

Erweiterung des Threadmodelles für den Einsatz in verteilten und heterogenen Systemumgebungen

Von der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von

Raik Nagel

aus Halle (Saale)

1.Gutachter: Prof. Dr. Thomas Rauber

2.Gutachter: Prof. Dr. Paul Molitor

Tag der Einreichung: 13.05.2009

Tag des Kolloquiums: 30.10.2009

Für Kathleen und Vincent

Erweiterung des Threadmodelles für den Einsatz in verteilten und heterogenen Systemumgebungen

Kurzfassung

Für die klassischen *distributed memory* und *shared memory* Architekturen kann bei der Programmierung auf bewährte nachrichten- und threadbasierte Programmiermodelle zurückgegriffen werden. Die weite Verbreitung und Akzeptanz dieser Modelle ermöglicht den portablen Einsatz und die Migration von bestehenden Programmen auf andere gleichartige Computersysteme. In verteilten heterogenen Systemumgebungen, die aus einer Vielzahl von unterschiedlichen Computerknoten bestehen und über ein Netzwerk miteinander verbunden sind, wird meistens ein nachrichtenbasiertes Programmiermodell verwendet. Neben praktischen Problemen, wie der Sicherstellung der korrekten Datendarstellung beim Nachrichtenaustausch zwischen Knoten mit unterschiedlicher Prozessorarchitektur, ergeben sich weitere Fragestellungen, die eine Erstellung und Portierung von Programmen für solche Ausführungsumgebungen erschweren. So ist man beispielsweise bei der Verwendung einer MPI-Bibliothek meist an ein Netzwerk gebunden und kann oft Systeme mit unterschiedlichen Prozessorarchitekturen nicht gleichzeitig verwenden.

Im ersten Teil der vorliegenden Arbeit wird das Modell einer hybriden Programmierumgebung vorgestellt. Durch die Verwendung von Konzepten aus den thread- und nachrichtenbasierten Programmiermodellen soll eine leichte Einarbeitung in das System und ein breites Einsatzgebiet ermöglicht werden. Im Anschluß an die Spezifikationen und Definitionen der Programmierumgebung und des Programmiermodelles wird die hierfür erstellte Prototypimplementierung beschrieben. Diese Prototypimplementierung erlaubt die Ausführung von Programmen in verteilten heterogenen Systemumgebungen und zeichnet sich durch eine hohe Modularität aus. Dies bedeutet, daß nahezu jede Komponente des Systems austauschbar ist und Funktionen bereitstellt, welche in der allgemeinen Spezifikation der Umgebung definiert werden. Für verschiedene Zielsetzungen des Anwenders, wie z.B. die Sicherheit der Datenübertragung oder eine hohe Fehlertoleranz der gesamten Kommunikation, lassen sich so unterschiedliche Komponenten in das Laufzeitsystem einbinden, ohne das Benutzerprogramm neu erstellen oder anpassen zu müssen. Die Benutzung und Einbindung von bereits existierenden, externen Implementierungen für solche Anforderungen ist ebenfalls möglich und reduziert den Entwicklungsaufwand neuer Komponenten des Laufzeitsystems. In der Arbeit wird das unter anderem anhand der *TSpaces*-Bibliothek von IBM als alternatives Speichersystem demonstriert.

Die sich im zweiten Teil der Arbeit anschließenden Kapitel befassen sich mit mehreren Umsetzungen von numerischen Algorithmen für die erstellte Prototypimplementierung der beschriebenen Programmierumgebung. Die erzielten Ergebnisse zeigen, daß die im Vorfeld definierten Ziele erreicht werden können und die resultierenden Programme mit unterschiedlichen Systemkonfigurationen lauffähig sind.

A Thread Model Extension for Distributed and Heterogeneous Environments

Abstract

For the classic *distributed memory* and *shared memory* programming it is possible to use the common message passing or multithreading concepts. The prevalence and acceptance of this kind of programming allows the portable use and the migration of existing programming code into similar computer environments. Message passing models are used basically for heterogeneous systems. These systems consist on many kinds of different computer nodes which are connected by a network. The preservation of the correct presentation of data encoding for the communication between nodes of different processor architectures is only one of many practical problems. There are some more issues which complicate the creation and the porting of programs for this kind of execution environment. Therefore the use of a MPI library is often only possible inside one network and mostly it is not possible to use systems with different processor architectures simultaneously.

