Diplomarbeit

Parallelisierung der Berechnung des Faltenwurfs von Textilien mit Partikelsystemen

vorgelegt von
Marcus Ritt

Tübingen, 8. Juli 1997

Betreuer:
Prof. Dr. Wolfgang Rosenstiel
Dipl. Inf. Tilmann Bubeck
Dr. Bernd Eberhardt

Ich versichere, die vorliegende Arbeit eigenständig und ohne Benutzung unzulässiger Hilfsmittel oder anderer als den angegebenen Quellen erstellt zu haben.

Tübingen, 8. Juli 1997
Marcus Ritt

Wilhelm-Schickard-Institut für Informatik
Lehrstuhl Technische Informatik
Eberhardt-Karls-Universität Tübingen
Sand 13
72076 Tübingen

Inhaltsverzeichnis

1 Einleitung **1**

2 Zielarchitekturen **3**
2.1 Klassifizierung paralleler Rechner 3
2.2 SUN Sparc MP 4
2.3 NEC SX-4 4
2.4 IBM RS/6000 SP System 6
2.5 SGI ONYX2 8
2.6 Workstation-Netz 9

3 Software-Basis **10**
3.1 Programmierung paralleler Rechner 10
3.2 PVM 11
3.3 DTS 12

4 Physikalisches Modell **14**
4.1 Graphik-Bibliotheken 14
4.2 Berechnung des Faltenwurfs 14
4.3 Algorithmen 15

5 Parallelisierung **21**
5.1 Vorbetrachtungen 21
5.2 Programme und Messungen 23
5.3 Gemeinsamer Speicher 24
5.3.1 SGI ONYX2 25
5.3.2 SUN Sparc MP 32
5.4 Verteilter Speicher 34
5.4.1 Workstation-Netz 36
5.4.2 Andere Architekturen 38
5.5 Fazit 39

6 Erweiterung von DTS **40**
6.1 Grenzen von DTS 40
6.2 Distributed Shared Memory 40
6.3 Entwurfskriterien und Lösungsansatz 43
6.4 Benutzerschnittstelle 46

6.5 Implementierung 49
6.6 Erweiterungsmöglichkeiten 56

7 C++-Codegenerator **57**
7.1 Verwendung von `CPPgen` 57
7.2 Erzeugung von Code für DCO 58
7.3 Implementierung 60

8 Zusammenfassung und Ausblick **62**

A Referenz der DSM-Primitive **64**

Literaturverzeichnis **66**

Abbildungsverzeichnis

2.1 Einteilung von MIMD-Rechnern 4
2.2 Bus-basierte Multiprozessor-Architektur 5
2.3 Kreuzschienenverteiler 5
2.4 Struktur des SP-Systems 7
2.5 ONYX2 Systemstruktur 9
2.6 Bus-basierte Multirechner-Architektur 9

3.1 Thread-Aufruf in DTS 13

4.1 Beispiel einer Szene des Partikelsystems 15
4.2 Klassenhierarchie des Partikelsystems 16
4.3 Flußdiagramm `rkqs` 18
4.4 Winkelabhängige Kräfte 20

5.1 Profiles auf der ONYX2 28
5.2 Overheads und Beschleunigungen auf der ONYX2 29
5.3 Programmstruktur bei verteiltem Speicher 37

6.1 Klassifizierung der Zugriffe bei schwacher Freigabe-Konsistenz 42
6.2 Lese- und Schreibphasen bei DSM 45
6.3 Beispiele zum Nachrichtenfluß der DSM-Implementierung 50

Tabellenverzeichnis

5.1 Parallelisierte Funktionen bei gemeinsamem Speicher 24
5.2 Normierung der Laufzeitwerte 27
5.3 Erreichbare Beschleunigungen ONYX2 30
5.4 Laufzeiten auf der ONYX2 31
5.5 Overheads auf der ONYX2 32
5.6 Meßergebnisse auf 4-Prozessor SUN-Rechnern 33
5.7 Meßergebnisse im Workstation-Netz 38

6.1 DSM-Nachrichtentypen 51

7.1 Schlüsselworte in `CPPgen` 58
7.2 Zusätzliche Optionen von `CPPgen` 58

Kapitel 1

Einleitung

Schon kurz nach Beginn der Computerentwicklung wurden Stimmen laut, die ein Ende der Leistungssteigerung bei Einprozessorsystemen prophezeiten und nur in Parallelrechnern ein genügendes Leistungspotential für zukünftige Anforderungen sahen. Daß diese nicht recht behielten hatte verschiedene Ursachen. Zum einen wurden damals die enormen technischen Entwicklungsmöglichkeiten von Einprozessorsystemen unterschätzt. Zum anderen gab es die vorhergehende Ausnutzung der Parallelität auf niedrigster Ebene durch Fließbandverarbeitung und parallel arbeitende Funktionseinheiten (Superskalarität). Gene Amdahl formulierte in seinem vielbeachteten Papier von 1967 [1] einige der grundsätzlichen Schwierigkeiten mit denen paralleles Rechnen verbunden ist. Dazu gehören beispielsweise inhomogene Problembereiche mit unregelmäßigen Rändern und globale Datenabhängigkeiten.
Am bekanntesten wurde das nach ihm benannte Gesetz

$$S_{\max} = \lim_{p \to \infty} \frac{1}{s + \frac{1-s}{p}} = \frac{1}{s} \quad ,$$

das die Obergrenze der erreichbaren Beschleunigung S in Abhängigkeit des sequentiellen Programmanteils s angibt. Es gilt, auch wenn zwischenzeitlich Gegenteiliges behauptet wurde[1], unverändert und schränkt die Klasse der Probleme deren parallele Lösung angemessenen Gewinn verspricht grundlegend ein.
Viele der dort angesprochenen Punkte bereiten auch heute noch Probleme, dennoch blieb paralleles Rechnen immer interessant. Die wichtigste Motivation dafür ist die Leistung, denn durch die Kopplung mehrerer Rechner wird die Lösung von Problemen greifbar, die für Einzelrechner aufgrund von Laufzeit- oder Speicheranforderungen zu komplex sind. Weitere Vorteile versprechen die leichte Erweiterbarkeit und erhöhte Ausfallsicherheit, obwohl diese stark von der eingesetzten Technik respektive der Programmierung abhängen. Mit der Entwicklung von Mikroprozessoren, die Groschs Gesetz der quadratischen Abhängigkeit der Rechnerleistung vom Preis ungültig machte [17], sowie der Weiterentwicklung der Kommunikationstechnik, wurde paralleles Rechnen für eine breite Schicht von Anwendern attraktiv. Lokale Workstation-Netze sind heutzutage fast überall anzutreffen, so daß es naheliegt, gerade auch dort Programme durch Ausnutzung von Parallelität zu beschleunigen.

[1] Man glaubte, durch eine Vergrößerung des Problems Amdahls Gesetz umgehen zu können.

Nicht zuletzt wird paralleles Rechnen dadurch immer bedeutender, daß, extrapoliert man die Trends der Prozessorentwicklung der letzten Jahre, in der ersten Dekade des nächsten Jahrhunderts erste physikalische Grenzen erreicht sein werden. Wenn sich nicht durch völlig andere Technologien neue Entwicklungsmöglichkeiten ergeben, kann die Leistung von Rechnersystemen nur durch die Ausnutzung der Parallelität weiter gesteigert werden.
Die Schwierigkeiten bei parallelem Rechnen bleiben die sequentiellen Programmanteile und die komplexe Programmierung. Während ein um einen Faktor zehn schnellerer Rechner, *jedes* Programm in einem Zehntel der Zeit ausführt, verhindert schon ein sequentieller Anteil von 10% das Erreichen dieser Grenze für parallele Programme, unabhängig von der Anzahl eingesetzter Rechner. Auch für die Programmierung liegen noch keine einheitlichen Techniken vor, da die konkrete Parallelisierung sehr stark von der zugrundeliegenden Architektur abhängt.
Beide Punkte, die Suche nach geeigneten parallelen Algorithmen und die Frage, wie man Parallelrechner benutzerfreundlich und trotzdem effizient programmiert, sind Gegenstand der aktuellen Forschung und auch Thema dieser Diplomarbeit.

Zielsetzung

Untersucht wurde eine Anwendung aus dem Bereich der physikalischen Simulation. Dabei wird der Faltenwurf von Textilien mit Hilfe eines Partikelsystems berechnet. Das zugrundeliegende physikalische Modell und die verwendeten Algorithmen sind in Kapitel 4 beschrieben.
Die Anwendung sollte auf parallele Programmanteile untersucht und diese dann auf Basis des „Distributed Threads System“ (DTS), einer Entwicklungsumgebung für parallele Programme, implementiert werden. Auf DTS wird in Kapitel 3 näher eingegangen, Kapitel 5 erläutert die Parallelisierung des Partikelsystems und diskutiert die erhaltenen Resultate. Dort werden auch die verschiedenen Architekturen, die zur Verfügung standen, am Beispiel der Anwendung verglichen. Technische Aspekte der verwendeten Rechner sind in Kapitel 2 zusammengefaßt.
Die Schwierigkeiten mit DTS die sich im Laufe der Diplomarbeit ergaben, fanden ihren Niederschlag in verschiedenen Erweiterungen. So führte die vorher umständliche Implementierung verteilter Algorithmen zur Ergänzung von DTS um „Distributed Shared Memory“ (DSM) und die fehlende Unterstützung von C++ zur Entwicklung eines Codegenerators. Diese Erweiterungen sind in Kapitel 6 und Kapitel 7 dokumentiert.

Kapitel 2

Zielarchitekturen

2.1 Klassifizierung paralleler Rechner

Die klassische Flynnsche Rechnereinteilung [11, 12] unterscheidet einzelne oder mehrere Instruktions- und Datenströme und kommt so zu vier verschiedenen Rechnerklassen. Sie ist heutzutage nur noch von beschränktem Nutzen, da SISD (single instruction, single data) sequentielle Rechner bezeichnet, die Kategorie MISD (multiple instructions, single data) eher künstlich ist und mit den verbleibenden Klassen SIMD (single instruction, multiple data) und MIMD (multiple instructions, multiple data) parallele Rechner nur sehr grob klassifiziert werden können. In die Sparte SIMD fallen vor allem Spezialrechner, etwa Vektor- oder Arrayprozessoren, alle anderen parallelen Architekturen, insbesondere sämtliche in dieser Diplomarbeit verwendeten Rechner, verarbeiten mehrere Instruktionen und verschiedene Daten gleichzeitig und sind damit vom Typ MIMD.

Eine differenziertere Einteilung ergibt sich, wenn man die Rechner nach Speichermodell und Verbindungsstruktur untersucht (Abbildung 2.1 zeigt eine Übersicht über diese Einteilung, nach [34]). Als wichtigstes Merkmal, sowohl in Bezug auf die weitere Realisierung der Hardware als auch die Programmierung des Rechners, muß die Lokalisation des Speichers angesehen werden. Bei *Multiprozessor-Rechnern* liegt ein gemeinsamer kohärenter Speicher vor, während *Multicomputer* auf verteiltem Speicher basieren, d.h. jeder Prozessor besitzt einen eigenen lokalen Speicher, Datenaustausch und Kohärenzalgorithmen fallen in die Verantwortung der Software.

Bei der Verbindungsstruktur unterscheidet man im wesentlichen zwischen Systemen die über einen Bus kommunizieren und solchen mit Punkt-zu-Punkt-Verbindungen. Bus-basierte Rechner sind einfacher zu realisieren, leiden aber unter ihrer schlechten Skalierbarkeit. Bei Punkt-zu-Punkt-Verbindungen hängt das Verhalten wesentlich von der gewählten Verbindungsstruktur ab.

Bandbreite und *Latenz* sind weitere wichtige Merkmale einer Verbindung. Dabei versteht man unter der Latenz die zeitliche Verzögerung bis zum Versand einer Nachricht (bedingt durch Wechsel in den Systemmodus, Protokoll-Overheads etc.) und unter Bandbreite die maximal erreichbare Übertragungsgeschwindigkeit. Man spricht von eng gekoppelten Systemen bei hoher Verbindungsgeschwindigkeit, im umgekehrten Fall von loser Kopplung. Tendenziell sind Multiprozessoren eher der

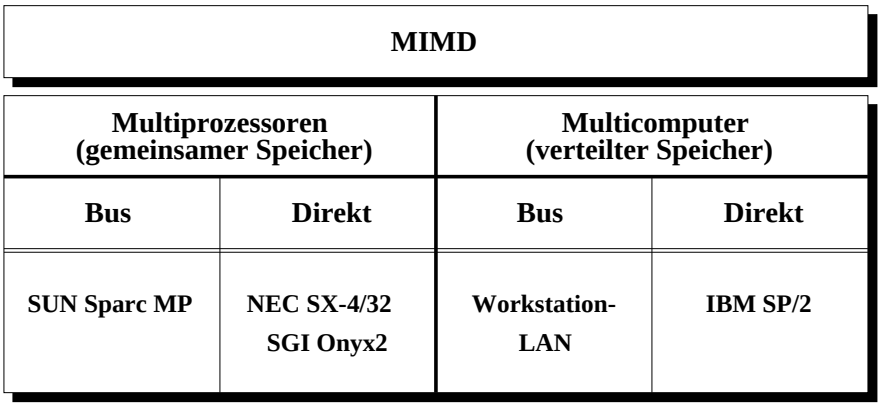

Abbildung 2.1: Einteilung von MIMD-Rechnern

ersten, Multicomputer der zweiten Verbindungsart zuzuordnen.
Im folgenden wird auf die verschiedenen Architekturen konkret eingegangen. Nicht auf allen Plattformen liegt eine parallele Implementierung des Partikelsystems vor, teils deswegen, weil die Architektur nur beispielhaft erwähnt wird und nicht zur Verfügung stand, teils weil aufgrund von technischen Problemen die vorliegende Software nicht portiert werden konnte.

2.2 SUN Sparc MP

Die SUN Sparc MP ist ein Beispiel für einen konventionellen, bus-basierten Multiprozessorrechner, wie in Abbildung 2.2 dargestellt. Der Rechner besteht aus maximal vier SUN HyperSPARC-Prozessoren. SUNs Betriebssystem, Solaris 2.5.1 unterstützt symmetrisches Multiprozessing[1] durch sogenannte „Leichtgewichtsprozesse“, vergleichbar mit normalen Threads (das Konzept eines Threads wird in Kapitel 3 genauer erläutert). Das vorliegende System war mit vier 40-Mhz-Prozessoren und 96 MByte Hauptspeicher ausgestattet.

2.3 NEC SX-4

Eine Hybrid-Architektur liegt mit der NEC SX-4 vor. Sie verbindet klassische Vektorrechner-Technik mit Multiprozessor-Eigenschaften. Die 64-Bit-Prozessoren werden mit 125 MHz getaktet und besitzen neben der gewohnten Skalar- noch eine Vektoreinheit. Bis zu 32 Prozessoren greifen über einen Kreuzschienenverteiler (engl. crossbar switch) auf gemeinsamen Speicher (maximal 16 GB SSRAM) zu (Abbildung 2.3).

[1]Bei symmetrischen Multiprozessoren kann jeder Prozessor Betriebssystemfunktionen ausführen. Die Leistung kann dadurch gesteigert werden, denn in Systemen bei denen ein dezidierter Prozessor für Betriebssystemaufgaben vorgesehen ist, stellt dieser einen potentiellen Flaschenhals dar.

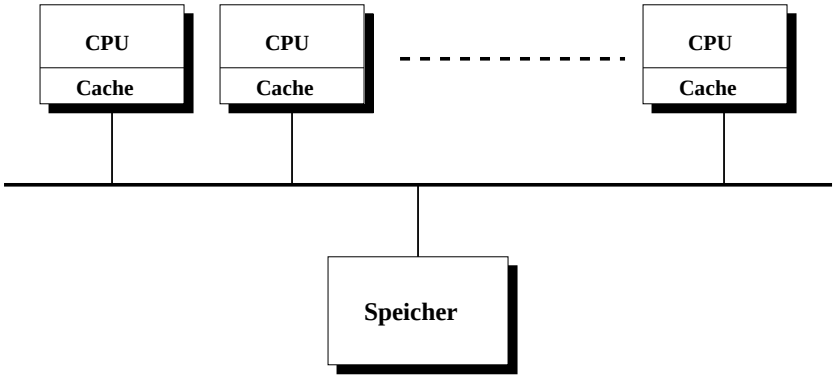

Abbildung 2.2: Bus-basierte Multiprozessor-Architektur

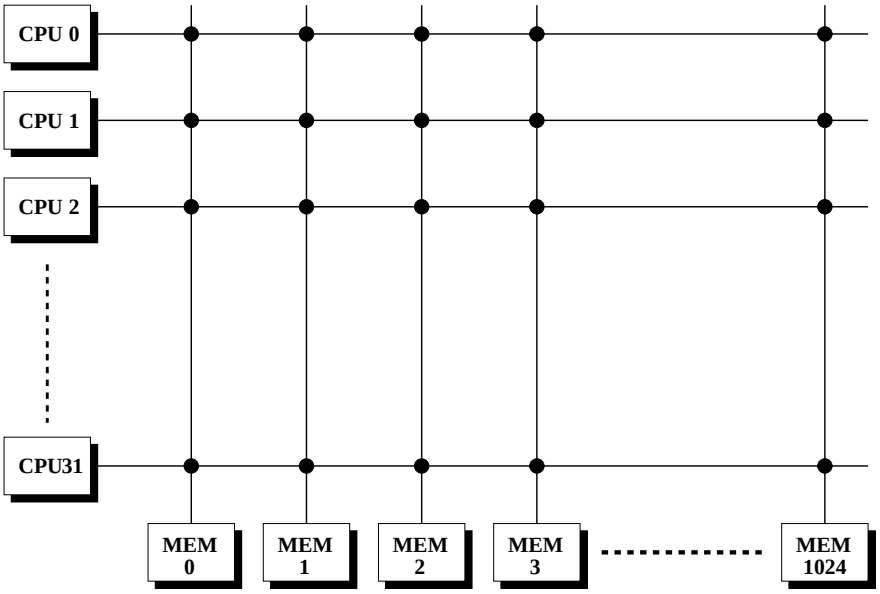

Abbildung 2.3: Kreuzschienenverteiler am Beispiel der NEC SX-4. Bis zu 32 Prozessoren sind mit 1024 Speicherbänken gekoppelt.

In der ursprünglichen Konzeption ging man noch einen Schritt weiter und plante bis zu 16 der oben beschriebenen *Knoten* über einen weiteren Kreuzschienenverteiler zu einem Gesamtsystem zu koppeln, das dann allerdings nicht mehr über gemeinsamen Speicher verfügt.
Zur Programmierung stehen neben POSIX Threads auch High Performance Fortran (HPF) und spezielle Versionen der Message-Passing-Systeme MPI und PVM zur Verfügung.

2.4 IBM RS/6000 SP System

Grundlage des SP-Systems (Scalable parallel) sind IBM RS/6000 Workstations mit POWER2-Prozessoren. Die Rechner werden in zwei Ausführungen eingesetzt, den sogenannten „thin-" und „thick-Nodes", die sich in Taktfrequenz, Busbreite und Speichergröße unterscheiden.
Neben der üblichen Kopplung über ein IP-Netzwerk wird das SP-System vor allem durch den High Performance Switch (HPS) zum Parallelrechner. Dieses spezielle Verbindungsnetzwerk erlaubt eine Punkt-zu-Punkt-Kommunikation zwischen den beteiligten Prozessoren mit hoher Bandbreite und geringer Latenzzeit. Der HPS beinhaltet keinerlei Speicher-Kohärenzmechanismen, d.h. das SP-System bleibt eine Architektur mit verteiltem Hauptspeicher.
Die Rechner werden dazu in Frames von jeweils 4-16 Knoten gruppiert, die über ein Omega[2]-Switchboard mit anderen Frames verbunden werden. Ein Omega-Netzwerk ist in Abbildung 2.4 (a) dargestellt. Je nach Anzahl der Frames werden diese direkt miteinander verbunden oder noch weitere Switchboards als Zwischenstationen eingefügt (Abbildungen 2.4 (b) und (c)). Aus Leistungsgründen werden zur Übertragung von Nachrichten keine festen Verbindungen aufgebaut (circuit-switched), sondern Paket für Paket durch das Netzwerk versandt (packet-switched) und zwar mit einem sogenannten „cut-through"-Verfahren, d.h. ist der betreffende Ausgabeport der Schaltelements frei, werden die Daten ohne Pufferung weitergegeben. Bei Blockierungen wird auf Byte-Basis gepuffert, die Daten des Pakets bleiben bis zur Freigabe auf verschiedenen Stationen entlang des Verbindungspfads gespeichert. Diese Vorgehensweise, das *„wormhole-cut-through"*, steigert die Leistung gegenüber normaler Pufferung nochmals.
Der HPS kann in zwei Modi, der IP- und US-Kommunikation genutzt werden. Bei der IP-Kommunikation teilen sich mehrere Prozesse auf einem Rechner den HPS-Adapter, bei der US-Kommunikation wird der Adapter exklusiv von einem Prozess belegt. In Abhängigkeit vom Kommunikations-Modus steht eine Bandbreite von etwa 20 MB/s bzw. 100 MB/s zur Verfügung.
Zur Programmierung stehen eine speziell auf den HPS zugeschnittene Message-Passing-Bibliothek und darauf aufbauend MPI und PVM zur Verfügung. Weitere Hilfsprogramme zur Ablaufverfolgung und Überwachung paralleler Programme ergänzen die parallele Systemumgebung.

[2]Ein Omega-Netzwerk ist eine spezielle Verbindungsstruktur zwischen n Eingängen und n Ausgängen mit n Zwischenstufen und insgesamt $n \log_2 n$ Schaltpunkten.

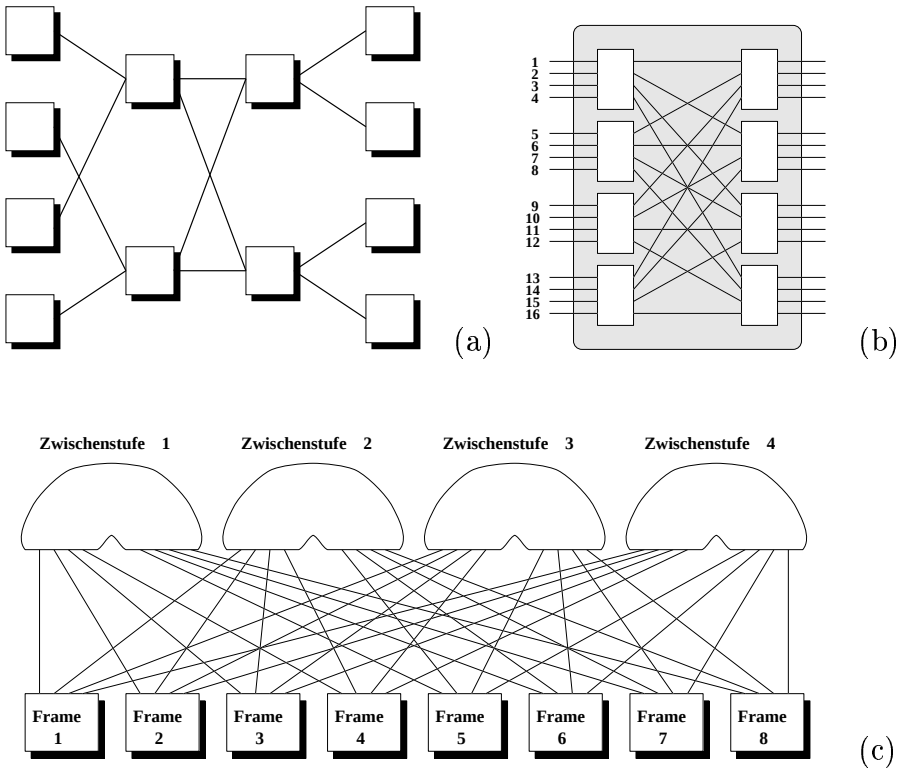

Abbildung 2.4: Struktur des SP-Systems. (a) Omega-Switch. (b) Switchboard mit bis zu 16 Rechnern (1 Frame). (c) System mit 128 Knoten.

