Plattformunabhängige Umgebung für verteilt paralleles Rechnen mit Rechnerbündeln

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der Fakultät für Informatik
der Universität Fridericiana zu Karlsruhe (TH)

genehmigte

Dissertation

von

Bernhard Haumacher

aus Pforzheim

Tag der mündlichen Prüfung:	25. Oktober 2005
Erster Gutachter:	Prof. Dr. Walter F. Tichy
Zweiter Gutachter:	Prof. Dr. Michael Philippsen

Inhaltsverzeichnis

Zusammenfassung **vii**

Abstract **ix**

1 Einleitung **1**
1.1 Konzepte für verteilt paralleles Rechnen 3
1.1.1 Nachrichtenaustausch 3
1.1.2 Entfernte Programmausführung 5
1.1.3 Virtuell gemeinsamer Speicher 6
1.1.4 Generative Kommunikation 8
1.2 Thesen 9
1.3 Ziel dieser Arbeit 9
1.4 Gliederung 11

2 Verwandte Arbeiten **13**
2.1 Anbindung plattformabhängiger nativer Bibliotheken 13
2.1.1 Generierung nativer Bibliotheksanbindungen 15
2.1.2 Native Kommunikationsbibliotheken 15
2.2 Java-Bibliotheken für verteilt paralleles Rechnen 15
2.2.1 Java-Bibliotheken zum Nachrichtenaustausch 16
2.2.2 Java-Bibliotheken zum entfernten Methodenaufruf 16
2.2.3 Gemeinsam benutzte Objekte 18
2.2.4 Generative Kommunikation 19
2.2.5 Bibliotheken für Grid-Computing 19
2.3 Präprozessorbasierte Ansätze 22
2.4 Lösungen mit konventionellem Übersetzer 24
2.5 Native Implementierungen einer verteilten virtuellen Maschine 25
2.5.1 Hauptspeicherorganisation 25
2.5.2 Objektmodell 26
2.5.3 Einordnung relevanter Arbeiten 26
2.5.4 Bewertung des Ansatzes einer verteilten virtuellen Maschine . 27
2.6 Replikation 28
2.6.1 Replikation zur Lokalitätsoptimierung 31
2.6.2 Verbindung von Kontroll- und Datenparallelität 36
2.6.3 Replikation für Fehlertoleranz 38

3 Konzepte der Programmierumgebung **39**
3.1 Basissprache und virtuelle Maschine 39
3.2 Bündellaufzeitsystem 41
3.3 Parallelitätsmodelle 41
3.3.1 Kontrollparallelität 41
3.3.2 Datenparallelität 43
3.4 Verteilter Objektraum 46
3.4.1 Das RMI-Modell 46
3.4.2 Das JavaParty-Modell 48
3.4.3 Objekte in der verteilten Umgebung 48
3.5 Grenzen der Einsetzbarkeit entfernter Objekte 50
3.5.1 Optimierung entfernter Zugriffe 50
3.5.2 Gemeinsam benutzte, aber selten modifizierte Daten 51
3.5.3 Verteilte Datenstrukturen 52
3.5.4 Koordination bei verteilten Datenstrukturen 53
3.6 Kollektive Replikation 54
3.6.1 Replikation für Algorithmen im BSP-Modell 55
3.6.2 Datenparallelität auf irregulären Datenstrukturen 57
3.6.3 Kollektiv replizierte Objekte 58
3.7 Integration der Konzepte 62
3.7.1 Transparent entfernte Klassen 63
3.7.2 Transparent maschinenüberspannende Kontrollfäden 64
3.7.3 Lokale Klassen 64
3.7.4 Transparent replizierte Klassen 65
3.8 Zusammenfassung 67

4 Schnelle Objektserialisierung **69**
4.1 Die Java-Objektserialisierung 71
4.1.1 Eigenschaften 71
4.1.2 Einschränkungen 72
4.2 Optimierungen am Serialisierungsprotokoll 74
4.2.1 Schlanke Typcodierung und Wiederverwendung 74
4.2.2 Spezialisierte Versenderoutinen 75
4.2.3 Eingeschränkte Sicherheitsgarantien 75
4.2.4 Transportschicht-Anbindung 76
4.2.5 Nichtrekursives Serialisieren 77
4.3 Optimierungen in den Serialisierungsmethoden 79
4.3.1 Einsparung von Zyklentests 79
4.3.2 Einsparung von Typinformation 81
4.4 Kompatibilität 82
4.4.1 Externe Serialisierungsmethoden 82
4.4.2 Rückfall auf die Standardserialisierung 83
4.5 Architektur 87
4.5.1 Die Objekt-Versendeschnittstelle 87
4.5.2 Technologieneutrale Abstraktionsschicht 89
4.5.3 Anbindung serialisierbarer Klassen 91

4.5.4 Ablauf von Serialisierung und Deserialisierung 91
4.6 Automatische Code-Erzeugung 94
4.6.1 Benutzerdefinierte Funktionalität während der Serialisierung . 95
4.6.2 Generierte Versenderoutinen 96
4.6.3 Generierte Kopiermethoden 100
4.6.4 Generierte Deskriptorklassen 101

5 Transparente effiziente Kommunikation 103
5.1 Einschränkungen bei Java-RMI 104
5.1.1 Effizienz 104
5.1.2 Transparenz 105
5.2 Entwurfsverbesserungen und Optimierungen 106
5.2.1 Steckbare Transporttechnologien 106
5.2.2 Exportpunkte statt TCP/IP-Ports 107
5.2.3 Effizienz durch Aufrufkontexte 108
5.2.4 Erkennung und Ausnutzung von Lokalität 111
5.3 Transparent maschinenüberspannende Kontrollfäden 113
5.3.1 Konsequenzen der segmentweisen Zusammensetzung 114
5.3.2 Transparente Synchronisation 115
5.3.3 Zuordnung von lokalen Kontrollfäden zu Segmenten 116
5.3.4 Transparente Unterbrechungen 118
5.4 Entfernte Sperranforderung 119
5.4.1 Problematik entfernter Sperranforderung 120
5.4.2 Sperranforderung über lokale Stellvertreter 121
5.4.3 Kompatibilität mit regulärer Synchronisation 122
5.5 Transparent entfernte Klassen 123
5.5.1 Transformation entfernter Klassen 123
5.5.2 Synchronisation 128
5.5.3 Statische Konstanten 130
5.5.4 Annotation der Objekterzeugung 134

6 Transparente Replikation 137
6.1 Transparent replizierte Klassen 137
6.1.1 Replizierter Zustand 139
6.1.2 Konsistenz 141
6.1.3 Lese- und Schreibsynchronisation 142
6.1.4 Kommunikation zwischen Kontrollfäden 145
6.1.5 Kollektive Synchronisation 146
6.1.6 Partielle Replikation 148
6.1.7 Felder als Teil des replizierten Zustandes 153
6.2 Erweiterungen in Objektserialisierung und Kommunikationsbibliothek 155
6.2.1 Replikatverwaltung 158
6.2.2 Sperroperationen 159
6.2.3 Änderungserkennung 162
6.2.4 Zustandsfortschreibung 165
6.2.5 Verschmelzen überlappender Änderungen 170

6.3 Transformation 172
6.3.1 Klassen und Instanzen 172
6.3.2 Klasseninitialisierung 173
6.3.3 Objekterzeugung 174
6.3.4 Koordination 174

7 Evaluation **177**
7.1 Evaluationsumgebung 177
7.2 Benchmarkkerne 177
7.2.1 Namensgebung 177
7.2.2 Objektserialisierung 179
7.2.3 Fernaufruf 182
7.2.4 Vergleich mit Manta/Ibis 184
7.2.5 Potenzieller Fernaufruf 184
7.2.6 Transparent entfernte Klassen 186
7.2.7 Transparente Kontrollfäden 188
7.2.8 Exklusive Synchronisation 190
7.2.9 Kollektive Synchronisation 192
7.3 Evaluation von Anwendungen 194
7.3.1 TSP 194
7.3.2 N-Body 200
7.3.3 Impact 206

8 Zusammenfassung und Ausblick **209**

A Inkrementelles verteiltes Färben von Graphen **213**
A.1 DLF-Algorithmus 213
A.2 Erweiterung zu IDLF 215
A.3 Aufwandsabschätzung 218

Literaturverzeichnis **219**

Zusammenfassung

Erstmals im Juni 2004 stellen Rechnerbündel mit 291 der 500 weltweit schnellsten Computer die Mehrheit der Supercomputer [70]. Damit hat sich die Bündelarchitektur als Plattform für das Hochleistungsrechnen durchgesetzt. Die Entwicklung von Sprachen und Programmierumgebungen für solche verteilt parallelen Systeme hat mit dieser Entwicklung allerdings nicht Schritt gehalten. Der bislang unangefochtene Standard bei der Entwicklung von Anwendungen für das Hochleistungsrechnen ist eine Kombination aus einer prozeduralen sequentiellen Programmiersprache wie Fortran oder C mit nachrichtenbasierter expliziter Kommunikation (z. B. MPI [26]). In anderen Anwendungsgebieten wurden prozedurale Sprachen längst durch objektorientierte wie C++ oder Java abgelöst, da Software damit schneller, mit weniger Fehlern und wartbarer entwickelt werden kann. Eine andere Entwicklung, die mit dem Erscheinen von Java eingesetzt hat, geht dahin, portable Programme zu schreiben, die ohne Neuübersetzung auf einer Vielzahl von Prozessorarchitekturen und Betriebssystemen sofort lauffähig sind.

Die vorliegende Arbeit macht sowohl Objektorientierung als auch Portabilität für verteilt paralleles Programmieren von Rechnerbündeln nutzbar. Dazu wurde Java um verteilungsrelevante Aspekte erweitert. Durch die Verwendung von transparenten entfernten Objekten kann der Programmierer ein paralleles Programm für ein Rechnerbündel ebenso wie für eine einzige virtuelle Java-Maschine schreiben. Kollektiv replizierte Objekte bieten neben der Optimierung verteilt paralleler Lesezugriffe ein objektorientiertes Ausdrucksmittel für datenparallele Operationen im bulk-synchronen parallelen Modell. Programme, die diesem Parallelitätsmodell entsprechen, sind in Hochleistungsanwendungen verbreitet, da ein hoher Parallelitätsgrad erreichbar ist und große Datenmengen verarbeitet werden können. In der vorliegenden Arbeit ermöglicht eine neue Replikationsform, die sog. kollektive Replikation, eine Kombination der aus Java bekannten und gut in objektorientierte Sprachen integrierbaren Kontrollparallelität und der in nachrichtenbasierten Programmen verbreiteten Datenparallelität. Dabei handelt es sich um eine wesentliche Neuerung, da Datenparallelität bisher nur auf reguläre Datenstrukturen wie (mehrdimensionale) Felder anwendbar war. Mit kollektiver Replikation wird Datenparallelität in eine objektorientierte Sprache integrierbar, ohne in Konflikt mit der Vererbung, der Modularität oder dem Geheimnisprinzip zu geraten.

Sämtliche Spracherweiterungen werden über eine Programmtransformation in reines Java zurückübersetzt. Dadurch wird volle Portabilität des erzeugten Codes garantiert. In der erweiterten Sprache bleiben alle Schlüsselkonzepte, die Java populär gemacht haben, wie einfache Objektorientierung, automatische Speicherbereinigung, eingebaute Parallelität und Koordination erhalten und unverändert in der verteilten Umgebung nutzbar. Das ermöglicht insgesamt eine einfache Programmierung von

Rechnerbündeln, ohne auf den Komfort einer objektorientierten Sprache verzichten zu müssen. Anhand eines Prototyps für die Programmierumgebung wird gezeigt, dass die entwickelten Konzepte zu einer besonders einfachen Portierung von parallelen Anwendungen auf verteilte Bündelarchitekturen führen. Wenn sich das Problem datenparallel zerlegen lässt, vereinfacht kollektive Replikation sogar die verteilte Parallelisierung sequentieller Programme. Dazu muss der Algorithmus selbst nicht verändert werden. Die verarbeiteten Datenstrukturen werden gesteuert durch eine Annotation automatisch transformiert, so dass Operationen bereitstehen, mit denen nach einer datenparallelen Modifikation ohne weiteren Codieraufwand die Konsistenz wiederhergestellt werden kann. Die Transformation stützt sich auf einen entfernten Methodenaufruf für Rechnerbündel, der ebenfalls im Rahmen der vorliegenden Arbeit entwickelt wurde und Primitive bereitstellt, auf welche kollektive Replikation und transparenter Fernzugriff abgebildet werden. Beides ermöglicht eine effiziente Ausführung rechenintensiver Anwendungen auf Rechnerbündeln.

Abstract

In June 2004, clusters obtained a majority with 291 out of the 500 world's fastest supercomputers [70]. The cluster architecture is now the established platform for high-performance computing. Unfortunately, the design of languages and runtime systems for those distributed systems did not keep up with this evolution. The unchallenged standard for high-performance application development is still a combination of a procedural language like Fortran or C with explicit message passing communication like MPI [26]. In other areas, procedural languages have been replaced for a long time with object-oriented ones like C++ or Java. The object-oriented paradigm makes application development faster, less error prone, and better maintainable. Another trend, which has started with the appearance of Java, is to write portable applications that run on a variety of processor architectures and operating systems without recompilation.

The work at hand makes object orientation as well as portability usable for parallel distributed programming of cluster computers. Therefore, Java was extended with aspects relevant to distribution. With transparent distributed objects, a programmer can write a parallel Program for a cluster like for a single virtual machine. Besides optimizing parallel distributed read access, collective replicated objects are a new object-oriented way of expressing data-parallel operations in the bulk-synchronous model. Data-parallel algorithms are widespread in high-performance computing applications, because they can reach a high degree of parallelism and are able to process large amounts of data. Collective replication is a new form of object replication, which was invented in the work at hand. It allows a seamless integration of control and data parallelism in an object-oriented language. This is an important contribution, since control parallelism was believed to match object-orientation, while data parallelism was restricted to array structures in procedural languages with explicit message passing. Collective replication integrates data parallelism into an object oriented language without conflicting with inheritance, modularization or encapsulation.

All language extensions are automatically transformed back to pure Java. This ensures full portability of the generated code. In the extended language, all concepts like easy object orientation, garbage collection, built-in parallelism and coordination, which made Java popular, are still usable in the distributed environment. This enables easy programming of cluster computers without abstaining from the comfort of an object-oriented language. A prototype shows that the extensions make porting of a parallel application to a distributed cluster environment particularly easy. If there is a data-parallel decomposition of the problem, collective replication even eases the distributed parallelization of sequential programs. With collective replication, the parallelization does not require modifications to the algorithm itself. Guided through an annotation, the data structures are automatically transformed to provide operations

for reestablishing consistency after a data-parallel modification. There is no need for any additional coding. The transformation is based on a library for extended remote method invocation for cluster computers, which was also developed within the scop of the work at hand. Extended remote method invocation provides communication primitives for collective replication and transparent remote access. Both enable the efficient execution of computing-intensive application in clusters.

Kapitel 1

Einleitung

Die Forschung beschäftigt sich schon seit gut zwei Jahrzehnten mit der systemtechnischen und programmiersprachlichen Beherrschung von Parallelrechnern ohne gemeinsamen Speicher. Dabei ruht seit gut einem Jahrzehnt die Hoffnung auf vernetzten Arbeitsplatzrechnern als kostengünstigem Ersatz für spezialisierte Parallelrechner, um den wachsenden Bedarf an Rechenleistung zu befriedigen. Diese Hoffnung gründet sich auf der Annahme, dass die Fertigungskosten für solche in Massenproduktion hergestellten Komponenten wesentlich günstiger sind, da sich die Entwicklungskosten in kürzeren Zeiträumen amortisieren und sich dadurch schnellere Innovationszyklen erreichen lassen. Allerdings ist noch nicht endgültig geklärt, wie sich die durch die relativ lose Kopplung der Systeme erhöhte Komplexität so gut beherrschen lässt, dass die günstigeren Anschaffungskosten nicht durch erhöhte Entwicklungskosten für die darauf eingesetzte Software aufgewogen werden, aber trotzdem ein effizientes Gesamtsystem entsteht. Die vorliegende Arbeit versteht sich als ein Teil zur Lösung des Problems der programmiertechnischen Beherrschbarkeit von Hochleistungsrechnern, bestehend aus gekoppelten Arbeitsplatzrechnern, kurz Rechnerbündeln.

Als ein wirksames Mittel zur Beherrschung von Komplexität hat sich die Ablösung strukturierter Programmiersprachen durch objektorientierte Sprachen herausgestellt. Die Forschung in den vergangenen zwanzig Jahren hat sich daher intensiv damit beschäftigt, wie die Vorteile von Objektorientierung mit einer effizienten Parallelverarbeitung zu vereinen sind. In einem Übersichtsartikel [81] hat Philippsen insgesamt 111 objektorientierte Sprachen untersucht, die in dem Zeitraum von 1981 bis 1997 vorgeschlagen wurden und die Objektorientierung mit Parallelverarbeitung verbinden. Von den untersuchten Vorschlägen zielen aber nur knapp die Hälfte auf *verteilt* parallele Systeme, welche aus einer Zusammenschaltung von Prozessoren ohne gemeinsamen Adressraum bestehen. Allerdings lässt sich auch bei den an dieser Stelle untersuchten Sprachen ein klarer Trend in Richtung auf verteilt paralleles Rechnen ausmachen. Von neunzehn in [81] zitierten Publikationen, die bis 1988 erschienen sind, behandelten nur drei eine Sprache, die auf verteilt paralleles Rechnen abzielt. Dagegen beschäftigen sich 1993 schon 33 von 47 Publikationen mit einer Sprache für ein verteiltes System.

Neben dem Trend hin zur Nutzung von Rechnerbündeln als Ersatz für spezialisierte Parallelrechner mit gemeinsamem Speicher und der Entwicklung von Hochsprachen, welche die daraus resultierende Systemarchitektur bestmöglich unterstützen, kristalli-

sierte sich eine weitere Stoßrichtung heraus, die sich größtmögliche Portabilität von Anwendungen zum Ziel setzt. Mit der Entwicklung von Java, einer Sprache, die ab 1991 bei SUN Microsystems entworfen und 1995 erstmals öffentlich vorgestellt [73] wurde, begann ein regelrechter Boom *mobilen Codes*. Unter mobilem Code versteht man ein Programm, das nicht für eine bestimmte Architektur fertigübersetzt und daher auf eine Kombination aus Hardware und Betriebssystem festgelegt ist. Ein solches Programm liegt in einer plattformunabhängigen Form vor und kann auf jedem Rechner in einer virtuellen Maschine (VM) ausgeführt werden. Der virtuellen Maschine steht es frei, das Programm bei der Ausführung zu interpretieren, beim Laden fertigzuübersetzen oder Teile bei Bedarf in die Maschinensprache des ausführenden Rechners zu transformieren. Die Sprache Java, in der solche plattformunabhängigen Programme geschrieben werden können, war ursprünglich als vereinheitlichende Plattform für die Programmierung von Endgeräten einer neuen Generation konzipiert. Den ersten Einsatz fand mobiler Code aber in interaktiven Webseiten. Java wird in einen maschinenunabhängigen Zwischencode übersetzt, welcher von der virtuelle Maschine ausgeführt wird. Dieser sog. Bytecode ist einerseits unabhängig von der ausführenden Zielmaschine und garantiert andererseits eine sichere Ausführung, da beim Empfänger die von der Sprache festgelegten Zugriffsregeln und Typüberprüfungen erneut durchgeführt werden können. Da Java weitere moderne Eigenschaften wie Objektorientierung, automatische Speicherbereinigung und die direkte Unterstützung von Parallelverarbeitung über nebenläufige Kontrollfäden in sich vereint, ist nicht verwunderlich, dass Java schon sehr bald auch für herkömmliche Anwendungen auf Klient und Dienstgeber eingesetzt wurde. Die dadurch erreichte kritische Masse beflügelte die Entwicklung der Sprache und ihrer Laufzeitumgebung derart, dass der anfänglich drastische Vorsprung konventionell übersetzter Sprachen bei der Ausführungszeit durch immer ausgefeiltere Techniken der Übersetzung zur Laufzeit heute in vielen Anwendungsfällen praktisch aufgeholt ist [11, 90].

Das Potenzial von Java wurde auch in der Welt des wissenschaftlichen Rechnens als mögliche Alternative wahrgenommen. Die Gemeinde des wissenschaftlichen Rechnens interessiert an Java in erster Linie die Vielzahl von Bibliotheken für graphische Visualisierung und Netzwerkinteraktion, die es ermöglichen, Anwendungen zur Berechnung, Aufbereitung und Weiterverarbeitung von Daten aus einem Guss zu erstellen. Eine weitere wichtige Rolle bei der Hinwendung zu Java spielt die Plattformunabhängigkeit der Java-Programme, die hoffen lässt, eine Anwendung, die für ein System entwickelt wurde, ohne Modifikationen auf einem anderen ausführen zu können. Da dies gleichfalls vom breiten Markt benötigt und nachgefragt wird, stehen all jene Eigenschaften schon zur Verfügung und werden entsprechend weiterentwickelt, ohne dass von den Anwendern aus dem Bereich des wissenschaftlichen Rechnens besondere Anstrengungen unternommen werden müssen. In anderen Bereichen besteht allerdings noch Bedarf an Verbesserungen, um Java uneingeschränkt einsatzfähig für wissenschaftliches Rechnen zu machen. Aus diesem Grunde hat sich 1998 die JavaGrande-Initiative [25] gegründet als eine Interessenvertretung für Themen, die für wissenschaftliches Rechnen mit Java wichtig sind, die aber nicht zwingend vom Massenmarkt benötigt werden. Ein Beispiel einer Beeinflussung des Java-Standards in diese Richtung ist die Änderung der Spezifikation der Gleitkomma-Arithmetik der virtuellen Maschine, die im Java-2-Standard jetzt die Verwendung von genaueren Zwi-

schenergebnissen (80 statt 64 Bit) erlaubt [63]. Allerdings stellt dieses Zugeständnis an die Nutzer von Java für Hochleistungsrechnen nur einen Kompromiss gegenüber dem ursprünglichen Vorschlag dar, und man kann an dem zurückgezogenen Wunsch auf Änderung der Java-Spezifikation (JSR 084 [91]) erkennen, wie schwierig die Einflussnahme auf Standards aus einer Anwendungsnische heraus ist. Dennoch stellt die Sprache Java durch ihr einfaches objektorientiertes Konzept, die automatische Speicherbereinigung, die eingebaute Unterstützung von Nebenläufigkeit, das Konzept der virtuellen Maschine und nicht zuletzt durch die Unmenge vorhandener Bibliotheken eine geradezu ideale Plattform für die Forschung dar, welche die universelle Nutzbarmachung von Rechnerbündeln für Hochleistungsrechnen anstrebt. Java bildet daher auch den Ausgangspunkt für die vorliegende Arbeit.

1.1 Konzepte für verteilt paralleles Rechnen

Wie so viele andere der in [81] untersuchten nebenläufigen objektorientierten Sprachen hat Java keine eingebaute Unterstützung für verteilt parallele Systeme. Um von der Verteilung zu abstrahieren oder wenigstens mit dem verteilten Adressraum in einem parallelen Programm umgehen zu können, haben sich im Laufe der Zeit vier verschiedene Konzepte entwickelt. Namentlich sind das der Nachrichtenaustausch, die entfernte Programmausführung, der virtuell gemeinsame Speicher und die generative Kommunikation. Dieser Abschnitt beleuchtet diesen Entwurfsraum für eine verteilt parallele Sprache. In Kapitel 2 werden verwandte Arbeiten in diese Kategorien eingeordnet.

1.1.1 Nachrichtenaustausch

Die absolute Mindestforderung an ein verteiltes Programm ist die Möglichkeit, für die verteilt ablaufenden Programmteile überhaupt Informationen austauschen zu können. Am einfachsten lässt sich diese Forderung erfüllen, wenn man den Teilen des verteilten Programms erlaubt, sich wechselseitig Botschaften zu senden. Die einzige Voraussetzung dafür ist ein Adressierungsschema für die Kommunikationspartner und die Verfügbarkeit von Routinen zum Senden und Empfangen von Daten. Zwei typische Vertreter dieses Konzepts sollen nun exemplarisch etwas näher besprochen werden.

Zeichenkanäle

Ein Zeichenkanal stellt eine Verbindung zwischen zwei Rechnern oder zwei Prozessen her, auf welcher beliebige Zeichenfolgen, also eine Ansammlung von unstrukturierten Daten, ausgetauscht werden können. Solche Zeichenkanäle bilden die fundamentale Grundlage des Internet und werden dort über Protokolle wie UDP [86] oder TCP [87] realisiert. Zur Adressierung des Kommunikationspartners dient bei den Zeichenkanälen des Internet eine Kombination aus der Internet-Adresse des Rechners und einer pro Kanal wählbaren Port-Nummer. Zeichenkanäle werden seit der Erfindung des ARPAnet/Internet in den frühen 70er Jahren als Basis für die verschiedensten höherwertigen Kommunikationsprotokolle zwischen Klienten und Dienstgebern eingesetzt.

Über Zeichenkanäle lassen sich zwar gut höherwertige Kommunikationsprotokolle implementieren, wie sie in der Weitverkehrskommunikation zahlreich eingesetzt werden, zur direkten Nutzung bei der Implementierung einer verteilt parallelen Anwendung sind sie allerdings ungeeignet, da die Anwendung selbst über ein eigenes Protokoll den übertragenen Informationen Struktur aufprägen muss. Solche speziellen Protokolle in jeder Anwendung neu zu implementieren, ist aber zu aufwendig und fehleranfällig.

Bibliotheken für Nachrichtenversand: PVM und MPI

PVM [93] und MPI [26] erlauben gegenüber reinen Zeichenkanälen ein höheres Maß an Abstraktion und zielen speziell auf verteilt parallele Anwendungen auf Rechnerbündeln. Beide Bibliotheken bieten eine Nachrichtenkopplung für prozedurale Programme nach dem Paradigma *single program, multiple data* (SPMD [50]). Ein SPMD-Programm wird ausgeführt, indem dasselbe Programm auf verschiedenen Rechnern gestartet wird. Jede Instanz des Programms wird mit einem unterschiedlichen Rang (einem unveränderlichen Startwert) initialisiert, der im folgenden auch zur Identifikation des Rechners für diesen speziellen Programmlauf dient. Unterschiedliche Programmabläufe auf den Prozessoren entstehen durch Einbeziehung des Prozessranges in Fallunterscheidungen, die dann zu unterschiedlichen Verzweigungen innerhalb der Programminstanzen führen.[1] Da ein SPMD-Programm auf jedem beteiligten Rechner mit Ausnahme der verarbeiteten Daten identisch ist, spricht man auch von einem *datenparallelen* Programm. Über eine Bibliothek können die parallel laufenden Instanzen miteinander kommunizieren. Das unhandliche Adressierungsschema der Zeichenkanäle des Internet wird durch den Startwert der parallel laufenden Instanz ersetzt. Darüber hinaus stehen für häufig auftretende Kommunikationsmuster vorgefertigte Schablonen zur Verfügung. Außerdem übernimmt das Laufzeitsystem das Starten der einzelnen Prozesse auf den Rechenknoten.

Der MPI-Ansatz fügt sich nicht nahtlos in das Konzept einer objektorientierten Sprache ein, da er nur reinen Datenaustausch ermöglicht und damit das Geheimnisprinzip verletzt. Trotzdem gibt es Bemühungen, für Java eine Bindung an MPI zu definieren oder die MPI-Funktionalität in einem objektorientierten Gewand für Java zu implementieren. Solche Anpassungen werden in Abschnitt 2.2.1 des folgenden Kapitels diskutiert. Die vorliegende Arbeit verwendet keine expliziten Kommunikationsoperationen, wie sie in Bibliotheken für Nachrichtenversand üblich sind, da diese eine Offenlegung der internen Datenstrukturen verlangen, damit gegen das objektorientierte Geheimnisprinzip verstoßen und daher nicht zu einem objektorientierten Programmierstil passen.[2] Allerdings zeigen die kollektiven Operationen beispielsweise von MPI, welche Kommunikationsmuster in einer verteilt parallelen Programmierumgebung benötigt werden.

[1] Der initiale Datenunterschied besteht lediglich aus dem Rang, dieser wird aber durch Fallunterscheidungen mit Einbeziehung des Ranges schnell größer.

[2] Zwar lässt sich der nachrichtenbasierte Ansatz auf die Kommunikation von Objekten statt auf die von reinen Daten erweitern, dies erzwingt dann aber die Kommunikation *kompletter* Objekte mit allen von ihnen verwalteten Teilzuständen.

1.1.2 Entfernte Programmausführung

Die entfernte Ausführung von Programmstücken abstrahiert von den zwischen den verschiedenen Adressräumen ausgetauschten Nachrichten, indem den Botschaften die Struktur von regulären Prozeduraufrufen aufgeprägt wird. Dadurch wird der Anwendungsprogrammierer davon entbunden, ein eigenes Protokoll für die Auswahl der entfernt auszuführenden Funktionalität und der zu übergebenen Daten zu entwerfen. Zudem treten Botschaften fast immer gepaart auf. Eine Botschaft fordert die entfernte Ausführung eines Programmstücks an und übermittelt die benötigten Argumente, während die zweite Botschaft in umgekehrter Richtung über die Abarbeitung informiert und optional einen Ergebniswert zurückliefert. Selbst bei einem asynchron ausgeführten Programmstück, bei dem der lokale Kontrollfluss weiterrechnet, während die Berechnung auf dem entfernten Knoten stattfindet, ist eine Rückmeldung erforderlich, falls später ein Ergebnis oder eine Benachrichtigung über erfolgreiche oder fehlgeschlagene Ausführung erwartet wird.

Entfernte Prozeduraufrufe

Der 1988 von SUN Microsystems entwickelte entfernte Prozeduraufruf (*remote procedure call, RPC [72]*) war der erste Vertreter der entfernten Programmausführung, der weite Verbreitung fand und heute in allen gängigen Betriebssystemen für die Implementierung von Netzwerkdiensten eingesetzt wird. Die Idee der entfernten Programmausführung geht aber mindestens bis ins Jahr 1976 zurück [107].[3] Der entfernte Prozeduraufruf basiert auf Zeichenkanälen als Transportmedium und fügt ihrem Adressierungsschema eine Kombination aus Programm- und Prozedurnummer hinzu, mit der das auszuführende Programmstück auf dem entfernten Rechner identifiziert wird. Mit dem entfernten Prozeduraufruf kann die Anwendung von der eigentlichen Kommunikation abstrahieren. Die Kommunikation versteckt sich hinter einem Aufruf an eine automatisch erzeugte Stellvertreterfunktion, die sich um die Übertragung ihrer Argumente an den Dienstgeber kümmert, diese mittels einer Kommunikationsoperation aktiviert und das Ergebnis mit Mitteln der verwendeten Programmiersprache als Rückgabewert liefert, so als ob es lokal berechnet worden wäre.

Methodenfernaufrufe

In einer objektorientierten Sprache ist Funktionalität (eine Methode) immer mit Zustand (dem zugehörigen Objekt) verknüpft. Das Adressierungsschema von entferntem Prozeduraufruf wird in einer objektorientierten Umgebung daher um das entfernt angesprochene Objekt zum sog. entfernten Methodenaufruf erweitert. Das Grundprinzip bleibt aber dasselbe: Eine an einem Stellvertreterobjekt aufgerufene Methode wird von diesem transparent in eine Kommunikationsoperation übersetzt. In dieser Kommunikationsoperation wird das auf einem anderen Rechner liegende Objekt kontaktiert, die an die Methode übergebenen Argumente werden zum Zielrechner kopiert, wo die eigentliche Methode ausgeführt wird. Das Ergebnis oder die Mitteilung über die erfolgreiche oder fehlgeschlagene Ausführung wird zurückübermittelt und vom Stellvertreter als

[3] Birrell et al. [10] geben einen guten Überblick die Anfänge der entfernten Programmausführung.

Ergebnis so an den Aufrufer zurückgegeben, als ob der Aufruf lokal stattgefunden hätte. Systeme für Methodenfernaufruf unterscheiden sich darin, wie gut die Objektsemantik, zu der u.a. Polymorphie, Überladung und Ausnahmebehandlung gehören, im verteilten Fall nachgebildet wird. Abschnitt 2.2.2 des folgenden Kapitels geht näher auf Systeme für entfernten Methodenaufruf ein.

1.1.3 Virtuell gemeinsamer Speicher

Die höchste Abstraktionsstufe für eine Umgebung mit verteiltem Speicher besteht darin, die Tatsache der Verteilung vor der Anwendung zu verbergen. Dies geschieht durch ein System für *virtuell* gemeinsamen Speicher (*distributed shared memory*, DSM). Der Applikation wird durch ein solches DSM-System die Illusion eines großen gemeinsamen Speichers für wahlfreien Zugriff auf allen beteiligten Prozessoren vermittelt. Die folgenden Abschnitte geben einen groben Überblick über die Techniken von virtuell gemeinsamem Speicher. Eine umfangreiche Übersicht über existierende DSM-Systeme und die in ihnen eingesetzten Konsistenzmodelle geben Protić et al. in [89].

Seitenbasierte DSM-Systeme

Ein klassisches seitenbasiertes DSM-System sorgt hinter den Kulissen dafür, dass bei einem Zugriff auf eine Speicheradresse, die im Hauptspeicher eines anderen Prozessors liegt, die zugehörige Speicherseite zum zugreifenden Prozessor transportiert wird. Die Seite wird darauf in den Adressraum dieses Prozessors eingeblendet. Für diesen Mechanismus werden die Fähigkeiten der Speicherverwaltungseinheit (MMU) moderner Prozessoren ausgenutzt. Die MMU wird normalerweise vom Betriebssystem dazu verwendet, virtuellen Speicher zu realisieren. Mit virtuellem Speicher muss nicht der gesamte benutzte Adressraum einer Anwendung mit physikalischem Speicher hinterlegt sein, sondern Seiten können auf die Festplatte ausgelagert sein. Erst beim Zugriff auf eine ausgelagerte Seite wird in einer Ausnahmebehandlungsroutine die Speicherseite unbemerkt von der Anwendung zurückgeladen und kann so von ihr benutzt werden, als ob sie die ganze Zeit im Hauptspeicher gestanden hätte. Genau jener Mechanismus wird von seitenbasierten DSM-Systemen dazu verwendet, eine Speicherseite bei Bedarf statt von der Festplatte von einem anderen Rechner herbei zu kopieren und in den lokalen Adressraum einzublenden.

Das Kopieren einer Speicherseite von einem auf einen anderen Rechner führt zu einer Replikation der betreffenden Speicherseite, da sie danach in beiden Adressräumen ansprechbar ist. Durch diese *Replikation* können Lesezugriffe auf allen Rechnern lokal bedient werden, die ein Replikat einer Seite halten. Generelles Problem bei der Replikation sind Schreiboperationen, die eine der Kopien verändern. Nach einem Schreibzugriff auf eines der Replikate sind alle anderen Kopien veraltet, da Leseoperationen dort noch den Zustand vor dem Schreibzugriff sehen. Die Regeln, wann ein Lesezugriff eines Prozessors den Effekt eines Schreibzugriffs eines anderen Prozessors beobachten kann, regelt ein Konsistenzmodell. Verschiedene Konsistenzmodelle für DSM-Systeme werden in [1] behandelt.

Die Idee, über Seitenaustausch einen virtuell gemeinsamen Adressraum für vernetzte Arbeitsstationen zu realisieren, stammt von Kai Li und Paul Hudak [59]. Ein

verbreitetes seitenbasiertes DSM-System ist TreadMarks [51].

Objektbasierte DSM-Systeme

DSM-Systeme unterscheiden sich sowohl hinsichtlich der kleinsten Einheit, aus welcher sich der virtuell gemeinsame Adressraum zusammensetzt, als auch bezüglich ihres Konsistenzmodells.

Bei der Wahl der Größe der kleinsten Verteilungseinheit spielen insbesondere voneinander unabhängige Schreibzugriffe (die unterschiedliche Adressen betreffen) auf dieselbe Speicherseite eine Rolle. Treten solche voneinander unabhängigen Schreibzugriffe auf, spricht man auch von *false sharing*. In einem solchen Fall müssen entweder die unterschiedlichen Änderungen zweier Replikate ineinandergemischt werden, oder die Seite muss zwischen den Schreibzugriffen zwischen den Rechnern hin- und herkopiert werden, um einen konsistenten Zustand zu erhalten. Das Hin- und Herkopieren führt zum sog. Seitenflattern (*page thrashing*) und beeinträchtigt die Effizienz des Gesamtsystems erheblich. Insbesondere, wenn ein objektorientiertes Programm mit vielen kleinen Objekten auf ein seitenbasiertes DSM-System abgebildet wird, ist die Wahrscheinlichkeit groß, dass mehrere Objekte, auf die unabhängig voneinander zugegriffen wird, auf derselben Seite zu liegen kommen und dadurch Seitenflattern auslösen.

Um die Wahrscheinlichkeit von Zugriffskonflikten und damit die Häufigkeit von Seitenflattern zu reduzieren, wurden DSM-Systeme entwickelt, bei denen sich die kleinste Speichereinheit näher an der Programmiersprache orientiert, in welcher Programme für das entsprechende System geschrieben werden. In einem objektbasierten DSM-System werden einzelne Objekte unabhängig verwaltet. Wird auf ein Objekt zugegriffen, wird nur genau dieses Objekt an den Ort des Zugriffs kopiert, ohne gleichzeitig alle Objekte mitkopieren zu müssen, die auf derselben Speicherseite liegen. Durch diesen Ansatz kann zwar die Häufigkeit von Zugriffskonflikten verringert werden, es ist aber nicht möglich, sie ganz auszuschließen, da auch unterschiedliche Teile ein und desselben Objektes von mehreren Prozessoren unabhängig voneinander modifiziert werden können.

Verteilte Objekträume

In einem verteilten Objektraum sind die Objekte der parallelen Anwendung über die Adressräume mehrerer Rechner verteilt, aber über entfernten Methodenaufruf von jedem Rechner aus ansprechbar. In einem verteilten Objektraum wird also beim Zugriff auf ein Objekt nicht dieses an den Ort des Zugriffs kopiert, sondern die ausführende Aktivität (*thread*) wechselt an den Ort, an dem sich das entsprechende Objekt befindet.

Wie in einem herkömmlichen DSM-System ist für die Anwendung in einem verteilten Objektraum der Zugriff auf alle Objekte des verteilten Systems möglich. Damit liegt für die Anwendung ebenso ein virtuell gemeinsamer Speicher vor. Dadurch, dass alle Zugriffe auf ein bestimmtes Objekt aber auf demjenigen Rechner stattfinden, auf welchem das Objekt angelegt ist, ist kein erweitertes Konsistenzmodell notwendig. Die Zugriffe auf ein Objekt können genauso koordiniert werden, wie das bei der Abarbeitung eines parallelen Programms auf einem einzelnen Rechner auch geschehen würde.

Bei einem ausschließlich über entfernten Methodenaufruf realisierten verteilten Objektraum erübrigt sich bei gemeinsamem Zugriff auch das Hin- und Herkopieren von Objekten zwischen mehreren Prozessen. In einem solchen Fall treffen und synchronisieren sich die logischen Aktivitätsstränge auf dem Rechner, auf dem das Objekt liegt.

Hybridansatz

Ein verteilter Objektraum löst aber nicht alle Probleme. Insbesondere wenn Objekte von mehreren Aktivitäten ausschließlich lesend benutzt werden, können diese Lesezugriffe in einem herkömmlichen DSM-System kollisionsfrei auf unterschiedlichen Rechnern nebenläufig bedient werden. In einem verteilten Objektraum, bei dem der Aufenthaltsort des Objektes den Ausführungsort von Zugriffen bestimmt, treffen sich zugreifende Aktivitäten auf einem Rechner. Dieser stellt für den Zugriff einen Flaschenhals dar, da sich die Zugriffe gegenseitig behindern. Ein möglicher Ausweg ist die Verbindung eines verteilten Objektraums mit Eigenschaften eines objektbasierten DSM-Systems. Dadurch können Objekte je nach ihren Zugriffseigenschaften entweder zentral auf einem Knoten angelegt werden oder über mehrere Knoten für unabhängigen lokalen Lesezugriff repliziert werden.

Die optimale Vorgehensweise ist Gegenstand aktiver Forschung und endgültige Ergebnisse fehlen noch. Die vorliegende Arbeit verfolgt einen Hybridansatz aus verteiltem Objektraum und Replikation. Der verteilte Objektraum zusammen mit einer neu entwickelten Replikationsstrategie ist in der Lage, die Vorteile der drei Ansätze „nachrichtenbasierte Kommunikation“, „entfernte Programmausführung“ und „virtuell gemeinsamer Speicher“ zu vereinen. Kollektive Operationen und Optimierung der Zugriffslokalität werden durch Einführung einer neuen Form von Replikation erreicht. Einfache Kommunikation und Koordination ist über entfernte Methodenaufrufe an Objekten des verteilten Objektraums möglich.

1.1.4 Generative Kommunikation

Neben den Konzepten Nachrichtenaustausch, entfernte Programmausführung und gemeinsamer Speicher schlägt Gelernter in [28] einen vierten Weg, die sog. generative Kommunikation, für die Strukturierung von verteilt parallelen Anwendungen vor. Die Kommunikation heißt generativ, weil die erzeugten und zur Kommunikation vorgesehenen Daten auch ohne einen im voraus feststehenden Kommunikationspartner und losgelöst vom Absender eine eigene Daseinsberechtigung haben. In der vorgeschlagenen Sprache Linda erzeugen die Prozesse einer verteilt parallelen Anwendung Datentupel und legen diese in einen gemeinsamen Tupelraum ab. Dort werden die Daten für potenzielle Interessenten zwischengespeichert. Eine Empfangsoperation spezifiziert ein Muster, das auf die zu empfangenden Daten passen muss. Der Tupelraum wählt aus den vorrätigen Datentupeln eines aus, das auf das Muster der Empfangsoperation passt, sofern ein solches Tupel vorliegt. Die Entkopplung von Sender und Empfänger findet statt, indem ein Sender immer senden kann und der Empfänger die passenden Daten erhält, egal ob die Sendeoperation zeitlich vor oder nach der Empfangsoperation stattgefunden hat. Der Sender spezifiziert keinen Empfänger, sondern der Empfänger sucht sich über Mustervergleich unter den gesendeten Daten die passenden aus. Die

Wegewahl der Daten bei konventioneller Kommunikation über Absender- und Empfängeradressen wird bei der generativen Kommunikation durch Mustervergleich während der Empfangsoperation ersetzt.

Kommunikation in dieser Form auszudrücken, ist deshalb besonders elegant, weil sich alle anderen Paradigmen auf solche Schreib-/Leseprimitive im verteilten Tupelraum zurückführen lassen. Leider ist ein verteilter Tupelraum aufgrund der Assoziativität noch viel schwieriger effizient zu implementieren als ein virtuell gemeinsamer Adressraum. Daher wird generative Kommunikation in dieser Arbeit nicht weiter betrachtet.

1.2 Thesen

Die für verteilt paralleles Rechnen existierenden Ansätze haben mit einem oder vielen der folgenden Probleme zu kämpfen: Das Schreiben verteilt paralleler Programme ist kompliziert und extrem fehleranfällig. Die Portierung eines parallelen Programms für eine verteilte Umgebung erfordert das Neuschreiben weiter Programmteile. Es kann keine moderne objektorientierte Sprache verwendet werden. Es fehlt eine nahtlose Verbindung von Konzepten für nebenläufige nichtverteilte und verteilt parallele Programme. Das Programm ist an eine spezielle Plattform gebunden und die verteilte Programmierumgebung selbst ist nur schwer auf ein neues System portierbar. Das Programmiermodell für ein verteilt paralleles Programm unterscheidet sich stark von bekannten, gelehrten und gelernten Modellen für nebenläufige nichtverteilte Programme. Die verteilt parallele Ausführung ist nicht effizient.

Die vorliegende Arbeit vertritt und belegt folgende Thesen:

- Ein anerkanntes und weit verbreitetes objektorientiertes Programmiermodell für nebenläufiges nichtverteiltes Programmieren kann für die verteilt parallele Programmierung eines Rechnerbündels nahtlos erweitert werden. Damit ist eine einfache Portierung einer parallelen nichtverteilten Anwendung für die verteilte Umgebung eines Rechnerbündels oder jedes anderen Parallelrechners mit verteiltem Speicher möglich.
- Es gibt eine nahtlose Verknüpfung von Konzepten einer parallelen objektorientierten Sprache mit Konzepten für datenparalleles Programmieren.
- Die Realisierung der verteilten Programmierumgebung für Rechnerbündel ist durch die Technik virtueller Maschinen plattformunabhängig möglich. Damit wird das verteilt parallele Programm unverändert auf jedem Parallelrechner einsetzbar, da die gesamte Laufzeitumgebung plattformunabhängig ist.
- Die verteilte Ausführung paralleler objektorientierter Programme in dieser portablen Umgebung ist effizient möglich.

1.3 Ziel dieser Arbeit

Das Ziel der vorliegenden Arbeit ist der Entwurf einer Programmierumgebung für Rechnerbündel, die es erlaubt, eine Produktivitätssteigerung über den gesamten Le-

benszyklus einer Anwendung zu erreichen. Produktivitätssteigerung ist hier im Sinne der HPCS-Initiative [16] gemeint, also angefangen vom Entwurf und der Implementierung der Anwendung über ihre effiziente Ausführung auf einem Rechnerbündel bis hin zu Wartung und möglichen Erweiterungen.

Um das Ziel gesteigerter Produktivität für Hochleistungsanwendungen auf Rechnerbündeln zu erreichen, werden bewährte Techniken aus der konventionellen Software-Technik eingesetzt. Dafür muss die eingesetzte Programmiersprache Objektorientierung unterstützen, um saubere Entwürfe zu ermöglichen. Sie muss moderne Eigenschaften wie automatische Speicherbereinigung, eingebaute Nebenläufigkeit und Typsicherheit bis zur Ausführung enthalten, um den Anwendungsentwickler genauso gut zu unterstützen, wie das für herkömmliche Software-Projekte zum Standard geworden ist. Nicht zuletzt muss die Sprache für Rechnerbündel eine um den Verteilungsaspekt nur minimal erweiterte etablierte Sprache sein, damit schon bei der Ausbildung der Anwendungsentwickler durch Rückgriff auf Bekanntes eine steile Lernkurve und damit eine Effizienzsteigerung erreicht wird.

Wegen der dynamischen Entwicklung der Zielsysteme für Hochleistungsanwendungen ist Portabilität ein zentraler Aspekt für hohe Produktivität bei Erweiterung und Wartung sowohl von Anwendungen als auch der Programmierumgebung selbst. Diese Anforderung wird durch Abstraktion von den konkreten Zielplattformen mit Hilfe von VM-Technologie und Modularisierung auf höchster Ebene erfüllt. Dazu muss die entworfene Programmierumgebung auf einer existierenden und möglichst weitverbreiteten virtuellen Maschine aufsetzen und diese um den Verteilungsaspekt zu einer virtuellen Bündelmaschine erweitern. Um die Forderung nach Modularität zu erfüllen, darf diese Erweiterung nicht invasiv durch Modifikation der virtuellen Basismaschine erfolgen, sondern muss durch Zusammenschaltung von unveränderten virtuellen Maschinen auf den Rechenknoten zu einer verteilten virtuellen Maschine erfolgen. Diese Trennung des Verteilungsaspektes von der Implementierung der virtuellen Maschine maximiert die Portabilität nicht nur der Anwendungen, sondern auch der verteilten Umgebung selbst. Die Modularität auf Ebene der Programmierumgebung ist die Grundlage für bestmögliche Effizienz durch Auswahl der aktuell besten Implementierung der virtuellen Maschine pro Zielplattform und durch Ausnutzung von stetigen Fortschritten in der Technologie für Laufzeitübersetzer.

Zusammenfassend sind Ziele und Alleinstellungsmerkmale dieser Arbeit die nahtlose Erweiterung einer modernen objektorientierten Sprache um verteilungsrelevante und datenparallele Aspekte, die modulare Erweiterung einer virtuellen Maschine zu einer verteilten virtuellen Maschine und die darauf möglich werdende effiziente Ausführung plattformunabhängiger, paralleler, objektorientierter Programme auf einem Rechnerbündel. Bei der Neuentwicklung einer verteilten parallelen Anwendung ist ein minimaler Lernaufwand bei Kenntnis der Basissprache notwendig und die Anzahl der Änderungen bei der Portierung einer bestehenden parallelen Anwendung für die verteilte Ausführung lässt sich auf eine Reihe von Annotationen beschränken.

1.4 Gliederung

Diese Arbeit gliedert sich folgendermaßen: Kapitel 3 entwirft die Architektur der Programmierumgebung. In den nächsten drei Kapiteln 4, 5 und 6 werden die zur Realisierung notwendigen Komponenten entwickelt. Dabei beschreibt Kapitel 4 einen effizienten Mechanismus für die Übertragung von Objektgraphen, Kapitel 5 stellt die Bausteine für Hochleistungskommunikation in Java und deren transparente Verwendung vor. Das Kapitel 6 führt transparent replizierte Objekte ein. Im Kapitel 7 wird die Leistung des Systems untersucht und bewertet. Abschließend fasst das letzte Kapitel 8 die gewonnenen Erkenntnisse zusammen und gibt einen Ausblick auf weitere in diesem Zusammenhang aufgeworfene Forschungsfragen.

Kapitel 2

Verwandte Arbeiten

Die vorliegende Arbeit zielt auf die plattformunabhängige Realisierung einer Programmierumgebung für Rechnerbündel. Zu diesem Zweck bietet sich die Technik virtueller Maschinen an. Diese Technik hat mit dem Auftauchen von Java einen enormen auch kommerziellen Anschub erhalten. Als sich abzeichnete, dass sich Java aus der Nische der Spielzeuganwendungen in Browser-Fenstern lösen und die Ausführungseffizienz des plattformunabhängigen Codes eine erfolgversprechende Aufholjagd beginnen würde, setzte eine rege Forschungstätigkeit ein mit dem Ziel, Java auch für Hochleistungsanwendungen auf verteilt parallelen Systemen nutzbar zu machen. Daraus resultierte eine Vielzahl von Publikationen über Systeme, die das für Einzelrechner so erfolgreiche Java-Modell auf verteilte Umgebungen übertragen.

Die Vorschläge hierzu lassen sich von ihrer Vorgehensweise her in fünf Kategorien einteilen. Erstens sind das Bindungen an schon existierende plattformabhängige Bibliotheken für Hochleistungsrechnen, zweitens die Entwicklung von solchen Bibliotheken in Java selbst, drittens Präprozessoren, welche eine neue, Java-ähnliche Sprache mit Konstrukten für verteilte Systeme in reines Java zurückabbilden, viertens die native Übersetzung von Java oder einer Obermenge in plattformabhängige Programme unter Verwendung von plattformabhängigen Bibliotheken und schließlich fünftens die Entwicklung einer virtuellen Java-Maschine, die verteilt in einem Rechnerbündel läuft und reguläre Java-Programme ausführt. Die vorliegende Arbeit verfolgt den Präprozessor-Ansatz, da dieser einen guten Kompromiss zwischen Ausdrucksmächtigkeit, Plattformunabhängigkeit und Effizienz verspricht.

In den nachfolgenden Abschnitten werden die unterschiedlichen Lösungsansätze bezüglich ihrer Vor- und Nachteile untersucht. Die Darstellung ist nach der Vorgehensweise des Ansatzes gegliedert, da der verwendete Grundansatz schon spezifische Vor- und Nachteile bedingt, und diese so gemeinsam dargestellt werden können.

2.1 Anbindung plattformabhängiger nativer Bibliotheken

Die virtuelle Java-Maschine bietet die Möglichkeit, über eine definierte Schnittstelle mit Bibliotheken zu kommunizieren, die nicht selbst in Java geschrieben sind und auch nicht im Java-Binärformat (Bytecode) vorliegen. Solche Bibliotheken im Ausfüh-

rungsformat der Zielmaschine werden zur Ausführungszeit dynamisch zur virtuellen Maschine hinzugebunden und sind über sog. native Methoden regulärer Java-Klassen ansprechbar. Auch die Java-Basisbibliothek besitzt eine Vielzahl solcher nativen Methoden für Funktionen, die selbst nicht in Java ausdrückbar sind. Dabei handelt es sich in der Regel um die Interaktion mit dem Betriebssystem, auf dem die virtuelle Maschine läuft, oder um Zugriffe auf die graphische Benutzerschnittstelle. Eine Klasse, die native Methoden enthält, ist auf einer bestimmten Plattform nur dann verwendbar, wenn auch eine entsprechende Bibliothek mit der Implementierung ihrer nativen Methoden vorrätig ist. Alle Ansätze, die sich auf die Einbindung nativer Bibliotheken stützen, um Hochleistungsrechnen in oder mit Java zu ermöglichen müssen prinzipiell drei Nachteile in Kauf nehmen:

- Eingeschränkte Portabilität: Enthält eine Klasse native Methoden, verliert sie die wesentliche Eigenschaft, dass sie auf jeder kompatiblen virtuellen Maschine ohne Neuübersetzen ausführbar ist. Um ein Anwendungsprogramm, dessen Klassen native Methoden verwenden, in einer virtuellen Maschine auf einer bestimmten Plattform ausführen zu können, muss zuerst die zugehörige native Bibliothek auf diese Plattform portiert und dort übersetzt werden, damit sie beim Laden der Java-Klasse dynamisch zur virtuellen Maschine hinzugebunden werden kann.

- Verlust der Sicherheitsgarantien: Native Methoden stellen einen Ausbruch aus der virtuellen Maschine dar, wobei alle Garantien verloren gehen, welche die virtuelle Maschine normalerweise bei der Ausführung zusichert. Wird ein Java-Programm ausgeführt, das native Methoden enthält, so kann die virtuelle Maschine weder einen Absturz aufgrund von illegalen Speicherzugriffen im nativen Code verhindern, noch kann sie die Einhaltung von Zugriffsbeschränkungen erzwingen, die in reinen Java-Programmen für Ressourcen des ausführenden Rechners definiert sind.

- Leistungsengpässe: Beim Übergang aus der VM in nativen Code entstehen Reibungsverluste, die beim Aufruf einer nativen Methode zu wesentlich höherem Aufwand führen als beim Aufruf einer regulären Java-Methode. Ferner ist aus einer nativen Methode kein direkter Zugriff auf Datenstrukturen der virtuellen Maschine möglich, sondern alle Zugriffe finden über das Java Native Interface (JNI [60]) statt. Diese zusätzliche Abstraktionsschicht verhindert, dass die native Bibliothek nicht nur von der Ausführungsplattform, sondern auch noch von Implementierungsdetails der virtuellen Maschine abhängig wird.

Beruht ein Vorschlag für wissenschaftliches Rechnen in Java auf der Einbindung von nativen Methoden, so verliert er zwei wesentliche Eigenschaften von Java, nämlich die Portabilität und die sichere Ausführung, ohne dabei unbedingt optimale Effizienz zu erreichen. Trotz deren Nachteile gibt es eine Reihe von Arbeiten, die sich mit der Nutzbarmachung nativer Bibliotheken in Java beschäftigen. Die vorliegende Arbeit vermeidet die Nutzung nativer Bibliotheken.[1]

[1] Eine native Bibliothek ist lediglich für die Anbindung von Netzwerkkarten für Hochgeschwindigkeitskommunikation notwendig.

2.1.1 Generierung nativer Bibliotheksanbindungen

Selbstverständlich gibt es aber eine Vielzahl von Bibliotheken insbesondere für wissenschaftliches Rechnen, die in konventionellen Sprachen geschrieben wurden und bereits einen langen Reife- und Optimierungsprozess durchlaufen haben. Daher ist es durchaus verständlich, dass man diese Funktionalität gerne unverändert weiterverwenden und nicht noch einmal nachimplementieren und -optimieren möchte. Eine Reihe von Projekten beschäftigt sich daher mit der Anbindung solcher Bibliotheken an Java. Getov beschreibt in [29] ein Werkzeug zur automatischen Erzeugung von Java-Bindungen für native Bibliotheken. Mit diesem Werkzeug wurden Bindungen an eine native MPI-Kommunikationsbibliothek und an verschiedene numerische Bibliotheken wie BLAS und ScaLAPACK erzeugt [75].

2.1.2 Native Kommunikationsbibliotheken

Andere Bindungen von Java an native Bibliotheken für Nachrichtenaustausch sind ebenfalls mit „mpiJava“ [6] für MPI und mit JavaPVM [95] für PVM realisiert worden. Mit OpusJava [57] ist es möglich, Fortran Module über Java-Verbinder in eine verteilte Umgebung einzubinden. Das Problem, dass die Anbindung an eine native Kommunikationsbibliothek aufgrund des Geschwindigkeitsengpasses nativer Bibliotheken keine optimale Leistung erzielt, geht das Jaguar-Projekt [103] an. Jaguar ist eine auf einem Laufzeitübersetzer beruhende Java-Umgebung, die eine alternative Möglichkeit bietet, einen Ausbruch aus der virtuellen Maschine durchzuführen, ohne die beschriebenen Geschwindigkeitseinbußen hinnehmen zu müssen. Der Jaguar-Übersetzer erkennt bestimmte Bytecode-Folgen, die Zugriffe auf spezielle „magische“ Klassen darstellen, und ersetzt diese durch direkte Zugriffe auf Speicherbereiche außerhalb der virtuellen Maschine oder auf andere Ressourcen der Ausführungsplattform. Auf diese Art und Weise lassen sich VM-externe Ressourcen direkt aus Java-Code ansprechen, aber dennoch können die Zugriffe mit Mitteln der virtuellen Maschine kontrolliert werden. Der eigentliche Ausbruch aus der virtuellen Maschine findet erst durch die spezielle Übersetzung statt. Dieser Trick, den Ausbruch aus der virtuellen Maschine über Zugriffe auf „magische“ Klassen zu realisieren, wird auch in der Jikes RVM [22] für die Implementierung der virtuellen Maschine in Java selbst verwendet.

Zusammenfassend liegt der Schwerpunkt bei Projekten, die nativen Code für wissenschaftliches Rechnen in die virtuelle Maschine importieren, eher darauf, in einer abwärtskompatiblen Art und Weise moderne Spracheigenschaften von Java für Projekte zu erschließen, die aufgrund der intensiven Nutzung existierender Bibliotheken ansonsten an konventionelle Sprachen gebunden blieben. Hierbei geht es aber nicht in erster Linie um eine einheitliche Plattform für portable, verteilt parallele Anwendungen auf Rechnerbündeln.

2.2 Java-Bibliotheken für verteilt paralleles Rechnen

Die Implementation von Bibliotheken für Hochleistungsrechnen auf Rechnerbündeln in reinem Java behebt die im letzten Abschnitt besprochenen Probleme der Plattform-

abhängigkeit. Bei solchen Ansätzen können bei der Programmausführung alle Garantien bezüglich Sicherheit und Zuverlässigkeit einer virtuellen Java-Maschine gegeben werden. Bibliotheksansätze haben den Vorteil, dass der Lernaufwand für den Anwender relativ gering ist und nur mit dem Umfang der Anwendungsschnittstelle wächst. Allerdings bleibt eine Bibliothek immer an die Ausdrucksmittel der Sprache gebunden.

2.2.1 Java-Bibliotheken zum Nachrichtenaustausch

Analog zu der Bindung von nativen Bibliotheken für Nachrichtenaustausch existieren Arbeiten, die Bibliotheken für PVM und MPI in reinem Java nachgebaut haben. Die PVM-Variante JPVM wird von Ferrari in [21] vorgestellt. Das MPI-Pendant MPIJ von Judd et al. wird im Zusammenhang mit der DOGMA Laufzeitumgebung in [48] beschrieben. Da PVM und MPI noch aus der Zeit stammen, bevor den objektorientierten Sprachen der Durchbruch gelang, passen aber die dort zur Verfügung stehenden Operationen schlecht zu einem objektorientierten Programmierstil, da lediglich Berechnungen auf Feldern von Basistypen oder bestenfalls Strukturen unterstützt werden. Aus diesem Grund schlagen Nelisse et al. in [77] eine Bibliothek vor, welche die in MPI auftretenden kollektiven Kommunikationsmuster auf Mengen von Objekten überträgt. Kollektive Operationen im MPI-Stil legen das Programmiermodell allerdings auf reine Datenparallelität fest, die nicht zum kontrollparallelen Modell von Java passt.

2.2.2 Java-Bibliotheken zum entfernten Methodenaufruf

Unterschiedliche Implementierungen des entfernten Methodenaufrufs unterschieden sich in der Objektsemantik, der Unterstützung von polymorphen Methoden, der Verwendung von Ausnahmebedingungen und der sprachübergreifenden Einsetzbarkeit. Im Folgenden werden Corba [33] und Java-RMI [71] besprochen.

Corba. Das von der Object Management Group (OMG) spezifizierte Corba ist ein plattformunabhängiger Mechanismus für Fernaufruf, der auch zur Integration von unterschiedlichen Programmiersprachen verwendet werden kann. Es existieren Sprachbindungen für Java, C, C++ und Fortran. Dies ermöglicht einen Aufruf über Sprachgrenzen hinweg. Ein in C++ implementiertes Corba-Objekt kann beispielsweise von einem in Java implementierten Klienten aufgerufen werden. So schön diese integrative Eigenschaft von Corba zur Einbindung von althergebrachten Komponenten auch sein mag, so bedingt sie doch natürlicherweise eine Einschränkung auf den kleinsten gemeinsamen Nenner aller unterstützten Sprachen bezüglich der unterstützen Objektsemantik. So sind in Corba weder polymorphe Methoden noch Ausnahmebedingungen mit Vererbungsstruktur möglich. Auch ist die Übergabe von Argumenten als Wert eingeschränkt; an das dynamische Nachladen von Argumentklassen während des Aufrufs ist gar nicht zu denken.

Corba hat zweifelsohne seine Stärken bei der Integration von Komponenten unterschiedlicher Herkunft und es existieren, je nach Implementierung, eine Vielzahl zusätzlicher Dienste für Namensauflösung, Transaktionen, Persistenz und vieles mehr. Um aber eine Sprachplattform zu bauen, die es ermöglicht, ein Rechnerbündel als Gesamtsystem zu programmieren, ist die sprachenverbindende Eigenschaft von Corba

wenig hilfreich. Besser geeignet scheint eine Lösung, die möglichst verlustfrei die objektorientierten Eigenschaften einer Sprache in ein verteiltes System hinüberrettet.

Java RMI. Java-RMI, das Standard-Paket für entfernten Methodenaufruf in Java, erlaubt es, Klassen zu definieren, deren Methoden durch Implementation einer ausgezeichneten Schnittstelle entfernt aufrufbar gemacht werden. Ein zusätzlicher Codegenerator erzeugt aus dem Java-Programm Stellvertreterklassen, die ein Objekt auf einer entfernten Maschine repräsentieren. Ein Methodenaufruf auf dem Stellvertreterobjekt bewirkt die Weiterleitung dieses Aufrufes über das Netzwerk zur Zielmaschine, den Aufruf derselben Methode auf dem Originalobjekt und die Rückübertragung des Ergebnisses zur anfragenden Seite. An einen solchen entfernten Methodenaufruf können beliebige Argumente[2] übergeben werden. Die Bibliothek kümmert sich selbständig um das Verpacken der Parameter in ein geeignetes Netzwerkformat und deren Übertragung zum Zielrechner, wo eine identische Kopie der ursprünglichen Argumente erstellt und an die aufzurufende entfernte Methode übergeben wird.

RMI ist Teil des Java-Standards und liegt somit jeder standardkonformen virtuellen Maschine bei. Da RMI speziell für Java entworfen ist und auch nur Kommunikation zwischen virtuellen Java-Maschinen erlaubt, ist die Semantik für entfernte Objekte eine gute Näherung an die Java-Objektsemantik. Für die Entwicklung verteilt paralleler Anwendungen in Rechnerbündeln eignet sich RMI aber nicht, da trotz der relativ einfachen Umwandlung einer regulären Klasse in eine entfernt ansprechbare Klasse die notwendigen Änderungen im Programm dennoch erheblich sind [85]. Unter anderem wegen mangelnder Unterstützung von Hochleistungs-Kommunikationshardware ist zudem die Kommunikationsgeschwindigkeit von Java-RMI im Rechnerbündel unbefriedigend [84]. Arbeiten, die sich dem Aspekt der Behebung von Leistungsengpässen in Java-RMI annehmen, werden im Folgenden diskutiert.

Leistungssteigerung von RMI

Java-RMI ist für die Kommunikation zwischen Klient und Dienstgeber in der Weitverkehrskommunikation ausgelegt. In diesem Anwendungsszenario liegt das Hauptaugenmerk auf der Behandlung unzuverlässiger Verbindungen und möglicher Ausfälle von Rechnern während einer Übertragung, da die Antwortzeiten von den Nachrichtenlaufzeiten dominiert werden. Beim Einsatz in einem Rechnerbündel resultiert daraus eine unzufriedenstellende Leistung bezüglich Latenz und Kommunikationsbandbreite. Verschiedenartige Optimierungen sind zur Umgehung diese Engpasses bereits vorgeschlagen worden.

Krishnaswamy et al. stellen in [54] eine alternative Transportschicht für RMI vor, die auf UDP anstatt auf TCP/IP aufsetzt. Gleichzeitig wird RMI dahingehend modifiziert, dass Objekte lokal zwischengespeichert werden können. Das Hauptproblem liegt bei dieser Arbeit aber darin, dass weder Hochleistungs-Kommunikationshardware berücksichtigt wird, noch dass andere Ineffizienzen im RMI-System, wie Speicheranforderungen, während eines entfernten Aufrufs angegangen werden.

Maassen et al. [67] schlagen eine effiziente Implementierung von RMI vor, die

[2] Die Klassen der Argumente müssen lediglich die Markierung `java.io.Serializable` tragen.

Hochleistungs-Kommunikationshardware unterstützt, verknüpfen dieses RMI aber untrennbar mit einem nativen Java-Übersetzer, der Java wie eine konventionelle Sprache direkt in ausführbaren Code für die Zielmaschine übersetzt. Bei einer solchen RMI-Implementation in nativem Code lässt sich einfacher eine effiziente Verarbeitung erreichen, da die Implementierung nicht an die Beschränkungen der virtuellen Maschine gebunden bleibt und das Speicherlayout der Java-Objekte dem nativen Übersetzer bekannt ist.

Die vorliegende Arbeit ersetzt Java-RMI durch eine für Rechnerbündel optimierte, aber dennoch in reinem Java geschriebene Bibliothek für Fernaufruf. Die Grundlagen hierzu wurden in [79], [39] und [38] gelegt. Die neue Bibliothek unterstützt Hochgeschwindigkeitsnetzwerke, ist portabel und ermöglicht ebenso effiziente Fernaufrufe wie der oben erwähnte native Übersetzer. Die sich rasant entwickelnde Technik der JIT-Übersetzung (*just-in-time*) in den virtuellen Maschinen hat ebenso zu diesem Erfolg beigetragen wie die in Kapitel 4 besprochene schnelle Objektserialisierung und der in Kapitel 5 vorgestellte für Hochleistungs-Kommunikation im Rechnerbündel optimierte Entwurf der Kommunikationsbibliothek.

2.2.3 Gemeinsam benutzte Objekte

Wie schon im einleitenden Abschnitt 1.1.3 angedeutet, reichen Fernaufrufe zur effizienten Realisierung eines verteilten Objektraums nicht aus. Insbesondere die Lokalitätsaspekte eines DSM-Systems können durch Fernaufrufe in einem verteilten Objektraum nicht erreicht werden. Aufgrund der Abstraktion von der zugrunde liegenden Hardware und dem Betriebssystem sind Techniken für verteilten gemeinsamen Speicher, die direkt an der Speicherverwaltung des Prozessors ansetzen, in Java nicht plattformunabhängig realisierbar. Statt dessen sind von vielen Autoren „gemeinsam benutzte“ Objekte vorgeschlagen worden. Objekte werden bei Bedarf an den Ort transportiert, an dem ein Zugriff stattfinden soll. Dieser Zugriff und mögliche Folgezugriffe können dann lokal abgearbeitet werden. Da hierbei eine Kopie (Replikat) des Originalobjektes entsteht, spricht man auch von *Objektreplikation*. Bei einer Bibliothekslösung kann der Zugriff auf ein Objekt allerdings nicht für die Applikation transparent überwacht werden. Daher müssen Zugriffe vorher explizit über einen Stellvertreter des Objektes angefordert werden. Nach Abschluss des Zugriffs ist eine Freigabe notwendig.

Aleph [41] stellt dieses Kommunikationsschema zur Verfügung. Die Applikation kann ein reguläres Objekt als gemeinsam benutzt deklarieren und später den Zugriff von verschiedenen Knoten aus anfordern. Wird der Zugriff gewährt, erhält die Applikation eine Referenz auf das Objekt oder eine Kopie dessen. Wurde lesender Zugriff beantragt, darf das Objekt danach nur in lesenden Kontexten verwendet werden. Wurde Schreibzugriff angefordert, darf der Zustand des Objektes modifiziert werden. Die Modifikationen werden nach Abschluss der Operation und anschließender Freigabe des Objektes anderen Lesern sichtbar gemacht, indem das veränderte Objekt mit einer Leseanforderung auf den Knoten transportiert wird. Wurde das Objekt nicht verändert, können alle seine Kopien weiterhin benutzt werden.

Mit der in Aleph implementierten Form replizierter Objekte kann immer nur ein ganzes Objekt auf einmal angefordert werden. Nach einer Modifikation muss immer das gesamte Objekt auf die anderen Knoten transportiert und dort gegen die veral-

tete Kopie ausgetauscht werden, unabhängig davon, wie groß die Änderung war. Da die Applikation beim Zugriff aber eine direkte Referenz auf das Replikat erhält, kann unerkannt ein Alias für eine veraltete Kopie entstehen.

Die vorliegende Arbeit reichert ebenfalls einen verteilten Objektraum mit replizierten Objekten an, vermeidet jedoch die oben genannten Nachteile durch Einsatz eines Präprozessors und inkrementelle Zustandsfortschreibung.

2.2.4 Generative Kommunikation

Das Paradigma der generativen Kommunikation wurde in Abschnitt 1.1.4 eingeführt. Mit JavaSpaces [27] existiert hierfür eine kommerziell verfügbare Implementation für Java. Die Tupel von JavaSpaces haben als Einträge Java-Objekte anstatt der Basistypen des Originalvorschlags, und der Mustervergleich zum Auffinden passender Operationen bezieht die Vererbungshierarchie dieser Java-Objekte mit ein. Wie RMI leidet aber auch das JavaSpaces-Paket unter einer für verteiltes Rechnen unzulänglichen Kommunikationsleistung und bleibt daher auf Anwendungsgebiete im Kontext von JINI beschränkt, wo es um das Auffinden von verfügbaren Diensten und Ressourcen in adaptiven Netzwerken geht [23].

In der vorliegenden Arbeit kommt generative Kommunikation nicht zum Einsatz, da ein verteilter Tupelraum nicht effizient implementierbar ist und das Senden und Empfangen von Nachrichtentupeln nicht zum objektorientierten Vorgehen passt.

2.2.5 Bibliotheken für Grid-Computing

Unter Grid-Computing oder Meta-Computing versteht man Arbeiten, die sich zum Ziel setzen, brach liegende Rechenkapazitäten in weitverteilten Netzen, wie beispielsweise dem Internet, für Anwendungen nutzbar zu machen. In diesem Zusammenhang fällt oft das Schlagwort der „Lost cycles“, was soviel heißen soll, dass es darum geht, die CPU-Zyklen auszunutzen, die Arbeitsplatzrechner überall auf der Welt in Warteschleifen verbringen, weil sie mit den meist interaktiven Anwendungen ihrer Nutzer nicht ausgelastet sind. Ein weiteres Problemfeld, das auch unter das Stichwort Grid-Computing fällt, ist der einheitliche Zugang zu Großrechnern in Rechenzentren und deren Kopplung über Weitverkehrsnetze.

Grid-Computing versus Cluster-Computing. Der zentrale Unterschied zwischen Grid-Computing und Hochleistungsrechnen auf Rechnerbündeln (sog. Cluster-Computing) ist die unterschiedliche Netztechnologie und -topologie der verbundenen Rechner. Während man bei Grid-Computing mit langsamen und fehlerträchtigen Weitverkehrsnetzen Vorlieb nehmen muss, setzt man bei Rechnerbündeln auf Hochleistungsnetzwerke, die eine besonders hohe Bandbreite und eine extrem kurze Latenz ermöglichen, um Anwendungen effizient ausführen zu können, die ansonsten einen herkömmlichen Supercomputer mit gemeinsamem Speicher benötigen würden. Die Klasse der Anwendungen, die für Grid-Computing-Anwendungen in Frage kommen, zeichnet sich durch eine besonders grobgranulare Parallelität aus, bei der nur wenige Daten ausgetauscht werden müssen, um einen Rechner hinreichend lange zu beschäftigen, so dass sich die Kommunikation selbst über ein langsames Netzwerk amortisiert. Da

bei diesen Ansätzen der Hauptaugenmerk nicht auf effizienter Kommunikation liegt, bieten einige der besprochenen Arbeiten – meist als Zusatzangebot – eine Form der generativen Kommunikation. Beispiele für Grid-Computing-Anwendungen sind das Brechen von Verschlüsselungen oder die Suche nach „außerirdischem Leben“ im Rahmen des Projekts SETI@home [104], bei dem sich Computer über das Internet bei der Auswertung von Radioteleskopsignalen beteiligen können.

Der Bibliotheksansatz beim Grid-Computing. Vor allem zwei Eigenschaften der Programmierumgebung sind bei Grid-Computing-Vorhaben wichtig. Aus der Sicht der Anwendung ist das die Plattformunabhängigkeit des Codes, der über eine sehr große Anzahl heterogener Rechner verteilt werden muss. Aus Sicht der Zielsysteme sind es die Sicherheitsgarantien, die bei der Ausführung des fremden Codes gegeben werden können. Für beide Problemfelder bietet Java optimale Voraussetzungen. Allerdings ist vor diesem Hintergrund auch einsichtig, dass die Mehrheit der Grid-Computing-Ansätze reine Bibliothekslösungen wählen. Aus Gründen der Portabilität und der Sicherheit dürfen keine nativen Bibliotheken beteiligt sein und das System muss auf den Zielmaschinen mit möglichst geringem Installationsaufwand auskommen. Am besten wird dies durch eine reine Java-Bibliothek erfüllt, die direkt bei der Ausführung zusammen mit der Anwendung auf den Rechner geladen wird. Im Extremfall sind auch die in WWW-Browsern eingebauten virtuellen Maschinen als Ausführungsplattformen geeignet.

Arbeiten im Bereich Grid-Computing. Sohda et al. beschreiben in [92] eine Art DSM-System für Java-Objekte. Die gemeinsam benutzten Objekte müssen allerdings schon beim Programmstart bekannt sein und beim Laufzeitsystem registriert werden. Der Zugriff und die Freigabe dieser Objekte müssen über Konsistenzoperationen explizit gemacht werden, was fehleranfällig ist, u. a. weil die korrekte Schachtelung dieser Operationen nicht durch den Übersetzer geprüft werden kann. Der unterstützte Programmierstil ist SPMD. In der vorliegenden Arbeit können gemeinsam benutzte Objekte zu einem beliebigen Zeitpunkt während der Laufzeit des Programms erzeugt werden. Konsistenzoperationen werden als erweiterte Java-Synchronisationsprimitive realisiert, so dass der Übersetzer die korrekte Paarung von Sperr- und Freigabeoperationen automatisch erzeugen kann.

Krishnaswamy et al. [53] beschreiben ebenfalls ein System, das gemeinsam benutzte Objekte in einer verteilten Umgebung ermöglicht. Es stehen mehrere Konsistenzprotokolle zur Verfügung, um unabhängig von der Güte der Netzwerkanbindung des Klienten gute Reaktionszeiten für interaktive Anwendungen erreichen zu können. Mocha [97] hat zusätzlich die Möglichkeit, den Ausfall einzelner Knoten der weitläufig verteilten Umgebung zu tolerieren, indem der Zustand der gemeinsam benutzten Objekte auf mehreren Knoten vorgehalten wird. Die vorliegende Arbeit realisiert eine verteilte Programmierumgebung für Rechnerbündel mit einer schnellen Netzwerkverbindung. Der Fokus liegt nicht auf interaktiven Anwendungen, sondern ein Rechnerbündel wird wie ein großer Parallelrechner programmierbar gemacht.

ProActive PDC stellt aktive Objekte zur Verfügung und wurde von Caromel et al. entwickelt [13]. Mit wenigen Einschränkungen ist es möglich, beliebige Java-Objekte

zu aktiven Objekten zu machen, die entfernt ansprechbar sind und anders als bei RMI asynchrone Aufrufe mit konfigurierbarem Synchronisationsschema unterstützen. Die Klassen dieser aktiven Objekte müssen dafür nicht im Quellcode vorliegen, was durch eine Erzeugung von Hüllklassen zur Laufzeit ermöglicht wird. Ein Hüllobjekt leitet Methodenaufrufe an sein entferntes Implementationsobjekt weiter. Die notwendige Typkompatibilität zwischen Hüll- und Implementationsobjekt wird dabei durch Unterklassenbildung erreicht. Dieses Vorgehen ist notwendig, weil es sich um einen reinen Bibliotheksansatz handelt, führt aber zu einem nicht speichereffizienten Programm, da die Hüllobjekte alle Felder ihrer Implementationsobjekte erben. Dieses Vorgehen ist auch deswegen gefährlich, da Feldzugriffe auf solche aktiven Objekte die Felder des Hüllobjektes statt des Implementierungsobjektes treffen, der Übersetzer dieses Problem aber nicht erkennen kann. Die vorliegende Arbeit setzt hingegen auf eine Quellcodetransformation. Bibliotheken, die nicht im Quellcode vorliegen, bleiben aber trotzdem verwendbar. Die Codetransformation ermöglicht die Erzeugung von Hüllobjekten, die eine transparente Weiterleitung von Zugriffen auf Feldelemente und sogar Klassenvariablen erlauben.

Javelin [15] konzentriert sich auf den Aspekt, wie virtuelle Java-Maschinen über das Internet zu einer Grid-Computing-Laufzeitumgebung zusammengeschaltet werden können. Dafür wird ein Broker-Ansatz verwendet. Teilnahmswillige Rechner melden sich beim Broker an, der zu bearbeitende Aufgaben der Anwendung an die Rechner verteilt. Darüber hinaus stellt Javelin Primitive zum Nachrichtenaustausch, zur Barrierensynchronisation und für Operationen auf einem Linda-Tupelraum zur Verfügung. Die vorliegende Arbeit geht von einem Rechnerbündel als Zielplattform aus, das komplett unter der administrativen Kontrolle einer Organisation steht. Ziel ist es, eine bestehende parallele Anwendung mit möglichst geringem Aufwand für eine verteilte Umgebung zu portieren. Dafür eignet sich weder ein Broker-Ansatz noch die Nachrichtenkommunikation über einen Tupelraum.

JavaSymphony [20] bietet eine Schnittstelle an, mit der eine Applikation explizit die Topologie einer Rechnerkonfiguration für die Programmausführung anfordern kann. Außerdem bietet es die Möglichkeit, jedes Java-Objekt von einem entfernten Knoten aus fernzusteuern. Da diese Form der Fernsteuerung beim Zugriff auf das ferngesteuerte Objekt aber einen Mechanismus ähnlich der dynamischen Typintrospektion verwendet, büßt ein JavaSymphony-Programm einen Großteil der Eleganz von Java ein und verzichtet darüber hinaus noch auf die wertvolle Unterstützung durch den Übersetzer bei der Typprüfung. Die vorliegende Arbeit bietet im Unterschied dazu volle Typprüfung für alle Klassen und Objekte, egal ob es sich um ein lokales oder entferntes Objekt handelt. Aufgrund schlechter Laufzeiteffizienz wird auf dynamische Typintrospektion, wenn möglich, verzichtet.

Während Javelin einen SPMD-Programmierstil unterstellt und JavaSymphony sich am Java-Modell für Parallelität orientiert, bietet JICOS/CX [12] Unterstützung für eine Task-Struktur an. Sog. Tasks können dabei die Bearbeitung von Teilaufgaben anfordern und werden nach deren Erledigung benachrichtigt. Unterschiedliche und nicht voneinander abhängige Teilaufgaben können dann automatisch auf unterschiedlichen Rechnern parallel ausgeführt werden. Die vorliegende Arbeit macht das von Java her bekannte Konzept der Programmierung mit mehreren Kontrollfäden einsetzbar in einer verteilten Bündelumgebung. Darauf aufbauend könnte je nach Bedarf der Anwendung

auch eine Task-Struktur von nebenläufigen Aufgaben mit wechselseitigen Abhängigkeiten realisiert werden.

Während sich Grid-Ansätze vorwiegend um die reine Zusammenschaltung von weitverteilten und unter keiner einheitlichen Kontrolle stehenden Betriebsmitteln kümmern, realisiert die vorliegende Arbeit eine transparent verteilte Maschine, die es dem Anwendungsprogrammierer erlaubt, ein Rechnerbündel als großen Parallelrechner zu betrachten.

2.3 Präprozessorbasierte Ansätze

Ein präprozessorbasierter Ansatz geht von einem Applikationsprogramm aus, das im Quellcode vorliegt, und wendet eine Programmtransformation auf dieses an. Die Ausgabe des Präprozessors ist wieder ein Quellprogramm, das anschließend mit einem regulären Übersetzer in seine ausführbare Form gebracht wird. Bezogen auf Java für verteilt paralleles Rechnen können durch einen Präprozessor Konstrukte, die zum Umgang mit der verteilten Umgebung notwendig sind, durch eine Programmtransformation automatisch erzeugt werden, anstatt sie von Hand codieren zu müssen. Zwar können solche Konstrukte, wie im Abschnitt 2.2 gesehen, prinzipiell auch in einer Bibliothek gekapselt werden, dieser Weg ist aber aus zwei Gründen nicht angemessen. Erstens sind in einer verteilten Umgebung Konstrukte zum Umgang mit der Verteilung allgegenwärtig, was zur Folge hat, dass sich bei Verwendung einer Bibliothek die Programmierung größtenteils auf der Meta-Ebene abspielt. Dies führt zu unleserlichen und unnötig komplexen Programmen. Zweitens kann der Übersetzer bei der Programmierung auf der Meta-Ebene keine Typüberprüfung vornehmen, was zu vermehrten Laufzeitfehlern und damit zu weniger zuverlässigen Programmen führt. Ein Präprozessor vermeidet diese Probleme, indem neue Konstrukte zum Umgang mit der verteilten Umgebung direkt in die Sprache aufgenommen werden. Der Präprozessor ist dabei eine besonders modulare Lösung für eine Spracherweiterung, da weder Übersetzer noch Laufzeitumgebung der Sprache geändert werden müssen. Gerade bei Java ist das ein wesentlicher Vorteil, da eine virtuelle Java-Maschine (JVM) mit ihrem Laufzeitübersetzer für das dynamische Klassenladen und der automatischen Speicherbereinigung an sich schon ein äußerst komplexes System darstellt. Bei einer modularen Lösung behält man die Freiheit, verschiedene virtuelle Maschinen als Ausführungsplattform zu wählen, was bei der rasanten Entwicklung der Technologie der Laufzeitübersetzung unverzichtbar ist. Aufgrund dieser Vorteile setzt die vorliegende Arbeit ebenfalls einen Präprozessor ein.

Das Do! Projekt [56] macht speziell markierte Klassen entfernt ansprechbar, beschränkt aber die Erzeugung von Parallelität auf eine vordefinierte Bibliothek für sog. Tasks und Sammlungen von Tasks mit entsprechenden Koordinationsmechanismen. Durch Austausch dieser Bibliothek soll wahlweise eine verteilte oder eine lokale Ausführung des Programms ermöglicht werden. Als Kommunikationsbibliothek wird RMI verwendet. Die vorliegende Arbeit zielt dagegen auf eine transparent verteilte Umgebung ab, in welcher derselbe Programmierstil wie in einer nichtverteilten Umgebung möglich ist.

J-Orchestra [96] ermöglicht das automatische Partitionieren reiner Java-Anwen-

dungen. Die Anwendung wird räumlich partitioniert, indem Komponenten auf unterschiedliche Rechner gelegt werden. Applikationsklassen werden so verpackt, dass ein transparenter Fernzugriff auf sie möglich ist. Applikationsobjekte können dadurch auch während der Laufzeit ihren Ort innerhalb der verteilten Umgebung wechseln. J-Orchestra setzt die Transformation auf Bytecode-Ebene an und kann daher sowohl Applikationen partitionieren, die nicht im Quellcode vorliegen, als auch Systemklassen nachträglich entfernt ansprechbar machen. Damit wird eine Unterscheidung in lokale und entfernte Objekte vermieden, da potenziell jedes Objekt entfernt ansprechbar gemacht werden kann. Ineffizienzen durch eine übermäßig große Anzahl entfernter Objekte werden reduziert, indem nur ausgewählte Klassen transformiert werden und ein Objektexport lediglich bei Bedarf stattfindet. Die Kommunikation findet ebenfalls über RMI statt. Der Fokus von J-Orchestra liegt aber nicht auf verteilt parallelen Programmen, sondern auf zentralisierten Programmen, bei denen die Notwendigkeit besteht, Komponenten voneinander örtlich zu trennen. Beispielsweise kann so die graphische Benutzeroberfläche auf einem anderen Rechner als die Applikationslogik oder die Schicht für den Datenbankzugriff ablaufen. Da die Auftrennung der Anwendung sich an Gruppen von Klassen orientiert, scheint die Methodik zur Verteilung paralleler Applikationen nicht geeignet. In einer parallelen Applikation entscheidet nicht die Klasse eines Objektes über einen günstigen Ort für seine Erzeugung, sondern der Kontext seiner Benutzung.

FarGo [44] führt Annotationen für entfernte Referenzen ein. Solche speziell markierten Referenzen können zusätzliche Aktionen auslösen, wenn entweder Quelle oder Ziel der Referenz auf einen anderen Knoten migriert. So kann beispielsweise spezifiziert werden, dass die Migration der Quelle einer Referenz die Migration des Zieles auf denselben Knoten anstößt. FarGo unterscheidet zwischen entfernten und lokalen Objekten, wobei ein entferntes Objekt zusammen mit der transitiven Hülle aller referenzierten lokalen Objekte eine Objektgruppe bildet. Von den Objekten einer Objektgruppe kann nur die Wurzel außerhalb der Gruppe referenziert werden. FarGo bildet Kommunikation und entfernte Referenzen ebenfalls auf RMI ab. FarGo adressiert nur Lokalitätsaspekte, die durch eine geschickte Anordnung der Objekte in einem verteilten Adressraum zu erreichen sind. Die vorliegende Arbeit deckt auch DSM-Aspekte und kollektive Operationen ab. Die in FarGo vorgeschlagenen Annotationen für entfernte Referenzen wären möglicherweise auch in der vorliegenden Arbeit gewinnbringend einsetzbar.

Die vorliegende Arbeit basiert auf Transformation von annotierten Java-Klassen, die dadurch entweder in entfernte oder replizierte Klassen der verteilten Umgebung verwandelt werden. Grundlagen für die Transformation entfernter Klassen wurden in [85] gelegt. Die Transformation ermöglicht, dass alle Klassen transparent, d.h. wie reguläre Klassen, verwendet werden können. Code für den Verteilungsaspekt wird automatisch erzeugt. Statt RMI kommt ein neuentwickelter Transportmechanismus zum Einsatz, der für eine Bündelumgebung optimiert ist und transparent verteilte Kontrollfäden realisiert (vgl. Abschnitt 5.3).

2.4 Lösungen mit konventionellem Übersetzer

Der im letzten Abschnitt beschriebene Ansatz, mit einem Präprozessor ein um Konstrukte für verteilt paralleles Programmieren erweitertes Java in reines Java zurück zu transformieren, muss mit der zusätzlichen Herausforderung kämpfen, dass das Ergebnis der Transformation an die restriktiven Sprachregeln bezüglich Typsicherheit und Zugriffsschutz gebunden bleibt. Diesen Einschränkungen versuchen Arbeiten aus dem Wege zu gehen, die ein Java für verteilte Systeme mit einem konventionellen Übersetzer direkt in plattformabhängigen Code für die Zielmaschine übersetzen. Ein weiterer Grund, den Weg der direkten Übersetzung einzuschlagen, war die anfänglich sehr schlechte Laufzeiteffizienz der verfügbaren virtuellen Maschinen. Mit der rasanten Entwicklung im Bereich der Laufzeitübersetzung in modernen virtuellen Maschinen dürfte dieser Gesichtspunkt, der von der Problematik der Erweiterung für verteilte Umgebungen unabhängig ist, nur noch eine untergeordnete Rolle spielen.

Spar [100] ist ein modifiziertes Java, in dem Kontrollfäden durch potenziell parallele Konstrukte wie `foreach` ersetzt und echte mehrdimensionale Felder veränderlicher Länge eingeführt werden. Spar ist gedacht für die Lösung von feldbasierten Problemen auf Systemen mit verteiltem Speicher. Spar fungiert dabei als Eingabesprache für ein Transformationssystem, das den Verteilungsaspekt automatisch zum parallelen Spar-Programm hinzugefügt. Die Verteilung von Code und Daten bleibt für die Anwendung transparent. Als Ausgabe liefert das Transformationssystem den C++-Code, der in ein ausführbares Programm für die Zielplattform übersetzt wird [55]. Die vorliegende Arbeit richtet den Fokus auf objektorientierte statt feldbasierte Programme. Das `foreach`-Konstrukt von Spar kollidiert mit dem Konzept paralleler Kontrollfäden in Java.

Ausgangspunkt von Manta [67] war die Beschleunigung des entfernten Methodenaufrufs durch native Übersetzung, die Anbindung an Hochleistungs-Kommunikationshardware und die Erzeugung von spezialisierten Versenderoutinen für Objekte unter Kenntnis ihres Speicherlayouts im Übersetzer. Die vorliegende Arbeit zeigt, dass ähnlich effiziente Versenderoutinen und die Anbindung an Hochleistungs-Kommunikationshardware ohne Neuentwicklung der virtuellen Maschine in reinem Java möglich sind.

Hicks et al. beschreiben in [43] einen Übersetzer, der Java direkt in ein ausführbares Programm für eine verteilte Zielplattform übersetzt. Kontrollfäden des Java-Programms werden dabei zu verteilten Kontrollfäden, die pro Knoten auf leichtgewichtige lokale Kontrollfäden abgebildet werden. Das Java-Objektmodell wird für alle Objekte unverändert beibehalten. Es wird nicht zwischen entfernt und nur lokal referenzierbaren Objekten unterschieden. Daraus resultiert eine extrem feine Granularität, da jeder Methodenaufruf zu einem Fernaufruf werden kann. Um Einfluss auf die Programmeffizienz zu haben, wird die Sprache erweitert. Mit diesen Erweiterungen ist es möglich, ein Objekt auf einem speziellen Knoten zu erzeugen, eine statische Methode auf einem angegebenen Knoten auszuführen, zu erfragen, ob ein Objekt lokal oder entfernt liegt, und eine explizite Kopie eines Objektes auf einem anderen Knoten zu erzeugen. Die vorliegende Arbeit vermeidet die Probleme feingranularer Verteilung, indem entfernte und replizierte Klassen entsprechend markiert werden. Dadurch entstehen automatisch Gruppen von zusammengehörigen Objekten (ein entferntes Objekt

und alle von ihm referenzieren regulären Objekte), die als Ganzes verteilt werden und daher garantiert lokal aufeinander zugreifen können.

Einen anderen Weg gehen die Autoren von Hyperion [69, 2]. Hyperion ist ein objektbasiertes DSM-System, welches das Java-Speichermodell ausnutzt, um Kopien derjenigen Objekte, die auf einem Knoten gebraucht werden, dort lokal vorzuhalten und die Konsistenz der Zugriffe aus unterschiedlichen Kontrollfäden konsistent zu halten. Das Java-Speichermodell erlaubt jedem Kontrollfaden eine lokale Sicht auf den gemeinsamen Speicher, die nur an Synchronisationspunkten konsistent sein muss. Das Software-DSM muss dazu allerdings alle Änderungen, die ein Kontrollfaden zwischen Synchronisationspunkten durchführt, Byte-genau mitprotokollieren, um sie beim Aktualisieren der Kopien einspielen zu können. Die vorliegende Arbeit setzt bei Konsistenzoperationen an replizierten Objekten ebenfalls eine genaue Zustandsfortschreibung nur der geänderten Instanzvariablen eines Objektes ein. Allerdings wird diese plattformunabhängig mit Hilfe eines Präprozessors für reine Java-Objekte realisiert.

2.5 Native Implementierungen einer verteilten virtuellen Maschine

Der Übergang zwischen der Übersetzung von Java in ein ausführbares Programm für die Zielmaschine und einer verteilten Implementierung der virtuellen Maschine selbst ist fließend. Will ein nativer Java-Übersetzer den vollen Umfang der Sprache unterstützen, so muss er das dynamische Nachladen von Klassen erlauben und dafür entweder auf Bytecode-Interpretation zurückfallen oder einen Laufzeitübersetzer beinhalten. Arbeiten in der Kategorie „nativer Übersetzer“ legen das Hauptaugenmerk auf die effiziente Ausführung eines möglicherweise für den Verteilungsaspekt abgewandelten Java-Dialektes, wogegen eine verteilte Implementierung der virtuellem Maschine die Ausführung von unmodifizierten Java-Programmen auf verteilten Systemen anstrebt. Dennoch kann die Einordnung einer Arbeit entweder in die Kategorie „native Übersetzung“ oder „verteilte virtuelle Maschine“ kontrovers diskutiert werden.

2.5.1 Hauptspeicherorganisation

Es gibt drei Strategien, wie eine so verteilte virtuelle Maschine mit Objekten der Anwendung umgehen kann. Entweder basiert der Speicher der virtuellen Maschine auf einem DSM-System, das allen Knoten die Illusion eines globalen gemeinsamen Speichers bietet. In diesem Fall finden alle Objektzugriffe lokal statt, da die betroffenen Daten bei Bedarf vom DSM-System an den Ort des Zugriffs transportiert wurden. Die verteilte virtuelle Maschine muss in diesem Fall „nur“ die parallelen Kontrollflüsse der Anwendung auf unterschiedliche Knoten verteilen. Das zugrundeliegende DSM-System sorgt dafür, dass die benötigten Objekte beim Zugriff lokal vorliegen. Die zweite Strategie speichert Objekte explizit auf einem dedizierten Knoten, macht diese aber von jedem Knoten aus entfernt ansprechbar. Bei einer verteilten virtuellen Maschine geschieht ein solcher entfernter Zugriff vollkommen transparent. Die virtuelle Maschine kann entscheiden, ob der Kontrollfluss zum Knoten des Objektes wandert oder aber

das Objekt auf denjenigen Knoten migriert, der den Zugriff ausführt. Die dritte Strategie ist eine Mischstrategie. Sie identifiziert Objekte, deren Zustand auf verschiedenen Knoten wie beim DSM-System repliziert wird, und solche, auf die entfernt zugegriffen werden kann. In der vorliegenden Arbeit ist ebenfalls eine Mischstrategie zwischen verteilt gespeichertem und repliziertem Zustand möglich. Die Auswahl wird über eine Kennzeichnung pro Klasse gesteuert.

