei repräsentiert ein *globaler Graph* die globale Datenstruktur im formalen Modell, der aus globalen Objekten und einer Referenzrelation zwischen diesen Objekten zusammengesetzt wird. Abb. 2.3 zeigt ein einfaches Beispiel eines solchen Graphen.

DEFINITION 1 (GLOBALES OBJEKT)
Ein globales Objekt $\tilde{o} \in \tilde{O}$ *sei definiert durch* $\tilde{o} = (\mathsf{type}, \mathsf{data})$ *mit Struktur* `type` *und Dateninhalt* `data`. $\tilde{O}$ *sei die Menge aller globalen Objekte.*

Globale Objekte können gemäß ihrer Strukturbeschreibung `type` klassifiziert werden; `type` beschreibt dabei den *internen Aufbau* eines bestimmten Objekttyps als Menge von *Elementen*, d.h. Tupeln (`position,size,element`), mit folgender Bedeutung:

- `position` erlaubt die Lokalisierung des beschriebenen Elements dieser Strukturdefinition.
- `size` $\in \mathbb{N}$ gibt die Größe (d.h. den Speicherbedarf) des beschriebenen Elements an.
- `element` $\in \tilde{E}$ gibt den Typ des Elements an; $\tilde{E}$ ist dabei die Menge der bei globalen Objekten zulässigen Elementtypen.

Der Dateninhalt `data` läßt sich als Menge von Paaren (`position,value`) schreiben; dabei sei jedem Element der Typbeschreibung ein solches Paar über die `position`-Angabe zugeordnet.

DEFINITION 2 (GLOBALE REFERENZRELATION)
Die Notation

$$\tilde{o}_i \overset{\mathsf{ref}}{\Leftrightarrow} \tilde{o}_j$$

bezeichne die Situation, daß ein (referenzierendes) Objekt $\tilde{o}_i$ ein zweites (referenziertes) Objekt $\tilde{o}_j$ referenziert. Die globale Referenzrelation $\tilde{R}$ sei nun definiert durch

$$\forall \tilde{o}_i, \tilde{o}_j \ : \quad (\tilde{o}_i, \tilde{o}_j) \in \tilde{R} \ \Leftrightarrow \ \tilde{o}_i \overset{\text{ref}}{\Leftrightarrow} \tilde{o}_j,$$

stellt also die Menge aller in der globalen Datenstruktur vorkommenden Referenzen dar.

Als *Typ* $\mathtt{type}(\tilde{r})$ der Referenz $\tilde{r} \in \tilde{R}$ soll im folgenden der Typ des referenzierten Objekts verstanden werden:

$$\forall (\tilde{o}_i, \tilde{o}_j) \in \tilde{R} \ : \quad \mathtt{type}((\tilde{o}_i, \tilde{o}_j)) = \mathtt{type}(\tilde{o}_j)$$

Aus den globalen Objekten und der Referenzrelation läßt sich nun leicht ein globaler Graph konstruieren, der die Struktur der Datenbasis genau repräsentiert.

DEFINITION 3 (GLOBALER GRAPH)
Der globale Graph $\tilde{G}$ (zur sequentiellen Programmversion) wird aus den globalen Objekten $\tilde{O}$ (Knoten) und den globalen Relationen $\tilde{R}$ (Kanten) durch $\tilde{G} = (\tilde{O}, \tilde{R})$ konstruiert.

$\tilde{G}$ ist ein gerichteter Graph, da die oben definierte Referenzrelation nicht symmetrisch sein muß. Für alle Objekte des Graphen wird die `data`-Komponente im allgemeinen unterschiedlich sein; jede `type`-Komponente entspricht allerdings im Normalfall genau einem (zusammengesetzten) Datentyp des Anwendungsprogramms, es gibt also eine beschränkte Zahl von Äquivalenzklassen bezüglich der `type`-Eigenschaft. Dies spiegelt die folgende Definition wider.

DEFINITION 4 (MENGE DER DDD-TYPEN)
Die Menge der DDD-Typen T sei die Menge der Typbeschreibungen aller Objekte aus $\tilde{O}$, also: $T = \{t \mid \exists \tilde{o} \in \tilde{O}$ mit $\mathtt{type}(\tilde{o}) = t\}$.

Zur vollständigen Spezifikation fehlt nun noch die Menge der Elementtypen; mit der `element`-Eigenschaft werden die im Programm vorkommenden Datentypen je nach ihrer Relevanz für das formale Modell klassifiziert. Die oben definierte Referenzrelation muß im Programm geeignet implementiert werden, dies kann je nach Programmiersprache beispielsweise durch hardwarenahe *pointer* (also Speicheradressen, z.B. in ANSI C) oder durch Indexrechnungen (z.B. in Fortran 77) geschehen. Elemente, welche die Referenzrelation implementieren, werden mit dem Elementtyp `ObjPtr` bezeichnet; alle anderen (Daten-)Elemente sind vom Typ `Data`.

DEFINITION 5 (MENGE DER GLOBALEN ELEMENTTYPEN)
Die Menge der globalen Elementtypen $\tilde{E}$ wird definiert als $\tilde{E} = \{$`Data`, `ObjPtr`$\}$; es wird also zwischen Daten- *und* Zeigerelementen *unterschieden.*

Die genaue Beziehung zwischen der Referenzrelation und den Elementen vom Typ `ObjPtr` regelt folgende Definition:

DEFINITION 6 (IMPLEMENTIERUNG VON REFERENZEN, NULLREFERENZ)
Für jedes `ObjPtr`*-Element $e \in t$ eines globalen Objekts $\tilde{o}_i$ mit der Typbeschreibung $\mathtt{type}(\tilde{o}_i) = t \in T$ gilt:*

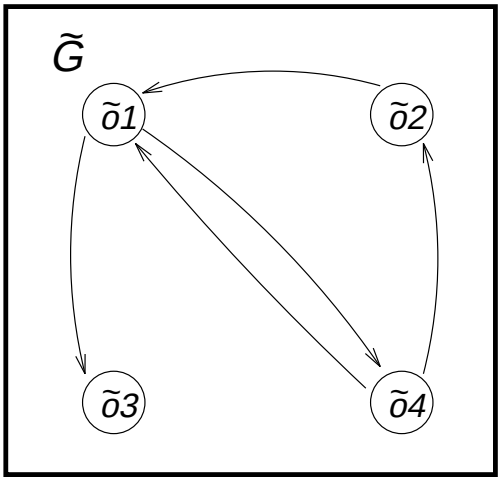


Abbildung 2.3: *Einfaches Beispiel eines globalen Graphen.*

- entweder *enthält das zugehörige Datum einen Verweis auf ein globales Objekt* $\tilde{o}_j$ *und implementiert damit die Referenzrelation* $\tilde{o}_i \overset{\text{ref}}{\Leftrightarrow} \tilde{o}_j$
- oder *es enthält die* Nullreferenz $\tilde{o}_i \overset{\text{ref}}{\Leftrightarrow} \oslash$.

Die ungültige Referenz (Nullreferenz) wird also als Referenz auf ein (nicht existierendes) Nullobjekt $\oslash$ betrachtet.

Aus praktischen Gründen ist es sinnvoll, darüberhinaus als weitere Elementtypen wiederum Objekte eines bestimmten DDD-Typs zuzulassen; diese hierarchische Form der Typdefinition geht einher mit Techniken zur Datentypdefinition, die aus diversen Programmiersprachen bekannt sind (z.B. Klassenhierarchien in C++). Dies vereinfacht in der Praxis die Definition von DDD-Typen, wird hier jedoch weggelassen, um das formale Modell nicht unnötig aufzublähen.

Als einfaches Beispiel zeigt Abb. 2.3 eine Datenstruktur bzw. den dazugehörigen Graphen $\tilde{G}$ mit vier globalen Objekten $\{ \tilde{o}_1, \tilde{o}_2, \tilde{o}_3, \tilde{o}_4 \}$ und folgender Referenzrelation:

$$\tilde{R} = \{ (\tilde{o}_1, \tilde{o}_3), (\tilde{o}_1, \tilde{o}_4), (\tilde{o}_2, \tilde{o}_1), (\tilde{o}_4, \tilde{o}_1), (\tilde{o}_4, \tilde{o}_2) \}$$

2.3.2 Lokale Sicht der Datenstruktur

Bei einer auf den Parallelrechner verteilten Datenstruktur gibt es erstmals einen *Lokalitätsbegriff*, da sich jedes Objekt in einem der verschiedenen Adreßräume aufhalten kann. Um diesen im Modell zu repräsentieren, wird die bisherige Objekt-Definition um eine *Prozessornummer* ergänzt.

DEFINITION 7 (LOKALES OBJEKT)
Ein (lokales) Objekt $o \in O$ *sei definiert durch* $o = (\mathsf{type}, \mathsf{data}, \mathsf{proc}, \mathsf{prop})$ *mit Struktur* type, *Dateninhalt* data, *Prozessornummer* $\mathsf{proc} \in P$ *und zusätzlicher Eigenschaft* prop. *O sei die Menge aller lokalen Objekte.*

Die Menge der (abstrakten) Prozessoren P kann dabei reale Prozessoren, aber auch Prozesse bezeichnen. Ebenso muß der Lokalitätsbegriff nicht unbedingt über verschiedene Adreßräume, sondern kann auch über den Zugriffsaufwand auf eigene bzw. fremde Objekte definiert werden (z.B. bei NUMA-Rechnerarchitekturen [74, S. 22f]). Die zusätzliche Objekt-Eigenschaft `prop` wird an dieser Stelle noch nicht benötigt und daher erst in Abschnitt 2.3.7 als Erweiterung genauer definiert.

Da der interne Aufbau bei lokalen bzw. globalen Objekten keine prinzipiellen Unterschiede aufweist, kann die Strukturbeschreibung `type` im wesentlichen beibehalten werden. Lediglich die Menge der vorkommenden Elementtypen wird erweitert: Lokale Objekte können einerseits Daten enthalten, die auf jedem Prozessor gültig sind (z.B. auf das Diskretisierungsgebiet bezogene Ortsvektoren), andererseits aber auch Daten, die nur auf einem bestimmten Prozessor Gültigkeit besitzen (z.B. Informationen zur Speicherverwaltung). Erstere Datenelemente bekommen nun den Typ `GData`, letztere den Typ `LData`.

DEFINITION 8 (MENGE DER LOKALEN ELEMENTTYPEN)
Die Menge der lokalen Elementtypen E wird definiert als $E = \{\texttt{GData}, \texttt{LData}, \texttt{ObjPtr}\}$; es wird also zwischen globalen *und* lokalen *Datenelementen und Zeigerelementen unterschieden.*

Zu jedem Prozessor $p \in P$ kann nun die lokale Objektmenge O^p definiert werden:

$$\forall p \in P: \quad O^p := \{o \mid \mathtt{proc}(o) = p\}.$$

Für die Menge aller lokalen Objekte O gilt

$$O = \bigcup_{p \in P} O^p,$$

die Mengen O^p bilden also eine Zerlegung von O.

DEFINITION 9 (LOKALE REFERENZRELATIONEN, LOKALE GRAPHEN)
Definiert man die lokalen Referenzrelationen R^p durch

$$\forall p \in P: \quad (o_i, o_j) \in R^p \;\Leftrightarrow\; o_i \in O^p \wedge o_j \in O^p \wedge o_i \overset{\mathsf{ref}}{\Longleftrightarrow} o_j,$$

so lassen sich (voneinander unabhängige) lokale Graphen konstruieren durch

$$\forall p \in P: \quad G^p = (O^p, R^p).$$

Dabei ist es wichtig festzustellen, daß bislang über den Zusammenhang zwischen den lokalen Referenzrelationen R^p und der globalen Referenzrelation $\tilde{R}$ keine Aussage getroffen wurde. Dieser Zusammenhang ist erst für die tatsächliche Umsetzung eines verteilten Graphen auf dem Parallelrechner wichtig und bleibt hier deshalb explizit unerwähnt.

In Abb. 2.4 ist ein einfaches Beispiel zweier lokaler Graphen dargestellt. Die Prozessormenge P enthält zwei Prozessornummern $\{A, B\}$; die Menge aller lokalen Objekte $O = \{\, o_1, o_2, o_3, o_4, o_5, o_6 \,\}$ wird durch die `proc`-Zuordnung in zwei lokale Objektmengen zerlegt:

$$O^A = \{\, o_2, o_5, o_6 \,\}$$
$$O^B = \{\, o_1, o_3, o_4 \,\}$$

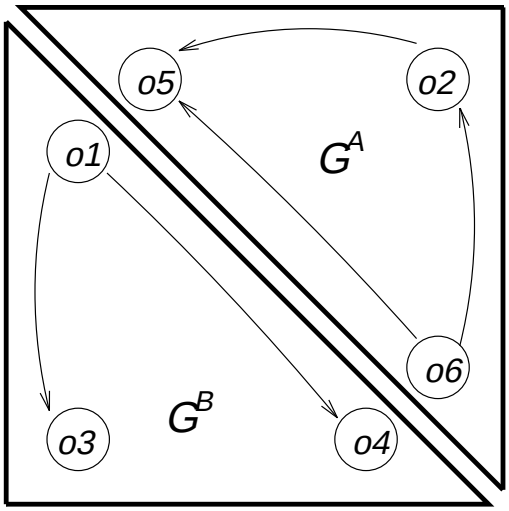


Abbildung 2.4: *Einfaches Beispiel zweier lokaler Graphen.*

Als wichtige Eigenschaft der Zerlegung in lokale Graphen ist anzumerken, daß Referenzen zwischen verschiedenen lokalen Graphen (d.h. über Prozessorgrenzen) durch Definition 9 explizit ausgeschlossen sind; die Zerlegung war im Beispiel nur deshalb möglich, da kein Objekt aus O^A eines der Objekte aus O^B referenziert und umgekehrt.

2.3.3 Verteilte Sicht der Datenstruktur

Die einzelnen lokalen Graphen sind *per definitionem* unzusammenhängend. Um aus diesen nun eine zusammenhängende, verteilte Datenstruktur zu konstruieren, sind zwei Modelle denkbar. Diese rühren von zwei Möglichkeiten zur exakten Definition der Referenzrelationen $\cdot \overset{\text{ref}}{\Leftrightarrow} \cdot$ her.

Modell L: Durch die zusätzliche Restriktion

$$\forall o_i, o_j \in O : o_i \overset{\text{ref}}{\Leftrightarrow} o_j \Rightarrow \mathsf{proc}(o_i) = \mathsf{proc}(o_j)$$

werden Referenzen über Adreßraumgrenzen verboten. Dieses Verbot erlaubt die effiziente Implementierung von indirekter Adressierung mittels hardwarenaher *pointer* (analog zum sequentiellen Programm).

Modell G: Die obige Restriktion wird nicht erhoben, daher sind Referenzen möglich, für die

$$\exists o_i, o_j \in O : o_i \overset{\text{ref}}{\Leftrightarrow} o_j \wedge \mathsf{proc}(o_i) \neq \mathsf{proc}(o_j)$$

gilt. Dies ist äquivalent zur Implementierung eines globalen Adreßraums.

Im DDD-Konzept wird *Modell L* benutzt; daher sind die Möglichkeiten zur Speicheradressierung beschränkt. Ohne Unterstützung durch geeignete Hardware scheint eine portable, aber gleichzeitig effiziente Implementierung der Funktionalität von Modell G nicht möglich. Die wesentliche

Schwierigkeit bei einer Umsetzung von Modell G in Form von Hardware oder Software ist, daß die Lokalität der Referenzen nicht berücksichtigt (d.h. nicht in Skalierbarkeit und Effizienz umgesetzt) werden kann. In einer typischen graphbasierten Anwendung verweisen Referenzen höchstens auf Adreßräume benachbarter Prozessoren; bei einer allgemeinen Bereitstellung von Modell G (also von Zeigern auf beliebige Speicherstellen im globalen Adreßraum, sog. *foreign pointer* [84]) kann jedoch keinerlei Lokalitätsinformation ausgenutzt werden. Jede Referenz kann überallhin zeigen. Dies führt zu Systemen mit Problemen bei der Erhaltung der Konsistenz, mangelnder Skalierbarkeit und erhöhtem Speicherbedarf (siehe auch Abschnitt 3.4.7). Einen praktischen Kompromiß könnte die Erweiterung der grundlegenden (und effizienten) Funktionalität von Modell L durch die komfortablere, aber ineffiziente Adressierungstechnik von Modell G darstellen.

Da sich durch die Einschränkung aus Modell L die Verkopplung der lokalen Graphen durch Referenzen verbietet, muß diese Verkopplung *auf Objektebene* stattfinden. Dies soll im folgenden formal konstruiert werden.

Die Abbildung $\mathsf{id} : O \mapsto I$ ordne jedem Objekt einen Index aus der beliebigen Indexmenge I zu. Lokale Objekte können somit zu Gruppen zusammengefaßt werden, von denen jede einen eindeutigen id-Index erhält. Weiterhin wird durch id die Abbildung $\mathsf{dobj} : I \mapsto \mathbf{P}(O)$ induziert gemäß

$$\mathsf{dobj}(i \in I) := \{o \mid \mathsf{id}(o) = i\},$$

wobei $\mathbf{P}(O)$ die Potenzmenge der Objekte darstellt. Jedem Index $i \in I$ wird also durch dobj eineindeutig eine Menge von Objekten (als Teilmenge von O) zugeordnet, die im folgenden als *verteiltes Objekt* betrachtet werden soll.

DEFINITION 10 (VERTEILTE OBJEKTE)
Zu jedem Index $i \in I$ wird ein verteiltes Objekt $\hat{o}_i \in \mathbf{P}(O)$ definiert durch

$$\forall i \in I : \quad \hat{o}_i := \mathsf{dobj}(i).$$

$\hat{O}$ sei die Menge aller verteilten Objekte ($\hat{O} \subset \mathbf{P}(O)$).

Ein verteiltes Objekt ist also eine Menge von (lokalen) Objekten, die auf verschiedene Prozessoren verteilt sein können. Üblicherweise werden die `type`-Strukturbeschreibungen innerhalb eines verteilten Objekts übereinstimmen, so daß sich das verteilte Objekt als Einheit von auf die Prozessoren verteilten *Objektkopien* verstehen läßt.

DEFINITION 11 (VERTEILTE REFERENZRELATION, VERTEILTER GRAPH)
Definiert man die verteilte Referenzrelation $\hat{R}$ durch

$$(\hat{o}_i, \hat{o}_j) \in \hat{R} \;\Leftrightarrow\; \exists o_k \in \hat{o}_i : \exists o_l \in \hat{o}_j : \exists p \in P : (o_k, o_l) \in R^p,$$

so läßt sich der verteilte Graph $\hat{G}$ einfach durch

$$\hat{G} = (\hat{O}, \hat{R})$$

aus den verteilten Objekten $\hat{O}$ (Knoten) und der verteilten Referenzrelation $\hat{R}$ (Kanten) konstruieren.

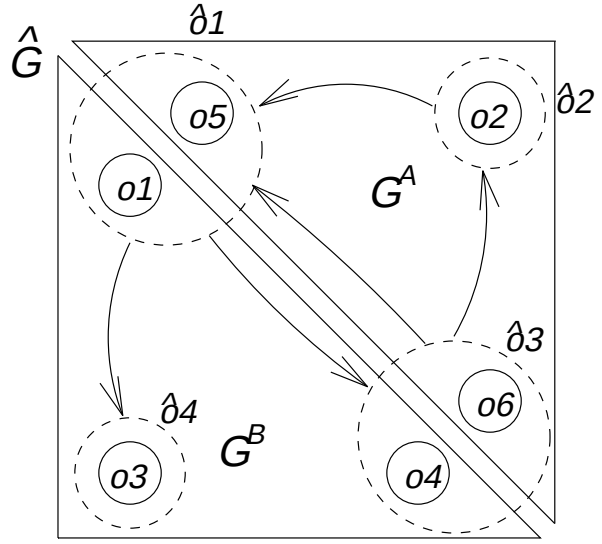


Abbildung 2.5: *Beispiel eines verteilten Graphen. Die Pfeile stellen die verteilte Referenzrelation dar und sind daher nicht den Prozessoren zugeordnet.*

Unter Modell L entsteht der verteilte Graph also durch Überlagerung aller lokalen Graphen: der verteilte Graph muß nun geeignet konstruiert werden, um für eine gegebene Datenstruktur die Modell L-Einschränkung zu erfüllen.

Abb. 2.5 zeigt als Beispiel einen verteilten Graphen $\hat{G}$ durch Überlagerung der beiden lokalen Graphen G^A und G^B aus Abb. 2.4. Die verteilten Objekte (gestrichelte Kreise) werden wie folgt konstruiert:

$$\hat{o}_1 = \{\, o_1, o_5 \,\}, \quad \hat{o}_2 = \{\, o_2 \,\}, \quad \hat{o}_3 = \{\, o_4, o_6 \,\}, \quad \hat{o}_4 = \{\, o_3 \,\}$$

Fordert man für eine gegebene verteilte Referenzrelation, daß jede Referenz auf allen Prozessoren repräsentiert wird, die beide beteiligten Objekte (in Kopie) speichern, so erhält man *konsistente* lokale Referenzrelationen.

DEFINITION 12 (KONSISTENTE LOKALE REFERENZRELATIONEN)
Durch eine gegebene verteilte Referenzrelation $\hat{R}$ wird eine konsistente lokale Referenzrelation R^p_{cons} *zu Prozessor $p \in P$ induziert gemäß:*

$$\forall(\hat{o}_i, \hat{o}_j) \in \hat{R} \quad : \quad \forall o_k \in \hat{o}_i : \forall o_l \in \hat{o}_j : \mathsf{proc}(o_k) = p \wedge \mathsf{proc}(o_l) = p \Rightarrow (o_k, o_l) \in R^p_{\text{cons}}.$$

Es gilt also stets $R^p \subseteq R^p_{\text{cons}}$, aber nicht unbedingt $R^p = R^p_{\text{cons}}$.

2.3.4 Parallelisierung/Verteilung

Bislang besteht keine formale Beziehung zwischen einem (gegebenen) globalen Graph $\tilde{G}$ und etwaigen verteilten Graphen $\hat{G}$. Auf der Basis des bisherigen formalen Modells läßt sich jedoch eine

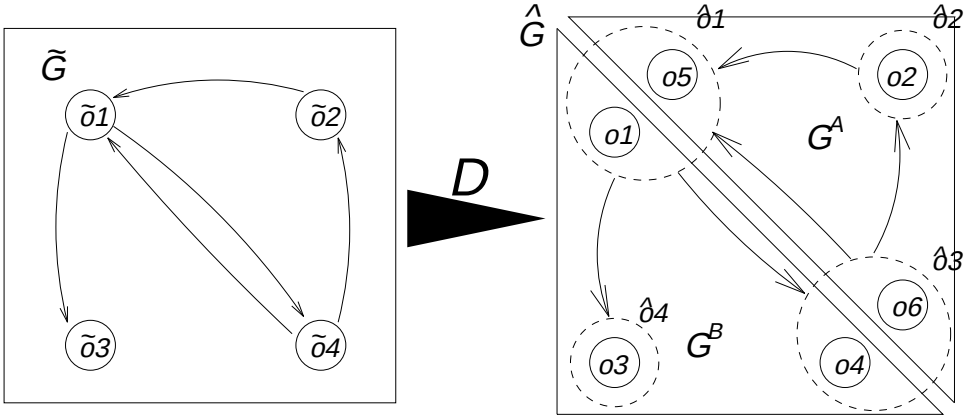


Abbildung 2.6: *Parallelisierung/Verteilung im Beispiel.*

konsistente Verteilung D (*distribution*) einfach als *Homomorphismus* $D : \tilde{O} \mapsto \hat{O}$ bezüglich der Relationen $\tilde{R}$ bzw. $\hat{R}$ definieren.

Die Homomorphismuseigenschaft

$$(\tilde{o}_1, \tilde{o}_2) \in \tilde{R} \quad \Rightarrow \quad (D(\tilde{o}_1), D(\tilde{o}_2)) \in \hat{R}$$

garantiert dabei das konsistente Ineinandergreifen der beteiligten Referenzrelationen. Wenn also im sequentiellen Programmablauf ein Objekt $\tilde{o}_1$ das Objekt $\tilde{o}_2$ referenziert, so muß es im parallelen Programmablauf einen Prozessor $p \in P$ und zwei lokale Objekte $o_1 \in D(\tilde{o}_1)$ bzw. $o_2 \in D(\tilde{o}_2)$ geben mit $\mathsf{proc}(o_1) = p$ und $\mathsf{proc}(o_2) = p$, so daß Prozessor p die Referenz lokal speichert (also $(o_1, o_2) \in R^p$).

Der Begriff der Verteilung D bezeichnet dabei keinen konkreten *Lastverteilungsvorgang*, sondern lediglich ein *formales Kriterium*, ob ein gegebener verteilter Graph eine korrekte Umsetzung des gewünschten globalen Graphen darstellt. Insbesondere ist die Existenz einer solchen Verteilung D für einen gegebenen verteilten Graphen $\hat{G}$ keine konstruktive Aussage über die Art und Weise, wie die einzelnen lokalen Graphen G^p hergestellt werden müssen.

Für das in Abb. 2.6 dargestellte Beispiel kann offensichtlich folgender Isomorphismus D angegeben werden, der die Korrektheit des verteilten Graphen $\hat{G}$ bezüglich des globalen (spezifizierten) Graphen $\tilde{G}$ sicherstellt:

$$D(\tilde{o}_1) = \hat{o}_1, \quad D(\tilde{o}_2) = \hat{o}_2, \quad D(\tilde{o}_3) = \hat{o}_4, \quad D(\tilde{o}_4) = \hat{o}_3$$

2.3.5 Vom globalen Graph zu lokalen Graphen

Die formale Konstruktion einer Verteilung D im Zusammenhang mit Def. 11 bewirkt bislang nur, daß jede Referenz aus dem globalen Graph *irgendwie* im verteilten Graph repräsentiert wird. Aus dem Vergleich der Abbildungen 2.4 und 2.6 erkennt man jedoch, daß z.B. die Referenz $(\hat{o}_1, \hat{o}_3) \in \hat{R}$ zwar im lokalen Graph von Prozessor B als $(o_1, o_4) \in R^B$ repräsentiert wird, nicht jedoch auf

Prozessor A, obwohl die lokalen Objekte o_5 und o_6 vorhanden wären. Es gilt also $(o_5, o_6) \notin R^A$ (wohl aber $(o_6, o_5) \in R^A$).

Nach Def. 11 ist *eine* lokale Referenz ausreichend, um eine globale Referenz zu repräsentieren. Im praktischen Einsatz wird man dagegen mit dieser inkonsistenten Speicherung von verteilten Referenzrelationen unzufrieden sein, da sie zu Fehlern und vor allem zur Entkopplung des verteilten Graphen bei häufiger Umverteilung führen kann (durch Verlust von Referenzen).

In solchen Fällen wird man als zusätzliche Bedingung fordern, daß jede verteilte Referenz auf jedem Prozessor gespeichert wird, dem dies möglich ist. Dies bedeutet, daß die lokalen Referenzrelationen *konsistent* im Sinn von Def. 12 sein müssen. Formal wird gefordert:

$$\forall p \in P : \quad R^p \stackrel{!}{=} R^p_{\text{cons}}$$

Das Beispiel aus den Abbildungen dieses Abschnitts wäre also unter dieser Forderung *keine* konsistente Verteilung.

2.3.6 Entwurf von Verteilungen

In der Praxis geht man vom Graphen $\tilde{G}$ aus und versucht, eine verteilte Struktur $\hat{G}$ mit den zugehörigen lokalen Graphen G^p dergestalt zu entwerfen, daß ein geeigneter Homomorphismus D existiert. Die Eigenschaften der Verteilung D wirken dabei konstruktiv für den Entwurf der verteilten Datenstruktur.

Entwurfsbeispiel

Legt man im gängigsten Fall eine bijektive Verteilung (d.h. einen Isomorphismus) zugrunde, so ergibt sich eine eineindeutige Zuordnung zwischen Objekten aus $\tilde{O}$ und solchen aus $\hat{O}$. Der minimale verteilte Graph (im Sinn der Anzahl von lokalen Objekten) läßt sich nun durch Erfüllen der Modell-L-Einschränkung problemlos entwerfen:

1. Bestimme eine echte *Zerlegung* der Datenstruktur $\tilde{G}$ in Teilstrukturen, die den vorhandenen Prozessoren zugeordnet werden.
2. Ordne jedem Objekt aus $\tilde{O}$ genau eine Objektkopie zu, deren `proc`-Zugehörigkeit aus obiger Zerlegung bestimmt wird.
3. Referenzen aus $\tilde{R}$, deren isomorphe Entsprechung aus $\hat{R}$ innerhalb eines Prozessors zu liegen kommt, können sofort als lokale Referenzen ausgedrückt werden.
4. Andere Referenzen sind im Modell L nicht zugelassen und müssen aufgelöst werden, indem auf einem der beteiligten Prozessoren eine neue Objektkopie des Objekts des anderen Prozessors hinzugenommen wird. Diese Objektkopie ermöglicht die lokale Darstellung der bisher globalen Referenz.

Oftmals wird sich aus dem vorgegebenen Algorithmus, der auf der Datenbasis aufsetzt, die Notwendigkeit ergeben, mehr lokale Objekte als im minimalen Fall zu verwenden. Dies kann mehrere Gründe haben:

- Der verteilte Graph repräsentiert zwar die gewünschte globale Referenzrelation, einzelne lokale Objekte werden jedoch weniger Referenzen enthalten als die zugehörigen verteilten Objekte. Für die Praxis heißt dies, daß Zeiger lokal inkonsistent sein können. Durch Hinzunahme zusätzlicher Kopien läßt sich dies vermeiden.
- Da Zugriffe auf nicht-lokale Objekte erhöhte Kommunikationszeit bedeuten, müssen ebenfalls Objektkopien ergänzt werden, um die Laufzeiteffizienz zu erhöhen. Diese Vorgehensweise wird in Kap. 5 in realistischem Anwendungskontext genauer beschrieben.

2.3.7 Erweiterung: zusätzliche Objekt-Eigenschaften

Die meisten Operationen auf der Objektmenge werden nicht auf die Gesamtheit der Objekte angewendet, sondern lediglich auf eine ihrer Teilmengen. Zur Auswahl dieser Teilmengen wird das obige Modell im folgenden um zwei Objekt-Eigenschaften erweitert: *Priorität* und *Attribut*. Dazu wird die bei der Definition des lokalen Objekts (Def. 7 in Abschnitt 2.3.2) eingeführte Komponente prop geeignet festgelegt:

$$\mathsf{prop}(o) = (\mathsf{prio}, \mathsf{attr})$$

Erweiterung: Priorität

Über die Objekt-Eigenschaft prio kann jedem lokalen Objekt o eine Priorität $\mathsf{prio}(o) \in \Pi \subset \mathbb{N}_0$ zugewiesen werden. Die Priorität kann dazu dienen, Konsistenz-Protokolle auf Anwendungsebene zu definieren; beispielsweise kann eine bestimmte Objektkopie jedes verteilten Objekts ausgezeichnet und bei entsprechenden Operationen bevorzugt behandelt werden (siehe Abschnitt 2.3.8).

Erweiterung: Attribut

Über die Objekt-Eigenschaft attr kann jedem verteilten Objekt $\hat{o}$ ein Attribut $\mathsf{attr}(\hat{o}) \in A$ zugewiesen werden, wobei die gesamte Objektmenge durch A in Teilmengen (Äquivalenzklassen) aufgespalten wird. Die Attribut-Eigenschaft dient dazu, verteilte Objekte in bezug auf die anzuwendenden Algorithmen zu unterscheiden; beispielsweise können einzelne Elemente ihrer jeweiligen Ebene in einer Multilevelhierarchie zugeordnet und dadurch getrennt behandelt werden.

2.3.8 Datenkonsistenz im DDD-Konzept

Die Erzeugung eines verteilten Graphen im Sinne der vorausgehenden Abschnitte impliziert die Existenz von redundanten Objektkopien, über welche die einzelnen lokalen Graphen aneinandergekoppelt werden. Die `GData`-Elemente dieser Objektkopien können als kleine Speicherabschnitte

betrachtet werden, die zwar verteilt sind, aber doch gleiche Daten enthalten sollten: sie sind also kleine *distributed-shared-memory*-Abschnitte.

Ein iteratives numerisches Verfahren, welches auf dem verteilten Graph aufsetzt, wird nun während jeder Iteration einzelne Daten jedes Objekts neu berechnen, was zur Inkonsistenz der zusammengehörenden Objektkopien und damit der verteilten Objekte führt. Die Effizienz des gesamten Verfahrens setzt sich dabei sowohl aus der *parallelen Effizienz* als auch der *numerischen Effizienz* zusammen [15, 126], wobei erstere durch erhöhte Prozessorsynchronisation bzw. -kommunikation, letztere durch Änderung des numerischen Algorithmus aufgrund der Parallelisierung bzw. den Verzicht auf konsistente Daten in den lokalen Objekten gemindert wird (soweit dies numerisch überhaupt sinnvoll ist). Konsistenzerhaltung erfordert jedoch gerade Kommunikation, was einen Kompromiß notwendig macht; dessen Parameter werden durch die eingesetzte Parallelrechner-Hardware einerseits und durch die Anforderungen des numerischen Verfahrens andererseits bestimmt. Die Einstellung des gewünschten Konsistenzgrades über ein entsprechendes Protokoll zur Synchronisation und Kommunikation muß deshalb dem Anwendungsprogramm obliegen.

Überträgt man nun dieselbe Situation auf eine Implementierung für Parallelrechner mit gemeinsamem Speicher (d.h. Multiprozessoren mit *shared-memory* oder skalierbare Architekturen auf *distributed-shared-memory*-Basis, kurz DSM [32], siehe auch Abschnitte 1.2 und 1.6), so lassen sich die dort etablierte Theorie und Terminologie der Datenkonsistenzmodelle auch hier anwenden.

Welche Konsistenzmodelle bietet die existierende Literatur also an? Hwang [74, S. 248ff] definiert das *Speichermodell* als das Verhalten des gemeinsamen Speichers aus der Sicht der Prozessoren. Als grundlegende Speicherprimitive gelten dabei Lese-/Schreiboperationen (*load/store*) und Synchronisationsoperationen (z.B. *swap*, atomares *load-store* oder *conditional store*). Bei gegebenen Instruktionsreihenfolgen des Multiprozessors lassen sich alle erlaubten Reihenfolgen der Speicherprimitive als partielle Ordnung definieren (sog. *ordering of events* [47]). Jedes Speichermodell bedeutet einen Kompromiß zwischen möglichst minimalen Einschränkungen des Anwendungsprogramms (bei Modellen mit strengen Konsistenzanforderungen) und einer möglichst effizienten Implementierbarkeit des Prozessor/Speicher-Gesamtsystems (bei Modellen mit niedrigen Konsistenzanforderungen).

Beim restriktivsten Modell, der *sequentiellen Konsistenz* (*sequential consistency* [92]), entspricht die Abarbeitungsreihenfolge der Speicherprimitive (sowohl für Daten als auch für Instruktionen) genau dem Programmablauf, es liegt also eine strenge Ordnung (*strong ordering*) vor. Dieses Modell bietet dem Benutzer zwar höchste Garantien in bezug auf Einhaltung programmierter Speicherzugriffsreihenfolgen, kann jedoch nicht effizient *und* skalierbar implementiert werden. Als Ausweg bieten sich schwächere Ordnungen der Speicherzugriffe an, die sich effizienter (aber auch komplexer) implementieren lassen. Als Beispiel seien hier *weak consistency* (strenge Ordnung nur der Synchronisationsoperationen [47]), *processor consistency* (strenge Ordnung der Schreiboperationen jedes Prozessors) und *release consistency* (Kombination aus *weak* und *processor consistency* [74, S. 487f]) genannt.

Im DDD-Konzept gelten nun folgende Randbedingungen:

- *Sequentielle Konsistenz* des Speichermodells ist für eine Vielzahl von iterativen numerischen Verfahren nicht notwendig; stattdessen bieten sich natürliche Synchronisationspunkte jeweils nach einer vollständig durchgeführten Iteration an.

- Die Hardwaregrundlage des Gesamtsystems sind Parallelrechner mit physikalisch verteiltem Speicher; wegen der von der Übertragungsmenge unabhängigen Aufsetzzeit der Interprozessorkommunikation ist die Anzahl der versendeten Nachrichten zu minimieren.
- Auf Anwendungsprogrammebene sollen alle Informationen über Objektkopien oder Kommunikationsverwaltung hinter der DDD-Schnittstelle verborgen sein; es sollten lediglich einfache, aber mächtige Grundoperationen zur Definition von Konsistenzprotokollen zur Verfügung stehen.

Um den obigen Bedingungen zu genügen, wurde ein flexibles, größtenteils vom Benutzer gesteuertes Konsistenzmodell gewählt, das den Bedürfnissen der jeweiligen (numerischen) Anwendung angepaßt werden kann (*lean consistency*, [20]). Dabei werden gemeinsame Daten (d.h. lokale Objekte) nicht automatisch synchronisiert; vom Anwendungsprogramm aus können jedoch Kommunikationen an geeigneten (wenigen) Synchronisationspunkten angestoßen werden. Die dazu nötigen Abstraktionen sind:

1. Durch die *Priorität* des lokalen Objekts werden eine oder mehrere Kopien ausgezeichnet (siehe Abschnitt 2.3.7).
2. *Interfaces* sind Mengen von verteilten Objekten, für die *gemeinsame* Synchronisationen durchgeführt werden können; dabei sind keine globalen, sondern nur lokale Synchronisationen bezüglich des Prozessor-Nachbarschaftsgraphs nötig (Details siehe Abschnitt 2.4.2).
3. *gather/scatter-Funktionen* definieren diejenigen Komponenten jedes lokalen Objekts, die der Synchronisation unterworfen werden.

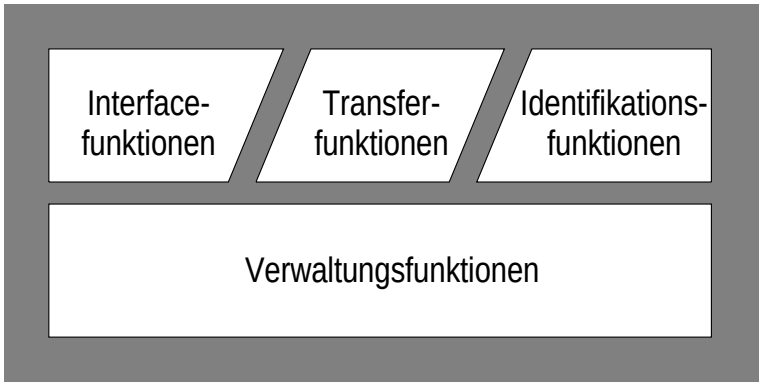


Abbildung 2.7: *Überblick zur Funktionalität von DDD.*

2.4 Funktionalität der DDD-Bibliothek

Das im vorigen Abschnitt 2.3 konstruierte Modell einer verteilten Datenstruktur soll nun in ein paralleles Programmiermodell eingebettet werden; die DDD-Bibliothek stellt eine adäquate Implementierung dieses Programmiermodells dar. Die zugehörige Gliederung der Schnittstelle ist in Abb. 2.7 dargestellt und soll im folgenden kurz umrissen werden.

Die in der DDD-Bibliothek realisierten Funktionen lassen sich vier Teilbereichen zuordnen:

Verwaltungsfunktionen: Der *TypeManager* dient zur Spezifikation der verteilten Anwendungsdatentypen; dies geschieht zur Laufzeit durch Deklaration bzw. Definition der `type`-Eigenschaft der lokalen Objekte. Damit können also dynamisch neue DDD-Typen erzeugt werden. Der *ObjectManager* stellt Funktionen zum Erzeugen und Löschen von lokalen Objekten sowie zur Manipulation der Objekteigenschaften *Priorität* und *Attribut* bereit.

Interfacefunktionen: Interfaces dienen zur Kommunikation auf statischen Datentopologien, wie sie z.B. zum Update von numerischen Daten an Gebietsrändern bei *Domain-Decomposition*-Verfahren nötig ist; in diesem Fall ist das Interface die Gesamtheit aller überlappenden Gebietsränder. Ein Interface wird zu Laufzeitbeginn als $X = (T_X, A, B)$ definiert, wobei $T_X \subseteq T$ eine Teilmenge aller DDD-Typen darstellt und A und B Mengen von Prioritäten zur Auswahl von Interfaceobjekten sind. Jedes einmal definierte Interface kann jederzeit zur Kommunikation verwendet werden und wird während aller topologischen Änderungen der verteilten Datenstruktur konsistent gehalten.

Transferfunktionen: Dieser Teilbereich enthält Funktionen zur dynamischen Veränderung der verteilten Datentopologie zur Laufzeit. Dies beinhaltet z.B. Operationen zur Erzeugung von lokalen Objektkopien auf anderen Prozessoren. Jeder Transferprozeß läuft optimiert (im Sinn der benötigten Anzahl von Nachrichten) ab; sämtliche Interfaces werden trotz topologischer Veränderung konsistent gehalten.

Identifikationsfunktionen: Ein Weg, eine verteilte Datenstruktur zu erhalten, ist die Erzeugung lokaler Objekte auf den verschiedenen Prozessoren, die nachträglich zu verteilten Objekten

zusammengefügt werden (*identification*). Alle Mechanismen zur Identifikation werden (konform zum formalen Modell im vorigen Abschnitt) in diesem Modul zur Verfügung gestellt. *Identifikationstupel* beliebiger Form und Größe dienen dazu, um (lokale) Objekte zu verteilten Objekten zu vereinen. Der Identifikationsprozeß läuft wie der Transferprozeß optimiert ab.

Die Verwaltungsfunktionen von DDD (also TypeManager und ObjectManager) bilden die Grundlage der DDD-Funktionalität und sind deshalb in Abb. 2.7 in der unteren Hälfte dargestellt. Die Funktionen aus den anderen Teilbereichen (*Interfaces, Transfer, Identifikation*) lassen sich sinnvoll nur im Zusammenhang mit den Verwaltungsfunktionen nutzen, sind aber ansonsten voneinander unabhängig.

Die DDD-Programmbibliothek wurde in ANSI C implementiert und ist somit auf einer Vielzahl von Parallelrechnerarchitekturen lauffähig; alle Funktionen sind im jeweils aktuellen, versionsabhängigen *DDD Reference Manual* [19] detailliert beschrieben. Die wichtigsten Aspekte zu Zweck, Kontext und Benutzung der Funktionen sollen jedoch im verbleibenden Teil dieses Abschnitts dargelegt werden, nach den obigen Teilbereichen gegliedert.

2.4.1 Verwaltungsfunktionen

Funktionalität des TypeManagers

Die zu parallelisierende Anwendung verwendet im allgemeinen Objekte verschiedener Datentypen, z.B. Knoten und Elemente zur Repräsentation der Diskretisierung eines Finite-Elemente-Verfahrens. Die Funktionen der DDD-Bibliothek kennen jedoch zunächst nur den einheitlichen Datentyp *lokales Objekt*, die Abbildung der Anwendungstypen auf die DDD-Schnittstelle geschieht erst zur Laufzeit.

Der DDD-*TypeManager* stellt eine Schnittstelle zur Verfügung, die es dem Benutzer erlaubt, dynamisch alle Informationen über die zu verteilenden Datentypen seines Programms zu spezifieren, die von der DDD-Bibliothek benötigt werden. Dies sind hauptsächlich folgende:

Struktur: Welche Komponenten hat der Anwendungsdatentyp (Position, Größe)?

Globale/lokale Daten: Welche Komponenten der Anwendungsdatentypen sind global, d.h. auf allen Prozessoren gültig? Welche sind dagegen nur von lokaler Bedeutung?

Referenzen: Welche Komponenten der Anwendungsdatentypen enthalten Verweise (d.h. Referenzen oder *pointer*) auf andere (verteilte) Datentypen? Welchen Typ haben diese referenzierten Daten?

Die Strukturkomponenten der Anwendungsdatentypen müssen also gerade auf die in Abschnitt 2.3.2 beschriebene Elementtypmenge E abgebildet werden. Dies geschieht in zwei Stufen: Zunächst werden die benötigten Datentypen *deklariert* (**DDD_TypeDeclare()**); mit der dabei erzeugten, eindeutigen Typ-ID (DDD_TYPE) wird die genaue Struktur *definiert* (**DDD_TypeDefine()**).

Um der Möglichkeit der hierarchischen Definition von Datentypen (z.B. verschachtelte *struct*-Definitionen in ANSI C, Vererbung und Klassenhierarchie in C++) Rechnung zu tragen, kann auch

die Definition der zu verteilenden Datentypen mit Hilfe des TypeManagers hierarchisch erfolgen (vgl. S. 41). Durch die Orientierung an den Gegebenheiten der Zielsprachen besteht für zukünftige DDD-Versionen die Möglichkeit, die TypeManager-Funktionalität in einen Präprozessor/Compiler zu integrieren und somit einen Schritt in Richtung der (semi-)automatischen Parallelisierung vorzubereiten.

Bereits bei Definition der DDD-Typen werden alle vorkommenden lokalen Objekte in die zwei Klassen *DDD-Objekte* und *registrierte Datenobjekte* eingeteilt. Diese unterscheiden sich sowohl im zusätzlichen Speicherbedarf für die Parallelisierung, als auch in ihrer Referenzierbarkeit im Sinn von Def. 9 (lokale Referenzrelationen). Die genauen Eigenschaften dieser beiden Objektklassen sind:

DDD-Objekte: diese lokalen Objekte werden mit einer zusätzlichen Komponente `DDD_HEADER` ausgestattet, die alle Verwaltungsinformationen des DDD-Laufzeitsystems enthält. Jedes lokale DDD-Objekt enthält Informationen über sein verteiltes Objekt und eine global eindeutige Identifikationsnummer und ist damit im Sinn von Definition 9 referenzierbar.

Registrierte Datenobjekte: der Typ dieser lokalen Objekte wird zwar ebenfalls beim TypeManager registriert, die Objekte selbst enthalten jedoch keine Komponente `DDD_HEADER`. Datenobjekte speichern somit keinerlei Information über ihr verteiltes Objekt sowie ihre Identifikationsnummer; daher sind sie zwar im Sinn von Definition 9 nicht referenzierbar, können aber selbst DDD-Objekte referenzieren. Damit erfordern registrierte Datenobjekte keinen zusätzlichen Speicheraufwand. Bei sehr kleinen Objekten (z.B. zur Implementierung dünnbesetzter Matrizen, vgl. Abschnitt 5.3.2), für welche die weitere Komponente einen nicht zu vernachlässigenden Zusatzaufwand bedeuten würde, muß diese Einschränkung in Kauf genommen werden.

Die Zuordnung der Anwendungsdatentypen zu diesen beiden Objektklassen kann in hohem Maß entscheidend für die Effizienz eines parallelen Programms sein; verwendet man nur DDD-Objekte, so ist die Verteilung der Datenbasis zwar auf jeden Fall durchführbar, der zusätzlich erforderliche Speicheraufwand kann jedoch so groß sein, daß die Anwendung des resultierenden parallelen Programms auf größere Probleme sinnlos wird. Datenobjekte schaffen hier Abhilfe. In Abschnitt 5.1.2 soll diese Problematik noch einmal genauer betrachtet werden, um dem Parallelisierer die notwendige Hilfestellung zu geben.

Funktionalität des ObjectManagers

Sobald mit dem TypeManager gültige DDD-Typen vereinbart (d.h. deklariert und definiert) wurden, können lokale Objekte dieser Typen erzeugt werden. Da das Erzeugen bzw. Löschen lokaler Objekte sowohl der zu parallelisierenden Anwendung als auch den Funktionen der DDD-Bibliothek selbst möglich sein muß, wurden mit dem DDD-*ObjectManager* drei Schnittstellen auf verschiedenen Abstraktionsstufen definiert (siehe Abb. 2.8):

Speicherschnittstelle: Es wird je eine Funktion zum Allokieren (**DDD_ObjNew()**) bzw. zur Freigabe (**DDD_ObjDelete()**) von Speicherplatz eines lokalen Objekts bereitgestellt. Dabei wird

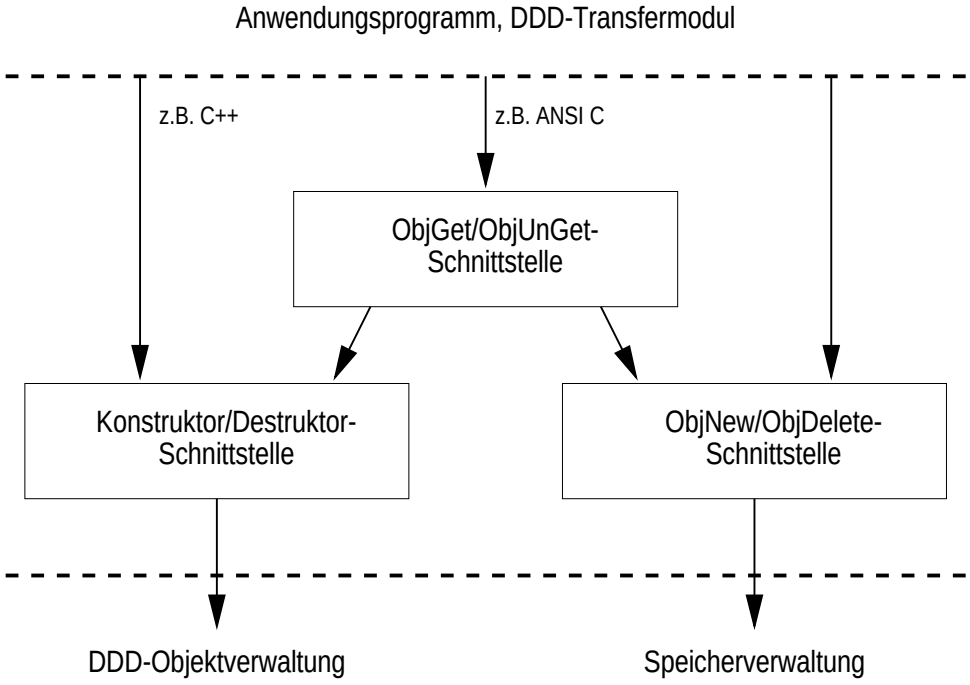


Abbildung 2.8: *Die drei Schnittstellen des DDD-ObjectManagers. Diese werden vom Anwendungsprogramm und vom DDD-Transfermodul benutzt, um neue Objekte zu kreieren bzw. um nicht mehr benötigte freizugeben.*

uninitialisierter Speicher umgesetzt; die Implementierung setzt anwendungsbezogen auf dem bereits beschriebenen *MemMgr*-Konzept (Abschnitt 2.1) auf.

Konstruktor/Destruktor-Schnittstelle: Eine Hauptaufgabe des ObjectManagers ist die An- bzw. Abmeldung von lokalen Objekten beim Laufzeitsystem von DDD. Diese Funktionalität kann durch die Funktionen der Konstruktor/Destruktor-Schnittstelle verwendet werden (Terminologie in Anlehnung an C++). Der *Konstruktor* initialisiert ein neues lokales Objekt, der *Destruktor* meldet es beim DDD-Laufzeitsystem ab.

Get/Unget-Schnittstelle: Diese Schnittstelle stellt die höchste Abstraktionsstufe dar; sie verwendet die Funktionen der beiden zuvor beschriebenen Schnittstellen, um fertige Objekte zu allokieren und beim Laufzeitsystem anzumelden (**DDD_ObjGet()**) bzw. um Objekte abzumelden und anschließend ihren Speicher freizugeben (**DDD_ObjUnGet()**).

2.4.2 Interfacekommunikation

Einmalige Interface-Definition

Jede Kommunikation auf einer bestehenden verteilten Datenstruktur zum Austausch von Informationen zwischen den lokalen Graphen dieser Struktur erfolgt über *Interfaces*. Dabei finden Kommunikationsvorgänge nur *innerhalb* von verteilten Objekten statt; im üblichen Fall also zwischen

verschiedenen lokalen Objektkopien *eines* verteilten Objekts. Quellen und Ziele der Kommunikation werden über die Zuordnung der in Abschnitt 2.3.7 eingeführten Priorität gesteuert.

Die Definition eines Interfaces X benötigt folgende Angaben:

$$X = (T_X, A, B)$$

mit einer Teilmenge der DDD-Typen $T_X \subseteq T$ und Teilmengen der Prioritäten $A, B \subseteq \Pi$. Für eine nichtleere Schnittmenge $A \cap B$ findet eventuell ein Sendevorgang an gleichpriore Kopien statt; gilt $A = B = \Pi$, so wird ein Datenaustausch zwischen allen Kopien durchgeführt.

Für die gesamte Datenstruktur wird automatisch ein *Standardinterface* vorgesehen, das ein Interface auf allen DDD-Typen $t \in T$ mit Prioritätsmengen $A = B = \Pi$ darstellt. Sofern andere Interfaces benötigt werden, müssen diese vor ihrer Verwendung einmalig mit der **DDD_IFDefine()**-Funktion beim DDD-Laufzeitsystem angemeldet werden.

Aus der Menge T_X läßt sich für einen gegebenen verteilten Graphen $\hat{G} = (\hat{O}, \hat{R})$ die Menge der für das Interface X relevanten (lokalen) Objekte O_X angeben:

$$O_X = \{o \in O \mid \mathsf{type}(o) \in T_X\}$$

Dabei soll unter der Objektmenge O_X die Menge all derer lokalen Objekte verstanden werden, die aufgrund ihres DDD-Typs am Interface X beteiligt sein *könnten* (und nicht etwa die Menge aller Objekte, die tatsächlich am Interface X beteiligt sind).

Nach der Definition eines Interfaces X enthält dieses stets alle lokalen Objekte $o \in O(X)$, deren Prioritäten sich in das durch die Mengen A und B gebildete Schema einordnen lassen (Details zur Implementierung siehe Abschnitt 3.2). Die Menge dieser tatsächlich am Interface beteiligten Objekte $O(X)$ ist stets eine Teilmenge von O_X (also $O(X) \subseteq O_X$). Wird die Topologie des verteilten Graphen zur Laufzeit geändert (durch Aufruf von Funktionen aus dem Transfer- bzw. Identifymodul), so bleiben alle zuvor definierten Interfaces konsistent, ohne daß die dazu notwendigen Anpassungen auf Ebene des Anwendungsprogramms sichtbar werden.

Häufige Verwendung von Interfaces: Kommunikationsarten

Die Definition eines Interface geschieht einmalig zur Laufzeit, danach kann das Interface verwendet werden, um das anwendungsorientierte Konsistenzmodell zu realisieren (siehe Abschnitt 2.3.8). Dabei werden jeweils Zeitpunkt der Synchronisation, Kommunikationsart und die zu übertragenden Datenmengen flexibel bestimmt. Zunächst lassen sich aus der Definition der Interfaces drei verschiedene Kommunikationsarten ableiten:

$A \rightharpoonup B$: jedes lokale Objekt

$$o_1 \in O_X \cap O^p \text{ mit } \mathsf{prio}(o_1) \in A$$

schickt Daten an alle Kopien

$$o_2 \in O_X \cap O^q \quad q \neq p \text{ mit } \mathsf{prio}(o_2) \in B.$$

Diese Vorwärtskommunikation wird durch die Funktion **DDD_IFOneway()** angestoßen.

$A \leftharpoondown B$**:** wie oben, aber *A* und *B* vertauscht. Diese Rückwärtskommunikation wird ebenfalls durch die Funktion **DDD_IFOneway()** angestoßen.

$A \rightleftharpoons B$**:** Austausch in beide Richtungen. Diese bidirektionale Kommunikation wird durch die Funktion **DDD_IFExchange()** angestoßen.

Die beiden Funktionen zur Kommunikation auf Interfaces benötigen folgende Parameter:

- ein vorher definiertes Interface X
- die Richtung *dir* der Kommunikation (nur bei unidirektionaler Kommunikation)
- die Größe *size* der zu übertragenden Daten pro *A*/*B*-Paar von lokalen Objekten (in $[\text{byte}]$)
- eine benutzerdefinierte Funktion *gather()* zum Einsammeln der Daten eines lokalen Objekts auf Senderseite
- eine benutzerdefinierte Funktion *scatter()* zum Austeilen (bzw. Bearbeiten) der Daten auf Empfängerseite

Die Angabe zweier Funktionen (*gather()/scatter()*) zum objektweisen Ein-/Auspacken der übertragenen Daten erlaubt neben dem einfachen Konsistenthalten gemeinsamer, aber verteilter Daten auch kompliziertere Operationen, z.B. beliebige Reduktionsoperationen auf den lokalen Kopien eines verteilten Objekts oder Versenden von Daten, die nirgendwo tatsächlich gespeichert, sondern nur nach Bedarf berechnet werden.

Alle Kommunikationen werden durch das Interfacemodul zu möglichst wenigen Nachrichten (*messages*) zusammengefaßt, d.h. von jedem Prozessor *p* zu jedem Prozessor *q* läuft höchstens *eine* Nachricht. Dieses Schema ermöglicht skalierbare Programme, da jeder Prozessor nur mit seiner unmittelbaren Umgebung im Prozessorgraph kommuniziert. Die Aufrufe zur Interfacekommunikation erfolgen blockierend oder nicht-blockierend (siehe dazu Abschnitt 3.2.3) und müssen von allen am jeweiligen Interface beteiligten Prozessoren angestoßen werden.

2.4.3 Transfer von Objekten

Bei der im ersten Kapitel beschriebenen Klasse der dynamischen Anwendungen (z.B. adaptive numerische Verfahren) unterliegt die verteilte Datentopologie zur Laufzeit Veränderungen, die den Transfer von lokalen Objekten zwischen beliebigen Prozessoren bzw. die Erzeugung von verteilten Objekten nötig machen; diese Funktionalität überhaupt und effizient bereitzustellen ist die wichtigste Neuerung des DDD-Konzepts (vgl. andere Ansätze, Abschnitt 1.6). Das Transfermodul beinhaltet dazu u.a. Operationen zur Erzeugung bzw. Vernichtung von Kopien lokaler Objekte auf beliebigen Prozessoren. Im einzelnen sind dies:

- die Funktion **DDD_XferCopyObj()**, die eine Kopie eines gegebenen lokalen Objekts auf einem fremden Prozessor mit einer bestimmten Priorität erzeugt; alle an dem zugehörigen verteilten Objekt beteiligten Prozessoren werden informiert.

- die Funktion **DDD_XferDeleteObj()**, die ein gegebenes lokales Objekt löscht; alle an dem zugehörigen verteilten Objekt beteiligten Prozessoren werden informiert.
- die Funktion **DDD_PrioritySet()**, die die Priorität eines gegebenen lokalen Objekts auf einen neuen Wert setzt; alle an dem zugehörigen verteilten Objekt beteiligten Prozessoren werden über diese Prioritätsänderung informiert.
- die Funktion **DDD_XferAddData()**, die einem zu transferierenden DDD-Objekt zusätzliche Datenobjekte zuordnet (siehe Abschnitt 2.4.1); diese werden mit übertragen und stehen somit der neuen lokalen Objektkopie wieder zur Verfügung.

Diese Operationen können natürlich im MIMD-Stil auf jedem Prozessor unterschiedlich ausgeführt werden; damit im Fall einer großen Nachrichtenaufsetzzeit im zugrundeliegenden *message-passing*-Modell noch effiziente Ausführungszeiten erzielt werden können, wurde als Rahmen ein transaktionsorientiertes Schema gewählt. Jeder Prozessor signalisiert den Beginn einer globalen Transferoperation durch einen Aufruf der Funktion **DDD_XferBegin()**. Danach können obige Operationen in beliebiger Folge aufgerufen werden; diese werden jedoch nicht sofort ausgeführt, sondern auf jedem Prozessor getrennt aufgezeichnet und erst bei globalem Aufruf der Funktion **DDD_XferEnd()** als Ganzes durchgeführt.

Dieses Vorgehen bietet mehrere Vorteile:

- Die sofortige Ausführung einzelner Transferoperationen würden einseitige Kommunikationsprimitive erfordern; dagegen stehen bei gesammelter Ausführung im SPMD-Stil Umfang, Sender und Empfänger aller nötigen Nachrichten vor der Kommunikation fest, es können also (allgemeiner verfügbare) Rendezvousprimitive verwendet werden.
- Aus denselben Gründen ist eine Optimierung der Kommunikation möglich, z.B. kann die Anzahl der benötigten Nachrichten minimiert werden, um die Aufsetzzeiten gering zu halten.
- Bis zur Ausführung des **DDD_XferEnd()**-Kommandos werden lediglich Transferoperationen aufgezeichnet, die Datenstruktur selbst wird noch nicht verändert. Dieser Vorteil des transaktionsorientierten Konzepts erleichtert die Benutzung des Transfermoduls.
- Da die topologische Veränderung des verteilten Graphen im Anwendungsprogramm als atomare Operation erscheint, können auch die im vorangehenden Abschnitt eingeführten Interfaces nach Abschluß des **DDD_XferEnd()**-Kommandos wieder konsistent zur Verfügung gestellt werden. Es werden also den Interfaces dynamisch lokale Objekte hinzugefügt oder aus diesen entfernt.

Betrachtet man die Transferfunktionen als atomare Kommandos an ein DDD-Laufzeitsystem, so sind beliebige Kombinationen dieser Kommandos auf jedes verteilte Objekt anwendbar. Da diese Kommandos in verteilter Weise von verschiedenen Prozessoren während einer Transferphase initiiert werden können, ist es nötig, die Wirkung auf das jeweilige verteilte Objekt genau festzulegen. Die detaillierte Spezifikation der Transferkommandos und ihrer Auswirkungen ist dieser Arbeit beigefügt (Anhang B).

2.4.4 Identifizierung lokaler Objekte

Das Transfermodul bietet die Möglichkeit, verteilte Objekte durch Kopieren lokaler Objekte zu generieren. In der Praxis reicht diese Möglichkeit jedoch oft nicht aus, z.B. wenn ein verteilter Graph (aus Dateien oder über Netzverbindungen) direkt in die verteilten Speicher des Parallelrechners gelesen werden soll. In dieser Situation ist es notwendig, lokale Objekte getrennt voneinander zu erzeugen und erst danach zu einem verteilten Objekt zu verkoppeln. Dies leistet das Identifymodul.

Die Grundoperation der Identifikation ist die Verkopplung von *genau zwei* lokalen Objekten zu einem verteilten Objekt, bei mehr als zwei Objekten müssen alle beteiligten lokalen Objekte jeweils paarweise behandelt werden. Die Voraussetzungen der Identifikation eines Objektpaars sind:

1. Beide beteiligten Prozessoren kennen den jeweiligen Partnerprozessor (Identifikation mit einem unbekannten Partner ist nicht möglich).
2. Pro Objektpaar können sich beide Prozessoren ohne Kommunikation auf einen lokal eindeutigen Identifikator einigen.