2.5 SGI ONYX2

Silicon Graphics geht mit ihren Multiprozessoren den zum SP-System entgegengesetzten Weg. Als Fortsetzung der bisherigen Produktlinie von bus-basierten Multiprozessorsystemen (Power Challenge) bieten auch die Origin 2000- und ONYX-Modelle hardware-gestützten gemeinsamen Speicher. Die physikalische Struktur der Rechner entspricht jedoch einem Hypercube mit auf die einzelnen Prozessoren verteiltem, lokalem Speicher. Damit wird versucht die technischen Grenzen des bus-basierten Ansatzes in Bezug auf die Anzahl der möglichen Prozessoren zu überwinden, d.h. bei diesem Rechner ist Distributed Shared Memory (siehe Kapitel 6) in Hardware implementiert.

Bausteine des Systems sind die R10000-Prozessoren der Tochterfirma MIPS, 64-Bit RISC-CPUs mit einer Taktfrequenz von 195 Mhz und 64 KB Primärcache. Die Prozessoren werden paarweise mit bis zu 4 GB lokalem Speicher in *Knoten* zusammengefaßt, die über sogenannte *Router* miteinander verbunden werden. Bei einem typischen System werden an jeden Router zwei Knoten gekoppelt und die Router untereinander in einer Hybercube-Struktur angeordnet. Mit den 6 Kommunikationsports jedes Routers ergibt sich für diese Verbindungsstruktur ein System mit maximaler Dimension 4, entsprechend 16 Routern oder 64 Prozessoren (Abbildung 2.5). Darüber hinaus können Systeme mit mehr als 64 Prozessoren durch andere Verbindungsstrukturen erhalten werden, etwa einem einfachen hierarchischen Hypercube für 128 Prozessoren oder durch Kopplung in einem Array. Bei dieser Art der Erweiterung steigen Durchmesser und durchschnittliche Entfernung der Prozessoren natürlich stärker an, als das bei einem regulären Hypercube der Fall wäre, was zu Leistungseinbußen führt. Hier werden die Unterschiede in Dimensionierung und Skalierbarkeit zum SP-System deutlich, die auf wesentlich größere Systeme ausgelegt wurde.

Der physikalische Adreßraum der ONYX2 verteilt sich gleichmäßig auf die im System enthaltenen Knoten. Damit sind nach der Konfiguration Lage und Verbindungspfad zu einem referenzierten Datum bekannt. Die Cache-Kohärenz wird durch einen *verzeichnisbasierten* Algorithmus sichergestellt. Zusätzlich zum lokalen Hauptspeicher besitzt jeder Knoten ein Verzeichnis in dem Status und Präsenzbits der einzelnen Seiten vermerkt sind. In den Präsenzbits wird festgehalten, welche Knoten eine Kopie der lokalen Daten besitzen. Bei jedem Cache-Miss wird nun die Anfrage an den Knoten weitergegeben, der den entsprechenden Teil des physikalischen Speichers verwaltet, und der dann, je nach Status und Anfrage reagiert und beispielweise einen weiteren Leser in das Verzeichnis einträgt oder Invalidates an andere Knoten schickt.

Die Verbindungsstruktur der ONYX2 führt zu einem als „non uniform memory access“ (NUMA) bekannten Phänomen: Je nach Lage eines referenzierten Datums, müssen andere Latenzzeiten in Kauf genommen werden. Trotz des gemeinsamen Speichers hängt damit die Leistung paralleler Programme stark von der Verteilung der Prozesse und der Kommunikationsstruktur ab, ein Effekt, der bei bus-basierten Systemen nicht auftritt. So gibt es folgerichtig in der Entwicklungsumgebung auch Programme zur Analyse und Beeinflussung der Plazierung von Prozessen sowie der Seitenallokation bzw. -migration.

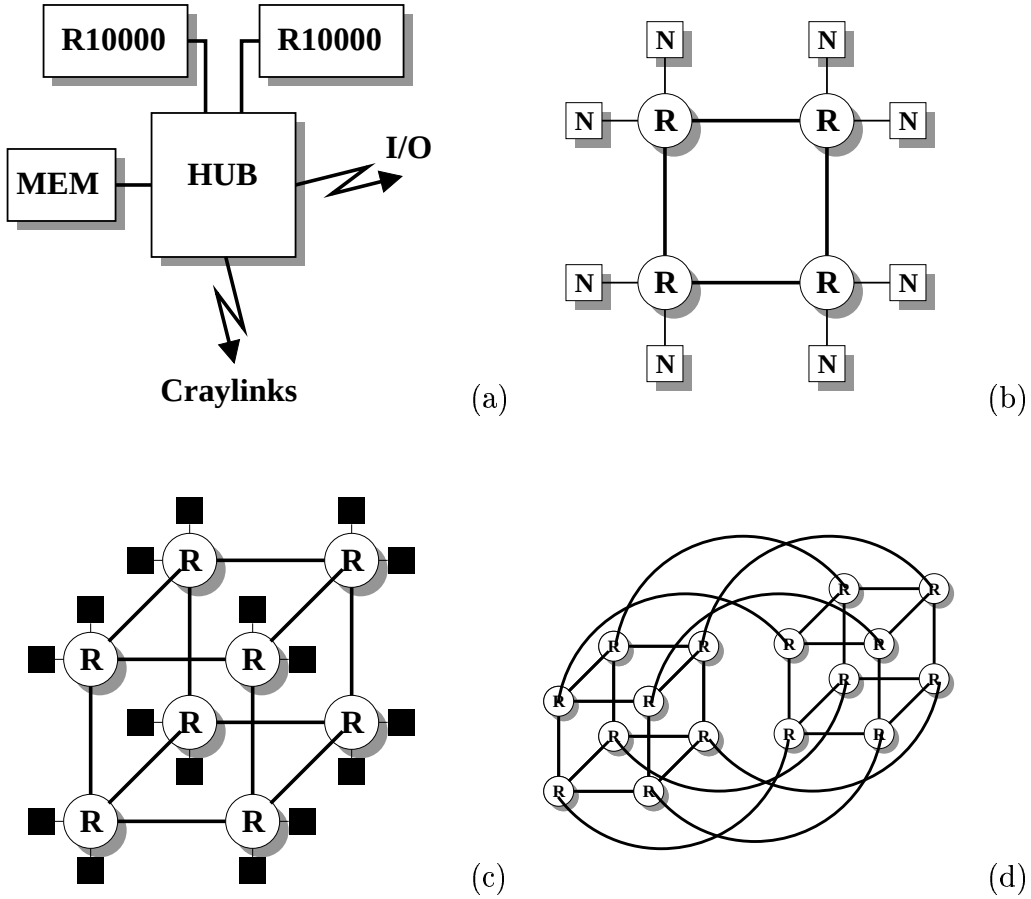

Abbildung 2.5: ONYX2-Systemstruktur. (a) Basisknoten des Systems mit zwei Prozessoren, lokalem Speicher und Ein-/Ausgabeeinheit (b) System mit 16 Rechnern. (c) System mit 32 Rechnern. (d) Maximal-System mit 64 Rechnern.

2.6 Workstation-Netz

Als letzte Architektur soll noch das Workstation-Netz als Beispiel eines bus-basierten Multicomputers angeführt werden (Abbildung 2.6). Im konkreten Fall lagen fünf SUN Sparcstation 10 mit 40-MHz-SPARC-Prozessoren und 64 MByte Hauptspeicher vor, die über 10 MBit-Ethernet verbunden waren.

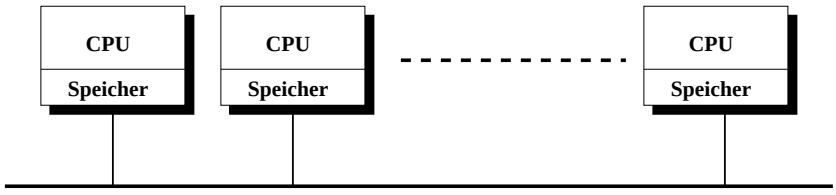

Abbildung 2.6: Bus-basierte Multirechner-Architektur

Kapitel 3

Software-Basis

3.1 Programmierung paralleler Rechner

Zur Programmierung paralleler Rechner gibt es derzeit noch sehr wenige allgemeingültige Techniken. Der Grund dafür liegt in dem enormen Einfluß den die zugrundeliegende Rechnerarchitektur auf die Art der Kommunikation zwischen den parallel rechnenden Einheiten hat. Im Falle von gemeinsamem Speicher wird üblicherweise mit Threads programmiert, bei Systemen mit verteiltem Speicher ist lediglich der kleinste gemeinsame Nenner, der direkte Austausch von Nachrichten (engl. Message-Passing), standardisiert.

Die Programmierung mit Threads auf gemeinsamem Speicher ist die einfachste Art parallel zu programmieren. Das Konzept des Thread entsteht, wenn man im Kontext eines Prozesses nicht einen, sondern mehrere Ausführungsstränge erlaubt. Zur Verwaltung innerhalb dieses Rahmens wird einem Thread nur das nötigste, nämlich Programmzähler, Register und Stack zugestanden. Alle anderen Eigenschaften werden mit den Threads desselben Prozesskontextes geteilt. Die Programmierung beruht auf dem *fork-join-Prinzip*, d.h. ein bereits laufender Thread kann mit Hilfe des *fork*-Primitivs einen weiteren, nebenläufigen Thread erzeugen. Die Vorgehensweise ähnelt einem Prozeduraufruf, mit dem Unterschied, daß während der Ausführung auch der Aufrufer weiterarbeiten kann. Nach Beendigung kann der Erzeuger die beiden Threads mit *join* wieder vereinen. Daß diese Primitive nicht ausreichen wird an einfachsten Beispielen klar, bei denen das Ergebnis der Berechnung von der Reihenfolge abhängt in der die Threads ausgeführt werden. Der Indeterminismus des Ergebnisses, sogenannte *Wettlaufbedingungen* (engl. race conditions) beim Zugriff auf gemeinsam genutzte Speicherbereiche, den *kritischen Regionen*, muß deshalb unter den einzelnen Threads synchronisiert werden. Dazu werden in den meisten Thread-Implementierungen Sperren und Bedingungsvariable, seltener Semaphore, Monitore, Schranken o.ä. verwendet. Sperren realisieren den wechselseitigen Ausschluß von Threads aus kritischen Regionen zumeist mit Hilfe von *aktivem Warten* (engl. busy-waiting). Bei längeren Wartezeiten sollte die Synchronisation deshalb mittels Bedingungsvariablen erfolgen. Ein Thread, der auf den Eintritt einer Bedingung wartet, schläft dabei, ohne CPU-Zeit zu verbrauchen, bis die Erfüllung der Bedingung von einem anderen Thread signalisiert wurde.

Man unterscheidet bei Threads zwischen Implementierungen auf *Benutzer*- und *System*-Ebene. Während die ersteren vom Betriebssystem-Kern völlig unabhängig sind, werden Threads im zweiten Fall explizit vom System unterstützt. Je nach Betriebssystem und Hersteller werden verschiedene Thread-Pakete angeboten, so zum Beispiel die „Leichtgewichtsprozesse“ unter Solaris oder die DCE-Threads der Open Software Foundation. Als Standard sind POSIX-Threads auf fast allen Plattformen erhältlich.
Am gängigsten in verteilten Systemen ist die Programmierung über Nachrichtenaustausch. Leider hat sie auch alle negativen Eigenschaften eines Ansatzes auf der niedrigsten Stufe: Die Programmierung ist fehleranfällig und komplex. Es gibt verschiedene Message-Passing Systeme, so z.B. PVM und MPI [28]. MPI ist dabei zum Standard zu werden. PVM wurde schon Anfang der neunziger Jahre entwickelt und war vor MPI der de-facto Standard. PVM wird als Grundlage des DTS-Systems und Beispiel für ein Message-Passing-System in Abschnitt 3.2 noch genauer beschrieben.
Zwischen diesen beiden extremen Ansätzen werden derzeit noch andere Modelle diskutiert. Recht aktuell ist BSP (engl. bulk synchronous processing) [16], bei dem ein bestimmtes Kommunikationsmuster festlegt wird, das immer noch eine genügend große Klasse von parallelen Applikationen erlaubt, aber durch die Spezialisierung erhebliche Vereinfachungen gegenüber reinem Nachrichtenaustausch zuläßt. Auch gibt es Versuche, eine Brücke zu Systemen mit gemeinsamem Speicher zu schlagen, indem dieser auf verteilten Systemen simuliert wird (Distributed Shared Memory).
In der Praxis birgt die parallele Programmierung noch weitere Stolpersteine. In vielen Anwendungsgebieten fehlen geeignete parallele Algorithmen. Zur Parallelisierung sequentieller Programme müssen vorab alle Datenabhängigkeiten analysiert werden, um parallelisierbare Algorithmen zu finden. Ein theoretisch oft unterschlagener Aspekt ist die Körnung der Parallelität. Bei zu feiner Körnung können Overheads, wie das Starten von Threads oder die Kommunikationkosten, den Gewinn der Parallelisierung neutralisieren. Schließlich wirkt sich die Neuheit und Uneinheitlichkeit des Gebiets auch auf die Entwicklungsumgebungen aus. Es existieren nur sehr wenige Programme zur Laufzeitmessung, zum Profiling oder zur Fehlersuche.
Damit ist bei der Beurteilung paralleler Programme die tatsächliche Gesamtlaufzeit als wichtigster Faktor zu betrachten. Der *Speedup* wird dabei als Verhältnis von paralleler Laufzeit zur Laufzeit des besten sequentiellen Programms definiert. Um eine Aussage über die Nutzung der Prozessoren zu erhalten betrachtet man oft noch die *Effizienz*, d.h. das Verhältnis von Speedup zur Anzahl der verwendeten Prozessoren. Als Optimum kann ein linearer Speedup und damit eine gleichbleibende Effizienz von 1 erreicht werden[1].

3.2 PVM

Ein Beispiel einer Message-Passing-Bibliothek ist PVM (Parallel Virtual Machine) [13]. PVM erlaubt es heterogene Netze von Computern zu einer virtuellen Ma-

[1] Abgesehen von eher pathologischen Fällen superlinearer Speedups, bei denen meist der zu kleine Hauptspeicher die sequentielle Ausführung zusätzlich verlangsamt und damit einen fairen Vergleich verhindert.

schine zusammenzuschließen. Die Kommunikation unter den Prozessen wird explizit über Nachrichtenaustausch programmiert, d.h. die Daten müssen vor dem Versenden in einen Sendepuffer gepackt und auf der Empfängerseite nach der Ankunft wieder ausgepackt werden. Das System garantiert dabei lediglich, daß Nachrichten eines Rechners in der Reihenfolge ihres Versendens bei anderen Rechners empfangen werden und sorgt dafür, daß die möglicherweise unterschiedlichen Datenformate der beteiligten Rechner konvertiert werden. Der Versand und Empfang von Nachrichten kann sowohl synchron als auch asynchron stattfinden.

Darüber hinaus enthält PVM die Möglichkeit der *Gruppenkommunikation*. Dabei wird eine Anzahl von Prozessen in einer Gruppe zusammengefaßt, in der dann mittels einfacheren Primitiven kommuniziert werden kann. Unterstützt werden Gruppennachrichten, Verteilungs- und Sammeloperationen (engl. scatter-gather) und Reduktionen, d.h. die Anwendung einer assoziativen Operation auf Daten die in der Gruppe verteilt sind.

PVM dient zur Zeit als Kommunikations-Basis des DTS-Systems.

3.3 DTS

Herkömmliche Thread-Implementierungen gehen von gemeinsamem Speicher aus. Das „Distributed Threads System“ (DTS) [4, 5] erweitert die Ausführungsmöglichkeit von Threads über die Maschinengrenze hinaus. Dazu geht man vorläufig von seiteneffektfreien, nur auf privaten Daten arbeitenden Threads aus, die dann auf einem beliebigen Rechner ausgeführt werden können, indem einfach alle Eingabe- und Ausgabeparameter versandt bzw. wieder empfangen werden. In seiner bisherigen Form war DTS damit ausschließlich auf datenunabhängige Threads ausgerichtet. Zur Programmierung boten sich vor allem *Master-Slave-* oder *Divide-and-Conquer-*Algorithmen an. Im *Master-Slave*-Fall verteilt ein zentraler Rechner, der *Master*, einzelne, unabhängige Rechnungen auf die *Slaves* im Netz und sammelt die Ergebnisse nach Beendigung der Berechnung wieder ein. Bei Divide-and-Conquer-Algorithmen wird die Arbeit dezentral, d.h. potentiell von jedem beteiligten Rechner, sukzessive in kleinere, wiederum unabhängige Rechnungen aufgeteilt, die vom System verteilt werden. Die Ergebnisse werden dann nach Beendigung wieder an den Aufrufer zurückübertragen. Die Erweiterung auf eine allgemeinere Klasse datenabhängiger paralleler Algorithmen ist in Kapitel 6 beschrieben.

Die automatische Übertragung der Parameter der aufgerufenen Funktionen wird durch einen Codegenerator, `Cgen`, unterstützt, der im Quelltext eingebettete spezielle Kommentare verarbeitet und entsprechende Versende- und Empfangsfunktionen erzeugt.

Weitere nennenswerte Eigenschaften von DTS sind die automatische Lastverteilung, die zur Laufzeit dafür sorgt, daß die vorhandene Arbeit möglichst gleichmäßig auf alle Rechner verteilt wird, und das automatische Recovery, das bei Ausfall einer Maschine die nicht beendeten Threads dieses Rechners neu startet.

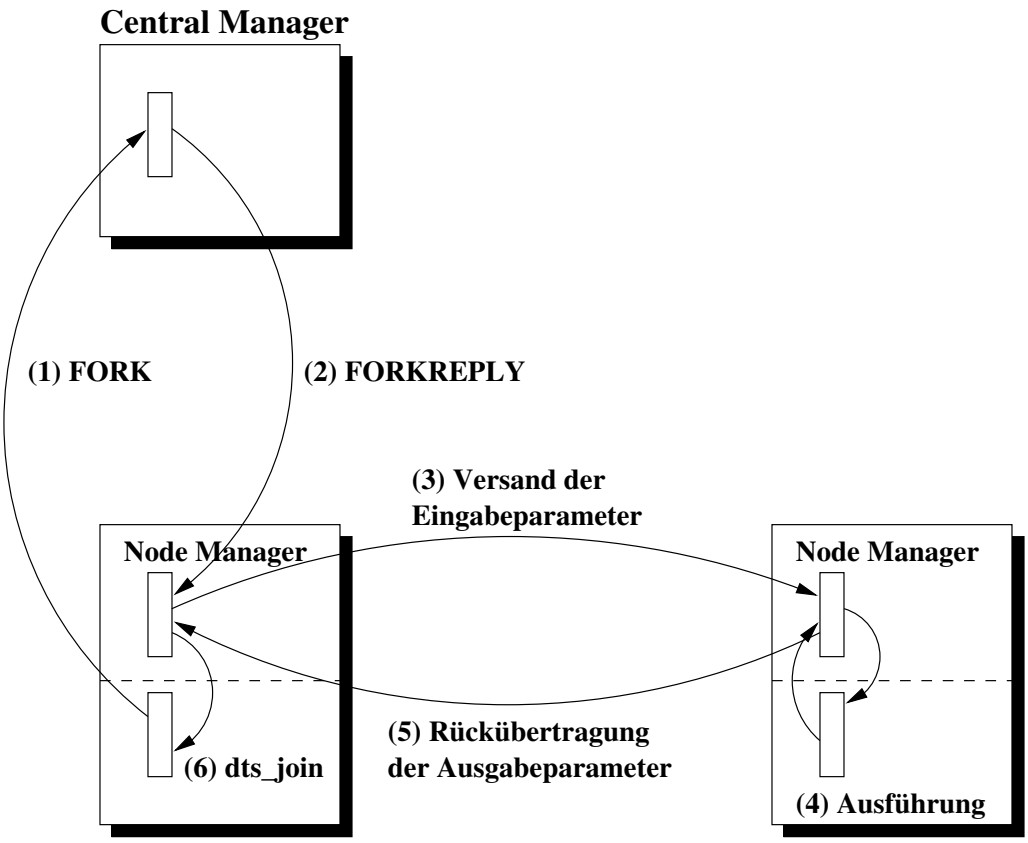

Abbildung 3.1: Thread-Aufruf in DTS

Funktionsweise

Um Threads verteilt auszuführen startet DTS die Applikation auf allen beteiligten Rechnern. Bei der Initialisierung wird dann auf dem Ausgangsrechner der *Central Manager* gestartet, der zentral die Verteilung der Threads verwaltet. Auf den Client-Maschinen läuft jeweils ein *Node Manager*, der dort auf Aufträge wartet.
Beim Start eines Threads mit `dts_fork` sendet der Aufrufer eine Nachricht an den Central Manager. Dieser antwortet, wenn eine freie Maschine zur Verfügung steht, mit deren PVM-Task-Id. Der Node Manager der aufrufenden Maschine sendet dann die Eingabeparameter an den ausführenden Rechner. Dessen Node Manager entpackt die empfangenen Parameter, startet den gewünschten Thread und schickt nach seiner Beendigung die Ausgabeparameter wieder an den Aufrufer zurück. Auf die Ergebnisse kann dort nach einem erfolgreichen `dts_join` zugegriffen werden. Der grundsätzliche Ablauf ist in Abbildung 3.1 nochmals zusammengefaßt.
Um ein Neuaufsetzen von nicht beendeten Threads beim Ausfall eines Rechners zu ermöglichen, sendet jeder Node Manager regelmäßig eine Nachricht, den sog. Heartbeat, an den Central Manager. Fällt dieser, beispielsweise bei einem Rechnerabsturz, über längere Zeit aus, kann der Central Manager den entsprechenden Rechner aus dem System entfernen und die an diesen vergebenen Aufträge neu starten.

Kapitel 4

Physikalisches Modell

Die Berechnung des Faltenwurfs von Textilien ist ein reizvolles und schwieriges Problem der Computergraphik. Der Drang zu immer mehr Realität in computergenerierten Szenen hat das Interesse an solchen Modellen in den letzten Jahren verstärkt. Auch zur Berechnung des Verhaltens von Stoffen im technischen Einsatz oder zur Präsentation von Kleidungsstücken können die Ergebnisse genutzt werden. An der Verwendung eines reduzierten Modells in Echtzeit-Virtual Reality Umgebungen wird gearbeitet. Das vorliegende Partikelsystem wurde am WSI/GRIS der Universität Tübingen entwickelt [9].