The first part of this work presents the model of a hybrid programming environment. The use of various message passing and multithreading concepts enables an easy orientation and allows the implementation of applications for a wide range of execution environments. This part includes the definitions and specifications of the programming environment and the programming model. Afterwards the developed prototype implementation is described. This prototype implementation allows the execution of parallel programs in distributed heterogeneous system environments and is characterized by a high modularity level. That means, that nearly each component of this prototype implementation can be replaced by another component and that it offers functions, which are defined in the basic specification of the programming environment. For different user requirements, for example a safe or fault-tolerant communication, it is possible to plug in different components into the prototype runtime system, without a reimplementaion of the user application. It is also possible to integrate already existing and external 3rd party implementations for this kind of applications. Such strategy reduces the development effort of new components of the prototype environment. In this work the *TSpaces* library by IBM is taken as an alternative runtime component for transparent and distributed memory access.

The following chapters in the second part of this work are dealing with several numerical algorithms for the prototype implementation of the described programming environment. The experimental results show that the predefined objectives can be achieved and the resulting programs are running at a wide range of possible system configurations.

Danksagung

An dieser Stelle möchte ich all denen danken, die mich auf unterschiedliche Weise unterstützt und so zur Entstehung dieser Arbeit in direkter oder indirekter Form beigetragen haben.

Herrn Prof. Dr. Rauber danke ich für die Möglichkeit zur Promotion und den gewährten Freiraum für die Verfolgung meiner Ideen.

Bedanken möchte ich mich bei den Mitarbeitern des Lehrstuhles für Angewandte Informatik II der Universität Bayreuth. Hier vor allem Dr. Sascha Hunold für die vielen inspirierenden Diskussionen und Simon Melzner für die schnelle unkomplizierte Hilfe bei Hardwareproblemen. Ein besonderer Dank geht an Ralf Hoffmann für seine stete Hilfsbereitschaft und wertvolle Unterstützung vor allem in der Endphase dieser Arbeit.

Natürlich möchte ich mich auch bei meiner Familie bedanken. An erster Stelle geht mein Dank an meine Mutter Christiane. Ohne Deine Unterstützung hätte ich diesen Ausbildungsweg nicht gehen können. Danke auch an Barbara und Klaus-Dieter für Eure Unterstützung.

Ronny, Dir danke ich für Deine nette Art, mich immer wieder daran zu erinnern, daß es auch andere Dinge außerhalb des Informatik-Universums gibt.

Zuletzt möchte ich mich bei Dir, Kathleen, aufs herzlichste für die aufgebrauchte Geduld beim Korrekturlesen und die ständige Fürsorge bedanken. Ohne Dich wäre vieles nicht möglich geworden!

Inhaltsverzeichnis

1	Einleitung	19
1.1	Motivation und Ziele	20
1.2	Aufbau und weitere Gliederung	23
2	Architekturen und Programmiermodelle	25
2.1	Architekturen	25
2.2	Speicherorganisation	26
2.2.1	Cache-Speicher	26
2.2.2	Gemeinsamer Speicher	29
2.2.3	verteilter Speicher	32
2.3	Programmiermodelle und Laufzeitsysteme	34
2.3.1	Speicher- und Konsistenzmodelle	35
2.3.2	Nachrichtenbasierte Programmierung - Message Passing	36
2.3.3	Multithreading	38
2.3.4	DSM und VSM Modelle	41
2.3.5	Linda TupleSpace	44
2.3.6	weitere parallele Programmiermodelle und Umgebungen	45
2.4	Java	47
2.4.1	Kritische Bereiche	47
2.4.2	Das Java Memory Modell	49
2.4.3	Der Garbage Collector	50
2.5	Zusammenfassung	54
3	Das DEE Programmiermodell	57
3.1	Kernel Level	60
3.2	DTM: Programmiermodell und Trails	62
3.2.1	Ausführungsmodell	63
3.2.2	Speichermodell	64
3.2.3	Kommunikation und Steuerung	65
3.3	Distributed Shared Memory	66
3.3.1	Server-Level	67
3.3.2	Protocol-Level	69
3.3.3	User-Level und DSM-Objekt	71
3.3.4	Erweiterte Funktionen des DSM-Subsystems	72
3.4	Weitere Kernkomponenten des <i>DEE</i>	74
3.4.1	Task-Provider	74