2.5.2 Objektmodell

Bei einer verteilten virtuellen Maschine gibt es unabhängig von der internen Organisation des Hauptspeichers aus Sicht der Anwendung nur eine Sorte von Objekten. Für alle Objekte gilt dieselbe Aufruf- und Parameterübergabesemantik – wenn die verteilte virtuelle Maschine den gemeinsamen Objektraum durch Fernaufruf realisiert, dann kann ein solcher Fernaufruf prinzipiell sowohl auf einer einfachen Zeichenkette wie auf einem Datenbankobjekt stattfinden. Für die Anwendung ist dieser Fernaufruf bis auf seine Ausführungsgeschwindigkeit aber nicht von einem lokalen Zugriff unterscheidbar.[3]

Die Forderung eines einheitlichen Objektmodells gibt es schon so lange wie verteilte objektorientierte oder objektbasierte[4] Sprachen. Schon 1987 hat Hutchinson mit Emerald [45] eine verteilte objektbasierte Sprache mit uniformem Objektmodell vorgeschlagen, bei der auf jedes Objekt potenziell entfernt zugegriffen werden kann. Der Übersetzer und das Laufzeitsystem treffen Vorkehrungen, um zur Laufzeit lokale Referenzen und lokale Zugriffe effizient bearbeiten zu können. Das uniforme Objektmodell soll die verteilte Programmierung vereinfachen. Allerdings ist Emerald nicht verteilungstransparent: Der Programmierer hat mit einer ganzen Reihe ortsabhängiger und ortsbeeinflussender Operationen die Möglichkeit, den Ort von Objekten zu erfahren, Objekte von Knoten zu Knoten umzuziehen oder dieses Umziehen zu verhindern. Von einer verteilten Java-Maschine würde man dagegen erwarten, dass sie eine Standard-Java-Applikation, die nicht für die verteilte Ausführung vorbereitet ist, effizient in einer verteilten Umgebung abarbeitet, ohne dass der Programmierer dafür spezielle Vorsorge getroffen haben müsste. Außerdem erlaubt Emerald keine Replikation von Objekten, was bei manchen Zugriffsmustern zwangsläufig zu schlechter Skalierung führt.

2.5.3 Einordnung relevanter Arbeiten

Java/DSM [108] und JESSICA [65] realisieren den gemeinsamen Hauptspeicher über das seitenbasierte DSM-System TreadMarks [51]. Dies deutet auf Probleme hin, die beim unabhängigen Zugriff auf Objekte derselben Seite entstehen. Beide Projekte setzen einen Interpretierer anstelle eines Laufzeitübersetzers ein, was nicht konkurrenzfähige Leistungsdaten bedingt.

[3] Intern wird ein solcher Fernzugriff ebenfalls über Stellvertreter organisiert. Innerhalb der virtuellen Maschine ist aber völlige Transparenz erreichbar, da die Stellvertreter dort nicht an das Java-Typsystem gebunden sind.

[4] Objektbasierte Sprachen ersetzen Klassen, Vererbung und Instanzierung durch Prototypen und Erzeugung von Duplikaten.

JESSICA2 [109] als Nachfolgeprojekt von JESSICA integriert Laufzeitübersetzung mit Strategien für die Migration von Kontrollfäden. Das Problem des seitenbasierten DSM-Systems bleibt bestehen und es sind keine Leistungsdaten des Laufzeitübersetzers im Vergleich zu einer konventionellen virtuellen Maschine publiziert worden.

Jackal [102] baut auf dem nativen Übersetzer Manta auf und erzeugt Code für ein heterogenes objektbasiertes DSM-Subsystem.

Bei einer DSM-basierten Implementierung der virtuellen Maschine werden die Kontrollfäden auf die Rechner verteilt. Bei Zugriffen auf gemeinsam benutzte Daten werden diese an den Ort des Zugriffs transportiert. Es findet nie ein Fernaufruf statt. Jede Aktivität ist an ihren Rechner gebunden und wechselt ihren Ausführungsort bei normaler Operation nie. Um ungünstiger Lastverteilung entgegenzuwirken, können wie bei JESSICA2 komplette Kontrollfäden vom Laufzeitsystem auf einen anderen Knoten umgezogen werden. Bei einem solchen Umzug wechselt ein Kontrollfaden mitsamt seiner „Vergangenheit“ auf einen anderen Knoten. Dies ist anders als beim Fernaufruf, bei dem der Kontrollfaden auf den Ausgangsknoten zurückkehrt, sobald die Schachtelung der Aufrufe wieder zum Kontext zurückkehrt, von dem der Fernaufruf ausging.

cJVM [3] entscheidet pro Objekt automatisch, ob Fernzugriff oder Replikation stattfindet, wenn von mehreren Knoten auf ein Objekt zugegriffen wird. cJVM verfolgt damit einen Hybrid-Ansatz zwischen objektbasiertem DSM und verteiltem Objektraum. Aufgrund der Vielzahl von Objekten in einem Programm gibt es für die Klassifikation in replizierte und entfernt ansprechbare Objekte und für die Verteilung der letztgenannten auf die Knoten sehr viele Freiheitsgrade. Im Vergleich dazu hat ein DSM-System lediglich die Möglichkeit, eine relativ geringe Menge von Kontrollfäden auf die Rechenknoten zu verteilten. In [4] wird eine Reihe von Optimierungstechniken beschrieben, mit denen cJVM entfernt ansprechbare und replizierte Objekte automatisch klassifiziert. Allerdings scheint auch in cJVM kein Laufzeitübersetzer integriert zu sein.

2.5.4 Bewertung des Ansatzes einer verteilten virtuellen Maschine

Eine verteilte Implementierung der virtuellen Maschine verspricht die größtmögliche Transparenz für die Anwendung, da idealerweise ein unverändertes Java-Programm, das lediglich die in Java eingebauten Mittel zum Ausdruck von Parallelität nutzt, durch Ausführung in der verteilten virtuellen Maschine zu einem verteilt parallelen Programm wird. Die virtuelle Maschine entscheidet dabei selbständig, auf welchem Knoten Objekte der Anwendung erzeugt werden und was bei einem Zugriff auf ein solches Objekt von einem entfernten Knoten passiert. Es gilt dieselbe Semantik ausnahmslos für alle Objekte, da die virtuelle Maschine vollständig von der Verteilung abstrahiert.

In einer verteilten virtuellen Maschine kann der Laufzeitübersetzer möglicherweise die Verteilung bei Optimierungen mitberücksichtigen. Damit sind eventuell bessere Ergebnisse möglich als bei der strikten Trennung in (nichtverteilte) virtuelle Maschine und verteilter Umgebung.

Aufgrund vollständiger Transparenz der Verteilung ist die Effizienz eines Programms abhängig von der Güte der automatischen Verteilungsstrategie. Da Java selbst

keine Konstrukte für Verteilung beinhaltet, hat der Programmierer keine Möglichkeit, sein Programm auf eine effiziente Ausführung in einer verteilten Umgebung vorzubereiten. Es ist nicht klar, ob eine *automatische* Verteilung eines *manuell* parallelisierten Programms optimale Ergebnisse liefern kann, weil die Parallelisierungsstrategie durchaus Einfluss darauf haben kann, ob sich das entstehende Programm verteilen lässt oder nicht.

Indem die Interna der virtuellen Maschine wie Laufzeitübersetzer und Speicherbereiniger mit dem Verteilungsaspekt verknüpft werden, können solche Ansätze nicht mehr direkt von der Weiterentwicklung dieser Technologien für den Massenmarkt von Einprozessor- und kleinen SMP-Maschinen[5] profitieren. Es ist zudem unklar, warum die Abstraktion von der zugrundeliegenden Hardware und dem Betriebssystem, welche die virtuelle Java-Maschine vornehmlich leistet, nicht klar von der Beherrschung des Verteilungsaspekt zu trennen ist.

Die vorliegende Arbeit verzichtet auf den Eingriff in die virtuelle Maschine und zeigt, wie durch reine Benutzung dieser Technologie zusammen mit einem Präprozessor eine plattformunabhängige verteilte Programmierumgebung mit Java-*ähnlicher* Semantik realisierbar ist. Dem Programmierer wird dadurch ein Großteil der *Handarbeit* abgenommen, die für eine verteilte Ausführung notwendig ist. Dadurch, dass die Verteilung aber nicht vollständig vor ihm verborgen wird, hat er die Möglichkeit eine *effiziente* verteilt parallele Ausführung bei minimaler Änderung des Programms zu erreichen. Indem er dafür zusätzlich in die Sprache eingeführte Konstrukte verwendet, ist er gezwungen, schon bei der Parallelisierung an den Verteilungsaspekt zu denken und so eine geeignete Parallelisierungsstrategie zu wählen.

2.6 Replikation

Ein wesentlicher Beitrag der vorliegenden Arbeit ist eine neuartige Form von Replikation. Daher beleuchtet dieser Abschnitt verwandte Arbeiten unter dem Aspekt von Replikationsstrategien in verteilten Umgebungen und vergleicht deren Methoden und Ziele. Der Fokus liegt hierbei auf verteilt parallelen, objektorientierten Sprachen, die auf plattformunabhängiges Programmieren von Rechnerbündeln abzielen.

Tabelle 2.1 klassifiziert Replikationsstrategien in verwandten, teilweise auch nicht Java-basierten, Arbeiten nach verschiedenen Kriterien.

- Die ersten beiden Merkmale *Plattformunabhängigkeit* und *Ansatz* ermöglichen eine Einordnung in die Übersicht über verwandte Arbeiten der Abschnitte 2.1 bis 2.5. Für die nicht Java-basierten Projekte entfällt diese Einordnung.

- Das Merkmal *Replikation von* gibt an, was in dem jeweiligen Ansatz replizierbar ist. Das können entweder einzelne Objekte, ganze Graphen von Objekten oder Felder sein.

- Das Merkmal *Eigenschaften* informiert genauer über die Replikationsstrategie.

[5] *symmetric multi processor*

Projekt	Ansatz			Repl. von			Eigensch.			Mittel				Ziele		
	plattformunabhängig	Bibliothek	Transformation	VM-Modifikation	Objekten	Graphen	Feldern	partiell	mit Direktzugriff	automatisch	Natives DSM-System	Objektkopie	Gruppenoperation	Inkrement	Lokalität	Datenparallelität	Fehlertoleranz
Braid [105]	–		n.a.				X	(X)		–		X	(X)		(X)	X	
Orca [36]	–		n.a.				X	(X)		–		X	(X)		(X)	X	
HPJava [61]	–			(X)			X	X		–		X	(X)		X	X	
ProActive [5]	X	X			X			–	–	–			X			X	
Mocha [97]	X	X			X			–		–		X					X
Manta/RepMI [66]	–			(X)	X	X		–	–	(X)			X		X		
Javanaise [34]	X		X		X	X		–	–	–		X			X		
Kan [47]	X		X		X			–	–	–		X			X		
JDSM [92]	X	X			X		X	–	–	–		X			X		
Aleph [42]	X	X			X			–	X	–		X			X		
Charlotte [8]	X	X			(X)			–	–	–		X		(X)	X		
Java/DSM [108]	–			X	X		X	–	X	X	X				X		
Hyperion [69]	–			X	X		X	–	X	X	X				X		
MultiJav [14]	–			X	X		X	–	X	X	X			X	X		
Jackal [101]	–			X	X		X	–	X	X	X				X		
CoJVM [64]	–			X	X		X	–	X	X	X				X		
JESSICA2 [109]	–			X	X		X	–	X	X	X				X		
cJVM [4]	–			X	X		X	–	?	X		X			X		
Diese Arbeit	X		X		X	X	X	X	X	–				X	X	X	

Tabelle 2.1: Übersicht über Methoden und Ziele von Replikation in verteilten objektorientierten Umgebungen.

  - Eine Struktur ist *partiell repliziert*, wenn nur gewisse Teile tatsächlich repliziert werden, während die übrigen Teile nur auf einem der Knoten gespeichert sind.
  - Replikate *mit Direktzugriff* benötigen keine Indirektion über einen lokalen Stellvertreter. Direktzugriff ermöglicht eine bessere Leistung, da die Kosten für den indirekten Zugriff über einen Stellvertreter eingespart werden.
  - Wenn im Programm kein Hinweis darauf notwendig ist, dass eine bestimmte Struktur repliziert werden soll, findet die Replikation *automatisch* statt.

- Die Projekte verwenden unterschiedliche *Mittel*, um Replikation und die Konsistenzerhaltung der Replikate zu realisieren.

  - Der Hauptspeicher einer verteilten Umgebung kann über ein *natives DSM-System* verwaltet werden, um automatische Replikation zu erhalten. In diesem Fall wird die Aufgabe der Konsistenzerhaltung an das DSM-System delegiert.
  - Es kann explizit eine *Objektkopie* des Originalobjektes angefertigt werden, wenn auf ein Objekt von mehreren Knoten aus lesend zugegriffen wird. Wird ein Schreibzugriff angefordert, müssen alle anderen Kopien invalidiert werden.
  - Ein Aggregat von Objekten kann auf mehreren Knoten konsistent gehalten werden, indem Modifikationsoperationen immer auf alle Mitglieder des Aggregates gleichzeitig angewendet werden. Bei einer solchen Operationen spricht man auch von einer *Gruppenoperation*, da sie statt auf ein einzelnes Objekt auf eine ganze Gruppe von Objekten wirkt.
  - Schließlich können Änderungen, die auf einem Replikat stattgefunden haben, anschließend *inkrementell* an alle anderen Replikate verteilt werden.

- *Ziele* der Replikation werden in der letzten Tabellenspalte vermerkt.

  - Replikation kann zur *Lokalitätsoptimierung* eingesetzt werden, weil auf alle Replikate verteilt parallel, lesend zugegriffen werden kann (vgl. Abschnitt 2.6.1).
  - Die Fehlertoleranz eines Programms kann durch Replikation erhöht werden, weil dadurch die Wahrscheinlichkeit steigt, dass immer einige Replikate überleben, auch wenn manche Knoten der verteilten Umgebung aufgrund von Fehlern nicht mehr verfügbar sind (vgl. Abschnitt 2.6.3).
  - Unterstützen replizierte Objekte nebenläufige Schreiboperationen, so kann das für *datenparallele Operationen* genutzt werden (vgl. Abschnitt 2.6.2).

Die folgende Darstellung ist nach den Zielen der Replikation gegliedert. Jeder Abschnitt beginnt mit einer Erklärung, wie das jeweilige Ziel mittels Replikation erreicht werden kann, und nimmt danach eine Einordnung der relevanten Arbeiten vor.

2.6.1 Replikation zur Lokalitätsoptimierung

Ein Hauptziel von Replikation ist die Erhöhung von Zugriffslokalität. Alle DSM-Systeme basieren auf dieser Eigenschaft. Wenn derselbe Zustand auf mehreren Rechnern repliziert vorrätig gehalten wird, kann er dort parallel lokal ausgelesen werden. Solange dieser Zustand nicht verändert wird, kann durch Replikation mit einem Fernzugriff, der den Zustand von einem anderen Rechner herkopiert, eine beliebige Zahl von darauffolgenden Lesezugriffen bedient werden. Erst wenn ein replizierter Zustand modifiziert wird, muss sichergestellt werden, dass auf anderen Rechnern nicht mit dem veralteten Zustand weitergerechnet wird. Das Hauptunterscheidungsmerkmal dabei ist die Strategie, mit der sichergestellt wird, dass die Anwendung trotz Replikation mit konsistenten Zuständen arbeitet.

Die im folgenden besprochenen Arbeiten verwenden (Objekt-)Replikation ausschließlich zur Lokalitätsoptimierung verteilt paralleler Lesezugriffe auf gemeinsame Objekte. Die Aufstellung ist nach der Strategie für die Konsistenzhaltung der Replikate gegliedert.

Konsistenz mittels Gruppenoperationen

Manta/RepMI [66] realisiert Replikation von abgeschlossenen Objektgraphen mittels Gruppenoperationen. Der Programmierer deklariert eine replizierte Klasse über eine vordefinierte Schnittstelle. Bei der Erzeugung einer Instanz einer replizierten Klasse entstehen Replikate auf allen Knoten der verteilten Umgebung. Schreibende Methodenaufrufe an einem Replikat werden über Rundruf an allen Replikaten gleichzeitig und global geordnet ausgeführt. Lesende Zugriffe werden lokal abgearbeitet. Der Übersetzer versucht, Schreibzugriffe automatisch zu erkennen. Scheitert diese Erkennung (ein vermeintlich lesender Methodenaufruf versucht eine Schreiboperation durchzuführen), wird ein lokal begonnener Zugriff zur Laufzeit zurückgesetzt und auf allen Replikaten neu gestartet.

Wird beispielsweise eine Klasse `A` als repliziert deklariert, so wird bei der Erzeugung einer Instanz von `A` ein Replikat auf allen Knoten der verteilten Umgebung angelegt. Wird auf dieser Instanz `x` eine Methode `foo()` aufgerufen, die den Zustand von `x` nicht verändert, so wird diese Methode am lokalen Replikat ausgeführt. Wird dagegen eine Methode `bar()` aufgerufen, die den Zustand von `x` verändert, wird `bar()` implizit an allen Replikaten von `x` gleichzeitig und in einer global festgelegten Reihenfolge aufgerufen.

Dadurch, dass alle modifizierenden Zugriffe auf ein repliziertes Objekt an allen Replikaten mit denselben Argumenten in einer einheitlichen Reihenfolge ausgeführt werden, ist nach Abarbeitung jedes Aufrufs Konsistenz gewährleistet. Damit die Mehrfachdurchführung von Operationen auf Kopien von Objekten diese tatsächlich in einem konsistenten Zustand hinterlässt und keine unerwarteten Nebeneffekte mit sich bringt, sind gravierende Einschränkungen notwendig: Der replizierte Objektgraph muss zum einen vollständig von der Außenwelt abgeschirmt werden und zum anderen ist es nicht möglich, Referenzen auf entfernte oder andere replizierte Objekte in einem replizierten Objekt zu speichern. Damit die parallele Ausführung von modifizierenden Methoden auf allen Replikaten diese in einem konsistenten Zustand hinterlässt, muss verhindert

werden, dass aus solchen Methoden auf Daten zugegriffen wird, die nicht zum replizierten Objekt gehören und sich daher in einem inkonsistenten Zustand befinden könnten (z. B. statische Variablen). Flösse ein solcher inkonsistenter Zustand in die Berechnung einer parallel ausgeführten modifizierenden Methode ein, lieferte diese kein konsistentes Ergebnis und könnte die Konsistenz des replizierten Objektes nicht sicherstellen. Enthielte der Zustand eines replizierten Objektes Referenzen auf entfernte oder andere replizierte Objekte, ergäben sich unerwartete Effekte beim Methodenaufruf an solchen Referenzen aus modifizierenden Methoden. Da solche modifizierenden Methoden zur Konsistenzerhaltung intern parallel an allen Replikaten ausgeführt werden, führte dies zu unerwarteten Mehrfachaufrufen an den referenzierten externen Objekten.

Mantas replizierte Objekte kapseln lediglich Informationen, die auch über Rundruf verteilt werden könnten. Es muss immer die gesamte Information auf alle Knoten verteilt werden, da alle Berechnungen an den Replikaten identisch ablaufen müssen. Führen Methoden außer dem reinen Schreiben in Instanzvariablen des Replikats auch noch Berechnungen aus, ist das kritisch, weil diese Berechnungen auf allen Replikaten ausgeführt werden und so Rechenzeit verschwenden. Die in der hier vorliegenden Arbeit eingesetzten replizierten Objekte leiden unter keinem dieser beschriebenen Probleme. Es können sowohl Referenzen auf entfernte sowie auf andere replizierte Objekte als replizierter Zustand gespeichert werden. Außerdem ist es beim hier vorgestellten Ansatz unkritisch – und sogar explizit vorgesehen, längere Berechnungen in schreibenden Methoden replizierter Objekte durchzuführen.

Konsistenz durch Objektkopie

Eine beliebte Methode bei Arbeiten, die eine plattformunabhängige verteilte Umgebung, basierend auf einer unveränderten virtuellen Java-Maschine, realisieren, ist es, Replikation durch einfache Kopien von Objekten zu realisieren. Wird ein Replikat auf einem Knoten angefordert, auf dem keine aktuelle Kopie vorrätig ist, wird das Objekt mit allen von ihm referenzierten Objekten dorthin kopiert. Auf dem anfordernden Knoten kann das Replikat dann anstelle des Originals lokal benutzt werden. Java bietet die dazu notwendigen Serialisierungsmechanismen bereits in der Basisbibliothek an. Allerdings ist damit keine Konsistenzerhaltung der durch Kopieren entstandenen korrespondierenden Objekte auf unterschiedlichen Knoten möglich. Nach dem Kopieren zerbricht diese Korrespondenz zwischen Originalobjekt und Kopie. Wird anschließend ein Replikat geändert, können die Änderungen nicht auf den anderen Replikaten nachgezogen werden. Die einzige Möglichkeit nach einer lokalen Änderung ist es, alle Kopien zu invalidieren und das Objekt erneut an die Stellen nachfolgender Lesezugriffe zu transportieren.

Dieses Vorgehen birgt zwei gravierende Probleme. Zum einen muss auch bei einer minimalen Änderung immer eine große Menge an Daten erneut übertragen werden. Dies verbraucht Netzwerkbandbreite und hat einen erheblichen Aufwand zur Folge, da viele Objekte neu instanziert und die veralteten Kopien vom Speicherbereiniger abgeräumt werden müssen. Zum anderen darf der Anwendung nie eine direkte Referenz auf ein repliziertes Objekt ausgehändigt werden. Andernfalls besteht die Gefahr, dass nach einer entfernt durchgeführten Zustandsfortschreibung Referenzen auf ver-

altete Kopien zurückbleiben. Um dieses Konsistenzproblem zu vermeinden, darf der Zugriff nur über einen Stellvertreter durchgeführt werden. Die zusätzliche Indirektion reduziert aber den Geschwindigkeitsgewinn, auf den die Replikation eigentlich abzielt. Diese Replikationsform eignet sich folglich höchstens dann, wenn Schreibzugriffe sehr selten sind oder replizierten Zustand immer komplett ändern.

Javanaise [34] realisiert einen gemeinsamen Objektraum für internetweite verteilte Anwendungen. Statt wie RMI dafür auf Fernzugriffe zu setzen, wird ein Objekt im Javanaise-Modell immer an den Ort des Zugriffs transportiert. Der Zugriff auf das Objekt kann dann lokal auf der Klienten-Seite durchgeführt werden. Nach der Änderung eines Replikats werden alle anderen Kopien invalidiert und bei der nächsten Anforderung neu erzeugt. Um auszuschließen, dass dadurch veraltete Kopien zurückbleiben, darf die Anwendung keine direkte Referenz auf das Replikat oder eines von ihm referenzierten Objektes erhalten. Dies wird erreicht, indem ein repliziertes Objekt nur über einen Stellvertreter angesprochen werden kann. Wird das lokale Replikat durch eine neue Kopie ersetzt, wird die Referenz im Stellvertreter auf die neue Kopie geändert. Da technisch gesehen der replizierte Zustand eines Javanaise-Objektes aus vielen einzelnen Java-Objekten bestehen kann, muss der Stellvertreter weiterhin gewährleisten, dass die Anwendung und der replizierte Zustand keine gemeinsamen Objekte referenzieren, um zu verhindern, dass die Anwendung auf veraltete Kopien eines replizierten Teilzustandes zugreift. Um dies garantieren zu können, müssen alle Methoden, die am Stellvertreter eines replizierten Objektes aufgerufen werden, alle ihre Argumente und Rückgabewerte kopieren. Somit besitzen Methodenaufrufe an replizierten Objekten in Javanaise Kopiersemantik. Bei der vorliegenden Arbeit sind dagegen weder der Zugriff über ein Stellvertreterobjekt noch Kopiersemantik beim Zugriff auf ein Replikat notwendig, um Referenzen auf veraltete Kopien zu vermeiden.

Objekte in Kan [47] migrieren entweder zum Ort ihres Zugriffs oder können für Leseoperationen repliziert werden. Schreiboperationen auf so replizierte Objekt werden in einem Fernaufruf auf dem Heimatknoten des Objektes abgewickelt. Dieser Ansatz hat im wesentlichen dieselben Nachteile wie Javanaise, da bei einer Zustandsfortschreibung auf dem Heimatknoten eines Objektes ebenfalls alle existierenden Kopien invalidiert werden müssen. Lediglich das Auffinden der aktuellen Version eines replizierten Zustandes vereinfacht sich. Dafür stellt der Heimatknoten einen Flaschenhals dar, da er für die Abwicklung aller Schreiboperationen auf einem replizierten Objekt verantwortlich ist. Im Unterschied dazu haben in der vorliegenden Arbeit replizierte Objekte keinen ausgezeichneten Heimatknoten.

JDSM [92] ist spezialisiert auf datenparallele Anwendungen. Es simuliert ein objektbasiertes DSM-System in der virtuellen Maschine. Vor dem Zugriff auf ein Objekt muss dieses über eine Bibliotheksfunktion angefordert und danach wieder freigegeben werden. Die Konsistenzhaltung wird ebenfalls über das Erzeugen neuer Kopien nach Schreibzugriffen realisiert. Alle replizierten Objekte müssen während einer Initialisierungsphase beim Programmstart registriert werden. In der vorliegenden Arbeit ist es demgegenüber möglich, neue replizierte Objekte zu jeder Zeit während des Programmlaufs zu erzeugen.

Aleph [42] offeriert nach Anforderung eine direkte Referenz auf das replizierte Objekt. Da der Zustand dieses Objektes nach einer Modifikation auf einem anderen Knoten aber nicht fortgeschrieben, sondern eine neue Kopie erzeugt wird, besteht die

Gefahr, dass Referenzen auf veraltete Kopien zurückbleiben. Aleph nimmt diese Gefahr von Inkonsistenzen in Kauf, um bessere Geschwindigkeit für den lokalen Replikatzugriff zu erreichen. Die vorliegende Arbeit bietet ebenfalls direkten Zugriff auf ein Replikat, schließt aber trotzdem Referenzen auf veraltete Kopien mittels inkrementeller Zustandsfortschreibung aus.

In Charlotte [8] werden nicht Objekte, sondern nur einzelne Basistypen repliziert. Dafür muss jede Instanzvariable in ein extra Objekt eingepackt werden. Dieses repräsentiert einen einzelnen replizierten Wert (beispielsweise eine einzelne Ganzzahl). Um nicht für jedes dieser zahllosen Einzelwert-Objekte Konsistenzoperationen beim Zugriff durchführen zu müssen, können diese gebündelt werden. Da jeder Wert in ein eigenes Objekt verpackt werden muss, scheint dieser Ansatz einen großen Mehraufwand zu verursachen. In der hier vorliegenden Arbeit können nicht nur Basistypen, sondern beliebige Klassen als repliziert deklariert werden. Eine solche Deklaration hat zur Folge, dass ihre kompletten Instanzen zu replizierten Objekten werden. Diese replizierten Objekte können beliebige Instanzvariablen besitzen, die alle zum replizierten Zustand des Objektes gehören. Auf diese Weise kapselt ein Objekt eine größere Menge replizierten Zustandes. Dieser Ansatz verspricht bessere Effizienz und passt zum objektorientierten Geheimnisprinzip.

Das komplette Kopieren eines Objektes ist eine inadäquate Methode für die Zustandsfortschreibung von Replikaten. Die Objektkopie verursacht erheblichen Mehraufwand, da auch bei kleinen Änderungen das komplette Replikat neu übertragen werden muss. Außerdem bedingt diese Form der Zustandsfortschreibung, dass der Zugriff auf ein Replikat mittels eines Stellvertreterobjektes geschützt werden muss, um Referenzen auf veraltete Kopien auszuschließen. Diese Stellvertreterschicht bremst lokale (Lese-)Zugriffe durch das unumgängliche Kopieren von Argumenten und Rückgabewerten unverhältnismäßig stark aus.

Die vorliegende Arbeit ermöglicht deshalb einen direkten Zugriff auf replizierte Objekte, Graphen und Felder, indem eine inkrementelle Zustandsfortschreibung eingesetzt wird. Änderungen an entfernten Replikaten werden während der Zustandsfortschreibung direkt in die veralteten Replikate eingespielt. Dadurch kann die Anwendung mit derselben Replikatreferenz weiterarbeiten. Ein Umschalten auf ein über Kopieren erzeugtes frisches Replikat ist nicht notwendig, was die Stellvertreterschicht für replizierte Objekte überflüssig macht. Darüber hinaus werden nur die tatsächlich geänderten Teile eines replizierten Objektes während der Zustandsfortschreibung neu übertragen, so dass Übertragungsbandbreite einspart wird.

Konsistenz mittels virtuell gemeinsamen Speichers

Eine ganze Reihe verteilter virtueller Java-Maschinen setzen auf einem System für virtuell gemeinsamen Speicher auf. Eine gute Übersicht über herkömmliche DSM-Systeme gibt [89]. Arbeiten, die nun besprochen werden, realisieren, darauf aufbauend, eine verteilte Ausführungsplattform für portablen Code. Die Systeme selbst sind dagegen nicht portabel, da sie auf einer Neuimplementierung oder Modifikation der virtuellen Java-Maschine beruhen. Dafür versprechen sie die Ausführung eines regulären parallelen Java-Programms ohne jegliche Modifikation. Um dies zu gewährleisten, muss die Java-Semantik in der verteilten Umgebung exakt nachgebildet werden.

Das Speichermodell der virtuellen Java-Maschine ist so spezifiziert, dass es sich auf vielen symmetrischen Multiprozessoren effizient implementieren lässt. Die Umsetzung für verteilten Speicher ist deutlich schwieriger. Viele der Arbeiten in dieser Kategorie realisieren daher auch keine virtuelle Java-Maschine im engeren Sinne, in der sich unmodifizierte parallele Applikationen effizient ausführen lassen, sondern erweitern die Sprache und verlangen vom Programmierer zusätzliche Annotationen.

Java/DSM [108] und JESSICA2 [109] verwenden das seitenbasierte DSM-System TreadMarks [51], um eine verteilte virtuelle Maschine über einen virtuell gemeinsamen Hautspeicher zu koppeln. Die Verwendung eines seitenbasierten DSM-Systems für eine objektorientierte Sprache hat das bekannte Problem, dass viele kleine Objekte auf einer Seite angelegt werden. Bei Benutzung dieser Objekte in unterschiedlichen Kontexten kommt es zu ständigem Hin- und Herkopieren der Seite.

Hyperion [69] vermeidet ungünstige Aufteilung der Objekte auf Seiten durch Einsatz eines objektbasierten DSM-Systems. Aber auch dies kann Konflikte beim Zugriffe nicht ausschließen, da in Java nebenläufige Modifikationen unterschiedlicher Teile eines Objektes in mehreren Aktivitäten konfliktfrei stattfinden können.

CoJVM [64] reduziert die Größe der kleinsten Verteilungseinheit auf ein 32-Bit Wort, da in Java atomare Operationen auf Datentypen dieser Größe beschränkt sind. Damit sind konkurrierende Zugriffe auf ein Objekt konfliktfrei möglich, wenn die Zugriffe unterschiedliche Instanzvariablen des Objektes betreffen. Allerdings steigt durch die feinere Granularität der Aufwand für die Verwaltung des virtuell gemeinsamen Speichers.

Das DSM-System in MultiJav [14] geht einen anderen Weg. In MultiJav wird durch das Subsystem für virtuell gemeinsamen Speicher das Java-Speichermodell simuliert. Dort ist festgelegt, wann ein Kontrollfaden spätestens die Änderungen sehen muss, die in einem anderen Kontrollfaden durchgeführt wurden. Das Java-Speichermodell geht von einem lokalen Zwischenspeicher pro Kontrollfaden aus, in dem dieser alle seine Änderungen durchführt. Der Zwischenspeicher wird bei Betreten eines synchronisierten Blocks geleert und vor dem Verlassen in den Hauptspeicher zurückgeschrieben. MultiJav erstellt daher vor dem Zugriff auf ein Objekt eine lokale Kopie und führt alle Modifikationen auf dieser Kopie aus. Beim Zurückschreiben der Änderungen werden genau die geänderten Datenelemente durch Differenzenbildung ermittelt.

Jackal [101] verwendet eine statische Programmanalyse, um Hinweise zu extrahieren, die es dem DSM-System erlauben, das Leeren des lokalen Zwischenspeichers, wenn immer möglich, zu unterlassen.

Die virtuelle Maschine cJVM [4] verwendet eine Mischstrategie zwischen Replikation und Fernzugriff. Ebenfalls durch eine Programmanalyse, in die auch Laufzeitinformationen einfließen, werden Objekte klassifiziert, für die sich entweder Replikation lohnt, weil auf sie oft verteilt lesend zugegriffen wird, oder für die Fernzugriff besser ist, weil sich ihr Zustand zu oft ändert und daher Replikation zu aufwendig ist.

Im Gegensatz zu den in diesem Abschnitt besprochenen Ansätzen ist die vorliegende Arbeit plattformunabhängig in reinem Java realisiert. Damit ist kein vollständig transparenter DSM-Ansatz möglich, da hierfür die Speicherverwaltung des zugrundeliegenden Prozessors verwendet werden müsste. Das ist nicht plattformunabhängig möglich. Vollkommene Verteilungstransparenz, wie sie durch die implizite Replikation in einem DSM-System möglich ist, wird aber in der vorliegenden Arbeit gar nicht an-

gestrebt, da die explizite Replikation und damit die Verteilung für datenparallele Operationen ausgenutzt werden kann. Durch diese explizite Replikation wird eine nahtlose Verbindung von Kontroll- und Datenparallelität erreicht.

2.6.2 Verbindung von Kontroll- und Datenparallelität

Ein zentraler Bestandteil der vorliegenden Arbeit ist die Verbindung von Kontroll- und Datenparallelität über eine neue Form von Replikation. Die Idee, das kontroll- und datenparallele Modell in einer Umgebung zu integrieren, ist nicht neu [7]. Bisherige Ansätze beschränken den datenparallelen Aspekt aber auf den regulären Fall mehrdimensionaler Felder, während hier eine Erweiterung auf beliebige irreguläre Strukturen, wie z. B. auf Graphen von Objekten, vorgestellt wird.

Braid [105] ist eine datenparallele Erweiterung von Mentat [32]. Eine datenparallele Klasse in Braid ist implizit ein Aggregat von Objekten der Klasse. Das Aggregat, also nicht das einzelne Objekt, ist über die Knoten der Umgebung verteilt gespeichert. Die Struktur eines Aggregates ist beschränkt auf mehrdimensionale Felder. Eine Methode eines „datenparallelen Objektes“ wird entweder auf einem einzelnen Element, auf allen Elementen oder auf einer regulären Teilmenge aller Elemente (einer Zeile oder Spalte der Aggregat-Matrix) ausgeführt. Datenparallelität ist folglich nur auf Feldern, nicht auf Objekten und erst recht nicht auf beliebigen Graphen von Objekten möglich. Replikation kommt nur am Rande ins Spiel, wenn datenparallele Methoden mit anderen Aggregaten parametrisiert werden. In einem solchen Fall kann die Verteilung der als Argument übergebenen Struktur so gewählt werden, dass ein Element des Argument-Aggregates während der Operation temporär bei verschiedenen Elementen des Ziel-Aggregates lokal vorrätig gehalten wird. Während einer datenparallelen Operation ist dann der Zustand des Argument-Aggregates repliziert und in den Teiloperationen auf den Elementen des Ziel-Aggregates lokal verwendbar. Die vorliegende Arbeit ermöglicht hingegen datenparallele Operationen auf allgemeinen Strukturen von Objekten.

Orca [36] bietet gemeinsam benutzbare Objekte an, um Datenparallelität auszudrücken. Die Sichtweise ist leicht anders als in Braid, die erreichbaren Effekte sind aber nahezu identisch. In Orca ist ein datenparalleles Objekt mit einer Verteilungsannotation versehen. Diese Verteilungsannotation bezieht sich auf jede einzelne Instanzvariable des Objektes in gleicher Weise und spezifiziert für sie eine Dimensionalität. Damit besteht ein datenparalleles Objekt aus Feldern gleicher Dimensionalität. In Orca ist also ein datenparalleles Objekt ein Objekt aus Feldern, während in Braid ein datenparalleles Aggregat ein Feld aus gleichartigen Objekten ist. Diese Sichtweisen sind gegeneinander austauschbar. Ein gemeinsam benutztes Objekt in Orca ist über die Knoten der Umgebung verteilt. Dabei kommen jeweils korrespondierende Partitionen seiner Instanzvariablen-Felder lokal auf einem Knoten zu liegen. Wird während einer datenparallelen Operation auf dem Objekt auf einen Feldeintrag einer fremden Partition zugegriffen, wird die gesamte Partition auf den lokalen Knoten kopiert und dort für weitere Zugriffe zwischengespeichert. In der vorliegenden Arbeit können die auf einem Knoten benötigten Bereiche im vorhinein spezifiziert werden, so dass die Daten bereits vorliegen, wenn Zugriffe stattfinden.

Sowohl in Braid als auch in Orca werden Teilfelder temporär auf einen anderen

Knoten kopiert. Gruppenoperationen werden für datenparallele Operationen benutzt, haben aber mit der Replikation oder der Konsistenzerhaltung des replizierten Zustandes nichts zu tun. Der temporär zwischengespeicherte Zustand wird nicht konsistent gehalten, sondern nach jeder Operation verworfen. Von Replikation kann man nur insoweit sprechen, als dass während einer datenparallelen Operation Teile von Feldern auf möglicherweise mehreren Knoten der verteilten Umgebung zwischengespeichert und daher lokal vorrätig gehalten werden. Diese Zwischenspeicherung geschieht, um durch einmaligen Transfer eines größeren Blocks Zugriffe zu bündeln und Latenz zu verdecken. Die Lokalitätsoptimierung durch verteilt parallelen Lesezugriff ist auf eine einzelne datenparallele Operation beschränkt. Das Kreuz in Tabelle 2.1 bei Lokalität ist daher bei beiden Sprachen eingeklammert.

HPJava [61] ist ein Java-Dialekt, der verteilte mehrdimensionale Felder unterstützt. Anders als in Braid und Orca werden diese Felder aber nicht als Objekte „getarnt“. Verteilte mehrdimensionale Felder werden über eine eigene Notation deklariert und mit einer Verteilungsannotation versehen. Diese Verteilung *partitioniert* das Feld aber nicht notwendigerweise auf die Knoten, sondern die Feldbereiche können möglicherweise *überlappen*. Ein so „verteiltes“ Feld kann man als *partiell repliziert* ansehen. Bei partieller Replikation sind nicht alle Teile der replizierten Struktur gleichermaßen überall verfügbar, sondern manche Teile sind nur auf einem Knoten angelegt (also verteilt), andere Teile sind auf *manchen* Knoten gleichzeitig verfügbar (also repliziert). In HPJava steht allerdings nur eine vordefinierte Auswahl an möglichen Verteilungsmustern zur Verfügung, die geeignet parametrisiert werden können. Bei einer blockweisen Verteilung kann so beispielsweise die Blockgröße und die Breite des Überlappungsbereichs festgelegt werden. Die vorliegende Arbeit erweitert die partielle Replikation von Feldern auf allgemeine Graphen von Objekten und macht so datenparallele Operationen auch auf solchen Strukturen möglich.

In HPJava hat jedes Feldelement einen Heimatknoten; nur dort darf es modifiziert werden. Zugriffe auf Replikate von Feldelementen dürfen ausschließlich lesend stattfinden. Unter diesen Randbedingungen kann das auf den Knoten der Umgebung partiell replizierte Feld datenparallel bearbeitet werden. Da keine Unterklassenbildung von verteilten Feldern erlaubt ist, gibt es in HPJava keine speziellen Methoden, die auf allen Feldpartitionen wie in Braid und Orca parallel ausgeführt werden. Um das auszugleichen, führt HPJava ein `overall`-Konstrukt ein, das die Aktivität zu jeder Partition eines verteilten Feldes verzweigt. Diese Form der Aktivitätserzeugung passt allerdings nicht gut zum ansonsten kontrollparallelen Java-Modell. Alle Zugriffe auf ein verteiltes Feld finden ausschließlich lokal statt (das Feldelement muss dazu auf dem Knoten beheimatet sein oder ein Replikat muss existieren). Durch Aufruf einer speziellen Methode werden die (möglicherweise) veränderten Feldelemente auf andere Knoten kopiert, wenn sie dorthin repliziert sind. In der vorliegenden Arbeit sind alle Replikate eines replizierten Objektes gleichberechtigt. Ein Replikat darf in einer datenparallelen Operation an jeder beliebigen Stelle modifiziert werden. Voraussetzung ist entweder, dass keine zwei Replikate gleichzeitig an derselben Stelle verändert werden, oder dass der Benutzer eine Operation spezifiziert, mit der die nebenläufigen Änderungen wieder zu einem konsistenten Zustand verschmolzen werden können.

ProActive [5] führt Gruppen aktiver Objekte ein. Ein Aufruf an eine Objektgruppe wird parallel an jedem Gruppenmitglied durchgeführt. Die Argumente eines Aufrufs

werden dabei entweder an alle Gruppenmitglieder übermittelt oder unter den Gruppenmitgliedern aufgeteilt. Der Aufruf entspricht somit in MPI einem Rundruf oder einer Scatter-Operation. Es wird kein konsistenter Zustand der Gruppenmitglieder angestrebt, da der Ansatz ausschließlich auf Datenparallelität abzielt. Allerdings ist es einem Gruppenmitglied nicht erlaubt, auf andere Mitglieder zuzugreifen. Die Funktionalität bleibt daher auf einfache Operationen beschränkt. Replikation findet nicht statt. Die vorliegende Arbeit ermöglicht über (partielle) Replikation datenparallele Operationen so zu formulieren, als ob an *einer* großen Struktur parallel gearbeitet wird, anstatt explizit Daten zu den parallelen Aktivitäten hinzuschicken und wieder zurückzuerhalten. Der Datenaustausch geschieht implizit während der Zustandsfortschreibung. Darüberhinaus ist es möglich, die von MPI her bekannten kollektiven Operationen für den reinen Datenaustausch durchzuführen, wobei ein repliziertes Objekt quasi als Kommunikator benutzt wird.

Als ein Alleinstellungsmerkmal dieser Arbeit kann festgehalten werden, dass über partielle Replikation von allgemeinen Strukturen eine nahtlose Verbindung von Kontroll- und Datenparallelität in einer objektorientierten Umgebung realisiert wird.

2.6.3 Replikation für Fehlertoleranz

Mocha [97] ist eine Java-Bibliothek für fehlertolerante verteilte Objekte. Um Robustheit gegenüber ausfallenden Rechenknoten zu erreichen, benutzt Mocha Replikation von Objekten. Der Zugriff auf ein verteiltes Objekt wird über eine ausgezeichnete Methode angefordert. Dies bewirkt, dass das gesamte Objekt auf den zugreifenden Knoten transportiert wird. Wurde Schreibzugriff angefordert, darf das Objekt lokal modifiziert werden. Nach der anschließenden Freigabe wird sein gesamter Zustand an so viele Knoten verschickt, wie beim Objekt als Replikationsgrad eingestellt ist. Bei einem Knotenausfall wird die Funktionalität von den überlebenden Replikaten übernommen, so dass der Ausfall von einigen Knoten tolerabel wird.

Bei Mocha geht es nicht um die Unterstützung paralleler Programme auf Rechnerbündeln, sondern um Anwendungen in Weitverkehrsnetzen. Die Effizienz der eingesetzten Replikationsform spielt daher eine untergeordnete Rolle. Replikation in der vorliegenden Arbeit kann nicht zum Erreichen von Fehlertoleranz eingesetzt werden. In einer Bündelumgebung scheint eine Strategie besser, bei der Sicherungspunkte während des Programmlaufs angelegt werden. Bei einem Knotenausfall kann der Sicherungspunkt auf einem anderen Knoten eingespielt werden, um die Anwendung fortzusetzen. Ein Mechanismus für Sicherungspunkte ist allerdings nicht Gegenstand der vorliegenden Arbeit.

Kapitel 3

Konzepte der Programmierumgebung

Dieses Kapitel beschreibt die Konzepte der Programmierumgebung und begründet getroffene Architekturentscheidungen. Nach Auswahl von Basissprache und zugrundeliegender virtueller Maschine werden die verteilungsrelevanten Konzepte diskutiert. Den Abschluss diese Kapitels bildet die Integration dieser Konzepte in den Lösungsvorschlag für die Sprache und die Laufzeitumgebung der Programmierumgebung. Die Umsetzung der dabei identifizierten Teilprobleme wird in den Folgekapiteln besprochen.

3.1 Basissprache und virtuelle Maschine

Java erfüllt alle Voraussetzungen, um als Basis für eine Bündelprogrammierumgebung zu dienen. Die Sprache hat ein einfaches Objektmodell, das saubere, objektorientierte Entwürfe fördert. Gleichzeitig ist Unterstützung für Nebenläufigkeit und automatische Speicherbereinigung in die Sprache integriert. Die Programmiersprache ist konzipiert für die Übersetzung in einen maschinenunabhängigen Zwischencode (Bytecode) und die Ausführung dessen auf einer virtuellen Maschine. Die virtuelle Maschine abstrahiert von der zugrundeliegenden Hardware und dem verwendeten Betriebssystem. Sie bietet der Anwendung eine auf allen Architekturen gleiche Schnittstelle an und bildet diese auf die zugrundeliegende Ausführungsumgebung ab. Durch diese erzwungene strikte Kapselung von plattformabhängigen Teilen in der virtuellen Maschine erreicht man optimale Portabilität von Applikationen. Eine Anwendung ist ohne Änderungen auf jeder virtuellen Maschine lauffähig. Damit eine Anwendung auf unterschiedlichen Kombinationen aus Hardware und Betriebssystem tatsächlich ausgeführt werden kann, setzt dies die Verfügbarkeit von Implementierungen der virtuellen Maschine auf diesen Plattformen voraus. Gerade diese Verfügbarkeit von Implementierungen der virtuellen Maschine auf vielen für Hochleistungsrechnen relevanten Plattformen ist bei Java besonders gut. Erfolgreiche Forschung hat eine rapide Weiterentwicklung der Laufzeitübersetzer möglich gemacht, so dass die Ausführung von portablem Bytecode mittlerweile die gleiche Geschwindigkeit erreicht wie konventionell übersetzter Code [11, 90]. Unterschiede werden auch immer vernachlässigbarer in Anbetracht der Tatsache, dass die Programmeffizienz stärker durch den Programmierer beeinflusst zu sein scheint als durch das Laufzeitsystem der Programmiersprache [88].

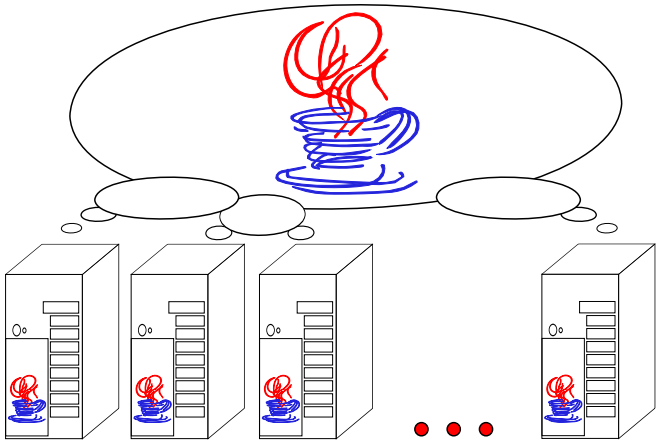


Abbildung 3.1: Verteilte virtuelle Maschine, zusammengesetzt aus kooperierenden virtuellen Maschinen in einem Rechnerbündel.

Aus den genannten Gründen wurde Java als Basis für die Bündelprogrammierumgebung ausgewählt. Prinzipiell wäre eine Realisierung aber auch auf Grundlage einer anderen modernen, objektorientierten Sprache mit Parallelitätsunterstützung wie beispielsweise C# möglich. Allerdings gibt es hierfür noch nicht genügend Implementierungen der virtuellen Maschine, um die Portabilität auf viele relevante Kombinationen aus Hardware und Betriebssystem für Bündelrechner sicherzustellen.

Die Wahl von Java stellt eine besondere Herausforderung dar, weil die zugehörige virtuelle Maschine für eine sichere Programmausführung auch von nicht-vertrauenswürdigem Code ausgelegt ist. Um dies zu erreichen, sind die Zugriffe des Programms auf seine Laufzeitumgebung streng reglementiert. Diese Beschränkungen werden vor dem Ausführen eines Programms durch die virtuelle Maschine überprüft. So ist es einem Java-Programm beispielsweise nicht möglich, auf Bereiche seines Stapelspeichers zuzugreifen, die außerhalb des aktuellen Rahmens liegen. Ebenso sind Zugriffe nur auf solche Objekte des Hauptspeichers möglich, auf welche das Programm eine explizite Referenz besitzt. Alle Referenzen sind typisiert und benennen immer ein ganzes Objekt. Innerhalb eines Objektes kann nur auf seine Instanzvariablen zugegriffen werden. Es gibt keine Zeiger auf Teile eines Objektes.

Die virtuelle Java-Maschine ist aufgrund der Unterstützung von Parallelität, Koordinationsprimitiven und ihrem Speichermodell auf parallele Zielarchitekturen vorbereitet, sie bietet jedoch keine Mechanismen an, um mit Verteilung umzugehen. In dieser Arbeit wird die virtuelle Java-Maschine als Baustein für eine verteilte Umgebung eingesetzt. Dafür wird Java um Konzepte für den Umgang mit Verteilung erweitert. Die Spracherweiterung ist so gemacht, dass sie sich durch einen Transformationsschritt von einem Präprozessor zurück auf Java abbilden lässt. Damit bleibt die virtuelle Java-Maschine unverändert einsetzbar. Auf den Knoten eines Rechnerbündels, die ihrerseits auch aus SMP-Maschinen bestehen können, abstrahiert die virtuelle Java-Maschine, wie in Abbildung 3.1 gezeigt, von der im Bündel eingesetzten Hardware. Die darauf aufbauende verteilte Programmierumgebung macht so die Verteilung beherrschbar und nutzbar für die Anwendung.

3.2 Bündellaufzeitsystem

Die Zielarchitektur der Programmierumgebung für Rechnerbündel ist eine verteilte virtuelle Java-Maschine, wie sie im folgenden definiert wird. Die verteilte virtuelle Maschine besteht aus regulären zusammengeschalteten virtuellen Java-Maschinen. Jeder Knoten des Rechnerbündels führt dabei eine solche virtuelle Java-Maschine aus. Die Zusammenschaltung erfolgt über ein Laufzeitsystem, welches die Knoten des Rechnerbündels für die Anwendung wie eine große virtuelle Maschine erscheinen lässt. Jede teilnehmende virtuelle Maschine bringt ihren lokalen Hauptspeicher in einen gemeinsamen Objektraum der verteilten virtuellen Maschine ein.

Eine Zusammenschaltung von einzelnen virtuellen Maschinen über ein Laufzeitsystem zu einer Bündelmaschine erfüllt die Portabilitäts- und Modularitätsanforderung aus Abschnitt 1.3, sofern hierfür unveränderte virtuelle Java-Maschinen eingesetzt werden. Die virtuellen Maschinen auf den Bündelknoten abstrahieren von der Hardware des zugrundeliegenden Rechners und erlauben damit sogar eine Zusammenschaltung von heterogenen Bündeln. Die Implementierung der virtuellen Java-Maschine auf den Bündelknoten kann unabhängig vom verteilten Laufzeitsystem gewählt werden. Damit kann z. B. diejenige virtuelle Maschine mit dem jeweils besten Laufzeitübersetzer eingesetzt werden.

3.3 Parallelitätsmodelle

Java hat Unterstützung für Kontrollparallelität in die Sprache eingebaut. Für viele Hochleistungsanwendungen, die mit großen Datenmengen umgehen, ist daneben Datenparallelität wichtig. Die verteilte Umgebung als Erweiterung einer zentralisierten virtuellen Maschine übernimmt die Kontrollparallelität des Java-Modells und bildet sie auf die verteilte Umgebung ab. Zusätzlich wird in das Programmiermodell für die verteilte Umgebung Datenparallelität so integriert, dass eine nahtlose Verbindung zwischen den Konzepten Kontroll- und Datenparallelität geschaffen wird.

3.3.1 Kontrollparallelität

Ein kontrollparalleles Programm erzeugt mehrere nebenläufige Aktivitätsstränge (engl. *threads*), die bis auf Synchronisationspunkte unabhängig voneinander parallel Aufgaben erledigen. Je weniger Abhängigkeiten die zu erledigenden Aufgaben untereinander aufweisen, desto einfacher ist es, ein solches kontrollparalleles Programm zu formulieren. Im Idealfall arbeitet jede Aktivität unabhängig ein eigenes Teilprogramm mit eigenen Daten ab. Aus diesem Grund wird der kontrollparallele Programmierstil auch *multiple instruction, multiple data* oder kurz MIMD [37] genannt.

Kontrollparallelität in Java

Java unterstützt Kontrollparallelität durch die Bereitstellung von Klassen für die Erzeugung nebenläufiger Aktivitäten in der Standardbibliothek und hat Sprachkonstrukte für Koordination und wechselseitige Benachrichtigung von Aktivitäten eingebaut.

Nach ihrer Erzeugung existiert eine Aktivität unabhängig von den Objekten der Anwendung. Die Aktivität führt die Anweisungen in den zu Objekten und Klassen gehörigen Methoden aus und hinterlässt eine Spur von Zustandsänderungen. Die Aktivität ist nicht an ein Objekt gebunden, sondern wechselt bei einem Methodenaufruf zum Zielobjekt. Demzufolge können durchaus mehrere Aktivitäten gleichzeitig in einem Objekt aktiv sein. Der Programmierer muss durch explizite Koordination Konflikte verhindern, die durch nebenläufige Zustandsänderungen an einem oder einer Gruppe von zusammenhängenden Objekten entstehen könnten. Koordination geschieht in Java durch Definition kritischer Abschnitte, die immer nur von genau einer Aktivität betreten werden können. Darüber hinaus können sich Aktivitäten über Zustandsveränderungen gegenseitig benachrichtigen.

Andere Ausprägungen von Kontrollparallelität

In anderen Sprachen werden andere Ausprägungen von Kontrollparallelität und zugehörigen Koordinationsmechanismen verwendet. Beispielsweise kann die Aktivität an ein Objekt gekoppelt sein. Alle an das Objekt „gesendeten“ Methodenaufrufe werden dann von dieser einen Aktivität nacheinander abgearbeitet. Die Ausprägungen von Kontrollparallelität unterscheiden sich hinsichtlich des erreichbaren Parallelitätsgrades und der Mächtigkeit bzw. Fehleranfälligkeit der Koordinationsmechanismen. Philippsen untersucht in [81] 111 parallele objektorientierte Sprachen hinsichtlich unterstützter Parallelitäts- und Koordinationsformen und wiegt Vor- und Nachteile gegeneinander ab. Ziel der hier vorliegenden Arbeit ist es nicht, die „beste“ dieser Parallelitätsformen zu suchen oder eine noch bessere zu erfinden, sondern ein weithin verwendetes Konzept in einer Bündelumgebung möglichst unverändert und effizient nutzbar zu machen. Diese Forderung triff auf das kontrollparallele Modell von Java zu. Dieses Modell wird auch in anderen modernen objektorientierten Sprachen verwendet. Auch C# erlaubt genau dieselbe Art von Aktivitätserzeugung und Koordination.

Kontrollparallelität in der verteilten Umgebung

Die verteilte Umgebung nimmt sich eine zentralisierte virtuelle Maschine zum Vorbild und bildet diese Sicht auf kooperierende virtuelle Maschinen im Rechnerbündel ab. In einem nichtverteilten Programm springt eine Aktivität beim Methodenaufruf zum Zielobjekt, arbeitet die aufgerufene Methode ab und kehrt danach zur aufrufenden Methode zurück. In der verteilten Umgebung sind die Objekte der Anwendung über die Knoten des Rechnerbündels verteilt. Ruft hier ein Objekt die Methode eines anderen Objektes auf, so wechselt die ausführende Aktivität nicht nur das Objekt, sondern möglicherweise auch den Rechenknoten. Stellt man sich vor, die Aktivität hinterließe eine Spur im ausgeführten Code, kann man sich die Metapher „Kontrollfaden“ für Aktivität bildlich vorstellen. Im verteilten Fall überspannt dieser Kontrollfaden dann Rechnergrenzen – es handelt sich um einen maschinenüberspannenden Kontrollfaden.

3.3.2 Datenparallelität

In einem datenparallelen Programm ist die ausführende Aktivität nicht notwendigerweise explizit repräsentiert. Im Mittelpunkt stehen Kollektionen von Daten, auf denen Berechnungen auszuführen sind. Solange eine Berechnung auf jedem Datenelement unabhängig von allen anderen Datenelementen ausgeführt werden kann, ist es möglich, diese parallel auf allen Datenelementen gleichzeitig durchzuführen. Eine Sprache unterstützt Datenparallelität beispielsweise, indem sie Konstrukte zur Verfügung stellt, die anzeigen, dass eine Berechnung auf einer Kollektion von Daten parallel ausgeführt werden kann. Das prominenteste Beispiel für ein solches datenparalleles Konstrukt ist das `forall` in HPF [24].

Die vorliegende Arbeit geht einen anderen Weg, indem sie das bulk-synchrone parallele Modell (BSP [98]) in einen objektorientierten Kontext überträgt und so datenparallele Berechnungen an allgemeinen Objektstrukturen ermöglicht. Dazu wird keine spezielle Form der Aktivitätserzeugung eingeführt, sondern über eine neue Form von Replikation die Möglichkeit geschaffen, dass sich reguläre kontrollparallele Aktivitäten temporär zu einem Kollektiv zusammenschließen, um eine datenparallele Berechnung an dem replizierten Zustand durchzuführen. Die neue Replikationsform unterstützt die parallele Modifikation des replizierten Zustandes zum Zwecke einer datenparallelen Berechnung und das anschließende Zusammenmischen der nebenläufigen Änderungen zu einer konsistenten Sicht. Da die beteiligten Aktivitäten dabei gemeinsam kooperativ vorgehen müssen, wird die neue Replikationsform *kollektive Replikation* genannt. Die folgenden Abschnitte motivieren den Zusamenhang zwischen datenparallelen Operationen und Replikation als geeignetes Mittel, um diese Parallelitätsform in einer objektorientierten Sprache auszudrücken.

Datenparallelität und Objektorientierung

Ein objektorientiertes Äquivalent für ein `forall`-Konstrukt ist der Methodenaufruf an einem Objektaggregat. Beim Aufruf einer Methode an einem Aggregat wird dieselbe Methode auf jedem Teil des Aggregates gleichzeitig parallel ausgeführt. Dabei geschieht die Erzeugung der Aktivitäten und deren Synchronisation implizit. Der Aufruf an dem Aggregat wird an jedem Mitglied des Aggregates konzeptuell in einer eigenen Aktivität durchgeführt. Der Aufruf kehrt dann zurück, wenn alle Teilaufrufe zurückgekehrt sind. Java bietet keine Unterstützung für Datenparallelität. Einige der in Abschnitt 2.6.2 diskutierten Arbeiten führen Aggregate in Java ein.

Eine datenparallele Operation lässt sich durch das Muster „Meister-Arbeiter“ kontrollparallel umformulieren. Dabei teilt der Meister die datenparallele Aufgabe in Unteraufgaben und übergibt diese an bereitstehende Arbeiter. Die Arbeiter führen ihre Teilaufgabe unabhängig von allen anderen Arbeitern aus und liefern das Ergebnis wieder beim Meister ab. Dieser konstruiert aus den Teilergebnissen die Gesamtlösung. Meister und Arbeiter sind eigenständige Aktivitäten, die nur während der Auftragsvergabe und der Abgabe der Teilergebnisse miteinander kommunizieren. In [80] wird eine Transformation vorgestellt, die ein datenparalleles `forall`-Konstrukt automatisch in ein kontrollparalleles Programm nach dem Meister-Arbeiter-Muster abbildet.

Objektaggregate und `forall`-Konstrukte fügen sich nicht nahtlos in eine ansonsten kontrollparallele Umgebung ein. Beide Ausdrucksformen führen implizit eine wei-

tere Form der Aktivitätserzeugung ein. Eine so erzeugte Aktivität ist aber nicht gleichwertig mit den übrigen von der Applikation explizit erzeugten Aktivitäten. Ihre Lebensdauer ist an einen syntaktischen Block oder eine Methode gebunden. Sie hat keine eigene Identität und darf daher nicht mit anderen Aktivitäten kommunizieren.

Das „bulk-synchrone“ parallele Modell

Das BSP-Modell ist ein Muster für datenparallele Programme. Dieses Musters setzt einfache und fehlerunempfindliche Synchronisationsmechanismen ein. Ein Programm im BSP-Modell gliedert sich in eine sequentielle Folge von Hauptschritten. Jeder dieser Hauptschritte führt parallele Berechnungen auf einer Kollektion von Daten aus. Alle diese Berechnungen können unabhängig voneinander ausgeführt werden. Nach jedem Hauptschritt findet ein Datenaustausch statt. Dieser Datenaustausch gewährleistet, dass die Berechnungen des nächsten Hauptschritts wieder unabhängig von einander erfolgen können. In einem BSP-Programm gibt es also keine Datenabhängigkeiten zwischen den parallelen Berechnungen eines Hauptschrittes. Datenabhängigkeiten zwischen zwei aufeinanderfolgenden Hauptschritten werden auf eine Kommunikationsoperation abgebildet.

Die Koordination der Aktivitäten, welche die parallelen Berechnungen durchführen, ist in BSP besonders einfach, weil Interaktionen während der Berechnung ausgeschlossen sind. Durch den anschließenden Datenaustausch findet automatisch eine Koordination statt, indem jede Aktivität diejenigen Daten bereitstellt, die andere Aktivitäten für den nächsten Schritt benötigen, und selbst solange wartet, bis entsprechende Daten für sie eingetroffen sind. In einer verteilten Umgebung ist das Abwechseln von Berechnung und Kommunikation besonders vorteilhaft, weil der Datenaustausch zwischen den Berechnungen am Stück erfolgen kann. Durch diese Bündelung wird die Netzwerklatenz verdeckt, und durch eine Blockung der übertragenen Daten kann die Netzwerkbandbreite bestmöglich ausgenutzt werden.

Das folgende Beispiel zeigt einen typischen BSP-Algorithmus: Um die statische Temperaturverteilung unter gegebenen Randbedingungen zu bestimmen, wird die Lösung der Laplace-Differentialgleichung diskret auf einem Gitter angenähert. Für eine gegebene Startbelegung A^0 eines Gitters $A = (a_{i,j})$ muss die Rechenvorschrift 3.1 für $n \in [1, \dots]$ solange iteriert werden, bis sich ein stabiler Zustand einstellt bzw. die Änderungen einen Schwellwert ϵ nicht mehr übersteigen.

$$\begin{aligned} A^{n+1} &= (a_{i,j}^{n+1}) \\ \text{mit } a_{i,j}^{n+1} &= \frac{1}{4}(a_{i+1,j}^{n} + a_{i-1,j}^{n} + a_{i,j+1}^{n} + a_{i,j-1}^{n}) \end{aligned} \tag{3.1}$$

Für jedes i und j ist die Rechenoperation 3.1 unabhängig von jeder anderen, da aus dem alten Zustand A^n der neue Zustand A^{n+1} berechnet wird. Verteilt man das Gitter wie in Abbildung 3.2 zeilenblockweise auf zwei Prozessoren P_0 und P_1, kann jeder Prozessor unabhängig arbeiten, wenn er zu Beginn seinen Teil der Datenelemente von A^n zuzüglich eines Randes der Breite eins erhält. In Abbildung 3.2 sind diejenigen Datenelemente, welche Prozessor P_0 für einen Hauptschritt benötigt von links unten nach rechts oben schraffiert. Entsprechend sind die Elemente für Prozessor P_1

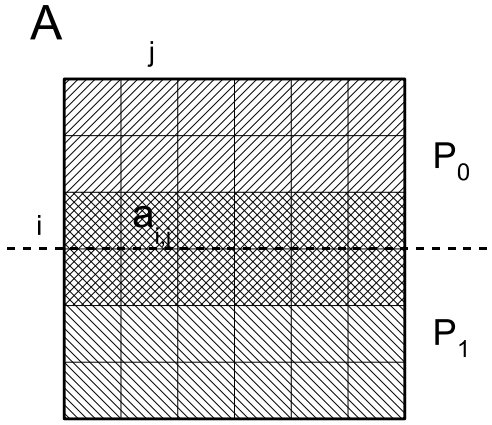


Abbildung 3.2: Datenstruktur für einen verteilt parallelen Algorithmus zur diskreten Näherungslösung der zweidimensionalen Laplace-Differentialgleichung.

von links oben nach rechts unten schraffiert. Am Ende eines Hauptschrittes muss der Überlappungsbereich zwischen den beiden Prozessoren ausgetauscht werden. Dieser Datenaustausch bewirkt, dass beide Prozessoren Zugriff auf denjenigen Teil der Daten von A^{n+1} haben, den sie benötigen, um ihren Teil von A^{n+2} zu berechnen usw.

Wechselt man die Sichtweise, stellt sich der Überlappungsbereich als gemeinsamer Zustand heraus, der auf beiden Prozessoren P_0 und P_1 also repliziert gespeichert werden muss. Da jeweils einer der beiden Prozessoren einen Teil dieses Zustandes modifiziert, muss nach der kollektiven Operation die Konsistenz dieser replizierten Bereiche wiederhergestellt werden. Betrachtet man den bearbeiteten Zustand als repliziertes Objekt, kann die im BSP-Modell an den Berechnungsschritt anschließende Kommunikationsoperation als Konsistenzoperation interpretiert werden.

Die vorliegende Arbeit überträgt das BSP-Modell in einen objektorientierten Kontext und ermöglicht so datenparallele Berechnungen an allgemeinen Objektstrukturen.

Datenparallelität in der verteilten Umgebung

Die vorliegende Arbeit führt keine alternativen Formen der Aktivitätserzeugung wie bei Objektaggregaten ein, da dies zu Konflikten mit dem kontrollparallelen Modell führt. Stattdessen dehnt sie den BSP-Ansatz auf beliebige irreguläre Kollektionen von Daten aus und integriert die Unterstützung dafür in die Sprache. In herkömmlichen Ansätzen lässt sich das BSP-Modell nur auf reguläre Kollektionen wie mehrdimensionale Felder anwenden, weil der explizite Datenaustausch zur Konsistenzerhaltung der Replikate am Ende eines jeden Hauptschrittes eine einfache Umrechnung von Sender- in Empfängeradressen voraussetzt. Am Beispiel aus Abbildung 3.2 heißt das, dass während des Datenaustausches die dritte Zeile der Matrix A von Prozessor P_0 auf Prozessor P_1 und die vierte Zeile in umgekehrter Richtung übertragen werden muss. Dieser Datenaustausch auf einer regulären Struktur lässt sich als Aufruf einer Kommunikationsbibliothek für Nachrichtenaustausch ausdrücken.

In einem objektorientierten Programm finden Berechnungen aber oft an irregulären Strukturen wie Graphen von Objekten statt. Möchte man eine solche irreguläre Struktur in einer datenparallelen Operation bearbeiten, lässt sich der Datenaustausch

am Ende eines Hauptschrittes nicht mehr statisch beschreiben. Insbesondere wenn sich der Graph durch Neueinfügen von Objekten und Austragen von alten Objekten dynamisch ändert, dann versagen alle statischen Ausdrucksmittel, um den Datenaustausch zu spezifizieren. Die Verallgemeinerung von datenparallelen Operationen auf irregulären Kollektionen wird über eine neue Ausdrucksform in der Sprache für die verteilte Programmierumgebung möglich gemacht. Ein *kollektiv* repliziertes Objekt kapselt eine beliebige Datenstruktur, die datenparallel nach dem BSP-Modell bearbeitet werden kann. Eine replizierte Klasse wird dazu im Programm mit einer Markierung versehen und durch einen Präprozessor auf die Replikation durch eine Programmtransformation vorbereitet (vgl. Abschnitt 3.6.3). Da der eigentliche Datenaustausch implizit während einer kollektiven Zustandsfortschreibung am Ende eines BSP-Hauptschritts stattfindet, kann der Programmierer vollständig davon abstrahieren.

3.4 Verteilter Objektraum

Die verteilte Umgebung muss zweierlei leisten: Die Objekte eines verteilt parallelen Programms müssen über die Hauptspeicher von kooperierenden virtuellen Maschinen transparent verteilt gespeichert werden können, und nebenläufige Aktivitäten müssen in dem so verteilten Objektraum wie in einer konventionellen virtuellen Maschine arbeiten können. Nur auf diese Weise wird das von Java vorgegebene Modell für Kontrollparallelität korrekt auf die verteilte Umgebung abgebildet.

In reinem Java ist ein Objekt an diejenige virtuelle Maschine gebunden, in der es geladen wurde. Referenzen auf Objekte innerhalb einer virtuellen Maschine sind nicht exportierbar, sondern nur innerhalb ihrer virtuellen Maschine gültig. Anders als in vielen anderen Sprachen gibt es weder die Möglichkeit, eine Referenz in einen anderen Datentyp (beispielsweise eine Ganzzahl) zu konvertieren, noch die Möglichkeit, andere Operationen als die Dereferenzierung auf Referenzen auszuführen. Diese Abgeschlossenheit der Java-Referenzen ist ein wichtiges Instrument für eine sichere Programmausführung, in der ausgeschlossen ist, dass Methoden an Objekten aufgerufen werden, welche diese Methoden nicht unterstützen und daher undefiniert reagieren. Ohne Modifikation der virtuellen Maschine können Java-Objekte also nicht *direkt* verwendet werden, um einen verteilten Objektraum aufzubauen, der mehrere virtuelle Maschinen umfasst. Auf Modifikation der virtuellen Maschine soll aber bewusst verzichtet werden, um die Portabilität der Lösung nicht in Frage zu stellen.

Für den verteilten Objektraum der vorliegenden Arbeit werden Konzepte von RMI und JavaParty [85] verwendet und um transparente entfernte Synchronisation und die Repräsentation von replizierten Objekten erweitert. Die folgenden Abschnitte erklären die Architektur des verteilten Objektraums, nachdem die verwandten Techniken besprochen wurden.

3.4.1 Das RMI-Modell

Die Java-Standardbibliothek enthält mit RMI ein Paket für entfernten Methodenaufruf. Mit diesem Paket lässt sich die Beschränkung der nur lokalen Referenzierbarkeit von Objekten aufheben. Im RMI-Modell wird ein Objekt, das eine separat definierte

Schnittstelle von entfernt aufrufbaren Methoden implementiert, durch Export zu einem Dienstgeber (*entferntes Objekt*). Ein solches Objekt wird über eine sog. *entfernte Referenz* angesprochen, die auch außerhalb seiner virtuellen Maschine gültig ist. RMI überwindet die Beschränkung von Referenzen auf ihre virtuelle Maschine durch eine Abbildung von Java-Referenzen nach Objektnummern. Wird ein Objekt exportiert, wird seine lokale Referenz in diese Tabelle eingetragen. Um das Objekt bei einem Zugriff von außerhalb zu identifizieren, dient die Nummer in dieser Objekttabelle. Bei einem Zugriff über das Netzwerk wird die Nummer zurück in eine lokale Referenz übersetzt und der Zugriff über diese Referenz durchgeführt. Objektnummer und Identifikation der virtuellen Maschine, auf der sich das entfernte Objekt befindet, werden in einem weiteren Objekt gekapselt und dienen als entfernte Referenz, die mittels Objektserialisierung auf andere virtuelle Maschinen kopiert werden kann.

Das Objekt, welches die entfernte Referenz kapselt, bietet dieselbe Schnittstelle an, die vom Dienstgeber exportiert wurde. Die entfernte Referenz dient als Stellvertreter des Dienstgebers in einer anderen virtuellen Maschine. Wird eine Methode an dem Stellvertreterobjekt aufgerufen, stellt dieses eine Verbindung zu der virtuellen Maschine des Dienstgebers her, den es repräsentiert. Es initiiert dort den angeforderten Methodenaufruf, indem es die Objektnummer des Dienstgebers, einen Identifikator für die aufgerufene Methode und die Argumente übermittelt. Nachdem der Dienstgeber die Methode ausgeführt hat, übermittelt er das Ergebnis an seinen Stellvertreter zurück, der dieses so an seinen Aufrufer zurückgibt, als wäre es lokal berechnet worden. Über entfernte Stellvertreter für Dienstgeberobjekte stellt RMI die Basis für einen transparenten Fernaufruf zur Verfügung.

Problematisch an RMI sind folgende Eigenschaften, die einer leichten Portierung eines regulären Java-Programms für eine verteilte Umgebung im Wege stehen, da erhebliche Änderungen notwendig sind:

- Entfernte Objekte können nur lokal erzeugt werden. Daher müssen Objekte unter einem wohlbekannten Namen registriert werden, um von einem entfernten Knoten aus benutzbar zu sein.

- Nach ihrer Erzeugung können entfernte Objekte ihren Knoten nicht mehr wechseln (keine Migration).

- Es ist weder Fernzugriff auf Instanzvariablen von Objekten möglich, noch ist die entfernte Klasse zur Laufzeit adäquat repräsentiert. Details werden in Abschnitt 5.5 besprochen.

- Bei jedem entfernten Methodenaufruf muss eine Ausnahmebehandlung angegeben oder die Ausnahme muss deklariert werden. Dies verkompliziert den Programmtext einer verteilten Anwendung im Rechnerbündel unnötig.

- Die Geschwindigkeit, mit der ein Fernaufruf über RMI durchgeführt werden kann, ist für eine Bündelumgebung nicht ausreichend.

In der vorliegenden Arbeit wird von RMI aufgrund der vielen Unzulänglichkeiten nur das Prinzip des Fernaufrufs von Methoden entfernter Objekte übernommen.

3.4.2 Das JavaParty-Modell

JavaParty [85] erweitert das RMI-Modell um entfernte Objekterzeugung, entfernten Feldzugriff, Objektmigration und entfernte Klassen. Ausnahmebedingungen, die bei Netzwerkproblemen während des Fernaufrufs auftreten können, werden automatisch behandelt. Damit wird die Verwendung einer entfernten Klasse und deren Instanzen in der verteilten Umgebung syntaktisch ununterscheidbar von regulären Klassen und Objekten in Java. JavaParty erreicht diese Transparenz durch eine Programmtransformation, die eine annotierte Klassendeklaration und ihre Verwendung in Benutzungen der RMI-Bibliothek übersetzt. Eine annotierte Klasse wird zu einer *transparent entfernten* Klasse.

JavaParty behebt bis auf die Ineffizienz alle oben genannten RMI-Probleme. Dadurch gleicht ein JavaParty-Programm syntaktisch einem regulären Java-Programm. Die verteilte Umgebung ist aber noch nicht perfekt:

- Koordination paralleler Aktivitäten über entfernte Objekte ist nur eingeschränkt möglich, da keine adäquate Repräsentation für maschinenüberspannende Kontrollfäden existiert. Dies kann zu Verklemmungen führen, weil Sperren von derselben Aktivität nicht mehrfach betreten werden können. Dieser Aspekt wird ausführlich in Abschnitt 5.3 diskutiert.

- Entfernte Objekte stellen bei häufigen verteilten Lesezugriffen einen Flaschenhals dar. Die Umgebung kennt keine Form von Replikation, um dieses Zugriffsmuster zu unterstützen.

- Viele Probleme lassen sich datenparallel formulieren. JavaParty sieht aber keine Mechanismen vor, um den Programmierer bei der Formulierung zu unterstützen.

Die vorliegende Arbeit übernimmt von JavaParty das Konzept transparent entfernter Klassen und ergänzt diese um die oben aufgeführten Punkte.

3.4.3 Objekte in der verteilten Umgebung

In der hier entworfenen Programmierumgebung werden die Objekte und Klassen nicht nur verstreut über die teilnehmenden virtuellen Maschinen gespeichert und dort entfernt ansprechbar gemacht, sondern der gesamte Programmfluss einer mehrfädigen Anwendung wird transparent auf die verteilte Umgebung abgebildet. In Java ist der in Objekte gekapselte Zustand nur ein Teil der Ausführungssemantik eines parallelen Programms. Wie in Abschnitt 3.3.1 diskutiert, existieren in Java parallele Aktivitäten größtenteils unabhängig von den Klassen und Objekten, deren Methoden sie abarbeiten. Eine parallele Anwendung nimmt über Synchronisations- und Kommunikationsprimitive nebenläufige Aktivitäten als Teil ihres Laufzeitzustandes wahr und kontrolliert sie über Koordinationsprimitive. Eine Aktivität hat in Java eine eigenständige Identität wie ein Objekt, nur dass sie nicht direkt referenziert werden kann. Die Aktivität wird zwar immer über ein sie repräsentierendes Objekt manipuliert, sie ist aber dennoch eine eigenständige Entität des Programms. Diese Eigenschaft offenbart sich für eine Anwendung daran, dass Aktivitäten nur diejenigen kritischen Abschnitte

betreten können, deren Monitor sie besitzen. Eine Definition der Semantik von den zugehörigen Sperroperationen findet sich im Java-Sprachstandard [31].

Werden die Objekte des Programms über mehrere virtuelle Maschinen verteilt, muss es für die Aktivitäten, welche die Methoden der Objekte abarbeiten, möglich sein, je nach Ort des angesprungenen Zielobjektes die virtuelle Maschine zu wechseln. Aktivitäten sind dann transparent maschinenüberspannend, wenn sich ihr Verhalten nicht mit der Verteilung der Objekte des Programms ändert. Dies ist nur möglich, wenn eine Aktivität ihre Identität während der Ausführung entfernter Methodenaufrufe behält. Nur so ist gewährleistet, dass sich Synchronisationsprimitive für die maschinenüberspannende Aktivität genauso verhalten wie in einem nichtverteilten Programm. Das ist keineswegs selbstverständlich, da Aktivitäten genauso wie Objekte an die virtuelle Maschine gebunden sind, in der sie erzeugt wurden. Damit kann eine Aktivität nur Methoden von Objekten abarbeiten, die in derselben virtuellen Maschine liegen. Einen Fernaufruf kann die Aktivität nicht selbst durchführen. RMI behilft sich damit, dass ein Fernaufruf an eine andere Aktivität delegiert wird, die vom Laufzeitsystem auf demjenigen Knoten erzeugt wird, auf dem sich das Zielobjekt des Fernaufrufs befindet. Die aufrufende Aktivität wartet so lange, bis das Ergebnis des Aufrufs vorliegt. Die beiden Aktivitäten stehen aber darüber hinaus in keiner Beziehung zueinander. Das führt dazu, dass Fernaufrufe über RMI nur insoweit transparent funktionieren, als keine Koordination von Aktivitäten notwendig ist. Ein zentraler Bestandteil eines parallelen Programms ist aber gerade die Koordination. Damit lässt sich auf Basis von RMI kein transparent verteilter Objektraum realisieren. Jede Abbildung einer verteilten Umgebung auf RMI erbt unvermeidlich dieses Problem.

In einem Programm mit zwei entfernten Objekten `x` und `y` tritt beispielsweise dann eine Verklemmung auf, wenn aus einer synchronisierten Methode `foo()` von `x` ein Fernaufruf an Objekt `y` stattfindet, der seinerseits eine ebenfalls synchronisierte Methode `x.bar()` aufruft. Ebenso gibt es in einer mit RMI verteilten Umgebung kein Äquivalent für das Java-Konstrukt `synchronized(x) {...}`.

Für einen auch bezüglich Synchronisation transparent verteilten Objektraum muss ein neuer Fernaufruf entwickelt werden. Dieser Fernaufruf für die Bündelumgebung muss neben der Ausführungsreihenfolge und Datenflüsse auch die Semantik bezüglich Koordination in einem verteilt parallelen Programm erhalten. Der neue Fernaufruf muss effizient sein, um die Kommunikationsleistung eines Rechnerbündels ausnutzen zu können. Die Abbildung von entfernten Klassen muss mit Hilfe dieses neuen Fernaufrufs erfolgen. Dabei kann der Teil der Abbildung, der sich mit dem Zugriff auf eine entfernte Klasse beschäftigt, von der JavaParty-Transformation übernommen werden. Sprachprimitive für Koordination müssen dagegen direkt auf Mechanismen des neuen Fernaufrufs abgebildet werden. Eine Simulation von transparent maschinenüberspannenden Kontrollfäden durch eine Transformation nach RMI ist nicht möglich.

Kapitel 4 legt mit einem schnellen Übertragungsmechanismus für Objektstrukturen die Grundlage für einen effizienten Fernaufruf. Darauf aufbauend wird in Kapitel 5 ein Fernaufruf realisiert, der transparent maschinenüberspannende Kontrollfäden unterstützt und für Hochleistungs-Kommunikation im Rechnerbündel ausgelegt ist. Die folgenden Abschnitte motivieren die Einführung von Replikation in die verteilte Umgebung und die direkte Unterstützung datenparalleler Operationen.

3.5 Grenzen der Einsetzbarkeit entfernter Objekte

Mit entfernten Klassen wird eine exakte Nachbildung des Java-Modells in der verteilten Umgebung vorgenommen: Beim Zugriff auf ein entferntes Objekt wechselt die Aktivität zu diesem Objekt und damit in der verteilten Umgebung zum Knoten, auf dem das Objekt angelegt ist. Berechnung und Synchronisation ist an den Aufenthaltsort des zugehörigen Objektes gebunden. Für dieses Modell spricht ein geringer Lernaufwand, wenn der Programmierer mit Kontrollparallelität in Java bereits vertraut ist.

Mit der Anordnung der Objekte auf den Knoten der verteilten virtuellen Maschine geht implizit eine Verteilung der Aktivitäten einher, die mit diesen Objekten arbeiten. Die verfügbare Parallelität wird dadurch ausgenutzt, dass unterschiedliche Aktivitäten gleichzeitig an verschiedenen Objekten arbeiten, die auf unterschiedlichen Knoten angelegt sind. Aktivitäten interagieren, indem sie auf gemeinsame Daten schreibend und lesend zugreifen. Ein Objekt, das von mehreren Aktivitäten gemeinsam genutzt wird, bewirkt, dass sich diese Aktivitäten beim Zugriff auf demselben Knoten „treffen“ und sich dort mittels Java-eigener Primitive synchronisieren.

Wenn Interaktionen zwischen den Aktivitäten selten auftreten und verhältnismäßig kurz sind, kann die verfügbare Parallelität durch entfernte Objekte optimal genutzt werden. Durch die Verteilung von Objekten, die jeweils ausschließlich durch eine Aktivität genutzt werden, auf unterschiedliche Knoten der verteilten virtuellen Maschine, können diese Aktivitäten ungehindert parallel ablaufen. Während einer Interaktion treffen sich die Aktivitäten auf demselben Knoten und teilen sich in dieser Zeit die dort verfügbaren Ressourcen. Hieran werden sofort auch die Grenzen der Verwendbarkeit entfernter Objekte deutlich. Jede gemeinsame Nutzung eines Objektes schränkt die mögliche Parallelität ein, da alle Nutzungen auf dem Rechnerknoten stattfinden, auf dem sich das betreffende Objekt aufhält. Je häufiger Zugriffe auf Objekte sind, die von mehreren Aktivitäten gemeinsam benutzt werden, desto unattraktiver werden entfernte Objekte für das verteilt parallele Programm.

3.5.1 Optimierung entfernter Zugriffe

Geschwindigkeitsprobleme, die durch zu häufige Fernaufrufe entstehen, könnten gelöst werden, wenn sich durch Optimierungen die Netzwerklatenz beim entfernten Zugriff verbergen ließe. Dies könnte durch folgende Maßnahmen erreicht werden:

- Viele kurze Zugriffe auf entfernte Objekte werden zu einem Zugriff gebündelt.
- Der lokale Kontrollfluss rechnet nach dem Absetzen eines Fernaufrufs solange weiter, bis das Resultat benötigt wird.
- Ein Fernaufruf wird abgeschickt, noch bevor der Kontrollfluss die Stelle des Aufrufs erreicht, damit das Ergebnis rechtzeitig vorliegt, wenn es benötigt wird.

Allerdings sind in objektorientierten Sprachen solche Optimierungen nicht oder nur sehr selten anwendbar. Methodenaufrufe haben Seiteneffekte oder lenken durch Auslösen einer Ausnahmebedingung den lokalen Kontrollfluss nach Rückkehr aus der Methode um. Solche Fälle können durch den Übersetzer aufgrund von Polymorphie

nur schlecht vorhergesagt werden. Daher sind die beschriebenen Optimierungsstrategien nicht zulässig, weil sie möglicherweise die Programmsemantik ändern. Um diese jedoch zu erhalten, muss ein entfernter Methodenaufruf bis auf wenige Trivialfälle genau an der Stelle im Programm durchgeführt werden, an welcher er im Programmtext auftaucht. Optimierungen zum Verbergen der Netzwerklatenz und zum Bündeln von Fernaufrufen sind also im allgemeinen nicht einsetzbar.

Die folgenden Abschnitte diskutieren Muster der gemeinsamen Nutzung von Objekten in einem parallelen Programm, bei denen deutlich wird, dass entfernte Objekte für verteilt paralleles Programmieren nicht ausreichen. Aus diesen Anwendungsbeispielen wird eine Erweiterung des Programmiermodells für Rechnerbündel abgeleitet.

3.5.2 Gemeinsam benutzte, aber selten modifizierte Daten

Beim Übergang von einer zentralisierten zu einer verteilten Umgebung wird die Datenrepräsentation dann zum Problem, wenn Daten häufig von mehreren Aktivitäten verteilt genutzt werden. Solche Daten ändern sich in der Regel relativ selten während des Programmablaufes, weshalb auf sie aus Effizienzgründen lokal zugegriffen werden müsste. Kapselt man solche Informationen aber in einem entfernten Objekt, finden alle Zugriffe entfernt statt, unabhängig davon, ob der Zugriff die gemeinsam benutzten Informationen ändert oder nicht. Diejenige Aktivität wird beim Zugriff begünstigt, die lokal zugreift. Alle anderen Aktivitäten führen Fernaufrufe für ihre Zugriffe aus. Bei jedem Fernaufruf muss dabei die volle Netzwerklatenz abgewartet werden, und jeder Zugriff behindert die lokal zugreifende Aktivität und schränkt die Parallelität ein.

In diesem Anwendungsfall ist die zentrale Annahme für den Einsatz entfernter Objekte verletzt. Bei der Verteilung eines Programms durch Zerlegen in entfernte Objekte geht man davon aus, dass durch eine geeignete Anordnung der resultierenden Objekte auf den Knoten der verteilten Umgebung die Lokalität der Zugriffe optimiert werden kann. Eine lokalitätsoptimierte Anordnung minimiert die notwendigen Fernaufrufe, die zur Kommunikation der parallelen Aktivitäten notwendig sind, und maximiert die mögliche Parallelität durch Separation von unabhängig verwendeten Objekten. Im beschriebenen Anwendungszenario gibt es aber für die gemeinsam verwendeten Informationen keinen Ort, der die Zahl der Fernzugriffe minimiert. Gleichgültig wo das zugehörige entfernte Objekt platziert wird, stets finden übermäßig viele Fernaufrufe von allen anderen Knoten her statt. Besser für die Lokalität der Zugriffe wäre es, wenn nicht der Ort, von dem der Zugriff ausgeht, darüber entscheiden würde, ob ein Fernaufruf notwendig ist, sondern die Art des Zugriffs. Optimale Lokalität läge vor, wenn die gemeinsam benutzte Information auf jedem Knoten lokal als Kopie vorrätig wäre. Lesezugriffe könnten dann auf jedem Knoten ohne Fernaufruf befriedigt werden, Schreibzugriffe müssen die Zustandsänderung an alle Kopien weitergeben.

Viele im Abschnitt 2.6 besprochenen Arbeiten gehen genau dieses Problem gemeinsam benutzter Objekte an. Der dargestellte Fall ist das Paradebeispiel gegen den Einsatz entfernter Objekte und für die Verwendung von Replikationsstrategien. Wie die folgenden Abschnitte zeigen, ist der oben besprochene Fall aber nicht der einzige, bei dem die Abstraktion entfernter Objekte versagt. In der vorliegenden Arbeit wird eine umfassende Lösung für alle diese Probleme eingeführt.

3.5.3 Verteilte Datenstrukturen

Ein entferntes Objekt ist zu einem bestimmten Zeitpunkt immer auf genau einem Knoten der verteilten Umgebung lokalisiert. Alle Berechnungen, die an diesem Objekt ausgeführt werden, finden auf dem Knoten statt, auf welchem sich das Objekt aufhält. Damit kann mit einem einzelnen entfernten Objekt nie eine verteilte Datenstruktur repräsentiert werden. Als Beispiel kann wieder das Feld aus Abbildung 3.2 dienen. Wenn jeder Prozessor diejenigen Feldelemente zugeschlagen bekommt, die er in einem parallelen Schritt berechnet, erhält Prozessor P_0 die obere Hälfte des Feldes und Prozessor P_1 entsprechend die untere Hälfte. Das Feld ist so über die beiden Prozessoren partitioniert. Jeder Prozessor kann bei dieser Verteilung auf diejenigen Elemente lokal zugreifen, die lokal bei ihm gespeichert sind. Bei einem verteilten Feld in HPF werden Zugriffe auf andere, nichtlokale Datenelemente auf Fernzugriffe abgebildet. Die folgenden Abschnitte untersuchen mögliche Formen verteilter Datenstrukturen. In allen Fällen stellen sich sowohl entfernte Objekte als auch herkömmliche Replikation als nicht adäquat heraus.

Verteilte Felder

Ein Feld kann nicht mit Hilfe eines einzigen entfernten Objektes verteilt werden. Da ein entferntes Objekt immer auf genau einem Rechenknoten angelegt ist, wird auf jedem Rechenknoten, auf den ein Teil des Feldes verteilt werden soll, mindestens ein entferntes Objekt benötigt. Diese Verteilung lässt sich nur durch eine Gruppe von entfernten Objekten erreichen. Jedes einzelne Objekt repräsentiert dann die jeweils lokale Partition des verteilten Feldes. Bei einem Feld ist dies vergleichsweise einfach zu realisieren, da sich am Feldindex die Zugehörigkeit zu einem Teil der Partition entscheiden lässt. Für den Zugriff auf andere Partitionen in HPF-Manier benötigt jedes Objekt zusätzlich noch entfernte Referenzen auf alle anderen Repräsentanten von Feldpartitionen. Jedes dieser entfernten Objekte kann dann Zugriffe auf seine lokale Partition direkt durchführen und Zugriffe auf Feldelemente fremder Partitionen an den entsprechenden Repräsentanten dieser Partition weiterleiten.

Mit diesem Entwurf könnte man – zugegebenermaßen etwas umständlich – verteilte Felder im Stil von HPF realisieren. Auf jedem Knoten finden Zugriffe auf die lokale Partition lokal statt, während Zugriffe auf andere Teile in Fernaufrufe umgewandelt werden. Greift ein datenparalleler Algorithmus, der alle Feldelemente gleichzeitig manipuliert, ausschließlich auf die jeweils lokale Partition zu, sind alle Zugriffe lokal und es finden keine Fernaufrufe statt. Dies ist aber nur für trivial parallele Algorithmen der Fall. Überlappen die Bereiche, auf die zugegriffen werden muss, finden viele kleine Fernaufrufe statt, um jeweils einen einzelnen Feldzugriff durchzuführen. Dadurch, dass sich diese Feldzugriffe aber hinter einer objektorientierten Fassade verbergen, können aus oben erwähnten Gründen Optimierungen zum Verbergen der Netzwerklatenz, wie sie Müller in [76] für HPF beschreibt, nicht eingesetzt werden.

Auch herkömmliche Strategien für Objektreplikation versagen bei einem verteilten Feld. Ein repliziertes Feldobjekt wird immer als Ganzes betrachtet und kann zu einem Zeitpunkt nur von einer Aktivität modifiziert werden. Bei einem seitenbasierten DSM-System kommt es bei Zugriffen auf die Ränder der Feldpartitionen zu einem Hin- und Herkopieren der Seite, an der die Partitionen aneinanderstoßen. Ein objektbasiertes

DSM-System benötigt eine Sonderbehandlung für Felder oder muss inkrementelle Zustandsfortschreibung unterstützen, um überhaupt mit verteilten Schreibzugriffen umgehen zu können. Herkömmliche Replikation lindert zwar die Probleme bei verteilt stattfindendem Zugriff auf *gemeinsame* Daten, versagt aber bei parallelen Operationen an *verteilten* Strukturen. Eine verteilte Programmierumgebung benötigt folglich eine umfassendere Lösung als die einfache Hinzunahme replizierter Objekte.

Verteilte Graphen

Möchte man statt eines Feldes einen Objektgraphen verteilt speichern und parallel verarbeiten, verschärft sich das Problem beim Einsatz von entfernten Objekten: Um den Objektgraphen auf N Knoten der Umgebung zu verteilen, muss der Graph in N Partitionen zerschnitten werden. Dazu müssen all jene Referenzen, die eine Partitionsgrenze überspannen, in entfernte Referenzen umgewandelt werden. Dies ist nur möglich, wenn die Klassen all jener Objekte, auf die eine partitionsgrenzen-überspannende Referenz zeigt, in entfernte Klassen umgewandelt werden. Besteht der Graph aus sehr vielen ($M \gg N$) Objekten, die Instanzen von wenigen ($K \ll M$) Klassen sind, ist die Wahrscheinlichkeit groß, dass auch bei einem geschickten Schnitt durch den Graphen fast jede Klasse in eine entfernte Klasse umgewandelt werden muss. Damit wird nahezu jedes der M Objekte des Graphen zu einem entfernten Objekt. Die Partitionierung des Graphen ist so zwar möglich, allerdings besteht jede einzelne Partition aus vielen (M/N) entfernten Objekten. Zurückübertragen auf das Feld-Beispiel hieße das, dass jedes einzelne Feldelement in ein eigenes entferntes Objekt gekapselt werden müsste. Für jedes dieser Objekte fällt Mehraufwand für die Verwaltung an. Alle Referenzen werden zu potenziell entfernten Referenzen, von denen nur wenige tatsächlich auf ein Objekt in einer anderen Partition zeigen. Jeder Zugriff auf eine solche Referenz ist aber teurer als ein direkter lokaler Zugriff. Eine datenparallele Berechnung kann jetzt zwar auf dem so verteilten Graphen durchgeführt werden. Das schon bei verteilten Feldern beschriebenen Problem bleibt aber bestehen, dass Zugriffe auf Nachbarelemente der Partitionsgrenzen zu Fernaufrufen führen, deren Latenz nicht verdeckt werden kann. Erschwerend kommt hier noch hinzu, dass auch die Berechnung innerhalb einer Partition durch die Umwandlung der Objekte in schwergewichtige entfernte Objekte behindert wird.