4.1 Graphik-Bibliotheken des GRIS

Basis und Schnittstelle zu den graphischen Elementen wie Szenenverwaltung und Ein-/Ausgabe sind die ebenfalls am GRIS entwickelten Graphik-Bibliotheken *RadioLab* und *OBMS* [24, 31]. OBMS (Object-based Modelling System) umfaßt verschiedene Bibliotheken mit C++-Klassen für Punkte, Vektoren und weiteren geometrischen Primitiven, die Verwaltung von Netzen und Oberflächen, Bounding-Boxen sowie Basisklassen zur Ein-/Ausgabe aller Datenstrukturen. Mit RadioLab stehen darauf aufbauend höhere Funktionen zur Verwaltung von Szenen und Beschleunigungsstrukturen und konkrete Rendering-Algorithmen zur Verfügung. OBMS besteht aus etwa 70000 Zeilen C++-Quellcode, RadioLab umfaßt weitere 50000 Quellzeilen. Die hier untersuchte Anwendung verwendet nur einen Teil des gesamten Funktionsumfangs, um auf Szenen zuzugreifen und die notwendigen Schnittpunktsberechnungen durchzuführen.

4.2 Berechnung des Faltenwurfs

Das Modell simuliert die zeitliche Dynamik einer Textilie und ihre Interaktion mit der Umgebung. Der Stoff wird durch ein regelmäßiges rechteckiges Gitter von Massenpunkten modelliert. Die statische Umgebung wird aus verschiedenen geometrischen Primitiven und Dreiecksnetzen zusammengesetzt und liegt als RadioLab-Szenenbeschreibung vor. Das Programm liefert dann sukzessive, in vom Benutzer

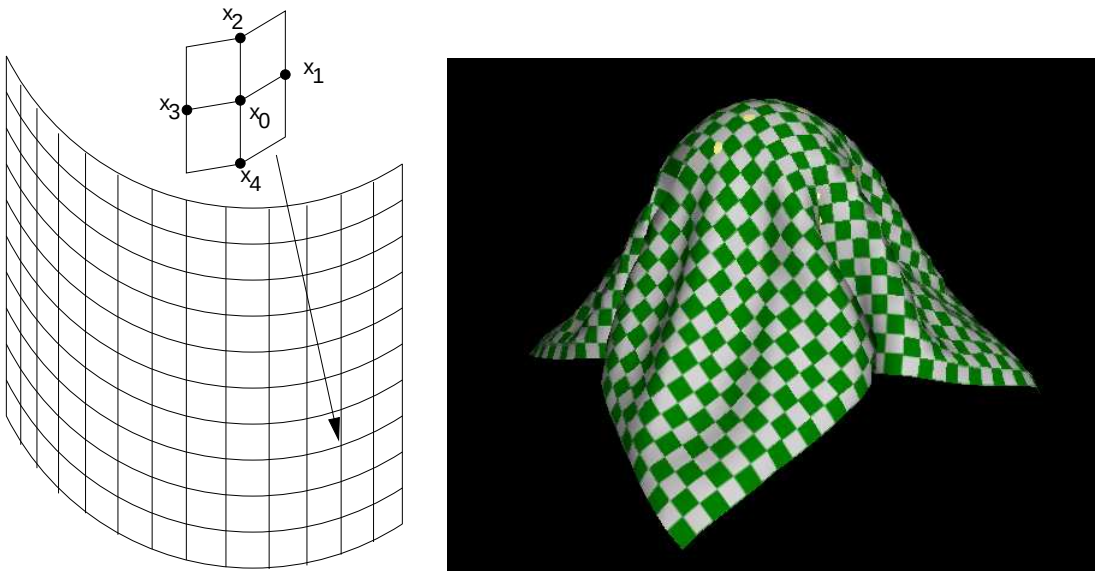

Abbildung 4.1: Links: Gitterstruktur der Textilie. Rechts: Beispielszene mit Tuch und Kugel

vorgebbaren Zeitintervallen, die Zustände der Textilie, die unter dem Einfluß der Schwerkraft „auf die Szene fällt“.
Zur Berechnung werden die Wechselwirkungen jedes Partikels mit den vier ihn umgebenden Nachbarn berücksichtigt (siehe Abbildung 4.1 links). Speziell zur Modellierung von Textilien müssen auftretende Zug- und Druckkräfte sowie die Auswirkung von Scherung und Biegung des Stoffes berücksichtigt werden. Scher- und Biegekräfte zeigen zudem Hystereseeffekte, die ebenfalls in das Modell einfließen, und bei konkreten Stoffen vorher durch standardisierte Meßverfahren ermittelt werden können. Weiterhin sind äußere Einflüsse wie Schwerkraft, Wind und Luftwiderstand zu berücksichtigen.
Die Bewegungsgleichungen der einzelnen Partikel erhält man über den Lagrangeschen Ansatz zur Beschreibung von Vielteilchensystemen. Für die Trajektorien ergibt sich dadurch ein Differentialgleichungssystem 2. Ordung. Dieses wird dann mit Hilfe von Differentialgleichungslösern numerisch berechnet. Schließlich muß auch noch die Interaktion des Stoffes mit sich selbst und der vorliegenden Szene berechnet werden, d.h. nach jeder Iteration werden Schnittpunktsberechnungen zur Kollisionsdetektion durchgeführt.

4.3 Algorithmen

Das Modell liegt als C++-Klassenbibliothek vor. Eine Übersicht über die Klassenhierarchie ist in Abbildung 4.2 zu finden. Es kann in seiner allgemeinen Form auch zur Berechnung anderer Systeme, etwa Galaxien oder Minimalflächen eingesetzt werden. Für die Berechnung des Faltenwurfs waren lediglich drei Klassen von Interesse:

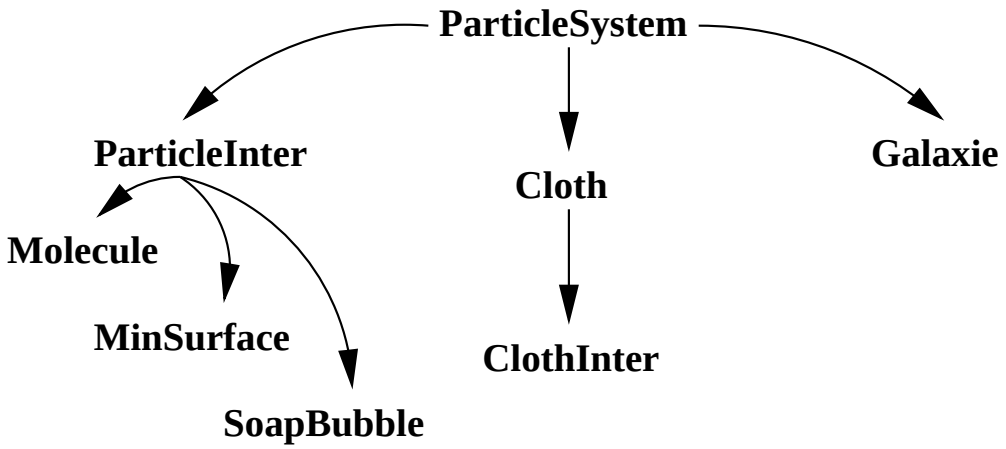

Abbildung 4.2: Klassenhierarchie des Partikelsystems

Die abstrakte Basisklasse `ParticleSystem`, die ein Ensemble von Partikeln mit beliebigen Wechselwirkungen modelliert, sowie die Klassen `Cloth` und `ClothInter`, die Textilien ohne bzw. mit Wechselwirkung zur Umgebung umsetzen.
Die Verwendung der Objekte wird durch folgendes Listing deutlich. Um das Verhalten eines Tuchs zu berechnen, wird ein Objekt vom Typ `ClothInter` erzeugt, das mit der Anfangskonfiguration des Tuchs, dem zu verwendenden Material und der Szeneninformation initialisiert wird (Zeile 7). Nach der Erzeugung des Objektes können noch verschiedene Parameter, im Beispiel der Typ des Lösers und die Startzeit der Berechnung in den Zeilen 9 und 10, gesetzt werden. Die Funktion `solveOde` erlaubt es dann die Dynamik des Tuchs zu berechnen. Dazu wird als einziger Parameter die zu berechnende Zeitspanne übergeben. Mit `writeCloth` kann schließlich der aktuelle Zustand des Systems in Form einer Netzbeschreibung ausgegeben werden. Für Animationen iteriert man diesen Vorgang, wie im Beispiel, mit relativ kleinen Zeitspannen. Ist man lediglich am Endergebnis interessiert, so genügt ein einziger Aufruf von `solveOde` mit dem zu berechnenden Zeitintervall.

```
1  int main(void) {
2    Mesh3D startmesh;
3    materialType material=cotton;
4    float timestep=0.005;
5
6    startmesh.read("20.mesh");
7    ClothInter cloth(startmesh, material, "umgebung.scene");
8
9    cloth.setSolverType(use_rkqs);
10   cloth.setStartTimes(0.001);
11
12   for (int i=0; i<1000; i++) {
13     cloth.solveOde(timestep);
14     sprintf(name,"simul.mesh.%3.3i",i);
15     cloth.writeCloth(name);
```

```
16    }
17  }
```

Die Methode `ParticleSystem::solveOde` ist die wesentliche Schnittstelle zum Benutzer. Eine – der Übersichtlichkeit halber vereinfachte Version – illustriert die einzelnen Lösungsschritte.

```
1   float ParticleSystem::solveOde(float min_time) {
2     float timeTry=startTimes, timeDid=startTimes, timeNext=startTimes;
3
4     currentPS=this;  // set current system for non-member getNRdydx
5
6     for(localTime=0.0; localTime<min_time; /* NoOp */) {
7       setZeroVelo();
8       switch(solverType) {
9
10      case use_rkqs:
11        getNRdydx(simulationTime, IValues-1, dydx-1);
12        rkqs(EValues-1, dydx-1, 6*NoNodes, &simulationTime, timeTry,
13             finalTolerance , scaleVector-1, &timeDid, &timeNext,
14             getNRdydx);
15        break;
16
17      default:
18        cerr << "Fatal error: solverType not defined!" << endl;
19      }
20
21      localTime+=timeDid;
22      if(localTime+timeNext>min_time) timeTry=min_time-localTime;
23      else timeTry=timeNext;
24
25      dointersection(timeDid);     // Durchdringungsrechnung
26
27      adjust_coefficients();       // Hysterese-Effekte
28
29      // Setzen der neuen Anfangswerte
30      for (int i=0; i<6*NoNodes; i++) IValues[i]=EValues[i];
31    }
32    timeLastOdeInvo = localTime;
33
34    return localTime;
35  } // end of solveOde
```

`solveOde` besteht aus den drei, bereits oben erwähnten Schritten: Lösung der Differentialgleichungen, Durchdringungsrechnung und Anpassung der Kraftkoeffizienten.

Lösung der Differentialgleichungen

Die Lösung der Differentialgleichungen wird in den Zeilen 7–20 angestoßen. Als Löser werden Standard-Zeitintegratoren für nicht-steife gewöhnliche Differential-

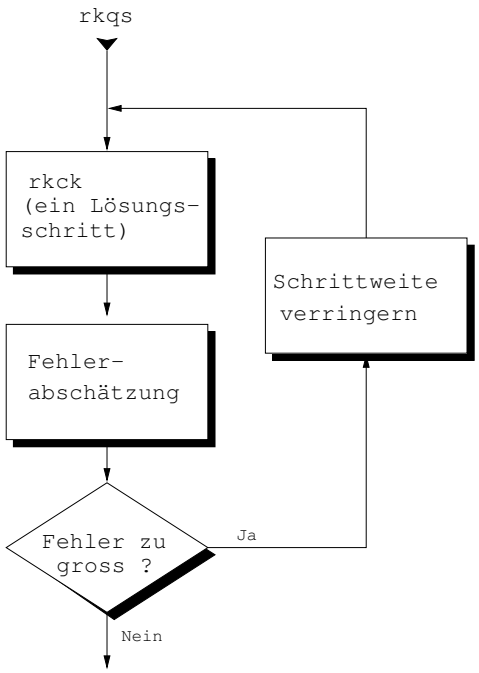

Abbildung 4.3: Flußdiagramm `rkqs`

gleichungssysteme mit Anfangswert aus der mathematischen Algorithmensammlung Numerical Recipes [29] verwendet. Im Beispiel wird die Treiberfunktion `rkqs` benutzt. Sie greift auf `rkck`, einen Runge-Kutta-Löser vierter Ordnung zurück, und ruft diesen wiederholt mit kleiner werdender Schrittweite auf, bis der Gesamtfehler die gewünschte Toleranz nicht mehr überschreitet (Abbildung 4.3). Zur Berechnung der rechten Seiten des Differentialgleichungssystems, d.h. der Ableitungen der Koordinaten und Geschwindigkeiten der Partikel, wird die Funktion `getNRdydx` verwendet, die ihrerseits indirekt die Methode `ParticleSystem::accelerInter` aufruft. Dort findet die eigentliche Auswertung der aktuellen Parameter statt. Der C-Code zur Berechnung der Ableitungen wurde automatisch mit Hilfe des Computeralgebra-Systems MAPLE generiert.

Kollisionsdetektion durch Schnittberechnung

Bei der Interaktion des Tuchs mit der statischen Szene sowie mit sich selbst kommen die Beschleunigungsstrukturen der GRIS-Bibliotheken zum Einsatz. Bounding-Box-Tests und Szenengitter tragen dazu bei den Aufwand der Schnittberechnung zu reduzieren.
Der erste Teil der Schnittberechnung testet die Bewegung der Partikel auf Kollision mit der Szene. Die Auswertung findet in der Methode `ParticleSystem::intersection_with_environment` statt. Dort wird das aktuelle Segment jeder Trajektorie auf Schnitt mit den in der Szene enthaltenen Objekten überprüft. Durch die Verwaltung der Graphikszenen mit RadioLab reduziert sich der Aufwand auf einen

einfachen Aufruf der Methode `isIntersected` des Umgebungsobjekts.
Für die Schnittberechnung des Tuchs mit sich selbst muß die Bewegung jedes Punktes auf einen möglichen Schnitt mit Netzelementen des Tuches geprüft werden. Die Schwierigkeit liegt darin, daß sich die Netzelemente selbst natürlich auch bewegen. Konkret werden zuerst in der Methode `findSelfInterCandidates` alle möglichen Paare sich schneidender Elemente bestimmt und in einem zweiten Schritt mit `checkSelfInterMeshEls` diese Paare eingehend auf Schnitt untersucht.
Aus beiden Schnittberechnungen erhält man Informationen über einen möglichen Schnittpunkt. Schließlich werden die Partikelorte und -geschwindigkeiten entsprechend verändert, d.h. der neue Ort des Partikels ist der erhaltene Schnittpunkt und sein Geschwindigkeitsvektor ergibt sich aus dem vorherigen durch einen elastischen Stoß mit Oberflächenreibung.

Neuberechnung der Kraftkoeffizienten

Da die Kraftkoeffizienten eine nicht-lineare Abhängigkeit (siehe Beispiel in Abbildung 4.4) von der jeweiligen Auslenkung sowie Hystereseeigenschaften zeigen, müssen diese nach jedem Schritt in Abhängigkeit von der aktuellen Position und der Vorgeschichte neu bestimmt werden. Dazu werden in `adjust_coefficients` die aktuellen Werte neu berechnet, mit den bisherigen Minimal- und Maximalauslenkungen verglichen und damit die aktuelle Position auf der Hysteresekurve ermittelt. Der Endwert ergibt sich dann als lineare Interpolation zwischen den in 1-Grad-Schritten vorgegebenen Stützstellen der Hysteresekurve.

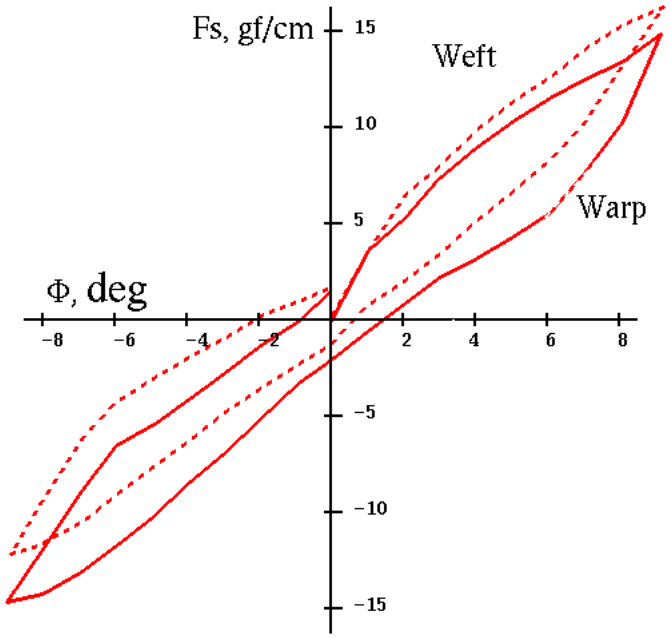

Abbildung 4.4: Winkelabhängigkeit der Biegekräfte. Bei anisotropen Materialien unterscheiden sich die Abhängigkeiten in Kettrichtung (engl. Weft, gestrichelt) und Schußrichtung (engl. warp, durchgezogen).

Kapitel 5

Parallelisierung

5.1 Vorbetrachtungen

Ausgehend von der algorithmischen Betrachtung des letzten Kapitels ergeben sich zwei Ansatzpunkte für eine Parallelisierung: Die Lösung der Differentialgleichungen sowie die Schnittberechnung der Textilie mit sich selbst und der Umgebung.

Parallele Lösung von Differentialgleichungen

Im Gegensatz zu vielen Algorithmen aus der linearen Algebra liegen im Bereich der parallelen numerischen Lösung von Differentialgleichungen noch sehr wenige Ansätze oder Algorithmen vor. Dies gilt selbst im vorliegenden, eingeschränkten Fall, dem der Lösung eines Anfangswertproblems eines System gewöhnlicher Differentialgleichungen

$$y'(t) = f(t, y), \quad y(t_0) = y_0 \qquad t \in \mathbb{R}, y \in \mathbb{R}^n \quad .$$

Man unterscheidet im wesentlichen drei Vorgehensweisen: Die Parallelisierung nach der Zeit, der Methode oder dem Problem [6, 32].
Die Lösung eines Anfangswertproblems nach der Zeit birgt keine natürliche Parallelität. Obwohl Ansätze mehrere Zeitschritte parallel auszuwerten existieren (Waveform-Relaxations-Methoden), liegen keine konkreten Algorithmen vor.
Die Parallelisierung nach der Methode erfordert eine eingehende Auseinandersetzung mit dem Lösungsverfahren. Auch in diesem Gebiet gibt es nur sehr wenige erfolgversprechende Ansätze. Viele Verfahren enthalten Datenabhängigkeiten, die eine parallele Ausführung verbieten. Beispielsweise können explizite k-stufige Runge-Kutta-Verfahren zwar in parallele Blöcke aufgeteilt werden, verlieren dabei aber gleichzeitig an Fehlerordnung, d.h. an Genauigkeit. Die parallele Ausführung verspricht daher keinen Gewinn, da für vergleichbare Ergebnisse die Schrittweiten entsprechend reduziert werden müssen. Andere Verfahren, wie die Methode von Bulirsch-Stoer, bei dem über einem gegebenen Intervall mehrere Einschrittverfahren mit abnehmender Schrittweite durchgeführt werden, deren extrapolierte Einzelergebnisse dann zu einer genaueren Näherung führen, können in beschränktem Umfang parallelisiert werden. Sie leiden jedoch unter einer sehr feinen Körnung und einer durch den Algorithmus bedingten, meist sehr niedrigen, Obergrenze der Parallelität. Auch können sie nur

bei Rechnern mit gemeinsamem Speicher eingesetzt werden, da die einzelnen parallel ausgeführten Teile jeweils auf dem gesamten Problem arbeiten. Ab einer gewissen Problemgröße macht die dafür erforderliche Kommunikation eine Parallelisierung auf Multicomputern unrentabel.
Als einfachste und erfolgversprechendste Lösung bietet sich die Parallelisierung nach dem Problem an. Dabei können in Systemen der Dimension n die „rechten Seiten", also die Funktion f, von maximal n Prozessoren gleichzeitig ausgewertet werden. Diese Methode hat die Vorteile, daß das konkrete Problem leicht auf die Zahl der vorhandenen Prozessoren aufgeteilt werden kann, und der Ansatz weitgehend unabhängig von der Wahl des Differentialgleichungslösers ist. Da die Variablen des Systems meist über die rechten Seiten gekoppelt sind, erfordert eine Parallelisierung nach dem Problem allerdings eine ständige, der Gleichungsstruktur entsprechende, Synchronisation zwischen den beteiligten Rechnern.

Parallele Kollisionsdetektion

Die Kollisionsdetektion der Textilie mit der Umgebung läßt sich auf den ersten Blick ohne weiteres parallel durchführen. Dabei treten jedoch zwei Probleme auf, der Zugriff auf interne Datenstrukturen der verwendeten Graphik-Bibliotheken bei Parallelisierung mit gemeinsamem Speicher und die potentielle Unregelmäßigkeit der Kollisionen. Bei der Durchführung einer Untersuchung auf Kollision werden interne Datenstrukturen der verwendeten Strahlen und Szenen modifiziert, so daß eine naive Parallelisierung zu *Wettlaufbedingungen* führt. Das Problem läßt sich umgehen, indem man alle internen Modifikationen vor der parallelen Ausführung durchführt und sicherstellt, daß die Datenstrukturen während der Schnittberechnung nicht mehr verändert werden.
Das zweite Problem hängt mit der Aufteilung der Textilie auf verschiedene Prozessoren zusammen. Im Normalfall, wenn keine Schnitte auftreten, führt jeder Prozessor für alle Partikel einen Bounding-Box-Test durch. Die parallele Ausführung ergibt dann leider auch keinen hohen Gewinn, da dieser Test nur aus wenigen Instruktionen besteht. Treten dagegen Schnitte auf, sind diese im allgemeinen nicht gleichverteilt. Schon die Einführung leichter Störungen, um die berechneten Szenen realistischer zu machen, ist eine Ursache dafür. Damit stellt sich beim Auftreten von Schnitten oft ein Ungleichgewicht ein, so daß die Parallelisierung nur wenig Gewinn bringen kann. Diese Schwierigkeit kann durch andere Zuteilungen nicht umgangen werden, da die auftretenden Kollisionen nicht im voraus bekannt sind.
Bei der Berechnung der Kollisionen der Textilie mit sich selbst treten weitere Probleme auf. Der Algorithmus enthält globale Datenabhängigkeiten, da potentiell jedes Netzelement sich mit jedem anderen durchdringen kann. Im Falle gemeinsamen Speichers müssen Wettlaufbedingungen durch zusätzliche Synchronisation vermieden werden. Bei verteilter Berechnung verhindern diese Abhängigkeiten eine effiziente Parallelisierung. Man befindet sich dort in einem grundsätzlichen Dilemma bei der Wahl der Zuteilung der Partikel zu den einzelnen Prozessoren. Soll die Textilie topologisch, d.h. nach ihrer Zusammenhangsstruktur, oder geometrisch, d.h. nach der aktuellen Raumstruktur zugewiesen werden? Der zweite Weg löst – bis auf Randeffekte – das oben erwähnte Problem der Datenabhängigkeiten. Leider muß das

mit einer veränderlichen Zuteilung von Partikeln zu Prozessoren erkauft werden, d.h. nach jeder Iteration muß zusätzlich ein globaler, aufwendiger Zuweisungsalgorithmus durchgeführt werden. Zudem sind die Ränder nicht mehr minimal, wie bei einer topologischen Aufteilung, was sich wiederum negativ auf die Leistung des parallelen Differentialgleichungslösers auswirkt.
Da der Weg der geometrischen Aufteilung schon in einer anderen Arbeit gegangen wurde [20] und aufgrund der genannten Nachteile, die in diesem Fall einen Gewinn eher in Frage stellen, wurde hier die topologische Zuteilung gewählt.