3.4.2	DEE-Service	74
3.5	Schlußbemerkungen	75
4	Die DEE Prototypimplementierung	77
4.1	DTM-Kernkomponente	78
4.1.1	Benutzerschnittstellen des DTM	78
4.1.2	Ausführungsmodell	82
4.1.3	Verteilte Synchronisation	85
4.1.4	Speicher- und Datenmodell	87
4.2	DSM-Kernkomponente	90
4.2.1	Benutzerschnittstellen des DSM	91
4.2.2	DSM-Server-Level	104
4.2.3	DSM-Protocol-Level und Datenkohärenz	108
4.2.4	DSM-User-Level	112
4.3	Kernel Level Implementierung	113
4.3.1	System-Core-Kernkomponente	114
4.3.2	Namespace-Modul	116
4.3.3	Kommunikationsmodul	117
4.4	DEE-Service-Kernkomponente	129
4.5	Schlußbemerkungen	131
5	Experimentelle Bewertung der Prototypimplementierung	133
5.1	Kommunikationsschicht	135
5.1.1	Verwaltungsmehraufwand	136
5.1.2	Erweiterung durch Java Fast Sockets	141
5.1.3	Fazit	142
5.2	DSM	143
5.2.1	Server-Level: direkter DSM-Serverzugriff	144
5.2.2	Protocol-Level: Cache und Kohärenz	147
5.2.3	Bandbreite	152
5.2.4	Zusammenfassende Auswertung	166
5.3	Laufzeitsystem	167
5.3.1	Matrixmultiplikation	168
5.3.2	N-Damen-Problem	172
5.3.3	Bewertung des Laufzeitsystems	174
5.3.4	Laufzeitvorhersage	174
5.3.5	Zusammenfassung	176
5.4	Schlußbemerkungen	176
6	Numerische Simulation	177
6.1	Simulationsverfahren	177
6.1.1	Simulationsraum	178
6.1.2	Verfahren zur Partikel- und Strömungssimulation	178
6.2	Kurzbetrachtung ausgesuchter Verfahren	179

6.2.1	Linked-Cell-Verfahren	179
6.2.2	Barnes-Hut-Algorithmus	180
6.2.3	Lattice-Boltzmann-Methode	181
6.3	Zusammenfassung	181
7	Linked-Cell-Verfahren	183
7.1	Beschreibung des Verfahrens	183
7.1.1	Grundlagen und Berechnungsvorschriften	183
7.1.2	Allgemeines Vorgehen	186
7.2	Sequentielle und threadparallele Implementierungen	187
7.2.1	Berechnung der Nachbarräume	188
7.2.2	Verwendete Datenstruktur	189
7.2.3	Implementierungen und Testergebnisse	189
7.3	DEE Implementierung	192
7.3.1	DSM-Datenstruktur Raum	192
7.3.2	Auswertung	192
7.4	Fazit	196
8	Barnes-Hut-Algorithmus	199
8.1	Beschreibung des Verfahrens	199
8.2	Sequentielle und threadparallele Varianten	201
8.2.1	sequentielle Implementierung	201
8.2.2	Threadparallele Variante	205
8.3	DEE Variante	208
8.3.1	Vorüberlegungen	208
8.3.2	Datenstruktur	210
8.3.3	Implementierung	212
8.3.4	Optimierungsansätze	213
8.4	Zusammenfassung	216
9	Zusammenfassung und Ausblick	217
A	Ergänzungen	221
A.1	Berechnungen für k -näre Bäume	221
A.2	Java-Optimierungen	222
B	Laufzeitsystem und Testumgebungen	223
B.1	<i>Beluga</i> - Benutzerschnittstellen und Klassen	223
B.1.1	Konfiguration	226
B.2	verwendete Rechnersysteme und Systemkonfigurationen	228
B.2.1	Übersicht verwendeter Rechnersysteme	228
B.2.2	Laufzeitkonfigurationen	232
B.2.3	Bestimmung von Leistungsmerkmalen	232
	Literaturverzeichnis	235