Soll beispielsweise der verteilte Graph aus Abb. 2.5 erzeugt werden, so haben die beteiligten Prozessoren nach Erzeugung der Objekte (Prozessor A erzeugt o_2, o_5 und o_6, Prozessor B erzeugt o_1, o_3 und o_4) folgende Identifikationen durchzuführen:

Prozessor A:	Prozessor B:
IdentifyBegin()	IdentifyBegin()
Identify(o_5, B, (*ObenLinks*))	Identify(o_4, A, (*UntenRechts*))
Identify(o_6, B, (*UntenRechts*))	Identify(o_1, A, (*ObenLinks*))
IdentifyEnd()	IdentifyEnd()

Es werden also jeweils das lokale Objekt, der Partnerprozessor und ein Identifikator angegeben, der hier beispielhaft aus geometrischen Beziehungen abgeleitet ist (z.B. *ObenLinks*, *UntenRechts*). Wie im Transfermodul werden die einzelnen Operationen mit einem Begin/End-Paar geklammert, was zu denselben Vorteilen führt.

Ein Identifikator Λ_i (z.B. (*UntenRechts*)) wird als Tupel aus Einzelidentifikatoren $\lambda_{i,j}$ gebildet, um maximale Flexibilität zu erreichen:

$$\Lambda_i = (\lambda_{i,j}), \text{ mit } j = 1 \ldots n_i$$

Hierbei ist n_i die Anzahl der Einzelidentifikatoren ($n_i \geq 1$). Zur Wahl des Einzelidentifikators gibt es folgende Möglichkeiten:

- $\lambda_{i,j}$ ist eine ganze Zahl (Funktion **DDD_IdentifyNumber()**)
- $\lambda_{i,j}$ ist eine Zeichenkette beliebiger Länge (Funktion **DDD_IdentifyString()**)
- $\lambda_{i,j}$ ist ein verteiltes Objekt, das beiden Prozessoren bereits bekannt ist oder das selbst in diesem Identifikationsvorgang identifiziert wird (Funktion **DDD_IdentifyObject()**)

Der letzte Einzelidentifikator beinhaltet dabei die wichtige Möglichkeit, sich bei der Identifikation hierarchisch auf Objekte zu beziehen, die wiederum erst identifiziert werden sollen. Dies ermöglicht es, komplexe Identifikationsvorgänge auf einfache topologische Beziehungen zwischen den Objekten zurückzuführen (Beispiele siehe Abschnitte 5.2.6 und 5.3.3). Diese topologischen Beziehungen müssen während des Identifikationsvorgangs partiell geordnet werden können und dürfen daher keine Zyklen enthalten.

Jedes Identifikationstupel Λ_i wird aus n_i einzelnen Aufrufen der jeweiligen **Identify**-Funktion zusammengesetzt; im Normalfall ist die Reihenfolge dieser n_i Aufrufe relevant und bestimmt den Aufbau des Tupels. Für Identifikationsvorgänge, in denen die Reihenfolge der Einzelidentifikatoren innerhalb eines Tupels nicht relevant sein soll oder sogar aus der lokalen Sicht jedes Prozessors nicht sinnvoll bestimmt werden kann, wird vom Identifymodul optional eine Sortierung durchgeführt; die Tupel werden in dieser Option also als *Mengen* von Einzelidentifikatoren aufgefaßt. Die Gesamtdatenmenge des Identifikationstupels ist nicht kritisch, da die Daten nur zur Sortierung benutzt, aber nicht verschickt werden müssen.

Die Funktionalität des DDD-Identifymoduls und vor allem die hierarchische Identifikation von verteilten Objekten steht in dieser Form in keinem anderen Programmpaket zur Verfügung. Die Verbindung aus einer allgemeinen, funktionalen Schnittstelle und einer effizienten Implementierung hat sich in vielen Anwendungen als sinnvoll erwiesen (z.B. parallele Ein-/Ausgabe, verteilte Gitterverfeinerung; Beispiele siehe Kap. 5).

2.4.5 Benutzerdefinierte Handler-Funktionen

Alle bisher beschriebenen DDD-Module stellen dem Anwendungsprogramm in der Hauptsache Funktionen zur Verfügung, die von diesem aufgerufen werden und Operationen auf der verteilten Datenstruktur durchführen. In einigen Fällen tritt jedoch der umgekehrte Fall ein: das DDD-Laufzeitsystem benötigt zur Durchführung der Operationen Funktionalität aus der Anwendung. Dies wird durch ein Handler-Konzept realisiert: DDD-Handler sind dabei benutzerdefinierte Funktionen (sog. *callback*-Funktionen), die vom Anwendungsprogramm zur Verfügung gestellt werden müssen, um der DDD-Bibliothek gewisse Operationen zu ermöglichen.

Die vorhandenen Handler lassen sich in folgende Kategorien einteilen (für Details siehe [19]):

Speicherverwaltung: Die Schnittstelle zum Speichermanager ist im wesentlichen eine Sammlung von Handlern (siehe Abschnitt 2.1).

Objektverwaltung: Dies sind Handler, die vom ObjektManager zur Initialisierung von Objekten benötigt werden.

Interfacekommunikation: Die im Abschnitt 2.4.2 beschriebenen *gather/scatter*-Funktionen sind Handler, die das Einsammeln bzw. Auspacken der Daten für die Interfacekommunikation übernehmen.

Transferkontrolle: Mit den Handlern dieser Kategorie können Abhängigkeiten von Objekten bezüglich des Transfers ausgedrückt werden; dies betrifft sowohl die Abhängigkeit verschiedener DDD-Objekte als auch die Zuordnung von zusätzlichen Datenobjekten.

Die Handler der Speicherverwaltung werden zur Linkzeit festgelegt; alle anderen Handler können zur Laufzeit beliebig dynamisch umkonfiguriert werden. Dabei gehört zu jedem DDD-Typ $t \in T$ ein Satz von Handlern, die im objektorientierten Sinn als Mitgliedsfunktionen der Klasse *t* gesehen werden können. Wird ein Handler eines DDD-Typs nicht konfiguriert, so wird gegebenenfalls ein Standardhandler (innerhalb der DDD-Bibliothek) aufgerufen. Diese Vorgehensweise entspricht dem objektorientierten Konzept der *virtuellen* Mitgliedsfunktion.

Kapitel 3

Dynamic Distributed Data: Implementierung

Nach der Vorstellung des DDD-Konzepts im vorausgehenden Kapitel sollen nun wichtige Implementierungsaspekte des aktuellen Prototyps umrissen werden. Die interne Struktur der Programmbibliothek spiegelt dabei die Gliederung ihrer Schnittstelle zum Anwendungsprogramm (siehe Abb. 2.7) wider, wobei eine Basisschicht an Datenstrukturen und zugehörigen Operationen verwendet wird, die allen Teilmodulen zugrundeliegt. Diese Schicht wird im ersten Abschnitt dieses Kapitels beschrieben. In den drei folgenden Teilen wird nacheinander auf die DDD-Module Interfaces, Identifikation und Transfer eingegangen. Wichtige Randbedingungen bei der Umsetzung des Konzepts in eine Programmbibliothek sind dabei *Effizienz* und *Portabilität*.

3.1 Basismodul: Datenstrukturen, Typ- und Objektmanager

Das Basismodul beinhaltet als gemeinsame Grundlage aller Funktionen der DDD-Bibliothek wichtige Datenstrukturen sowie die Funktionen von Typ- und Objektmanager (d.h. grundlegende Verwaltungsfunktionen). Die Datenstruktur `DDD_HEADER` faßt die für jedes DDD-Objekt nötige Verwaltungsinformation zusammen, *Couplinglisten* speichern lokale Kopplungsinformationen zur Repräsentation verteilter Objekte. Die globale *Typbeschreibungstabelle* enthält die Definitionen aller zur Laufzeit angemeldeten DDD-Typen. Die *lokale Objekttabelle* ist eine Tabelle von Zeigern auf alle lokalen Objekte eines Prozessors; die zugehörige *lokale Couplingtabelle* ist eine Tabelle von Zeigern auf die Couplinglisten der lokalen Objekte. Die folgenden Abschnitte behandeln obige Datenstrukturen und Tabellen im einzelnen.

3.1.1 Die Datenstruktur `DDD_HEADER`

Die wichtigsten Eigenschaften der von DDD verwalteten Objekte werden in der Datenstruktur `DDD_HEADER` gespeichert. Diese umfaßt sowohl Einträge, die für alle lokalen Kopien eines verteilten Objekts gelten, als auch individuelle Einträge. Tabelle 3.1 gibt einen Überblick.

Tabelle 3.1: *Die Teilstruktur* DDD_HEADER *eines DDD-Objekts.*

Datenstruktur DDD_HEADER			
Datentyp	Name	Bedeutung	Einheitlich für:
DDD_TYPE	*type*	DDD-Typ	verteiltes Objekt
DDD_GID	*gid*	globale ID	verteiltes Objekt
DDD_ATTR	*attr*	Attribut	verteiltes Objekt
DDD_PRIO	*prio*	Priorität	lokales Objekt
int	*index*	Index in lokaler Objekttabelle	lokales Objekt
int	*flags*	Flags, verschieden genutzt	lokales Objekt

Der Parameter *type* beschreibt den DDD-Typ des verteilten Objekts (Abschnitt 2.3.1, Def. 4), wie er beim Erzeugen des Objekts mittels **DDD_ObjGet()** angegeben wurde. Der Datentyp DDD_TYPE wird dabei als Index in eine globale *Typbeschreibungstabelle* aufgefaßt, in welcher der interne Aufbau jedes definierten Objekttyps gespeichert wird.

Der zu jedem verteilten Objekt gehörende Index aus der Indexmenge I ($\mathsf{id} : O \mapsto I$, S. 44) wird als global eindeutige ID ebenfalls in der DDD_HEADER-Struktur gespeichert. Das Verhältnis von verteiltem Objekt und der zugehörigen Menge von lokalen Objekten wird somit direkt in die Implementierung übernommen. Die Erzeugung einer global eindeutigen ID für jedes lokal erzeugte Objekt geschieht durch Konkatenation eines Referenzzählers mit der jeweils lokalen Prozessornummer. Jeder Prozessor verwaltet dazu einen eigenen Referenzzähler, der bei Erzeugung eines neuen lokalen Objekts inkrementiert wird. Diese lokale Art der Erzeugung gewährleistet die Skalierbarkeit des Mechanismus. Die Konkatenation wird maschinennah auf Bitebene durchgeführt; durch die Breite des für die globale ID verwendeten Datentyps wird zwar die theoretische Obergrenze für die Anzahl gleichzeitig existierender verteilter Objekte eingeschränkt, die reale Beschränkung stellt in der Praxis jedoch der auf dem Parallelrechner verfügbare Hauptspeicher dar. Bei extensiver Erzeugung und Vernichtung von Objekten (z.B. im Rahmen einer parallelen, instationären Berechnung) kann es dennoch geschehen, daß einer der Referenzzähler überläuft; dies macht die Neunumerierung der global eindeutigen IDs notwendig, die zu diesem Zeitpunkt gültig sind.

Als zusätzliche Eigenschaft des verteilten Objekts wird als Komponente *attr* das *Attribut* ebenfalls im DDD_HEADER gespeichert. Der DDD_HEADER jedes lokalen Objekts enthält außerdem die Eigenschaft *Priorität* als Komponente *prio* (Abschnitt 2.3.7). Neben einer Komponente *flags*, die von anderen Modulen der DDD-Bibliothek zu verschiedensten Aufgaben genutzt wird, enthält die DDD_HEADER-Struktur schließlich noch den Index in der lokalen Objekt- bzw. Couplingtabelle, welche in Abschnitt 3.1.3 näher beschrieben werden.

Um nun den Funktionen der DDD-Bibliothek stets Zugriff auf die objektbezogenen Verwaltungdaten zu ermöglichen, wird in jedem lokalen Objekt zusätzlich zu allen Komponenten des sequentiellen Programms eine DDD_HEADER-Struktur gespeichert. Diese kann innerhalb des Objekts an eine beliebige Position gesetzt werden, wobei der Offset für jeden DDD-Typ spezifisch in der Typbeschreibungstabelle festgelegt wird (vgl. Tabelle 3.3). Abb. 3.1 verdeutlicht dies und stellt zwei zusätzliche Datentypen vor: DDD_OBJ als Zeiger auf von DDD verwaltete Objekte, DDD_HDR als Zeiger auf deren jeweiligen DDD_HEADER. Das Anwendungsprogramm verwendet ausschließlich

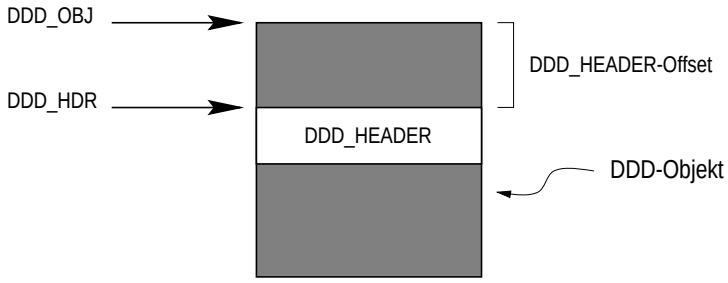


Abbildung 3.1: *Die* DDD_HEADER*-Struktur als Teil eines DDD-Objekts und dazugehörige Datentypen.* DDD_HDR *und* DDD_OBJ *sind Zeiger auf die* DDD_HEADER*-Struktur bzw. auf das DDD-Objekt selbst.

Zeiger vom Typ DDD_OBJ; die Übergabe eines Objekts an die DDD-Bibliothek geschieht mittels des Typs DDD_HDR.

3.1.2 Die Couplingliste

Zur Repräsentation des im vorigen Kapitel vorgestellten verteilten Datenmodells genügt strenggenommen bereits die im DDD-Header jedes lokalen Objekts gespeicherte *gid*-Komponente (Kap. 2). Für den Einsatz in der Praxis muß jedoch eine Zusatzdatenstruktur geschaffen werden, die den effizienten Zugriff auf die Menge der lokalen Objekte eines verteilten Objekts erlaubt. Diese wurde, in etwas verallgemeinerter Form entnommen aus [10], in Form von sogenannten *Couplinglisten* implementiert.

Für jedes lokale Objekt, das in mehr als einer Kopie auf dem Parallelrechner existiert, wird eine Liste von COUPLING-Strukturen gespeichert. Für ein gegebenes verteiltes Objekt $\hat{o}$ speichert also jeder beteiligte Prozessor $p \in P_{\hat{o}}$ mit

$$P_{\hat{o}} = \{\mathsf{proc}(o) \mid o \in \hat{o}\}$$

zum lokalen Objekt $o_{\text{lokal}} \in \hat{o} \cap O^p$ eine Couplingliste $\mathsf{cpl}^p_{\hat{o}}$ als Menge von Prozessor/Prioritäts-Tupeln wie folgt:

$$\mathsf{cpl}^p_{\hat{o}} \;=\; \{(o_{\text{lokal}}, \mathsf{proc}(o), \mathsf{prio}(o)) \mid o \in \hat{o} \wedge \mathsf{proc}(o) \neq p\}$$

Jede der COUPLING-Strukturen $c \in \mathsf{cpl}^p_{\hat{o}}$ repräsentiert dabei eine Kopie des Objekts auf einem anderen Prozessor. Alle Prozessoren $q \notin P_{\hat{o}}$ speichern keine lokalen Objekte aus $\hat{o}$ und daher auch keine zugehörige Couplingliste $\mathsf{cpl}^q_{\hat{o}}$ (formal jedoch: $\mathsf{cpl}^q_{\hat{o}} = \emptyset$). Die *lokale Couplingmenge* cpl^p eines Prozessors p sei schließlich die Vereinigung aller Couplinglisten auf p:

$$\mathsf{cpl}^p = \bigcup_{\hat{o} \in \hat{O}} \mathsf{cpl}^p_{\hat{o}}$$

Die erforderlichen COUPLING-Komponenten sind (Tabelle 3.2): Prozessornummer der Objektkopie (vom Typ DDD_PROC), Priorität der Objektkopie (vom Typ DDD_PRIO) und wiederum diverse

Tabelle 3.2: *Einzelner Eintrag der Couplingliste: die Struktur* `COUPLING`.

Datenstruktur `COUPLING`		
Datentyp	Name	Bedeutung
`DDD_PROC`	*proc*	Partnerprozessor
`DDD_PRIO`	*prio*	Priorität
unsigned char	*flags*	Flags, verschieden genutzt
`DDD_HDR`	*obj*	Zeiger auf o_{lokal}
`COUPLING`*	*next*	Verkettung in Couplingliste

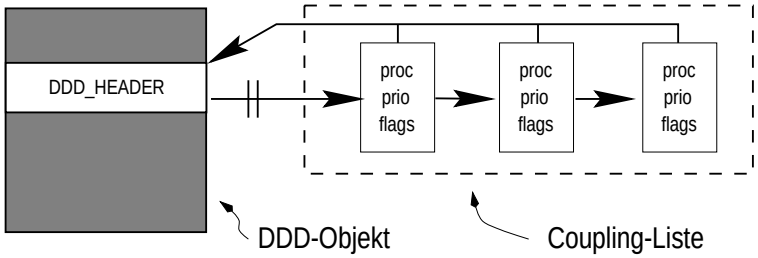


Abbildung 3.2: *Couplingliste eines verteilten Objekts, bestehend aus vier lokalen Objekten. Die Couplingliste enthält drei* `COUPLING`*-Strukturen, das vierte Objekt ist lokal vorhanden und benötigt daher keinen Platzhalter.*

Flags. Die Speicherung erfolgt mittels einer einfach verketteten Liste, deren erstes Element über die lokale Couplingtabelle adressiert wird (Abschnitt 3.1.3). Die Details sind für ein verteiltes Objekt, bestehend aus vier lokalen Kopien, in Abb. 3.2 veranschaulicht, wobei die Adressierung über die Couplingtabelle mit einem Doppelstrich symbolisiert wird. Aus der Abbildung läßt sich ebenfalls ersehen, daß eine Rückverzeigerung von jeder `COUPLING`-Struktur zum zugehörigen lokalen Objekt o_{lokal} existiert, was zur Repräsentation der DDD-Interfaces benötigt wird.

3.1.3 Lokale Objekttabelle, lokale Couplingtabelle

Von vielen Funktionen der DDD-Bibliothek wird der Zugriff auf die lokale Objektmenge O^p jedes Prozessors p benötigt. Daher wird auf jedem Prozessor eine Tabelle von Zeigern auf die lokalen Objekte gespeichert, die in zwei Bereiche unterteilt wird: im ersten Abschnitt werden alle DDD-Objekte mit nichtleeren Couplinglisten referenziert, im zweiten Abschnitt die übrigen lokalen Objekte. Beide Abschnitte sind ansonsten unsortiert; jeweils nach Bedarf kann eine nach beliebigen Kriterien sortierte Kopie dieser *lokalen Objekttabelle* angelegt werden.

Die Couplinglisten der lokalen Objekte aus dem ersten Abschnitt der lokalen Objekttabelle können mittels der *lokalen Couplingtabelle* referenziert werden. Durch den *index*-Eintrag in der Struktur `DDD_HEADER` wird also sowohl die Couplingliste eines lokalen Objekts (für $1 \leq \mathit{index} \leq n_{cpl}$) als auch der Verweis aus der lokalen Objekttabelle erreicht (für $1 \leq \mathit{index} \leq n_{obj}$). Dabei ist $n_{obj} = |O^p|$

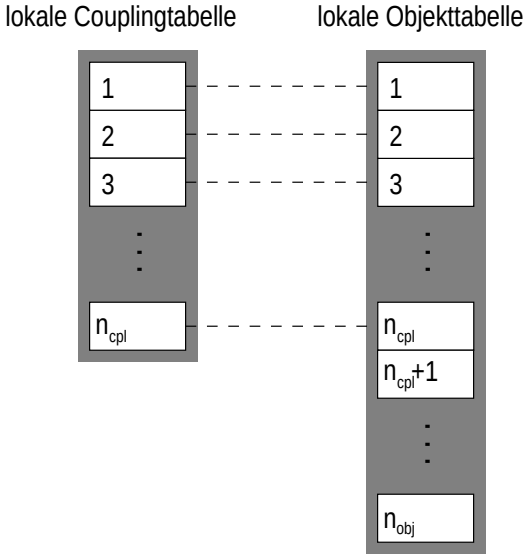


Abbildung 3.3: *Zusammenhang von lokaler Couplingtabelle und lokaler Objekttabelle. Es gilt auf jedem Prozessor stets die Beziehung* $0 \leq n_{cpl} \leq n_{obj} = |O^p|$.

die Anzahl der lokalen Objekte und n_{cpl} die Anzahl der lokalen Objekte mit nichtleerer Couplingliste; Abb. 3.3 soll diesen Zusammenhang verdeutlichen.

Die Operationen *Einfügen* und *Löschen* auf dieser Doppeltabelle werden im Rahmen des ObjectManagers implementiert; die Konsistenz z.B. beim Löschen von Objekten mit $index \leq n_{cpl}$ wird durch geeignete Tauschoperationen auf den Tabellenindizes gewahrt. Der Zugriff auf alle Couplinglisten bzw. alle lokalen Objekte mit Kopien kann über die beschriebene Datenstruktur effizient erfolgen.

3.1.4 Dynamische Typbeschreibung

In Abschnitt 2.4.1 wurde bereits die Funktionalität des TypeManagers beschrieben. Um die gewünschte Möglichkeit der dynamischen Definition von DDD-Typen bereitstellen zu können, stützt sich der TypeManager auf die globale *Typbeschreibungstabelle*, welche auf allen Prozessoren repliziert wird. Tabelle 3.3 zeigt die wesentlichen Komponenten eines Typbeschreibungsdatensatzes.

Zur Ausgabe von Informationen zur Laufzeit erhält jeder DDD-Typ bei seiner Deklaration einen Typnamen *name*. Die Strukturbeschreibung `type` (Definition 7, S. 41) wird in einem Feld *element[ ]* als Tabelle (der Länge *nElements*) von Elementbeschreibungsstrukturen `ELEM_DESC` gespeichert (Tabelle 3.4). Jede dieser Elementbeschreibungstrukturen enthält neben dem Typ des Elements (laut Definition 8, S. 42) dessen Größe und seinen Offset in der Gesamtdatenstruktur. Zusätzlich wird für Elemente vom Typ `ObjPtr` der referenzierte DDD-Typ *reftype* gespeichert, da diese Information während des Objekttransfers benötigt wird.

Während der Erzeugung von lokalen Objekten im Transfermodul müssen lokale Objekte zwischen Nachrichtenpuffer und lokalem Speicher (und umgekehrt) kopiert werden. Diese Operation wird allerdings nur auf den `GData`-Elementen des Objekts durchgeführt (vgl. S. 42); daher wird in der

Tabelle 3.3: *Einzelner Eintrag der Typbeschreibungstabelle.*

Datenstruktur `TYPE_DESC`		
Datentyp	Name	Bedeutung
char	*name[ ]*	Typname (in Klartext)
`ELEM_DESC`	*element[ ]*	Elementtabelle (siehe Abschnitt 2.3.1)
int	*nElements*	Anzahl der Elemente
HandlerPtr	*handler[ ]*	Handlertabelle
int	*size*	Größe eines Objekts (in [byte])
int	*nPointers*	Anzahl von Referenzen
int	*offsetHeader*	`DDD_HEADER`-Offset (in [byte])
unsigned char*	*cmask*	Kopiermaske

Tabelle 3.4: *Die Struktur* `ELEM_DESC` *zur Elementbeschreibung.*

Datenstruktur `ELEM_DESC`		
Datentyp	Name	Bedeutung
int	*offset*	Offset des Elements im Objekt
size_t	*size*	Größe des Elements (in [byte])
int	*type*	Elementtyp $\in E$ (laut Def. 8, S. 42)
`DDD_TYPE`	*reftype*	referenzierter DDD-Typ (für *type*=`ObjPtr`)

Typbeschreibung eine Kopiermaske *cmask* vorgehalten, die schematisch alle `LData`-Elemente ausblendet und so einen effizienten Kopiervorgang ermöglicht (siehe Abb. 3.4). Zusätzliche Informationen wie die Gesamtgröße *size* eines Objekts des gegebenen Typs oder die Anzahl aller gespeicherten Referenzen pro Objekt *nPointers* dienen zur schnellen Berechnung von Nachrichten- und Puffergrößen beim Transfer des Objekts (vgl. Abschnitt 3.4).

Schließlich wird in jeder Typbeschreibungsstruktur eine Tabelle von Funktionen *handler[ ]* gespeichert, welche die für diesen DDD-Typ gültigen DDD-Handler implementieren (vgl. Abschnitt 2.4.5). Somit können Handler zur Laufzeit beliebig konfiguriert und dynamisch umkonfiguriert werden. Steht ein Handler eines bestimmten DDD-Typs zum Aufruf an, wird zunächst mittels des zugehörigen Eintrags der Tabelle *handler[ ]* geprüft, ob eine Funktion konfiguriert wurde. Falls keine Handlerfunktion eingetragen ist, wurde der Tabelleneintrag auf 0 gesetzt; in diesem Fall kann eine Standardfunktion aufgerufen werden.

3.2 Interfaces: Repräsentation, Konstruktion und Benutzung

Das Interfacemodul bietet Kommunikation auf statischen Datentopologien (Abschnitt 2.4.2). Um diese Aufgabe effizient durchführen zu können, benötigt die Implementierung eine geeignete Repräsentation der bisher nur abstrakt definierten DDD-Interfaces; diese soll im folgenden Abschnitt

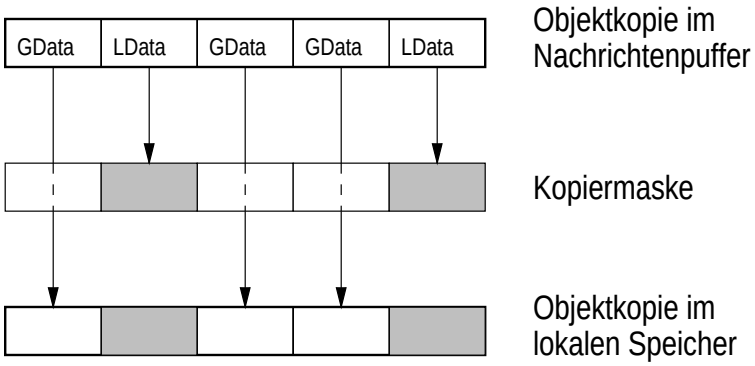


Abbildung 3.4: *Beispiel für den Einsatz der Kopiermaske* cmask. *Die Objektkopie im Nachrichtenpuffer wird in einer effizienten Programmschleife in den lokalen Speicher übertragen, wobei die einzelnen* `GData`*-Komponenten kopiert, die* `LData`*-Komponenten jedoch durch logische Verknüpfung ausgeblendet werden. Damit ist effizientes Kopieren bis auf Bitebene möglich.*

behandelt werden. Die Grundidee der Interface-Implementierung ist, die bereits erläuterten Couplinglisten zu nutzen, um durch gleiche Sortierung auf verschiedenen Prozessoren möglichst viel Information über Objektkopplungen ohne Kommunikation (d.h. implizit) herzuleiten und nur die Nutzdaten über das Interface zu übertragen.

Die folgende Implementierungsbeschreibung der Interfacefunktionalität ist in drei Teile gegliedert: zunächst werden die Datenstrukturen zur Repräsentation der Interfaces beschrieben, danach der dynamische Aufbau dieser Datenstrukturen und schließlich die Benutzung zur eigentlichen Kommunikation auf Interfaces.

3.2.1 Repräsentation der Interfaces

Interfaces als Couplinglisten

Die Repräsentation der Interfaces fußt auf den oben beschriebenen `COUPLING`-Strukturen. Ein Interface $X = (T_X, A, B)$ wird als verteilte Datenstruktur implementiert, wobei jeder Prozessor p eine Datenstruktur X^p speichert. Den Kern dieser lokalen Repräsentation X^p bildet eine sortierte Liste von Zeigern auf `COUPLING`-Strukturen, die aus der lokalen Couplingmenge cpl^p gewonnen wird; dazu wird zunächst die Couplingmenge $\mathsf{cpl}^p_X \subseteq \mathsf{cpl}^p$ bestimmt, die alle `COUPLING`-Strukturen $c := (o_{\text{lokal}}, \mathsf{proc}(o), \mathsf{prio}(o)) \in \mathsf{cpl}^p$ enthält, für die mindestens einer der beiden Fälle

- Fall $A \rightharpoonup B$: $\mathsf{prio}(o_{\text{lokal}}) \in A \wedge \mathsf{prio}(o) \in B$
- Fall $A \leftharpoonup B$: $\mathsf{prio}(o) \in A \wedge \mathsf{prio}(o_{\text{lokal}}) \in B$

erfüllt ist. Die *Couplingrichtung* $\mathsf{dir}(c)$ ergibt sich dann zu

- $\mathsf{dir}(c) =$`DirAB`, falls $(A \rightharpoonup B) \wedge \neg(A \leftharpoonup B)$

- $\mathsf{dir}(c) = \texttt{DirBA}$, falls $(A \leftharpoondown B) \wedge \neg(A \rightharpoonup B)$
- $\mathsf{dir}(c) = \texttt{DirABA}$, falls $(A \rightharpoonup B) \wedge (A \leftharpoondown B)$.

Die Zeigerliste zu X^p wird schließlich aus der Menge cpl_X^p durch Sortierung erzeugt, und zwar gemäß dem Schlüssel

$$1.\ \mathsf{proc}(o), \quad 2.\ \mathsf{dir}(c), \quad 3.\ \mathsf{attr}(\hat{o}), \quad 4.\ \mathsf{id}(\hat{o}) \tag{3.1}$$

Auf das Attribut des zur jeweiligen Couplingliste gehörenden verteilten Objekts $\hat{o}$ sowie auf die globale ID kann jeweils durch die DDD_HEADER-Struktur des lokalen Objekts o_{lokal} zugegriffen werden.

Abb. 3.5 gibt eine vereinfachte Übersicht der Datenstrukturen, die ein DDD-Interface auf einem Prozessor repräsentieren. Im Zentrum dieser Abbildung ist die sortierte Zeigerliste beispielhaft dargestellt; jeder Eintrag (d.h. jede Zeile) der Proc/IFDir/Attr/GID-Tabelle entspricht einem Zeiger auf eine COUPLING-Struktur. Man erkennt zunächst die Sortierung der Liste in Abschnitte gleichen Zielprozessors (Spalte Proc, Einträge $P1$ und $P2$), die weitere Unterteilung je nach Couplingrichtung (Spalte IFDir) und die absteigende Sortierung nach dem jeweiligen Attributswert des verteilten Objekts (Spalte Attr).

Die letzte Spalte GID deutet die feinste Sortierung nach global eindeutiger ID an, wobei im Beispiel o.B.d.A. eine konstante Anzahl von drei COUPLING-Strukturen pro Attributswert gewählt wurde: vom ersten Eintrag der Tabelle an gerechnet enthält das Interface zunächst drei Einträge mit Couplingrichtung DirAB und $\mathsf{attr} = 7$, dann drei Einträge mit $\mathsf{attr} = 2$, danach drei Einträge mit Couplingrichtung DirBA und $\mathsf{attr} = 4$, etc. Im allgemeinen werden die Anzahl verschiedener globaler IDs bei gleichem Attributswert und ebenfalls die Anzahl verschiedener Attributswerte bei gleicher Couplingrichtung von der Beschaffenheit der verteilten Datenbasis abhängen.

Jeder Abschnitt gleichen Zielprozessors q der Gesamtliste X^p soll im folgenden mit $X^{p\to q}$ bezeichnet werden; jedes dieser Teilinterfaces $X^{p\to q}$ wird in der Implementierung durch eine Datenstruktur vom Typ IF_PROC repräsentiert, die jeweils (neben der Prozessornummer q selbst) Anfang und Länge der Abschnitte gleicher Couplingrichtung speichert. Sie bildet damit eine summarische Beschreibung der im zugehörigen Abschnitt der Zeigerliste gespeicherten Daten (linker Teil in Abb. 3.5).

Sämtliche IF_PROC-Strukturen werden linear verkettet und bilden somit wieder die Gesamtliste X^p. Die Abschnitte gleicher Couplingrichtung dir sollen mit $X_{\mathsf{dir}}^{p\to q}$ bezeichnet werden; die sortierte Liste der verteilten Objekte zum Interfaceabschnitt $X_{\mathsf{dir}}^{p\to q}$ sei $\hat{O}(X_{\mathsf{dir}}^{p\to q})$.

Benötigt man keine Unterteilung eines Interfaces nach seinen Attributswerten, so reicht der bisher beschriebene Teil der Datenrepräsentation aus (vgl. Abschnitt 3.2.3). Soll jedoch Kommunikation nur auf Teilmengen eines Interfaces stattfinden (z.B. zur Kommunikation auf *einem* verteilten Gitter eines Mehrgitterverfahrens), so wird die im nächsten Abschnitt beschriebene Datenstruktur genutzt.

Repräsentation der Attribute

Für viele Anwendungsklassen (z.B. Multilevelverfahren) ist es nötig, nur Teile eines Interfaces X, ausgewählt nach der Eigenschaft *Attribut* der verteilten Objekte, ansprechen und verwenden

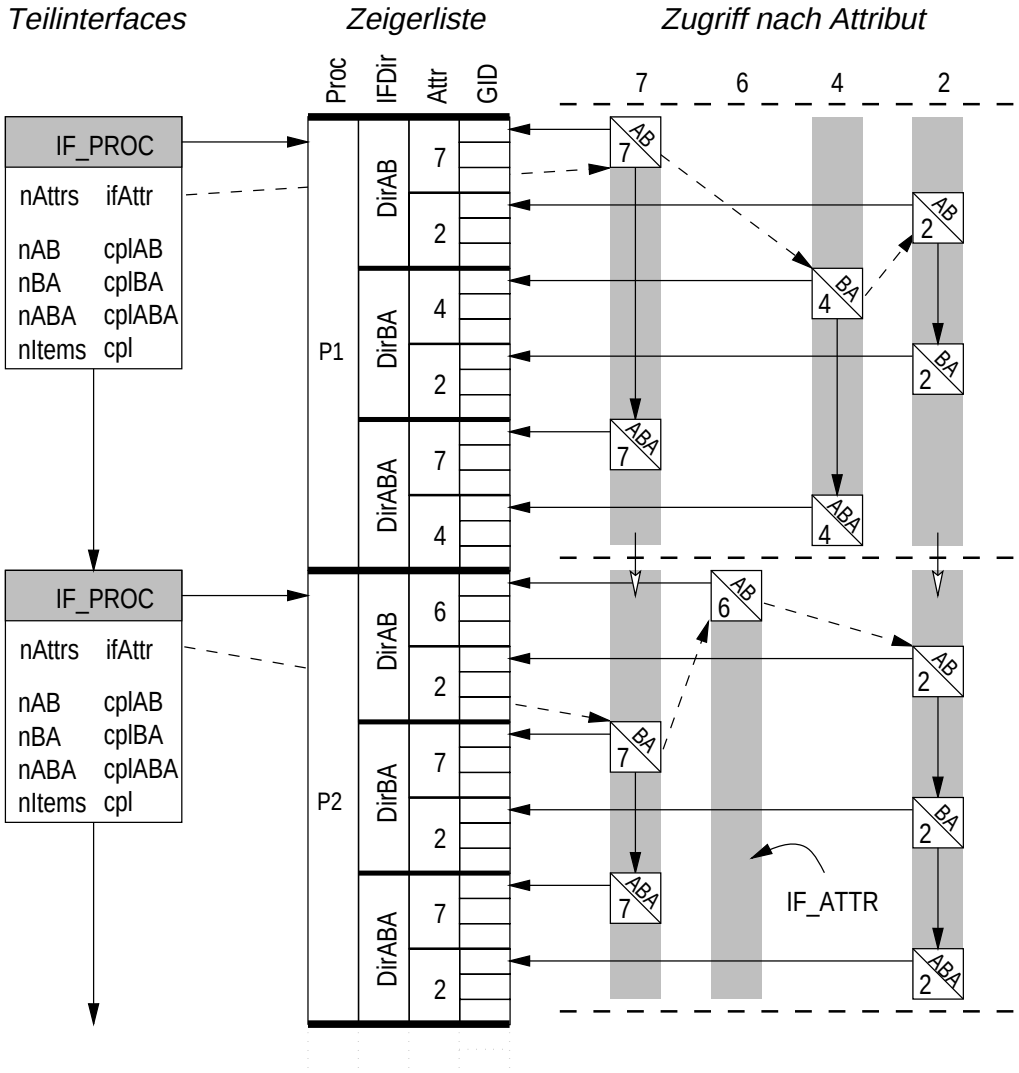


Abbildung 3.5: *Datenstrukturen eines DDD-Interfaces, aus der lokalen Sicht eines Prozessors.*

zu können. Daher muß eine Erweiterung der Interface-Datenstruktur vorgesehen werden, die den Zugriff auf diese Untermengen eines Interfaces erlaubt.

Für ein gegebenes Objekt-Attribut $a \in A$ sei $X^p|_a$ die gemäß Schlüssel 3.1 sortierte Couplingmenge $\mathtt{cpl}^p_X|_a \subseteq \mathtt{cpl}^p_X$, die nur Couplings zu den verteilten Objekten $\hat{o} \in \hat{O}(X^p)$ enthält, für die $\mathtt{attr}(\hat{o}) = a$ gilt. Bei analoger Vorgehensweise für die Teilinterfaces $X^{p\to q}$ entsteht eine Zerlegung in kleinere Teilinterfaces $X^{p\to q}|_a$ zu jedem Attribut $a \in A$, die in der Implementierung durch die Datenstruktur `IF_ATTR` repräsentiert werden. Diese Datenstruktur, im rechten Teil von Abb. 3.5 durch senkrechte graue Balken dargestellt, speichert wie die Datenstruktur `IF_PROC` Anfang und Länge der Abschnitte gleicher Couplingrichtung, allerdings beschränkt auf Couplings aus $\mathtt{cpl}^p_X|_a$.

Ein Beispiel: zur Kommunikation aller Objekte mit $\mathtt{attr} = 4$ im dargestellten Interface wird nur eine `IF_ATTR`-Struktur gefunden; diese befindet sich im Zeigerlistenbereich mit $\mathtt{proc} = P1$ und be-

wirkt damit eine Nachricht an/von Prozessor $P1$. Dabei stehen für eine Kommunikation in Richtung `DirBA` (Rückwärtskommunikation) drei Objekte zur Verfügung, und in Richtung `DirABA` (Kommunikation ohne ausgezeichnete Richtung) ebenfalls drei Objekte. In Richtung `DirAB` befinden sich keine Objekte im Teilinterface $X^p|_4$.

Die `IF_ATTR`-Strukturen gleichen Attributs werden linear (in vertikaler Richtung) verkettet und bilden somit die Repräsentation von $X^p|_a$; zusätzlich wird eine horizontale Verkettung gespeichert (gestrichelte Pfeile in Abb. 3.5), welche die Zerlegung von $X^{p\to q}$ in die einzelnen $X^{p\to q}|_a$ darstellt. Die horizontale Verkettung dient dazu, das gesuchte Attribut aufzufinden; die vertikale Verkettung wird sodann zum Durchlaufen aller Objekte im Interface mit dem gegebenen Attribut benutzt.

Zusammenhang der lokalen Teilinterfaces

Für den späteren Einsatz der Interfaces im verteilten System müssen diese ebenfalls auf die Prozessoren verteilt gespeichert werden. Um den Zusammenhang zweier Teilinterfaces $X^{p\to q}$ und $X^{q\to p}$ zu einer Interfacedefinition $X = (T_X, A, B)$ zu verstehen, betrachte man exemplarisch ein verteiltes Objekt $\hat{o}$ und zwei lokale Objekte $o_1, o_2 \in \hat{o}$, wobei gerade $p = \mathsf{proc}(o_1)$ und $q = \mathsf{proc}(o_2)$ gelten soll. Aus Symmetriegründen tritt auf Prozessor q der Fall $A \leftharpoondown B$ genau dann auf, wenn auf Prozessor p der Fall $A \rightharpoonup B$ gilt und umgekehrt. Somit gilt für Coupling $c_1 = (o_1, q, prio(o_2))$ auf Prozessor p und Coupling $c_2 = (o_2, p, prio(o_1))$ auf q genau einer der folgenden vier Fälle:

1. $\hat{o}$ ist nicht im Interface X (wegen $c_1 \notin \mathsf{cpl}_X^p$ und $c_2 \notin \mathsf{cpl}_X^q$)
2. $dir(c_1) = \mathtt{DirAB}$ und $dir(c_2) = \mathtt{DirBA}$ (d.h. die Vorwärtsrichtung der Kommunikation geht von p nach q)
3. $dir(c_1) = \mathtt{DirBA}$ und $dir(c_2) = \mathtt{DirAB}$ (d.h. die Vorwärtsrichtung der Kommunikation geht von q nach p)
4. $dir(c_1) = \mathtt{DirABA}$ und $dir(c_2) = \mathtt{DirABA}$ (d.h. es gibt keine ausgezeichnete Richtung der Kommunikation)

Wegen dieses Zusammenhangs und der Sortierung von X^p bzw. X^q nach Attribut und globaler ID der verteilten Objekte innerhalb von Bereichen gleicher Couplingrichtung gilt folgende Beziehung, die zusätzlich in Abb. 3.6 grafisch dargestellt ist:

$$\begin{aligned} \hat{O}(X^{p\to q}_{\mathtt{DirAB}}) &= \hat{O}(X^{q\to p}_{\mathtt{DirBA}}) \\ \hat{O}(X^{p\to q}_{\mathtt{DirBA}}) &= \hat{O}(X^{q\to p}_{\mathtt{DirAB}}) \\ \hat{O}(X^{p\to q}_{\mathtt{DirABA}}) &= \hat{O}(X^{q\to p}_{\mathtt{DirABA}}) \end{aligned} \tag{3.2}$$

Diese Beziehung drückt die zu Beginn dieses Abschnitts angesprochene implizite Information aus: die Sortierungen der Teilinterfaces auf den Prozessoren p und q entsprechen einander, so daß keine weitere Identifizierung der am Interface beteiligten Objekte nötig ist. Schickt man als Nachricht zwischen den Prozessoren Nutzdaten, die dieser Sortierung entsprechen, so kann die Zuordnung auf der Empfängerseite implizit stattfinden (Anmerkung: das gleiche Prinzip kommt ebenfalls in den Modulen Identifikation und Transfer zur Anwendung).

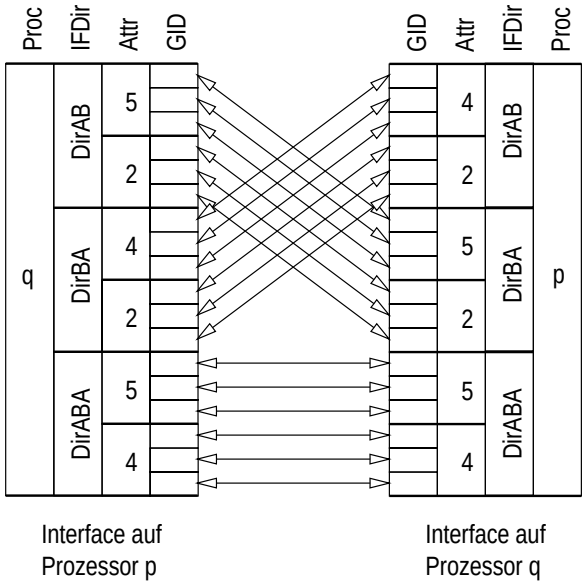


Abbildung 3.6: *Die Beziehung zweier Teilinterfaces $X^{p\to q}$ und $X^{q\to p}$ auf zwei Prozessoren p und q. Die Reihenfolge bei entsprechender Couplingrichtung ist auf beiden Prozessoren gleich.*

3.2.2 Konstruktion der Interfaces

Um eine modulare und fehlersichere Implementierung des Interfacemoduls zu ermöglichen, wurde die Konstruktion und Benutzung von Interfaces streng gekapselt. Die Schnittstelle des Interfacemoduls sieht aus diesem Grund keine Funktionen vor, um die Interface-Datenstruktur inkrementell an kleinere topologische Veränderungen des verteilten Graphen durch das Transfermodul bzw. das Identifymodul anzupassen; in der Praxis sind solche Änderungen jedoch ohnehin meist so umfangreich, daß sich inkrementelle Anpassungen nicht lohnen und stattdessen die Interfaces nach jeder Transfer- bzw. Identify-Operation wieder neu aufgebaut werden.

Die Schnittstelle zur Konstruktion von Interfaces besteht somit aus nur einer Funktion

IFCreateFromScratch(`DDD_IF` X**)**,

die das gegebene Interface X gegebenenfalls löscht, initialisiert und schließlich neu aufbaut. Dazu greift die Funktion auf die Grunddaten zum jeweiligen Interface zurück, die aus den Definitionsdaten (aus der Anmeldung des Interfaces) und den Laufzeitdaten (aus der letzten Konstruktionsphase) bestehen. Die Konstruktion eines Interfaces erfolgt ohne jegliche Kommunikation auf Basis der bestehenden Couplingmenge $\mathtt{cpl}^p$ und gliedert sich in folgende Phasen:

1. Bestimmung der für das Interface X relevanten Couplingmenge $\mathtt{cpl}^p_X$ aus der gesamten Couplingmenge $\mathtt{cpl}^p$ auf jedem Prozessor p. Dazu wird die lokale Couplingtabelle durchlaufen, wobei jedes lokale Objekt auf Zugehörigkeit zum Interface überprüft wird.
2. Lokale Sortierung der Couplingmenge $\mathtt{cpl}^p_X$ gemäß Schlüssel 3.1.

3. Aufbau der erforderlichen IF_PROC- und IF_ATTR-Strukturen auf Basis der sortierten Liste $\mathtt{cpl}_X^p$. Dazu wird die sortierte Liste aus (2) einmal durchlaufen.

4. Aufbau zusätzlicher Datenstrukturen zur Gewährleistung effizienten Zugriffs während der Interfacebenutzung. Dies ist im wesentlichen eine Liste von Zeigern auf die lokalen Objekte aus $\hat{O}(X^{p\to q})$ für die jeweiligen Teilinterfaces.

5. Aufbau der Kommunikationskanäle, die für Kommunikationen auf dem Interface benötigt werden.

Zeit- und Speicheraufwand zur Interfacekonstruktion

Der pro Interface erforderliche Speicherbedarf ist gering, da nur Listen von Zeigern auf Couplingstrukturen und zusätzlich wenige IF_PROC- bzw. IF_ATTR-Strukturen benötigt werden. Die Größe der Zeigerlisten ist durch $|\,\mathtt{cpl}_X^p\,|$ gegeben. Der Zeitbedarf des Interfaceaufbaus ist ebenfalls unbedeutend, da neben einem mehrmaligen Durchlaufen der Liste zur Couplingmenge $\mathtt{cpl}_X^p$ nur ein Sortiervorgang derselben Liste notwendig ist ($O(n\log(n))$ im mittleren Fall, mit $n = |\,\mathtt{cpl}_X^p\,|$). In Abschnitt 4.3.2 sind zeitliche Meßdaten zur Konstruktion von Interfaces niedergelegt.

3.2.3 Kommunikation auf Interfaces

Die Benutzerschnittstelle des Interfacemoduls bietet mehrere Funktionen zur Kommunikation auf Interfaces; diese lassen sich nach mehreren, orthogonalen Eigenschaften klassifizieren:

Kommunikationsrichtung: Die Interfacekommunikation kann in eine bestimmte Richtung (**DDD_IFOneway()**) oder bidirektional (**DDD_IFExchange()**) stattfinden (vgl. 2.4.2).

Attribut: Die Kommunikation kann auf dem ganzen Interface X oder auf einem Teil eines Interfaces $X|_a$ (ausgewählt nach dem Objektattribut a) stattfinden.

Mechanismus: Die Kommunikation kann blockierend oder nichtblockierend stattfinden. Die nichtblockierende Variante ermöglicht es, zwischen dem Start und der Abfrage auf Beendigung der Interfacekommunikation sinnvolle Rechenarbeit einzuschieben, um die Überlappung von Kommunikation und Berechnung zu ermöglichen.

Durch die oben beschriebenen Möglichkeiten ergeben sich insgesamt acht verschiedene Funktionen; bezieht man weitere Ergänzungen der hier beschriebenen Funktionalität mit ein (z.B. Variabilität der Datengröße pro Objekt im gleichen Interface), so vervielfacht sich diese Zahl noch. Um nun eine übersichtliche Implementierung der vielfältigen Kommunikationsfunktionen zu erreichen, die aber gleichzeitig effizient arbeitet, wurden alle nach außen sichtbaren Funktionen einheitlich aus parametrisierbaren Teilschritten zusammengesetzt, die in Abb. 3.7 als Struktogramm dargestellt sind und im folgenden in der tatsächlich ausgeführten Reihenfolge kurz beschrieben werden:

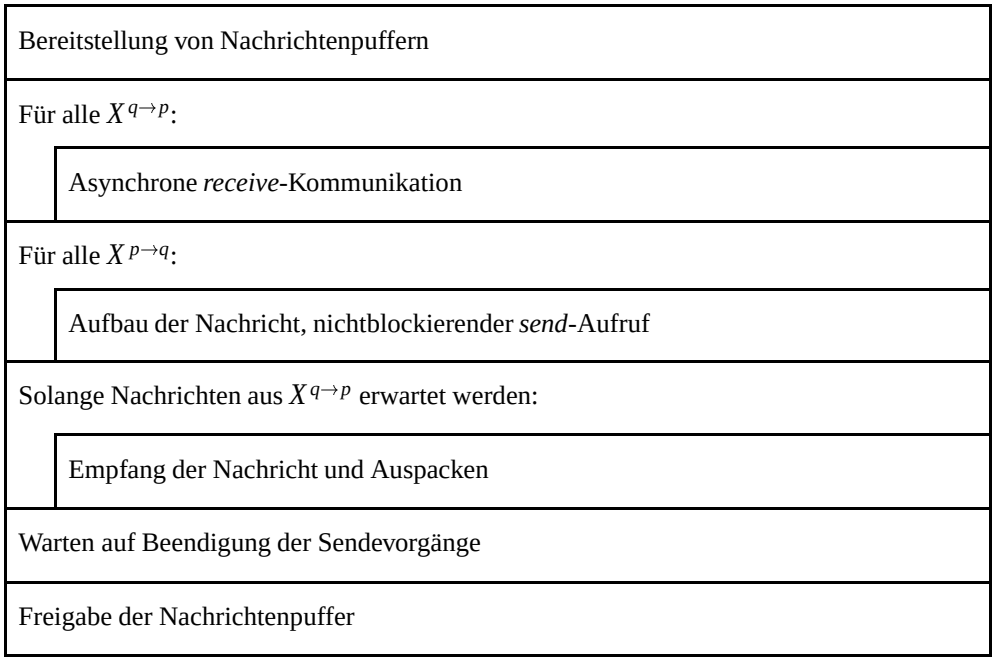


Abbildung 3.7: *Ablaufschema der Interfacekommunikation, auf Prozessor p.*

1. Bereitstellung von Speicher für zu versendende und für zu empfangende Nachrichten. Die Größe der beiden Nachrichtenpuffer pro Teilinterface wird aus der Anzahl der Objekte im jeweiligen Teilinterface $X^{p\to q}$ (bzw. einer Teilmenge davon) und der zu übertragenden Datengröße pro Objekt berechnet und in der zugehörigen `IF_PROC`-Struktur gespeichert.

2. Initiierung der asynchronen *receive*-Kommunikationen. Je nach Implementierung des zugrundeliegenden *message-passing*-Modells ist es unter Umständen vorteilhaft, den Empfangsteil einer asynchronen Kommunikation vor dem Empfang der Nachricht anzustoßen, um das Anlegen eines zusätzlichen Datenpuffers und das nachträgliche Kopieren der Daten in den benutzerdefinierten Puffer zu vermeiden (z.B. bei Intel Paragon, vgl. Abschnitt 4.6).

3. Aufbau der zu versendenden Nachrichten und Senden. Dieser Teilschritt setzt sich aus einer Anzahl von Schleifen über die Objekte im Interface zusammen, die den Sendenachrichtenpuffer durch objektweise Aufrufe der benutzerdefinierten *gather*-Funktion mit Daten füllen. Aus Effizienzgründen werden dazu nicht die Couplinglisten $\mathtt{cpl}_X^p$ benutzt, sondern die während der Interfacekonstruktion aufgebauten Zeigerlisten auf die lokalen Objekte aus $\hat{O}(X^{p\to q})$.

Je nach Kommunikationsrichtung werden dabei die folgenden Teilinterfaces eingepackt:

Unidirektional/vorwärts: $(X^{p\to q}_{\mathtt{DirAB}}, X^{p\to q}_{\mathtt{DirABA}})$
Unidirektional/rückwärts: $(X^{p\to q}_{\mathtt{DirBA}}, X^{p\to q}_{\mathtt{DirABA}})$
Bidirektional: $(X^{p\to q}_{\mathtt{DirAB}}, X^{p\to q}_{\mathtt{DirBA}}, X^{p\to q}_{\mathtt{DirABA}})$

Direkt nach dem Einpacken der Daten eines Teilinterfaces $X^{p\to q}$ wird die zugehörige Nachricht an den Empfängerprozessor q gesendet. Die nichtblockierende Version der jeweiligen Kommunikationsfunktion endet, nachdem alle Teilinterfaces behandelt wurden. Alle folgenden Schritte werden bei der blockierenden Version direkt ausgeführt, bei der nichtblockierenden Version dagegen durch den Benutzer zu einem späteren Zeitpunkt aufgerufen.

4. Warten auf Nachrichtenempfang und Auspacken sofort nach Empfang. In einer Schleife werden nacheinander alle Teilinterfaces zyklisch abgefragt; sobald eine Nachricht vom zugehörigen Sendeprozessor empfangen wurde, kann diese ausgepackt werden. Dazu wird (analog zum vorherigen Einpackschritt) durch objektweise Aufrufe der benutzerdefinierten *scatter*-Funktion der Empfangsnachrichtenpuffer geleert und dessen Daten in die vorhandenen Objekte im Interface integriert. Durch einen *timeout*-Mechanismus werden Verklemmungen durch Hardware-Ausfall in gewissem Maß abgefangen und erkannt bzw. ignoriert (Fehlertoleranzaspekt).

Aus der Beziehung der lokalen Teilinterfaces 3.2 (S. 70) ergeben sich je nach Kommunikationsrichtung Auspackvorgänge auf folgenden Interfaceabschnitten:

Unidirektional/vorwärts:	$(X^{q\to p}_{\texttt{DirBA}}, X^{q\to p}_{\texttt{DirABA}})$
Unidirektional/rückwärts:	$(X^{q\to p}_{\texttt{DirAB}}, X^{q\to p}_{\texttt{DirABA}})$
Bidirektional:	$(X^{q\to p}_{\texttt{DirBA}}, X^{q\to p}_{\texttt{DirAB}}, X^{q\to p}_{\texttt{DirABA}})$

5. Warten auf Beendigung der Sendevorgänge. Wurden alle Nachrichten empfangen und ausgepackt, sind im Normalfall auch alle lokal angestoßenen Sendevorgänge beendet; dies wird in diesem Teilschritt sichergestellt.

6. Freigabe des Speichers für Nachrichtenpuffer. Zu Beginn des Kommunikationsvorgangs wurden für jedes Teilinterface zwei Nachrichtenpuffer angelegt; diese werden nun nicht mehr benötigt und können freigegeben werden. Optional können die Nachrichtenpuffer bis zum nächsten Gebrauch des Interfaces vorgehalten werden, um den Aufwand für Speicherverwaltung zu sparen.

3.3 Der Identifikationsmechanismus

3.3.1 Globale IDs und Grundidee der Identifikation

In Abschnitt 3.1.1 wurde bereits auf die Erzeugung von globalen IDs für verteilte Objekte eingegangen. Damit diese global eindeutige Namensgebung in einem verteilten Namensraum effizient und skalierbar implementiert werden kann, mußten folgende Eigenschaften des Mechanismus zur Namensverwaltung eingehalten werden:

- Globale IDs (d.h. Namen) werden nur lokal erzeugt und sind zunächst nur dem sie erzeugenden Prozessor bekannt. Durch Eincodierung der Prozessornummer kann die Erzeugung verteilt, ohne Kommunikation und doch global eindeutig durchgeführt werden.
- Es können nur solche Objekte referenziert werden, die lokal vorhanden sind (Einschränkung in Modell L, siehe 2.3.3). Damit muß für eine Referenzierung auch die globale ID des referenzierten Objekts vorhanden sein.
- Die globale ID eines Objekts, das auf einem gegebenen Prozessor nicht gespeichert wird (und daher auch nicht von anderen seiner lokalen Objekte referenziert werden kann), ist diesem nicht bekannt. Entsprechend wird bei Löschung eines lokalen Objekts dessen globale ID nicht länger gespeichert.
- Durch den Transfer von Objekten werden deren globale IDs dem Empfängerprozessor bekannt gemacht, dies ist eine von zwei Möglichkeiten zur Erzeugung von (echten) verteilten Objekten.

Das Identifymodul bietet nun die andere Möglichkeit, um lokale erzeugte IDs anderen Prozessoren bekanntzumachen; die Grundidee bei der Identifikation zweier lokaler Objekte in bezug auf die verteilte Namensvergabe ist, mittels einer implizit definierten Operation aus den alten globalen IDs der beiden lokalen Objekte die globale ID des neu erzeugten, verteilten Objekts zu errechnen und diese den beteiligten lokalen Objekten zuzuweisen. In der hier beschriebenen Implementierung wird dazu die Minimumoperation verwendet, d.h. die kleinere der beiden alten globalen IDs setzt sich durch. Beide beteiligten Prozessoren können diese Operation unabhängig voneinander, jedoch mit demselben Ergebnis durchführen, wenn ihnen beide alten globalen IDs zur Verfügung stehen. Die in die globale ID eincodierte Prozessornummer gibt somit im allgemeinen nicht den Prozessor wieder, auf dem ein verteiltes Objekt erzeugt wurde.

Sollen nun zwischen zwei Prozessoren in einem Vorgang mehrere lokale Objekte identifiziert werden, so ist es notwendig, die jeweils zu identifizierenden Objekte paarweise einander zuzuordnen. Der naive Ansatz, die Identifikatoren in einer Nachricht an den anderen Prozessor zu schicken, scheitert spätestens bei komplexen Identifikationstupeln an der Effizienz der Datenübertragung. Gelingt es jedoch, auf beiden Prozessoren die gleiche Sortierung der Identifikationstupel herzustellen, so ist es ausreichend, die sortierte Liste der globalen IDs zwischen den Prozessoren auszutauschen. Der Kernpunkt der Identify-Implementierung ist also, eine totale Ordnung auf allen möglichen Identifikationstupeln zu finden.

Die beiden folgenden Abschnitte erklären zunächst den groben Ablauf eines Identifikationsvorgangs und schließlich die oben erwähnte totale Ordnung.

3.3.2 Ablauf des Identifikationsvorgangs

Das transaktionsorientierte Schema des Identifikationsvorgangs führt zu folgenden Phasen während der Programmausführung (vgl. Abschnitt 2.4.4):

1. Kommandophase. Das Anwendungsprogramm leitet die Identifikation durch globalen Aufruf der Funktion **DDD_IdentifyBegin()** ein. Die dann folgenden **Identify**-Aufrufe werden auf jedem Prozessor lokal zwischengespeichert. Der globale Aufruf von **DDD_IdentifyEnd()** beendet diese Phase; der eigentliche Identifikationsvorgang beginnt.

Die vom Anwendungsprogramm geforderten Identifikatoren $\Lambda_i = (\lambda_{i,j})$ der lokalen Objekte werden je nach Wahl des Einzelidentifikators $\lambda_{i,j}$ (d.h. einer der Typen *ganze Zahl*, *Zeichenkette* oder ein bereits bekanntes bzw. ebenfalls zu identifizierendes *verteiltes Objekt*) über je eine **Identify**-Funktion bereitgestellt; zur temporären Speicherung wird für jeden dieser Funktionsaufrufe intern eine `IDENTINFO`-Struktur erzeugt. Gleichzeitig wird für jeden Partnerprozessor eine Liste angelegt, in der alle auf diesen Prozessor bezogenen `IDENTINFO`-Strukturen gespeichert werden, um eine Vorsortierung der Kommandos zu erreichen.

2. Vorbereitungsphase. Zu jeder der in der Kommandophase angelegten Listen muß später eine Nachricht an den jeweiligen Partnerprozessor gesendet bzw. von ihm empfangen werden, die alle notwendigen Informationen über die zu identifizierenden lokalen Objekte enthält; in dieser Phase wird nun zunächst Speicher für jede zu versendende bzw. zu empfangende Nachricht bereitgestellt.

Danach wird die Sortierung gemäß der im folgenden Abschnitt vorgestellten totalen Ordnung auf Identifikationstupeln listenweise durchgeführt; aus diesem Sortiervorgang geht zu jedem Partnerprozessor eine sortierte Liste hervor, bei der jeder Eintrag folgende Form hat:

Eintrag in lokaler Identifikationstabelle		
`DDD_HDR`	*hdr*	`DDD_HEADER` des lokalen Objekts
`DDD_GID`	*gid*	globale ID des lokalen Objekts
`DDD_PRIO`	*prio*	Priorität des lokalen Objekts

Schließlich wird der Sendepuffer zu jedem Partnerprozessor mit den Einträgen *gid* und *prio* aus der zugehörigen Tabelle gefüllt.

3. Kommunikation. Wie im oben beschriebenen Interfacemodul werden zunächst Kanäle zu den Kommunikationspartnern aufgebaut und anschließend die Empfangsfunktionen der nichtblockierenden Kommunikation initiiert. Danach werden die Sendepuffer an den Partnerprozessor geschickt. Wie schon im Interfacemodul ist der Kernpunkt des verteilten Algorithmus, notwendige Informationen lokal von den Kommunikationspartnern nach einem implizit bekannten Verfahren zu generieren anstatt sie mittels Kommunikation zu verbreiten.

4. Identifikation. Nachdem jeweils eine der Nachrichten empfangen wurde, können die darin beschriebenen Identifikationsvorgänge durchgeführt werden. Dies beinhaltet zwei Schritte, die von beiden Prozessoren unabhängig voneinander auf ihrem jeweils lokalen Objekt durchgeführt werden:

1. Aus der globalen ID des lokalen Objekts (zugänglich über den Eintrag *gid* in der gesendeten Tabelle) und der globalen ID des fremden Objekts (Eintrag *gid* in der empfangenen Tabelle) wird über die Minimumfunktion von beiden Prozessoren dieselbe neue globale ID berechnet und dem jeweils lokalen Objekt zugewiesen. Die Zuweisung erfolgt über den Verweis auf die `DDD_HEADER`-Struktur des lokalen Objekts in der gesendeten Tabelle (Eintrag *hdr*).
2. Der Couplingliste des lokalen Objekts o_{lokal} wird ein neuer Eintrag $(o_{\text{lokal}}, \mathsf{proc}, \mathsf{prio})$ hinzugefügt, wobei proc den Partnerprozessor der Identifikation und prio den *prio*-Eintrag in der empfangenen Tabelle bezeichnet. Der Couplingeintrag steht für das fremde, jetzt identifizierte Objekt.