Sonstige Parallelität

Der Rückgriff auf die Graphik-Bibliotheken entzieht einen Teil des Programms einer weiteren Parallelisierung, da diese nach Umfang und Komplexität zu groß waren, um in dieser Diplomarbeit auf parallelisierbare Programmteile oder alternative parallele Algorithmen untersucht werden zu können. Die restlichen Programmteile boten noch einen geringen Ansatz zur Parallelisierung, hauptsächlich nach dem Problem, d.h. durch Aufteilung der Partikel auf die Prozessoren. Der Struktur nach waren dies zumeist sehr einfache Berechnungen, die bei der Implementierung auf gemeinsamem Speicher teilweise genutzt werden konnten, bei verteiltem Speicher jedoch viel zu feinkörnig waren, um einen Leistungsgewinn zu erzielen.

5.2 Programme und Messungen

In den folgenden zwei Kapiteln werden die verschiedenen Parallelisierungen des Partikelsystems auf Rechnern mit gemeinsamem Speicher und auf verteilten Systemen beschrieben. Der Grund für die Programmierung zweier unterschiedlicher Versionen liegt in dem geringen Potential an Parallelität dieser Anwendung. Um den Leistungsgewinn möglichst groß zu halten, mußte die jeweilige Zielarchitektur durch spezielle Programmierung ausgenutzt werden. Zudem ist DTS noch nicht allgemein genug, um eine gemeinsame Version in beiden Fällen einzusetzen ohne wesentlich an Geschwindigkeit zu verlieren.
Am Ende des jeweiligen Kapitels finden sich für jede untersuchte Architektur die Messungen und Analysen der parallelen Ausführungszeiten. Zum Vergleich mit der parallelen Implementierung und der Abschätzung des sequentiellen Anteils wurden vorab Messungen und Profiling-Läufe mit der sequentiellen Version des Programms durchgeführt. Dabei bestätigten sich in fast allen Fällen die Lösung der Differentialgleichungen und die Schnittberechnung als grundsätzliche Ansatzpunkte für eine Parallelisierung.
Die Auswertungen der parallelen Programme wurden mit jeweils drei Konfigurationen durchgeführt:

A Quadratisches Tuch mit 20×20 Partikeln fällt über eine Kugel

B Quadratisches Tuch mit 52×52 Partikeln fällt über einen Tisch

C Poncho (2056 Partikel) fällt über eine Kleiderpuppe.

Funktion	Erläuterung
`accelerations`	Berechnung der Beschleunigungen zur Lösung der DGL
`intersection_with_environment`	Kollisionsdetektion mit der Umgebung
`calc_new_grids`	Neuberechnung der internen Datenstrukturen der Szenen
`findSelfInterCandidates`	Parallelisierbarer Teil der Kollisionsdetektion der Textilie mit sich selbst
`adjust_coefficients`	Neuberechnung der Hysteresekoeffizienten

Tabelle 5.1: Parallelisierte Funktionen bei gemeinsamem Speicher

Die verwendete Schrittzahl wurde abhängig von der speziellen Architektur ausgewählt um die Laufzeiten bei der Messung innerhalb vertretbarer Grenzen zu halten, die immer noch groß genug waren, um aussagekräftige Meßergebnisse zu erhalten.

5.3 Gemeinsamer Speicher

Im Falle des gemeinsamen Speichers wurden sämtliche Algorithmen parallelisiert, die eine Bearbeitung für alle Partikel erforderten. Dabei war allerdings festzustellen, daß in vielen Fällen die Körnung der Parallelität so gering ist, daß kaum Leistungssteigerungen erzielt werden konnten. Konkret blieben die Auswertung der rechten Seiten der Differentialgleichungen, die Kollisionsdetektion mit der Szene, Teile der Selbstdurchdringungsrechnung und die Anpassung der Hysteresekoeffizienten als gewinnbringende Routinen übrig. Tabelle 5.1 faßt die parallelisierten Programmteile und die zugehörigen Funktionen nochmals zusammen.
Da bei gemeinsamem Speicher der Overhead einer Gabelung normalerweise sehr gering ist, wurde nur die Berechnung der Beschleunigungen parallelisiert, die von den Differentialgleichungslösern verwendet wird. Das hat den entscheidenden Vorteil, daß die Löser nicht umgeschrieben werden müssen, d.h. weiterhin alle zur Verfügung stehenden Löser verwendet werden können.
Die Auswertung der rechten Seiten der Beschleunigungen mußte geringfügig modifiziert werden, um Wettlaufbedingungen zu vermeiden. Datenabhängigkeiten ergaben sich an drei Stellen, der Initialisierung des Ergebnis-Felds, bei der eigentlichen Berechnung, wo mehrere Threads sowohl lesend als auch schreibend auf dasselbe Element zugreifen, und bei der abschließenden Nachverarbeitung der Beschleunigungen. Die Initialisierung und Nachbehandlung wurden von der parallelen Bearbeitung ausgeschlossen, da die geringe Bearbeitungszeit für eine Parallelisierung nicht lohnenswert war, und gleichzeitig die Abhängigkeiten damit beseitigt werden konnten. Die Überlappungen bei der Berechnung wurden dadurch umgangen, daß bei der parallelen Implementierung die einzelnen Threads die Beschleunigungen in separaten Ergebnisräumen ablegen. Die verschiedenen Ergebnisse werden nach Ende der

Berechnung aufsummiert. Die Summation selbst kann wiederum parallel ausgeführt werden. Durch die Aufteilung mußte leider ein zusätzlicher Overhead in Kauf genommen werden.
Bei der Schnittberechnung greift man auf Bibliotheksfunktionen des RadioLab zurück, die beim Aufbau der Raumgitter zur Beschleunigung der Schnittberechnung interne Objektstrukturen verändern, was zu Wettlaufbedingungen führen kann. Der Gitteraufbau mußte deshalb vorab erfolgen.
Die Szenenaktualisierung konnte ohne Eingriff in die Basisbibliotheken nicht parallelisiert werden. Da jedoch in jeder Iteration zwei Szenengitter neu berechnet werden müssen, wurde versucht, wenigstens diesen Schritt mit zwei Threads parallel durchzuführen.
Alle anderen Routinen enthielten keine Datenabhängigkeiten und konnten ohne weitere Modifikationen parallelisiert werden.
Die eigentliche Implementierung erfordert dann lediglich, die Berechnung für alle Partikel auf die vorhandenen Prozessoren aufzuteilen, d.h. im Wesentlichen werden Schleifen parallelisiert. Dazu wurde eine Klasse `ParClothInter` von `ClothInter` abgeleitet, deren Benutzerschnittstelle mit `ClothInter` identisch ist. Programme, die auf dem Partikelsystem aufbauen, können also durch den Wechsel zur Klasse `ParClothInter` parallel ausgeführt werden.
Da die Verwendung der Löser aus den Numerical Recipes den Zugriff von Nicht-Mitgliedsfunktionen auf Objektdaten erfordert, der mittels eines statischen Zeigers auf das aktuelle Objekt realisiert ist, kann bei paralleler Ausführung jeweils nur ein Objekt des Typs `ParClothInter` verwendet werden. Bei der Initialisierung wird dies überprüft und danach die Zuteilung der Partikel zu den Prozessoren berechnet und in dem internen Feld `start_node` gespeichert. Die Parallelisierung einer Schleife der Form

```
for(i=0; i<NoNodes; i++) do(i);
```

reduziert sich dann auf die Transformation nach

```
for(i=0; i<NoProcs; i++) fork_do_wrapper(i);
for(i=0; i<NoProcs; i++) join_do_wrapper(i);

/*pure*/
void do_wrapper(int i) {
  for(k=start_node[i]; k<start_node[i+1]; k++) do(k);
}
```

Dabei werden die Makros `fork_do_wrapper` und `join_do_wrapper` vom Codegenerator aufgrund der `pure`-Deklaration automatisch erzeugt (siehe Kapitel 7).

5.3.1 SGI ONYX2

Vorarbeit

Nach dem Programmierhandbuch der ONYX2 [10] gibt es lediglich zwei Möglichkeiten der parallelen Programmierung auf gemeinsamem Speicher für diese Architektur: Die Verwendung von POSIX-Threads und der IRIX-spezifische `sproc()`-

Mechanismus, bei dem mehrere Prozesse sich denselben Adreßraum teilen. Die bisherige Implementierung basierte auf diesem Mechanismus. Im Zuge dieser Diplomarbeit wurde DTS zusätzlich auf die Basis der POSIX-Threads portiert. Zeitmessungen des Gabelungs-Overheads ergaben einen eindeutigen Vorteil für die Verwendung der Threads, der `sproc()`-Mechanismus war um ein bis zwei Größenordnungen langsamer.
Leider nimmt man mit den POSIX-Threads andere Nachteile in Kauf. Am schwerwiegendsten ist die hohe Anzahl der zu verwendenden Threads für den maximalen Geschwindigkeitsgewinn. Bei acht Prozessoren sollten im Prinzip acht Threads das Optimum der Laufzeit ergeben können. Die Implementierung verwendet jedoch ein n-über-m-Schema, d.h. n Threads werden auf $m < n$ Prozesse aufgeteilt. Dabei ist die Anzahl der verwendeten Prozesse nur indirekt, d.h. über die Zahl der erzeugten Threads, beeinflußbar. Damit ergeben sich zwei zusätzliche Quellen für Leistungsverluste. Zum einen steigt der parallele Overhead für $n > 8$ Threads stärker als nötig an, zum anderen kann die Zuteilung von Threads zu den Prozessoren nicht beeinflußt werden, was bei dem zugrundeliegenden Hardware-DSM zu höherer Kommunikation führt.
Messungen mit einem einfachen, rein parallelen Programm zeigen, daß das Geschwindigkeitsoptimum bei etwa 40 Threads liegt. Die dabei gemessene Beschleunigung von 7.5 ist bei acht Prozessoren nahezu optimal und wird nur durch einen geringen parallelen Overhead geschmälert. Erhöht man die Körnung des Testprogramms, indem mehr Gabelungen bei gleichbleibender Laufzeit durchgeführt werden, erhält man deutlichere Overhead-Effekte. Die optimale Beschleunigung reduziert sich und wird bereits bei geringeren Thread-Zahlen erreicht.
Aufgrund dieser Thread-Implementierung konnte die Parallelität der Szenenaktualisierung nicht ausgenutzt werden, da dort nur zwei Threads eingesetzt werden können. Eine tatsächliche parallele Ausführung ist damit nicht sichergestellt. Versuche mit der parallelen Programmversion zeigten, daß bei Ausnahme der Szenenaktualisierung von der parallelen Ausführung die Beschleunigung im Vergleich zur parallelen Ausführung aller Programmteile immer leicht größer war. Also wurden beide Threads auf einem Prozessor ausgeführt und nur der Overhead stieg an.

Messungen

Die Laufzeitmessungen wurden mit den in der Einleitung angegebenen drei Konfigurationen durchgeführt. Jede Konfiguration wurde mit vier verschiedenen Anzahlen von Berechnungsschritten getestet und dabei jeweils die Laufzeit der sequentiellen und der parallelen Programmversion gemessen. Außerdem wurde die Anzahl der Threads in sieben Stufen im Bereich 1-40 variiert. Die Meßergebnisse sind in Tabelle 5.4 zu sehen. Für jede Konfiguration und Schrittzahl sind jeweils die Gesamtlaufzeiten und die Beschleunigungen angegeben.
Für die noch folgende Analyse wurden Profiles der sequentiellen Programmversion erstellt. In Abbildung 5.1 sind die prozentualen Anteile an der Gesamtlaufzeit für die in Tabelle 5.1 beschriebenen parallelisierten Funktionen dargestellt.
Um den Overhead der parallelen Programmversion zu ermitteln, wurde zusätzlich für die jeweils größte Konfiguration die Laufzeit bei verschiedenen Thread-Zahlen

A_{200}	Anzahl Threads							
	Seq.	1	5	10	16	24	32	40
t [s]	491.84	493.61	370.81	318.55	258.61	340.85	315.67	318.69
S	1287	1204	1169	1173	1099	1356	1209	1169
t_{norm}[s]	491.84	527.64	408.24	349.51	302.85	323.51	336.04	350.86

Tabelle 5.2: Normierung der Laufzeitwerte. Mit t sind die Laufzeiten, mit S die Anzahl der Lösungsschritte bezeichnet. Die gemessenen parallelen Laufzeiten werden aufgrund der unterschiedlichen Zahl der Lösungsschritte auf die Laufzeit der sequentiellen Version normiert. Es gilt $t_{\text{norm}} = t \cdot S_{\text{parallel}}/S_{\text{sequentiell}}$.

unter Beschränkung auf einen Prozessor gemessen. Die Overheads sind in Tabelle 5.5 zu finden, dort sind jeweils die Laufzeiten sowie die Differenz zur entsprechenden sequentiellen Laufzeit, d.h. der reine Overhead, angegeben.

Bei der Untersuchung der Laufzeiten ergab sich eine weitere Schwierigkeit. Die gemessenen Laufzeiten sind nicht direkt vergleichbar, weil die Anzahl der Lösungsschritte mit der Anzahl der verwendeten Threads schwankt. Die Abweichungen sind Resultat einer geänderten Ausführungsreihenfolge, wodurch sich die Ergebnisse durch Rechenfehler leicht unterscheiden. Diese verstärken sich im Laufe der iterativen Lösung der Differentialgleichungen. Der Effekt war bereits bei der sequentiellen Ausführung zu beobachten, wenn man die Ausgaben optimierter und nichtoptimierter Programme verglich, und tritt bei der parallelen Ausführung verstärkt auf, da sich dort systembedingt die Berechnungsreihenfolge von Lauf zu Lauf ändert. Um dennoch einen aussagekräftigen Vergleich durchzuführen zu können, sind alle angegebenen parallelen Laufzeiten mit Hilfe der tatsächlich ausgeführten Anzahl von Lösungsschritten auf den Referenzwert der sequentiellen Ausführung normiert worden. Das Vorgehen wird in Tabelle 5.2 deutlich, wo an einem Beispiel der Konfiguration A die Anzahl der Lösungsschritte sowie die tatsächlich gemessenen und normierten Laufzeiten gegenübergestellt sind.

Analyse der Messungen

Aus den Profiling-Ergebnissen (Abbildung 5.1) erkennt man, daß bei der Szenenaktualisierung (Funktion `calc_new_grids`) ein nicht unerheblicher sequentieller Anteil zurückbleibt. Tabelle 5.3 faßt die darauf beruhenden Erwartungswerte der Beschleunigungen für die ONYX2 zusammen.

Die tatsächlich erzielten Beschleunigungen liegen in allen drei Fällen unter den Erwartungswerten aus Tabelle 5.3. Man erkennt allerdings aus den Overhead-Messungen, daß bei der parallelen Ausführung ein zusätzlicher, mit wachsender Zahl von Threads steigender Zeitverlust hinzukommt. Diese Overheads und eine lineare Interpolation sind für die drei Fälle in der linken Spalte von Abbildung 5.2 zu sehen.

Um die Overheads mit einzubeziehen und die gemessenen Werte zu erklären, läßt sich das Rechenmodell für die Beschleunigung bei n Prozessoren und einer Gesamt-

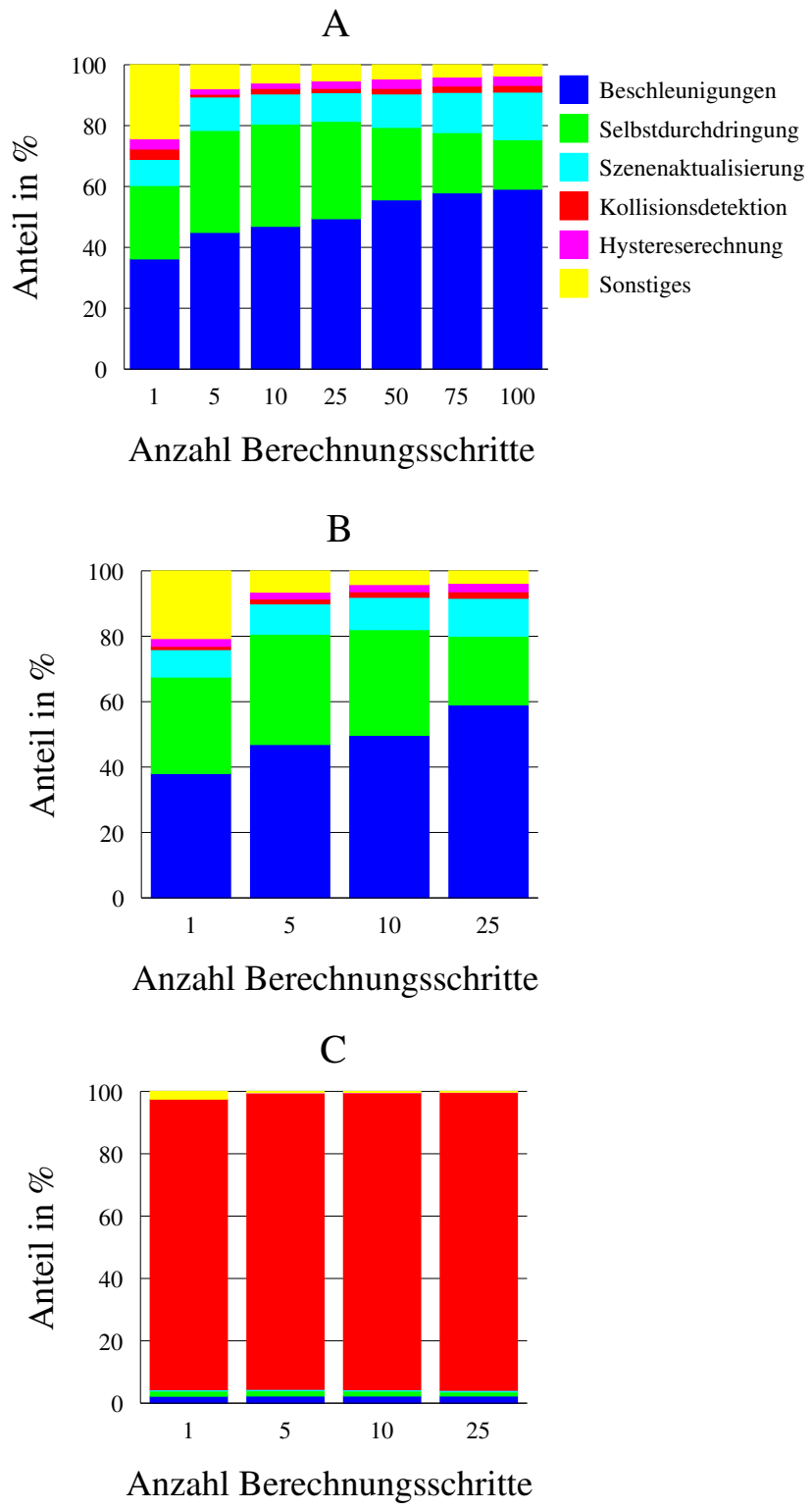

Abbildung 5.1: Ergebnisse der Profiling-Läufe auf der ONYX2. Gezeigt sind die prozentualen Laufzeitanteile der sequentiellen Version für fünf zentrale Funktionen (vgl. Tabelle 5.1.

A

B

C

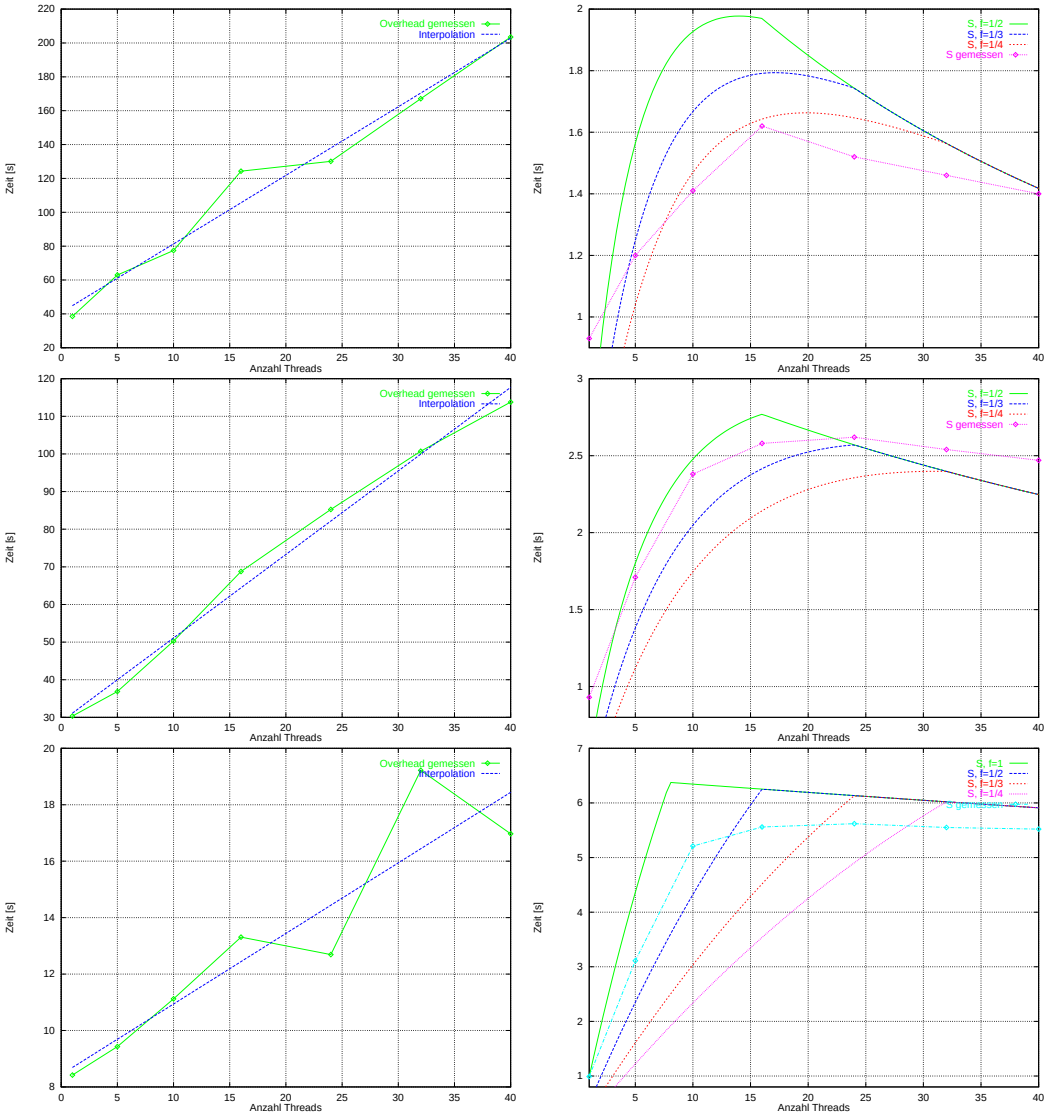

Abbildung 5.2: Parallele Overheads (linke Spalte) und Beschleunigungen (rechte Spalte) auf der ONYX2 für die Konfigurationen A, B und C. Bei den Overheads sind die Meßwerte und eine lineare Interpolation zu sehen. Bei den Beschleunigungen sind zusätzlich zu den Meßwerten die Modellkurven für $f = 1/2, 1/3$ und $1/4$ eingetragen.