Eine Graphstruktur lässt sich wie beschrieben zwar elegant durch Umwandlung aller Knotenobjekte in entfernte Objekte verteilen, aber die zu erwartende Effizienz einer datenparallelen Operation auf einem so verteilten Graphen ist inakzeptabel. Systeme, die auf die verteilt parallele Verarbeitung von Feldern spezialisiert sind, versagen völlig. Herkömmliche Objektreplikation kann aufrund der nebenläufigen Schreibzugriffe wieder nicht angewendet werden.

3.5.4 Koordination bei verteilten Datenstrukturen

Bei Verwendung entfernter Objekte führt ein Programm, das auf mehr Daten zugreift, als zur lokalen Partition gehören – wie im letzten Abschnitt beschrieben – zu Fernaufrufen, die sich die fehlenden Daten bei Bedarf beschaffen. Bei einer datenparallelen Operation ist dies allerdings problematisch, da gleichzeitig an allen anderen Partitio-

nen der Datenstruktur ebenfalls gearbeitet wird. Ohne weitere Maßnahmen liefert der Fernzugriff in diesem Fall inkonsistente Daten. Die Aktivitäten, welche die datenparallele Operation auf den Partitionen der Datenstruktur durchführen, arbeiten während ihrer Berechnung unabhängig voneinander. Deshalb ist nicht klar, in welchem Zustand die Daten zum Zeitpunkt des Fernzugriffs auf eine fremde Partition sind. Möglicherweise ist die Operation auf den fremden Daten bereits ausgeführt oder der Zugriff greift mitten in die Operation oder die Daten sind noch unverändert. Dieser Indeterminismus ist nicht tolerabel. Daher müssen die parallelen Aktivitäten auch während der Ausführung eines parallelen Schrittes koordiniert werden. Koordination erfordert aber Mehraufwand und schränkt die mögliche Parallelität ein.

An dieser Stelle kann man erkennen, dass auch ein DSM-basiertes System mit dieser Situation nicht umgehen kann. Zwar wird sich die Verteilung der Objekte innerhalb einer Partition automatisch an die verteilte Benutzung in den unterschiedlichen Aktivitäten anpassen, der Zugriff auf die Partitionsränder müsste von der Applikation aber koordiniert werden. Ansonsten würden Zugriffe ebenso indeterministische Ergebnisse liefern wie bei der Realisierung mit entfernten Objekten. Dies liegt daran, dass das DSM-System die Verteilung komplett vor der Anwendung zu *verbergen* sucht. In einem BSP-Programm aber kann die Verteilung sogar *ausgenutzt* werden, um den Zustand einer Struktur deterministisch auf verschiedenen Knoten in unterschiedlichen Versionen zu sehen und nur zu bestimmten Zeitpunkten einen Abgleich vorzunehmen.

Zusammenfassend kann man feststellen, dass entfernte Objekte für die Realisierung einer verteilten Datenstruktur für ein datenparalleles Programm nicht ausreichend sind. Dafür können drei Gründe identifiziert werden:

- Die Umwandlung aller (Teil-)Objekte der verteilten Datenstruktur in entfernte Objekte hat einen negativen Einfluss auf die Ausführungsgeschwindigkeit der eigentlichen Berechnung.
- Die Latenz der Fernzugriffe, welche Daten aus benachbarten Partitionen beschaffen, kann nicht überdeckt werden.
- Es ist zusätzliche Koordination während des datenparallelen Schritts notwendig, um Fernzugriffe auf inkonsistente Daten zu vermeiden.

Der folgende Abschnitt zeigt, wie alle oben genannten Probleme mit einem weiteren Konzept gelöst werden können.

3.6 Kollektive Replikation

Um das BSP-Modell für objektorientierte Programme nutzbar zu machen, muss von dem reinen Datenaustausch am Ende eines Hauptschrittes abstrahiert werden. Eine Betrachtung der Gesamtdatenstruktur ist hier hilfreich. Im Beispiel aus Abbildung 3.2 wird letztendlich mit einem auf die Prozessoren P_0 und P_1 verteilten Gitter gerechnet. Da jeder Prozessor für seine Berechnung mehr als die Hälfte der Gitterpunkte benötigt, reicht eine einfache Partition der Daten aber nicht aus. Das Gitter aus Abbildung 3.2 ist stattdessen partiell repliziert. Bei dieser partiellen Replikation sind diejenigen Gitterpunkte, die nur von genau einem Prozessor benötigt werden, auf diesem Prozessor

angelegt. Diejenigen Gitterpunkte, die von beiden Prozessoren für ihre Berechnung benötigt werden, sind auf beiden Prozessoren repliziert gespeichert. Der Datenaustausch am Ende eines jeden Hauptschritts dient dazu, eine konsistente Sicht beider Prozessoren auf das partiell replizierte Gitter herzustellen. In dieser Konsistenzoperation werden replizierte Gitterpunkte von demjenigen Prozessor, der sie im letzten Hauptschritt verändert hat, zu demjenigen Prozessor übertragen, der auf sie im nächsten Hauptschritt lesend zugreifen wird.

Betrachtet man einen BSP-Algorithmus als eine Abfolge kollektiver Operationen auf einer partiell replizierten Datenstruktur, ist die Übertragung des Modells auf objektorientierte Umgebungen und irreguläre Datenstrukturen wie z. B. Graphen von Objekten möglich. Mit dieser Sichtweise kann von der expliziten Kommunikationsoperation am Ende eines jeden Hauptschrittes abstrahiert werden. Die explizite Kommunikation wird zu einer Konsistenzoperation eines partiell replizierten Objektes. Im Beispiel der Lösung der Laplace'schen Differentialgleichung wäre das Gitter das partiell auf den Prozessoren P_0 und P_1 replizierte Objekt. Dabei sind genau diejenigen Teile des Objektes auf jedem Prozessor lokal verfügbar, die dieser für seine Berechnungen benötigt. Während der parallelen Berechnungen eines Hauptschrittes divergiert der Zustand des partiell replizierten Objektes. Jeder Prozessor modifiziert seine Partition des Gitters, während er den jeweils nur lesend benutzten replizierten Randbereich unverändert lässt. Da der zum unveränderten Randbereich korrespondierende Zustand auf dem jeweils anderen Prozessor modifiziert wurde, weisen die Randbereiche nach der parallelen Operation auf den beiden Prozessoren unterschiedliche Werte auf. Durch die Konsistenzoperation am Ende des Hauptschrittes wird die Konsistenz wiederhergestellt, indem die Zustandsänderung der Randbereiche auf den jeweils anderen Prozessor übertragen und dort nachvollzogen wird. Am Anfang des nächsten Hauptschrittes starten beide Prozessoren dann wieder mit einer konsistenten Sicht auf das partiell replizierte Objekt.

Die folgenden Abschnitte stellen anhand von Beispielen eine Verbindung zwischen datenparallelen Algorithmen im BSP-Modell und Replikation her. Diese Verbindung wird in der vorliegenden Arbeit genutzt, um ein Ausdrucksmittel für datenparallele Operationen in einem objektorientierten Kontext bereitzustellen. Als Lösung werden *kollektiv replizierte Objekte* vorgeschlagen.

3.6.1 Replikation als Ausdrucksmittel für Algorithmen im BSP-Modell

Um den Zusammenhang zwischen datenparallelen Algorithmen im BSP-Modell und Replikation zu verstehen, verdeutliche man sich die verteilten Datenstrukturen eines nachrichtenbasierten MPI-Programms. Das Beispiel aus Abschnitt 3.3.2 wird hier noch einmal aufgenommen. Abbildung 3.3 zeigt die Realisierung einer verteilten Matrix, wie sie typischerweise mit MPI realisiert wird. Beide Prozessoren P_0 und P_1 haben nur denjenigen Bereich der Matrix lokal in ihrem Speicher, den sie tatsächlich benötigen. Den Prozessoren fehlt dabei die Sicht auf die Gesamtstruktur. Eine solche Sicht muss manuell über eine Umrechnungsvorschrift von lokalen in globale Koordinaten bereitgestellt werden. Wenn beispielsweise Prozessor P_1 auf die Position $(2, 1)$ zugreift, findet er dort das Element $a_{4,1}$ vor. Während eines parallelen Hauptschrittes berech-

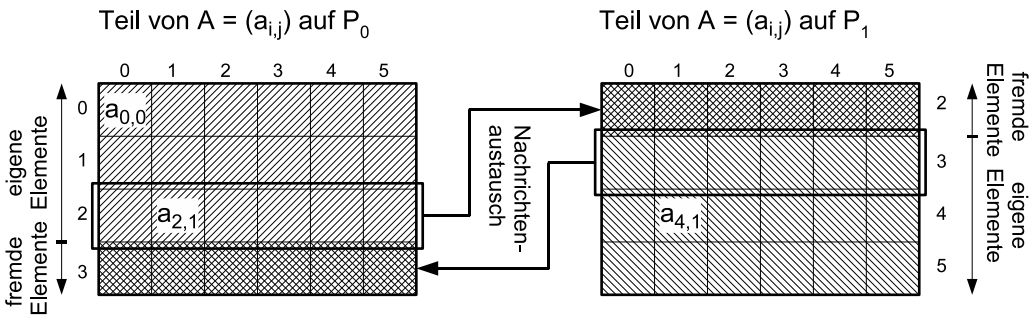


Abbildung 3.3: Verteilte Datenstruktur aus Abbildung 3.2 aus Sicht eines nachrichtenbasierten MPI-Programms.

net Prozessor P_0 den Teil der Matrix neu, der seine eigenen Elemente enthält. Dieser Bereich ist im linken Teil der Abbildung 3.3 durch einfache Schraffur gekennzeichnet. Dieser Teil umfasst nur einen Teil des auf Prozessor P_0 lokal abgelegten Bereichs der Matrix. Der andere Teil wird für die Berechnung von P_0 nur lesend benötigt. Allerdings wird der korrespondierende Teil im Speicher von P_1 während des Berechnungsschrittes dort modifiziert. Um die Konsistenz der Datenstruktur nach dem parallelen Hauptschritt wiederherzustellen, muss Prozessor P_0 den umrandeten Bereich aus seinem Speicher extrahieren und an Prozessor P_1 senden. Dieser muss die empfangenen Daten in den schraffierten Bereich in seinem lokalen Speicher einfügen. Prozessor P_1 verfährt entsprechend. Um die richtigen Speicherpositionen für die Extraktion und das Einfügen zu ermitteln, muss ständig von globalen in lokale Koordinaten umgerechnet werden und umgekehrt.

Betrachtet man die verteilte Datenstruktur als Ganzes, sieht man wieder die Matrix aus Abbildung 3.2, bei der die ersten drei Zeilen an Prozessor P_0 und die letzten drei Zeilen an Prozessor P_1 verteilt sind. Die mittleren beiden Zeilen sind auf beiden Prozessoren *repliziert*. Während des parallelen Hauptschrittes wird der replizierte Teil der Matrix parallel auf beiden Prozessoren modifiziert. Während des Kommunikationsschrittes werden die nebenläufigen Änderungen wieder zu einer konsistenten Sicht verschmolzen.

Mit den Mitteln der vorliegenden Arbeit kann die gesamte Matrix als ein *kollektiv* repliziertes Objekt betrachtet werden. Die Replikation ist dabei nur *partiell*, da nicht alle Teile des Objektes auf allen Prozessoren repliziert sind. Kollektiv replizierte Objekte unterstützen zwei Operationsmodi. Im Modus 1 kann ein kollektiv repliziertes Objekt entweder von einem einzelnen Prozessor exklusiv verändert oder von vielen Prozessoren gleichzeitig gelesen werden. Dieser Modus entspricht der herkömmlichen Replikation. Nachdem eine exklusive Modifikation abgeschlossen ist, wird die getätigte Änderung an alle anderen Replikate verteilt. Im Modus 2 kann es parallel von allen Prozessoren modifiziert werden. Während der anschießenden Konsistenzoperation werden die nebenläufig ausgeführten Zustandsänderungen automatisch verschmolzen. Durch die automatische Verschmelzung nebenläufiger Änderungen entfällt der besonders fehlerträchtige Teil eines nachrichtenbasierten verteilten Programms, in welchem auf jedem Prozessor der Überlappungsbereich der verteilten Struktur extrahiert und auf einem anderen Rechner passend eingesetzt werden muss.

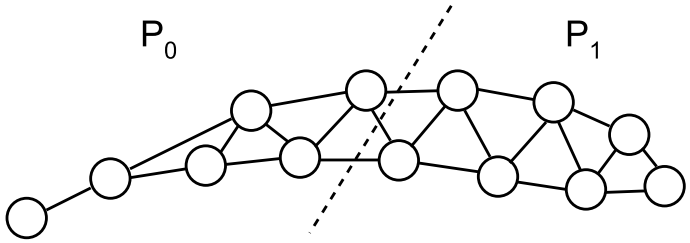


Abbildung 3.4: Drahtmodell für eine Finite-Elemente-Berechnung.

Ein kollektiv repliziertes Objekt kapselt eine über Rechnergrenzen hinweg verteilte Datenstruktur. Die wechselseitige Unabhängigkeit der parallelen Berechnungen eines Hauptschrittes im BSP-Modell wird gewährleistet, indem kollektiv replizierte Objekte mehrere gleichzeitige Schreiber zulassen. Die Applikation muss lediglich zusichern, dass die nebenläufigen Schreibzugriffe nicht überlappen. Bei Überlappungsfreiheit der Schreibzugriffe können die nebenläufig ausgeführten Modifikationen während der anschließenden Konsistenzoperation vollautomatisch zu einem konsistenten Zustand verschmolzen werden. Kann der Algorithmus nicht so vormuliert werden, dass die Schreibzugriffe überlappungsfrei sind, muss vom Programmierer zusätzlich eine Verschmelzungsoperation angegeben werden.

3.6.2 Datenparallelität auf irregulären Datenstrukturen

Ein kollektiv repliziertes Objekt, welches ein partiell repliziertes Gitter kapselt, versteckt sowohl die Koordinatenumrechnung als auch die Kommunikationsoperation für die Konsistenzoperation vor der Anwendung. Die volle Stärke kollektiv replizierter Objekte kommt aber erst bei irregulären Datenstrukturen zum Tragen. Abbildung 3.4 zeigt ein Drahtmodell, wie es für eine Finite-Elemente-Berechnung eingesetzt wird. Das Modell ist als Objektgraph repräsentiert. Ein Knoten im Drahtmodell ist mit denjenigen Nachbarn über Referenzen verbunden, mit denen er wechselwirkt.

Gleichzeitig ist in Abbildung 3.4 bereits angedeutet, das diese irreguläre Struktur zur datenparallelen Berechnung wieder auf zwei Prozessoren verteilt werden soll. Auch ein Finite-Elemente-Algorithmus lässt sich als BSP-Problem formulieren. Hier werden in zwei sich abwechselnden parallelen Hauptschritten jeweils die Kräfte berechnet, mit denen ein Knoten des Modells auf seine Nachbarknoten wirkt. Danach werden die Verformungen berechnet, die diese Kräfte verursachen. Nach den Berechnungen der Kräfte und Verformungen muss jeweils ein Datenaustausch unter den Prozessoren stattfinden, der die Zustandsänderungen in den Modellknoten am Rand der beiden Modell-Partitionen dem jeweils anderen Prozessor als Grundlage für die nächste Berechnung zur Verfügung stellt.

Abbildung 3.5 zeigt, welche Teilmenge der Modellknoten für die Rechenoperationen auf welchem Prozessor benötigt werden. Es gibt auch in diesem Beispiel wieder Daten, die nicht repliziert sind, weil sie nur auf einem Prozessor benötigt werden. Diese sind in der Abbildung weiß gezeichnet. Modellknoten, die sich am Rand der Partitionsgrenze befinden, sind dagegen auf beide Prozessoren repliziert. Korrespondierende Knoten sind jeweils gleich schraffiert und über gestrichelte Verbinder aneinanderge-

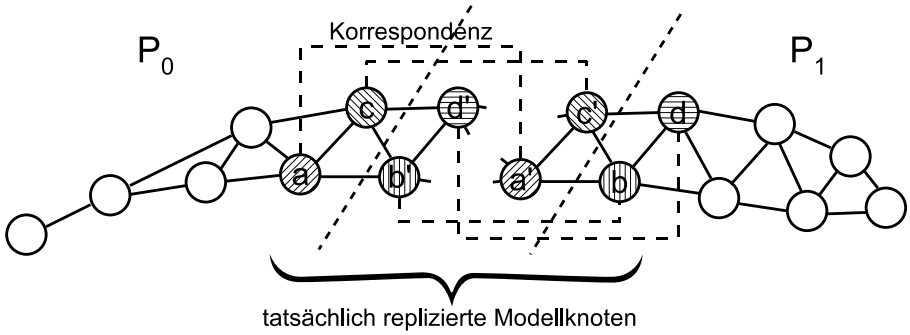


Abbildung 3.5: Aufteilung des Graphen aus Abbildung 3.4 auf die beiden Prozessoren.

koppelt. Nachdem beispielsweise Prozessor P_0 die Kräfte auf Knoten a und Prozessor P_1 die Kräfte auf Knoten b berechnet hat, müssen diese Zustandsänderungen auf den jeweiligen replizierten Zuständen a' und b' nachgezogen werden, bevor die Prozessoren die Verformung der Knoten a und b berechnen können.

In diesem Beispiel lässt sich der auszutauschende Datenbereich weder als zusammenhängender Speicherbereich noch durch einen MPI-Datentypen beschreiben. Wenn die Modellknoten als Objekte implementiert sind, die Referenzen auf ihre Nachbarknoten besitzen, stehen korrespondierende Replikate nur noch über ihre Verzeigerungsstruktur in Beziehung zueinander. Ein partiell repliziertes Objekt verwaltet diese Korrespondenzen von replizierten Teilen eines Objektes automatisch. Das gesamte Modell aus Abbildung 3.4 kann in ein partiell repliziertes Objekt gekapselt werden. Nachdem die Verteilung der Modellknoten auf die Prozessoren spezifiziert ist, werden nebenläufig ausgeführte Änderungen an dieser verteilten Datenstruktur in einer Konsistenzoperation automatisch verschmolzen. Dazu werden Änderungen an Modellknoten auf Prozessor P_0 an ihren Replikaten auf Prozessor P_1 nachgezogen und umgekehrt. Die Applikation muss lediglich die Berechnungen am Gesamtmodell, verteilt über die Prozessoren, durchführen und die BSP-Hauptschritte markieren. Damit findet automatisch nach jedem Hauptschritt eine Verschmelzung der Zustandsänderungen zu einem konsistenten Gesamtzustand statt. Die für die Konsistenzoperation notwendige Extraktion der geänderten Zustände, die Übertragung auf die jeweils anderen Prozessoren und das Wiedereinspielen übernimmt das Laufzeitsystem.

3.6.3 Kollektiv replizierte Objekte

Die vorliegende Arbeit ergänzt das kontrollparallele Programmiermodell von Java um *kollektiv replizierte Objekte*. Kollektive Replikation erweitert konventionelle Replikation um einen koordinierten parallelen Schreibzugriff auf den replizierten Zustand. Darüber lassen sich datenparallele Operationen nicht nur elegant formulieren, sondern es wird die Anwendung von Datenparallelität auf Objektgraphen im allgemeinen überhaupt erst möglich.

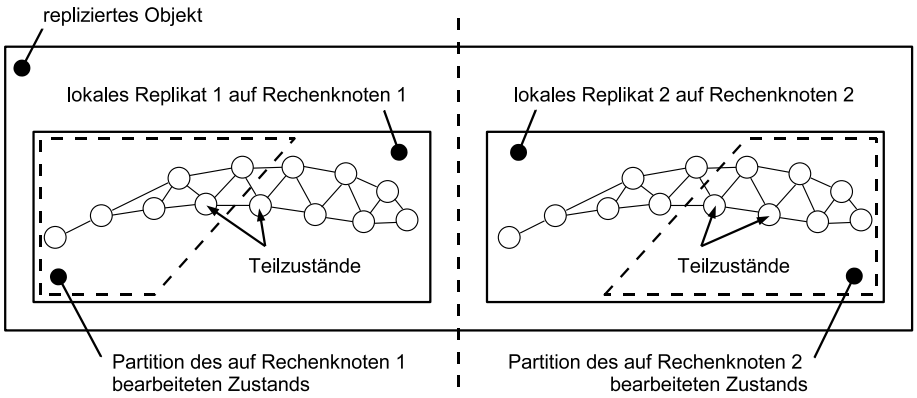


Abbildung 3.6: Konzeptuelle Sicht auf ein repliziertes Objekt.

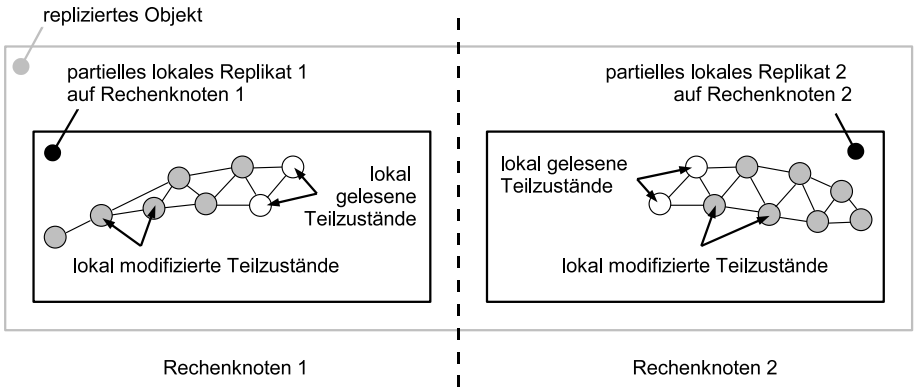


Abbildung 3.7: Sicht der lokalen Rechenknoten auf ein repliziertes Objekt.

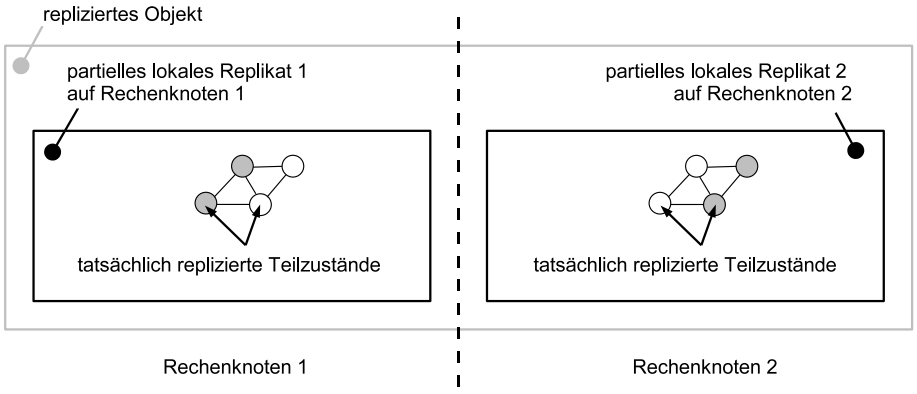


Abbildung 3.8: Sicht auf die tatsächlich replizierten Teile eines replizierten Objektes.

Begriffsklärung

Ein *kollektiv repliziertes Objekt* oder hier auch kurz *repliziertes Objekt* kapselt einen datenparallel verarbeitbaren Gesamtzustand. Man kann drei Sichtweisen auf ein solches Objekt unterscheiden, die in den Abbildungen 3.6, 3.7 und 3.8 dargestellt sind.

Konzeptuelle Sicht. Konzeptuell besteht ein *repliziertes Objekt* aus *lokalen Replikaten* auf jedem Rechenknoten, der an der parallelen Verarbeitung des in dem replizierten Objekt gekapselten Zustandes teilnimmt. Diese lokalen Replikate sind *konsistent* in dem Sinne, dass sie einen identischen Zustand aufweisen. Diese konzeptuelle Sicht ist in Abbildung 3.6 gezeigt. Jeder Rechenknoten hat durch sein lokales Replikat konzeptuell eine Komplettsicht auf die bearbeitete Gesamtstruktur. Jedes der lokalen Replikate besteht intern aus einer Menge von *Teilzuständen*, bei denen es sich um reguläre Java-Objekte handelt. Diese Teilzustände stellen den Zustand dar, den der datenparallele Algorithmus bearbeitet. Dazu unterhält die Applikation auf denjenigen Rechenknoten, die an der datenparallelen Berechnung teilnehmen, Kontrollfäden, welche *kollektiv*, d.h. koordiniert, die parallele Modifikation der Gesamtstruktur durchführen. Dazu betreten diese Kontrollfäden vor Beginn der datenparallelen Operation einen am replizierten Objekt *kollektiv synchronisierten Abschnitt*. In diesem kollektiv synchronisierten Abschnitt darf jeder Kontrollfaden parallel eine *Partition*[1] des Gesamtzustandes modifizieren. Dazu sind auch Lesezugriffe auf die übrigen Teile der konzeptuell vollständigen lokalen Replikate erlaubt. Vor dem Verlassen des kollektiv synchronisierten Abschnitts sorgt das Laufzeitsystem vollautomatisch dafür, dass die parallel durchgeführten Änderungen am replizierten Zustand ausgetauscht werden. Dadurch wird die Konsistenz aller lokalen Replikate, welche durch die parallele Modifikation verloren gegangen war, wiederhergestellt. Mit Wiederherstellung der Konsistenz ist die Voraussetzung für die Durchführung eines weiteren datenparallelen Schrittes geschaffen.

Sicht der lokalen Rechenknoten. Tatsächlich wird ein Rechenknoten, der an einer datenparallelen Operation beteiligt ist, nie auf alle Teilzustände seines konzeptuell vollständigen lokalen Replikats zugreifen müssen. Stattdessen benötigt jeder Rechenknoten nur denjenigen Teil der Gesamtstruktur, für dessen Berechnung er zuständig ist, und zusätzlich einen gewissen *Randbereich*, der in seine Berechnung mit einfließt.[2] In Abbildung 3.7 sind die Teilzustände der Gesamtstruktur grau hinterlegt, die auf dem jeweiligen Rechenknoten modifiziert werden. Teilzustände, auf die nur lesend zugegriffen wird, sind weiß dargestellt. Alle übrigen Teilzustände, die zwar konzeptuell zum jeweiligen lokalen Replikat gehören, aber auf dem betreffenden Rechenknoten nicht für die Berechnung benötigt werden, sind auf dem entsprechenden Rechenknoten auch nicht angelegt. Diese Teilzustände fehlen in der Abbildung 3.7 daher ganz. Auf jedem Rechenknoten existiert damit nur genau die Teilmenge der Teilzustände des konzeptuell vollständigen lokalen Replikats, die dieser Rechenknoten auch tatsächlich

[1] Parallele Modifikationen, die nicht überlappungsfrei sind, werden ebenfalls unterstützt, wenn das Programm eine Verschmelzungsvorschrift spezifiziert (vgl. Abschnitt 6.2.5).

[2] Die Größe und die Struktur dieses Randbereiches ist durch den konkreten datenparallelen Algorithmus bestimmt.

benötigt. Da die tatsächlichen Replikate keine vollständigen Kopien des Gesamtzustandes sind, heißen sie *partielle lokale Replikate* oder kurz *lokale Replikate*, wenn im Zusammenhang das tatsächliche Speicherlayout auf den Rechenknoten offensichtlich oder unwichtig ist. Ein solches lokales Replikat ist damit die lokale Sicht eines Rechenknoten auf die verarbeitete Gesamtstruktur. Diese Gesamtstruktur ist aber in der Regel auf keinem Rechenknoten tatsächlich als Ganzes angelegt, sondern stellt lediglich die gedachte Grundmenge dar, auf welcher der datenparallele Algorithmus formuliert ist.

Sicht auf die tatsächlich replizierten Teilzustände. Streicht man aus Abbildung 3.7 alle diejenigen Teilzustände, die nur zu einem einzigen lokalen Replikat beitragen, so erhält man die Sicht aus Abbildung 3.8. Diese Abbildung stellt die Sicht des Laufzeitsystems auf ein repliziertes Objekt dar. Darin sind nur noch diejenigen Teilzustände enthalten, von denen es Kopien auf mehr als einem Rechenknoten gibt. Nur solche Teilzustände müssen bei der Wiederherstellung der Konsistenz vor Verlassen eines kollektiv synchronisierten Abschnitts berücksichtigt werden. Von allen übrigen Teilzuständen gibt es keine Kopien, die auf den neusten Stand gebracht werden müssten. Insgesamt ist also von dem replizierten Objekt nur eine Teilmenge seiner Teilzustände tatsächlich auf mehr als einem Rechenknoten angelegt. Manche Teilzustände sind also nicht repliziert. Daher heißt ein solches repliziertes Objekt auch *partiell repliziertes Objekt*. Anders als in Abbildung 3.8 vereinfacht dargestellt, ist es nicht zwingend, sondern eher die Ausnahme, dass ein tatsächlich replizierter Teilzustand auch Teil aller lokalen Replikate eines replizierten Objektes ist. Besteht ein repliziertes Objekt aus vielen lokalen Replikaten auf entsprechend vielen Rechenknoten, wird ein replizierter Teilzustand tatsächlich nur auf wenigen dieser Rechenknoten repliziert sein.

Funktionsweise

Bei der Entwicklung eines datenparallelen Algorithmus können replizierte Objekte aus der konzeptionellen Sicht der Vollreplikation betrachtet werden. Diese Sicht dient dazu, vollständig von Kommunikation *und* Verteilung abstrahieren zu können. Damit lässt sich auf objektorientierten irregulären Strukturen auf einfache Weise ein datenparalleler Algorithmus im BSP-Modell formulieren. Dazu muss lediglich der eigentliche Berechnungskern für ein einzelnes Datenelement codiert werden. Bei der anschließenden Parallelisierung und Verteilung werden die Datenelemente auf die Kontrollfäden der Anwendung aufgeteilt. Aus den Zugriffseigenschaften des Berechnungskerns ergibt sich die Sicht der lokalen Replikate. Jeder Kontrollfaden der Anwendung bearbeitet danach sein lokales Replikat der nur in der Vorstellung des Programmierers existierenden Gesamtstruktur. Tatsächlich ist diese aber auf keinem der beteiligten Rechenknoten als Ganzes angelegt. Jeder einzelne Kontrollfaden arbeitet dabei auf regulären Java-Objekten. Dabei ist unwichtig, ob ein einzelnes Objekt Teil von mehreren lokalen Replikaten und damit tatsächlich repliziert ist, oder ob dieses Objekt der einzige Repräsentant des entsprechenden Teilzustandes der gedachten Gesamtstruktur und damit nicht repliziert ist. Diese Sicht der lokalen Replikate macht die Verteilung explizit, abstrahiert aber weiterhin von der Kommunikation, die notwendig ist, um nach einer parallelen Modifikation die Konsistenz wiederherzustellen. Die Wiederherstellung einer konsistenten Sicht am Ende eines kollektiv synchronisierten Bereiches ist

alleinige Aufgabe des Laufzeitsystems. Nur dieses muss wissen, welche Teile eines lokalen Replikates auf mehreren Rechenknoten repräsentiert sind und daher einer Zustandsfortschreibung bedürfen.

Abgrenzung zu konventioneller Replikation

Kollektive Replikation ist ein neuer Beitrag, der über gewöhnliche Objektreplikation, wie sie in vielen anderen Projekten für verteilt paralleles Rechnen eingesetzt wird, hinausgeht. Bei normaler Objektreplikation wird ein repliziertes Objekt zu einem bestimmten Zeitpunkt entweder von einem Schreiber exklusiv modifiziert oder mehrere Leser greifen gleichzeitig auf seinen Zustand zu. Diese einfache Form der Replikation kann nur eingesetzt werden, um häufige verteilte Lesezugriffe zu optimieren, wenn die entsprechenden Daten nur selten modifiziert werden. Auch die hier vorgeschlagenen kollektiv replizierten Objekte können in einem Modus normaler Replikation verwendet werden. In diesem Modus kann entweder eine exklusive Schreibsperre oder eine gemeinsame Lesesperre gesetzt werden. Damit ist es möglich, auch ein kollektiv repliziertes Objekt in herkömmlicher Weise für die Verbesserung der Zugriffslokalität einzusetzen.

Indem kollektiv replizierte Objekte aber in beiden Modi betrieben werden können (entweder kollektiv zur Durchführung datenparalleler Operationen oder exklusiv zur Lokalitätsoptimierung von häufigen verteilten Lesezugriffen), stellen sie eine echte Erweiterung bekannter Replikationsstrategien dar und decken alle Fälle aus Abschnitt 3.5 ab, bei denen der Einsatz entfernter Objekte scheitert.

3.7 Integration der Konzepte

Dieser Abschnitt skizziert den Lösungsentwurf, der die in den vorangegangenen Abschnitten eingeführten Konzepte so in die verteilte Programmierumgebung integriert, dass sie mit minimalen Spracherweiterungen ausgedrückt werden können.

Alle im vorigen Abschnitt beschriebenen Teile einer verteilten Umgebung – ein gemeinsamer Objektraum mit Fernaufruf und kollektive Replikation – ließen sich ohne weiteres mit reinen Bibliotheken realisieren. Die Erfahrung zeigt allerdings, dass bei der Programmierung in einer verteilten Umgebung die Verwendung von Konzepten zum Umgang mit der Verteilung allgegenwärtig sind. Daher liegt es nahe, die Konzepte zum Umgang mit der Verteilung in die Sprache zu integrieren. Ansonsten verliert der Programmierer über immer gleichen Mustern von umständlichen Bibliotheksaufrufen den eigentlichen Zweck seines Programms aus dem Auge. Aus dem Betrachtungswinkel von Sprachen, die auf ihr spezielles Einsatzgebiet zurechtgeschneidert werden (*domain-specific languages*), ist hier die Domäne die verteilt parallele Anwendung. Durch geringfügige Erweiterungen der Basissprache können immer wiederkehrende Muster, die spezifisch für das Einsatzgebiet sind, automatisch erzeugt werden.

Die Spracherweiterungen für die hier entwickelte verteilte Programmierumgebung sind so gemacht, dass bekannte Konzepte aus der nichtverteilten Umgebung in die verteilte Umgebung übertragen werden. Dies erleichtert sowohl das Erlernen der Spracherweiterungen als auch das Portieren einer Anwendung für die verteilte Umgebung.

Die zentralen Bausteine von Java-Programmen sind Klassen und deren Instanzen. Beide sind auf eine einzige virtuelle Maschine beschränkt. Die Spracherweiterungen für die verteilte Umgebung ergänzen eine Klasse je nach Anwendungsfall um das Konzept *Fernaufruf* oder *Replikation*. Die Klasse wird dadurch entweder zu einer entfernten oder zu einer replizierten Klasse. Eine entfernte Klasse bietet in der verteilten Umgebung Zugriff von überall her auf sich selbst und auf ihre Instanzen. Eine replizierte Klasse ist überall in der verteilten Umgebung lokal in einem konsistenten Zustand verfügbar und erlaubt die Replikation ihrer Instanzen auf Knoten der verteilten Umgebung. An einem replizierten Objekt können Operationen in beiden Replikationsmodi (kollektiv oder exklusiv) abwechselnd durchgeführt werden.

3.7.1 Transparent entfernte Klassen

Die Idee bei transparent entfernten Klassen ist die Umkehrung der Sichtweise von RMI. Statt von lose gekoppelten virtuellen Maschinen wird von einer großen verteilten virtuellen Maschine ausgegangen, die sich aus regulären virtuellen Java-Maschinen auf den Knoten eines Rechnerbündels zusammensetzt. In einem verteilt parallelen Programm auf dieser Zielplattform sind entfernte Zugriffe daher nicht die Ausnahme, sondern die Regel. Da das Netzwerk integraler Bestandteil eines Parallelrechners ist, können dort Netzwerkverbindungen als genauso stabil angesehen werden wie der Speicherbus in einem herkömmlichen Rechner. In einem verteilt parallelen Programm ist es nicht sinnvoll, bei jedem der unzähligen entfernten Zugriffe eine Ausnahmebehandlung für den Fall anzugeben, dass dieser Fernaufruf wegen Netzwerkproblemen fehlschlägt. In einem Rechnerbündel ist daher ein größeres Maß an Transparenz möglich und notwendig, als dies im RMI-Modell der Fall ist.

Eine transparent entfernte Klasse im verteilten Objektraum des Rechnerbündels ist das Pendant zu einer regulären Java-Klasse in einer einzelnen virtuellen Maschine. Ihre Verwendung ist syntaktisch nicht von der Verwendung einer regulären Java-Klasse zu unterscheiden. Die entfernte Klasse selbst besitzt eine Laufzeitrepräsentation in der verteilten Umgebung. Die Klasse wird bei ihrer ersten Verwendung geladen und auf einem Rechenknoten initialisiert. Alle Zugriffe auf statische Anteile der entfernten Klasse werden danach transparent an diese Laufzeitrepräsentation weitergeleitet, unabhängig davon, auf welchem Rechenknoten sie stattfinden. Eine Instanz einer transparent entfernten Klasse wird wie gewohnt durch Aufruf eines Konstruktors über die `new`-Anweisung erzeugt. Die neue Instanz wird dabei auf einem Rechenknoten der verteilten Umgebung angelegt, der nicht notwendigerweise mit dem Rechenknoten übereinstimmt, auf dem die Erzeugung angestoßen wurde. Die Erzeugung liefert eine entfernte Referenz auf das neue Objekt. Entfernte Referenzen sind selbst transparent. Über sie kann auf das entfernte Objekt so wie auf ein reguläres Java-Objekt zugegriffen werden. Weder Methodenaufruf noch Zugriff auf eine Instanzvariable noch Synchronisation an dem entfernten Objekt sind von einem lokalen Zugriff unterscheidbar. Wenn das referenzierte entfernte Objekt auf einem anderen Rechenknoten als der Aufrufer liegt, wird transparent ein entsprechender Fernzugriff durchgeführt.

3.7.2 Transparent maschinenüberspannende Kontrollfäden

Durch die transparente Verteilung der entfernten Objekte auf die Rechenknoten wird ein Kontrollfaden, der Methoden dieser Objekte abarbeitet, unwillkürlich zu einem maschinenüberspannenden Kontrollfaden: In jedem Aufruf an einem entfernten Objekt, das auf einem anderen Rechenknoten als der Aufrufer liegt, wechselt das aktive Ende des Kontrollfadens zum Rechenknoten des Zielobjektes. Der Aufrufstapel eines solchen Kontrollfadens befindet sich folglich nicht mehr auf einem einzelnen Rechenknoten, sondern ist über die Rechenknoten der Umgebung verteilt.

Die Tatsache, dass Segmente eines Kontrollfadens auf unterschiedlichen Rechenknoten liegen, muss für die Anwendung ebenfalls transparent sein. Die Herausforderung besteht darin, für einen maschinenüberspannenden Kontrollfaden dieselbe Semantik bezüglich gehaltener Sperren und Unterbrechungssignale zu gewährleisten, wie der Programmierer dies von einem regulären Java-Kontrollfaden erwartet, der auf die virtuelle Maschine beschränkt ist, auf der er erzeugt und gestartet wurde.

3.7.3 Lokale Klassen

In der verteilten Umgebung können nicht alle Klassen in entfernte Klassen umgeschrieben werden. Erstens fällt für jedes entfernte Objekt ein gewisses Maß an Zusatzaufwand sowohl für seine Verwaltung als auch bei jedem Zugriff an. Da ein Programm typischerweise mit sehr vielen kleinen Objekten umgeht, kann dieser Zusatzaufwand nicht für jedes Objekt in Kauf genommen werden. Zweitens ist das Umschreiben in eine entfernte Klasse gar nicht immer möglich. Für Bibliotheken, die nicht im Quellcode vorliegen, könnte noch auf eine Bytecode-Transformation zurückgegriffen werden. Bei der Java-Basisbibliothek, bei der Verwendung von nativen Methoden oder bei eingebauten Typen wie Feldern sind dem allerdings Grenzen gesetzt. In der verteilten Umgebung sollen aber sowohl existierende Bibliotheken als auch alle Möglichkeiten der Java-Basisbibliothek unverändert weiterverwendet werden können. Nicht entfernte Klassen bleiben in der verteilten Umgebung nutzbar, sind jedoch auf ihre lokale virtuelle Maschine beschränkt. Daher werden reguläre Java-Klassen in der verteilten Umgebung als *lokale* Klassen (und ihre Instanzen als *lokale* Objekte) bezeichnet.

Durch die Koexistenz von lokalen und entfernten Klassen treten deren semantische Unterschiede bei gemischter Verwendung zutage. Entfernte Objekte werden in entfernten Methodenaufrufen als Referenz übergeben, wie das für lokale Objekte in regulären lokalen Methodenaufrufen gilt. Offensichtlich ist aber eine Übergabe als Referenz für lokale Objekte in entfernten Methodenaufrufen nicht möglich, da lokale Objekte nicht von außerhalb ihrer virtuellen Maschine referenzierbar sind. Aus Sicht der verteilten virtuellen Maschine handelt es sich bei lokalen Objekten um Werte, die als Kopie übergeben werden. In entfernten Methodenaufrufen werden lokale Objekte also wie Basistypen in der nichtverteilten virtuellen Maschine behandelt. Dort gibt es keine Referenzen auf Basistypen, weswegen sie in Methodenaufrufen ebenfalls als Kopien übergeben werden.

Aufgrund der geänderten Übergabesemantik für Instanzen lokaler Klassen beim Aufruf entfernter Methoden ist ausgerechnet die Verwendung der „originalen“ Java-Klassen in der verteilten virtuellen Maschine nicht mehr ganz transparent. Allerdings

stellt die verteilte Umgebung eine ortsunabhängige Semantik bereit: Die Semantik der Argumentübergabe (Übergabe als Kopie oder als Referenz) hängt nicht vom Ort des entfernten Zielobjektes ab. Ein Methodenaufruf an einem von seinem Typ her entfernten Objekt wird immer als entfernter Methodenaufruf behandelt, unabhängig davon, auf welcher virtuellen Maschine der verteilten Umgebung das Zielobjekt angelegt ist. Insbesondere, wenn das Zielobjekt sich auf derselben virtuellen Maschine angelegt ist, in welcher der Aufruf stattfindet, erfolgt dennoch eine Übergabe als Kopie. Durch diese verlässliche Übergabesemantik werden sowohl entfernte als auch lokale Methodenaufrufe transparent bezüglich der Platzierung der entfernten Objekte auf den Knoten der verteilten virtuellen Maschine. Die Semantik eines Programms für die verteilte virtuelle Maschine ist ortsunabhängig.

3.7.4 Transparent replizierte Klassen

Kollektive Replikation wird in der verteilten Programmierumgebung über transparent replizierte Klassen verwendet. Eine transparent replizierte Klasse kapselt replizierten Zustand und stellt diesen in ihren Replikaten für den lokalen Zugriff auf unterschiedlichen Rechenknoten zur Verfügung.

Transparente Erzeugung

Auch eine replizierte Klasse wird wie eine entfernte Klasse genau einmal in der verteilten Umgebung geladen, das heißt, ihr statischer Initialisierer wird einmal ausgeführt. Danach wird sie aber auf jeden Knoten repliziert und ist dort lokal verfügbar. Anders als bei ihren Instanzen existiert von einer replizierten Klasse ein lokales Replikat auf allen Knoten der verteilten Umgebung. Allerdings können diese lokalen Replikate wie bei ihren Instanzen partiell sein, d.h. dass nicht alle Teilzustände auf allen Rechenknoten repliziert sein müssen. Die Instanzierung einer replizierten Klasse erzeugt ein neues repliziertes Objekt. Dabei wird der Konstruktor der Klasse wie gewohnt genau einmal ausgeführt. Erst im Anschluss werden lokale Replikate auf mehreren oder allen Knoten der verteilten Maschine angelegt. Der Konstruktor einer replizierten Klasse liefert als Ergebnis eine Referenz auf das lokale Replikat des neu initialisierten replizierten Objektes. In der Regel ist es nicht sinnvoll, alle Teilzustände eines replizierten Objektes direkt in seinem Konstruktor anzulegen, da diese sonst alle auf dem Rechenknoten angelegt würden, auf dem das replizierte Objekt erzeugt wird. Stattdessen sollte im Konstruktor lediglich ein Gerüst aufgebaut werden, das in einer anschließenden kollektiven Operation parallel gefüllt wird. Auf diese Weise können replizierte Objekte erzeugt werden, deren gesamter Zustand nie auf einen einzelnen Rechenknoten passen würde.

Direkter Zugriff

Nach seiner Erzeugung geschieht der Zugriff auf das replizierte Objekt immer lokal. Anders als bei entfernten Objekten, die nach ihrer Erzeugung von anderen Knoten über eine entfernte Referenz angesprochen werden, liefert die Erzeugung eines replizierten Objektes eine normale Java-Referenz auf das lokale Replikat zurück. Damit

kann ein repliziertes Objekt ohne Mehraufwand wie jedes andere Objekt einer lokalen Klasse benutzt werden. Insbesondere ist kein Stellvertreterobjekt notwendig, um ein Replikat vor direkten Zugriffen abzuschirmen. Durch den Direktzugriff kann ein repliziertes Objekt wie ein lokales Objekt verwendet werden. Unterschiede werden nur bei der Synchronisation und der Übergabe als Argumente in entfernten Methodenaufrufen deutlich.

Übergabe als Argument in Fernaufrufen

Ein lokales Objekt würde bei der Parameterübergabe in einem entfernten Methodenaufruf als Wert behandelt und kopiert. Anders verhält es sich bei einem replizierten Objekt: Der entfernte Methodenaufruf transportiert hier keine Kopie des lokalen Replikats auf den entfernten Knoten, sondern nur einen Identifikator für das replizierte Objekt. Bevor der entfernte Methodenaufruf beginnt, wird dieser Identifikator durch das bestehende Replikat auf dem Zielknoten ersetzt. Über diesen Mechanismus kann ein repliziertes Objekt wie ein entferntes Objekt als Referenz in entfernten Methodenaufrufen übergeben werden. Dazu muss ein Replikat eines replizierten Objektes auf beiden Seiten des entfernten Methodenaufrufs vorrätig sein. Diese beiden Replikate sind die Repräsentanten des replizierten Objektes auf den Knoten. Referenzübergabe wird realisiert, indem das Replikat auf dem aufrufenden Knoten bei der Übergabe in entfernten Methodenaufrufen durch das entsprechende Replikat auf dem Zielknoten ersetzt wird. Den Unterschied zu der Übergabe als Kopie bei lokalen Objekten erkennt man, wenn man sich mehrere entfernte Methodenaufrufe in Folge vorstellt, die immer dasselbe Objekt als Argument übergeben. Handelt es sich bei dem Argument um ein lokales Objekt, wird dies in jedem dieser Methodenaufrufe kopiert. Auf der Empfängerseite kommt folglich in jedem Methodenaufruf eine neue Kopie (also ein anderes Objekt) an. Handelt es sich dagegen um ein repliziertes Objekt, so wird in jedem Methodenaufruf das Argument auf Empfängerseite zum dortigen Repräsentanten des replizierten Objektes (dem lokalen Replikat) aufgelöst. In allen Methodenaufrufen kommt folglich auf Empfängerseite dasselbe Objekt (eine Referenz auf das lokale Replikat) an.

Koordination

Bei replizierten Objekten wird zwischen lesenden, schreibenden und kollektiven Zugriffen unterschieden. Die Art des Zugriffs muss durch das Setzen einer entsprechenden Sperre (zum Lesen, Schreiben oder zur Durchführung einer datenparallelen kollektiven Operation) an dem replizierten Objekt angekündigt werden. Da es in Java nur Synchronisation als einzige Ausdrucksmöglichkeit für das Setzen von Sperren gibt, muss an dieser Stelle eine Erweiterung vorgenommen werden. Replizierte Objekte sind daher bezüglich Synchronisation nicht vollkommen transparent. Für replizierte Klassen werden drei unterschiedliche Synchronisationsprimitive eingeführt. Mit *exklusiver Synchronisation* lässt sich ein repliziertes Objekt für die exklusive Modifikation in einem einzigen Kontrollfaden sperren. Mit *gemeinsamer Synchronisation* können mehrere Kontrollfäden unabhängig voneinander gemeinsamen Lesezugriff auf ein repliziertes Objekt erhalten. Schließlich können sich über *kollektive Synchronisation*

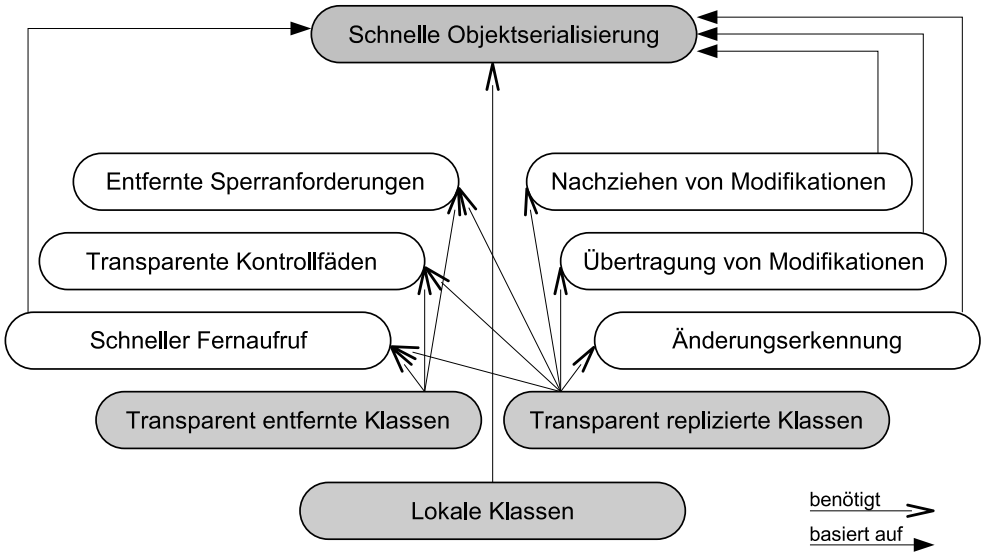


Abbildung 3.9: Zusammenspiel der Komponenten in der verteilten Umgebung.

Kontrollfäden zu einem Kollektiv zusammenfinden, um kooperativ eine datenparallele Operation auf dem replizierten Zustand durchzuführen.

3.8 Zusammenfassung

Transparent entfernte und transparent replizierte Klassen bilden die Grundbausteine der in der vorliegenden Arbeit realisierten Programmierumgebung für Rechnerbündel. Daneben können reguläre Java-Klassen als lokale Klassen unverändert weiterbenutzt werden. Abbildung 3.9 zeigt das Zusammenspiel der Komponenten, die zur Realisierung notwendig sind. Diese Realisierung wird in den Folgekapiteln besprochen. Aufgrund der starken Abhängigkeiten wird die Reihenfolge der Darstellung so gewählt, dass zuerst die Basistechnologien und anschließend die höherwertigen Komponenten besprochen werden. Das Folgekapitel 4 führt die schnelle Objektserialisierung ein, auf der alle folgenden Techniken aufsetzen. Kapitel 5 beschreibt anschließend den schnellen Fernaufruf und die Techniken für die Transparenz von entfernten Klassen, Kontrollfäden und entfernten Sperren. In Kapitel 6 wird schließlich die Realisierung von kollektiver Replikation über transparent entfernte Objekte erklärt. Die dafür notwendigen Erweiterungen des Fernaufrufs und der schnellen Objektserialisierung werden ebenfalls dort eingeführt.

Kapitel 4

Schnelle Objektserialisierung

Die Effizienz einer verteilten objektorientierten Umgebung hängt entscheidend vom Reibungsverlust ab, der auftritt, wenn eine Datenstruktur von einem Rechner auf einen anderen kopiert werden muss. In einem objektorientierten Programm bestehen solche Datenstrukturen im allgemeinen aus Objektgraphen. Da ein Objektgraph in der Regel nicht zusammenhängend im Speicher eines Rechners abgelegt ist, muss dieser zur Übertragung in eine Netzwerkrepräsentation übersetzt werden. Auf der Empfängerseite wird die empfangene Information interpretiert und wieder in einen Objektgraphen zurückverwandelt. Diesen Vorgang nennt man Serialisierung auf der Senderseite und Deserialisierung auf der Empfängerseite.

Während eines entfernten Methodenaufrufs wird Serialisierung und Deserialisierung zur Argumentübergabe und Rückübermittlung des Resultats verwendet. Für einen effizienten entfernten Methodenaufruf ist daher die Geschwindigkeit der Objektserialisierung entscheidend. In einem über RMI abgewickelten Fernaufruf wird ca. ein Drittel der Zeit für die Objektserialisierung verbraucht [68, 84]. Systeme zur Objektserialisierung müssen mit den folgenden Problemen umgehen:

Heterogenität: Primitive Datentypen sind auf unterschiedlichen Hardwarearchitekturen verschieden repräsentiert. Soll eine Kommunikation über Architekturgrenzen hinweg möglich sein, muss eine Übersetzung in ein architekturunabhängiges Format durchgeführt werden.

Da die virtuelle Java-Maschine von solchen Details der Ausführungsplattform ohnehin abstrahiert, muss eine verteilte Java-Umgebung keinen Mehraufwand leisten, um über Architekturgrenzen hinweg zu kommunizieren. Innerhalb der virtuellen Maschine ist es gar nicht möglich eine Netzwerkrepräsentation zu erzeugen, die von der Hardwarearchitektur des Sende-Rechners abhängt.

Referenzen: Eine objektorientierte Datenstruktur besteht aus vielen Objekten, die einander wechselseitig referenzieren. Die Serialisierung wandert diesen Referenzgraphen ab und überträgt alle erreichbaren Teilobjekte. Die Empfängerseite muss daraufhin in der Lage sein, einen äquivalenten Graphen zu rekonstruieren. Mithilfe der Referenzen wird von der relativen Lage der Teildaten im Speicher des Quell- und Zielrechners abstrahiert. Während der Serialisierung werden Daten aus nicht zusammenhängenden Speicherbereichen übertragen, deren relative Lage erst zur Laufzeit des Pro-

gramms feststeht. Darüberhinaus kann sich die relative Lage bei mehrfacher Übertragung derselben Struktur beispielsweise durch kopierende Speicherbereiniger ändern.

Zyklen: Die Referenzverfolgung während der Serialisierung muss Zyklen im zu übertragenden Graphen erkennen und in der Netzwerkrepräsentation codieren. Nur so ist es möglich, allgemeine Objektstrukturen zu übertragen. Die Zyklenerkennung ist ein Hauptverursacher für Aufwand während der Objektserialisierung. Daher sind Strategien besonders interessant, die es erlauben, auf Zyklenerkennung unter gewissen Bedingungen zu verzichten.

Selbstbeschreibungsfähigkeit: Zur vollständigen Beschreibung des Zustandes eines Objektes ist neben den Belegungen aller seiner Instanzvariablen auch der Typ des Objektes relevant. Ohne Kenntnis des Typs ist es dem Empfänger nicht möglich, ein gleichwertiges Objekt zu konstruieren. Zu diesem Zweck muss der Sender ein gewisses Maß an Typinformation in den Datenstrom eincodieren, welche die übertragenen Daten beschreibt. Je nachdem, wieviel gemeinsames Wissen auf Sender- und Empfängerseite vorausgesetzt werden kann, muss das verwendete Format mehr oder weniger Fähigkeit zur Selbstbeschreibung besitzen. Ein besonders hohes Maß an Selbstbeschreibungsfähigkeit ist notwendig, wenn Daten durch persistente Speicherung in der Zeit transportiert werden sollen. Bei der Kommunikation von Daten – dem Transport zwischen verschiedenen Orten – hängt die erforderliche Selbstbeschreibungsfähigkeit vom Grad der Kopplung der beteiligten Systeme ab. In eng gekoppelten Systemen und in Abwesenheit von Polymorphie kann im Extremfall ganz auf Selbstbeschreibung verzichtet werden, wenn zwischen Sender und Empfänger Einvernehmen über die kommunizierten Datentypen herrscht. In lose gekoppelten Systemen muss Softwareversionierung berücksichtigt werden. Unterschiedliche Versionen einer Software müssen Daten austauschen können, obwohl ihre Klassen in unterschiedlichen Versionen mit unterschiedlichen Mengen von Instanzvariablen vorliegen. In einem solchen Anwendungsszenario ist ein besonders hohes Maß an Selbstbeschreibung beim verwendeten Format notwendig. Dies ist insbesondere auch dann der Fall, wenn ein Serialisierungsformat allen Anwendungsfällen gerecht werden will. Selbstbeschreibung verursacht aber einen hohen Aufwand beim Schreiben und Lesen des Formates. Ein Serialisierungsformat, das auf einen bestimmten Anwendungsfall zurechtgeschnitten ist, kann dagegen die Codierung unnötiger Selbstbeschreibungsinformationen vermeiden.

Kopieroperationen: Netzwerkprotokolle sind dann besonders effizient, wenn sie ohne Kopieren der Daten auskommen (*zero-copy*) [19]. In diesem Fall führt weder die Kommunikationsbibliothek noch das Betriebssystem Kopieroperationen durch, während sich die Daten auf dem Weg von der Anwendungsschnittstelle zur Übergabe an die Netzwerkhardware befinden. Beim Versenden einer Graphstruktur ist aber klar, dass *eine* Kopieroperation notwendig ist, welche die unzusammenhängend liegenden verzeigerten Daten vor dem Versenden in einem zusammenhängenden Puffer sammelt und dabei kopiert. Allerdings sollte der Serialisierungsmechanismus so ausgelegt sein, dass er mit *genau einer* Kopieroperation auf Sender- und Empfängerseite auskommt.

Klonen: Zur Realisierung transparent entfernter Objekte (vgl. Kapitel 5.5) ist statt der Kopie eines Graphen auf einen anderen Rechner unter gewissen Umständen, die in Kapitel 5.2.4 beschrieben werden, die Erzeugung einer lokalen tiefen Kopie notwendig. Die Erzeugung eines lokalen Klons sollte durch den Serialisierungsmechanismus direkt ohne den Umweg über Serialisierung/Deserialisierung unterstützt werden. Zwar kann eine tiefe Kopie über die Operationsfolge Serialisieren und lokales Deserialisieren ohne Versenden der Daten erzeugt werden, aber dieser Ansatz ist äußerst ineffizient, da zwei Kopieroperationen stattfinden (die eine in einen Datenpuffer bei der Serialisierung und die andere in den Ziel-Objektgraphen). Außerdem wird bei der Serialisierung aufwendig in eine Netzwerkrepräsentation übersetzt, die sofort anschließend wieder in einen Objektgraphen umgewandelt werden muss. Daher sollte der Serialisierungsmechanismus auch das direkte tiefe Kopieren eines Objektgraphen unterstützen.

Begrenzter Stapelspeicher: In einer mehrfädigen Anwendung ist die Größe des verfügbaren Stapelspeichers pro Kontrollfaden notwendigerweise begrenzt. Findet während der Serialisierung ein rekursiver Abstieg zur Verfolgung der Referenzen statt, ist bei einer großen, stark verzeigerten Struktur[1] der verfügbare Stapelspeicher schnell erschöpft. In diesem Fall hat die maximale Rekursionstiefe während der Serialisierung dieselbe Größenordnung wie die Anzahl der Objekte im zu serialisierenden Graphen. Um auch große, stark verzeigerte Datenstrukturen übertragen zu können, darf der Serialisierungsmechanismus nicht rekursiv formuliert sein.

4.1 Die Java-Objektserialisierung

Die Java-Objektserialisierung [74] ist Bestandteil der Java-Basisbibliothek. Sie ist einerseits konzipiert für die Übertragung von Objekten zwischen virtuellen Java-Maschinen und andererseits für die persistente Speicherung von Objektgraphen auf Datenträgern. Das ebenfalls zur Basisbibliothek gehörende Java-RMI [71] setzt die Java-Objektserialisierung ein, um Parameter an einen entfernten Methodenaufruf als Kopie zu übertragen, wenn es sich bei den betreffenden Objekten nicht selbst um entfernte Objekte handelt.

4.1.1 Eigenschaften

Die Java-Objektserialisierung kann mit Graphen von verzeigerten Objekten umgehen, führt eine Zyklenerkennung durch und rekonstruiert ein exaktes Abbild des Originalgraphen auf der Empfängerseite. Neben den Daten der übertragenen Objekte codiert die Java-Objektserialisierung zusätzlich die vollständige Typinformation in das Übertragungsformat. Diese ausführliche Typinformation erlaubt es, die Java-Objektserialisierung auch für die persistente Speicherung von Daten oder für die Datenübertragung zwischen nicht gekoppelten Systemen zu verwenden. Obwohl es sich beim Übertragungsformat der Java-Objektserialisierung um ein Binärformat handelt, ist es bezüg-

[1] Von jedem Objekt einer stark verzeigerten Struktur ist jedes andere über eine Referenzenkette zu erreichen.

lich der übertragenen Daten selbstbeschreibend. So ist es möglich, Daten aus einem mit der Java-Objektserialisierung geschriebenen Strom auch dann noch zu lesen, wenn auf der Empfängerseite die zugehörigen Klassen in einer neueren Version vorrätig sind. Aufgrund dieser Eigenschaft ist das Übertragungsformat der Java-Objektserialisierung mit einem auf XML basierenden Format vergleichbar, obwohl es sich um ein Binärformat handelt.

Benutzung

Die Java-Objektserialisierung kann, ohne dass dies Mehraufwand beim Programmieren bedeutet, benutzt werden. Eine Anwendungsklasse muss zur Markierung lediglich die leere Schnittstelle `java.io.Serializable` implementieren. Indem eine Klasse diese Markierungsschnittstelle implementiert, erklärt sie sich damit einverstanden, dass ihre Instanzen von den Serialisierungsklassen ausgelesen und in einen Bytestrom konvertiert werden dürfen. Um diese Schnittstelle zu implementieren, muss keine Funktionalität bereitgestellt werden. Fehlt die Markierung, dann darf der Zustand des zugehörigen Objektes nicht ausgelesen werden oder der Zustand ist nicht persistent speicherbar oder auf eine andere virtuelle Maschine übertragbar (entweder aus Gründern der Sicherheit oder weil das Objekt nur in seiner virtuellen Maschine Gültigkeit besitzt).

Funktionsweise

Führt eine Klasse die Serialisierbarkeitsmarkierung in ihrer Liste der implementierten Schnittstellen auf, geschieht das Auslesen der Daten auf Senderseite und das Erzeugen des kopierten Objektes sowie das Befüllen seiner Instanzvariablen auf Empfängerseite über dynamische Typintrospektion allein durch die Serialisierungsklassen ohne Zutun des Anwenders. Optional kann eine Klasse aber eigene Serialisierungsfunktionalität bereitstellen. Das ist dann notwendig, wenn entweder Änderungen am Zustand vor der Serialisierung durchzuführen sind oder ein abgeänderter Zustand übertragen werden soll (beispielsweise zur Kompression oder Verschlüsselung). Der Serialisierungsmechanismus verwendet benutzerdefinierte Methoden mit spezieller Signatur, falls diese vorhanden sind (`writeObject()` bzw. `writeExternal()`).

4.1.2 Einschränkungen

Eine Reihe von Eigenschaften der Java-Objektserialisierung schränken ihre Verwendbarkeit für die Datenübertragung in einer verteilten Rechnerbündelumgebung stark ein.

Tiefe Kopie

Eine tiefe Kopie eines ganzen Objektgraphen, bei der alle von einem Objekt aus erreichbaren Objekte ebenfalls kopiert werden, lässt sich mit Hilfe der Java-Objektserialisierung nur über den Umweg des Schreibens in einen temporären Puffer und des anschließenden Wiedereinlesens der Daten realisieren. Daraus resultieren eine unnötige Kopieroperation der Daten, ein erhöhter Speicherplatzverbrauch und ein aufwendiger aber überflüssiger Codierungs- und Decodierungsschritt.

Rekursion

Die Java-Objektserialisierung verwendet einen rekursiven Abstieg durch den Objektgraphen beim Senden und Empfangen des Datenstroms. Aufgrund dessen ist es nicht möglich, eine große, stark verzeigerte Struktur mit der Java-Objektserialisierung zu übermitteln.

Effizienz

Die Laufzeit der Java-Objektserialisierung ist für die Datenübertragung in einer verteilten Umgebung für Rechnerbündel inakzeptabel langsam. Dies liegt hauptsächlich an ihrem aufwendig zu erzeugenden selbstbeschreibenden Übertragungsformat und der relativ teuren dynamischen Typintrospektion für das Auslesen und Zurückschreiben der Daten.

Sicherheit

Interessanterweise kann die Laufzeit auch durch Anbieten von spezialisierten Versenderoutinen pro Klasse nicht wesentlich verbessert werden. Dies macht zwar die dynamische Typintrospektion überflüssig, setzt aber einen Sicherheitsmechanismus in Kraft, der mit zusätzlichen Laufzeitkosten zu Buche schlägt: Die Java-Objektserialisierung garantiert die wechselseitige Abschirmung von Objekten, welche nacheinander in demselben Strom übertragen werden. Indem die Serialisierung darüber Buch führt, welche Daten von der Senderoutine eines Objektes geschrieben wurden, und nur genau diese an die zugehörige Empfangsroutine ausliefert, wird verhindert, dass fehlerhafte Versende-/Empfangsroutinenpaare auf einen Zustand zugreifen können, der zu später im Strom übertragenen Objekten gehört. (Eine „bösartige“ Klasse könnte versuchen, in ihrer Leseroutine mehr zu lesen als ihre Schreibroutine ausgegeben hat, um damit den Zustand anderer im selben Strom übertragener Objekte auszuspähen oder den Zustand von nachfolgenden Objekten zu manipulieren, indem die Schreibroutine mehr ausgibt, als die zugehörige Leseroutine liest). Die wechselseitige Abschirmung von Objekten während der Übertragung garantiert ein definiertes Verhalten, selbst wenn nicht vertrauenswürdige Objekte mitübetragen werden. Diese Sicherheitsgarantien sind aber nur zum Preis einer Extra-Verpackung für Daten zu erreichen, welche von Anwendungsklassen in den Serialisierungsstrom eingemischt werden. Dieses Zusatzprotokoll wiegt die eingesparte Typintrospektion wieder auf, so dass in der Java-Objektserialisierung anwendungsspezifische Versenderoutinen nur für Zusatzfunktionalität, nicht aber für verbesserte Effizienz zu gebrauchen sind.

Geschwindigkeit

Aufgrund der rasanten Entwicklung der Laufzeitübersetzer in den vergangenen Jahren ist die Laufzeit der Java-Objektserialisierung, absolut gesehen, besser geworden. Aber wie im weiteren Verlauf des vorliegenden Kapitels gezeigt wird, profitiert die hier vorgestellte optimierte Objektserialisierung ebenfalls von der verbesserten Laufzeitumgebung, so dass sich ihr relativer Geschwindigkeitsvorteil gegenüber den Messungen in [84] sogar noch steigern lässt.

Die folgenden Abschnitte beschreiben die Optimierungstechniken, die es erlauben, eine schelle Objektserialisierung in reinem Java und ohne Eingriff in die Java-Basisbibliotheken zu implementieren. Ein erster in [84] beschriebener Ansatz erforderte die Modifikation und das Neuübersetzen von Java-Basisklassen.

4.2 Optimierungen am Serialisierungsprotokoll

Ein Großteil der Arbeit wird bei der Java-Objektserialisierung in die Erzeugung des sicheren und für die persistente Speicherung geeigneten Formates investiert. Die hier entworfene Serialisierung ist auf die reine Datenübertragung in eng gekoppelten Rechnerbündeln spezialisiert und kann daher den Mehraufwand der Java-Objektserialisierung einsparen. Die Optimierungsidee ist die Reduktion der Selbstbeschreibungsfähigkeit des Formats auf ein Minimum, die Vermeidung von Redundanzen bei mehreren Übertragungen in Folge und der völlige Verzicht auf dynamische Typintrospektion durch pro Klasse generierte Sende- und Empfangsroutinen.

4.2.1 Schlanke Typcodierung und Wiederverwendung

Wenn die Serialisierung von einer gemeinsamen Codebasis auf Sender- und Empfängerseite ausgeht, genügt es auf Empfängerseite, den Typ (die Klasse) jedes übertragenen Objektes zu kennen. Mit dieser Information lässt sich ein Objekt desselben Typs auf der Empfängerseite erzeugen, um es anschließend mit den übertragenen Zustandsdaten zu füllen. Beim ersten Kontakt ist die kompakteste Repräsentation des Typs eines zu übertragenden Objektes der Klassenname. Über den Klassennamen kann die Empfängerseite die zugehörige Klasse laden und neue Objekte erzeugen.

Allerdings ist die Übertragung des Klassennamens als Zeichenkette und das Nachschlagen des Klassenobjektes über seinen Namen in einer Tabelle für jedes übertragene Objekt zu teuer. Daher wird der Klassenname nur für das jeweils erste Objekt einer Klasse verwendet, um ein gemeinsames Typalphabet zwischen Sender und Empfänger zu etablieren. Während der Übertragung des ersten Objektes einer Klasse weisen Sender und Empfänger dieser Klasse eine eindeutige fortlaufende Nummer zu, über welche diese Klasse in allen weiteren Übertragungen angesprochen werden kann.

Nach der erstmaligen Verwendung einer Klasse für die Serialisierung bzw. Deserialisierung reduziert sich die übertragene Typinformation pro Objekt auf die vereinbarte Nummer. Dieser Mechanismus funktioniert ähnlich wie die Zyklenerkennung für Objektreferenzen. Wird die Serialisierung zur Übertragung von Argumenten in entfernten Methodenaufrufen verwendet, kann das in vorangegangenen Aufrufen etablierte Typalphabet (anders als die übertragenen Objektreferenzen) für folgende Aufrufe weiterverwendet werden. Dies macht die erneute Übertragung von Typinformation überflüssig.

Zur Realisierung werden getrennte Tabellen für die Zyklenerkennung in Objektgraphen und für den Aufbau eines Typalphabets zwischen Paaren aus Serialisierer und Deserialisierer verwendet. Dadurch kann die Tabelle zur Zyklenerkennung vor jedem entfernten Methodenaufruf gelöscht werden, während das gemeinsame Typalphabet erhalten bleibt. Die Java-Serialisierung verwendet nur eine einzige Tabelle zum

Auflösen von Objektreferenzen und Typbezeichnern, weswegen die komplette Typinformation für jeden entfernten Methodenaufruf erneut übertragen werden muss.

4.2.2 Spezialisierte Versenderoutinen

Die optimierte Objektserialisierung verzichtet völlig auf dynamische Typintrospektion. Stattdessen müssen die Klassen von Objekten, die mit der optimierten Objektserialisierung übertragen werden sollen, spezialisierte Methoden für alle Aufgaben anbieten, die in der Java-Objektserialisierung durch dynamische Typintrospektion erledigt werden. Dazu gehören Methoden, die den Zustand des Objektes auslesen und in den Strom schreiben, und Methoden, welche die Daten des Objektes aus dem Strom lesen und die Instanzvariablen des neuen Objektes damit befüllen. Diese Methoden sind mit den optionalen Methoden `writeObjekt()` und `readObjekt()` bei der Java-Objektserialisierung vergleichbar. Darüber hinaus werden Routinen benötigt, die ein neues Objekt auf Empfängerseite erzeugen, und andere, die für die schnelle tiefe Kopie eines Objektgraphen benutzt werden.

Wie im Abschnitt 4.6 dargestellt werden wird, lassen sich alle für die optimierte Objektserialisierung benötigten Routinen automatisch erzeugen, so dass die Optimierungen keinen Verlust an Komfort für den Programmierer zur Folge haben. Verwendet man allerdings Bibliotheksklassen, für die eine nachträgliche Erzeugung der notwendigen Routinen nicht möglich ist, wäre es wünschenswert, auf den Mechanismus der Java-Objektserialisierung zurückzufallen, um diese Klassen – wenn auch weniger schnell – dennoch weiterverwenden zu können. Dies ist über die Einbettung eines mit der Java-Objektserialisierung erzeugten Datenstromes in den Datenstrom der hier beschriebenen Serialisierung möglich. Dazu wird ein unverändertes Java-Serialisierungsobjekt verwendet, an das die Aufgabe der Serialisierung in einem solchen Fall delegiert wird. Damit erreicht die schnelle Objektserialisierung volle Kompatibilität zur Java-Serialisierung. Ein Geschwindigkeitsgewinn ist allerdings nur dann zu erwarten, wenn die Klassen der übertragenen Objekte spezialisierte Versenderoutinen anbieten.

4.2.3 Eingeschränkte Sicherheitsgarantien

Da die optimierte Objektserialisierung auf spezialisierte Senderoutinen pro Klasse setzt, um einen Geschwindigkeitsgewinn zu erreichen, dürfen diese nicht wie bei der Java-Serialisierung zu Mehraufwand durch erzwungene wechselseitige Abschirmung führen (vgl. Abschnitt 4.1.2). In der optimierten Objektserialisierung gehört der von spezialisierten Senderoutinen geschriebene Datenstrom ebenso zum Protokoll, wie der von der Java-Objektserialisierung mittels dynamischer Typintrospektion erzeugte Datenstrom. Da die optimierte Objektserialisierung zur Kommunikation einer verteilten Anwendung innerhalb eines Rechnerbündels verwendet werden soll, ist das Risiko gering, dass eine bösartige Senderoutine den Datenstrom so verändert, dass eine Empfangsroutine Daten, die zu einem anderen Objekt gehören, lesen kann.

Aus software-technischer Sicht wäre es allerding fatal, wenn durch mangelnde wechselseitige Abschirmung der Objekte während des Transports, eine unbemerkt fehlerhafte Senderoutine einen Programmfehler an einer entfernten Stelle im Programm

verursachen könnte. Ein solcher Fehler wäre extrem schwierig zu finden, da seine Ursache im Programmfluss sehr weit entfernt liegen kann. Der Effekt ist vergleichbar mit einem unbemerkt gebliebenen illegalen Speicherzugriff in einer unsicheren Programmiersprache. Dieses Problem der Fehleranfälligkeit wird in der optimierten Objektserialisierung durch automatische Erzeugung der notwendigen Sende- und Empfangsroutinen vermieden. Dadurch wird sichergestellt, dass eine Empfangsroutine immer zu ihrer Senderoutine passt und nur genau diejenigen Daten aus dem Strom konsumiert, die von ihrer zugehörigen Senderoutine erzeugt wurden. Dadurch wird zwar nicht vermieden, dass durch mutwillige Fälschung einer Routine das Stromformat korrumpiert werden könnte, aber es kann ausgeschlossen werden, dass sich unbemerkte Fehler in die Sende-/Empfangsroutinen einschleichen.

4.2.4 Transportschicht-Anbindung

Die Objektserialisierung steht in einem System für entfernten Methodenaufruf an der Schnittstelle zur Transportschicht. Sie definiert die Übersetzung von Objekten der Programmiersprache in eine für die Netzwerkübertragung verwendbare Darstellung. Um die Objektserialisierung flexibel verwenden zu können, darf sie nicht auf eine bestimmte Transporttechnologie zugeschnitten sein.

Strom- vs. Paketorientierung

Die Transporttechnologie kann wahlweise eine strom- oder paketorientierte Schnittstelle für die Datenübertragung anbieten. Eine stromorientierte Schnittstelle geht davon aus, dass man Daten in beliebiger Stückelung senden und empfangen kann. Durch die Stromabstraktion ist es irrelevant, ob sich die Stückelung, in welcher der Sender die Daten schickt, von der Stückelung unterscheidet, in welcher der Empfänger die Daten anfordert. Eine paketorientierte Schnittstelle geht davon aus, dass Daten in größeren Paketen verschickt und in genau derselben Stückelung auch wieder empfangen werden. Schickt der Sender ein Paket einer bestimmten Größe, dann muss der Empfänger bereit sein, ein Paket von mindestens dieser Größe in Empfang zu nehmen. Die Java-Objektserialisierung geht von einer Stromabstraktion aus, während die hier vorgestellte optimierte Objektserialisierung mit beiden Arten von Schnittstellen zusammenarbeiten kann.

Vermeidung von Kopieroperationen

Viele Protokolle für Hochgeschwindigkeitskommunikation bieten ausschließlich eine paketorientierte Schnittstelle an, um Kopieroperationen beim Anpassen der Stückelungsgröße der Daten zu vermeiden. Wird eine solche Transporttechnologie verwendet, ist es entscheidend, dass kein Reibungsverlust durch Schnittstellenanpassungen zwischen Objektserialisierung und Transporttechnologie auftritt. Ein solcher Reibungsverlust tritt bei der Java-Serialisierung auf, da diese ausschließlich eine stromorientierte Schnittstelle anbietet. Die Adaption einer paketorientierten an eine stromorientierte Schnittstelle ist aber nur über Pufferung möglich, und das bedingt eine zusätzliche Kopieroperation der Daten. Die optimierte Objektserialisierung arbeitet sowohl mit

einer strom- als auch mit einer paketorientierten Transporttechnologie zusammen, indem sich die Pufferungsschicht austauschen lässt. Bei Verwendung einer paketorientierten Transporttechnologie kann mit einer spezialisierten Pufferungsschicht direkt in die Netzwerkpuffer der Transporttechnologie serialisiert und damit eine Übertragung von Objektgraphen mit genau einer Kopieroperation auf Sende- und Empfangsseite ermöglicht werden.

4.2.5 Nichtrekursives Serialisieren

Objektserialisierung ist ein Graph-Algorithmus und damit nach erstem Anschein ein inhärent rekursives Problem. Um den kompletten Zustand eines Objektes zu serialisieren, müssen sowohl seine primitivwertigen Instanzvariablen als auch alle seine Referenzen auf weitere Objekte übertragen werden. Da eine Referenz auf ein Objekt nur übertragen werden kann, wenn auch das referenzierte Objekt selbst übertragen wird, führt dies direkt zu einer Rekursion: An der Stelle, an welcher während der Serialisierung eines Objektes eine Referenz auf ein weiteres Objekt geschrieben werden soll, wird die Serialisierungsfunktion rekursiv mit diesem Objekt aufgerufen. Wurde das referenzierte Objekt bereits früher übermittelt, wird nur die Referenz wiederholt und die Rekursion bricht ab. Ansonsten unterbricht die Serialisierung des referenzierten Objektes die Serialisierung des referenzierenden Objektes rekursiv. Der resultierende Serialisierungsstrom entspricht einer „in-order“ Traversierung des serialisierten Objektgraphen.

Eine direkte Umsetzung des inhärent rekursiven Serialisierungsalgorithmus führt zu einer Implementierung, die unbrauchbar ist, um große, stark verzeigerte Strukturen zu übertragen. Schon die Serialisierung einer langen verketteten Liste lässt die Rekursionstiefe linear mit der Läge der Liste anwachsen und sprengt damit schnell jede vorgegebene Größe des Stapelspeichers. Die Java-Objektserialisierung leidet an genau diesem Problem. Für parallele Hochleistungsanwendungen darf aber nicht die Größe des Stapelspeichers, sondern nur die Menge des verfügbaren Hauptspeichers eine Beschränkung für die Größe ihrer Datenstrukturen sein.

Durch Offenlegen des Stapelspeichers lässt sich ein beliebiger rekursiver Algorithmus (also auch ein nicht endrekursiver) in einen iterativen Algorithmus umwandeln. Dabei wird der Zustand, der bei der Rekursion implizit auf dem Programmstapel zwischengespeichert wird, in eine separate Datenstruktur gerettet. Die Größe dieser Hilfsdatenstruktur ist nicht mehr durch den Stapelspeicher, sondern nur noch durch den verfügbaren Hauptspeicher beschränkt.[2]

Angewendet auf die Serialisierung würde dieses Vorgehen bedeuten, dass ein Objekt, das erst zum Teil ausgelesen, bzw. mit Daten gefüllt ist, in einer Hilfsdatenstruktur zwischengespeichert wird, solange diejenigen Objekte bearbeitet werden, die von dem Ausgangsobjekt referenziert werden. Dieses Vorgehen ist aus zwei Gründen so nicht durchführbar: Zum einen findet die Serialisierung des Objektzustandes in einer spezialisierten Methode des Objektes selbst statt. Diese Methode müsste an der Stelle, an der eine Referenz auf ein noch nicht übertragenes Objekt entdeckt wird, ohne rekursiven Abstieg solange suspendiert werden, bis das referenzierte Objekt übertragen ist.

[2]Siehe dazu beispielsweise im Portland Pattern Repository (`http://c2.com/`) unter dem Stichwort „ExternalizeTheStack“.

Erst danach dürfte die Ausführung der Serialisierungsmethode an der Stelle fortgeführt werden, an der sie verlassen wurde. Ein solcher Programmablauf würde das Vorhandensein von Co-Routinen[3] (oder Co-Methoden) voraussetzen. Das zweite Problem tritt nur während der Deserialisierung auf der Empfängerseite auf. Hier muss ein halb initialisiertes Objekt (bei dem noch nicht alle Instanzvariablen initialisiert sind) in eine Hilfsdatenstruktur eingereiht werden, um die Initialisierung später fortzusetzen, wenn die von ihm referenzierten Objekte eingelesen sind. Eine solche Teilinitialisierung ist insbesondere in Java dann nicht möglich, wenn das zu lesende Objekt finale Instanzvariablen beinhaltet. Eine final deklarierte Instanzvariable muss während der Ausführung des Konstruktors initialisiert werden und darf nach Verlassen des Konstruktors nicht mehr modifiziert werden.

Lösung Teil 1: Geänderte Traversierungsreihenfolge

Die Unverträglichkeit von spezialisierten Serialisierungsmethoden mit der Umwandlung von Rekursion in Iteration über einen offengelegten Stapelspeicher wird vermieden, indem die Traversierungsreihenfolge von „in-order“ in „pre-order“ geändert wird. Bei der „pre-order“ Traversierung wird in der Serialisierungsmethode nur der Objektzustand selbst geschrieben, nicht aber der Abstieg in die Serialisierungsmethoden der referenzierten Objekte durchgeführt. Erst nachdem die Serialisierungsmethode eines Objektes zurückkehrt, werden die Zustände aller referenzierten Objekte geschrieben, indem deren Serialisierungsmethoden iterativ eine nach der anderen aufgerufen werden.

Damit dieser Ansatz durchführbar wird, muss eine Serialisierungsmethode in der Lage sein, eine Referenz auf ein Objekt zu schreiben, das selbst noch nicht übertragen wurde. Mit den Daten dieser „Vorwärtsreferenz“ muss die empfangende Seite in der Lage sein, ein Objekt zu konstruieren und dessen Referenz in die entsprechende Instanzvariable des referenzierenden Objektes zu schreiben. Das so vorab konstruierte Objekt muss aber noch nicht vollständig mit Daten gefüllt sein. Dieses geschieht erst, nachdem die Deserialisierung des Referenten abgeschlossen ist, da die Daten des referenzierten Objektes erst hinter den Daten des referenzierenden Objektes übertragen werden.

Lösung Teil 2: Aufteilung in rekursiven und nichtrekursiven Teil

Die Teilinitialisierung von Objekten scheitert bei finalen Instanzvariablen. Diese Instanzvariablen müssen im Konstruktor des deserialisierten Objektes während dessen Erzeugung initialisiert werden. Da es durchaus möglich ist, Zyklen in Objektgraphen zu erzeugen, die komplett aus finalen Referenzen bestehen, ist klar, dass mit keiner Traversierungsreihenfolge ein rekursiver Abstieg aus dem Konstruktor eines gerade deserialisierten Objektes vermieden werden kann.

Eine Aufteilung der Methoden für Serialisierung und Deserialisierung in jeweils zwei Methoden löst dieses Problem: Beim Schreiben einer Objektreferenz in den Serialisierungsstrom wird die erste Serialisierungsmethode aufgerufen, die eine rekursive

[3] Siehe beispielsweise in Wikipedia (`http://wikipedia.org/`) unter dem Stichwort Coroutine.

Serialisierung aller finalen Felder des Objektes durchführt. Die zweite Serialisierungsmethode schreibt die restlichen Daten des Objektes. Sie wird erst aufgerufen, nachdem das Objekt, welches die Serialisierung für das referenzierte Objekt angestoßen hat, vollständig bearbeitet ist. Auf der Empfängerseite werden in einem Deserialisierungskonstruktor alle finalen Felder des Objektes initialisiert, die zusammen mit der Objektreferenz übertragen wurden. Das so teilinitialisierte Objekt wird danach in der offengelegten Stapelstruktur solange aufbewahrt, bis die Deserialisierung des referenzierenden Objektes abgeschlossen ist. Danach wird in der zweiten Deserialisierungsmethode der restliche Zustand initialisiert.

Mit diesem Vorgehen kann eine vollständig rekursive Serialisierung in eine nur noch teilrekursive Serialisierung umgewandelt werden. Rekursion findet nur noch entlang finaler Referenzen statt. Zwar ist damit die Gefahr eines Stapelüberlaufs nicht gänzlich gebannt, für die Verwendung zur Datenübertragung in verteilt parallelen Programmen ist die hier vorgeschlagene Lösung der teilrekursiven Objektserialisierung aber akzeptabel, da sich ein eventuell auftretendes Problem mit knappem Stapelspeicher einfach durch Entfernen einiger Final-Markierungen beheben lässt.

4.3 Optimierungen in den Serialisierungsmethoden

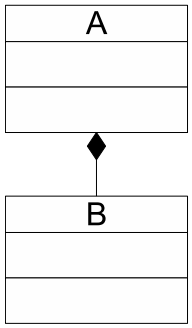
Den größten Geschwindigkeitsgewinn erreicht das Vorhandensein von Serialisierungsmethoden pro Klasse, indem dynamische Typintrospektion vermieden wird und die Serialisierungsmethoden durch den Laufzeitübersetzer optimiert werden können. Mit Kenntnis der Programmstruktur können diese Serialisierungsmethoden noch weiter optimiert werden. Dabei geht es darum, explizite Tests und Datenübertragung zur Laufzeit einzusparen, wenn die Information statisch bekannt ist.

4.3.1 Einsparung von Zyklentests

Um Graphen mit Zyklen serialisieren zu können, müssen diese erkannt und so codiert werden, dass jedes Objekt des Graphen genau einmal übertragen wird. Ohne Kenntnis der Struktur des Graphen muss bei jeder Referenz überprüft werden, ob das Zielobjekt bereits übertragen ist oder nicht. Für den Fall, dass ein Objekt gefunden wird, das bereits übertragen ist, darf nur eine symbolische Referenz auf dieses Objekt in den Strom geschrieben werden, ohne die Zustandsinformation des Objektes zu wiederholen. Ohne Zyklentests würde eine zyklische Struktur in einen unendlichen Strom ausgerollt.

Einsparpotenzial

Um den Zyklentest durchzuführen, muss sich der Serialisierer alle bereits übertragenen Objekte in einer Hash-Tabelle merken. Beim Verfolgen einer Referenz muss ein Hash-Wert des Zielobjektes berechnet und in der Tabelle der bereits übertragenen Objekte nachgeschlagen werden. Wird das Objekt nicht gefunden, muss seine Referenz in die Tabelle aufgenommen werden, bevor sein Zustand serialisiert werden kann. Dieser Vorgang ist zeitaufwendig, so dass bei der Übertragung vieler kleiner Objekte die Zeit für die Serialisierung leicht vom Aufwand für den Zyklentest dominiert werden kann. Daher ist es wichtig, möglichst viele Zyklentests zu vermeiden.




```
public class A {
   @embedded B b;

   ...
}
```

Abbildung 4.1: Kompositum als UML-Klassendiagramm.

Einsparung bei Kompositionsreferenzen

Selbst wenn ein übertragener Objektgraph Zyklen enthält, ist es trotzdem nicht notwendig, bei jeder Referenz einen Zyklentest durchzuführen. Dies gilt sogar für Referenzen, die auf dem Zyklus liegen. Es muss lediglich sichergestellt werden, dass ein Zyklentest bei solchen Referenzen durchgeführt wird, die einer Traversierung möglicherweise eine Rückwärtsreferenz auf ein bereits serialisiertes Objekt darstellen.

Referenzen, die nie Rückwärtsreferenzen sein können, lassen sich aus dem Entwurf der Anwendung erkennen. In Abbildung 4.1 ist ein solches Muster in einem UML-Klassendiagramm dargestellt. Wenn eine Komponente `A` eine Komposition aus einer oder mehreren Komponenten `B` ist, bedeutet das, dass die Komponente `A` die alleinige Verantwortung für die Verwaltung der Komponente `B` hat. Es existiert also außerhalb von `A` keine direkte Referenz auf `B`. In einem Kompositum kann folglich die Referenz, welche die Beziehung von `A` nach `B` implementiert, nie eine Rückreferenz sein. Immer wenn `A` serialisiert wird, ist sichergestellt, dass `B` vorher noch nicht in den Strom geschrieben worden ist, da `A` der alleinige Eigentümer einer Referenz auf `B` ist. In diesem Fall kann der Zyklentest in der Serialisierungsroutine von `A` beim Schreiben der Komponente `B` eingespart werden.

Zur Markierung von Kompositionsreferenzen wird eine Metadaten-Annotation[4] eingeführt, die es erlaubt, solche Referenzen beim Generieren der Serialisierungsfunktionalität speziell zu behandeln. Auf der rechten Seite von Abbildung 4.1 ist diese Markierung verwendet, um die Kompositionsreferenz `b` zu markieren. Diese Annotation wird bei der Generierung von Serialisierungsmethoden genutzt, indem der Zyklentest bei der Serialisierung der Referenz `b` eingespart wird. Wird ein Werkzeug für den Entwurf der Anwendung verwendet, so könnte die Annotation `@embedded` automatisch aus dem UML-Klassendiagramm bei der Erzeugung eines Code-Skelettes miterzeugt werden.

Wertklassen

Bei Wertklassen im Sinne der vorliegenden Arbeit handelt es sich um solche Klassen, deren Objekte so verwendet werden, als besäßen sie keine eigene Identität. Das ist genau dann der Fall, wenn Objekte nach ihrer Konstruktion unveränderlich sind und die

[4]Die @-Syntax für die Annotation von Metainformationen an Deklarationen wird in Java 1.5 neu eingeführt.

```
@faceless class C {
   ...
}
```

Abbildung 4.2: Deklaration einer Wertklasse über Annotation.

```
final class D {        class E {
   ...                    D d;
}                      }
```

Abbildung 4.3: Einsparung von Typinformation für monomorphe Referenzen

Referenz auf ein Objekt jederzeit durch eine Kopie desselben ersetzt werden kann. Per Definition dürfen daher Referenzen auf Wertobjekte keine Zyklen bilden. Zur Deklaration einer Wertklasse wird ebenfalls eine Annotation eingeführt. Abbildung 4.2 zeigt die Verwendung. Java kennt Wertklassen wie sie hier definiert sind implizit als Entwurfsmuster. Ein Beispiel dafür ist die Klasse `String` aus der Java-Basisbibliothek. Mit der oben beschriebenen Annotation kann dieses Entwurfsmuster explizit gemacht werden, um bei der Generierung von Sende- und Empfangsroutinen ausgenutzt werden zu können.

Bei der Serialisierung werden keine Zyklentests für Referenzen auf Objekte von Wertklassen durchgeführt. Da für solche Objekte Referenzen und Kopien austauschbar sind, spielt es keine Rolle, wenn während der Serialisierung eine mehrfache Referenz auf dasselbe Objekt in zwei Kopien aufgelöst wird. Da der Zyklentest eine teure Operation ist, amortisiert sich die dadurch eventuell vergrößerte Menge an übertragenen Daten. Außerdem werden die hier eingeführten Wertklassen ebenfalls für Optimierungen beim Fernaufruf (vgl. Abschnitt 5.2.4) und bei der Replikation (vgl. Abschnitt 6.2.3) verwendet.

C# bietet ein Sprachkonstrukt an, das dem oben beschriebenen Verhalten nahe kommt (Kapitel 18. *Structs* in [46]). Für Java gab es ebenfalls Bestrebungen, sog. leichtgewichtige Klassen zu integrieren [94]; bislang ist allerdings hier keine Umsetzung im Sprachstandard in Reichweite. Beide Vorschläge, sowohl für C# als auch für Java, orientieren sich an den Struktur-Datentypen von C. Derartige Wertklassen setzen eine geänderte Zuweisungssemantik (Kopie statt Referenz) voraus, um das Entkommen von Referenzen auf Wertobjekte in den Hauptspeicher zu verhindern. Dadurch wird eine billige Allokation dieser Objekte auf dem Stapelspeicher möglich, weil die Lebenszeit der Objekte an die Lebenszeit des Stapelspeicherrahmens gebunden ist, der zu der erzeugenden Methode gehört. Zyklenerkennung während der Serialisierung wäre für solche Wertobjekte ebenfalls unnötig, da überhaupt keine Referenzen auf solche Objekte im Hauptspeicher existieren (sie treten dort lediglich als in andere Objekte eingebettete Strukturen auf). Allerdings ließen sich derartige Wertobjekte nicht für die Optimierung beim transparenten Fernaufruf einsetzen (vgl. Abschnitt 5.2.4).

4.3.2 Einsparung von Typinformation

Damit auf der Empfängerseite ein Objekt desselben Typs konstruiert werden kann, wie auf der Senderseite serialisiert wurde, muss in der Regel vor jedem Objekt der Typ des folgenden Objektes angekündigt werden. Anders verhält es ich in Abbildung 4.3 bei

der Klasse `E`. Sie enthält eine Referenz auf die finale Klasse `D`. Bei der Serialisierung einer Instanz von `E` muss lediglich geprüft werden, ob die Referenz null ist oder das referenzierte Objekt bereits übertragen wurde. Ist dies nicht der Fall, können nach der Ankündigung einer monomorphen Referenz ohne Vorspann direkt die Daten von `D` übertragen werden. Der Empfänger weiß aufgrund des Kontextes, dass nur ein Objekt der Klasse `D` übertragen werden kann, da `D` keine Ableitung von Unterklassen zulässt.