Abbildungsverzeichnis

2.1	Speicherhierarchie des <i>Intel Core 2 Quad</i>	27
2.2	Cache-Kohärenz-Fehlzugriffe	28
2.3	Einteilung virtueller Speichermodelle nach Raina	42
2.4	OpenCL	45
2.5	Beispiel für eine zyklische Datenstruktur	51
2.6	generationelle Speicherbereinigung	53
2.7	Java-Heap-Aufteilung	54
3.1	Übersicht <i>DEE</i>	59
3.2	Verbindungsnetzwerk mit Gateways	62
3.3	<i>DEE</i> Speicherhierarchie	64
3.4	DSM-Subsystem	66
3.5	DSM-Beispielkonfiguration	69
3.6	Beispiel eines verteilten Feldes im DSM	72
3.7	LocalLockManager	73
3.8	DEE Schichtenmodell	74
4.1	DTM-Teamerzeugung	84
4.2	<i>shared</i> Datenmodell der DTM	89
4.3	<i>private</i> Datenmodell der DTM	90
4.4	vereinfachtes <i>Beluga</i> DSM-Modell	91
4.5	<i>Beluga</i> DSM	92
4.6	DSM-Zelle und lokale Kopien	93
4.7	Varianten für die Erzeugung von DSM-Zellen	105
4.8	Central- und Spread-DSM	106
4.9	<i>Beluga</i> Data-Policy	110
4.10	Vorgänge beim Erzeugen von DSM-Objekten	112
4.11	Registry-Server	114
4.12	<i>Beluga</i> L_w -Werte	116
4.13	<i>Beluga</i> I/O Aufbau	119
4.14	PingPong-Benchmark	124
4.15	PingPong-Benchmark 2	125
4.16	<i>Beluga</i> Kommunikationsmodul	126
4.17	Übertragungsrate des PingPong-Tests	127
4.18	<i>Beluga</i> Shell	130
5.1	Testkonfigurationen	134

5.2	Netzwerklatenzen und Bandbreiten	137
5.3	Bandbreiten mit Objektserialisierung	138
5.4	<i>Beluga</i> Sendeoverhead und Bandbreiten	139
5.5	<i>Beluga</i> Zeitoverhead	141
5.6	Gegenüberstellung JFS Bandbreiten	142
5.7	direkter DSM-Zugriff	146
5.8	TSpaces-WriteTest	148
5.9	DSM-Zellen anlegen	148
5.10	DSM-Cachetests A	149
5.11	DSM Cachetest4	151
5.12	DSM-Cachetests B	152
5.13	Hauptspeicherbandbreiten einiger Testsysteme	155
5.14	verwendete Zellen für Bandbreitentests	157
5.15	lokale DSM-Bandbreiten Fall a	159
5.16	lokale DSM-Bandbreiten Fall b	159
5.17	DSM-Speicherbedarf	161
5.18	verteilte DSM-Bandbreiten	163
5.19	verteilte DSM-Bandbreiten	163
5.20	TSpaces-DSM-Bandbreiten	164
5.21	TSpaces-DSM-Speicherbelegung	165
5.22	DSM-Bandbreite mit Locks	166
5.23	DSM-Bandbreite mit Locks (ohne Warmup)	167
5.24	Matrixmultiplikation	169
5.25	Matrixmultiplikation DSM	170
5.26	Trailverteilung Konfiguration K3	171
5.27	Matrixmultiplikation in einer heterogenen Systemkonfiguration	171
5.28	Matrixmultiplikation mit DSM	172
5.29	Laufzeiten für das N-Damen-Problem	173
6.1	Octree	180
6.2	Lattice-Boltzmann	181
7.1	Lennard-Jones-(12,6)-Potential	185
7.2	Linked-Cell	188
7.3	Linked-Cell:Aufteilung des Simulationsraumes	189
7.4	Linked-Cell: Datenstruktur	190
7.5	Linked-Cell: Laufzeiten	191
7.6	Linked-Cell: verschiedene Speedups	191
7.7	Linked-Cell: DSM-Datenobjekt	193
7.8	Linked-Cell: Laufzeiten pro Iteration	194
7.9	Linked-Cell: verteilte Laufzeiten	195
7.10	Linked-Cell: Speedups und Laufzeiten im verteilten Fall	195
7.11	Linked-Cell: Speedup für Central- und TSpaces-DSM	196

8.1	Barnes-Hut-Partikelwechselwirkungen	200
8.2	Barnes-Hut-Simulation	202
8.3	Barnes-Hut-Abbruchkriterium	204
8.4	Barnes-Hut-Rechenzeitverteilung	207
8.5	Barnes-Hut-Iterationen	210
8.6	Barnes-Hut-Version für <i>DEE</i>	213
8.7	Barnes-Hut-Optimierungsvarianten 1	214
8.8	Barnes-Hut-Optimierungsvarianten 2	215
A.1	Quadtree	222