Szene	Sequentielle Anteile [%] Gitter	Sonstige	Beschleunigungen S_∞	S_8
A	13.1	4.1	5.80	3.63
B	7.3	3.9	8.93	4.48
C	0.5	0.5	100	7.48

Tabelle 5.3: Theoretisch erreichbare Beschleunigungen ONYX2 (nach Amdahls Gesetz). Mit S_∞ sind die maximalen, mit S_8 die Beschleunigungen mit 8 Prozessoren bezeichnet. Die sequentiellen Programmanteile wurden in die Neuberechnung der Szenengitter und Sonstige unterteilt.

laufzeit $T_S = T_s + T_p$ bei sequentiellem Anteil T_s und parallelem Anteil T_p

$$S(n) = \frac{T_S}{T_s + T_p/n}$$

an die tatsächlichen Gegebenheiten anpassen.
Berücksichtigt man, daß die Anzahl der Prozessoren nur indirekt mit der Anzahl t der Threads über eine unbekannte Konstante $f = n/t$ gekoppelt ist und nimmt den beobachteten parallelen Overhead in Abhängigkeit der Zahl der Threads $o(t)$ in das Modell mit auf, ergibt sich für die erwartete Beschleunigung

$$S^*(t) = \frac{T_S}{T_s + T_p/(tf) + o(t)} \quad .$$

Für die Overheads wurde auf Basis der Messungen eine lineare Abhängigkeit $o(t) = at+b$ angenommen. Für das unbekannte Verhältnis f wurden die Werte 1/2, 1/3 und 1/4 betrachtet. Die Ergebnisse des Modells sind in Abbildung 5.2 mit den aktuellen Meßwerten verglichen. Sie erklären die gemessenen Beschleunigungen qualitativ als auch quantitativ hinreichend. Abweichungen sind den zusätzlichen Kommunikationskosten bei tatsächlicher paralleler Rechnung und der Begrenztheit des Modells zuzuordnen, da das Scheduling der Threads sehr wahrscheinlich nicht, wie idealisiert angenommen, in einem festen Verhältnis, sondern dynamisch implementiert ist.
Die starken Effekte des parallelen Overhead lassen sich nachvollziehen, wenn die Körnung der Parallelität, d.h. die Anzahl der Gabelungen im Verhältnis zur Problemgröße betrachtet wird. So sind beispielsweise in Konfiguration A die Trajektorien von 400 Partikeln zu berechnen, wobei jede Iteration schon bei sequentieller Ausführung weniger als 400 ms Laufzeit benötigt. Bei paralleler Ausführung teilt sich diese Zeit auf die verwendeten Threads und 16 Gabelungen pro Iteration auf. Da nach den Profiles kein Programmteil mehr als 15% erreicht (der Anteil von 60% der Differentialgleichungslösung teilt sich auf 12 Gabelungen auf), bleiben bei angenommenen 24 Threads etwa 2.5 ms Rechenzeit pro Thread. Dem gegenüber stehen, bezieht man die Gesamtzahl der Gabelungen auf den Overhead, etwa 0.25 ms Laufzeitverlust pro Thread. Damit sind fast 10% der Laufzeit reiner Overhead.
Das schlechte Rechen- zu Systemzeit-Verhältnis bei den Konfigurationen A und B zeigen auch die Ergebnisse bei Konfiguration C, bei der die Problemgröße deutlich höher liegt, und die Beschleunigungen entsprechend anwachsen.

	Anzahl Threads							
Konf.	Seq.	1	5	10	16	24	32	40
A_{50} [s]	64.8	69.42	47.85	38.20	30.69	30.18	32.41	34.27
Beschl.	1.00	0.93	1.35	1.70	2.11	2.15	2.00	1.89
A_{100} [s]	174.99	186.48	138.04	116.08	98.91	103.33	108.72	113.64
Beschl.	1.00	0.94	1.27	1.51	1.77	1.69	1.61	1.54
A_{150} [s]	318.71	341.29	252.19	217.00	188.21	191.43	206.19	362.69
Beschl.	1.00	0.93	1.26	1.47	1.69	1.66	1.55	0.88
A_{200} [s]	491.84	527.64	408.24	349.51	302.85	323.51	336.04	350.86
Beschl.	1.00	0.93	1.20	1.41	1.62	1.52	1.46	1.40
B_6 [s]	65.79	69.95	45.91	27.18	23.60	23.33	23.81	24.32
Beschl.	1.00	0.94	1.43	2.42	2.79	2.82	2.76	2.71
B_{12} [s]	176.01	189.02	101.78	72.77	60.26	61.34	62.58	64.19
Beschl.	1.00	0.93	1.73	2.42	2.92	2.87	2.81	2.74
B_{18} [s]	363.52	391.44	205.05	153.42	135.14	133.27	136.74	141.55
Beschl.	1.00	0.93	1.77	2.37	2.69	2.73	2.66	2.57
B_{24} [s]	635.51	683.72	371.72	267.28	246.33	242.38	249.97	257.49
Beschl.	1.00	0.93	1.71	2.38	2.58	2.62	2.54	2.47
C_6 [s]	152.43	153.66	49.59	30.64	28.10	27.74	27.92	28.05
Beschl.	1.00	0.99	3.07	4.97	5.42	5.49	5.46	5.43
C_{12} [s]	303.31	306.07	98.68	59.33	55.15	54.47	54.78	55.37
Beschl.	1.00	0.99	3.07	5.11	5.50	5.57	5.54	5.48
C_{18} [s]	454.65	471.23	147.76	87.54	82.25	81.43	81.62	82.31
Beschl.	1.00	0.96	3.08	5.19	5.53	5.58	5.57	5.52
C_{24} [s]	648.38	654.15	208.30	124.43	116.66	115.37	116.81	117.45
Beschl.	1.00	0.99	3.11	5.21	5.56	5.62	5.55	5.52

Tabelle 5.4: Meßergebnisse auf der ONYX2. Die Indizes bezeichnen die Anzahl der Schritte. Für jede Messung sind die Laufzeiten in Sekunden (erste Zeile) und die Beschleunigungen (zweite Zeile) angegeben. Alle Zeiten wurden normiert (siehe Text), die größten Beschleunigungen sind hervorgehoben.

Konf.		Anzahl Threads 1	5	10	16	24	32	40
A_{200}	[s]	530.36	554.81	569.26	616.14	621.94	658.92	695.32
Δt	[s]	38.52	62.97	77.42	124.30	130.10	167.08	203.48
B_{24}	[s]	665.84	672.37	685.78	704.28	720.77	736.23	749.30
Δt	[s]	30.33	36.86	50.27	68.77	85.26	100.72	113.79
C_{24}	[s]	656.80	657.81	659.50	661.69	661.07	667.60	665.35
Δt	[s]	8.42	9.43	11.12	13.31	12.69	19.22	16.97

Tabelle 5.5: Overhead-Messungen auf der ONYX2. Die Indizes bezeichnen die Anzahl der Schritte. Für jede Messungen sind die Laufzeiten in Sekunden (erste Zeile) und die reinen Overheads (zweite Zeile), d.h. die Laufzeitdifferenz zur entsprechenden sequentiellen Ausführung angegeben.

5.3.2 SUN Sparc MP

Die Messungen wurden auch auf SUNs Multiprozessorrechnern mit vier Prozessoren durchgeführt. Die gemessenen Laufzeiten sind in Tabelle 5.6 zusammengefaßt. Im Vergleich mit der ONYX2 fällt auf, daß mit nur vier Prozessoren bei kleinen Szenen nahezu dieselben Beschleunigungen erreicht werden. Dieses Resultat ist auf den geringeren Overhead zurückzuführen, da in diesem Fall die Anzahl der Threads gleich der Anzahl der Prozessoren gewählt werden konnte. Die Effizienzen sind bei allen Messungen erheblich besser als auf der ONYX2, vor allem wenn kleine Konfigurationen berechnet werden, wo der Overhead-Einfluß stärker ist.
Trotzdem sind deutliche Verluste durch die feine Körnung und dem damit verbundenen Overhead zu verzeichnen. Das läßt sich auch daran erkennen, daß bei Konfiguration A die größten Beschleunigungen schon bei drei Prozessoren erreicht werden, und die Ausführung mit vier Prozessoren leicht langsamer ist, während bei den Konfigurationen B und C, die beide grobkörniger sind, immer bei vier Prozessoren die maximale Beschleunigung erreicht wird.

Konf.	Seq.	1	2	3	4
A_5 [s]	41.07	41.09	27.00	23.08	24.05
Beschl.	1.00	1.00	1.52	1.78	1.71
A_{10} [s]	79.04	80.05	51.04	45.02	46.00
Beschl.	1.00	0.99	1.55	1.76	1.72
A_{20} [s]	154.05	158.00	100.02	87.04	89.05
Beschl.	1.00	0.98	1.54	1.77	1.73
A_{40} [s]	338.06	347.04	220.02	192.09	196.04
Beschl.	1.00	0.97	1.54	1.76	1.72
B_5 [s]	332.02	339.05	213.03	165.06	157.01
Beschl.	1.00	0.98	1.56	1.89	2.11
B_{10} [s]	828	843.09	523.02	456.09	368.06
Beschl.	1.00	0.98	1.58	1.82	2.20
C_5 [s]	191.08	186.08	105.01	79.04	65.04
Beschl.	1.00	1.03	1.82	2.42	2.94
C_{10} [s]	378.04	366.01	205.09	156.09	127.00
Beschl.	1.00	1.03	1.84	2.42	2.98
C_{15} [s]	569.08	545.06	306.06	229.02	187.01
Beschl.	1.00	1.04	1.86	2.48	3.04

Tabelle 5.6: Meßergebnisse auf 4-Prozessor SUN-Rechnern. Die Indizes bezeichnen die Anzahl der Berechnungsschritte. Für jede Messung sind die Laufzeiten in Sekunden (erste Zeile) und die Beschleunigungen (zweite Zeile) angegeben. Die größten Beschleunigungen sind hervorgehoben.

5.4 Verteilter Speicher

Bei der verteilten Implementierung hat man grundsätzlich, soweit die Anwendung dies zuläßt, die Wahl zwischen den beiden Extremen eines einfachen Master-Slave-Algorithmus und einer echt verteilten Berechnung. Die Implementierung eines Master-Slave-Schemas setzt geringe Datenabhängigkeiten und eine recht grobe Körnung voraus, um die Kommunikationskosten, vor allem bei langsameren Verbindungsnetzwerken, auszugleichen. Bei verteilter Berechnung ist oftmals eine feinere Körnung möglich, wenn bei Problemen, wie dem vorliegenden, ein „Oberflächeneffekt“ die Kommunikation reduziert. Dabei müssen nicht nach jedem Schritt die gesamten Problemdaten eingesammelt und später neu übertragen werden, es genügt, wenn die einzelnen Rechner nur mit logisch benachbarten Rechnern die Randelemente ihres Problembereichs austauschen.
Der bei gemeinsamem Speicher verfolgte Ansatz, bei der Lösung der Differentialgleichungen nur die Berechnung der Beschleunigungen zu parallelisieren, greift hier nicht. Untersucht man die in Frage kommende Funktion `accelerInter` auf implizite und explizite Eingabedaten, ergeben sich Abhängigkeiten von 12 veränderlichen Koeffizienten sowie den Partikelorten und -geschwindigkeiten. Diese in jedem Schritt für jeden berechneten Partikel zu übertragenden Daten machen einen einfachen Master-Slave Ansatz unbrauchbar. Selbst unter idealen Umständen darf die Kommunikationszeit die Rechenzeit nicht überschreiten. Die Übertragung von insgesamt 108 Bytes pro Partikel erfordert selbst mit 500 kB/s ohne Overhead etwa 0.2 ms. Da in der Praxis eher geringere Bandbreiten erreicht werden, und ein zusätzlicher Betriebssystemoverhead pro Übertragung hinzukommt, lohnt sich dieser Ansatz im Vergleich zu 0.9 ms Rechenzeit nicht. Für die Lösung der Differentialgleichungen lag deshalb eine verteilte Berechnung nahe.
Auf der anderen Seite verbieten zwei Probleme einen rein verteilten Ansatz. Die Berechnung der Selbstdurchdringung enthält globale Datenabhängigkeiten, die entweder eine sequentielle Rechnung und damit verbunden eine Rückübertragung aller Daten an einen Rechner impliziert, oder eine aufwendige und teure Kommunikation zwischen allen Rechnern erfordert. Da zudem in vielen Anwendungsfällen eine Animation der Dynamik der Textilie gefordert ist, d.h. die Daten für die Speicherung in jedem Fall zurückübertragen werden müssen, wurde die erste Lösung gewählt.
Damit ergibt sich im gesamten ein Hybrid-Ansatz. Die Berechnung der Beschleunigungen und die Lösung der Differentialgleichungen geschehen verteilt. Sie erfordern nur an bestimmten Synchronisationspunkten den Austausch von Informationen über die aneinandergrenzenden Datenbereiche. Die nicht parallelisierbaren Schritte werden zentral auf einem Rechner ausgeführt. Für die verteilte Berechnung werden vorher an jeden Rechner die entsprechenden Daten übertragen, und die Ergebnisse vor der Ausführung der sequentiellen Programmteile wieder auf einem Rechner vereint. In Abbildung 5.3 ist der gesamte Programmfluß für einen Berechnungsschritt wiedergegeben.
Zur parallelen Berechnung wird die Textilie in eine der Zahl der Rechner entsprechende Zahl von „Streifen“ zerlegt. Um die Kommunikationskosten klein zu halten, werden die Partikel vor der Zuteilung zu den Prozessoren nach der Koordinate der größten Ausdehnung der die Textilie umgebenden Bounding-Box sortiert. Bei re-

gulären Strukturen, etwa einer rechteckigen Einteilung mit jeweils vier Nachbarn, erhält man dadurch das Minimum der notwendigen Kommunikation.
Um trotz der hohen Kommunikationskosten bei der Verteilung und Rückübertragung der Daten viel an Geschwindigkeit zu gewinnen, mußte darauf geachtet werden, daß der parallel gerechnete Anteil möglichst hoch ist. Für die verteilte Berechnung wurde deshalb auf der Stufe der Differentialgleichungslöser parallelisiert. Da eine exakte Untersuchung und Parallelisierung aller zur Verfügung stehenden Löser den Rahmen der Diplomarbeit gesprengt hätte, wurde ein spezieller Löser für nicht-steife Differentialgleichungen betrachtet. Er basiert auf einem Runge-Kutta-Verfahren vierter Ordnung, das in der Praxis gute Ergebnisse geliefert hat. Der Löser und die entsprechende Treiberfunktion `rkqs` wurden schon in Kapitel 4 beschrieben. Die ursprüngliche Treiberfunktion `rkqs` wurde durch die Methode `ParClothInter::do_rkqs` ersetzt und dahingehend modifiziert, daß sie die parallele Berechnung eines Lösungsschritts anstößt und die von den einzelnen Rechnern zurückgelieferten lokalen Fehlerwerte weiterverarbeitet. Da die Beschleunigungen von den Nachbarelementen eines Streifens abhängen, müssen diese im Löser selbst nach jedem Teilschritt neu synchronisiert werden. Dazu wird die in DTS integrierte Shared-memory-Komponente genutzt[1].
Der folgende Programmausschnitt zeigt die relevanten Teile des parallelen Differentialgleichungslösers. Die Methode entspricht einem Lösungsschritt und wird auf allen Rechnern gleichzeitig ausgeführt. In den Zeilen 4–11 werden die gemeinsamen Speicherbereiche deklariert, mit deren Hilfe der Ausführer die lokalen Ergebnisse mit den logisch benachbarten Rechnern synchronisiert. Da die Zugriffe auf die benachbarten Bereiche eine unregelmäßige Struktur haben, kann die Synchronisation nicht direkt erfolgen, sondern wird über die Hilfsfelder `tleft, fleft, tright` und `fright` abgewickelt. Im zweiten Teil (Zeilen 13–25) wird die eigentliche Berechnung durchgeführt. Dabei wird nach jedem Lösungschritt mit Hilfe der Funktionen `synchronize_neighbours` und `get_synchronized` eine Synchronisation durchgeführt, welche die lokalen Änderungen über die Hilfsfelder an die Nachbarn weitergibt. Schließlich werden im letzten Abschnitt (Zeilen 27–34) die gemeinsamen Speicherregionen wieder freigegeben.

```
1  void
2  ParClothInter::par_rkck(int host, float x, float h, float *errmax) {
3  ...
4    if (host>0) {
5      dts_shm_attach(TO_LEFT(host),   (char *)tleft, leftsize,  2);
6      dts_shm_attach(FROM_LEFT(host), (char *)fleft, leftsize,  2);
7    }
8    if (host<num_hosts-1) {
9      dts_shm_attach(FROM_RIGHT(host), (char *)fright, rightsize, 2);
10     dts_shm_attach(TO_RIGHT(host),   (char *)tright, rightsize, 2);
11   }
12 ...
```

[1]Das DSM erleichterte in diesem Fall zwar die Programmierung, war aber aufgrund der irregulären Datenzugriffe nicht optimal einsetzbar. Bei einer Einschränkung auf einfachere, reguläre Textilien hätte man das DSM wesentlich besser ausnutzen können, dabei aber viele interessante Anwendungen ausgeschlossen.

```
13    // Aktuelle Koordinaten und Ableitungen für diesen Streifen und
14    // die 2.Nachbarelemente sind bekannt. Berechne Stützstellen
15    // an allen Stellen innerhalb dieses Streifens und den 1.Nachbarn ...
16    for (i=0, idx=index; i<idx_size; i++,idx++)
17      ytemp[*idx]=IValues[*idx]+b21*h*dydx[*idx];
18
19    // ... und damit erste Approximation in diesem Streifen.
20    getNRdydx(host, x+a2*h, ytemp, ak2);
21
22    // ... Synchronisation der berechneten Werte mit den Nachbarn
23    //     (Austausch 1. und 2. Nachbarn)
24    synchronize_neighs(INITIAL, AK2_DATA, host, ak2, tleft, tright);
25    get_synchronized(AK2_DATA, host, ak2, fleft, fright);
26  ...
27    if (host>0) {
28      dts_shm_detach(FROM_LEFT(host));
29      dts_shm_detach(TO_LEFT(host));
30    }
31    if (host<num_hosts-1) {
32      dts_shm_detach(FROM_RIGHT(host));
33      dts_shm_detach(TO_RIGHT(host));
34    }
35  ...
36  }
```

5.4.1 Workstation-Netz

Zur Messung wurde ein Workstation-Netz aus fünf Sparc 10-Rechnern verwendet. Um die Vergleichbarkeit der parallelen Laufzeiten zu verbessern, wurde der Central Manager von DTS auf einem zusätzlichen, eigenen Rechner ausgeführt, und die vier verbleibenden Rechner der Anwendung zugeteilt. Die Ergebnisse sind in Tabelle 5.7 zu sehen. Die Messungen geben leider nur Tendenzen wieder, da die Laufzeiten der parallelen Ausführungen sehr stark von der aktuellen Netzlast abhingen. Je nach Belastung waren Laufzeitunterschiede von bis zu 20% meßbar.

Der Gewinn durch die parallele Lösung der Differentialgleichungen, wurde durch die hohen Kommunikationskosten bei der Verteilung und Rückübertragung der Daten egalisiert. Man erkennt an der Laufzeitabnahme von einem zu vier Rechnern, daß die parallele Ausführung allein eine Beschleunigung von etwa zwei bewirkt, was dem Anteil der Differentialgleichungslösung an der Gesamtlaufzeit entspricht.

Echte Beschleunigungen wären nur zu erzielen, wenn man sich auf einen reinen Differentialgleichungslöser beschränkte und nicht an den Zwischenschritten interessiert wäre. Dann entfiele die Kommunikation in jedem Schritt und wäre nur am Anfang und Ende der Berechnung notwendig.

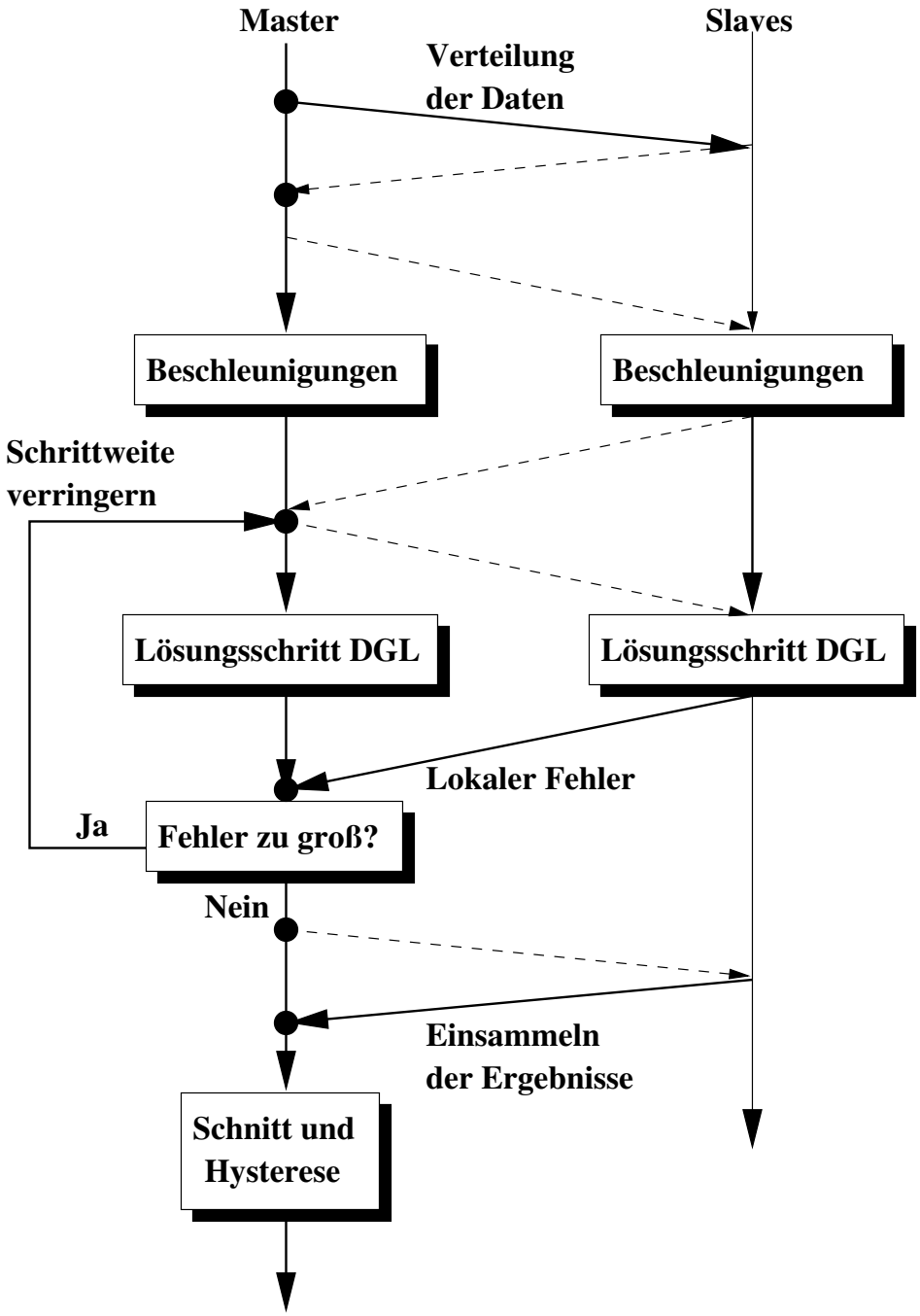

Abbildung 5.3: Programmstruktur bei verteiltem Speicher. Gestrichelte Pfeile repräsentieren reine Nachrichten, bei durchgezogenen Pfeilen werden die angegebenen Daten mitübertragen.