Enthält ein Objekt eine Instanzvariable, deren Typ eine finale Klasse ist, kann Polymorphie in dieser Referenz zur Laufzeit ausgeschlossen werden. Ist die Referenz zur Laufzeit nicht `null` und wurde das referenzierte Objekt noch nicht früher im Strom übertragen, so kann der Empfänger ein Objekt der richtigen Klasse konstruieren, ohne dass für die Referenz zusätzliche Typinformation übertragen werden müssen. Diese Optimierung spart gleich doppelt: Einerseits wird die Menge der übertragenen Daten reduziert, andererseits entfällt für den Sender auch das Nachschlagen, ob der Typ bereits übertragen wurde. Diese Überprüfung ist ähnlich der Zyklenerkennung mit einem Nachschlagen in einer Hash-Tabelle verbunden.

Besonders effektiv ist diese Optimierung, wenn die Referenz `d` zusätzlich als eingebettet deklariert ist oder es sich bei `D` um eine Wertklasse handelt. In diesem Fall muss nur noch die Möglichkeit einer null-Referenz gesondert behandelt werden. Diese Unterscheidung ist aber viel weniger aufwendig als der Zyklentest oder das Nachschlagen der Typinformation.

4.4 Kompatibilität

Es stellt sich die Frage, inwieweit die effiziente Objektserialisierung mit der Standard-Objektserialisierung von Java zusammenarbeitet. Offensichtlich kann aufgrund der Optimierungen kein binärkompatibler Strom erzeugt werden. Demzufolge kann weder ein Java-Deserialisierer einen Datenstrom verarbeiten, der mit der schnellen Objektserialisierung erzeugt wurde, noch umgekehrt. Die Kompatibilitätsfrage ist dennoch interessant, wenn es darum geht, Objekte mit der schnellen Objektserialisierung zu übertragen, deren Klassen nicht darauf vorbereitet sind. Da sich die effiziente Objektserialisierung auf Serialisierungsmethoden pro Klasse abstützt, können Objekte, die keine solchen Methoden vorsehen, nicht (direkt) verarbeitet werden. Das kann besonders dann sehr ärgerlich sein, wenn eine Bibliothek verwendet wird, deren Klassen zwar mit der Java-Objektserialisierung übertragbar wären, aber keine Serialisierungsmethoden für die effiziente Objektserialisierung anbieten. Damit in diesem Fall nicht gänzlich auf die schnelle Serialisierung verzichtet werden muss, gibt es zwei Möglichkeiten, solche Objekte dennoch zu verwenden. In der hier vorgestellten optimierten Serialisierung können beide – externe Serialisierungsmethoden oder der Rückfall auf die Standardserialisierung – verwendet werden. Die erste Möglichkeit erzielt eine bessere Geschwindigkeit als die zweite, erfordert aber dafür Zusatzarbeit vom Programmierer.

4.4.1 Externe Serialisierungsmethoden

Um Serialisierungsmethoden für Klassen erzeugen zu können, die nicht durch Einfügen von Methoden verändert werden können oder dürfen, gibt es die Möglichkeit, die

notwendige Funktionalität für die Serialisierung in einer sogenannten Deskriptorklasse extern zu implementieren. Damit dies möglich ist, muss der gesamte Zustand eines solchen Objektes über seine Schnittstelle auslesbar sein. Dies ist beispielsweise für Java-Beans-Klassen und alle Container-Klassen aus der Standardbibliothek möglich.

Ein Deskriptor repräsentiert in der effizienten Objektserialisierung einen übertragbaren Typ. Er ist dafür zuständig, Objekte seiner repräsentierten Klasse zu erzeugen und Aufrufe für Serialisierung und Deserialisierung an die Methoden des Objektes weiterzuleiten. Um Klassen an die effiziente Objektserialisierung anzupassen, ohne ihre Implementierung selbst zu ändern, ist es möglich, stattdessen die Deskriptorklasse abzuleiten und für die Serialisierungsfunktionalität der repräsentierten Klasse zu spezialisieren. Anstatt sich auf die Funktionalität der Serialisierungsmethoden der Zielklasse abzustützen, liest der Deskriptor dann den Zustand des Zielobjektes über öffentliche Methodenaufrufe aus und implementiert die Serialisierung selbst. Ein Beispiel für eines Deskriptorklasse wird in 4.6.4 gegeben.

4.4.2 Rückfall auf die Standardserialisierung

Die im letzten Abschnitt beschriebene externe Serialisierung ist dann nicht möglich, wenn eine Klasse es nicht erlaubt, den kompletten Zustand auszulesen, um damit ein gleichwertiges Objekt zu rekonstruieren. Für diesen Fall besteht die Möglichkeit des Rückfalls auf die Standardserialisierung. Immer wenn die effiziente Objektserialisierung eine Objektreferenz vorfindet, deren Zielobjekt zwar mit der Standardserialisierung serialisierbar wäre, für das aber kein Deskriptor geladen werden kann, fällt sie in den Modus der Standardserialisierung zurück. In diesem Fall wird ein mit der Standardserialisierung für dieses Objekt erzeugter Datenstrom in den Datenstrom der effizienten Objektserialisierung eingebettet.

Notwendigkeit trotz Effizienzverlust

Offensichtlich geht beim Rückfall auf die Standardserialisierung nicht nur der Effizienzgewinn verloren, sondern es muss für die Einbettung sogar noch Zusatzaufwand getrieben werden. Daher lohnt sich dieses Vorgehen nur, wenn in einer großen Datenmenge sehr wenige Objekte mitübertragen werden müssen, für die keine Serialisierungsmethoden vorliegen. Trotzdem ist dieser Rückfall für die praktische Anwendbarkeit der effizienten Objektserialisierung in einem allgemeinen System für entfernten Methodenaufruf entscheidend. Insbesondere bei der Übermittlung von Ausnahmen, die bei der Abarbeitung entfernter Methoden auftreten können, muss eine Vielfalt von Ausnahme-Klassen übertragen werden können, die größtenteils in der Standardbibliothek definiert sind. Weder sind diese Klassen auf die schnelle Objektserialisierung vorbereitet, noch lohnt es sich, eine eigene Deskriptorklasse für jede einzelne zu erzeugen. Wäre ein Rückfall auf die Standardserialisierung in solchen Fällen nicht möglich, bestünde eine starke Einschränkung für die Benutzbarkeit der schnellen Objektserialisierung.

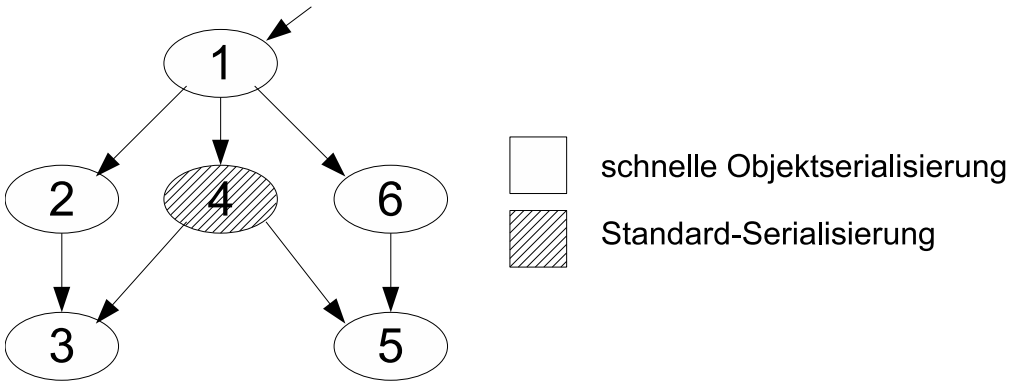


Abbildung 4.4: Kritische Serialisierungsreihenfolge beim Rückfall auf die Standard-Serialisierung.

Nahtlose Integration

Die Kombination beider Serialisierungspakete während der Serialisierung eines einzigen Objektgraphen stellt weitere Herausforderungen. So ist vorstellbar, dass ein Objekt, das nur mit der Java-Serialisierung verarbeitbar ist, Referenzen auf andere Objekte hält, die auch mit der schnellen Serialisierung übertragen werden könnten. In diesem Fall verarbeitet die Java-Serialisierung aber alle Objekte, die von ihrem Wurzelobjekt aus erreichbar sind. Ein Beispiel für einen solchen Graphen ist in Abbildung 4.4 dargestellt. Objekte sind durch Ellipsen symbolisiert. Weiß bedeutet, dass das betreffende Objekt mit der schnellen Serialisierung verarbeitbar ist, schraffiert bedeutet, dass ein Rückfall auf die Standard-Serialisierung notwendig ist. Die Zahlen in den Ellipsen geben an, in welcher Reihenfolge die Serialisierung den Objektgraph abläuft.

Zwei mögliche Problemfälle können auftreten: Erstens kann die Standardserialisierung auf ein Objekt treffen, das bereits mit der schnellen Objektserialisierung übertragen wurde. Dies kann geschehen, wenn für dieses Objekt ein Alias außerhalb des mit der Standardserialisierung verarbeiteten Graphen existiert, und dieser Teil des Graphen bereits bearbeitet ist. Ein Beispiel ist Objekt 3 in Abbildung 4.4, da seine Referenz bereits während der Serialisierung von Objekt 2 das erste mal gefunden wurde. Während der Übertragung von Objekt Nummer 4 findet die Standard-Serialisierung ebenfalls eine Referenz auf Objekt 3. Ohne weitere Maßnahmen würde dadurch Objekt 3 wiederholt übertragen, da die beiden Serialisierungsmechanismen über keine gemeinsamen Tabellen für die Zyklenerkennung verfügen. Der umgekehrte Fall tritt bei Objekt Nummer 5 auf. Auf dieses wird eine Referenz zuerst von der Standard-Serialisierung gefunden. Dadurch serialisiert sie das Objekt mit, obwohl es ebenfalls mit der schnellen Objektserialisierung verarbeitbar wäre. Wenn danach Objekt 6 wieder mit der schnellen Objektserialisierung geschrieben wird, darf diese den Zustand von Objekt 5 nicht wiederholen. Ansonsten würde das Objekt 5 auf Empfängerseite verdoppelt.

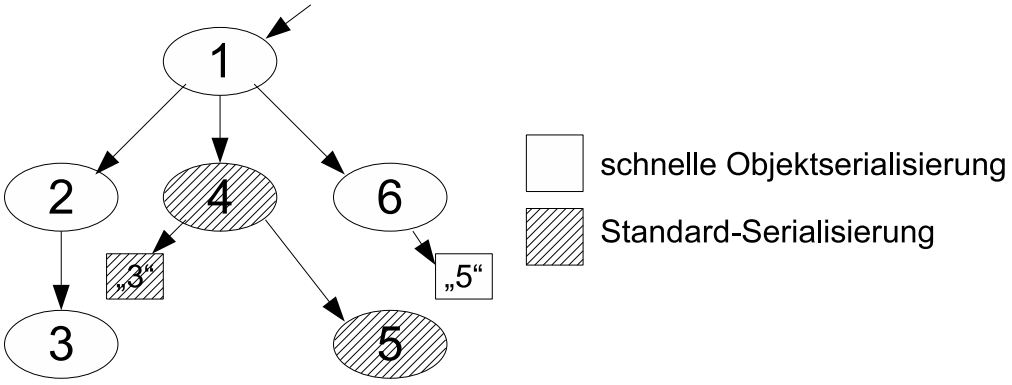


Abbildung 4.5: Tatsächliche Zuordnung der Objekte zu den Serialisierungspaketen.

Lösung des Integrationsproblems

In beiden Fällen kann der Objekt-Ersetzungsmechanismus der Standard-Serialisierung zur Lösung eingesetzt weren. Dieser Ersetzungsmechanismus präsentiert jedes Objekt, das gerade im Begriff ist, übertragen zu werden, und jedes Objekt, das empfangen wurde, einer benutzerdefinierten Methode. Diese Ersetzungsmethode ist auf Senderseite dafür vorgesehen, um ein anderes Objekt an Stelle des ursprünglichen zu übertragen. Auf Empfängerseite kann ein empfangenes Objekt durch ein anderes substituiert werden, bevor es an die Applikation ausgeliefert wird. Ursprünglich ist dieser Mechanismus gedacht, um beispielsweise die transparente Ersetzung von Server-Implementierungsobjekten durch ihre Stellvertreter in entfernten Methodenaufrufen zu realisieren. Derselbe Mechanismus kann, wie im folgenden beschrieben wird, auch dafür eingesetzt werden, um Interoperabilität zwischen zwei Serialisierungsmechanismen zu erreichen.

Erkennen von Rückwärtsreferenzen

Bei der Serialisierung von Objekt 4 in Abbildung 4.4 wird auf die Standard-Serialisierung umgeschaltet. Diese würde als nächstes versuchen, das referenzierte Objekt 3 zu serialisieren. Zuvor wird dieses aber dem Ersetzungsmechanismus präsentiert. Indem die schnelle Objektserialisierung die Ersetzungsmethode für die Standard-Serialisierung definiert, kann diese erkennen, dass Objekt 3 bereits mit der schnellen Objektserialisierung übertragen wurde. Wie in Abbildung 4.5 gezeigt, wird daher das Objekt 3 durch eine symbolische Referenz ersetzt und die Kante im Graph aus Sicht der Standard-Serialisierung getrennt. Auf Empfängerseite präsentiert die Standard-Serialisierung die übertragene symbolische Referenz ebenfalls dem Ersetzungsmechanismus, der dort das bereits vorrätige Objekt 3 aus der Empfängertabelle der schnellen Objektserialisierung holen und die symbolische Referenz dagegen austauschen kann.

Integration der Zyklenerkennungen

Als nächstes wird Objekt 5 von der Standardserialisierung verarbeitet. Wenn dieses Objekt dem Ersetzungsmechanismus präsentiert wird, kann dafür keine symbolische

Referenz erzeugt werden, da dieses Objekt noch nicht übertragen wurde. Es muss daher von der Standard-Serialisierung mitübertragen werden, obwohl es für die effiziente Serialisierung vorbereitet wäre. Damit eine weitere Referenz, die während der anschließenden wieder aufgenommenen schnellen Serialisierung gefunden wird, nicht zu einer Doppelübertragung führt, registriert die Ersetzungsmethode auf Senderseite Objekt 5 aber vorsorglich in der Tabelle zur Zyklenerkennung der schnellen Serialisierung. Genau dieser Fall tritt bei der Übertragung von Objekt 6 ein. Beim Verfolgen der Referenz auf Objekt 5 erkennt jetzt die schnelle Objektserialisierung, dass dieses Objekt bereits übertragen wurde und übermittelt lediglich seine ihm zugewiesene Objektnummer. In Abbildung 4.5 bedeutet das schraffierte Rechteck die in der Ersetzungsmethode der Standard-Serialisierung erzeugte symbolische Referenz und das weiße Rechteck die von der schnellen Serialisierung übermittelte Objektnummer, die statt einer Wiederholung von Objekt 5 übermittelt wird. Das Senden einer Objektnummer bei Detektion der schließenden Kante eines Zyklus ist dabei der normale Weg, wie Zyklen in Objektgraphen durch die Serialisierung codiert werden. Dieses Verfahren arbeitet nahtlos mit der in Abschnitt 4.3.1 beschriebenen Zyklenoptimierung zusammen, da sowohl das zusätzliche Eintragen als auch das Nachschlagen in der Tabelle zur Zyklenerkennung während der Standard-Serialisierung unter Kontrolle der optimierten Serialisierung abläuft.

Unterschiedliche Aufrufreihenfolgen der Ersetzungsmethoden

Normalerweise vergibt der Sender für jedes Objekt eine fortlaufende Nummer. Da die Objekte in derselben Reihenfolge empfangen werden, wie sie gesendet werden, kann der Empfänger ebenfalls für jedes empfangene Objekt eine fortaufende Nummer vergeben. Dadurch entstehen auf Sender- und Empfängerseite korrespondierende Tabellen, die für die Zyklenerkennung benutzt werden können. Das besondere an der für Objekt 5 vergebenen Objektnummer ist, dass die schnelle Serialisierung diese Nummer gar nicht selbst vergeben hat, sondern dass das Objekt in der Ersetzungsmethode der Standard-Serialisierung eingetragen wurde. Die einzige Gelegenheit, Objekt 5 auf Empfängerseite in die Tabelle der schnellen Objektserialisierung einzutragen, ist die dort aufgerufene Ersetzungsmethode der Standard-Serialisierung. Die Reihenfolgen, in denen die Ersetzungsmethoden auf Sender- und Empfängerseite aufgerufen werden, unterscheiden sich aber: Auf Senderseite wird ein Objekt der Ersetzungsmethode präsentiert, *bevor* es verarbeitet wird. Auf Empfängerseite dagegen wird ein Objekt der Ersetzungsmethode erst dann präsentiert, wenn es *komplett* eingelesen wurde (wenn alle von ihm referenzierten Objekte bereits vollständig deserialisiert sind). Die Traversierungsreihenfolgen aus Sicht der Ersetzungsmethoden sind damit auf Senderseite „pre-order“ Tiefensuche und auf Empfängerseite „post-order“ Tiefensuche.

Um nach Abschluss einer teilweisen Serialisierung mit der Standard-Serialisierung konsistente Objektnummern zu erhalten, müssen die Nummern, unter denen die Objekte auf Senderseite geführt werden, explizit mitübertragen werden. Dieses geschieht, indem nach Abschluss der Serialisierung von Objekt 4 durch die Standard-Serialisierung die senderseitige Objektnummer für jedes während des Rückfalls übertragene Objekt explizit wiederholt wird.

Obwohl der Rückfall auf die Standard-Serialisierung Mehraufwand verursacht, ist

er aus Gesichtspunkten der Benutzbarkeit dennoch unerlässlich. Mit der beschrieben Technik lässt sich eine nahtlose Integration der Standard-Serialisierung mit einem alternativen Serialisierungsmechanismus ohne Eingriff in den Code der Standard-Serialisierung erreichen.

4.5 Architektur

Die Architektur der effizienten Objektserialisierung ist in drei Schichten gegliedert, die in das in Abbildung 4.6 gezeigte Klassendiagramm eingetragen sind. Die oberste Schicht ist die Objekt-Versendeschnittstelle, über welche eine Anwendung die schnelle Objektserialisierung verwendet. Eine Anwendung könnte hier beispielsweise ein Paket für entfernten Methodenaufruf sein. Die dünne Mittelschicht stellt eine Abstraktionsebene dar, die eine effiziente Anbindung der schnellen Objektserialisierung an verschiedene Transporttechnologien zur Netzwerkübertragung ermöglicht. Die unterste Schicht ist kein integraler Bestandteil der Serialisierung mehr, sondern zeigt, wie verschiedene Netzwerktechnologien, die entweder eine strom- oder eine paketorientierte Schnittstelle anbieten, an die schnelle Objektserialisierung angebunden werden.

Die folgenden Abschnitte 4.5.1 und 4.5.2 besprechen die Schnittstelle zum Objektversand gegenüber der Anwendung und die technologieneutrale Anbindung an die Transportschicht. Danach stellt Abschnitt 4.5.3 die Voraussetzungen vor, die Anwendungsklassen erfüllen müssen, um mit der hier beschriebenen optimierten Serialisierung übertragen werden zu können. Abschließend wird in Abschnitt 4.5.4 der Ablauf einer Serialisierungsoperation gezeigt.

4.5.1 Die Objekt-Versendeschnittstelle

Die beiden Klassen `MarshalStream` und `UnmarshalStream` (ganz oben in Abbildung 4.6) stellen die zentrale Funktionalität bereit, die in der Standard-Serialisierung von `ObjectOutputStream` und `ObjectInputStream` angeboten wird. Die Klasse `MarshalStream` implementiert die Serialisierung, während `UnmarshalStream` den Deserialisierungsteil realisiert. `MarshalStream` und `UnmarshalStream` implementieren dazu die von der Standard-Serialisierung vorgegebenen Schnittstellen `ObjectOutput` und `ObjectInput`, so dass ein problemloser Austausch der Standard-Serialisierung durch die schnelle Serialisierung in einer Anwendung möglich ist.

Während Serialisierer und Deserialisierer als Endpunkte von Netzwerkverbindungen gedacht sind, um diese für die Übertragung von Objektgraphen zu verwenden, bietet die schnelle Serialisierung eine weitere Klasse `DeepClone`, die eine tiefe Kopie eines Objektgraphen *innerhalb* einer virtuellen Maschine erstellt. Konzeptuell kann diese Klasse als ein „kurzgeschlossenes“ Paar aus Serialisierer und Deserialisierer betrachtet werden, das einen Objektgraphen in einen Puffer schreibt und sofort wieder aus diesem zurückliest. Die Klasse `DeepClone` arbeitet jedoch effizienter, indem eine strukturierte Kopie des Objektgraphen direkt erzeugt und dadurch die Übersetzung in und aus einer Netzwerkrepräsentation mit den damit verbundenen zusätzlichen Kopierschritten eingespart wird.

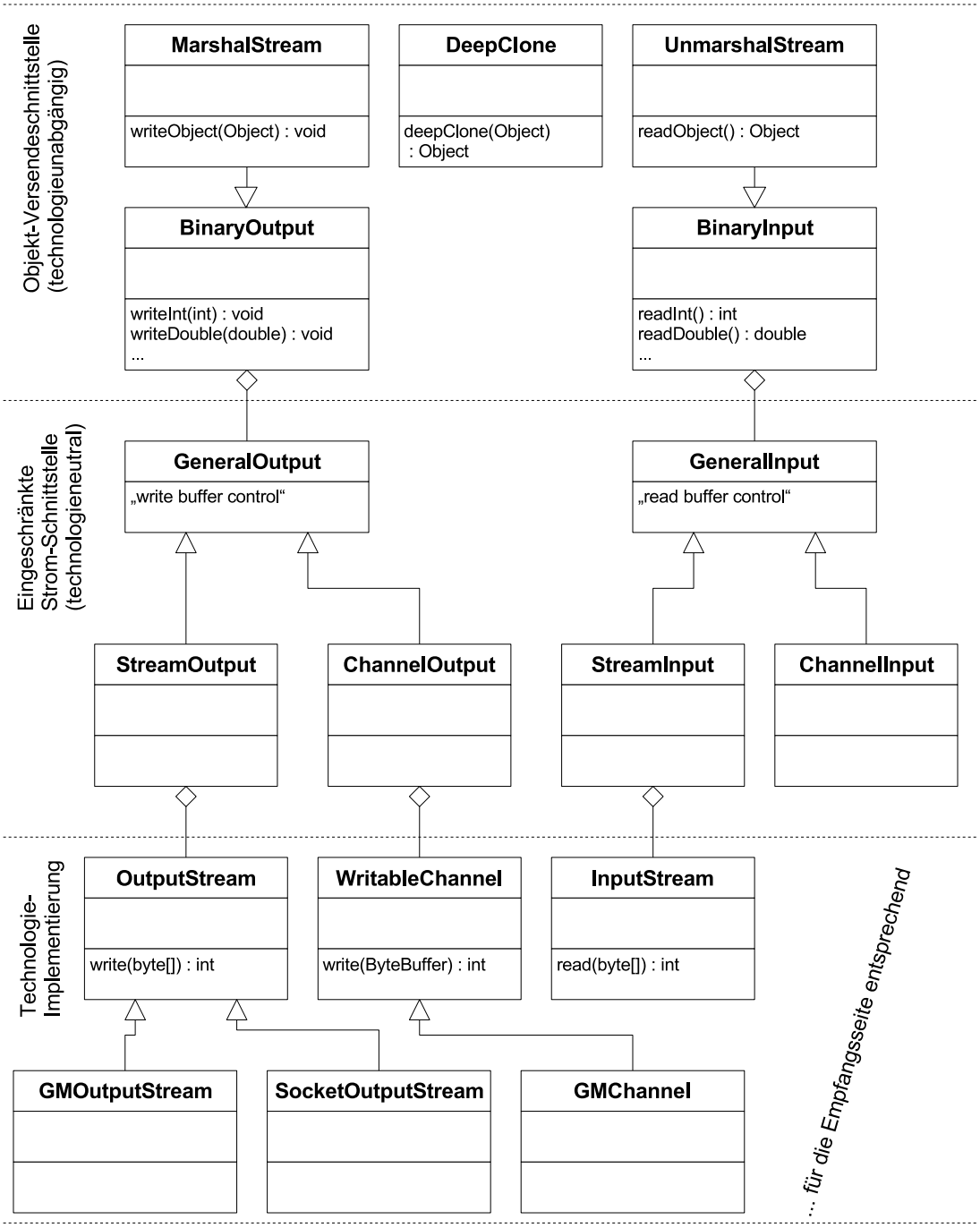


Abbildung 4.6: Klassen-Diagramm der schnellen Objektserialisierung.

Während `DeepClone` bei der Traversierung eine direkte Kopie erstellt, bauen `MarshalStream` und `UnmarshalStream` auf Klassen für die Konvertierung von Binärdaten in eine plattformunabhängige Netzwerkrepräsentation auf. `BinaryOutput` konvertiert Basisdatentypen und Zeichenketten in Bytefolgen und arbeitet dafür mit der darunterliegenden allgemeinen Pufferungsschicht zusammen. Auf der Eingabeseite funktioniert `BinaryInput` entsprechend.

4.5.2 Technologieneutrale Abstraktionsschicht

Die mittlere Schicht in Abbildung 4.6 abstrahiert von der konkreten Implementierung der Übertragungspufferung. Dies ermöglicht Objektübertragungen sowohl über Technologien, die eine stromorientierte Übertragungsschnittstelle anbieten, als auch über solche, die paketorientiert sind.

Strom- vs. paketorientierter Transportschicht

Eine stromorientierte Transportschicht bietet der Anwendung eine Schnittstelle, die es erlaubt, beliebige Datenfragmente in einen Strom zu schreiben und in einer beliebigen Stückelung wieder zu lesen. Für die Übertragung führt eine solche Transportschicht eine interne Pufferung durch, um nicht für viele kleine Schreibanfragen einzelne Nachrichten versenden zu müssen und um beliebige Leseanfragen passend befriedigen zu können. Diese Form der Kommunikationsschnittstelle ist weitverbreitet, da sie von der Anwendung denkbar einfach zu handhaben ist. Insbesondere funktioniert so die Kommunikation über TCP-Sockets, die in Java durch die Klassen `SocketOutputStream` und `SocketInputStream` abgebildet wird. Diese automatische Anpassung der Stückelungsgröße von Nachrichten bedingt eine zusätzliche Pufferung innerhalb der Transportschicht, die zu einer weiteren Kopieroperation der übertragenen Daten führt.

Um eine Übertragung ohne zusätzliches Kopieren der Daten zu ermöglichen, bieten Kommunikationsbibliotheken für Hochgeschwindigkeitsnetzwerke eine paketorientierte Schnittstelle[5] an. Eine paketorientierte Schnittstelle führt keine automatische Anpassung der Stückelungsgröße durch, sondern sendet für jede Schreiboperation eine Nachricht und erwartet, dass eine Leseoperation in der Lage ist, eine komplette Nachricht entgegenzunehmen. Bei einer paketorientierten Schnittstelle muss die Anwendung selbst für eine vernünftige Stückelungsgröße der gesendeten Nachrichten sorgen, behält also die Kontrolle über die Pufferung. Im Gegenzug kann die Kommunikationstechnologie eine Datenübertragung ohne zusätzliche Kopieroperationen auf dem Übertragungsweg gewährleisten.

Pufferung durch die Serialisierung

Während der Serialisierung muss in jedem Fall eine Pufferung der übertragenen Daten vorgenommen werden, um auf Senderseite das Übertragungsprotokoll zu erzeugen und es auf Empfängerseite wieder zu zerteilen und in eine Objektstruktur zurück-

[5] Paketorientiert bezieht sich hier lediglich auf die fehlende Anpassung der Stückelungsgröße. Paketfolgen bleiben reihenfolgetreu, verlust- und verdopplungsfrei.

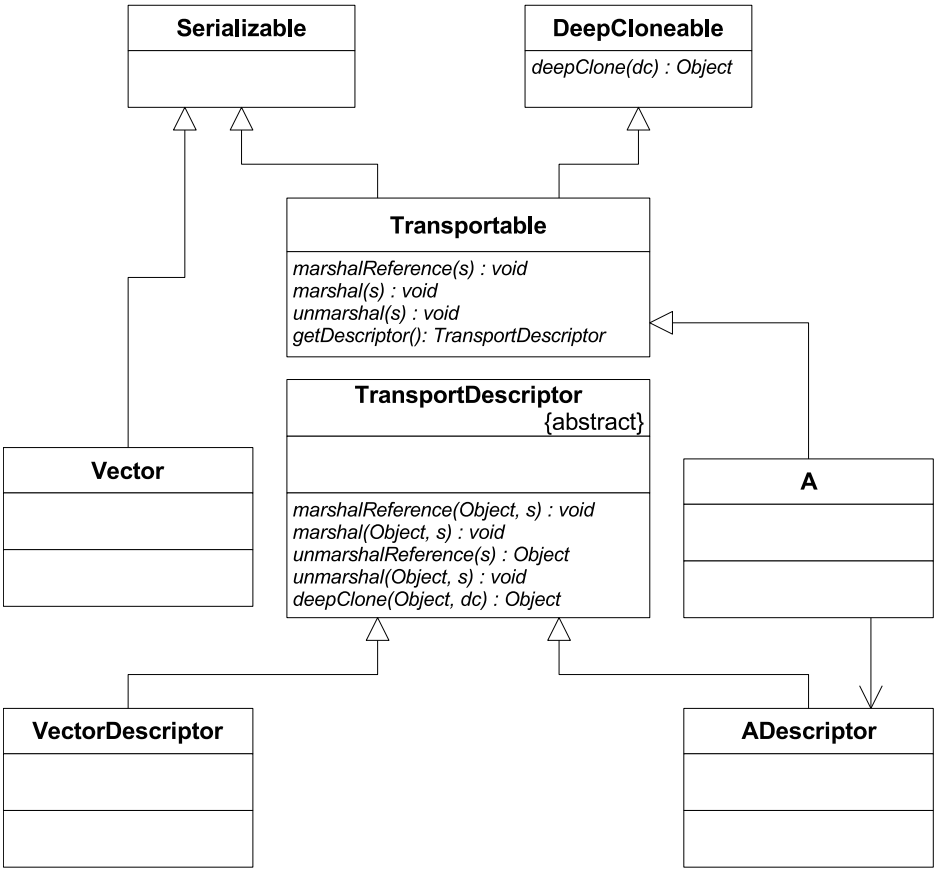


Abbildung 4.7: Schnelle Objektserialisierung für eine Anwendungs- und eine Bibliotheksklasse.

zuübersetzen. Die Objektserialisierung als Anwendung eignet sich daher bestens zur Zusammenarbeit mit einer paketorientierten Transportschicht. Die technische Herausforderung besteht darin, einerseits eine effiziente Anbindung der Pufferverwaltung zu erreichen, aber andererseits die Serialisierungsfunktionalität so von der Pufferung zu trennen, dass eine Zusammenarbeit mit den unterschiedlichen Schnittstellen der Transporttechnologien möglich wird.

Java bietet zwei unterschiedliche Schnittstellen für stromorientierte und paketorientierte Transporttechnologien. Im Paket `java.io` befinden sich die Klassen `OutputStream` und `InputStream`, die als Oberklassen für alle stromorientierte Kommunikation dienen. Über die `Channel`-Schnittstelle im Paket `java.nio.channels` steht ein Rahmen für paketorientierte Kommunikation mit expliziter Pufferverwaltung zur Verfügung. Um die schnelle Objektserialisierung anders als die Standard-Serialisierung mit beiden Schnittstellen verwenden zu können, abstrahiert die in Abbildung 4.6 gezeigte mittlere Schicht von der konkreten Implementierung der Pufferverwaltung. Bei Verwendung einer auf der `Channel`-Schnittstelle aufbauenden Transporttechnologie werden die dort definierten Puffer verwendet, um direkt in einen Puffer der darunterliegenden Kommunikationsbibliothek serialisieren zu können. Bei Verwendung der Stromschnittstelle wird ein eigener Puffer für Schreib- und Leseanforderungen erzeugt. Diese Schreib- und Leseanforderungen werden in diesem Fall aber

so gepaart abgesetzt, dass für die darunterliegende Kommunikationstechnologie keine Notwendigkeit für eine Anpassung der Stückelungsgröße der Daten besteht. Über diese so eingeschränkte Sicht auf den Strom kann auch eine ansonsten paketorientierte Technologie wie GM/Myrinet über die Stromschnittstelle an die Serialisierung angebunden werden, ohne dass eine zusätzliche Pufferung notwendig wird. Eine solche Anbindung ist in der Klasse `GMOutputStream` realisiert.

4.5.3 Anbindung serialisierbarer Klassen

Die schnelle Objektserialisierung kann prinzipiell jedes Objekt verarbeiten, das die von der Standard-Serialisierung vorgegebene Schnittstelle `Serializable` implementiert. Damit Objekte allerdings *effizient* verarbeitbar sind, muss für ihre Klasse ein sogenannter Deskriptor verfügbar sein. Abbildung 4.7 zeigt die Schnittstellen und Klassen, die für eine zu serialisierende Anwendungsklasse relevant sind. Eine Anwendungsklasse implementiert normalerweise die Schnittstelle `Transportable`, welche die Schnittstellen `Serializable` und `DeepCloneable` kombiniert und erweitert. Eine Klasse bietet durch Implementierung der in `Transportable` deklarierten Methoden und Bereitstellung eines speziellen Konstruktors zur Erzeugung neuer Objekte während der Deserialisierung die Funktionalität an, die für ihre effiziente Behandlung während der Serialisierung ohne Rückfall auf die Standard-Serialisierung notwendig ist. Die Deskriptorklasse erfüllt zwei Funktionen: Sie stellt eine Fassade für die Serialisierungsfunktionalität bereit und dient als Fabrik bei der Konstruktion neuer Objekte während der Deserialisierung. Da alle Serialisierungsfunktionalität ausschließlich über den Deskriptor einer Klasse angesprochen wird, ist es möglich, auch Bibliotheksklassen mit Serialisierungsfunktionalität nachzurüsten, die nicht modifiziert werden können bzw. dürfen. In diesem Fall leitet der Deskriptor Zugriffe nicht an das betroffene Objekt weiter, sondern implementiert die Serialisierungsfunktionalität quasi extern selbst.

In Abbildung 4.7 sind zwei Beispiele gegeben. Die Bibliotheksklasse `Vector` implementiert selbst nur die Schnittstelle `Serializable`, stellt also keine eigene Serialisierungsfunktionalität bereit. Ihr Deskriptor `VectorDescriptor` implementiert die Serialisierung und Konstruktion neuer Vektoren extern. Die Verbindung zwischen der Anwendungsklasse und ihrem Deskriptor wird in diesem Fall über Namenskonvention hergestellt. Die Klasse `A` implementiert selbst die Schnittstelle `Transportable` und stellt damit die notwendige Serialisierungsfunktionalität bereit. Ihr Deskriptor `ADescriptor` realisiert lediglich eine Fassade für den Zugriff auf diese Funktionalität. In diesem Fall ist keine Konvention für die Benennung von Klasse und Deskriptor notwendig, da das zu serialisierende Objekt selbst eine Referenz auf seinen Deskriptor liefern kann.

4.5.4 Ablauf von Serialisierung und Deserialisierung

Die Zusammenarbeit der Klassen des Serialisierungspakets mit den Anwendungsklassen und ihren Deskriptoren ist im Sequenzdiagramm in Abbilung 4.8 gezeigt.

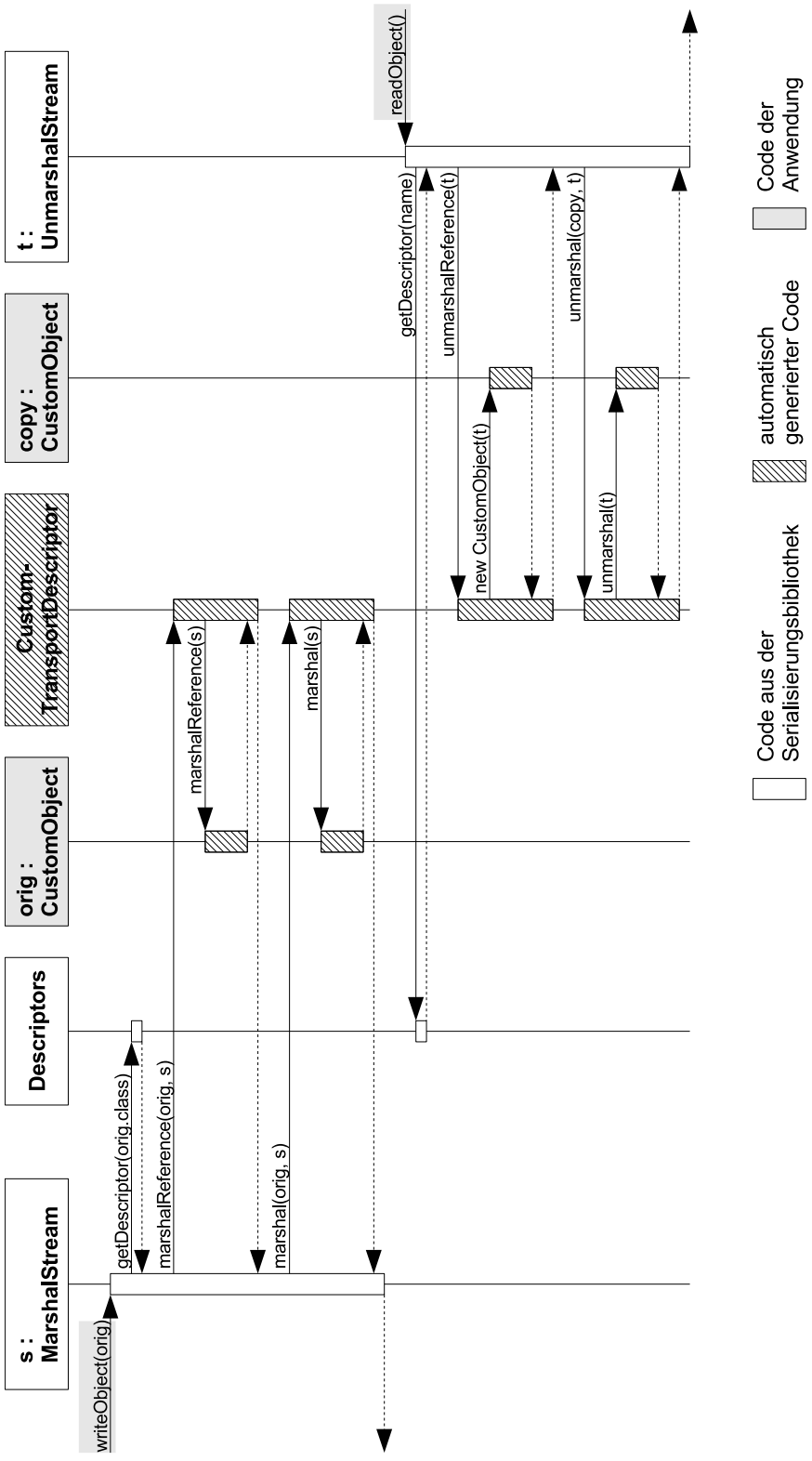


Abbildung 4.8: Sequenzdiagramm der schnellen Objektserialisierung.

Serialisierung

Wie bei der Standard-Serialisierung beginnt der Prozess mit dem Aufruf der Methode `writeObject()` von `MarshalStream`. Diese besorgt sich das passende Deskriptorobjekt, das für die Serialisierung als Stellvertreter für den zu übertragenden Typ dient. Als nächstes wird die Referenz auf das Objekt serialisiert. Diese Objektreferenz besteht aus dem konkreten Typ des Objektes zuzüglich aller Informationen, die auf Empfängerseite notwendig sind, um ein neues Objekt desselben Typs anzulegen. Das sind bei Objekten, welche die Schnittstelle `Transportable` implementieren und daher eigene Serialisierungsmethoden haben, alle finalen Instanzvariablen, da diese während der Ausführung des Deserialisierungs-Konstruktors initialisiert werden müssen.

Für die Serialisierung der Referenz wird die Methode `marshalReference()` auf dem konkreten Deskriptor-Objekt aufgerufen, das zur Klasse des zu übertragenden Objektes gehört. In dem vorliegenden Fall implementiert diese Klasse (`Custom-Object`) die Schnittstelle `Transportable`. Daher reicht der Deskriptor den Programmfluss an die Methode `marshalReference()` des zu übertragenden Objektes weiter.

Nach dem Schreiben der Objektreferenz wird der Datenteil aller bis dahin noch nicht vollständig übertragenen Objekte verschickt. Im konkreten Fall muss genau die Serialisierung des Wurzelobjektes abgeschlossen werden. Dazu dient die Methode `marshal()` aus der Schnittstelle `Transportable`. Diese Methode wird wieder über den Klassendeskriptor aufgerufen. Die `marshal()`-Methode schickt alle diejenigen Daten, die noch nicht bei der Übermittlung der Referenz übertragen wurden. Befinden sich darunter weitere Referenzen auf andere Objekte, findet anders als bei der Standard-Serialisierung kein rekursiver Aufruf von `writeObjet()` statt. Für diese Referenzen wird statt dessen nur der erste Teil – die Übertragung der Referenz mit `marshalReference()` – durchgeführt. Alle so teilserialisierten Objekte werden in eine Warteschlange eingereiht. Bevor der initiale Aufruf `writeObject()` schließlich zurückkehrt, werden die Datenanteile aller noch nicht vollständig übertragenen Objekte gesendet. Auf diese Art und Weise wird Rekursion während der Serialisierung auf den finalen Teilgraphen des zu übertragenden Objektgraphen beschränkt.

Deserialisierung

Auf Empfängerseite beginnt der Prozess mit dem Aufruf von `readObject()`. Diese Methode liest als erstes den Typ des zu empfangenden Objektes. Mit diesem Typ wird der passende Deskriptor gefunden. Der Deskriptor ist in `unmarshalReference()` dafür zuständig, ein neues Objekt der durch ihn repräsentierten Klasse zu erzeugen. Dazu ruft er bei `Transportable`-Objekten einen spezialisierten Konstruktor auf. Als Argument erwartet dieser den Strom, aus dem das konstruierte Objekt gelesen werden soll. Dieser Deserialisierungskonstruktor liest und initialisiert alle finalen Instanzvariablen des neuen Objektes und konsumiert dabei genau diejenigen Daten, die während der Serialisierung der Referenz geschrieben wurden. Alle anderen Felder des Objektes bleiben uninitialisiert, wieder um den rekursiven Abstieg auf einen Teilgraphen zu beschränken, der so klein wie möglich ist. Enthält das zu lesende Objekt finale

Referenzen auf andere Objekte, werden diese Objekte auch nur soweit initialisiert, wie unbedingt notwendig.

Nach Abschluss der Objekterzeugung auf Empfängerseite werden die restlichen Daten all derjenigen Objekte gelesen, die noch nicht vollständig initialisiert sind. Dazu wird für jedes dieser Objekte die Methode `unmarshal()` aufgerufen. Diese liest die Werte derjenigen Instanzvariablen, deren Initialisierung zum Zeitpunkt der Objekterzeugung aufgeschoben wurde. Werden währenddessen Referenzen auf weitere Objekte gelesen, werden auch diese nur teilinitialisiert und in die Schlange der Objekte eingereiht, die auf den Abschluss ihrer Initialisierung warten.

Vergleich mit der Standard-Serialisierung

Die schnelle Serialisierung benötigt weder dynamische Introspektion noch einen Ausbruch aus der virtuellen Maschine über native Methoden. In beiderlei Hinsicht ist sie damit besser als die Standard-Serialisierung, die den kompletten Zustand der Objekte über langsame dynamische Introspektion liest und initialisiert. Die Objekterzeugung in der Standard-Serialisierung geschieht in einer nativen Methode außerhalb der virtuellen Maschine, weil ein uninitialisiertes Objekt ohne Aufruf eines Konstruktors erzeugt werden muss. Das Initialisieren von finalen Instanzvariablen außerhalb des Konstruktors und der Zugriff auf private Variablen von außerhalb ihrer Klasse ist in der Standard-Serialisierung nur möglich, da sie in einem privilegierten Modus der virtuellen Maschine abgearbeitet wird. Die schnelle Serialisierung vermeidet diese Tricks, indem Methoden der Klasse selbst das Auslesen und Befüllen von Instanzvariablen durchführen und damit legal Zugriff auf alle Instanzvariablen besitzen. Durch Aufruf eines spezialisierten Konstruktors ist selbst das korrekte Initialisieren von Objekten mit finalen Variablen möglich. Selbst der Aufruf des Deserialisierungskonstruktors benötigt keinen reflektiven Zugriff, da dieser durch das ebenfalls spezialisierte Deskriptorobjekt durchgeführt wird. Die Rekursionstiefe während der schnellen Objektserialisierung ist beschränkt durch die Länge der längsten finalen Referenzkette.

4.6 Automatische Code-Erzeugung

Um keinen Verlust an Komfort bei der Verwendung der schnellen Serialisierung gegenüber der Standard-Serialisierung zu erleiden, muss die notwendige Zusatzfunktionalität pro Anwendungsklasse automatisch generiert werden. Da die Zusatzfunktionalität in die Anwendungsklassen eingebettet werden muss, ist es notwendig die Anwendungsklassen selbst zu modifizieren. Dazu könnte entweder eine Nachbehandlung der fertigübersetzten Klassen oder ein Vorverarbeitungsschritt vor der Übersetzung eingeschoben werden.

Beide Vorgehensweisen wurden realisiert. Es ist möglich, fertigübersetzte serialisierbare Klassen einzulesen, zu analysieren und die notwendigen Code-Fragmente zu erzeugen, welche die Zusatzfunktionalität für die schnelle Serialisierung realisieren. Da die gesamte verteilte Programmierumgebung auf einer Reihe von Vorverarbeitungs- und Transformationsschritten beruht, ist das Einfügen eines weiteren Transformationsschrittes für die Erzeugung der Serialisierungsfunktionalität jedoch einfacher

```
private void restoreAfterUnmarshal()
   throws ClassNotFoundException, IOException;

private void prepareBeforeMarshal()
   throws IOException;
```

Abbildung 4.9: Methodensignaturen zur Spezifikation von Vor- und Nachbereitungsarbeiten während der Serialisierung.

in der Handhabung. Mit diesem Ansatz werden Klassen, die mit dem Übersetzer für die verteilte Programmierumgebung verarbeitet werden, bei ihrer Übersetzung automatisch mit Serialisierungsfunktionalität ausgerüstet.

4.6.1 Einmischen benutzerdefinierter Funktionalität während der Serialisierung

In einigen Anwendungsklassen kann es notwendig sein, Zusatzfunktionalität anzugeben oder im Extremfall die automatische Generierung der Serialisierungsmethoden ganz durch handgeschriebene Methoden zu ersetzen. Beides ist mit der schnellen Objektserialisierung, wie folgt, möglich.

Vor- und Nachbereitung der Serialisierung

In vielen Fällen kann eine benutzerdefinierte Aktion vor der Serialisierung eines Objektes oder nach erfolgreicher Deserialisierung notwendig sein. Beispielsweise ist eine Nachbereitung im Anschluss an die Deserialisierung notwendig, wenn die Klasse transiente Instanzvariablen besitzt. Da transienter Zustand wird nicht mitübertragen wird, bleibt er nach abgeschlossener Deserialisierung uninitialisiert. Der transiente Zustand eines eingelesenen Objektes muss in der Regel nach der Deserialisierung neu berechnet werden.

In der Standard-Serialisierung gibt es keine gesonderte Möglichkeit, benutzerdefinierte Vor- und Nachbereitungsarbeiten bei der Serialisierung getrennt zu spezifizieren. Allerdings kann eine Methode bereitgestellt werden, welche die Serialisierung oder Deserialisierung einer Klasse komplett übernimmt. Ist lediglich Zusatzfunktionalität vor oder nach der Serialisierung notwendig, muss eine solche Serialisierungsmethode geschrieben werden, welche die Zusatzfunktionalität beinhaltet und für die eigentliche Serialisierung einen rekursiven Aufruf in den allgemeinen Serialisierungsmechanismus vornimmt. Dieser Entwurf macht eine nichtrekursive Implementierung der Standard-Serialisierung unmöglich.

Vor- und Nachbereitungsfunktionalität kann bei der schnellen Serialisierung deshalb in benutzerdefinierten Methoden angegeben werden. Die automatische Code-Erzeugung registriert das Vorhandensein dieser Methoden und baut Aufrufe in die generierten Versende- und Empfangsroutinen ein. Abbildung 4.9 gibt die Methodensignaturen an, die von benutzerdefinierten Vor- und Nachbereitungsmethoden benutzt werden müssen.

Benutzerdefinierte Serialisierungsfunktionalität

In manchen Fällen kann es günstiger sein, auf die automatische Generierung der Serialisierungsfunktionalität zu verzichten und Serialisierungsmethoden per Hand zu schreiben. Dies ist der Fall, wenn sich der Zustand eines Objektes auf Sende- und Empfangsseite wegen des Einflusses äußerer Faktoren unterscheiden muss. Beispielsweise ist das bei der Übertragung einer Hash-Tabelle der Fall, welche als Hash-Funktion die von der virtuellen Maschine bereitgestellte Hash-Funktion für die Objektidentität verwendet. In diesem Fall unterscheidet sich die Zuordnung von Objekten zu den Hash-Containern der Tabelle auf Sende- und Empfangsseite. Dies kommt daher, dass die Hash-Werte dieser Funktion nur für eine virtuelle Maschine gültig sind. Daher muss die Struktur einer solchen Hash-Tabelle bei der Übertragung auf eine andere virtuelle Maschine wegen der Änderung der Hash-Funktion umgebaut werden. Dies kann entweder in einem Nachbereitungsschritt geschehen oder direkt, während der Zustand ausgelesen und wieder eingespielt wird. Die zweite Alternative ist effizienter, weil die Umbauoperation eingespart wird.

4.6.2 Generierte Versenderoutinen

Abbilung 4.10 zeigt zwei Anwendungsklassen `A` und `B`. Nachfolgend wird die für diese Klassen generierte Serialisierungsfunktionalität skizziert. Klasse `A` enthält sowohl finale als auch nicht-finale Instanzvariablen, wobei `A.b` eine referenzwertige finale Instanzvariable ist. Klasse `B` erweitert `A` und fügt unter anderem eine transiente Instanzvariable `B.t` hinzu. Um den Zustand von `B.t` nach der Deserialisierung zu initialisieren, ist in `B` eine Nachbereitungsmethode `restoreAfterUnmarshal()` angegeben.

Abbildung 4.11 zeigt eine vereinfachte Version der generierten Serialisierungsfunktionalität für Klasse `A`. In diesem und den folgenden Code-Beispielen wird von der Ausnahmebehandlung abstrahiert, und das Puffermanagement mit dem Einfügen und der Extraktion von primitiven Datenwerten ist nur in Pseudocode-Blöcken `{*...*}` angedeutet.

Die Methode `marshalReference()` schreibt direkt alle finalen Instanzvariablen von `A` in den Strom, da diese bei der Konstruktion eines neuen Objektes während der Deserialisierung benötigt werden. Der rekursive Teil der Serialisierung ist im Aufruf `writeReference()` zu sehen. Die nicht-finalen Teile von `A` werden in der Methode `marshal()` geschrieben. Beim Schreiben von referenzwertigen Instanzvariablen wird nicht rekursiv die `writeObject()`-Methode des Stroms aufgerufen, sondern `writeReference()`, so dass die Rekursion auf den finalen Teilgraphen des zu serialisierenden Objektgraphen beschränkt wird.

Der Deserialisierungskonstruktor in Abbildung 4.11 erhält den Strom als Argument, aus dem das zu deserialisierende Objekt seinen Zustand initialisiert und zusätzlich die Objektnummer, unter der es zur Zyklenerkennung im Strom geführt wird. Die erste Aktion im Deserialisierungskonstruktor ist die Registrierung des neu konstruierten Objektes unter seiner Objektnummer beim Strom. Das ist notwendig, damit Zyklen, die im finalen Teilgraphen auf das gerade konstruierte Objekt zurückzeigen, aufgelöst werden können. Im konkreten Beispiel könnte das als nächstes direkt im Dese-

```
class A implements Serializable {
   final int x;
   final B b;

   float y;
   B c;
}

class B extends A {
   final int z;
   final B d;

   transient double t;
   A e;

   private void restoreAfterUnmarshal()
      throws IOException, ClassNotFoundException
   {
      t = 2.5 * x;
   }
}
```

Abbildung 4.10: Anwendungsklassen mit Vererbungshierarchie

rialisierungskonstruktor gelesene Objekt der Klasse `B` eine finale Referenz zurück auf das gerade konstruierte Objekt besitzen. In diesem Fall muss das aktuelle Objekt beim Strom registriert sein, noch bevor sein Konstruktor abgeschlossen ist. Offensichtlich kann der Strom das Objekt also nicht selbst registrieren, da er vor Aufruf des Deserialisierungskonstruktors noch keine Objektreferenz besitzt und die Registrierung nach Aufruf des Konstruktors schon zu spät sein kann.

Der Aufruf des Deserialisierungskonstruktors ist gleichbedeutend mit dem Einlesen einer Referenz auf ein Objekt. Beim Lesen der Referenz muss eine teilrekursive Initialisierung durchgeführt werden, bei der die Werte aller finalen Instanzvariablen aus dem Strom gelesen und initialisiert werden. Im obigen Beispiel gehört eine Referenz auf ein Objekt der Klasse `B` zum Zustand, der schon beim Lesen einer Referenz auf ein Objekt der Klasse `A` benötigt wird. Daher wird diese Referenz schon im Deserialisierungskonstruktor gelesen, direkt nachdem sich das Objekt beim Strom registriert hat.

Die Methode `unmarshal()` liest danach alle nicht-finalen Instanzvariablen des Objektes. Sie wird in einem zweiten Durchlauf wieder vom Strom aus aufgerufen. Während dieses Durchgangs wird die Initialisierung der bis dahin nur teilinitialisierten Objekte abgeschlossen.

Abbildung 4.12 zeigt die Serialisierungsfunktionalität für die abgeleitete Klasse `B`. Man erkennt, dass alle Methoden und Konstruktoren ihr Pendant der Oberklasse aufrufen, bevor sie selbst mit der Ausführung beginnen. Insbesondere leitet der Deserialisierungskonstruktor von `B` die übergebene Objektnummer an den Konstruktor der Oberklasse weiter. Dadurch wird sichergestellt, dass die Registrierung des neuen Objektes die allererste Operation ist, die bei der Konstruktion und vor der Initialisierung

```
// Schreiben einer Referenz
public void marshalReference(MarshalStream stream) {
   {* Reserve buffer space for primitive data,
      stream.insert(this.x); *}
   stream.writeReference(this.b);
}

// Schreiben des Objektzustandes
public void marshal(MarshalStream stream) {
   {* Reserve buffer space for primitive data,
      stream.insert(this.y); *}
    stream.writeReference(this.c);
}

// Deserialisierungskonstruktor (Lesen
// einer Referenz)
public A(UnmarshalStream stream, int id) {
   stream.register(this, id);

   {* Request primitive data,
      this.x = stream.extractInt(); *}
   this.b = (B) stream.readReference();
}

// Lesen des Objektzustandes
public void unmarshal(UnmarshalStream stream) {
   {* Request primitive data,
      this.y = stream.extractFloat(); *}
   this.c = (B) stream.readReference();
}
```

Abbildung 4.11: Skizze der generierten Serialisierungsfunktionalität für Klasse `A`.

```
// Schreiben einer Referenz
public void marshalReference(MarshalStream stream) {
   super.marshalReference(stream);

   {* Similar to A.marshalReference() *}
}

// Schreiben des Objektzustandes
public void marshal(MarshalStream stream) {
   super.marshal(stream);

   {* Similar to A.marshal() *}
}

// Deserialisierungskonstruktor (Lesen
// einer Referenz)
public B(UnmarshalStream stream, int id) {
   super(stream, id);

   {* Request primitive data,
      this.z = stream.extractInt(); *}
   this.d = (B) stream.readReference();
}

// Lesen des Objektzustandes
public void unmarshal(UnmarshalStream stream) {
   super.unmarshal(stream);

   this.e = (A) stream.readReference();

   this.restoreAfterUnmarshal();
}
```

Abbildung 4.12: Skizze der generierten Serialisierungsfunktionalität für Klasse `B`.

```
public final Object deepClone(DeepClone helper) {
   Object copy = clone();

   helper.add(this, copy);

   ((A) copy).deepCloneReferences(helper);

   return copy;
}

protected void deepCloneReferences(DeepClone helper) {
   this.c = (B) helper.doDeepClone(this.c);
}
```

Abbildung 4.13: Skizze der generierten Funktionalität für tiefes Kopieren von `A`.

durchgeführt wird.

Die Klasse `B` in Abbildung 4.10 definiert eine Methode `restoreAfterUnmarshal()` zur Initialisierung der transienten Instanzvariable `t`. Diese wird von `B.unmarshal()` in Abbildung 4.12 nach Abschluss der Initialisierung aufgerufen. Nimmt man an, es gäbe noch eine weitere Ableitung `C` von `B`, so würde beim Deserialisieren eines Objektes vom Typ `C` die private Methode `B.restoreAfterUnmarshal()` aufgerufen, bevor die Methode `C.unmarshal()` den zusätzlichen Zustand von `C` initialisiert hätte. Dieses Verhalten ist hinnehmbar, da der Zustand von `B.t` allein durch den in `B` und seinen Oberklassen definierten Zustand bestimmt sein sollte. Um dies zu gewährleisten, sollten aus `restoreAfterUnmarshal()` keine nicht-finalen Methoden desselben Objektes aufgerufen werden. Diese Regel wird vom Übersetzer allerdings nicht erzwungen.

4.6.3 Generierte Kopiermethoden

Abbildung 4.13 zeigt die für tiefes Kopieren zur Klasse `A` hinzugefügte Funktionalität. Auch hier unterscheidet sich der generierte Code für eine serialisierbare Basisklasse von dem erzeugten Code in Unterklassen.

Die Methode `deepClone()` erzeugt zuerst eine flache Kopie des aktuellen Objektes. Dafür wird die von `Object` ererbte `clone()`-Methode verwendet. Als nächstes wird die flache Kopie für Zyklenerkennung bei dem übergebenen Hilfsobjekt registriert. An dieser Stelle sind alle primitivwertigen Instanzvariablen der Kopie bereits korrekt initialisiert. Lediglich referenzwertige Instanzvariablen verweisen noch auf dieselben Objekte wie das Original. Um die referenzierten Objekte ebenfalls zu kopieren, bzw. um die Referenzen in Referenzen auf Objekte des kopierten Graphen umzuschreiben, wird die ebenfalls generierte Methode `deepCloneReferences()` aufgerufen. Diese präsentiert jede referenzwertige Instanzvariable dem Hilfsobjekt, welches entweder eine Kopie des übergebenen Objektes anfertigt oder eine Referenz auf ein korrespondierendes bereits kopiertes Objekt zurückliefert.

Die Methode `deepClone()` ist final in `A`. Die Erweiterung in `B` findet lediglich im Überschreiben von `deepCloneReferences()` statt. Wie in Abbildung 4.14 zu

```
protected void deepCloneReferences(DeepClone helper) {
   super.deepCloneReferences(helper);

   this.e = (A) helper.doDeepClone(this.e);

   this.restoreAfterUnmarshal();
}
```

Abbildung 4.14: Skizze der generierten Funktionalität für tiefes Kopieren von `B`.

sehen, werden nach Aufruf der überschriebenen Version die in `B` hinzugekommenen referenzwertigen Instanzvariablen geklont. Da `B` eine Methode `restoreAfterUnmarshal()` deklariert, wird diese ebenfalls beim tiefen Kopieren nach Abschluss der Initialisierung der Kopie aufgerufen.

4.6.4 Generierte Deskriptorklassen

Die Deskriptor-Klasse für `A` ist in Abbildung 4.15 zu sehen. Sie ist als innere Klasse von `A` realisiert, und wird zusammen mit den Versende- und Kopiermethoden in die Klasse `A` hineingeneriert. Die Methode `unmarshalReference()` erzeugt eine neue Instanz von `A` über den Aufruf des Deserialisierungskonstruktors. Alle anderen Methoden leiten die Aufrufe an Methoden von `A` weiter. Bei einer Klasse, die nicht die Schnittstelle `Transportable` implementiert, könnte die notwendige Funktionalität extern in den Methoden der Deskriptor-Klasse eingebettet werden. Die Deskriptorklasse wird durch die automatische Code-Erzeugung als anonyme innere Klasse derjenigen Klasse realisiert, die sie beschreibt. Der Deskriptor für `B` sieht analog aus.

```
class A {
   ...

   public static
   final TransportDescriptor TRANSPORT_DESCRIPTOR =
      new TransportDescriptor() {
         public Object unmarshalReference(
            UnmarshalStream s, int id)
         { return new A(s, id); }

         public void unmarshal(
            Object obj, UnmarshalStream s)
         { ((A) obj).unmarshal(s); }

         public void marshalReference(
            Object obj, MarshalStream s)
         { ((A) obj).marshalReference(s); }

         public void marshal(
            Object obj, MarshalStream s)
         { ((A) obj).marshal(s); }

         public Object deepClone(
            Object obj, DeepClone helper)
         { return ((A) obj).deepClone(helper); }
      };
}
```

Abbildung 4.15: Der generierte Deskriptor für `A`.

Kapitel 5

Transparente effiziente Kommunikation

In der hier vorgestellten verteilten Programmierumgebung entsteht durch Zusammenschaltung mehrerer virtueller Java-Maschinen eine große verteilte virtuelle Maschine. Hierfür muss das Problem der Zusammenarbeit von Objekten über die Grenzen ihrer virtuellen Maschine hinweg gelöst werden. Die notwendige Kommunikation muss einerseits effizient sein, damit der Reibungsverlust bei den Operationen minimiert wird, welche Objekte aus mehreren Teilen der verteilten Programmierumgebung betreffen. Andererseits muss die Kommunikation für den Programmierer transparent sein, damit dieser Programme für eine verteilte Umgebung schreiben kann, ohne sich um die Umwandlung von Konstrukten der Programmiersprache in explizite Kommunikationsoperationen zu kümmern. In der verteilten Programmierumgebung werden Zugriffe von einem Objekt auf ein anderes dann automatisch zu einer Kommunikationsoperation, wenn bei dem Zugriff eine Adressraumgrenze überschritten wird.

Java-RMI stellt einen näherungsweise transparent entfernten Methodenaufruf für Java-Objekte zur Verfügung. Für die prinzipielle Funktionsweise siehe Abschnitt 2.2.2 der Übersicht über verwandte Arbeiten. Die angebotene Funktionalität legt nahe, dass die RMI-Bibliothek als Basis für eine Programmierumgebung für Rechnerbündel verwendet werden kann, um den Programmfluss eines parallelen Programms transparent auf eine verteilte Umgebung abzubilden. Wie der folgende Abschnitt zeigt, sprechen allerdings sowohl funktionale als auch nichtfunktionale Mängel gegen die Verwendung von RMI in einer Bündelumgebung. Aus diesem Grund ist eine Neuentwicklung des entfernten Methodenaufrufs speziell für Rechnerbündel notwendig.

Dieses Kapitel identifiziert zuerst die Mängel in RMI und stellt danach den Entwurf eines neuen entfernten Methodenaufrufs (KaRMI) vor, der eine effiziente Kommunikation in einem Rechnerbündel ermöglicht und als Basis für eine transparente Verteilung dienen kann. Im zweiten Teil dieses Kapitels wird der entwickelte entfernte Methodenaufruf zu transparent entfernten Klassen erweitert. Entfernte Klassen und deren Instanzen bilden als kleinste verteilbare Einheiten die Grundbausteine der verteilten Programmierumgebung. Entfernte Klassen werden dabei wie reguläre Klassen deklariert, aber zusätzlich mit einer Markierung versehen. Diese Markierung wird von einem Präprozessor ausgewertet und stößt die automatische Erzeugung von entfernt ansprechbaren Klassen an, die den effizienten entfernten Methodenaufruf als Kommu-

nikationsmedium verwenden.

5.1 Einschränkungen bei Java-RMI

Java-RMI wurde als Abstraktionsmittel für die Kommunikation mit Dienstgebern im Internet entworfen. Aus dieser Zielvorgabe resultieren Einschränkungen, die einer Verwendung für Hochleistungsanwendungen im Rechnerbündel entgegenstehen. Einerseits ist dies eine langsame Kommunikationsgeschwindigkeit, andererseits sind es Einschränkungen bezüglich der erreichbaren Transparenz. Beide Gesichtspunkte werden im Folgenden untersucht.

5.1.1 Effizienz

Auf Weitverkehrsverbindungen im Internet sind Latenzzeiten von etlichen Millisekunden für den reinen Nachrichtenaustausch normal. Aus diesem Grund spielt die Effizienz beim Entwurf einer Abstraktionsschicht für die Kommunikation dort eine untergeordnete Rolle. Die Zeit für die reine Nachrichtenübertragung wird in jedem Fall die benötigte Gesamtzeit für eine Interaktion dominieren. Im Gegenzug muss bei der Kommunikation im Internet besonderes Augenmerk auf die Behandlung möglicher Fehlerzustände gelegt werden, die auftreten, wenn einzelne Rechner nicht erreichbar sind oder die Verbindung während der Übertragung zusammenbricht. Insgesamt können drei Ursachen für die schlechte Effizienz von Java-RMI in Hochgeschwindigkeitsnetzwerken wie folgt identifiziert werden.

Fixierung auf TCP/IP: Die einzige im Internet relevante Übertragungstechnologie ist TCP/IP. Java-RMI verwendet daher TCP/IP-Sockets als Schnittstelle zur Transporttechnologie. Technologien, die keine Socket-Schnittstelle anbieten, sind daher mit Java-RMI nicht verwendbar oder aber es muss eine Socket-Emulation verwendet werden. Hochleistungsnetzwerke haben in der Regel keine Socket-Schnittstelle, oder eine Socket-Schnittstelle wird mit deutlichen Geschwindigkeitseinbußen emuliert. Damit ist Java-RMI nicht bzw. nur eingeschränkt für die Kommunikation im Rechnerbündel einsetzbar. Die Fixierung auf TCP/IP ist dabei kein Implementationsdetail von Java-RMI, sondern schlägt bis zur Anwendungsschnittstelle durch. So kann die Anwendung Objekte direkt an sogenannten Ports exportieren, die aus der Socket-Schnittstelle stammen. KaRMI abstrahiert von Port-Nummern mittels sog. Export-Punkte und führt steckbare Kommunikationstechnologien ein, über die, transparent für die Anwendung, entweder das Hochgeschwindigkeitsnetzwerk im Rechnerbündel oder TCP/IP für die Kommunikation mit externen Ressourcen verwendet werden kann. Die folgenden Abschnitte 5.2.1 und 5.2.2 präsentieren die neuen Ideen.

Entwurfsbedingte Probleme: Aufgrund der Aufgabenverteilung über die Schichten der RMI-Bibliothek kommt es zum Anlegen vieler temporärer Objekte während der Durchführung eines entfernten Methodenaufrufs. Die Stellvertreterschicht in Java-RMI verpackt die Argumente eines entfernten Methodenaufrufs in eine generische

Datenstruktur, um sie durch die Referenzschicht an die Transportschicht weiterreichen zu können. Erst in der Transportschicht werden die Argumente mit der Java-Objektserialisierung in ein für die Netzwerkübertragung geeignetes Format konvertiert. Das Anlegen neuer Objekte ist eine relativ teure Operation, insbesondere wenn man bedenkt, dass ein temporäres Objekt zusätzlich den Speicherbereiniger in Anspruch nimmt. Bei einer Kommunikationslatenz von einigen Millisekunden im Internet fällt dieser Zusatzaufwand nicht ins Gewicht. Bei einer Latenz von wenigen Mikrosekunden, wie sie bei der Hochgeschwindigkeitskommunikation im Rechnerbündel erreicht wird, fällt jede Zusatzbelastung des Rechenknotens ins Gewicht. KaRMI vermeidet daher das Anlegen temporärer Objekte mit Hilfe einer geänderten Aufgabenverteilung der einzelnen Schichten, die in Abschnitt 5.2.3 beschrieben wird.

Lokalität: Java-RMI enthält keine Optimierungen für potenziell entfernte Aufrufe, die sich aber zur Laufzeit an ein Objekt in derselben virtuellen Maschine richten. In einer verteilten Laufzeitumgebung wird durch Lokalitätsoptimierung der Effekt angestrebt, dass Komponenten, die sich häufig gegenseitig aufrufen, auf demselben Knoten angelegt oder zur Laufzeit auf denselben Knoten migriert werden. Durch Verbesserung der Lokalität der Aufrufe erhofft man sich einen Geschwindigkeitsgewinn, da die Netzwerklatenz bei den jetzt lokal ablaufenden Aufrufen eingespart werden kann. RMI berücksichtigt jedoch die Lokalität von Aufrufen nicht. Daher wird auch bei einem lokalen Aufruf die gesamte Maschinerie für den Fernzugriff bis zum Protokollstapel des Betriebssystems aktiviert. Demzufolge kann durch Lokalitätsoptimierung die Programmlaufzeit nur unwesentlich verkürzt werden. Für das RMI-Szenario ist dies tolerabel, da bei einem Fernzugriff auf einen Dienstgeber im Internet das Klientenprogramm nie in derselben virtuellen Maschine wie der Dienstgeber abläuft. In einer verteilten Laufzeitumgebung für parallele Programme können aber sehr wohl verschiedene Komponenten in derselben virtuellen Maschine ausgeführt werden. Der in dieser Arbeit entworfene Fernaufruf nutzt daher vorhandene Lokalität aus. Im besten Fall benötigt ein lokaler Zugriff auf ein potenziell entferntes Objekt nur eine einzige zusätzliche Indirektion. Mehr dazu folgt in Abschnitt 5.2.4.

5.1.2 Transparenz

Der Fernzugriff mit Java-RMI ist weder transparent, noch lässt sich auf Basis von Java-RMI eine Programmierumgebung mit transparentem Fernzugriff realisieren. Zu diesem Problem kommt es, weil Java-RMI die durch den Fernaufruf entstehenden verteilten Kontrollfäden nicht ausreichend berücksichtigt. Beim Fernaufruf entsteht aus einem lokalen Kontrollfaden, dessen Wirkungsbereich auf eine virtuelle Maschine beschränkt ist, ein verteilter Kontrollfaden, dessen Aufrufstapel über mehrere Maschinen verteilt ist und dessen Ausführungspunkt von einer Maschine zu einer anderen wechselt. Da die Synchronisation in Java an den ausführenden lokalen Kontrollfaden gebunden ist, RMI darauf aber keine Rücksicht nimmt, kommt es zu Unverträglichkeiten von Synchronisation und Fernaufrufen. Für ein Programm, das bei der Verwendung einer einzigen virtuellen Maschine korrekt synchronisiert wäre, würde der Übergang zu einer mit RMI verteilten Umgebung möglicherweise zu einer Verklemmung führen, weil ein verteilter Kontrollfaden seine eigenen Sperren nicht mehr überwinden

kann, wenn zwischen der Sperranforderung und dem Wiedereintritt eine Folge von Fernaufrufen liegt (Details werden in Abschnitt 5.3.1 besprochen). Auch die Anforderung einer Sperre an einem entfernt liegenden Objekt in einem synchronisierten Block hat nicht den erwarteten Effekt. Dieses Manko lässt sich in einer RMI-Anwendung nicht umgehen, da kein Einfluss auf die Art und Weise genommen werden kann, wie lokale Kontrollfäden zu verteilten Kontrollfäden zusammengesetzt werden. Wie die Abschnitte 5.3 und 5.4 zeigen werden, berücksichtigt KaRMI dagegen die Abhängigkeiten zwischen verteilten Kontrollfäden und der Java-Synchronisationsemantik. In KaRMI ist sowohl eine entfernte Sperranforderung möglich als auch der Wiedereintritt in einen bereits angeforderten Monitor. Dabei spielt es keine Rolle, ob zwischen Monitoranforderung und Wiedereintritt Fernaufrufe stattgefunden haben.

5.2 Entwurfsverbesserungen und Optimierungen

Dieser Abschnitt beschreibt Eigenschaften des Entwurfs und verwendete Techniken, die einen effizienten Einsatz von KaRMI in Rechnerbündeln ermöglichen. Das Resultat ist ein modulares Paket für einen entfernten Methodenaufruf, mit dem sich die Kommunikationleistung in einem Rechnerbündel ausschöpfen lässt.

5.2.1 Steckbare Transporttechnologien

Hochgeschwindigkeitsnetzwerke für Rechnerbündel bringen in der Regel ihre eigenen Kommunikationsbibliotheken mit, die einen schnellen, direkten Kommunikationspfad von der Anwendung zur Hardware eröffnen. Da diese Kommunikation i. d. R. das Betriebssystem nicht involviert (*user-level* Kommunikation), ist diese Kommunikationshardware auch nicht über die Socket-Schnittstelle des Betriebssystems ansprechbar. Damit eine Bibliothek für entfernten Methodenaufruf in einem Rechnerbündel den schnellen Kommunikationspfad zur Netzwerkhardware nutzen kann, muss sie modular aufgebaut sein, damit verschiedene Kommunikationstechnologien eingesteckt werden können. Je nachdem welche Netzwerkhardware in einem Rechnerbündel vorrätig ist, wird dann das passende Modul für die Anbindung geladen.

In KaRMI kapselt ein Technologieobjekt neben der eigentlichen Kommunikation auch das in der Kommunikationstechnologie verwendete Adressierungsschema. Das Adressierungsschema für die TCP/IP-Kommunikationstechnologie bilden die wohlbekannten IP-Adressen mit ihren zugehörigen Ports. In einer auf ein Bündel beschränkten Technologie werden aber einfachere Adressierungschemata wie beispielsweise die Knotennummer verwendet, um den Kommunikationspartner ausfindig zu machen. Damit mehrere verschiedene Kommunikationstechnologien gleichzeitig benutzt werden können, führt KaRMI eine Abstraktionsebene für die Adressierung von Kommunikationsendpunkten ein. Die abstrakte Oberklasse für einen allgemeinen Kommunikationsendpunkt wird in jeder konkreten Technologieimplementierung entsprechend dem nativen Adressierungsschema der Kommunikationstechnologie spezialisiert.

An seiner Schnittstelle zur KaRMI-Bibliothek stellt ein Technologieobjekt Verbindungen zu einem Kommunikationspartner zur Verfügung, der über dieselbe Netzwerktechnologie verfügt und daher in das Adressierungsschema des Technologieobjektes

passt. Anders als bei herkömmlichen Netzwerkverbindungen des Betriebssystems, die eine unstrukturierte Byte-Folge transportieren, sitzen an den Verbindungsendpunkten von KaRMI-Technologieobjekten Paare aus Serialisierungs- und Deserialisierungsobjekten. Dadurch ist es möglich, über solche Verbindungen nicht nur primitive Datentypen, sondern auch Objektstrukturen zu übertragen, was für die Argumentübergabe und die Ergebnisrückgabe von entfernten Methodenaufrufen wichtig ist. Der Serialisierer sitzt somit innerhalb der Transportschicht, und dass ist insofern naheliegend, als er an der Erzeugung des Übertragungsprotokolls wesentlich beteiligt ist. Die Verbindungsendpunkte von KaRMI-Technologien sind aus drei Gründen bereits mit Serialisierungsfunktionalität versehen:

– Jede Kommunikation in KaRMI benötigt potenziell Serialisierungsfunktionalität.

– Der verwendete Serialisierungsmechanismus ist technologieneutral, kann aber aus Effizienzgründen eine unterschiedliche Anbindung an die Transportschicht verwenden, je nachdem, ob diese eine paket- oder eine stromorientierte Schnittstelle anbietet (vgl. 4.5.2). Das Technologieobjekt kann somit den Serialisierungsmechanismus optimal für seine Netzwerkschnittstelle konfigurieren.

– Verbindungen werden vom Technologieobjekt verwaltet und können für mehrere aufeinanderfolgende entfernte Methodenaufrufe wiederverwendet werden. Durch Wiederverwendung der Verbindung zusammen mit dem zugehörigen Paar aus Serialisierer und Deserialisierer ist eine besonders effiziente Typcodierung bei der Serialisierung möglich (vgl. 4.2.1). In bereits über dieselbe Verbindung abgewickelten entfernten Methodenaufrufen hat sich ein gemeinsames Typalphabet aufgebaut, das eine besonders effiziente Typcodierung ermöglicht und für weitere entfernte Methodenaufrufe wiederverwendet wird. Die Schnittstelle eines Technologieobjektes bilden die in Abschnitt 5.2.3 eingeführten Aufrufkontexte.

5.2.2 Exportpunkte statt TCP/IP-Ports

In Java-RMI werden TCP/IP-Ports an der Schnittstelle zur Anwendung verwendet, um Objekte an bekannten Ports für die Dienstfindung zu exportieren. Die Portnummer ist dabei – neben dem Knoten und der Objektnummer – Teil des Namens, unter welchem ein Objekt exportiert wird. KaRMI erlaubt neben TCP/IP das Einstecken beliebiger Kommunikationstechnologien, wie im vorangegangenen Abschnitt beschrieben. Da der Anwendung nicht bekannt ist, welche Kommunikationstechnologien auf einem Knoten verfügbar sind, ist ein abstrakteres Adressierungsschema notwendig, das TCP/IP-Ports zur Benennung des Kommunikationsendpunktes ablöst.

Die technologieunabhängige Form, mit der KaRMI Objekte eines Rechners unter verschiedenen Namen exportiert, ist der Exportpunkt. Ein Exportpunkt wird unabhängig von den auf einem Rechenknoten installierten Transporttechnologien konfiguriert und ist über eine Nummer identifiziert. Jede Transporttechnologie kann sich an den Exportpunkten eines Knotens registrieren und muss dabei eine Übersetzung in das eigene

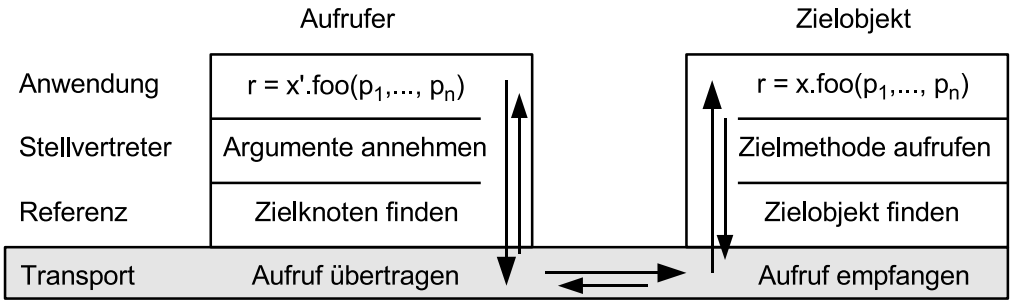


Abbildung 5.1: Schichten beim entfernten Methodenaufruf.

Adressierungschema vornehmen. Das Technologieobjekt für TCP/IP führt beispielsweise eine Abbildung des Exportpunktes auf einen bestimmten TCP/IP-Port durch und öffnet diesen Port für die Annahme von Verbindungen. Über diesen Port lässt sich dann entfernt auf Objekte zugreifen, welche unter dem zugehörigen Exportpunkt exportiert werden.

Durch die Indirektionsstufe der Exportpunkte ist die Anwendung vom direkten Umgang mit technologieabhängigen TCP/IP-Ports enthoben. Damit kann eine Anwendung ohne Kenntnis der auf dem Rechnerbündel verfügbaren Kommunikationtechnologien unterschiedliche Namensräume zum Export von Objekten verwenden. Die Zuordnung zu konkreten Ports ist nicht hart in die Anwendung eincodiert, sondern kann nachträglich mittels Konfiguration geändert oder für andere Kommunikationstechnologien angepasst werden.

5.2.3 Effizienz durch Aufrufkontexte

Ein entfernter Methodenaufruf durchläuft im wesentlichen drei Schichten wie in Abbildung 5.1 zu sehen. Die Anwendung ruft eine Methode an einem generierten Stellvertreterobjekt `x'` (dem Objekt, das die entfernte Referenz kapselt) auf und stößt damit den entfernten Methodenaufruf an. Der Stellvertreter `x'` nimmt die Argumente des Aufruf entgegen und codiert die aufgerufene Methode `foo()`. Die Referenzschicht findet den Rechenknoten, auf dem sich das Zielobjekt `x` befindet. Daraufhin wird das Technologieobjekt der Transportschicht aktiviert, welches eine Verbindung zum Zielrechner herstellt, um den Methodenaufruf mitsamt seinen Argumenten zu übermitteln. Auf der entfernten Seite werden die Schichten in umgekehrter Reihenfolge durchlaufen. Die Transportschicht nimmt den Aufruf entgegen, die Referenzschicht findet das angesprochene entfernte Objekt `x` und die Stellvertreterschicht ruft schließlich die angeforderte Methode `x.foo()` auf. Das Ergebnis des entfernten Methodenaufrufs wird in entgegengesetzter Richtung zurückübermittelt.

Beim Entwurf stellt sich die Frage, was mit der Argumentliste des Aufrufs passiert, während dieser seinen Weg durch die oben beschriebenen drei Schichten nimmt. Klar ist, dass die Stellvertreterschicht der Anwendung die Illusion eines Methodenaufrufs mit Argumentübergabe vermittelt. Die Stellvertreterschicht nimmt die Argumentliste mit Hilfe des programmiersprachlichen Konstrukts des Methodenaufrufs entgegen und liefert das Resultat als Rückgabewert zurück. Die Kommunikationsoperation findet hinter den Kulissen statt. Klar ist auch, dass die Transportschicht für das Leitungs-

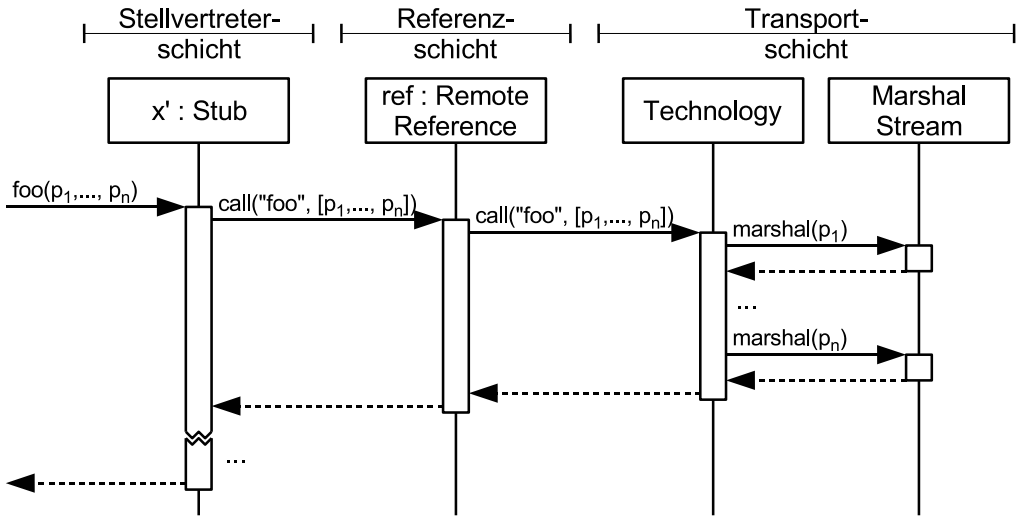


Abbildung 5.2: Ablauf beim Absenden des Aufrufs als Sequenzdiagramm (Java-RMI).

protokoll und damit für die Netzwerkübertragung der Methodenidentifikation, der Argumentliste und des Rückgabewertes zuständig ist.

In Java-RMI wird die Argumentliste für den Transport von der Stellvertreter- über die Referenz- hin zur Transportschicht in eine generische Datenstruktur[1] verpackt, die bei jedem Aufruf neu angelegt wird. Abbildung 5.2 zeigt, wie diese Datenstruktur in generischen Methoden (die nicht mehr speziell für die Methode des Zielobjektes generiert wurden) durch die Referenzschicht an die Transportschicht weitergeleitet wird. Die in eine generische Datenstruktur verpackten Argumente sind in Abbildung 5.2 durch die Notation $[p_1, \ldots, p_n]$ angedeutet, welche die Erzeugung eines dynamischen Feldes symbolisieren soll. Allerdings verursacht das Verpacken der Argumentliste durch die Erzeugung des Feldes (und evtl. von Behälterobjekten für primitive Datentypen) und die daraus resultierende Beanspruchung des Speicherbereinigers unnötigen Mehraufwand, da die Datenstruktur nur für den Transport der Argumente des entfernten Methodenaufrufs durch die Schichten der Bibliothek benötigt wird.

KaRMI vermeidet die temporäre Erzeugung von Objekten pro Fernaufruf durch einen leicht geänderten Ablauf des Fernaufrufs, ohne dabei die Kapselung der Funktionalität in die oben genannten drei Schichten aufzugeben. Ein Fernaufruf in KaRMI läuft in zwei Phasen ab, die in Abbildung 5.3 zu sehen sind. In der ersten Phase wird ein Aufrufkontext ausgewählt und initialisiert, während in der zweiten Phase der Fernaufruf über diesen Aufrufkontext abgewickelt wird.

Phase 1: Die Auswahl des Aufrufkontextes läuft wie die komplette Abwicklung des Fernaufrufs im Java-RMI-Modell ab, nur dass die Argumente des Fernaufrufs nicht durch die Schichten der Bibliothek hindurchgereicht werden. Der Aufruf kann also bei Erreichen der Transportschicht noch nicht abgeschickt werden, da die Argumente in der Stellvertreterschicht verblieben sind. Stattdessen wird ein Aufrufkontext mit allen notwendigen Informationen initialisiert und als Ergebnis der ersten Phase an die Stellvertreterschicht zurückgegeben. Für die Initialisierung des Aufrufkontextes

[1] ein Feld aus Objektreferenzen, das bei Bedarf mit Behältern für primitive Werte gefüllt wird.

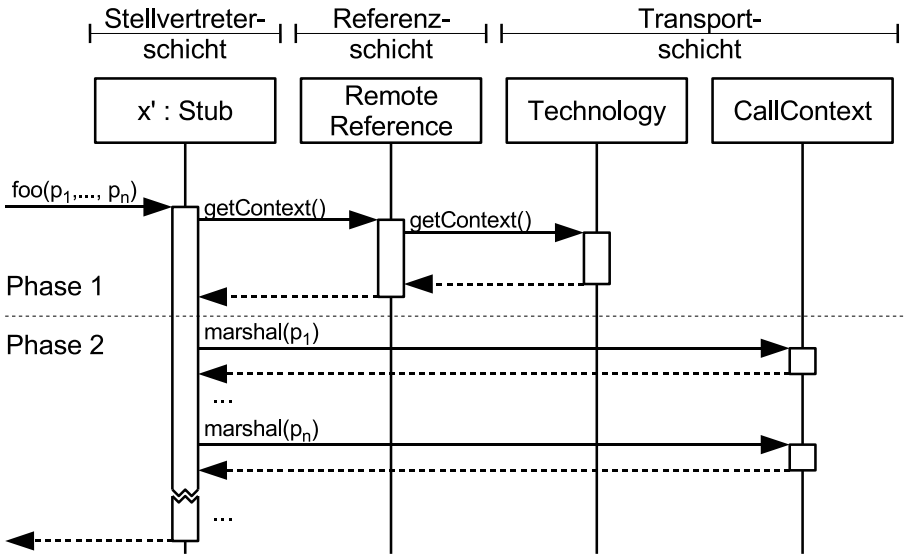


Abbildung 5.3: Ablauf beim Absenden des Aufrufs als Sequenzdiagramm (KaRMI).

ist keine Objekterzeugung notwendig, da der Kontext ebenso wie die Netzwerkverbindung einen Aufruf überdauert und für Folgeaufrufe wiederverwendet wird.

Phase 2: Nachdem die Stellvertretermethode einen Aufrufkontext erhalten hat, führt sie den Fernaufruf unter eigener Regie durch. Dazu ruft sie sequentiell Methoden aus der Schnittstelle des Aufrufkontextes nach einem vordefinierten Protokoll auf. Diese Methoden haben die folgenden Bedeutungen: Starten des Aufrufs, Übergabe eines Arguments, Abschicken des Aufrufs, Erwarten des Ergebnisses, Anfordern des Ergebnisses oder der aufgetretenen Ausnahmebedingung und schließlich der Abschluss des Aufrufs.[2] Dieses Aufrufprotokoll macht ein Verpacken der Argumente und des Ergebnisses eines entfernten Methodenaufrufs in eine generische Datenstruktur für den Transport durch die Schichten überflüssig, da jedes Argument einzeln an den Aufrufkontext übertragen wird und das Resultat direkt vom Aufrufkontext zurückgeliefert wird. Auch innerhalb des Aufrufkontextes ist kein Zwischenspeichern der Argumente notwendig, da sie direkt an den während der Initialisierungsphase angeschlossenen Serialisierer weitergeleitet werden können. Primitive Werte werden in spezialisierten Methoden übergeben, so dass auch hier kein aufwendiges Ein- und Auspacken notwendig wird.

Da das Leitungsprotokoll letztendlich im Aufrufkontext zusammengebaut wird, gehört seine Implementierung zur Transportschicht und kann technologieabhängig realisiert werden. Die Schnittstelle des Aufrufkontextes, die das allgemeine Ablaufprotokoll eines Fernaufrufs mit Argumentübergabe und Ergebnisrückgabe beschreibt, ist jedoch technologieunabhängig und kann daher direkt aus der Stellvertreterschicht heraus benutzt werden. Die Referenzschicht, welche nur in der ersten Phase beteiligt ist, arbeitet lediglich beim Auffinden des richtigen Aufrufkontextes mit, wird aber während der Durchführung des Aufrufs nicht benötigt. Durch diesen zweiphasigen Ent-

[2] Davon ist in Abbildung 5.3 lediglich die Übergabe der Argumente gezeigt.

wurf wird in KaRMI das Durchschleifen von Argumentliste und Ergebnis durch die Referenzschicht vermieden, wodurch sich eine temporäre Objekterzeugung pro Aufruf gänzlich einsparen lässt.

5.2.4 Erkennung und Ausnutzung von Lokalität

In einem verteilten Programm können Fernaufrufe zwischen häufig miteinander interagierenden Komponenten vermieden werden, indem diese auf denselben Knoten gelegt werden. Ein solcher Optimierungsschritt heißt Lokalitätsoptimierung. Die Lokalitätsoptimierung ändert nichts an der Semantik des Programms, sondern wird mit der Hoffnung auf eine erhöhte Geschwindigkeit durchgeführt. Da nach der Lokalitätsoptimierung die Zielkomponente auf demselben Knoten wie die aufrufende Komponente liegt, wird aus einem entfernten ein lokaler Aufruf. Dieser lokale Aufruf ist aber *potenziell entfernt*, da die Komponenten auch dann noch miteinander interagieren könnten, wenn sie wieder voneinander getrennt würden. Lokalitätsoptimierung ist nur dann effektiv, wenn ein lokaler, aber potenziell entfernter Methodenaufruf in Bezug auf seine Latenz in der Größenordnung eines lokalen Methodenaufrufs liegt. Um dies zu erreichen, verfügt KaRMI über einen besonders effizienten Mechanismus zur schnellen Durchführung von lokalen, aber potenziell entfernten Methodenaufrufen. Dieser besteht aus zwei Teilen, die in den folgenden Abschnitten beschrieben werden.

Direktaufruf aus der Stellvertreterschicht

Die Stellvertreterschicht in KaRMI kann feststellen, ob ein referenziertes Objekt lokal liegt. In diesem Fall kann die Stellvertretermethode zur Durchführung des Aufrufs statt eines Aufrufkontextes eine direkte Referenz auf das Zielobjekt erlangen. Die schnellste Möglichkeit des Aufrufs bestünde jetzt im direkten Aufruf der entsprechenden Methode des Zielobjektes unter Weiterleitung der übergebenen Argumente. Leider dürfen die Argumente nicht direkt an die Methode des Zielobjektes übergeben werden, weil sich ansonsten die Semantik des Programms ändern könnte. (Gleiches gilt für die Rückgabe des Resultats.) In einem Fernaufruf werden die Argumente bei der Übertragung auf einen anderen Knoten nämlich kopiert. Das heißt, dass der Empfänger die Argumente eines entfernten Methodenaufrufs ändern kann, ohne dass der Aufrufer dies nach Rückkehr der Methode feststellen könnte, wenn nicht das geänderte Argument explizit als Resultat zurückgeliefert wird. Um überhaupt Lokalitätsoptimierung mit einem Programm durchführen zu können, muss seine Semantik ortsunabhängig sein. Das bedeutet, dass das Ergebnis der Berechnung nicht davon abhängen darf, wie die Komponenten des Programms über die Rechenknoten verteilt sind. Diese Forderung verbietet eine Abhängigkeit der Semantik der Parameterübergabe vom Ort des Zielobjektes. Als Konsequenz muss aus Gründen der Ortsunabhängigkeit die Kopiersemantik auch bei lokal ablaufenden, aber potenziell entfernten Methodenaufrufen durch Kopieren der Argumente simuliert werden.

Bei einem Fernaufruf eines lokal liegenden Zielobjektes werden die Argumente des Aufrufs also vom Stellvertreterobjekt kopiert. Dann wird mit diesen Kopien als Argument direkt die Methode des Zielobjektes aufgerufen. Das Resultat wird anschließend im Stellvertreter ebenfalls kopiert und an den Aufrufer zurückgegeben. Für das schnel-

le Kopieren der Argumente kommt jener Teil der in Kapitel 4 beschriebenen schnellen Objektserialisierung zum Einsatz, mit welchem Objektgraphen direkt im Speicher dupliziert werden können. Als Resultat erhält man eine tiefe Kopie des Graphen, wie sie entstünde, wenn die Objektstruktur mit der Serialisierung geschrieben und anschließend wieder eingelesen würde (vgl. Abschnitt 4.6.3). Mit dieser Funktionalität der schnellen Objektserialisierung kann effizient das für die Erhaltung der Programmsemantik notwendige Kopieren der Argumentliste und des Resultats von lokal ablaufenden potenziell entfernten Methodenaufrufen durchgeführt werden.

Optimierung beim Kopieren der Argumente

Der Aufwand für das Anfertigen einer tiefen Kopie der in einem lokalen Aufruf übergebenen Argumente ist nicht zu vernachlässigen. Da das Kopieren einzig und allein zur Erhaltung der Fernaufrufsemantik dient, stellt sich die Frage, unter welchen Umständen auf das Kopieren verzichtet werden kann. Steht zur Übersetzungszeit (wenn das Stellvertreterobjekt generiert wird) fest, dass für einen Fernaufruf weder eines der Argumente noch das Resultat eine explizite Kopieroperation erfordert, wird durch den Generator ein Direktaufruf der Zielmethode eingesetzt. Ein potenziell entfernter Aufruf an einem lokal liegenden Zielobjekt kommt dann mit einem einzigen Test und einer zusätzlichen Indirektion aus. Im lokalen Fall erreicht diese Optimierung eine Aufruflatenz in der Größenordnung eines regulären Methodenaufrufs. Die folgenden Abschnitte untersuchen Fälle, in denen auf die Erstellung von Kopien verzichtet werden kann.