Kapitel 1

Einleitung

Nach der heute üblichen Auslegung des Moore'schen Gesetzes [90] verdoppelt sich die Anzahl von Transistoren auf einem handelsüblichen Prozessorchip etwa alle 18 bis 24 Monate. Neben der Vergrößerung des direkt in einem Prozessorchip integrierten Speichers (*Cache*) nimmt auch die Komplexität und Mächtigkeit der Berechnungseinheiten zu, was zu einer Steigerung der allgemeinen Leistungsfähigkeit eines Prozessors führt. Beispielsweise ist heute ein Mobiltelefon in der Lage, Anwendungen auszuführen, für die vor einigen Jahren noch teure Spezialcomputer nötig waren (*Multimedia*). Allerdings läßt sich dieser Trend der stetigen Leistungssteigerung mit der bisher von den meisten Herstellern dieser Prozessoren verfolgten Strategie, durch Erhöhung der Taktraten einen Leistungsgewinn zu erzielen, kaum halten. Da es durch die höheren Taktraten auch zu einer gesteigerten elektrischen Leistungsaufnahme¹ der Prozessoren kommt, welche wiederum zu thermischen Problemen führen, versuchen die Hersteller durch Verkleinerung der Strukturgrößen und einer geringeren elektrischen Betriebsspannung diesem entgegenzuwirken [101]. Es ist aber absehbar, daß die weitere Verkleinerung der Strukturgrößen in naher Zukunft durch die gegebenen physikalischen Gesetzmäßigkeiten nicht mehr im bisherigen Maße erfolgen kann. Um aber die Entwicklung weiterhin auf diesem Niveau fortsetzen zu können, besitzen die meisten der heutigen Prozessoren mehrere Kerne und in Zukunft wird sich die Anzahl der Kerne pro Prozessor wohl noch weiter erhöhen [50, 85, 113]. Mit diesem Vorgehen wird eine Steigerung der relativen Leistungsfähigkeit solcher Prozessoren erzielt, da die vorhandenen Kerne in der Lage sind, Berechnungen gleichzeitig auszuführen. Hierdurch vervielfacht sich die Anzahl der theoretisch durchführbaren Operationen pro Sekunde im Vergleich zu einem Prozessor mit nur einem Kern, ohne die Taktfrequenz weiter zu erhöhen.

Damit dieses Leistungspotential für den Benutzer voll verfügbar ist, müssen die auszuführenden Anwendungen dementsprechend entwickelt sein und die Durchführung der Berechnungen durch parallele Kontrollflüsse erlauben. Viele der heutigen Programme benutzen jedoch nur einen Kontrollfluß (*sequentiell*) oder sind so entworfen, daß eine effektive parallele Bearbeitung nicht möglich ist. Deshalb wächst die Notwendigkeit, bei der Entwicklung von neuen Anwendungen gezielt auf deren Parallelisierung zu achten.

¹Die maximale Verlustleistung, d.h. die von einem Prozessor direkt in Wärme umgewandelte elektrische Leistung (*TDP* - *Thermal Design Power*), ist bei einem Pentium 4 Prozessor von Intel mit 86W angegeben [64].

Hierfür ist das Vorhandensein von einfach handhabbaren, robusten und universell einsetzbaren parallelen Programmiermodellen eine Grundvoraussetzung.

1.1 Motivation und Ziele

Parallel arbeitende Computersysteme können anhand der Organisation ihres Adreßraumes unterteilt werden. Es ergeben sich dann zwei dominierende Architekturen: *shared memory* und *distributed memory*. In Abhängigkeit von dieser Architektur eignet sich für die parallele Programmierung entweder ein Threadmodell (*shared*) oder ein auf Nachrichtenaustausch basierendes Modell (*distributed*). Beide Programmiermodelle unterscheiden sich grundlegend und es ist oftmals nicht möglich oder besonders effektiv, Programme des einen Modelles auf einem System mit der Architektur des anderen Modelles auszuführen.

Der *MPI*-Standard², als Umsetzung eines nachrichtenbasierten Programmiermodelles, ist das dominierende System für parallele wissenschaftliche Berechnungen [11]. Dies liegt aber vor allem an den zur Verfügung stehenden parallelen Systemen, welche heute (noch) häufig aus einzelnen unabhängigen Knoten zusammengesetzt sind. Durch die oben beschriebene Entwicklung hin zu Prozessoren mit vielen einzelnen Kernen, die alle Zugriff auf einen gemeinsamen Hauptspeicher haben, sind Systeme mit vergleichbarer Leistung, für die sich das Threadmodell eignet, immer häufiger anzutreffen. Deshalb ist anzunehmen, daß die Anzahl von multithreaded Anwendungen weiter steigt und dieses Programmiermodell von vielen Programmierern eingesetzt wird.