Konf.	Seq.	Anzahl Rechner 1	2	4
A_{10} [s]	149.3	321.89	222.31	162.43
Beschl.	1.00	0.46	0.67	0.92
A_{15} [s]	170.73	369.23	307.50	268.43
Beschl.	1.00	0.46	0.56	0.64
A_{20} [s]	224.71	603.95	426.27	374.52
Beschl.	1.00	0.37	0.53	0.60
B_2 [s]	235.26	480.23	288.25	205.40
Beschl.	1.00	0.49	0.82	1.15
B_4 [s]	502.29	994.64	588.08	411.31
Beschl.	1.00	0.50	0.85	1.22
B_6 [s]	779.35	1572.39	901.92	630.53
Beschl.	1.00	0.50	0.86	1.24

Tabelle 5.7: Meßergebnisse im SUN Workstation-Netz. Die Indizes bezeichnen die Anzahl der Berechnungsschritte. Für jede Messung sind die Laufzeiten in Sekunden (erste Zeile) und die Beschleunigungen (zweite Zeile) angegeben. Die größten Beschleunigungen sind hervorgehoben.

5.4.2 Andere Architekturen

Zur Untersuchung der parallelisierten Programmversion standen noch zwei weitere Architekturen zur Verfügung. Auf der NEC SX/4, einer Architektur mit gemeinsamem Speicher, standen die Basisbibliotheken OBMS und RadioLab nicht zur Verfügung. Eine Portierung war, trotz Unterstützung der Entwickler, nicht mehr rechtzeitig durchführbar.

Das IBM RS/6000 SP-System wurde nach den Ergebnissen im Workstation-LAN nicht weiter untersucht. Dabei waren zwei Gründe ausschlaggebend. Auf dem SP-System standen zur Parallelisierung eine spezielle, auf den High Performance Switch zugeschnittene Version von PVM, PVMe [21], sowie die Public-Domain-Version von PVM zur Verfügung. Die Ergebnisse im Workstation-Netz zeigen, daß die normale Ethernet-Kommunikationsbandbreite für das vorliegende Problem nicht ausreichend ist, womit die Verwendung von PVM keinen Gewinn verspricht. Um PVMe zu nutzen, wäre dagegen eine erneute Portierung von DTS unumgänglich, da es nicht den vollen Funktionsumfang von PVM zur Verfügung stellt, auf den DTS aufbaut. Gegen diesen Aufwand sprechen die Ergebnisse der Profiles auf dem SP-System. Diese zeigen, daß mit einem parallelisierbaren Anteil von 60% bei der Differentialgleichungslösung eine maximale Beschleunigung von 2.5 erreichbar wäre, die durch die hinzukommenden Kommunikationskosten noch weiter reduziert werden würde.

5.5 Fazit

Bei den untersuchten Szenen, die von der Problemgröße in den realistischen Bereich einzuordnen sind, können bei Verwendung von gemeinsamem Speicher Beschleunigungen von zwei bis drei erreicht werden. Begrenzender Faktor ist der Overhead durch die feine Körnung des Problems. Mit steigender Problemgröße wächst die erreichbare Beschleunigung. Desweiteren verursachen Routinen der zugrundeliegenden Graphik-Bibliotheken einen sequentiellen Anteil von bis zu 10%, womit die Beschleunigung ohne Eingriff in diese selbst im Idealfall in der Größenordnung 10 liegt.
Im Falle von verteiltem Speicher machen die Kommunikationskosten die zu erwartende geringe Beschleunigung von höchstens 3 in den meisten Fällen zunichte. Die Anwendung ist damit in der vorliegenden Form für eine Parallelisierung auf Rechnern mit verteiltem Speicher ungeeignet. Die Ursache liegt zum einen darin, daß der äußere Aufbau der Anwendung grundsätzlich der eines Iterationsverfahrens ist, bei dem die einzelnen Schritte sequentiell ausgeführt werden müssen. Es bleibt also nur eine Parallelisierung der Schritte selbst, deren Laufzeit je nach Architektur zwischen einer halben und zehn Sekunden lagen. Zum anderen stellt die Berechnung der Selbstdurchdringung ein globales, d.h. von allen Daten abhängiges Problem dar, für dessen effiziente Parallelisierung die Modifikation der Graphik-Bibliotheken und eine völlige Neuentwicklung der Algorithmen notwendig wäre.

Kapitel 6

Erweiterung von DTS

6.1 Grenzen von DTS

DTS war — in der ursprünglichen Form — zur parallelen Programmierung mit datenunabhängigen Threads konzipiert. Dieses Programmiermodell ist für eine gewisse Klasse von Problemen ausreichend. Es verbietet jedoch wichtige parallele Programmiertechniken, so zum Beispiel datenparallele Programme, bei denen verschiedene Threads gemeinsam an einem Problem arbeiten, dabei aber über Speicher, Sperren, Bedingungsvariablen o.ä. kooperieren. Im verteilten Fall entspricht dies einem Algorithmus bei dem die Problemdaten unter mehreren Rechnern aufgeteilt werden, die sich im Laufe der Berechnung an bestimmten Punkten durch direkte Kommunikation synchronisieren. Die Implementierung solcher Algorithmen war in der bisherigen Version von DTS zwar durchführbar, aber nur unter unnötigem Aufwand und der Umgehung des eigentlichen Systems, d.h. im Grunde wurde nur auf das darunterliegende Nachrichtensystem zurückgegriffen.

6.2 Distributed Shared Memory

Eine vielversprechende Lösung des obigen Problems stellt die Implementierung von Distributed Shared Memory (DSM) dar. Sie erweist sich in DTS sogar als einziger gangbarer Weg, denn DTS wurde entworfen um eine Abstraktion von dem darunterliegenden Message-Passing-System zu erreichen. Ziel war es, die herkömmliche Thread-Programmierung so gut wie möglich auf verteilten Systemen nachzubilden. Auch wegen der vereinfachten Programmierung bietet sich eine DSM-Implementierung in DTS an.
Grundsätzlich gliedert man DSM-Systeme nach dem zugrundeliegenden Konsistenzmodell. Die wichtigsten Konsistenzmodelle werden im folgenden kurz vorgestellt.

Sequentielle Konsistenz Nach Lamports Definition [25] muß bei der sequentiellen Konsistenz das Ergebnis der parallelen Ausführung dem Ergebnis *irgendeiner* sequentiellen Ausführung entsprechen, wobei die von einem bestimmtem Prozessor ausgeführten Operationen in der lokalen, vom Programm vorgegebenen Reihenfolge, in der Gesamtsequenz auftauchen. Aufgrund des enormen Aufwands der zur Aufrechterhaltung der sequentiellen Konsistenz nötig ist (jede Änderungen muß sofort für alle anderen Prozessoren sichtbar werden), wird sie fast ausschließlich bei Hardware-Lösungen eingesetzt. Eine Ausnahme ist das IVY-System [26], eine frühe Lösung, die auf reiner Software-Basis seitenbasierte sequentielle Konsistenz implementiert.

Prozessor-Konsistenz Goodman und Woest [15] beobachteten, daß selbst einige Multiprozessor-Rechner die Anforderungen der sequentiellen Konsistenz nicht erfüllen, sondern lediglich die der Prozessor-Konsistenz. Dabei müssen Schreibzugriffe eines Prozessors bei jedem anderen Prozessor in derselben Reihenfolge ausgeführt werden, Schreibzugriffe *verschiedener* Prozessoren brauchen dagegen nicht überall in derselben Reihenfolge beobachtet werden.

Schwache Konsistenz Speicherzugriffe lassen sich bei genauerer Betrachtung in zwei Klassen einteilen: Synchronisations- und normale Speicherzugriffe [8]. Da normalerweise der Zugriff auf gemeinsam genutzte Daten auf höherer Ebene mit Synchronisationsprimitiven geregelt wird, können die Anforderungen an das Speichersystem bei korrekter Programmierung abgeschwächt werden. Konkret genügt es, wenn nicht alle, sondern lediglich die Synchronisationszugriffe sequentiell konsistent sind[1].

Freigabe-Konsistenz Die Vorgehensweise bei schwacher Konsistenz kann bei feinerer Unterscheidung innerhalb der Zugriffe noch weiter optimiert werden. Gharachorloo et al. [14] teilen gemeinsame Speicherzugriffe in freie und konkurrierende Zugriffe ein. Die letzteren werden in solche die der Synchronisation dienen und sonstige eingeteilt. Bei den Synchronisationszugriffen werden schließlich die Klassen „Erwerb“ und „Freigabe“ unterschieden (siehe Abbildung 6.1). Mit *passend ausgezeichneten* Programmen (engl. properly-labeled programs) deren Zugriffe korrekt nach diesem Schema eingeteilt werden, genügt es, Speicherzugriffe bei einer Freigabe anderen Prozessoren zur Verfügung zu stellen (die sogenannte eifrige Freigabe-Konsistenz, engl. eager release concistency). Man unterscheidet in diesem Fall Invalidate- und Update-Protokolle, d.h. Verfahren bei denen nur die Information über eine notwendige Aktualisierung weitergegeben wird, die dann der betreffende Rechner bei Bedarf selbst durchführt, oder Verfahren, bei denen nach der Freigabe alle Rechner auf den aktuellen Stand gebracht werden. Bei der *faulen Freigabe-Konsistenz* (engl. lazy release consistency) [23] werden die Auswirkungen der Zugriffe bis zum nächsten Erwerb verzögert. Damit reduziert sich die Anzahl der notwendigen Aktualisierungen noch weiter. Eine eifrige Version der Freigabe-Konsistenz wurde erstmals im Munin-System [7] implementiert.

[1] Im Grunde genügt sogar die Prozessor-Konsistenz der Synchronisationen.

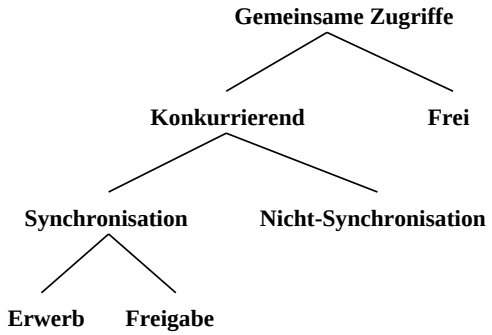

Abbildung 6.1: Klassifizierung der Zugriffe bei schwacher Freigabe-Konsistenz

Eintritts-Konsistenz Die Eintrittskonsistenz (engl. entry consistency) geht noch einen Schritt weiter. Vom Protokoll her mit der faulen Freigabe-Konsistenz identisch, werden bei der Eintrittskonsistenz Synchronisations- und Datenobjekte explizit gekoppelt. Damit kann die Aktualisierung der Daten an einem Synchronisationspunkt auf die assoziierte Speicherregion beschränkt werden. Bershad et al. beschreiben diesen Ansatz und seine Implementierung im Midway-System in [3].

Die Konsistenzmodelle versuchen einen Ausgleich zwischen erschwerter Programmierung und akzeptabler Leistung zu finden. Je höher die Konsistenz, desto einfacher wird die Programmierung, desto höher sind aber auch die Leistungsverluste. Die Entwicklung der höheren Konsistenzmodelle ergibt sich aus der einfachen Beobachtung, daß für parallele Programmierung in den meisten Fällen Synchronisationsprimitive notwendig sind. Selbst bei sequentieller Konsistenz, die normalerweise nur bei Multiprozessor-Rechnern vorliegt, kann nicht ohne weitere Synchronisation parallel programmiert werden. Die schwächeren Konsistenzmodelle setzen genau an dieser Stelle ein. Wenn aber beim Zugriff auf gemeinsamen Speicher Synchronisationsprimitive verwendet werden müssen, liegt es nahe, die Konsistenzmechanismen mit diesen zu koppeln. Die Konsistenz wird dabei mit der Forderung einer korrekten Programmierweise erkauft. Ein wichtiger Aspekt stellt in diesem Zusammenhang auch der Gegenstand der Konsistenz dar. Bei stärkeren Modellen wird oft der gesamte virtuelle Speicher auf Seitenbasis konsistent gehalten. Das zieht eine starke Hardwareabhängigkeit und unnötige Aktualisierungen nach sich. Bei schwächeren Ansätzen können Nachrichten eingespart werden, indem nur explizite, als gemeinsamer Speicher deklarierte Regionen berücksichtigt werden.
Neuere DSM-Systeme gehen über diese Möglichkeiten noch hinaus, um die Leistung bei stärkerer Konsistenz zu verbessern und damit gleichzeitig den Programmierer zu entlasten. So ist beispielsweise in Treadmarks [22] ein Protokoll implementiert, das mehrere Schreiber zuläßt. Die Änderungen werden bei Synchronisationen laufweitenkodiert (engl. runlength encoded) an die anderen Rechner weitergegeben. Andere Ansätze versuchen über dynamische Protokollwahl [2] (in Abhängigkeit vom Zugriffsmuster der Applikation) oder Schleifenvorhersagen (über sog. Inspektoren [27]) die Leistung und Anwendungsmöglichkeiten von DSM-Systemen zu erweitern.

6.3 Entwurfskriterien und Lösungsansatz

Die vorliegende DSM-Implementierung beschränkt sich auf den im Rahmen einer Diplomarbeit gangbaren und von DTS vorgegebenen Weg. Auch die zu parallelisierende Anwendung hatte Einfluß auf die konkrete Umsetzung.
Beim Entwurf der DSM-Komponente in DTS standen Allgemeinheit, Effizienz und gute Ausnutzung von vorhandenem gemeinsamem Speicher im Vordergrund. Das System sollte auf eine möglichst große Problemklasse anwendbar sein, dabei aber eine sehr geringe Anzahl von Synchronisationsnachrichten verwenden, um auch bei einem langsameren Verbindungsnetzwerk noch einsetzbar zu sein. Da DTS auch auf Rechnern mit gemeinsamem Speicher eingesetzt wird, war es ebenfalls wichtig, daß die Implementierung im lokalen Fall nur sehr wenig Zusatzaufwand erfordert, d.h. bei Vorhandensein von gemeinsamem Speicher so gut wie möglich darauf zurückgreift. Die Implementierung in DTS erfordert zudem einen rein softwarebasierten Ansatz, um die Portabilität des Systems zu erhalten. Die Möglichkeit, auch in heterogenen Umgebungen zu arbeiten, verbietet darüber hinaus das System auf einem Speicherschutz-Mechanismus aufzubauen. Eine weitere Randbedingung war die Vermeidung von Spracherweiterungen (und damit die Programmierung von speziellen Compilern). Bei der Umsetzung der notwendigen Synchronisationsprimitive boten sich, nach dem zu erwartenden Einsatzbereich und aufgrund der vereinfachten Programmierung, vorläufig zentrale Algorithmen an. Die in DTS vorhandene Eigenschaft des Wiederaufsetzens (engl. recovery) bei Ausfall eines Rechners geht naturgemäß verloren, da bei datenparallelen Algorithmen mit dem Ausfall eines Rechners auch einen Teil der Daten verlorengeht und der Algorithmus keinesfalls mehr zu einen korrekten Ergebnis führen kann.
Da im Hinblick auf die recht feinkörnige Parallelität der vorliegenden Anwendung möglichst wenig ausgetauschte Nachrichten erstrebenswert waren, wurde das schwächste der oben besprochenen Modelle, die Eintrittskonsistenz mit fauler Weitergabe der Daten implementiert. Dieser Ansatz kommt so dem in Midway implementierten am nächsten, der ebenfalls eine sehr schwache Konsistenz realisiert und lediglich einzelne Synchronisationsobjekte und nicht den gesamten Speicher konsistent hält. Eine weitere Ähnlichkeit ist die Kopplung von Synchronisation- und Speicherobjekten.
Die Bereitstellung von DSM alleine reicht in den meisten Fällen für eine korrekte Parallelisierung nicht aus. Um Wettlaufbedingungen zu vermeiden oder den Zugriff bei Datenabhängigkeiten in einer festen Reihenfolge zu sequentialisieren werden zusätzlich Synchronisationsprimitive benötigt. Als grundsätzliche Methode zur Vermeidung von Wettlaufbedingungen bietet sich die *Sperre* an, die den wechselseitige Ausschluß von Threads aus kritischen Regionen garantiert. Speziell bei verteiltem Speicher ergeben sich aber noch weitere Probleme, die an folgendem Beispiel, einem einfachem Produzent-Konsument-Schema ohne Pufferung, verdeutlicht werden sollen:

```
condition data_access;
int data_ready;
char shared_buffer[SIZE];

void consumer(void) {
  while (1) {
    cond_lock(data_access);
    while (!data_ready) cond_wait(data_access);
    consume(shared_buffer);
    data_ready=FALSE;
    cond_signal(data_access);
    cond_unlock(data_access);
  }
}

void producer(void) {
  while (1) {
    cond_lock(data_access);
    while (data_ready) cond_wait(data_access);
    produce(shared_buffer);
    data_ready=TRUE;
    cond_signal(data_access);
    cond_unlock(data_access);
  }
}
```

Zuerst fällt auf, daß bei einer einfachen Sperre nicht zwischen Lesern und Schreibern unterschieden werden kann. Während diese Tatsache bei gemeinsamem Speicher nicht ins Gewicht fällt, muß ohne diese Information bei verteiltem Speicher der gemeinsame Speicherbereich nach dem Zugriff *in jedem Fall* an andere Rechner übertragen werden, auch wenn der Thread nur lesend darauf zugriff. Um diesen erhöhten Aufwand zu vermeiden, werden bei der Implementierung Lese- und Schreibsperren unterschieden. Das hat den weiteren Vorteil, daß im Fall von Lesesperren mehreren Rechnern gleichzeitig der Zugriff erlaubt werden kann, d.h. es kann potentiell mehr Parallelität ausgenutzt werden.
Als weiterer kritischer Punkt entpuppt sich die Verwendung der Synchronisationsvariable `data_ready`. Die Berechtigung dieses Vorgehens liegt in der Voraussetzung von gemeinsamem Speicher. Prinzipiell genügt eine Implementierung von gemeinsamem Speicher, um darauf aufbauend einen solchen Synchronisationsmechanismus zu realisieren. Diese Vorgehensweise bringt allerdings eine Vielzahl unnötiger Nachrichten mit sich und macht das Verfahren zu ineffizient. Die Konsequenz daraus ist die Bereitstellung einer zusätzlichen Synchronisationsmöglichkeit.
Um die Synchronisation so effizient wie möglich zu halten, wurde eine Eintrittskonsistenz implementiert. Dort sind die Synchronisations- und Sperrmechanismen integriert sind, und sparen so überflüssige Nachrichten ein. Die Idee der hier implementierten Synchronisation liegt in der Einbindung der oben frei verwendeten Variable `data_ready` in den Sperrmechanismus. Mit jeder gemeinsamen Speicherregion wird eine zusätzliche Variable, die *Sequenznummer*, assoziiert, die je nach

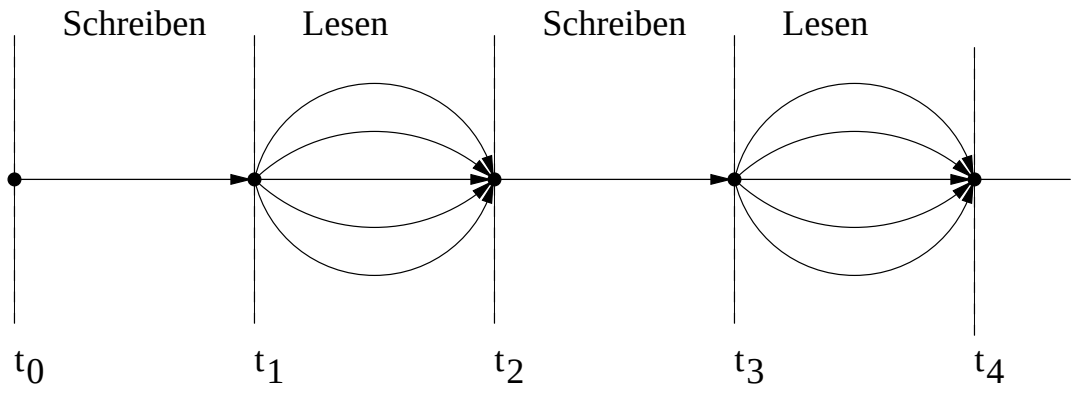

Abbildung 6.2: Lese- und Schreibphasen beim Zugriff auf eine gemeinsame Speicherregion

Status des Objekts verändert werden kann. Zur Synchronisation kann die Sperre bedingt angefordert werden, d.h. gekoppelt mit dem Erreichen einer bestimmten Wertes. Beim Schreiben der Daten kann gleichzeitig der Wert der Sequenznummer verändert werden.

Ein besonderes Problem ergibt sich bei Lesezugriffen. Als Beispiel soll der Fall eines Schreibers mit mehreren Lesern dienen. Würde man Lesern erlauben die Sequenznummer zu verändern, ergäbe sich daraus eine Sequentialisierung der Lesezugriffe in einer vom Programmierer vorgegebenen Reihenfolge. Um dieses Problem zu vermeiden, dürfen Leser die Sequenznummer nicht explizit verändern, sie wird *implizit* bei jeder Freigabe einer Lesesperre um 1 erhöht. Zudem wirkt sich die Erhöhung der Sequenznummer erst beim nächsten Übergang von einer Lese- zu einer Schreibphase aus. Damit können mehrere Leser parallel, in einer beliebigen Reihenfolge, auf den gemeinsamen Speicherbereich zugreifen. Ein folgender Schreiber kann dennoch kontrolliert das Ende der Lesephase abwarten. In Abbildung 6.2 sind die abwechselnden Schreib- und Lesezugriffe illustriert. Zusammengefaßt gilt damit für die Sequenznummer

$$
\begin{aligned}
s(t_0) &= 0 \\
s(t_{n+1}) &= \begin{cases} s(t_n) & \text{nach Schreiben ohne Modifikation der Sequenznummer} \\ s^* & \text{nach Schreiben mit Setzen neuer Sequenznummer } s^* \\ s(t_n)+l & \text{nach Lesephase mit } l \text{ Lesern} \end{cases}
\end{aligned}
$$

Der Ausnutzung von vorhandenen gemeinsamem Speicher wurde durch ein zweistufiges Verwaltungsschema Rechnung getragen. Der Central Manager sorgt sich dabei nur um die Vergabe und Synchronisation auf Rechnerebene, jeder Node Manager muß sich selbst um die Vergabe der Ressourcen an die lokalen Threads kümmern.