5. Abschluß. Schließlich werden die ordnungsgemäße Beendigung aller Sendevorgänge kontrolliert, der Speicher für alle Nachrichtenpuffer freigegeben und zuletzt die Konsistenz der zuvor angemeldeten Interfaces wiederhergestellt.

Aus der Sicht eines Prozessors p muß pro Partnerprozessor q in der Identifikation eine Nachricht gesendet bzw. eine Nachricht empfangen werden. Der Kommunikationsaufwand für eine solche Nachricht hängt linear von der Anzahl n_q der den Prozessor q betreffenden *Identify*-Tupeln Λ_i ab, da für jedes Λ_i konstant zwei Werte gesendet werden müssen. Der lokale Rechenaufwand für die Identifikation wird asymptotisch von der Sortierung der Tupel bestimmt, die im mittleren Fall die Komplexität $O(n_q \log(n_q))$ aufweist. Alle anderen Schritte (z.B. die Identifikation selbst) haben lineare Komplexität $O(n_q)$, was von der symmetrischen Struktur der *Identify*-Kommandos an der Anwendungsschnittstelle herrührt.

3.3.3 Eine Ordnung auf Identifikationstupeln

Gemäß Abschnitt 3.3.2 werden die Identifikationskommandos (d.h. die `IDENTINFO`-Strukturen) zunächst nach dem jeweiligen Partnerprozessor vorsortiert, so daß es an dieser Stelle ausreicht, nur Identifikationskommandos an *einen* Partnerprozessor zu betrachten. Darüberhinaus werden die Kommandos nach der (alten) globalen ID der zu identifizierenden Objekte sortiert, so daß neben der Sichtweise auf die Kommandos selbst (und damit auf die Einzelidentifikatoren) auch der Zugriff auf die Identifikationstupel Λ_i möglich ist.

Das Ziel ist es nun, eine totale Ordnung auf Identifikationstupeln so zu definieren, daß von beiden beteiligten Prozessoren die gleiche Sortierung der Identifikationstupel erzeugt werden kann. Laut Definition der Identifikationskommandos muß zu jedem Kommando eines Prozessors ein entsprechendes Kommando des anderen Prozessors aufgerufen worden sein; daher existieren die Kommandos stets paarweise, was eine Grundbedingung für diese Sortierung ist. Ebenso muß sich die Struktur jedes Tupelpaars eindeutig entsprechen, dadurch kann die Ordnung auf Tupeln auf die Ordnungen der Einzelidentifikatoren zurückgeführt werden. Die meisten dieser Einzelordnungen sind sofort klar: für **DDD_IdentifyNumber()** und **DDD_IdentifyString()** können die betreffenden Ordnungen auf ganzen Zahlen bzw. Zeichenketten verwendet werden. Für bereits identifizierte Objekte (bei **DDD_IdentifyObject()**) kann ebenfalls die Ordnung ganzer Zahlen auf Basis der globalen IDs eingesetzt werden.

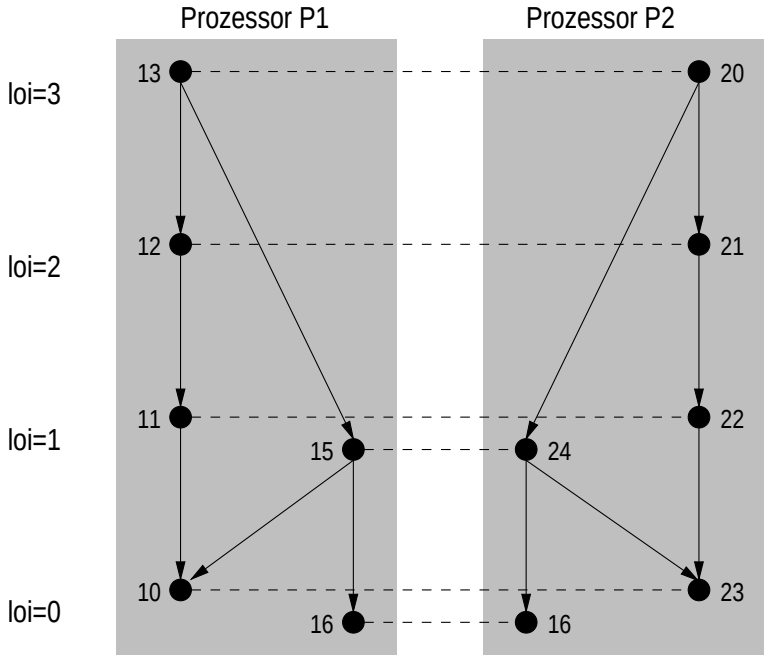


Abbildung 3.8: *Beispiel einer Identifikation mittels zu identifizierender Objekte. Darstellung der Identifikationskommandos als partielle Ordnung; Pfeile deuten eine* wird-identifiziert-durch-Identifikator-*Relation an, gestrichelte Linien zeigen die zu identifizierenden Objekte an. Die Zahlen sind beispielhafte globale IDs* vor *der Identifikation.*

Ein besonderes Problem stellt jedoch die Identifikation mittels Objekten dar, die selbst erst während des aktuellen Vorgangs identifiziert werden. Diese Identifikatoren sind *a priori* nicht beiden Prozessoren bekannt, können also nicht ohne weiteres als Sortiergrundlage eingesetzt werden. Abb. 3.8 zeigt ein Beispiel, unter anderem wurden folgende Identifikationskommandos erteilt:

Prozessor P1:	Prozessor P2:
IdentifyBegin()	IdentifyBegin()
IdentifyNumber(o_{10}, P2, λ)	IdentifyNumber(o_{23}, P1, λ)
IdentifyObject(o_{12}, P2, o_{11})	IdentifyObject(o_{21}, P1, o_{22})
IdentifyObject(o_{11}, P2, o_{10})	IdentifyObject(o_{22}, P1, o_{23})
...	...
IdentifyEnd()	IdentifyEnd()

Im Beispiel soll also das lokale Objekt o_{10} auf Prozessor P1 mit Objekt o_{23} auf Prozessor P2 identifiziert werden; dies geschieht hier über den Identifikator λ (ganze Zahl), der beiden Prozessoren bekannt ist. Gleichzeitig soll auch das Objekt o_{11} auf P1 mit o_{22} auf P2 identifiziert werden; als Identifikator wird dazu auf das verteilte Objekt zurückgegriffen, das erst aus der Identifikation von o_{10} mit o_{23} entsteht. In Abb. 3.8 ist dieser Zusammenhang durch jeweils einen Pfeil dargestellt. Ebenso wird die Identifikation von o_{12} auf P1 und o_{21} auf P2 mit dem verteilten Objekt durchgeführt, daß aus der gleichzeitigen Identifikation von o_{11} mit o_{22} hervorgeht.

Die Verwendung der globalen IDs der Identifikator-Objekte als Grundlage der Ordnung auf den Identifikationstupeln führt nun zu einem Fehler bei der Sortierung und damit der Zuordnung von zu identifizierenden Objekten. Denn im Beispiel würde auf Prozessor P1 wegen $o_{10} < o_{11}$ die Sortierreihenfolge (o_{11}, o_{12}) und auf P2 wegen $o_{22} < o_{23}$ die Reihenfolge (o_{21}, o_{22}) gelten. Die Identifikation erfolgt gemäß der Sortierreihenfolge, damit würden das Objekt o_{11} mit o_{21} und das Objekt o_{12} mit o_{22} identifiziert. Dies ist offensichtlich falsch.

Tatsächlich bilden die **DDD_IdentifyObject()**-Kommandos einen zyklenfreien, gerichteten Graphen, wobei die Identifikation die Bestimmung der Isomorphie zweier Graphen darstellt. Zur Konstruktion des Graphen gilt die Beziehung

$$o_1 \leftrightarrows o_2 \qquad \Leftrightarrow \qquad \text{IdentifyObject}(o_1, \text{P}, o_2)$$

für Partnerprozessor P. Jeder Knoten des so entstehenden Graphen wird mit einem *level-of-indirection*-Wert (kurz: *loi*-Wert) markiert, der angibt, wieviele indirekte Identifikationen für das betreffende Objekt nötig sind. Objekte des Graphen, von denen keine Pfeile ausgehen, sind entweder bereits verteilte Objekte (z.B. Objekt o_{16} in Abb. 3.8) oder werden selbst über **DDD_IdentifyNumber()** oder **DDD_IdentifyString()** identifiziert (z.B. Objekte o_{10} und o_{23}). Diese Objekte (die „Blätter“ des gerichteten Graphen) werden mit *loi*$= 0$ markiert. Alle anderen Objekte o_i erhalten den *loi*-Wert

$$\text{loi}(o_i) \quad = \quad 1 + \max_{o \in \text{succ}(o_i)} \text{loi}(o)$$

wobei $\text{succ}(o_i)$ die Menge der auf o_i folgenden Objekte im Graphen darstellt. In Abb. 3.8 wurden die Objekte beider Graphen bereits nach ihrem *loi*-Wert ebenenweise dargestellt.

Zur korrekten Sortierung werden nun die Identifikationstupel zunächst nach dem *loi*-Wert des betreffenden Objekts vorsortiert und danach innerhalb des gleichen *loi*-Werts endgültig geordnet, ausgehend von loi$= 0$. Als Sortierschlüssel dient jeweils nicht die globale ID des Identifikator-Objekts, sondern dessen Index in seiner jeweiligen *loi*-Ebene. Im Beispiel gilt also zunächst folgende Ordnung:

Prozessor P1

Objekt	Ident-Tupel	*loi*	Index
o_{16}	–	0	**0**
o_{10}	(λ)	0	**1**
o_{11}	(10)	1	
o_{15}	(10,16)	1	
o_{12}	(11)	2	
o_{13}	(12,15)	3	

Prozessor P2

Objekt	Ident-Tupel	*loi*	Index
o_{16}	–	0	**0**
o_{23}	(λ)	0	**1**
o_{24}	(23,16)	1	
o_{22}	(23)	1	
o_{21}	(22)	2	
o_{20}	(21,24)	3	

Die Tupel wurden also jeweils nach ihrem *loi*-Wert sortiert, innerhalb der Ebene loi$= 0$ wurden die Tupel dann endgültig sortiert und in der Spalte *Index* durchnumeriert. In allen restlichen Tupeln wird sodann die globale ID der Objekte mit loi$= 0$ durch diesen Index ersetzt, danach wird die Ebene loi$= 1$ sortiert. Dies ergibt folgende Ordnung:

Prozessor P1			
Objekt	Ident-Tupel	*loi*	Index
o_{16}	–	0	**0**
o_{10}	(λ)	0	**1**
o_{11}	(**1**)	1	**2**
o_{15}	(**1**,**0**)	1	**3**
o_{12}	(11)	2	
o_{13}	(12,15)	3	

Prozessor P2			
Objekt	Ident-Tupel	*loi*	Index
o_{16}	–	0	**0**
o_{23}	(λ)	0	**1**
o_{22}	(**1**)	1	**2**
o_{24}	(**1**,**0**)	1	**3**
o_{21}	(22)	2	
o_{20}	(21,24)	3	

In zwei weiteren Schritten wird genauso verfahren, bis der Algorithmus beim höchsten *loi*-Wert und mit einer kompletten Sortierung der Tupel endet:

Prozessor P1			
Objekt	Ident-Tupel	*loi*	Index
o_{16}	–	0	**0**
o_{10}	(λ)	0	**1**
o_{11}	(**1**)	1	**2**
o_{15}	(**1**,**0**)	1	**3**
o_{12}	(**2**)	2	**4**
o_{13}	(**4**,**3**)	3	**5**

Prozessor P2			
Objekt	Ident-Tupel	*loi*	Index
o_{16}	–	0	**0**
o_{23}	(λ)	0	**1**
o_{22}	(**1**)	1	**2**
o_{24}	(**1**,**0**)	1	**3**
o_{21}	(**2**)	2	**4**
o_{20}	(**4**,**3**)	3	**5**

Nun ist die Tabelle eine Repräsentation der beiden korrespondierenden Graphen aus Abb. 3.8, die Identifikation kann stattfinden:

Prozessor P1	Prozessor P2
o_{10}	o_{23}
o_{11}	o_{22}
o_{15}	o_{24}
o_{12}	o_{21}
o_{13}	o_{20}

Aus Effizienzgründen werden die Identifikationstupel Λ_i innerhalb jeder *loi*-Ebene grob vorsortiert: zu jedem Identifikationstupel Λ_i wird aus der Anzahl und den Typen der Einzelwerte $\lambda_{i,j}$ eine Tupel-ID berechnet, nach der die Tupel dann sortiert werden. Erst bei Objekten gleicher Tupel-ID werden tatsächlich die Einzelidentifikatoren verglichen. Falls vom Benutzer als Option die Reihenfolgeunabhängigkeit der Einzelidentifikatoren innerhalb jedes Tupels gewählt wurde, werden die Einzelidentifikatoren pro Tupel noch sortiert, bevor die Tupel-IDs berechnet und die Tupel innerhalb der *loi*-Ebene sortiert werden. In obigem Beispiel wurde diese Option aber nicht ausgewählt, so daß die Reihenfolge der *Identify*-Kommandos für die Ordnung der Einzelidentifikatoren innerhalb des Tupels maßgebend ist (Objekte o_{15}, o_{13}, o_{24} und o_{20}).

3.4 Das Transfermodul

3.4.1 Teilprobleme der Implementierung

Die Grundfunktionalität des Transfermoduls erlaubt jedem beteiligten Prozessor, Kopien lokaler Objekte auf beliebigen anderen Prozessoren zu erzeugen, lokale Objekte zu löschen und die Priorität von lokalen Objekten zu ändern. Zur Implementierung dieser Funktionalität müssen folgende Teilprobleme gelöst werden:

Minimierung von Nachrichten: Die jeweils lokal von der Anwendung erteilte Menge von **DDD_XferCopyObj()**-Transferkommandos (Anzahl n) muß nach Zielprozessor geordnet auf möglichst wenige zu sendende Nachrichten (Anzahl $O(1) < n$) abgebildet werden, die alle notwendigen Daten der zu kopierenden Objekte enthalten müssen. Auf reinen *message-passing*-Rechnern mit hoher Latenzzeit je Nachricht kann dadurch der Transfer möglichst effizient durchgeführt werden.

Auspackvorgang: Beim Auspacken von Objektkopien müssen diese in die vorhandene Datenstruktur eingepaßt werden. Dies beinhaltet rein technische Speicherverwaltungsaspekte, aber auch die Vorgehensweise bei Kollisionen mit bereits vorhandenen Kopien desselben verteilten Objekts. Die Details sind in einer Spezifikation der Transferkommandos und ihrer Auswirkungen niedergelegt (s. Anhang B).

Couplingkonsistenz: Auf die Erzeugung neuer lokaler Objekte mittels **DDD_XferCopyObj()**, die Vernichtung von lokalen Objekten mittels **DDD_XferDeleteObj()** und die Prioritätsänderung einzelner lokaler Objekte mittels **DDD_PrioritySet()** eines verteilten Objekts $\hat{o}$ muß eine Anpassung der verteilt gespeicherten Couplinglisten $\mathsf{cpl}_{\hat{o}}^{p}$ auf allen Prozessoren $p \in P_{\hat{o}}$ erfolgen. Für die Auswirkungen mehrerer Transferkommandos auf das entstehende verteilte Objekt siehe wieder Anhang B.

Transfer von Referenzen: Beim Transfer von Objekten, die Referenzen auf andere lokale Objekte enthalten (z.B. *pointer* in ANSI C), muß eine Abbildung der Referenzen vom Adreßraum des sendenden Prozessors in den Adreßraum des empfangenden Prozessors erfolgen. Dazu wird den lokalen Objekten aus einer Transfernachricht eine Symboltabelle beigegeben, die eine Abbildung von Referenzen aus diesen Objekten auf die globalen IDs der referenzierten Objekte darstellt. Beim Empfang einer solchen Nachricht kann die Symboltabelle dann mit den bereits vorhandenen lokalen Objekten abgeglichen werden. Damit werden schließlich die Referenzen aus den transferierten Objekten auf die neuen Ziele umgesetzt.

Der nächste Abschnitt gibt einen Überblick des Ablaufs eines Transfervorgangs; die folgenden Abschnitte widmen sich den oben angesprochenen Teilproblemen ausführlicher.

3.4.2 Ablauf eines Transfervorgangs

Wie bereits in Abschnitt 2.4.3 dargelegt, werden die eigentlichen Transferkommandos vom Anwendungsprogramm erst dann abgesetzt, wenn von allen Prozessoren durch die Funktion

DDD_XferBegin() die Transferphase eingeleitet wurde. Alle Transferkommandos werden danach durch DDD-interne Datenstrukturen geeignet repräsentiert, der eigentliche Transfervorgang mit sämtlichen Kommunikationen findet erst nach dem globalen Aufruf der **DDD_XferEnd()**-Funktion statt. Die Implementierung dieser **DDD_XferEnd()**-Funktion soll nun beschrieben werden.

Zunächst läßt sich der Transfervorgang in drei große Phasen gliedern: Senden von Objekten, Senden von Informationen zur Couplingkonsistenz, Aufbau aller DDD-Interfaces. Genauer betrachtet ist die Implementierung der **DDD_XferEnd()**-Funktion zwar komplizierter, bleibt aber dennoch eine Serie von strikt aufeinanderfolgenden Phasen. Die Reihenfolge dieser Phasen ist zwingend und ergibt sich aus deren Datenabhängigkeiten bzw. der Spezifikation der Transferkommandos und ihrer Auswirkungen (Anhang B). Im Rest des Kapitels wird auf die Regeln C1 bis C4, P1, D1, M1 bis M4 Bezug genommen; dies meint stets die Regeln gemäß dieser Spezifikation.

1. Aufbereitung der DDD_XferCopyObj()-Kommandos. Die Liste der auf jedem Prozessor erteilten **DDD_XferCopyObj()**-Kommandos wird sortiert und um nach Regel C1 bzw. Regel C4 überflüssige Kommandos bereinigt. Dem Anwendungsprogramm wird dadurch ermöglicht, ohne den Einsatz von Flags zur Kontrolle von Doppelnennungen zu arbeiten und gegebenenfalls mehrmals das gleiche Transferkommando zu erteilen, ohne daß der Transfermechanismus tatsächlich die doppelte Arbeit leisten muß.

Für jedes Kommando zur Erzeugung eines lokalen Objekts $o_{\text{neu}} \in \hat{o}$ von Prozessor p an Prozessor q (mit $q \notin P_{\hat{o} \setminus o_{\text{neu}}}$) wird q als *NewOwner* vermerkt; in den Objektnachrichten wird später die (bisherige) Couplingliste $\mathsf{cpl}_{\hat{o}}^{p}$ komplett an q verschickt, damit dieser im Fall der Annahme des lokalen Objekts seinerseits eine konsistente Couplingliste aufbauen kann. Darüberhinaus werden *NewOwner*-Prozessoren zum gleichen verteilten Objekt (falls vorhanden) gegenseitig über ihre neuen lokalen Objekte informiert.

Die bisherigen Eigentümer der lokalen Kopien von $\hat{o}$ (also $p_i \in P_{\hat{o}}$ mit $p_i \neq p$) werden ebenfalls über diesen potentiellen neuen Couplingeintrag informiert; auch dies soll bereits in den Objektnachrichten geschehen. Die zusätzlichen, in den Objektnachrichten zu verschickenden Informationen werden der nächsten Phase in Form von zwei Listen übergeben: die *OldCpl*-Liste für bereits existierende Couplings an *NewOwner*-Prozessoren, die *NewCpl*-Liste für neu einzutragende Couplings.

2. Vorbereitung der Objektnachrichten. Die Verwaltung und Versendung sämtlicher Nachrichten während des DDD-Transfers geschieht mittels der *LowComm*-Kommunikationsschicht (s. Anhang A.3). Diese bietet abstrakte Kommunikationsfunktionen im Nachbarschaftsgraph der Prozessoren, mit deren Hilfe beliebig strukturierte Nachrichten (bestehend aus Tabellen und einfachen Datenblöcken) verschickt werden können. Damit *LowComm* die Nachrichtenpuffer einrichten kann, müssen zuerst Anzahl, Empfänger und jeweilige Größe der Nachrichten ermittelt werden.

Sämtliche Informationen, die in den Objektnachrichten verschickt werden müssen, werden gesammelt und nach Empfängerprozessor sortiert. Die Anzahl der Einträge der Tabellen und die Größe der Datenblöcke innerhalb der Nachrichten werden berechnet und dem *LowComm*-Modul mitgeteilt. Die genaue Struktur der Objektnachrichten wird in Abschnitt 3.4.3 beschrieben.

Die *LowComm*-Schicht stellt für jede zu sendende Nachricht einen Pufferspeicher passender Größe bereit; mit Hilfe des *Notify*-Moduls (s. Anhang A.2) werden die Empfänger informiert, worauf deren *LowComm*-Schicht die Empfangspuffer anlegt.

3. Einpacken/Verschicken der Objektnachrichten. Die von *LowComm* bereitgestellten Sendepuffer können nun mit den notwendigen Informationen gefüllt werden; dazu gehören zunächst die Objekte selbst, aber auch die *OldCpl*- und *NewCpl*-Listen. In Abschnitt 3.4.3 wird der Einpackvorgang genauer beschrieben. Nach Fertigstellung der Nachrichten werden diese vom *LowComm*-Modul an die jeweiligen Empfänger gesendet.

Während das Kommunikationssystem des jeweiligen Parallelrechners die Nachrichten überträgt, können lokale Aufgaben durchgeführt werden. Dazu gehört hauptsächlich das Löschen derjenigen lokalen Objekte, die dafür während der Kommandophase durch einen **DDD_XferDeleteObj()**-Aufruf ausgewählt wurden, sowie die Ausführung der **DDD_PrioritySet()**-Kommandos. Durch die Überlappung von Kommunikation und Durchführung lokaler Operationen wird das Gesamtsystem optimal ausgelastet.

4. Aufbereitung der DDD_PrioritySet()-Kommandos. Die Liste der auf jedem Prozessor erteilten **DDD_PrioritySet()**-Kommandos wird sortiert und um nach Regel P1 überflüssige Kommandos bereinigt. Dies ermöglicht der Anwendung (analog zu Phase 1), ohne lästige Verwaltung von Flags zur Vermeidung von Doppelnennungen auszukommen.

5. Ausführung der DDD_XferDeleteObj()-Kommandos. Die Liste der auf jedem Prozessor erteilten **DDD_XferDeleteObj()**-Kommandos wird sortiert und um nach Regel D1 überflüssige Kommandos bereinigt. Die verbleibenden Kommandos werden in die vorherige Ordnung zurücksortiert, damit die Objekte in der vorgeschriebenen Reihenfolge gelöscht werden können. Auch hier ist die Reihenfolge der Phasen wichtig, da das Löschen von Objekten im allgemeinen erst erfolgen darf, nachdem in Phase 3 die Objektnachrichten zusammengestellt wurden.

Das Löschen lokaler Objekte wird je nach verwendeter ObjectManager-Schnittstelle (vgl. Abschnitt 2.4.1) von verschiedenen benutzerdefinierten Handlern oder vom DDD-ObjectManager selbst durchgeführt. In jedem Fall ist es möglich, während dieser Phase weitere lokale Objekte zu löschen: die erforderlichen Couplinganpassungen werden noch in diesem Transfervorgang bearbeitet. Dadurch kann das Löschen einer Hierarchie von verteilten Objekten einfach und übersichtlich implementiert werden.

Zur Implementierung dieser Eigenschaft wird eine *DelCpl*-Liste aufgebaut, deren Einträge anderen Prozessoren die Löschung eines lokalen Objekts signalisieren sollen. Diese Liste wird während der Phase 9 an die entsprechenden Prozessoren geschickt und dort ausgewertet (evtl. auch an die *NewOwner*-Prozessoren aus Phase 1). Gleichzeitig werden in den *DelCpl*-Einträgen die Couplingliste zum jeweiligen verteilten Objekt gespeichert; falls Prozessor p ein lokales Objekt $o_1 \in \hat{o}$ löscht, aber Prozessor q gleichzeitig ein anderes lokales Objekt $o_2 \in \hat{o}$ an p schickt, so kann die Couplingliste mit entsprechenden Änderungen wiederhergestellt werden. Andernfalls wird sie weggeworfen.

6. Ausführung der DDD_PrioritySet()-Kommandos. In dieser Phase werden die nach Phase 4 verbliebenen **DDD_PrioritySet()**-Kommandos ausgeführt, sofern gemäß Regel M1 kein

DDD_XferDeleteObj()-Kommando für das gleiche Objekt erteilt wurde (in diesem Fall würde das Objekt sowieso gelöscht).

Die Ausführung beinhaltet zwei Teile: zunächst werden die Priorität des lokalen Objekts $o \in \hat{o}$ auf den neuen Wert umgesetzt und gegebenenfalls ein benutzerdefinierter Handler aufgerufen, der Seiteneffekte dieser Prioritätsänderung berücksichtigen kann. Danach werden alle Prozessoren $p \in P_{\hat{o}}$ und die *NewOwner*-Prozessoren (falls vorhanden) von der Änderung informiert; dies geschieht in Form einer *ModCpl*-Liste, die in Phase 9 an die entsprechenden Prozessoren geschickt wird.

7. Auspacken empfangener Objektnachrichten. Sobald alle Nachrichten vom *LowComm*-Modul in Empfang genommen wurden, beginnt der Auspackvorgang. Dazu gehört die Integration der empfangenen lokalen Objekte in die vorhandene Datenbasis, aber auch die Auswertung der mitgeschickten Konsistenzinformationen. In Abschnitt 3.4.4 wird dieser komplexe Vorgang genauer beschrieben.

Nach dem Auspackvorgang sind alle neuen Objekte in den lokalen Speichern etabliert, zu löschende Objekt tatsächlich entfernt und die Prioritätsänderungen durchgeführt. Was die lokalen Objekte angeht, ist der Transfervorgang also abgeschlossen. Viele Informationen über die Konsistenz der Objektkopien, also über den erforderlichen Zustand der Couplinglisten, wurden jedoch noch nicht propagiert; die Couplingkonsistenz muß erst in den folgenden beiden Phasen wiederhergestellt werden.

8. Vorbereitung der Couplingnachrichten. Die in den vorherigen Phasen erstellten Listen (d.h. *DelCpl*-Liste aus Phase 5, *ModCpl*-Liste aus Phasen 6 und 7, *AddCpl*-Listen aus Phase 7) werden sortiert und um überflüssige Einträge bereinigt. Über die zugrundeliegenden logischen Abläufe informiert Abschnitt 3.4.5.

9. Austausch/Anwendung der Couplingnachrichten. Jede der drei in Phase 8 erwähnten Listen kann als eine Liste von Kommandos an andere Prozessoren aufgefaßt werden. Diese Kommandos werden nun sortiert und wiederum mit Hilfe des *LowComm*-Moduls an die Empfängerprozessoren geschickt. Diese führen dann folgende Aktionen aus:

- für jeden *DelCpl*-Eintrag wird der entsprechende Couplinglisteneintrag gelöscht,
- für jeden *ModCpl*-Eintrag wird der entsprechende Couplinglisteneintrag modifiziert,
- für jeden *AddCpl*-Eintrag wird ein entsprechender Couplinglisteneintrag hinzugefügt.

10. Wiederherstellung der Interfaces. Durch die Auspackvorgänge für die Objekt- bzw. Couplingnachrichten wurde die verteilte Datenstruktur verändert. Dies beinhaltet nicht nur die Menge der lokalen Objekte auf jedem Prozessor, sondern natürlich auch die Couplinglisten der Objekte. Da die zuvor definierten Interfaces direkt auf den Couplinglisten aufgebaut sind, müssen diese der neuen verteilten Datenstruktur angepaßt werden. Dies geschieht durch Aufruf der Interfacefunktion **IFCreateFromScratch()**, die Interfaces auf der Basis der neuen Couplingsituation neu konstruiert.

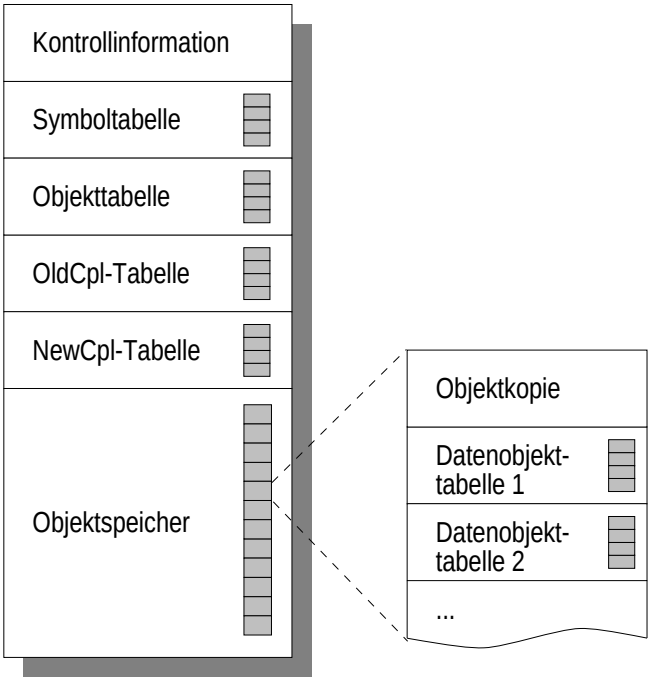


Abbildung 3.9: *Die Nachrichtenstruktur im Transfermodul.*

3.4.3 Einpackvorgang

Die Organisation der Objektnachrichten und ihre Verschickung an die korrekten Empfänger wird dem *LowComm*-Modul überlassen; die Aufgabe, die relevanten Daten in die Nachrichtenpuffer einzufüllen, verbleibt beim Transfermodul. Im folgenden sollen die Struktur der Objektnachrichten und das Einpacken der Objektdaten genauer beschrieben werden.

Die Struktur der Objektnachrichten

Abbildung 3.9 zeigt die Struktur einer Transfernachricht in schematischer Form. Neben wenigen Kontrollinformationen, die von *LowComm* zur internen Beschreibung der Nachricht benötigt werden, und einigen kleineren Tabellen machen den Großteil einer Nachricht üblicherweise die eigentlichen Objektkopien aus, die aus dem lokalen Speicher eines Prozessors direkt in die Nachricht kopiert werden. Jedem Objekt sind gegebenenfalls noch von ihm abhängige, registrierte Datenobjekte zugeordnet (vgl. Abschnitt 2.4.1).

Die vier Tabellen in jeder Nachricht haben folgende Bedeutung:

- Die *Symboltabelle* enthält eine global eindeutige Abbildungsfunktion für Referenzen innerhalb der Nachricht (vgl. Abschnitt 3.4.6).
- Die *Objekttabelle* dient als Inhaltsverzeichnis des Objektspeichers am Ende der Nachricht. Hier werden z.B. die Position der zugehörigen Objektkopie im Objektspeicher oder die Priorität des neuen lokalen Objekts abgelegt.

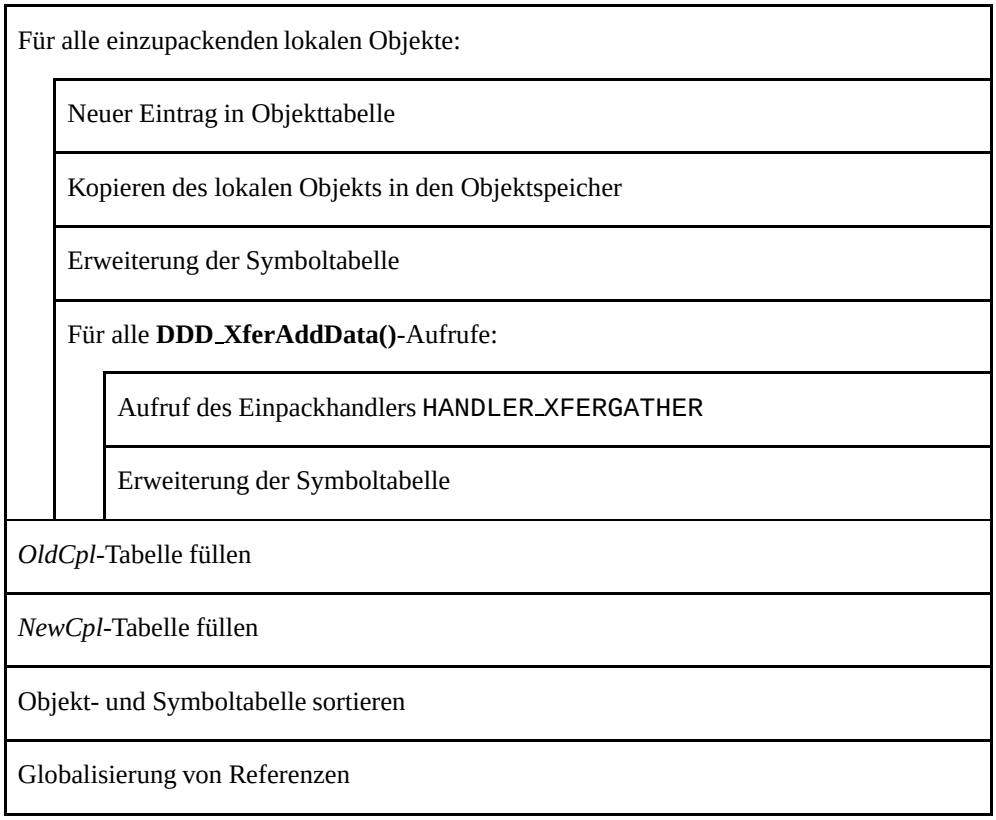


Abbildung 3.10: *Einpackvorgang einer Objektnachricht.*

- Die *OldCpl-Tabelle* enthält die in Phase 1 erzeugten *OldCpl*-Einträge, die dem *NewOwner*-Empfangsprozessor dazu dienen sollen, eine Vorabversion der Couplingliste eines neuen lokalen Objekts zu erstellen.
- Die *NewCpl-Tabelle* enthält die ebenfalls in Phase 1 erzeugten *NewCpl*-Einträge, die dem Empfangsprozessor die Entstehung von neuen lokalen Objekten auf anderen Prozessoren signalisieren sollen. Mit dieser Information aus den eingehenden Objektnachrichten kann jeder Prozessor seinen Couplinglisten neue Einträge hinzufügen.

Einpacken der Objektdaten

Das Struktogramm in Abb. 3.10 zeigt die Vorgänge beim Einpacken der Objektdaten einer einzelnen Nachricht. Für jedes in einer Transfernachricht enthaltene Objekt o_{lokal} werden folgende Schritte durchgeführt:

1. Der Objekttabelle wird ein Eintrag hinzugefügt, der als wichtigste Daten Offset und Länge des Bereichs im Objektspeicher bezeichnet, welcher die Objektkopie und die davon abhängigen

Datenobjekte enthält. Hier wird auch die Gesamtlänge der Datenobjekttabellen abgelegt sowie die Objektgröße, sofern diese von der anfangs deklarierten Objektgröße abweicht.

2. Das Objekt o_{lokal} wird aus dem lokalen Speicher in die Nachricht kopiert. Die Priorität des neu zu erzeugenden Objekts wird direkt in seine Kopie innerhalb der Nachricht eingefügt.

3. Die Symboltabelle wird erweitert (Details siehe Abschnitt 3.4.6).

4. Während der Kommandophase wurden neben Transferkommandos für eigentliche DDD-Objekte nach Bedarf auch Transfervorgänge für registrierte Datenobjekte mittels des **DDD_XferAddData()**-Kommandos angestoßen (vgl. Abschnitte 2.4.1, 2.4.3). Dabei wurden der DDD-Typ des Datenobjekts und die Anzahl der Datenobjekte je DDD-Typ als Parameter angegeben. Nun wird für jeden **DDD_XferAddData()**-Aufruf eine Datenobjekttabelle generiert, die im Objektspeicher der Nachricht direkt hinter dem zugehörigen lokalen Objekt zu liegen kommt (siehe Abb. 3.9). Dies geschieht in folgenden Schritten:

   (a) Ein benutzerdefinierter Transferkontrolle-Handler wird aufgerufen. Dies ist eine Funktion des Anwendungsprogramms, welche die zusätzlichen Datenobjekte eines DDD-Typs in die Nachricht kopiert (Transferhandler `HANDLER_XFERGATHER`).

   (b) Die Symboltabelle der Nachricht wird erweitert. Dies geschieht analog zu den transferierten DDD-Objekten wie in Abschnitt 3.4.6 beschrieben.

Aufgrund der Einbeziehung in die Symboltabelle können Datenobjekte genauso wie DDD-Objekte Referenzen auf DDD-Objekte enthalten, die beim Transfer korrekt umgerechnet werden. Diese wichtige Funktion der DDD-Bibliothek steht also zur Verfügung, obwohl kein zusätzlicher Speicher für etwaige `DDD_HEADER`-Strukturen pro Objekt belegt wird. Dies ist z.B. zur Implementierung dünnbesetzter Matrizen in numerischen Lösungsverfahren (vgl. Abschnitt 5.3.2) oder zur Repräsentation der Referenzen als eigene Klasse (vgl. Abschnitt 5.2.4) von großem Vorteil.

Als zusätzliche Funktionalitäten stehen darüberhinaus das gleichzeitige Verschicken von Datenobjekten variabler Größe und das Verschicken von ungetypten Datenströmen beliebiger Länge zur Verfügung. Damit können jedem verschickten DDD-Objekt beliebige Daten hinzugefügt werden.

Nach der Schleife über alle Objektkopien in dieser Nachricht werden die verbleibenden Tabellen (d.h. *OldCpl*-Tabelle und *NewCpl*-Tabelle) gefüllt. Dann werden die Objekt- und die Symboltabelle sortiert, um auf der Auspackseite lineare Vergleiche effizient zu ermöglichen. Danach werden die Referenzen innerhalb der Nachricht mit Hilfe der Symboltabelle globalisiert (genauer in Abschnitt 3.4.6). Sobald alle Daten eingepackt wurden, trägt das *LowComm*-Modul noch die notwendigen Kontrollinformationen ein und schickt die Nachrichten an ihre Empfänger. Da die Empfangsvorgänge bereits in einer früheren Phase nichtblockierend angestoßen wurden, stehen auf Empfängerseite bereits Puffer zur Verfügung, um die ankommenden Nachrichten aufzunehmen.

In Abb. 3.11 sind die Aktionen beim Ein- und Auspackvorgang aus Sicht eines Schichtenmodells dargestellt. Auf der Einpackseite werden zunächst die Transferhandler `HANDLER_XFERGATHER` für alle DDD-Objekte aufgerufen, denen Datenobjekte zugeordnet wurden. Danach werden die in den Nachrichten enthaltenen Referenzen globalisiert und die Nachrichten in den Kanal geschickt (waagrechter Pfeil).

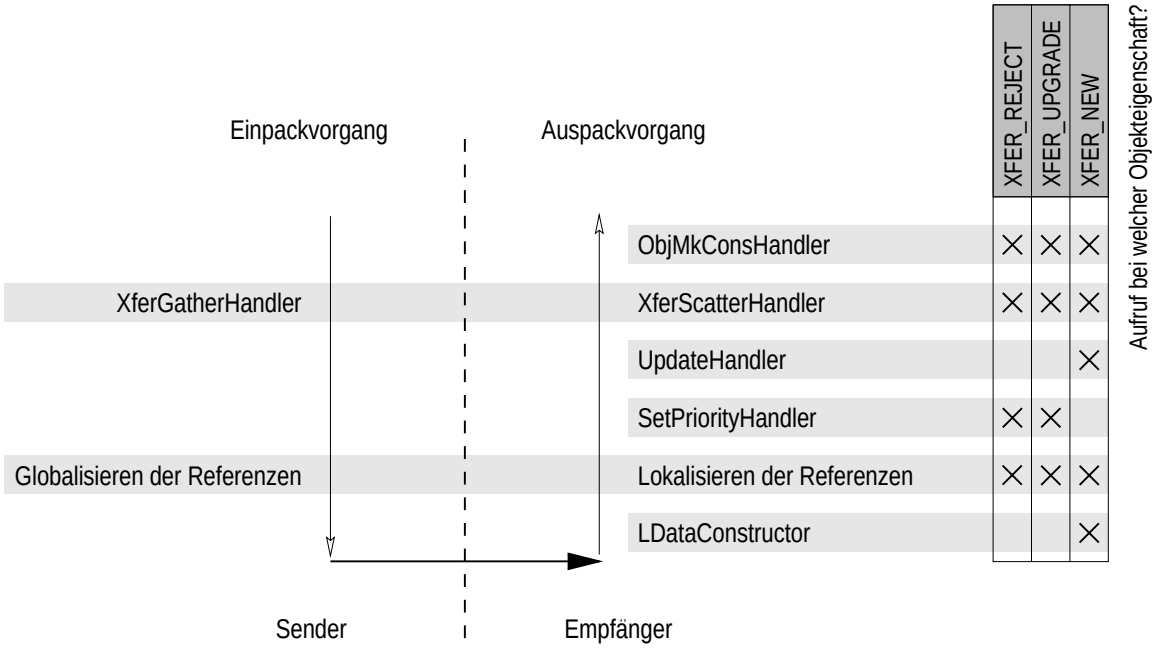


Abbildung 3.11: *Übersicht zum Transfer als Schichtenmodell zur Kommunikation; dargestellt sind wichtige Aktionen beim Ein- und Auspackvorgang. Die Tabelle (rechter Bildteil) zeigt die Bedingungen, die für eine Aktion beim Auspacken gelten müssen (siehe Abschnitt 3.4.4).*

3.4.4 Auspackvorgang

Sobald der Empfang aller Objektnachrichten abgeschlossen ist, können diese auf dem Empfängerprozessor ausgepackt werden. Die Hauptaufgaben während des Auspackens sind das Etablieren der in den Nachrichten enthaltenen Objektkopien im lokalen Speicher, das Wiederherstellen der Referenzen unter Zuhilfenahme der Symboltabellen und die (zumindest teilweise) Wiederherstellung der Konsistenz der verteilten Datenstruktur.

Integration empfangener Objekte

Für den Auspackvorgang ist zunächst ein Vorverarbeitungsschritt notwendig, im dem die *zusammengefaßte Objekttabelle* als ein Kompendium aller Inhaltsverzeichnisse der eingehenden Nachrichten erstellt wird; diese beschreibt somit die Menge aller neu zu erzeugenden lokalen Objekte. Aus dieser Objekttabelle werden alle gemäß Regel C2 (Anhang B) überflüssigen Einträge als solche markiert. Dieser Fall tritt ein, wenn das Kommando zur Erzeugung eines verteilten Objekts auf mehreren Prozessoren gegeben wurde; nach Regel C2 wird das Kommando mit der höchsten Priorität berücksichtigt.

Danach wird für jeden verbleibenden Eintrag der Transfer des lokalen Objekts aus der Nachricht in den lokalen Speicher gemäß Regel C3 durchgeführt. Die Kopierfunktion, die das Objekt aus dem Objektspeicher der Nachricht in den lokalen Adreßraum überträgt, basiert aus Effizienzgründen auf

der in der DDD-Typbeschreibung gespeicherten Kopiermaske *cmask*; wie bereits in Abschnitt 3.1.4 beschrieben wurde, werden dabei nur die `GData`-Elemente eines Objekts kopiert. Regel C3 bestimmt, wie bei Kollisionen zwischen eingehenden Objekten und bereits existierenden lokalen Objekten zu verfahren ist.

Um Kollisionen nach Regel C3 überhaupt festzustellen, werden die sortierte, zusammengefaßte Objekttabelle und eine Tabelle der lokalen Objekte jeweils nach globaler ID sortiert. Beide Tabellen werden dann eintragsweise verglichen, um mit linearem Aufwand gleiche Einträge zu finden. Bei diesem Vorgehen können lokale Objekte mit leeren Couplinglisten außer acht gelassen werden, da diese ohnehin nur auf dem empfangenden Prozessor bekannt sind und daher nicht von einem anderen Prozessor geschickt werden konnten. In der Praxis (z.B. bei numerischen Lösungsverfahren für partielle Differentialgleichungen, siehe Kap. 5) werden lokale Objekte mit nichtleeren Couplinglisten (d.h. Objekte an den Interfaces) die Ausnahme und lokale Objekte mit leeren Couplinglisten (d.h. Objekte ohne Kopien) die Regel bilden, was die effiziente Durchführbarkeit des Transfers garantiert. Die dort auftretenden geometrischen Problemgebiete werden durch Gitter diskretisiert, die in Gittergraphen mit großem Volumen, aber kleiner Oberfläche resultieren.

Bei der Bestimmung von Kollisionen nach den Regeln C2 und C3 wird jedes eingehende Objekt mit einer „Neuheits“-Markierung versehen (vgl. rechte Tabelle in Abb. 3.11):

- neue Objekte werden mit `XFER_NEW` markiert,
- überschriebene Objekte werden mit `XFER_UPDATE` markiert,
- abgewiesene Objekte werden mit `XFER_REJECT` markiert.

Nachdem alle Objekte in die bestehende Datenbasis integriert wurden, können die Symboltabellen der empfangenen Nachrichten aktualisiert werden; danach können die Referenzen der neuen lokalen Objekte korrekt eingesetzt werden (siehe Abschnitt 3.4.6).

Transferhandleraufrufe und abhängige Datenobjekte

Als nächster Schritt während des Auspackvorgangs werden die jeweils zu den Objekten gespeicherten Datenobjekttabellen ausgewertet. Dazu werden die einzelnen Datenobjekte zunächst innerhalb des Nachrichtenpuffers mit dem Umrechnungsmechanismus aus Abschnitt 3.4.6 behandelt, um die in ihnen enthaltenen Symboltabellenindizes ebenfalls in lokale Referenzen umzuwandeln. Danach wird dann wiederum eine benutzerdefinierte Handlerfunktion aufgerufen, welche die jetzt mit korrekten Referenzen auf lokale Objekte ausgestatteten Datenobjekte in die bereits teilweise etablierte Datenstruktur hinzufügen soll (Transferhandler `HANDLER_XFERSCATTER`).

Für die empfangenen Objekte werden nun – je nach ihrer Behandlung in den Regeln C2 und C3 – weitere Transferhandler aufgerufen, die es der Anwendung erlauben, auf die möglichen Änderungen der Datenstruktur mit entsprechenden Seiteneffekten differenziert zu reagieren; ihre Aufrufreihenfolge und die Bedingungen ihrer Aufrufe ist in Abb. 3.11 dargestellt (weitere Details siehe [19]).

Auswertung der OldCpl- bzw. NewCpl-Tabellen

Aus den beiden Couplingtabellen, die vorab in den Objektnachrichten versendet wurden, können bereits wichtige Informationen gewonnen werden; der Zustand allgemeiner Konsistenz der verteilten Objekte und Couplinglisten wird jedoch erst nach Phase 9 erreicht. Abschnitt 3.4.5 beschäftigt sich genauer mit der Erhaltung der Couplingkonsistenz und damit der verteilten Objekte.

3.4.5 Couplingkonsistenz

Die Auswirkungen der Transferkommandos **DDD_XferDeleteObj()** und **DDD_PrioritySet()** auf einem Prozessor p beziehen sich jeweils auf nur ein lokales Objekt $o \in \hat{o}$. Damit die Couplingliste des zugehörigen verteilten Objekts $\hat{o}$ konsistent bleibt, würde eine Nachricht an alle Prozessoren $q \in P_{\hat{o}}$ mit $q \neq p$ genügen. Das dritte Transferkommando **DDD_XferCopyObj()** erlaubt es jedoch, neue lokale Objekte auf bisher nicht beteiligten (d.h. *NewOwner*-) Prozessoren $p' \notin P_{\hat{o}}$ zu erzeugen. In der dadurch erzeugten Prozessormenge $P'_{\hat{o}} \supset P_{\hat{o}}$ reicht ein Kommunikationsschritt im allgemeinen nicht aus, um alle beteiligten Prozessoren $p \in P'_{\hat{o}}$ über alle Änderungen zu informieren, was zu inkonsistenten Couplinglisten führt (Beispiele siehe Anhang D.1).

Um obigem Problem zu begegnen, wurde den Objektnachrichten eine zweite Kommunikationsphase (Phase 9) nachgeschaltet, in der alle Prozessoren direkt auf die Couplinglisten anderer Prozessoren Einfluß nehmen können. Dazu werden aufgrund von bestimmten Ereignissen entweder sofortige Änderungen der lokalen Couplinglisten durchgeführt oder Kommandos zur Anpassung fremder Couplinglisten generiert. In der folgenden Liste werden die Ereignisse und ihre Folgen im einzelnen beschrieben.

- Aus den *OldCpl*-Einträgen werden die Couplinglisten neuer Objekte konstruiert; da Entscheidungen dritter Prozessoren in diesem Stadium noch nicht bekannt sind, werden die so erzeugten Couplinglisten im allgemeinen noch nicht korrekt sein.

- In Phase 5 wurden die Couplinglisten für gelöschte lokale Objekte in Form einer *DelCpl*-Liste zwischengespeichert. Falls ein anderer Prozessor die Erzeugung eines lokalen Objekts desselben verteilten Objekts veranlaßt hat, wird die Couplingliste wieder rekonstruiert (entsprechend Regel M3).

- Jeder *NewCpl*-Eintrag, der sich auf ein empfangenes Objekt bezieht, wird in einen Couplinglisteneintrag umgesetzt. Ebenso wird jeder *NewCpl*-Eintrag behandelt, der sich auf ein bereits vorhandenes Objekt bezieht; dies geschieht als Ausführung der in Phase 1 angestoßenen Konsistenzaktionen.

  Hierbei muß für mehrere *NewCpl*-Einträge an ein bereits vorhandenes Objekt verteilt eine konsistente Entscheidung gemäß Regel C2 getroffen werden, Abb. 3.12 zeigt ein Beispiel: Das verteilte Objekt $\hat{o} = \{o_1, o_2, o_3\}$ sei vor dem Transfer auf den Prozessoren $P1$, $P2$ und $P3$ gespeichert. Beim Transfer senden P2 und P3 jeweils ihr lokales Objekt an Prozessor $P4$. Gleichzeitig wird in Phase 1 je ein *NewCpl*-Eintrag erzeugt und an Miteigentümer P1 gesendet. Prozessor P4 endscheidet gemäß Regel C2 die Annahme von o_2 (wegen höherer Priorität); diese Entscheidung muß P1 aus den beiden *NewCpl*-Einträgen rekonstruieren.

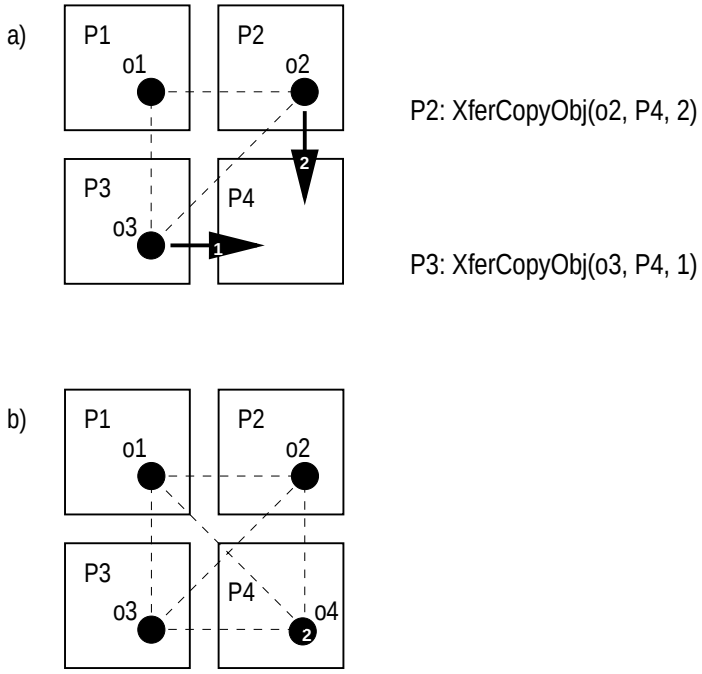


Abbildung 3.12: *Beispiel zur verteilten Erzeugung von konsistenten Couplinglisten.* **DDD_XferCopyObj()***-Kommandos beim Transfervorgang (a) und notwendige Couplinginformation nach dem Transfer (b). Gestrichelte Linien deuten Couplinglisteneinträge an.*

- Falls ein Prozessor ein lokales Objekt verschickt hat, dessen Priorität sich durch die Annahme eines soeben empfangenen lokalen Objekts erhöhte, muß dieser Prozessor den Empfänger nachträglich über diese Änderung informieren. Dies geschieht über das Anstoßen einer Couplingänderung durch Erzeugung eines Eintrags in die *ModCpl*-Liste.
- Für **DDD_PrioritySet()**-Kommandos, die nicht nach Regel M1 durch ein **DDD_XferDeleteObj()**-Kommando ausgelöscht wurden, wird die Couplingänderung auf entfernten Prozessoren durch Erzeugung eines Eintrags in die *ModCpl*-Liste ausgelöst.
- Für **DDD_XferDeleteObj()**-Kommandos, die nicht nach Regel M3 durch erneutes Empfangen eines lokalen Objekts ausgelöscht wurden, wird die Löschung des Couplingeintrags auf entfernten Prozessoren durch Erzeugung eines Eintrags in die *DelCpl*-Liste ausgelöst.

Aus den generierten Tabellen (d.h. *AddCpl*, *ModCpl* und *DelCpl*) werden schließlich Nachrichten erstellt, mit Hilfe des *LowComm*-Moduls verschickt und von den Empfängern lokal ausgeführt (Phase 9). Danach ist die gesamte verteilte Datenstruktur wieder konsistent.

3.4.6 Transfer von Referenzen

Das letzte Teilproblem, das hier gesondert beschrieben wird, ist der korrekte Transfer von Referenzen zwischen DDD-Objekten. Diese sind im Normalbetrieb nur im lokalen Adreßraum gültig und

müssen deshalb beim Einpacken in die Objektnachricht auf die globalen IDs der referenzierten Objekte abgebildet werden. Beim Auspacken der Nachricht muß der umgekehrte Vorgang stattfinden.

Erstellung der Symboltabelle auf Einpackseite

Während des Einpackvorgangs (Abschnitt 3.4.3) wird zu jeder Objektnachricht eine Symboltabelle erstellt. Dabei wird zu jeder vom Objekt o_{lokal} ausgehenden Referenz

$$o_{\text{lokal}} \overset{\text{ref}}{\Leftrightarrow} o_{\text{ref}}$$

auf ein Objekt o_{ref} ein Symboltabelleneintrag erzeugt, der neben der globalen ID von o_{ref} auch einen Verweis auf die Stelle in der Nachricht enthält, an der das zugehörige Strukturelement vom Typ `ObjPtr` $\in E$ in der Objektkopie von o_{lokal} zu liegen kommt. Damit kann von jedem Eintrag der Symboltabelle die Kopie der zugehörigen Referenz adressiert und manipuliert werden.

Die globale ID des referenzierten Objekts o_{ref} erhält man dabei wie folgt: Bei der Definition des DDD-Typs von o_{lokal} (vgl. *TypeManager*, Abschnitt 3.1.4) wurde als *reftype*-Komponente der Elementbeschreibungsstruktur der DDD-Typ von o_{ref} angegeben. Damit kann aus der Typbeschreibungstabelle der Offset der `DDD_HEADER`-Struktur im Objekt o_{ref} bestimmt werden; aus dieser Struktur läßt sich schließlich die globale ID entnehmen. Die Möglichkeit eines variablen `DDD_HEADER`-Offsets bedarf also wegen obiger Ableitung einer exakten Definition der DDD-Typen aller Referenzen.

Globalisieren der Referenzen vor dem Senden

Bevor die Objektnachricht verschickt werden kann, müssen alle Referenzen globalisiert werden, dabei werden die bisher nur im lokalen Adreßraum gültigen Zeiger auf DDD-Objekte durch deren globale ID ersetzt. Abb. 3.13 zeigt diesen Vorgang am Beispiel. Nach der Abarbeitung aller Objekte nach dem obigen Schema enthält die Symboltabelle einen Eintrag für jede Referenz innerhalb der Nachricht. Abb. 3.13a zeigt ein Objekt o_1 innerhalb der Nachricht, das über sein Element ref_1 das lokale Objekt o_2 referenziert. Entsprechend gibt es einen Symboltabelleneintrag, der die Abbildung von der globalen ID von o_2 (also gid_2) auf die Speicherstelle des lokal gültigen Zeigers repräsentiert. Der *ref*-Eintrag der Symboltabelle zeigt also auf die Speicherstelle innerhalb der Nachricht, an der sich die Referenz ref_1 befindet.

Nun wird die im vorigen Schritt konstruierte Symboltabelle nach ihrem *gid*-Eintrag sortiert, damit die Zuordnung auf der Empfängerseite effizient durchgeführt werden kann (vgl. 3.4.4). Diese Sortierung hat im mittleren Fall die Komplexität $O(n\log(n))$, wobei n durch die Anzahl der Referenzen in der Symboltabelle gegeben ist.

Die eigentliche Globalisierung besteht darin, daß alle Referenzen durch den Index in der Symboltabelle ersetzt werden. Da auch auf der Empfängerseite der DDD-Typ jedes empfangenen Objekts (im Beispiel o_1) und damit die Struktur der `ObjPtr`-Elemente bekannt ist, kann dort aus der globalen ID in der Symboltabelle der neue, gültige Zeiger auf das referenzierte Objekt bestimmt werden, falls dieses überhaupt existiert. Im Beispiel (Abb. 3.13b) wird also der Zeiger im Element ref_1 durch den

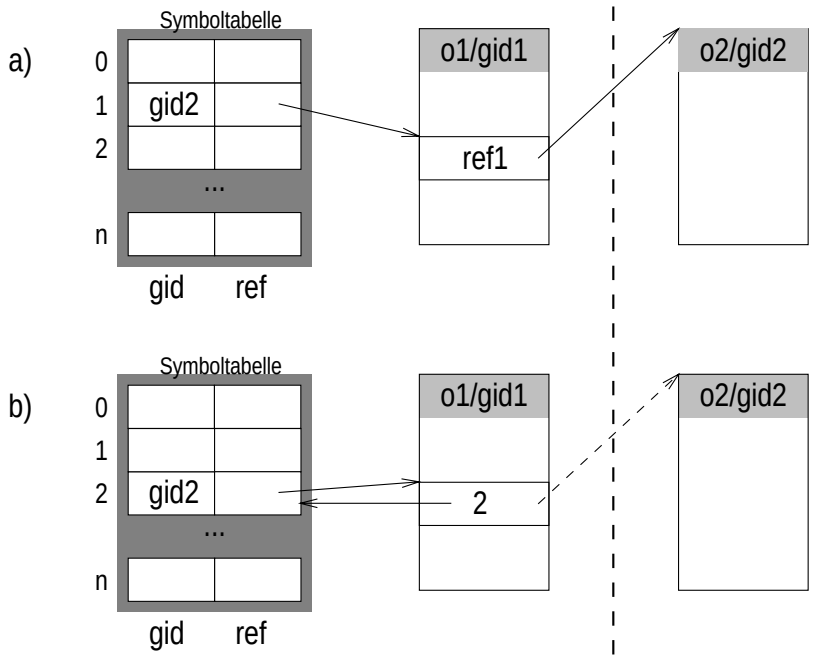


Abbildung 3.13: *Beispiel einer Objektreferenz als Eintrag in die Symboltabelle auf der Senderseite, die gestrichelte Linie trennt den Nachrichtenpuffer (links) vom lokalen Speicher (rechts). Eintrag eines Zeigers auf die referenzierende Stelle und der globalen ID des referenzierten Objekts (a); danach Sortierung der Symboltabelle nach globaler ID und Umsetzung der Referenz auf den Index der Symboltabelle (b).*

Symboltabellenindex 2 ersetzt. Der Ersetzungsvorgang kann effizient durch einmaliges Durchlaufen der Symboltabelle durchgeführt werden, da ihr *ref*-Eintrag genau die Stelle angibt, wo der Index eingetragen werden muß.

Lokalisieren der Referenzen auf der Empfangsseite

Nachdem alle ankommenden Objekte in den lokalen Speicher übertragen wurden, müssen die Symboltabellen in den empfangenden Nachrichten bearbeitet werden. Dazu wird zu jeder globalen ID aus einem Symboltabelleneintrag das zugehörige lokale Objekt gesucht; sofern dieses vorhanden ist, wird im *ref*-Eintrag der Symboltabelle ein Zeiger auf dieses Objekt gespeichert. Die Menge der lokalen Objekte, die überhaupt als referenzierte Objekte in Frage kommen, beschränkt sich auf lokale Objekte mit nichtleeren Couplinglisten (siehe S. 89) unter Hinzunahme der soeben empfangenen und neu erzeugten Objekte (vgl. Abschnitt 3.3).

Nachdem die Symboltabelle einer Nachricht komplettiert wurde, können die bereits etablierten lokalen Objekte dieser Nachricht mit den korrekten Referenzen versehen werden. Abb. 3.14 zeigt diesen Vorgang am Beispiel aus Abb. 3.13. Das lokale Objekt o_1 wurde bereits im Speicher des empfangenden Prozessors etabliert, statt der korrekten Referenz auf das lokale Objekt o_2' mit der globalen ID gid_2 enthält es aber noch den Symboltabellenindex 2 (Abb. 3.14a). In den *ref*-Eintrag der Sym-

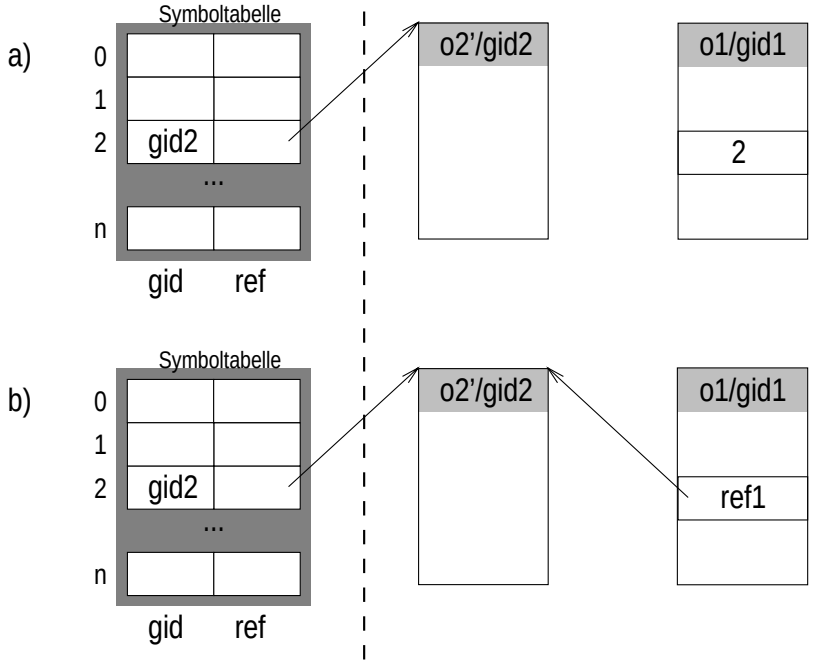


Abbildung 3.14: *Beispiel zur Lokalisierung einer Objektreferenz auf der Empfängerseite, die gestrichelte Linie trennt den Nachrichtenpuffer (links) vom lokalen Speicher (rechts). (a) Zuordnung einer globalen ID in der Symboltabelle zu einem vorhandenen Objekt o_2'; (b) danach Umsetzung des Symboltabellenindex im neuen Objekt o_1 auf die gültige Referenz.*

boltabelle wurde aber bereits der Zeiger auf das hier tatsächlich vorhandene Objekt o_2' eingetragen. Im Lokalisierungsschritt werden nun alle Index-Einträge in den neuen Objekten durch den entsprechenden Zeiger ersetzt, sofern das referenzierte Objekt lokal existiert. Im Beispiel (Abb. 3.14b) wird also als Referenz ref_1 der Zeiger auf o_2' aus der Symboltabelle übertragen, wodurch die Situation auf der Senderseite wiederhergestellt ist (siehe Abb. 3.13a). Der *ref*-Eintrag der Symboltabelle wird also auf der Einpackseite als Zeiger auf eine Referenz und auf der Auspackseite als Zeiger auf ein referenziertes Objekt benutzt.