Primitive Typen: In Java werden primitive Typen bei der Parameterübergabe in Methodenaufrufen immer als Wert übergeben. Eine Methode erhält also standardmäßig bei Übergabe eines primitiven Arguments eine Kopie. Demzufolge muss bei primitivwertigen Argumenten kein Zusatzaufwand für das Anlegen einer Kopie betrieben werden (RMI tut dies trotzdem). Alle primitivwertigen Argumente werden daher nicht explizit kopiert, sondern direkt aus dem Stellvertreter an die Zielmethode weitergeleitet.

Klassen mit Wertsemantik: In Abschnitt 4.3.1 wurden bereits Eigenschaften von Klassen untersucht, die eine Einsparung von Zyklentests während der Objektserialisierung erlauben. Bei diesen sogenannten Wertklassen sind Referenzen und Kopien austauschbar. Während der Serialisierung wird diese Eigenschaft ausgenutzt, um Zyklentests einzusparen, da es keine Rolle spielt, wenn beim Empfänger statt zweier Referenzen auf dasselbe Objekt zwei identische Kopien ausgeliefert werden. Beim Optimieren von potenziell entfernten, aber zur Laufzeit lokalen Methodenaufrufen kann diese Eigenschaft in umgekehrter Richtung ausgenutzt werden: Anstatt eine Kopie des Wertobjektes an die Zielmethode zu übergeben, kann genauso gut eine direkte Referenz auf das Argument weitergeleitet werden. Die Semantik des Programms wird dadurch nicht verändert, weil das Argument während des Methodenaufrufs nicht modifiziert werden kann.

Beide vorgestellten Optimierungen erlauben es, einen potenziell entfernten Aufruf

an einem lokal liegenden Objekt dann direkt durchzuführen, wenn aufgrund des Typs der Parameter und des Rückgabewertes ein Kopieren vermieden werden kann. In diesem Fall erfordert ein solcher nur potenziell entfernter Methodenaufruf lediglich einen zusätzlichen Test und eine Indirektion gegenüber einem regulären lokalen Methodenaufruf.

5.3 Transparent maschinenüberspannende Kontrollfäden

Ein System für entfernte Methodenaufrufe erweitert einen lokalen Kontrollfaden zu einem verteilten bzw. maschinenüberspannenden Kontrollfaden, indem die Semantik des lokalen Methodenaufrufs auf eine verteilte Umgebung übertragen wird. Die folgenden Abschnitte untersuchen die daraus resultierenden Konsequenzen und leiten prinzipielle Anforderungen an die Semantik von Kontrollfäden unter Berücksichtigung von Fernaufrufen ab. Es werden nur reguläre synchrone Methodenaufrufe betrachtet, da asynchrone Aufrufe eine zusätzliche Quelle von Nebenläufigkeit darstellen und daher nicht zum Konzept paralleler Kontrollfäden in Java passen.

Lokaler Methodenaufruf

Bei einem lokalen Methodenaufruf wird die aufrufende Methode suspendiert und mit der Abarbeitung der aufgerufenen Methode in einem neuen Kontext begonnen. Erst wenn die aufgerufene Methode vollständig abgearbeitet ist, wird das Resultat in den aufrufenden Kontext übernommen und die Abarbeitung der ursprünglichen Methode an genau der Stelle wieder aufgenommen, an der sie durch den Methodenaufruf unterbrochen wurde. Die gesamte Abfolge findet dabei in demselben Kontrollfaden statt. Die Methodenkontexte (in denen lokale Variablen gespeichert werden) bilden einen Stapel, so dass immer nur der oberste Kontext, welcher zur aktuell bearbeiteten Methode gehört, sichtbar ist. Das aktive Ende eines Kontrollfadens markiert den Punkt im Programm, an welchem die Abarbeitung gerade steht. Der Stapel aus Aufrufkontexten repräsentiert die Aufrufhistorie und liefert den Rücksprungpunkt, wenn das Ende einer Methode erreicht ist.

Entfernter Methodenaufruf

Beim entfernten Methodenaufruf wird bezüglich des Programmablaufs das Verhalten eines lokalen Methodenaufrufs nachgebildet. Der Unterschied besteht nur darin, dass sich das Ziel des Aufrufs auf einer anderen Maschine als der Aufrufer befindet. Auch beim entfernten Methodenaufruf wird die aufrufende Methode so lange suspendiert, bis die entfernt aufgerufene Methode vollständig abgearbeitet ist und ein Ergebnis zurückliefert. Da ein Aufrufkontext immer dort liegt, wo die zugehörige Methode abgearbeitet wird, führt der Fernaufruf zu einem verteilten Stapel aus Aufrufkontexten. Gleichzeitig wechselt das aktive Ende eines Kontrollfadens im Fernaufruf vom aufrufenden Rechner zu demjenigen mit der aufgerufenen Methode. Aus der Vogelperspektive betrachtet, sieht man einen verteilten Kontrollfaden mit einem verteilten

Stapelspeicher und einem aktiven Ende, welches beim Aufruf und der Rückkehr aus entfernten Methoden die Maschine wechseln kann.

Maschinenüberspannende Kontrollfäden

Entfernte Methodenaufrufe führen auf natürliche Weise zur Sichtweise von maschinenüberspannenden Kontrollfäden. Aufgrund der starken Abstraktion von der Kommunikation durch das Konzept des Fernaufrufs erscheinen die Kontrollfäden eines Programms so, als könne ihr aktives Ende die ausführende Maschine wechseln. Da in Java ein Kontrollfaden in Wirklichkeit aber an diejenige virtuelle Maschine gebunden ist, in der er erzeugt wurde, muss sich der konzeptuell maschinenüberspannende Kontrollfaden abschnittsweise aus mehreren lokalen Kontrollfäden zusammensetzen.

Blickt man auf die Details, so wird der lokale Kontrollfaden, welcher einen Fernaufruf anstößt, suspendiert, sobald er den Aufruf abgeschickt hat. Während der Fernaufruf läuft, repräsentiert der inaktive Kontrollfaden lediglich ein weiter unten liegendes Segment des verteilten Stapels aus Aufrufkontexten. Da der aufrufende Kontrollfaden selbst nur auf die Rückkehr des Fernaufrufs wartet, trägt er so lange nicht zum Fortkommen des Programms bei, bis der Fernaufruf zurückkehrt. Die entfernt aufgerufene Methode selbst wird in einem neuen, durch die Bibliothek bereitgestellten, Kontrollfaden abgearbeitet. Sein aktives Ende repräsentiert während der Laufzeit des Fernaufrufs das aktive Ende des konzeptuell maschinenüberspannenden Kontrollfadens. In Wirklichkeit wechselt das aktive Ende eines maschinenüberspannenden Kontrollfadens also im Fernaufruf nicht die Maschine, sondern der aufrufende Kontrollfaden wird suspendiert, während ein neuer den Rumpf der entfernt aufgerufenen Methode ausführt. In der Kette der lokalen Kontrollfäden, aus der sich der konzeptuell maschinenüberspannende Kontrollfaden zusammensetzt, ist immer nur der letzte Kontrollfaden aktiv.

5.3.1 Konsequenzen der segmentweisen Zusammensetzung

Solange ein Programm nur einen einzigen Kontrollfaden benutzt und somit keine Koordination zwischen mehreren (maschinenüberspannenden) Kontrollfäden notwendig ist, erfüllt die segmentweise Zusammensetzung eines maschinenüberspannenden Kontrollfadens aus lokalen Kontrollfäden alle Erwartungen. Arbeiten allerdings mehrere maschinenüberspannende Kontrollfäden in einem parallelen Programm zusammen, wird Koordination wichtig. In einem verteilten Programm, das Koordinationsprimitive verwendet, wird aber die Zusammensetzung seiner maschinenüberspannenden Kontrollfäden aus einzelnen lokalen Kontrollfäden sichtbar und führt zu unvorhergesehenem Verhalten. Java-RMI leidet unter genau diesem Problem.

In Java gibt es Koordinationsprimitive zum Schutz kritischer Abschnitte und zur Kommunikation zwischen Kontrollfäden. Ein geschützter kritischer Abschnitt kann nur von einem einzigen Kontrollfaden betreten werden. Dazu wird zu Beginn eines kritischen Abschnitts ein Monitor angefordert, um die Sperre zu setzen, und vor Verlassen des Abschnitts der Monitor freigegeben, um die Sperre wieder zu lösen. Versucht ein weiterer Kontrollfaden einen Abschnitt zu betreten, in dem noch ein anderer Kontrollfaden aktiv ist, wird der zweite solange blockiert, bis der erste den geschütz-

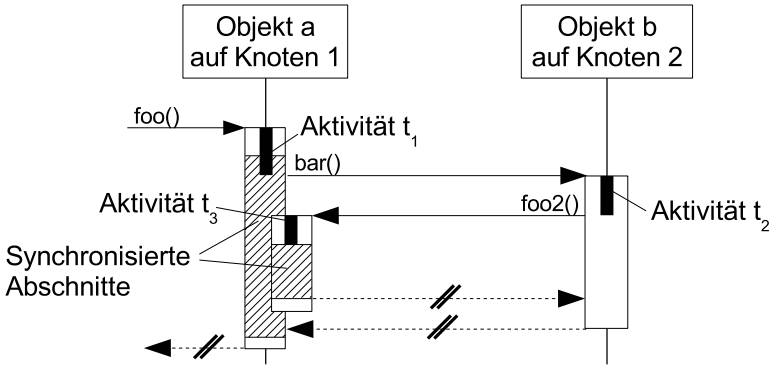


Abbildung 5.4: Kritische Situation beim Fernaufruf mit Potenzial zur Verklemmung.

ten Bereich verlässt und die Sperre freigibt. Ein Kontrollfaden darf allerdings von ihm selbst geschützte Bereiche – beispielsweise in rekursiven Aufrufen – mehrfach betreten. In diesem Fall hält der Kontrollfaden schon den zu einem geschützten Bereich gehörenden Monitor. Die Sperre wird in diesem Fall erst dann freigegeben, wenn der Monitor sooft verlassen wurde, wie er angefordert wurde (vgl. Kapitel 17 im Java-Sprachstandard [31]).

Natürlich ist sich die virtuelle Maschine dabei der nur konzeptuell existierenden maschinenüberspannenden Kontrollfäden nicht bewusst. Führt ein Kontrollfaden, so wie in Abbildung 5.4 gezeigt, während der Abarbeitung der Methode `foo()` aus einem geschützten Bereich heraus einen Fernaufruf `bar()` aus, so bleibt der aufrufende Kontrollfaden t_1 Besitzer des zugehörigen Monitors. Eine Verklemmung tritt ein, wenn im Verlauf der Ausführung der entfernt aufgerufenen Methode `bar()` ein weiterer Fernaufruf `foo2()` zurück zum Ausgangsknoten stattfindet und in diesem Aufruf versucht wird, einen kritischen Abschnitt desselben Objektes zu betreten. Der Kontrollfaden t_1, welcher sich im Besitz des zugehörigen Monitors befindet, wartet auf die Rückkehr des ersten Fernaufrufs `bar()`. Der neue Kontrollfaden t_3, welcher den entfernten Rückaufruf `foo2()` durchführt, kann daher den kritischen Abschnitt nicht betreten und blockiert. Das Problem besteht darin, dass die beiden Kontrollfäden t_1 und t_3, welche unterschiedliche Segmente des konzeptuell maschinenüberspannenden Kontrollfadens repräsentieren, in keiner Beziehung zueinander stehen. Da der Besitzer des Monitors t_1 den kritischen Abschnitt nicht verlassen wird, bevor der erste Fernaufruf zurückkehrt, dies aber nie geschehen wird, weil das aktive Ende t_3 des maschinenüberspannenden Kontrollfadens blockiert ist, ist eine Verklemmungssituation eingetreten.

5.3.2 Transparente Synchronisation

Um Transparenz bezüglich Koordination zu erreichen, muss gewährleistet sein, dass maschinenüberspannende Kontrollfäden als Ganzes gesehen den Besitz von Monitoren auch dann nicht einbüßen, wenn zwischen Betreten des geschützten Bereichs und dem Wiedereintritt ein möglicherweise mehrstufiger Fernaufruf liegt. Da die virtuelle Maschine nicht geändert werden soll, gibt es keine Möglichkeit für einen Kontrollfaden, einen Monitor zu betreten, der noch von einem anderen Kontrollfaden gehalten

wird. Dies ist auch dann nicht möglich, wenn beide lokale Kontrollfäden Segmente eines einzigen konzeptuell maschinenüberspannenden Kontrollfadens repräsentieren, da die virtuelle Maschine nur lokale Kontrollfäden kennt und diese sich in geschützten Bereichen wechselseitig ausschließen.

Der Wiedereintritt in einen geschützten Bereich ist nur für denjenigen lokalen Kontrollfaden möglich, der den zugehörigen Monitor besitzt. Folglich müssen alle Segmente eines maschinenüberspannenden Kontrollfadens, die auf demselben Rechenknoten ausgeführt werden, auf denselben lokalen Kontrollfaden abgebildet werden. Um dies zu erreichen führt KaRMI das Konzept des lokalen Stellvertreters für einen konzeptionell maschinenüberspannenden Kontrollfaden ein. Auf jedem Rechenknoten, auf dem zur Laufzeit ein Segment eines maschinenüberspannenden Kontrollfadens ausgeführt wird, existiert ein lokaler Kontrollfaden als eindeutiger Stellvertreter. Dieser Stellvertreter führt alle Segmente eines maschinenüberspannenden Kontrollfadens auf seinem Knoten aus. Alle Monitore, die von dem maschinenüberspannenden Kontrollfaden auf einem Knoten angefordert werden, gehören damit seinem lokalen Stellvertreter auf diesem Knoten. Dadurch spielt es keine Rolle mehr, ob zwischen der Anforderung des Monitors und einem möglichen Wiedereintritt eine Folge entfernter Methodenaufrufe liegt.

Dass ein maschinenüberspannender Kontrollfaden auf unterschiedlichen Rechenknoten immer noch durch verschiedene lokale Stellvertreter repräsentiert wird, spielt für die Transparenz der Koordination keine Rolle. Da Monitore an Objekte gekoppelt sind und diese an ihren Knoten gebunden[3] sind, kann ein Wiedereintritt in einen geschützten Bereich nur auf demselben Knoten stattfinden, auf dem er initial betreten wurde. Mit der beschriebenen Modifikation bei der Auswahl des ausführenden lokalen Kontrollfadens für ein Segment eines maschinenüberspannenden Kontrollfadens wird somit vollständige Transparenz bezüglich Synchronisation erreicht. Koordinationsprimitive verhalten sich somit für lokale und maschinenüberspannende Kontrollfäden identisch.

5.3.3 Zuordnung von lokalen Kontrollfäden zu Segmenten

Beim Eintreffen eines entfernten Methodenaufrufs bei einem Rechenknoten muss entschieden werden, ob dort bereits ein lokaler Stellvertreter für den zugehörigen maschinenüberspannenden Kontrollfaden existiert. Wenn ja, dann muss dieser lokale Kontrollfaden für die Ausführung der aufgerufenen Methode wiederverwendet werden. Um dazu in der Lage zu ein, müssen zwei Anforderungen erfüllt werden, die im Folgenden besprochen werden.

Identifikator für maschinenüberspannende Kontrollfäden

Um einen Rückaufruf zu erkennen, bei dem ein maschinenüberspannender Kontrollfaden in einem entfernten Methodenaufruf auf einen Knoten zurückkehrt, muss für jeden solchen Kontrollfaden ein Identifikator existieren, der auch während eines Fernaufrufs erhalten bleibt. Dieser Identifikator kennzeichnet den ansonsten nur konzeptuell existierenden maschinenüberspannenden Kontrollfaden eindeutig.

[3] Objektmigration kann nur stattfinden, wenn das Objekt momentan nicht benutzt wird.

In KaRMI wird daher für jeden lokalen Kontrollfaden, der von der Applikation erzeugt wird, zum Zeitpunkt, zu dem der erste Fernaufruf von ihm ausgeht, ein solcher Identifikator erstellt. In diesem Moment wird der lokale Kontrollfaden zum ersten Segment eines maschinenüberspannenden Kontrollfadens und damit Repräsentant dieses Kontrollfadens auf seinem Knoten. Der Identifikator eines maschinenüberspannenden Kontrollfadens wird als unsichtbares Argument jedem entfernten Methodenaufruf mitgegeben.

Der Identifikator eines maschinenüberspannenden Kontrollfadens wird in einer kontrollfadenlokalen Variable gespeichert, die dynamisch erzeugt wird, wenn ein Kontrollfaden den ersten Fernaufruf durchführt.[4] Somit kommt es zu keinem Mehraufwand bei der Ausführung von lokalen Methodenaufrufen. Insbesondere muss dieser Identifikator nicht durch alle Methodenaufrufe eines Programms hindurchgeschleift werden, wie das bei anderen Vorschlägen [106] der Fall ist.

Umlenkung wartender Kontrollfäden

In dem Moment, in dem ein entfernter Methodenaufruf an einem Knoten ankommt, muss entschieden werden, welcher lokale Kontrollfaden die Ausführung der Methode übernehmen soll. Wird der Fernaufruf in einem maschinenüberspannenden Kontrollfaden ausgeführt, der auf der aktuellen Maschine bereits einen lokalen Repräsentanten besitzt, muss die Ausführung an diesen delegiert werden. Andernfalls stellt KaRMI einen lokalen Kontrollfaden zur Verfügung, der zum lokalen Repräsentanten des aufrufenden maschinenüberspannenden Kontrollfadens gemacht wird und die Ausführung übernimmt.

Existiert bereits ein lokaler Repräsentant, so ist sicher, dass dieser im Moment inaktiv ist und auf die Rückkehr eines von der aktuellen Maschine ausgehenden Fernaufrufs wartet. Über den beim Fernaufruf mitübermittelten Identifikator wird dieser Kontrollfaden mit Hilfe einer Hash-Tabelle ausfindig gemacht. Um die Ausführung des ankommenden Fernaufrufs an ihn delegieren zu können, muss anschließend sein Warten unterbrochen werden. Leider kann man normale Netzwerkoperationen – und der Empfang eines Ergebnisses eines entfernten Methodenaufrufs ist eine solche Netzwerkoperation – nicht unterbrechen [58]. Um dennoch einen existierenden lokalen Repräsentanten für die Ausführung eines ankommenden Fernaufrufs wiederverwenden zu können, kommen zwei Auswege in Frage:

- Verwendung der Channel-Schnittstelle: Mit Java 1.4 wurde die Basisbibliothek um sog. Channels im Paket `java.nio` erweitert. Kommunikation über diese Kanäle ist im Gegensatz zu den Kommunikationsprimitiven in `java.io` unterbrechbar. Allerdings schafft das nur schnelle Abhilfe für die Kommunikation über TCP/IP. Für alle anderen Kommunikationstechnologien müssten Implementierungen der Channel-Schnittstelle nachgebaut werden.

- Delegation der Empfangsoperation: Um den wartenden lokalen Repräsentanten unterbrechen zu können, darf dieser nicht selbst in einer Kommunikationsopera-

[4] Eine kontrollfadenlokale Variable ist seit Java 1.2 über die Bibliotheksklasse `ThreadLocal` realisierbar.

tion stecken. Zu diesem Zweck kann das Warten auf das Ergebnis eines Fernaufrufs an einen Helfer delegiert werden. Der lokale Repräsentant wartet dann lediglich auf eine Benachrichtigung seines Helfers. Diese Form des Wartens kann einfach durch eine andere Benachrichtigung über das Eintreffen eines rekursiven Aufrufs unterbrochen werden.

KaRMI verwendet die zweite Möglichkeit, um lokale Repräsentanten zur Ausführung entfernter Methodenaufrufe wiederzuverwenden, da diese unabhängig von der eingesetzten Kommunikationstechnologie verwendbar ist.

5.3.4 Transparente Unterbrechungen

Nicht nur bei der Koordination mittels geschützter Bereiche kann die Anwendung die stückweise Zusammensetzung von maschinenüberspannenden Kontrollfäden durchscheinen sehen. Spezielle Vorsorge ist notwendig, damit Unterbrechungssignale, welche an einen maschinenüberspannenden Kontrollfaden geschickt werden, zu demselben Verhalten führen, wie das bei rein lokalen Kontrollfäden der Fall ist.

Ein Unterbrechungssignal wird in Java an einen Kontrollfaden geschickt, indem die Methode `interrupt()` auf dem zugehörigen Thread-Objekt aufgerufen wird. Diese Unterbrechung wird an den betroffenen Kontrollfaden ausgeliefert, sobald er den nächsten unterbrechbaren Zustand erreicht.[5] Wie bei der Koordination richtet sich auch die Unterbrechung immer an denjenigen lokalen Kontrollfaden, an dessen Thread-Objekt die Unterbrechung ausgelöst wurde. Repräsentiert dieser lokale Kontrollfaden ein Segment eines maschinenüberspannenden Kontrollfadens, das nicht sein aktives Ende ist, trifft die Unterbrechung einen auf die Beendigung eines Fernaufrufs wartenden lokalen Stellvertreter. In Java-RMI führt eine solche Unterbrechung abhängig von der verwendeten Java-Version entweder zum Abbruch der Verbindung oder zur verzögerten bzw. unterbleibenden Auslieferung des Unterbrechungssignals.

In KaRMI wird ein Unterbrechungssignal, das an ein inaktives Segment eines maschinenüberspannenden Kontrollfadens geschickt wird, vom lokalen Stellvertreter abgefangen und weitergeleitet. Da das Unterbrechungssignal nur dort an die Anwendung ausgeliefert werden kann, wo sich momentan das aktive Ende des maschinenüberspannenden Kontrollfadens befindet, muss das Unterbrechungssignal in einer Zusatzverbindung neben dem momentan aktiven Fernaufruf hergeschickt werden (*out-of-band signaling*). Auf dem entfernten Knoten wird die Unterbrechung erneut ausgelöst, was den Weiterleitungsprozess unter Umständen rekursiv fortsetzt, bis das aktive Ende des maschinenüberspannenden Kontrollfadens erreicht ist. Tritt eine Wettlaufsituation dergestalt ein, dass die Weiterleitung des Unterbrechungssignals erst nach Beendigung des Fernaufrufs auf dem entfernten Knoten eintrifft, wird die Unterbrechung auf der weiterleitenden Seite erneut ausgelöst. Der weiterleitende lokale Stellvertreter wird nach Rückkehr des Fernaufrufs zum aktiven Ende des maschinenüberspannenden Kontrollfadens und damit zum richtigen Adressaten für das Unterbrechungssignal.

Durch Transparenz von Koordination und Unterbrechung bei maschinenüberspannenden Kontrollfäden in KaRMI eignet sich diese neue Bibliothek für Fernaufruf, um

[5] Unterbrechungen sind typsichere Ausnahmen, die nur an Stellen auftreten können, an denen das Programm darauf vorbereitet ist.

```
synchronized void foo() {
   //
   // Die gesamte Methode
   // ist als kritischer
   // Abschnitt markiert.
   //
}
```

```
synchronized(x) {
   //
   // Ein einzelner Block
   // als kritischer Ab-
   // schnitt geschützt
   // über den Monitor des
   // Objektes x.
   //
}
```

Abbildung 5.5: Synchronisierte Methode und synchronisierter Block in Java.

als Basis für eine verteilte Programmierumgebung mit Fernzugriff zu dienen.

5.4 Entfernte Sperranforderung

Java sieht vor, Monitore von Objekten nicht nur bei der Ausführung synchronisierter Methoden anzufordern, sondern auch einen einzelnen Block mit dem Monitor eines beliebigen Objektes zu schützen. In Abbildung 5.5 sind beide Formen geschützter Bereiche gezeigt. Für eine einzelne virtuelle Maschine spielt der verwendete Monitor keine große Rolle. In einer verteilten Umgebung erfordert der synchronisierte Block jedoch besondere Aufmerksamkeit.

Auch in einer verteilten Umgebung ist bei der synchronisierten Methode sichergestellt, dass sich das Objekt, dessen Methode ausgeführt wird, und der Monitor, der zum Schutz des kritischen Abschnitts eingesetzt wird, auf derselben Maschine befinden. Dies ist der Fall, da bei einer synchronisierten Methode der zum aktuellen Objekt (`this`) gehörende Monitor implizit angefordert wird. Anders verhält es sich mit einem synchronisierten Block, da dieses Konstrukt den Monitor eines beliebigen Objektes verwendet. Verwendete man zur Synchronisation in einer verteilten Umgebung eine entfernte Referenz, die auf ein Objekt einer anderen virtuellen Maschine zeigt, müssten Monitoranforderung und -freigabe als entfernte Operationen erfolgen, während der Code des geschützten Bereiches lokal ausgeführt würde.

Da eine virtuelle Java-Maschine keine entfernten Referenzen kennt, verbirgt sich hinter einer *konzeptuell* entfernten Referenz nur ein reguläres lokales Stellvertreterobjekt, das mit Mitteln der Bibliothek Methodenaufrufe an sein Original weiterleitet. Benutzt man daher synchronisierte Blöcke zusammen mit entfernten Referenzen aus Java-RMI, behalten diese ihre lokale Semantik. Das heißt, dass der Block nicht mit dem Monitor des entfernten Objektes geschützt wird, sondern mit dem Monitor seines Stellvertreters. Dies führt dazu, dass unbeabsichtigt andere Codeblöcke parallel ausgeführt werden könnten, die mit dem Monitor des Originals oder anderer Stellvertreterobjekte geschützt sind.

Beim synchronisierten Block handelt es sich um ein Sprachkonstrukt, und deswegen ist klar, dass eine reine Bibliothekslösung hier nicht transparent Abhilfe schaffen kann. Zwar ist eine entfernte Sperranforderung als Bibliotheksfunktion notwendig, aber zusätzlich braucht man eine Transformation des Synchronisationskonstrukts für den Fall, dass ein Block an einer entfernten Referenz synchronisiert wird. Java-

```
x.rmAquire();
{
   // kritischer Abschnitt
}
x.rmRelease();
```

Abbildung 5.6: Vermeintliche Umsetzung eines an einer entfernten Referenz `x` synchronisierten Blocks.

RMI bietet keine Möglichkeit der entfernten Sperranforderung. Wie die folgenden Abschnitte zeigen, lässt sich in Java eine entfernte Sperranforderung auch nicht in reguläre entfernte Methodenaufrufe abbilden. KaRMI löst das Problem der entfernten Sperranforderung dadurch, dass zwei lokale Stellvertreter eines maschinenüberspannenden Kontrollfadens während eines entfernt synchronisierten Blocks verschränkt aktiviert werden.

5.4.1 Problematik entfernter Sperranforderung

Eine einleuchtende, aber mit RMI nicht durchführbare Umsetzung eines an einer entfernten Referenz synchronisierten Blocks wäre die Verwendung von zwei entfernten Methodenaufrufen `rmAquire()` und `rmRelease()`, wie in Abbildung 5.6 gezeigt. Dabei ist die Intention, dass in `rmAquire()` die Sperre des Objektes `x` angefordert, danach lokal der kritische Abschnitt ausgeführt und anschließend die Sperre im Aufruf `rmRelease` wieder freigegeben wird. Das Problem bei diesem Vorgehen liegt darin, dass Monitoranforderung und Monitorfreigabe in zwei getrennten und voneinander unabhängigen entfernten Methodenaufrufen durchgeführt werden müssten. Um effektiv zu sein, müsste der Monitor des entfernten Objektes `x` in der Methode `rmAquire()` angefordert und später in `rmRelease()` wieder freigegeben werden.

Solche Methoden `rmAquire()` und `rmRelease()`, die eine einzelne Monitoranforderung oder -freigabe enthalten, sind in Java nicht möglich. Eine Sperranforderung kann immer nur als Paar zusammen mit der Freigabe am Anfang und Ende eines syntaktischen Blocks auftreten. Obwohl die Monitoranforderung und -freigabe separate Befehle der virtuellen Maschine sind, wird die Eigenschaft der Paarung dieser Befehle auch im Bytecode verlangt. Beim Laden des Programms versucht die virtuelle Maschine im sog. Verifier durch statische Analyse zu beweisen, dass für jeden Befehl zur Monitoranforderung innerhalb einer Methode genau ein passender Befehl zur Monitorfreigabe in derselben Methode ausgeführt werden muss. Lässt sich diese Eigenschaft nicht statisch nachweisen, wird die Ausführung verweigert. Offensichtlich ist diese Forderung weder für die Methode `rmAquire()` noch für `rmRelease()` erfüllt. Demzufolge sind die für die Umsetzung der entfernten Sperranforderung über reguläre entfernte Methodenaufrufe notwendigen Methoden in Java nicht einzeln implementierbar. Mit Java-RMI lassen sich nur einzelne Methoden entfernt aufrufen, und damit ist keine adäquate Umsetzung des synchronisierten Blocks in der verteilten Umgebung möglich.

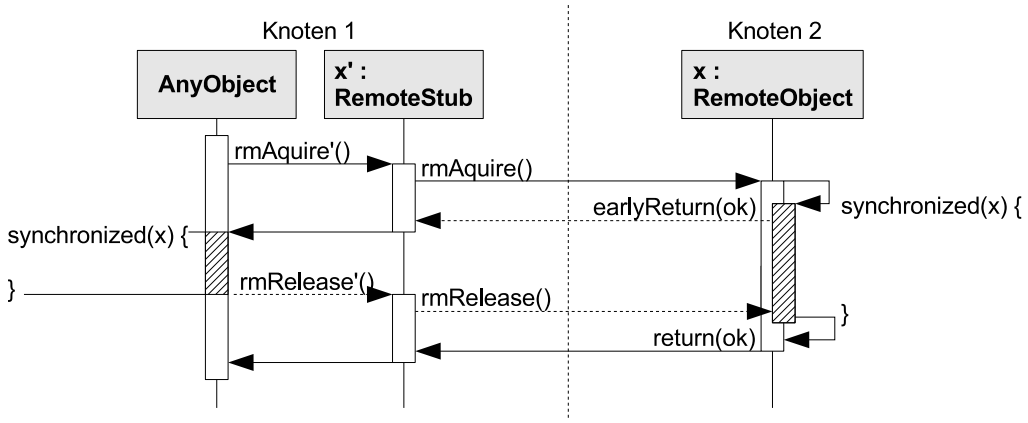


Abbildung 5.7: Sequenzdiagramm der entfernten Sperranforderung.

5.4.2 Sperranforderung über lokale Stellvertreter

Die Lösung für die entfernte Anforderung von Sperren in KaRMI stellen die lokalen Stellvertreter für maschinenüberspannende Kontrollfäden dar. Wie in Abschnitt 5.3.3 beschrieben, wird ein maschinenüberspannender Kontrollfaden auf jedem Knoten, den er überspannt, durch einen lokalen Kontrollfaden vertreten. Dieser lokale Stellvertreter bearbeitet alle Methodenaufrufe des maschinenüberspannenden Kontrollfadens, die auf seinem lokalen Knoten ausgeführt werden.

Die Erweiterung, die es ermöglicht, entfernte Sperranforderungen zu realisieren, besteht darin, dass jeder lokale Stellvertreter zusätzlich Sperren anfordern und wieder freigeben kann. Beides geschieht im Auftrag des maschinenüberspannenden Kontrollfadens, der als Gesamtheit zu sehen ist. Dabei verlässt der lokale Stellvertreter den synchronisierten Block, welchen er beim Ausführen der Sperranforderung betritt, solange nicht, bis die Aufforderung zur Freigabe ebenfalls entfernt erteilt wird. Auf dem entfernten Knoten, auf dem sich das zu sperrende Objekt befindet, läuft damit eine ganz normale Monitoranforderung und -freigabe in einem syntaktischen Block ab, die aber durch das aktive Ende des maschinenüberspannenden Kontrollfadens entfernt gesteuert wird. Realisiert wird dies in KaRMI dadurch, dass `rmAquire()` als spezieller Fernaufruf implementiert wird, der über die Möglichkeit einer frühzeitigen Rückkehr verfügt (*early return*).

Abbildung 5.7 zeigt die entfernte Sperranforderung im Sequenzdiagramm. Auf der linken Seite sieht man den geschützten Bereich der Anwendung auf Knoten 1, der an dem entfernten Objekt `x` auf Knoten 2 synchronisiert werden soll. Vor dem Betreten dieses geschützten Bereiches wird die Methode `rmAquire'()` an dem Stellvertreter `x'` von `x` aufgerufen. Aus dieser Methode wird der spezielle Fernaufruf `rmAquire()` angestoßen. Dieser betritt lokal auf Knoten 2 einen an `x` synchronisierten Abschnitt und sendet danach eine verfrühte Rückkehr zurück. Diese bestätigt, dass die entfernte Monitoranforderung erfolgreich war. Dabei verbleibt aber der lokale Stellvertreter des maschinenüberspannenden Kontrollfadens selbst in dem synchronisierten Bereich auf Knoten 2 und hält damit die Sperre fest. Wenn `rmAquire'()` die verfrühte Rückkehr empfängt, kehrt diese *lokale* Methode zurück und der entfernt geschützte Bereich wird auf Knoten 1 betreten.

Nach Verlassen des entfernt synchronisierten Bereiches wird die Methode `rmRelease'()` an der entfernten Referenz `x'` aufgerufen. Diese wiederum sendet die Nachricht `rmRelease()` an das Objekt `x` und wartet anschließend, bis der durch `rmAquire()` angestoßene Fernaufruf endgültig zurückkehrt. Sobald der lokale Stellvertreter des maschinenüberspannenden Kontrollfadens auf Knoten 2 die Nachricht `rmRelease()` empfängt, verlässt er den an `x` synchronisierten Bereich und gibt damit die entfernt angeforderte Sperre frei. Anschließend kehrt der konzeptuell bei `rmAquire()` begonnene Fernaufruf zurück. Was auf dem anfordernden Knoten 1 aussieht wie zwei getrennte Aufrufe `rmAquire'()` und `rmRelease'()`, wird intern in einen einzigen Fernaufruf umgesetzt. Der Fernaufruf bleibt zwischen diesen beiden Methodenaufrufen aktiv und hält die Sperre am entfernten Objekt, bis mit `rmRelease'()` die Aufforderung zur Freigabe erteilt wird.

Während der lokale Stellvertreter die Sperre auf Knoten 2 durchsetzt, bleibt er gleichzeitig ansprechbar für die Ausführung entfernter Methodenaufrufe. Sollte die Anwendung aus dem entfernt geschützten Bereich einen Fernaufruf auf Knoten 2 durchführen, würde dieser Fernaufruf ebenfalls durch den lokalen Stellvertreter dort bearbeitet. Dazu wird der synchronisierte Block nicht verlassen, sondern der ankommende Fernaufruf in einem lokalen Methodenaufruf aus dem geschützten Bereich heraus bearbeitet. Die entfernte Sperranforderung und der entfernte Aufruf werden somit wie zwei Segmente eines maschinenüberspannenden Kontrollfadens behandelt, die auf demselben Knoten ausgeführt und damit rekursiv ineinandergeschachtelt werden.

Damit ermöglicht KaRMI über die beiden Spezialmethoden `rmAquire'()` und `rmRelease'()` die entfernte Anforderung und Freigabe von Sperren, womit sich geschützte Bereiche realisieren lassen, die als Sperre ein entferntes Objekt verwenden. Wie in Abbildung 5.7 gut zu sehen ist, führen diese Methoden keine zwei separaten Fernaufrufe durch, sondern sie steuern einen einzigen Fernaufruf, der über die Möglichkeit verfrühter Rückkehr verfügt. Damit wird es möglich, dass die Monitoranforderung und -freigabe auf der entfernten Seite – wie von der virtuellen Maschine vorgeschrieben – in einem syntaktischen Block erfolgen kann. Nur auf Klientenseite erfolgt die Anforderung und Freigabe über zwei getrennte Methodenaufrufe.

5.4.3 Kompatibilität mit regulärer Synchronisation

Die Kombination aus entfernter Sperranforderung und segmentweiser Abbildung maschinenüberspannender Kontrollfäden auf lokale Stellvertreter ermöglicht volle Kompatibilität von (entfernt) synchronisierten Blöcken und synchronisierten Methoden. Ein maschinenüberspannender Kontrollfaden kann dabei einen Monitor wie im lokalen Fall rekursiv mehrfach anfordern und die Synchronisationsformen dabei auch mischen.

In Abbildung 5.7 könnte aus dem entfernt geschützten Bereich auf Knoten 1 eine synchronisierte Methode des Objektes `x` aufgerufen werden. Da der maschinenüberspannende Kontrollfaden durch die entfernte Sperranforderung den dazu notwendigen Monitor bereits besitzt, darf der entfernte Aufruf einer synchronisierten Methode an `x` nicht blockieren. Dass dies auch tatsächlich nicht passieren wird, kann man folgendermaßen zeigen: Während der Abarbeitung des entfernt synchronisierten Blocks auf Knoten 1 existiert ein lokaler Stellvertreter auf Knoten 2, der den Monitor von `x` hält. Geschieht ein Fernaufruf aus diesem Kontext auf Knoten 2, wird er von genau

diesem Stellvertreter abgearbeitet. Dieser Stellvertreter des maschinenüberspannenden Kontrollfadens steigt dazu rekursiv aus dem geschützten Bereich ab und ruft die angeforderte Methode auf. Da er dabei den Monitor von `x` nicht aufgibt, ist sichergestellt, dass er die synchronisierte Methode von `x` sofort betreten kann, so als hätte lokal ein Wiedereintritt in einen bereits angeforderten Monitor stattgefunden.

5.5 Transparent entfernte Klassen

Mit den in den vergangenen Abschnitten beschriebenen Mitteln lässt sich ein verteilter Objektraum realisieren, in welchem nebenläufige Kontrollfäden transparent abgearbeitet werden können. Der aktuelle Abschnitt befasst sich mit der Programmierung dieses verteilten Objektraums. Das Ziel ist es, mit dem verteilten Objektraum eine verteilte virtuelle Maschine zu simulieren. Dazu soll ein weitgehend unverändertes Java-Programm für eine reguläre virtuelle Maschine durch Transformation verteilt ausführbar gemacht werden. Ziel der Transformation ist es, eine Klassendefinition so umzuschreiben, dass die Instanzen der Klasse auf den Knoten der verteilten virtuellen Maschine angelegt werden können und transparent miteinander kommunizieren können, wie das auch in einer regulären virtuellen Maschine der Fall ist. Zu diesem Zweck versieht der Programmierer die Klassendefinition mit einer Markierung, die aus einer regulären Java-Klasse eine sog. entfernte Klasse macht. Nur solche entfernte Klassen werden von der Transformation entfernt ansprechbar gemacht. Diese selektive Transformation geschieht einerseits aus Effizienzgründen, andererseits wäre die Transformation gar nicht für alle Klassen durchführbar.

Die in der vorliegenden Arbeit verwendete Transformation für entfernte Klassen basiert auf den in [85] beschriebenen Ideen. Das dort vorgestellte System übersetzt markierte Klassen zurück in reines Java unter Verwendung von Java-RMI. Die vorliegende Arbeit ersetzt Java-RMI durch die KaRMI Bibliothek, und das ermöglicht eine schnelle Kommunikation im Rechnerbündel, die Erzeugung von effizienteren Stellvertretern für entfernte Objekte und eine adäquate Transformation von Blöcken, die an entfernten Objekten synchronisiert sind. Darüberhinaus wurde die Transformation um die effiziente Behandlung von statische Konstanten in entfernten Klassen und um die Möglichkeit von Verteilungsannotationen ergänzt.

Der unmittelbar folgende Abschnitt 5.5.1 umreißt die Transformationsidee für entfernte Klassen, soweit diese für das Verständnis der Erweiterungen in den darauffolgenden Abschnitten wichtig ist. Der folgende Abschnitt kann bei Kenntnis von [85] übersprungen werden.

5.5.1 Transformation entfernter Klassen

Abbildung 5.8 zeigt die Deklaration einer entfernten Klasse `R`. Bis auf die Markierung `remote` an der Klassendeklaration unterscheidet sie sich nicht von einer regulären Java-Klasse. Die Markierung stößt während der Übersetzung die Transformation für entfernte Klassen an. Ziel dieser Transformation ist es, die Klassensemantik auf die verteilte Umgebung abzubilden. Objekte dieser Klasse sollen auf beliebigen Knoten der verteilten Umgebung angelegt werden können und von überall ansprechbar sein.

```
public remote class R {
   int x;
   static long y;
   public R(int x) {this.x = x; }
   public void foo() {...}
   public static void bar() {...}
   static {
      y = System.currentTimeMillis();
   }
}
```

Abbildung 5.8: Deklaration einer entfernten Klasse.

Der Zugriff (Methodenaufruf und Feldzugriff) soll unabhängig vom Aufenthaltsort des Zielobjektes sein und soll sich im Programmtext nicht vom Zugriff auf ein reguläres Java-Objekt unterscheiden. Der Aufenthaltsort eines entfernten Objektes wird bei seiner Erzeugung festgelegt. Ein entferntes Objekt kann dabei auf einem beliebigen Rechenknoten erzeugt werden (unabhängig vom Rechenknoten, auf dem die Erzeugung angestoßen wurde). Ein transparenter Umzug von einem Rechenknoten auf einen anderen ist dann möglich, wenn die entfernte Klasse nicht explizit durch Deklaration der Markierungsschnittstelle `Resident` als stationär gekennzeichnet wurde.[6] Ferner soll die Klasse `R` selbst in der verteilten Umgebung repräsentiert sein. Auch auf sie soll wie gewohnt zugegriffen werden können. Die Zugriffe auf die entfernte Klasse selbst werden ebenfalls in Fernzugriffe auf denjenigen Knoten abgebildet, wo die Klasse angelegt wurde.

Die wesentlichen Punkte der Transformation sind die Trennung von Instanzen- und Klassenanteil, die Umschreibung von Feldzugriffen und Anweisungen zur Objekterzeugung und die Einhaltung der korrekten Initialisierungsreihenfolge trotz Verteilung. Auf diese Teile der Transformation gehen die folgenden Abschnitte ein.

Trennung von Instanzen- und Klassenanteil

Da die Knoten der verteilten Umgebung aus unveränderten virtuellen Maschinen bestehen, können dort nur vollständige Klassen geladen werden und für jede dieser Klassen gilt die Java-Klassensemantik. In der verteilten Umgebung muss es aber möglich sein, auf einem Knoten eine Instanz einer entfernten Klasse zu erzeugen, ohne auch die zugehörige Klasse auf demselben Knoten zu laden. Die entfernte Klasse ist möglicherweise bereits auf einem anderen Knoten geladen und darf demzufolge nicht ein weiteres Mal initialisiert werden. Da keine Instanz einer Java-Klasse erzeugt werden kann, bevor nicht ihre Klasse geladen und initialisiert ist, muss eine entfernte Klasse in mehrere Klassen zerlegt werden, damit ihre Instanzen auf allen Knoten angelegt werden können, während die Klasse selbst nur auf einem einzigen Knoten geladen wird und dort den statischen Anteil repräsentiert.

Für die Struktur des generierten Codes ist weiterhin wichtig, dass Fernzugriff nur auf Instanzen möglich ist, nicht aber auf den statischen Anteil einer Klasse. Um Zugriff

[6] Durch Kennzeichnung einer entfernten Klasse als stationäre Klasse kann die Effizienz des transparenten Zugriffs deutlich erhöht werden (vgl. Abschnitt 7.2.6).

```
public class R_instance {
   // Instance variables
   int x;

   // Zugriffsmethoden für x
   public int getX() {
      return x;
   }

   public void setX(int x) {
      this.x = x;
   }

   // Instance constructor
   public R_instance(int x) {
      this.x = x;
   }

   // Instance methods
   public void foo() {...}
}
```

```
public class R_class {
   // Class variables
   long y;

   // Zugriffsmethoden für y
   public int getY() {
      return y;
   }

   public void setY(int y) {
      this.y = y;
   }

   // Class methods
   void bar() {...}

   // Class initializer
   void _clinit() {
      y = System
         .currentTimeMillis();
   }

   // Instance factory
   RemoteReference newR(
      int x
   ) {
      return getReference(
         new R_instance(x));
   }
}
```

Abbildung 5.9: Klasse mit Instanzenanteil von `R` (links) und Klasse mit statischem Anteil (rechts).

```
public class R {
   // Entfernte Referenz auf das Instanzobjekt
   RemoteReference thisRef;

   // Entfernte Referenz auf das Klassenobjekt
   static RemoteReference classRef =
      findClassObject("R");

   // Zugriffsmethoden für x und y
   int getX() {...}
   void setX(int x) {...}
   static long getY() {...}
   static void setY(long y) {...}

   // Konstruktor der Stellvertreterklasse
   public R(int x) {...}

   // Stellvertretermethoden für Instanzmethoden
   public void foo() {
      // Entfernter Aufruf von foo() an thisRef;
   }

   // Stellvertretermethoden für statische Methoden
   public static void bar() {
      // Entfernter Aufruf von bar() an classRef;
   }
}
```

Abbildung 5.10: Stellvertreterklasse für `R` mit Zugriffsmethoden.

auf den statischen Anteil einer entfernten Klasse zu erlauben, muss dieser nicht nur in eine eigene Klasse verlagert, sondern auch zu nicht-statischem Code umgewandelt werden. Die entfernte Klasse wird somit nach der Transformation durch ein entfernt ansprechbares Singleton-Objekt repräsentiert.

Die entfernte Klasse `R` aus Abbildung 5.8 wird während der Transformation im wesentlichen in drei Klassen zerlegt.[7] Zwei davon sind `R_instance` und `R_class`. Sie werden auf Fernzugriff vorbereitet und kapseln den Instanzen- bzw. den Klassenanteil der Ausgangsklasse. Abbildung 5.9 zeigt (vereinfacht) diese beiden Teile des Transformationsergebnisses. Die Merkmale der Ausgangsklasse werden auf die beiden Klassen `R_instance` und `R_class` aufgeteilt. Alle nicht-statischen Merkmale werden in die Klasse `R_instance` verschoben. Alle statischen Merkmale der Ausgangsklasse gelangen als nicht-statische Merkmale in die Klasse `R_class`. Nach der Transformation repräsentiert folglich ein Objekt der Klasse `R_instance` eine Instanz der entfernten Ausgangsklasse, während ein Objekt von `R_class` die entfernte Klasse selbst repräsentiert.

Die in Abbildung 5.10 dargestellte dritte Klasse trägt denselben Namen wie die Ausgangsklasse. Sie hat zwei Aufgaben. Diese sog. Stellvertreterklasse fungiert zum

[7] Für den Fernzugriff werden aus technischen Gründen noch weitere Klassen erzeugt.

einen als Bindeglied zwischen dem Instanzen- und Klassenanteil eines entfernten Objektes. Zum anderen repräsentieren ihre Instanzen lokale Stellvertreterobjekte für Instanzen der entfernten Klasse. Die Stellvertreterklasse wiederholt alle Methoden der Ausgangsklasse. Die Methoden in der Stellvertreterklasse führen selbst aber keinen Anwendungscode aus, sondern leiten Aufrufe in einem Fernaufruf an die entsprechende Methode des zugehörigen Instanzen- oder Klassenobjektes weiter. Ein Aufruf von `foo()` an einer Instanz der Stellvertreterklasse wird diesen Aufruf an die entsprechende Methode des vertretenen Objektes der Klasse `R_instance` weiterleiten. Ein Aufruf der statischen Methode `bar()` an der Stellvertreterklasse wird den Aufruf an die Methode `bar()` des Klassenobjektes der entfernten Klasse weiterleiten. Ein Stellvertreterobjekt stellt damit eine Fassade für den Zugriff auf ein entferntes Objekt und dessen Klasse dar. Die Zugriffe auf beide Teile finden potenziell entfernt statt, werden aber an zwei unterschiedliche Objekte delegiert, je nachdem ob der Instanzen- oder der Klassenanteil betroffen ist. Damit kann die generierte Stellvertreterklasse die Ausgangsklasse in allen Kontexten, die Methodenaufrufe durchführen, transparent ersetzen. Um eine entfernte Klasse zu verwenden, muss der Code nur an wenigen Stellen geändert werden. Die folgenden Abschnitten beleuchten andere Kontexte der Verwendung und beschreiben eventuell notwendige zusätzliche Transformationen.

Variablenzugriffe

Zugriffe auf Variablen entfernter Klassen müssen gesondert behandelt werden, da sie primitive Operationen der virtuellen Maschine sind und nicht überschrieben werden können. In der Stellvertreterklasse aus Abbildung 5.10 tauchen weder Instanzvariablen noch Klassenvariablen auf. Stattdessen sind diese durch Zugriffsmethoden zum Auslesen und Schreiben eines Wertes ersetzt. Diese Umschreibung ist unerlässlich, da es keine Möglichkeit gibt, einen Variablenzugriff der Form `r.x` im Stellvertreterobjekt an das entfernte Objekt vom Typ `R_instance` weiterzuleiten.

Da Variablenzugriffe von einem Objekt in Java nicht überschrieben werden können, kann sich die Transformation entfernter Klassen nicht auf die markierten Klassen beschränken. Stattdessen müssen die Methodenrümpfe aller Klassen in die Transformation einbezogen werden, die eine entfernte Klasse direkt benutzen. Die Transformation der Methodenrümpfe schreibt Variablenzugriffe in die generierten Zugriffsmethoden um. Ein Zugriff `r.x` auf ein entferntes Objekt der Ausgangsklasse `R` (Abbildung 5.8) wird dabei in einen Zugriff `x.getX()` an einem Objekt der korrespondierenden Stellvertreterklasse (Abbildung 5.10) umgeschrieben. Die dadurch aufgerufene Methode `getX()` der Stellvertreterklasse leitet diesen Zugriff an die ebenfalls generierte Methode `getX()` der Instanzenklasse (Abbildung 5.9 links) weiter.

Für eine feldwertige Instanzvariable werden mehrere Zugriffsmethoden generiert, so dass entfernte Zugriffe auch auf einzelne Elemente möglich sind. So würden für eine Instanzvariable `int[] z` der entfernten Klasse `R` neben der Methode zum Lesen und Schreiben des gesamten Feldes auch Methoden erzeugt, die eine einzelne Feldposition lesen (`getZ(int index)`) und schreiben (`setZ(int index, int value)`).

Instanzierung

Die Erzeugung eines neuen entfernten Objektes wird von der Stellvertreterklasse angestoßen. Abbildung 5.10 zeigt den Konstruktor der Stellvertreterklasse, der dieselbe Signatur wie der Konstruktor der Ausgangsklasse besitzt. Zum Zeitpunkt, zu dem ein neues Stellvertreterobjekt erzeugt wird, existiert das entfernte Objekt, das es vertritt, noch nicht. Stattdessen stößt die Erzeugung eines neuen Stellvertreterobjektes die Erzeugung des neuen entfernten Objektes an. Somit kann zwar die Stellvertreterklasse bei der transformierenden Übersetzung die Ausgangsklasse in allen Programmteilen transparent ersetzen, in denen eine Objekterzeugung stattfindet. Der für den Konstruktor der Stellvertreterklasse generierte Code unterscheidet sich aber von dem für reguläre Stellvertretermethoden, weil der Konstruktor der Stellvertreterklasse zwei Aufgaben hat. Erstens muss er den Knoten auswählen, auf dem das entfernte Objekt erzeugt werden soll und zweitens die Erzeugung durch einen Fernaufruf auf den entsprechenden Knoten anstoßen. Die Ortsauswahl wird an ein Distributorobjekt delegiert, welches von der Anwendung konfiguriert werden kann. Dadurch geht allerdings der Kontext der Objekterzeugung verloren und die Ortsauswahl muss allein aufgrund des Klassennamens des zu erzeugenden Objektes getroffen werden. Mit dieser Information allein ist es in vielen Fällen nicht möglich, eine überlegte Entscheidung zu treffen. Die Objekterzeugung wird daher in Abschnitt 5.5.4 um eine optionale Annotation ergänzt, die gezieltere Objektplatzierungen erlaubt. Die eigentliche Objekterzeugung wird von der Fabrikmethode im Klassenobjekt durchgeführt, die passend zu jedem Konstruktor der Originalklasse generiert wird. Die rechte Seite von Abbildung 5.9 zeigt eine solche Fabrikmethode `newR()` für den einzigen Konstruktor der Originalklasse. Diese Fabrikmethode ist notwendig, weil nur *Instanzmethoden* entfernt aufgerufen werden können und die Fabrikmethode entfernt auf demjenigen Knoten aufgerufen werden muss, auf dem das entfernte Objekt erzeugt werden soll.[8] Die Fabrikmethode erzeugt ein Objekt der Instanzklasse durch Aufruf des passenden Konstruktors. Die Konstruktorargumente werden vom Konstruktor der Stellvertreterklasse an die Fabrikmethode und von dort an den Konstruktor der Instanzklasse weitergeleitet. Die Fabrikmethode liefert schließlich eine entfernte Referenz auf das neu erzeugte Objekt zurück.

Die folgenden Abschnitte beschäftigen sich mit Spezialthemen der Transformation für entfernte Klassen. Diese sind zum Teil spezifisch für die Kommunikationsbibliothek KaRMI und zum Teil allgemein anwendbare Optimierungen.

5.5.2 Synchronisation

Wie in Abschnitt 5.3 beschrieben, ermöglicht KaRMI transparent maschinenüberspannende Kontrollfäden, ohne dass dafür zusätzlich die Transformation geändert werden müsste. Anders verhält es sich bei synchronisierten Blöcken. Für die Transformation synchronisierter Blöcke muss der in Abschnitt 5.4 beschriebene Mechanismus in KaRMI herangezogen werden, mit dem eine entfernte Sperranforderung möglich ist.

[8] Um entfernte Objekte einer Klasse auch auf anderen Knoten als dem Heimatknoten des Klassenobjektes anlegen zu können, werden von der Klasse `R_class` weitere uninitialisierte Instanzen vorgehalten, die ausschließlich als Fabrikobjekte für die entfernte Objekterzeugung dienen.

```
RemoteObject x;
...
synchronized(x) {
   // Code
}
```

```
RemoteObject x;
...
if (isLocal(x)) {
   synchronized(getImpl(x)) {
      // Code
   }
} else {
   rmAquire(x);
   try {
      // Code
   } finally {
      rmRelease(x);
   }
}
```

Abbildung 5.11: Transformation des synchronisierten Blocks.

Allerdings beschränkt sich die Transformation nicht auf die schon in Abbildung 5.6 gezeigte, da die entfernte Sperranforderung dann nicht verwendet werden kann, wenn die entfernte Referenz auf ein Objekt auf demselben Knoten verweist, auf dem der synchronisierte Block ausgeführt werden soll.

Wie in Abschnitt 5.4.2 beschrieben wird bei der entfernten Sperranforderung der Monitor des entfernten Objektes durch einen anderen Kontrollfaden – den eindeutigen Stellvertreter des anfordernden Kontrollfadens auf dem Heimatknoten des entfernten Objektes – während der Ausführung des geschützten Blocks gesperrt gehalten. Ist der Knoten, auf dem die Sperranforderung angestoßen wird, identisch mit dem Knoten, auf dem sich das entfernte Objekt befindet, wäre offensichtlich der anfordernde Kontrollfaden mit dem eindeutigen Stellvertreter auf dem Knoten des entfernten Objektes identisch und eine Delegation der Sperranforderung daher nicht möglich.

Abbildung 5.11 zeigt die Transformation eines an einer entfernten Referenz synchronisierten Blocks. Auf der linken Seite ist ein Ausschnitt aus dem Originalprogrammtext zu sehen, auf der rechten Seite das Transformationsergebnis. In der Abbildung wird vorausgesetzt, dass zur Übersetzungszeit schon bekannt ist, dass es sich bei der Referenz um eine entfernte Klasse handelt. Wäre dies nicht der Fall (wenn es sich z. B. beim Typ von `x` um `Object` oder eine Schnittstelle handeln würde), würde eine zusätzliche Abfrage mit einem Laufzeittyptest eingesetzt. Der transformierte Code prüft zuerst, ob es sich bei der entfernten Referenz `x` um ein lokal liegendes entferntes Objekt handelt. Ist dies der Fall, wird ein regulärer synchronisierter Block eingesetzt. Dabei wird statt der entfernten Referenz eine direkte Referenz auf das Implementierungsobjekt als Sperrobjekt verwendet. Nur wenn die entfernte Referenz tatsächlich auf ein Objekt auf einem entfernten Knoten verweist, kann der oben beschriebene Mechanismus der entfernten Sperranforderung eingesetzt werden.

Wie aus der rechten Seite von Abbildung 5.11 zu erkennen ist, bewirkt die Transformation des synchronisierten Blocks eine Codeverdopplung des geschützten Bereichs. Dies rührt daher, dass im lokalen Fall Monitoranforderung und -freigabe gepaart an einem syntaktischen Block erfolgen müssen. Diese Codeverdopplung ist nicht dramatisch, da synchronisierte Blöcke in der Praxis relativ selten vorkommen und dann

```
public remote class R {
   static final int ID = 42;
   static final long TIME;
   static final String NAME;
   static final int[] ARRAY;
   static final String[] MESSAGE;
   static final HashMap MAP;

   static {
      TIME = System.currentTimeMillis();
      NAME = InetAddress.getLocalHost().getHostName();
      ARRAY = new int[] {13, 42};
      MESSAGE = new String[] {"hello", "world", "!"};
      MAP = new HashMap();
   }
}
```

Abbildung 5.12: Entfernte Klasse mit Konstanten.

auch nur Teile von Methoden umfassen. Die Codeverdopplung könnte vermieden werden, indem man den geschützten Code in eine eigene Methode herausfaktorisiert und diese Methode aus beiden Zweigen der Transformation aufruft.

Mit der Kombination aus den in KaRMI realisierten maschinenüberspannenden Kontrollfäden und der oben beschriebenen Transformation synchronisierter Blöcke unter Verwendung von entfernter Sperranforderung wird die Synchronisation im verteilten Adressraum transparent. Ihre Semantik unterscheidet sich nicht von der in einer einzelnen virtuellen Maschine. Dadurch wird ein wesentlicher Beitrag zum Übergang von einer nichtverteilten in eine verteilte Umgebung geleistet.

5.5.3 Statische Konstanten

Die Transformation, wie sie in Abschnitt 5.5.1 beschrieben ist, verschiebt alle statischen Anteile einer entfernten Klasse `R` in die Klasse des Klassenobjektes `R_class`. Zur Laufzeit wird dieses Klassenobjekt auf einem Knoten der verteilten Umgebung initialisiert und alle Zugriffe auf solche statischen Bestandteile der Ausgangsklasse in Fernzugriffe auf das Klassenobjekt umgelenkt. Konstanten werden in Java mit der Markierung `static final` versehen. Sie müssen beim Laden der Klasse initialisiert werden und können sich danach nicht mehr ändern. Konstanten sind Teil der Klasse und würden daher bei einer naiven Transformation in das Klassenobjekt verschoben und bei jedem Zugriff über Fernaufruf gelesen. Für Konstanten ist ein Fernaufruf bei jedem Zugriff überflüssig, da sich ihr Wert nicht ändert und daher ohne Konsistenzprobleme lokal auf jedem Knoten zwischengespeichert werden kann.

Der nächste Abschnitt leitet eine Regel her, wann für eine Konstante oben beschriebene Optimierung durchgeführt werden kann, ohne die Programmsemantik zu verändern. Der darauf folgende Abschnitt beleuchtet die Technik der Transformation.

```
public remote class R {
   static final const int[] ARRAY;
   static final const String[] MESSAGE;

   ...
}
```

Abbildung 5.13: Entfernte Klasse mit arraywertigen Konstanten.

Zulässigkeit der Optimierung

Abbildung 5.12 zeigt eine entfernte Klasse `R` mit sechs statisch finalen Feldern. Von diesen Feldern sind streng genommen nur die ersten zwei „echte“ Konstanten. Und von diesen ist nur `ID` eine Übersetzungszeitkonstante im Sinne der Java-Sprachspezifikation (vgl. Abschnitt 15.28 in [31]). Nur solche Übersetzungszeitkonstanten dürfen in `case`-Anweisungen als Muster angegeben werden. `TIME` dagegen wird zum Zeitpunkt bestimmt, zu dem die Klasse geladen und initialisiert wird. Alle weiteren „Konstanten“ sind Referenzen auf Objekte. Bei solchen konstanten Referenzen ist nur die Referenz selbst konstant, der Zustand des referenzierten Objektes kann sich aber möglicherweise zur Laufzeit des Programms ändern.

Während das Zwischenspeichern echter Konstanten zum Vermeiden von Fernaufrufen unproblematisch ist, kann ohne besondere Maßnahmen eine solche Transformation bei konstanten Referenzen zu einer falschen Programmsemantik führen. Wird das mit einer konstanten Referenz referenzierte Objekt nach der Initialisierung der Klasse von der Anwendung noch verändert, so kann es in der verteilten Umgebung zu einer inkonsistenten Sichtweise auf die Klasse kommen, wenn solche Objekte auf den einzelnen Knoten lokal zwischengespeichert werden.

Die Konstante `NAME` verweist zur Laufzeit auf ein Objekt des Typs `String`. Die Klasse `String` ist selbst eine unveränderliche Klasse. Ihre Objekte können nach ihrer Erzeugung nicht mehr modifiziert werden. Daher macht die Kombination der Eigenschaften – konstante Referenz und unveränderliches Objekt – `NAME` doch zu einem Kandidaten, für den die Fernzugriffe vermieden werden können. Da Fernzugriff auf eine Variable mit Kopiersemantik verbunden ist, reicht, wie schon in Abschnitt 4.3.1 beschrieben, die Unveränderlichkeit der Klasse alleine nicht aus, um gefahrlos Kopien mit Referenzen vertauschen zu können. Wird die Optimierung durchgeführt, findet eine solche Vertauschung statt, weil anstelle neuer Kopien bei jedem Fernzugriff immer dieselbe lokale Referenz auf dasselbe zwischengespeicherte Objekt geliefert wird. Daher müssen zusätzlich zur Unveränderlichkeit die Bedingungen aus Abschnitt 4.3.1 erfüllt sein, damit ein über eine konstante Referenz referenziertes Objekt lokal zwischengespeichert werden darf.

Die beiden Konstanten `ARRAY` und `MESSAGE` verweisen beide jeweils auf ein Array-Objekt. Da Arrays in Java immer veränderbare Objekte darstellen, ist die Optimierung hier im allgemeinen unzulässig. Wenn die Anwendung die Variablen `ARRAY` und `MESSAGE` aber tatsächlich wie Konstanten verwendet, d.h. sie nach ihrer Initialisierung nicht mehr modifiziert, dann wäre es aus Geschwindigkeitsgründen sehr ärgerlich, wenn trotzdem für jeden Zugriff ein Fernaufruf durchgeführt werden müsste. Leider gibt es in Java keine Möglichkeit, arraywertige Konstanten zu deklarieren. Um

die Möglichkeit zu bieten, effiziente arraywertige Konstanten in entfernten Klassen zu deklarieren, wird diese Möglichkeit hinzugefügt. Dazu wird die in Java als reserviertes Wort vorgesehene, aber in der Sprache nicht benutzte Markierung `const` verwendet. Abbildung 5.13 zeigt die Anwendung auf die Konstanten `ARRAY` und `MESSAGE`. Als `static final` deklarierte arraywertige Variablen dürfen *zusätzlich* mit `const` markiert werden. Dies bedeutet, dass nicht nur die Referenz auf das Array konstant sein soll, sondern auch sein Inhalt. Die Konstanz der Feldelemente wird vom Übersetzer geprüft. Außerhalb des statischen Initialisierers einer Klasse darf ein als `const` deklariertes Array nicht modifiziert werden.[9]

Offensichtlich darf die letzte Variable `MAP` aus Abbildung 5.12 nicht lokal zwischengespeichert werden, da sie ein nicht konstantes Objekt referenziert, das sich nach Initialisierung der Klasse noch verändern kann.

Zusammenfassend ist die Optimierung, statische Konstanten lokal zwischenzuspeichern, dann zulässig, wenn eine der folgenden Bedingungen zutrifft:

- Es handelt sich um eine Übersetzungszeitkonstante gemäß Java-Sprachstandard.
- Der Typ der Konstanten ist ein Basistyp (kann als Spezialfall des nächsten Punktes betrachtet werden).
- Die Konstante referenziert ein Wertobjekt gemäß Abschnitt 4.3.1.
- Die Konstante referenziert ein Array mit einem primitiven Typ oder einer Wertklasse als Basistyp und trägt zusätzlich die Markierung `const`.

Transformation statischer Konstanten

Wie bei der Initialisierung der Konstanten in Abbildung 5.12 zu sehen, kann der Wert einer solchen Konstanten vom Zeitpunkt (`TIME`) oder dem Ort seiner Initialisierung (`NAME`) abhängen, oder die Initialisierung kann Seiteneffekte haben. Daher ist es bei allen Konstanten außer den Übersetzungszeitkonstanten wichtig, dass ihre Initialisierung zentral zusammen mit der Initialisierung des Klassenobjektes abläuft und nur ihre Werte auf den anderen Knoten zwischengespeichert werden. Es werden zwei unterschiedliche Transformationen für Übersetzungszeitkonstanten und andere lokal zwischenspeicherbare Konstanten durchgeführt.

Transformation von Übersetzungszeitkonstanten Die Initialisierung von Übersetzungszeitkonstanten kann auf jedem Knoten der verteilten Umgebung unabhängig geschehen, da ihre Initialisierung überall denselben Wert liefern wird und keine Seiteneffekte auslösen kann. Dazu werden solche Konstanten wieder aus dem Klassenobjekt in die Stellvertreterklasse zurückverlagert. De facto findet also für solche Konstanten im Endeffekt überhaupt keine Transformation statt. Die Konstante `ID` der Ausgangsklasse `R` taucht im transformierten Code wieder als statisch finale Variable der Klasse `R` auf, die jetzt die Funktion der Stellvertreterklasse übernimmt. Entsprechend wird der Zugriff auf solche Konstanten nicht in Zugriffsmethoden umgeschrieben.

[9] Modifikationen nach der Initialisierung werden verhindert, indem die Erzeugung von Aliasnamen für solche Konstanten ebenfalls verboten wird. Ansonsten könnte die Modifikation über eine zweite, nicht mehr `const` deklarierte Referenz durchgeführt werden.

```
public class R_class {
   static long TIME;

   public long getTIME() {
      return R_class.TIME;
   }

   // Class initializer
   void _clinit() {
      R_class.TIME = System.currentTimeMillis();
   }

   // Static initializer
   static {
      if (isFactoryObject) {
         TIME = getClassObject().getTIME();
      }
   }
}
```

Abbildung 5.14: Die für die Konstante `TIME` relevanten Teile des Klassenobjektes.

Transformation dynamisch initialisierter Konstanten Exemplarisch für die dynamisch initialisierten Konstanten `TIME`, `NAME`, `ARRAY`, und `MESSAGE` zeigt Abbildung 5.14 den Ausschnitt aus dem Klassenobjekt `R_class`, der für die Konstante `TIME` aus der Originalklasse in Abbildung 5.12 erzeugt wird. Im Unterschied zu der statischen, aber nicht finalen Variablen `y` aus der Originalklasse in Abbildung 5.8 wird die Konstante `TIME` zu einer statischen Variablen der in Abbildung 5.14 gezeigten Klasse des Klassenobjektes. Wie bei der Beschreibung der entfernten Objektinstanzierung in Abschnitt 5.5.1 schon angesprochen, erfüllen die Instanzen der Klasse `R_class` zwei Funktionen. Auf genau einem Knoten der verteilten Umgebung repräsentiert eine Instanz von `R_class` die entfernte Klasse. Bei der Erzeugung dieses Klassenobjektes wird der in die Methode `_clinit()` verlagerte statische Initialisierer der Originalklasse ausgeführt. Wie in Abbildung 5.14 zu sehen ist, wird dabei auch die Konstante `TIME` initialisiert. Die andere Funktion der Klasse `R_class` ist das entfernte Erzeugen von Instanzen der Klasse `R_instance`. Auf Knoten der verteilten Umgebung, auf denen nicht das Klassenobjekt liegt, wird zu diesem Zweck ebenfalls eine Instanz von `R_class` angelegt, aber nicht initialisiert (die Methode `_clinit()` wird nicht aufgerufen). Wie im statischen Initialisierer in Abbildung 5.14 zu sehen, wird auf Knoten, welche die Klasse `R_class` nur zur Aufgabe der entfernten Objekterzeugung laden, die Konstante `TIME` ebenfalls initialisiert. Dazu wird eine entfernte Referenz auf das Klassenobjekt benutzt und die dort initialisierte Konstante über Fernzugriff ausgelesen.

Um für alle Konstanten einen einheitlichen Zugriff zu gewährleisten, tauchen dynamisch initialisierte Konstanten, wie in Abbildung 5.15 zu sehen, ebenfalls neben den Übersetzungszeitkonstanten in der Stellvertreterklasse auf. Sie werden dort aus der korrespondierenden Variablen der Klasse des Klassenobjektes initialisiert. In der

```
public class R {
   static final int  ID = 42;
   static final long TIME = R_class.TIME;
}
```

Abbildung 5.15: Die für die Konstanten `ID` und `TIME` relevanten Teile der Stellvertreterklasse.

Stellvertreterklasse ist `TIME` auch wieder eine Konstante, da sie wieder die Markierung `final` trägt. Code, der die Konstante `TIME` der entfernten Klasse `R` benutzt, greift nach der Transformation auf die Konstante in der Stellvertreterklasse zu. Interessant ist dabei die Initialisierungsreihenfolge. Geschieht ein Zugriff auf `R.TIME` in einem Knoten, so wird dort die Stellvertreterklasse geladen. Während ihrer Initialisierung wird aufgrund des Zugriffs `R_class.TIME` dort auch die Klasse `R_class` des Klassenobjektes geladen und initialisiert. Die Initialisierung der Stellvertreterklasse wird währenddessen blockiert. Der statische Initialisierer von `R_class` stößt seinerseits das eigentliche Laden und Initialisieren der entfernten Klasse an. Ihr Klassenobjekt wird entweder auf demselben Knoten erzeugt und initialisiert in `_clinit()` die statische Variable `R_class.TIME`, oder das Klassenobjekt wird auf einem anderen Knoten geladen und per entfernter Referenz verfügbar gemacht. Im zweiten Fall kopiert der statische Initialisierer von `R_class` den Wert von `TIME` in einem Fernaufruf in die eigene statische Variable `TIME`. Erst danach ist die Klasse `R_class` vollständig geladen und die Initialisierung der Stellvertreterklasse wird fortgesetzt. Sie initialisiert ihre Konstante `TIME` aus der jetzt lokal vorrätigen statischen Variablen der Klasse des Klassenobjektes. Ab diesem Zeitpunkt finden alle Zugriffe auf `R.TIME` so statt, als wäre nie eine entfernte Klasse im Spiel gewesen. Nachdem die Stellvertreterklasse geladen ist, stehen alle Konstanten der entfernten Klasse wieder als Konstanten der Stellvertreterklasse lokal zur Verfügung, wodurch alle Optimierungen für Konstanten der virtuellen Maschine unbeeinträchtigt greifen können.

5.5.4 Annotation der Objekterzeugung

Die Erzeugung eines Objektes in der verteilten Umgebung hat gegenüber einer regulären virtuellen Maschine den Ort der Erzeugung als zusätzlichen Freiheitsgrad. Da die Programmierumgebung verteilungstransparent ist – die Verteilung der Objekte hat keinen Einfluss auf die Programmsemantik – kann die Verteilung zum Zwecke der Lokalitätsoptimierung ausgenutzt werden und unabhängig vom eigentlichen Programm bestimmt werden. Prinzipiell sind drei Möglichkeiten denkbar, um die Objektverteilung von der Programmlogik zu trennen.

- Die Verteilung wird, getrennt von der Anwendungslogik, in einem eigenen Distributorobjekt spezifiziert. Das Distributorobjekt kann bei der Ausführung passend zur Anwendung konfiguriert werden.
- Zur Laufzeit kann durch das Laufzeitsystem die Kommunikation der Komponenten einer Anwendung beobachtet werden und daraus eine geeignete Verteilung der Objekte abgeleitet werden, welche die Kommunikation über Rechnergrenzen hinweg minimiert.

```
/** @at NodeExpression */
R r = new R(...);
```

Abbildung 5.16: Annotation der Erzeugung eines entfernten Objektes.

– Die Verteilung kann durch statische Analyse des Programms vom Übersetzer automatisch generiert werden. Dadurch können vom Übersetzer geeignete Direktiven in den Programmcode eingemischt werden, die zur Laufzeit den Ort von Objekterzeugungen steuern.

Der erste Ansatz hat sich als ungeeignet herausgestellt. Es ist im allgemeinen kaum möglich, einen Distributor unabhängig von der Anwendung zu implementieren. Bei einer Objekterzeugung müsste dieser Distributor einen Knoten für das neue Objekt auswählen, ohne den Kontext zu kennen, in dem dieses Objekt erzeugt wurde. Insbesondere bei der Objekterzeugung in mehreren parallelen Kontrollfäden kann der Distributor aus Mangel an Kontextinformationen keine informierte Entscheidung über eine gute Platzierung treffen.

Die zweite Alternative hat ebenfalls Nachteile. Um optimale Ergebnisse zu erzielen, muss ein geeigneter Ort für das neue Objekt vor seiner Erzeugung bestimmt werden. Durch Beobachtung des Kommunikationsverhaltens einer Anwendung kann aber lediglich die Güte einer gegebenen Verteilung abgeschätzt werden. Um einen guten Ort für eine Objekterzeugung zu bestimmen, müsste demzufolge das zukünftige Kommunikationsverhalten des gerade erzeugten Objektes bekannt sein. Diesen Blick in die Zukunft kann aber eine reine Beobachtung der Anwendung nicht leisten, da diese immer nur in die Vergangenheit blickt. Die Beobachtung der Anwendung und ein notwendig werdender nachträglicher Umzug von Objekten sind ebenfalls mit Laufzeitkosten verbunden.

Die letzte Alternative hat prinzipiell die Möglichkeit, an der Stelle der Objekterzeugung einen Blick in die Zukunft zu werfen, indem der von dort ausgehende Programmfluss analysiert wird und zukünftige Interaktionen extrapoliert werden, die mit dem an einer bestimmten Stelle erzeugten Objekt stattfinden werden. Kann die Analyse ausreichend Informationen gewinnen, ist hiermit eine informierte Entscheidung über den Ort für eine Objekterzeugung möglich. Dieser Ansatz wurde vom Autor in [82, 83] untersucht. Im Ergebnis kann eine solche statische Analyse in manchen Situationen erfolgreich sein, während sie in anderen aufgrund von Polymorphie zu ungenaue Informationen über den Programmfluss extrahieren kann, um gute Platzierungsentscheidungen abzuleiten.

Daher wurde in der verteilten Umgebung eine Möglichkeit eingeführt, mit welcher der Programmierer in die Platzierungsentscheidungen des Systems eingreifen kann. Zu diesem Zweck werden Verteilungshinweise eingeführt, die in Form einer Annotation der Objekterzeugung auftreten. Dadurch ist der Verteilungshinweis syntaktisch nahe genug an der Stelle der Objekterzeugung, um den Kontext mit in die Platzierungsentscheidung einbeziehen zu können. Verteilungshinweise können dabei allgemein genug spezifiziert werden, um eine Verteilung unabhängig von der tatsächlichen Größe der Laufzeitumgebung während der Programmausführung anzugeben.

Abbildung 5.16 zeigt die Syntax einer Annotation der Objekterzeugung. Die Annotation verwendet einen Dokumentationskommentar mit einem neuen Etikett `@at`.

Als Argument kann ein beliebiger Java-Ausdruck `NodeExpression` vom Typ `int` gegeben werden, der im Kontext der Erzeugung zu einer virtuellen Knotennummer ausgewertet wird. Die Auswahl eines Knotens der verteilten Umgebung für die Objekterzeugung zur Laufzeit ist zweistufig. Zuerst wird `NodeExpression` ausgewertet, das Ergebnis als virtuelle Knotennummer interpretiert und danach auf eine reale Knotennummer der verteilten Umgebung abgebildet. Die Abbildung von virtueller (vnr) in eine reale Knotennummer (rnr) erfolgt reihum (rnr = vnr modulo Knotenanzahl).

Mit der vorgestellten Annotationssyntax lassen sich beliebige Verteilungen angeben. Insbesondere ist es auch möglich, neue Objekte relativ zu schon existierenden Objekten zu erzeugen. Beispielsweise lässt sich mit `@at getLocation(s)` ein neues Objekt auf demselben Knoten anlegen, auf dem bereits ein anderes entferntes Objekt `s` liegt. Die Anwendung hat weiterhin die Wahl, eine eigene Abstraktionsebene für die Objektverteilung zu verwenden. Beispielsweise ist denkbar, dass eine Bibliothek eine geeignete Parametrisierung verwendet, um eine Abstimmung der Verteilung der durch sie erzeugten Objekte mit der Anwendung zu ermöglichen.

Kapitel 6

Transparente Replikation

Die in Abschnitt 3.6 eingeführte kollektive Replikation wird in der vorliegenden Arbeit mit Hilfe *replizierter* Klassen ausgedrückt. Während eine Instanz einer entfernten Klasse und die entfernte Klasse selbst zur Laufzeit des Programms auf einem dedizierten Knoten der verteilten Umgebung angelegt sind und der Kontrollfluss beim Zugriff auf diesen Knoten wechselt, ist ein repliziertes Objekt bzw. die replizierte Klasse auf mehreren Knoten gleichzeitig lokal vorrätig. Ihre Kopien (Replikate) befinden sich auf allen Knoten in einem konsistenten Zustand, solange aktuell keine Zustandsfortschreibung durchgeführt wird. Konsistenz bedeutet intuitiv, dass die Replikate in allen ihren Werten übereinstimmen. Abschnitt 6.1.2 definiert Konsistenz für ein repliziertes Objekt genauer. Im Gegensatz zu einem entfernten Objekt wird auf ein repliziertes Objekt immer lokal zugegriffen. Der Zugriff findet an demjenigen Replikat statt, das sich auf dem Rechenknoten befindet, auf welchem der Zugriff durchgeführt wird. Dabei spielt es keine Rolle, ob es sich dabei um einen lesenden oder einen schreibenden Zugriff handelt. Auch ein schreibender Zugriff modifiziert nur das lokale Replikat, wodurch die Konsistenz zwischenzeitlich verloren geht. Erst die auf eine Modifikation folgende Zustandsfortschreibung zieht die von der Anwendung an einem Replikat lokal durchgeführte Änderung auf allen anderen Replikaten nach und bringt sie dadurch wieder in einen konsistenten Zustand.

In den folgenden Abschnitten werden transparent replizierte Klassen als neues Ausdrucksmittel eingeführt und anschließend die Techniken beschrieben, die deren Umsetzung in reinem Java mittels einer Programmtransformation erlauben.

6.1 Transparent replizierte Klassen

Abbildung 6.1 zeigt die Deklaration einer replizierten Klasse `P`. Sie unterscheidet sich von einer regulären Java-Klasse durch die Markierung `replicated` an der Klassendeklaration. Diese Markierung bewirkt eine Transformation durch den Übersetzer, die zur Laufzeit eine Replikation von `P` und ihrer Instanzen in der verteilten Umgebung bewirkt. Eine replizierte Klasse kann Instanz- und Klassenvariablen enthalten. Im Beispiel aus Abbildung 6.1 ist der Typ von `x` ein Basistyp, während die Variable `s` zur Laufzeit auf ein lokales Objekt, die Variablen `r` und `p` auf ein entferntes bzw. anderes repliziertes Objekt verweisen. Die möglichen Typen von Variablen einer replizierten

```
replicated class P {
   int x;
   String s;
   R r; // entferntes Objekt
   P p; // anderes repliziertes Objekt

   static int y = 42;

   P(R r, int x) {
      this.r = r;
      this.x = x;
   }

   void foo() {...}
   static void bar() {...}
}
```

Abbildung 6.1: Deklaration einer replizierten Klasse mit primitivwertigen Instanzvariablen und Referenzen auf entfernte und andere replizierte Objekte.

```
/** @at new int[] {0, 1, 2} */
P p = new P();
```

Abbildung 6.2: Erzeugung eines replizierten Objektes mit Replikaten auf den Rechenknoten 0, 1 und 2.

Klasse sind also nicht eingeschränkt. Wie jede andere Klasse kann auch eine replizierte Klasse Konstruktoren, einen statischen Initialisierer und sowohl statische als auch nicht statische Methoden haben. Damit kann eine replizierte Klasse wie eine normale Java-Klasse deklariert werden, was die Bezeichnung als transparent replizierte Klasse rechtfertigt.

Erzeugt wird eine Instanz einer replizierten Klasse wie bei lokalen und entfernten Klassen über den Aufruf eines Konstruktors in einem `new`-Ausdruck. Ähnlich der für entfernte Klassen in Abschnitt 5.5.4 eingeführten Annotation der Objekterzeugung kann auch die Erzeugung eines replizierten Objektes annotiert werden. Während die Annotation für entfernte Objekte die Nummer des Rechenknotens angibt, auf dem das entfernte Objekt erzeugt werden soll, wird bei der Erzeugung eines replizierten Objektes eine Knotenmenge erwartet. In diesem Fall wird auf jedem angegebenen Knoten ein Replikat des replizierten Objektes erzeugt. Über diese Annotation wird also sowohl der Replikationsgrad des neuen Objektes bestimmt als auch der Ort, an dem seine Replikate erzeugt werden. Fehlt die Annotation, so wird auf jedem Rechenknoten der verteilten virtuellen Maschine ein Replikat angelegt. Von replizierten Klassen selbst wird immer auf jedem Rechenknoten ein Replikat erstellt, da ihre Erzeugung implizit beim Klassenladen geschieht und daher nicht annotiert werden kann. Abbildung 6.2 zeigt ein Beispiel für die Instanzierung der Klasse aus Abbildung 6.1, wobei ein Replikat nur auf den Rechenknoten 0, 1 und 2 angelegt wird. Die `@at`-Annotation wird dazu mit einem Feld aus Ganzzahlen parametrisiert, welche die Nummern der Rechenknoten angeben, auf denen Replikate erzeugt werden sollen.

Wie schon in der Einführung von replizierten Klassen in Abschnitt 3.7.4 begründet, bedarf es beim Zugriff auf replizierte Objekte in einer verteilten Umgebung erweiterter Koordinationsmechanismen, um eine konsistente Sicht auf den replizierten Zustand zu gewährleisten und gleichzeitig einen Gewinn aus der Replikation ziehen zu können. Abschnitt 6.1.1 klärt, was zum replizierten Zustand eines replizierten Objektes gehört, und Abschnitt 6.1.2 definiert, wann dieser Zustand als konsistent gilt. Die Abschnitte 6.1.3 sowie 6.1.5 führen erweiterte Synchronisationsprimitive ein, mit denen eine konsistente Sicht auf ein repliziertes Objekt sichergestellt wird bzw. die eine Zustandsfortschreibung von einem konsistenten Zustand in einen neuen konsistenten Zustand ermöglichen. Synchronisation in Java kann immer auch zur Interaktion zwischen Kontrollfäden benutzt werden. Abschnitt 6.1.4 erweitert die bekannten Konstrukte für den Austausch von Benachrichtigungen zwischen Kontrollfäden für replizierte Objekte. Der abschließende Abschnitt 6.1.6 stellt eine Erweiterung von replizierten Objekten zu partiell replizierten Objekten vor, die unabdingbar für den effizienten Einsatz bei großen, datenparallel verarbeiteten Strukturen sind.

Die folgenden Abschnitte diskutieren nur replizierte Objekte, da die replizierte Klasse selbst analog zu einer entfernten Klasse als ein spezielles, repliziertes Einzelobjekt (das Klassenobjekt) betrachtet werden kann. Die einzige Besonderheit des Klassenobjektes ist, dass es in jedem Kontext über den Klassennamen direkt angesprochen werden kann, ohne eine explizite Referenz zu besitzen. Die Behandlung der replizierten Klasse als solche wird erst wieder bei der Programmtransformation für replizierte Klassen in Abschnitt 6.3 wichtig.

6.1.1 Replizierter Zustand

Um zu verstehen, was zum replizierten Zustand eines replizierten Objektes gehört, soll wieder die Klasse `P` aus Abbildung 6.1 als Beispiel dienen. Für den Moment wird vorausgesetzt, dass ein repliziertes Objekt zur Laufzeit aus einer Menge von Replikaten auf unterschiedlichen Knoten besteht. Bei einer Instanz von `P` besteht dann jedes dieser Replikate aus den Instanzvariablen `x`, `s`, `r` und `p`. Die Variable `x` beinhaltet einen Wert eines Basistyps und gehört daher direkt zum Zustand des Replikats. Die Variable `s` dagegen hält eine Referenz auf ein anderes Objekt, in diesem Fall auf eine Zeichenkette. Nur die Referenz auf dieses Objekt (oder der Wert `null`) ist unmittelbar Teil des Replikats. Allerdings gehören alle von einem replizierten Objekt referenzierten lokalen Objekte implizit mit zu seinem replizierten Zustand. Da es sich bei dem über `s` referenzierten Objekt um ein solches lokales Objekt handelt, wird es implizit mitrepliziert. Für ein lokales Objekt ist die Replikation seiner Referenz (in den Instanzvariablen aller Replikate eines replizierten Objektes) allein gar nicht möglich, ohne bei jedem Replikat des replizierten Objektes ebenfalls Replikate des referenzierten lokalen Objektes anzulegen. Dies rührt daher, dass eine Referenz auf ein lokales Objekt nur auf der virtuellen Maschine gültig ist, auf der sich das zugehörige Objekt befindet.

Anders verhält es sich mit den Referenzen `r` und `p`. Die erste referenziert ein entferntes Objekt der Klasse `R` (beispielsweise der Klasse aus Abbildung 5.8), während die zweite auf ein weiteres repliziertes Objekt der Klasse `P` zeigt. In diesen beiden Fällen ist es nicht nur möglich, sondern durchaus sinnvoll und notwendig, nur die *Referenzen* zu replizieren. Das mit `r` referenzierte entfernte Objekt befindet sich mögli-

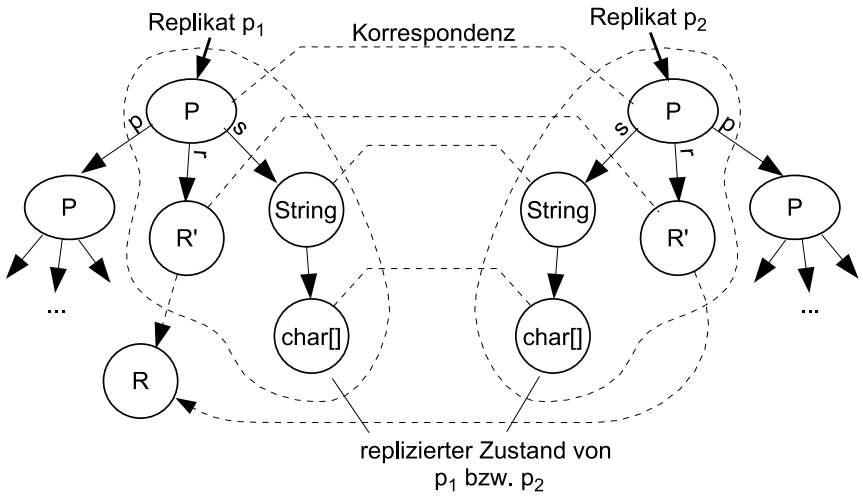


Abbildung 6.3: Replizierter Zustand einer Instanz von `P` aus Abbildung 6.1 bestehend aus zwei Replikaten p_1 und p_2.

cherweise auf einem entfernten Knoten und kann somit nicht Teil des Zustandes eines bestimmten Replikats sein. Ein entferntes Objekt bleibt also auch dann ein entferntes Objekt auf einem dedizierten Knoten, wenn es aus einem replizierten Objekt heraus referenziert wird. Nur die entfernte Referenz (die ihrerseits nichts anderes als ein lokales Stellvertreterobjekt ist) wird mitrepliziert. Ein anderes repliziertes Objekt besteht seinerseits aus mehreren Replikaten und kann somit nicht sinnvoll Teil des Zustandes eines einzigen Replikats sein. Daher gehört nur die Referenz `p`, die auf ein anderes repliziertes Objekt zeigt, zum replizierten Zustand des untersuchten Ausgangsobjektes.

Die Regel für die Zugehörigkeit zum replizierten Zustand muss transitiv fortgesetzt werden, da auch lokale Objekt, die über mehrere Indirektionsstufen von diesem Replikat aus referenziert werden, mitrepliziert werden müssen. Der replizierte Zustand eines replizierten Objektes erstreckt sich somit, ausgehend von einem Replikat als Wurzel, auf alle von dort aus über lokale Referenzen erreichbaren Objekte. Er endet bei entfernten Referenzen oder Referenzen auf andere replizierte Objekte. Ein Beispiel für ein repliziertes Objekt der Klasse `P` mit zwei Replikaten p_1 und p_2 ist in Abbildung 6.3 zu sehen. Der zu den Replikaten gehörende replizierte Zustand ist als gestrichelter Bereich eingezeichnet. Dieser replizierte Zustand umfasst sowohl das lokale Objekt der Klasse `String` als auch alle von diesem referenzierten Objekte (angedeutet durch den Zeichenpuffer des Zeichenkettenobjektes). Da eine entfernte Referenz auf ein Objekt der Klasse `R` ebenfalls durch ein lokales Objekt (der Stellvertreterklasse `R'`) repräsentiert wird, ist das entfernte Referenzobjekt im replizierten Zustand enthalten, während sich die Instanz der entfernten Klasse `R` außerhalb befindet. In der Abbildung 6.3 ist zu sehen, dass die beiden replizierten entfernten Referenzen auf dasselbe entfernte Objekt zeigen. Die Referenz `p` auf ein anderes repliziertes Objekt ist eine reguläre Java-Referenz auf sein Replikat. Daher verlässt diese Referenz direkt den replizierten Zustand. Die Referenzen `p` in beiden Replikaten zeigen dabei auf unterschiedliche Objekte (auf die jeweiligen Replikate des referenzierten replizierten Objektes). Indem lokale Objekte mitrepliziert werden, besteht der Zustand jedes Replikats aus einer Menge korrespondierender Objekte. Zum Objekt $p_1.s$ im Zustand von p_1 gehört ein entspre-

chendes Objekt im Zustand von p_2. Diese beiden lokalen Objekte $p_1.s$ und $p_2.s$ stehen wie p_1 und p_2 in einer Replikationsbeziehung. Diese Korrespondenz der Objekte ist in Abbildung 6.3 durch gestrichelte Verbindungslinien eingezeichnet.

Im Folgenden meint die Bezeichnung „Replikat“ den gesamten replizierten Zustand auf einem Rechenknoten, also jeweils eine der beiden mit einer gestrichelten Linie umschlossenen Mengen von Objekten in Abbildung 6.3. Nur wo eine Unterscheidung zwischen der Instanz der replizierten Klasse (dem Objekt der Klasse `P` jeweils ganz oben in Abbildung 6.3) und den von ihr referenzierten und daher mitreplizierten lokalen Objekten wichtig ist, wird von dem Wurzelobjekt eines Replikats und seinen Teilzuständen gesprochen. Mit „repliziertem Objekt“ ist immer die Gesamtheit aller seiner Replikate gemeint. Abschnitt 6.1.6 untergliedert die Teilzustände eines Replikats für partielle Replikation weiter in sog. „Hauptobjekte“ und deren Teilzustände.

6.1.2 Konsistenz

Als nächstes soll definiert werden, wann sich zwei Replikate p_i und p_j eines replizierten Objektes in einem konsistenten Zustand befinden. Intuitiv muss dafür gelten, dass ihre replizierten Zustände übereinstimmen. Bei primitivwertigen Instanzvariablen $p_i.x$ und $p_j.x$ ist klar, dass sie genau dann übereinstimmen, wenn sie identische Werte enthalten. Für referenzwertige Instanzvariablen ist die Forderung „identischer Werte“ (mit Ausnahme von `null`) auf unterschiedlichen Knoten keine sinnvolle Bedingung für die Konsistenz. Die folgenden Abschnitte definieren Konsistenz für referenzwertige Instanzvariablen je nach Art der Referenz. Dabei wird nach Referenzen auf lokale, entfernte und replizierte Objekte unterschieden.

Ein von einem Replikat p_i referenziertes lokales Objekt $p_i.s$ gehört gemäß Abschnitt 6.1.1 ebenfalls zum seinem replizierten Zustand. Folglich müssen lokale Objekte s_i und s_j existieren, die ebenfalls in einer Replikationsbeziehung zueinander stehen. Diese Replikationsbeziehung ist eine Bijektion $\rho_{i,j} \subseteq \mathbb{O}_i \times \mathbb{O}_j$ zwischen den Objektmengen $\mathbb{O}_i$ und $\mathbb{O}_j$ der Replikate p_i und p_j. Die Replikationsbeziehung $\rho_{i,j}$ ordnet jedem Objekt s_i im Replikat p_i über $(s_i, s_j) \in \rho_{i,j}$ umkehrbar eindeutig ein Objekt s_j im Replikat p_j zu. Die lokalen Referenzen $p_i.s$ und $p_j.s$ sind genau dann konsistent, wenn gilt $(p_i.s, p_j.s) \in \rho_{i,j}$. Korrespondierende lokale Objekte sind Instanzen derselben Klasse und die hier beschriebenen Konsistenzregeln gelten rekursiv für ihre eigenen Instanzvariablen. Die Forderung, dass $\rho_{i,j}$ eine Bijektion ist, bewirkt, dass die Objektgraphen der Replikate dieselbe Struktur aufweisen müssen, um konsistent zu sein. Diese Bijektion ist in Abbildung 6.3 durch gestrichelte Linien zwischen den Objekten der beiden Replikate eingezeichnet.

Zwei entfernte Referenzen $p_i.r$ und $p_j.r$ sind dann konsistent, wenn sie dasselbe entfernte Objekt r referenzieren. Auf einer abstrakten Ebene stimmen damit in einem konsistenten Zustand tatsächlich die Referenzen überein. Auf der konkreten Ebene der Java-Objekte sind entfernte Referenzen aber nichts anderes als spezielle lokale Objekte, die ein entferntes Objekt identifizieren und für dieses als Stellvertreter fungieren. Konkret referenzieren $p_i.r$ und $p_j.r$ zwei korrespondierende lokale Stellvertreterobjekte für dasselbe entfernte Objekt r. So gesehen ist die Konsistenz von entfernten Referenzen ein Spezialfall der Konsistenz lokaler Referenzen.