6.4 Benutzerschnittstelle

Zur Deklaration und dem Zugriff auf den mit einer gemeinsamen Region assoziierten lokalen Speicherbereich dient

```
void *dts_shm_attach(int id, char *mem, int length, int number);
```

Die Region wird über eine eindeutige, vom Benutzer frei wählbare Identifikation `id` angesprochen. Sie muß bei späteren Zugriffen angegeben werden. Beim Anbinden kann der aufrufende Thread gleichzeitig mittels `mem` und `length` den zugehörigen lokalen Speicherbereich deklarieren. Damit wird einerseits die Notwendigkeit einer Compilerunterstützung vermieden, die sich bei der Identifikation gemeinsamer Speicherbereiche über Variablennamen ergibt, zum anderen entspricht diese Art der Anbindung dem dynamischen Charakter von DTS. Die Teilnahme an gemeinsamem Speicher ist in dieser Form eine Thread-Eigenschaft und kann jederzeit zur Laufzeit verändert werden. Auch die Anzahl der gemeinsamen Regionen muß nicht im voraus festgelegt werden.
Bei der Anbindung ergibt sich in der Praxis ein zusätzliches Synchronisationsproblem. Oft ist es erforderlich zu warten, bis alle beteiligten Threads ihre lokalen Speicherbereiche deklariert haben. Diesem Problem wird durch die Bereitstellung eines synchronen Anbindungs-Mechanismus Rechnung getragen. Er entspricht einer Schranke mit über die Variable `number` spezifizierbarer Anzahl beteiligter Threads. Bei unsynchronisierter Anbindung kann `number=0` angegeben werden.
Verschiedene Mechanismen zur Festlegung eines assoziierten Speicherbereichs sind denkbar. Zum einen kann der Thread einen bestimmten Speicherbereich dem System übergeben, auf der anderen Seite könnte auch das System selbst Speicher der gewünschten Länge bereitstellen. Der erste Mechanismus hat den Vorteil, daß der Thread die Adresse des Speicherbereichs bestimmen kann. Das kann zu einer erheblichen Vereinfachung des Algorithmus beitragen, man denke an Nachbarelemente in einem Array auf die gemeinsam zugegriffen wird, das vom einzelnen Thread aber als Gesamtes angesprochen wird. Bei der zweiten Methode ergibt sich dagegen eine Möglichkeit zu lokalen Optimierung. Teilen sich mehrere Threads auf demselben Rechner eine Speicherregion, so kann das System jedem dieser Threads denselben Speicherbereich zuteilen. Beide Methoden wurden implementiert und können vom Benutzer je nach Bedarf angewandt werden. Übergibt der Benutzer `mem=NULL`, stellt das System Speicher bereit und gibt bei Aufruf von `dts_shm_attach` einen Zeiger darauf zurück.

Die Teilnahme an einem gemeinsamen Speicherbereich kann mit

```
int dts_shm_detach(int id)
```

wieder aufgegeben werden.
Zum Zugriff auf die Sequenznummer dienen die Primitive

```
int dts_shm_getseq(int id);
int dts_shm_setseq(int id, int value);
```

Mit `dts_shm_getseq` kann von jedem Thread, der gerade Zugriff auf die Speicherregion besitzt, die aktuelle Sequenznummer gelesen werden. Threads mit Schreibzugriff ist es darüber hinaus erlaubt, mit `dts_shm_setseq` der Sequenznummer einen neuen Wert zuzuweisen. Zentral sind die Primitive

```
int dts_shm_lock(int id, int type, int seq);
int dts_shm_unlock(int id, int type);
#define dts_shm_readlock(i, s) dts_shm_lock(i,SHM_READ_LOCK,s)
#define dts_shm_writelock(i,s) dts_shm_lock(i,SHM_WRITE_LOCK,s)
#define dts_shm_readunlock(i)  dts_shm_unlock(i,SHM_READ_LOCK)
#define dts_shm_writeunlock(i) dts_shm_unlock(i,SHM_WRITE_LOCK)
```

zum Erwerb und der Freigabe von Schreib- oder Leserechten auf den Regionen. Die Region wird, wie oben, über eine Ganzzahl `id` identifiziert, die Unterscheidung zwischen Schreib- und Lesesperren trifft die Variable `type`, die jeweils die Werte `SHM_READ_LOCK` oder `SHM_WRITE_LOCK` annehmen kann. Zum bedingten Erwerb einer Sperre dient der Parameter `seq`. Die Sperre wird erst bei Erreichen der angegebenen Sequenznummer `seq` vergeben. Ist keine Synchronisation gewünscht, kann `DTS_ANY_SEQ` angegeben werden.

Die Verwendung der Primitive wird in Programm 6.1 illustriert, in dem ein Vektor parallel normiert wird. Dazu wird im ersten Schritt die Norm des Vektors bestimmt, danach können die neuen Vektorelemente berechnet werden. Diese Art der Interaktion wäre in DTS ohne DSM nicht realisierbar, da das Ergebnis eines einzelnen Threads von den gesamten Daten und damit von den Ergebnissen anderer Threads abhängt. In DTS müßte entweder die Norm vorab berechnet werden, oder über zwei parallele Aufrufe zuerst die Norm bestimmt und später die Skalierung durchgeführt werden. In jedem Fall sinkt die Leistung, da die erste Lösung Parallelität verschenkt und bei der zweiten Lösung die Eingabedaten *zweimal* versandt werden müssen.
Bei der Verwendung von DSM wird die Variable `norm` vom Typ `float` als gemeinsamer Speicherbereich definiert. Sie dient als Synchronisations- und Kommunikationspunkt zwischen den parallel arbeitenden Threads. Jeder Thread addiert die lokale Quadratsumme zu diesem Element, und kann, nach Zugriff aller anderen Threads, das Ergebnis auslesen und damit den lokalen Vektorabschnitt normieren.

Programm 6.1 Parallele Normierung eines Vektors

```
#define NUM_THREADS 10                    /* Zahl der Threads         */
#define STRIPE 1000                       /* Länge eines Streifens    */
#define ARRAY_LENGTH (NUM_THREADS*STRIPE) /* Vektor-Länge             */
#define NORM 1                            /* beliebige Identifikation */

/*pure*/
thread(int number, int length, /*inout*/float array[/*dtslen length*/) {
  float local_norm=0, norm;
  int i;

  /* Anbindung */
  dts_shm_attach(NORM, &norm, sizeof(norm), NUM_THREADS);

  /* Lokale Quadratsumme berechnen */
  for(i=0; i<length; i++) local_norm+=array[i]*array[i];

  /* Addition zu gemeinsamer Summe */
  dts_shm_writelock(NORM, number);
  norm+=local_norm;
  dts_shm_setseq(NORM, number+1);
  dts_shm_writeunlock(NORM);

  /* Auslesen gemeinsamer Summe nachdem alle Threads den lokalen
     Beitrag aufsummiert haben */
  dts_shm_readlock(NORM, NUM_THREADS);
  local_norm=sqrt(norm);
  dts_shm_readunlock(NORM);

  /* Normierung des Vektors */
  for(i=0; i<length; i++) array[i]/=local_norm;

  /* Freigabe */
  dts_shm_detach(NORM);
}

void main(void) {
  float global_array[ARRAY_LENGTH];
  dts_t tid[NUM_THREADS];
  int i;

  /* Aufruf der einzelnen Threads mit jeweils einem Streifen */
  for(i=0; i<NUM_THREADS; i++)
    tid[i]=fork_thread(i, STRIPE, global_array+i*STRIPE);

  /* Einsammeln der Ergebnisse */
  for(i=0; i<NUM_THREADS; i++) join_thread(tid[i]);
}
```

6.5 Implementierung

Wie bereits angedeutet, wurde der gemeinsame Speicher in einem zweistufigen Schema implementiert. Die Systemebene verwaltet die einzelnen Rechner, während jeder Rechner sich um die Synchronisation der lokalen Threads kümmert. Durch diese Aufteilung erhält man einerseits eine gute Leistung im Fall von realem gemeinsamem Speicher, auf der anderen Seite bleiben die Algorithmen der Systemebene architekturunabhängig. Die folgenden beiden Abschnitte diskutieren Details der jeweiligen Implementierung.

Implementierung der System-Ebene

Der Central Manager verwaltet alle Regionen und die daran beteiligten Rechner. Da alle Aktionen zentral über den Central Manager durchgeführt werden, kennt dieser zu jedem Zeitpunkt die aktuellen Zustände der an die Rechner vergebenen Ressourcen. Er ist verantwortlich für die Korrektheit der faulen Weitergabe der gemeinsamen Daten bei Anforderung von Sperren.
Zur Verwaltung der einzelnen Regionen existiert eine spezielle Datenstruktur. Die Regionen werden in einer doppelt verketteten Liste verwaltet, die für jede aktuelle Region einen Eintrag enthält. Darin enthalten sind im wesentlichen die Kennung der Region, eine Liste aller beteiligten Rechner und die Warteschlangen für die Vergabe von Sperren. Dabei wird für jede angeforderte Sequenznummer eine eigene Warteschlange verwaltet.

```
typedef struct global_region {
  int id;                       /* Kennung */
  int len;
  int seq;                      /* Sequenznummer */
  int seq_delta;
  region_state state;           /* Zustand der Region */
  int barrier;

  pvm_task_id passing_target;   /* Zur asynchronen Weitergabe */
  int passing_state;
  hostlist *attached;           /* Liste beteiligter Rechner */
  int lockers;                  /* Anzahl vergebener Sperren */
  sequence_wait *wait_queue;    /* Liste der Warteschlangen  */

  global_region_t *next, *prev;
} global_region_t;
```

Zur Umsetzung in DTS wurden sieben neue Nachrichten definiert, zwei davon zum Erstellen und Lösen einer Speicherassoziation, zwei weitere für den Erwerb und die Freigabe von Sperren. Die restlichen drei Nachrichten wurden für die Behandlung von Sonderfällen benötigt, die sich aus der faulen Weitergabe von Daten ergeben. Die Nachrichten sind in Tabelle 6.1 zusammengefaßt, in Abbildung 6.3 wird der Nachrichtenfluß an einigen Beispielen illustriert.

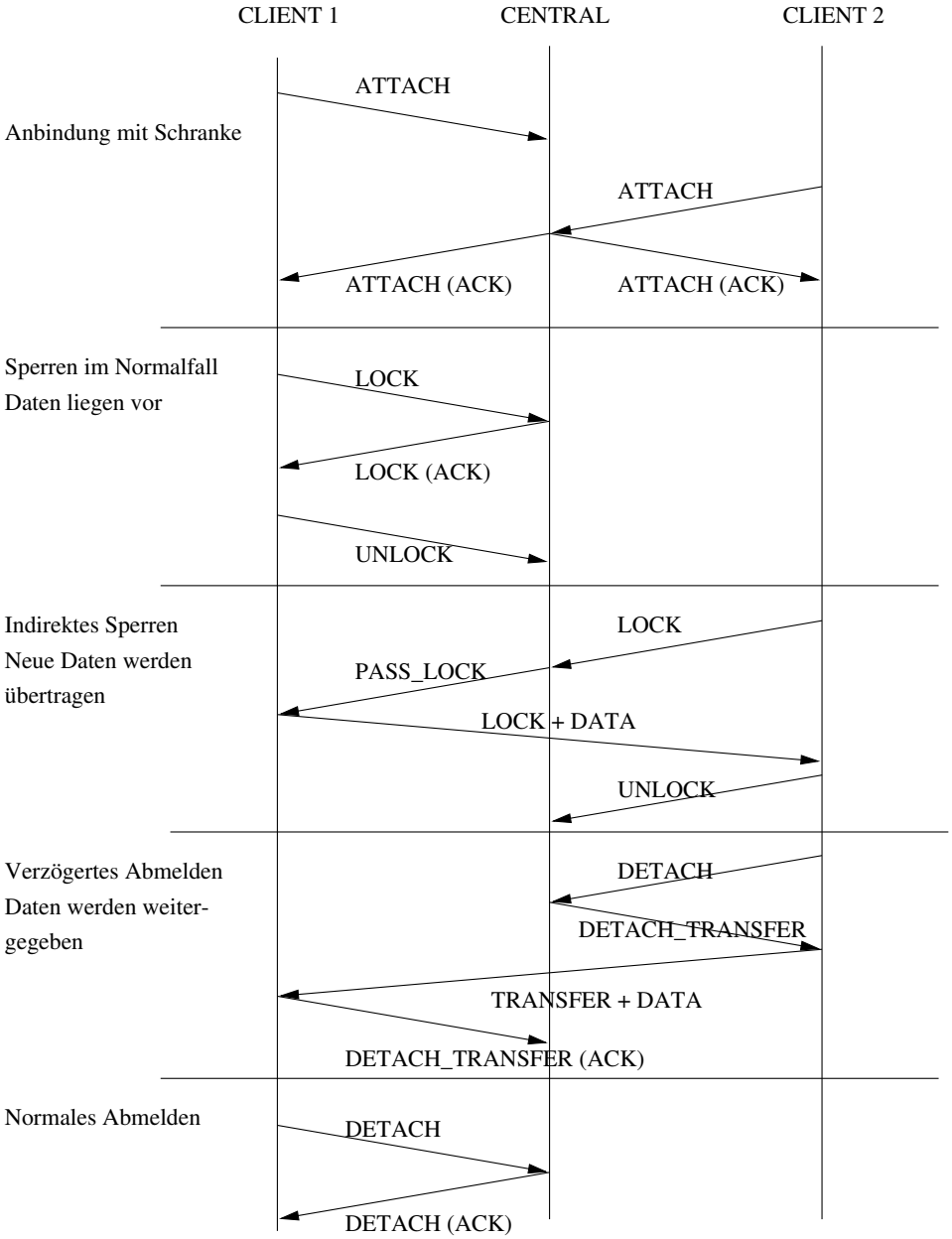

Abbildung 6.3: Beispiele zum Nachrichtenfluß der DSM-Implementierung

Nachrichtentyp	Richtung	Beschreibung
ATTACH	C←N	Anbindung an eine gemeinsame Speicherregion.
ATTACH	C→N	Bestätigung der Anbindung. Bei synchronisierter Anbindung gleichzeitig Freigabe der zugehörigen Schranke.
DETACH	C←N	Freigabe einer gemeinsamen Speicheregion.
DETACH	C→N	Bestätigung der Freigabe.
LOCK	C←N	Anforderung einer Sperre.
LOCK	C→N	Vergabe einer Sperre.
LOCK	N←N	Weitergabe einer Sperre mit aktuellen Daten.
UNLOCK	C←N	Entsperren einer Region.
PASS_LOCK	C→N	Indirekte Vergabe einer Sperre. Zielrechner fügt aktuelle Daten an.
DETACH_TRANSFER	C→N	Freigabe mit Datenweitergabe.
TRANSFER	N←N	Asynchrone Datenweitergabe.
DETACH_TRANSFER	C←N	Bestätigung der Weitergabe vom Empfänger.

Tabelle 6.1: DSM-Nachrichtentypen. In Spalte 2 ist die Senderichtung zwischen Central Manager (C) und Node Manager (N) angegeben.

Bei der Anbindung sendet der Central Manager, wenn die beim Aufruf angegebene Schranke erreicht wurde, jedem Rechner eine Nachricht zur Bestätigung.

```
void shm_attach(host caller) {
  if (new id) create_region();
  waiters++;
  if (waiters>barrier) {
    send MESS_SHM_ATTACH to all hosts;
  }
}
```

Ist die Sperre bei einer Anforderung frei, wird sie an den anfordernden Rechner vergeben, sonst wird dieser in die entsprechende Warteschlange eingetragen. Vor der Vergabe einer Sperre wird anhand des aktuellen Zustands festgestellt, ob ein Rechner veraltete Daten besitzt, und die Sperre gegebenenfalls *indirekt* vergeben. Der Central Manager fordert dazu ein Mitglied der Gruppe mit aktuellen Daten auf, diese zusammen mit der Sperre an den anfordernden Rechner weiterzuleiten. Diese Vorgehensweise vermeidet überflüssige Nachrichten und eine mehrfache Übertragung der Daten.
Die Freigabe der Sperre erfolgt ohne Bestätigung. Dabei werden die Warteschlangen überprüft, und eine erneute Vergabe angestoßen, sollte die Wartebedingung für einen oder mehrere der wartenden Rechner erfüllt sein.

```
void shm_lock(host caller) {
  if (caller owns a lock) error();
  if (caller does not wait for current sequence) {
    put_into_wait_queue(caller);
    return;
  }
  give_lock(caller);   /* else: host can get the lock */
}

void shm_unlock(host caller) {
  check();
  update_state();
  check_wait_queue();
}

void give_lock(host H) {
  if (H is uptodate) {
    /* simply grant access */
    send MESS_SHM_LOCK to H;
    return;
  } else {
    search uptodate host H';
    send MESS_PASS_LOCK to H';
  }
}
```

Die faule Weitergabe bringt in seltenen Fällen zusätzliche Komplikationen mit sich, die für eine korrekte Implementierung berücksichtigt werden müssen. Der Gewinn im Normalfall rechtfertigt jedoch den zusätzlichen Implementierungsaufwand. Dadurch, daß die gemeinsamen Daten erst bei Bedarf weitergegeben werden, kann der Fall eintreten, daß nur noch ein Rechner über aktuelle Daten verfügt. Löst dieser nun die Assoziation mit den anderen Rechnern, müssen vorher die aktuellen Daten asynchron an einen anderen, in der Gruppe verbleibenden Rechner weitergegeben werden. Bis diese Übergabe sicher beendet ist, muß der Central Manager die weitere Vergabe von Sperren verzögern. Der Empfänger bestätigt die Weitergabe der Daten an den Central Manager, der den ursprünglichen Rechner daraufhin aus der Rechnerliste entfernt.

```
void shm_detach(host caller) {
  if (caller waits for update) {
    mark host for pending detach;
    return;
  }
  if (caller has last uptodate copy and there are other hosts) {
    // request to send uptodate data to another host before detaching
    send MESS_SHM_DETACH_TRANSFER to caller
    return;
  }
```

```
  /* else: simple detachment */
  remove_host();
  send MESS_SHM_DETACH to caller;
  if (no more hosts) free_region();
}

void transfer_ack(host caller) {
  set caller uptodate;
  check wait queues;
}
```

Implementierung der Rechner-Ebene

Der Zugriff der lokalen Threads auf gemeinsame Speicherregionen wird vom Node Manager verwaltet. Dazu existiert, analog zum Central Manager, für jede Region eine Datenstruktur, die unter anderem eine Liste der beteiligten Threads und der zugehörigen Datenblöcken sowie Informationen zur Verwaltung der Sperren enthält. Da auf der Rechner-Ebene mehrere nebenläufige Threads um die vorhandenen Ressourcen konkurrieren können, mußte bei der Implementierung besonders auf die Vermeidung von Wettlaufbedingungen geachtet werden. Alle gemeinsam genutzten Strukturen werden deshalb mit Sperren geschützt, die in der folgenden Diskussion nicht explizit angeführt werden.

Die Anbindung an eine Region birgt keine Schwierigkeiten. Der Node Manager stellt, falls erforderlich, Speicher für den Benutzer bereit und sendet eine entsprechende Nachricht an den Central Manager. Der aufrufende Thread wartet, bis die angegebene Schranke, ausgelöst durch die Bestätigungsnachricht des Central Manager, erreicht wurde.

```
void *dts_shm_attach(int id, char *p, int len, int num) {
  if (p==NULL)
    create new memory block for user;

  if (new id) {
    create_new_region(id, p, len, num);
  }
  send MESS_SHM_ATTACH to CM;
  condition_wait(region_attached);
}

void attachment_ack() {
  condition_broadcast(region_attached);
}
```

Bei der Freigabe einer Region gilt es, den schon bei der Implementierung der zentralen Verwaltung angesprochenen Fall zu berücksichtigen, daß der aktuelle Rechner als letzter über aktuelle Daten verfügt. Deswegen wird, falls der letzte lokale Thread die Region freigibt, eine interne Kopie der Daten angelegt und die Zerstörung der Region bis zur Empfang der Bestätigung vom Central Manager verzögert. Je nach

Art der Bestätigung müssen zuerst die lokalen Daten an einen anderen Rechner verschickt werden, bevor die Region zerstört werden kann.

```
int dts_shm_detach(int id) {
  if (last thread detaching) {
    if (user memory) make_internal_copy();
  }
  delete thread and memory;
  if (no more local threads) {
    set region state to IN_DESTRUCTION;
    send MESS_SHM_DETACH to CM;
  }
}

/* Normal detachment */
void detachment_ack() {
  remove region;
}

/* Exceptional detachment: Only this host owns uptodate data.
   Send data before detaching */
void detachment_transfer(host target) {
  pack local data into message;
  send MESS_SHM_TRANSFER to target;
  remove region;
}
```

Die lokale Vergabe von Sperren an einzelne Threads wird vom Node Manager auf Rechnerebene verwaltet. Bei der ersten Anforderung wird eine entsprechende Nachricht an den Central Manager geschickt. Erhält der Rechner die Sperre, wird diese an die lokal wartenden Threads weitergegeben. Liegen keine lokalen Anforderung mehr vor, wird die Sperre an den Central Manager zurückgegeben.
Auf der lokalen Ebene werden keine Warteschlangen verwaltet, für jede Sequenznummer gibt es lediglich zwei Zähler für die Anzahl der Schreiber und Leser sowie dazugehörige Bedingungsvariablen. Muß ein Thread bis zur Vergabe einer Sperre warten, wird `condition_wait` auf der entsprechenden Bedingungsvariable ausgeführt. Das System weckt den Thread, wenn die Bedingungen für die Vergabe erfüllt sind, d.h. wenn die gewünschte Sequenznummer erreicht ist, die Daten aktualisiert wurden und keine Konkurrenz mit anderen Threads besteht. Durch diesen Ansatz werden explizite Warteschlangen vermieden und die Warteschlangen des zugrundeliegenden Thread-Systems genutzt.

```
int dts_shm_lock(int id, int type, int seq) {
  if (type==READ_LOCK) {
    if (first thread) {
      send MESS_SHM_LOCK(read) to CM;
      condition_wait(lock_avail);
    } else {
```

```
      if (lock is pending) condition_wait(lock_avail);
    }
   } else { /* type == WRITE_LOCK */
     if (first thread) send MESS_SHM_LOCK(write) to CM;
     condition_wait(lock_available);
   }
}

int dts_shm_unlock(int id, int type) {
  if (type==READ_LOCK) {
    if (last thread) {
      send MESS_SHM_UNLOCK(read) to CM;
      if (no more readers or writers waiting) free_sequence();
    }
  } else { /* type==WRITE_LOCK */
    synchronize local memory copies;
    send MESS_SHM_UNLOCK(write) to CM;
    if (no more readers or writers waiting) free_sequence();
  }
}

void got_lock() {
  if (data present) copy data to local memory;
  /* wakeup waiters */
  if (read_lock)
    condition_broadcast(lock_available);
  else /* write lock */
    condition_signal(lock_available);
}
```