Wird nach Regel C3 ein existierendes lokales Objekt mit einem ankommenden Objekt vereinigt, so dürfen keine Referenzen verlorengehen. Die bereits existierenden Referenzen bleiben gültig, neue Referenzen werden zusätzlich eingetragen. In Abbildung 3.15 sind die fünf Möglichkeiten für das verteilte Objekt $\hat{o}$ mit $\{o_1, o_2, o_3\} \subset \hat{o}$ in einem Beispiel dargestellt:

1. Beide Referenzen sind nicht gültig. Nicht gültige (Null-)Referenzen (d.h. Referenzen auf $\oslash$ bzw. *null*, siehe Def. 6 auf S. 40) verweisen auf Objekte, die lokal nicht vorhanden sind (es gibt evtl. Objekte $o \in \hat{o}$, bei welchen diese Referenz tatsächlich auf ein Objekt verweist). In diesem Fall entsteht wieder eine nicht gültige Referenz.

2. Nur die ankommende Referenz ist gültig. Das referenzierte Objekt o' war zum Zeitpunkt des Empfangs von o_1 noch nicht vorhanden, wurde jedoch in der Zwischenzeit etabliert. Die Referenz wird auf dieses Objekt gesetzt.

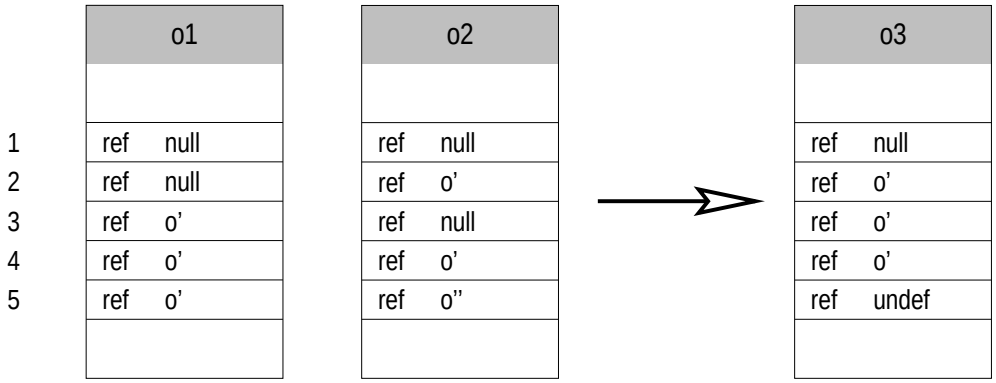


Abbildung 3.15: *Überlagerung von Referenzen auf der Empfängerseite. Objekt o_1 existiert bereits, während Objekt o_2 transferiert wurde. Aus beiden soll ein Objekt o_3 durch Überlagerung erzeugt werden.*

3. **Nur die existierende Referenz ist gültig.** Auf dem Sendeprozessor war zum Zeitpunkt des Sendens von o_2 kein lokales Objekt $o \in \mathtt{dobj}(o')$ vorhanden, deshalb enthält o_2 keine gültige Referenz. Das Objekt o' ist jedoch auf dem Empfangsprozessor vorhanden, also muß die Referenz in o_3 erhalten bleiben.

4. **Beide Referenzen verweisen auf das gleiche Objekt o'.** Die erzeugte Referenz wird ebenfalls auf dieses Objekt gesetzt.

5. **Die Referenzen verweisen auf verschiedene Objekte.** Formal darf dieser Fall nicht eintreten und löst daher einen Laufzeitfehler aus. Optional kann auch eine der beiden Referenzen (stets die ankommende oder die bereits existierende) ausgewählt werden.

Der Mechanismus zur Globalisierung/Lokalisierung von Referenzen arbeitet natürlich für beliebig komplexe Fälle, wie z.B. bei verteilter Speicherung von Referenzen mit Transfer zwischen disjunkten Paaren von Prozessoren; er ist damit die direkte algorithmische Umsetzung der Definition der verteilten Referenzrelation (Def. 11, Kap. 2). Eine effiziente Umsetzung wird ermöglicht, indem von allen beteiligten Objekt- bzw. ID-Mengen sortierte Listen vorgehalten werden, die geeignet simultan durchlaufen werden können. Dies stellt in jedem Fall lineare Ausführungskomplexität sicher.

3.4.7 Nachbemerkungen

Lokale vs. globale Referenzen

Der programmtechnische Aufwand zur Realisierung des Transfermoduls ist hoch und macht ca. ein Drittel des gesamten Programmcodes der DDD-Bibliothek aus. Ein Großteil davon resultiert aus der Vermeidung von globalen Referenzen; dies macht aber gerade den besonderen Vorteil des Parallelisierungskonzepts aus: eine Implementierung von globalen Referenzen hätte nämlich die Ergänzung *jeder Referenz* durch die referenzierte Prozessornummer zur Folge, was vor allem aus Speicherplatzgründen in der Praxis oftmals nicht tragbar wäre. In den weiter unten beschriebenen numerischen Anwendungen auf unstrukturierten Gittern (Kap. 5) stellen C-*pointer* (also Referenzen in

Form von Speicheradressen) einen erheblichen Anteil am gesamten Speicherbedarf des jeweiligen Programms dar, dies würde bei zusätzlicher Speicherung der referenzierten Prozessornummer wie z.B. in *CHAOS++* [34, 35] vorgesehen den Bedarf an Hauptspeicher drastisch erhöhen. Außerdem kann aus implementierungstechnischen Gründen der Speicher für die Objekte, welche nur lokale Referenzen enthalten, höchstens mit erheblichem Laufzeitverlust dynamisch reduziert werden. Bei jeder Referenzierung müßte dann abgefragt werden, ob diese lokal oder global geschehen muß.

Im DDD-Modell dagegen sind Referenzen stets lokal, die Kopplungen zwischen den Adreßräumen finden über die Verknüpfung von lokalen Objekten statt. So kann je nach der zugrundeliegenden Architektur und nach der Compilerumgebung stets die effizienteste verfügbare Implementierung für Referenzen gewählt werden.

Iterativer Lasttransfer

Bereits in [10] wurde beschrieben, daß ein Transfervorgang für große Datenmengen bei erheblichen topologischen Veränderungen des verteilten Graphen sehr speicherintensiv sein kann. Sobald der verfügbare Speicher auf jedem Prozessor zu ungefähr einem Drittel verbraucht ist, die Umverteilung sich aber auf den kompletten Graphen erstreckt, wird für die zu sendenden Nachrichtenpuffer mindestens das zweite und für die zu empfangenden Nachrichtenpuffer auch noch das dritte Speicherdrittel verwendet. Bezieht man in diese Abschätzung ebenfalls die Hilfsstrukturen (z.B. Symboltabellen) mit ein, so reicht der lokale Speicher bald nicht mehr aus.

In diesem Fall ist es unabdingbar, das Auftragsvolumen eines Transfervorgangs in kleinere Teilaufträge zu zerlegen. In [10] war die parallele Anwendung vom Lasttransferalgorithmus nicht durch eine abstrakte Schnittstelle getrennt, so daß zur Auswahl der Kommandos in jedem Teilauftrag Informationen aus beiden Programmteilen verwendet werden konnten. Die DDD-Bibliothek muß jedoch ohne zusätzliches Wissen über die Struktur des zu verteilenden Graphen eine Aufteilung vornehmen können.

Der iterative Lasttransfer wurde in der aktuellen DDD-Programmbibliothek noch nicht realisiert; derzeit kann die Aufteilung des Transfervolumens nur auf Anwendungsebene stattfinden. Dazu muß das Anwendungsprogramm die DDD-Transferkommandos eines Transfervorgangs selbständig derart in Teilmengen aufspalten, daß nach jedem Teiltransfer wieder ein konsistenter Graph entsteht. Dies ist jedoch mit zusätzlichem Aufwand an Laufzeit verbunden, da zur Herstellung von Graphkonsistenz in den Zwischenstadien zusätzliche Kommunikation und Berechnung notwendig ist (z.B. müssen alle DDD-Interfaces nach jedem Teilschritt wieder konsistent hergestellt werden).

Als Erweiterung der derzeitigen DDD-Transferfunktionalität ist daher ein transparenter, iterativer Transfervorgang vorgesehen. Dazu werden zunächst alle Daten verschickt, die zur Konsistenz des gesamten Transfervorgangs nötig sind (z.B. Symboltabellen). Danach wird über die Menge an zu verschickenden lokalen Objekten iteriert (der Objektspeicher hat den größten Anteil an der Nachrichtengröße im Transfer); in jeder Iteration wird eine Teilmenge von zu verschickenden Objekten eingepackt, verschickt und im Zielspeicher etabliert. Der Speicher für Nachrichtenpuffer kann so öfter verwendet werden. Gleichzeitig werden lokale Objekte gelöscht, wo dies erforderlich und im Rahmen des iterativen Vorgehens möglich ist. Abschließend wird die Couplingkonsistenz gemäß Abschnitt 3.4.5 wiederhergestellt.

Kapitel 4

Leistungsmerkmale zur Implementierung

In diesem Kapitel werden Aussagen zu Leistungsmerkmalen der im vorigen Kapitel beschriebenen Implementierung der DDD-Bibliothek zusammengefaßt. Dazu wurden Messungen zu Laufzeitverhalten und Effizienz für anwendungsorientierte Szenarien auf verschiedenen Parallelrechnerarchitekturen durchgeführt, die so konstruiert wurden, daß daraus aussagekräftige Kenngrößen abgeleitet werden können. Einige der Messungen werden darüberhinaus mit Modellen zur Implementierung verglichen. Hierbei soll auf eine Fülle von Details verzichtet und stattdessen einige für den DDD-Anwender wichtige Grundregeln herausgearbeitet werden, wiederum thematisch nach den einzelnen DDD-Modulen geordnet.

Zur Durchführung der Messungen wurde eigens eine Meßapplikation entwickelt, die ebenso portabel wie die DDD-Bibliothek sofort auf neue Architekturen gebracht und dort zur Effizienzbewertung benutzt werden kann. Diese Anwendung wurde in eine Meßumgebung eingebettet, die es erlaubt, entweder gezielt einzelne Messungen durchzuführen oder eine komplette Benchmarksuite samt Auswertung und Berechnung von abgeleiteten Kenngrößen ablaufen zu lassen.

4.1 Verwendete Parallelrechner

Die nachfolgenden Meßreihen und Experimente wurden auf einer Auswahl verschiedener Parallelrechnerplattformen durchgeführt. An dieser Stelle soll jedoch kein ausführlicher Vergleich von konkurrierenden Rechnersystemen präsentiert, sondern ein Eindruck des Verhaltens der vorliegenden DDD-Implementierung gegeben werden. Als Hardwaregrundlagen für die folgenden Messungen wurden folgende Rechner eingesetzt:

- Cray T3E. 512 Rechenknoten mit jeweils einem DEC-Alpha-EV5-Prozessor (300 MHz, 600 MFlops, 1200 MIPS) und 128 MByte Hauptspeicher (Speicherbandbreite $1.2\frac{\text{GB}}{\text{sec}}$). Cache-Kohärenz-Protokoll in Hardware. Kommunikationsnetzwerk als 3D-Torus, Virtual-Cut-Through-Routing, (Einzelverbindung mit Bandbreite $500\frac{\text{MB}}{\text{sec}}$ bidirektional). Betriebssystem UNICOS/mk (mit CHORUS-Microkernel). Message-Passing-Modell MPI, Kennzeichnung: CRAYT3E.

- Hitachi SR2201. 32 Rechenknoten mit jeweils einem HARP-1E (Hitachi Advanced RISC Processor; 150 Mhz, 300 MFlops) und 256 MByte Hauptspeicher (Bandbreite Prozessor/Speicher $1.2\frac{\text{GB}}{\text{sec}}$). Verbindungsnetzwerk als 2D- bzw. 3D-Torus über Crossbar (Bandbreite $300\frac{\text{MB}}{\text{sec}}$ je Prozessor). Betriebssystem Hi-UX/MPP (mit Mach 3.0 Microkernel) auf Rechenknoten, OSF/1 auf sonstigen Knoten. Message-Passing-Modell MPI, Kennzeichnung: HITACHI.
- NEC SX4. 1 Rechenknoten mit 32 SX-4 Vektorprozessoren (jew. 125 MHz, 2 GFlops Vektorleistung in 8 Pipelines, 250 MFlops skalare Leistung), und 8 GByte gemeinsamer Hauptspeicher (SSRAM) über Crossbar (Bandbreite Prozessor/Speicher $16\frac{\text{GB}}{\text{sec}}$). Betriebssystem Super-UX. Message-Passing-Modell MPI-SX (PThreads), Kennzeichnung: NECSX4.
- Intel Paragon. 113 Rechenknoten mit jeweils zwei i860XP-Prozessoren (einer für Rechenleistung, einer als Kommunikationsprozessor; jeweils 50 MHz, 75 MFlops, 50 MIPS) und 32 bzw. 64 MByte Hauptspeicher (Speicherbandbreite $400\frac{\text{MB}}{\text{sec}}$). Kommunikationsnetzwerk als 2D-Gitter, Wormhole-Routing (Einzelverbindung mit 25μsec Startupzeit und Bandbreite von $200\frac{\text{MB}}{\text{sec}}$). Betriebssystem OSF/1 (basierend auf Mach Microkernel). Message-Passing-Modell NX, Kennzeichnung: PARAGON.

4.2 Kommunikation mittels PPIF

Szenario

Als Grundlage der DDD-Funktionalität dient die Schnittstelle zum *message-passing*-Modell des Parallelrechners; diese wurde in Form der PPIF-Kommunikationsschicht gekapselt (vgl. Anhang A.1). In diesem Abschnitt werden nun einige Leistungsmerkmale der PPIF-Kommunikationsprimitive für die im weiteren Verlauf dieses Kapitels betrachteten Plattformen beschrieben: dies sind blockierende Kommunikationsprimitive (*SendSync/RecvSync* für synchrone Kommunikation) sowie nichtblockierende Kommunikationsprimitive (*SendASync/RecvASync* für asynchrone Kommunikation).

Die Prozessoren werden dabei entsprechend ihrer logischen Nummer in einem Ring angeordnet, jeder Prozessor erhält je einen Kanal zum linken und rechten Nachbarprozessor im Ring. Dann wird eine Nachricht durch den Ring gereicht, wobei im Fall der nichtblockierenden Kommunikationsprimitive sofort nach der Ausführung der Sende- bzw. Empfangsaufrufe auf Freigabe des Kanals gewartet wird. Die Kommunikationsleistung wird also unidirektional gemessen. Beim Aufbau des Szenarios wird davon ausgegangen, daß die Entfernung von je zwei Prozessoren im Kommunikationsgraphen vernachlässigt werden kann; gängige Routing-Techniken (z.B. *cut-through routing* [91, S. 45]) legen dies nahe. In der Praxis sollten jedoch die vordefinierten Kommunikationsstrukturen der PPIF-Schnittstelle benutzt werden, da dann ein möglichst optimales *Mapping* der Kommunikationstopologie auf die Hardware erfolgen kann, wo dies möglich ist.

Messungen

Aus den Parametern *Anzahl der Ringdurchläufe* n_{cycles} und *Prozessorzahl P* kann die durchschnittliche Zeit t_{hop} pro einzelner Kommunikation ermittelt werden; dabei werden die Parameter derart

Tabelle 4.1: *Aufsetzzeit und Durchsatz der PPIF-Kommunikationsschicht. auf verschiedenen Architekturen, synchrone und asynchrone Kommunikation.*

Architektur	synchron		asynchron	
	t_s [μsec]	b_{net} [$\frac{MB}{sec}$]	t_s [μsec]	b_{net} [$\frac{MB}{sec}$]
CRAYT3E	30.4	277.0	24.3	77.7
HITACHI	48.7	208.2	67.7	208.2
NECSX4	506.9	3198.6	1489.0	378.3
PARAGON	49.5	69.0	57.8	69.3

eingestellt, daß alle Startup-Effekte (z.B. Speicherallokierung) verschwinden. In jeweils einer Meßreihe wird die Nachrichtengröße variiert und daraus die Aufsetzzeit pro Nachricht t_s (*startup* oder *latency*) und der Durchsatz des Verbindungsnetzwerks b_{net} (*bandwidth* oder *throughput*, mit der Transferzeit pro Byte $t_{byte} = 1/b_{net}$) bestimmt. Tabelle 4.1 gibt t_s und b_{net} der synchronen bzw. asynchronen Kommunikation für die getesten Plattformen wieder ($n_{cycles} = 200, P = 16$).

Diskussion

Sofern der jeweiligen PPIF-Implementierung wieder ein *message-passing*-Modell zugrundeliegt (z.B. die NX-Schicht auf PARAGON), würde man durch direkte Benutzung dieser Schicht eine kleine Leistungssteigerung erzielen. Der Overhead durch PPIF-Benutzung ist jedoch sehr gering (jeweils ca. ein Funktionsaufruf und Setzen von Kontrollvariablen) und beträgt stets weniger als 2 % für t_s bzw. 1 % für b_{net} der in Tabelle 4.1 angegeben Werte.

Die Implementierung der MPI-SX-Schicht für die NECSX4-Architektur basiert auf Posix-Threads. Der hohe Durchsatzwert b_{net} der synchronen Kommunikation stimmt mit direkten Messungen von MPI-SX-Funktionen überein; durch den gemeinsamen Hauptspeicher können die *message-passing*-Funktionen auf Kopiervorgänge im SSRAM abgebildet werden. Die gemessenen Aufsetzzeiten und der Durchsatz im asynchronen Fall sind (verglichen mit Messungen der Kommunikationsbandbreite auf zwei Prozessoren) eher schlecht; durch die threadbasierte Implementierung hängen die Aufsetzzeiten und b_{net} (asynchron) von der Strategie des Thread-Schedulings und der Prozessorsynchronisation ab, was sich im gemessenen Szenario (Ring) negativ auswirkt. Bei praktischen Anwendungen ist jedoch eher mit den hier gemessenen Werten zu rechnen als mit einem einfachen 2-Prozessor-Benchmark.

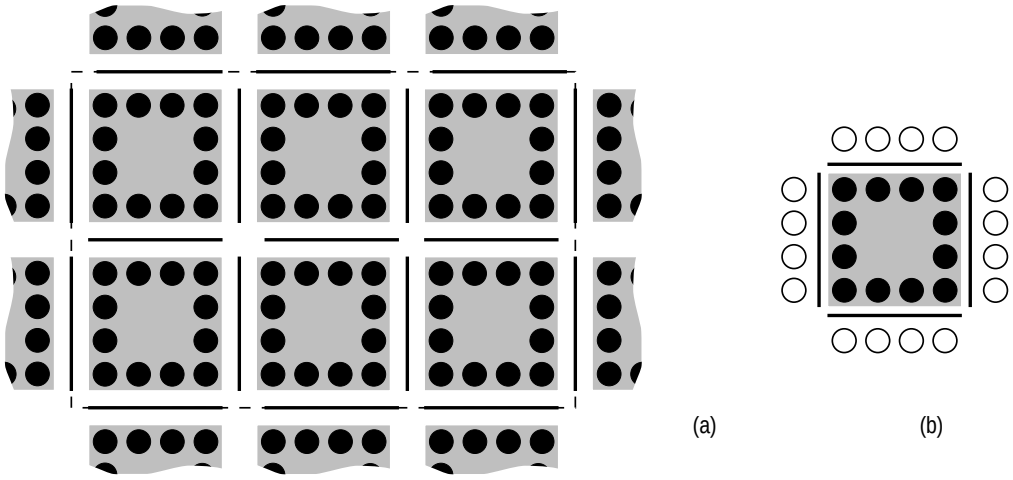


Abbildung 4.1: *(a) Beispiel einer Torus-Struktur auf sechs Prozessoren, mit $P_x = 3$, $P_y = 2$, $n = 4$ und damit $l_p = 16$. (b) Gitter eines Prozessors mit einer Überlappungsreihe. Die Interfaces sind jeweils durch Balken zwischen den Gebieten hervorgehoben.*

4.3 Performance der DDD-Interfaces

4.3.1 Kommunikation über Interfaces

Szenario

Als Szenario zur Messung der Kommunikationsleistung über DDD-Interfaces wird ein regelmäßiges zweidimensionales Gitter aus einheitlichen Objekten (*Knoten*) gewählt, das zu einem Torus geschlossen wird. Die Aufteilung auf eine variierbare Anzahl von Prozessoren wird analog durchgeführt, so daß die (virtuelle) Topologie der Prozessoren ebenfalls einen 2D-Torus bildet.

Der Prozessorgraph habe also die Größe $P_x \times P_y$, wobei jeder beteiligte Prozessor einen quadratischen Gitterausschnitt der Seitenlänge n Knoten erhält, was zu einer Gesamtlänge des Interfaces pro Prozessor von $l_p = 4n$ unidirektional und somit $8n$ bidirektional führt (Abb. 4.1). Die inneren Knoten jedes Prozessors haben keinen Einfluß auf die Kommunikationsleistung der Interfaces und müssen daher in diesem Benchmark weder erzeugt noch gespeichert werden.

Folgende Parameter werden in die Messungen einbezogen:

- Übertragene Information pro Knoten s_n (in byte).
- Gesamtlänge des Interfaces pro Prozessor l_p.

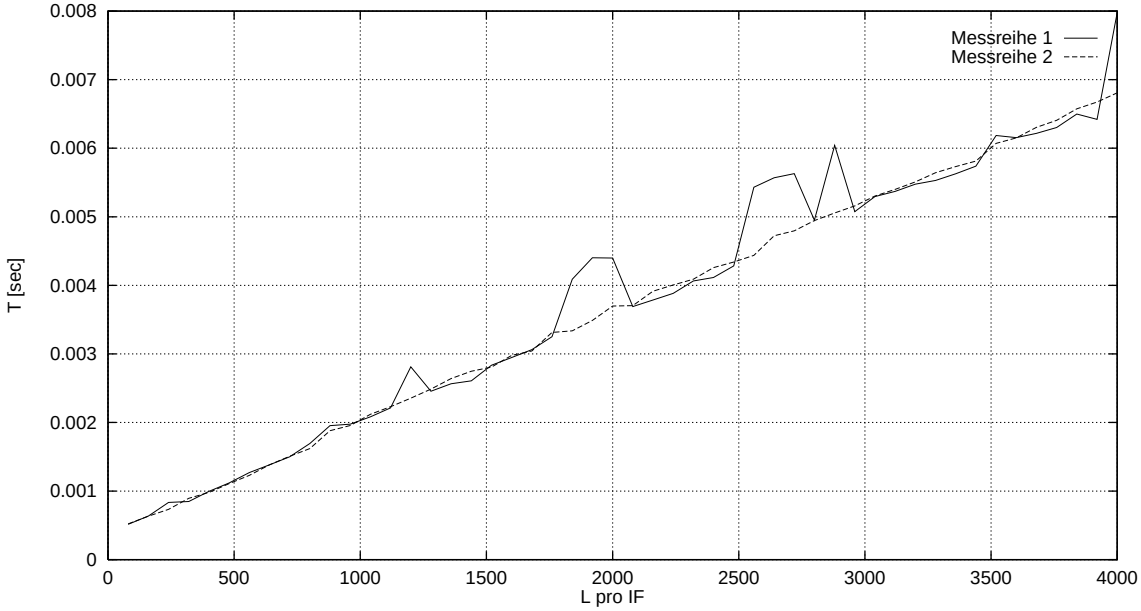


Abbildung 4.2: *Meßreihen zur Interface-Kommunikation: Meßprogramm als eine von mehreren, gleichzeitigen Anwendungen (1) und als Einzelanwendung (2).* (PARAGON, 16 Prozessoren, Mittel über 100 Kommunikationen, $s_n = 8\,\text{byte}$, L pro IF = l_p).

Messungen

Bei der ersten Messung wird die Interface-Gesamtlänge l_p im Bereich 0–4000 variiert; dabei werden Messungen auf einer durch mehrere zur selben Zeit laufenden Anwendungen belasteten Intel Paragon mit Messungen auf der freien Maschine (*dedicated use*) verglichen. Abb. 4.2 zeigt exemplarisch je einen Fall: beide Kurven weisen lineare Abhängigkeit von l_p auf und sind auch quantitativ gleich. Im belasteten Fall kann es jedoch zu Störungen durch den Restbetrieb kommen, die maximal ca. 25 % betragen.

In der zweiten Messung wird die Abhängigkeit der Kommunikationszeit von den übertragenen Datenmengen erfaßt; dazu wird die Informationsmenge pro Knoten s_n bei konstanter Interfacelänge variiert (Abb. 4.3). Um den lokalen Aufwand für die Ausführung der *gather/scatter*-Funktionen, welche die Daten aus den Knoten in die Nachrichtenpuffer bzw. umgekehrt kopieren, vom restlichen Aufwand (im wesentlichen zum Senden/Empfangen der Einzelnachrichten) abzugrenzen, wird eine zusätzliche Meßreihe mit abgeschalteten *gather/scatter*-Funktionen (also leeren Nachrichten) durchgeführt. Die Nachrichten werden trotzdem weiterhin (bei gleicher Länge der Nachrichtenpuffer) versendet. Der Aufwand zur Ausführung der Kopierfunktionen (d.h. vor allem *memcpy()*) beträgt somit ca. 39 % der gesamten Kommunikationszeit.

Auswertung

Sämtliche Messungen zur Interface-Kommunikation legen lineare Abhängigkeiten der Laufzeit von beiden Parametern nahe; daher werden die gemessenen Werte durch Geraden approximiert und

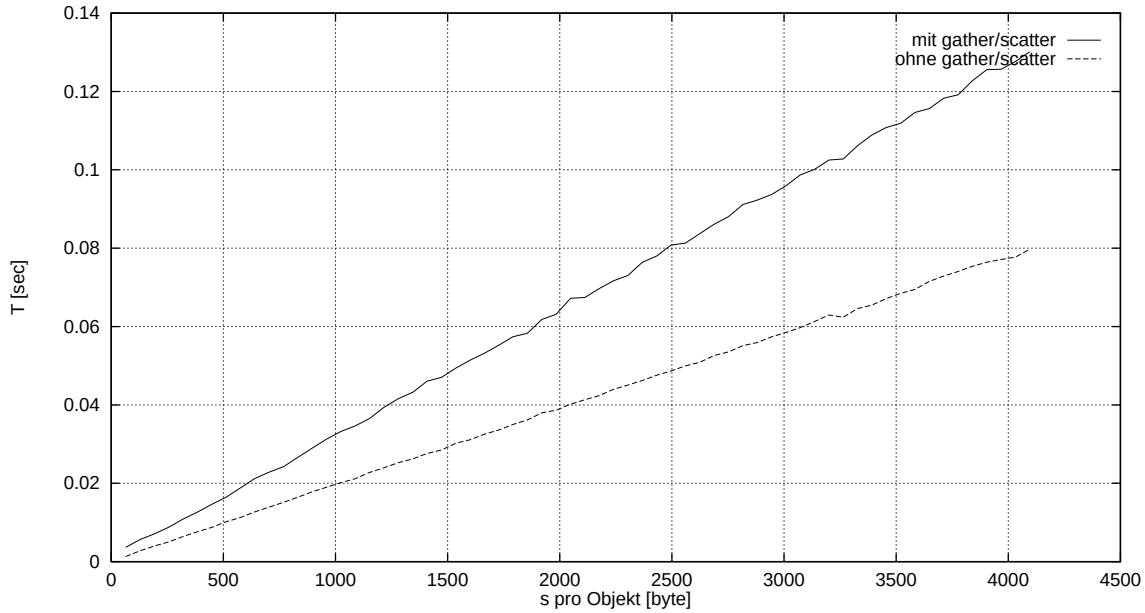


Abbildung 4.3: *Meßreihen zur Interface-Kommunikation: Vergleich von gesamter Kommunikation (*mit gather/scatter*) und Kommunikation ohne Einpack- bzw. Auspackvorgänge (*ohne gather/scatter*)* (PARAGON, 16 Prozessoren, Mittel über 100 Kommunikationen, $l_p = 400$ Objekte im Interface).

der mittlere quadratische Fehler als Gütemaß angegeben. Aus den ermittelten Geradensteigungen können für gegebene Parallelrechnerarchitekturen und die zugrundliegenden Programmiermodelle Abschätzungen angegeben werden, welche dem Anwender von DDD als Richtlinie für die Parallelisierungsarbeit dienen können.

Danach läßt sich die durchschnittliche Laufzeit für eine Kommunikation auf einem DDD-Interface nach folgender Formel ermitteln:

$$T_{\text{if}} \;=\; C^{4,T}_{\text{setup}} + l_p(C^{4,T}_{\text{elem}} + s_n(C^{4,T}_{\text{data}} + C^{4,T}_{\text{copy}})) \tag{4.1}$$

Die maschinenabhängigen Konstanten haben folgende Bedeutung:

$C^{4,T}_{\textbf{setup}}$**:** Aufsetzzeit pro Aufruf einer DDD-Interface-Kommunikation. Dieser Wert setzt sich aus DDD-internem Aufwand (z.B. Bereitstellung der Nachrichtenpuffer) und maschinenabhängigem Aufwand (z.B. Aufsetzzeit der Nachrichten) zusammen. Er ist insbesondere bei kurzen Interfaces und geringem Datenaufkommen am Interface relevant.

$C^{4,T}_{\textbf{elem}}$**:** Aufsetzzeit pro Objekt im Interface. Dieser Wert rührt vom Durchlaufen der Interfacelisten und den Aufrufen der *gather/scatter*-Funktionen her. Dies ist besonders bei kleinem Datenaufkommen je Objekt und langen Interfaces relevant.

$C^{4,T}_{\textbf{data}}$**:** Übertragungszeit pro Datenmenge. Dieser Wert bestimmt den Durchsatz der DDD-Interfaces und ist nach oben durch den Durchsatz des zugrundeliegenden *message-passing*-Modells begrenzt. Für großes Datenaufkommen ist dieser Wert bestimmend für die Kommunikationsleistung der Interfaces.

Tabelle 4.2: *Kommunikation mittels DDD-Interfaces: architekturabhängige Kenngrößen.*

Architektur	$C^{4,T}_{\text{setup}}$ [μsec]	$C^{4,T}_{\text{elem}}$ [μsec]	$C^{4,T}_{\text{data}}$ [$\frac{\mu\text{sec}}{\text{byte}}$]	$C^{4,T}_{\text{copy}}$ [$\frac{\mu\text{sec}}{\text{byte}}$]
CRAYT3E	189	1.17	0.0149	0.0195
HITACHI	394	1.21	0.0104	0.0055
PARAGON	301	1.58	0.0431	0.0408

$C^{4,T}_{\text{copy}}$: kumulierte Ausführungszeit aller *gather/scatter*-Funktionen. Da diese Funktionen hauptsächlich Daten zwischen den lokalen Objekten und dem Nachrichtenpuffer kopieren, wird dieser Wert von der lokalen Rechnerarchitektur (d.h. Prozessor, Cache, Speicher, Bus) bestimmt, jedoch nicht von der Leistung des Kommunikationsnetzwerks.

Tabelle 4.2 gibt die Konstanten wieder, wie sie aus einer Vielzahl von Messungen auf der jeweiligen Parallelrechnerarchitektur gewonnen werden können (mittlerer quadratischer Fehler der linearen Approximation stets kleiner als 1%). Aufgrund der hohen Aufsetzzeit im Verhältnis zu sehr guten Durchsatzwerten (vgl. Tabelle 4.1) können für die NECSX4-Architektur obige Konstanten nicht ermittelt werden, die Kommunikationszeit T_{if} liegt im gewählten Szenario unabhängig von den Parametern l_p und s_n stets im Bereich von $140 \Leftrightarrow 160$ msec.

Diskussion

Im gewählten Szenario (Torus von Prozessoren und Daten) hat die Prozessoranzahl unter der Bedingung $P_x \times P_y > 8$ auf keiner der getesteten Architekturen nennenswerten Einfluß auf die gemessenen Größen (bis auf NECSX4, s.o.). Um diese Eigenschaft genauer beschreiben zu können, wird die Prozessoranzahl bei festgehaltenen lokalen Parametern und dadurch entsprechend die Torus-Struktur variiert. Abb. 4.4 zeigt zwei verschiedene Plattformen im Vergleich; die qualitative Analyse ergibt eine geringere Laufzeit für $P_x < 3 \vee P_y < 3$. In solchen Fällen sind nämlich zwei benachbarte Prozessoren auch über die gegenüberliegende Seite ihres lokalen Gitters benachbart, was zu einer geringeren Zahl von Nachbarn im Prozessorgraph und damit zu kürzeren Aufsetzzeiten bei der Kommunikation führt. Das DDD-Interfacemodul verpackt dann die Objekte beider Seiten des Gitters in *eine* Nachricht. Sobald dagegen jeder Prozessor vier *echte* Nachbarn besitzt, ist der Nachbarschaftsgraph gesättigt.

Gleichzeitig ist es wichtig festzustellen, daß die berechneten Konstanten nur das gewählte Szenario beschreiben. Dies ist zwar im allgemeinen sinnvoll, da sich die reale Anwendungssituation nicht so regelmäßig darstellt wie im Szenario der Messung, was ohnehin eine Mittelung erfordert. Stellt man sich jedoch z.B. ein Szenario mit einer um eine Dimension höheren Konnektivität vor, so hat jeder Prozessor ca. sechs Nachbarn. Die Aufsetzzeit $C^{6,T}_{\text{setup}}$ beispielsweise wird dann quantitativ nicht der gemessenen aus Tabelle 4.2 entsprechen. Zudem stellt sich die Frage, warum die Aufsetzzeiten $C^{4,T}_{\text{setup}}$ um soviel größer sind als die reinen asynchronen PPIF-Aufsetzzeiten t_{s} aus Tabelle 4.1.

Um beide Fragen zu beantworten wurde ein alternatives Szenario erarbeitet. Dabei wird anstatt einer Torus-Konnektivität ein sternförmiger Nachbarschaftsgraph aufgebaut. Prozessor 0 besitzt

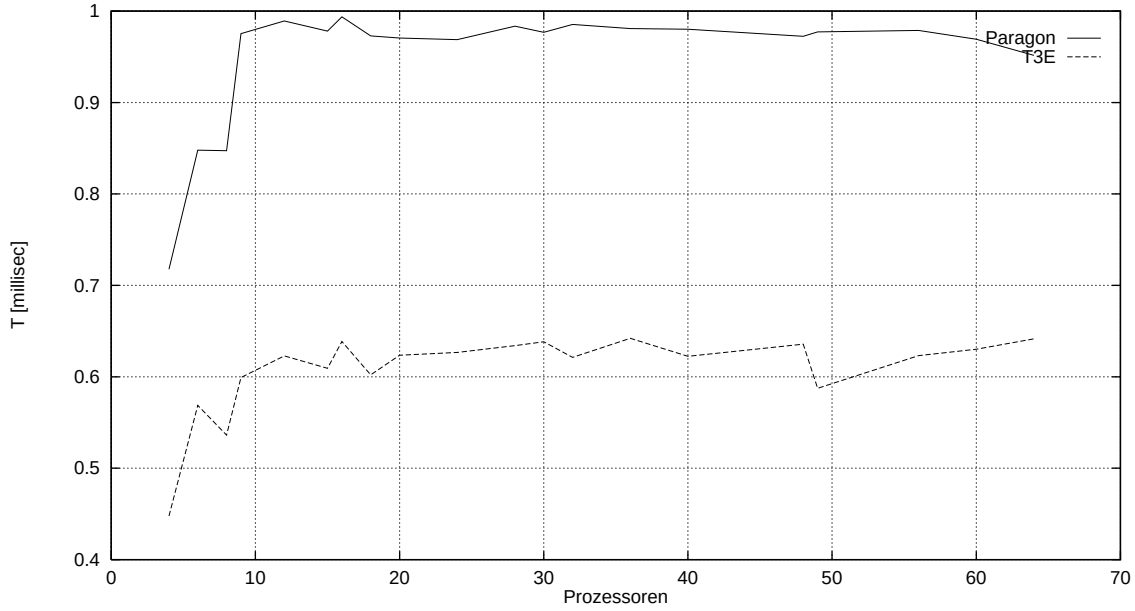


Abbildung 4.4: *Meßreihen zur Interface-Kommunikation: Zeit für die Interfacekommunikation bei verschiedenen Prozessorzahlen und jeweils entsprechend skaliertem Gitter; verschiedene Plattformen im Vergleich* (Mittel über 1000 Kommunikationen, $l_p = 400$ Objekte im Interface, $s_n = 8$ byte).

je ein Interface zu jedem anderen Prozessor; somit hat ersterer die gesamte Interfacelänge $P \times l_p$, die anderen dagegen nur die Länge l_p (übertragene Information pro Objekt $s_n = 8$ byte). Variiert man nun sowohl die Prozessorzahl P als auch die Interfacelänge l_p, so läßt sich aus den Meßreihen folgende Formel für die Kommunikationszeit $T_{\text{if}}(P, l_p)$ interpolieren:

$$T_{\text{if}}(P,l_p) = \overbrace{(C_1 + C_2 P)}^{C^{P,\star}_{\text{setup}}} + \overbrace{(C_3 + C_4 P)}^{C^{P,\star}_{\text{elem}}} \cdot l_p \quad [\mu\text{sec}] \qquad (P \geq 1)$$

$$\text{mit} \quad C_1 = \Leftrightarrow 79.0, \quad C_2 = 79.3, \quad C_3 = \Leftrightarrow 2.0, \quad C_4 = 1.65 \qquad (\text{für PARAGON})$$

Man erkennt, daß sowohl die Aufsetzzeit pro Interfacekommunikation $C^{P,\star}_{\text{setup}}$ als auch die Zeit pro Objekt im Interface $C^{P,\star}_{\text{elem}}$ von der Anzahl der Prozessoren in der Sterntopologie abhängt. Setzt man $l_p = 0$, erhält man die (prozessorabhängige) Aufsetzzeit, die für Nachrichten der Nutzlänge 0 auftreten würde; natürlich wird die reale Implementierung für $l_p = 0$ keine echte Kommunikation anstoßen. Für $P = 4$ entspricht $C^{4,\star}_{\text{setup}}$ in etwa der Aufsetzzeit $C^{4,T}_{\text{setup}}$ aus obigem Szenario. Die kleine Zeitdifferenz läßt sich auf unterschiedliche Synchronisation der Prozessoren $p \neq 0$ zurückführen: im Torus kommuniziert jeder Prozessor mit mehreren Nachbarn, im Stern nur Prozessor 0.

4.3.2 Konstruktion von Interfaces

Sobald vom Anwendungsprogramm ein neues DDD-Interface definiert wird oder nach einer früheren Interface-Definition verteilte Objekte erzeugt, umverteilt oder vernichtet werden, muß

DDD die Datenstrukturen des Interfaces neu aufbauen. Dieser Prozeß der Konstruktion von Interfaces wurde bereits in Abschnitt 3.2 genau beschrieben. Insbesondere werden nach jedem Transfer- oder Identifikationsvorgang alle Interfaces neu konstruiert. Die Kosten für diesen Vorgang sollen im folgenden ermittelt werden.

Die Implementierung der Interface-Konstruktion hat folgende Eigenschaften:

- Das Standardinterface muß auf jeden Fall konstruiert werden, dieses enthält alle verteilten Objekte eines Prozessors. Danach werden alle benutzerdefinierten Interfaces konstruiert.
- Es sind ausschließlich die Objekte betroffen, die tatsächlich Kopien auf anderen Prozessoren besitzen. Rein lokale Objekte sind für die Interfacekonstruktion nicht relevant.
- Die Konstruktion ist eine lokale Arbeit und daher ohne Kommunikation durchführbar.

Szenario

Tabelle 4.3 zeigt den Zeitaufwand zur Konstruktion von vier verschiedenen Interfaces. In jeder Messung werden durch Identifikation eine Torus-Struktur ähnlich der des vorherigen Abschnitts erzeugt und danach vier verschiedene Interfaces explizit konstruiert.

- Das Standard-Interface X_0 enthält alle echt verteilten Objekte.
- Interface X_1 enthält alle Objekte mit bestimmten, gegebenen Prioritäten. Diese Prioritäten werden so gewählt, daß X_1 ebensoviele Objekte enthält wie das Standard-Interface X_0.
- Die Prioritätsmengen A und B von Interface X_2 werden so gewählt, daß nur jedes zweite lokale Objekt ins Interface eingeordnet wird. Somit enthält X_1 doppelt soviele Objekte wie X_2.
- Schließlich werden die Prioritätsmengen von Interface X_3 so gewählt, daß kein Objekt ins Interface gelangt. X_3 ist also ein leeres Interface.

Die Anzahl von verteilten Objekten je Prozessor l_p wird variiert; jeder Prozessor konstruiert also sowohl zum Standard-Interface X_0 als auch zu X_1 vier Teilinterfaces der Länge $l_p/4$ und zu Interface X_2 vier Teilinterfaces der Länge $l_p/8$ (jeweils zu den vier Nachbarprozessoren).

Die in Tabelle 4.3 angegebenen Phasen entsprechen der Vorgehensweise in der aktuellen Implementierung (wie in Abschnitt 3.2.2 angegeben). Die Summe der Laufzeiten der fünf Phasen und der hier nicht angegebenen (vernachlässigbaren) Zeit für Speicherverwaltung ergibt die Gesamtzeit zur Interfacekonstruktion T_{ifc}.

Diskussion

In Phase 1 muß für jedes Interface die lokale Couplingmenge durchlaufen werden, wobei für das Standard-Interface keine weiteren Operationen notwendig sind. Für alle anderen (benutzerdefinierten) Interfaces muß jedes lokale Objekt auf Zugehörigkeit zum Interface überprüft werden. Die Zeit

Tabelle 4.3: *Laufzeiten zur Konstruktion von vier Interfaces (auf* CRAYT3E*-Architektur).*

		Phasen (Zeiten in [msec])				
l_p	T_{ifc} [msec]	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5
Standard-Interface X_0 ($\mid \mathsf{cpl}^p_{X_0} \mid = l_p$)						
4	0.07	0.01	0.02	0.03	0.00	0.00
16	0.11	0.01	0.05	0.03	0.00	0.00
64	0.43	0.02	0.28	0.10	0.00	0.02
256	1.69	0.06	1.52	0.09	0.00	0.01
1024	8.88	0.30	8.17	0.39	0.00	0.01
4096	48.36	1.08	44.42	2.83	0.00	0.01
16384	262.47	4.16	246.37	11.90	0.00	0.01
65536	1282.71	16.74	1217.84	48.09	0.00	0.01
Interface X_1 ($\mid \mathsf{cpl}^p_{X_1} \mid = l_p$)						
4	0.09	0.01	0.01	0.03	0.01	0.00
16	0.13	0.02	0.04	0.03	0.01	0.00
64	0.41	0.06	0.27	0.04	0.01	0.00
256	1.99	0.20	1.59	0.11	0.04	0.00
1024	9.71	0.99	7.95	0.46	0.25	0.01
4096	55.17	4.82	44.56	3.09	2.61	0.01
16384	290.66	20.36	246.61	12.96	10.66	0.01
65536	1392.91	80.79	1217.56	51.65	42.82	0.01
Interface X_2 ($\mid \mathsf{cpl}^p_{X_2} \mid = \frac{l_p}{2}$)						
4	0.05	0.01	0.00	0.02	0.00	0.00
16	0.07	0.01	0.02	0.02	0.00	0.00
64	0.16	0.03	0.09	0.02	0.01	0.00
256	0.70	0.14	0.46	0.05	0.02	0.00
1024	3.62	0.69	2.60	0.19	0.10	0.01
4096	20.57	3.64	14.10	1.59	1.18	0.01
16384	107.01	15.23	80.01	6.40	5.31	0.01
65536	493.27	60.31	385.78	25.70	21.40	0.01
Interface X_3 ($\mid \mathsf{cpl}^p_{X_3} \mid = 0$)						
4	0.02	0.00	0.00	0.00	0.00	0.00
16	0.03	0.01	0.00	0.00	0.00	0.00
64	0.04	0.02	0.00	0.00	0.00	0.00
256	0.11	0.08	0.00	0.00	0.00	0.00
1024	0.48	0.44	0.00	0.00	0.00	0.00
4096	2.48	2.43	0.00	0.00	0.00	0.00
16384	10.53	10.49	0.00	0.00	0.00	0.00
65536	42.04	41.99	0.00	0.00	0.00	0.00

für Phase 1 ist demzufolge bei X_0 stets klein, bei den anderen Interfaces höher. Bei nichtleeren Interfaces macht die Laufzeit dieser Phase nur einen relativ kleinen Anteil an der Gesamtlaufzeit aus.

Phase 2 ist im wesentlichen ein Sortiervorgang der Objekte, die in Phase 1 ausgewählt wurden (also Sortierung der Couplingmengen cpl_X^p). Die Komplexität dieses Sortierprozesses ist im mittleren Fall $O(n\log(n))$, mit $n = |\,\mathsf{cpl}_X^p\,|$. Die Auswertung des Sortierschlüssels ist verhältnismäßig aufwendig, wodurch sich eine große Konstante für den Sortieraufwand ergibt. Für die Interfaces X_0, X_1 und X_2 bildet Phase 2 den größten Anteil an der Gesamtrechenzeit, bei Interface X_3 entfällt dieser Schritt aufgrund der leeren Couplingmengen $\mathsf{cpl}_{X_3}^p$.

Der Aufwand der Phasen 3 und 4 hängt linear von der Größe der Couplingmenge ab; in diesen Phasen werden die Interface-Datenstrukturen weiter aufbereitet. Phase 5 ist aus Sicht der Laufzeit nicht relevant.

Setzt man die Werte aus Tabelle 4.3 zu den Ergebnissen der anderen Messungen ins Verhältnis und bezieht dies auf eine gegebene parallele Applikation, so gewinnt man einen Eindruck von der Ausgewogenheit der benutzten DDD-Funktionalität. Die Zeit zur Konstruktion von X_1 mit $l_p = 65536$ entspricht beispielsweise 15 Kommunikationen auf X_1 oder beträgt ca. 77% des vorausgehenden Identifikationsvorgangs (siehe auch im nächsten Abschnitt).

Für Anwendungen, bei welchen das Verhältnis Konstruktion/Benutzung von Interfaces ungünstig ausfällt, würde sich die weitere Optimierung der DDD-Implementierung zur Konstruktion von Interfaces lohnen. Dabei bietet sich die Möglichkeit an, die Couplingmengen cpl_X^p vor Phase 2 mit Hilfe geeigneter Datenstrukturen vorzusortieren und/oder in kleinere Teilmengen aufzuspalten, um den Sortiervorgang in Phase 2 zu beschleunigen. So könnte ein großer Anteil der Konstruktionszeit eingespart werden.

4.4 Laufzeiten zur Identifikation

Szenario

Das Szenario zur Bestimmung der Leistung des Identify-Moduls wird wie im Fall der Interface-Kommunikation an einen typischen Anwendungsfall angelehnt. Auch hier wird eine Torus-Struktur aufgebaut, um die Messungen soweit wie möglich unabhängig von der Anzahl der Prozessoren zu machen. Jeder Prozessor hat vier Nachbarn, wobei er mit jedem dieser Nachbarprozessoren n verteilte Objekte erzeugt. Dies ergibt eine Gesamtzahl von Identifikationen von $l_p = 4n$ (ähnlich wie im Interface-Szenario). Ebenfalls wird das Innere jedes Prozessorgebiets nicht abgespeichert, um große Werte für l_p testen zu können, die sonst auf den zur Verfügung stehenden Maschinen nicht in den Hauptspeicher passen würden. Der einzige Unterschied zum Interface-Szenario ist, daß pro Prozessornachbarschaft nicht zwei Reihen von Objekten, sondern nur eine erzeugt wird; dies ist für die gemessenen Größen jedoch irrelevant.

Messungen

In den Meßreihen werden absolute Laufzeiten zu Identifikationsvorgängen für verschiedene Hardware-Plattformen erfaßt und mit anderen Kennzahlen verglichen. Jeder Prozessor identifiziert je $l_p/4$ Objekte mit vier Nachbarprozessoren. Jeder der dazu nötigen l_p Identifikatoren Λ_i wird durch eine feste Anzahl N_{id} von Einzelidentifikatoren $\lambda_{i,j}$ gebildet (vgl. Abschnitt 2.4.4). In der Praxis heißt dies, daß pro zu identifizierendem Objekt für $N_{\mathrm{id}} = 1$ ein **DDD_IdentifyNumber()**-Aufruf und für $N_{\mathrm{id}} = 2$ zwei **DDD_IdentifyNumber()**-Aufrufe benutzt werden. Im zweiten Fall wird hier **DDD_IdentifyNumber()** zunächst für alle l_p Objekte einmal und danach in umgekehrter Reihenfolge ein zweites Mal aufgerufen, um die Messungen nicht durch gute Vorsortierungen zu verfälschen. Obwohl hier jedes Objekt mit genau N_{id} Einzelidentifikatoren identifiziert wird, ist es in realen Anwendungen möglich (und meistens nötig), die Anzahl der Einzelidentifikatoren n_i pro Identifikator Λ_i frei zu variieren.

Tabelle 4.4 zeigt folgende Werte in Abhängigkeit von der Anzahl zu identifizierender Objekte l_p:

N_{id}: Anzahl der Einzelidentifikatoren $\lambda_{i,j}$ pro Identifikator Λ_i eines zu identifizierenden Objekts.

T_{ident}: Die absolute, gemittelte Zeit für den Identifikationsvorgang (Laufzeit der Funktion **DDD_IdentifyEnd()** ohne Konstruktion von Interfaces) in [sec].

$T_{\mathrm{ident}}/T_{\mathrm{if}}$: Anzahl der Interface-Kommunikationen, deren Dauer dem Identifikationsvorgang entspricht. Dies dient als Anhaltspunkt für die relativen Kosten der Identifikation.

T_{ident}/l_p: Dauer des Identifikationsvorgangs, bezogen auf die Anzahl identifizierter Objekte.

T_{ifc}: Die mittlere Zeit zum Aufbau des Standardinterfaces für die gegebene Konstellation (Architektur und l_p, vgl. Abschnitt 4.3.2).

Tabelle 4.4: *Laufzeiten von Identifikationsvorgängen im typischen Torus-Szenario, für ein und zwei Einzelidentifikatoren pro Objekt; dazu Vergleich mit Interface-Kommunikation und -Konstruktion.*

	$N_{\text{id}} = 1$			$N_{\text{id}} = 2$			
l_p	$T_{\text{ident}}[\text{sec}]$	$\frac{T_{\text{ident}}}{T_{\text{if}}}$	$\frac{T_{\text{ident}}}{l_p}[\mu\text{sec}]$	$T_{\text{ident}}[\text{sec}]$	$\frac{T_{\text{ident}}}{T_{\text{if}}}$	$\frac{T_{\text{ident}}}{l_p}[\mu\text{sec}]$	$T_{\text{ifc}}[\text{sec}]$
CRAYT3E							
4	0.0004	2	98	0.0004	2	104	0.0001
16	0.0005	3	34	0.0006	3	40	0.0001
64	0.0010	3	15	0.0021	7	32	0.0004
256	0.0074	13	29	0.0098	18	38	0.0017
1024	0.0185	11	18	0.0661	42	65	0.0089
4096	0.0870	14	21	0.3962	66	97	0.0484
16384	0.4003	17	24	2.1130	88	129	0.2625
65536	1.7976	19	27	20.0905	212	307	1.2827
HITACHI							
4	0.0106	27	2656	0.0069	17	1719	0.0013
16	0.0119	29	742	0.0100	24	625	0.0013
64	0.0131	27	205	0.0138	29	215	0.0019
256	0.0144	20	56	0.0181	25	71	0.0013
1024	0.0269	15	26	0.0794	45	78	0.0094
4096	0.1131	19	28	0.4225	72	103	0.0456
16384	0.5269	24	32	2.3994	108	146	0.2288
65536	2.5162	29	38	21.4344	243	327	1.0562
PARAGON							
4	0.0241	78	6036	0.0245	79	6127	0.0001
16	0.0242	72	1511	0.0244	72	1525	0.0003
64	0.0270	61	421	0.0299	67	467	0.0011
256	0.0342	39	134	0.0501	57	196	0.0057
1024	0.0923	35	90	0.1979	76	193	0.0277
4096	0.3085	32	75	0.9769	103	239	0.1462
16384	1.3009	35	79	4.8769	131	298	0.6940
65536	5.6256	38	86	43.7139	296	667	3.3965

Diskussion

Aus Tabelle 4.4 geht hervor, daß die Zeit für einen Identifikationsvorgang T_{ident} superlinear mit der Anzahl der zu identifizierenden Objekte steigt. Im Fall $N_{\text{id}} = 1$ ist dieser Effekt noch kaum zu bemerken, im Fall $N_{\text{id}} = 2$ wachsen die Laufzeiten (vor allem bei $l_p = 65536$) überproportional. Gliedert man einen Identifikationsvorgang mittels genauerer Messungen (hier nicht aufgeführt) in die Phasen auf, die in Abschnitt 3.3.2 informell aufgelistet sind, so zeigt sich, daß Phase 2 (Vorbereitungsphase) den Hauptanteil der Laufzeit ausmacht.

Laut Abschnitt 3.3.3 wird in dieser zweiten Phase eine Sortierung der Identifikationskommandos nach der globalen ID des zu identifizierenden Objekts durchgeführt, damit der Zugriff auf die Identifikationstupel Λ_i möglich wird. Bei ungeeigneter Vorsortierung (im schlechtesten Fall) kann dieser Sortiervorgang die Komplexität $O(n^2)$ besitzen. Bei einer hohen Anzahl von zu sortierenden Kommandos führt dies zu überhöhten Laufzeiten. Im obigen Szenario bedeutet dies, daß Phase 2 für den schlechtesten Fall die Komplexität $4 \times O(N_{\text{id}}^2 l_p^2)$ besitzt; dies führt z.B. bei Vervierfachung von l_p im Fall $N_{\text{id}} = 2$ zur Verzehnfachung von T_{ident} bei allen betrachteten Architekturen (Tabelle 4.4). Die Identifikationskommandos wurden ja gerade so aufgerufen, daß eine ungünstige Vorsortierung entstehen mußte.

Alle anderen Phasen, darunter auch die eigentliche Kommunikation der Prozessoren, spielen anteilsmäßig für die Identifikation nur eine geringe Rolle. Eine verbesserte Implementierung des Identifikationsmoduls muß bereits in der Kommandophase alle Kommandos in eine geeignete Datenstruktur (z.B. einen balancierten Baum) einsortieren, um die teure Sortierung überflüssig oder zumindest billiger zu machen.

Durch die Identifikation im vorliegenden Szenario werden verteilte Objekte erzeugt, die ein Interface bilden. Die Kenngröße $T_{\text{ident}}/T_{\text{if}}$ gibt die Anzahl der Kommunikationen auf diesem Interface an, die in der Identifikationszeit hätten durchgeführt werden können (für $s_n = 8\,\text{byte}$). Da die Interfacelänge im wesentlichen linear in die Kommunikationszeit, aber superlinear in die Identifikationszeit eingeht, ist die Kenngröße $T_{\text{ident}}/T_{\text{if}}$ asymptotisch nicht beschränkt. In der Praxis wird jedoch meistens nur eine kleine Zahl von neuen Objekten identifiziert, dadurch jedoch ein bereits vorhandenes Interface vergrößert (z.B. nach der Verfeinerung bei adaptiven Mehrgitterverfahren). Außerdem können auf einem durchgeführten Identifikationsvorgang aufbauend beliebig viele Interfaces konstruiert werden. Die Zeit zur Interfacekonstruktion T_{ifc} (Standard-Interface) auf der jeweils identifizierten Datenbasis ist in der rechten Spalte von Tabelle 4.4 eingetragen, um den Laufzeitaufwand vergleichen zu können.

Rechnet man die gesamte Zeit zur Identifikation auf die Anzahl der identifizierten Objekte um (Spalten T_{ident}/l_p in Tabelle 4.4), so ergibt sich ein Anhaltspunkt für den Zeitaufwand pro Identifikationstupel. An erhöhten Werten von T_{ident}/l_p kann man im Bereich kleiner l_p die Aufsetzzeit (Fixkosten) des Vorgangs und im Bereich großer l_p die oben bereits angesprochene Asymptotik erkennen (vor allem für $N_{\text{id}} = 2$).

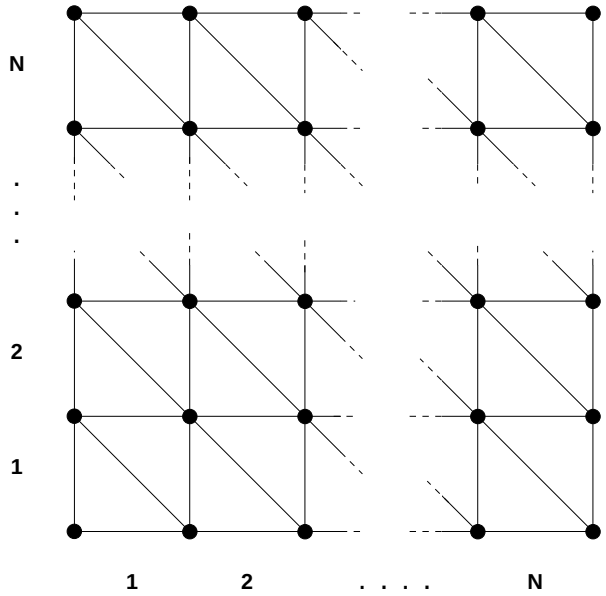


Abbildung 4.5: *Unstrukturiertes Gitter aus* $2 \times N^2$ *Dreieckselementen.*

4.5 Merkmale des Transfermoduls

Szenario

Die Implementierung des Transfermoduls hat bei weitem die höchste Codekomplexität der gesamten DDD-Bibliothek. Die Konstruktion eines brauchbaren Performance-Modells für den Transfervorgang ist nicht möglich, da zu viele Einflußfaktoren berücksichtigt werden müßten: Hardware (Prozessor, Netzwerk, Speicherhierarchie), Kommunikation (z.B. MPI-Implementierung, Mapping der Prozessoren), bis hin zu Datenstrukturen und Algorithmen des parallelen Anwendungsprogramms.

Daher wird in diesem Abschnitt versucht, ähnlich wie in den vorherigen Abschnitten ein realistisches Szenario aufzusetzen, um zumindest einen Anhaltspunkt für die Leistungsfähigkeit und -merkmale des Transfermoduls zu geben. In der Musteranwendung wird ein quasi-unstrukturiertes Gitter aus Dreiecken erzeugt und durch drei Datentypen *Dreieck*, *Kante* und *Knoten* repräsentiert. Dabei referenziert jedes Dreieckselement seine drei Kanten und Knoten; jede Kante referenziert die beiden anliegenden Dreiecke und beide Knoten. Die Implementierung erlaubt also jedes beliebige Dreiecksgitter, in den Messungen werden aber stets Gitter wie das in Abb. 4.5 skizzierte verwendet.

Die Verteilung eines Gitters auf die Prozessoren des Parallelrechners erfolgt ausgehend von den Dreiecksobjekten. Vor der Verteilung der Gitterdatenbasis werden diese Objekte zunächst vom Lastbalancierer mit ihrer neuen Partitionsnummer markiert; danach erfolgt die eigentliche Verteilung, wozu das DDD-Transfermodul benutzt wird. Die Dreieckselemente werden den Prozessoren eindeutig zugeordnet; an Prozessorgrenzen herrscht also lediglich Überlappung von Kanten- und Knotenobjekten, verteilte Dreiecksobjekte existieren nicht. Verteilte Kantenobjekte bestehen dadurch stets aus zwei lokalen Objekten, Knotenobjekte aus beliebig vielen.

Jeder Meßvorgang beinhaltet zwei verschiedene Transfervorgänge und läuft wie folgt ab:

0. Konstruktion. Das Gitter wird auf Prozessor 0 konstruiert.

1a. Lastbalancierung. Die Dreiecke werden mittels einer RCB-Strategie (*Recursive Coordinate Bisection*) mit neuen Partitionsnummern markiert. Die Anwendung des RCB-Verfahrens geschieht nur auf Prozessor 0.

1b. Lasttransfer. Der DDD-Transfer wird durchgeführt; dadurch entsteht aus einem lokalen Gitter eine gleichmäßige Verteilung der Gitterdatenbasis mit kurzen Interfaces.

2a. Lastbalancierung. Jeder Prozessor markiert seine in *1b)* erhaltenen Dreieckselemente zyklisch mit den Prozessornummern. Dadurch entsteht eine quasi zufällige Partitionierung.

2b. Lasttransfer. Der DDD-Transfer wird durchgeführt; dadurch entsteht aus einem verteilten Gitter wiederum eine gleichmäßige Verteilung der Gitterdatenbasis, diesmal jedoch mit langen Interfaces.

Messungen

Der oben beschriebene Meßvorgang wird in Meßreihen mit den variierenden Parametern Gittergröße N und Prozessoranzahl P durchgeführt. Ein Gitter wie in Abb. 4.5 besteht aus $(N+1)^2$ Knoten, $3N^2+2N$ Kanten und $n_T = 2N^2$ Dreieckselementen. Die Gesamtzahl n_{grid} aller Objekte im lokal erzeugten Gitter der Größe N ist also durch

$$n_{\text{grid}} = 6N^2 + 4N + 1$$

gegeben. Mit dem (architekturabhängigen) Speicherbedarf s_T, s_E bzw. s_N für einzelne Objekte vom Typ Dreieck, Kante bzw. Knoten und der oben angegeben Anzahl der jeweiligen Objekte läßt sich somit der Gesamtspeicherbedarf s_{grid} berechnen.