Im dritten Fall, der Referenz auf ein anderes repliziertes Objekt, besteht dann Kon-

sistenz, wenn beide Referenzen $p_i.p$ und $p_j.p$ auf Replikate desselben replizierten Objektes verweisen. Da der Zustand des referenzierten replizierten Objektes nicht Teil des replizierten Ausgangsobjektes ist, ist für die Konsistenz der Replikate p_i und p_j die Konsistenz der Replikate $p_i.p$ und $p_j.p$ des anderen replizierten Objektes keine Voraussetzung. Damit können Replikate eines replizierten Objektes konsistent sein, obwohl sie auf andere replizierte Objekte verweisen, die sich nicht in einem konsistenten Zustand befinden.

Ein repliziertes Objekt als Ganzes befindet sich genau dann in einem konsistenten Zustand, wenn alle seine Replikate paarweise konsistent sind. Nach seiner Erzeugung (bei Rückkehr des Konstruktors einer replizierten Klasse) befindet sich ein repliziertes Objekt in einem konsistenten Zustand. Die nächsten beiden Abschnitte führen Mechanismen ein, mit denen sich dieser initial konsistente Zustand so fortschreiben lässt, dass das Objekt von mehreren Kontrollfäden benutzt werden kann und trotzdem alle Kontrollfäden immer einen konsistenten Zustand beobachten.

6.1.3 Lese- und Schreibsynchronisation

Dieser Abschnitt beschreibt die Verwendung des ersten Modus der Replikation, in dem replizierte Objekte zur Optimierung verteilter Leseoperationen eingesetzt werden können. Befindet sich ein repliziertes Objekt in einem konsistenten Zustand, so können Lesezugriffe auf dieses Objekt verteilt parallel ausgeführt werden. Jeder dieser Lesezugriffe beobachtet dann dieselben Daten. Im Vergleich zum Einsatz eines entfernten Objektes finden aber keine Fernaufrufe statt, da der Zustand eines replizierten Objektes lokal vorliegt. Wird allerdings eine Schreiboperation auf einem replizierten Objekt durchgeführt, müssen die Lesezugriffe solange unterbunden werden, bis sich wieder alle Replikate in einem konsistenten Zustand befinden.

Analog zu regulärem Java wird ein Synchronisationskonstrukt eingesetzt, um nebenläufige Zugriffe auf ein repliziertes Objekt so voneinander abzugrenzen, dass jede zugreifende Aktivität das Objekt immer in einem konsistenten Zustand wahrnimmt. Anders als in regulärem Java, wo Synchronisation immer ein Objekt exklusiv sperrt, wird bei replizierten Objekten zwischen einer exklusiven Sperre für einen Schreibzugriff und einer gemeinsamen Sperre für einen Lesezugriff unterschieden. Abbildung 6.4 zeigt die Syntax an einem Beispiel. Im oberen Teil der Abbildung sieht man zwei synchronisierte Methoden `setX()` und `getX()` einer replizierten Klasse `P`. Neben dem aus Java bekannten Schlüsselwort `synchronized` tragen diese Methoden die Markierung `shared` bzw. `exclusive`. Der untere Teil der Abbildung 6.4 zeigt zwei synchronisierte Blöcke. Auch diese Form der Synchronisation lässt sich mit der Markierung `shared` bzw. `exclusive` versehen.

Von ihrer Sperrsemantik entspricht die exklusive Sperre der regulären Java-Sperre an lokalen Objekten. Bei der Synchronisation an replizierten Objekten muss die Art der Sperre immer explizit angegeben werden. Ein Synchronisationskonstrukt ohne Markierung ist für replizierte Objekte nicht erlaubt. Diese Bedingung zwingt den Programmierer über die Art der Sperre nachzudenken. Dies ist wichtig, da bei ausschließlicher Verwendung von exklusiven Sperren kein Vorteil durch die Replikation erzielt werden kann. Bei Gleichsetzung von exklusiver Synchronisation mit regulärer Synchronisation wäre aber die Gefahr gegeben, dass die Markierung `shared` aus Bequemlichkeit

```
replicated class P {
   int x;

   exclusive synchronized void setX(int x) {
      this.x = x;
   }

   shared synchronized int getX() {
      return x;
   }
}

P p;
...
shared synchronized(p) {
   ...
}

exclusive synchronized(p) {
   ...
}
```

Abbildung 6.4: Deklaration von synchronisierten Methoden zum Schreiben und Lesen.

vergessen wird.

Bei den Sichtbarkeitsregeln unterscheidet sich die exklusive Synchronisation an replizierten Objekten leicht von der regulären Java-Synchronisation an lokalen Objekten. Das Java-Speichermodell schreibt vor, dass nach dem Freigeben einer Sperre alle Änderungen für andere Kontrollfäden sichbar werden, nachdem diese einen beliebigen anderen synchronisierten Bereich betreten haben [31].[1] Beim Freigeben einer exklusiven Sperre an einem replizierten Objekt werden dagegen nur die Änderungen an dem replizierten Zustand des Objektes sichtbar, an dem synchronisiert wurde. Anders ausgedrückt findet nur eine Zustandsfortschreibung für das replizierte Objekt statt, das für die Synchronisation verwendet wurde. Der Unterschied zur Java Synchronisation besteht darin, dass dort die Änderungen an allen Objekten sichtbar werden, unabhängig davon, an welchem Objekt synchronisiert wurde. Diese Einschränkung gilt allerdings nur für replizierte Objekte. Alle Änderungen, die während einer exklusiven Synchronisation an lokalen Objekten getätigt wurden, werden nach Freigeben der Sperre genauso sichtbar, wie bei regulärer Java-Synchronisation.

Hält ein Kontrollfaden eine exklusive Synchronisation (`exlusive synchronized`) an einem replizierten Objekt, verhindert er dadurch alle weiteren Synchronisationsversuche an diesem Objekt. Die beim Betreten einer exklusiv synchronisierten Methode oder eines exklusiv synchronisierten Blocks gesetzte Sperre entspricht von ihrer Semantik der Sperre, die in Java durch Synchronisation an einem Objekt gesetzt wird. Bevor ein Kontrollfaden eine exklusive Sperre an einem replizierten Objekt

[1] Diese Forderung wird in Java 1.5 leicht abgeschwächt, indem nur noch solche Kontrollfäden die Änderungen sehen müssen, die sich im folgenden an derselben Sperre synchronisieren [62].

durchsetzen kann, müssen alle anderen (gemeinsamen oder exklusiven) Sperren an demselben Objekt freigegeben worden sein. Während ein Kontrollfaden eine exklusive Sperre hält, kann kein anderer Kontrollfaden eine Sperre an demselben replizierten Objekt durchsetzen. Während einer exklusiven Synchronisation darf ein Kontrollfaden Modifikationen am Zustand des replizierten Objektes durchführen. Dadurch geht zwar die Konsistenz des replizierten Objektes zwischenzeitlich verloren, jedoch kann bei korrekter Synchronisation[2] kein anderer Kontrollfaden diese Inkonsistenz beobachten. Bevor ein Kontrollfaden eine exklusive Sperre beim Verlassen eines synchronisierten Bereiches wieder aufgibt, wird eine Zustandsfortschreibung durchgeführt, welche die an einem Replikat des replizierten Objektes gemachten Änderungen an alle seine Replikate verteilt. Die Zustandsfortschreibung am Ende eines exklusiv synchronisierten Bereiches stellt somit die Konsistenz des replizierten Objektes wieder her. Nachdem eine exklusive Sperre freigegeben wurde, führen damit Lesezugriffe auf seine Replikate wieder zu konsistenten Beobachtungen.

Befindet sich ein Kontrollfaden in einem gemeinsam synchronisierten Bereich eines replizierten Objektes (`shared synchronized`), darf er seinen Zustand auslesen, aber keine Modifikationen daran durchführen. Im Gegensatz zu Synchronisation in Java schließt sich gemeinsame Synchronisation nicht wechselseitig aus, sondern verhindert lediglich die nebenläufige Ausführung von exklusiv synchronisierten Bereichen desselben Objektes. Es ist damit möglich, dass sich mehrere Kontrollfäden gleichzeitig in einem gemeinsam synchronisierten Bereich desselben replizierten Objektes befinden. Dadurch können insbesondere Lesezugriffe auf einem replizierten Objekt verteilt parallel durchgeführt werden, ohne dass sie sich gegenseitig behindern. Im Beispiel aus Abbildung 6.4 könnte sich somit ein Kontrollfaden innerhalb der Methode `getX()` aufhalten, während ein anderer Kontrollfaden gleichzeitig den gemeinsam synchronisierten Block im unteren Teil ausführt. Die parallele Ausführung mehrerer gemeinsam synchronisierter Bereiche ist lediglich eine Möglichkeit, aber keine Garantie. Wenn sich demnach ein Kontrollfaden innerhalb der synchronisierten Lesemethode `getX()` befindet und ein anderer Kontrollfaden währenddessen versucht, dieselbe Methode zu betreten, kann das Laufzeitsystem entweder die nebenläufige Ausführung zulassen oder die beiden Ausführungen sequentialisieren.

Ein und derselbe Kontrollfaden kann dieselbe Synchronisationsart (`shared` oder `exclusive`) an demselben replizierten Objekt mehrfach blockierungsfrei anfordern. Es ist ebenfalls problemlos möglich, aus einem exklusiv synchronisierten Bereich eine gemeinsame Sperre anzufordern. Eine solche Anforderung wird ebenfalls immer blockierungsfrei erteilt. Die Verschachtelung einer exklusiven Synchronisation in eine gemeinsame Synchronisation ist möglich, sie wird aber nicht blockierungsfrei erteilt. Die Gefahr einer Verklemmung wird in dieser Situation vermieden, indem die gemeinsame Synchronisation vor Anforderung der exklusiven Sperre kurzfristig aufgegeben wird (wie bei einem Aufruf der Methode `wait()` aus der Klasse `Object`). Nach Betreten der exklusiven Synchronisation kann die Anwendung daher nicht sicher sein, denselben Zustand zu beobachten wie vor dem Betreten der exklusiven Synchronisation, da einem anderen Kontrollfaden zwischenzeitlich Zugriff auf das Objekt gewährt worden sein könnte.

[2] Ist das Programm nicht korrekt synchronisiert, so können wie in regulärem Java ebenfalls inkonsistente Zustände beobachtet werden.

```
synchronized(x) {
   // Wächter, der das Erfülltsein der Bedingung
   // CONDITION_ON_THE_STATE_OF_X zusichert.
   while (! CONDITION_ON_THE_STATE_OF_X) {
      x.wait();
   }
   ...
}

synchronized(x) {
   // Zustandsveränderung von x, welche die Bedingung
   // CONDITION_ON_THE_STATE_OF_X erfüllt.
   x.notifyAll();
}
```

Abbildung 6.5: Muster für den Einsatz von Primitiven zur Kommunikation zwischen Kontrollfäden in Java.

6.1.4 Kommunikation zwischen Kontrollfäden

Koordination nebenläufiger Kontrollfäden ist nicht durch geschützte Bereiche allein zu erreichen. Ebenso wichtig sind Kommunikationsmechanismen zwischen Kontrollfäden, um Signale über eingetretene Ereignisse zu senden und zu empfangen. Java stellt dazu die Primitive `wait()` und `notifyAll()` zur Verfügung, die als Methoden von `Object`, der gemeinsamen Oberklasse aller Objekte, deklariert sind. Abbildung 6.5 stellt ein Musterbeispiel für die Verwendung dieser Kommunikationsprimitive vor. Mit `wait()` wartet ein Kontrollfaden auf das Eintreten einer Bedingung, die nicht durch ihn selbst, sondern nur durch einen anderen Kontrollfaden erfüllt werden kann. Mit `notifyAll()` sendet ein Kontrollfaden ein Signal, wenn er den Zustand eines Objektes so verändert hat, dass eine Bedingung erfüllt wird, auf die ein anderer Kontrollfaden möglicherweise wartet. Sowohl um ein Signal zu senden als auch um auf das Eintreffen eines Signals an einem Objekt zu warten, muss sich ein Kontrollfaden in einem geschützten Bereich des zugehörigen Objektes befinden. Während er auf das Eintreten einer Bedingung wartet, gibt er die von ihm gehaltene Sperre auf, damit ein anderer Kontrollfaden den geschützten Bereich betreten kann, um die Bedingung durch Zustandsänderung zu erfüllen.

Derselbe Mechanismus steht auch für replizierte Objekte zur Verfügung. Aufgrund der Unterteilung in Schreib- und Lesesynchronisation, sind aber nicht alle Kombinationen der Kommunikationsprimitive sinnvoll. Das Senden eines Signals setzt eine Änderung des zum synchronisierten Objektes gehörigen Zustandes voraus. Nur durch eine solche Zustandsänderung kann an anderer Stelle eine erwartete Bedingung erfüllt werden. Demzufolge ist die Verwendung von `notifyAll()` nur in exklusiv synchronisierten Bereichen sinnvoll und erlaubt,[3] da nur dort der Zustand eines replizierten Objektes modifiziert werden darf. Das Warten auf das Eintreten einer Bedingung ist dagegen sowohl innerhalb einer gemeinsamen wie einer exklusiven Synchronisation

[3]Die Verwendung von `notifyAll()` in einem gemeinsam synchronisierten Bereich führt zu einer Ausnahmebedingung während der Laufzeit.

möglich. In beiden Fällen wird die Sperre zwischenzeitlich freigegeben, damit ein anderer Kontrollfaden während einer exklusiven Synchronisation für die Erfüllung der erwarteten Bedingung sorgen kann.

Handelte es sich bei `x` in Abbildung 6.5 um ein repliziertes Objekt, so müsste der obere synchronisierte Block die Markierung `shared` oder `exclusive` tragen, während für den unteren Block nur die Markierung `exclusive` erlaubt wäre. Die zusätzliche Bedingung, dass die Verwendung von `notifyAll()` auf exklusiv synchronisierte Bereiche beschränkt ist, stellt dabei aber keine Einschränkung gegenüber dem Programmiermodell von Java dar. Diese Bedingung ist lediglich eine Konsequenz aus der dort formulierten Forderung, das Senden von Signalen immer mit einer Zustandsänderung zu verbinden. Während diese Forderung für reguläre Java-Programme nur implizit gestellt wird, ist sie für replizierte Objekte explizit gemacht.

Erzeugt ein Kontrollfaden in einem exklusiv synchronisierten Bereich neben einer Zustandsänderung ein Signal, muss dieses vor dem Verlassen des geschützten Bereichs zusammen mit der Zustandsänderung an alle Replikate des replizierten Objektes ausgeliefert werden. Denn dort warten möglicherweise verteilt mehrere Kontrollfäden in gemeinsam oder exklusiv synchronisierten Bereichen auf das Eintreten einer Bedingung, die mit dem Einspielen der Zustandsfortschreibung erfüllt wird. Ein an einem replizierten Objekt erzeugtes Signal ist somit Teil der Zustandsfortschreibung und wird zusammen mit dieser an allen seinen Replikaten ausgeliefert.

Die exklusive Synchronisation erlaubt immer nur einem einzigen Kontrollfaden gleichzeitig den Zustand eines replizierten Objektes zu ändern. Mit den in diesem Abschnitt eingeführten Kommunikationsprimitiven kann die Reihenfolge solcher Änderungen koordiniert werden. Allerdings werden alle in exklusiven Synchronisationen durchgeführten Änderungen sequentialisiert, so dass damit keine Parallelität bei Zustandsänderungen replizierter Objekte erreicht werden kann. Der folgende Abschnitt führt eine weitere Synchronisationsform ein, mit der koordinierte parallele Modifikationen an einem replizierten Objekt durchgeführt werden können.

6.1.5 Kollektive Synchronisation

Mit den bisher eingeführten Mitteln sind Berechnungen nur dann parallel formulierbar, wenn sie den Zustand eines replizierten Objektes ausschließlich lesen, da alle Änderungen am replizierten Zustand exklusiv durchgeführt werden müssen und daher sequentialisiert werden. Dies ist nur dann ein gangbarer Weg, wenn der replizierte Zustand sehr selten geändert wird. Dieser Abschnitt beschreibt wie auch Zustandsänderungen an einem replizierten Objekt parallel durchgeführt werden können.

Unkoordinierte nebenläufige Änderungen aus parallelen Kontrollfäden an gemeinsamem Zustand haben das inhärente Problem, dass dadurch inkonsistente Teiländerungen beobachtet werden können. Dennoch sind parallele Modifikationen erforderlich, um das Potenzial an Parallelität eines Rechnerbündels auszuschöpfen. Um beides zu erreichen – Änderungen parallel durchführen zu können und trotzdem die Beobachtung inkonsistenter Zustände auszuschließen – müssen die parallelen Änderungen eng verzahnt werden. Dazu müssen die durchführenden Kontrollfäden während ihrer Modifikationsoperation kooperativ vorgehen. Ein Muster, nach dem solche eng verzahnten Änderungen quasi kollektiv ausgeführt werden können, wurde in Abschnitt 3.6

```
// Abarbeitung auf jedem Replikat von p
collective synchronized(p) {
   // Nebenläufige Änderung eines überlappungsfreien
   // Teilzustandes von p
}
```

Abbildung 6.6: Markierung eines parallelen Hauptschritts durch kollektive Synchronisation.

mit der Übertragung des bulk-synchronen Modells auf replizierte Objekte vorgestellt. Dort schließen sich eine Menge von parallelen Kontrollfäden zu einem Kollektiv zusammen, die in einem parallelen Hauptschritt jeweils ihren Teil ihres lokalen Replikats des replizierten Objektes modifizieren. Zur Berechnung der Modifikation steht jedem dieser Kontrollfäden die konsistente Sicht auf den Zustand des replizierten Objektes zu Beginn des parallelen Hauptschritts zur Verfügung. Die kollektive Modifikation wird abgeschlossen, indem die nebenläufig durchgeführten Änderungen am Ende des Hauptschrittes wieder zu einem konsistenten Zustand verschmolzen und an jedes Replikat verteilt werden. Auf diese Weise startet die parallele Modifikationsoperation mit einer konsistenten Sicht auf das replizierte Objekt. Ausgehend davon führt jeder einzelne Kontrollfaden seine Teiländerung durch. Jede dieser Teiländerungen kann aber erst dann abgeschlossen werden, wenn alle parallel durchgeführten Teiländerungen vorliegen und zu einer konsistenten Gesamtänderung verschmolzen sind. Durch diese enge Verzahnung wird trotz Nebenläufigkeit nie ein inkonsistenter Zwischenzustand nach außen sichtbar.

Zur Synchronisation eines solchen datenparallelen BSP-Hauptschrittes wird durch die vorliegende Arbeit eine spezielle Form der Synchronisation zur Verfügung gestellt. Diese neue Form der Synchronisation heißt „kollektiv“, da für ihr Zustandekommen mehrere Kontrollfäden benötigt werden. Für die kollektive Synchronisation wird die Markierung `collective` verwendet, welche syntaktisch wie die Markierungen `shared` und `exclusive` aus Abschnitt 6.1.3 eingesetzt wird. Ein Beispiel ist in Abbildung 6.6 zu sehen. Eine kollektive Synchronisation kann genau dann betreten werden, wenn sie an jedem Replikat eines replizierten Objektes von einem Kontrollfaden angestoßen wird. Während der Ausführung der kollektiven Synchronisation gehen diese Kontrollfäden eine Kooperation ein, um eine parallele Zustandsfortschreibung des synchronisierten replizierten Objektes zu berechnen. Jeder Kontrollfaden darf „sein“ Replikat während der kollektiven Synchronisation modifizieren.[4] Allerdings müssen die nebenläufigen Änderungen an allen Replikaten kooperativ durchgeführt werden. Dabei müssen die teilnehmenden Kontrollfäden garantieren, dass ihre Änderungen überlappungsfrei sind. Das heißt, jeder Teilzustand des replizierten Objektes darf nur von genau einem Kontrollfaden modifiziert werden.[5] Diese Einschränkung ist notwendig, damit eine automatische Verschmelzung der Änderungen zu einem neuen

[4] Auf Rechnerbündeln aus SMP-Rechenknoten ist auch die Verwendung von mehreren Kontrollfäden pro Replikat denkbar und sinnvoll. Dies kann realisiert werden, indem sich die Kontrollfäden, die ein Replikat bearbeiten sollen, lokal synchronisieren und nur einer der Kontrollfäden die kollektive Synchronisation durchführt.

[5] Diese Einschränkung wird in Abschnitt 6.2.5 durch die Einführung benutzerdefinierter Verschmelzungsalgorithmen abgemildert.

konsistenten Gesamtzustand durchgeführt werden kann. Die kollektive Synchronisation kann erst dann wieder verlassen werden, wenn alle kooperierenden Kontrollfäden das Ende ihres kollektiv synchronisierten Bereichs erreicht haben. Erst dann stehen alle Änderungen zur Verfügung, die mittels einer kollektiven Zustandsfortschreibung in den neuen konsistenten Zustand eingebracht werden. Nachdem ein Kontrollfaden einen kollektiv synchronisierten Bereich verlassen hat, wird durch das Laufzeitsystem garantiert, dass sein lokales Replikat wieder eine konsistente Sicht auf das replizierte Objekt darstellt. Alle nebenläufigen Änderungen, die von anderen Kontrollfäden durchgeführt wurden und die Teilzustände seines lokalen Replikats betreffen sind in diese Sicht eingegangen. Entweder kann auf einen kollektiven Schritt ein weiterer folgen, oder das betreffende replizierte Objekt kann mit exklusiver und gemeinsamer Synchronisation weiterbearbeitet werden.

Von ihren Eigenschaften her steht die kollektive Synchronisation zwischen exklusiver und gemeinsamer Synchronisation. Wie in der exklusiven Synchronisation dürfen Modifikationen durchgeführt werden, sie müssen aber kooperativ mit den anderen nebenläufigen Modifikationen ablaufen. Wie bei der gemeinsamen Synchronisation wird die kollektive Synchronisation in mehreren Kontrollfäden parallel durchgeführt. Anders als bei der gemeinsamen Synchronisation ist dies aber nicht optional, sondern verpflichtend. Um erfolgreich zu sein, muss eine kollektive Synchronisation auf allen Replikaten eines replizierten Objektes parallel durchgeführt werden. Anders als bei der exklusiven und gemeinsamen Synchronisation sind während einer kollektiven Synchronisation keine Benachrichtigungsoperationen zwischen Kontrollfäden über `wait()` und `notify()` erlaubt.[6] Da sich die an einer kollektiven Synchronisation beteiligten Kontrollfäden bereits in einer wechselseitigen Abhängigkeitsbeziehung befinden, wäre die Gefahr einer Verklemmung ansonsten zu groß.

6.1.6 Partielle Replikation

Die im letzten Abschnitt beschriebene kollektive Synchronisation eignet sich besonders, um sehr große Datenstrukturen parallel zu verarbeiten. Dabei arbeitet jeder Kontrollfaden an einem Teil der Struktur. Benötigt er dazu die komplette Datenstruktur in seinem Hauptspeicher, dann ist die Gesamtgröße der verarbeitbaren Daten durch die Hauptspeichergröße eines einzelnen Rechners beschränkt. Ebenso schlimm ist in diesem Fall der Aufwand für die kollektive Zustandsfortschreibung, da der von jedem Kontrollfaden modifizierte Teil an alle anderen Replikate verteilt werden muss. Wie groß der Teil der Gesamtdatenstruktur ist, der benötigt wird, um eine Berechnung auf einem Teil der Struktur auszuführen, hängt vom Algorithmus der Anwendung ab. Viele Algorithmen benötigen zur Berechnung auf einer Teilstruktur nicht alle Daten als Eingabe. Der in Abschnitt 3.6.2 als Beispiel verwendetet Finite-Elemente-Algorithmus benötigt für die Berechnung des Zustandes zum nächsten Zeitpunkt nur den Teil des Drahtmodells, auf dem die Berechnung durchgeführt werden soll, zuzüglich der Knoten in der direkten Nachbarschaft. Partielle Replikation hat das Ziel, einem Kontrollfaden in seinem lokalen Replikat nur genau die Teilzustände des replizierten Objektes zur Verfügung zu stellen, die er für seinen Teil der Berechnung benötigt. Weist ein

[6] Der Verstoß gegen diese Regel führt zu einer Ausnahmebedingung während der Laufzeit.

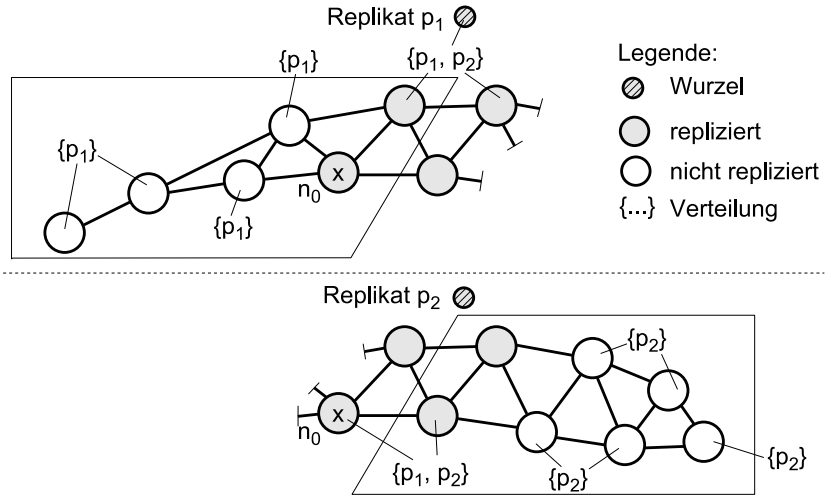


Abbildung 6.7: Replikate eines partiell replizierten Objektes.

Algorithmus günstige Lokalitätseigenschaften auf, muss ein lokales Replikat nur wenig mehr Teilzustände enthalten als von dem zugehörigen Kontrollfaden bearbeitet werden. Dies sind die Teilzustände, für deren Berechnung der entsprechende Kontrollfaden zuständig ist, zuzüglich der Teilzustände in der unmittelbaren Umgebung, welche in die Berechnung miteinfließen. Um partielle Replikation zu erreichen, wird für die Teilzustände eines replizierten Objektes explizit spezifiziert, in welchen lokalen Replikaten sie enthalten sein müssen. Damit wird auch die Bearbeitung einer großen Datenstruktur möglich, die nicht als Ganzes in den Hauptspeicher eines einzelnen Rechenknotens passt. Außerdem wird so der Aufwand für eine kollektive Zustandsfortschreibung erheblich reduziert.

Wie in Abschnitt 6.1.1 ausgeführt, besteht ein Replikat aus einem Wurzelobjekt, dessen Klasse die Markierung `replicated` trägt, und einer Menge von lokalen Objekten, den Teilzuständen des replizierten Objektes. Bei partieller Replikation spiegelt ein einzelnes lokales Replikat keine komplette Kopie des gedachten Gesamtzustandes wider, sondern es wird nur der Teil des Gesamtzustandes lokal vorgehalten, der für die Berechnung auf einem Rechenknoten auch tatsächlich benötigt wird. Abbildung 6.7 nimmt das Beispiel aus Abschnitt 3.6.2 wieder auf. Zu sehen sind zwei partielle lokale Replikate eines Drahtmodells. Das komplette Drahtmodell ist auf keinem der Rechenknoten als Ganzes verfügbar. In seiner Gesamtheit existiert es lediglich in der Vorstellung des Programmierers, der den datenparallelen Algorithmus so formulieren kann, als wäre auf jedem an der Berechnung beteiligten Rechenknoten eine komplette Kopie dieses Zustandes abgelegt. Tatsächlich umfasst aber jede dieser Kopien nur genau diejenigen Teilzustände, die auf dem jeweiligen Rechenknoten für die Durchführung der Berechnung benötigt werden. Die Menge der lokal verfügbaren Knoten des Drahtmodells besteht aus den Knoten, deren Zustand auf dem jeweiligen Rechenknoten neu berechnet werden soll (eingerahmte Knoten) und denjenigen Knoten des Drahtmodells, deren Zustand mit in die Berechnung einfließt. Da in eine Berechnung an einem Knoten des Drahtmodells nur die Zustände der jeweiligen Nachbarknoten einfließen und der Schnitt für die Aufteilung der Berechnung geschickt gewählt ist, muss nur etwas mehr als die Hälfte der Teilzustände auf jedem Rechenknoten vor-

gehalten werden. Aus Sicht des Laufzeitsystems sind daher die meisten Teilzustände nicht im eigentlichen Sinne repliziert, da sie nur in einer einzigen Kopie vorliegen. Nur von den grau eingefärbten Knoten existieren jeweils zwei Kopien – nur diese sind also echt repliziert. Aus Sicht des Programmierers sind dagegen alle lokalen Replikate (partielle) Kopien des gedachten Gesamtzustandes. Für den Programmierer spielt es keine Rolle, ob von einem speziellen Teilzustand noch weitere Kopien auf anderen Rechenknoten existieren oder nicht. Der Algorithmus behandelt alle Teilzustände des jeweils lokalen Replikats gleich – so wie wenn das lokale Replikat eine vollständige Kopie des Gesamtzustandes wäre.

Partielle Replikation wird erreicht, indem man für die Teilzustände eines replizierten Objektes eine Verteilung spezifiziert. Diese Verteilung gibt an, bei welchen Replikaten der jeweilige Teilzustand benötigt wird. In Abbildung 6.7 ist die spezifizierte Verteilung als Menge von Replikaten an die Teilzustände annotiert. Die Annotation $\{p_1, p_2\}$ bedeutet dabei, dass der entsprechende Teilzustand bei Replikat p_1 und p_2 benötigt wird und damit repliziert wird. Mit der Annotation $\{p_1\}$ hingegen wird der Teilzustand nur bei Replikat p_1 benötigt. Er wird daher nicht repliziert, sondern ist ausschließlich Teil eines einzigen Replikats. Wie man an dem mit x markierten Knoten erkennt, benötigt partielle Replikation eine erweiterte Definition von Konsistenz. Für das Modell in Abbildung 6.7 gilt Konsistenz, obwohl sich der Zustand des Knotens x offensichtlich von Replikat p_1 zu p_2 unterscheidet. Während x in Replikat p_1 Referenzen auf vier Nachbarknoten hat, referenziert seine Kopie in Replikat p_2 nur zwei Nachbarknoten. Die beiden übrigen Referenzen sind wegen der partiellen Replikation gekappt, da die beiden anderen Nachbarknoten eine Verteilungsannotation tragen, die ihre Einbeziehung in Replikat p_2 verhindert. Gekappte Referenzen werden durch den Wert `null` repräsentiert. Demzufolge ist die Referenz $x.n_0$ gleich y in Replikat p_1, während $x.n_0$ gleich `null` in p_2 ist.

Um partieller Replikation Rechnung zu tragen, wird die Definition konsistenter Referenzen auf lokale Objekte aus Abschnitt 6.1.2 folgendermaßen angepasst: Zwei lokale Referenzen $p_i.s$ und $p_j.s$ sind genau dann konsistent, wenn sie entweder beide den Wert `null` annehmen oder auf Objekte s_i bzw. s_j verweisen, die ebenfalls in einer Replikationsbeziehung stehen oder $p_i.s$ auf ein Objekt s verweist und $p_j.s$ gleich `null` ist und p_j nicht in der Verteilungsannotation von s enthalten ist.

Die Verwendung des Wertes `null`, um gekappte Referenzen darzustellen, ist zweischneidig. Eine korrekte Anwendung wird nie versuchen, auf eine gekappte Referenz zuzugreifen, da das Objekt, das hinter dieser Referenz stand, für die Berechnung per Definition nicht benötigt wird. Eine fehlerhafte Anwendung könnte allerdings trotzdem versuchen, eine solche Referenz zu dereferenzieren. In diesem Fall muss ein Fehler signalisiert werden. Dies ist bei `null` zwar der Fall, aber der dabei erzeugte Fehler lässt sich nicht von einem anderen illegalen Zugriff auf eine normale uninitialisierte Referenz unterscheiden. Außerdem wäre es wünschenswert nicht nur bei der Dereferenzierung, sondern bei *jeder* Benutzung einer referenzwertigen Instanzvariablen, die eine gekappte Referenz enthält, einen Fehler zu erzeugen. Dies ist aber bei Verwendung des Wertes `null` nicht möglich. Beispiele sind der Referenzvergleich und das Zuweisen der Referenz an eine andere referenzwertige Variable. Bei Verwendung des Wertes `null` wird in diesen Fällen kein Fehler ausgelöst. Dies erschwert es, den zugrundeliegenden Programmfehler zu finden, da der Effekt erst spät eintritt. Leider ist

```
void distribute(
   ReplicatedObject p,
   Object x, int[] distribution);
```

Abbildung 6.8: Bibliotheksmethode zur Spezifikation einer Verteilung.

das ideale Verhalten im Fehlerfall nicht oder nur durch eine globale Programmtransformation aller referenzwertigen Variablen und der Zugriffe darauf zu erreichen. Dies würde zu einem prohibitiven Laufzeitaufwand führen.

Eine Verteilungsannotation wird über den Aufruf der Bibliotheksmethode `distribute()` spezifiziert. Ihre Signatur ist in Abbildung 6.8 gezeigt. Die Methode `distribute()` erwartet neben einer Referenz auf das replizierte Wurzelobjekt das Objekt, dessen Verteilung spezifiziert werden soll, und die gewünschten Verteilung, angegeben als Menge von Replikatnummern, als Argumente. Wurden von einem replizierten Objekt p zwei Replikate p_1 und p_2 angelegt, so spezifiziert der Aufruf `distribute(p, x, new int[]{0})` für das Objekt x die Verteilung $\{p_1\}$. Es ist gleichbedeutend, an welchem Replikat des replizierten Objektes (p_1 oder p_2) der Aufruf von `distribute()` stattfindet. Genausowenig spielt es eine Rolle, ob das Replikat, an dem der Aufruf stattfindet, in der spezifizierten Verteilung enthalten ist oder nicht. Findet in obigem Beispiel der Aufruf an Replikat p_2 statt, die spezifizierte Verteilung ist aber $\{p_1\}$, so wird das Objekt in der darauffolgenden Zustandsfortschreibung zum angegebenen Replikat transportiert und im Replikat p_2 gelöscht.

Die Verteilungsannotation wurde dynamisch über einen Methodenaufruf (anstatt statisch über eine Deklaration) realisiert, um auch eine nachträgliche Umverteilung mit denselben Mitteln spezifizieren zu können. Darüberhinaus ist es für das Laufzeitsystem während der Erzeugung eines lokalen Objektes nicht wichtig, bei welchen Replikaten später Kopien des neuen Objekt erzeugt werden. Ein lokales Objekt wird immer lokal erzeugt und erst während einer Zustandsfortschreibung Teil eines replizierten Objektes.

Wird für ein Objekt keine Verteilung spezifiziert, ist das gleichbedeutend mit der Spezifikation voller Replikation für dieses Objekt. Allerdings bedeutet die Tatsache, dass die spezifizierte Verteilung für einen Teilzustand die Einbeziehung in alle Replikate *ermöglicht*, nicht automatisch, dass dieser Teilzustand zur Laufzeit auch *tatsächlich* Teil jedes Replikats wird. Ein Objekt wird nur dann Teil eines Replikats, wenn es dort auch tatsächlich referenziert wird. Diese Replikation bei Bedarf erlaubt, die Spezifikation von Verteilungsannotationen auf eine kleine Anzahl von Objekten zu reduzieren. Dass dies auch absolut notwendig ist, um nicht das objektorientierte Geheimnisprinzip zu verletzen, kann man sich folgendermaßen klarmachen: Angenommen, ein Objekt a enthält als privaten Unterzustand Referenzen auf Objekte x und y. Wenn die Spezifikation einer Verteilung für das Objekt a die identische Spezifikation für alle seine Unterzustände x und y (und deren mögliche Unterzustände) erforderte, dann wäre für die Verteilung eines replizierten Objektes ein direkter Zugriff auf den gesamten gekapselten Zustand notwendig. Dieses Vorgehen stünde im Gegensatz zum objektorientierten Geheimnisprinzip. Die Replikation bei Bedarf ermöglicht es, eine Verteilung nur für das Objekt a zu spezifizieren und die Verteilung für x und y offen zu lassen. Damit können diese Objekte prinzipiell Teil jedes Replikats werden. Da es sich bei ihnen aber

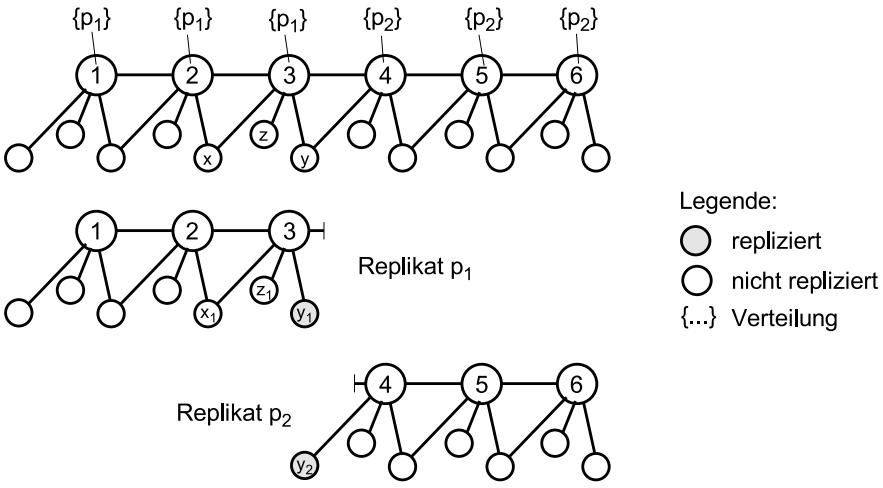


Abbildung 6.9: Adaptive Verteilung von Teilzuständen eines replizierten Objektes.

um private Unterzustände von a handelt, werden sie nur aus a heraus referenziert und passen sich damit automatisch der Verteilung von a an.

Replikation bei Bedarf ist aber nicht auf private Unterzustände beschränkt, die nur von genau einem Objekt referenziert werden. Ein Beispiel ist in Abbildung 6.9 zu sehen. Nur für die nummerierten Objekte $1-6$ ist explizit eine Verteilung spezifiziert. Jedes dieser Objekte besitzt drei Referenzen, eine auf ein gemeinsames Objekt mit seinem linken Nachbarn, eine auf ein privates Objekt und eine auf ein gemeinsames Objekt mit seinem rechten Nachbarn. Die Verteilungsannotation ist so gewählt, dass die Objekte $1-3$ nur in Replikat p_1 enthalten sind und die Objekte $4-6$ nur in Replikat p_2. Für die als kleine Kreise eingezeichneten Objekte ist keine Verteilungsannotation gegeben. Dennoch stellt sich genau die dargestellte Verteilung ein, bei der lediglich Objekt y repliziert ist. Seine Kopie y_1 ist Teil von p_1, weil es aus Objekt 3 referenziert wird, das aufgrund seiner Verteilungsannotation Teil von p_1 ist. Seine Kopie y_2 ist in p_2 enthalten, weil sie von dem dorthin verteilten Objekt 4 referenziert wird. Alle anderen Objekte sind in nur genau einem Replikat enthalten. Insbesondere gilt das für die Objekte x und z. Objekt z wird ausschließlich von Objekt 3 referenziert und passt sich daher dessen Verteilung an. Objekt x wird sowohl von Objekt 2 als auch von Objekt 3 referenziert. Da aber sowohl 2 als auch 3 nur Teil von Replikat p_1 sind, wird auch x nur aus p_1 referenziert und somit ausschließlich Teil von p_1.

Die Bedarfsreplikation arbeitet mit dem verteilten Speicherbereiniger zusammen. Sobald die letzte Referenz auf einen replizierten Teilzustand in einem Replikat gelöscht wird, wird dieser Teilzustand zuerst vom lokalen Speicherbereiniger der virtuellen Maschine abgeräumt. Während der nächsten Zustandsfortschreibung des replizierten Objektes wird daraufhin die Verteilung für diesen Teilzustand entsprechend angepasst. Wurde die letzte Kopie eines replizierten Teilzustandes gelöscht und somit die leere Verteilung erreicht, wird dieser Teilzustand endgültig aus den Verwaltungsstrukturen des replizierten Objektes entfernt. Selbstverständlich kann sich der Bedarf während der Laufzeit des Programms ändern, wodurch sich die Replikation von Teilzuständen entsprechend anpasst. Würde in Abbildung 6.9 ein neues Objekt 7 eingefügt, das eine Referenz auf Objekt x hält und Teil beider Replikate p_1 und p_2 wird, so würde

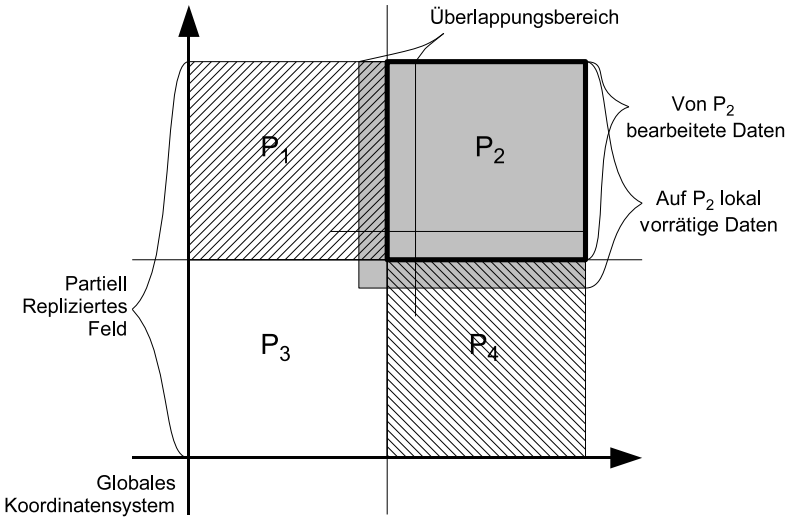


Abbildung 6.10: Globale Sicht auf ein repliziertes Feld.

dadurch die Verteilung von Objekt x auch entsprechend angepasst werden.

Die Anwendung kann die Spezifikation der Verteilung der Teilzustände eines replizierten Objektes auf wenige Objekte beschränken. Zweckmäßigerweise wird es sich dabei um genau die Objekte handeln, auf denen der parallele Algorithmus operiert und deren Bearbeitung unter den beteiligten Kontrollfäden aufgeteilt werden soll. Die Verteilung aller anderen Objekte wird danach über Replikation bei Bedarf an die Verteilung der explizit verteilten Objekte angepasst. Im Beispiel der Finite-Elemente-Berechnung auf einem Drahtmodell wird es sich bei den explizit verteilten Objekten um die Elementobjekte handeln, während die Verteilung der Objekte, die Materialien oder Nebenbedingungen repräsentieren, der Automatik überlassen werden kann. Bei der Parallelisierung wird die Berechnung an den Elementobjekten unter den parallelen Kontrollfäden aufgeteilt. Dazu muss der Programmierer ohnehin die Verteilung der Elementobjekte bestimmen. Alle übrigen Objekte müssen überall da verfügbar sein, wo eine Referenz auf sie existiert.

6.1.7 Felder als Teil des replizierten Zustandes

Dieselbe Abstraktion, die eine lokale Sicht auf ein partiell repliziertes Objekt für die Durchführung einer kollektiven Operation bietet, ist auch auf reguläre Strukturen anwendbar. Abbildung 6.10 zeigt ein zweidimensionales Feld, das auf vier Rechenknoten parallel verarbeitet werden soll. Das Feld ist dazu logisch so unterteilt, dass auf jedem Rechenknoten P_1 bis P_4 der Zustand eines rechteckigen Teilbereiches fortgeschrieben werden soll. Jeder der Rechenknoten berechnet dabei die in seiner Farbe eingefärbten Feldelemente. Unter der Annahme, dass in das Ergebnis der Berechnung eines Feldelementes Werte in seiner Nachbarschaft eingehen, benötigt jeder Rechenknoten Zugriff auf einen etwas größeren Feldbereich als ihm zur Berechnung zugeordnet ist. Dieser Bereich ist für Rechenknoten P_2 grau hinterlegt eingezeichnet. Entsprechende Feldbereiche müssen auch auf den anderen Rechenknoten lokal vorrätig gehalten werden.

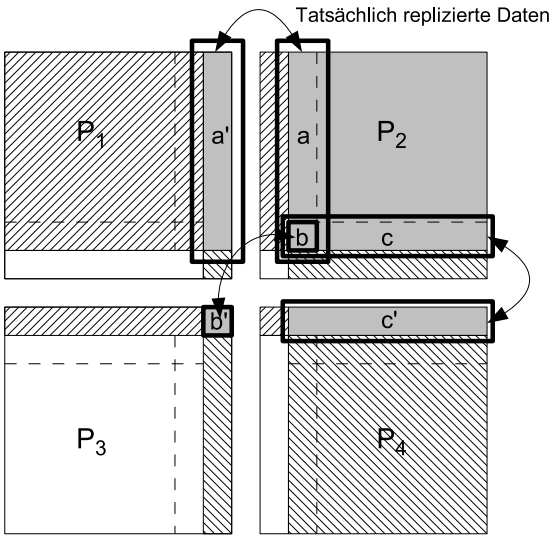


Abbildung 6.11: Sicht auf die prozessorlokalen Teilfelder mit Replikationsbeziehungen.

Abbildung 6.11 zeigt die aus den Überschneidungsbereichen hervorgehenden knotenlokalen Teilfelder. Der auf einem Knoten lokal vorrätig gehaltene Feldbereich setzt sich dabei aus den Feldelementen zusammen, die der lokale Rechenknoten berechnet zuzüglich der Feldelemente, die für die lokale Berechnung lesend verwendet werden (aber von anderen Rechenknoten berechnet werden). Die Färbung der Feldelemente zeigt an, welcher Prozessor das jeweilige Feldelement berechnet. Man erkennt, dass sich die von Rechenknoten P_2 berechneten (grau hinterlegten) Feldelemente in vier Klassen einteilen lassen: Die Feldelemente im Bereich a ohne Bereich b werden auf den Rechenknoten P_2 und P_1 lokal vorrätig gehalten. Entsprechend sind die Feldelemente im Bereich c ohne Bereich b auf den Rechenknoten P_2 und P_4 gespeichert. Die Feldelemente in Bereich b sind auf allen vier Rechenknoten abgelegt und alle übrigen grau hinterlegten Feldelemente sind private Feldelemente von Rechenknoten P_2.

Wie der Objektgraph aus Abbildung 6.7 lässt sich das gesamte Feld somit als partiell repliziert betrachten. In Abbildung 6.12 sind die Feldelemente nach ihrem Replikationsgrad zusammengefasst. Ein Großteil der Feldelemente des Gesamtfeldes ist dabei nur auf einem einzelnen Rechenknoten angelegt und somit nicht „tatsächlich“ repliziert. Lediglich die Feldelemente in den Randbereichen sind entweder zweifach oder vierfach repliziert. Dennoch ist die Betrachtung des Feldes als partiell repliziertes Gesamtfeld gerechtfertigt und sinnvoll, weil die Formulierung des Berechnungsalgorithmus von seiner physikalischen Verteilung auf mehrere Rechenknoten abstrahieren kann. Der Zugriff auf das Feld erfolg auf jedem Rechenknoten uniform in globalen Feldkoordinaten und der Datenaustausch zwischen einzelnen Berechnungsphasen einer kollektiven Operation geschieht automatisch. Um das Feld parallel auf mehreren Rechenknoten verarbeiten zu können, muss der Algorithmus lediglich logisch so partitioniert werden, dass die Berechnungen auf den einzelnen Feldpartitionen unabhängig voneinander parallel durchgeführt werden können. Der Berechnungsalgorithmus ist jedoch auf jeder einzelnen Feldpartition identisch.

Die in der vorliegenden Arbeit entworfene Programmierumgebung realisiert parti-

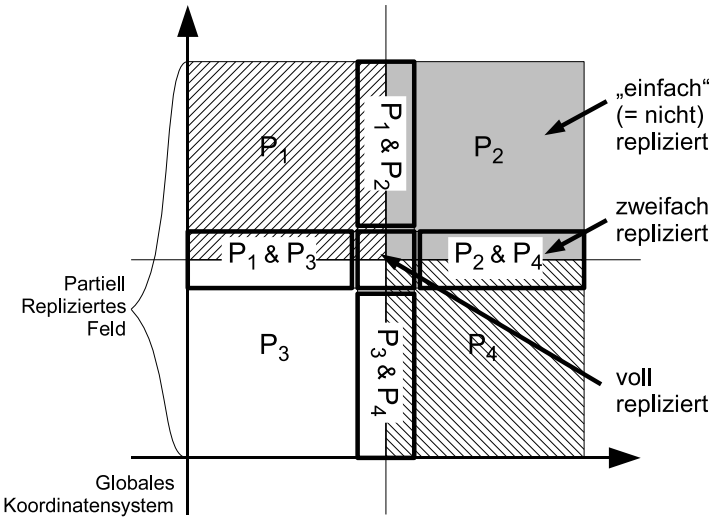


Abbildung 6.12: Replikationsgrade des partiell replizierten Feldes aus Abbildung 6.10.

ell replizierte Felder als Bibliothek. Ein repliziertes Feld ist dabei kein eigenständiges repliziertes Objekt, sondern ist immer als Teil des Zustandes einer Instanz einer replizierten Klasse (vgl. Abschnitt 6.1). Die Synchronisation und damit die Steuerung der Zustandsfortschreibung geschieht immer am umschließenden replizierten Objekt. Abbildung 6.13 zeigt ein Programmierbeispiel. In diesem Programmierbeispiel wird ein kollektiv repliziertes Feld `matrix` in einer exklusiven Synchronisation an einem umschließenden replizierten Objekt erzeugt und eine Referenz darauf (beispielsweise durch Speicherung in einer Instanzvariable des replizierten Objektes) allen beteiligten Kontrollfäden bekanntgemacht. In einer anschließenden kollektiven Synchronisation schließen die berechnenden Kontrollfäden die Initialisierung des replizierten Feldes ab und erhalten eine lokale Sicht auf das Feld. Diese Sicht benutzen die Kontrollfäden in aufeinanderfolgenden kollektiv synchronisierten Abschnitten, um eine parallele Berechnung auf dem partiell replizierten Feld durchzuführen. Während dieser Berechnung greifen die Kontrollfäden lesend auf ihren und einen umliegenden Feldbereich zu und schreiben den Zustand ihrer eigenen Feldpartition fort. Am Ende jeder kollektiven Synchronisation sorgt die Bibliotek für partiell replizierte Felder in Kooperation mit der Laufzeitumgebung automatisch für den Datenaustausch, der Konsistenz unter allen lokalen Sichten auf das Gesamtfeld herstellt.

6.2 Erweiterungen in Objektserialisierung und Kommunikationsbibliothek

Die Einführung von replizierten Objekten erfordert eine Erweiterung der Kommunikationsbibliothek aus Kapitel 5. Zwar verhalten sich replizierte Objekte aus Anwendungssicht wie reguläre lokale Objekte, da immer nur das lokale Replikat angesprochen wird, dennoch müssen Replikate intern zum Zweck der Zustandsfortschreibung entfernt ansprechbar sein. Daraus ergibt sich, dass ein repliziertes Objekt ebenso wie

```
// Erzeugung des kollektiven Feldes im Zustand eines
// gegebenen replizierten Objektes:
exclusive synchronized (replicatedObject) {
   // Gegeben ist die Anzahl von Spalten, Zeilen,
   // Anzahl von Blöcken in x- und y-Richtung und
   // die Breite des Überlappungsbereiches dieser
   // Blöcke.
   matrix = new CollectiveArray2D(
      colCnt, rowCnt, 2, 2, overlap);
}

// Erzeugen einer lokalen Sicht in jedem
// berechnenden Kontrollfaden:
collective synchronized (replicatedObject) {
   // Die lokale Sicht erlaubt den lokalen Zugriff
   // auf das partiell replizierte Feld.
   view = matrix.init(replicatedObject);
}

// In jedem berechnenden Kontrollfaden:
while (computation-not-finished) {
   collective synchronized (replicatedObject) {
      // Für jedes Feldelement (x0, x1), das im
      // lokalen Kontrollfaden berechnet wird:
      {
         // Berechnung des Feldelements (x0, y0):
         newValue = computeNewValue(
            view.get(x0, x1),
            view.get(x0 + 1, x1),
            view.get(x0, x1 - 1), ...);

         view.set(x0, x1, newValue);
      }
   }
}

// Ergebnis der Berechnung liegt im partiell
// replizierten Feld als Teil des Zustands eines
// replizierten Objektes vor.
```

Abbildung 6.13: Programmierbeispiel für die Verwendung eines partiell replizierten Feldes als Teil des Zustands eines replizierten Objektes.

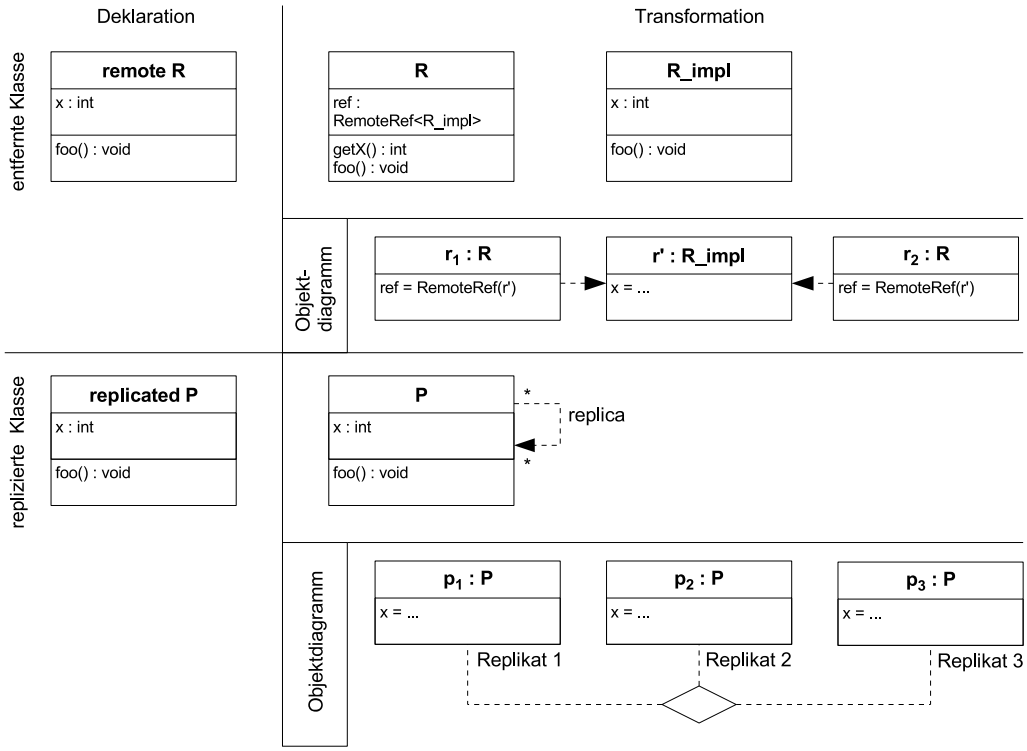


Abbildung 6.14: Klassen- und Objektstruktur eines replizierten Objektes im Vergleich zu einem entfernten Objekt.

ein entferntes eine von der Kommunikationsbibliothek zu verwaltende Identität in der verteilten Umgebung besitzen muss. Anders als bei einem entfernten Objekt, bei dem es mehrere Stellvertreter (die entfernten Referenzen), aber nur ein zugehöriges Implementierungsobjekt gibt, ist bei einem replizierten Objekt das lokale Replikat gleichzeitig Referenz und Implementierungsobjekt. Es hält dazu intern Verweise auf alle anderen Replikate des replizierten Objektes. In Abbildung 6.14 ist diese Situation mit Hilfe eines Klassen- und Objektdiagramms veranschaulicht. Die entfernte Klasse `remote R` zerfällt durch die Transformation in die Stellvertreterklasse `R` und die Implementierungsklasse `R_Impl`. Zur Laufzeit existieren zu einem entfernten Objekt genau ein Implementierungsobjekt r', aber möglicherweise viele Stellvertreterobjekte r_1 und r_2. Die Stellvertreterobjekte r_1 und r_2 referenzieren das einzelne Implementierungsobjekt r' entfernt. Dagegen bleibt die replizierte Klasse `replicated P` durch die Transformation im wesentlichen unverändert.[7] Zur Laufzeit existieren für ein repliziertes Objekt allerdings mehrere Replikate. In diesem Beispiel ist der Replikationsgrad drei, das heißt, dass das replizierte Objekt aus den drei Replikaten p_1, p_2 und p_3 besteht. Diese Replikate sind alle Instanzen derselben Klasse `P` und über die Beziehung `replica` verbunden. Diese Beziehung wird ebenfalls über entfernte Referenzen realisiert, so dass jeweils ein Replikat die beiden anderen, welche sich auf anderen Rechenknoten befinden, ansprechen kann.

Auch die schnelle Objektserialisierung aus Kapitel 4 bedarf einer Erweiterung. Für die Übertragung von Argumenten in entfernten Methodenaufrufen ist es ausreichend,

[7] Details der Transformation replizierter Klassen finden sich in Abschnitt 6.3.

immer komplette Objekte zu übertragen, die auf der Empfangsseite als neue Kopien entstehen. Um aber eine Zustandsfortschreibung für ein repliziertes Objekt zu übermitteln, darf nur ein Teil des Zustands eines Objektes (der geänderte Teil eines Replikats, vgl. Abschnitt 6.2.3) übertragen werden, und dieser Teilzustand muss auf Empfangsseite in ein bestehendes Objekt (ein veraltetes Replikat) *eingespielt* werden.

Die folgenden Abschnitte erklären die notwendige Bibliotheksunterstützung, auf der die in Abschnitt 6.3 besprochene Programmtransformation für transparent replizierte Klassen aufbaut. Abschnitt 6.2.1 beschreibt die Verwaltung replizierter Objekte durch die Kommunikationsbibliothek und das Zusammenspiel mit entfernten Objekten, insbesondere bezüglich der Argumentübergabe in Fernaufrufen. Abschnitt 6.2.2 erweitert die entfernte Sperranforderung für entfernte Objekte aus Abschnitt 5.4 zum Sperren von replizierten Objekten. In Abschnitt 6.2.3 wird die Technik erklärt, die es erlaubt, die an einem Replikat durchgeführten Änderungen zu erkennen, sie in eine Änderungsmenge zu extrahieren und dieselben Änderungen an einem anderen Replikat desselben replizierten Objektes nachzuvollziehen. Abschnitt 6.2.4 kombiniert diese Basistechniken zur exklusiven bzw. kollektiven Zustandsfortschreibung für replizierte Objekte.

6.2.1 Replikatverwaltung

Die Replikate replizierter Objekte unterscheiden sich von entfernten Objekten darin, dass sie keine separaten Stellvertreter besitzen und in der Art und Weise, wie sie in entfernten Methodenaufrufen als Referenz übergeben werden. Ein Replikat wird bei seiner Erzeugung exportiert, so dass es wie ein entferntes Objekt Fernzugriffe entgegennehmen kann. Anders als ein entferntes Objekt unterstützt ein Replikat keine benutzerdefinierten fernaufrufbaren Methoden. Fernzugriffe auf Replikate werden ausschließlich intern durch das System über ein Reihe fest vordefinierter Methoden für Sperranforderungen und Zustandsfortschreibungen durchgeführt. Dabei dient das Replikat selbst als Stellvertreterobjekt für alle anderen Replikate des zugehörigen replizierten Objektes.

Bei der Erzeugung eines neuen replizierten Objektes wird sein Replikationsgrad festgelegt. Die Erzeugung wird angestoßen, indem die Anwendung eine neue Instanz einer Unterklasse von `ReplicatedObject` anlegt. Damit wird das lokale Replikat des neuen replizierten Objektes erzeugt. Der Konstruktor der Oberklasse führt wie im Fall der Erzeugung eines entfernten Objektes den Objektexport durch. Zusätzlich erzeugt der Konstruktor von `ReplicatedObject` die anderen Replikate des replizierten Objektes und speichert entfernte Referenzen auf diese Replikate für darauffolgende Zustandsfortschreibungen. Die im vorangegangenen Abschnitt besprochene partielle Replikation ist nur für Teilzustände des replizierten Objektes, aber nicht für das replizierte Wurzelobjekt selbst möglich. Nach der Erzeugung eines neuen replizierten Objektes erhält demnach die Anwendung eine einzige Referenz auf ein lokales Replikat. Jedes Replikat kann alle anderen Replikate desselben replizierten Objektes über entfernte Referenzen ansprechen, allerdings wird für die Operationen zur Synchronisation und Zustandsfortschreibungen immer nur von einem Teil dieser Referenzen tatsächlich Gebrauch gemacht.

Replizierte Objekte können genauso wie entfernte Objekte als Referenz in ent-

fernten Methodenaufrufen übergeben werden. Über diesen Mechanismus bekommt die Anwendung, ausgehend von einem Replikat, Zugriff auf die Replikate desselben replizierten Objektes auf anderen Rechenknoten. Zur Erklärung soll angenommen werden, dass die Replikate des replizierten Objektes aus Abbildung 6.14 so auf die Rechenknoten verteilt sind, dass Replikat p_1 auf Knoten 1 liegt, Replikat p_2 auf Knoten 2 und so weiter. Wenn in einem Fernaufruf, der von Knoten 1 ausgeht und an einem entfernten Objekt auf Knoten 2 abgearbeitet wird, eine Referenz auf Replikat p_1 als Argument übergeben wird, soll diese Referenz während der Abarbeitung des Fernaufrufs transparent für die Anwendung durch eine Referenz auf Replikat p_2 ersetzt werden. Durch diese Austauschoperation erhält die Anwendung die Illusion der Referenzübergabe von replizierten Objekten in entfernten Methodenaufrufen. Der Austausch des Replikats auf dem aufrufenden Knoten gegen das entsprechende Replikat auf dem Zielknoten sorgt für die passende Referenzsemantik, da ein repliziertes Objekt immer über sein lokales Replikat referenziert wird. Ist auf dem Zielknoten kein Replikat des übergebenen replizierten Objektes verfügbar, schlägt der Fernaufruf mit einer Ausnahmebedingung fehl.

Eine ähnliche Technik wurde auch für die Referenzübergabe von entfernten Objekten in Fernaufrufen eingesetzt. Dort funktioniert die Referenzübergabe so, dass auf Seite des Aufrufers ein entferntes Objekt der Klasse `R_Impl` durch ein entsprechendes Stellvertreterobjekt der Klasse `R` ersetzt wird. Dieses Stellvertreterobjekt wird im Fernaufruf als Argument auf den Zielknoten kopiert und verweist dort zurück auf das auf dem Ausgangsknoten zurückgebliebene entfernte Objekt. Da replizierte Objekte im Gegensatz zu entfernten Objekten keine Stellvertreterobjekte besitzen und immer nur über ihre lokalen Replikate referenziert werden, müssen bei der Argumentübergabe in einem Fernaufruf zwei Ersetzungsoperationen – eine auf Sender- und eine auf Empfängerseite – stattfinden. Ein in einem Fernaufruf als Argument übergebenes Replikat p_1 wird auf Senderseite durch einen Identifikator ID_p für das zugehörige replizierte Objekt ersetzt. Dies verhindert, dass das gesamte Replikat p_1 in dem Fernaufruf auf den Zielknoten kopiert wird. Auf Empfängerseite wird der übertragene Identifikator ID_p dazu benutzt, das entsprechende Replikat p_2 des replizierten Objektes ausfindig zu machen. Bevor der Fernaufruf ausgeführt wird, wird der Identifikator ID_p durch das Replikat p_2 auf dem Zielknoten ausgetauscht.

6.2.2 Sperroperationen

In diesem Abschnitt werden die Synchronisationsprimitive für replizierte Objekte auf Java-Synchronisation abgebildet. Dazu müssen für die drei auf replizierte Objekte anwendbaren Synchronisationsformen `shared`, `exclusive` und `collective` Abbildungen auf Java-Synchronisation an den Replikaten des replizierten Objektes gefunden werden. Diese Abbildung muss gewährleisten, dass bei der Verwendung eines replizierten Objektes in mehreren nebenläufigen Kontrollfäden keine inkonsistenten Zustände beobachtet werden können. Dazu muss ein exklusiv synchronisierter Bereich das Betreten von allen weiteren synchronisierten Bereichen am selben replizierten Objekt verhindern. Ein gemeinsam synchronisierter Bereich dagegen muss zwar eine gleichzeitige exklusive oder kollektive Sperranforderung verhindern, soll aber weitere gemeinsame Sperranforderungen zulassen. Eine kollektive Sperre erfordert das neben-

läufige Anstoßen dieser Sperroperation auf allen Replikaten und schließt das gleichzeitige Setzen anderer Sperren aus. Zusätzlich zu diesen Anforderungen, die sich direkt aus der Semantik der drei Synchronisationsformen herleiten, sollen die Sperroperationen möglichst direkt durch Java-eigene Sperren ausdrückbar sein. Da bei der Weiterentwicklung der virtuellen Java-Maschine besonderes Augenmerk auf eine effiziente Handhabung von Sperroperationen gelegt wird, muss andernfalls eine Implementierung von Sperren beispielsweise in einer eigenen Bibliothek als Ersatz der eingebauten Primitive zu einer ineffizienten Lösung führen.

Die einzige Form der in Java zur Verfügung stehenden Sperre ist die exklusive Sperre eines einzelnen Objektes. Die gemeinsame Sperre eines replizierten Objektes wird daher (etwas zu streng) durch Sperren des lokalen Replikats über reguläre Java-Synchronisation realisiert. Damit eine exklusive Synchronisation eines replizierten Objektes gemeinsame Sperren an allen seinen Replikaten verhindert, muss die exklusive Sperre eine Java-Synchronisation an allen seinen Replikaten durchführen. Indem die kollektive Synchronisation schließlich ebenfalls eine Java-Synchronisation am lokalen Replikat durchführt, behindern sich die kollektiven Aktivitäten gegenseitig nicht, schließen aber eine nebenläufig durchgeführte exklusive oder gemeinsame Synchronisation aus.

Da die gemeinsame Synchronisation einzig aus einer regulären Java-Synchronisation am lokalen Replikat besteht, ist ihre Durchführung besonders effizient. Dies entspricht der Anforderung, dass das Lesen (im Gegensatz zum exklusiven Schreiben) von replizierten Objekten besonders effizient sein muss, damit sich der Mehraufwand für die Replikation und die Zustandsfortschreibung beim Schreiben amortisiert. Durch die direkte Abbildung von gemeinsamer Synchronisation auf Java-Synchronisation des lokalen Replikats ergeben sich aber offensichtlich Einschränkungen bezüglich der Nebenläufigkeit mehrerer solcher gemeinsamer Synchronisationen. Da eine Java-Sperre exklusiv ist, können folglich auch nicht mehrere gemeinsame Sperren an *demselben* Replikat gleichzeitig gesetzt werden. Die Nebenläufigkeit ist durch diese Umsetzung auf gleichzeitige gemeinsame Sperren an *unterschiedlichen* Replikaten beschränkt. Unter dem Gesichtspunkt, dass in Java ausschließlich exklusive Sperren einzelner Objekte zur Verfügung stehen, scheint diese Einschränkung tolerabel. Allerdings sollte die Anwendung gemeinsam synchronisierte Bereiche so klein wie möglich wählen, wenn mehrere Aktivitäten nebenläufig auf dasselbe Replikat zugreifen.

Wie oben beschrieben, soll die Anforderung einer gemeinsamen Sperre eine rein lokale Operation sein, die unmittelbar auf das Anfordern einer regulären Java-Sperre abgebildet wird. Daher müssen für die Anforderung einer exklusiven Sperre alle Replikate eines replizierten Objektes „gleichzeitig“ gesperrt werden. Mit anderen Worten, eine exklusive Sperre ist an allen Replikaten repliziert. Damit mehrere solche exklusive Sperranforderungen, welche alle Replikate eines replizierten Objektes sperren, verklemmungsfrei ablaufen, müssen die einzelnen Sperranforderungen an den Replikaten in einer global festgelegten Reihenfolge durchgesetzt werden. Versuchen zwei Kontrollfäden gleichzeitig eine exklusive Sperre an einem replizierten Objekt zu setzen, muss verhindert werden, dass der eine zuerst das Replikat p_1 sperrt und der andere zuerst das Replikat p_2. Ansonsten kann keine der beiden exklusiven Sperren durchgesetzt werden und eine Verklemmung tritt ein. Eine altbekannte Methode, Verklemmungen bei der gleichzeitigen Anforderung mehrerer Betriebsmittel zu verhindern, ist ihre

```
synchronized(r1) {
   synchronized(r2) {
      ...
      synchronized(rn) {
         // Repliziertes
         // Objekt
         // exklusiv
         // gesperrt.
         ...
      }
      ...
   }
}
```

```
ticket = r1.newTicket();
// geordneter Sperr-Rundruf
foreach r (r1,..., rn) {
    r.rmAquire(ticket);
}
// Repliziertes Objekt
// exklusiv gesperrt.
...
foreach r (r1,..., rn) {
    r.rmRelease();
}
```

Abbildung 6.15: Verklemmungsfreie Durchsetzung einer exklusiven Sperre an einem replizierten Objekt. Links: Ordnung über geschachtelte Sperranforderung. Rechts: Ordnung über Ticketvergabe.

hierarchische Ordnung [18]. Dies könnte durch eine verschachtelte Anforderung der Sperren erreicht werden, wie es auf der linken Seite von Abbildung 6.15 schematisch gezeigt ist. Für die (möglicherweise entfernte) Anforderung einer Sperre an einem Replikat `r1,...,rn` ist die schematische Darstellung als synchronisierter Block statt der Transformation aus Abschnitt 5.5.2 verwendet. Allerdings wäre bei diesem Vorgehen keine Überlappung der Einzelsperranforderungen möglich, und die Latenz für eine exklusive Sperre würde linear mit der Anzahl der Replikate wachsen. Um größtmögliche Überlappung zu erreichen, wird daher die global festgelegte Reihenfolge der einzelnen Sperranforderungen durch Verwendung eines geordneten Rundrufs sichergestellt. Dabei werden die Sperranforderungen an alle Replikate per geordnetem Rundruf übermittelt und dort in der durch den Rundruf vorgegebenen Reihenfolge ausgeführt. Die Ordnung der Nachrichten des Rundrufs wird durch einen Sequentialisierer [49] realisiert. Dazu übernimmt eines der Replikate die Funktion des Sequentialisierers, indem es auf Anfrage aufsteigende Ticket-Nummern ausstellt. Auf der rechten Seite von Abbildung 6.15 ist dieses Schema zu sehen. Bevor der Sperr-Rundruf startet, wird über einen Methodenfernaufruf eine neue Ticket-Nummer angefordert und diese dem Sperr-Rundruf mitgegeben. Die Sperranforderungen werden daraufhin an allen Replikaten in der durch die Ticket-Nummer vorgegebene Reihenfolge abgearbeitet. Dies wird gewährleistet, indem eine in `rmAquire()` angeforderte Sperre erst dann gesetzt wird, wenn das mitübergebene Ticket gültig geworden ist, d.h. dass alle Tickets mit kleineren Nummern bereits für vorangegangene Sperranforderungen verbraucht sind. Es gibt zwar noch ausgefeiltere Techniken für geordneten Rundruf [52], aber der Einsatz eines Sequentialisierers erscheint hier ausreichend, da der Rundruf ausschließlich als Sperr-Rundruf verwendet wird. Da exklusive Sperren an replizierten Objekten relativ selten vorkommen, ist die Gefahr somit gering, dass der Sequentialisierer einen Flaschenhals bildet.

Das entfernte Anfordern von regulären Java-Sperren an Replikaten zur exklusiven Synchronisation wird mit den in Abschnitt 5.4 beschriebenen Mitteln der entfernten Sperranforderung realisiert. Hierzu wird die Ähnlichkeit von Replikaten und entfern-

ten Objekten genutzt. Ein Kontrollfaden kann eine Sperre eines Replikats genauso wie die eines entfernten Objektes anfordern. Anders als beim entfernten Sperren eines entfernten Objektes müssen für eine exklusive Sperre eines replizierten Objektes alle seine Replikate gesperrt werden. Über die Technik des geordneten Rundrufs lassen sich solche mehrfachen Sperranforderungen überlappend durchführen, ohne dass man die Gefahr einer Verklemmung eingeht.

Kollektive Synchronisation setzt in den Kontrollfäden, die an der Synchronisation teilnehmen, auf jedem Replikat eine reguläre Java-Sperre. Damit wird die gleichzeitige Durchführung einer gemeinsamen und einer kollektiven Synchronisation effektiv unterbunden, allerdings besteht die Gefahr einer Verklemmung, wenn eine kollektive und eine exklusive Synchronisation konkurrierend durchgeführt werden. Diese Problematik ließe sich zwar durch die Anforderung eines Tickets auch für die kollektive Synchronisation auflösen, darauf wird aber aus Effizienzgründen verzichtet. Die kollektive Synchronisation wird im Kern einer datenparallelen Löserschleife eingesetzt, weswegen eine weitere Kommunikationsoperation zu Beginn der Synchronisation für die Ticket-Anforderung nicht vertretbar ist. Stattdessen muss die Anwendung sicherstellen, dass eine kollektive und eine exklusive Synchronisation nicht konkurrierend durchgeführt werden können. Dies ist in der Regel einfach über eine Barrierenoperation möglich.

6.2.3 Änderungserkennung

Sowohl nach einer exklusiven als auch nach einer kollektiven Synchronisation müssen Änderungen am lokalen Replikat erkannt, extrahiert und an anderen Replikaten eingespielt werden. Dieser Abschnitt beschreibt die Technik, die es ermöglicht, Änderungen an einem Replikat zu erkennen und diese an einem anderen Replikat nachzuvollziehen. Der folgende Abschnitt 6.2.4 verwendet diese Basistechnik, um am Ende einer exklusiven oder kollektiven Synchronisation einen konsistenten Zustand aller Replikate eines replizierten Objektes wiederherzustellen.

Um Änderungen erkennen zu können, hält ein Replikat eine Kopie seines Zustandes parat. Wird ein Replikat modifiziert, so können Änderungen anhand von Abweichungen zwischen Kopie und Original erkannt werden. Vom gesamten replizierten Zustand eines Replikats gemäß Abschnitt 6.1.1 wird eine Kopie nur für solche Teilobjekte vorgehalten, die tatsächlich repliziert sind (vgl. Abschnitt 6.1.6 über partielle Replikation). Änderungen an Objekten, die nur Teil eines einzigen Replikats sind, müssen nicht erkannt werden, da sie für keines der anderen Replikate relevant sind. Eine erkannte Änderung wird in einem Format ähnlich dem der Objektserialisierung codiert, so dass sie auf einem anderen Replikat desselben replizierten Objektes auf einem anderen Knoten nachvollzogen werden kann.

Dadurch dass nur von tatsächlich replizierten Teilzuständen eine Kopie angelegt werden muss, kann sowohl der Speicherplatzverbrauch als auch der Aufwand für die Änderungserkennung auch bei einem großen replizierten Objekt relativ gering gehalten werden. Dazu ist eine gute Partitionierung des replizierten Objektes notwendig, welche die Überlappungsbereiche zwischen seinen einzelnen Replikaten minimiert.

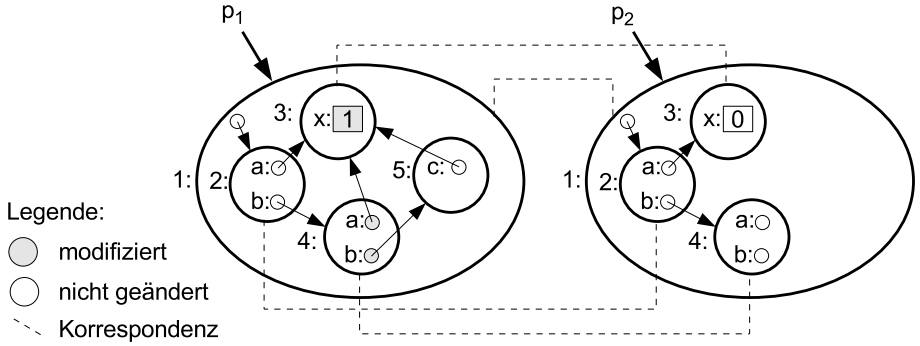


Abbildung 6.16: Repliziertes Objekt, bestehend aus zwei Replikaten, bei kompletter Replikation. Inkonsistenter Zustand zwischen p_1 und p_2.

Korrespondierende Objekte

Wie in Abschnitt 6.1.2 beschrieben, besteht der Zustand eines replizierten Objektes aus vielen Teilobjekten. Eine an einem dieser Objekte detektierte Änderung muss extrahiert und an dem entsprechenden Teilobjekt eines anderen Replikats nachvollzogen werden. Dazu muss für ein Objekt o_i eines Replikats p_i das korrespondierende Objekt o_j in einem anderen Replikat p_j gefunden werden, mit dem o_i in einer Replikationsbeziehung gemäß Abschnitt 6.1.2 steht ($(o_i, o_j) \in \rho_{i,j}$). Da $\rho_{i,j}$ Objekte verschiedener virtueller Maschinen in Beziehung setzt, ist diese Relation nicht direkt implementierbar. Stattdessen implementiert jedes Replikats p_i eine (injektive) Identifikatorfunktion $id_i : \mathbb{O}_i \rightarrow \mathbb{N}$, so dass $id_i(o_i) = id_j(o_j)$ genau dann wenn $(o_i, o_j) \in \rho_{i,j}$. Diese Identifikatorfunktion gibt also einem Objekt eines Replikats eine Adresse, die in allen Replikaten gültig und damit vergleichbar ist.

Abbildung 6.16 zeigt als Beispiel ein repliziertes Objekt, bestehend aus zwei Replikaten p_1 und p_2. Vor jedem Teilobjekt o_i steht seine Identifikationsnummer $id_i(o_i)$ mit Doppelpunkt getrennt. Korrespondierende Objekte, die dieselbe Identifikationsnummer in beiden Replikaten besitzen, sind durch gestrichelte Linien miteinander verbunden. Von einem Replikat ist nur das Wurzelobjekt, wie in Abschnitt 6.2.1 beschrieben, entfernt ansprechbar. Wie leicht zu sehen ist, befindet sich das in Abbildung 6.16 gezeigte replizierte Objekt nicht in einem konsistenten Zustand. Das Replikat p_1 ist lokal geändert, was durch graue Hinterlegung der betreffenden Instanzvariablen kenntlich gemacht ist. So ist die Instanzvariable x von Objekt $id_1^{-1}(3)$ von 0 auf 1 geändert worden und die beiden referenzwertigen Instanzvariablen a und b von $id_1^{-1}(4)$ wurden ebenfalls modifiziert. Die Referenz a wurde auf das schon existierendes Objekt $id_1^{-1}(3)$ gesetzt, während die Referenz b mit einem neuen Objekt initialisiert wurde. Diese Änderungen werden detektiert, indem jedes Teilobjekt mit seiner Kopie (welche in der Abbildung aus Platzgründen nicht gezeigt ist) verglichen wird. Über die Objektnummer wird entschieden, in welche Teilobjekte von Replikat p_2 diese Änderungen eingespielt werden müssen. So wird beispielsweise die Änderung von Objekt $id_1^{-1}(3)$ an Objekt $id_2^{-1}(3)$ von Replikat p_2 nachvollzogen, indem die Instanzvariable x von 0 auf 1 geändert wird. Zusammen mit der Änderung wird das neu in Replikat p_1 eingefügte Objekt an Replikat p_2 übertragen und erhält dabei die bereits eingezeichnete Identifikationsnummer 5.

```
objects[3].x = 1
objects[4].a = reference(3)
objects[4].b = object(5)
```

Abbildung 6.17: Änderungsbeschreibung für das Replikat p_1 aus Abbildung 6.16.

Codierung und Einspielen von Änderungen

Das Auslesen der Änderungen von Replikat p_1 und das Einspielen in Replikat p_2 läuft folgendermaßen ab: Es findet ein Durchlauf durch die Teilobjekte von Replikat p_1 statt. Bei diesem Durchlauf wird jedes Teilobjekt mit seiner Kopie verglichen. Wird eine Modifikation festgestellt, wird für dieses Objekt eine Änderungsbeschreibung erstellt und seine Kopie aktualisiert. Die Änderungsbeschreibung enthält die Identifikationsnummer des betroffenen Objektes und einen Änderungseintrag für jede modifizierte Instanzvariable. Eine Änderungsbeschreibung für die Situation in Abbildung 6.16 ist in Klartext in Abbildung 6.17 gezeigt.[8] Die erste Zeile beschreibt die Änderung an der Instanzvariable x, indem der neue Wert für diese Variable codiert wird. Die zweite Zeile beschreibt die Änderung der Referenz a, indem die Identifikationsnummer des nach der Änderung referenzierten Objektes übertragen wird. Dieses Vorgehen ist für die Änderung in b nicht ausreichend, weil das referenzierte Objekt noch nicht Teil des Replikats p_2 ist. Durch bloße Nennung der lokal neu vergebenen Identifikationsnummer (5) kann die Änderung an p_2 nicht nachvollzogen werden. Aus diesem Grund muss das gesamte Objekt 5 zusammen mit der Änderungsbeschreibung übertragen werden.

Die Funktionalität zum Übertragen von Objekt(graphen) wurde in Kapitel 4 besprochen. Dieselbe Funktionalität wird während der Übertragung einer Änderungsbeschreibung eingesetzt, wenn neue Objekte in den replizierten Zustand eingefügt werden. Allerdings wird eine Integration der Referenzauflösung während der Serialisierung mit den Identifikationsnummern für Objekte eines Replikats notwendig. Die Objektserialisierung überträgt ein Objekt mit allen von ihm referenzierten Objekten. Wie an der Instanzvariable c des eingefügten Objektes 5 in Abbildung 6.16 zu sehen ist, kann ein Objekt, das neu in den replizierten Zustand eingefügt wird, durchaus Objekte referenzieren, die bereits zum replizierten Zustand gehören. In dem konkreten Fall darf die Objektserialisierung bei der Übertragung von Objekt 5 aber das alte Objekt 3 nicht mitübertragen. Objekt 3 existiert nämlich bereits im replizierten Zustand des Replikats p_2, weswegen nur eine Referenz übertragen werden darf. Normalerweise verwendet die Objektserialisierung eigene Tabellen für die Referenzauflösung, die es erlauben, zyklische Strukturen zu übertragen. Während der Übertragung einer Änderungsbeschreibung wird die Tabelle zur Zyklenerkennung der Objektserialisierung auf Sender- und Empfängerseite gegen die Tabelle mit Identifikationsnummern der Replikate ausgetauscht.

Das Einspielen einer Änderungsbeschreibung auf Empfängerseite geschieht analog zu ihrer Erzeugung. Mit der Identifikationsnummer wird das Objekt gefunden, das von der Änderung betroffen ist. Anschließend werden genau die geänderten Instanzvaria-

[8] Das in der Implementierung verwendete Format ist binär und kommt ohne die Nennung von Namen der betroffenen Instanzvariablen aus. Das Auslesen und Einspielen von Änderungen geschieht analog zur Objektserialisierung in spezialisierten Methoden pro Klasse und ist daher mit ähnlicher Effizienz möglich, da keine dynamische Typintrospektion verwendet wird.

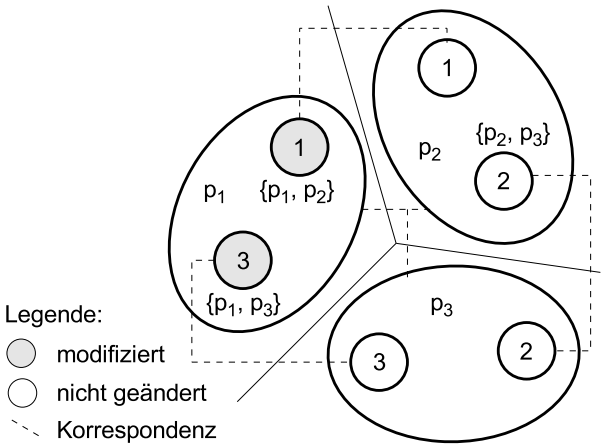


Abbildung 6.18: Zustandsfortschreibung bei partieller Replikation.

blen dieses Objektes mit den in der Änderungsbeschreibung übertragenen Werten initialisiert. Wurden Referenzen auf neue Objekte zum replizierten Zustand hinzugefügt, übernimmt die darunterliegende Objektserialisierung deren Erzeugung und Initialisierung. Auch hierbei werden die Identifikationsnummern der Teilobjekte des Replikats für die Zyklenerkennung verwendet. Das Einspielen einer Änderung in ein Replikat ist auch dann möglich, wenn der Zustand dieses Replikats ebenfalls modifiziert wurde, was für die Zustandsfortschreibung bei kollektiver Synchronisation wichtig ist.

Unter der Voraussetzung, dass sich die eingespielte und die direkt durchgeführte Änderung nicht überlappen, wird dieser folgendermaßen erreicht: Das Einspielen einer Änderung wirkt gleichzeitig auf das Replikat und seine Kopie. Demnach unterscheidet sich eine an einem Replikat von der Applikation durchgeführte direkte Modifikation von einer, die durch das Einspielen einer Änderung bewirkt wird. Das Einspielen einer Änderung in ein von der Applikation unverändertes Replikat hinterlässt dieses in einem Zustand, der wieder mit seiner Kopie übereinstimmt. Eine auf diesem Replikat anschließend durchgeführte Änderungserkennung liefert daher die leere Änderungsmenge. Wird eine Änderung in ein Replikat eingespielt, das ebenfalls von der Anwendung modifiziert wurde, liefert eine anschließende Änderungserkennung auf dem Replikat mit den eingespielten Änderungen dieselbe Änderungsmenge, die vor dem Einspielen der Änderung erkannt worden wäre, da sich die beiden Änderungen wie oben vorausgesetzt nicht überlappen. Überlappen sich die Änderungen und werden keine zusätzlichen Maßnahmen getroffen, gewinnt die Änderung des Rechenknotens, der als erster seine lokalen Änderungen übermittelt. Die anderen Änderungen werden überschrieben und gehen dabei verloren. Abschnitt 6.2.5 beschreibt einen zusätzlichen Mechanismus, mit dem überlappende Änderungen beim Einspielen zu einer Gesamtänderung so verschmolzen werden können, dass keine Änderung verloren geht.

6.2.4 Zustandsfortschreibung

Am Ende einer Synchronisation, die eine Zustandsänderung erlaubt, muss eine Zustandsfortschreibung der veralteten Replikate durchgeführt werden. Diese Zustandsfortschreibung hat das Ziel, alle Replikate wieder in einen konsistenten Zustand zu

bringen. Eine solche Zustandsfortschreibung muss sich aus den im letzten Abschnitt beschriebenen elementaren Operationen für das Auslesen und Einspielen von Änderungen an einzelnen Replikaten zusammensetzen.

Zustandsfortschreibung bei exklusiver Synchronisation

Am Ende einer exklusiven Synchronisation weist nur eines der Replikate eine Modifikation auf, welche an den übrigen Replikaten nachgezogen werden muss. Bei vollständiger Replikation muss diese Modifikation auch tatsächlich auf alle Replikate übertragen werden. Liegt partielle Replikation gemäß Abschnitt 6.1.6 vor, so sind im allgemeinen nicht alle anderen Replikate von der Änderung betroffen, da nicht alle modifizierten Objekte Teil aller Replikate sind. Bei partieller Replikation kann sich aber sogar die Änderungsmenge, die von dem geänderten Replikat zu den anderen Replikaten übertragen werden muss, von Empfänger zu Empfänger unterscheiden. Eine solche Situation ist in Abbildung 6.18 zu sehen. Ein repliziertes Objekt, bestehend aus drei Replikaten p_1, p_2 und p_3, ist so verteilt, dass p_1 und p_2 nur das Teilobjekt 1 und dass p_1 und p_3 nur das Teilobjekt 3 gemeinsam haben. Durch Schattierung ist angedeutet, dass an Replikat p_1 der Zustand der Teilobjekte 1 und 3 exklusiv geändert wurde. Im Folgenden muss der Zustand von Teilobjekt 1 von Replikat p_2 und der Zustand von Teilobjekt 3 von Replikat p_3 entsprechend fortgeschrieben werden. Dazu wird von Replikat p_1 aus die Änderungsmenge betreffend Teilobjekt 1 an Replikat p_2 übermittelt, während die Änderungsmenge, betreffend Teilobjekt 3, nur für Replikat p_3 eine Bedeutung hat.

Wenn die zugrundeliegende Kommunikationstechnologie Mehrfachversand unterstützt, könnte bei vollständiger Replikation die Änderungsmenge per Rundruf verteilt werden. Ohne die Möglichkeit des Rundrufs kommt eine baumartige Verteilung der Änderungsmenge unter den Replikaten in Frage, um einen Rundruf zu simulieren. Bei partieller Replikation erscheint dieses Vorgehen allerdings nicht sinnvoll, da von einer großen Änderungsmenge (wenn überhaupt etwas) nur ein kleiner Teil bei einem Replikat benötigt wird und ein zusätzlicher Aufwand für das Herausfiltern der relevanten Teile verursacht würde. Bei partieller Replikation erscheint es daher besser, in einzelnen Punkt-zu-Punkt-Verbindungen die jeweils für den Empfänger relevanten Teile der Änderung vom ändernden Rechenknoten aus zu verteilen. Da weder die Kommunikationsbibliothek KaRMI noch die zugrundeliegenden Kommunikationstechnologien Mehrfachversand unterstützen, kommen nur die beiden letzteren Alternativen für die Implementierung in Frage. Die vorliegende Implementierung verwendet für das Anfordern und Freigeben von exklusiven Sperren eine baumartige Nachrichtenverteilung, setzt aber für die Verteilung einer Änderungsmenge nur die dritte für partielle Replikation optimierte Strategie ein. Dies erscheint auch sinnvoll, da eine große replizierte Struktur mit vielen Replikaten ohnehin nur mit partieller Replikation effizient zu handhaben ist.

Zustandsfortschreibung bei kollektiver Synchronisation

Am Ende einer kollektiven Synchronisation sind alle Replikate modifiziert. Um einen konsistenten Zustand wiederherzustellen, muss jeder an der kollektiven Synchronisati-

on beteiligte Kontrollfaden seine lokal getätigten Änderungen extrahieren und (zumindest konzeptionell) in alle anderen Replikate einspielen. Allerdings ist es wiederum nur bei vollständiger Replikation erforderlich, die Änderungen eines Kontrollfadens an alle Replikate zu verteilen. In diesem Fall wären bei n Replikaten n^2 Kommunikationsoperationen für die Zustandsfortschreibung notwendig. Das Kommunikationsschema entspricht dann der kollektiven Operation `Alltoall` in MPI. Wenn bei partieller Replikation die Daten allerdings gut verteilt sind und ein Replikat nur mit wenigen anderen Replikaten einen nichtleeren Überlappungsbereich hat, besteht die Hoffnung, mit deutlich weniger Kommunikationoperationen ($O(n)$) auszukommen. Dies ist immer dann der Fall, wenn die während der kollektiven Synchronisation durchgeführte Berechnungsvorschrift gute Lokalitätseigenschaften hat (wenn von einem Datenelement nur Nachbarelemente in geringem Abstand für die Berechnung benötigt werden) und eine Datenverteilung gewählt wird, welche diese Lokalitätseigenschaften zur Minimierung der Überlappungsbereiche ausnutzt.

Eine effiziente Abwicklung der Zustandsfortschreibung bei partieller Replikation nach einer kollektiven Synchronisation erfordert eine unter den beteiligten Kontrollfäden abgestimmte Reihenfolge von Extraktion und Einspielen der Zustandsänderung. Wie dieser Abschnitt zeigt, führt der naive Ansatz zu einer Verklemmung. Im naiven Ansatz würde jeder Kontrollfaden direkt nach der Operation am Replikat seine lokalen Änderungen extrahieren und aufgeteilt an diejenigen Replikate versenden, mit denen er gemeinsame Objekte hat. Anschließend würde er die Änderungsmenge von seinen Nachbarreplikaten empfangen und die Änderung auf seinem Replikat einspielen. Dieses Vorgehen bewirkt eine optimale Überlappung der Aktivitäten, da alle Kontrollfäden sowohl im Sende- als auch im Empfangsschritt beschäftigt sind und außerdem die Latenz der Datenübertragung verdeckt werden kann. Allerdings ist unschwer zu erkennen, dass die Daten, welche im ersten Schritt erzeugt und versendet werden, solange gepuffert werden müssen, bis die Empfänger bereit zur Abnahme der Daten sind. Bei einer kleinen Änderungsmenge wird die ohnehin vorhandene Pufferung der Netzwerkschicht ausreichen, um eine Verklemmung zu vermeiden. Bei einer großen Änderungsmenge kann sich die Pufferkapazität erschöpfen, und das führt zu einem Blockieren der Sendeoperation und damit insgesamt zu einer Verklemmung der Zustandsfortschreibung.

Bei Programmierung in einem rein nachrichtenbasierten Kontext läge die Lösung auf der Hand. Bevor die Kontrollfäden mit der Sendeoperation beginnen, würden sie Puffer für den asynchronen Empfang bereitstellen, die ausreichen, um die erwarteten Daten aufzunehmen. Bei der Replikation von allgemeinen Objektstrukturen ist aber nicht von vorneherein klar, wie viele Daten von jedem Sender erwartet werden, so dass im Voraus keine ausreichende Pufferkapazität zur Verfügung gestellt werden kann. Auch das dynamische Vergrößern der Pufferkapazität stellt keine befriedigende Lösung dar, da die benötigte Pufferkapazität in der Größenordnung des durch das lokale Replikat belegten Speichers liegen kann. Anders als bei einem nachrichtenbasierten MPI-Programm ist nämlich der Empfangspuffer nicht die Zieladresse der Daten, sondern diese müssen noch in das lokale Replikat eingepflegt werden, stellen also nur eine Zwischenrepräsentation dar. Auch das Verwenden von zusätzlichen Kontrollfäden auf Empfängerseite, welche die empfangenen Daten direkt beim Empfang in das lokale Replikat einspielen, ist nicht praktikabel. Um an einem Replikat eine Ände-

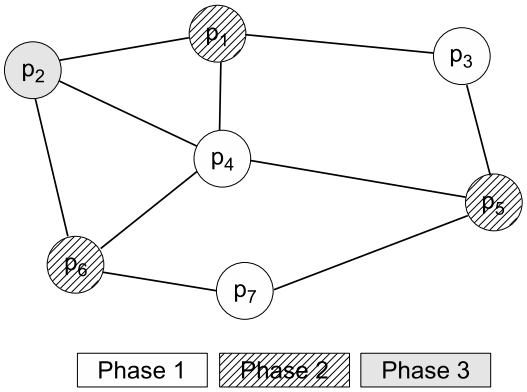


Abbildung 6.19: Phasen für den reibungslosen Ablauf einer kollektiven Zustandsfortschreibung.

rungsmenge auszulesen und nebenläufig mehrere Änderungsmengen einzuspielen, ist eine Synchronisation der beteiligten Kontrollfäden an gewissen Punkten notwendig. Da eine solche Synchronisation für den auslesenden und den einspielenden Kontrollfaden einer Änderung notwendig ist, gibt es eine zyklische Abhängigkeit, die ebenfalls zu einer Verklemmung führen kann.