Die beiden letzten Funktionen behandeln asynchrone, vom Central Manager ausgelöste Ereignisse. Bei einer *indirekten Vergabe* einer Sperre wird der Rechner aufgefordert, diese zusammen mit den lokalen Daten an einen anderen Rechner weiterzuschicken.

```
void pass_lock(host target) {
  pack data into message;
  send MESS_SHM_LOCK to target;
}
```

Die asynchrone Datenweitergabe ist Folge einer Loslösung des einzigen Rechners mit aktuellen Daten. Der Empfänger aktualisiert die lokalen Speicherbereiche und bestätigt dem Central Manager den erfolgreichen Abschluß der Weitergabe.

```
void transfer() {
  copy data to local memory;
  send MESS_SHM_DETACH_TRANSFER to CM;
}
```

6.6 Erweiterungsmöglichkeiten

Dem vorgesehenen Einsatz gemäß wurde kein besonderes Augenmerk auf die Steuerung von mehreren Schreibern gelegt. Zwar sind Schreibzugriffe immer sequentiell, dennoch fehlt der Implementierung eine Arbitrierungsmöglichkeit. Der Programmierer hat sich im voraus auf eine bestimmte Reihenfolge der Schreibzugriffe festzulegen. Diese Schwäche kann durch eine Verallgemeinerung des Wartemechanismus beseitigt werden, indem man Threads erlaubt, nicht nur auf das Eintreffen einer bestimmte Sequenznummer zu warten, sondern den Zugriff ermöglicht, wenn die aktuelle Sequenznummer *größer oder gleich* der geforderten ist. Dabei muß die Sequenznummer allerdings auf monotones Wachsen eingeschränkt werden. Der entstehende Mechanismus ist dann äquivalent zu den in [30] beschriebenen Ereigniszählern (engl. eventcounts).
Ein Ansatz zur lokalen Optimierung ergibt sich, wenn dem Benutzer die Möglichkeit gegeben wird, bestimmte Regionen als *offen* oder *geschlossen* zu deklarieren. Die Bedeutung deckt sich mit der von offenen bzw. geschlossenen Gruppen bei Gruppenkommunikation. Bei geschlossenen Gruppen ist nach der Erzeugung, d.h in diesem Fall nach Erreichen der bei der Anbindung angebenen Zahl von beteiligten Threads, kein weiterer Zugang mehr möglich. Das System kann mit dieser Information Nachrichten einsparen. Wurden etwa alle beteiligten Threads auf einem Rechner gestartet, müssen überhaupt keine Nachrichten mehr versandt werden, sind nur zwei Rechner beteiligt kann eine direkte Kommunikation erfolgen.
Durch den Verzicht auf eine explizite Anbindung und Loslösung von gemeinsamen Speicherregionen könnte eine Vereinfachung der Programmierung erreicht werden. Damit wären implizit alle Rechner an einer gemeinsamen Region beteiligt, ohne das dadurch zusätzliche Nachrichten notwendig würden, da bei der faulen Weitergabe ein Nachrichtenaustausch erst bei Anforderung einer Sperre erfolgt.
Ein wesentlich grundlegenderer Ansatz wäre ein systemnaher Konsistenzmechanismus, der über Speicherschutzmechanismen Seitenfehler abfängt[2] und sich implizit um die Synchronisation der beteiligten Rechner kümmert. Dieser Ansatz nimmt dem Programmierer die explizite Kopplung von Speicherregionen und Synchronisationsobjekten ab, erfordert dafür aber einen wesentlich höheren Implementierungsaufwand und aufwendige Algorithmen, soll dieselbe Leistung erreicht werden. Darüber hinaus muß die bisher in DTS vorhandene Möglichkeit der Verwendung inhomogener Rechnergruppen aufgegeben werden.
Beseitigt wäre damit auch ein anderes Problem der vorliegenden Implementierung. Derzeit können nur einfache, zusammenhängende Speicherbereiche geteilt werden. Diese Einschränkung könnte durch mehrere assoziierte Speicherbereiche etwas geweitet werden, das generelle Problem mit irregulären oder sogar überlappenden geteilten Daten kann aber nur mittels einem systemnahen Mechanismus gelöst werden.
Der Ansatz konnte aufgrund der Aufwands und der umfangreichen notwendigen Änderungen an DTS nicht weiter verfolgt werden und könnte Gegenstand einer weiterführenden Arbeit sein.

[2] Bei UNIX-Systemen dient dazu der `mprotect`-Systemaufruf

Kapitel 7

C++-Codegenerator

DTS unterstützt in der vorliegenden Form nur die Sprachen C und Fortran. Bei der Verwendung von C++ ist der Benutzer daher gezwungen, die bislang automatisch erzeugten Transferfunktionen wieder selbst bereitzustellen. Darüber hinaus können keine entfernten Methodenaufrufe von Objekten erfolgen, da es in DTS keine Möglichkeit gibt auf die zugehörigen Objektdaten zuzugreifen. Deshalb liegt als weiteres Ziel die Bereitstellung einer C++-Umgebung zur parallelen Programmierung nahe. Dieses System, die Distributed Concurrent Objects (DCO), wird derzeit von Tobias Grundmann verwirklicht [18]. Dazu wurde im Rahmen dieser Diplomarbeit der Codegenerator `CPPgen` entwickelt. Er ist ein Analogon zu `Cgen` in DTS, und in der Lage, sowohl Transferfunktionen für C++-Code auf Basis von DTS, als auch für DCO selbst zu generieren. Der für DTS erzeugte Code unterscheidet sich nur unwesentlich von durch `Cgen` erzeugtem Code, neu ist lediglich die Möglichkeit auch C++-Quellen zu verarbeiten. Die Konzepte dieser Codeerzeugung finden sich in [19]. Motivation sowie Begründung für die in DCO gewählten Ansätze sind in [18] beschrieben. An dieser Stelle wird lediglich auf die Implementierung und die Codeerzeugung für DCO eingegangen.
Der Parser baut auf der C++-Grammatik von James A. Roskind auf, die, abgesehen von fehlender Template-Unterstützung, den C++-Sprachumfang vollständig abdeckt.

7.1 Verwendung von `CPPgen`

Die Codegenerierung wird durch in den Quellcode eingebettete Kommentare gesteuert, die vom Codegenerator verarbeitet werden. Die Syntax deckt sich mit der bereits in `Cgen` verwendeten und soll deshalb hier nur in aller Kürze beschrieben werden. Eine Übersicht über die verwendeten Schlüsselworte und deren Bedeutung findet sich in Tabelle 7.1.
Das folgende Beispiel zeigt die Deklaration einer einfachen Funktion, die Werte eines Ganzzahl-Arrays summiert. Das Array wird explizit als Eingabeparameter deklariert und muß somit nicht mehr zurückübertragen werden. Desweiteren ist erkennbar, daß auch Arrays mit zur Übersetzungszeit unbekannter Größe versandt werden können, indem ein zugehöriger Längenparameter deklariert wird.

Schlüsselwort	Deklariert...
`pure`	eine seiteneffektfreie Funktion für die Code erzeugt wird.
`in`	einen Eingabeparameter der nur versandt wird.
`out`	einen Ausgabeparameter der nur empfangen wird.
`inout`	einen Ein- und Ausgabeparameter.
`dtslen`	einen Längenbezeichner für Array- oder Zeigertypen.
`dtsglen`	einen globalen Längenbezeichner für Array- oder Zeigertypen.

Tabelle 7.1: Schlüsselworte in `CPPgen`

Option	Beschreibung
`--c-mode`	Parsen von C-Quellcode
`--c-output-mode`	Erzeugen von Transferfunktionen für DTS
`-v S,S,...`	Ausgabe von Zusatzinformationen zu den angegebenen Programmteilen. Als Schlüsselworte S können `commands`, `parser`, `stack`, `symtab` oder `all` angegeben werden.
`--verbose`	

Tabelle 7.2: Zusätzliche Optionen von `CPPgen`

```
/*pure*/
int sum(int len, /*in*/int x[/*dtslen len*/], /*out*/ int *y);
```

Beim Aufruf von `CPPgen` kann zwischen Codeerzeugung für DTS oder DCO gewählt werden. Dazu dienen die Optionen `--c-mode` und `--c-output-mode`, mit denen 3 Modi selektiert werden können:

1. Option `--c-mode` (impliziert `--c-output-mode`)
   Erzeugung von DTS-Transferfunktionen aus C-Quellcode.

2. Option `--c-output-mode`
   Erzeugung von DTS-Transferfunktionen aus C++-Quellcode.

3. Ohne Optionen: Standard-Modus
   Erzeugung von reifizierten Klassen für DCO.

Die Modi 1 und 2 unterscheiden sich bisher lediglich in der Behandlung zusammengesetzter Typen, die in C++ im Unterschied zu C direkt ohne Angabe einer Identifikation (`struct, union, class, ...`) angesprochen werden können.
Alle von `Cgen` abweichenden Optionen sind in Tabelle 7.2 nochmals angeführt.

7.2 Erzeugung von Code für DCO

DCO erlaubt es, Objekte auf einem bestimmten Rechner einer virtuellen Maschine zu erzeugen. Die Objektdaten verbleiben dort während ihrer Existenz. Von jedem anderen Rechner können Mitgliedsfunktionen des Objekts aufgerufen werden. Dazu

wird beim Aufrufer ein Proxy-Objekt angelegt, das die Kommunikation und den Datentransfer zu dem entfernten Objekt kapselt. In der Praxis muß dazu für jede entfernt aufgerufene Methode eine eigene Klasse von DCO Standard-Klassen abgeleitet werden, in der Methoden zum Versand und Empfang der Parameter mittels des darunterliegenden Nachrichtensystems definiert sind. Bei Konstruktoren ist zudem eine Methode zur Erzeugung von Objekten, bei normalen Methoden eine zum Aufruf der entfernten Methode vom Programmierer bereitzustellen. Als Beispiel sei eine einfache Klasse angeführt:

```
class X {
public:
  X() : data(0) {}
  void inc(int skip) { data+=skip }

private:
  int data;
};
```

Soll die Methode `inc` entfernt aufgerufen werden, muß der Programmierer, wie in [18] ausführlich beschrieben, folgende Schnittstelle bereitstellen:

```
class X__inc : public Dco_reify<X,X__inc> {
public:
  X__inc(){}
  X__inc(int  arg_skip) : skip(arg_skip) {}
  void deserialize(Dco_msg_data& msg) {
    msg.extract(&(skip));
  }
  void serialize(Dco_msg_data& msg) const {
    msg.insert(&(skip));
  }
  void call(void) {
    get_object()->inc(skip );
  }
private:
  int  skip;
};
```

Die Erzeugung dieses Code wird von `CPPgen` automatisch durchgeführt, wenn die entsprechende Methode als `pure` deklariert wird, im obigen Beispiel muß also lediglich die Deklaration der Methode `inc` in

```
  /*pure*/ void inc(int skip) { data+=skip; }
```

abgewandelt werden. Danach erzeugt ein Aufruf von `CPPgen` mit der Quelldatei als einziges Argument die reifizierten Klassen. In obigem Beispiel würden im Unterverzeichnis `DCO` die Dateien `dco_X__inc.C` und `dco_X__inc.h` erzeugt. Der Benutzer muß die erzeugten Dateien zusätzlich übersetzen und binden. Ein Compiler-Frontend, analog zu `dtscc`, wird diese Arbeit in Zukunft automatisieren.

7.3 Implementierung

Da sich die Implementierung an `Cgen` anlehnt und dessen Funktionalität auf der C-Untermenge in `CPPgen` übernommen wurde, sollen hier nur in aller Kürze die für C++ veränderten Konzepte angesprochen werden.
Um Transferfunktionen für Parameter automatisch zu erzeugen, muß der gesamte Quellcode geparst und dabei eine Typtabelle aufgebaut werden. Die Typen müssen bei einem späteren Zugriff vollständig aus dieser Information rekonstruierbar sein. Dazu werden interne Strukturen für Bezeichner und Typen angelegt, die während des Parsen in eine Symboltabelle eingetragen und geeignet verkettet werden. So enthält beispielsweise eine Klasse oder Struktur eine Liste ihrer Mitglieder, die selbst wiederum aus komplexen Typen aufgebaut sein kann, oder eine Funktion eine Liste ihrer Argumente. Bei Klassen wird auch die Ableitungshierarchie intern vermerkt.
Ein Beispiel für die Anforderungen von C++ an das Typsystem zeigt der folgende Programmausschnitt, bei dem der Typ `T` bei Verwendung in der Klasse `X` auf drei verschiedene Weisen aufgelöst werden kann:

```
1  typedef int T;
2
3  class S {
4    typedef char T;
5  };
6
7  class X {
8  public:
9    typedef double T;
10
11   T x1;
12   ::T x2;
13   S::T x3;
14 };
```

Die Verwendung in Zeile 11 greift auf die lokale Umdefinition zurück und wird als `double` aufgelöst, während in Zeile 12 die Definition `int` aus dem globalen Namensraum verwendet wird. In Zeile 13 muß der Parser den Bezug auf die lokale Definition einer anderen Klasse auflösen und den Typ `char` generieren.
Damit war als erster, grundlegender Schritt in der Verwaltung von Bezeichnern und Typen eine Abkehr von der „flachen“ Typverwaltung in `Cgen` notwendig. Während des Parsevorgangs wird mit einer Hierarchie verschachtelter Gültigkeitsbereiche gearbeitet, um die Zuordnung eines Typbezeichners zum richtigen Typ zu gewährleisten. Zudem werden sämtliche Typen, auch mehrfache Zeiger und lokale Typdefinitionen, komplett in die interne Repräsentation übernommen. Darüber hinaus wurde die C++-Grammatik zur Verarbeitung der neuen, in Kommentare eingebetteten Schlüsselworte ergänzt.

Kapitel 8

Zusammenfassung und Ausblick

In dieser Diplomarbeit wurde die Parallelisierbarkeit einer konkreten Anwendung zur Simulation eines physikalischen Systems untersucht. Die Analyse der Algorithmen und die gemessenen Laufzeiten machten die Grenzen der hier erreichbaren Geschwindigkeitssteigerung deutlich. Die Gründe dafür liegen einerseits im hohen, teilweise versteckten, sequentiellen Anteil der Berechnung sowie der feinen Körnung der vorhandenen Parallelität und dem damit verbundenen Overhead. Auf der anderen Seite stand die Schwierigkeit, die objektorientierte Programmierung und eine praktische Implementierung von verteilten Algorithmen in DTS zu ermöglichen. Der zweite Problempunkt wurde durch die Erweiterung des Codegenerators `Cgen` [19] zur Verarbeitung von C++-Quellen gemildert und wird mit der Fertigstellung von DCO [18] in Kombination mit diesem Codegenerator überwunden sein. Mit der Integration von Distributed Shared Memory in DTS wurde die Grundlage für eine einfache Implementierung von verteilten Algorithmen geschaffen.
Aus den Erfahrungen die im Verlauf dieser Arbeit gewonnen wurden, ergeben sich im wesentlichen drei zentrale Ansatzpunkte für zukünftige Arbeiten:

- Kombination des Prinzips der verteilten Thread-Programmierung mit aktuellen DSM-Konzepten.

  Nimmt man das Konzept des Distributed Shared Memory als Grundlage des Thread-Systems, entfallen die bisherigen Einschränkungen in DTS. Das Ziel wäre eine universelle, transparente Thread-Programmierumgebung für verteilte Rechner.

- Entwicklung eines Systemmodells mit dynamischer Parallelisierung.

  Die Ergebnisse der Diplomarbeit zeigen, daß die Unterschiede in den Architekturen immer noch einen erheblichen Einfluß auf die Art der Parallelisierung haben. Mit einem universellen Architekturmodell, das eine Voraussage der Laufzeiten und Kommunikationskosten erlaubt, könnten auf Basis desselben Quellcodes an jeweilige Architektur angepaßte Programme erzeugt werden, bei denen nur diejenigen Teile parallel ausgeführt werden, die einen Gewinn versprechen.

- Bereitstellung paralleler numerischer Bibliotheken

  Die nachträgliche Parallelisierung von Anwendungen, die auf numerische Algorithmen zurückgreifen ist aufwendig und fehlerträchtig. Durch die Bereitstellung von Bibliotheken zur Behandlung spezieller numerischer Probleme, etwa der Lösung von gewöhnlichen Differentialgleichungen, könnte auf Basis des obigen Gesamtsystems eine einheitliche Programmierschnittstelle bereitgestellt werden, die bei Bedarf ohne weiteren Programmieraufwand eine parallele Ausführung erlaubt.

Anhang A

Referenz der DSM-Primitive

```
/*****************************************************************
   Herstellen und Lösen einer Verknüpfung
 *****************************************************************/
void *dts_shm_attach(int id,
                     char* memory, int length, int barrier);
int dts_shm_detach(int id);

/*****************************************************************
 * Zugriff auf gemeinsame Regionen
 *****************************************************************/
/* Zugriffsarten */
#define SHM_READ_LOCK    0x1
#define SHM_WRITE_LOCK   0x2

/* Erwerb und Freigabe der Region */
int dts_shm_lock(int id, int type, int sequence);
int dts_shm_unlock(int id, int type);

/* Makros */
#define dts_shm_readlock(m,s)  dts_shm_lock(m,SHM_READ_LOCK,s)
#define dts_shm_writelock(m,s) dts_shm_lock(m,SHM_WRITE_LOCK,s)
#define dts_shm_readunlock(m)  dts_shm_unlock(m,SHM_READ_LOCK)
#define dts_shm_writeunlock(m) dts_shm_unlock(m,SHM_WRITE_LOCK)

/*****************************************************************
   Zugriff auf Sequenznummern
 *****************************************************************/
int dts_shm_setseq(int id, int sequence);
int dts_shm_getseq(int id);
```

Danksagung

Dank gebührt Prof. Rosenstiel, an dessen Lehrstuhl diese Diplomarbeit durchgeführt wurde, sowie den Betreuern Tilmann Bubeck und Bernd Eberhardt für ihre zahlreichen Anregungen und Ratschläge. Ebenfalls möchte ich mich bei Prof. Bülthoff vom Max-Planck-Institut in Tübingen für die Gelegenheit bedanken, auf der dort betriebenen ONXY2 rechnen zu können. Auch die Unterstützung des dortigen Systemverwalters Phillip Georg bei technischen Problemen aller Art war sehr hilfreich.

Literaturverzeichnis

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Proc. of the SJCC*, volume 30, pages 483–485, 1967.

[2] C. Amza, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM protocols that adapt between single writer and multiple writers. In *Proc. of the 3rd High Performance Computer Architecture Conf.*, pages 261–271, Feb. 1997.

[3] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 528–537, Feb. 1993.

[4] T. Bubeck. Eine Systemumgebung zum verteilten funktionalen Rechnen. Technical Report WSI-93-8, Eberhard-Karls-Universität Tübingen, August 1993.

[5] T. Bubeck. *Distributed Threads System DTS User's Guide.* SFB 382/C6, Universität Tübingen, Sep 1995.

[6] K. Burrage and J. Butcher. Parallel methods for initial value problems. *Applied Numerical Mathematics*, 11:5–25, 1993.

[7] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP'91)*, pages 152–164, Oct. 1991. Also available as Rice University, Dept. of Computer Science technical report COMP TR91-150.

[8] M. Dubois, C. Scheurich, and F. A. Briggs. Memory Access Buffering in Multiprocessors. In *Proc. of the 13th Annual Int'l Symp. on Computer Architecture (ISCA'86)*, pages 434–442, June 1986.

[9] B. Eberhardt, A. Weber, and W. Strasser. A fast, flexible, particle-system model for cloth draping. *IEEE Computer Graphics and Applications*, 16(5):52–60, September 1996.

[10] J. Fier. *Performance Tuning and Optimization for Origin 2000 and ONYX2.* SGI, first edition, 1996.

[11] M. J. Flynn. Very high speed computing systems. In *Proceedings of IEEE*, volume 54:12, pages 1901–1909, Dec. 1966.

[12] M. J. Flynn. Some computer organizations and their effectiveness. In *IEEE Trans. on Computers*, volume C-21, pages 948–960, Sept. 1972.

[13] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM 3 User's Guide and Reference Manual.* Oak Ridge National Laboratory, Tennessee, September 1994.

[14] K. Gharachorloo, D. E. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 15–26, May 1990.

[15] J. R. Goodman and P. J. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. In *Proc. of the 15th Annual Int'l Symp. on Computer Architecture (ISCA'88)*, pages 422–431, May 1988.

[16] M. W. Goudreau, J. M. D. Hill, K. Lang, B. McColl, S. B. Rao, D. C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for the BSP worldwide standard library. Technical report, April 1996.

[17] H. A. Grosch. Grosch's law revisited. *Computerworld*, 8(16):24, April 1975.

[18] T. Grundmann. Distributed Concurrent Objects. Ein objektorientiertes System zum verteilten Rechnen. Master's thesis, Universität Tübingen, Technische Informatik, 1997.

[19] T. Grundmann, T. Bubeck, and W. Rosenstiel. Automatisches Generieren von DTS-Kopierfunktionen für C. Technical Report 48, Universität Tübingen, SFB 382, July 1996.

[20] S. Hüttemann. Methoden zur Parallelisierung von Smoothed Particle Hydrodynamics. Master's thesis, Universität Tübingen, Institut für Astronomie und Astrophysik, 1996.

[21] IBM Corp. *IBM PVMe for AIX, User's Guide and Subroutine Reference*, first edition, October 1995.

[22] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter Usenix Conf.*, Jan. 1994.

[23] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory . In *Proc. of the 18th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.

[24] R. Klein and P. Slusallek. Object-oriented framework for curves and surfaces. In J. Warren, editor, *Curves and Surfaces in Computer Vision and Graphics III*, pages 284–295. SPIE, November 1992.

[25] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, C-28(9):690–691, Sept. 1979.

[26] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proc. of the 5th ACM Annual Symp. on Principles of Distributed Computing (PODC'86)*, pages 229–239, Aug. 1986.

[27] H. Lu, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *Proc. of the 6th Symposium on Principles and Practice of Parallel Programming*, June 1997. To appear.

[28] Message Passing Interface Forum. MPI: A message passing interface. In *Proc. Supercomputing '93*, pages 878–883. IEEE Computer Society, 1993.

[29] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, New York, second edition, 1992.

[30] D. P. Reed and R. K. Kanodia. Synchronization with eventcounts and sequencers. *Comm. of the ACM*, 22:115–123, 1979.

[31] R. Sonntag. *RadioLab: An object oriented global illumination system.* Dissertation, Universität Tübingen, Tübingen, Germany, 1997. Forthcoming.

[32] K. Strehmel and R. Weiner. *Numerik gewöhnlicher Differentialgleichungen.* Teubner, 1995.

[33] B. Stroustrup. *The C++ programming language.* Addison-Wesley, second edition, 1992.

[34] A. S. Tanenbaum. *Modern operating systems.* Prentice-Hall Inc., 1992.