In Tabelle 4.5 sind die Laufzeiten der beiden Transfervorgänge (1b und 2b) aufgelistet. Da der Aufwand hier von der Anzahl der zu verschickenden Objekte n_{grid} bzw. vom gesamten zu verschickenden Speichervolumen (in Form des Gitter-Speicherbedarfs s_{grid}) abhängt, sind beide Werte in Tabelle 4.5 angegeben. Die obere Hälfte der Tabelle gibt die Zeiten für den jeweils ersten Transfervorgang an (Verteilung von Prozessor 0 auf alle Prozessoren), die untere Hälfte die Zeiten für den zweiten Transfervorgang (Umverteilung eines bereits verteilten Gitters).

Tabelle 4.6 gibt als Vergleichswert die Laufzeit des auf Prozessor 0 lokal angewendeten RCB-Verfahrens an. Da die Lastbalancierungsstrategie auf der Menge von Dreiecken arbeitet, ist die Anzahl von Dreiecken n_T in Tabelle 4.6 zusätzlich angegeben.

Diskussion

Bereits in diesem einfachen, jedoch anwendungsnahen Szenario lassen sich (neben den absoluten Transferzeiten) wichtige Effekte bei DDD-Transfervorgängen erkennen. Betrachtet man in Tabelle 4.5 die Zeilen für $N \in \{50, 70, 100, 140\}$, so findet man jeweils ungefähr eine Verdopplung in den

Tabelle 4.5: *Zeiten für zwei Transfervorgänge (Angaben in* [sec]. *Messungen auf* CRAYT3E*).*

		Prozessoranzahl P					
N	n_{grid}	2	4	8	16	32	s_{grid}[MB]
Transfer 1 (lokal $\rightarrow$ verteilt)							
10	641	0.059	0.069	0.079	0.093	0.110	0.090
20	2481	0.280	0.312	0.340	0.384	0.412	0.340
30	5521	0.652	0.787	0.862	0.912	0.955	0.750
40	9761	1.353	1.442	1.588	1.666	1.695	1.320
50	15201	2.023	2.305	2.701	2.751	2.886	2.060
60	21841	3.092	3.491	3.818	4.054	4.247	2.960
70	29681	4.384	4.972	5.527	5.639	5.760	4.030
80	38721	5.624	6.646	7.321	7.368	7.450	5.260
90	48961	7.238	8.720	10.497	9.944	9.869	6.650
100	60401	9.487	10.689	12.255	12.085	12.616	8.200
110	73041	11.687	13.387	15.365	15.493	15.052	9.920
120	86881	14.250	16.660	18.285	18.278	17.920	11.800
130	101921	17.004	19.542	21.599	21.689	21.954	13.850
140	118161	18.770	23.386	26.473	25.372	25.616	16.060
150	135601	23.291	27.054	29.947	29.761	30.274	18.430
Transfer 2 (verteilt $\rightarrow$ verteilt)							
10	641	0.046	0.054	0.040	0.041	0.063	0.090
20	2481	0.212	0.233	0.200	0.131	0.115	0.340
30	5521	0.550	0.611	0.356	0.231	0.202	0.750
40	9761	1.055	1.158	0.664	0.557	0.347	1.320
50	15201	1.791	1.973	1.723	0.897	0.494	2.060
60	21841	2.637	2.908	2.645	0.898	0.530	2.960
70	29681	3.708	4.368	2.390	1.572	1.040	4.030
80	38721	4.996	5.970	3.278	2.735	1.439	5.260
90	48961	6.550	7.498	6.513	2.768	1.849	6.650
100	60401	8.235	9.805	8.497	4.341	2.352	8.200
110	73041	9.975	11.749	6.580	5.340	2.856	9.920
120	86881	12.916	14.648	7.956	6.441	3.519	11.800
130	101921	15.305	17.904	15.383	4.539	3.382	13.850
140	118161	17.575	21.022	17.960	9.157	4.875	16.060
150	135601	20.812	24.749	13.336	10.610	5.614	18.430

Tabelle 4.6: *Zeiten für lokale Lastbalancierung mittels RCB-Verfahren (Angaben in* [sec]. *Messungen auf* CRAYT3E*).*

		Prozessoranzahl P				
N	n_T	2	4	8	16	32
10	200	0.001	0.002	0.002	0.004	0.003
20	800	0.006	0.013	0.015	0.024	0.022
30	1800	0.015	0.037	0.036	0.065	0.060
40	3200	0.030	0.065	0.080	0.124	0.130
50	5000	0.051	0.119	0.126	0.225	0.209
60	7200	0.077	0.171	0.218	0.344	0.344
70	9800	0.112	0.248	0.276	0.465	0.475
80	12800	0.150	0.363	0.422	0.683	0.709
90	16200	0.198	0.463	0.503	0.882	0.859
100	20000	0.250	0.551	0.672	1.107	1.124
110	24200	0.311	0.726	0.781	1.410	1.379
120	28800	0.376	0.869	1.078	1.691	1.875
130	33800	0.449	1.139	1.187	2.226	2.089
140	39200	0.533	1.226	1.491	2.415	2.480
150	45000	0.616	1.438	1.596	2.782	2.858

quadratisch abhängigen Größen n_{grid} bzw. s_{grid}. An den Laufzeiten der beiden Transfervorgänge für eine Prozessorzahl P kann man dann näherungsweise die Komplexität des Transfervorgangs ablesen: die Laufzeiten entwickeln sich etwas schlechter als linear mit den abgeleiteten Größen n_{grid} und s_{grid}. Aus der Implementierungsbeschreibung des Transfermoduls (siehe Abschnitt 3.4) läßt sich dies bestätigen; lediglich einige Sortiervorgänge bringen die Komplexität $O(n\log(n))$ mit sich (im mittleren Fall).

Beim ersten Transfervorgang wird ein vorher lokales Gitter auf den Parallelrechner verteilt, der Einpack- und Sendevorgang auf Prozessor 0 stellt hier den sequentiellen Flaschenhals dar. In Tabelle 4.5 läßt sich dies ablesen: die Laufzeiten für feste Gittergröße N nehmen meist mit der Prozessoranzahl zu. Wie eine genauere Analyse der Laufzeiten einzelner Phasen im Transfervorgang auf den jeweiligen Prozessoren zeigt, verbringen alle Prozessoren mit $P \neq 0$ den Großteil ihrer Laufzeit mit Wartezyklen.

Im zweiten Transfervorgang wird das bereits verteilte Gitter umverteilt. Hier sind alle Prozessoren ausgelastet, die Phasen des Transfervorgangs werden echt parallel durchlaufen, Wartezeiten entstehen kaum. Der untere Teil von Tabelle 4.5 zeigt die Konsequenz: für feste Gittergröße N nimmt die Laufzeit mit der Prozessoranzahl ab.

In obigen Messungen wurde der Einfluß einiger Systemparameter auf die Transferlaufzeit aufgezeigt: Objektgrößen und damit Übertragungsvolumen, Objektanzahl und Ausgewogenheit der Verteilung. Ein weiterer Parameter, nämlich die Anzahl der zu verschickenden Referenzen, kann ebenfalls eine Rolle für die Laufzeit eines Transfervorgangs spielen. Die Effekte durch den Aufwand zur Globalisierung/Lokalisierung von Referenzen im Transfer sind jedoch kompliziert und treten erst ab

circa 10-20 Referenzen pro Objekt auf; daher wird dieser Effekt in vorliegender Arbeit nicht näher untersucht.

4.6 Schlußbemerkungen

Skalierbarkeit paralleler Anwendungen

In den obigen Untersuchungen wurde mittels ausführlicher Meßreihen der *praktische* Nachweis der effizienten Anwendbarkeit des DDD-Konzepts geführt. In diesem Abschnitt werden dazu einige Aussagen zur asymptotischen Skalierbarkeit der DDD-Funktionalität ergänzt. Soll der gesamte parallele Algorithmus skalierbar für jede beliebige Prozessorzahl sein, dürfen natürlich auch die DDD-internen Verfahren und Kommunikationsschemata keine rein sequentiellen Teile bzw. globale Synchronisationen besitzen.

Interfaces. Die Konstruktion von Interfaces erfolgt auf Basis der lokalen Couplinglisten und beinhaltet keine Kommunikation. Die Kommunikation auf Interfaces synchronisiert nur mit den jeweiligen Nachbarn im Prozessorgraph und ist daher ebenfalls skalierbar. Einzelne Bereiche des Prozessorgraphs können also anderen, weiter entfernten Bereichen im Programmfluß vorauseilen. Eine Voraussetzung dafür ist natürlich, daß im Anwendungsprogramm Algorithmen angewendet werden, die selbst skalierbar sind. Dazu muß die Konnektivität des Prozessor-Nachbarschaftsgraphen unabhängig von der Anzahl der Prozessoren sein. Für numerische Verfahren auf unstrukturierten Rechengittern ist dies beispielsweise der Fall.

Identifikation. Der Identifikationsvorgang läuft vollständig verteilt ab und enthält nur Kommunikationen in dem Graphen, der durch die Identify-Kommandos des Anwendungsprogramms aufgebaut wird. Wie oben unter **Interfaces** beschrieben ist dies genau dann der Fall, wenn das Anwendungsprogramm selbst auf skalierbaren Strukturen aufsetzt. Ein wichtiges Beispiel ist wiederum die Klasse von numerischen Verfahren auf unstrukturierten Gittern, entsprechende Anwendungsfälle sind in Kapitel 5 beschrieben.

Transfer. Fast alle Kommunikationen beim Transfervorgang laufen in einem Prozessor-Nachbarschaftsgraph ab, sämtliche Algorithmen (z.B. Umrechnung von Referenzen) sind vollständig verteilt implementiert. Da die Kommunikationsanforderungen des Transfermoduls jedoch sendergetrieben sind (z.B. **DDD_XferCopyObj()**), die PPIF-Schicht dagegen nur zweiseitige Kommunikationsprimitive unterstützen kann, muß vor den eigentlichen Nachrichten eine Mitteilung an die Empfänger dieser Nachrichten erfolgen. Dies leistet der sog. *Notify*-Algorithmus, welcher *n*-zu-*m*-Kommunikation auf einem Prozessorbaum abwickelt, was zumindest theoretisch die Skalierbarkeit gefährdet (zu Details siehe Anhang A.2). Sollte sich dieser logarithmische Zusatzfaktor (d.h. $O(P\log(P))$) in der Praxis als Einschränkung herausstellen, muß die Funktionalität des *Notify*-Algorithmus effizienter implementiert werden. Bislang liegt der Zeitaufwand für *Notify* allerdings um Größenordnungen unter dem Aufwand für andere DDD-Funktionen oder auch unter dem Aufwand für die Algorithmen der Anwendung selbst.

Bis auf den Notify-Algorithmus sind also alle implementierten DDD-Funktionen vollständig skalierbar und können auf beliebig großen Parallelrechnersystemen angewendet werden.

Vergleich: DDD-Parallelisierung und Parallelisierung von Hand

Zum Schluß dieses Kapitels über die Leistungsmerkmale der DDD-Bibliothek sollen DDD-basierte Anwendungen mit von Hand parallelisierten Anwendungen verglichen werden. Dazu werden einige Einwände diskutiert, die scheinbar für eine aufwendige, manuelle Parallelisierung sprechen.

Einwand: Handparallelisierte Programme kommunizieren mit weniger Nachrichten.
In den Funktionen der DDD-Bibliothek werden alle Daten an gleiche Partnerprozessoren phasenweise in einer Nachricht zusammengefaßt; dies gilt für alle Module (Identifikation, Transfer, Interfaces). Eine weitere Reduktion der Nachrichtenanzahl ist auch von Hand nicht möglich.

Einwand: Handparallelisierte Programme kopieren weniger Nachrichtenpuffer.
Alle DDD-Funktionen sind darauf ausgelegt, die *receive*-Anweisungen asynchroner Kommunikationsvorgänge vorab aufzurufen, damit niedrigere Kommunikationsschichten die Möglichkeit haben, den DDD-verwalteten Puffer zu verwenden. Sollte die zugrundeliegende Kommunikationsschicht diese Information ignorieren, hat auch der Handparallelisierer keine Möglichkeit das Kopieren des Nachrichtenpuffers zu vermeiden. Darüberhinaus fallen in DDD keine weiteren Kosten für Datenkopiervorgänge an.

Einwand: In handparallelisierten Programmen überlappen Kommunikation und Berechnung.
Die Vorgänge während eines DDD-Transfers betreffen die gesamte Datenbasis; sollte eine ähnliche Funktionalität wie das DDD-Transfermodul für eine bestimmte Anwendung von Hand implementiert werden, ist es aufgrund der Softwarekomplexität nicht möglich, gleichzeitig noch Berechnungsvorgänge auf nicht betroffenen Daten durchzuführen. Für die Kommunikation auf DDD-Interfaces ist in der Implementierung der DDD-Bibliothek bereits vorgesehen, die Interface-Aufrufe in ein *Begin/End*-Paar aufzugliedern, um während der Interfacekommunikation andere Berechnungen durchzuführen.

Sieht man von zusätzlichem Verwaltungsaufwand (für Laufzeit und Speicher) bei Benutzung der DDD-Bibliothek ab, der ja auch im handparallelisierten Programm erbracht werden müßte, verbleiben als einziger Mehraufwand durch Benutzung von DDD die Kosten für zusätzliche Funktionsaufrufe bei der Interfacekommunikation (*gather/scatter*-Funktionen). In einem handparallelisierten Programm könnten diese Funktionsaufrufe eingespart und durch Schleifen ersetzt werden, allerdings auf Kosten einer allgemeinen Schnittstelle. Die Messungen des Interfacemoduls zu Anfang dieses Kapitels legen jedoch nahe, diesen kleinen Aufwand in Kauf zu nehmen und dafür eine klare und praktische Schnittstelle bereitstellen zu können.

Kapitel 5

Anwendungsbeispiele

In den vorangehenden Kapiteln dieser Arbeit wurde ein abstraktes Konzept zur Parallelisierung von graphbasierten Anwendungen und eine konkrete Realisierung dieses Konzepts in Form der DDD-Bibliothek beschrieben. Nun soll dieses Konzept tatsächlich angewendet werden; dazu werden einige der Parallelisierungsprojekte beschrieben, in denen DDD eine zentrale Rolle spielt.

Wichtig sind an dieser Stelle jedoch nicht die Anwendungen selbst; eine genaue Beschreibung der Anwendung samt ihrer Schnittstelle zur DDD-Bibliothek scheint im Rahmen dieser Arbeit nicht angebracht. Stattdessen werden die Anwendungen jeweils aus den Blickwinkeln beleuchtet, die ein tieferes Verständnis von spezifischen Eigenschaften und Funktionen der DDD-Bibliothek begründen können.

An einen allgemeinen Abschnitt zum praktischen Einsatz der DDD-Bibliothek schließen sich ausführliche Beschreibungen der beiden ausgewählten Anwendungen an. Beide Parallelisierungsprojekte sind im wesentlichen Lösungsverfahren für partielle Differentialgleichungen auf unstrukturierten, adaptiv verfeinerten Gittern; dieses Anwendungsfeld bildet auch den Schwerpunkt des bisherigen Einsatzgebietes von DDD.

Daneben wurde DDD allerdings ebenso zur Parallelisierung von Verfahren eingesetzt, bei denen andere Datenstrukturen bzw. Zugriffsmechanismen vorherrschen:

- Im Rahmen einer Diplomarbeit [102] wurde z.B. die parallele Variante eines 2D-Delauney-Triangulierungsalgorithmus implementiert. Hierbei sind andere Kommunikationsmuster und -protokolle zur dynamischen Erzeugung des gewünschten Dreiecksgitters notwendig als bei den unten beschrieben adaptiven, numerischen Verfahren. Das DDD-Konzept trägt auch hier zu einer erheblichen Vereinfachung des parallelen Programms bei gleichzeitiger Aufrechterhaltung der gewünschten Effizienz bei.

- In einem prototypischen Parallelisierungsprojekt wurde ein bestehendes Programm zur Multilevel-Lösung von Markovketten-Problemen [73] auf DDD aufsetzend parallelisiert. Die minimalen Änderungen am sequentiellen Programmcode konnten innerhalb weniger Tage durchgeführt werden. Werkzeuge zur Performancemodellierung benutzen Markovketten als mathematische Modelle der zu modellierenden Systeme.

5.1 Praktischer Einsatz der DDD-Bibliothek

Hier soll ein Überblick zur Vorgehensweise bei der Parallelisierung existierender Programme mittels DDD bzw. beim Entwurf neuer, bereits für Parallelrechner ausgelegter Programme gegeben werden. Die Reihenfolge der einzelnen Abschnitte entspricht der Reihenfolge der Parallelisierungsschritte.

5.1.1 Überlegungen zum parallelen Kontrollfluß

Das DDD zugrundeliegende Programmiermodell ist das SPMD-Modell (*single program, multiple data*). Die Phasen der Programmausführung bleiben auch bei der parallelen Version bestehen, lediglich auf feingranularer Ebene werden die Aktionen der einzelnen Prozessoren voneinander abweichen. Wichtige Entwurfsentscheidungen betreffen in dieser Anfangsphase hauptsächlich Vor- und Nachverarbeitungsschritte, z.B. die Erzeugung der Ausgangsdatenbasis bzw. die Ausgabe der Resultate eines parallelen Laufs.

Zur Erzeugung der Datenbasis zu Laufzeitbeginn ist es denkbar, alle erforderlichen Daten zunächst lokal auf einem Prozessor zu erzeugen und erst später auf alle Prozessoren zu verteilen. Diese Lösung ist für ein prototypisches Programm und für Verfahren praktikabel, die mit einem kleinen Graphen beginnen und diesen während der Laufzeit sukzessive erweitern. Treten schon zu Beginn der Laufzeit umfangreiche Datenmengen auf, so muß bereits bei der Dateneingabe eine parallele Lösung gefunden werden, um einen sequentiellen Flaschenhals zu vermeiden (siehe auch Abschnitt 5.1.8).

Ist Benutzerinteraktion während der Laufzeit erforderlich (z.B. über eine Kommando- bzw. grafische Schnittstelle), so müssen ebenfalls Entscheidungen bezüglich des groben Programmablaufs getroffen werden. Als einfaches Modell bietet sich an, die Interaktion zwischen parallelem Programm und Benutzer einem Prozessor zuzuordnen, der die Benutzereingaben an die anderen Prozessoren verteilt und deren Ausgaben zusammenfaßt und an den Benutzer weitergibt. Das Einsatzfeld der DDD-Bibliothek liegt jedoch in den Berechnungsphasen dazwischen und ist deshalb weitgehend unabhängig von den Details dieser Problematik.

5.1.2 Analyse der Datenbasis

Die graphartige Datenstruktur, die der zu parallelisierenden Anwendung zugrundeliegt und die mit DDD verteilt werden soll, wird bereits in der sequentiellen Implementierung durch Objekte verschiedener Typen bzw. Klassen repräsentiert. Diese Datentypen müssen zur Laufzeit beim DDD-TypeManager angemeldet werden (siehe Abschnitt 2.4.1), zuvor muß jedoch für jeden beteiligten Datentyp eine Auswahl unter folgenden Möglichkeiten getroffen werden:

Datentyp wird DDD-Objekt. Objekte dieses Typs sollen verteilt im Speicher vorliegen können; sie sollen von anderen Objekten referenziert werden können. Auf diesen Objekten ist Kommunikation mittels der DDD-Interfacefunktionen nötig. Darüberhinaus sollte ihr Speicherbedarf groß genug sein, damit der von DDD zusätzlich benötigte Speicher gegenüber dem Objektspeicher selbst nicht ins Gewicht fällt.

Datentyp wird registriertes Datenobjekt. Objekte dieses Typs lassen sich eindeutig einem (verteilten) DDD-Objekt zuordnen, sind aber selbst im Sinne des DDD-Modells nicht verteilt. Sie referenzieren selbst DDD-Objekte, werden aber nicht referenziert. Datenobjekte haben üblicherweise einen kleinen Speicherbedarf und treten in hoher Zahl im System auf (z.B. Abschnitte 5.2.4 und 5.3.2).

Datentyp wird nicht bei DDD registriert. Objekte dieses Typs enthalten nur Datenkomponenten, die entweder stets für alle Prozessoren gleich (also global konsistent) oder rein lokaler Natur sind. Jedes dieser Objekte kommt auf jedem Prozessor vor, durch gleiche Operationen ändern sich ihre Komponentenwerte stets global gleichzeitig. Dadurch muß nie Kommunikation über DDD-Interfaces stattfinden; der Transfer dieser Objekte ist ebenfalls nicht nötig.

In der Praxis ist die übliche Methode, zunächst alle Objekte, die verteilt vorliegen müssen, als DDD-Objekte anzumelden und später alle Objekte mit kleinem Speicherbedarf daraufhin zu untersuchen, ob sie in Datenobjekte umgewandelt werden können. Dieses Vorgehen erlaubt die schnelle Erstellung eines Prototyps mit nachträglicher, schrittweiser Optimierung des Speicheraufwands.

Bei der Definition der zulässigen DDD-Typen (mit DDD-TypeManager-Funktionen) muß für jedes Element eines zusammengesetzten Datentyps der Elementtyp gemäß Definition 8 (S. 42) angegeben werden. Referenzen auf DDD-Objekte haben den Elementtyp `ObjPtr`, global gültige Datenelemente den Elementtyp `GData` und rein lokal gültige Elemente den Typ `LData`.

5.1.3 Entwurf der verteilten Datenbasis

Der in Abschnitt 2.3 formal definierte Verteilungsbegriff läßt noch Raum für Entwurfsentscheidungen, z.B. Grad und Art von Redundanz (Gitterüberlappung), Konsistenzprotokolle und dazugehörige Prioritäten, Multilevel-Eigenschaften und Objekt-Attribute. Bereits in Abschnitt 2.3.6 wurde ein Beipiel für die Festlegung solcher Entwurfsdetails gegeben, allerdings noch in abstrakter, auf keine bestimmte Anwendungsklasse bezogene Form.

Zwei Anforderungen bestimmen, wie die Datenbasis auf die Prozessoren verteilt wird, wobei es genügt, nur die in Schritt 5.1.2 ausgewählten DDD-Objekte zu betrachten:

- Alle Referenzen, die im globalen Graph vorkommen, müssen auch im verteilten Graph repräsentiert werden.
- An allen Stellen eines Programms, wo über Referenzen auf ein anderes DDD-Objekt zugegriffen wird, muß eine Kopie dieses Objekts angelegt worden sein, damit die Referenz gültig ist. Falls keine lokale Kopie vorhanden ist, muß der Algorithmus den Zugriff über eine ungültige Referenz (Nullreferenz, Def. 6 auf S. 40, z.B. *null-pointer*) abfragen und verhindern.

Die erste Anforderung besagt, daß der verteilte Graph stets so mit sich überlappenden Objektkopien angereichert werden muß, daß der Graph nicht in unzusammenhängende Teilgraphen zerbricht. Besteht der globale Graph z.B. aus Objekten nur eines Typs, deren Topologie über eine Nachbarschaftsrelation definiert ist, muß bei einer Verteilung zweier benachbarter Objekte auf zwei Prozessoren das jeweils andere Objekt ebenfalls lokal vorhanden sein, d.h. beide verteilten Objekte

bestehen aus mindestens zwei lokalen Objekten. Damit kann die Nachbarschaftsrelation in Form von lokalen Referenzen eindeutig dargestellt werden. Auf den gesamten Graph übertragen bedeutet dieses Vorgehen eine Überlappung der den Prozessoren zugeordneten Teilgebieten der Tiefe 1. In Anhang C wird eine Methode vorgestellt, um Eigenschaften verteilter Gitter formal zu spezifizieren.

Die zweite Anforderung schließt sich direkt an: falls die Algorithmen, die auf die Datenstruktur des obigen Beispiels zugreifen, von einem Objekt ausgehend nur Daten der direkt benachbarten Objekte benötigen, so genügt die Überlappungstiefe 1 zusammen mit geeigneten Konsistenzoperationen über DDD-Interfaces (siehe Abschnitt 5.1.7). Falls jedoch Zugriffe auf die Nachbarobjekte der Nachbarobjekte erforderlich sind, muß entweder über zusätzliche lokale Objekte eine Überlappungstiefe von zwei Objektreihen erzeugt werden (was erheblichen zusätzlichen Speicherbedarf bedeuten kann) oder der Algorithmus derart abgeändert werden, daß dieser Zugriff durch eine Abfolge von Zugriffen auf direkte Nachbarn und dazwischenliegende Interface-Kommunikationen ersetzt werden kann. Derartige Entwurfsentscheidungen sind von Kompromissen zwischen Laufzeiteffizienz und Speicherbedarf geprägt.

5.1.4 Integration mit der Speicherverwaltung

Falls eine bestehende Anwendung parallelisiert werden soll, so wird in der sequentiellen Version der Speicher für die Objekte entweder vom Betriebssystem oder von einem anwendungseigenen Speichermanager angefordert. Damit in der parallelen Version die DDD-Bibliothek die Kontrolle über die verteilten Objekte übernehmen kann, müssen die Aufrufe zur Allokierung bzw. zur Freigabe von Objektspeicher durch Aufrufe des DDD-ObjectManagers ersetzt werden (siehe Abschnitt 2.4.1). Obwohl dieser Schritt rein technische Probleme betrifft, ist es doch wichtig, sich über Zuständigkeiten bei der Objektverwaltung klarzuwerden. Je nach Zielsprache der Anwendung (ANSI C, C++, Fortran) wird die Schnittstelle zwischen Anwendung und DDD-ObjectManager verschieden ausgelegt werden müssen.

5.1.5 Benutzerdefinierte Handler

Das Konzept der DDD-*Handler* wurde bereits in Abschnitt 2.4.5 beschrieben; an dieser Stelle des Parallelisierungsprozesses werden erste Handler für die angemeldeten DDD-Objekte entworfen. Für alle Details zu diesen benutzerdefinierten Funktionen wird auf [19] verwiesen.

5.1.6 Lastbalancierung und -transfer

Nach der Bereitstellung der nötigsten Handler-Funktionen und der Anbindung an den DDD-ObjectManager kann ein lokaler Graph auf einem Prozessor kreiert werden. Um diesen auf alle Prozessoren zu verteilen, ist es zunächst nötig, den lokal gespeicherten Graph mit einem Lastbalancierungsverfahren mit Partitionsnummern zu versehen, um ihn im Anschluß tatsächlich auf die anderen Prozessoren verteilen zu können. Der Zusammenhang zwischen der Lastbalancierungsstrategie und einer DDD-Anwendung wurde bereits in einem früheren Kapitel beschrieben (Abschnitt 2.2). Die Lastbalancierungsstrategie ist nicht in DDD enthalten und muß durch ein externes Werkzeug oder eine speziell auf die Anwendung zugeschnittene Strategie verwirklicht werden.

Als Eingabedaten für den Lastbalancierer wird man im allgemeinen nicht Objekte aller DDD-Typen heranziehen, sondern auf die Objekte eines zentralen DDD-Typs zurückgreifen. In einem durch Dreiecke, Kanten und Knoten repräsentierten, zweidimensionalen Gitter werden Kanten und Knoten für die Lastbalancierung ungeeignet sein; stattdessen verwendet man die Dreieckselemente zusammen mit ihrer Nachbarschaftsrelation und/oder ihrem geometrischen Schwerpunkt im Problemgebiet (siehe Abschnitt 4.5). Diese Auswahl ist anwendungsabhängig und muß von Fall zu Fall entschieden werden.

Wurden die lokalen Objekte eines bestimmten Typs mit ihrer neuen Prozessornummer markiert, dann können diese durch **DDD_XferCopyObj()**-Kommandos des DDD-Transfermoduls auf andere Prozessoren migriert werden. Die lokalen Objekte anderer Typen werden durch Handlerfunktionen verschickt, welche die Abhängigkeiten von den ersteren lokalen Objekten ausdrücken (z.B. müssen zu jedem Dreieck in obigem Szenario jeweils dessen Kanten und Knoten mitverschickt werden). Die Überlappung der Teilgraphen, die in Abschnitt 5.1.3 entworfen wurde, muß durch geeignete zusätzliche Transferkommandos erzeugt werden.

Liegt vor dem Lastbalancierungsschritt bereits ein verteilter Graph vor, so sind zusätzliche Probleme zu berücksichtigen:

- Entweder muß die Lastbalancierungsstrategie verteilt arbeiten können oder ihre Eingabedaten müssen auf einem Prozessor gesammelt werden. Damit im letzteren Fall kein sequentieller Engpaß entsteht, müssen die Daten vor der Konzentration auf einem Prozessor geeignet komprimiert werden, z.B. durch Methoden zur *Cluster*-Bildung (vgl. [10, 93]).
- Der existierende verteilte Graph war bereits konsistent im Sinn von Abschnitt 5.1.3, er enthielt also bereits die dort definierte Überlappung der einzelnen Teilgraphen. Bei der Umverteilung muß nun nicht nur eine neue Überlappung berechnet, sondern die alte Überlappung entfernt werden. Dies kann nicht in getrennten, aufeinanderfolgenden Schritten geschehen, weil sonst der Graph nach Entfernung der alten Überlappung auseinanderbrechen würde. Damit keine Referenzen verlorengehen, muß also die Abbildung von der alten auf die neue Überlappung gleichzeitig erfolgen (vgl. Abschnitt 2.3.5).

5.1.7 Synchronisation und Konsistenz

Basierend auf dem Verteilungsentwurf (Abschnitt 5.1.3) und dem benutzerdefinierten Konsistenzprotokoll (Abschnitt 2.3.8) werden nun Synchronisationspunkte ins parallele Programm eingeführt, die Konsistenz an den Stellen garantieren, wo dies benötigt wird. Dazu wird das DDD-Interfacemodul verwendet, da jeder Synchronisation ein vorher definiertes Interface zugrundeliegt.

DDD-Interfaces existieren nur in Bereichen des Graphen, wo verteilte Objekte vorhanden sind, d.h. in den Überlappungsbereichen der verschiedenen Partitionen. Kommunikation der Prozessoren erscheint dort auch sinnvoll, um berechnete Daten über das Gesamtsystem zu verbreiten. Es obliegt dem DDD-Benutzer, nach Berechnungsphasen solche Kommunikationen zum Konsistentmachen der Daten anzustoßen, indem er einfache DDD-Interfacefunktionen ins sequentielle Programm einfügt (siehe Abschnitt 2.4.2). Kann dies an zentraler Stelle geschehen, so ist eine transparente Parallelisierung vieler iterativer Anwendungen möglich.

5.1.8 Parallele Ein-/Ausgabe

Ein großer, während einer parallelen Berechnung erzeugter Graph kann mit seinen gesamten Daten nicht mehr problemlos auf eine Datei ausgegeben werden; die erforderliche Synchronisation der beteiligten Prozessoren würde einen ernsten sequentiellen Flaschenhals darstellen. Stattdessen wird man die Dateiausgabe so auslegen, daß jeder Prozessor seinen Teil des Graphen auf eine separate Datei ausgibt; sofern das Betriebssystem diese Operation effizient anbietet, kann man mit dieser Technik während der Laufzeit ohne Verlust der Parallelität Dateiausgaben erzeugen (z.B. zur Erzeugung von *checkpoints*, die ein späteres Wiederaufsetzen einer langwierigen Berechnung ermöglichen).

Beim Einlesen dieser Dateien (eine Datei pro Prozessor) wird der verteilte Graph zunächst als Menge von lokalen Graphen ohne weitere Kopplung/Überlappung vorliegen. Diese Kopplung muß erst durch Kommunikation der Prozessoren aufgebaut werden, wozu sich die Funktionalität des DDD-Identifymoduls in Form der Identifizierung von am Prozessorrand liegenden Objekten anbietet. Ein derartiges Anwendungsbeispiel ist in Abschnitt 5.2.6 beschrieben.

Auf jeder einzelnen Parallelrechnerarchitektur werden eine oder mehrere spezifische Methoden zur parallelen Ein- und Ausgabe von Daten angeboten; diese Lösungen bieten zwar oft effiziente Durchsatzwerte, sind aber im allgemeinen bisher nicht portabel und somit im hier beschriebenen Kontext nicht zufriedenstellend. Ansätze zu portablen Ein-/Ausgabeschnittstellen bestehen zwar bereits oder werden derzeit entwickelt (z.B. MPI-2-I/O [109], Scalable I/O Initiative [130], Passion [116], auch [108]), wegen der rasanten Entwicklung neuer Parallelrechnerhardware wurde aber die erwünschte Generalität im Bereich des parallelen I/O noch nicht erreicht. Nicht nur im *Passion*-Projekt [116] wurde die Schlußfolgerung gezogen, die Funktionalität zum parallelen I/O auf Anwendungsebene anzusiedeln; auch in DDD-basierten Anwendungen empfiehlt sich diese Vorgehensweise.

5.1.9 Dynamische Veränderungen der Datenbasis

Neben der Lastbalancierung gibt es oftmals andere Aspekte eines parallelen Programms, die eine Änderung der Datentopologie beinhalten. Ein wichtiges Beispiel ist die Gitterverfeinerung/-vergröberung bei adaptiven numerischen Verfahren. DDD unterstützt deren Parallelisierung durch seine Identifikations- bzw. Transferfunktionen. Beispiele zur Implementierung von verteilten Gitterverfeinerungsmechanismen sind in den Abschnitten 5.2.5 und 5.3.3 beschrieben.

Auf Basis der atomaren DDD-Transferkommandos sind beliebige Kommunikationsvorgänge implementierbar; je nach Anwendung kann somit eine individuelle Lösung erstellt werden. Neben den typischen Anwendungen von Lösungsverfahren für partielle Differentialgleichungen auf unstrukturierten Gittern sind andere Anwendungsbereiche vorstellbar, in denen Teile der Datentopologie erst zur Laufzeit entstehen; ein Beispiel für eine solche Spezialanforderung stellt die Parallelisierung eines Delauney-Triangulierers auf der Basis von DDD dar (siehe S. 117).

5.2 CEQ – Gleichungslöser auf beliebig unstrukturierten Gittern

Dieser Abschnitt beschreibt die erste Anwendung des DDD-Konzepts im Rahmen eines Softwareprojekts von realistischen Ausmaßen. Das gewählte numerische Verfahren stellt durch die Komplexität und Dynamik seiner Datenstrukturen hohe Anforderungen an die Parallelisierungsschnittstelle; darüberhinaus soll die Eignung des DDD-Konzepts im objektorientierten Kontext überprüft werden.

5.2.1 Das sequentielle Programm

Die Ausgangsbasis für die Parallelisierung ist die sequentielle Implementierung eines numerischen Lösungsverfahrens für partielle Differentialgleichungen, welches den Kern des am *Rechenzentrum Universität Stuttgart* (RUS) lokalisierten Projekts *CEQ* (Conservation EQuation) darstellt [67]. Im Zentrum steht dabei die Entwicklung eines dimensionsunabhängigen Finite-Volumen-Verfahrens zunächst zur Lösung der Euler-Gleichung (mit *cell-centered* Diskretisierung und expliziter Zeitintegration), dessen Problemgitter aus beliebigen Polygonen bzw. Polyedern zusammengesetzt ist. Wegen der Komplexität der dazu erforderlichen Datenstrukturen gehen mit der Entwicklung des Verfahrens selbst zwei Teilprojekte zur Parallelisierung (basierend auf der hier vorgestellten Arbeit) sowie zur Visualisierung von Daten und Prozessen auf Gittern beliebiger Elemente [131] einher.

Die Zielsetzung des CEQ-Projekts ist die Integration von Gittergenerierung und Strömungssimulation und schließlich des gesamten Entwicklungszyklus, ausgehend vom CAD-Entwurf. Gerade im industriellen Rahmen ist dieser Ansatz höchst relevant: die Gittergenerierung hat bislang am gesamten Zeitaufwand einer Simulation einen unangemessen hohen Anteil. Damit die mittels CAD erzeugte Geometrie direkt als Ausgangspunkt der Simulation dienen kann, wurde ein adaptives Finite-Volumen-Verfahren entwickelt, das sich von einigen wenigen Elementen (bzw. Kontrollvolumen) ausgehend durch dynamische Verfeinerung an kritischen Stellen im Problemgebiet immer feinere Gitter schafft. Dabei werden beliebige Elemente zugelassen, sogar nicht-konvexe und mehrfach zusammenhängende Polygone bzw. Polyeder.

Für die lokale Gitteradaption sind Fehlerkriterien nötig, die das gegebene Gitter an den notwendigen Stellen zur Verfeinerung markieren; da die Verfeinerung selbst in Form der Teilung eines Elements durch eine Schnittgerade bzw. Schnittebene durchgeführt wird, müssen neben den Markierungen auch Verfeinerungsrichtungen für die markierten Elemente bestimmt werden. Zur Vermeidung von kurzen Kantenstücken bzw. zu kleinen Polyederseiten kann der Schnittebene eine Dicke zugeordnet werden; falls der Abstand der Ebene von einem Gitterpunkt innerhalb dieser Toleranz liegt, wird die Ebene so verbogen, daß sie direkt auf dem Gitterpunkt zu liegen kommt. Durch dieses Verfahren verbessern sich zwar Eigenschaften des Gitters, gleichzeitig kann nun jedoch die Bearbeitungsreihenfolge der Elementverfeinerung eine Rolle für die Gittergeometrie spielen.

Die sequentielle Implementierung erfolgte in der Programmiersprache C++ nach einem streng objektorientierten Konzept. Dies ermöglichte eine klare Abgrenzung der geometrischen von den numerischen Programmteilen. In Abb. 5.1 sind die beiden zugrundeliegenden Klassenhierarchien dargestellt. Die geometrischen Begriffe *Node*, *Edge*, *Face* und *Cell* werden von der Basisklasse *GeometricObject* abgeleitet und sind paarweise durch Konnektivitätsrelationen verknüpft. In der Basisklasse ist z.B. der geometrische Schwerpunkt des Objekts gespeichert, bei Objekten vom Typ

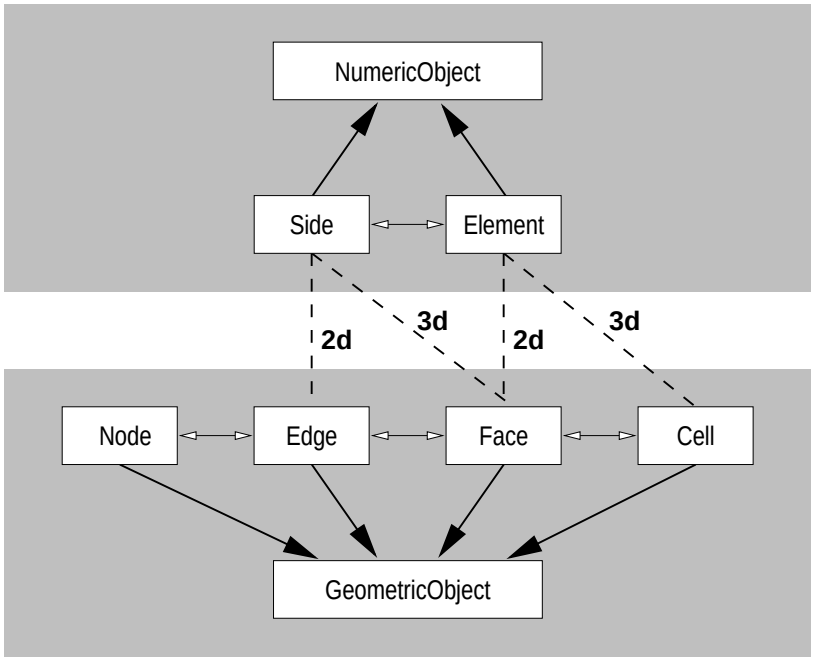


Abbildung 5.1: *Objektorientierte Datenstrukturen zur Trennung von Geometrie und Numerik in* CEQ *durch Ableitung von getrennten Basisklassen* NumericObject *und* GeometricObject*. Waagerechte Doppelpfeile stellen die Gittertopologie-Information in Form von Konnektivitätsrelationen dar. Je nach Problemdimension werden zur Kompilierzeit entweder die 2d- oder die 3d-Beziehungen hergestellt.*

Node ist dies die geometrische Position im Problemgebiet. Kanten (*Edge*-Objekte) verweisen auf beide Knoten (*Node*-Objekte), die Knoten jeweils auf alle angrenzenden Kanten usw., die Topologie wird also vollständig repräsentiert. Analog wird mit den numerischen Begriffen verfahren, hier definiert man also die abgeleiteten Klassen *Side* und *Element* nach Finite-Volumen-Terminologie.

Die Verknüpfung des geometriebezogenen und des numerischen Programmteils findet zur Kompilierzeit statt; je nach Problemdimension entsteht so automatisch das passende Programm, wobei folgende Verknüpfungen hergestellt werden:

Problem	*Side* verknüpft mit	*Element* verknüpft mit
1-dim.	*Node*	*Edge*
2-dim.	*Edge*	*Face*
3-dim.	*Face*	*Cell*

Damit kann der numerische Teil des Programms komplett dimensionsunabhängig gehalten werden; nur durch obige Verknüpfung setzt er auf definierte Weise auf der Geometrie bzw. Gittertopologie auf. Auf ähnliche Weise werden die Konnektivitätsinformationen (waagerechte Pfeile in Abb. 5.1) implementiert: zunächst werden nur *Up*- und *Down*-Relationen unterschieden, durch Parametrisie-

rung mit den geometrischen Klassen erhält man schließlich die Konnektivitätsrelationen auf den verschiedenen Ebenen (*Node*↔*Edge*, *Edge*↔*Face* und *Face*↔*Cell*).

An den geometrischen Rändern des Problemgebiets ist die Repräsentation von speziellen Randinformationen nötig (z.B. Randbedingungen, Geometrie). Diese wird in CEQ nicht in den *Side*-Objekten gespeichert, die direkt auf dem Gebietsrand liegen, sondern in speziellen *Randelementen*. Diese liegen zwar selbst logisch im Äußeren des Problemgebiets, sind aber über eine Konnektivitätsrelation mit dem benachbarten *Side*-Objekt und daher mit dem Inneren des Gebiets verknüpft. Es ist also stets sichergestellt, daß ein *Element* des Gitters über jedes seiner Seiten ein Nachbarelement besitzt, wobei dieses entweder ein *innerer Nachbar* (d.h. ein normales *Element*-Objekt) oder ein solcher *Randelement-Nachbar* sein kann.

Im numerischen Teil des Programms gliedert sich der Ablauf einer adaptiven Simulation in die sich abwechselnden Phasen

- Simulation auf aktuellem Gitter und
- adaptive Verfeinerung.

Dieser Zyklus wird wiederholt, bis die gewünschte Genauigkeit der Lösung erreicht ist. Die Simulation setzt sich dabei aus einer Folge von Iterationen zusammen, in welchen folgende Schritte der Reihe nach ausgeführt werden:

1. Berechnung der Flußintegrale
2. Aktualisierung der Elementzustände
3. Berechnung der Gradienten auf den Elementen
4. evtl. Limiting auf Seiten und/oder Elementen
5. Aktualisierung der Ausflußränder
6. Neuberechnung der Flüsse
7. Bestimmung der Zeitschrittweite

5.2.2 Architektur des parallelen Programms

Der Aufbau des parallelen Programms stimmt mit der in Abschnitt 2.1 beschriebenen allgemeinen Struktur überein [23, 65]. Das sequentielle Programm aus dem vorigen Abschnitt wurde über neu entwickelte Schnittstellenfunktionen bzw. -klassen an die Schnittstelle der DDD-Bibliothek angebunden. Die DDD-Bibliothek setzt wiederum auf dem CEQ-Speichermanager auf, so daß eine Gesamtstruktur wie in Abb. 2.1 (S. 34) dargestellt entsteht.

Da die CEQ-Software als objektorientiertes C++-Programm entwickelt wurde, die DDD-Bibliothek jedoch eine reine ANSI C-Schnittstelle bietet, wurden einige Klassen entworfen, die sowohl die Funktionalität der DDD-Bibliothek als auch die Schnittstelle Anwendung/DDD kapseln sollen und so eine klare Strukturierung auf Codeebene erzielen. Die wichtigsten dieser Klassen sind:

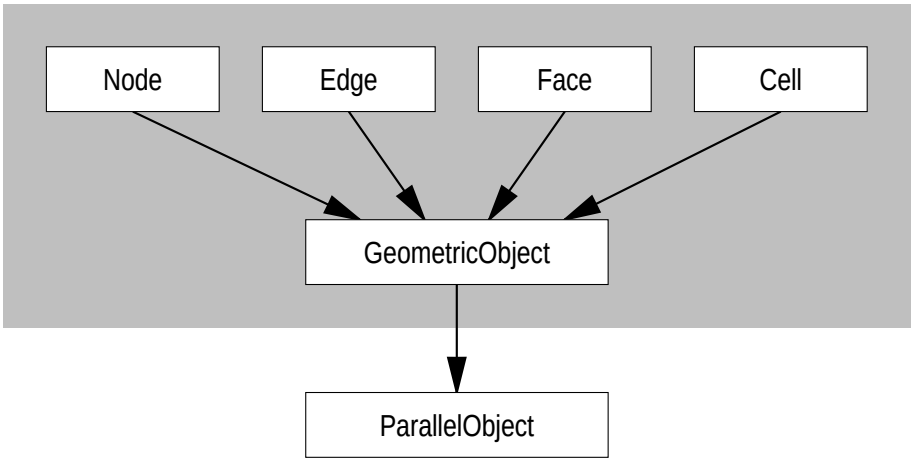


Abbildung 5.2: *Zusätzliche Vereinfachung der Parallelisierung durch objektorientierte Datenstrukturen. Die Klasse* ParallelObject *kapselt alle Eigenschaften von DDD-Objekten.*

DDDLibrary. Diese Klasse kapselt alle globalen (d.h. nicht objektbezogenen) Funktionen der DDD-Bibliothek; außerdem enthält sie die für die parallele CEQ-Version nötigen DDD-Interfaces und anwendungsbezogene Kommunikationsoperationen. In der Implementierung dieser Klasse sind ebenfalls die Anmeldung von DDD-Typen der vorkommenden Anwendungsklassen als auch die Registrierung der benutzerdefinierten Handlerfunktionen verborgen. Die Klasse DDDLibrary stellt somit die DDD-Bibliothek samt der Anwendungsschnittstelle dar.

ParallelObject. Diese Klasse kapselt alle objektbezogenen Funktionen der DDD-Bibliothek, die Implementierung der Handlerfunktionen für die verschiedenen DDD-Typen, mögliche Prioritäten der vorkommenden DDD-Objekte, und zusätzlich notwendige Daten jedes verteilten Objekts. Die Klasse ParallelObject stellt somit ein DDD-Objekt samt seiner Anwendungsschnittstelle dar (siehe auch Abs. 5.2.3).

LoadBalancer. Wie in Abschnitt 2.2 dargestellt, ist der Lastbalancierungsalgorithmus weder Bestandteil von DDD noch des Anwendungsprogramms. Aus diesem Grund wurde eine eigene Klasse geschaffen, die alle Funktionen der verschiedenen Lastbalancierungsstrategien enthält. Die Funktionalität dieser Klasse wird in Abschnitt 5.2.4 näher beschrieben.

5.2.3 Datenstrukturen und ihre Abbildung auf den verteilten Graph

Zur Parallelisierung wurde eine Analyse des sequentiellen Programms gemäß den Abschnitten 5.1.2 und 5.1.3 durchgeführt. Da der zu verteilende Graph rein durch die geometrischen Klassen repräsentiert wird, ist es ausreichend, zunächst nur diese zu betrachten. Jedes geometrische Objekt soll als verteiltes Objekt vorliegen können; dieser Zusammenhang kann nun einfach durch Ableitung der Klasse *GeometricObject* von der oben beschriebenen Basisklasse *ParallelObject* implementiert

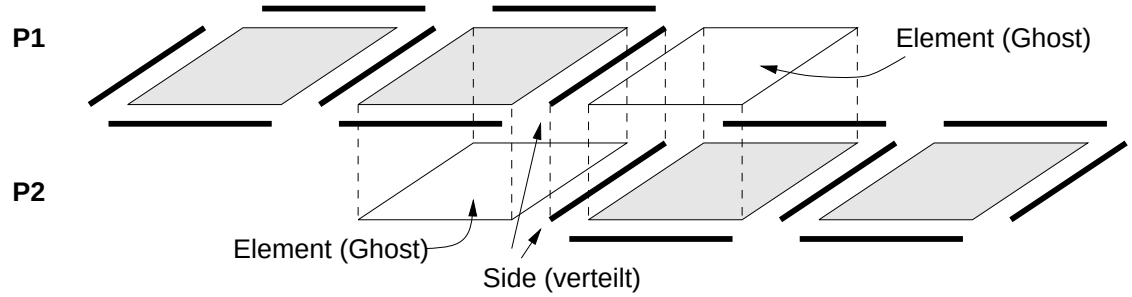


Abbildung 5.3: *Überlappungsstrategie der CEQ-Datenstrukturen am zweidimensionalen Beispiel. Im Bereich des Gitters an der Grenze zweier Prozessoren P1 und P2 entsteht eine Überlappung von einer Elementreihe. Elemente haben entweder* Master*-Priorität (ausgefüllt) oder (als Kopie)* Ghost*-Priorität. Dazwischenliegende Elementseiten (*Side*) werden doppelt gespeichert.*

werden. Diese Vorgehensweise ist elegant, denn große Teile des sequentiellen Codes können nun unverändert wiederverwendet werden. Abb. 5.2 zeigt den Zusammenhang schematisch.

Damit sind nun auch die numerischen Klassen *Element* und *Side* DDD-Objekte und können verteilt gespeichert werden. Die Analyse der Datenzugriffe im numerischen Teil des Programms ergibt:

- Gradienten werden im Element gespeichert, zur Neuberechnung muß auf die Elementseiten zugegriffen werden.
- Flüsse werden in den Elementseiten gespeichert, zur Neuberechnung muß auf die Gradienten der beiden benachbarten Elemente zugegriffen werden.

Damit diese Datenzugriffe weiterhin prozessorlokal stattfinden können, müssen auf einem Prozessor zu jedem Element auch alle Seiten gespeichert werden und zu jeder dieser Seiten wiederum die beiden angrenzenden Elemente. Liegt die Seite jedoch auf der Grenze zwischen zwei Partitionen, so werden vom außerhalb liegenden Element nur diese Seite (und evtl. andere seiner Seiten auf Partitionsgrenzen) und alle numerischen Daten gespeichert (Abb. 5.3). Damit gibt es von jedem Element genau eine sog. *Master*-Kopie (d.h. ein lokales Objekt mit Priorität *Master*) und beliebig viele sog. *Ghost*-Kopien (d.h. lokale Objekte mit Priorität *Ghost*). Die Seiten (*Side*-Objekte) an der Prozessorgrenze werden stets doppelt gespeichert, für beide benachbarten Prozessoren je einmal. Diese Überlappungsstrategie wurde auf den numerischen Begriffen definiert und ist somit unabhängig von der Problemdimension. Eine formale Spezifikation dieser Überlappungsstrategie ist der Regelsatz 1-OVPE aus Anhang C.

5.2.4 Lastbalancierung und Lasttransfer

Strategie der Lastbalancierung

Da die Objekte der numerischen *Element*-Klasse eine vollständige Überdeckung des Problemgebiets darstellen und die Rechenarbeit pro Prozessor in etwa proportional zur Anzahl seiner Elemente ist,

wurden eben diese Elemente als Grundlage der Lastbalancierungsstrategie herangezogen. In Anbetracht des hohen Speicherbedarfs pro Gitterelement wird die Anzahl der Elemente im Gitter im wesentlichen durch den verfügbaren Hauptspeicher beschränkt sein; deshalb konnte zunächst von einer verteilten Lastbalancierungsstrategie Abstand genommen werden. Stattdessen wurde innerhalb der Klasse *LoadBalancer* folgende Vorgehensweise implementiert:

1. Einsammeln aller Daten auf einem Prozessor, die zur Durchführung der Lastbalancierungsstrategie notwendig sind
2. Aufrufen einer Lastbalancierungsstrategie
3. Verteilen der neuen Partitionsnummern an alle anderen Prozessoren
4. Durchführen des Lasttransfers mit Hilfe des DDD-Transfermoduls

Das vorherige Einsammeln der Daten und das Austeilen der Partitionierungsinformationen geschieht effizient auf der Baum-Kommunikationsstruktur, die durch die PPIF-Schicht im Prozessorgraph bereitgestellt wird (siehe Anhang A.1). Als Lastbalancierungsstrategie in Schritt 2 wurden zwei Ansätze exemplarisch ausgeführt; beides sind zentrale, quasistatische Verfahren (vgl. Abschnitt 2.2):

Rekursive Koordinatenbisektion (RCB). Zur Anwendung dieses einfachen Verfahrens müssen die Schwerpunktkoordinaten aller Elemente gesammelt werden. Diese Variante dient als Beispiel für die Anbindung eigener Lastbalancierungsverfahren an DDD-basierte Anwendungen.

Metis. Innerhalb der *LoadBalancer*-Klasse wurde eine Schnittstelle zur Metis-Bibliothek geschaffen [83]. Dazu müssen Metis topologische Informationen in Form eines Graphen zur Verfügung gestellt werden; im Fall von CEQ bedeutet dies, den Nachbarschaftsgraph der Elemente aus dem verteilten Gitter zu erzeugen und auf einem Prozessor (auf dem danach Metis gestartet wird) zu versammeln. Diese Variante zeigt die Anbindung fertiger Pakete zur Lastbalancierung an DDD-basierte Anwendungen.

Lasttransfer mittels DDD-Funktionen

Von der *LoadBalancer*-Klasse wird jedem Element o_e die neue Partition $\mathrm{dest}(o_e)$ mitgeteilt; danach wird das DDD-Transfermodul benutzt, um die Verteilung des Gitters zu ändern. An dieser Stelle wird die im letzten Abschnitt beschriebene Überlappungsstrategie erzeugt, gleichzeitig muß eine vorher bereits vorhandene Gitterüberlappung entfernt werden. Der in Abb. 5.4 schematisch dargestellte Algorithmus leistet dies. Dabei wird nur auf die *Element*-Objekte eingegangen, *Side*-Objekte werden im Programm jedoch gleichzeitig behandelt.

Der Algorithmus in Abb. 5.4 hat zwei Vorverarbeitungsschritte: zunächst wird ein DDD-Interface zwischen *Master*- und *Ghost*-Objekten verwendet, um den *Ghost*-Elementen die neue Partition der zugehörigen *Master*-Elemente mitzuteilen. Im zweiten Schritt wird über dasselbe Interface ein codiertes Kommando geschickt, das dem *Ghost*-Element mitteilt, ob es in der neuen Partitionierung noch benötigt oder gar zu einem *Master*-Element befördert wird.

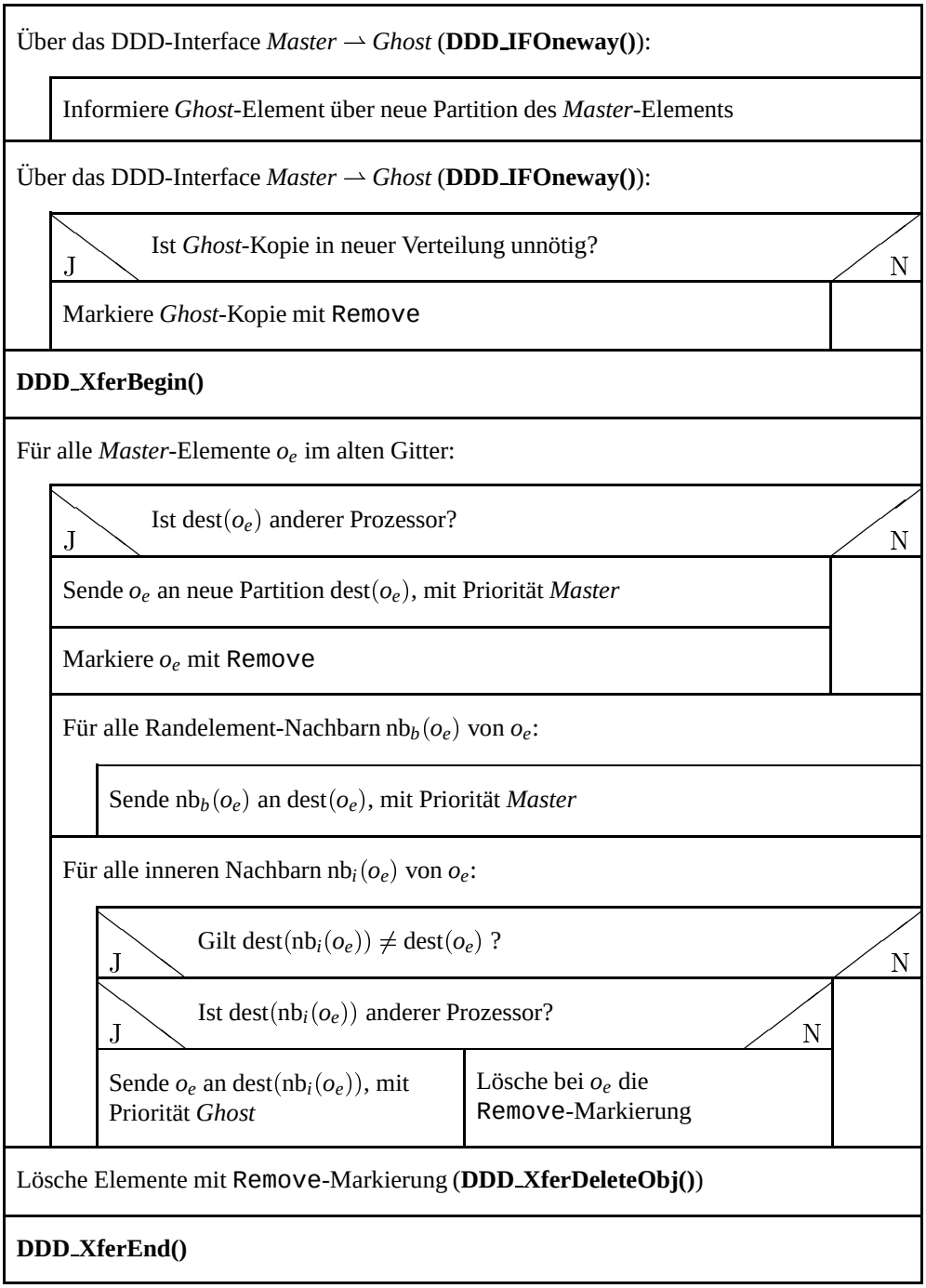


Abbildung 5.4: *Algorithmus zum Lasttransfer in CEQ. Sowohl das Senden von Elementen (mittels* **DDD_XferCopyObj()***) als auch das Löschen von Elementen (mittels* **DDD_XferDeleteObj()***) werden zunächst von DDD aufgezeichnet und erst in* **DDD_XferEnd()** *ausgeführt.*

Im eigentlichen Transferschritt setzen nur *Master*-Elemente **DDD_XferCopyObj()**- und **DDD_XferDeleteObj()**-Kommandos ab, es ist also für die Neuverteilung jedes verteilten Elementobjekts stets der Prozessor verantwortlich, welcher in der alten Verteilung die Kopie mit *Master*-Priorität besitzt. Die `Remove`-Markierung wird zunächst gesetzt, falls ein Element lokal überflüssig wird; später kann diese Markierung jedoch wieder entfernt werden, falls dieses Element in der neuen Verteilung als *Ghost*-Element benötigt wird.

Durch den Lasttransfer ändern sich die Gitterverteilung, die Gitterüberlappung und damit auch sämtliche definierten DDD-Interfaces. Die Funktion **DDD_XferEnd()** stellt jedoch selbständig konsistente Interfaces her, so daß dieser komplizierte Aspekt in der DDD-Bibliothek versteckt bleibt und nicht die Anwendung unnötig mit technischen Details anreichert.

Konnektivitätsrelationen als DDD-Datenobjekte

Zur Repräsentation von Referenzen zwischen Objekten stehen unterschiedliche Möglichkeiten zur Verfügung; in der bisherigen Arbeit wurden hauptsächlich einfache Implementierungen der Referenzen zugrundegelegt, z.B. *pointer* in ANSI C (d.h. Speicheradressen). Im CEQ-Löser jedoch wurden Referenzen in Form einer eigenen Klassenhierarchie implementiert, die in abstrakter Weise die *Konnektivitätsrelation* (S. 124) zwischen den verschiedenen Objekttypen implementiert. Jedes geometrische Objekt in CEQ speichert also eine Liste der Aufwärts- und eine Liste der Abwärtskonnektivitäten (vgl. Abb. 5.1). Jedes dieser Konnektivitätsobjekte enthält lediglich die Speicheradresse (also wiederum den C-*pointer*) des referenzierten Objekts und minimale Kontrollinformation.

Würde man nun die Konnektivitätsobjekte als DDD-Objekte darstellen, hätte jede einzelne Referenz den Speicheroverhead der `DDD_HEADER`-Struktur (vgl. Abschnitt 3.1.1). Dies wäre nicht tragbar, da die im sequentiellen Programm sehr kleinen Konnektivitätsobjekte in hoher Zahl auftreten. In diesem Fall ist der Einsatz von bei DDD registrierten Datenobjekten sehr sinnvoll, da diese keinen zusätzlichen `DDD_HEADER` und damit auch keinen Speicheroverhead erhalten. Referenzen von diesen Objekten auf DDD-Objekte können jedoch aufgelöst werden (Abschnitt 2.4.1).

Beim Transfer von Elementen und davon abhängigen geometrischen Objekten muß nun jeweils Aufwärts- und Abwärtskonnektivität zusätzlich verpackt und auf der Empfängerseite neu etabliert werden. Dies ist mit dem DDD-Transferkommando **DDD_XferAddData()** möglich (vgl. Abschnitte 2.4.3, 3.4.3). Dem Objekt werden beim Transfer also zwei Datenobjekttabellen zugeordnet, in denen die Konnektivitätslisten gespeichert werden. Auf Senderseite muß ein benutzerdefinierter Handler die Konnektivitätslisten in die Datenobjekttabellen füllen; auf der Empfängerseite muß ein entsprechender Handler die Datenobjekttabellen auspacken und deren Inhalt mit eventuell bestehenden Konnektivitätslisten vereinigen.