Eine reibungslose Abwicklung der kollektiven Zustandsfortschreibung, die mit einer beschränkten Puffergröße auskommt und nur einen Kontrollfaden pro Replikat involviert, muss folglich in mehreren Phasen ablaufen. In jeder dieser Phasen, kann ein einzelner Kontrollfaden entweder Daten empfangen und diese in sein lokales Replikat einspielen oder seine eigenen Änderungen extrahieren und an die davon betroffenen anderen Replikate versenden. In Abbildung 6.19 ist eine solche Verteilung auf Phasen für ein repliziertes Objekt mit sieben Replikaten gezeigt. Die Kreise symbolisieren die Replikate, und die Verbindungen zwischen den Replikaten geben an, welche Replikate gemeinsame Objekte besitzen. Der Austausch von Änderungsmengen nach einer kollektiven Synchronisation ist folglich nur zwischen solchen Replikaten zu erwarten, zwischen denen eine Kante eingezeichnet ist. Die Zuordnung eines Replikats zu einer Phase gibt an, wann der zugehörige Kontrollfaden seine Extraktions- und Sendeoperation durchführt. Damit alle gesendeten Daten auch abgenommen werden, dürfen folglich keine zwei über Kanten verbundene Replikate derselben Phase zugeordnet sein. Die Zuordnung der Replikate zu Phasen läuft folglich auf das Problem der Knotenfärbung des Replikatgraphen hinaus.[9] Hat man wie im Beispiel in Abbildung 6.19 eine gültige Färbung gefunden, so können in Phase 1 die Änderungsmengen an den Replikaten p_3, p_4 und p_7 extrahiert und versendet werden, während die Replikate p_1, p_2, p_5 und p_6 diese empfangen und einspielen. Phase 2 und 3 läuft entsprechend ab, wobei in Phase 3 die Replikate p_3, p_5 und p_7 keine Operation durchzuführen haben, da sie keine Nachbarn besitzen, die in dieser Phase senden. Durch die Färbung werden die Knoten demnach so in Phasen eingeteilt, dass sich keine zwei benachbarten Knoten gleichzeitig Änderungen zuschicken.

[9]Eine gültige Knotenfärbung ordnet jedem Knoten eines ungerichteten Graphen so eine Farbe (hier Phase) zu, dass keine zwei benachbarten Knoten (die direkt durch eine Kante verbunden sind) dieselbe Farbe erhalten (hier in derselben Phase ihre Änderungen übermitteln).

Für eine schnelle Abwicklung des Austausches von Änderungen ist es günstig, eine möglichst geringe Anzahl von Phasen durchzuführen. Dazu muss eine gültige Färbung mit möglichst wenig Farben gefunden werden. Für jeden Graphen gibt es eine kleinste Anzahl k von Farben, für die eine gültige k-Färbung[10] des Graphen existiert. Diese Zahl heißt die chromatische Zahl $\chi(G)$ des Graphen. Leider ist die Berechnung der chromatischen Zahl $\chi(G)$ und damit die optimale Färbung eines Graphen np-schwer. Jedoch gilt für jeden Graphen die obere Schranke $\Delta(G) + 1$ für die chromatische Zahl [17], $\Delta(G)$ bezeichnet dabei den maximalen Grad von G. Es gibt effiziente Algorithmen, welche eine Färbung mit dieser Farbenanzahl auch garantiert erreichen. Für das Einsatzgebiet hier ist eine solche suboptimale Färbung ausreichend, da bei der vorausgesetzten Lokalität des Anwendungsalgorithmus und einer guten Verteilung der Daten der resultierende Replikatgraph einen kleinen Grad Δ besitzt. Dies ist der Fall, da jedes Replikat Objekte nur mit wenigen anderen Replikaten gemeinsam hat. Wichtiger ist die Forderung, dass die Färbung des Graphen selbst auch verteilt parallel berechnet werden muss. Hansen et al. geben hierfür in [35] einen heuristischen Algorithmus (DLF) an, der für alle Graphen mit n Knoten in $O(\Delta(G)^2 n)$ Schritten eine gute Färbung und für viele spezielle Klassen von Graphen sogar eine optimale oder nahezu optimale Färbung berechnet. DLF ist allerdings in synchronen Runden formuliert und daher nicht direkt effizient auf einem Rechnerbündel ausführbar. Da sich außerdem mit jeder Zustandsfortschreibung der Replikatgraph ändern kann (wenn neue Überlappungen zwischen Replikaten entstehen oder alte verschwinden), müsste mit DLF bei jeder kollektiven Synchronisation eine komplett neue Färbung berechnet werden. Dies ist aber zu aufwendig, zumindest dann, wenn sich der Graph nur leicht oder überhaupt nicht geändert hat. Dies ist aber nur lokal feststellbar und erschwert eine globale Entscheidung darüber, ob DLF überhaupt angestoßen werden muss.

Zwei Schritte führen zur Lösung des Problems. Erstens wurde DLF ohne global synchrone Runden umformuliert, indem seine parallelen Kontrollfäden asynchron ablaufen und sich nur über ohnehin während des Algorithmus ausgetauschte Daten implizit synchronisieren. Zweitens wurde der DLF-Algorithmus zum IDLF-Algorithmus erweitert, der eine inkrementelle Färbung eines Graphen ermöglicht. Eine inkrementelle Färbung geht von einem bereits gefärbten Graphen aus, bei dem die Färbung jedoch durch Modifikationen des Graphen ungültig geworden ist. In einem inkrementellen Schritt wird diese defekte Färbung repariert. IDLF benötigt nur einen einzigen Schritt, wenn sich der Graph nicht verändert hat oder durch die Änderung die Färbung nicht ungültig geworden ist. Bei einem echt modifizierten Graphen ist der Aufwand abhängig von der Änderung. Allerdings ist der Aufwand von IDLF im schlimmsten Fall durch die Anzahl der Runden von DLF plus eins beschränkt und auch durch wiederholtes Reparieren einer Färbung mit IDLF treten keine Degenerationserscheinungen auf (die Anzahl der benötigten Farben ist vergleichbar mit einer initialen Färbung durch DLF). Sowohl die Umformulierung von DLF zur Selbstsynchronisation und seine Erweiterung zu IDLF mit Aufwandsbeweis sind in Anhang A gegeben.

Mit Hilfe der Graphfärbung werden für die Abwicklung der kollektiven Zustandsfortschreibung keine zusätzlichen Kontrollfäden benötigt außer denen, die an der kollektiven Synchronisation beteiligt waren. Die Zustandsfortschreibung arbeitet dennoch

[10] Färbung mit k Farben.

verklemmungsfrei und kommt mit einer beschränkten Puffergröße bei der Datenübertragung aus. Jeder Kontrollfaden führt nach seiner lokalen Modifikation die folgenden Schritte aus, um den Zustand seines Replikats zu einem konsistenten Zustand fortzuschreiben, in den alle anderen kollektive Änderungen eingeflossen sind, die Teilobjekte des lokalen Replikats betreffen.

- Inkrementelles Färben des Replikatgraphen mit IDLF.
- Empfangen und Einspielen der Änderungsmengen von Nachbarreplikaten mit kleinerer Farbnummer.
- Auslesen der eigenen Änderungen und Versenden an alle Nachbarreplikate. Dabei wird die Änderungsmenge so auf die Nachbarn aufgeteilt, dass nur Änderungen an einen Empfänger versendet werden, für die er auch das zugehörige Objekt besitzt.
- Empfangen und Einspielen der Änderungen von Nachbarreplikaten mit größerer Farbnummer.

6.2.5 Verschmelzen überlappender Änderungen

Der in Abschnitt 6.2.3 beschriebene Prozess der Erkennung, Extraktion und des Einspielens von Änderungen beruht auf einem Vergleich der Instanzvariablen von Originalobjekt und Kopie, dem Auslesen von geänderten Instanzvariablen und dem Nachziehen der Änderung an anderen Replikaten. Die Granularität der Zustandsfortschreibung ist daher die Änderung einer einzelnen Instanzvariablen eines Objektes. Eine solche Änderung wird durch Überschreiben der entsprechenden Instanzvariablen mit neuen Werten in den Replikaten nachgezogen. Dadurch wird gewährleistet, dass während einer kollektiven Zustandsfortschreibung ein Objekt von verschiedenen Kontrollfäden geändert werden kann, wenn diese Änderungen nur unterschiedliche Instanzvariablen betreffen. Während der Zustandsfortschreibung werden solche nicht überlappenden Änderungen automatisch in den Replikaten zu einer Gesamtänderung zusammengefügt. Von dieser Fähigkeit kann das Programm profitieren, wenn ein Objekt mehrere unabhängig voneinander änderbare Zustände kapselt. Allerdings kann ebenso der dazu orthogonale Fall auftreten, dass mehrere Kontrollfäden ein Teilobjekt des replizierten Zustandes nebenläufig so ändern müssen, dass davon zwar dieselben Instanzvariablen betroffen sind, die Änderungen aber unabhängig voneinander sind, so dass es keine Rolle spielt, in welcher Reihenfolge die Änderungen durchgeführt werden. In diesem Fall handelt es sich um eine überlappende Änderung. Ohne besondere Maßnahmen würden von der Automatik der Zustandsfortschreibung alle bis auf eine solche Änderung beim Einspielen einer überlappenden Änderung überschrieben werden. Dieser Abschnitt behandelt Mittel, mit denen ein Verschmelzen solcher überlappender Änderungen erreicht werden kann. Das Ergebnis der Verschmelzung ist dann wie im ersten Fall die Gesamtänderung, die entstanden wäre, wenn die einzelnen Änderungen sequentiell in irgendeiner Reihenfolge durchgeführt worden wären.

Die Art und Weise, wie überlappende Änderungen zu einer Gesamtänderung verschmolzen werden sollen, kann nicht automatisch ermittelt werden, sondern ist abhängig von der Anwendungssemantik. Der Programmierer hat drei Möglichkeiten, die

```
class A {
   /** @merge AdditiveInt */
   int cnt;

   ...
}
```

Abbildung 6.20: Klasse mit Verschmelzungsannotation an einer Instanzvariablen.

```
/**
 * Additive merging algorithm for int values.
 */
class AdditiveInt extends MergeValueInt {
   public boolean isChanged(int original, int copy) {
      return original != copy;
   }

   public int diff(int original, int copy) {
      return original - copy;
   }

   public int patch(int original, int delta) {
      return original + delta;
   }
}
```

Abbildung 6.21: Additiver Verschmelzungsalgorithmus.

Verschmelzung zu spezifizieren, je nachdem ob sich die überlappende Änderung auf eine einzelne Instanzvariable beschränkt oder Objekte als Ganzes betroffen sind. Der Einfachheit halber wird hier beispielhaft nur der Fall einer einzelnen Instanzvariablen besprochen.

Abbildung 6.20 zeigt eine Klasse `A` mit einer Instanzvariablen `cnt`. Dieses Beispiel nimmt an, dass diese Instanzvariable `cnt` eine Anzahl darstellt und nebenläufig in einer kollektiven Synchronisation inkrementiert oder dekrementiert wird. Als Verschmelzung dieser nebenläufigen Änderungen erwartet die Anwendung die Gesamtänderung, die entstanden wäre, wenn alle Änderungen nicht an unterschiedlichen Kopien, sondern an einer einzigen Instanzvariablen vollzogen worden wären. Dieser Effekt wird durch eine additive Verschmelzung erreicht, bei der anstatt von Absolutänderungen nur Deltawerte übertragen werden. Zu diesem Zweck ist an die Instanzvariable `cnt` der passende Verschmelzungsalgorithmus `AdditiveInt` annotiert. Seine Implementierung ist in Abbildung 6.21 gezeigt.

Der Verschmelzungsalgorithmus besitzt drei Methoden. Davon überprüft die Methode `isChanged()`, ob die entsprechende Instanzvariable überhaupt verändert wurde. Die Methode `diff()` berechnet aus dem geänderten Wert im Originalobjekt und seiner Kopie einen Deltawert, der an einem veralteten Replikat der `patch()`-Methode als zweites Argument übergeben wird. Die Methode `patch()` zieht schließlich die Änderung an einem veralteten Wert nach. Dazu erhält sie als erstes Argument den veralteten Wert und als zweites Argument die Veränderung.

Allgemein erwartet eine Instanzvariable eines Typs `V` die Annotation eines Verschmelzungsalgorithmus vom Typ `MergeValue<V>`. Da Java keine Konkretisierung generischer Typen mit Basistypen (z. B. `MergeValue<int>`) vorsieht, stehen als Ersatz expandierte Versionen dieser Schnittstelle zur Verfügung. Davon wurde in obigem Beispiel die Variante `MergeValueInt` verwendet.

6.3 Transformation

Dieser Abschnitt erklärt die Transformation einer mit dem Schlüsselwort `replicated` markierten Klasse. Aufgrund dessen, dass auf ein repliziertes Objekt direkt zugegriffen werden kann und daher keine Stellvertreterobjekte benötigt werden, gestaltet sich die Transformation einfacher als die in Abschnitt 5.5.1 besprochene Transformation entfernter Klassen. Wie entfernte Klassen benötigen auch replizierte Klassen eine Zerlegung in statische und nicht statische Anteile. Dieser Teil der Transformation wird in Abschnitt 6.3.1 beschrieben. Die darauffolgenden Abschnitte 6.3.2 und 6.3.3 erklären die Initialisierung einer replizierten Klasse und die Erzeugung ihrer Instanzen. Abschließend diskutiert Abschnitt 6.3.4 die Transformation der verschiedenen Synchronisationsmechanismen für replizierte Objekte.

6.3.1 Klassen und Instanzen

Replizierte Klassen werden wie entfernte Klassen in ihre statischen und nicht statischen Anteile zerlegt. Die Notwendigkeit dafür leitet sich daraus ab, dass Replikate für die Zustandsfortschreibung entfernt ansprechbar sein müssen. Da aber ein Fernaufruf nur an Objekten und nicht an Klassen durchführbar ist, muss der statische Anteil einer replizierten Klasse in ein Klassenobjekt transformiert werden, um seine Replikate aktualisieren zu können.

Die Klasse aus Abbildung 6.1 wird dazu in die zwei Klassen `P` und `P_class` zerlegt, wobei `P_class` alle in der Ursprungsklasse statischen Anteile (`y` und `bar()`) als nicht statische Anteile erhält. Beide Klassen `P` und `P_class` erben nach der Transformation von der Bibliotheksklasse `ReplicatedObject`, welche die Funktionalität für die Steuerung der Zustandsfortschreibung liefert. Damit der Code, welcher die Klasse `P` verwendet, funktionsfähig bleibt, werden alle Zugriffe auf ursprünglich statische Anteile von `P` auf die entsprechenden Felder und Methoden von `P_class` umgeleitet. Zu diesem Zweck wird in die transformierte Klasse `P` als einzige statische Variable `P._class` ein Verweis auf das lokale Replikat des Klassenobjektes von `P` eingefügt. `P._class` zeigt zur Laufzeit also auf eine Instanz von `P_class` mit den ursprünglich statischen Anteilen von `P`. Damit können Zugriffe auf `P.y` und `P.bar()` direkt in `P._class.y` und `P._class.bar()` umgeschrieben werden. Anders als bei entfernten Klassen findet keine Umschreibung von Zugriffen auf Instanzvariablen in entsprechende `get`- und `set`-Methoden statt.

Der folgende Abschnitt behandelt die Initialisierung des Klassenobjektes und die für den Zugriff darauf notwendige oben verwendete statische Referenz `P._class`.

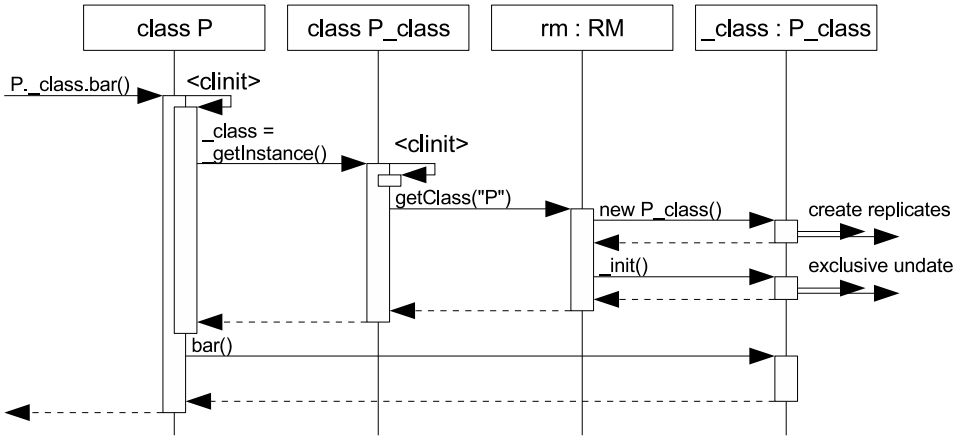


Abbildung 6.22: Laden einer replizierten Klasse im Sequenzdiagramm.

6.3.2 Klasseninitialisierung

Nach der Transformation müssen sich die beiden Klassen für statische und nicht statische Anteile beim Laden in der virtuellen Maschine so wie eine einzige reguläre Java-Klasse verhalten. Dazu muss das Klassenobjekt vor der ersten Benutzung der replizierten Klasse geladen und initialisiert werden. Abbildung 6.22 zeigt den Ablauf im Sequenzdiagramm. Dabei wird angenommen, dass die Methode `P.bar()` der replizierten Klasse `P` aufgerufen wird, die Klasse aber noch nicht geladen ist. Wie in Abschnitt 6.3.1 erläutert, wird der Aufruf der statischen Methode `P.bar()` in `P._class.bar()` transformiert. Die Ausführung dieser Anweisung stößt als erstes das Laden der Klasse `P` an. Die Verknüpfung zwischen `P` und `P_class` wird über den statischen Initialisierer von `P` hergestellt, welcher beim Laden von `P` durch die virtuelle Maschine automatisch ausgeführt wird. Wie im Sequenzdiagramm zu sehen, versucht dieser die Variable `_class` über einen Aufruf von `P_class._getInstance()` zu initialisieren. Dieser Aufruf stößt das Laden der Klasse `P_class` an. Anschließend erfolgt ein Aufruf an den `RuntimeManager`, um das Klassenobjekt nachzuschlagen. Dieser Aufruf ist potenziell ein Fernaufruf an die zentrale Verwaltungsinstanz der Laufzeitumgebung. Dies ist notwendig, um festzustellen, ob das Laden der Klasse nicht bereits auf einer anderen virtuellen Maschine angestoßen wurde. Der `RuntimeManager` stellt entweder fest, dass das Klassenobjekt bereits existiert und liefert eine Referenz darauf zurück (in der Abbildung nicht gezeigt) oder erzeugt selber das neue Klassenobjekt. Dazu ruft er den Konstruktor von `P_class` auf, der ein neues repliziertes Objekt mit allen Replikaten erzeugt. Anschließend ruft der `RuntimeManager` die `_init()`-Methode auf, in welche der statische Initialisierer der replizierten Klasse `P` verschoben wurde. Diese führt im Anschluss eine Zustandsfortschreibung durch, so dass gewährleistet ist, dass nach dem Laden alle Replikate des Klassenobjektes mit einem konsistenten Zustand starten. Erst nachdem die Variable `P._class` mit dem neu erzeugten Klassenobjekt initialisiert ist, wird der Aufruf von `bar()` tatsächlich durchgeführt.

Wie im obigen Ablauf zu sehen, ist gewährleistet, dass vor jedem Zugriff auf die replizierte Klasse `P` ihr Klassenobjekt erzeugt und der Zugriff auf sein lokales Replikat über die Variable `P._class` möglich ist. Der statische Initialisierer der Original-

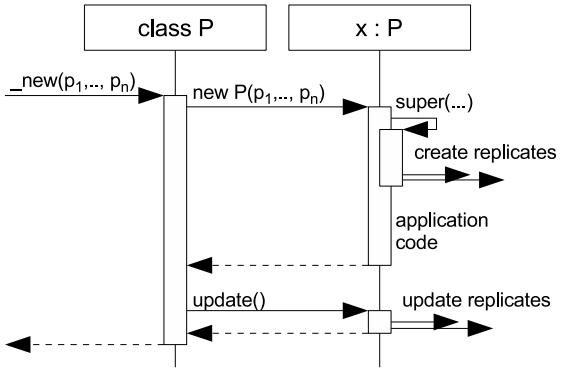


Abbildung 6.23: Erzeugen eines neuen replizierten Objektes im Sequenzdiagramm.

klasse `P` wird dabei genau einmal ausgeführt und sein Effekt in einer Zustandsfortschreibung an alle Replikate verteilt. Für die Verknüpfung der beiden Klassen `P` und `P_class` wird der Java-Mechanismus für das Klassenladen ausgenutzt.

6.3.3 Objekterzeugung

Um zu gewährleisten, dass nach der Erzeugung einer Instanz einer replizierten Klasse alle Replikate des neuen replizierten Objektes konsistent sind, muss die Erzeugung in einer generierten Fabrikmethode abgewickelt werden. Ein Sequenzdiagramm des Ablaufs dieser Fabrikmethode ist in Abbildung 6.23 zu sehen. Die Transformation ersetzt folglich alle Konstruktoraufrufe einer replizierten Klasse durch den Aufruf einer Fabrikmethode `_new()` mit der entsprechenden Signatur. Diese ruft als erstes den entsprechenden Konstruktor der replizierten Klasse auf und leitet ihre Argumente an diesen weiter. Anschließend stößt sie eine Zustandsfortschreibung an dem neu erzeugten Objekt an.

Wie in Abbildung 6.23 zu sehen ist, erzeugt der von `ReplicatedObject` ererbte Konstruktor alle weiteren Replikate des neuen replizierten Objektes. Während der Erzeugung der Replikate wurde der Code der Anwendung im Konstruktor von `P` allerdings noch nicht ausgeführt. Daher befindet sich das replizierte Objekt nicht in einem konsistenten Zustand, nachdem sein Konstruktor zurückkehrt. Die fehlende Zustandsfortschreibung wird daher im Anschluss durch die Fabrikmethode durchgeführt. Am Ende des Konstruktors von `P` eine Zustandsfortschreibung einzufügen, ist nicht günstig, da bei tieferer Vererbungshierarchie beim Aufruf mehrerer Super-Konstruktoren mehrere Zustandsfortschreibungen während einer Objekterzeugung durchgeführt werden müssten. Anders als bei allen weiteren Zustandsfortschreibungen ist die initiale Zustandsfortschreibung nicht mit einer Synchronisation verknüpft, da nur der erzeugende Kontrollfaden eine Referenz besitzt.

6.3.4 Koordination

Die Koordinationsmechanismen der gemeinsamen, exklusiven und kollektiven Synchronisation werden entweder auf reguläre Java-Synchronisationen an Replikaten oder entfernte Sperranforderungen gemäß Abschnitt 5.4 und Operationen für die Zustands-

```
exclusive synchronized(x) {
   // exclusive section
}
```

```
Lock l = preAquire(x);
synchronized(x) {
   postAquire(l, x);

   // exclusive section

   preRelease(l, x)
}
postRelease(l, x);
```

Abbildung 6.24: Transformation der exklusiven Synchronisation. Links vor und rechts nach der Transformation.

fortschreibung gemäß Abschnitt 6.2.4 abgebildet. Da die erweiterten Formen der Synchronisation nur auf replizierte Objekte anwendbar sind, muss die Transformation eine Laufzeitüberprüfung einführen, wenn eine Synchronisation an einem Schnittstellen-Typ oder `java.lang.Object` durchgeführt wird. In diesem Fall ist zur Übersetzungszeit nicht zu entscheiden, ob es sich bei dem betreffenden Objekt um ein repliziertes Objekt handeln wird. Die Laufzeitüberprüfung löst dann eine Ausnahmebedingung aus, wenn eine reguläre Synchronisation an einem replizierten Objekt oder eine erweiterte Synchronisation an einem nicht replizierten Objekt durchgeführt wird. Für die gemeinsame Synchronisation ist keine besondere Transformation notwendig, da sie direkt auf reguläre Java-Synchronisation am lokalen Replikat abgebildet wird. Exklusive und kollektive Synchronisation werden im folgenden besprochen.

Exklusive Synchronisation

Exklusive Synchronisation wird nach dem Schema in Abbildung 6.24 transformiert. Die Transformation benutzt neben der Java-Synchronisation des lokalen Replikats die vier Bibliotheksmethoden `preAquire()`, `postAquire()`, `preRelease()` und `postRelease()`. Die Methode `preAquire()` holt ein neues Ticket[11], stößt damit einen Sperr-Rundruf an allen übrigen Replikaten an und liefert den Sperrkontext als `Lock`-Objekt zurück. Anschließend wird das lokale Replikat in einer regulären Java-Synchronisation gesperrt. Der Aufruf von `postAquire()` wartet daraufhin solange, bis das im ersten Schritt angeforderte Ticket gültig geworden ist und alle anderen Replikate erfolgreich gesperrt wurden. Das Warten findet dabei am lokalen Replikat `x` statt, wodurch die in der vorangegangenen Synchronisation erworbene Sperre zwischenzeitlich wieder freigegeben wird. Dadurch wird ein verklemmungsfreies Anfordern aller Sperren der Replikate erreicht, da die endgültige Sperranforderung (bei der Rückkehr aus der Methode `postAquire()`) nur mit einem gültigen Ticket und daher bei nebenläufiger Anforderung mehrerer exklusiver Sperren in einer definierten Reihenfolge stattfindet. Nachdem der exklusiv synchronisierte Bereich abgearbeitet ist, führt `preRelease()` eine exklusive Zustandsfortschreibung durch und gibt anschließend den Befehl zur Freigabe der entfernt angeforderten Sperren. Die Methode `postRelease()` schließt die exklusive Synchronisation ab, indem sie die Freigabebestätigung der entfernt angeforderten Sperren abwartet.

[11] Vgl. Abschnitt 6.2.2.

```
collective synchronized(x) {
   // exclusive section
}
```

```
synchronized(x) {
   // collective section

   collectiveUpdate(x)
}
```

Abbildung 6.25: Transformation der kollektiven Synchronisation. Links vor und rechts nach der Transformation.

Kollektive Synchronisation

Die Transformation der kollektiven Synchronisation ist in Abbildung 6.25 zu sehen. Sie besteht lediglich aus seiner Java-Synchronisation am lokalen Replikat und einem abschließenden Aufruf der Bibliotheksmethode `collectiveUpdate()`, welche eine kollektive Zustandsfortschreibung durchführt. Da jeweils nur das lokale Replikat synchronisiert wird, kann dieselbe Operation gleichzeitig an anderen Replikaten des replizierten Objektes durchgeführt werden. Damit die abschließende Zustandsfortschreibung erfolgreich durchgeführt werden kann, *muss* diese Operation sogar an allen Replikaten gleichzeitig durchgeführt werden, da `collectiveUpdate()` einen Kontrollfaden pro Replikat für die Extraktion der lokalen Änderung und das Einspielen der von den anderen Replikaten übermittelten Änderungen voraussetzt. Die Möglichkeit muss ausgeschlossen werden, dass eine kollektive Synchronisation nebenläufig mit einer exklusiven Synchronisation angestoßen wird. Wird diese Regel verletzt, kann es zu einer Verklemmung kommen, wenn manche Replikate bereits von der exklusiven Synchronisation gesperrt sind und an anderen Replikaten Kontrollfäden den kollektiv synchronisierten Bereich betreten haben. In diesem Fall kann weder die exklusive noch die kollektive Synchronisation erfolgreich durchgeführt werden, da keine von beiden eine Sperre an allen Replikaten durchsetzen kann. Diese Regel ermöglicht es, eine zusätzliche Kommunikation zu Beginn einer kollektiven Synchronisation einzusparen, in der ebenfalls analog zur exklusiven Synchronisation ein Ticket angefordert und verteilt werden müsste. Benötigt die Anwendung eine abwechselnde Verwendung von kollektiver und exklusiver Synchronisation, muss sie die Verklemmung durch Einsatz zusätzlicher Barrieren zwischen den unterschiedlichen Synchronisationsformen verhindern. Dies erscheint vorteilhaft, weil kollektive Synchronisation im Gegensatz zu exklusiver Synchronisation sehr häufig (meist in einer Schleife) durchgeführt wird, da sie einen datenparallelen Hauptschritt markiert. Eine zusätzliche Kommunikationsoperation zu Beginn jeder kollektiven Synchronisation würde aber genau diese Operation unnötig bremsen.

Kapitel 7

Evaluation

7.1 Evaluationsumgebung

Alle Messungen dieses Kapitels wurden an dem Rechnerbündel *Carla* vorgenommen, das im Institut für Programmstrukturen und Datenorganisation der Universität Karlsruhe installiert ist. Dieses Rechnerbündel besteht aus 16 Knoten, ausgestattet mit jeweils zwei Pentium III-Prozessoren mit 800 MHz Taktfrequenz. Die Rechenknoten sind sowohl über Fast Ethernet als auch über Myrinet Netzwerk miteinander verbunden. Zum Zeitpunkt der Messungen war SuSE Linux in der Kernversion 2.4.21-smp4G, die Myrinet Treibersoftware GM 2.016 und die Java-Entwicklungsumgebung der Version 1.4.2_06 von Sun Microsystems installiert. In allen Messungen wurde die Serverversion der virtuellen Java-Maschine verwendet. Die absoluten Messergebnisse sind insbesondere von der Güte des in der virtuellen Maschine verwendeten Laufzeitübersetzers abhängig. Die Tendenz der Messergebnisse lässt sich aber auch auf anderen virtuellen Maschinen bestätigen. Bei Verwendung der entsprechenden virtuellen Maschine von IBM werden ähnliche Geschwindigkeitsgewinne, wie im folgenden beschrieben, erzielt. Interessanterweise gewinnt die in der vorliegenden Arbeit vorgestellte Lösung bei besserem Laufzeitübersetzer überproportional, da sich der generierte Code besonders gut optimieren lässt.

7.2 Benchmarkkerne

Die folgenden Untersuchungen arbeiten die Einzelergebnisse der Arbeit heraus. Dabei dient die Java-Objektserialisierung und der entfernte Methodenaufruf RMI als Referenz für die Optimierungen der Kommunikationsleistung. Die folgenden Abschnitte führen dazu eine getrennte Betrachtung der schnellen Objektserialisierung, der Kommunikationsbibliothek KaRMI und der transparenten Replikation durch. Dabei werden die Effekte der speziellen Optimierungen aufgezeigt.

7.2.1 Namensgebung

Die folgenden Abschnitte verwenden eine Reihe synthetischer Objekte und Methoden, anhand derer die Serialisierungs- und Kommunikationsleistung vermessen wird. Ihre

Namenskürzel werden hier eingeführt.

Die folgenden Objekte werden für die Benchmarks verwendet:

- `Obj4` bezeichnet ein kleines Objekt, das aus vier einzelnen Integerwerten besteht.
- `FObj4` heißt ein Objekt `Obj4`, dessen Klasse zusätzlich mit der Annotation `@faceless` versehen ist (vgl. Abschnitt 4.3.1 über die Einsparung von Zyklentests).
- `Obj32` bezeichnet ein größeres Objekt, das 32 Instanzvariablen vom Typ Integer hat.
- `Tree3` steht für einen Objektgraphen mit 15 Einzelobjekten in Form eines balancierten Binärbaumes der Tiefe 3. Neben den Verweisen auf den linken und rechten Nachfolger enthält jedes Knotenobjekt zusätzlich 4 weitere Instanzvariablen vom Typ Integer.
- `byte[]` bezeichnet ein Byte-Feld der Länge 100.
- `float[]` bezeichnet ein Feld aus Fließkommazahlen von ebenfalls der Länge 100.
- `empty` ist ein repliziertes Objekt ohne Instanzvariablen. Es wird benutzt zum Vermessen der reinen Synchronisationszeiten.
- `grid` ist ein repliziertes zweidimensionales Gitter der Größe 8×8. Jeder Gitterpunkt speichert einen Fließkommawert und Verweise auf den Nachbar-Gitterpunkt in Nord-, Süd-, West- und Ostrichtung.

Für die Evaluation des Fernaufrufs wird die folgende Familie von Methoden verwendet:

- Mit `ping(arg)` wird eine Methode bezeichnet, welche mit einem optionalen Argument `arg` aufgerufen wird und sofort ohne weitere Berechnungen und ohne Rückgabe eines Ergebnisses (`void`) zurückkehrt. Die ping-Methode wird verwendet, um die Latenz eines (Fern-)Aufrufs zu bestimmen. Dabei wird die Summe aus der Latenz von Hin- und Rückrichtung gemessen, da ein Methodenaufruf erst dann abgearbeitet ist, wenn eine Bestätigungsbotschaft empfangen wurde (auch dann, wenn kein Ergebnis zurückübermittelt wurde).

Die folgenden Kürzel bezeichnen die verglichenen Systeme. Diese werden gegebenenfalls mit einer Option `opt` parametrisiert, die im zugehörigen Text näher erklärt wird:

- `JDK <opt>` steht für die Standard-Objektserialisierung in der Java-Entwicklungsumgebung.
- `RMI <opt>` steht für den regulären entfernten Methodenaufruf.

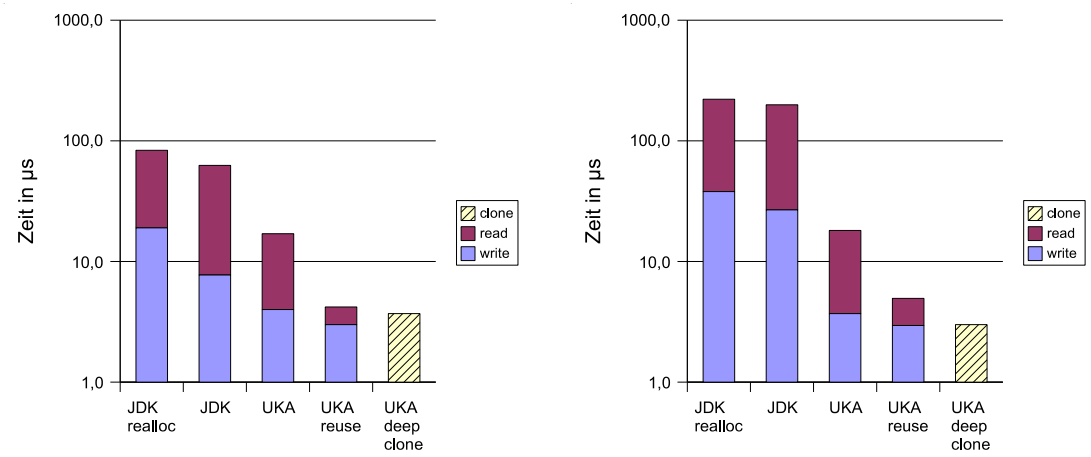


Abbildung 7.1: Zeitaufwand für Serialisierung, Deserialisierung und tiefe Kopie eines `Obj4`-Objektes (links) bzw. eines `Obj32`-Objektes (rechts). Beide Graphiken verwenden logarithmischen Maßstab.

- `UKA <opt>` bezeichnet die in dieser Arbeit eingeführte schnelle Serialisierung.
- `KaRMI <opt>` meint den in dieser Arbeit vorgestellten schnellen Fernaufruf für Rechnerbündel.

Die optionale Parametrisierung `opt` der untersuchten Pakete wird direkt bei den Messungen näher erklärt.

7.2.2 Objektserialisierung

Bei der Evaluation der Objektserialisierung wird für die Objekte aus Abschnitt 7.2.1 jeweils getrennt die Zeit gemessen, die das Schreiben des Objektes in einen Strom und das Wiederauslesen des Objektes aus dem Strom kostet. Dabei wird das Szenario nachgebildet, in welchem die Serialisierung im zugehörigen Paket für Fernaufruf verwendet wird. Abbildung 7.1 links zeigt die Zeiten für ein `Obj4`-Objekt. Um die erforderliche Messgenauigkeit zu erreichen, wird bei der Messung eine große Anzahl von Serialisierungs- und Deserialisierungsoperationen gemittelt. Die Anzahl dieser Operationen wird automatisch eingestellt.

Die Option `realloc` (die Säule ganz links) für die `JDK`-Serialisierung bedeutet, dass der betreffende Serialisierungs- und Deserialisierungsstrom pro serialisiertem Objekt neu angelegt wird. Dies entspricht der Verwendung der Serialisierung im `RMI`-Paket, wenn ein einzelnes Objekt als Argument einer entfernt aufgerufenen Methode übertragen wird.

`JDK` ohne Option misst die aufeinanderfolgende Serialisierung vieler `Obj4`-Objekte unter Benutzung desselben Stroms. Die so gemessene Zeit könnte mit der `JDK`-Serialisierung beim Fernaufruf erreicht werden, wenn kein Neuanlegen der Serialisierungskomponenten pro Aufruf durchgeführt würde.

Die Säule `UKA` ohne Option zeigt die der Säule `JDK` entsprechende Zeit, die durch Umschalten auf die `UKA`-Serialisierung erreicht werden kann. Hierbei wird ebenfalls

kein Neuanlegen des Serialisierungsstromes pro Aufruf durchgeführt, allerdings wird der Strom vor jeder neuen Serialisierung komplett zurückgesetzt. Das Zurücksetzen des Stroms ist notwendig, damit in aufeinanderfolgenden entfernten Methodenaufrufen Objekte neu übertragen werden und die Tabellen zur Zyklenerkennung im Serialisierungsmechanismus nicht im Laufe der Zeit alle je übertragenen Objekte ansammeln.

Mit der Option `reuse` der vorletzten Säule wird die zusätzliche Optimierung der `UKA`-Serialisierung eingeschaltet, die es erlaubt, beim Zurücksetzen des Stromes die schon übertragene Typinformation zu behalten und so ein Typalphabet zwischen Sender und Empfänger zu etablieren, das eine besonders effiziente Typcodierung ermöglicht.

Die letzte Säule mit der Option `deepclone` zeigt die Zeit, die für eine optimierte tiefe Kopie des Objektes benötigt wird. Sie entspricht einem kurzgeschlossenen Serialisierungs-/Deserialisierungsprozess, der für das allgemeine Kopieren von Objektgraphen in Fernaufrufen anfällt, welche ein (potenziell entferntes) Objekt derselben virtuellen Maschine zum Ziel haben.

Man erkennt, dass bei der Serialisierung dieses extrem kleinen Objektes das vermeiden der Neuallokation des Serialisierers 25% der Zeit einspart. Der Übergang zur `UKA`-Serialisierung mit pro Objekt generierten Versende- und Empfangsroutinen spart von der verbleibenden Zeit wiederum 73% ein. Die optimierte Typcodierung der `UKA`-Serialisierung, welche erst durch Wiederverwenden des Serialisierers für mehrere Serialisierungs- und Deserialisierungsprozesse möglich wird, spart davon weitere 75% ein, so dass eine Gesamtersparnis von 99,95% gegenüber der `JDK`-Serialisierung mit Neuanlegen des Serialisierers erreicht wird. Dies entspricht einer Geschwindigkeitssteigerung um Faktor 20.

Bei dem sehr kleinen Objekt `Obj4` ist der Geschwindigkeitsgewinn durch „tiefes“ Kopieren gegenüber der Summe aus `UKA`-Serialisierung und Deserialisierung vernachlässigbar. Bei dem etwas größeren Objekt `Obj32` aus Abbildung 7.1 rechts sieht man dagegen, dass die Objektkopie gegenüber der Kombination aus Serialisierung und Deserialisierung weitere 39% einsparen kann.

Weiter fällt auf, dass sich beim Übergang von `Obj4` zu `Obj32` die Ersparnis durch Vermeiden des Neuanlegens der Serialisierungskomponenten von 25 auf 10% reduziert, dagegen aber die Ersparnis durch Übergang auf die `UKA`-Serialisierung von 72 auf 91% anwächst. Die Ersparnis durch die optimierte Typcodierung bleibt mit 73% ungefähr konstant. Insgesamt erhöht sich die Geschwindigkeitssteigerung durch die `UKA`-Serialisierung auf Faktor 45.

In Abbildung 7.2 wird die Serialisierung und die tiefe Kopie von verzeigerten Strukturen untersucht. Bei dem Testobjekt handelt es sich um einen balancierten Binärbaum mit Tiefe 3 (bestehen aus 15 Einzelobjekten). Die Abbildung ist wie die vorhergehenden Abbildungen aufgebaut. Nur die mit `embedded` gekennzeichneten Säulen fünf und sieben sind eingeschoben und zeigen die Verbesserung der Serialisierungsleistung bei Einbettung der internen Referenzen des Baumes. Durch Vergleich der Säulen vier und fünf bzw. sechs und sieben in Abbildung 7.2 erkennt man, dass die Einbettung der Referenzen für die Serialisierung bzw. tiefe Kopie ausgenutzt werden kann und dort zu einer Zeiteinsparung von 37 bzw. 35 % führt. Insgesamt wird damit eine Zeitersparnis gegenüber den besten mit der JDK-Serialisierung erreichbaren Werten von 83 % für die Serialisierung und 93 % für die tiefe Kopie erreicht. Dies

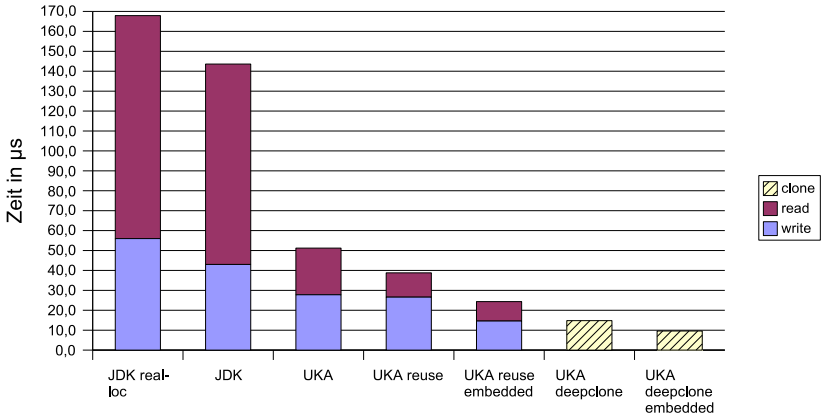


Abbildung 7.2: Zeitaufwand für Serialisierung und Deserialisierung eines `Tree3` Objektes (mit und ohne eingebettete Referenzen).

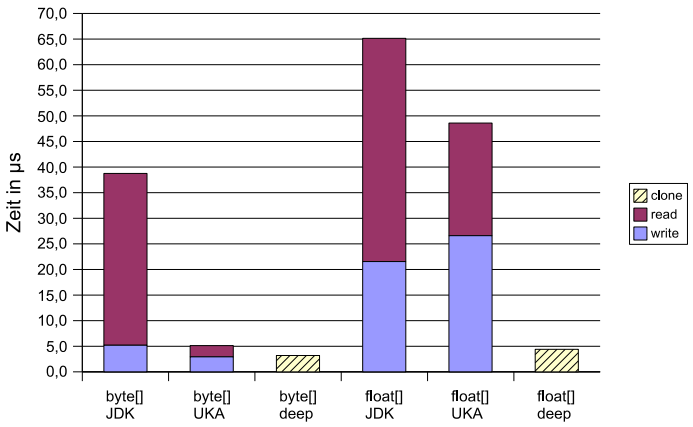


Abbildung 7.3: Serialisierungsleistung bei Feldern.

entspricht einer Geschwindigkeitssteigerung um Faktor 6 bzw. 15.

Abbildung 7.3 zeigt die Serialisierungsleistung bei Feldern. Das schlechteste Ergebnis erreicht die `UKA`-Serialisierung bei der Übertragung des Feldes von Fließkommazahlen. Dies ist aber nicht weiter verwunderlich, da in Java die Konvertierung einer Fließkommazahl in die entsprechende Bitrepräsentation nicht innerhalb der virtuellen Maschine möglich ist. Für die Konvertierung muss für jede einzelne Fließkommazahl eine native Methode aus der Standardbibliothek aufgerufen werden, da die `UKA`-Serialisierung plattformunabhängig in reinem Java ohne zusätzlichen nativen Code und ohne Eingriff in die virtuelle Maschine implementiert ist. Die `JDK`-Serialisierung verwendet dagegen eine native Routine der Serialisierungsbibliothek, welche die Konvertierung für das gesamte Feld auf einmal durchführt (je nach zugrundeliegender Hardwarearchitektur besteht diese Konvertierung aus einer einfachen Kopie oder einer Kopie mit Vertauschung der Bytereihenfolge). Diese Konvertierungsfunktion ist aber nicht öffentlich und kann ausschließlich von der JDK-Serialisierung verwendet werden. Die UKA-Serialisierung muss die öffentlich zur Verfügung stehenden Standard-Konvertierungsroutinen verwenden, die einen teuren Aus- und Wieder-

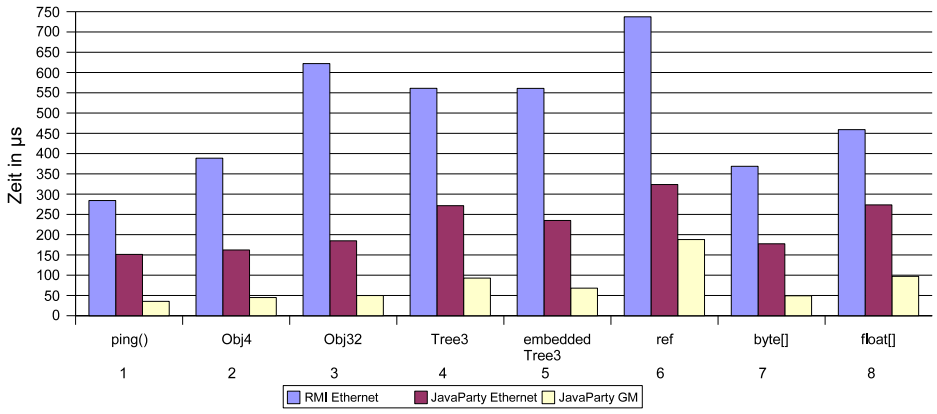


Abbildung 7.4: Vergleich der Latenz entfernter Methodenaufrufe über RMI (Fast Ethernet) und KaRMI (sowohl über Fast Ethernet als auch über Myrinet/GM).

eintritt in die virtuelle Maschine pro Feldelement verursachen. Um so erstaunlicher ist es, dass die `UKA`-Serialisierung auch im Fall der Übertragung des Fließkommazahlenfeldes dennoch einen kleinen Geschwindigkeitsvorteil von 25% erreichen kann. Das Kopieren eines Feldes im Speicher wird von der UKA-Serialisierung auf die eingebaute `clone()`-Methode abgebildet und kann deshalb einen Geschwindigkeitsgewinn um Faktor 15 erreichen.

7.2.3 Fernaufruf

Abbildung 7.4 zeigt die Zeiten für komplette Fernaufrufe inklusive der Übergabe eines Argumentes (in eine Richtung). Als Argumente kommen dieselben Objekte zum Einsatz, deren Serialisierung in Abschnitt 7.2.2 untersucht wurde. Es sind immer drei Säulen in einer Gruppe zusammengefasst. Die jeweils linke Säule stellt die Zeit für einen `ping()`-Aufruf über RMI und Fast Ethernet dar. Die mittlere Säule stellt dem als direkten Vergleich die Zeit desselben Fernaufrufs über KaRMI gegenüber. Die jeweils rechte Säule zeigt die Zeit des Fernaufrufs, wenn als KaRMI-Transporttechnologie das Myrinet-Hochgeschwindigkeitsnetzwerk verwendet wird. Die verschiedenen Säulengruppen zeigen die Zeiten bei Übertragung unterschiedlicher Argumentobjekte.

Die Säulengruppe ganz links zeigt die Zeit für einen `ping()`-Aufruf ganz ohne Argument. Sie dient als Vergleichsmaßstab für den Fall, dass keine Objektserialisierung notwendig ist. In diesem Fall ist KaRMI bei Verwendung identischer Kommunikationshardware mit $151\ \mu s$ um 47% schneller als RMI, das etwas über $280\ \mu s$ benötigt. Beim Einsatz von Myrinet über die nativ angebundene Treibersoftware GM gewinnt KaRMI um weitere 77% und benötigt für den Fernaufruf noch lediglich $36\ \mu s$. Daraus ergibt sich ein Gesamtgeschwindigkeitsvorteil von 88%.

Wie erwartet zeigt sich bei RMI kein Geschwindigkeitsunterschied zwischen den Varianten `Tree3` und `embedded Tree3`, da die dort verwendete `JDK`-Serialisierung keine Optimierungen für als eingebettet markierte Referenzen vorsieht. Dagegen kann man bei Verwendung von KaRMI den Geschwindigkeitsvorteil bei der Übertragung der Variante mit eingebetteten Referenzen bei der Verwendung beider Transporttechnologien erkennen. Der relative Geschwindigkeitsvorteil kann hier allerdings

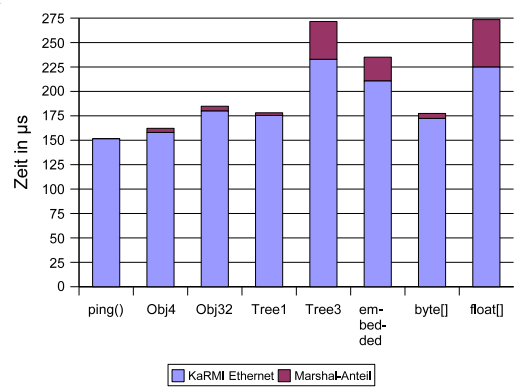


Abbildung 7.5: Zeit für Serialisierung und Netzwerkübertragung mit KaRMI über Fast Ethernet.

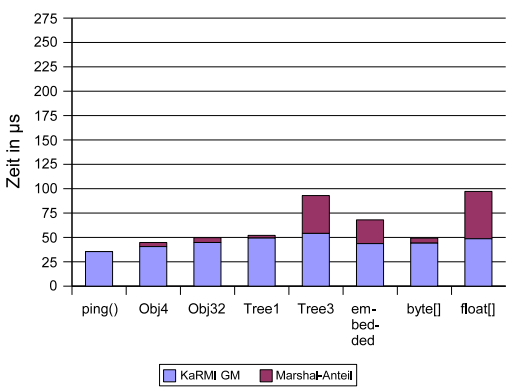


Abbildung 7.6: Zeit für Serialisierung und Netzwerkübertragung mit KaRMI über Myrinet/GM.

aufgrund der Zeit für die Netzwerkübertragung und des übrigen Aufwands für den Fernaufruf nicht so deutlich ausfallen wie bei der Einzelbetrachtung der Serialisierung in Abschnitt 7.2.2. Interessanterweise ist der absolute Geschwindigkeitsvorteil von `embedded Tree3` gegenüber `Tree3` mit $25\ \mu s$ bei Ethernet und $37\ \mu s$ bei GM überproportional groß gegenüber dem Vorteil bei der reinen Serialisierung, der bei $15,2\ \mu s$ für `embedded Tree3` lag. Dieser große Unterschied kann kaum allein mit der eingesparten Datenmenge erklärt werden, die sich aus dem Wegfall der übertragenen Informationen zur Zyklenerkennung ergibt.

Die Säulengruppe an Position 6 in Abbildung 7.4 zeigt die Zeit für die Übertragung einer entfernten Referenz in einem entfernten Methodenaufruf. Diese Situation ist besonders komplex, da hier der verteilte Speicherbereiniger mit involviert ist. Aus diesem Grund lässt sich der reine Serialisierungsaufwand hierfür nicht separat messen. Sowohl mit RMI als auch mit KaRMI ist diese Operation besonders teuer. Vergleicht man allerdings den Mehraufwand von `ref` gegenüber einem `ping()` ohne Argument, so schneidet auch hier KaRMI mit $200\ \mu s$ bei Ethernet und $62\ \mu s$ bei GM gegenüber RMI mit $292\ \mu s$ gut ab.

Insgesamt liegt der Geschwindigkeitsvorteil von KaRMI gegenüber RMI bei den hier vermessenen Operationen (und bei Verwendung derselben Transporttechnologie) zwischen 40 und 70%. Durch Verwendung des Hochgeschwindigkeitsnetzwerks Myrinet lässt sich mit KaRMI darüberhinaus eine weitere Geschwindigkeitssteigerung zwischen 42 und 77% erreichen, was insgesamt einem Geschwindigkeitsvorteil von 75 bis 92% entspricht. Am größten ist die Ersparnis bei der Übertragung von `Obj32` und am geringsten bei der Übertragung einer entfernten Referenz `ref`.

In den gegenübergestellten Abbildungen 7.5 und 7.6 sind nochmals die Latenzzeiten für die `ping()`-Aufrufe mit KaRMI über Ethernet und GM eingezeichnet. Zusätzlich ist in jeder Säule der Zeitanteil markiert, den die reine Serialisierung und Deserialisierung des entsprechenden Argumentobjektes alleine in Anspruch nimmt. Aus der verbleibenden Säulenhöhe erkennt man, dass es beim Fernaufruf neben der Netzwerklatenz, der Objektserialisierung und der (konstanten) Verwaltungszeit für den Fernaufruf einen weiteren von der Art des Argumentes abhängigen Zeitanteil gibt. Dieser ist

bei Ethernet deutlich größer als bei GM und kann zumindest zum Teil mit der Zeit für die Übertragung der eigentlichen Nutzdaten erklärt werden.

7.2.4 Vergleich mit Manta/Ibis

Das Manta-Projekt, das ursprünglich Java für Hochleistungsrechnen nativ übersetzt hat, hat diesen Ansatz offenbar aufgegeben und mit Ibis eine Bibliothek für plattformunabhängige Hochleistungskommunikation in Java entwickelt. Folgende Tabelle vergleicht die in [99] veröffentlichen Werte für Latenz und Durchsatz entfernter Methodenaufrufe mit denen von KaRMI. Für die Messungen wird das JDK von IBM auf vergleichbaren Rechner verwendet (1 GHz Pentium III für Ibis und 800 MHz Pentium III für KaRMI).

	Ethernet/TCP		Myrinet/GM	
Latenz (μs)	Ibis	KaRMI	Ibis	KaRMI
ping()	131,3	152,8	42,2	38,8
Durchsatz (MB/s)				
100 KB byte[]	10,3	10,6	76,0	87,7
100 KB int[]	9,6	10,5	76,0	67,5
100 KB double[]	9,1	10,3	76,0	27,3
1023 Knoten Binärbaum	4,3	6,1	22,9	13,7

Vermessen wird der leere Methodenaufruf `ping()`, die Übertragung von Feldern primitiver Datentypen und die Übertragung eines balancierten Binärbaumes mit 1023 Knotenobjekten. Jedes Objekt des Baumes besitzt wie das Testobjekt `Tree3` zusätzlich vier Ganzzahlwerte. Bei den Methodenaufrufen mit Argument ist der Durchsatz in MB/s für die reinen Anwendungsdaten (ohne Verzeigerungsstruktur beim Binärbaum) angegeben.

KaRMI kann das Ethernet bei Übertragung der Felder besser sättigen und erreicht einen ca. 40 % höheren Durchsatz bei Übertragung des Binärbaumes. Die Latenz für den leeren Methodenaufruf ist über RMI dagegen 16 % größer. Über Myrinet erzielt KaRMI eine geringfügig bessere Latenz und einen um 15 % höheren Durchsatz bei der Übertragung des Byte-Feldes. Bei der Übertragung des Ganzzahl- und Fließkommafeldes verwendet Ibis nativen Code für die direkte Übertragung ohne Konvertierung in ein plattformunabhängiges Format und daher identische Werte wie für die Übertragung des Byte-Feldes. KaRMI führt in beiden Fällen eine Konvertierung durch, so dass die Übertragungsleistung einbricht. Interessanterweise ist Ibis bei Verwendung von Myrinet um 67 % schneller bei der Übertragung des Binärbaumes.

Ibis hat keine Optimierungen für den lokalen Zugriff auf potenziell entfernte Objekte und kennt weder transparent entfernte Klassen noch maschinenüberspannende verteilte Kontrollfäden. Für den Aufbau eines verteilten Objektraumes, wie er für die vorliegende Arbeit benötigt wird, ist Ibis daher nicht geeignet.

7.2.5 Potenzieller Fernaufruf

Wie in Abschnitt 5.2.4 besprochen, ist es in einer verteilten Bündelumgebung nicht nur möglich, sondern explizit erwünscht, dass potenzielle Fernaufrufe zur Laufzeit des

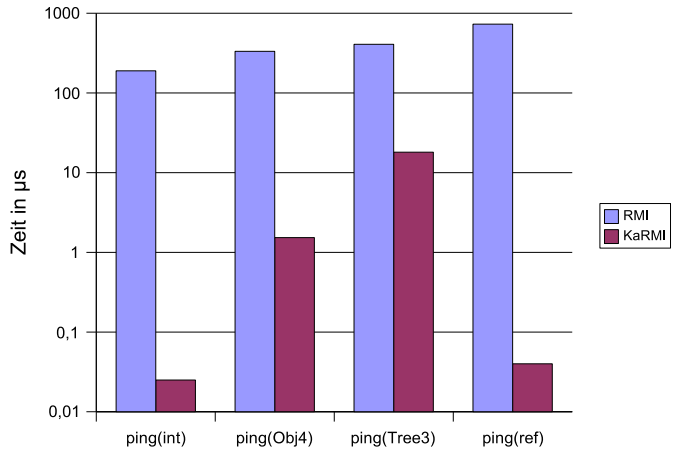


Abbildung 7.7: Zeit für lokale, aber potenziell entfernte Methodenaufrufe in RMI und KaRMI.

Programms an Objekte gerichtet sind, welche sich in derselben virtuellen Maschine wie der Aufrufer aufhalten. Damit diese Lokalitätsoptimierung den gewünschten Effekt hat, müssen solche lokalen (aber potenziell entfernten) Aufrufe besonders effizient abgehandelt werden können.

Abbildung 7.7 zeigt die Latenzzeiten eines solchen lokalen, aber potenziell entfernten Methodenaufrufs bei Verwendung von RMI bzw. KaRMI. Dabei wird die entsprechende Methode an einem Referenzobjekt aufgerufen, das auf ein (vom Typ her) entferntes Objekt auf derselben virtuellen Maschine verweist. Damit die dafür benötigten Zeiten von RMI und KaRMI überhaupt in einem Diagramm darstellbar sind, wurde hier eine logarithmische Skala für die Zeit gewählt. Die Latenzzeit bei einem Lokalaufruf über RMI liegt dabei in derselben Größenordnung wie bei einem echten Fernaufruf. Offensichtlich hat damit eine Lokalitätsoptimierung in einem verteilt parallelen Programm keine Aussicht auf Erfolg. KaRMI erreicht durch spezielle Optimierung für lokale Aufrufe einen Geschwindigkeitsvorteil von fast 4 Größenordnungen oder Faktor 7500.

Auch bei KaRMI sieht man deutliche Geschwindigkeitsunterschiede bei den vier Säulengruppen in Abbildung 7.7. Diese rühren daher, dass für einen transparenten Fernaufruf unter Umständen eine tiefe Kopie des übergebenen Objektgraphen angefertigt werden muss. Wie in Abschnitt 5.2.4 beschrieben, kann dieses Kopieren unter gewissen Randbedingungen vermieden werden. Dies ist u.a. bei den Aufrufen der Fall, deren Messergebnisse an erster und letzter Stelle gezeigt werden. Bei `ping(int)` haben alle Argumente primitive Typen, die auch in regulären Java-Aufrufen per Wert übergeben werden. Im letzten Fall bei `ping(ref)` wird eine entfernte Referenz übergeben. Dieser Fall ist mit RMI besonders teuer, mit KaRMI dagegen findet der lokale Aufruf besonders effizient statt, da es sich bei einer entfernten Referenz um ein unveränderliches Objekt handelt, das ohne Verletzung der Transparenz per Referenz übergeben werden kann. In den beiden anderen Fällen fertigt auch KaRMI eine tiefe Kopie der übergebenen Objekte an. In dieser Situation wird deutlich, warum beim Anfertigen von Objektkopien bestmögliche Effizienz besonders wichtig ist.

Abbildung 7.8 untersucht die Laufzeit von KaRMI bei `ping(Obj4)` näher (zwei-

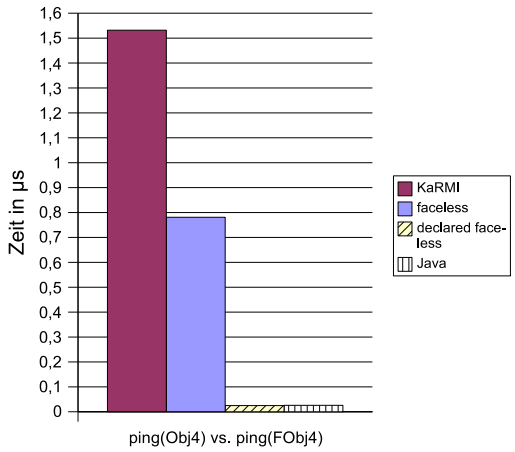


Abbildung 7.8: Zeiteinsparung durch Vermeidung von Kopieroperationen für `@faceless` markierte Objekte.

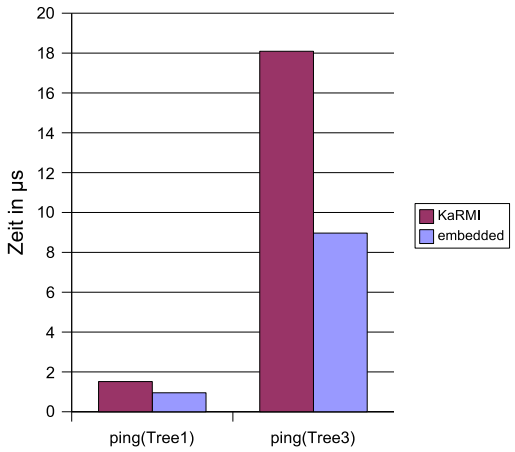


Abbildung 7.9: Zeiteinsparung beim tiefen Kopieren bei Objekten mit eingebetteten Referenzen.

te Säule in Abbildung 7.7). Die erste Säule stellt dabei den lokalen KaRMI Aufruf bei Übergabe eines `Obj4` Objektes ohne weitere Optimierungen dar. Anhand der zweiten Säule kann man erkennen, wie sich der Aufwand verringert, wenn dieselbe Methode (`ping(Obj4)`) aufgerufen wird, aber ein typkompatibles Objekt übergeben wird, dessen Klasse `@faceless` annotiert wurde. In diesem Fall wird der Mechanismus für die Erstellung einer Objektkopie in Gang gesetzt, da zur Übersetzungszeit nicht auszuschließen war, dass ein Kopievorgang zur Laufzeit notwendig werden könnte. Allerdings erkennt dieser Mechanismus am dynamischen Typ des tatsächlich übergebenen Objektes, dass auf das Kopieren verzichtet werden kann. Dies spart immerhin die Hälfte der ansonsten aufzuwendenden Zeit ein. Wird, wie in der durch die dritte Säule repräsentierten Situation, statt der Methode `ping(Obj4)` direkt eine Methode `ping(FObj4)` aufgerufen, bei der bereits zur Übersetzungszeit feststeht, dass auch auf den Kopiervorgang ganz verzichtet werden kann, schrumpft die benötigte Zeit fast auf die Latenz eines regulären lokalen Java-Methodenaufrufs (vierte Säule).

In Abbildung 7.9 ist zu sehen, wie sich die Annotation für eingebettete Objekte bei deren Übergabe an lokale, aber potenziell entfernte Aufrufe auswirkt. Im wesentlichen ist hier genau der bereits in Abschnitt 7.2.2 gemessene Geschwindigkeitsvorteil beim Erstellen einer tiefen Kopie für das entsprechende Argumentobjekt zu sehen. Da durch die KaRMI-Optimierung für lokale Aufrufe fast kein weiterer Aufwand anfällt, kommt der Geschwindigkeitsvorteil beim Kopierprozess voll zum Tragen.

7.2.6 Transparent entfernte Klassen

Die Abbildungen 7.10, 7.11 und 7.12 zeigen die Untersuchungsergebnisse des durch die JavaParty-Transparenzschicht verursachten Mehraufwands. Die erste Abbildung zieht dazu wiederum den transparenten Lokalaufruf heran, die beiden weiteren Abbildungen zeigen die Messergebnisse für tatsächlich entfernt durchgeführte Aufrufe über Ethernet bzw. Myrinet/GM.

Die in Abbildung 7.10 gezeigten Säulen vergleichen die Latenzen eines regulären

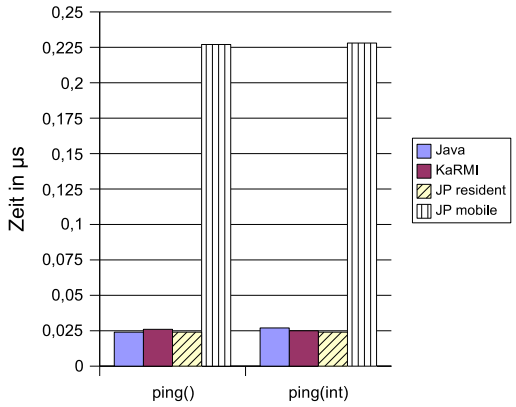


Abbildung 7.10: Kosten der Transparenzschicht beim Lokalaufruf an einem potenziell entfernten Objekt.

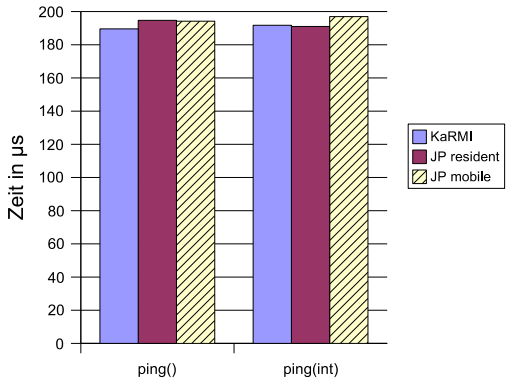


Abbildung 7.11: Kosten der Transparenz beim Fernzugriff über KaRMI/Ethernet

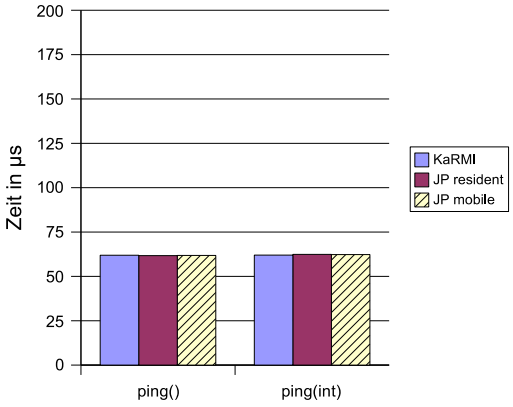


Abbildung 7.12: Kosten der Transparenz beim Fernzugriff über KaRMI/GM

Java-Methodenaufrufes mit einem entsprechenden potenziell entfernten Methodenaufruf über eine entfernte KaRMI-Referenz (Säule 2), ein JavaParty-Stellvertreterobjekt, welches ein stationäres Objekt referenziert (Säule 3) und ein JavaParty-Stellvertreterobjekt (Säule 4), welches ein mobiles Objekt anspricht, das nach seiner Erzeugung seinen Aufenthaltsort wechseln kann. Die Säulen eins bis drei sind innerhalb der Messtoleranz gleich groß. Die letzte Säule überragt die vorhergehenden allerdings um ungefähr Faktor acht. Der Grund für den erheblichen Geschwindigkeitsnachteil beim Ansprechen mobiler Objekte liegt darin begründet, dass bei solchen Objekten bei jedem Aufruf eine Buchhaltungsoperation durchgeführt werden muss, die verhindert, dass ein Objekt umgezogen werden kann, während ein Kontrollfaden eine Methode dieses Objektes ausführt.

Die in den Abbildungen 7.11 und 7.12 dargestellten Messergebnisse relativieren die obige Aussage. Bei tatsächlich entfernt durchgeführten Aufrufen lässt sich innerhalb der Messtoleranz kein Geschwindigkeitsunterschied zwischen Aufrufen über KaRMI oder die JavaParty-Transparenzschicht nachweisen. Dabei spielt es auch keine Rolle, ob es sich bei dem angesprochenen Objekt um ein mobiles oder stationäres Objekt handelt. Allerdings kann der Mehraufwand für mobile Objekte beim transparenten lokalen Zugriff nicht vernachlässigt werden. Dies ist insbesondere deswegen schade, da sich mobile Objekte vor allem dafür eignen, zur Laufzeit durch geschickten Objektumzug günstige Lokalitätseigenschaften herzustellen und so die Anzahl der lokalen Aufrufe gegenüber den Fernaufrufen zu maximieren. Allerdings liegt die Latenz absolut gesehen bei einem Lokalzugriff auf ein mobiles Objekt immer noch im Submikrosekundenbereich. Im Vergleich zu einem Fernzugriff auf ein Objekt auf einen anderen Knoten ist dies auch bei Verwendung schneller Kommunikationstechnologie immer noch um Faktor 300 günstiger. Mit der Optimierung für den schnellen transparent lokalen Zugriff auf potenziell entfernte (mobile) Objekte eröffnet sich erst die Möglichkeit, dynamisch Lokalitätsoptimierung durchzuführen. Wird von der Möglichkeit des Objektumzugs in der Anwendung allerdings kein Gebrauch gemacht, sollte eine Klasse aufgrund der dargestellten Messergebnisse als stationär deklariert werden, um in den Genuss der vollen Leistung beim lokalen Zugriff zu kommen.

7.2.7 Transparente Kontrollfäden

Wie in Abschnitt 5.3 beschrieben, bietet KaRMI völlige Transparenz bezüglich Synchronisation und Kontrolle von maschinenüberspannenden Kontrollfäden. Diese Eigenschaft verursacht einen gewissen Mehraufwand bei jedem entfernten Aufruf und ist daher optional zuschaltbar. Die Ergebnisse der Untersuchungen dieses Mehraufwandes sind in Abbildung 7.13 zusammengefasst. Die drei Säulen auf der linken Hälfte zeigen Messergebnisse für einen entfernten `ping()`-Aufruf, während die Säulen auf der rechten Hälfte die Latenz für eine entfernte Sperranforderung angeben.

Links in jeder Dreiergruppe ist als Referenz die Latenz des entsprechenden RMI-Aufrufs dargestellt (der keine Transparenz von maschinenüberspannenden Kontrollfäden bietet). Die unteren Bereiche der beiden jeweils folgenden Säulen zeigen die Messergebnisse für KaRMI über Ethernet bzw. GM (Säulen 2 und 5 bzw. 3 und 6), die bereits in Abschnitt 7.2.3 bei der Evaluation des Fernaufrufs vorgestellt wurden. Zur Herstellung der Transparenz für Kontrollfäden fällt ein Mehraufwand an, der durch die

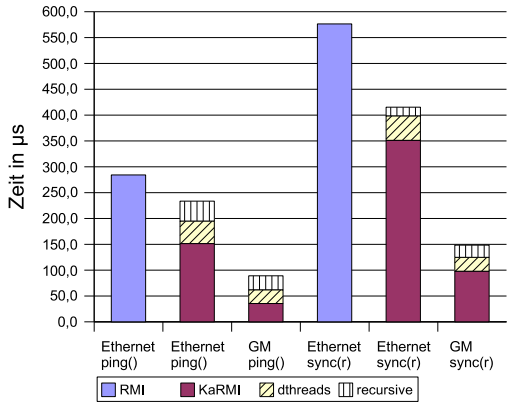


Abbildung 7.13: Aufwand beim Methodenaufruf für die Realisation transparent verteilter Kontrollfäden.

auf diese Säulen aufgesetzten Blöcke verdeutlicht wird.

Der jeweils untere mit `dthreads` bezeichnete Teil wird dadurch verursacht, dass der Kontrollfaden auf Senderseite die Rückkehr des Fernaufrufes mittelbar über einen Stellvertreterkontrollfaden abwartet, um für den Fall ansprechbar zu bleiben, dass ein rekursiver Rückaufruf in demselben (maschinenüberspannenden) Kontrollfaden eintrifft. Zur Herstellung der Transparenz der Synchronisation ist es notwendig, dass ein potenzieller Rückruf in dem Kontrollfaden ausgeführt wird, welcher der lokale Stellvertreter des maschinenüberspannenden Kontrollfadens auf der Ausgangsmaschine ist. Dieser lokale Stellvertreter ist aber gerade jener Kontrollfaden, der auf die Rückkehr des von ihm angestoßenen Fernaufrufs wartet.

Der jeweils obere, mit `recursive` bezeichnete Teil fällt in dem Fall an, wenn auf Empfängerseite des Fernaufrufs tatsächlich ein Wechsel des Kontrollfadens stattfinden muss, da auf der lokalen Maschine bereits ein Segment des maschinenüberspannenden Kontrollfadens angelegt ist. In diesem Fall wartet dort bereits ein lokaler Stellvertreter auf die Rückkehr eines von ihm angestoßenen Fernaufrufs. Statt dieser Rückkehr trifft dort aber ein weiterer rekursiver Fernaufruf im selben maschinenüberspannenden Kontrollfaden ein. Dieser ankommende Fernaufruf muss in diesem Fall durch einen Wechsel des Kontrollfadens zur Abarbeitung an den lokalen Stellvertreter übergeben werden.

Der mit `dthreads` bezeichnete Zusatzaufwand ist hauptsächlich von Implementierungsdetails bestimmt, was die KaRMI-interne Verwendung von Netzwerkverbindungen betrifft. Der Wechsel des Kontrollfadens beim Empfang der Rückkehr eines Fernaufrufs ist nur daher notwendig, weil das Warten an Netzwerkverbindungen in Java nicht definiert unterbrochen werden kann. Dieses Problem wurde mit Version 1.4 der Java-Umgebung durch Einführung des zusätzlichen Paketes `java.nio` behoben, welches genau die benötigte Funktionalität von kontrolliert unterbrechbaren Netzwerkoperationen zur Verfügung stellt. Durch Umstellung von KaRMI von `java.io` auf `java.nio` ließe sich der Wechsel der Kontrollfäden auf Aufrufseite und damit ein Großteil des hier entstehenden Mehraufwandes einsparen.

Der mit `recursive` bezeichnete Teil des Mehraufwandes zur Herstellung von

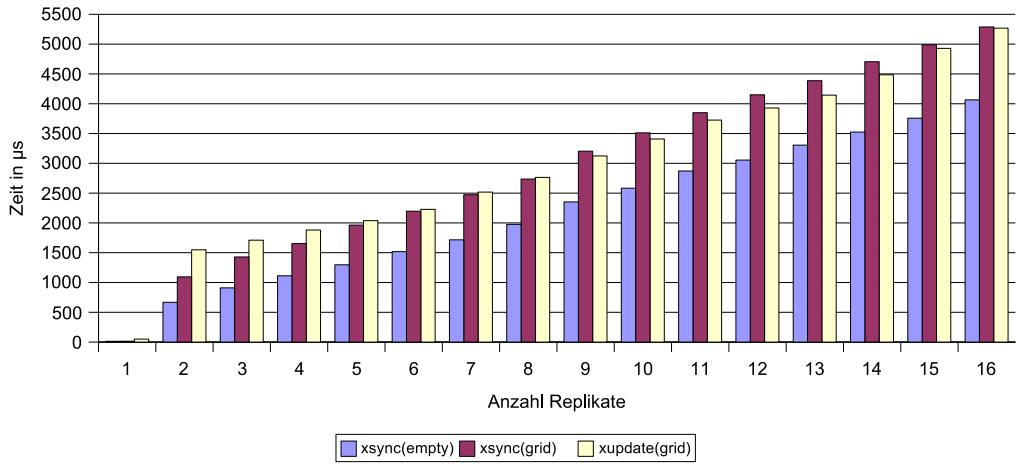


Abbildung 7.14: Exklusive Synchronisation an verschieden großen replizierten Objekten unterschiedlichen Replikationsgrades.

Transparenz bezüglich maschinenüberspannender Kontrollfäden fällt nicht bei jedem Fernaufruf an. Der Wechsel des Kontrollfadens auf Empfängerseite ist nur dann notwendig, wenn es sich bei dem Fernaufruf tatsächlich um einen rekursiven Rückruf handelt.

Insgesamt kann festgehalten werden, dass trotz Zuschaltens der Transparenz für maschinenüberspannende Kontrollfäden ein Fernaufruf bzw. eine entfernte Synchronisation über KaRMI und Ethernet immer noch 18 bzw. 28 % schneller abgearbeitet wird, als dies mit RMI möglich ist, obwohl dort keinerlei Rücksicht auf die Problematik maschinenüberspannender Kontrollfäden gelegt wird. Ärgerlich ist der im Vergleich zum eigentlichen Fernaufruf relativ große Zeitanteil insbesondere beim Einsatz von Hochgeschwindigkeitsnetzwerken wie Myrinet, da, wie in Abbildung 7.13 zu sehen, der Mehraufwand zur Bereitstellung transparent maschinenüberspannender Kontrollfäden die Latenz des Aufrufs verdoppeln kann. Andererseits ist dieser Mehraufwand konstant und fällt bei Aufrufen mit größeren Argumenten nicht mehr so stark ins Gewicht.

7.2.8 Exklusive Synchronisation

Abbildung 7.14 zeigt die Zeiten, welche für eine exklusive Synchronisation an unterschiedlichen replizierten Objekten mit unterschiedlichem Replikationsgrad benötigt werden. Auf der X-Achse ist die Anzahl der Knoten abgetragen, auf denen das Objekt repliziert ist, auf der Y-Achse die Zeit in μs. Eine Gruppe von Säulen symbolisiert die Zeiten für unterschiedliche Objekte bzw. unterschiedliche Operationen. Die jeweils erste Säule zeigt die Zeit für eine Synchronisation an einem leeren replizierten Objekt (`empty`). Die zweite und dritte Säule zeigen die Zeiten für eine Synchronisation an einem replizierten Gitterobjekt (`grid`) mit 256 Gitterpunkt-Objekten. Während der mit `xsync` bezeichneten Messung, wurde das replizierte Objekt nicht modifiziert, so dass nur das reine Durchsuchen auf Änderungen und die Übermittlung einer leeren Ände-

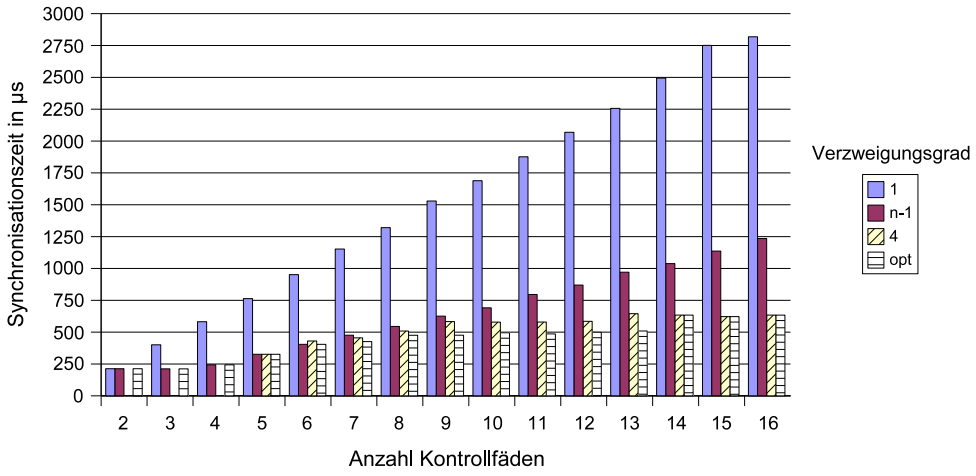


Abbildung 7.15: Zeit für Barrierenoperation bei optimalem Verzweigungsgrad des Synchronisationsbaumes und variabler Anzahl Kontrollfäden.

rungsmenge gemessen wird. Bei der Messung `xupdate` wurde das Gitter während der Synchronisation geändert, so dass hier die Kombination aus Änderungserkennung und Zustandsfortschreibung zu sehen ist.

Es ist zu erkennen, dass bei Replikationsgrad eins, wenn also keine Replikation vorliegt, weder Aufwand für Änderungserkennung noch Zustandsfortschreibung anfallen. In diesem Fall verhält sich das replizierte Objekt nahezu identisch mit einem regulären Java-Objekt. Dies wird erreicht, indem Änderungserkennung und Zustandsfortschreibung nur für diejenigen Teile eines replizierten Objektes durchgeführt werden, die tatsächlich auf mehr als einem Knoten vorrätig gehalten sind. Für alle anderen Teile (im Fall des Replikationsgrades eins für das gesamte replizierte Objekt) werden weder lokale Kopien für die Änderungserkennung angelegt, noch werden sie während der Zustandsfortschreibung überhaupt berücksichtigt. Teile eines replizierten Objektes, die nicht repliziert sind, befinden sich definitionsgemäß immer in einem konsistenten Zustand.

Weiterhin ist in Abbildung 7.14 ab Replikationsgrad zwei ein linearer Trend in den Synchronisationszeiten zu erkennen. Dabei spielt es offensichtlich keine Rolle, wie groß das replizierte Objekt ist und ob dieser Zustand überhaupt modifiziert wurde. Dies liegt, wie in Abschnitt 6.2.4 beschrieben, daran, dass die Zustandsfortschreibung bei exklusiver Synchronisation immer in einer (nicht überlappenden) eins-zu-n-Kommunikation stattfindet. Dabei spielt es keine Rolle, ob dabei tatsächlich eine nichtleere Zustandsfortschreibung übermittelt wird oder ob nach dem Überprüfen auf Änderungen mitgeteilt wird, dass keine Änderungen durchgeführt wurden. In beiden Fällen erfolgt dieser Vorgang sequentiell, was den linearen Trend in den Synchronisationszeiten für die exklusive Synchronisation erklärt. Dieses Vorgehen ist eindeutig suboptimal, da mit einer baumartigen Verteilung der Zustandsfortschreibung unter den Replikaten eine bessere Skalierbarkeit erreicht werden könnte. Dass dies auch tatsächlich realisierbar wäre, zeigt die im Anschluss diskutierte Abbildung 7.15. Wie allerdings

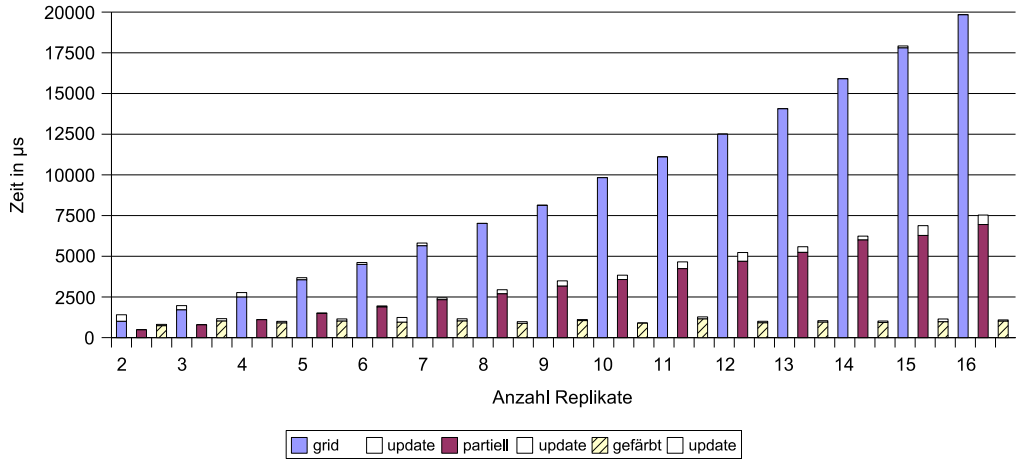


Abbildung 7.16: Zeiten für kollektive Zustandsfortschreibung in unterschiedlichen Optimierungsstufen.

ebenfalls in Abschnitt 6.2.4 ausgeführt, ist eine baumartige Verteilung von Änderungen bei partiell replizierten Objekten (vgl. Abschnitt 6.1.6) nur schwierig realisierbar, da hier der zulässige Verteilungsbaum durch die Art der Partition des Objektes eingeschränkt ist. Die vorliegende Arbeit interessiert sich hauptsächlich für den kollektiven Aspekt (insbesondere partiell) replizierter Objekte, weswegen die Optimierung der exklusiven Synchronisation nicht weiter verfolgt wurde.

Dass über Verteilungsbäume eine bessere Skalierung auch bei exklusiver Synchronisation erreichbar wäre, zeigt Abbildung 7.15. Anhand einer manuell implementierten Barrierenoperation wird hier die reine Synchronisationszeit untersucht. Diese Operation entspricht ungefähr dem Kommunikationsmuster der reinen Sperroperation ohne Änderungserkennung und Zustandsfortschreibung, das bei der exklusiven Synchronisation auftritt. In der Abbildung ist die benötigte Synchronisationszeit für eine variable Anzahl von Kontrollfäden n (2 bis 16) unter Verwendung eines Synchronisationsbaumes mit unterschiedlichem Verzweigungsgrad v aufgetragen. Sowohl bei $v = 1$ (jeweils erste Säule einer Gruppe) als auch bei $v = n-1$ (jeweils zweite Säule) wächst die Synchronisationszeit linear mit der Anzahl der beteiligten Kontrollfäden. Der optimale Verzweigungsgrad v_{opt} liegt zwischen diesen beiden Extremwerten. Die Synchronisationszeiten für Verzweigungsgrad 4 und den optimalen Verzweigungsgrad v_{opt} sind in den Säulen drei und vier einer Gruppe eingezeichnet. Der optimale Verzweigungsgrad variiert zwischen 1 und 5. Abbildung 7.15 zeigt, dass 4 eine brauchbare Näherung für den optimalen Verzweigungsgrad eines Verteilungsbaumes bei der vorliegenden Netzwerktopologie ist.

7.2.9 Kollektive Synchronisation

Abbildung 7.16 ist der Auswertung kollektiver Zustandsfortschreibung gewidmet. An ihr sind die Effekte der in den Abschnitten 6.1.6 und 6.2.4 beschriebenen Optimierungstechniken für die kollektive Zustandsfortschreibung zu erkennen.

Bei der hier durchgeführten kollektiven Operation handelt es sich um die Lösung einer Laplace-Differentialgleichung auf einem Gitter mit 256 Gitterpunkten in Torusform gemäß der Beschreibung in Abschnitt 3.3.2. Als Veranschaulichung kann Abbildung 3.2 dienen. Der alleinige Zweck ist hier allerdings die Vermessung der Kosten kollektiver Zustandsfortschreibungen. Dazu sind Problemgröße und Implementierung so gewählt, dass auch unter optimalen Systemvoraussetzungen keine Skalierung (verringerte Zeit bei Hinzunahme von Rechenknoten) zu erwarten ist, sondern nur die Synchronisationskosten sichtbar werden (da bei dieser Dimensionierung des Problems die Zeit für die eigentliche Problemlösung vernachlässigt werden kann).

In Abbildung 7.16 sind die Zeiten für einen datenparallelen BSP-Hauptschritt (realisiert als kollektive Synchronisation) auf dem bereits im letzten Abschnitt verwendeten replizierten Gitterobjekt bei unterschiedlichem Replikationsgrad und unterschiedlichen Optimierungsoptionen abgetragen. Der weiß gelassene Säulenkopf zeigt an, wieviel der Zeit für den eigentlichen Datenaustausch aufgewendet wird. Die jeweils erste Säule in einer Gruppe (`grid`) symbolisiert die notwendige Zeit bei Totalreplikation. In diesem Fall ist das gesamte Objekt auf allen über den Replikationgrad angegebenen Knoten repliziert. Während der Zustandsfortschreibung muss daher das gesamte Objekt auf allen Knoten auf Änderungen untersucht und die gefundenen Änderungen an alle anderen Replikate versandt werden. In diesem Fall sieht man ebenso wie bei der exklusiven Synchronisation einen linearen Trend in der Zeit für die Zustandsfortschreibung. Anders als bei der exklusiven Synchronisation sind die absoluten Zeiten allerdings deutlich größer. Dies verwundert aber nicht weiter, da die Zustandsfortschreibung der kollektiven Synchronisation ohne weitere Optimierungen, wie in Abschnitt 6.2.4 beschrieben, wie bei n miteinander verbundenen, aber nacheinander ausgeführten exklusiven Synchronisationen durchgeführt werden muss.

Die jeweils zweite Säule einer Gruppe in Abbildung 7.16 zeigt die Zeit für dieselbe datenparallele Operation bei Verwendung eines partiell replizierten Objektes. Der Torus ist hier zeilenblockweise partitioniert, so dass jedes Replikat nur mit seinem oberen und unteren Nachbarn jeweils zwei Zeilen von gemeinsamen Gitterpunkt-Objekten besitzt. Die Zeiten für die Zustandsfortschreibung bei partieller Replikation sind deutlich geringer als bei Totalreplikation, allerdings lässt sich durch Vergleich der Säulenhöhen immer noch ein linearer Trend ablesen. Dieser zeigt an, dass die Synchronisationszeit immer noch linear mit dem Replikationsgrad wächst und daher auch bei einem realistischer dimensionierten Problem keine bzw. keine gute Skalierung zu erwarten ist, da weiterhin der Parallelisierungsgewinn durch einen wachsenden Aufwand bei der Zustandsfortschreibung aufgefressen würde. Die absoluten Zeiten sind besser als bei Totalreplikation, da nur ein kleiner Teil des gesamten Gitters auf jedem Knoten auf Änderungen untersucht werden muss und diese Änderung auch nur an zwei anstatt an alle anderen Replikate übermittelt werden muss. Allerdings findet auch hier die Zustandsfortschreibung nicht überlappend sequentiell für alle Replikate statt. Diese suboptimale Reihenfolge bedingt den weiterhin linearen Trend der Synchronisationszeiten bei wachsendem Replikationsgrad.

Bei der mit *gefärbt* bezeichneten Datenreihe wird jetzt zusätzlich noch die Optimierung der Reihenfolge der Zustandsfortschreibung durch Färben des Replikatgraphen eingeschaltet. Bei dem hier beschriebenen Problem besteht dieser Graph gerade aus einem einzelnen Kreis, dessen Länge dem Replikationsgrad entspricht. Ein solcher

Kreis lässt sich mit zwei oder drei Farben einfärben, je nachdem, ob seine Länge gerade oder ungerade ist. Statt in Knotenreihenfolge findet die Zustandsfortschreibung dann überlappend in Reihenfolge aufsteigender Farbnummern statt. Die Eigenschaft, dass keine Replikate mit gemeinsamen Objekten dieselbe Farbe erhalten können, sichert dabei zu, dass trotz überlappender Zustandsfortschreibung keine Konflikte entstehen können. Offensichtlich bringt genau diese Optimierung einen durchschlagenden Erfolg. Die Zeiten für die Zustandsfortschreibung bleiben mit dieser Optimierung konstant, unabhängig vom Replikationsgrad des replizierten Objektes. Genau dieses Verhalten ist für eine gute Skalierung bei der Lösung einer realen Problemstellung notwendig.

Zwei weitere Beobachtungen lassen sich bei genauerem Vergleich der Synchronisationszeiten bei partieller Replikation und bei zusätzlich durch Graphfärbung optimierter Reihenfolge machen: Offensichtlich hat das Färben seinen Preis. Bei Replikationsgrad zwei und drei ist die Variante ohne Färbung schneller. Das ist damit zu erklären, dass (mindestens beim gewählten Problem) in diesen beiden Fällen die triviale Färbung (bei der jedes Replikat seine eigene Farbe erhält) optimal ist. Jeglicher Zusatzaufwand, der bei der Berechnung der Färbung und bei ihrer Überprüfung nach Modifikation des replizierten Objektes entsteht, muss sich negativ auf die Geschwindigkeit der Zustandsfortschreibung auswirken. Ab Replikationsgrad vier amortisiert sich allerdings dieser Aufwand. Weiterhin sind die Säulen bei geradzahligem Replikationsgrad (mit Ausnahme von Replikationsgrad sechs) etwas kleiner als bei ungeradzahligem Replikationsgrad. Dies lässt darauf schließen, dass der Färbealgorithmus fast immer eine optimale Färbung gefunden hat (diese enthält bei ungeradzahliger Länge des Kreises eine Farbe mehr als bei geradzahliger Länge, was eine schlechtere Überlappung der Zustandsfortschreibung und damit eine längere Zeit für die kollektive Synchronisation bedeutet).

7.3 Evaluation von Anwendungen

7.3.1 TSP

Das Problem des Handlungsreisenden (*Traveling Salesman Problem*, TSP) ist ein bekanntes kombinatorisches Problem. In seiner allgemeinen Form, wird in einem gewichteten Graphen ein Hamiltonkreis mit minimalem Kantengewicht gesucht. Dieser Hamiltonkreis repräsentiert eine geschlossene Reiseroute minimaler Länge, mit der ein Handlungsreisender eine vorgegebene Anzahl von Städten besuchen kann. Das Lösen von TSP ist NP-vollständig und daher im allgemeinen schon für eine relativ kleine Stadtanzahl nicht mehr praktisch durchführbar.

TSP wird als Benchmark untersucht, da das Lösungsverfahren übertragbar auf viele Optimierungs- und Suchprobleme ist. Um die kürzeste Route zu finden, müssen im Prinzip alle Routen aufgezählt und unter diesen die kürzeste ermittelt werden. Beim Aufzählen können allerdings Routenanfänge zusammen mit allen möglichen Ergänzungen direkt verworfen werden, wenn keine Chance mehr besteht, die Anfangsroute so zu vervollständigen, dass eine fertige Route entsteht, die kürzer ist als eine bis dahin bereits bekannte kurze Route. Dieses Beschneiden des Suchraumes (engl. *pruning*)

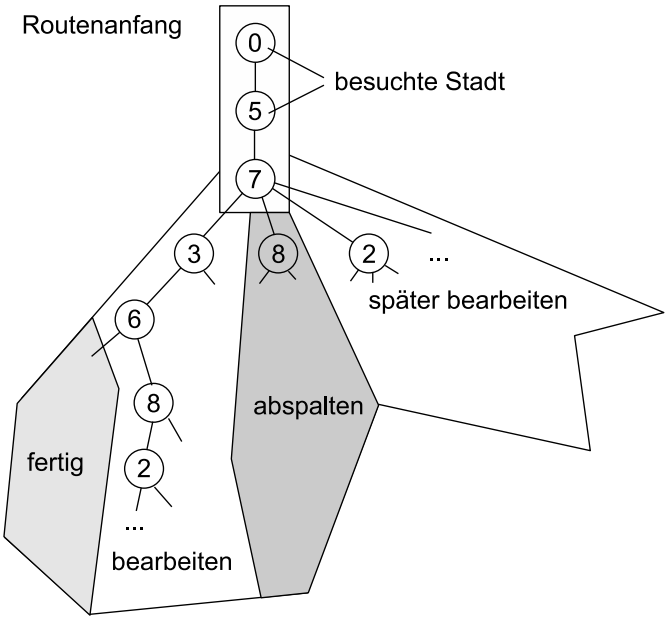


Abbildung 7.17: Arbeitspaket für eine Aktivität des parallelen TSP-Algorithmus.

ist eine häufig eingesetzte Technik bei vielen Such- und Optimierungsalgorithmen. So suchen beispielsweise Schachprogramme einen Antwortzug, indem sie alle möglichen Folgestellungen berechnen und bewerten, die von einem Ausgangszustand über eine Zugfolge bestimmter Länge erreicht werden können. Da auch hierbei die Anzahl der möglichen Folgestellungen exponentiell mit der Zuglänge wächst, werden nur vielversprechende Zuganfänge bei der Suche weiter berücksichtigt. TSP eignet sich allerdings besser für Benchmarkzwecke, da der Suchalgorithmus (Aufzählung der Routen) einfach und das Bewertungskriterium (Länge der Route) offensichtlich ist.