Mit der fortschreitenden Entwicklung der Hardware und dem Zusammenwachsen der Netze ist es heute keine Seltenheit mehr, Systeme mit gemischten Architekturen zur Verfügung zu haben. Die Erstellung von Programmen für **beliebige** Rechnersysteme, die aus einzeln im Netzwerk miteinander verbundenen Multiprozessorknoten oder aus einem einzelnen Multiprozessorsystem bestehen, kann mit den konventionellen Programmiermodellen nicht - oder nur mit hohem Aufwand - effektiv erfolgen. Setzt man einige Systemparameter für die Zielplattform einer zu erstellenden Anwendung voraus³, läßt sich eine solche Umgebung sehr effektiv mit *MPI* programmieren. Hinzu kommen verschiedene Implementierungen des *MPI*-Standards in Form von Programmierbibliotheken, welche besondere Systemparameter ausnutzen und voraussetzen. Bei einigen dieser Implementierungen wird beispielsweise auf Multiprozessorknoten mit n Prozessoren bzw. n Kernen, im Gegensatz zu den üblicherweise erzeugten n *MPI*-Prozessen, nur ein einzelner Kommunikationsprozeß gestartet. Die bisherigen Aufgaben der Prozesse übernehmen dann Threads, da deren Ausführung und *Intranode*-Kommunikation erheblich günstiger ist [13, 61].

Andere Umsetzungen des *MPI*-Standards erlauben die Ausführung von Programmen in einer *Grid*-Umgebung [40]. Eine solche Variante wird z.B. durch die *MPICH-G2*-

²*MPI* steht für *Message Passing Interface*, <http://www.mpi-forum.org/>

³z.B. homogene Systemknoten

Bibliothek⁴ verwirklicht. Diese basiert auf dem *Globus Toolkit*⁵ und kann deshalb auf ein mächtiges und ausgereiftes Laufzeitsystem zurückgreifen. *MPICH-G2*-Programme sind hierdurch auf einer Vielzahl unterschiedlicher Architekturplattformen verfügbar und lauffähig [72]. Da der MPI-Standard durch die hier genannten Bibliotheken vollständig umgesetzt wird, ändert sich das grundsätzliche Programmiermodell nicht oder nur geringfügig. Aufgrund der vorausgesetzten Systemeigenschaften der Zielplattform werden dem Benutzer zum Teil einige zusätzliche Kontrollmöglichkeiten angeboten, um die speziellen Eigenschaften der MPI-Implementierung gezielt einsetzen zu können. Werden diese Kontrollmöglichkeiten nicht genutzt, d.h. es werden nur die im MPI-Standard definierten Funktionen benutzt, sind die Programme portabel und lassen sich auf unterschiedlichen Zielumgebungen einsetzen. Handelt es sich bei der Zielumgebung um ein System aus heterogenen Knoten, ist u.U. ein höherer Konfigurationsaufwand notwendig, da für jeden dieser Knoten eine plattformabhängige Version des Programmes erstellt werden muß. Dies gilt vor allem bei der Verwendung zusätzlicher Bibliotheken durch das Programm, was z.B. bei numerischen Simulationsanwendungen (Crashtests, Wettersimulation usw.) häufig vorkommen kann. In diesen Fällen können vielfältige Abhängigkeiten zu anderen Softwarekomponenten bestehen, die zum Teil erst während der Laufzeit des Programmes aufgedeckt werden.

Neben dem MPI-Standard gibt es noch eine ganze Reihe weiterer Alternativen (z.B. *PVM* [109], *Orca* [12], *UPC* [131], *Satin* [98] oder *JavaParty* [53]). Mit diesen kann man in der Regel ebenfalls gute Resultate in verteilten Systemumgebungen erzielen. Das jeweils angebotene Programmiermodell mit der zugehörigen Laufzeitumgebung ist aber meist für eine bestimmte Systemumgebung entworfen. Als Konsequenz hieraus ist eine so erstellte Software oft nicht ohne Änderungen im Quellcode auf eine andere Zielplattform portierbar. Die notwendigen Anpassungen erfordern die genaue Kenntnis über die zur Verfügung stehenden Mechanismen und über die zum Teil komplexen Zusammenhänge im verwendeten Programmiermodell.

Ziel dieser Arbeit ist es, eine Programmierumgebung zu entwickeln, deren Programmiermodell sich für *distributed memory* und *shared memory* Architekturen gleichermaßen eignet. Dies schließt auch hybride und heterogene Systemumgebungen aus mehreren Multiprozessorknoten mit ein.