5.2.5 Parallele Gitterverfeinerung

Beim Prozeß der Gitterverfeinerung werden zunächst Elemente abhängig von einem Fehlerkriterium zur Verfeinerung markiert, zusätzlich werden Schnittebenen (bzw. -geraden) festgelegt, die

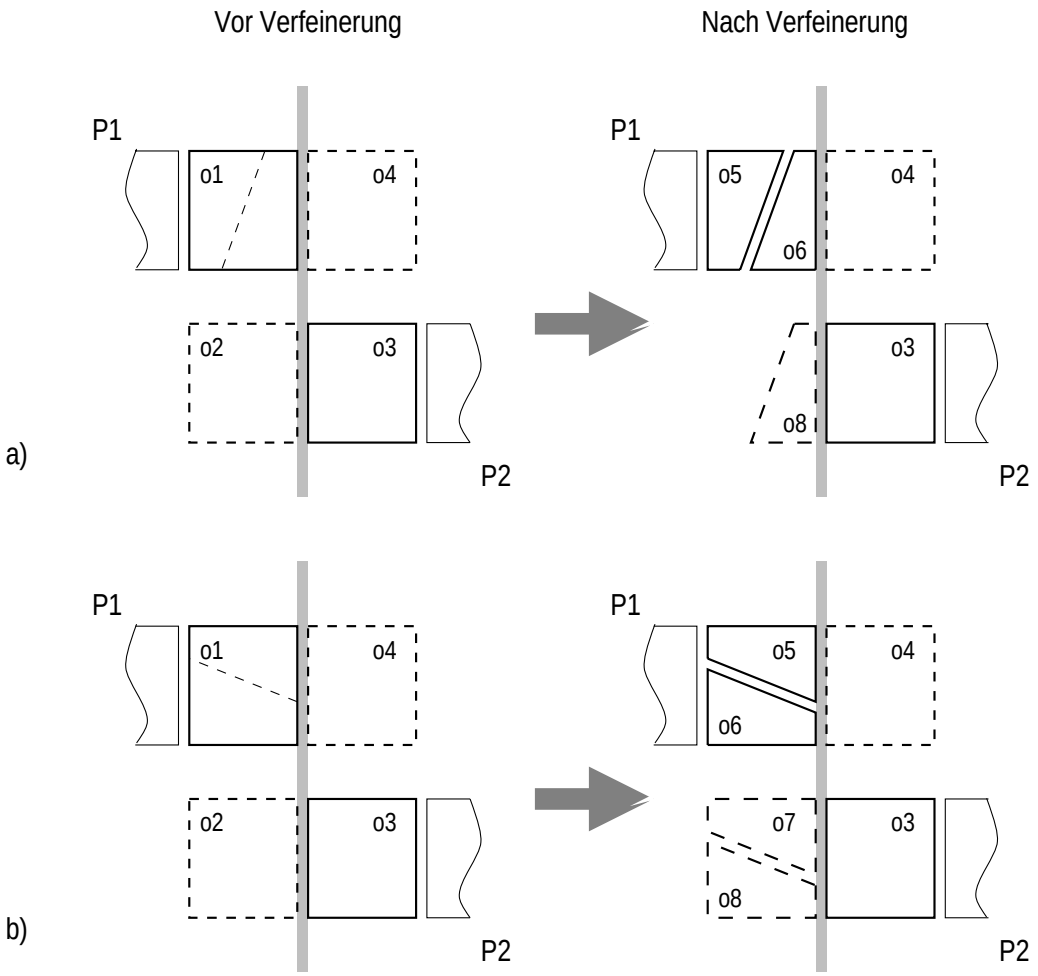


Abbildung 5.5: *Zwei Fälle der Gitterverfeinerung in CEQ am zweidimensionalen Beispiel. Jeweils wird Element o_1 auf Prozessor P1 zur Verfeinerung markiert; da eine* Ghost*-Kopie o_2 auf Prozessor P2 existiert, muß diese angepaßt bzw. erweitert werden. Die Prozessorgrenze ist durch einen grauen Balken dargestellt. Die* Master*-Elemente sind mit durchgezogenen, ihre* Ghost*-Kopien mit gestrichelten Linien dargestellt.*

die Verfeinerungsrichtung bestimmen. In einem zweiten Schritt werden alle markierten Elemente tatsächlich in zwei kleinere Elemente geteilt (Abschnitt 5.2.1). Im parallelen Fall muß zusätzlich dafür Sorge getragen werden, nach der Gitterverfeinerung erneut die in Abschnitt 5.2.3 beschriebene konsistente Überlappung der Elemente zu erhalten. Im Inneren jeder Partition ist dies unproblematisch, da sowieso keine Überlappung existiert. Im Überlappungsbereich müssen jedoch ggf. alte *Ghost*-Elemente entfernt und durch neue ersetzt bzw. ergänzt werden.

In Abb. 5.5 sind die beiden möglichen Fälle dargestellt. Dabei gibt es vier verteilte Objekte $\{o_1, o_2\}$, $\{o_3, o_4\}$, $\{o_5, o_7\}$ und $\{o_6, o_8\}$ (*Master*-Element jeweils erstgenannt); links ist jeweils die Markierung von o_1 samt Verfeinerungsrichtung dargestellt, rechts die konsistente Überlappungssituation nach der Verfeinerung. In Fall (a) ist durch die Elementteilung auf Prozessor P1 die Elementseite

auf der Prozessorgrenze nicht betroffen, daher wird das *Ghost*-Element o_2 nur durch das kleinere Element o_8 ersetzt. Das neu entstandene Element o_6 liegt also an der Prozessorgrenze, Element o_5 hingegen im Inneren des Prozessors P1. In Fall (b) wird durch die Elementteilung auf P1 die Elementseite auf der Prozessorgrenze ebenfalls geteilt; dies hat zur Folge, daß auf P2 zwei *Ghost*-Elemente o_7 und o_8 entstehen müssen.

Damit es durch die verteilte Ausführung der Teilungsoperationen bei gleichzeitiger Teilung benachbarter Elemente (z.B. Elemente o_1 und o_3 in Abb. 5.5) nicht zu Kollisionen kommt, wird der gesamte Verfeinerungsvorgang durch geeignete Colorierung des Nachbarschaftsgraphen an den Prozessorgrenzen künstlich in wenige Teilschritte aufgebrochen und somit sequentialisiert (stets ca. 2-4 Phasen). Aus Mengen von Elementen, die paarweise nicht gleichzeitig verfeinert werden dürfen, wird stets nur ein Objekt zur Verfeinerung zugelassen. Nach jeder dieser Iterationen wird das DDD-Transfermodul eingesetzt, um in einem **DDD_XferBegin()**/**DDD_XferEnd()**-Zyklus die neu entstandenen Elemente zusammen mit allen abhängigen Daten (Seiten, Konnektivitätslisten etc.) an die Nachbarprozessoren als *Ghost*-Kopien zu schicken.

Alle Einzelschritte der Parallelisierungsstrategie für den Verfeinerungsalgorithmus sind ebenso für dreidimensionale Gitter gültig. Sämtliche Entscheidungen basieren auf den Elementseiten, die bei jeder Problemdimension stets zwei angrenzende Elemente besitzen.

Zur Vermeidung des iterativen Vorgehens bei der verteilten Gitterverfeinerung könnte ein Vorverarbeitungsschritt eingeführt werden, der für zwei Elemente an einer Prozessorgrenze bereits die genauen Teilungspunkte ermittelt. Dafür gibt es in der zweidimensionalen Version folgende vier Möglichkeiten (Fall (a) und Fall (b) jeweils aus Abb. 5.5):

1. Die Seite an der Prozessorgrenze wird dreigeteilt (zweimal Fall (b)).
2. Die Seite an der Prozessorgrenze wird zweigeteilt, da der Abstand der beiden Teilungspunkte kleiner ist als die gewählte Schnittebenendicke (zweimal Fall (b)).
3. Die Seite an der Prozessorgrenze wird zweigeteilt (ein angrenzendes Element wird nach Fall (a) und das andere nach Fall (b) geteilt).
4. Die Seite an der Prozessorgrenze wird nicht geteilt (zweimal Fall(a)).

Jede dieser vier Möglichkeiten muß vorab beiden Prozessoren bekannt sein; damit kann jeder Prozessor auch die Teilung der *Ghost*-Elemente selbst durchführen. Nach dem einzigen Teilungsschritt müssen mit Hilfe des DDD-Identifymoduls die neuen *Master*- an die neuen *Ghost*-Elemente angekoppelt werden. Diese Möglichkeit wäre weniger aufwendig im Hinblick auf Kommunikation zwischen den Prozessoren und damit effizienter (die dreidimensionale Version hätte allerdings weit mehr Fälle zu berücksichtigen). Ein Beispiel für den Einsatz der DDD-Identifikation im Kontext der Gitterverfeinerung gibt Abschnitt 5.3.3.

5.2.6 Parallele Ein-/Ausgabe von Gittern

Während einer aufwendigen parallelen Rechnung ist es ratsam, von Zeit zu Zeit Zwischenergebnisse auf dem Dateisystem zu sichern, um im Falle eines ungewollten Abbruchs des Rechenvorgangs

(z.B. beim Ausfall einer Komponente des Parallelrechners) ein Wiederaufsetzen zu ermöglichen (sog. *checkpointing*). In einer Rechnung auf unstrukturierten, lokal verfeinerten Gittern müssen sowohl die Gittertopologie als auch die numerischen Daten gesichert werden, um einen Wiederaufsetzpunkt zu erzeugen. Die Datenmenge einer großen Simulationsrechnung verbietet es jedoch, alle Daten auf einem Prozessor zu sammeln und dort sequentiell dem Dateisystem zu übergeben; dieser sequentielle Flaschenhals würde die Effizienz des gesamten parallelen Verfahrens vernichten.

In der Parallelisierung des CEQ-Verfahrens wurde eine prozessorweise Methode zum Schreiben und Lesen von verteilten Gittern (mit ihren numerischen Daten) implementiert. Jeder Prozessor schreibt sein Teilgitter und die darauf bezogenen numerischen Daten mit standardisierten Betriebssystembefehlen auf je eine Datei. Zusätzlich von diesen bereits in der sequentiellen Version nötigen Daten werden im parallelen Fall zusätzliche Informationen geschrieben, die die Verkopplung der Teilgitter nach dem Einlesen ermöglichen sollen.

Bei der Ausgabe von Gittern wird den Elementseiten an der Grenze zwischen zwei Prozessoren von diesen jeweils die globale ID und die Prozessornummer des benachbarten Prozessors beigefügt. Wegen der symmetrischen Situation der Gitterüberlappung entstehen für jede Elementseite, die nicht im Inneren einer Prozessorpartition liegt, in zwei Dateien zwei symmetrische Einträge.

Beim Einlesen kann mit dieser Information die Verkopplung der Teilgitter in folgenden Schritten durchgeführt werden:

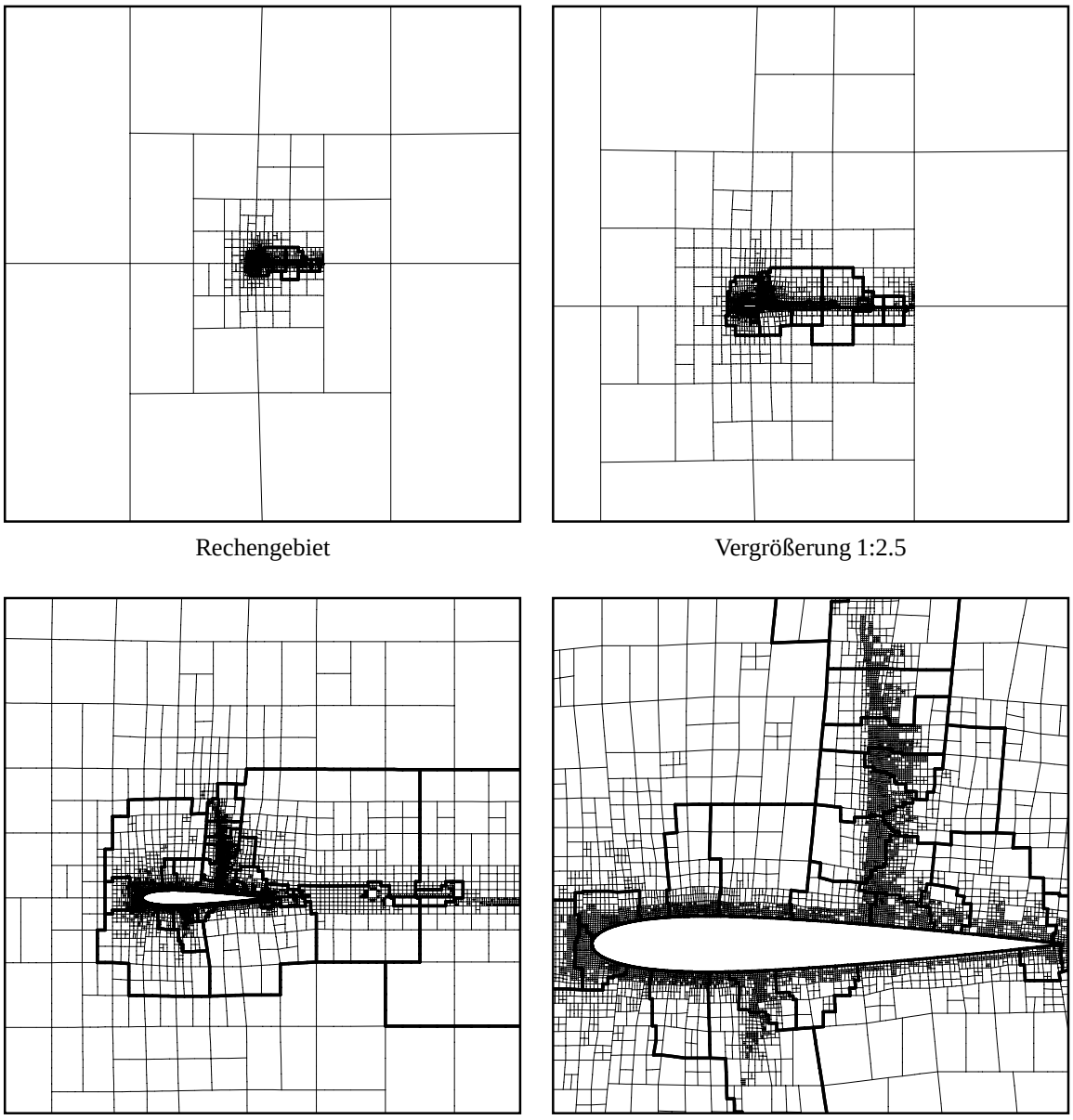
1. Die Elementseiten werden mit Hilfe der alten globalen IDs und der jeweiligen Nachbarprozessornummer identifiziert. Dazu wird für jede solche Elementseite einmal **DDD_IdentifyNumber()** aufgerufen, wobei als Identifikator eben diese alte globale ID dient. Der Partnerprozessor gibt aus Symmetriegründen ein entsprechendes Kommando. Nach diesem Schritt sind alle Elementseiten verknüpft.

2. Im gleichen Identifikationsvorgang werden die Ränder der Elementseiten identifiziert. Dies sind im zweidimensionalen Programm die Eckpunkte (*Node*-Objekte), im dreidimensionalen die Kanten (*Edge*-Objekte). Als erster Einzelidentifikator dient dabei die zu identifizierende Elementseite selbst, diese Angabe wird erst durch die in Abschnitt 3.3.3 beschriebene Technik der hierarchischen Identifikation möglich.

   Damit die Randobjekte derselben Elementseite vom DDD-Identifymodul unterschieden werden können, muß als zusätzlicher Einzelidentifikator eine lokale Numerierung angegeben werden, z.B. die Kantenorientierung im zweidimensionalen CEQ.

3. In einem abschließenden DDD-Transfervorgang werden für jede an einer Prozessorgrenze liegende Elementseite durch **DDD_XferCopyObj()** *Ghost*-Elemente auf benachbarten Prozessoren erzeugt. Nach diesem Schritt ist die verteilte Überlappungssituation wie in Abschnitt 5.2.3 beschrieben wiederhergestellt.

Unter Verwendung der hierarchischen Identifikation des DDD-Identifymoduls und des DDD-Transfermoduls kann also ein komplexer Ein-/Ausgabevorgang mit kleinen Ergänzungen der Ausgabedaten einfach und effizient durchgeführt werden. Eine weitere, ähnliche Anwendung der hier vorgestellten Identifikationstechnik wird im nächsten Beispielprojekt in Abschnitt 5.3.3 beschrieben.



Rechengebiet | Vergrößerung 1:2.5

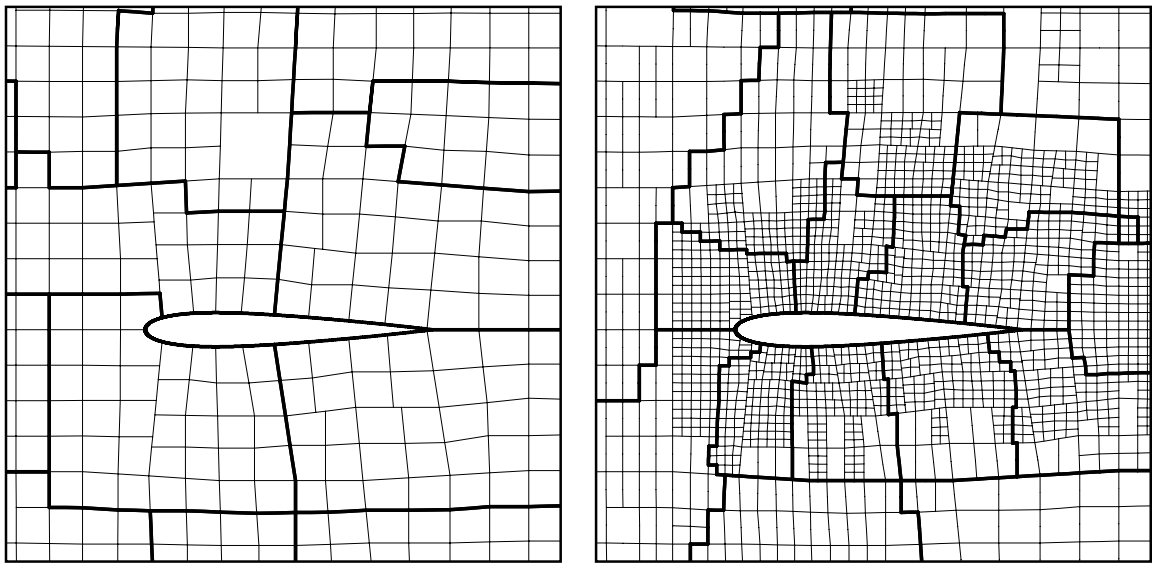
Vergrößerung 1:8 | Vergrößerung 1:30

Abbildung 5.6: *Adaptives Rechengitter und drei Vergrößerungen mit Prozessorgrenzen, für Simulationszeit* $t = 0.292$. *Testfall NACA-0012, 2D-Profil, Anstellwinkel* 1.25°, *Machzahl 0.8. Die Berechnung erfolgte auf 32 Prozessoren einer Intel Paragon.*

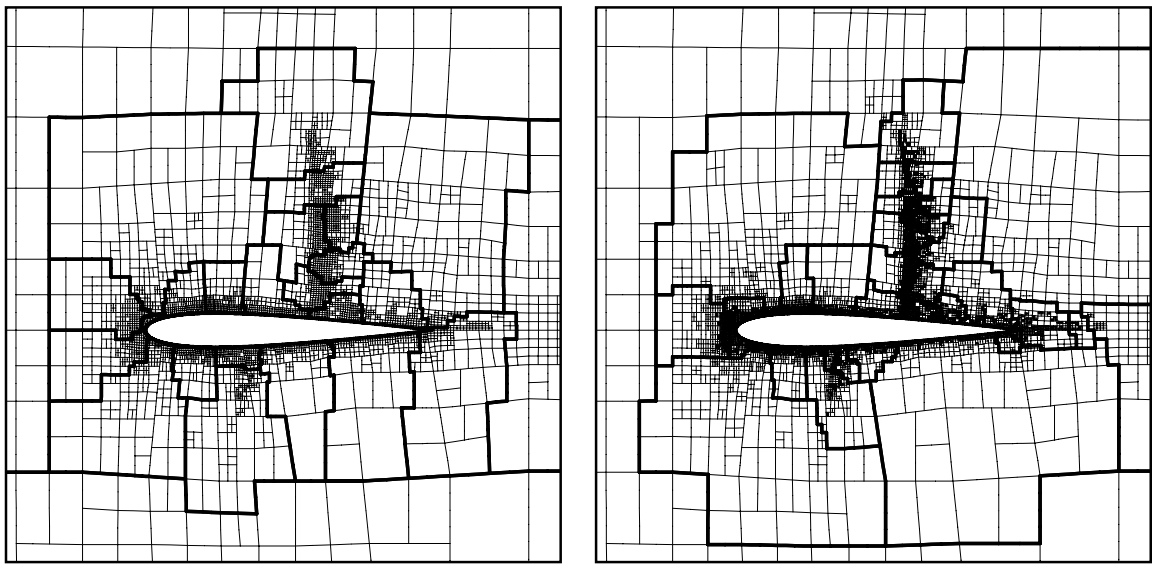
5.2.7 Beispiele und Effizienzbetrachtungen

Testbeispiel NACA-0012

Im folgenden Abschnitt sollen die qualitativen und quantitativen Leistungsmerkmale der Parallelversion von CEQ untersucht werden. Dazu wird nun als Beispielsimulation folgender zweidimensiona-



$t = 0.066$, $N_{\text{elem}} = 1186$ $t = 0.226$, $N_{\text{elem}} = 2613$



$t = 0.276$, $N_{\text{elem}} = 5073$ $t = 0.292$, $N_{\text{elem}} = 11373$

Abbildung 5.7: *Vier adaptive Rechengitter und die durch Lastverteilung entstehenden Prozessorgrenzen zu verschiedenen Simulationszeiten (Ausschnitte, Vergrößerung 1:16). Testfall NACA-0012, 2D-Profil, Anstellwinkel* 1.25°*, Machzahl 0.8. Simulation auf 32 Prozessoren einer Intel Paragon.*

ler Testfall untersucht: Ein NACA-0012-Profil (Anstellwinkel 1.25°) wird von links mit 0.8 Mach angeströmt. Im diskretisierten Problemgebiet wird die reibungsfreie Strömung (Eulergleichung) berechnet. Zunächst besteht das Rechengitter aus *einem* Polygon, das Profil wird durch einen inneren Rand repräsentiert. Konzeptionell kann ein solcher Datensatz ohne vorherige Gittergenerierung direkt vom CAD-System importiert werden [66]. Aus dieser Anfangsgeometrie wird automatisch

durch wenige horizontale und vertikale Gitterteilungen ein Startgitter aus 28 polygonalen Elementen erzeugt, mit dem die Simulation gestartet wird.

Abb. 5.6 zeigt das diskretisierte Problemgebiet und drei Vergrößerungen am Ende des Simulationsprozesses. Die Unstetigkeiten der Lösung durch die entstehenden Druckstöße ober- und unterhalb des Profils wurden durch den adaptiven Gittergenerierungsprozeß detektiert und sind in der detailliertesten Vergrößerung (rechts unten in Abb. 5.6) gut zu erkennen. In Abb. 5.7 sind ausschnittweise vier Zustände des Gitters zu verschiedenen Simulationszeiten im expliziten Zeitschrittverfahren wiedergegeben. Durch den Prozeß der Gitterverfeinerung (und -vergröberung [144]) steigt die Zahl der Kontrollvolumen von $N_{\text{elem}} = 28$ zu Beginn der Simulation auf $N_{\text{elem}} = 11373$ Elemente bei Erreichen der vorgegebenen Genauigkeit. Diese Entwicklung wird durch geeignete Fehlerschranken und Schwellwerte im vorgegebenen „Ablaufplan“ gesteuert.

Die parallele Rechnung erfolgte auf 32 Prozessoren eines Parallelrechners vom Typ Intel Paragon; die hervorgehobenen Linien in den Abb. 5.6 und 5.7 stellen die Prozessorgrenzen dar. Die Lastbalancierung wurde unter Verwendung der *Metis*-Bibliothek [83] durchgeführt; dadurch entstehen Prozessorgrenzen mit möglichst wenigen Elementen am Interface. Das ebenfalls einsetzbare Verfahren der *rekursiven Koordinatenbisektion* (RCB) ergibt tendenziell längere Interfaces und damit einen höheren Kommunikationsaufwand.

In Abb. 5.6 erkennt man die Häufung von Partitionen in der Umgebung des NACA-Profils. Ca. 99% der Fläche des Problemgebiets ist nur einem der Prozessoren zugeordnet, die restlichen 1% Fläche enthalten ca. 97% der Kontrollvolumen und können somit 31 Prozessoren beschäftigen. Im Simulationsablauf (Abb. 5.7) läßt sich erkennen, wie sich die Partitionen durch die dynamische Lastbalancierung (unter Verwendung des DDD-Transfermoduls) mit feiner werdendem Gitter immer mehr in der unmittelbaren Umgebung des Profils und den Unstetigkeiten der Lösung konzentrieren.

Abb. 5.8 zeigt die Lösung in Form von Isolinien des Drucks. Als dünne Linien sind wiederum die Grenzen der Prozessorpartitionen dargestellt. Die Isolinien sind unstetig, da die nur stückweise stetige Repräsentation der Lösung in der Finite-Volumen-Diskretisierung exakt visualisiert wird. Die Lösung zeigt jedoch keine Abhängigkeiten oder Artefakte durch kreuzende Partitionsgrenzen, an denen während der Simulation die Interface-Kommunikationen stattfinden.

Quantitative Ergebnisse

Für den oben angegebenen Testfall werden nun noch Ergebnisse aus Rechnungen mit verschiedenen Prozessoranzahlen P aufgeführt und diskutiert. Die Zielsetzung des Verfahrens liegt in der transparenten Verfügbarkeit im Rahmen des gesamten Entwicklungszyklus (vgl. Abschnitt 5.2.1). Daher werden im folgenden keine Werte für einzelne Bestandteile der Simulation (z.B. einzelne Iterationen) gemessen, sondern alle Werte auf den gesamten Ablauf bezogen angegeben. Damit der Simulationsvorgang auch auf einem Prozessor sinnvoll durchgeführt werden kann, wird die Gesamtzahl von Elementen im Lösungsgitter auf $N_{\text{elem}} \approx 3000$ Kontrollvolumen beschränkt.

Die wichtigsten Resultate sind in Tabelle 5.1 zusammengefaßt. Die Tabelle ist dreiteilig: im ersten Teil werden Laufzeiten, im Mittelteil statistische Zahlen aus CEQ und im unteren Teil DDD-bezogene Werte angegeben. Die Gesamtlaufzeit T_{all} (in $[\text{sec}]$) beinhaltet alle Vorgänge vom Start des

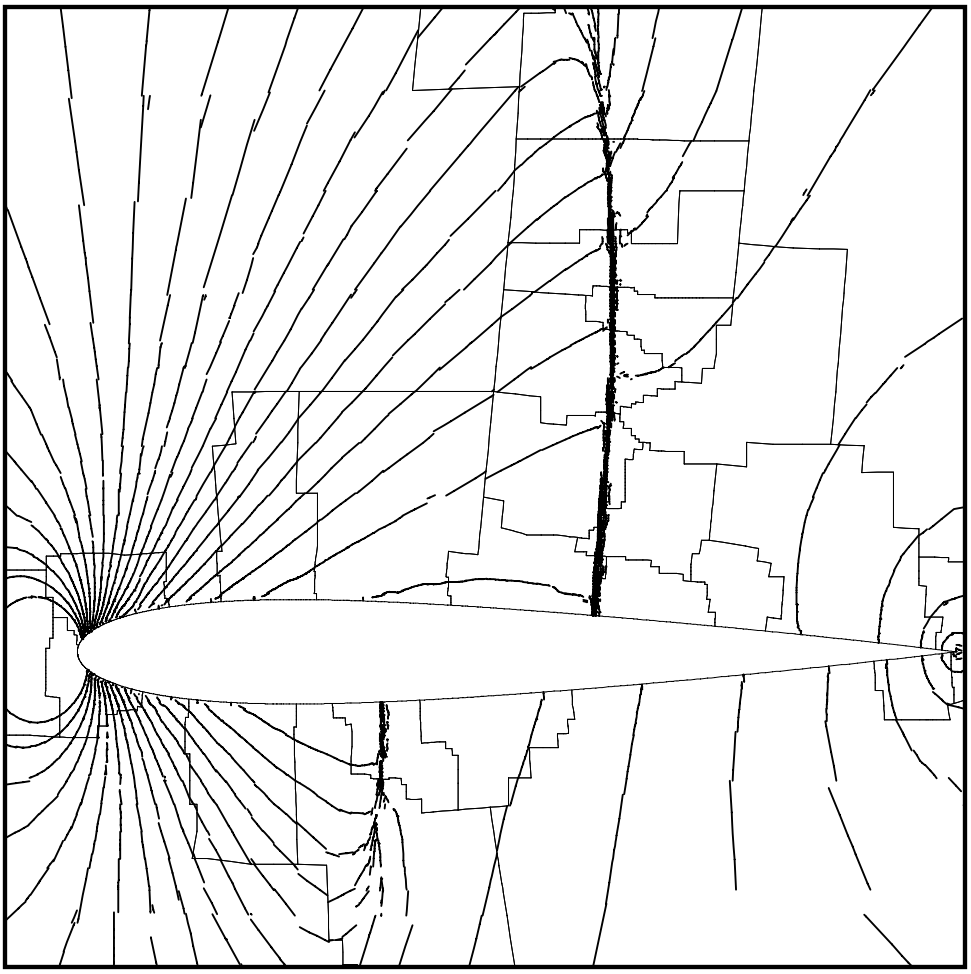


Abbildung 5.8: *Isolinien des Drucks mit Prozessorgrenzen; Simulationszeit* $t = 0.292$*, Vergrößerung 1:30. Die Druckstöße ober- und unterhalb des Profils sind gut erkennbar. Testfall NACA-0012, 2D-Profil, Anstellwinkel* 1.25°*, Machzahl 0.8. Simulation auf 32 Prozessoren einer Intel Paragon.*

Programms bis zur Terminierung (sog. *wallclock time*). Daraus läßt sich die Effizienz des parallelen Programms bei fester Problemgröße durch

$$E^f_{all}(P) = \frac{T_{all}(1)}{PT_{all}(P)}$$

berechnen. Setzt man die in allen DDD-Interfacekommunikationen verbrauchte Zeit zur Gesamtzeit ins Verhältnis, bekommt man ein Maß für den Kommunikationsanteil des Programms T_{if}/T_{all} (in %).

Im Mittelteil der Tabelle ist zunächst die Anzahl der bis zum Erreichen der vorgegebenen Genauigkeit gerechneten Iterationen (Zeitschritte) N_{IT} angegeben. Dieser Wert und die Gesamtzahl der Kontrollvolumen zum Schluß der Simulation N_{elem} sind in den einzelnen Läufen nicht exakt gleich,

Tabelle 5.1: *Testfall NACA-0012 auf verschiedenen Prozessorzahlen eines Parallelrechners vom Typ Intel Paragon.*

P	1	2	4	8	16	32
Laufzeiten und Effizienzen						
T_{all} [sec]	15537	8251	4626	2786	1668	982
E^f_{all} in %	100	94	84	70	58	49
T_{if}/T_{all} in %	0	5	19	22	40	46
CEQ-Statistik						
N_{IT}	9298	9294	9294	9294	9295	9295
N_{elem}	3001	2922	2895	2855	2851	2877
$\bar{N}^p_{\text{elem}}$	3001	1461	723	356	178	89
DDD-Statistik						
$\bar{n}_{obj}$	16526	8100	3957	1888	1057	525
$\bar{n}_{cpl}$	0	258	295	271	132	194
$\overline{\mathsf{cpl}}^p_X$	0	258	304	281	138	216
$\overline{\mathsf{cpl}}^p_X/\bar{n}_{obj}$ in %	0	3.2	7.6	14.9	13.1	41.1

da die parallele Gitterverfeinerung je nach Lastverteilung minimal unterschiedliche Gitter erzeugt (siehe Abschnitt 5.2.5). Diese Unterschiede wirken sich wiederum auf die Markierung zur folgenden Verfeinerung aus. Die durchschnittliche Anzahl von Elementen pro Prozessor $\bar{N}^p_{\text{elem}}$ sinkt mit steigender Prozessorzahl ab und erreicht für $P = 32$ schließlich 89 Elemente.

Aus Sicht der DDD-Bibliothek speichert jeder Prozessor am Ende der Simulation durchschnittlich etwa $\bar{n}_{obj}$ Objekte; an der Differenz zur Elementzahl $\bar{N}^p_{\text{elem}}$ lassen sich die Anzahl der *Ghost*-Elemente und die Anzahl von Kanten- bzw. Knoten-Objekten ablesen. Die durchschnittliche Anzahl von Objekten mit nichtleeren Couplinglisten $\bar{n}_{cpl}$ ist für $P > 2$ stets kleiner als die Anzahl von durchschnittlichen Interfaceeinträgen $\overline{\mathsf{cpl}}^p_X$, da Objekte an Teilinterfaces zu mehreren Prozessornachbarn partizipieren können (beim Zusammenstoßen von mehreren Partitionen). Das Verhältnis $\overline{\mathsf{cpl}}^p_X/\bar{n}_{obj}$ ermöglicht eine Aussage über den Anteil der Oberfläche des Teilgraphen zum logischen Volumen der Partition.

Durch die feste Problemgröße wird das Verhältnis von Kommunikation zu Rechenaufwand mit wachsender Prozessoranzahl immer größer; dies läßt sich an den Verhältniszahlen T_{if}/T_{all} (Anteil der Interface-Kommunikation an der Gesamtlaufzeit) und $\overline{\mathsf{cpl}}^p_X/\bar{n}_{obj}$ (Oberfläche zu Volumen der Partitionen) ablesen. Im Fall von 32 Prozessoren ist schließlich das Verhältnis von Oberfläche zu Volumen ca. 41%, was die Ursache für erhöhten Rechenaufwand an der Prozessorgrenze und den hohen Kommunikationsaufwand von 46% darstellt. Gelingt es, eine große Zahl von Prozessoren mit einer Simulation angemessener Größe zu beschäftigen, werden die Anteile für Interfacekommunikation und Doppelberechnungen an der Gesamtlaufzeit kleiner ausfallen.

Zusätzlich enthält die Gesamtzeit T_{all} noch die Laufzeiten für Lastbalancierung und Lasttransfer sowie Synchronisationen zur Bestimmung von globalen Größen und Zuständen (z.B. Evaluierung von Abbruchkriterien). Trotz der Einbeziehung dieser Teilprobleme in die Gesamtbilanz und unangemessen kleiner Problemgrößen für hohe Prozessorzahlen wurde für die Gesamtzeit ein Speedupwert von fast 16 bei 32 Prozessoren erreicht, was durchaus zufriedenstellend ist. Gleichzeitig mit dem erreichten Grad an Modularität und Objektorientierung im Design des parallelen Softwareprodukts CEQ wurde also durch Benutzung des DDD-Modells eine effiziente und portable Implementierung erreicht.

5.3 Das Simulationswerkzeug UG

Das Simulationswerkzeug UG (Unstructured Grids) zur Lösung von Problemen partieller Differentialgleichungen wird am *Institut für Computeranwendungen III* der Universität Stuttgart seit einigen Jahren entwickelt [13]. Der Funktionsumfang des Pakets wächst ständig, sein Programmcode besteht aus ca. 350.000 Zeilen (Stand Juni 1997). Das UG-Paket verbindet effiziente Techniken zur Lösung partieller Differentialgleichungen: unstrukturierte Gitter zur Auflösung komplexer Problemgeometrien, lokal-adaptive Gitterverfeinerung, Mehrgittermethoden zur schnellen Lösung linearer Gleichungssysteme und nicht zuletzt die Parallelisierung in vollem Funktionsumfang auf Basis des DDD-Modells. Sowohl die hierarchische Gliederung des sequentiellen UG-Pakets im besonderen als auch das klare Schichtenmodell von DDD-basierten Anwendungen im allgemeinen unterstützen die programmtechnische Bewältigung der durch die Parallelisierung zusätzlich hinzukommenden Komplexitätsebene.

Abb. 5.9 gibt den Aufbau einer mit dem UG-Paket erstellten Anwendung vereinfacht wieder. Den Kern des Programms macht das *Gittermanager-Subsystem* aus, das unstrukturierte Gitter und hierarchisch aufgebaute Mehrgitter erzeugt und verwaltet. Dabei stehen für zweidimensionale Gitter die Elementtypen Dreieck und Viereck, für dreidimensionale die Elemente Tetraeder, Pyramide, Prisma und Hexaeder zur Verfügung. Der Gittermanager enthält einen mächtigen Algorithmus zur Erzeugung von lokal verfeinerten Gitterebenen auf der Basis des bestehenden Mehrgitters sowie zur selektiven Entfeinerung (also Rücknahme vorheriger Verfeinerungsoperationen) dieser Gitter [94]. Im parallelen Fall setzt der Gittermanager auf der DDD-Bibliothek auf, um verteilte Mehrgitter bereitzustellen.

Der Gittermanager bildet die Basis für das *Numerik-Subsystem*. Dieses enthält zunächst grundlegende BLAS-Routinen (*basic linear algebra* [24]), vor allem aber modular gefaßte numerische Algorithmen von Standard-Glättungsverfahren über CG- und Mehrgitterlöser bis hin zum Lösungsverfahren für nichtlineare, zeitabhängige partielle Differentialgleichungen. In der parallelen Version werden der BLAS-Schicht zusätzliche Routinen hinzugefügt, die über DDD-Interface-Aufrufe elementare Kommunikationsfunktionen implementieren. Da diese Routinen dieselbe Terminologie und Parameterübergabekonventionen wie die sequentiellen BLAS-Routinen verwenden, fügen sie sich nahtlos in das übrige Numerik-Subsystem ein. Die numerischen Algorithmen in der darüberliegenden Schicht werden durch Aufrufe der Kommunikationsroutinen ergänzt, somit wird z.B. aus einem sequentiellen Jacobi-Glätter auf dem Parallelrechner automatisch ein Block-Jacobi-Verfahren mit Kommunikation an den Prozessorgrenzen. Somit ist (zumindest für den Einsatz von

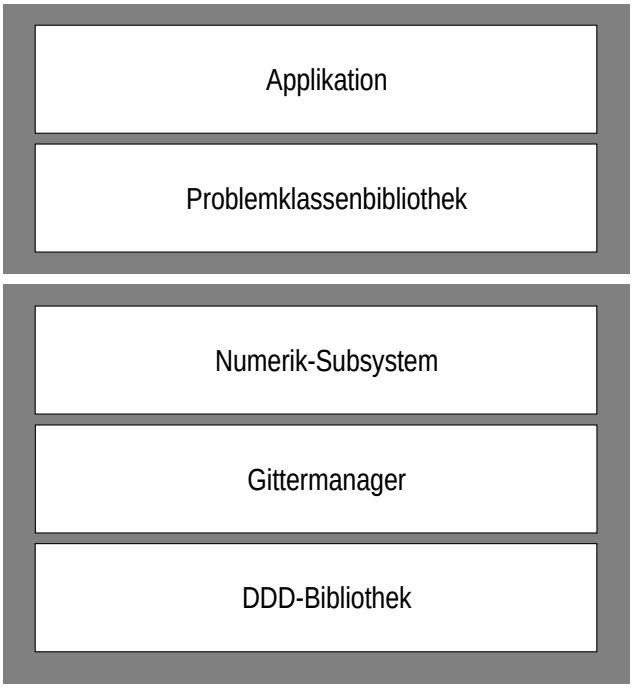


Abbildung 5.9: *Vereinfachtes Schichtenmodell des UG-Pakets.*

Standardverfahren) die Parallelisierung von dieser Schicht des Numerik-Subsystems an aufwärts nahezu transparent gekapselt.

Die verschiedenen *Problemklassenbibliotheken* setzen auf den Funktionen und Algorithmen des Numerik-Subsystems auf. Sie ermöglichen es, neue Applikationen mit einem Minimum an zusätzlichem Programmcode zu entwickeln. Eine Vielzahl von Problemklassen-Modulen wurde entwickelt, z.B. für Konvektions-Diffusions-Probleme, lineare Elastizität, inkompressible Navier-Stokes-Gleichungen, Dichte- und Zweiphasenströmung. Da die numerischen Verfahren aus dem Numerik-Subsystem bereits parallel auf verteilten Gittern zur Verfügung stehen, können Applikationen dieser Problemklassen ohne zusätzlichen Aufwand auf Parallelrechner portiert werden.

5.3.1 Mehrgitter und DDD

Die Abbildung eines einzelnen zwei- bzw. dreidimensionalen Gitters auf das DDD-Graphmodell erfolgt wie in den Abschnitten 5.1.2 bzw. 5.1.3 beschrieben. Ähnlich wie im CEQ-Beispiel wird eine Gitterüberlappung von einer Elementreihe eingeführt (vgl. Abschnitt 5.2.3). Zu dieser *horizontalen* Überlappung innerhalb eines Gitters kommt beim UG-Gittermanager noch die *vertikale* Überlappung zwischen den Gitterebenen, die aus der Forderung entsteht, daß jedes durch Verfeinerung entstandene Element prozessorlokal auf sein Vaterelement im nächstgröberen Gitter oder auf eine entsprechende Kopie dieses Vaterelements zugreifen können muß. Der genaue Algorithmus zum Lasttransfer in UG (entsprechend Abb. 5.4 im CEQ-Projekt) soll hier aus Platzgründen nicht wiedergegeben werden; eine formale Spezifikation der zugrundeliegenden Überlappungsstrategie

ergibt sich jedoch durch Kombination der deklarativen Regelsätze MG und 1-OVPE aus Anhang C. Diese regeln die Überlappung zwischen Gittern (MG) bzw. innerhalb jedes Gitters (1-OVPE).

Für die Kommunikationsroutinen des Numerik-Subsystems wird eine *gitterebenenweise* Kommunikation benötigt. Zu deren Umsetzung wird die von DDD bereitgestellte Objekt-Eigenschaft $\texttt{attr}(\hat{o})$ (Attribut) eines verteilten Objekts $\hat{o}$ benutzt (Abschnitt 2.3.7). Jedem verteilten Objekt wird als DDD-Attribut seine Gitterebene im Mehrgitterkontext zugeordnet; natürlich haben alle lokalen Objekte eines verteilten Objekts dasselbe Attribut. Die Interface-Kommunikationsroutinen können nun selektiv nach Attribut angesteuert werden; dadurch wird nur auf einzelnen Gittern kommuniziert (vgl. S. 68).

Neben den normalen Konsistenzroutinen für verteilte Gitter, die von der Existenz horizontaler Überlappung herrühren, werden im Mehrgitterkontext (d.h. für die Gittertransfer-Operatoren) ebenfalls Konsistenzroutinen für vertikale Überlappungen benötigt. Diese sammeln Daten von *Ghost*-Objekten, die durch vertikale Überlappung zwischen einzelnen Gittern entstehen, auf den jeweiligen *Master*-Objekten. Dazu werden die Objekt-Prioritäten der verteilten Objekte geeignet festgelegt und DDD-Interfaces auf diesen Prioritäten definiert, die alle nötigen Kommunikationsvorgänge abstrakt fassen. Es ist an dieser Stelle wichtig, die Art der Überlappung in Form der Priorität des *Ghost*-Objekts festzuhalten, die Möglichkeiten sind (vgl. erneut Anhang C):

- $\mathit{Ghost}_{\mathbf{H}}$ für *Ghost*-Objekte aus horizontaler Überlappung
- $\mathit{Ghost}_{\mathbf{V}}$ für *Ghost*-Objekte aus vertikaler Überlappung
- $\mathit{Ghost}_{\mathbf{VH}}$ für *Ghost*-Objekte aus horizontaler *und* vertikaler Überlappung

Darauf aufbauend können zwei DDD-Interfaces

$$X_{\mathbf{H}} = (T\,, \{\mathit{Ghost}_{\mathbf{H}}, \mathit{Ghost}_{\mathbf{VH}}\}, \{\mathit{Master}\})$$

und

$$X_{\mathbf{V}} = (T\,, \{\mathit{Ghost}_{\mathbf{V}}, \mathit{Ghost}_{\mathbf{VH}}\}, \{\mathit{Master}\})$$

zur horizontalen bzw. vertikalen Kommunikation definiert werden. Weiterhin unterstützt das DDD-Transfermodul das Kombinieren von Prioritäten mittels frei definierbarer Operationen, siehe dazu die Nachbemerkung in Anhang B.3.

Bereits in [11, Kap. 3] wurden von P. Bastian entsprechende Kommunikationsroutinen aus Sicht einer numerischen Anwendung vorgestellt und wichtige Begriffe definiert (z.B. konsistente, inkonsistente und eindeutige Repräsentation von Vektoren, Kommunikation in Restriktion und Prolongation); die DDD-Bibliothek bietet nun zusätzlich einfache Implementierbarkeit allgemeiner Konsistenzroutinen, klare Trennung von Aspekten der Numerik, der Datenkonsistenz und der Kommunikation durch allgemein definierte Schnittstellen und nicht zuletzt eine effiziente und portable Umsetzung der zugrundeliegenden abstrakten Konzepte. Zusätzlich bleibt das DDD-Modell allgemein einsetzbar und nicht rein auf eine Anwendung beschränkt; dies wird ermöglicht durch die Bereitstellung von abstrakten verteilten Objekten mit dynamischer Typdefinition, durch die Trennung von DDD-Objekten und registrierten Datenobjekten und durch die allgemein gehaltene Funktionalität der drei DDD-Module *Interfaces*, *Transfer* und *Identifikation*.

5.3.2 Verteilung der Datenstrukturen

Die numerischen Algorithmen in UG setzen nicht direkt auf der geometrischen (und topologischen) Gitterbeschreibung auf; stattdessen wurde eine Zwischenebene aus *Vektor*- und *Matrix*-Strukturen eingeführt, die es ermöglicht, beliebigen geometrischen Objekten (d.h. Knoten, Kanten, Elementseiten, Elementen) Freiheitsgrade zuzuordnen und diese beliebig zu verknüpfen. Jedes Vektorobjekt verwaltet eine Liste von Matrixstrukturen, deren Einträge jeweils eine Kopplung zu einem anderen Vektorobjekt repräsentieren. Vektoren werden als DDD-Objekte definiert, dadurch können die oben beschriebenen Interfaces $X_{\mathbf{H}}$ und $X_{\mathbf{V}}$ aufgebaut und zur Kommunikation z.B. im Sinn von [11, Kap. 3] verwendet werden.

Die Matrixobjekte hingegen erfüllen alle Kriterien eines registrierten Datenobjektes gemäß Abschnitt 5.1.2: sie benötigen nur wenig Speicher, existieren in großer Zahl, referenzieren DDD-Objekte (nämlich Vektorobjekte) und sind selbst eindeutig einem DDD-Objekt zuzuordnen (nämlich dem jeweiligen Vektorobjekt). Daher werden Matrixobjekte beim DDD-TypeManager nicht als DDD-Objekte, sondern als Datenobjekte registriert. Dies führt dazu, daß die DDD-Funktionalität (z.B. Umrechnen von Referenzen beim Transfer) auch für Matrixobjekte ohne zusätzlichen Speicherbedarf genutzt werden kann.

Beim Versenden von Vektorobjekten im DDD-Transfer muß jeweils deren Matrixliste eingepackt und auf der Empfängerseite ausgepackt werden; ist das Vektorobjekt auf der Empfängerseite bereits vorhanden, so müssen die empfangenen Matrixstrukturen mit den schon vorhandenen verschmolzen werden. Dies obliegt den benutzerdefinierten Ein- und Auspackhandlern (vgl. Abschnitte 3.4.3 und 3.4.4). Die so implementierte verteilte Struktur aus Vektorkomponenten und Matrixeinträgen ist also ein Prototyp für die Realisierung verteilter dünnbesetzter Matrizen.

5.3.3 Besonderheiten der parallelen Gitterverfeinerung

Das Gitter, das vom UG-Gittermanager-Subsystem erzeugt und verwaltet wird, besteht im zweidimensionalen Fall aus Dreiecken und Vierecken, im dreidimensionalen aus Tetraedern, Hexaedern, Pyramiden und Prismen. Durch Gitterverfeinerung können bei geeigneter Teilung dieser Gitterelemente neue, feinere Gitterebenen geschaffen werden; bei uniformer Verfeinerung bedeckt die neue Gitterebene das ganze Gebiet, bei lokaler Verfeinerung entstehen nur Teilgitter. In zur Verfeinerung markierten Elementen werden *reguläre Verfeinerungsregeln* angewendet, mit welchen stumpfe Innenwinkel der entstehenden Feingitterelemente vermieden werden. Die benachbarten, nicht regulär verfeinerten Elemente werden entsprechend ihrer Kantenmarkierungen behandelt.

Abb. 5.10 zeigt den Satz von regulären Verfeinerungsregeln. Da bei der parallelen Verfeinerung an Prozessorgrenzen neue Gitterebenen und damit neue verteilte Objekte zunächst lokal entstehen, muß zwischen den Prozessoren Kommunikation stattfinden, um die Nachbarschaften im neu entstandenen Gitter abzugleichen. Dazu wird eine ähnliche Kombination der DDD-Module Transfer und Identifikation angewendet, wie sie bereits für die parallele Ein-/Ausgabe von Gittern in Abschnitt 5.2.6 beschrieben wurde. Die Ähnlichkeit ist dabei nicht zufällig, in beiden Fällen entstehen in verteilter Weise sich überlappende Gitter: in Abschnitt 5.2.6 nach dem Einlesen von Dateien, hier durch den Verfeinerungsalgorithmus.

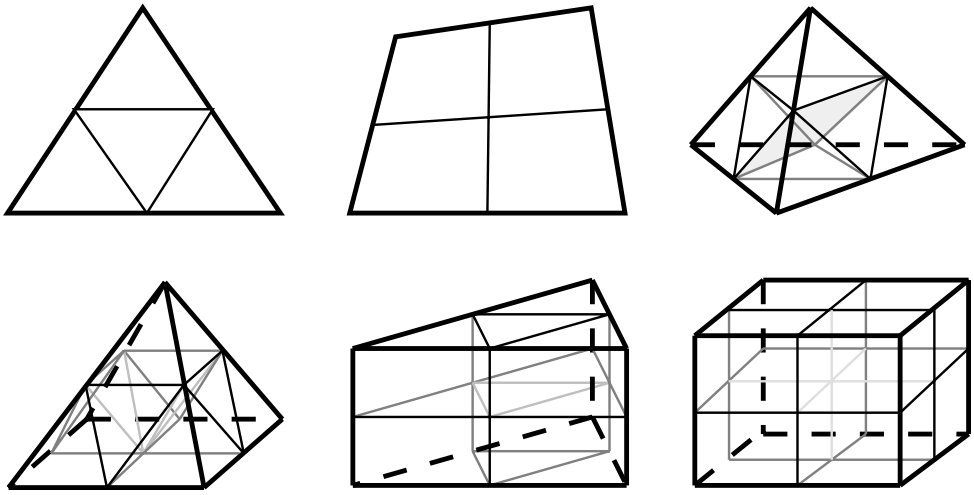


Abbildung 5.10: *Reguläre Verfeinerungsregeln im UG-Gitterverfeinerungsalgorithmus, bei denen die durch Verfeinerung neu entstehenden Winkel die numerischen Eigenschaften nicht beeinträchtigen. Dies sind Regeln für Dreieck und Viereck in zweidimensionalen Gittern bzw. Regeln für Tetraeder, Pyramide, Prisma und Hexaeder in dreidimensionalen Gittern. Es werden jeweils alle Kanten markiert und geteilt, im Fall von Vierecken oder Hexaedern wird ein zusätzlicher innerer Knoten eingefügt.*

Abb. 5.11 gibt den Vorgang der parallelen Verfeinerung in UG vereinfacht wieder; dabei wird die uniforme Verfeinerung zweier Dreieckselemente eines zweidimensionalen Gitters direkt an einer Prozessorgrenze beschrieben. Jedes der beiden *Master*-Elemente (grau) auf dem groben Gitter hat eine *Ghost*-Kopie, alle anderen Objekte auf der Prozessorgrenze sind ebenfalls verteilt (z.B. Knoten, horizontale Pfeile in Abb. 5.11a). Durch Verfeinerung der Grobgitter-*Master*-Elemente wurden bereits je vier Feingitter-*Master*-Elemente erzeugt, auf dem feinen Gitter besteht jedoch weder eine Verkopplung der Knoten noch eine Überlappung der Elemente wie in Abschnitt 5.3.1 spezifiziert. Als nächster Schritt wird daher über DDD-Identify-Kommandos die Identifikation aller Objekte *auf* der Prozessorgrenze eingeleitet (vertikale Pfeile in Abb. 5.11a). Dabei gilt folgende Strategie:

- Alle Feingitterknoten mit direkter Entsprechung (gestrichelte senkrechte Linie) im groben Gitter verwenden diese zur Identifikation. Die Grobgitterknoten sind ja bereits identifiziert. Dies ist stets eindeutig.

- Alle Feingitterknoten ohne direkte Entsprechung (z.B. solche auf der Mitte einer Grobgitterkante) geben beide benachbarten Grobgitterknoten als Identifikatoren an. Ebenso könnten beide benachbarten Feingitterknoten als Identifikatoren angegeben werden, diese werden im gleichen Vorgang identifiziert (siehe vorherigen Punkt).

- Zusätzliche Objekte auf der Prozessorgrenze können sich schließlich über die jeweiligen Feingitterknoten identifizieren. Wurde z.B. jeder Elementseite im Sinn von Abschnitt 5.3.2 ein Vektor zugeordnet (Rauten in Abb. 5.11a), so können die beiden Knoten dieser Elementseite als Identifikatoren benutzt werden.

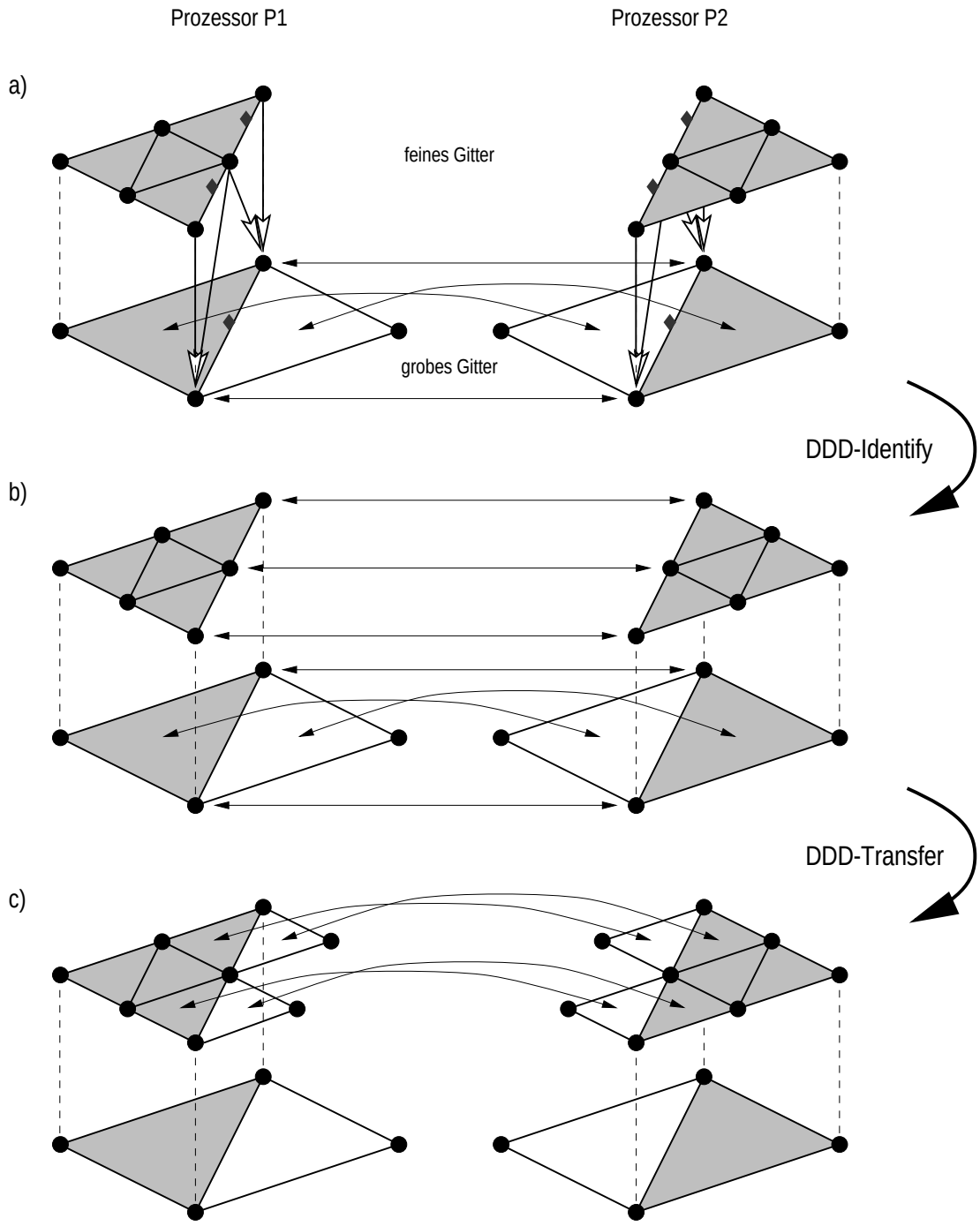


Abbildung 5.11: *Parallele Verfeinerung am 2D-Beispiel. Nach der Erzeugung des feinen Gitters für* Master-*Elemente (grau) folgt Identifikation über bekannte Objekte (a). Nachdem alle Objekte auf der Grenze identifiziert wurden (b), wird die Gitterüberlappung durch DDD-Transferkommandos erzeugt (c).*

Zur Identifikation wird also von der wichtigen Möglichkeit gebraucht gemacht, lokale Objekte über Objekte zu identifizieren, die selbst noch nicht identifiziert sind. Entsprechend der Implementierung des DDD-Identifymoduls (vgl. S. 77) ist also zur Verkopplung der neu entstehenden Gitterteile nur ein Nachrichtenaustausch zwischen benachbarten Prozessoren nötig. Im zweidimensionalen Fall entstehen Abhängigkeitsgraphen mit drei Ebenen, im dreidimensionalen Fall kommt noch eine Ebene hinzu. Die Vorgänge bei der Identifikation sind also kompliziert, die konkrete Anwendung dagegen einfach. Die DDD-Identify-Funktionalität in ihrer abstrakten, mächtigen Form steht bislang in keinem anderen Programmpaket vergleichbar zur Verfügung.

In Abb. 5.11b hat die Identifikation bereits stattgefunden, dies ist durch horizontale Pfeile zwischen den Feingitterknoten angedeutet. Im letzten Schritt kann nun die Gitterüberlappung erzeugt werden, indem die direkt am Rand liegenden *Master*-Elemente auf dem feinen Gitter als *Ghost*-Kopien an den benachbarten Prozessor geschickt werden. Dieser Vorgang basiert auf dem DDD-Transfermodul; über **DDD_XferCopyObj()**-Aufrufe werden Elemente und alle abhängigen Objekte bzw. Daten versendet. Durch die vorherige Identifikation finden die ankommenden *Ghost*-Objekte bereits bekannte verteilte Objekte vor und fügen sich automatisch in das feine Gitter des benachbarten Prozessors ein. Schließlich erhält man das Ergebnis wie in Abb. 5.11c dargestellt: durch die Erzeugung von Feingitter-*Ghost*-Elementen wurde eine zum Grobgitter analoge Überlappungssituation erzeugt.

5.3.4 Quantitative Ergebnisse

Die Betrachtung der Parallelisierung des Simulationswerkzeugs UG soll nun mit einigen quantitativen Resultaten abgeschlossen werden. Dabei soll nicht nur die Laufzeiteffizienz der Parallelisierung untersucht werden, sondern auch der Aufwand zur Erstellung des parallelen Programms selbst.

Programmieraufwand

Zu Beginn dieses Abschnitts 5.3 wurde die programmtechnische Komplexität betont, die aus der Parallelisierung eines komplexen numerischen Verfahrens entsteht. Dabei wurde ein klares Schichtenmodell zur Lösung des Softwareproblems vorgeschlagen; dieser Ansatz soll im folgenden validiert werden.

Ein übliches Maß zur Bestimmung des Arbeitsaufwands zur Erstellung sequentieller und paralleler Programme ist die Anzahl von programmierten Codezeilen (*lines of code*, kurz: LOC). Orientiert am Schichtenmodell von Abb. 5.9 wurden jeweils die gesamte Zeilenzahl und der Anteil der nur zur parallelen Version nötigen Zeilen bestimmt. Tabelle 5.2 zeigt die so ermittelte Statistik.

Das Programmsystem verwendet zusätzlich noch externe Werkzeuge zur Lastbalancierungsstrategie (z.B. Chaco) und die DDD-Bibliothek selbst (zusammen ca. 60.000 Zeilen Programmcode). Unter *Andere Subsysteme* sind vor allem die parallele graphische Ausgabe sowie Benutzerschnittstellen und Geometrieschnittstellen zusammengefaßt. Folgende Punkte sind besonders bemerkenswert:

- Der Gesamtanteil parallelen Programmcodes ist mit 5,5% erfreulich niedrig. Der Löwenanteil von 94,5% ist unabhängig von der Parallelisierung und läßt sich somit getrennt weiterentwickeln und warten.

Tabelle 5.2: *Anteil des Programmieraufwands für die Parallelisierung (Anzahl Codezeilen LOC_{par}) am Gesamtaufwand (Anzahl Codezeilen LOC), aufgegliedert nach Schichtenmodell (Stand Juni 1997). Als Beispiel dient die Finite-Elemente-Applikation.*

Subsystem/Modul	LOC	LOC_{par}	Anteil
Applikation	1700	0	0,0%
Problemklassenbibliothek	31400	100	0,3%
Numerik-Subsystem	50700	1400	2,8%
Gittermanager	60900	1300	2,2%
Andere Subsysteme	95300	3200	3,3%
Schnittstelle UG/DDD		7300	
Gesamt	240000	13300	5,5%

- Der Benutzer von UG hat an der Problemklassenschnittstelle keine zusätzliche Codezeile für Parallelisierung zu entwickeln. Er benutzt alle darunterliegenden Schichten als sequentielles oder paralleles Simulationswerkzeug.

- Die betrachtete Problemklassenbibliothek enthält noch ca. 100 Zeilen parallelen Codes. Auf Dauer werden auch diese Zeilen in den Kern des Pakets übertragen, dann ist zur Parallelisierung neuer Problemklassen keinerlei zusätzlicher Aufwand mehr nötig.

- Tabelle 5.2 betrachtet *nur eine* Problemklasse. Das UG-Konzept sieht jedoch eine beliebige Anzahl von Problemklassen vor, was den Anteil von 5.5% weiter reduziert.

- Der wenige noch notwendige Code zur Parallelisierung ist übersichtlich gekapselt, über die Hälfte (7300/13300) ist in der Schnittstelle UG/DDD versammelt.

Mit den in Abschnitt 6.3.3 vorgeschlagenen Ideen wird gerade der Anteil der Schnittstelle Anwendung/DDD noch reduziert, so daß sich der parallele Codeanteil und damit die Parallelisierungsarbeit in zukünftigen Projekten weiter verringern wird.