Damit die Suchraumbeschneidung überhaupt greift, ist es wichtig, dass möglichst schnell eine relativ gute Route gefunden wird. Zu diesem Zweck wird für die Aufzählung der Routen die Heuristik verwendet, dass ein Routenanfang mit Städten in der Reihenfolge wachsenden Abstandes zur jeweils letzten besuchten Stadt ergänzt wird. Die erste Route entspricht damit der Greedy-Lösung, bei der immer die nächstliegende Stadt besucht wird. Zwar kann diese Initial-Lösung immer noch beliebig weit von der Optimallösung entfernt sein, in der Regel ist sie aber deutlich besser als eine zufällige Lösung, die sich bei der Aufzählung der Städte in willkürlicher Reihenfolge ergäbe.

Parallelisierung

Die Parallelisierung der Suche nach der kürzesten Route erscheint trivial. Der Suchraum muss auf die an der Suche beteiligten Aktivitäten aufgeteilt werden. Jede parallele Aktivität untersucht eine Teilmenge aller Routen, wobei sie für die Beschneidung ihres Teils des Suchraums dasselbe Kriterium wie in der sequentiellen Variante einsetzt. Eine Aktivität, welche eine neue kürzeste Route gefunden hat, meldet diese an eine Zentrale. Die neue beste Route dient danach allen Aktivitäten als Kriterium für die Beschneidung des Suchraumes.

Allerdings ergeben sich bei der Parallelisierung zwei Probleme, die eine gute Ska-

lierung schwierig machen. Das Aufbauen einer *einzelnen* Route, die Berechnung ihrer Länge und die Entscheidung, ob eine neue beste Route gefunden wurde, ist eine Operation, die nur sehr geringe Zeit benötigt. Damit scheiden einzelne Routen als Basis für die Aufteilung des Suchraums und damit für die Parallelisierung aus, da ansonsten der Parallelisierungsgewinn durch die Mehrarbeit bei der Verteilung mehr als aufgewogen würde. Als Grundlage für die Aufteilung der Arbeit auf die parallelen Aktivitäten kommen also nur Routenanfänge in Frage, die für alle möglichen Ergänzungen zu kompletten Routen stehen und so einen Unterraum des Suchraums repräsentieren. Das zweite Problem ist allerdings, dass aufgrund des Kriteriums zur Suchraumbeschneidung kaum abschätzbar ist, wie viele Routen aus einer gegebenen Anfangsroute tatsächlich generiert werden müssen. Daher ist eine dynamische Lastbalance zwischen den Aktivitäten notwendig. Da einem Arbeitspaket (repräsentiert durch einen Routenanfang) nicht anzusehen ist, wie lange seine Bearbeitung dauern wird, muss die bearbeitende Aktivität in der Lage sein, auf Anfrage einen Teil der ihr zugewiesenen Arbeit abzuspalten und einer anderen Aktivität zu übertragen. Eine Aktivität, die mit ihrem Teil des Suchraums fertig ist, bemüht sich danach, ein neues Arbeitspaket von einer anderen Aktivität zu erhalten, welche ihre Suche noch nicht beendet hat. Die Suche terminiert, wenn alle Arbeitspakete bearbeitet sind.

Kritisch für die Effektivität der beschriebenen Lastverteilung ist die Maximierung der Größe von abgespaltenen Arbeitspaketen. Sind die abgespaltenen Arbeitspakete zu klein, wird auch hier der Parallelisierungsgewinn durch die Zeit für Kommunikation und Synchronisation aufgewogen. Das benutzte Schema zur Abspaltung von Arbeitspaketen, ist in Abbildung 7.17 veranschaulicht: Der bearbeitenden Aktivität ist der Routenanfang $(0, 5, 7)$ zugewiesen, zu dem alle Vervollständigungen zu einer kompletten Route untersucht werden müssen. Dieser Suchraum ist durch das Gebiet unter dem Routenanfang angedeutet. Die Grenzen des Suchraumes sind unregelmäßig eingezeichnet, da nicht in allen Teilen des Suchraums die Anfangsroute zu einer kompletten Route vervollständigt werden muss, da mittels des Beschneidungskriteriums sichergestellt ist, dass dort keine neue beste Route gefunden werden kann. Das linke Gebiet ist schon vollständig durchsucht. Während die bearbeitende Aktivität das mit „bearbeiten“ gekennzeichnete Gebiet analysiert trifft eine Anfrage auf Abspaltung eines Teils der Arbeit ein. In diesem Fall wird der nächste noch unbearbeitete Routenanfang mit minimaler Länge $(0, 5, 7, 8)$ an die anfragende Aktivität abgegeben. Dies erfordert zwar eine etwas komplexere Verwaltung des noch zu durchsuchenden Restraumes, als wenn der momentan bearbeitete Routenanfang abgespalten würde, erhöht aber die Chance deutlich, dass ein hinreichend großes Arbeitspaket abgegeben wird, für das sich der Kommunikationsaufwand lohnt.

Änderungen für die verteilte Ausführung

Abbildung 7.18 zeigt die beiden zentralen Klassen des TSP-Algorithmus. Die zentrale Instanz `TSP` kapselt die momentan beste Route. Mit `updateBestTour()` wird eine neue beste Route installiert, wenn die übergebene Route kürzer als die bisherige Lösung ist. Von der Klasse `Solver` existieren mehrere Instanzen, welche den Suchprozess kapseln. Alle Suchinstanzen laufen in eigenen Kontrollfäden und melden neue kürzeste Routen an die Zentrale und benutzen die bisher beste gefundene Lösung, um

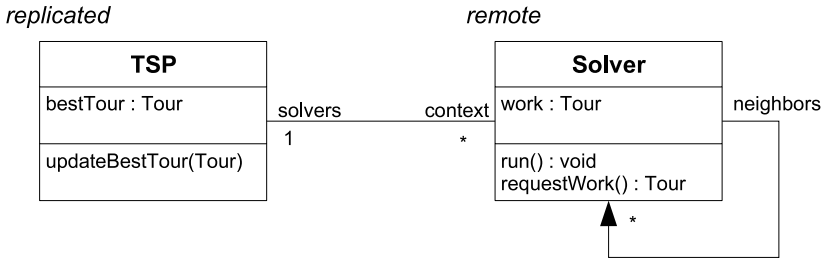


Abbildung 7.18: Annotation der zentralen Klassen des parallelen TSP-Algorithmus.

ihren eigenen Suchraum zu beschneiden. Hat ein Sucher sein Arbeitspaket abgearbeitet, fragt er über `requestWork()` reihum andere Suchprozesse nach abspaltbarer Arbeit.

Um diesen parallelen Algorithmus verteilt ausführen zu können, müssen die Suchinstanzen auf die Knoten des Rechnerbündels verteilt werden. Diese benötigen aber weiterhin schnellen lokalen Zugriff auf die gemeinsam genutzte Instanz von `TSP`, um die aktuell beste Route für die Beschneidung des verteilt abgesuchten Raumes benutzen zu können. Dies wird erreicht, indem die Klasse `Solver` mit dem Schlüsselwort `remote` und die Klasse `TSP` mit dem Schlüsselwort `replicated` markiert werden. Neben dieser Änderung ist lediglich noch die Synchronisation in der Methode `updateBestTour()` in eine exklusive Synchronisation umzuwandeln, da diese Methode den Zustand des replizierten Kontextobjektes modifiziert. Daher muss ausgeschlossen werden, dass zwei Sucher gleichzeitig versuchen eine neue kürzeste Route zu installieren.

Diese Lösung für die Verteilung schließt auch die transparente Verwendung von SMP-Knoten im Rechnerbündel ein. Auf einem SMP-Rechenknoten können entsprechend der Anzahl der installierten CPUs mehrere Suchinstanzen angelegt werden. Alle Suchinstanzen auf einem Rechenknoten haben damit direkten Zugriff auf das lokale Replikat des Kontextobjektes und können transparent entweder mit anderen Suchinstanzen auf demselben Rechner oder mit Suchinstanzen auf anderen Rechnern zwecks der Abspaltung von Arbeitspaketen kommunizieren.

Versuchsaufbau

Das Kriterium für die Suchraumbeschneidung führt nicht bei allen Probleminstanzen zu einer gleichmäßigen Reduktion der Anzahl von Routen, die untersucht werden müssen. Ist die Varianz der Entfernungen zwischen unterschiedlichen Städten groß, müssen u.U. alle Pfade durch nahegelegene Städte untersucht werden, da das Kriterium für die Suchraumbeschneidung in diesem Fall nicht (oder erst spät) anspricht. Bei 101 zufällig generierten Probleminstanzen mit 18 Städten wurde eine Streuung der Laufzeit des sequentiellen Algorithmus von 0,2 s bis über 300 s beobachtet, was einem Faktor von 1500 entspricht. Eine Probleminstanz, bei der das Kriterium für die Suchraumbeschneidung zu einer stärkeren Reduktion der untersuchten Routen und damit zu einer kürzeren Laufzeit führt, heißt im folgenden ein einfaches Problem. Für die Evaluation der Skalierbarkeit der parallelen Version des Algorithmus wird sowohl ein einfaches Problem mit mehr Städten und ein schwieriges Problem mit weniger Städten verwen-

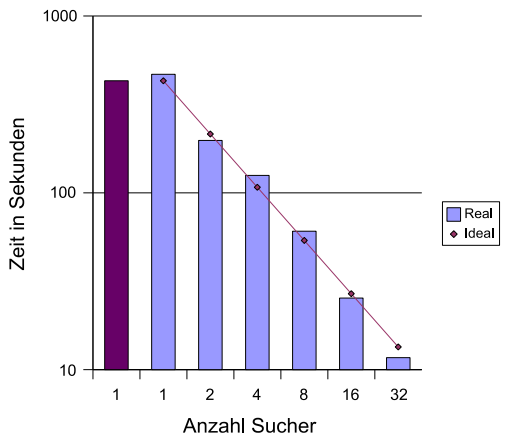


Abbildung 7.19: Laufzeiten der sequentiellen und parallelen Version für ein *einfaches* Problem mit 22 Städten.

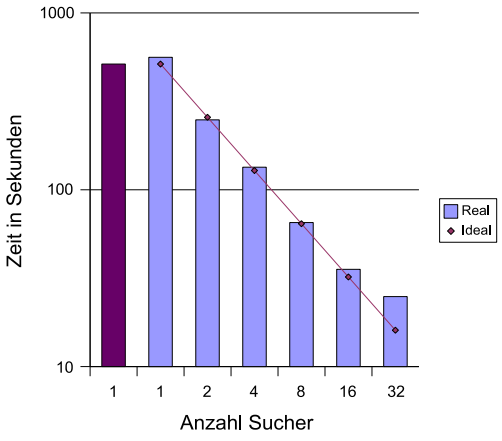


Abbildung 7.20: Laufzeiten der sequentiellen und parallelen Version für ein *schwieriges* Problem mit 19 Städten.

det. Die Anzahl der Städte wurde so gewählt, dass die sequentielle Version für die Lösung beider Probleme ungefähr gleich lange benötigt.

Messergebnisse

Die Abbildungen 7.19 und 7.20 zeigen Laufzeiten der sequentiellen Version des Algorithmus (die jeweils erste Säule) im Vergleich zu der parallelen Version mit unterschiedlich vielen Suchaktivitäten (die jeweils folgenden Säulen). Beide Abbildungen haben eine logarithmische Skala an der Y-Achse. Abbildung 7.19 zeigt die Zeiten für die Lösung eines einfachen und Abbildung 7.20 die eines schwierigen Problems. In beiden Graphiken erkennt man den Mehraufwand der parallelen Version durch Vergleich der ersten beiden Säulen. In beiden Fällen benötigt die parallele Version auf einem einzelnen Prozessor ca. 9 % mehr Zeit als die sequentielle Version. Dies ist ein sehr gutes Ergebnis und zeigt die Effektivität der Optimierungen für den transparenten lokalen Zugriff auf replizierte und potenziell entfernte Objekte. Eine genauere Betrachtung wird im nächsten Abschnitt bei einem Vergleich mit der RMI-Version desselben Programms angestellt. In beiden Abbildungen sind die optimal erwarteten Zeiten mit einer die Säulen verbindenden Linie eingezeichnet. Man erkennt, dass die parallele Version gut skaliert. Insbesondere beim Übergang von einem auf zwei Suchaktivitäten tritt eine Laufzeitreduktion ein, die größer ist als die eingezeichnete Optimalkurve erwarten ließe. Dieser Synergieeffekt kann damit erklärt werden, dass sich bei der parallelen Suche die Suchreihenfolge ändert. Dadurch kann es vorkommen, dass eine gute Lösung früher gefunden wird. Aufgrund der Abhängigkeit der Suchraumbeschneidung von der aktuell besten Lösung verringert sich die Größe des insgesamt zu durchsuchenden Raumes. Die Laufzeit der parallelen Version (mit zwei Suchern) kann dadurch unter die Hälfte der Laufzeit der sequentiellen Version fallen. Abbildung 7.19 zeigt weiterhin, dass der Algorithmus bis 32 Suchaktivitäten gut mit einer um eins schwankenden Effizienz skaliert. Bei dem in Abbildung 7.20 untersuchten schwierigen Problem wird ebenfalls eine nahezu perfekte Skalierung bis acht Suchaktivitäten

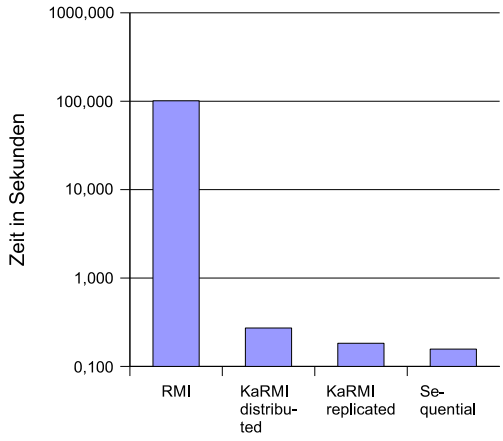


Abbildung 7.21: Laufzeiten für ein einfaches Problem mit nur 12 Städten. Vergleich unterschiedlicher Programmvarianten.

erreicht, danach fällt die Effizienz auf ca. 65 % bei 32 Suchaktivitäten.

Obwohl die Suche nach der kürzesten Wegstrecke sehr gut parallelisierbar erscheint, ist eine gute Skalierung bei einer großen Anzahl von Suchaktivitäten keineswegs selbstverständlich. Zwar scheinen die Suchaktivitäten auf den ersten Blick bis auf den gemeinsamen Zugriff auf die momentan kürzeste bekannte Route unabhängig voneinander arbeiten zu können. Dies ist aber nicht der Fall, da eine weitere Kopplung über die Zuteilung der Suchgebiete an die Aktivitäten existiert. Diese Aufteilung des Suchraums entscheidet über die Ausgewogenheit der Lastbalance und damit über die Güte der Skalierung. Je ausgefeilter aber das Kriterium für die Beschneidung des Suchraumes ist, desto mehr hängt die Effizienz der verteilten Suche an der schnellen Aktualisierung des Beschneidungskriteriums und desto unvorhersagbarer ist die von einer Suchaktivität tatsächlich aufzuwendenden Arbeit für ein ihr zugeteiltes Gebiet im Suchraum. Die Aufteilung des Suchraums an die Suchaktivitäten kann daher nicht statisch beim Programmstart erfolgen, sondern muss dynamisch angepasst werden. Diese dynamische Lastbalance erfordert zusätzliche Kommunikation zwischen den Suchaktivitäten. Bei dem untersuchten schwierigen Problem ist die Anzahl von zu untersuchenden Routen für Weganfänge mit gleich vielen Städten sehr unterschiedlich, da bei manchen Weganfängen sehr früh und bei anderen erst sehr spät abgebrochen werden kann. Der implementierte Lastverteilungsalgorithmus kommt damit nicht gut zurecht, und das führt dazu, dass viele zu kleine Suchaufträge an andere Aktivitäten abgegeben werden, aber große Teilbereiche von einer einzelnen Aktivität untersucht werden.

Abbildung 7.21 vergleicht die Laufzeiten verschiedener paralleler Programmvarianten bei der Ausführung auf nur einem Rechenknoten gegenüber der sequentiellen Version. Damit wird die Effektivität von Optimierungen für den transparenten Zugriff auf replizierte und potenziell entfernte Objekte evaluiert. Als Vergleichsmaßstab dient eine Programmvariante, welche die Java-Standard-Serialisierung und RMI verwenden (erste Säule von links). Bei der zweiten Säule werden die Optimierungen in der Serialisierung aus Kapitel 4 und die Optimierungen für den transparenten Fernzugriff aus Kapitel 5 verwendet. Die beiden ersten Varianten verwenden keine Objektreplikation, sondern verwalten die aktuell beste Route in einem entfernten Objekt. Da alle Versio-

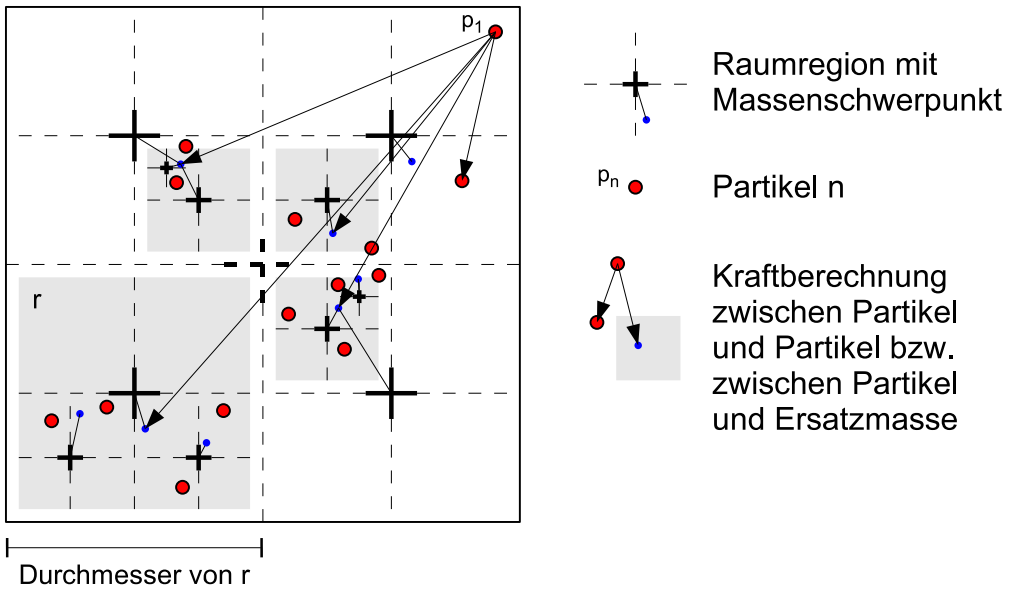


Abbildung 7.22: Aufteilung der Ebene in Regionen. Kraftberechnung zwischen Partikel und Partikel oder zwischen Partikel und Ersatzmasse einer Region.

nen nur auf einem einzelnen Rechenknoten ablaufen, finden alle potenziellen Fernzugriffe lokal statt. Die Variante mit KaRMI ohne Replikation ist bereits mehr als 600 mal schneller als die RMI-Variante, aber immer noch ca. 70 % langsamer als die sequentielle Version ganz rechts. Dies rührt von dem indirekten Zugriff auf entfernte Objekte über deren Stellvertreter her. Erst dadurch, dass replizierte Objekte in der vorliegenden Arbeit einen direkten Zugriff ohne Stellvertreterobjekte ermöglichen, kommt die Variante mit KaRMI und Replikation bis auf 17 % an die sequentielle Version heran. Wie oben gesehen verbessert sich dieses Verhältnis noch bei größeren Problemen. Die Werte zeigen, dass die vorgenommenen Optimierungen beim transparenten Zugriff absolut notwendig sind, um eine gute Effizienz gemessen an der sequentiellen Version zu erzielen. Nur mit effizientem transparentem Zugriff ist es möglich, eine Anwendung einfach in Komponenten zu zerlegen, die eine verteilte Ausführung ermöglichen. Mit der vorliegenden Arbeit ist diese Zerlegung allein durch die Annotation von Klassen mit den Schlüsselworten `remote` und `replicated` möglich. Dadurch wird die transparente Verwendung der Komponenten in einer verteilten Umgebung ermöglicht, ohne die lokale Interaktion mit zu großem Mehraufwand zu behindern.

7.3.2 N-Body

Die N-Körper-Simulation approximiert die Bewegung aufgrund wechselseitiger Massenanziehung von n Partikeln im Raum. Die Simulation ist in Zeitschritte diskretisiert, wobei in jedem Zeitschritt (theoretisch) die Anziehungskraft zwischen jedem der n Partikel und jedem anderen Partikel berechnet werden muss. Anschließend wird für jedes Partikel die Beschleunigung und Veränderung von Geschwindigkeit und Ortskoordinaten ausgerechnet. Zur Reduktion der quadratischen Komplexität bei der Kräfteberechnung benutzt der hier verwendete Benchmark die von Barnes und Hut vorgeschla-

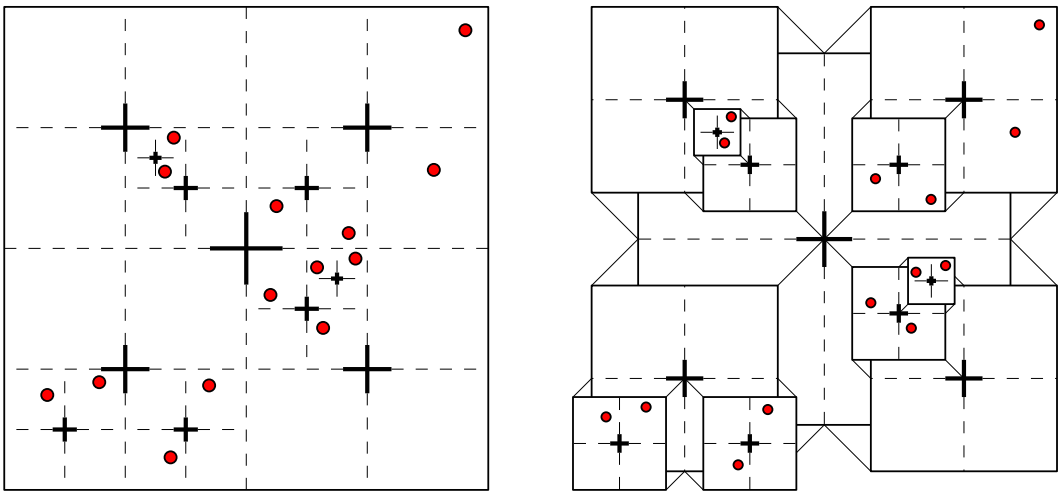


Abbildung 7.23: Explosionszeichnung der verwendeten Quartärbaum-Datenstruktur.

gene Approximation [9], welche durch hierarchische Gruppierung und Ersatzmassen-Bildung für entfernte Raumregionen zu einer Reduktion auf $O(n \log(n))$ Rechenoperationen je Zeitschritt führt.

Das Hauptaugenmerk bei diesem Benchmark liegt auf der direkten Parallelisierung und Verteilung durch den Einsatz kollektiv replizierter Objekte. Als Ausgangspunkt wird eine direkte objektorientierte Umsetzung des Algorithmus von Barnes und Hut verwendet, die im folgenden beschrieben wird. Abbildung 7.22 zeigt die Optimierungsidee bei der Kräfteberechnung von Barnes und Hut für den zweidimensionalen Fall. Der Raum ist soweit hierarchisch in quadratische Regionen unterteilt, dass sich in jeder Region höchstens ein Partikel befindet. Wenn das Verhältnis aus dem Durchmesser einer Region und der Entfernung eines Partikels p_n vom Massenschwerpunkt der Region einen Schwellwert θ unterschreitet, so darf die Kräfteberechnung zwischen dem Partikel p_n und dem Massenschwerpunkt der Region durchgeführt werden, anstatt einzelne Kräfteberechnungen mit den Partikeln dieser Region durchführen zu müssen. Der Parameter θ beeinflusst dabei die Genauigkeit der Simulation. In Abbildung 7.22 sind die Regionen grau eingefärbt, welche bei der Kräfteberechnung für das Partikel p_1 durch ihre Ersatzmasse substituiert werden dürfen (bei $\theta = 0,5$). Bei den insgesamt 14 Partikeln in Abbildung 7.22 reduzieren sich damit die Kräfteberechnungen für Partikel p_1 von ursprünglich 13 (bei ausschließlicher Berücksichtigung von Partikel-Partikel-Interaktionen) auf 5 Berechnungen.

Die Partikel der Simulation sind in einem Quartärbaum organisiert, um schnell die Raumregionen identifizieren zu können, welche bei der Kräfteberechnung zusammengefasst werden können. Abbildung 7.23 zeigt die Korrespondenz der Raumeinteilung und der Baumstruktur. Die Wurzel des Baumes (die tiefste Ebene in der Explosionszeichnung) repräsentiert den kompletten untersuchten Bereich. Ein Baumknoten ist entweder ein Blattknoten, welcher höchstens ein Partikel enthält, oder er hat vier Kinder, welche die vier Quadranten des von ihm aufgespannten Raumes repräsentierten.

Für jedes Partikel müssen pro Zeitschritt identische Operationen durchgeführt werden (Berechnung von Kräften, Beschleunigung, Geschwindigkeits- und Ortsveränderung). Zur Parallelisierung bietet sich daher ein datenparalleles Vorgehen an, bei dem

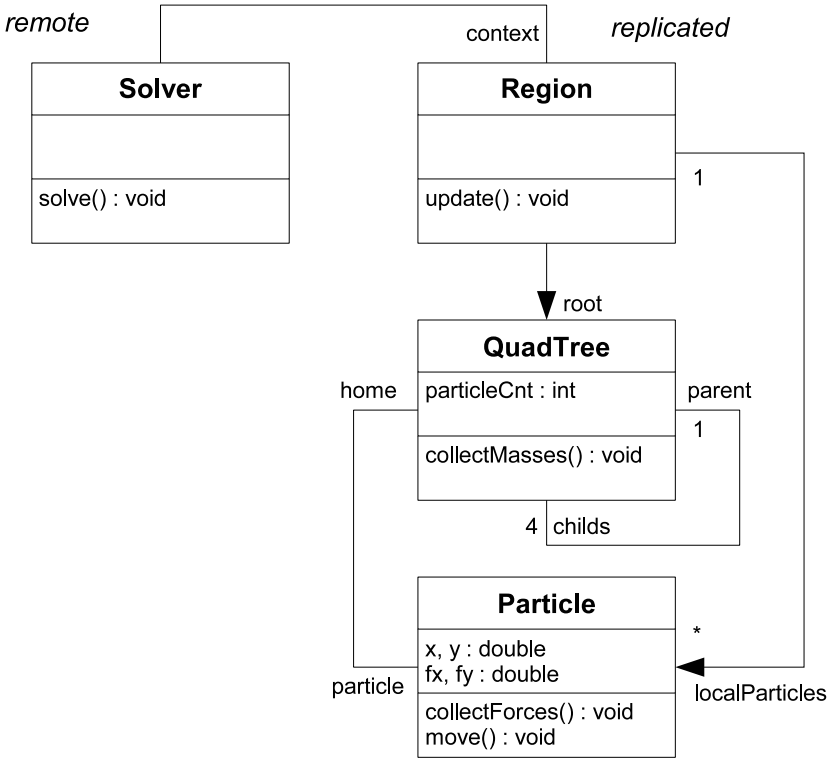


Abbildung 7.24: Programmstruktur des sequentiellen und parallelen Barnes-Hut-Algorithmus.

die Partikel (die Daten) zur Berechnung auf unterschiedliche Rechenknoten aufgeteilt werden. Neben den ihm zugeordneten Partikeln benötigt jeder Rechenknoten mindestens den Teil des Quartärbaumes, der alle auf ihm berechneten Partikel enthält. Zusätzlich werden auf einem Rechenknoten auch noch die Teile des Baumes für entfernte Regionen benötigt, um Kräfteberechnungen zwischen den Partikeln des Rechenknotens und den gemäß obiger Formel bestimmten Ersatzmassen durchführen zu können. Der Quartärbaum ist damit Teil der Daten bei der datenparallelen Operation. Der Baum wird dabei keinesfalls nur lesend verwendet. Nachdem die Ortskoordinaten der Partikel am Ende eines Zeitschrittes verändert wurden, müssen Baumknoten umgebaut werden, wenn Partikel die Region verlassen haben, in der sie vor ihrer Bewegung gespeichert waren.

Durch den Einsatz kollektiver Replikation lässt sich der datenparallele Ansatz direkt auf die Baumstruktur übertragen, ohne die Struktur des sequentiellen Algorithmus zu ändern. Das Klassendiagramm in Abbildung 7.24 zeigt die Programmstruktur sowohl für die sequentielle als auch die verteilt parallele Lösung. Ein `Solver`-Objekt (in der parallelen Version sind es mehrere) arbeitet an einem `Region`-Objekt, das eine Fassade für die in der Quartärbaumstruktur organisierten Partikeldaten darstellt. Die Berechnung eines Zeitschrittes findet in der `update()`-Methode von `Region` statt. Durch Annotation der `Solver`-Klasse mit `remote` und der Klasse `Region` mit `replicated` lassen sich mehrere `Solver`-Objekte auf unterschiedliche Rechenknoten so verteilen, dass sie alle eine Referenz auf das zentrale replizierte Datenobjekt `Region` halten. Jedes `Solver`-Objekt arbeitet aufgrund der Annotation

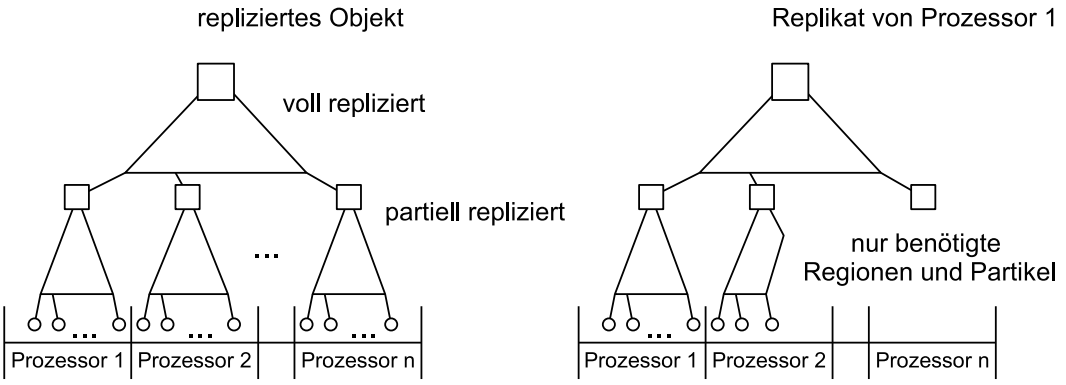


Abbildung 7.25: Gesamtsicht des replizierten Zustandes des Barnes-Hut-Algorithmus (links) und das entsprechende Replikat auf Rechenknoten 1 (rechts).

`replicated` an der Klasse `Region` an einem lokalen Replikat der Quartärbaumstruktur. Durch direkten Zugriff auf die entsprechenden lokalen Replikate können alle `Solver`-Objekte unabhängig voneinander ihren Rechenschritt durchführen. In der parallelen Version des Algorithmus verwendet die `update()`-Methode eine kollektive Synchronisation am `Region`-Objekt, um die Änderungen am gemeinsamen Zustand auszutauschen und in einen neuen konsistenten Zustand zu verschmelzen.

Durch Verteilungshinweise bei der Erzeugung von Baumknoten wird in der parallelen Version erreicht, dass sie nur dort repliziert werden, wo sie für die Berechnungen an Partikeln benötigt werden. Ein Rechenknoten benötigt alle diejenigen Baumknoten für seine lokalen Berechnungen, welche die ihm zugewiesenen Partikel speichern, und alle deren Vorgänger im Baum. Zusätzlich werden Knoten für die lokale Berechnung benötigt, die Regionen repräsentieren, welche so weit von den lokal berechneten Partikeln entfernt sind, dass sie durch ihre Ersatzmassen substituiert werden können. Für die Partikel-Objekte sind keine Verteilungshinweise notwendig. Der hier besprochene Benchmark nimmt darüberhinaus noch die folgenden Vereinfachungen vor:

- Der Raum ist statisch unter den Rechenknoten aufgeteilt, so dass für alle Partikel aufgrund der Ortskoordinaten feststeht, welcher Rechenknoten für ihre Berechnung zuständig ist.
- Die Verteilungshinweise für Baumknoten richten sich nicht nach der aktuellen Position der Partikel, sondern verwenden eine Abschätzung des schlimmsten Falles für die Entscheidung, ob ein räumlich entfernter Baumknoten im lokalen Replikat zur Berechnung benötigt wird. Dabei wird davon ausgegangen, dass sich überall in den einem Rechenknoten zugewiesenen Raumregionen ein Partikel befinden könnte.

Mit obigen Vereinfachungen müssen die Knoten des Quartärbaumes nie umverteilt werden. Allerdings passt der Algorithmus die Lastverteilung nicht an eine Änderung der Verteilung der Partikel im Raum an. Durch die Annahme des schlimmsten Falles bei der Berechnung der Verteilungshinweise werden außerdem möglicherweise zu viele Baumknoten repliziert, was zu einem erhöhten Aufwand bei der Zustandsfortschreibung führt. Dafür müssen für die Partikel-Objekte keine Verteilungshinweise

gegeben werden. Für sie gilt Replikation bei Bedarf. Damit wird ein Partikel-Objekt überall dahin repliziert, wo es von einem Baumknoten referenziert wird. Dies ist immer auf dem Rechenknoten der Fall, welchem die Raumregion zugeteilt ist, in der sich das betreffende Partikel momentan befindet. Zusätzlich kann ein Partikel-Objekt auch auf Rechenknoten repliziert werden, welche räumlich angrenzende Regionen bearbeiten, wenn dort ein Ast des Quartärbaumes bis zu seiner untersten Ebene für die Kräfteberechnung benötigt wird. Verlässt ein Partikel eine Region, von der mehrere Rechenknoten Replikate besitzen, verschwinden auf diesen anderen Rechenknoten die Referenzen auf das Partikel. Dies hat zur Folge, dass seine Kopie auf denjenigen Rechenknoten vom Speicherbereiniger abgeräumt wird, auf denen es nicht länger benötigt wird.

Abbildung 7.25 zeigt auf der linken Seite die Gesamtsicht auf die verteilte Datenstruktur des Algorithmus. Allerdings ist diese Gesamtsicht auf keinem der Rechenknoten tatsächlich realisiert. Stattdessen besitzt jeder Rechenknoten nur eine partielle Sicht auf den gesamten Quartärbaum. Eine solche lokale Sicht von Rechenknoten 1 ist auf der rechten Seite von Abbildung 7.25 dargestellt. Rechenknoten 1 hält den oder die Teilbäume, welche die ihm zugeteilten Raumregionen mit den darin befindlichen Partikeln repräsentieren. Zusätzlich hat er Kopien von Baumknoten und Partikeln, welche auf anderen Knoten berechnet werden, aber deren Massen für die Kräfteberechnungen an den Partikeln von Rechenknoten 1 benötigt werden. Ein einzelner Baumknoten oder ein Partikel-Objekt ist daher entweder auf mehreren Rechenknoten repliziert oder nur lokal auf einem einzelnen Rechenknoten gespeichert, je nachdem wo es benötigt wird. Insgesamt resultiert daraus zur Laufzeit des Programms eine komplexe Datenstruktur eines teilweise replizierten und teilweise verteilten Baumes. Die gesamte Komplexität dieser Struktur ist jedoch hinter der Fassade des kollektiv replizierten Objektes vor dem Programmierer versteckt. Der zur Realisierung notwendige Verteilungscode wird automatisch durch die Transformation aufgrund der Annotation der Klasse `Region` erzeugt. Die Struktur des sequentiellen Programms muss für die verteilt parallele Ausführung nicht geändert werden. Die einzigen notwendigen Änderungen sind die Annotation der Klassen `Solver` und `Region`, die Deklaration der Methode `update()` als `collective synchronized` und das Einfügen von Verteilungshinweisen bei der Erzeugung von Baumknoten. Diese Verteilungshinweise leiten sich direkt aus der Berechnungsvorschrift für die zu berücksichtigenden Kräfte auf Partikel.

Abbildung 7.26 zeigt die Laufzeiten der sequentiellen und parallelen Version der N-Körpersimulation auf einem bis 16 Rechenknoten. In dem untersuchten Lauf werden 100 000 Partikel simuliert. Die Laufzeiten sind für einen Simulationszeitschritt in Sekunden angegeben. Der Übergang von der sequentiellen zur parallelen Version auf einem Rechenknoten ist mit keinen Mehrkosten verbunden. In diesem Fall liegt das daran, dass auf den replizierten Quartärbaum der parallelen Version ohne Mehrkosten gegenüber seinem Pendant der sequentiellen Version zugegriffen werden kann. Die in der parallelen Version zusätzlich notwendige (kollektive) Synchronisation reduziert sich bei Verwendung nur eines Rechenknotens auf reguläre Java-Synchronisation und fällt im Vergleich zur Laufzeit des Rechenschrittes nicht ins Gewicht. Das relativiert sich beim Übergang auf zwei Rechenknoten. Hier wird der Mehraufwand durch die Synchronisation und die damit verbundene Zustandsfortschreibung als deutlicher Abstand zur idealerweise erwarteten Laufzeit sichtbar. Bei zwei Rechenknoten wird

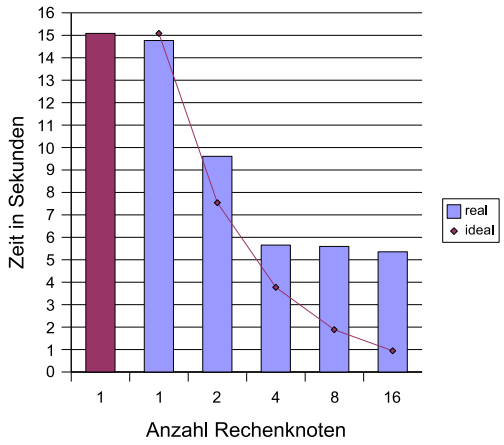


Abbildung 7.26: Laufzeiten eines Simulationszeitschrittes der N-Körper-Simulation in der sequentiellen und parallelen Version.

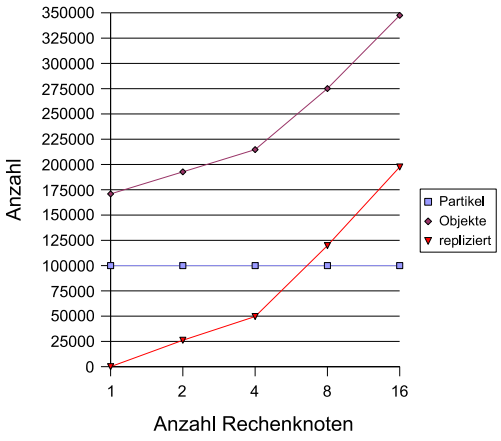


Abbildung 7.27: Anzahl von Teilobjekten des replizierten Quartärbaumes bei Verwendung unterschiedlich vieler Rechenknoten.

eine Effizienz von 79 % und bei vier Rechenknoten von nur noch 67 % erreicht. Bei Hinzunahme von weiteren Rechenknoten wird keine weitere Geschwindigkeitssteigerung erreicht. Eine Begründung für dieses Verhalten findet sich in Abbildung 7.27. In dieser Abbildung ist die gesamte für die Realisierung des Quartärbaumes notwendige Objektanzahl gegenüber der Anzahl replizierter Objekte aufgetragen. Diese Anzahlen sind die Summen der Objektanzahlen über alle beteiligten Rechenknoten. Von den als repliziert ausgewiesenen Objekten existiert eine Kopie auf mehr als einem Rechenknoten. Solche Teilzustände des replizierten Quartärbaumes sind also tatsächlich repliziert, weswegen für sie eine Zustandsfortschreibung am Ende jeder Synchronisation notwendig wird. Man erkennt, dass die Anzahl der tatsächlich replizierten Teilzustände bei wachsender Anzahl von Rechenknoten überproportional wächst. Bei 16 Rechenknoten ist der Anteil der tatsächlich replizierten Objekte bereits auf über die Hälfte aller Objekte angestiegen und der Aufwand für die Zustandsfortschreibung dominiert den Rechenaufwand pro Zeitschritt.

Grund für die schlechte Skalierung im Beispiel liegt vor allem an dem zu stark vereinfachenden Vorgehen bei der Aufteilung des Problems auf die Rechenknoten. Wie in Abbildung 7.25 zu sehen, wird das von seiner Struktur her hierarchische Problem flach auf die Rechenknoten verteilt. Dies hat zur Folge, dass zum einen ein sehr großer Teil des Zustandes tatsächlich repliziert werden muss, da die Abhängigkeiten zwischen den Teilzuständen nicht berücksichtigt werden, und zum anderen dass die Struktur der resultierenden Überlappungen von gemeinsamen Objekten in den Replikaten besonders ungünstig ist. Aufgrund dieser Aufteilung hat jeder Rechenknoten gemeinsame Objekte mit jedem anderen Rechenknoten. Demzufolge ist die in Abschnitt 6.2.4 beschriebene Optimierung der kollektiven Zustandsfortschreibung durch Färben des Replikatgraphen nicht effektiv. Aufgrund der Vollständigkeit des Replikatgraphen entsteht daher zwangsläufig immer eine Färbung mit einer Farbanzahl gleich der Anzahl der beteiligten Replikate. Deswegen findet die kollektive Zustandsfortschreibung sequentiell statt, und das führt wie in Abschnitt 7.2.9 vermessen zu einem in der Anzahl der beteiligten

Replikate linearen Anstieg der Synchronisationszeiten.

Aus der verteilten Parallelisierung des N-Körper-Problems kann dennoch das Fazit gezogen werden, dass eine gute Skalierung möglich ist, wenn die Partitionierung des replizierten Zustandes die Anzahl der tatsächlich replizierten Objekte klein hält. Dazu muss gegebenenfalls Arbeit in die geeignete Parallelisierung des Problems investiert werden. Der Codieraufwand für die verteilte Ausführung aber ist durch den Einsatz kollektiver Replikation bei der datenparallelen Verarbeitung komplexer Strukturen sogar vernachlässigbar gering.

7.3.3 Impact

Impact ist eine Finite-Elemente-Simulationsumgebung für dynamische Phänomene wie die Verformung von Materialien. Impact ist in Java geschrieben und konsequent objektorientiert strukturiert. Impact ist ein quelloffenes Projekt, das unabhängig von der vorliegenden Arbeit bei SourceForge entwickelt wird.[1] Die Parallelisierung von Impact stellt daher einen echten Anwendungsbenchmark dar.

Das Modell des untersuchten dreidimensionalen Werkstücks ist in Impact als Objektgraph, bestehend aus Element- und Knotenobjekten, repräsentiert. Ein Elementobjekt beschreibt dabei die Finite-Elemente-Näherung für ein Raum- oder Flächenstück des untersuchten Gegenstandes. Elementobjekte tragen Eigenschaften wie Material, Elastizität oder Festigkeit. Es existieren unterschiedliche Klassen von Elementobjekten, die es ermöglichen, ein Modell aus unterschiedlich geformten Basiselementen zusammenzusetzen. Elementobjekte sind untereinander durch Knotenobjekte verbunden, die Massepunkte im Raum repräsentieren. Über gemeinsame Knotenobjekte interagieren angrenzende Elemente. Knotenobjekte sammeln Kräfte, die von angrenzenden Elementen auf sie ausgeübt werden. Resultierende Kräfte an Knotenobjekten führen zur deren Beschleunigung, Ortsveränderung und damit zu einer Verschiebung oder Verformung der an sie anstoßenden Elemente, wodurch sich möglicherweise die auf die Knotenobjekte ausgeübten Kräfte ändern.

Parallelisierung

Das Ausgangsprogramm ist nicht parallelisiert. Zur Motivation der Parallelisierungsidee dient die Darstellung des algorithmischen Kerns von Impact. Die Berechnung der Modelldynamik läuft in Pseudocode wie folgt ab:

for all Zeitschritt **do**
  for all Element **do**
    Berechne Spannungen, Verformungen und die daraus resultierenden Kräfte.
    Propagiere die Kräfte an die Knoten des Elements.
  for all Knoten **do**
    Aus der resultierenden Kraft berechne Beschleunigung, Geschwindigkeits- und Positionsveränderung.

Die äußere Schleife ist nicht parallelisierbar, da der folgende Zeitschritt direkt von der Berechnung des letzten Zeitschritts abhängt. Innerhalb eines Zeitschritts wird über

[1] Siehe `http://impact.sourceforge.net/`

die Elemente und Knoten des Modells iteriert. Diese beiden Schleifen sind datenparallel formulierbar. Nur die erste Schleife (die Iteration über die Elemente) hat Datenabhängigkeiten (beim Propagieren der Kräfte in die Knotenobjekte). Aufgrund der Kommutativität bei der Addition von Kraftvektoren lassen sich diese aber auflösen.

Die Parallelisierungsidee ist die folgende: Das Modell wird in ein repliziertes Objekt gekapselt. Von dem replizierten Objekt werden Replikate auf den verfügbaren Rechnern angelegt. Die beiden inneren Schleifen werden parallel auf diesen Rechnern gestartet, laufen dort aber jeweils nur über eine Partition der Element- bzw. Knotenobjekte. Um die Datenabhängigkeiten zwischen den Schleifen aufzulösen, werden beide jeweils in eine kollektive Synchronisation eingeschlossen. Dadurch wird jeweils am Ende der jetzt parallel ausgeführten Schleifen über die Elemente und die Knoten eine konsistente Sicht auf das Modell hergestellt. Während der Berechnung der Positionsveränderungen der Knoten stehen damit alle Kräfte bereit, auch wenn die Elemente, welche diese Kräfte ausüben auf anderen Rechenknoten berechnet wurden. Umgekehrt kann während der Berechnung des nächsten Zeitschrittes von allen Elementen wieder auf die aktuellen Positionen der Modellknoten zugegriffen werden.

Die oben erwähnte Datenabhängigkeit beim Propagieren der Kräfte von den Elementen in die Knotenobjekte wird durch die Angabe einer (additiven) Verschmelzungsoperation bei den Knotenobjekten aufgelöst. Werden während der ersten kollektiven Synchronisation mehrere Replikate desselben Knotenobjektes modifiziert – weil Elementobjekte, die an dieses Knotenobjekt angrenzen, in verschiedenen Partitionen und damit auf verschiedenen Rechenknoten berechnet werden – so werden diese nebenläufigen Modifikationen während der kollektiven Zustandsfortschreibung zu einer konsistenten Gesamtänderung verschmolzen.

Über Verteilungshinweise kann das replizierte Modellobjekt entsprechend der für die Parallelisierung der Schleifen durchgeführten Partitionierung der Menge der Elementobjekte ebenfalls partitioniert werden. Damit wird verhindert, dass das gesamte Modell auf allen Rechenknoten geladen und bei jeder Synchronisation aktualisiert werden muss. Jedes Elementobjekt muss nur auf demjenigen Rechenknoten vorrätig gehalten werden, auf dem es berechnet wird, da weder die Berechnung von Elementobjekten noch die von Knotenobjekten Zugriff auf andere Elementobjekte benötigen. Knotenobjekte müssen dagegen auf alle Rechenknoten repliziert werden, auf welchen angrenzende Elemente berechnet werden, da über sie das Propagieren der ausgeübten Kräfte abläuft. Die Partitionierung der Menge der Elementobjekte hat somit einen direkten Einfluss auf den Replikationsgrad der Knotenobjekte. Ein Knotenobjekt muss immer dann repliziert werden, wenn angrenzende Elementobjekte unterschiedlichen Partitionen zugeschlagen werden. Eine gute Partitionierung minimiert die Anzahl der Knotenobjekte, welche sich derart an Partitionsgrenzen befinden.

Programmtechnisch muss für die verteilte Parallelisierung die Modellklasse repliziert deklariert und der algorithmische Kern in eine entfernte Klasse verlagert werden, um ihn parallel auf verschiedenen Rechenknoten ablaufen lassen zu können. Beides sind rein mechanische Änderungen, die keine Umstrukturierungen des Lösungsalgorithmus erfordern. Schwieriger ist es, eine geeignete Partitionierung der Elementobjekte zu finden, so dass die Kommunikation während der kollektiven Zustandsfortschreibung minimiert wird. Hierbei bietet das System bisher keine Unterstützung an. Um eine solche Partitionierung für allgemeine Modelle zu finden, müsste zusätzlich

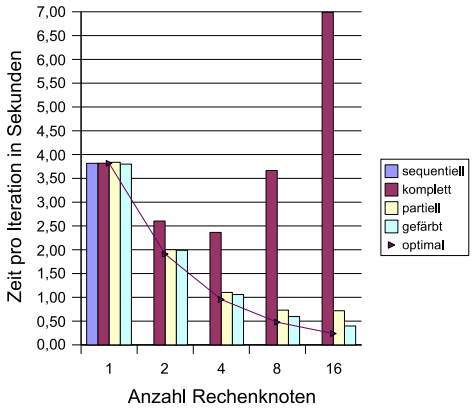


Abbildung 7.28: Laufzeiten von Impact-Varianten mit unterschiedlichen Optimierungsstufen.

ein Graph-Partitionierungsalgorithmus implementiert werden. Dies ist in Impact nicht realisiert. Stattdessen geht die Anwendung davon aus, dass das Modell bereits in einer geeigneten Partitionierung vorliegt.

Skalierungsmessungen

Abbildung 7.28 zeigt Messergebnisse an einem Modell mit 2825 Knoten und 5574 Elementen. Simuliert werden die Deformationen, die beim Stauchen eines Holzylinders auftreten. Wieder wird die sequentielle Programmversion (erste Säule) der parallelen Version mit unterschiedlichen Optimierungstufen gegenübergestellt. Bei den Messungen für die jeweils zweite Säule[2] ist das Modell voll repliziert. Bei der dritten Säule wird partielle Replikation verwendet, allerdings ohne die Optimierung der kollektiven Zustandsfortschreibung zu benutzen. Die jeweils letzte Säule verwendet alle Optimierungen. In diesem Fall ist das Modell partiell repliziert und der Prozess der Zustandsfortschreibung wird über Färbung des Replikatgraphen optimiert. Zur Orientierung ist die optimale Skalierung als durchgezogene Linie in das Diagramm eingezeichnet.

Man erkennt, dass der Mehraufwand durch die Parallelisierung vernachlässigbar gering ist. Während die komplette Replikation bei Verwendung von nur zwei Rechenknoten noch akzeptable Ergebnisse liefert (Effizienz von 73 %), lässt sich bei Erhöhung der Anzahl von Rechenknoten bei kompletter Replikation kein nennenswerter Laufzeitgewinn mehr erzielen. Partielle Replikation verbessert die Situation erheblich. Bis vier Rechenknoten wird eine gute (86 %) und bei acht Rechenknoten immerhin noch eine akzeptable Effizienz von (65 %) erreicht. Die Färbung des Replikatgraphen kann darüberhinaus noch eine weitere Verbesserung erzielen, so dass bis acht Rechenknoten eine gute (80 %) Effizienz erreicht wird. Eine weitere Verdopplung der Anzahl von Rechenknoten kann mit der Optimierung der kollektiven Zustandsfortschreibung noch in eine Geschwindigkeitssteigerung umgesetzt werden, allerdings fällt die Effizienz auf 60 %.

[2]Die erste Säule für die sequentielle Version ist nur bei Rechenknotenanzahl eins vorhanden.

Kapitel 8

Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde das weit verbreitete Java-Programmiermodell für die verteilt parallele Programmierung von Rechnerbündeln erweitert. Die Erweiterung ist nahtlos: Zum einen werden bereits bekannte Sprachkonstrukte mit Semantik für die verteilte Umgebung angereichert und zum anderen können bestehende Bibliotheken, die nicht verteilungsrelevante Aspekte betreffen, unverändert weiterverwendet werden. Klassen werden durch Annotation entweder zu transparent entfernten oder transparent replizierten Klassen, je nachdem welche Zugriffseigenschaften in der verteilten Umgebung erwünscht sind. Die Modifikation der Zugriffseigenschaften ist transparent für die sonstige Verwendung der Klassen und deren Instanzen. Eine entfernte oder replizierte Klasse wird somit genauso instanziert und verwendet, wie eine reguläre Klasse auch. Die Portierung der in Abschnitt 7.3 besprochenen Beispielanwendungen hat gezeigt, dass mit den zur Verfügung stehenden Mitteln eine einfache Portierung einer parallelen nicht verteilten Anwendung für die verteilte Umgebung eines Bündel-Parallelrechners möglich ist. Mit Impact konnte sogar eine bestehende Anwendung, die unabhängig von der vorliegenden Arbeit entwickelt wurde, ohne Eingriff in ihre bestehende Struktur direkt auf ein Rechnerbündel portiert werden. Mittlerweile existiert von Impact eine von seinen Autoren mit den Mitteln der vorliegenden Arbeit für Rechnerbündel parallelisierte Variante. Dies zeigt, dass die zur Verfügung gestellten Konstrukte einfach genug zu handhaben sind, um ohne besondere Vorkenntnisse bezüglich verteilt parallelen Rechnens von Experten der Anwendungsdomäne eingesetzt werden zu können.

Das in der vorliegenden Arbeit entwickelte Konzept der kollektiven Replikation ermöglicht nicht nur eine nahtlose Verbindung von Kontroll- und Datenparallelität, sondern macht datenparalleles Programmieren auf komplexen, oft irregulären, objektorientierten Datenstrukturen erst möglich. Kontroll- und Datenparallelität wird verbunden, nicht indem eine mit kontrollparallelen Konstrukten unverträgliche zusätzliche Form der Aktivitätserzeugung bereitgestellt wird, sondern indem reguläre kontrollparallele Aktivitäten zeitlich begrenzt in einer kollektiven Synchronisationsoperation zur Durchführung der datenparallelen Berechnung zusammengeschaltet werden. Damit behält die Anwendung einerseits die volle Kontrolle über die von ihr erzeugten Aktivitäten, andererseits bleiben die Mechanismen der erweiterten Sprache voll kompatibel zu den bereits in der Basissprache Java bereitstehenden Konzepten für Erzeugung und Kontrolle paralleler Aktivitäten. Kollektive Replikation ermöglicht, wie in

den Anwendungsbeispielen der N-Körper-Berechnung und Impact gesehen, nicht nur eine Portierung einer parallelen Anwendung für eine verteilte Umgebung, sondern vereinfacht sogar die Parallelisierung selbst. Mit kollektiver Replikation kann ohne Eingriff in die Datenstrukturen der Anwendung eine Parallelisierung vorgenommen werden, wenn sich das Problem datenparallel zerlegen lässt. Dazu wird die Datenstruktur, an der die Berechnung stattfinden soll, auf den Rechenknoten (partiell) repliziert, dort durch gemeinsam arbeitende Aktivitäten parallel modifiziert und die durchgeführten Änderungen in einer kollektiven Zustandsfortschreibung zu einem konsistenten Zustand verschmolzen.

Die gesamte in der vorliegenden Arbeit entwickelte Programmierumgebung ist plattformunabhängig in Java realisiert. Die Spracherweiterungen werden durch Programmtransformationen nach Java übersetzt, wodurch die mit ihnen formulierten Programme in allen Rechnerbündeln ausführbar sind, auf denen eine virtuelle Java-Maschine zur Verfügung steht. Die Evaluation der Anwendungen hat gezeigt, dass trotz dieses für rechenintensive Anwendungen unkonventionellen Ansatzes eine gute Skalierung möglich ist.

Dass die Beschränkung auf die virtuelle Java-Maschine einer effizienten Parallelisierung nicht im Wege steht, konnte im vorigen Kapitel nachgewiesen werden. Dass diese Parallelisierung bei einem datenparallelen Problem auch dann direkt und ohne Umstrukturierung der Anwendung möglich ist, haben insbesondere die Parallelisierungen der N-Körpersimulation und Impact gezeigt. Allerdings ist für eine effiziente Ausführung der parallelisierten Anwendung eine gute Partitionierung der replizierten Objekte notwendig. Diese Partitionierung muss die Anzahl von tatsächlich replizierten Objekten minimieren, um den Aufwand bei der Änderungserkennung und Zustandsfortschreibung zu minimieren. Eine gute Partitionierung hängt in der Regel von den Anwendungsdaten ab und ist wie bei Impact schwierig zu finden. Um die Anwendungsentwicklung weiter zu vereinfachen, sollte das Laufzeitsystem Hilfestellung bei der Partitionierung von replizierten Datenstrukturen bieten. Dazu könnte ein Graph-Partitionierungsalgorithmus in das Laufzeitsystem integriert werden, welcher die Verteilung eines replizierten Objektes nach Vorgaben der Anwendung umordnet, so dass die Überlappungsbereiche zwischen Rechenknoten minimiert werden. Eine offene Frage in diesem Zusammenhang ist, wie Verteilungsanforderungen der Anwendung an das Laufzeitsystem kommuniziert und umgekehrt Informationen über die Partitionierungsentscheidungen zurückgeliefert werden können, ohne eine Offenlegung der Datenstrukturen der Anwendung zu verlangen.

Die in der vorliegenden Arbeit eingeführte kollektive Replikation deckt den Bereich der Optimierung paralleler verteilter Lesezugriffe mit ab. Zusätzlich ist kollektive Replikation ein verständliches und einfach anwendbares Ausdrucksmittel für datenparallele Algorithmen insbesondere auf irregulären objektorientierten Datenstrukturen, das eine nahtlose Verbindung von Kontroll- und Datenparallelität herstellt. Die bei der Replikation inhärente Redundanz wird bei kollektiver Replikation ausgenutzt, um Datenabhängigkeiten paralleler Aktivitäten aufzulösen und diese datenparallel arbeiten zu lassen. Daher steht diese Redundanz nicht mehr zur Verfügung, um auch noch den Aspekt der Fehlertoleranz und Ausfallsicherheit zu adressieren. Aus diesem Grund benötigt die in der vorliegenden Arbeit entworfene Programmierumgebung ein alternatives Konzept, um die laufende Anwendung vor unvorhersehbaren Ausfällen von

Bündelknoten abzuschirmen. Hierfür bietet es sich an, einen Mechanismus zu integrieren, der die Anwendung dabei unterstützt, Sicherungspunkte in ihrem Programmablauf anzulegen. Der Sicherungspunkt muss alle Zustände beinhalten, die notwendig sind, um die Anwendung nach einem unvorhergesehenen Abbruch wiederanlaufen zu lassen. Dieses Vorgehen ist insbesondere bei rechenintensiven Anwendungen der Replikation aus Gründen der Fehlertoleranz vorzuziehen, da der Zusatzaufwand über die Häufigkeit eingestellt werden kann, mit der Sicherungspunkte gesetzt werden. Gegenüber herkömmlichen Mechanismen für das Setzen von Sicherungspunkten benötigt ein Sicherungsmechanismus für die in der vorliegenden Arbeit vorgestellte Programmierumgebung aufgrund der maschinenüberspannenden Kontrollfäden und der partiell replizierten Datenstrukturen eine Gesamtsicht auf das verteilte System. Mit einer solchen Gesamtsicht wäre es möglich, die Größe des Sicherungspunktes zu minimieren, indem replizierte Teile nur auf einem Rechenknoten gesichert werden und bei Wiederanfahren des Systems von dort erneut repliziert würden. Eine weitere interessante Forschungsfrage in diesem Zusammenhang ist, ob Sicherungspunkte vollautomatisch gesetzt werden können oder ob eine Kooperation mit der Anwendung notwendig ist. Insbesondere ist es nur dann möglich, die redundante Speicherung von replizierten Strukturen zu vermeiden, wenn sich das betreffende Objekt in einem konsistenten Zustand befindet.

Das Laufzeitsystem der in der vorliegenden Arbeit entwickelten verteilten Programmierumgebung erlaubt derzeit noch keine Hinzu- oder Wegnahme von Rechenknoten während der Laufzeit. Eine solche Dynamik kann aber gerade bei langlaufenden Anwendungen sinnvoll sein, wenn zu bestimmten Zeiten in einem Rechnerbündel mehr Knoten zur Verfügung stehen oder Knoten für Wartungsarbeiten heruntergefahren werden müssen. Bei der Wegnahme von Knoten müssen die vorher auf diesem Knoten angelegten Objekte auf andere Knoten migriert und replizierte Objekte, die Replikate auf dem abzuschaltenden Knoten hatten, neu verteilt werden. Bei Hinzunahme von Knoten müssen bestehende replizierte Objekte ebenfalls umverteilt werden, um die hinzukommende Rechenleistung ausnutzen zu können. Im laufenden Betrieb ist eine solche Dynamik nur schwer vorstellbar, da dafür insbesondere mittlere Segmente aus einem verteilten Aufrufstapel eines maschinenüberspannenden Kontrollfadens auf einen anderen Knoten migriert werden müssten. Es erscheint aussichtsreicher, aufbauend auf einen oben beschriebenen Sicherungsmechanismus, eine Möglichkeit zu schaffen, eine verteilte Anwendung aus einem Sicherungspunkt heraus in einer modifizierten Umgebung wiederanlaufen zu lassen. In dem Sicherungspunkt liegt der gesamte Zustand der Anwendung externalisiert vor und ist vielversprechend, diesen Sicherungspunkt mit einer geänderten Menge von Rechenknoten wiedereinzulesen. Bei diesem Vorgehen bleibt zu klären, wie die Anwendung auf eine solche Situation vorbereitet werden muss oder ob ein Umkonfigurieren der verteilten Umgebung für die Anwendung transparent gehalten werden kann.

Anhang A

Inkrementelles verteiltes Färben von Graphen

Das Problem der Knotenfärbung besteht darin, jedem Knoten $v \in V$ eines gegebenen ungerichteten Graphen $G = (V, E)$ eine Farbe so zuzuordnen, dass keine zwei Knoten v_1, v_2, welche durch eine Kante $\{v_1, v_2\} \in E$ verbunden sind, dieselbe Farbe erhalten. Existiert für einen gegebenen Graphen eine Färbung mit insgesamt k Farben, so heißt der Graph k-färbbar. Die kleinste Zahl k, für welche ein gegebener Graph k-färbbar ist, heißt die chromatische Zahl $\chi(G)$ des Graphen G (vgl. [17]).

A.1 DLF-Algorithmus

Da die Berechnung der chromatischen Zahl eines Graphen und einer dazu passenden optimalen Färbung mit möglichst wenigen Farben np-schwer ist, begnügt man sich in der Praxis damit, eine „gute“ Färbung mit einer kleinen, aber nicht notwendigerweise minimalen Farbenanzahl zu finden. Der DLF-Algorithmus [35] ist ein solcher heuristischer Algorithmus, der zu einem gegebenen Graphen $G = (V, E)$ eine „gute“ Knotenfärbung verteilt berechnet.

Eine Variante des DLF-Algorithmus zur verteilten Graphfärbung aus [35] ist in Prozedur 1 mit den Unterprozeduren 2 und 3 dargestellt. Als Erweiterung von [35] setzt der hier gezeigte Algorithmus keine synchron ablaufenden Runden auf den beteiligten Rechenknoten voraus. Stattdessen synchronisiert sich der hier gezeigte Algorithmus über die versendeten und empfangen Nachrichten selbst.

Zur Berechnung der Färbung muss jeder Knoten nur seine direkten Nachbarknoten kennen. Solange ein Knoten noch aktiv ist, d.h. solange seine Farbe noch nicht feststeht, wählt er eine noch nicht vergebene Farbe mit möglichst niedriger Nummer und eine Zufallszahl aus. Diese beiden Informationen sendet er zusammen mit seinem Rang in einer Anfragenachricht an alle seine Nachbarn, die noch nicht gefärbt sind. Durch Vergleich der Daten mit denen seiner Nachbarn kann jeder Knoten lokal entscheiden, ob er selbst entweder die höchste Priorität besitzt oder einen Farbvorschlag gemacht hat, der nicht im Konflikt mit den Farbwünschen seiner Nachbarknoten steht. Die Idee der verwendeten Heuristik ist, dass Knoten mit vielen Nachbarn, also mit großem Rang, Priorität bei der Farbauswahl haben, da es für diese Knoten besonders

Prozedur 1 Collective DLF for node k

Require: Node k is not colored.
Require: All neighbors of k are marked `active`.
  while node k is not colored **do**
    Choose color $c \in \mathbb{N}_0$ as $\min(\mathbb{N}_0 \backslash \{c_n \mid \text{color } c_n \text{ is } \texttt{used}\})$.
    Choose a random number r.
    Send CHECK$(c, \deg_k, r)$ to all `active` neighbors.
    Receive CHECK$(c_n, \deg_n, r_n)$ from all `active` neighbors n.
    Compare $(c, \deg_k, r)$ against all received parameters.
    if k has highest priority or color c produces no conflict **then**
      Send message COLOR(c) to all neighbors.
      Set node k colored.
    else
      Send CANCEL() to non-conflicting and lower priority neighbors.
    call Receive reply messages.
  call Receive missing color messages.
Ensure: Node k is colored.
Ensure: All neighbors have a color assigned.
Ensure: All neighbors are not `active`.

Prozedur 2 Receive reply messages

  for all `active` neighbors n with higher priority or no conflict **do**
    Receive message m from neighbor n.
    switch message m:
      COLOR(c_n):
        Assign color c_n to neighbor n.
        Mark color c_n as `used`.
        Mark neighbor n as not `active`.
      CANCEL():
        Do nothing.

Prozedur 3 Receive missing color messages

  for all `active` neighbors n **do**
    Receive message COLOR(c_n) from neighbor n.
    Assign color c_n to neighbor n.
    Mark color c_n as `used`.
    Mark neighbor n as not `active`.

schwierig ist, eine noch unbenutzte Farbe zu finden, wenn alle ihre Nachbarn schon gefärbt sind. Die Zufallszahl wird bei gleichem Rang für die Prioritätsentscheidung herangezogen, um in regulären Bereichen des Graphen durch Randomisierung die Laufzeitgarantien des Algorithmus von $O(\Delta^2 \log(n))$ Runden einhalten zu können. Hat ein Knoten höchste Priorität oder steht sein Farbvorschlag nicht im Konflikt mit Vorschlägen seiner Nachbarn, erklärt er sich für gefärbt und teilt dies seinen Nachbarknoten in einer Antwortnachricht mit. In allen anderen Fällen widerruft der Knoten seinen Färbungsvorschlag. Nach Auswertung aller Antwortnachrichten weiß jeder Knoten, welche Nachbarn in der nächsten Runde noch am Farbwahl-Prozess teilnehmen werden.

In [35] wird gezeigt, dass nach $O(\Delta^2 \log(n))$ Runden alle Knoten gefärbt sind und dass die Färbung für viele Klassen von Graphen entweder optimal ist oder nahe am Optimum liegt. Diese Eigenschaften gelten unverändert für den hier gezeigten Algorithmus, da es sich lediglich um eine Umformulierung handelt, um ihn in einem nachrichtengekoppelten System einsetzen zu können.

A.2 Erweiterung zu IDLF

Der Algorithmus DLF aus Abschnitt A.1 kann eine bereits vorhandene Färbung, die durch geringfügige Veränderungen der Struktur des Graphen möglicherweise ungültig geworden ist, nicht wiederherstellen. Mit DLF kann bei Modifikation des Graphen lediglich eine ganz neue Färbung berechnet werden. Hat sich der Graph gegenüber dem Zustand, in dem eine Färbung berechnet wurde, nur geringfügig oder überhaupt nicht verändert, verspricht eine inkrementelle „Reparatur“ der Färbung gegenüber der kompletten Neufärbung erhebliche Geschwindigkeitsvorteile. Inkrementelle Färbung kann mit dem in Prozedur 4 dargestellten Algorithmus IDLF tatsächlich erreicht werden. Nach Modifikation des Graphen bestätigt IDLF entweder die Gültigkeit der bestehenden Färbung oder färbt diejenigen Teile des Graphen neu, in denen durch zusätzliche Kanten die Färbung ungültig geworden ist oder in denen durch Wegfall von Kanten eine günstigere Färbung mit weniger Farben erreicht werden kann. Dazu benötigt IDLF im schlimmsten Fall (wenn der Graph stark verändert wurde) nur eine Iteration mehr als DLF. Im günstigsten Fall (wenn die bestehende Färbung auch nach der Modifikation noch gültig ist) terminiert IDLF in einem Schritt und bestätigt die Gültigkeit der bestehenden Färbung.

Der Algorithmus IDLF geht davon aus, dass jeder Knoten bereits über eine Farbe verfügt und die Farben derjenigen Nachbarn kennt, mit denen er auch schon verbunden war, als die bestehende Färbung berechnet wurde. Als erstes entscheidet jeder Knoten darüber, ob er aufgrund der lokal verfügbaren Informationen eine Farbe mit kleiner Nummer auswählen könnte. Dies kann der Fall sein, wenn von dem Knoten ausgehende Kanten weggefallen sind. Daraufhin findet genau ein Abstimmungsschritt mit seinen Nachbarn ähnlich wie in DLF statt, der über das Ausmaß der partiellen Neufärbung entscheidet. Die partielle Neufärbung wird danach mit Rückgriff auf den DLF-Algorithmus auf den ausgewählten Knoten durchgeführt. Ist für keinen Knoten eine Neufärbung notwendig, terminiert IDLF bereits nach dem Abstimmungsschritt und bestätigt die bestehende Färbung.

Prozedur 4 Collective incremental DLF for node k

Require: Node k is colored with color c.
Require: All neighbors of k are marked `active`.
  if there is a color $c_u < c$ that is not `used` **then**
    Set c to c_u.
  Choose a random number r.
  Send CHECK$(c, \deg_k, r)$ to all `active` neighbors.
  Receive CHECK$(c_n, \deg_n, r_n)$ from all `active` neighbors n.
  Compare $(c, \deg_k, r)$ against all received parameters.
  if color c produces no conflict **then**
    Set `accept` to TRUE.
    for all `active` neighbors n with higher degree **do**
      call Receive reply message m from neighbor n.
      if received message m was CANCEL() **then**
        Set `accept` to FALSE.
    if accept **then**
      Send message COLOR(c) to all neighbors.
    else
      Send message CANCEL() to all `active` neighbors.
    for all `active` neighbors n with lower or equal degree **do**
      call Receive reply message m from neighbor n.
    if `accept` **then**
      call Receive missing color messages.
    else
      call Collective DLF for node k.
  else if k has heighest priority **then**
    Send message COLOR(c) to all neighbors.
    Set node k colored.
    call Receive reply messages.
    call Receive missing color messages.
  else
    Send message CANCEL() to all neighbors with lower priority or no conflict.
    call Receive reply messages.
    call Collective DLF for node k.

Falls der Knoten im initialen Abstimmungsschritt mit der höchsten Priorität die Abstimmung gewinnt, bestätigt er seine Farbe wie in DLF und ist direkt nach Empfang der Farbbotschafen von seinen Nachbarn fertig. Verliert er die Abstimmung mit einem Farbkonflikt, ist er Teil der Knotenmenge, die neu gefärbt werden muss, und ruft den DLF-Algorithmus auf. Die Neufärbung wird dabei so auf den geänderten Teil des Graphen beschränkt, dass bei fortgesetzter inkrementeller Färbung keine Verschlechterung der Färbung (durch unnötig viele Farben) verursacht wird. Der Trick, der dieses Verhalten ermöglicht, liegt in der Auswahl der neu zu färbenden Knoten. Für den Fall, dass der lokale Farbvorschlag zwar keinen Konflikt verursacht, der Knoten aber nicht die höchste Priorität zur Durchsetzung seines Vorschlags besitzt, wartet der Knoten alle Antwortnachrichten von seinen Nachbarn mit höherem Rang ab. Diese besitzen garantiert höhere Priorität als er selbst. Nur wenn alle diese Nachbarn ihre Farbe im ersten Schritt bestätigen, bestätigt auch der konfliktfreie Knoten mit kleinerem Rang seine Farbe und ist direkt nach Empfang der restlichen Antwortnachrichten und Farbbotschaften fertig. Ansonsten widerruft er seinen Farbvorschlag, obwohl dieser zu keinem Konflikt geführt hätte und fährt daraufhin mit dem regulären DLF-Algorithmus fort.

Wenn aus Sicht eines Knotens mit konfliktfreiem Farbvorschlag ein Knoten mit größerem Rang im initialen Schritt seinen Farbvorschlag widerrufen muss, ist dies ein Hinweis, dass sich der Graph dort geändert hat und eine lokale Neufärbung notwendig ist. Diese lokale Neufärbung wird mit Rückgriff auf das DLF-Verfahren durchgeführt. Damit dies reibungslos und ohne Einbußen der Färbungsqualität ablaufen kann, muss sichergestellt werden, dass Knoten mit hoher Priorität ihre Farbvorschläge gegenüber Knoten mit niedrigerer Priorität durchsetzen können. Aus diesem Grund darf ein konfliktfreier Knoten mit kleinerem Rang als seine Nachbarn seinen Farbvorschlag, der aufgrund unvollständiger Informationen im initialen Schritt getätigt wurde, nicht vorzeitig festschreiben. Insgesamt haben obige Regeln zur Folge, dass bei Modifikation des Graphen alle Knoten, die direkt von der Modifikation betroffen sind, und rekursiv alle deren Nachbarn mit kleinerem Rang ihre Farbvorschläge widerrufen und zum Färben mit DLF übergehen. Der in diesem Fall angestoßene DLF-Algorithmus arbeitet daraufhin auf einer Teilmenge der Knoten genauso, als ob für die übrigen Knoten, deren initiale Farbvorschläge erfolgreich waren, in vorausgehenden DLF-Runden bereits eine Färbung bestimmt worden wäre. Nur Knoten mit lokal maximalem Rang setzen ihre Färbung im initialen Schritt durch und nur solche Knoten bestätigen einen konfliktfreien Farbvorschlag, die dadurch nicht die Neufärbung von Knoten mit höherem Rang behindern. Daher stellt das Ergebnis des initialen Abstimmungsschrittes von IDLF eine Situation dar, die auch während einer Komplettfärbung mit DLF (bei entsprechender Wahl der Zufallszahlen) hätte auftreten können. Auf dem neu zu färbenden Teil des Graphen startet DLF damit aus einer Situation heraus, die bei komplettem Neufärben nach einigen Anfangsschritten hätte erreicht werden können. IDLF spart diese Anfangsschritte ein und führt zu einem Resultat mit gleicher Färbungsqualität wie DLF.

A.3 Aufwandsabschätzung

Offensichtlich ist die Laufzeit von IDLF durch die Laufzeit von DLF plus einer zusätzlichen Runde beschränkt, da ein Knoten entweder schon im initialen Schritt seine Farbe bestätigt oder eine Neufärbung mit DLF durchführt. Wurde der Graph nicht modifiziert, ist mit IDLF garantiert, dass das Verfahren in genau einem Schritt terminiert, da die bestehende Färbung von allen Knoten bestätigt wird. Aufgrund der Auswahl von Knoten für die Neufärbung läuft der DLF-Algorithmus auf der Teilmenge der neu zu färbenden Knoten ungehindert ab, so dass die Färbungsqualität mit einer kompletten Neufärbung vergleichbar ist. Die gutartigen Eigenschaften von fortgesetztem inkrementellem Neufärben konnten durch Experimente bestätigt werden.

Literaturverzeichnis

[1] Gabriel Antoniu and Luc Bougé. DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. In *Proceedings of the 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'2001)*, volume 2026 of *Lecture Notes in Computer Science*, pages 55–70, San Francisco, April 2001. Springer Verlag.

[2] Gabriel Antoniu and Philip J. Hatcher. Remote object detection in cluster-based Java. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, pages 104–110, San Francisco, April 2001.

[3] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: A single system image of a JVM on a cluster. In *Proceedings of the International Conference on Parallel Processing*, pages 4–12, Fukushima, Japan, September 1999.

[4] Yariv Aridor, Michael Factor, Avi Teperman, Tamar Eilam, and Assaf Schuster. A high performance cluster JVM presenting a pure single system image. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 168–177, San Francisco, California, United States, June 2000.

[5] Laurent Baduel, Françoise Baude, and Denis Caromel. Efficient, flexible, and typed group communications in Java. In *Proceedings of the 2002 joint ACM-ISCOPE Conference on Java Grande*, pages 28–36, Seattle, Washington, United States, November 2002. ACM Press.

[6] Mark Baker, Bryan Carpenter, Sung Hoon Ko, and Xinying Li. mpiJava: A Java interface to MPI. In *Proceedings of the First UK Workshop on Java for High-Performance Network Computing (Euro-Par 1998)*, Southampton, England, September 1998. `http://www.cs.cr.ac.uk/hpjworkshop/`.

[7] Henri E. Bal and Matthew Haines. Approaches for integrating task and data parallelism. *IEEE Concurrency*, 6(3):74–84, July-September 1998.

[8] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the web. *Future Generation Computer Systems*, 15(5-6):559–570, October 1999.

[9] Josh Barnes and Piet Hut. A hierarchical O(N Log N) force calculation algorithm. *Nature*, 324(4):446–449, December 1986.

[10] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, February 1984.

[11] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for scientific applications. In *Proceedings of the 2001 joint ACM-ISCOPE Conference on Java Grande*, pages 97–105, Stanford University, Palo Alto, California, United States, June 2001. ACM Press.

[12] Peter Cappello and Dimitrios Mourloukos. A scalable, robust network for parallel computing. In *Proceedings of the 2001 joint ACM-ISCOPE Conference on Java Grande*, pages 78–86, Stanford University, Palo Alto, California, United States, June 2001.

[13] Denis Caromel, Wilfried Klauser, and Julien Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10(11-13):1043–1061, September-November 1998.

[14] X. Chen and V. H. Allan. MultiJav: A distributed shared memory system based on multiple Java virtual machines. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'1998)*, pages 91–98, Las Vegas Hilton, Las Vegas, Nevada, United States, July 1998.

[15] Bernd O. Christiansen, Peter Cappello, Mihai F. Ionescu, Michael O. Neary, Klaus E. Schauser, and Daniel Wu. Javelin: Internet-based parallel computing using Java. *Concurrency: Practice and Experience*, 9(11):1139–1160, November 1997.

[16] DARPA/IPTO. DARPA HPCS website, 2004. `http://www.darpa.mil/ipto/programs/hpcs/`.

[17] Reinhard Diestel. *Graphentheorie*. Springer-Verlag, Heidelberg, Germany, 2000.

[18] Edsger Wybe Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, October 1971.

[19] Peter Druschel and Larry L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 189–202, Asheville, North Carolina, United States, 1994. ACM Press.

[20] Thomas Fahringer. JavaSymphony: A system for development of locality-oriented distributed and parallel Java applications. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2000)*, pages 145–152, Chemnitz, Germany, December 2000.

[21] Adam Ferrari. JPVM: network parallel computing in Java. *Concurrency: Practice and Experience*, 10(11-13):985–992, September-November 1998.

[22] Stephen Fink and Michael Hind. The design and implementation of the Jikes RVM optimizing compiler, June 2002. Talk held at the ACM Conference on Programming Language Design and Implementation (PLDI'2002), Berlin, Germany.

[23] Robert Flenner. *Jini and Javaspaces Application Development*. Sams Publishing, Indianapolis, Indiana, United States, December 2001.

[24] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[25] Java Grande Forum. The Java Grande forum charter, 1998. `http://www.javagrande.org/jgcharter.html`.

[26] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, Message Passing Interface Forum, 1994.

[27] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces(TM) Principles, Patterns, and Practice*. Addison-Wesley, June 1999.

[28] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, January 1985.

[29] Vladimir Getov, Susan Flynn Hummel, and Sava Mintchev. High-performance parallel programming in Java: exploiting native libraries. *Concurrency: Practice and Experience*, 10(11-13):863–872, September-November 1998.

[30] Matthias Gimbel, Michael Philippsen, Bernhard Haumacher, Peter C. Lockemann, and Walter F. Tichy. Java as a basis for parallel data mining in workstation clusters. In *Proceedings of the International Conference on High Performance Computing and Networking (HPCN'1999)*, volume 1593 of *Lecture Notes in Computer Science*, pages 884–894, Amsterdam, April 1999. Springer Verlag.

[31] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, June 2000.

[32] A.S. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. *IEEE Computer*, 26(5):39–51, May 1993.

[33] Object Management Group. CORBA/IIOP specification (2.6), December 2001.

[34] D. Hagimont and D. Louvegnies. Javanaise: distributed shared objects for internet cooperative applications. In *Middleware'1998*, pages 339–355, The Lake District, England, 1998.

[35] Jennie Hansen, Marek Kubale, Łukasz Kuszner, and Adam Nadolski. Distributed largest-first algorithm for graph coloring. In *Proceedings of the 10th International Euro-Par Conference*, volume 3149 of *Lecture Notes in Computer Science*, pages 804–811, Pisa, Italy, 2004. Springer Verlag.

[36] Saniya Ben Hassen and Henri Bal. Integrating task and data parallelism using shared objects. In *Proceedings of the International Conference on Supercomputing (ICS'1996)*, pages 317–324, Philadelphia, Pennsylvania, United States, May 1996. ACM Press.

[37] P.J. Hatcher, M.J. Quinn, A.J. Lapadula, B.K. Seevers, R.J. Anderson, and R.R. Jones. Data-parallel programming on MIMD computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):377–383, July 1991.

[38] Bernhard Haumacher, Thomas Moschny, Jürgen Reuter, and Walter F. Tichy. Transparent distributed threads for Java. In *Proceedings of the 5th International Workshop on Java for Parallel and Distributed Computing in conjunction with the International Parallel and Distributed Processing Symposium (IPDPS 2003)*, page 136 and CDROM, Nice, France, April 2003. IEEE Computer Society.

[39] Bernhard Haumacher and Michael Philippsen. More efficient object serialization. In *Parallel and Distributed Processing*, volume 1586 of *Lecture Notes in Computer Science*, pages 718–732, San Juan, Puerto Rico, April 1999. Springer Verlag.

[40] Bernhard Haumacher and Michael Philippsen. Exploiting object locality in JavaParty, a distributed computing environment for workstation clusters. In *CPC2001, 9th Workshop on Compilers for Parallel Computers*, pages 83–94, Edinburgh, Scotland, UK, June 2001.

[41] Maurice P. Herlihy. The Aleph toolkit: Support for scalable distributed shared objects. In *Communication, Architecture, and Applications for Network-Based Parallel Computing*, pages 137–149, Orlando, Florida, January 1999.

[42] Maurice P. Herlihy and Michael P. Warres. A tale of two directories: implementing distributed shared objects in Java. *Concurrency: Practice and Experience*, 12(7):555–572, May 2000.

[43] Michael W. Hicks, Suresh Jagannathan, Richard Kelsey, Jonathan T. Moore, and Cristian Ungureanu. Transparent communication for distributed objects in Java. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 160–170, San Francisco, California, United States, June 1999.

[44] Ophir Holder, Israel Ben-Shaul, and Hovav Gazit. Dynamic layout of distributed applications in FarGo. In *International Conference on Software Engineering*, pages 163–173, Los Angeles, California, United States, May 1999.

[45] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming*. PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, United States, January 1987. Also available as Technical Report 87-01-01.

[46] ECMA International. C# language specification (standard ECMA-334), December 2002.

[47] Jerry James and Ambuj K. Singh. Design of the Kan distributed object system. *Concurrency: Practice and Experience*, 12(8):755–797, July 2000.

[48] Glenn Judd, Mark Clement, and Quinn Snell. DOGMA: distributed object group metacomputing architecture. *Concurrency: Practice and Experience*, 10(11-13):977–983, September-November 1998.

[49] M. Frans Kaashoek, Andrew S. Tanenbaum, Susan Flynn Hummel, and Henri E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, 1989.

[50] Alan H. Karp. Programming for parallelism. *IEEE Computer*, 20(5):43–57, May 1987.

[51] P. Keleher, S. Dwarkadas, Alan L. Cox, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

[52] Jongsung Kim and Cheeha Kim. A total ordering protocol using a dynamic token-passing scheme. *Distributed Systems Engineering*, 4(2):87–95, June 1997.

[53] Vijaykumar Krishnaswamy, Ivan B. Ganev, Jaideep M. Dharap, and Mustaque Ahamad. Distributed object implementations for interactive applications. In *Middleware*, volume 1795 of *Lecture Notes in Computer Science*, pages 45–70. Springer Verlag, 2000.

[54] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol, and Mustaque Ahamad. Efficient implementations of Java remote method invocation (RMI). In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems*, pages 19–36, Santa Fe, New Mexico, April 1998.

[55] F. Kuijlman, C. van Reeuwijk, A. J. C. van Gemund, and H. J. Sips. Code generation techniques for the task-parallel language Spar. In *Proceedings of the 7th International Conference on Compilers for Parallel Computers (CPC'1998)*, pages 1–11, Linkoping, Sweden, June 1998.

[56] Pascale Launay and Jean-Louis Pazat. A framework for parallel programming in Java. In *Proceedings of the International Conference on High Performance Computing and Networking (HPCN'1998)*, pages 628–637, Amsterdam, The Netherlands, April 1998.

[57] E. Laure. Distributed high performance computing with OpusJava. In E. H. D'Hollander, J. R. Joubert, F. J. Peters, and H. Sips, editors, *Proceedings of the International Conference ParCo'1999*, pages 590–597, Delft, The Netherlands, August 2000. Imperial College Press.

[58] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, November 1999.

[59] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, November 1989.

[60] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley, 1999.

[61] Sang Boem Lim, Bryan Carpenter, Geoffrey Fox, and Han-Ku Lee. Collective communication for the HPJava programming language. *Concurrency and Computation: Practice and Experience*, 17(7-8):867–894, June - July 2005.

[62] Tim Lindholm. JSR-133: Java memory model and thread specification. Technical Report Proposed Final Draft, Sun Microsystems, April 2004. Java Community Process, `http://jcp.org/en/jsr/detail?id=133`.

[63] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification Second Edition*. Addison-Wesley, May 1998.

[64] Marcelo Lobosco, Claúdio Amorim, and Orlando Loques. A Java environment for high-performance computing. Technical Report RT-03/01, Instituto de Computação, Universidade Federal Fluminense, Rio de Janeiro, Brazil, May 2001.

[65] Matchy J. M. Ma, Cho-Li Wang, Francis C. M. Lau, and Zhiwei Xu. JESSICA: Java-enabled single system image computing architecture. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'1999)*, pages 2781–2787, Las Vegas, Nevada, United States, June 1999.

[66] Jason Maassen, Thilo Kielmann, and Henri E. Bal. Efficient replicated method invocation in Java. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 88–96, San Francisco, California, United States, June 2000.

[67] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, Thilo Kielmann, Ceriel Jacobs, and Rutger F. H. Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):747–775, November 2001.

[68] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. An efficient implementation of Java's remote method invocation. *ACM SIGPLAN Notices*, 34(8):173–182, August 1999.

[69] M. W. Macbeth, K. A. McGuigan, and Philip J. Hatcher. Executing Java threads in parallel in a distributed-memory environment. In *Proceedings of the CASCON'1998*, pages 40–54, Missisauga, ON, 1998. IBM Canada and the National Research Council of Canada.

[70] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst D. Simon. TOP500 supercomputer sites, June 2004. `http://www.top500.org/`.

[71] Sun Microsystems. Java remote method invocation (RMI). `http://java.sun.com/products/jdk/rmi/`.

[72] Sun Microsystems. RPC: Remote procedure call protocol specification: Version 2 (RFC1057), June 1988.

[73] Sun Microsystems. Java technology: An early history, 1998.

[74] Sun Microsystems. Java object serialization specification, August 2001. Available online from http://java.sun.com/j2se/.

[75] Sava Mintchev and Vladimir Getov. Towards portable message passing in Java: Binding MPI. In *Proceedings of EuroPVM-MPI*, volume 1332 of *Lecture Notes in Computer Science*, pages 135–142, Kraków, Poland, November, 1997. Springer Verlag.

[76] Matthias M. Müller. Compiler-generated vector-based prefetching on architectures with distributed memory. In *High Performance Computing in Science and Engineering '01*, volume 2001 of *Transactions of the High Performance Computing Center Stuttgart (HLRS)*, pages 527–539. Springer Verlag, 2001.

[77] A. Nelisse, Thilo Kielmann, Henri E. Bal, and Jason Maassen. Object-based collective communication in Java. In *Proceedings of the 2001 joint ACM-ISCOPE Conference on Java Grande*, pages 11–20, Stanford University, Palo Alto, California, United States, June 2001.

[78] Christian Nester, Michael Philippsen, and Bernhard Haumacher. Effizientes RMI für Java. In C. H. Cap, editor, *Proceedings of JIT'1999*, Informatik Aktuell, pages 135–148, Düsseldorf, Germany, September 1999. Springer Verlag.

[79] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient RMI for Java. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 152–159, San Francisco, California, United States, June 1999.

[80] Michael Philippsen. Data parallelism in Java. In *Proceedings of the International Symposium on High Performance Computing Systems and Applications (HPCS'1998)*, pages 85–99, Edmonton, Alberta, Canada, May 1998.

[81] Michael Philippsen. A survey of concurrent object-oriented languages. *Concurrency: Practice and Experience*, 12(10):917–980, August 2000.

[82] Michael Philippsen and Bernhard Haumacher. Locality optimization in JavaParty by means of static type analysis. In *Proceedings of the First UK Workshop on Java for High Performance Network Computing (Euro-Par 1998)*, Southampton, England, September 1998.

[83] Michael Philippsen and Bernhard Haumacher. Locality optimization in JavaParty by means of static type analysis. *Concurrency: Practice and Experience*, 12(8):613–628, July 2000.

[84] Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, May 2000.

[85] Michael Philippsen and Matthias Zenger. JavaParty - transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[86] J. Postel. *User Datagram Protocol (STD 0006)*. Information Sciences Institute (ISI) of the University of Southern California (USC), August 1980. Internet Standard.

[87] J. Postel. *Transmission Control Protocol (STD 0007)*. Information Sciences Institute (ISI) of the University of Southern California (USC), September 1981. Internet Standard.

[88] Lutz Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, October 2000.

[89] Jelica Protić, Milo Tomašević, and Veljko Milutinović. An overview of distributed shared memory. In *Distributed Shared Memory: Concepts and Systems*, pages 12–41. John Wiley & Sons, Ltd., Chichester, West Sussix, August 1997.

[90] Christopher J. Riley, Siddhartha Chatterjee, and Rupak Biswas. High-performance Java codes for computational fluid dynamics. *Concurrency and Computation: Practice and Experience*, 15(3-5):395–415, February 2003.

[91] Marc Snir. JSR 84: Floating point extensions, October 2000. Java Community Process, `http://jcp.org/jsr/detail/84.jsp`.

[92] Yukihiko Sohda, Hidemoto Nakada, Satoshi Matsuoka, and Hirotaka Ogawa. Implementation of a portable software DSM in Java. In *Proceedings of the 2001 joint ACM-ISCOPE Conference on Java Grande*, pages 163–172, Stanford University, Palo Alto, California, United States, June 2001. ACM Press.

[93] Vaidy S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–340, December 1990.

[94] George K. Thiruvathukal, Fabian Breg, Ronald Boisvert, Joseph Darcy, Geoffrey C. Fox, Dennis Gannon, Siamak Hassanzadeh, Jose Moreira, Michael Philippsen, Roldan Pozo, and Marc Snir. Java Grande Forum Report: Making Java work for high-end computing. In *Supercomputing'98: International Conference on High Performance Computing and Communications*, Orlando, Florida, United States, November 1998. panel handout.

[95] David A. Thurman. JavaPVM: The Java to PVM interface, June 1996. `http://www.cis.ksu.edu/Systems/Info/JavaPVM/`.

[96] Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Automatic Java application partitioning. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 178–204, University of Málaga, Spain, June 2002. Springer Verlag.

[97] Brad Topol, Mustaque Ahamad, and John T. Stasko. Robust state sharing for wide area distributed applications. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS'1998)*, pages 554–561, Amsterdam, The Netherlands, May 1998.

[98] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[99] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, and Henri E. Bal. Ibis: a flexible and efficient Java-based grid programming environment. *Concurrency: Practice and Experience*, 17(7-8):1079–1107, June-July 2005.

[100] C. van Reeuwijk, A. J. C. van Gemund, and H. J. Sips. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, November 1997.

[101] Ronald Veldema, Raoul A. F. Bhoedjang, and Henri E. Bal. Distributed shared memory management for Java. Technical report, Vrije Universiteit Amsterdam, November 1999.

[102] Ronald Veldema, Rutger F. H. Hofman, Raoul A. F. Bhoedjang, and Henri E. Bal. Runtime optimizations for a Java DSM implementation. In *Proceedings of the 2001 joint ACM-ISCOPE Conference on Java Grande*, pages 153–162, Stanford University, Palo Alto, California, United States, June 2001. ACM Press.

[103] Matt Welsh and David Culler. Jaguar: Enabling efficient communication and I/O in Java. *Concurrency: Practice and Experience*, 12(7):519–538, May 2000.

[104] Dan Werthimer, Jeff Cobb, Matt Lebofsky, David Anderson, and Eric Korpela. Seti@home: massively distributed computing for SETI. *Computing in Science and Engineering*, 3(1):78–83, January 2001.

[105] Emil A. West and Andrew S. Grimshaw. Braid: Integrating task and data parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'1995)*, pages 211–219, McLean, Virginia, United States, February 1995. IEEE Computer Society Press.

[106] Danny Weyns, Eddy Truyen, and Pierre Verbaeten. Distributed threads in Java. In *Proceedings of the International Symposium on Parallel and Distributed Computing (ISPDC 2002)*, pages 94–109, Iaş, Romania, July 2002.

[107] James E. White. A high-level framework for network-based resource sharing. In *Proceedings of the National Computer Conference*, June 1976.

[108] Weimin Yu and Alan L. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, November 1997.

[109] Wenzhang Zhu, Cho-Li Wang, and Francis C. M. Lau. JESSICA2: A distributed Java virtual machine with transparent thread migration support. In *IEEE Fourth International Conference on Cluster Computing*, pages 381–389, Chicago, Illinois, United States, September 2002.