Die für diese Programmierumgebung erstellten Anwendungen sollen ohne Änderungen im Quellcode auf möglichst vielen Computersystemen mit unterschiedlichen Architekturen lauffähig sein. Das bisher oftmals verwendete Paradigma einer Anpassung der Anwendungssoftware an geänderte Systemeigenschaften oder veränderte Anforderungen (z.B. eine manipulationssichere Kommunikation) soll hier durch die Anpassung der Programmierumgebung bzw. des Laufzeitsystems an die erforderlichen Gegebenheiten ersetzt werden. Dadurch wird es für die Anwendungen möglich, ohne Änderungen am Quellcode auf einer Vielzahl von Systemen effizient lauffähig zu sein.

⁴<http://www3.niu.edu/mpi/>

⁵<http://www.globus.org/>

Im nachfolgenden Abschnitt werden die Anforderungen an eine solche Programmierumgebung in einem kurzen Überblick zusammenfassend betrachtet.

Programmiermodell und Programmierumgebung

Dem Programmierer soll eine weitestgehend automatisiert arbeitende parallele Umgebung angeboten werden. Das zu entwickelnde Framework setzt sich zum Ziel, ein aus mehreren heterogenen Rechnerknoten bestehendes Netz zu einem virtuellen Rechner zu verbinden. Weiterhin sollen die folgenden Eigenschaften erfüllt werden:

⇒ Bereitstellung eines verständlichen und leicht zu verwendenden **Programmiermodelles für die Spezifizierung von parallelen Kontrollflüssen**:

Das Threadmodell und das Modell der nachrichtenbasierten Programmierung sollen gleichzeitig unterstützt werden. Da beide Modelle weit verbreitet sind, wird die schnelle Einarbeitung durch den Anwender in das hier definierte Programmiermodell erleichtert. Weiterhin können bereits für andere Programmiermodelle implementierte Algorithmen⁶ leichter portiert werden. Im Kontext des hier vorgestellten erweiterten Programmiermodelles werden die parallelen Kontrollflüsse als Trails bezeichnet.

⇒ Unterstützung eines zentralen Speichermodelles für die vom Benutzer definierten Kontrollflüsse:

Dieses Speichermodell soll, unabhängig von der Systemkonfiguration der Ausführungsplattform (z.B. mehrere durch ein Netzwerk verbundene Knoten), die Sicht auf einen **gemeinsamen virtuellen Speicher** ermöglichen. Die Existenz eines solchen globalen gemeinsamen Speicherbereiches ist notwendig, um ein dem Threadmodell nachempfundenen Programmiermodell in verteilten Ausführungsumgebungen bereitstellen zu können.

⇒ Flexible **Konfiguration** und leichte **Anpassbarkeit** der Laufzeitumgebung an unterschiedliche Anforderungen:

Der gesamte notwendige Funktionsumfang der Laufzeitumgebung soll durch mehrere in einem Schichtenmodell angeordnete und logisch unabhängige Softwarekomponenten bereitgestellt werden. Jede dieser Komponenten verfügt über einen festgelegten Funktionsumfang. Das Verhalten, d.h. die zu erwartenden Aktionen beim Aufruf einer Funktion, sowie die zugehörigen Schnittstellen werden ebenfalls definiert. Hierdurch wird es möglich, einzelne Softwarekomponenten auszutauschen und durch äquivalente Entwicklungen, z.B. des Benutzers, zu ersetzen. Durch eine konsequente Trennung der einzelnen Funktionsblöcke ist neben einer vollständigen Neuimplementierung einzelner Softwarekomponenten auch die Verwendung bereits bestehender und externer Softwarebibliotheken denkbar. Beispielsweise könnte in einem Benutzerszenario eine optimierte Kommunikationsbibliothek für die Nachrichtenübertragung zwischen einzelnen Rechnerknoten existieren. Der Programmie-

⁶gemeint sind hier thread- oder nachrichtenparallele Implementierungen

rer soll diese Bibliothek nutzen können, indem einmalig eine sogenannte Wrapperimplementierung erstellt wird, welche alle Aufrufe des Laufzeitsystems an diese Bibliothek weiterleitet. Dieses Vorgehen reduziert den notwendigen Entwicklungsaufwand bei der Anpassung und Optimierung der Laufzeitumgebung an Systemgegebenheiten der Ausführungsplattform durch den Anwender. Gleichzeitig können somit mehrere Implementierungen für einen Funktionsblock⁷ bereitgestellt und durch den Anwender ausgewählt werden.