Numerische Ergebnisse

In diesem Abschnitt werden schließlich einige numerische Ergebnisse des parallelen UG-Pakets vorgestellt. Dazu werden Messungen von zwei Applikationen beschrieben. Beispiel 1 wurde auf einem Parallelrechner des Typs Intel Paragon durchgeführt, Beispiel 2 auf einem Cray T3E-Rechner (zu Hardware-Details siehe Abschnitt 4.1).

Beispiel 1. Laplace-Gleichung. Das 2D-Problemgebiet (Einheitsquadrat) wird mittels einer Finite-Volumen-Diskretisierung durch ein Grobgitter aus 2x2 Viereckselementen und weiteren, uniform verfeinerten Gittern beschrieben. Die beiden gröbsten Gitter werden auf einem Prozessor gespeichert (Agglomeration). Die Messung in Tabelle 5.3 gibt die Zeit in [sec] für einen Standard-Mehrgitter-V-Zyklus mit lokalem ILU-Glätter und globalem Block-Jacobi-Glätter an

Tabelle 5.3: *Zeit pro parallelem Standard-Mehrgitterzyklus in* [sec]*; Iterationszahl für* 10^{-8} *Reduktion des Residuums. Variierende Gitterweiten h und Prozessorzahlen P.*

P / h (2D)	1	4	16	64
64	0.569 / 8	0.157 / 8	0.063 / 8	0.042 / 9
128	2.234 / 8	0.595 / 8	0.174 / 8	0.069 / 8
256		2.333 / 8	0.627 / 8	0.185 / 8
512			2.346 / 8	0.629 / 8
1024				2.353 / 8

Tabelle 5.4: *Effizienzen je Mehrgitter-Iteration:* E_{IT}^f *bei festem Gitter von 128x128 Elementen bzw.* E_{IT}^s *bei 128x128 Elementen pro Prozessor.*

P / Maß	4	16	64
$E_{IT}^f(P)$ in %	93.9	80.2	50.6
$E_{IT}^s(P)$ in %	95.7	95.2	94.9

(Dämpfungsfaktor 0,9; ein Vor- und ein Nachglättungsschritt). Zusätzlich ist die Anzahl notwendiger Iterationen für eine Reduktion des Residuums von 10^{-8} angegeben (diese Anzahl variiert aufgrund des globalen Block-Jacobi-Glätters mit der Anzahl der Prozessoren).

Aus Tabelle 5.3 lassen sich feste und skalierte Effizienzen entnehmen. Hält man die Problemgröße X bei variabler Anzahl von Prozessoren P konstant, ergibt sich die Effizienz E_X^f aus

$$E_X^f(P) = \frac{T_X(1)}{PT_X(P)},$$

wobei $T_X(p)$ die Ausführungszeit für Problem X auf p Prozessoren bezeichnet. Falls die Problemgröße proportional mit der Anzahl der Prozessoren wächst, ergibt sich die Effizienz E_X^s (*scaled efficiency*) aus

$$E_X^s(P) = \frac{T_X(1)}{T_{P\cdot X}(P)}.$$

Im Falle fester Problemgröße (feinstes Gitter mit 128x128 Elementen) und Prozessoranzahl P speichert jeder Prozessor $n_p = 16384/P$ Elemente; die Mehrzahl der Prozessoren (im Gebietsinneren) hat dann vier Teilinterfaces zu Nachbarprozessoren mit der Gesamtlänge

$$l_p = 4\cdot(\sqrt{n_p}+1).$$

Da sowohl der Kommunikationsaufwand als auch der Aufwand für redundante Berechnungen am Prozessorrand proportional zu l_p ist (vgl. Abschnitt 4.3.1), nimmt der relative Mehraufwand mit steigender Prozessorzahl zu und damit die Effizienz $E_{IT}^f(P)$ ab. Liegt die Prozessorzahl dagegen

Tabelle 5.5: *Iterationszahl für 10^{-6} Reduktion des Residuums, für verschiedene Prozessorzahlen P und Gitterebenen l.*

P	1					4						16							64							
l	4	5	6	7	8	4	5	6	7	8	9	4	5	6	7	8	9	10	4	5	6	7	8	9	10	11
it	4	4	4	5	5	7	6	5	6	5	6	8	7	6	7	6	7	7	8	8	7	7	7	8	8	7

in einem angemessenen Verhältnis zur Problemgröße (wie im Fall von $E_{IT}^{s}(P)$), so läßt sich die Skalierbarkeit des Verfahrens gut demonstrieren.

Beispiel 2. Elliptisches Problem mit variablen Koeffizienten [14].

$$\Leftrightarrow \text{div } a(x) \text{ grad } u = 10 \quad \text{in} \quad \Omega = \{x \in \mathbf{R}^2 \mid |x| < 1\}, \quad u(x) = 0 \quad \text{für} \quad |x| = 1$$

$$a(x) = \begin{cases} 100 & x_1^2 + 2x_2^2 > 0.6 \\ 0.1 & 2x_1^2 + x_2^2 < 0.1 \\ 1 & \text{sonst} \end{cases}$$

Es werden folgende Diskretisierungen verwendet: lineare komforme Finite Elemente (P_1), quadratische konforme Finite Elemente (P_2), nichtkonforme P_1-Elemente (CR), Raviart-Thomas-Elemente der Ordnungen 0 und 1 (RT_0, RT_1) und Brezzi-Douglas-Martini-Elemente (BDM). Eine genaue Beschreibung dieser Elemente läßt sich in [147] nachlesen. Eine Gitterebene l besteht aus 4^l Dreieckselementen, gerechnet wurde bis zu Gitterebene 11 mit 4194304 Dreiecken. Zur Lösung wird eine ähnliche Konfiguration wie in Beispiel 1 benutzt; unterschiedlich ist nur die Anzahl der Vor- bzw. Nachglättungsschritte (hier jeweils zwei).

Wie bereits in Beispiel 1 erwähnt, variiert die Konvergenzrate mit der Prozessoranzahl P. Bei einer festen Prozessoranzahl ist die Konvergenzrate dagegen unabhängig von der Gitterweite. Tabelle 5.5 zeigt diese Zusammenhänge (für P_1-Elemente).

In Tabelle 5.6 sind die Zeiten in [sec] für eine Mehrgitter-Iteration (V-Zyklus) für verschiedene Verfeinerungsstufen und Prozessorzahlen aufgestellt. Entsprechend zu Beispiel 1 erkennt man die gute Skalierbarkeit und darüberhinaus die Flexibilität des Programms.

Tabelle 5.6: *Zeit pro paralleler Mehrgitter-Iteration in* [sec] *für verschiedene Diskretisierungen, Prozessorzahlen und Verfeinerungsstufen.*

Disk.	P	$l=4$	5	6	7	8	9	10	11
P_1	1	0.055	0.106	0.337	1.327	5.735			
	4	0.054	0.065	0.123	0.358	1.357	5.807		
	16	0.059	0.069	0.079	0.137	0.386	1.452	6.300	
	64	0.065	0.078	0.077	0.092	0.155	0.406	1.522	6.720
P_2	1	0.253	0.888	3.550	15.280				
	4	0.127	0.287	0.920	3.606	15.260			
	16	0.109	0.150	0.308	0.978	3.844	16.620		
	64	0.117	0.146	0.183	0.352	1.055	4.084	18.820	
CR	1	0.107	0.370	1.533	6.847				
	4	0.075	0.136	0.402	1.574	6.881			
	16	0.085	0.097	0.160	0.441	1.669	7.159		
	64	0.085	0.091	0.123	0.176	0.438	1.649	7.209	
RT_0	1	0.082	0.234	0.867	3.603				
	4	0.062	0.100	0.258	0.903	3.649			
	16	0.068	0.079	0.130	0.318	1.136	4.462		
	64	0.088	0.092	0.113	0.219	0.379	1.191	4.662	
RT_1	1	0.198	0.629	2.404	9.855				
	4	0.121	0.232	0.674	2.467	9.960			
	16	0.121	0.147	0.263	0.714	2.545	10.150		
	64	0.146	0.163	0.195	0.326	0.766	2.579	10.170	
BDM	1	0.144	0.554	2.248	9.371				
	4	0.084	0.184	0.608	2.321	9.484			
	16	0.074	0.115	0.220	0.645	2.397	9.651		
	64	0.103	0.106	0.175	0.256	0.682	2.424	9.664	

Kapitel 6

Zusammenfassung, Diskussion und Ausblick

Im Vergleich zu bisherigen Ansätzen bietet das DDD-Konzept eine neuartige Kombination aus folgenden Merkmalen:

- Verteilung von beliebig komplexen Datentopologien (allgemeine Graphen)
- Unterstützung von dynamischen Veränderungen der Datenbasis
- Dimensionsunabhängigkeit, Unterstützung beliebiger Datenstrukturen
- Globale Kopplungen auf lokalen Adreßräumen mittels lokaler Referenzen
- Frei definierbares Konsistenzmodell
- Effiziente Programme, weitreichende Portabilität
- Entwicklung gut wartbarer, in die sequentielle Version integrierter Parallelprogramme

Das Konzept deckt einen weiten Bereich von Applikationen ab, wobei vor allem viele moderne numerische Verfahren eingeschlossen sind. Im abschließenden Kapitel sollen nun nach einer kurzen Zusammenfassung die Entwurfsziele validiert und noch offene Aspekte diskutiert werden. In einem Ausblick wird auf Arbeiten und Zielsetzungen verwiesen, die über das hier beschriebene DDD-Konzept hinausgehen.

6.1 Zusammenfassung

Ein Konzept zur Parallelisierung von Verfahren auf graphbasierten, dynamischen Datenstrukturen wurde vorgestellt. Als Motivation diente dabei die Einsicht, daß bestehende parallele Programmiermodelle auf *distributed-memory*-Architekturen entweder nur ungenügende Abstraktionen zur

Verfügung stellen (z.B. *message-passing*), welche die Realisierung solcher Verfahren schwierig oder gar unmöglich machen, oder im Gegenteil zwar komfortable Möglichkeiten der parallelen Programmentwicklung bieten, aber stets zu ineffizienten Implementierungen (in bezug auf Rechenzeit und Ressourcenbedarf) führen (Kap. 1). Aus dieser Einsicht und vorangegangenen Erfahrungen mit der Parallelisierung spezieller Applikationen mit obigen Eigenschaften (vgl. [10], [17]) wurde ein Parallelisierungsmodell abgeleitet, das einen Kompromiß zwischen der nötigen Abstraktionsebene und der zu erreichenden Effizienz darstellt [22].

Dem vorgestellten Konzept *Dynamic Distributed Data* (DDD) wurde zunächst ein formales Modell von verteilten Datenbasen zugrundegelegt, das auf allgemeinen Graphen basiert (Kap. 2). Dabei wurde ein verteilter Graph, der aus den Prozessoren zugeordneten lokalen Graphen zusammengefügt wird, zu einem globalen Graph in Beziehung gesetzt, der die vom Algorithmus benötigten Datenstrukturen repräsentiert. In diesem Modell wurden viele grundlegende Begriffe dieser Arbeit (z.B. lokales und verteiltes Objekt, Referenzrelation, globale bzw. lokale Referenzen, Objekttypen) vorgestellt und ihre Beziehungen herausgearbeitet. Aus einer Vielzahl von bestehenden Datenkonsistenzmodellen wurde für DDD das Prinzip der *lean consistency* abgeleitet, das die Kontrolle der Konsistenz von Daten verteilter Objekte dem Anwendungsprogramm überträgt, welches Synchronisationen je nach den exakten Bedürfnissen des auszuführenden Algorithmus anstoßen kann (Abschnitt 2.3.8).

Der Systemarchitektur von DDD-basierten parallelen Anwendungen liegt ein Schichtenmodell zugrunde (Abschnitt 2.1), den Kern stellt dabei die DDD-Programmbibliothek dar. Die Lastbalancierungsstrategie zur Partitionierung der Datenbasis und zur Abbildung der Partitionen auf Prozessoren ist nicht in der DDD-Bibliothek enthalten; diese Entwurfsentscheidung wurde wegen der Anwendungsabhängigkeit der Problematik und der Möglichkeit der Integration von DDD-Applikationen mit völlig neuentwickelten Lastbalancierungsverfahren getroffen (Abschnitt 2.2).

Die DDD-Programmbibliothek bietet Funktionen aus vier Teilbereichen an, die auf dem verteilten Graph arbeiten, der durch das formale Modell spezifiziert wurde. Dies sind Verwaltungsfunktionen zur Definition von verteilten Datentypen (*Typemanager*, Abschnitt 2.4.1) und zur Kontrolle lokaler Objekte (*ObjectManager*, Abschnitt 2.4.1), Interfacefunktionen zur Kommunikation auf statischen verteilten Datenbasen (Abschnitt 2.4.2), Transferfunktionen zur dynamischen Veränderung der Datentopologie (Abschnitt 2.4.3) sowie Identifikationsfunktionen zur Erzeugung von verteilten aus lokalen Objekten (Abschnitt 2.4.4). Benutzerdefinierte Handler-Funktionen kapseln in abstrakter Form anwendungsinterne Informationen; diese werden von den DDD-Bibliotheksfunktionen bei Bedarf aufgerufen (Abschnitt 2.4.5).

In Kap. 3 wurde detailliert auf die Implementierung der DDD-Bibliothek eingegangen, da die Umsetzung des formalen Graphmodells und der an der Programmierschnittstelle definierten Funktionen den Löwenanteil dieser Arbeit ausmacht. Zahlreiche Datenstrukturen und effiziente Algorithmen mußten entwickelt werden; erst deren Zusammenspiel garantiert die erforderliche Effizienz des implementierten Programmiermodells. Im wesentlichen findet sich die Aufteilung der DDD-Programmierschnittstelle in vier Funktionsbereiche als Teilmodule in der Implementierung wieder, wobei auf zusätzliche Hilfsmodule zurückgegriffen wird (z.B. Anhang A), um dem entstandenen Softwareprodukt eine wartbare und dokumentierbare hierarchische Struktur zu verleihen.

Je ein Kapitel wurde schließlich der Bestimmung von Leistungsmerkmalen der Programmbibliothek (Kap. 4) und der Beschreibung von verschiedenen DDD-basierten Anwendungen gewidmet

(Kap. 5). Der Schwerpunkt lag dabei auf adaptiven numerischen Verfahren auf unstrukturierten Gittern, da diese einerseits die Entwicklung von DDD überhaupt motivierten und andererseits typische Beispiele von graphbasierten, dynamischen Anwendungen darstellen.

6.2 Diskussion

Zur Validierung der Ergebnisse dieser Arbeit sollen nun die Entwurfsziele, die in Abschnitt 1.5 aus einer allgemeinen Betrachtung der Problematik abgeleitet wurden, diskutiert und mit den erzielten Ergebnissen verglichen werden.

- Die Parallelisierung vorhandener sequentieller Programme wird durch die Verwendung des DDD-Programmiermodells erheblich vereinfacht. War es bisher üblich, mehrjährige europäische oder nationale Projekte (z.B. EUROPORT-ESPRIT-Projekt [50]) bzw. komplette Dissertationen (z.B. [101]) für die Parallelisierung *eines* Anwendungscodes anzusetzen, so reichen auf DDD-Basis nun einige Mannmonate, um ein entsprechend komplexes Softwareprodukt auf Parallelrechnerplattformen zu portieren. Das in der DDD-Bibliothek kumulierte Knowhow kann modular und einfach wiederverwendet werden, die Ersparnis an Zeitaufwand und anderen Ressourcen wird somit für inhaltliche Details frei.

- Durch den Einsatz des Schichtenmodells DDD-basierter Applikationen (Abb. 2.1 auf S. 34) kann der sequentielle Anwendungsteil größtenteils unabhängig von der Parallelisierung weiterentwickelt werden, die Verzahnung der sequentiellen mit den parallelen Programmteilen findet nur an der Schnittstelle Anwendung/DDD statt. Sofern das Anwendungsprogramm ebenfalls nach hierarchischen Gesichtspunkten gegliedert ist (wie im Beispiel UG, Abschnitt 5.3), kann die Algorithmenentwicklung auf einer komplett von der parallelen Implementierung getrennten Ebene stattfinden (vgl. Abschnitt 5.3.4).

- Andererseits bietet das DDD-Modell eine sichere Grundlage, auf der neue parallele Programme entwickelt werden können, ohne z.B. konkrete *message-passing*-Funktionalität zu benutzen (z.B. Abschnitt 5). Nach einigen Zyklen im Entwurf der Programmierschnittstelle besitzt das DDD-Modell nunmehr die nötige Flexibilität, um parallele Programmentwicklungen aus verschiedensten Bereichen zu unterstützen.

- Für die Portierung von DDD-basierten Anwendungsprogrammen auf die verschiedenen Parallelrechnerplattformen hat sich erfahrungsgemäß eher das Anwendungsprogramm selbst als die DDD-Bibliothek als Hindernis erwiesen. Durch die Verwendung der *message-passing*-Zwischenschicht PPIF (Anhang A.1) und den konsequenten Einsatz von ANSI C ist die DDD-Bibliothek als solche auf jeder neuen Plattform sofort lauffähig, nachdem die verfügbare Compiler- und Betriebssystemumgebung geklärt wurde. Alle Supercomputer, für die eine PPIF-Implementierung verfügbar ist, lassen sich unmittelbar für DDD-Applikationen einsetzen.

- Die erzielbare Effizienz von Anwendungen auf DDD-Basis bleibt kaum hinter der von Programmen zurück, die aufwendig von Hand parallelisiert wurden (siehe auch 4.6). Als

hauptsächlicher Speicheroverhead ist die Einführung einer DDD_HEADER-Struktur pro lokalem DDD-Objekt zu nennen; ein Teil dieses Mehraufwands wäre sicher auch für die weit aufwendigere von Hand durchgeführte Parallelisierung nötig, ein anderer Teil läßt sich nachträglich durch die Ersetzung von DDD-Objekten durch Datenobjekte vermeiden, wobei diese Ersetzung am bereits laufenden Programm durchgeführt werden kann (s. Abschnitte 5.1.2, 4.6).

Der bei DDD-Interfacekommunikationen auftretende Mehraufwand für das Umkopieren von Daten ließe sich für unstrukturierte Anwendungen auch bei manuellem Vorgehen nicht vermeiden (Abschnitt 4.3.1). Alle in den Modulen Transfer und Identifikation vorkommenden Konsistenzberechnungen und Kommunikationsvorgänge sind unvermeidbar und stehen in realistischer Relation zur geforderten Funktionalität. Hier liegt die Verantwortung für den vernünftigen Einsatz von dynamischen Datentopologieänderungen ohnehin beim Anwendungsprogramm.

- Ausführlich wurde auf die durch DDD eingeführten Abstraktionen für verteilte Graphen und Operationen auf diesen Graphen eingegangen. In der Praxis hat sich gezeigt, daß diese Abstraktionsebenen sinnvolle Schnittstellen zwischen Applikation und parallelem Programmiermodell bereitstellen, ohne zu Effizienzverlusten beim Einsatz der parallelen Programme zu führen. Die Graph-Abstraktion erlaubt dem Entwickler eine Denkweise, die sich nicht mit Details z.B. der Prozessorsynchronisation oder Kommunikation aufhält, sondern die gewohnten Datentypen durch die Beziehung von lokalen und verteilten Objekten automatisch auf den Parallelrechner übertragt.

- Die Definition und Benutzung von Interfaces erlaubt es dem Entwickler einer parallelen Anwendung, gezielt die Konsistenz der verteilten Datenbasis herbeizuführen. Dadurch entsteht ein schlankes Konsistenzmodell, das einerseits keine Zeit mit dem Abgleichen von vielleicht ohnehin schon konsistenten Daten vergeudet, aber durch die Abstraktion *Interface* andererseits einfache Möglichkeiten zum Herstellen einer konsistenten Datenbasis bietet, sofern dies nötig ist. Andererseits ist es während der Entwicklungsphase möglich, an allen kritischen Stellen eines prototypischen Parallelprogramms die Konsistenz der Daten einfach herbeizuführen und erst in einem späteren Optimierungsschritt abzuschwächen.

Die in Abschnitt 1.5 aufgestellten Entwurfsziele konnten also durch das DDD-Modell durchaus erreicht werden. Aus dem Prototyp der DDD-Bibliothek wurde mittlerweile ein fertiges, komfortabel einsetzbares Softwareprodukt entwickelt, das in einer Reihe von Projekten zum Einsatz kommt. Dennoch oder gerade wegen des möglichst reibungslosen Einsatzes als ein Baustein innerhalb von größeren Softwareprodukten treten einige Aspekte zutage, bei denen durch den Einsatz von Entwicklungszeit Verbesserungen des Gesamtkonzepts möglich sein werden. Diese sollen nun kurz diskutiert werden, im abschließenden Ausblick werden einige Punkte noch einmal aufgegriffen.

Die Implementierung der DDD-Bibliothek selbst wurde in ANSI C durchgeführt, da diese Sprache auf sämtlichen Rechnerplattformen verfügbar ist. Die Codestruktur ist streng modular und hierarchisch; dies ist unabdingbar, wenn man die Komplexität der zu implementierenden Funktionalität bedenkt. Gerade die wesentlichen Eigenschaften einer objektorientierten Sprache (Klassenbildung, Vererbung, Polymorphie, Inlining) hätten jedoch während der Entwicklungsphase wesentlich zur

Vereinfachung des Implementierungsprozesses beitragen können [152]. Auch in der jetzigen, produktionsreifen Version der DDD-Bibliothek hätte der Einsatz z.B. von C++ klare Vorteile für Wartung, nachträgliche Effizienzverbesserungen und Funktionserweiterungen der DDD-Bibliothek. Im Bereich des Supercomputing hat die objektorientierte Softwareentwicklung mittlerweile an Stellenwert zugenommen, die Existenz von entsprechenden Compilerumgebungen und viele Arbeiten belegen dies (z.B. [7, 33, 45, 104, 120]). Auch im Bereich der numerischen Applikationen, die ja eine Hauptanwendungsklasse für die DDD-Parallelisierung darstellen, sind objektorientierte Techniken nach anfänglichen Diskussionen [3] über Effizienz (z.B. gegenüber Fortran 77) oder Compilerverfügbarkeit mittlerweile allgemein anerkannt [7, 9, 132].

Der Einsatz des DDD-Modells in konkreten Projekten hat gezeigt, daß ein paralleler Prototyp mit Grundfunktionalität zufriedenstellend schnell erzeugt werden kann. Bis ein gesamtes Softwareprodukt parallelisiert ist, sind jedoch viele Detailprobleme zu lösen, so daß tatsächlich ein Arbeitsaufwand von einigen Mannmonaten bis hin zu wenigen Mannjahren verbleibt, trotz des Einsatzes der DDD-Bibliothek. Die Analyse der Vorgehensweise bei der DDD-Parallelisierung aus Abschnitt 5.1 zeigt, daß gerade in den Arbeitsschritten

- *Analyse der Datenbasis* (Abschnitt 5.1.2),
- *Entwurf der verteilten Datenbasis* (Abschnitt 5.1.3) und
- *Lasttransfer* (Abschnitt 5.1.6)

viel Schnittstellencode entworfen werden muß, der sich in allen Parallelisierungsprojekten ähnelt. An dieser Stelle ist es möglich, Entwicklungszeit einzusparen (vgl. Ausblick, Abschnitt 6.3).

Ein weiterer kritischer Punkt ist die Fehlersuche in parallelen Programmen. Dazu wurde in DDD mit folgenden Ansätzen bereits eine solide Basis geschaffen:

- Soweit möglich, werden die Parameter von DDD-Bibliotheksfunktionen zur Laufzeit auf Plausibilität kontrolliert; gegebenenfalls wird das Programm mit einer ausführlichen, dokumentierten Fehlermeldung abgebrochen. Dies ermöglicht die Kontrolle von Fehlern durch Fehlverwendungen der DDD-Funktionen.
- Ein interaktives Werkzeug zur Fehlersuche wurde implementiert, das vom parallelen Anwendungsprogramm aus aufgerufen werden kann und auf Benutzerbefehle mit der Ausgabe von Detailinformationen aus dem Zustand der DDD-Bibliothek reagiert. Dieser interaktive *Debugger* wird in die Schnittstelle Anwendung/DDD integriert; dadurch können ebenfalls Informationen über Anwendungsobjekte abgefragt werden.
- Ein ausführlicher Algorithmus zur Konsistenzüberprüfung wurde in die DDD-Bibliothek integriert, der z.B. die Konsistenz aller verteilten Objekte oder die vom Benutzer definierten DDD-Interfaces überprüft. In der Testphase können so in das Anwendungsprogramm an kritischen Stellen Aufrufe dieses Überprüfungsprogramms eingebaut werden; stellt dieses Fehler fest, können diese mit Hilfe des interaktiven Debugging-Werkzeugs lokalisiert und behoben werden.

Mit obigen Hilfsmitteln wird die Fehlersuche in DDD-basierten Programmen erleichtert; es wäre jedoch denkbar, moderne Hilfsmittel z.B. grafische Debugger zu integrieren bzw. zu entwerfen. Beispielsweise stehen der DDD-Bibliothek zur Laufzeit alle Informationen über den verteilten Graph zur Verfügung; durch Einführung eines benutzerdefinierten Handlers zur Angabe der geometrischen Position jedes verteilten Objekts könnte der verteilte Graph geeignet visualisiert werden. Dies würde bei kritischen Programmfehlern dazu beitragen, Inkonsistenzen der verteilten Datenstruktur schnell zu lokalisieren.

6.3 Ausblick

6.3.1 Zusätzliche Programmierschnittstellen

Die DDD-Bibliothek ist einsatzfähig und wird in mehreren Projekten eingesetzt. Um den möglichen Benutzerkreis zu erweitern, wird derzeit zusätzlich zur bestehenden C-Anwendungsschnittstelle eine Fortran 77-Schnittstelle entworfen und implementiert, die den Begriff der verteilten Graphen auf die in Fortran 77 implementierbaren Datenstrukturen erlaubt. Erste Testbeispiele zeigen, daß trotz einiger Unterschiede im Programmierstil und Datenlayout die in DDD vereinigten Algorithmen gewinnbringend auch für F77-Programme eingesetzt werden können. Auf Basis der F77-Schnittstelle wird DDD zur Zeit in bereits laufenden bzw. beantragten ESPRIT-Projekten einer kritischen Prüfung unterzogen.

Darüberhinaus wird derzeit eine zusätzliche C++-Schnittstelle entworfen, welche die DDD-Funktionalität in Form einer Klassenhierarchie kapselt. Zwar war es bisher bereits möglich, DDD im Kontext von objektorientierten Programmen einzusetzen (siehe dazu Beispielanwendung CEQ in Abschnitt 5.2), doch mit einer echten C++-Fassade kann die DDD-Bibliothek wesentlicher einfacher in ein objektorientiertes Softwareprodukt eingebaut werden.

Beide Ansätze bedürfen ständiger Überprüfung an echten Anwendungsprogrammen; in beiden Fällen wird daher kein übereilter Schnittstellenentwurf durchgeführt, sondern Schritt für Schritt eine Entsprechung zur bestehenden ANSI C-Schnittstelle aufgebaut. Hinter den drei Schnittstellen steckt im übrigen dieselbe Implementierung, um Wartung und Weiterentwicklung der DDD-Programmbibliothek zu erleichtern.

6.3.2 Dokumentation und Fehlersuche

Die bestehende Dokumentation der DDD-Bibliothek wird ständig ausgebaut, um den Einstieg in die Benutzung des Parallelisierungskonzepts zu erleichtern. Neben dem bereits existierenden Referenzhandbuch [19] und den existierenden Publikationen soll ein Benutzerhandbuch erstellt werden, das anhand von begrenzten Beispielen die Konzepte und Funktionen von DDD erklärt.

Die im letzten Abschnitt angesprochenen Ideen zu Diagnosewerkzeugen, die den DDD-Anwender in seiner Parallelisierungsarbeit unterstützen können, werden weitergedacht und realisiert. Ein grafisches Analyseverfahren für Objekte verschiedener DDD-Typen und ihre Referenzen ist bereits in Planung. Die bestehenden Werkzeuge zur Ermittlung von Statistik- und Leistungswerten DDD-basierter Programme müssen ausgebaut und auf einen Produktionsstandard gebracht werden.

6.3.3 Automatisierung der Parallelisierungsarbeit

Die zukünftige Arbeit soll das DDD-Konzept in einen größeren Zusammenhang einbetten. Auf Basis der formalen, deklarativen Spezifikationen verteilter Datenstrukturen (Anhang C) soll der Programmcode für die Schnittstelle Anwendung/DDD automatisch generiert werden können.

Als Vorstufe werden aus geeigneten Datenstruktur-Spezifikationen konsistente, verteilte Datenbasen erzeugt (z.B. überlappende Gitter), auf denen komplexe Aufgaben formal definiert werden (z.B. Lasttransfer, der ein verteiltes Gitter auf ein anderes abbildet). Die Erzeugung z.B. eines konsistenten Überlappungsbereichs wird mit Methoden der Modellgenerierung [30] durchgeführt. Über Planungsverfahren bzw. Suchverfahren in diskreten Zustandsräumen (vgl. [137, Kap. 20]) kann eine Operationensequenz errechnet werden, die eine verteilte Datenbasis in eine andere überführt; die verfügbaren Operationen sind dabei z.B. alle Kommandos des DDD-Transfermoduls.

Da die beschriebenen Algorithmen eine weit schlechtere Komplexität als die zu parallelisierenden numerischen Verfahren aufweisen, kann obige Methodik nicht direkt eingesetzt werden, sondern muß als Grundlage zu einem weiteren Schritt dienen: zur Erzeugung von effizientem Programmcode, der die formal gestellte komplexe Aufgabe lösen kann. Indem die Datenbasen und die gesuchte komplexe Operation so aufgestellt werden, daß sie eine endliche Beschreibung aller möglicher Problemsituationen darstellen, kann ein Programm zur Lösung dieser speziellen Aufgabe auch alle gleichartigen Aufgaben bewältigen.

Bei den hier nur kurz beschriebenen Techniken wird das DDD-Konzept als Grundlage benutzt, um die Parallelisierungsarbeit weiter zu automatisieren. Die Methodik der deklarativen Spezifikation erlaubt es, die Lokalität der Datenzugriffe eines Verfahrens noch präziser zu formulieren und die eher technischen Aspekte zu automatisieren. Dies bringt mehrere Vorteile:

- Die Parallelisierung einer graphbasierten Anwendung reduziert sich im besten Fall auf die Auswahl von Eigenschaften des verteilten Graphen.
- Obwohl komplizierte Such- und Modellgenerierungsprozesse eingesetzt werden, können doch effiziente Programme erzeugt werden.
- Das Verfahren liefert mit dem Programm auch einen Beweis für dessen Fehlerfreiheit.
- Der Zeitaufwand zur Parallelisierung komplizierter Programme reduziert sich über das durch DDD erreichte Maß hinaus.
- Zusätzlich können obige Techniken zur automatischen Erzeugung von Werkzeugen zur Konsistenzüberprüfung bzw. im sequentiellen Bereich eingesetzt werden (z.B. Vektorisierung, Gitterverfeinerung).

Die weitere, auf DDD aufbauende Arbeit hat also drei Stoßrichtungen: Erweiterung des Anwenderkreises, Verbesserung der Entwicklungsumgebung und Generalisierung des Parallelisierungskonzepts.

Anhang A

Grundlegende Kommunikationsschnittstellen

Im folgenden Anhang sollen die Kommunikationsschichten vorgestellt werden, die der DDD-Implementierung zugrundeliegen. Dabei werden die Schichten ausgehend vom gegebenen Parallelrechner-Programmiermodell in der Reihenfolge zunehmender Mächtigkeit beschrieben; Abb. A.1 zeigt eine Übersicht. Die erste Schnittstelle (PPIF) ist auch vom parallelisierten Anwendungsprogramm benutzbar, die folgenden Schnittstellen existieren zunächst nur innerhalb der DDD-Bibliothek und werden nicht nach außen exportiert.

A.1 PPIF: Schnittstelle zum Parallelrechner

Ein auf Basis der DDD-Bibliothek parallelisiertes Programm soll auf möglichst vielen (allen) Parallelrechnerkonfigurationen lauffähig sein. Deshalb wurden (bereits in [12]) Minimalanforderungen an einen Parallelrechner definiert und in Form einer Kommunikationsschnittstelle zusammengefaßt. Diese Schnittstelle (*parallel processing interface*, kurz PPIF) hat folgende Eigenschaften:

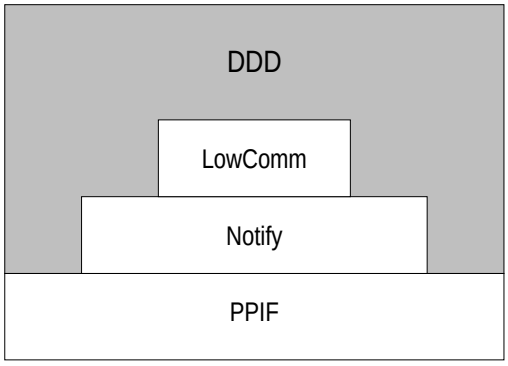


Abbildung A.1: *Übersicht der grundlegenden Kommunikationsschichten in DDD.*

- Bevor Nachrichten gesendet werden können, muß eine Verbindung zwischen den Kommunikationspartnern in Form eines Kanals (*virtual channel*) aufgebaut werden. Dies ermöglicht den Einsatz auf Rechnersystemen, deren Programmiermodell solche Kanäle voraussetzt.
- Es werden durchweg zweiseitige Kommunikationsprimitive eingesetzt; einseitige Kommunikation (z.B. *get/put*) stellt PPIF nicht zur Verfügung. Dies ermöglicht den Einsatz auf Rechnersystemen, die keine einseitige Kommunikation anbieten.
- Es wird blockierende und nichtblockierende Kommunikation angeboten. Für (ältere) Programmiermodelle, die keine nichtblockierende Kommunikation vorsehen, muß diese selbst implementiert werden (mit einem zusätzlichen Prozeß).
- Einige Kommunikationsstrukturen sind in PPIF vordefiniert und sofort benutzbar (Prozessorbaum, 2D/3D-Feld). Die Baumstruktur läßt sich u.a. zur bequemen Ausführung von globalen Reduktionsoperationen nutzen (Summe, Minimum, Maximum etc.).

Die vordefinierten Kommunikationsstrukturen der PPIF-Schnittstelle können im Hinblick auf Effizienz als *Angebot* an das benutzende Programm verstanden werden: falls für eine gegebene Rechnerarchitektur eine dieser Topologien (Baum bzw. Feld) besonders effizient auf die Hardware abgebildet werden kann (*mapping*), so sollte die PPIF-Implementierung für diese Hardware diesen Umstand berücksichtigen. Das aufrufende Programm kann dann davon ausgehen, daß bei Benutzung dieser vordefinierten Strukturen die Kommunikation möglichst effizient durchgeführt wird. Baut das aufrufende Programm an der PPIF-Schnittstelle eigene Strukturen aus Kommunikationskanälen auf, so können diese im allgemeinen nicht optimal auf das Kommunikationsnetz des Parallelrechners abgebildet werden.

PPIF-Implementierungen für bestimmte Programmiermodelle sind relativ einfach zu erstellen; derzeit existieren Implementierungen für alle gängigen Plattformen (MPI, PVM, NX, SHMEM, Parix). PPIF wird als Teil der DDD-Bibliothek eingesetzt, ist aber ebenfalls als Einzelpaket für Anwendungen ohne DDD verwendbar.

A.2 Notify: Aufbau einer n-zu-m Kommunikationsstruktur

Einige der Kommunikationsanforderungen der DDD-Bibliothek sind sendergetrieben, z.B. die Durchführung von **DDD_XferCopyObj()**-Kommandos im DDD-Transfermodul. Dabei muß jeder Prozessor lokal die Erzeugung von Objekten auf anderen Prozessoren initiieren können. Dieser einseitigen Kommunikation an der Schnittstelle zum Anwendungsprogramm steht das zugrundeliegende *message-passing*-Programmiermodell (PPIF) gegenüber, so daß eine Abbildung auf zweiseitige Kommunikationsvorgänge nötig wird. In der Praxis heißt dies, daß die jeweiligen Empfängerprozessoren vor dem eigentlichen Transfer informiert werden müssen, von welchem Prozessor sie Nachrichten welcher Größe erhalten werden.

Die zu dieser Information nötigen Vorabnachrichten haben die Form

$$(p_{\text{send}}, s_{\text{msg}}, p_{\text{recv}}),$$

wobei p_{send} der Sendeprozessor der späteren, eigentlichen Nachricht ist, s_{msg} die Größe der Nachricht und p_{recv} ihr Empfänger. Zu Beginn kennt jeder Prozessor p also die Nachrichten $\{(p_i, s_i, q_i) \mid p_i = p\}$, die von ihm gesendet werden sollen. Für den Transfer muß Prozessor p jedoch auch die Nachrichten $\{(p_i, s_i, q_i) \mid q_i = p\}$ kennen, um die nötigen Empfangsoperationen initiieren zu können.

Um zu verhindern, daß die Anzahl der dazu nötigen Vorabnachrichten im schlechtesten Fall quadratisch mit der Prozessoranzahl wächst, wurde ein Algorithmus entwickelt, der diese Informationen über eine baumartige Kommunikationstopologie verteilt, so wie sie bereits standardmäßig vom zugrundeliegenden *Parallel Processing Interface* bereitgestellt wird. In diesem sog. *Notify*-Algorithmus wird die Baumstruktur in zwei entgegengesetzten Wellen durchlaufen: in der ersten Welle (von den Blättern zur Wurzel) teilt jeder Prozessor seinem Vaterprozessor im Kommunikationsbaum mit, welche Nachrichten seinen Unterbaum verlassen müssen; in der darauffolgenden, entgegengesetzten Welle erhält jeder Prozessor vom Vaterknoten die Nachrichten, die für seinen Unterbaum bestimmt sind. Nachrichten von p nach q verbleiben also im kleinsten Teilbaum, der noch p und q enthält.

Nach dem Durchlaufen des *Notify*-Algorithmus erhält jeder Prozessor eine Liste, die den Sender und die Größe der im Transfer zu empfangenden Nachrichten enthält. Basierend auf dieser Liste können nun zunächst Kanäle zu den Kommunikationspartnern aufgebaut, der Speicher für die zu empfangenen Nachrichten bereitgestellt und schließlich die Empfangsfunktionen der asynchronen Kommunikation initiiert werden.

A.3 LowComm: Abstrakte Kommunikation im Prozessorgraph

Eine typische Anforderung der DDD-Bibliothek an die zugrundeliegenden Kommunikationsfunktionen ist das Versenden von kompliziert strukturierten Nachrichten an im Prozessorgraph benachbarte Prozessoren. Vor allem in der Implementierung des Transfermoduls wird diese Kommunikationsart benötigt: zur Ausführung der vielen Einzelkommandos, die ein Anwendungsprogramm während einer Transferkommunikation absetzt, sollen möglichst wenige tatsächliche Nachrichten generiert werden. Dazu ist es nötig, die zu sendenden Informationen nach ihrem Empfangsprozessor zusammenzufassen und diesem zunächst die Nachrichtengröße mitzuteilen, damit dieser einen Empfangspuffer bereitstellen kann. Letzteres erledigt der oben vorgestellte *Notify*-Algorithmus; die Zusammenfassung der Nachrichten, die Strukturierung der enthaltenen Informationen in Tabellenform und das transparente Senden/Empfangen von mehreren Nachrichten pro Prozessor wurde in der vorliegenden DDD-Implementierung erneut gekapselt; das sogenannte *LowComm*-Modul stellt diese Funktionalität bereit.

Bevor Nachrichten mit dem *LowComm*-Modul verschickt werden können, muß der allgemeine Aufbau einer Nachricht definiert werden. Dazu werden zur Laufzeit beliebig viele Nachrichtentypen (*MsgType*) definiert, von denen jeder eine Nachricht beschreibt, die aus einer Anzahl von Tabellen mit beliebig strukturierten Einträgen (sog. *MsgTable*) und/oder einzelnen ansonsten unstrukturierten Datenbereichen (sog. *MsgChunk*) besteht. Für die einzelnen Bereiche zu einem Nachrichtentyp werden nachrichtenabhängige Identifikatoren vergeben, über die diese Bereiche zugänglich sind. In

dieser Anmeldephase wird nur die Struktur der Nachricht festgelegt; die Größe der einzelnen Bereiche und ebenso die Größe der tatsächlichen Nachrichten werden erst beim Aufbau der Nachricht festgelegt.

Soll die tatsächliche Kommunikation anlaufen, werden zunächst alle zu sendenden Nachrichten mit ihrem jeweiligen Zielprozessor beim *LowComm*-Modul angemeldet. Sodann werden die Anzahl der Tabelleneinträge für alle Tabellen jeder Nachricht sowie die Größe der Datenbereiche festgelegt. Aus diesen Angaben errechnet das *LowComm*-Modul die Größen der erforderlichen Nachrichtenpuffer, legt den Speicher dafür an und informiert unter Verwendung des oben beschriebenen *Notify*-Algorithmus die Empfänger über die Größe der Nachrichten.

Nachdem das aufrufende Programm die leeren Nachrichtenpuffer (d.h. Tabellen und Datenbereiche) mit den eigentlichen Daten gefüllt hat, erledigt das *LowComm*-Modul das Versenden an die richtigen Empfänger. Über die bereits angesprochenen Identifikatoren können schließlich auf Empfängerseite die Daten aus den einzelnen Tabellen und Datenbereichen der empfangenen Nachrichten extrahiert und weiterverarbeitet werden.

Mit dem *LowComm*-Modul steht somit eine Schnittstelle zur Verfügung, die die restliche DDD-Implementierung von den Widrigkeiten eines reinen *message-passing*-Betriebs entlastet.

Anhang B

Spezifikation der Transfer-Funktionalität

Dieser Anhang soll die Wirkung der DDD-Transferkommandos **DDD_XferCopyObj()**, **DDD_XferDeleteObj()** und **DDD_PrioritySet()** und ihren Zusammenhang genau spezifizieren. Da die Wirkung jedes Transferkommandos auf genau *ein* verteiltes Objekt beschränkt bleibt, beziehen sich die unten angegebenen Regelsätze tatsächlich auf nur ein verteiltes Objekt, dieses kann jedoch selbstverständlich aus mehreren lokalen Objekten zusammengesetzt sein. Erst bei Einbeziehung von Referenzen zwischen Objekten und der Implementierung des Referenz-Transfer-Mechanismus kommt die Beziehung *zwischen* verteilten Objekten zum Tragen. Dies soll nicht Gegenstand dieses Anhangs sein.

Die folgende Spezifikation besteht aus zwei Teilen: zunächst werden die drei relevanten Kommandotypen und ihre Auswirkungen auf die verteilte Datenbasis einzeln spezifiziert, danach wird ihr Zusammenwirken festgelegt.

B.1 Spezifikation der einzelnen Transferkommandos

In diesem Abschnitt wird die Wirkung von mehreren (gegebenenfalls verteilt abgesetzten) gleichartigen Transferkommandos auf ein verteiltes Objekt spezifiziert.

B.1.1 Das Transferkommando XferCopyObj

Der folgende Regelsatz spezifiziert die Wirkung von mehreren **DDD_XferCopyObj()**-Kommandos auf dasselbe verteilte Objekt $\hat{o}$. Die Prozessoren P_A, P_B und P_C seien paarweise ungleich.

Regel C1: Besitzt P_A eine Kopie $o_1 \in \hat{o}$ mit Priorität p_1 und schickt P_B eine Kopie $o_2 \in \hat{o}$ mit Priorität p_2 und gleichzeitig eine Kopie $o_3 \in \hat{o}$ mit Priorität p_3 an P_A, dann wird bereits von P_B das Maximum von p_2 und p_3 ausgewählt. Der nicht ausgewählte Kopierbefehl wird verworfen.

Regel C2: Schickt P_B eine Kopie $o_2 \in \hat{o}$ mit Priorität p_2 an P_A und schickt P_C eine Kopie $o_3 \in \hat{o}$ mit Priorität p_3 an P_A, dann wird das Maximum der Prioritäten p_2 und p_3 ausgewählt. Der nicht ausgewählte Kopierbefehl wird verworfen. Bei $p_2 = p_3$ wird einer der beiden Kopierbefehle zufällig ausgewählt (abhängig von den Nachrichtenlaufzeiten im Kommunikationsnetz).

Regel C3: Besitzt P_A eine Kopie $o_1 \in \hat{o}$ mit Priorität p_1 und schickt P_B eine Kopie $o_2 \in \hat{o}$ mit Priorität p_2 an P_A, dann gibt es folgende Fälle:

$p_2 > p_1$**:** o_1 wird durch o_2 überschrieben, d.h. o_1 wird gelöscht, o_2 wird kreiert. Nullreferenzen aus o_2 werden jedoch durch vorhandene Referenzen aus o_1 ersetzt.

$p_2 = p_1$**:** Aktion wie bei Fall $p_2 > p_1$ (willkürlich gewählt).

$p_2 < p_1$**:** o_2 wird abgewiesen, da bereits o_1 vorhanden; Kopierbefehl wird also verworfen. Nullreferenzen aus o_1 werden jedoch gegebenenfalls durch vorhandene Referenzen aus o_2 ersetzt.

Regel C4: Besitzt P_A eine Kopie $o_1 \in \hat{o}$ mit Priorität p_1 und schickt eine Kopie $o_2 \in \hat{o}$ mit Priorität p_2 an sich selbst, so wirkt dieser Kopierbefehl wie ein PrioritySet-Befehl, d.h. die Priorität von o_1 wird auf p_2 gesetzt. *Achtung:* Die Spezifikation sieht hier kein Abweisen des Kommandos für $p_2 < p_1$ vor.

Durch Regel C1 werden bereits auf der Senderseite gleichartige Kommandos zusammengefaßt, zwei lokale Objekte mit gleicher globaler ID sind nicht zugelassen. Aus allen eingehenden Nachrichten auf dem empfangenden Prozessor P_A werden durch Regel C2 gleichartige Kommandos von verschiedenen Sendern gebündelt. Regel C3 entscheidet beim Vorhandensein lokaler Kopien, ob eingehende Kommandos berücksichtigt und damit diese Kopien überschrieben werden. Regel C4 berücksichtigt schließlich Transferkommandos eines Prozessors an sich selbst und deren Umsetzung als Prioritätsänderung.

Abb. B.1 zeigt das Zusammenspiel der Regeln. Die linke Seite der Grafik bezeichnet dabei den nicht-lokalen Teil (*remote commands*), die rechte Seite den lokalen Teil (jeweils aus der Sicht von Prozessor A). Die gestrichelte Linie in der Mitte deutet die Kommunikationskanäle an.

B.1.2 Das Transferkommando PrioritySet

Der folgende Regelsatz spezifiziert die Wirkung von mehreren **DDD_PrioritySet()**-Kommandos auf dasselbe verteilte Objekt $\hat{o}$. Da das **DDD_PrioritySet()**-Kommando laut Parameterliste keinen anderen Prozessor betrifft, genügt es, einen Prozessor P_A zu betrachten. Wegen seiner Auswirkung auf die Prozessoren, die ebenfalls Kopien $o \in \hat{o}$ besitzen, wird dieses Kommando trotzdem zur Gruppe der Transferkommandos gerechnet.

Regel P1: Besitzt P_A eine Kopie $o_1 \in \hat{o}$ mit Priorität p_1 und setzt er gleichzeitig zwei (oder mehr) **DDD_PrioritySet()**-Kommandos mit den Prioritäten p_2 und p_3 ab, so wird nur das Kommando mit der maximalen Priorität ausgeführt. *Achtung:* Priorität p_1 wird in die Maximumsbildung nicht einbezogen.

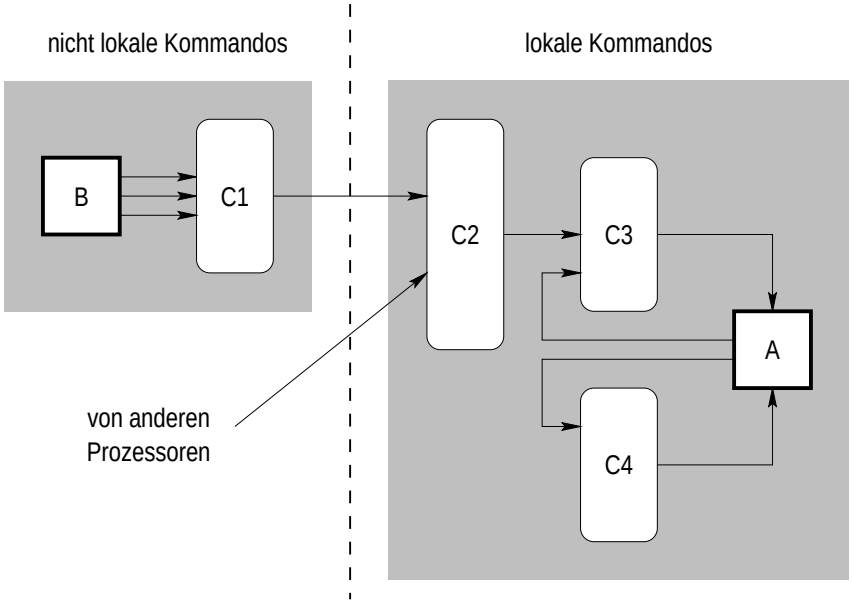


Abbildung B.1: *Übersicht der XferCopyObj-Regeln und ihrer Anwendung im Transfermodul.*

B.1.3 Das Transferkommando XferDeleteObj

Der folgende Regelsatz spezifiziert die Wirkung von mehreren **DDD_XferDeleteObj()**-Kommandos auf dasselbe verteilte Objekt $\hat{o}$. Da das **DDD_XferDeleteObj()**-Kommando laut Parameterliste keinen anderen Prozessor betrifft genügt es, einen Prozessor P_A zu betrachten. Wegen seiner Auswirkung auf die Prozessoren, die ebenfalls Kopien $o \in \hat{o}$ besitzen, wird dieses Kommando trotzdem zur Gruppe der Transferkommandos gerechnet.

Regel D1: Besitzt P_A eine Kopie $o_1 \in \hat{o}$ mit Priorität p_1 und setzt er für o_1 gleichzeitig zwei oder mehr **DDD_XferDeleteObj()**-Kommandos ab, so ist deren Wirkung äquivalent mit der Wirkung eines einzigen **DDD_XferDeleteObj()**-Kommandos.

B.2 Zusammenwirken der Transferkommandos

Dieser Abschnitt spezifiziert die Ausführungsreihenfolge von mehreren Transferkommandos verschiedenen Typs für dasselbe verteilte Objekt und damit deren Zusammenwirken. Die Kommandos werden in folgender Reihenfolge ausgeführt:

1. Zunächst werden alle **DDD_PrioritySet()**- bzw. alle nach Regel C4 lokalen **DDD_XferCopyObj()**-Kommandos bearbeitet.
2. Danach werden durch **DDD_XferDeleteObj()**-Kommandos betroffene lokale Objekte gelöscht.

3. Schließlich werden alle echten (d.h. nicht-lokalen gemäß Regel C4) **DDD_XferCopyObj()**-Kommandos bearbeitet.

Durch diese Abarbeitungsreihenfolge gehorcht das Transfermodul folgenden Regeln:

Regel M1: Werden **DDD_PrioritySet()** und **DDD_XferDeleteObj()** für dasselbe Objekt abgesetzt, so ist **DDD_PrioritySet()** wirkungslos.

Regel M2: Werden **DDD_PrioritySet()** und **DDD_XferCopyObj()** für dasselbe Objekt abgesetzt, so wird bereits die neue Priorität zur Entscheidung in Regel C3 herangezogen.

Regel M3: Werden **DDD_XferDeleteObj()** und **DDD_XferCopyObj()** für dasselbe Objekt abgesetzt, so wird es zunächst vernichtet, dann wieder etabliert. Regel C3 ist also außer Kraft, jedes eingehende Objekt (ungeachtet seiner Priorität) angenommen.

Regel M4: Werden **DDD_PrioritySet()**, **DDD_XferDeleteObj()** und **DDD_XferCopyObj()** für dasselbe Objekt abgesetzt, so ist nach Regel M1 das **DDD_PrioritySet()**-Kommando wirkungslos; danach wird nach Regel M3 jedes eingehende Objekt (ungeachtet seiner Priorität) angenommen.

Dabei werden bei mehreren konkurrierenden Kommandos gleichen Typs zunächst die Regeln C1-C4, P1 und D1 eingesetzt, für die verbleibenden (maximal drei) Kommandos verschiedenen Typs die Regeln M1-M4.

B.3 Nachbemerkung

Implizit wurde der obigen Spezifikation stets die Maximumoperation zugrundegelegt, falls aus einer Menge von Prioritäten eine ausgezeichnete Priorität ermittelt werden sollte (Regeln C1-C4 und P1). Es ist jedoch möglich, andere Operationen zu definieren, falls die Maximumoperation zur Implementierung des anwendungsorientierten Konsistenzmodells nicht ausreicht. Die Operation wird als dreistellige Relation in Form von einzelnen Beispielen für einen bestimmten DDD-Typ $t \in T$ definiert, wobei zunächst eine Ausgangseinstellung festgelegt werden kann.

Soll beispielsweise beim Vergleich zweier Prioritäten im allgemeinen das Minimum, bei den Prioritäten 5 und 3 jedoch die Priorität 11 entstehen, so wird durch eine einfache Befehlssequenz (links) eine (stets symmetrische) Operation definiert (rechts):

Default(t, MINIMUM);
Define (t, 3, 5, 11);

$\Rightarrow$

	3	5	11
3	3	11	3
5	11	5	5
11	3	5	11

Dabei ist zu beachten, daß die Operation assoziativ sein muß, da z.B. die Reihenfolge mehrerer eingehender Nachrichten nicht eindeutig ist (obiges Beispiel wäre also falsch!). Bei Definition einer korrekten Operation wird schließlich die gesamte Logik des Transfermoduls auf diese umgestellt.

Anhang C

Formale Spezifikationen verteilter Gitter

Das DDD-Modell beschreibt eine allgemeine Spezifikation verteilter Graphstrukturen. Die Kopplung zwischen den einzelnen lokalen Graphen bleibt dabei auf die Kopplung der verteilten Objekte beschränkt, aus denen der verteilte Graph besteht. Beschränkt man die Anwendung des Modells auf die Klasse der *gitterbasierten Verfahren* wie die in Kap. 5 beschriebenen Lösungsverfahren für Probleme partieller Differentialgleichungen, so geschieht die Kopplung der lokalen Teilgitter durch redundante Kopien von Gitterelementen im Überlappungsbereich an Prozessorgrenzen.

Dieser Anhang listet nun für typische Anwendungsfälle formale Spezifikationen auf, die zur Grundlage von Implementierungen verteilter Graphen und deren Überprüfung dienen können. Dabei werden die elementaren Operationen in Form der Funktionen **DDD_XferCopyObj()**, **DDD_XferDeleteObj()** und **DDD_PrioritySet()** vom DDD-Transfermodul konsistent zur Verfügung gestellt (vgl. Anhang B). Der Algorithmus, der diese Operationen aufruft und dadurch verteilte Gitter mit Überlappung erzeugt, muß jedoch abhängig von den Erfordernissen der Anwendung und somit von der Gitter-Spezifikation entwickelt werden (ein Beispiel gibt Abschnitt 5.2.4).

Jede Spezifikation ist ein Satz von Regeln in prädikatenlogischer Schreibweise. In der Überschrift erhält jede Spezifikation ein Kürzel, das den einzelnen Regeln als Präfix vorangestellt wird, um die komplette Regelsammlung eindeutig zu gestalten (1-OVP, 1-OVPE, MG). Die Schreibweise $master(e, p)$ bedeutet, daß eine lokale Kopie des Elements e auf Prozessor p mit Priorität *Master* existiert (analog für $ghost(e, p)$ und $copy(s, p)$).

Die deklarative Formulierung von Gittereigenschaften mittels prädikatenlogischer Formeln hat u.a. drei wesentliche Vorteile:

- Durch die Beschränkung auf wesentliche Relationen als Grundlage einer Spezifikation können andere Eigenschaften der spezifizierten Gitter sehr allgemein miterfaßt werden. Beispielsweise gelten alle unten aufgeführten Spezifikationen unabhängig von der Raumdimension des Gitters.
- Wie im Bereich der logischen Programmierung können die Regelsätze verschiedener deklarativer Spezifikationen einfach kombiniert werden, um kompliziertere Spezifikationen abzuleiten. Diese Eigenschaft erlaubt es, Hierarchien von Regelsätzen zu definieren, die nur über ihre Schnittstellen verknüpft sind. Ein einfaches Beispiel ist die 1-OVPE-Spezifikation (Abschnitt C.2).

- Durch die präzise Formulierung ist es möglich, diese Spezifikationen als Grundlage einer automatischen Erzeugung von konsistenten Gittern oder gar von parallelen, effizienten Programmcodes einzusetzen. Dies kann z.B. auf Basis von Methoden der Modellgenerierung [30] geschehen; unter Einsatz des automatischen Theorembeweisers und Werkzeugs zur Generierung von (minimalen) Modellen SATCHMO [1,29] wurden bereits vielversprechende Resultate erzielt [21] (siehe auch Abschnitt 6.3.3).

C.1 Einfache Überlappung von Elementen (1-OVP)

Diese Spezifikation ist zwar so vereinfacht, daß sie in realen Anwendungen nur selten in Reinform anwendbar ist. Sie drückt jedoch klar den Zusammenhang von Nachbarschaftsrelation, Prozessorzuordnung und Überlappung aus und ist damit die Grundlage für alle folgenden Spezifikationen. Der Name der Spezifikation 1-OVP steht für Überlappung (*OVerlaP*) von einer Elementreihe (*1*).

C.1.1 Informelle Beschreibung

Das Gitter besteht aus Elementen $e \in \mathsf{elem}$. Es wird vom Lastbalancierer auf Prozessoren $p \in \mathsf{proc}$ verteilt. Die Topologie des Gitters wird durch eine symmetrische Nachbarschaftsrelation $\mathrm{nb}(\cdot,\cdot)$ bestimmt. An den Grenzen zwischen den Partitionen soll eine einreihige Überlappung von Elementen je Prozessor entstehen. Elemente im Inneren einer Partition seien mit *Master*-Priorität ausgezeichnet, durch Überlappung entstandene Elemente mit *Ghost*-Priorität.

C.1.2 Formale Spezifikation

Die Relation $ghost_H(e,p)$ steht für *Ghost*-Objekte, die aufgrund horizontaler Nachbarschaftsrelationen entstehen.

1-OVP-1 (Lokalität): Alle Nachbarelemente eines *Master*-Elements werden lokal gespeichert, und zwar entweder mit *Ghost*- oder *Master*-Priorität.

$$\forall\, \mathsf{elem}\ e_1, e_2 : \forall\, \mathsf{proc}\ p : \quad nb(e_1,e_2) \wedge master(e_1,p) \;\rightarrow\; ghost_H(e_2,p) \vee master(e_2,p)$$

1-OVP-2 (Eindeutigkeit): Zu jedem Element gibt es höchstens eine *Master*-Kopie.

$$\forall\, \mathsf{elem}\ e : \forall\, \mathsf{proc}\ p_1, p_2 : \quad master(e,p_1) \wedge master(e,p_2) \;\rightarrow\; p_1 = p_2$$

1-OVP-3 (Speicher-Effizienz): Ein Prozessor kann nur dann eine *Ghost*-Kopie eines Elementes besitzen, wenn er keine *Master*-Kopie dieses Elementes besitzt.

$$\forall\, \mathsf{elem}\ e : \forall\, \mathsf{proc}\ p : \quad master(e,p) \;\rightarrow\; \neg ghost_H(e,p)$$

C.2 Einfache Überlappung mit Elementseiten (1-OVPE)

Diese Spezifikation ist komplizierter, aber auch realistischer als 1-OVP. Die dort verwendete Nachbarschaftsrelation nb ist hier nicht vorgegeben, sondern wird aus einer vorgegebenen Relation zwischen Elementen und Elementseiten abgeleitet. Der Name der Spezifikation 1-OVPE steht für Überlappung (*OVerlaP*) von einer Elementreihe (*1*) unter Berücksichtigung von Kanten (*with Edges*). 1-OVPE besteht aus den 1-OVP-Regeln und zwei zusätzlichen Regeln; dies ist das erste Beispiel zur additiven Komposition von Gitter-Spezifikationen.

C.2.1 Informelle Beschreibung

Das Gitter besteht aus Elementen $e \in \mathsf{elem}$ und Elementseiten $s \in \mathsf{side}$. Die Elemente werden vom Lastbalancierer auf Prozessoren $p \in \mathsf{proc}$ verteilt. Die Relation $e \overset{\mathsf{ref}}{\Leftrightarrow} s$ gibt an, welche Elementseiten durch das Element e referenziert werden. Es existiert keine direkte Nachbarschaftsrelation zwischen den Elementen, stattdessen wird die Gittertopologie rein durch die Relation $\overset{\mathsf{ref}}{\Leftrightarrow}$ bestimmt. An den Grenzen zwischen den Partitionen soll eine einreihige Überlappung von Elementen je Prozessor entstehen. Elemente im Inneren einer Partition seien mit *Master*-Priorität ausgezeichnet, durch Überlappung entstandene Elemente mit *Ghost*-Priorität. Es sollen nur die Elementseiten von *Master*-Elementen gespeichert werden; damit bestehen die Elementseiten auf Prozessorgrenzen aus genau zwei lokalen Objekten. Die Elementseiten besitzen stets die Priorität *Copy*.

C.2.2 Formale Spezifikation

1-OVPE-1 (Elementnachbarschaft): Die Nachbarschaftsrelation auf Elementen $\mathrm{nb}()$ wird von der Referenzrelation $\overset{\mathsf{ref}}{\Leftrightarrow}$ abgeleitet.

$$\forall\, \mathsf{elem}\, e_1, e_2 : \forall\, \mathsf{side}\, s : \quad e_1 \overset{\mathsf{ref}}{\Leftrightarrow} s \wedge e_2 \overset{\mathsf{ref}}{\Leftrightarrow} s \wedge e_1 \neq e_2 \;\rightarrow\; nb(e_1, e_2)$$

1-OVPE-2 (Lokale Elementseiten): Jeder Prozessor speichert alle Seiten seiner *Master*-Elemente.

$$\forall\, \mathsf{elem}\, e : \forall\, \mathsf{side}\, s : \forall\, \mathsf{proc}\, p : \quad master(e, p) \wedge e \overset{\mathsf{ref}}{\Leftrightarrow} s \;\rightarrow\; copy(s, p)$$

1-OVP: zusätzlich alle Regeln der 1-OVP-Spezifikation.

C.3 Vertikale Überlappung bei verteilten Mehrgittern (MG)

Falls durch Gitterverfeinerung mehrere Gitter unterschiedlicher Auflösung entstehen, gibt es vertikale Nachbarschaftsbeziehungen zwischen den aufeinanderfolgenden Gittern. Der Name der Spezifikation MG steht für Überlappung bei Mehrgittern (*MultiGrid*). Die Spezifikation MG definiert einen Lokalitätsbegriff für solche verteilten, hierarchischen Mehrgitterstrukturen. Auch diese Spezifikation kommt einzeln betrachtet in Anwendungen nicht vor, sie dient aber als Grundlage für verteilte Mehrgitterverfahren (z.B. Abschnitt 5.3.1).

C.3.1 Informelle Beschreibung

Das Gitter besteht aus Elementen $e \in \mathsf{elem}$. Es wird vom Lastbalancierer auf Prozessoren $p \in \mathsf{proc}$ verteilt. Die durch Gitterverfeinerung entstehenden Beziehungen zwischen den Elementen auf benachbarten Gitterebenen wird durch die Relation son bestimmt, wobei $son(e_1, e_2)$ genau dann gilt, wenn Element e_1 durch Teilung von Element e_2 entstanden ist (e_2 ist Vater von e_1). Der Vater jedes Elements soll prozessorlokal zugreifbar sein. Dadurch entsteht eine vertikale Überlappung der Elemente, wobei die durch Überlappung entstandenen Vaterelemente die Priorität *Ghost* besitzen sollen. Alle anderen Elemente sollen die Priorität *Master* erhalten.

C.3.2 Formale Spezifikation

Die Relation $ghost_V(e, p)$ steht für *Ghost*-Objekte, die aufgrund vertikaler Nachbarschaftsrelationen (d.h. son-Relationen) entstehen. Die Regeln zu MG entsprechen dem 1-OVP-Regelsatz, da in beiden Spezifikationen aus einer Topologierelation eine Überlappungseigenschaft abgeleitet wird.

MG-1 (Lokalität): Alle Vaterelemente eines *Master*-Elements werden lokal gespeichert, und zwar entweder mit *Ghost*- oder *Master*-Priorität.

$$\forall\, \mathsf{elem}\ e_1, e_2 : \forall\, \mathsf{proc}\ p : \quad son(e_1, e_2) \wedge master(e_1, p) \ \rightarrow\ ghost_V(e_2, p) \vee master(e_2, p)$$

MG-2 (Eindeutigkeit): Zu jedem Element gibt es höchstens eine *Master*-Kopie.

$$\forall\, \mathsf{elem}\ e : \forall\, \mathsf{proc}\ p_1, p_2 : \quad master(e, p_1) \wedge master(e, p_2) \ \rightarrow\ p_1 = p_2$$

MG-3 (Speicher-Effizienz): Ein Prozessor kann nur dann eine *Ghost*-Kopie eines Elementes besitzen, wenn er keine *Master*-Kopie dieses Elementes besitzt.

$$\forall\, \mathsf{elem}\ e : \forall\, \mathsf{proc}\ p : \quad master(e, p) \ \rightarrow\ \neg ghost_V(e, p)$$

C.4 Nachbemerkung

Die oben beschriebenen Regelsätze wurden ausgewählt, da sie in direkter Beziehung zu den DDD-Anwendungsbeispielen in Kap. 5 stehen. Viele weitere Spezifikationen sind denkbar: mehrfache Überlappung, komplizierte Datenstrukturen mit mehreren Objekttypen, Gitterverfeinerung und Gitterentfeinerung bzw. -vergröberung und andere. Eine Spezifikationsbibliothek befindet sich im Aufbau, die hierarchisch strukturiert ist und eine Vielzahl möglicher Gitteranwendungen abdeckt. Die jeweils ausgewählte Spezifikation beschreibt das Verhältnis der anwendungsbezogenen Datenstrukturen zum Modell des verteilten Graphen aus DDD und damit die Schnittstelle zwischen der DDD-Bibliothek und dem Anwendungsprogramm.

Anhang D

Automatisierter Funktionstest der Transferimplementierung

In Anhang B wurde spezifiziert, welche Wirkung jedes DDD-Transferkommando auf ein verteiltes Objekt haben soll und wie sich das Transfermodul bei mehreren Kommandos für *ein* verteiltes Objekt verhalten soll. Die Implementierung dieser Spezifikation stellt sich als schwierig heraus, zumal auf die Effizienz des Transfers besonderes Augenmerk gerichtet werden sollte. Dabei verteilen sich die Schwierigkeiten im wesentlichen auf die Teile *Referenzkonsistenz* und *Couplingkonsistenz*. Während die Erhaltung von Referenzen zwischen den Objekten und ihre Umrechnung beim Transfer ein überschaubares Problem darstellt, ist die Erhaltung der Konsistenz von Couplinglisten aufwendig und fehleranfällig: das Anwendungsprogramm kann völlig beliebig und auf die Prozessoren verteilt Transferkommandos absetzen, die das DDD-Transfermodul mit möglichst wenig Kommunikationsaufwand in eine verteilte Änderung der betroffenen Objekte und ihrer Couplinglisten umsetzen muß.

Da eine *Verifikation* der Transferimplementierung schon wegen des Programmumfangs nicht durchgeführt werden kann, müssen *Testverfahren* eingesetzt werden, um das korrekte Arbeiten gemäß der Spezifikation aus Anhang B sicherzustellen. Es hat sich gezeigt, daß die Kombinationen aus Transferkommandos, die in allen bisherigen DDD-Anwendungen benutzt wurden, nur einen Bruchteil aller möglichen Situationen ausmachen. Eine Auflistung einer ausreichenden Menge an Testfällen von Hand erscheint deshalb unmöglich.

Deshalb wurde ein zweistufiges Testverfahren entwickelt, das eine automatisierte Testdurchführung erlaubt: ein auf der zu testenden DDD-Implementierung basierendes, künstliches Anwendungsprogramm liest eine Reihe von Testsituationen ein, führt jeweils den Transfervorgang durch und kontrolliert danach die verteilten Couplinglisten und die Konsistenz des verteilten Objekts. Diese Testanwendung wird sodann als Werkzeug benutzt, um automatisch generierte Testreihen durchzuführen. Die Generierung der Testreihen obliegt einem Prolog-Programm, das für bestimmte Vorgaben (z.B. die Anzahl der beteiligten Prozessoren) umfassende Testdateien erzeugt (mit ca. 10^5 bis 10^7 Testfällen).

In diesem Anhang wird zunächst ein einfaches Beispiel für die Probleme verteilter Couplingkonsistenz beschrieben, um die Sensibilität für die Testproblematik zu erhöhen. Danach wird die DDD-basierte Testanwendung vorgestellt; schließlich wird der Algorithmus zur Erzeugung der Testfälle erklärt.

D.1 Beispiel zur Couplingkonsistenz

Ein wichtiges Ziel der Implementierung des Transfermoduls ist, den Aufwand an Kommunikation zu minimieren. Dies betrifft sowohl die Anzahl der nötigen Nachrichten (d.h. Minimierung der Aufsetzzeiten) als auch das Datenvolumen je Nachricht. Die Problematik des DDD-Transfers erlaubt es jedoch nicht, mit nur einer Kommunikationsphase (d.h. maximal eine Nachricht pro Prozessorpaar) auszukommen; das folgende (Gegen-)Beispiel belegt dies. Gleichzeitig läßt das Beispiel erkennen, wie eine unzureichende Transferimplementierung einen typischen Couplingkonsistenz-Fehler verursacht und wie dieser im verteilten System aufscheint. Achtung: die unten aufgeführten Kommunikationsvorgänge passieren innerhalb von DDD, auf Anwendungsebene ist nur das Anstoßen von zwei Transferkommandos sichtbar.

Betrachtet man also das verteilte Objekt $\hat{o}_1 = \{o_1, o_2\}$ mit $\mathsf{proc}(o_1) = p_1$ und $\mathsf{proc}(o_2) = p_2$, sowie die neu zu erzeugenden lokalen Objekte o_3 und o_4 mit $\mathsf{proc}(o_3) = p_3$ und $\mathsf{proc}(o_4) = p_4$ (siehe Abb. D.1), so muß die Kommunikation notwendigerweise in zwei Schritten erfolgen:

Ausgangspunkt: p_1 speichert o_1 und kennt o_3 mit Zielprozessor p_3, daher gilt auf p_1 die Annahme

$$p_1 : \hat{o}_1' = \{o_1, o_2, o_3\};$$

analog speichert p_2 das Objekt o_2 und kennt o_4 mit Zielprozessor p_4, daher gilt auf p_2 die Annahme

$$p_2 : \hat{o}_1' = \{o_1, o_2, o_4\}.$$

Erste Kommunikation: p_1 informiert p_2 von der bevorstehenden Erzeugung von o_3; analog informiert p_2 Prozessor p_1 von der bevorstehenden Erzeugung von o_4; danach gilt auf beiden Prozessoren die gleiche Annahme

$$p_1, p_2 : \hat{o}_1' = \{o_1, o_2, o_3, o_4\},$$

die auch tatsächlich korrekt ist.

Zweite Kommunikation: p_1 erzeugt das Objekt o_3 auf p_3 und teilt gleichzeitig die Annahme $p_1 : \hat{o}_1'$ mit; analog erzeugt p_2 das Objekt o_4 auf p_4 und teilt die Annahme $p_2 : \hat{o}_1'$ mit. Dadurch gilt nun auf p_3 dieselbe Annahme wie auf p_1 und analog auf p_4 dieselbe Annahme wie auf p_2.

Da nun durch die erste Kommunikation die Annahme über die Struktur des neuen verteilten Objekts $\hat{o}_1'$ abgeglichen wurde, haben alle vier beteiligten Prozessoren die gleiche Sicht. Ohne die erste Kommunikation würde gelten

$$\begin{aligned} p_1 &: \hat{o}_1' = \{o_1, o_2, o_3, o_4\}, \\ p_2 &: \hat{o}_1' = \{o_1, o_2, o_3, o_4\}, \\ p_3 &: \hat{o}_1' = \{o_1, o_2, o_3\} \text{ und} \\ p_4 &: \hat{o}_1' = \{o_1, o_2, o_4\}, \end{aligned}$$

damit wäre die verteilte Datenstruktur inkonsistent (obwohl p_1 und p_2 um die Inkonsistenz wüßten).

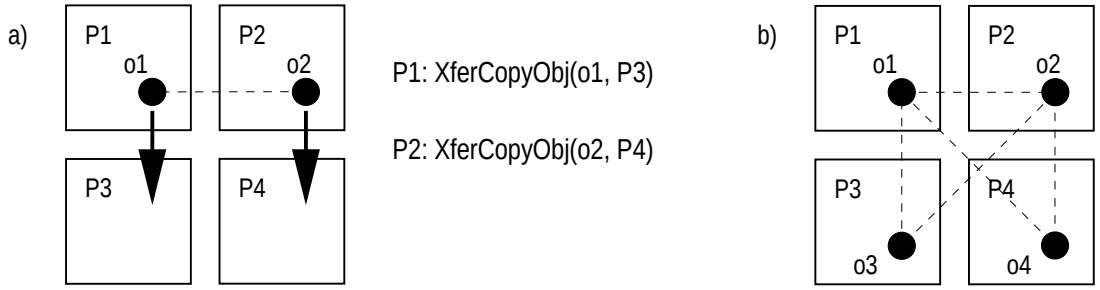


Abbildung D.1: *Beispiel zur fehlerhaften Konsistenz von Couplinginformationen. Zwei* **DDD_XferCopyObj()***-Kommandos beim Transfervorgang (a) und fehlerhafte Couplinginformationen nach dem Transfer (b). Die Prozessoren p_3 und p_4 wissen gegenseitig nichts von der Existenz des anderen lokalen Objekts.*

D.2 Die Testanwendung `dddic`

Eine Testumgebung für die Durchführung von Testreihen mit Situationen wie der im vorherigen Beispiel muß folgende Einzelschritte leisten:

1. Einlesen eines Testfalles von Datei
2. Aufbau der Testsituation auf den beteiligten Prozessoren
3. Versenden der Transferkommandolisten an die Prozessoren
4. Durchführung des Transfervorgangs mittels DDD
5. Lokale Konsistenzkontrolle durch alle Prozessoren
6. Einsammeln der Ergebnisse
7. Globale Konsistenzkontrolle
8. Statistik und (sofern nötig) Fehlerreport
9. Rücksetzen der Testumgebung zur Vorbereitung auf nächsten Testfall

Das DDD-basierte Anwendungsprogramm `dddic` (kurz für: *DDD Implementation Check*) leistet gerade dies; dabei werden die beteiligten Prozessoren nach einem *master/slave*-Schema aufgeteilt: ein Kontrollprozesser liest die Testfälle von Datei ein, verteilt diese an die *slave*-Prozessoren und sammelt nach der Testdurchführung die Ergebnisse ein.

Die Spezifikation eines Testfalles in der Eingabedatei sähe für das Beispiel aus dem vorigen Abschnitt folgendermaßen aus:

Datei	*Bedeutung*
`1 0 2 0`	lokales Objekt auf p_1 und p_2 mit Priorität 0
	Beginn der Kommandoliste
`1 c 3 0`	Kommandos von p_1: XferCopy an p_3 mit Priorität 0
`2 c 4 0`	Kommandos von p_2: XferCopy an p_4 mit Priorität 0
`0`	Ende der Kommandoliste

In der Kommandoliste können für jeden Prozessor eine beliebige Menge an Transferkommandos gegeben werden, die dieser auf sein lokales Objekt anwendet. Diese Kommandoliste wird allen *slave*-Prozessoren vom *master*-Prozessor geschickt, damit diese lokal entsprechende DDD-Kommandos aufrufen können. Die gesamte Eingabedatei wird einerseits im `ASCII`-Format angegeben, um eine schnelle manuelle Erstellung von Testfällen zu ermöglichen; andererseits wurde eine möglichst prägnante Darstellungsform gewählt, um die Dateigrößen für umfangreiche Testreihen handhabbar zu halten.

Sowohl die lokale als auch die globale Konsistenzkontrolle werden mit dem Standard-Algorithmus des in DDD integrierten Konsistenzchecks durchgeführt, so daß die Testanwendung `dddic` selbst unabhängig von der genauen Definition der DDD-Transferkommandos und der gewünschten Konsistenz bleibt. Der `dddic`-Fehlerreport für das vorherige Beispiel und eine entsprechend fehlerhafte Implementierung des DDD-Transfermoduls würde lauten:

```
Warning: obj 000400 on P1 has cpl from P3, but P4 hasn't!
Warning: obj 000400 on P1 has cpl from P4, but P3 hasn't!
Warning: obj 000400 on P2 has cpl from P3, but P4 hasn't!
Warning: obj 000400 on P2 has cpl from P4, but P3 hasn't!
```

Wie bereits durch Überlegung festgestellt, sind die Couplinglisten der Prozessoren p_3 und p_4 inkonsistent; die Meldungen darüber kommen allerdings von p_1 und p_2, die beide um die Inkonsistenz wissen.

Die Durchführung von Testreihen läßt sich mit Hilfe von `dddic` effizient gestalten: auf einem Parallelrechner vom Typ *Intel Paragon* lassen sich ca. 70 Testfälle pro Sekunde durchführen, auf einem Rechner vom Typ *Cray T3E* ca. 120 Testfälle pro Sekunde. Somit benötigt ein T3E-Rechner für eine umfangreiche Testreihe von 10^6 Fällen ungefähr zweieinhalb Stunden (bei einer Prozessoranzahl von vier bis sechs Prozessoren). Nutzt man zusätzlich noch die Trivialparallelität durch Aufteilen der gesamten Testmenge, läßt sich diese Zahl beliebig skalieren.

D.3 Automatisierte Testfallgenerierung

Zur automatisierten Testfallgenerierung wurde in der deklarativen Programmiersprache Prolog [137] ein Algorithmus (ca. 200 Programmzeilen) entwickelt, der aus den Eingabedaten

- Menge der zulässigen Prioritäten Π (mit $p = |\Pi|$),
- Menge der zulässigen Prozessoren P (mit $P = |P|$) und

- Menge von unzulässigen Transferkommando-Paaren

alle relevanten Testfälle erzeugt. Jeder Testfall besteht dabei aus einer *Anfangsbelegung* und einer *Kommandoliste* für jeden beteiligten Prozessor.

Die *Anfangsbelegung* ist ein verteiltes Objekt $\hat{o}$, das aus n lokalen Objekten $o_1 \ldots o_n$ (mit $n = |\hat{o}|$) mit den Prioritäten $\mathsf{prio}(o)$ ($o \in \hat{o}$) besteht. Alle Prozessoren $q \in P \setminus P_{\hat{o}}$ besitzen kein lokales Objekt von $\hat{o}$ und benötigen daher keine Kommandoliste. Alle anderen Prozessoren $r \in P_{\hat{o}} \subset P$ erhalten eine Liste von Transferkommandos, die sie für ihr lokales Objekt $o \in \hat{o}$ auszuführen haben. Für Paare von abstrakten Transferkommandos kann konfiguriert werden, ob sie vom gleichen Prozessor auf das gleiche lokale Objekt angewendet werden dürfen.

Um die Anzahl der möglichen Testfälle und damit den Umfang der Testreihe vor der Erzeugung abschätzen zu können, müssen die Anzahl von Anfangsbelegungen und die Anzahl von verschiedenen Kommandolisten kombinatorisch ermittelt werden. Legt man die Anzahl der beteiligten Prozessoren $|P_{\hat{o}}|$ und damit die Anzahl der lokalen Objekte $n = |\hat{o}|$ fest, so läßt sich die Anzahl von Anfangsbelegungen N_{start} berechnen durch den Binomialkoeffizient

$$N_{\text{start}} = \binom{p+n \Leftrightarrow 1}{p \Leftrightarrow 1} = \frac{(p+n \Leftrightarrow 1)!}{(p \Leftrightarrow 1)! \quad n!}$$

wobei p die Anzahl der erlaubten Prioritäten darstellt. Die Anzahl von relevanten Anfangsbelegungen N_{start} wurde bereits um Symmetrien bereinigt.

Zur Bestimmung der Anzahl von relevanten Kommandolisten für ein lokales Objekt $o \in \hat{o}$ muß zunächst die Anzahl der möglichen Transferkommandos N_c bestimmt werden. Dabei werden drei grundlegende Kommandotypen unterschieden:

- Für jedes lokale Objekt kann *genau ein* **DDD_XferDeleteObj()**-Kommando ausgeführt werden.
- Für jedes lokale Objekt können $p \Leftrightarrow 1$ verschiedene **DDD_PrioritySet()**-Kommando ausgeführt werden (das Objekt hat ja in der Anfangsbelegung bereits eine Priorität).
- Für jedes lokale Objekt o können $(P \Leftrightarrow 1)p$ verschiedene **DDD_XferCopyObj()**-Kommandos ausgeführt werden (an alle Prozessoren $q \in P \setminus \{\mathsf{proc}(o)\}$ kann eine Kopie beliebiger Priorität geschickt werden).

Für die Anzahl der möglichen Transferkommandos N_c für ein lokales Objekt gilt also:

$$N_c = 1 + (p \Leftrightarrow 1) + (P \Leftrightarrow 1)p = Pp$$

Wenn keine weiteren Einschränkungen an die Transferkommandos je Prozessor definiert werden, ist die Gesamtanzahl N_{cl} von Möglichkeiten zur Bildung von Kommandolisten

$$N_{\text{cl}} = 2^{N_c} = 2^{Pp},$$

da jedes Kommando entweder in der Liste enthalten ist oder nicht. Bei n Prozessoren mit lokalen Objekten ergibt dies eine Gesamtzahl von Testfällen N_{tests} gemäß

$$N_{\text{tests}} = N_{\text{start}}(N_{\text{cl}})^n = \binom{p+n \Leftrightarrow 1}{p \Leftrightarrow 1} 2^{nPp},$$

d.h. die Testreihengröße unterliegt sehr starkem Wachstum. Durch die Angabe unzulässiger Paare von Transferkommandos (Verringerung von N_c) bzw. Ausnützung von Symmetrien (z.B. Verkleinerung der Zielprozessormenge von **DDD_XferCopyObj()**-Kommandos) läßt sich dieses Wachstum einschränken; die entstehenden Testreihen haben eine Größe zwischen 10^5 und 10^7.

Das Programm zur Testfallgenerierung nutzt die Eigenschaften des Prolog-Interpreters und kann deshalb kompakt und übersichtlich formuliert werden. Über *backtracking* werden alle möglichen Anfangsbelegungen bestimmt. Zu jeder Anfangsbelegung wird für jedes lokale Objekt eine Kommandoliste generiert. Über erneutes *backtracking* werden alle möglichen Kommandolisten errechnet. Sobald alle lokalen Objekte einer Anfangsbelegung mit Kommandolisten ausgestattet sind, kann ein Testfall ausgegeben werden. Die Ausgabe kann entweder direkt oder über Datei dem Testprogramm `dddic` zugeführt werden, welches die Tests auf dem Parallelrechner effizient durchführt. Auf weitere (Implementierungs-)Details soll an dieser Stelle nicht eingegangen werden.

Anhang E

Konzepte für verteilte, dynamische Graphen bzw. Gitter im Vergleich

Die Problematik der Implementierung verteilter, dynamischer Graphstrukturen auf Rechnern mit verteiltem Speicher (*distributed memory architectures*) war und ist Gegenstand einer Vielzahl von Forschungs- und Entwicklungsarbeiten. Das in dieser Arbeit vorgeschlagene DDD-Konzept bietet die Möglichkeit, beliebige Graphstrukturen mit beliebiger topologischer Veränderung der Datenbasis zur Laufzeit effizient zu verwalten und gleichzeitig gut handhabbaren, portablen Programmcode zu erhalten.

Die folgende Tabelle stellt das DDD-Konzept in einem einheitlichen Schema anderen, vergleichbaren Ansätzen gegenüber. Für Bewertung und Vergleich der verschiedenen Konzepte sei auf Abschnitt 1.6 (Seite 21) verwiesen.

In der Tabelle werden folgende Abkürzungen verwendet:

AP	Anwendungsprogramm
CFD	Computational Fluid Dynamics
Dim	Dimension(en)
FD	Finite-Differenzen-Verfahren
FE	Finite-Elemente-Verfahren
FV	Finite-Volumen-Verfahren
KS	Kommunikationsschicht
MG	Mehrgitter-Verfahren
ML	Multilevel-Verfahren
OO	objekt-orientiert
RSB	Recursive Spectral Bisection [134]
VuM	Vektoren und Matrizen
WsC	Workstation-Cluster

Die Tabelleneinträge sind alphabetisch nach dem Akronym des jeweiligen Konzepts sortiert; die Felder jedes Tabelleneintrags haben folgende Bedeutungen:

Bezeichnung			Institution
Modell. Art der Abstraktion			
dynamisch/statisch Gitterart		Datentypen Zielsprache	MP-Modell Plattformen
Funktionalität			
S&K	Synchronisation Konsistenzerhaltung/Kommunikation		
LB&LT	Lastbalancierung Lasttransfer		
Applikationsbereich			
Literatur			Internet-Referenz (Homepage im WWW)

A++/P++			Univ. of Colorado & Los Alamos National Lab.
SPMD. verteilte Datenfelder, *array-extensions*, VSG (*virtual shared grids*)			
dynamisch (gitterweise) blockstrukturierte Gitter		integer, double C++	PVM, MPI WsC
Felder wie in F90, dynamische Verteilung			
S&K	Synchronisation durch *array*-Befehle Kommunikation implizit		
LB&LT	automatische Verteilung in jeder Dim, optional AP		
Anwendungen auf strukturierten Gittern			
[98, 99]			www.c3.lanl.gov/∼dquinlan/A++P++.html

BlockComm			Argonne National Laboratory
SPMD. verteilte Datenfelder			
statisch strukturierte Gitter		F77-Datentypen F77	Chameleon [61] Cray, IBM, Intel u.a., WsC
statische Verteilung und Datenaustausch			
S&K	Synchronisation durch AP Kommunikation in Überlappungsbereich		
LB&LT	blockweise in jeder Dim		
Anwendungen auf strukturierten Gittern			
[60]			–

CHAOS (⊃ PARTI)			University of Maryland
SPMD. verteilte unstrukturierte Gitter, globale Indizes			
dynamisch unstrukturierte Gitter		C-/F77-Datentypen C, F77	PVM, MPI, NX u.a. Intel, IBM SP1, CM-5
wie PARTI, dazu Umverteilung von Datenfeldern			
S&K	wie PARTI wie PARTI		
LB&LT	dynamisch (z.B. RSB), AP-gesteuert automatische Anpassung der *translation table*		
adaptive unstrukturierte Anwendungen			
[107, 119, 133]			www.cs.umd.edu/projects/hpsl/chaos.html

CHAOS++ (⊃ CHAOS)			University of Maryland
SPMD. verteilte Datenstrukturen und globale Pointer			
dynamisch allgemeine Graphen		C++ Klassen und Typen C++	PVM, MPI, NX u.a. Intel, IBM, CM-5
wie CHAOS, dazu Verwaltung hierarchischer Objekte mit Pointern			
S&K	expliziter Update von *ghost*-Objekten durch AP Datenaustausch zwischen Objektkopien		
LB&LT	beliebig pack/unpack sind AP-Funktionen		
OO-Anwendungen mit dynamischen Datenstrukturen			
[34, 35]			nhse.cs.rice.edu/CRPC/newsletters/sum95/wip.chaos.html

CLIC (auch: GMD Communications Library)			GMD, St. Augustin
SPMD. überlappende Gitterblöcke			
dynamisch auf Blockebene blockstrukturierte Gitter		F77-Datentypen F77	PARMACS 6.0 [69] Cray, IBM, Intel u.a., WsC
dynamische, verteilte Gittererzeugung			
S&K	explizite Synchronisation Kommunikation an Blockgrenzen		
LB&LT	automatisch, blockweise		
blockstrukturierte Anwendungen			
[122]			www.gmd.de/SCAI/num/clic/clic.html

Canopy			Fermi National Accelerator Laboratory
SPMD. verteilte Gitter und ihre Relationen			
statisch strukturierte Gitter		C-/F77-Datentypen C, F77	eigene KS *CHIP* Cray T3D, Sequent
S&K	explizite Synchronisation Kommunikation *intra-grid* und *inter-grid*		
LB&LT	automatisch		
Anwendungen auf strukturierten Gittern			
[52]			slacvm.slac.stanford.edu:5080/FIND/FREEHEP/NAME/CANOPY/FULL

Concurrent Graph (oder: SCPlib)			California Institute of Technology
taskparallel. Graph aus Prozessen und Datenabhängigkeiten			
dynamisch auf Threadebene allgemeine Graphen		beliebig ?	eigene KS Cray, IBM, Intel u.a., WsC
Taskmigration/-split/-merge, Visualisierung			
S&K	globale Synchronisation durch *barriers* mittels *communication lists*		
LB&LT	dynamisch auf Prozeßebene, diffusiv Ein-/Auspacken durch AP		
CFD, Partikelmethoden, Monte Carlo			
[139, 145]			www.scp.caltech.edu

DAGH			University of Texas at Austin, TX
SPMD. verteilte Gitter			
dynamisch blockstrukturierte Gitter		Fortran Datentypen C++ (und F77/F90)	MPI Cray, IBM, WsC
Gitterumverteilung, -verfeinerung, Kommunikation, Aufruf von F77/F90-Subroutinen			
S&K	explizit durch AP Datenaustausch an inneren Rändern		
LB&LT	benutzerdefiniert transparent		
adaptive Mehrgitterverfahren (AMR)			
[113, 115]			www.ticam.utexas.edu/∼parashar/public_html/DAGH

DDD			Universität Stuttgart
SPMD. verteilte Objekte, lokale Referenzen, Interfaces			
dynamisch allgemeine Graphen		F77/C/C++ Typen/Klassen C, C++ und F77	eigene KS *PPIF* alle gängigen Plattformen
Objekttransfer und -identifikation, abstrakte Kommunikation			
S&K	Synchronisation über Interfaces (durch AP) Kommunikation auf Interfaces		
LB&LT	externe Bibliotheken abstraktes Verschicken von Objektmengen		
dynamische Anwendungen auf beliebigen Graphstrukturen			
[19, 20, 22]			www.ica3.uni-stuttgart.de/∼birken/ddd

DIME			California Institute of Technology
SPMD. lokale Gitterelemente und verteilte Knoten			
dynamisch Dreiecksgitter		C Datentypen C	Express OS NCube, Transputer, Intel
Datenaustausch, Gitterverfeinerung und Umverteilung			
S&K	Synchronisation durch AP abstrakte Kommunikation auf verteilten Knoten		
LB&LT	orthogonale RCB Ein-/Auspacken automatisch		
FE (für Navier-Stokes)			
[148, 150]			www.npac.syr.edu/copywrite/pcw/node232.html

Dome			Carnegie Mellon University
SPMD. transparente verteilte Vektoren und Matrizen			
dynamisch strukturierte Gitter		VuM von C++-Typen C++	PVM heterogene WsC
heterogene Multiuser-Umgebung, Checkpointing, Fehlertoleranz			
S&K	explizite Reduktionsoperationen		
LB&LT	transparente Umverteilung bei Lastveränderung		
Anwendungen auf strukturierten Gittern			
[4]			www.cs.cmu.edu/∼Dome

Global Arrays (GA)			Pacific Northwest Laboratory
MIMD. verteilte Datenfelder			
dyn. *array*-Verwaltung strukturierte Gitter		integer, double C, F77	*interrupt-driven*, TCGMSG Intel, IBM, KSR, WsC
direkter, asynchroner Zugriff auf globale Felder, asynchrones Gather/Scatter			
S&K	konsistente *update*-Operation, ansonsten durch AP *accumulate*-Operation		
LB&LT	blockweise Verteilung, höchstens ein Block pro Prozessor		
Computer-Chemie			
[110]			www.emsl.pnl.gov:2080/docs/global/ga.html

GRIDS			Universität Stuttgart
SPMD. beliebige Topologien, skriptgesteuert			
statisch unstrukturierte Gitter		F77 Datentypen F77	eigene KS *SPPL* IBM, Intel Paragon
verteilte unstrukturierte Gitter, AP-Framework			
S&K	explizite Synchronisation transparenter Update		
LB&LT	automatisch		
gitterbasierte Anwendungen (z.B. CFD)			
[56–58]			www.informatik.uni-stuttgart.de/ipvr/as/projekte/grids/grids.html

LOCO			K. U. Leuven, Belgien
SPMD. grobgranulare *Units* im Nachbarschaftsgraph, Programm-Phasen			
dynamisch beliebig		C Typen, Strukturen C	PVM, NX Intel, WsC
Datenaustausch zwischen, Lastbalancierung auf *Units*			
S&K	Konsistmachen der *Unit*-Daten zwischen den Berechnungsphasen AP stellt Ein- und Auspackroutinen		
LB&LT	RCB, RIB, evol. Verfahren, u.a. Ein-/Auspacken von *Units* durch Anwendungsprogramm		
adaptive numerische Verfahren, auch MG			
[84–86]			–

LPARX (vorher: GenMP [5])			University of California, San Diego
SPMD. Zuordnung von dynamischen Feldern zu Prozessoren			
dyn. *array*-Verwaltung blockstrukturierte Gitter	C++ Typen/Klassen C, C++ und F77		eigene KS *MP++* PVM, u.a.
Domain Calculus: Erzeugung von Gitterüberlappungen			
S&K	Synchronisation durch AP Kommunikation im Überlappungsbereich		
LB&LT	Standardverfahren eingebaut oder AP Ein- und Auspacken durch AP		
ML, adaptive FD, Partikelmethoden			
[87, 88]			alchemy.ucsd.edu/skohn/lparx.html

Nearest Neighbor Tool (NNT)			Forecast Systems Laboratory
SPMD. verteilte Datenfelder			
statisch strukturierte Gitter	F77-Datentypen F77		PCL (Forecast Systems Lab) Intel u.a., WsC
statische Verteilung und Datenaustausch			
S&K	Synchronisation durch AP Kommunikation synchron und asynchron		
LB&LT	blockweise in jeder Dim (alle Felder gleich)		
Wettervorhersagemodelle (mittels FD)			
[123]			www-ad.fsl.noaa.gov/mvpab/hpcs/nnt.html

OPlus			Oxford University
SPMD. Objektmengen mit Daten, Inter-Konnektivität und parallelen Operationen			
statisch unstrukturierte Gitter	F77-Datentypen F77		PVM, MPI PVM, MPI
Operationen auf verteilten Datenfeldern (Objektmengen), parallele I/O-Unterstützung			
S&K	implizite Synchronisation vor Schleifen des AP Kommunikation, Überlappung mit Berechnung		
LB&LT	RCB eingebaut oder durch AP automatisch		
gitterbasierte Anwendungen (z.B. CFD)			
[31]			www.comlab.ox.ac.uk/oucl/oxpara/parallel/oplus.htm

PARTI			ICASE, VA; Syracuse Univ., NY; Yale Univ., CT
SPMD. verteilte unstrukturierte Gitter, globale Indizes			
statisch unstrukturierte Gitter	C-/F77-Datentypen C, F77		NX Intel, IBM SP1, CM-5
nichtlokaler Zugriffe mittels *translation table* (Inspector/Executor)			
S&K	explizite Synchronisation optimierte Zugriffe (Hashtabellen, Nachrichtenanzahl)		
LB&LT	statisch (z.B. RSB) als *Preprocessing*-Schritt		
unstrukturierte (Mehrgitter-)Verfahren (z.B. CFD)			
[38, 138]			www.cs.umd.edu/projects/hpsl/chaos.html

PetSc			Argonne National Laboratory
SPMD. transparente Datenverteilung, verteilte Indexmengen			
statisch strukturiert/unstrukturiert		verteilte VuM, Datenfelder C, C++, F77	MPI MPI
Vektor- bzw. Matrix-Assemblierung, globale Indizes wie bei Chaos			
S&K	Synchronisation durch AP (Assemblierung, Gather/Scatter) Kommunikation, Überlappung mit Berechnung		
LB&LT	automatisch		
numerische Anwendungen (z.B. CFD)			
[6, 7]			www.mcs.anl.gov/petsc/petsc.html

Runtime System Library (RSL)			Argonne National Laboratory
Host/Node. verteilte Datenfelder			
dyn. Gitterschachtelung strukturierte Gitter		F77-Datentypen F77	Chameleon [61], PICL IBM SP-1, Intel Delta
vordefinierte Kommunikation auf Prozessorgrenzen (*Stencil*)			
S&K	Synchronisation durch AP Kommunikation *intra-grid* und *inter-grid*		
LB&LT	2D blockweise		
Modelle zur Wettervorhersage und Geophysik			
[106]			www.mcs.anl.gov/Projects/RSL

SUMAA3d			Argonne National Laboratory
SPMD. verteilte Dreiecksgitter			
dynamisch Dreiecksgitter		integer, double C	MPI MPI
linearer Löser (BlockSolve [76, 79]), Gitterverfeinerung, Umverteilung			
S&K	Synchronisation innerhalb von *BlockSolve* Kommunikation durch *BlockSolve*		
LB&LT	eigenes heuristisches Verfahren [77] Ein-/Auspacken automatisch		
Anwendungen auf unstrukturierten Gittern			
[77, 78]			www.mcs.anl.gov/home/freitag/SC94demo

Literaturverzeichnis

[1] S. Abdennadher, F. Bry, N. Eisinger, und T. Geisler. The theorem prover Satchmo: Strategies, heuristics, and applications (system description). Forschungsbericht PMS-FB-1995-3, Institut für Informatik, LMU München, 1995.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, und B. Smith. The Tera computer system. In *4th International Conference on Supercomputing*, Amsterdam, 1990.

[3] I.G. Angus. Are object oriented programming methods suitable for numerical computing on parallel machines? In Kowalik und Grandinetti [90].

[4] J. N. C. Árabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, und P. Stephan. Dome: Parallel programming in a distributed computing environment. In *Proceedings of the International Parallel Processing Symposium 1996*, 1996.

[5] S. B. Baden. Programming abstractions for dynamically partitioning and coordinating localized scientific calculations running on multiprocessors. *SIAM J. Sci. Stat. Comput.*, 12 (No. 1):145–157, 1991.

[6] S. Balay, W. Gropp, L. C. McInnes, und B. Smith. PETSc 2.0 user's manual. Technical Report ANL-95/11, Argonne National Laboratory, 1995.

[7] S. Balay, W. D. Gropp, L. C. McInnes, und B. F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. In E. Arge, A. M. Bruaset, und H. P. Langtangen, Hrsg., *Modern Software Tools in Scientific Computing*. Birkhauser Press, 1997.

[8] R. E. Bank. *PLTMG: A Software Package for Solving Elliptic Partial Differential Equiations, User's Guide 7.0*. SIAM, Philadelphia, 1994.

[9] J.J. Barton und L.R. Nackman. *Scientific and Engineering C++*. Addison-Wesley Inc., 1994.

[10] P. Bastian. *Parallele adaptive Mehrgitterverfahren*. Dissertation, Universität Heidelberg, 1994.

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

[12] P. Bastian. Parallel adaptive multigrid methods. Preprint 93-60, Interdisziplinäres Zentrum für Wissenschaftliches Rechnen, Heidelberg, Oktober 1993.

[13] P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuss, H. Rentz-Reichert, und C. Wieners. UG – a flexible software toolbox for solving partial differential equations. *Computation and Visualization in Science*, (1), 1997.

[14] P. Bastian, K. Birken, S. Lang, und C. Wieners. Large scale PDE applications on parallel computers. In *Submitted for Mannheim SuParCup '97*, 1997.

[15] P. Bastian und G. Horton. Parallelization of robust multigrid methods: ILU factorization and frequency decomposition method. *SIAM J. Sci. Stat. Comput.*, 12(6):1457–1470, 1991.

[16] F. Bellosa. Implementierung adaptiver numerischer Verfahren auf komplexen Geometrien mit leichtgewichtigen Prozessen. Interner Bericht TR-14-7-94, Universität Erlangen-Nürnberg, IMMD IV, 1994.

[17] K. Birken. Ein Parallelisierungskonzept für adaptive, numerische Berechnungen. Diplomarbeit im Fach Informatik, Friedrich-Alexander-Universität Erlangen-Nürnberg, IMMD III, 1993.

[18] K. Birken. Parallelisierung von Multilevelverfahren bei adaptiver Verfeinerung. In *Beiträge zum 1. Berichtskolloquium des GK PVS*, Graduiertenkolleg Parallele und Verteilte Systeme, Universität Stuttgart, Oktober 1993.

[19] K. Birken. Dynamic Distributed Data in a parallel programming environment – DDD Reference Manual. Forschungs- und Entwicklungsberichte RUS–23, Rechenzentrum der Universität Stuttgart, Germany, September 1994.

[20] K. Birken. An efficient programming model for parallel and adaptive CFD-algorithms. In *Proceedings of Parallel CFD Conference 1994*, Kyoto, Japan, 1995. Elsevier Science.

[21] K. Birken. Towards semi-automatic parallelization for dynamic, grid-based applications. Technical report, Universität Stuttgart, Institut für Computeranwendungen III, 1997. in Vorbereitung.

[22] K. Birken und P. Bastian. Dynamic Distributed Data (DDD) in a parallel programming environment – specification and functionality. Forschungs- und Entwicklungsberichte RUS–22, Rechenzentrum der Universität Stuttgart, Germany, September 1994.

[23] K. Birken und R. Rühle. Dynamic Distributed Data: Efficient, portable and easy to use. In D'Hollander et al. [44], S. 303–310.

[24] BLAS frequently asked questions. www.netlib.org/blas/faq.html.

[25] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, und Y. Zhou. Cilk: An efficient multithreaded runtime system. In *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, S. 207–216, Santa Barbara, CA, 1995.

[26] G. Botorog und H. Kuchen. Algorithmic skeletons for adaptive multigrid methods. In *Proceedings Irregular '95*, LNCS 980, S. 27–41. Springer Verlag, 1995.

[27] G. Botorog und H. Kuchen. Algorithmic skeletons in an imperative language for distributed programming. Technical Report 9504, University of Giessen, Germany, 1995.

[28] G. Bringmann, K.-P. Gulden, D. Vitt, K. Birken, und C. Helf. AdaptivSearch: A novel iterative algorithm for the generation of n-dimensional hypersurfaces using an adaptive numerical strategy. *J. Mol. Model.*, 1:161–175, 1995.

[29] T. Brüggemann, F. Bry, N. Eisinger, T. Geisler, S. Panne, H. Schütz, S. Torge, und A. Yahya. Satchmo: Minimal model generation and compilation (system description). Forschungsbericht PMS-FB-1996-5, Institut für Informatik, LMU München, 1996.

[30] F. Bry und A. Yahya. Minimal model generation with positive unit hyperresolution tableaux. In *5th Workshop on Theorem Proving with Tableaux and Related Methods*, LNAI. Springer, 1996.

[31] D. A. Burgess, P. I. Crumpton, und M. B. Giles. A parallel framework for unstructured grid solvers. In Wagner et al. [142].

[32] J. B. Carter. *Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency*. Dissertation, Rice University, 1993.

[33] K. M. Chandy und C. Kesselman. CC++: A declarative concurrent object oriented programming notation. Technical Report CS-TR-92-01, Department of Computer Science, California Institute of Technology, 1992.

[34] C. Chang, A. Sussman, und Joel Saltz. Object-oriented runtime support for complex distributed data structures. Technical Report CS-TR 3428, Dept. of Computer Science, University of Maryland, MD, 1995.

[35] C. Chang, A. Sussman, und Joel Saltz. Support for distributed dynamic data structures in C++. Technical Report CS-TR 3266, Dept. of Computer Science, University of Maryland, MD, 1995.

[36] A. Chien. Efficient runtime support for concurrent objects: the Illinois Concert System. Vortrag, International Workshop on Run-Time Systems for Parallel Programming, 1997.

[37] A. Chien, J. Dolby, B. Ganguly, V. Karamcheti, und X. Zhang. Supporting high level programming with high performance: The Illinois Concert system. In Hellwagner [68], S. 15–24.

[38] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, und J. Saltz. Software support for irregular and loosely synchronous problems. *Computing Systems in Engineering*, 3 (No. 1-4):43–52, 1992.

[39] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[40] D. E. Culler, A. C. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, und K. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, S. 262–273. IEEE Computer Society Press, 1993.

[41] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, und J. Wawrzynek. Fine-grain parallelism with minimal hardware support: a compiler-controlled Threaded Abstract Machine. In *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, 1991.

[42] J. Darlington, A. J. Field, P. G. Harrison, et al. Parallel programming using skeleton functions. In *Proceedings of PARLE '93*, LNCS 694, S. 146–160. Springer Verlag, 1993.

[43] P. Deuflhard, P. Leinen, und H. Yserentant. Concepts of an adaptive hierarchical finite element code. *IMPACT of Computing in Science and Engineering*, 1(3-35), 1989.

[44] E. D'Hollander, G.R. Joubert, F.J. Peters, und D. Trystram, Hrsg. *Parallel Computing: State-of-the-Art and Perspectives, Proceedings of ParCo '95*, Amsterdam, Netherlands, 1996. Elsevier Science.

[45] J. Dongarra, R. Pozo, und D. Walker. LAPACK++: A design overview of object-oriented extensions for high performance linear algebra. In *Proceedings of Supercomputing '93*, S. 162–171. IEEE Computer Society Press, 1993.

[46] K. Dowd. *High Performance Computing*. O'Reilly & Associates, 1993.

[47] M. Dubois, C. Scheurich, und F. A. Briggs. Synchronization, coherence and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–21, February 1988.

[48] D. R. Emerson und Y. F. Hu, Hrsg. *Proceedings of Dynamic Load Balancing on MPP systems: Progress, Challenges and Issues*, November 1995.
www.dl.ac.uk/TCSC/Staf/Hu_Y_F/MEETING/meeting.html.

[49] B. Erdmann, J. Lang, und R. Roitzsch. KASKADE Manual, Version 2.0. Technical Report TR 93-5, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Germany, December 1993.

[50] HPCN at large for industrial applications: The ESPRIT initiative EUROPORT. GMD/SCAI, St. Augustin, Germany. www.gmd.de/SCAI/europort/Welcome.html.

[51] C. Farhat und H. D. Simon. TOP/DOMDEC: a software tool for mesh partitioning and parallel processing. Technical Report CU-CSSC-93-11, University of Colorado, College of Engineering, Boulder, Colorado, 1993.

[52] M. Fishler, G. Hockney, P. Mackenzie, und M. Uchima. Canopy version 7.0. Technical Report TW 185, Computing R& D, Fermilab, Batavia, IL 60510, 1994.

[53] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computers*, 21(9):948–960, 1972.

[54] High Performance Fortran Forum. High Performance Fortran, language specification, May 1993.

[55] L. Freitag, M. T. Jones, und P. E. Plassmann. Interactive adaptive mesh refinement in the CAVE. Argonne National Laboratory.
www.mcs.anl.gov/home/freitag/SC94demo/project/cave_refine.html.

[56] U. Geuder, M. Härdtner, B. Wörner, und R. Zink. The GRIDS parallel model for grid-based scientific applications. In *Proceedings of the 8th International Parallel Processing Symposium*, Cancun, Mexico, 1994.

[57] U. Geuder, M. Härdtner, B. Wörner, und R. Zink. Implementation of a parallel euler solver with GRIDS. In *High Performance Computing and Networking*, München, 1994.

[58] U. Geuder, M. Härdtner, B. Wörner, und R. Zink. Scalable execution control of grid-based scientific applications on parallel systems. In *Proceedings of the 1994 Scalable High-Performance Computing Conference*, Knoxville, TN, 1994.

[59] A. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. *IEEE Computer*, 5(26):39–51, May 1993.

[60] W. Gropp. BlockComm for Fortran. Technical Report ANL-93/00 (UC-405), Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, 1993.

[61] W. Gropp und B. Smith. User's manual for Chameleon parallel programming tools. ANL Report ANL-93/23, Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, Ill., 1993.

[62] Haskell homepage. Universität Aachen, Department of Computer Science. www-i2.informatik.rwth-aachen.de/Forschung/FP/Haskell.

[63] P. J. Hatcher und M. J. Quinn. *Data-Parallel Programming*. Scientific and Engineering Computation Series. MIT Press, Cambridge, MA, 1991.

[64] A. Heirich. Scalable load balancing by diffusion. Technical Report TR-94-04, California Institute of Technology, Pasadena, CA, 1994.

[65] C. Helf, K. Birken, und U. Küster. Parallelization of a highly unstructured Euler-solver based on arbitrary polygonal control volumes. In N. Satofuka, J. Periaux, und A. Ecer, Hrsg., *Proceedings of the Parallel CFD '95 Conference*, Pasadena, USA, 1995. Elsevier.

[66] C. Helf, K. Birken, und U. Küster. Integrating grid generation and flow simulation based on a highly unstructured euler solver. In N.N., Hrsg., *Sixth International Symposium on Computational Fluid Dynamics - A Collection of Technical Papers*, Lake Tahoe, Nevada, USA, September 4-8, 1995.

[67] C. Helf und U. Küster. A finite volume method with arbitrary polygonal control volumes and high order reconstruction for the Euler equations. In Wagner et al. [142].

[68] H. Hellwagner, Hrsg. *Second International Workshop on High-Level Programming Models and Supportive Environments.* IEEE Computer Society Press, 1997.

[69] R. Hempel, H.-C. Hoppe, und A. Supalov. PARMACS 6.0 library interface. Technical report, Gesellschaft für Mathematik und Datenverarbeitung mbH, St. Augustin, Germany, 1992.

[70] B. Hendrickson. Can static load balancing algorithms be appropriate in a dynamic setting? In Emerson und Hu [48].

[71] B. Hendrickson und R. Leland. The Chaco User's Guide Version 1.0. Report SAND93-2339, Sandia National Laboratories, Albuquerque, NM, 1993.

[72] G. Horton. A multi-level diffusion method for dynamic load balancing. *Parallel Computing*, 19:209–218, 1993.

[73] G. Horton und S.T. Leutenegger. A multi-level solution algorithm for steady-state markov chains. ICASE Report 93-81, ICASE, NASA Langley Research Center, Hampton, VA, November 1993.

[74] K. Hwang. *Advanced Computer Architecture*. McGraw-Hill, 1993.

[75] IEEE. POSIX Guide to an Open System Environment. PASC/POSIX Committee, P1003.0/D15.

[76] M. T. Jones und P. E. Plassmann. BlockSolve V1.0: Scalable library software for the parallel solution of sparse linear systems. ANL Report ANL-92/46, Argonne National Laboratory, Argonne, Ill., 1992.

[77] M. T. Jones und P. E. Plassmann. A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–669, 1993.

[78] M. T. Jones und P. E. Plassmann. Parallel algorithms for the adaptive refinement and partitioning of unstructured meshes. In *Proceedings of the 1994 SHPCC*, S. 726–733. IEEE, 1994.

[79] M. T. Jones und P. E. Plassmann. BlockSolve95: Scalable library software for the parallel solution of sparse linear systems. ANL Report ANL-95/48, Argonne National Laboratory, Argonne, Ill., 1995.

[80] L. V. Kale und Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, Band 28(10) der *ACM Sigplan Notes*, S. 91–108. ACM, 1993.

[81] L. V. Kale, B. Ramkumar, A. B. Sinha, und A. Gursoy. The Charm parallel programming language and system: Part I – description of language features. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[82] L. V. Kale, B. Ramkumar, A. B. Sinha, und A. Gursoy. The Charm parallel programming language and system: Part II – the runtime system. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[83] G. Karypis und V. Kumar. METIS – unstructured graph partitioning and sparse matrix ordering system. Technical report, University of Minnesota, Minneapolis, MN, 1995.

[84] J. De Keyser. Load balancing data parallel programs on distributed memory computers. *Parallel Computing*, 19(11):1199–1219, 1993.

[85] J. De Keyser. LOCO 2.1: A library supporting data parallelism on MIMD computers. Report TW 185, K. U. Leuven, Leuven, Belgium, 1994.

[86] J. De Keyser und D. Roose. A software tool for load balanced adaptive multiple grids on distributed memory computers. In *Proceedings of the Sixth Distributed Memory Computing Conference*, S. 122–128. IEEE Computer Society Press, 1991.

[87] S. R. Kohn. *A Parallel Software Infrastructure for Dynamic Block-Irregular Scientific Calculations*. Dissertation, University of California at San Diego, 1995.

[88] S. R. Kohn und S. B. Baden. A robust parallel programming model for dynamic non-uniform scientific computations. CSE Technical Report CS94-354, Dept. of Comp. Science and Eng., Univ. of California, San Diego, California, 1994.

[89] C. Koppe. Sleeping Threads: A kernel mechanism for support of efficient user level threads. In *Proceedings of the 7th International Conference on Parallel and Distributed Computing and Systems*, Anaheim, 1995. IASTED-ACTA Press.

[90] J. S. Kowalik und L. Grandinetti, Hrsg. *Software for Parallel Computation*, Band 106 der *NATO ASI Series*. Springer Verlag, 1992.

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

[92] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):241–248, 1979.

[93] S. Lang. Lastverteilung für parallele, adaptive numerische Mehrgitterberechnungen. Diplomarbeit im Fach Informatik, Friedrich-Alexander-Universität Erlangen-Nürnberg, IMMD III, 1994.

[94] S. Lang. Adaptive refinement and derefinement of unstructured grid hierarchies. Technical report, Universität Stuttgart, Institut für Computeranwendungen III, 1997. in Vorbereitung.

[95] K. Langendoen, M. Haines, und G. Benson, Hrsg. *International Workshop on Run-Time Systems for Parallel Programming*. Vrije Universiteit Amsterdam, 1997. Report IR-417.

[96] J. R. Larus, B. Richards, und G. Viswanathan. C**: A large-grain, object-oriented, data parallel programming language. Technical Report 1126, University of Wisconsin-Madison, 1992.

[97] P. Leinen. *Ein schneller adaptiver Löser für elliptische Randwertprobleme auf Seriell- und Parallelrechnern*. Dissertation, Universität Dortmund, 1990.

[98] M. Lemke. *Multilevelverfahren mit selbstadaptiven Gitterverfeinerungen für Parallelrechner mit verteiltem Speicher*. R. Oldenbourg, München/Wien, 1994.

[99] M. Lemke und D. Quinlan. P++, a C++ virtual shared grids based programming environment for architecture-independent development of structured grid applications. In *Proceedings of the Second Joint International Conference on Vector and Parallel Processing*, LNCS, S. 121–126. Springer Verlag, 1992.

[100] D. Lenoski, J. Laudon, T. Joe, D. Nakashira, L. Stevens, A. Gupta, und J. Hennessy. The DASH prototype: Implementation and performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, S. 92–103, Gold Coast, Australia, 1992.

[101] J. Lepper. Simulation dreidimensionaler Strömungsvorgänge unter Verwendung einer Domain-Decomposition-Methode auf Parallelrechnersystemen. In *Beiträge zum 3. Berichtskolloquium des GK PVS*, Graduiertenkolleg Parallele und Verteilte Systeme, Universität Stuttgart, Juli 1995.

[102] W. Lohr. Parallelisierung eines Delauney-Triangulierers für 2-dimensionale Punktmengen. Diplomarbeit (Interner Bericht Nr. 129), Universität Stuttgart, Rechenzentrum der Universität Stuttgart, Allmandring 30, D-70550 Stuttgart, Oktober 1996.

[103] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, und F. Bodin. Performance analysis of pC++: A portable data-parallel programming system for scalable parallel computers. In *Proceedings of the 8th International Parallel Processing Symposium (IPPS)*, Cancun, Mexico, 1994.

[104] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, F. Bodin, und S. Kesavan. Implementing a parallel C++ runtime system for scalable parallel systems. In *Proceedings of Supercomputing '93*, S. 588–597. IEEE Computer Society Press, 1993.

[105] M. Metcalf und J. Reid. *Fortran 90 Explained*. Oxford University Press, 1990.

[106] J. Michalakes. RSL: A parallel runtime system library for regular grid finite difference models using multiple nests. Technical Report ANL/MCS-TM-197, Argonne National Laboratory, Argonne, Illinois, 1994.

[107] B. Moon, M. Uysal, und J. Saltz. Index translation schemes for adaptive computations on distributed memory multiprocessors. In *Proceedings of the Ninth International Parallel Processing Symposium*, S. 812–819, 1995.

[108] J.A. Moore, P.J. Hatcher, und M.J. Quinn. Stream*: Fast, flexible, data-parallel I/O. In D'Hollander et al. [44], S. 287–294.

[109] MPI-2: Extensions to the Message Passing Interface. University of Tennessee. www.mcs.anl.gov/mpi2/mpi2.index.

[110] J. Nieplocha, R.J. Harrison, und R.J. Littlefield. Global arrays: A portable „shared memory" programming model for distributed memory computers. In *Proceedings of Supercomputing '94*. IEEE Computer Society Press, 1994.

[111] Technical overview of the Origin family. www.sgi.com/Products/hardware/servers/technology/overview.html.

[112] G. Papakonstantinou und P. Tsanakas. Distributed shared memory: Principles and implementation. In Kowalik und Grandinetti [90], S. 100–110.

[113] M. Parashar und J. C. Browne. Distributed dynamic data-structures for parallel adaptive mesh-refinement. In *Proceedings of the International Conference for High Performance Computing*, 1995.

[114] M. Parashar und J. C. Browne. Scalable distributed dynamic grids: A survey of existing support. Technical report, Department of Cmputer Science & Center for Relativity, University of Texas at Austin, 1995. godel.ph.utexas.edu/Members/parashar/survey/mainhtml.html.

[115] M. Parashar und J. C. Browne. On partitioning dynamic adaptive grid hierarchies. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, 1996.

[116] Passion – parallel and scalable software for I/O. Syracuse University. www.cat.syr.edu/passion.html.

[117] F. Pearce. Long range forces and hydrodynamics. In Emerson und Hu [48].

[118] F. Pellegrini. Scotch 3.0 User's Guide. Technical report, Université Bordeaux, Talence, France, 1995.

[119] R. Ponnusamy, J. Saltz, und A. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Proceedings of Supercomputing '93*, S. 361–370. IEEE Computer Society Press, 1993.

[120] R. Pozo. A stream-based interface in C++ for programming heterogeneous systems. In J.J. Dongarra und B. Tourancheau, Hrsg., *Environments and Tools for Parallel Scientific Computing*. North-Holland, 1993.

[121] R. Preis. The PARTY partitioning-library – User's Guide. Technical report, Universität Paderborn, Paderborn, Germany, 1995.

[122] H. Ritzdorf und K. Stüben. Adaptive multigrid on distributed memory computers. Arbeitspapiere der GMD 781, Gesellschaft für Mathematik und Datenverarbeitung, Germany, September 1993.

[123] B. Rodriguez, L. Hart, und T. Henderson. NNT version 2.0 user's guide. Technical report, High Performance Computing Section, National Oceanic and Atmospheric Administration, Boulder, Colorado, 1994.

[124] A. Rogers, M. C. Carlile, J. H. Reppy, und L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[125] A. Rogers, J. H. Reppy, und L. J. Hendren. Supporting SPMD execution for dynamic data structures. In U. Banerjee, D. Gelernter, A. Nicolau, und D. Padua, Hrsg., *Proceedings of the Languages and Compilers for Parallelism Workshop 1992*. Springer Verlag, 1993.

[126] D. Roose und R. Van Driessche. Distributed memory parallel computers and computational fluid dynamics. In H. Deconinck, editor, *Lecture Notes on Computational Fluid Dynamics, LS 1993-04,* Von Karman Institute for Fluid Dynamics, Belgium, 1993.

[127] M. Rudgyard, T. Schönfeld, R. Striujs, G. Audemar, und P. Leyland. A modular approach for computational fluid dynamics. In Wagner et al. [142].

[128] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994.

[129] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Publishing Company, 1989.

[130] Scalable I/O Initiative – homepage. California Institute of Technology. www.cacr.caltech.edu/ jpool/.

[131] T. Schmidt und R. Rühle. On–line visualization of arbitrary unstructured, adaptive grids. In *Proceedings of Sixth Eurographics Workshop on Visualization in Scientific Computing,* Chia, Italy, May 1995.

[132] Workshop Software Engineering im Scientific Computing (SESC). Technische Hochschule Hamburg-Harburg, Arbeitsbereich Mathematik, 1995. www.tu-harburg.de/mat/sesc/.

[133] S. D. Sharma, R. Ponnusamy, B. Moon, Y.-S. Hwang, R. Das, und J. Saltz. Run-time and compile-time support for adaptive irregular problems. In *Proceedings of Supercomputing '94,* S. 97–106. IEEE Computer Society Press, 1994.

[134] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering,* 2(2/3):135–148, February 1991.

[135] Sisal - a high performance, portable, parallel programming language. Lawrence Livermore National Laboratory. www.llnl.gov/sisal/SisalHomePage.html.

[136] I. Sommerville. *Software Engineering.* Addison-Wesley, third edition, 1989.

[137] L. Sterling und E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Massachusetts, 1994.

[138] A. Sussman, J. Saltz, R. Das, S. Gupta, D. Mavriplis, R. Ponnusamy, und K. Crowley. PARTI primitives for unstructured and block structured problems. *Computing Systems in Engineering*, 3 (Nos 1-4):73–86, 1992.

[139] S. Taylor, J. Watts, M. Rieffel, und M. Palmer. The Concurrent Graph: Basic technology for irregular problems. *IEEE Parallel and Distributed Technology*, 4(2):15–25, 1996.

[140] J. F. Thompson und N. P. Weatherill. Aspects of numerical grid generation: Current science and art. In *Proceedings of 11th AIAA Applied Aerodynamics Conference*, Monterey, California, 1993. American Institute of Aeronautics and Astronautics.

[141] T. von Eicken, D. E. Culler, S. C. Goldstein, und K. E. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architectures*. ACM Press, 1992.

[142] S. Wagner, E.H. Hirschel, J. Périaux, und R. Piva, Hrsg. *Proceedings of the Second European Computational Fluid Dynamics Conference*, Stuttgart, Germany, 1994. Wiley & Sons.

[143] C. Walshaw, M. Cross, und M. G. Everett. Dynamic mesh partitioning: A unified optimisation and load-balancing algorithm. Mathematics Research Report 95/IM/06, Centre for Numerical Modelling and Process Analysis, University of Greenwich, London, SE18 6PF, UK, 1995.

[144] K. Warendorf und R. Rühle. A parallel self-adaptive grid generation strategy for a highly-unstructured euler solver. In *Proceedings of the Parallel CFD '97 Conference*, Manchester, GB, 1997.

[145] J. Watts, M. Rieffel, und S. Taylor. Practical dynamic load balancing for irregular problems. In *Proceedings of Irregular '96*, Band 1117 der *LNCS*, S. 299–306. Springer Verlag, 1996.

[146] C.-P. Wen, S. Chakrabarti, E. Deprit, A. Krishnamurthy, und K. Yelick. Runtime support for portable distributed data structures. In *Workshop on Languages, Compilers and Runtime Systems for Scalable Computers*, 1995.

[147] C. Wieners. The implementation of parallel multigrid methods for finite elements. SFB 404 Preprint 97/13, Universität Stuttgart, Institut für Computeranwendungen III, 1997.

[148] R. D. Williams. DIME - distributed irregular mesh environment. Report C3P 861, California Institute of Technology, Februar 1990.

[149] R. D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. Report C3P 913, California Institute of Technology, Pasadena, CA, 1990.

[150] R. D. Williams und R. Glowinski. Distributed irregular finite elements. In C. Taylor, Hrsg., *Numerical Methods in Laminar and Turbulent Flow*, Band 6, S. 3–13, Swansea, UK, 1989. Pineridge Press. Caltech Report C3P 715.

[151] K. Yelick, S. Chakrabarti, E. Deprit, J. Jones, A. Krishnamurthy, und C. Wen. Data structures for irregular applications. In *Workshop on Parallel Algorithms for Unstructured and Dynamic Problems*, Piscataway, NJ, 1993.

[152] E. Yourdon. *Object-Oriented Systems Design – An Integrated Approach*. Prentice Hall (Yourdon Press), 1994.

Index

Lebenslauf

Persönliche Daten

Name	Klaus Birken
Adresse	Fritz-Reuter-Str. 22, 70193 Stuttgart
Geburtsort	Sulzbach-Rosenberg
Geburtstag	16. September 1968
Staatsangehörigkeit	deutsch
Familienstand	verheiratet

Ausbildung

Grundschule, Sulzbach-Rosenberg	1974-1978
Herzog-Christian-August-Gymnasium,	
Sulzbach-Rosenberg	1978-1987
Abitur	26. Juni 1987
Friedrich-Alexander-Universität,	
Erlangen-Nürnberg	1987-1993
Vordiplom im Fach Informatik	25. Oktober 1989
Diplom im Fach Informatik	25. Februar 1993
Graduiertenkolleg Parallele und Verteilte Systeme,	
am Rechenzentrum der Universität Stuttgart	April 1993 bis März 1996

Anstellung

wiss. Angestellter, Universität Stuttgart,	
Institut für Computeranwendungen III	seit 1. April 1996