Durch die Programmierumgebung (Programmiermodell und Laufzeitsystem) soll der Programmierer in die Lage versetzt werden, Anwendungen für Systeme zu schreiben, ohne den konkreten Aufbau des zur Laufzeit verfügbaren Rechnersystems kennen zu müssen. Dies soll es dem Programmierer ermöglichen, sich besser auf die wesentlichen Dinge des zu implementierenden Verfahrens zu konzentrieren. Zur weiteren Unterstützung dieser Modellphilosophie verzichtet das zur Verfügung stehende hybride Programmiermodell aus thread- und nachrichtenbasierter Programmierung auf die Definition von zusätzlichen und zum Teil komplexen Mechanismen, wie z.B. den *divide and conquer* Anweisungen in einem verteilten *Satin* Programm [98, 138].

1.2 Aufbau und weitere Gliederung

Die vorliegende Arbeit gliedert sich wie folgt: nach einer kurzen Übersicht über aktuelle Rechner- und Prozessorarchitekturen beschäftigt sich das hieran anschließende Kapitel mit Programmiermodellen und Programmierumgebungen für parallele Berechnungen. Vorrangig finden sich hier aktuelle Ansätze und Entwicklungen, aber auch bereits gefestigte und bewährte Konzepte, welche im Zusammenhang mit der in dieser Arbeit formulierten Programmierumgebung stehen. Vertiefend werden dort auch einige Aspekte der Java-Programmiersprache und einer dazugehörigen Laufzeitumgebung von Sun Microsystems betrachtet, da die entwickelte Prototypimplementierung der hier beschriebenen Laufzeitumgebung auf der Basis von Java erstellt wurde.

Nach diesem einführenden Überblick erfolgt im Kapitel 3 die allgemeine Beschreibung des zum Erreichen der Zielsetzung erarbeiteten parallelen Programmiermodelles und der zugehörigen Programmierumgebung. Hierfür werden der Aufbau der Laufzeitumgebung, welche aus einzelnen Softwarekomponenten besteht, erläutert und die erforderlichen Funktionen bzw. Mechanismen beschrieben. Das sich daran anschließenden Kapitel 4 beschreibt eine Prototypimplementierung für das im vorangegangenen Kapitel vorgestellte Modell und kommentiert die getroffenen Designentscheidungen bei der Umsetzung einzelner Softwarekomponenten. Zusätzlich werden bei der Betrachtung der Softwarekomponenten und Systembestandteile die zur Verfügung gestellten Benutzerschnittstellen dokumentiert und anhand kurzer Quellcodebeispiele erläutert.

In Kapitel 5 erfolgt eine Laufzeitanalyse der Grundfunktionalitäten der erstellten Prototypimplementierung. Da das Laufzeitsystem der Prototypimplementierung modular

⁷z.B. die angesprochene Nachrichtenübertragung im Netzwerk

aufgebaut ist, werden neben der Betrachtung des Gesamtsystemes auch einzelne Komponenten der Laufzeitumgebung getrennt untersucht. Vertiefend wird hierbei auf die Komponenten für die Kommunikation zwischen Knoten in einem verteiltem System und für die Bereitstellung eines gemeinsamen Speichersystems auf allen Knoten eingegangen. Die daraus gewonnenen Erkenntnisse helfen bei der Einteilung und Bewertung des Gesamtsystems.

In den dann folgenden Kapiteln 6 bis 8 wird anhand konkreter Fallbeispiele aus dem Gebiet der numerischen Simulation das Verhalten der Laufzeitumgebung mit praktischen Anwendungen untersucht. Insbesondere werden hier die im Vorfeld gestellten Zielsetzungen⁸ evaluiert, indem zunächst sequentielle und threadparallele Implementierungen der ausgewählten Algorithmen erstellt werden. Aus diesen Implementierungen werden dann jeweils die Varianten für die hier beschriebene parallele Programmierumgebung abgeleitet. Die durch das beschriebene Vorgehen erhaltenen Programmvarianten für die Prototypimplementierung werden danach in unterschiedlichen Testszenarien analysiert und ihr Verhalten dokumentiert.

⁸z.B. direkte Portierbarkeit aus dem Threadmodell

Kapitel 2

Architekturen und Programmiermodelle

We can only see a short distance ahead, but we can see plenty there that needs to be done.

(Alan Turing)

Die existierenden und am weitesten verbreiteten parallelen Programmiermodelle orientieren sich stark an der zur Verfügung stehenden Hardware. Deshalb sollen hier zunächst die wichtigsten Unterscheidungskriterien für parallele Architekturen präsentiert und eine kurze Klassifizierung vorgenommen werden. Anschließend werden einige parallele Programmiermodelle vorgestellt und kurz besprochen. Für einen ausführlichen Überblick zu dieser Thematik sei auf [109] verwiesen.

Vertiefend werden in diesem Kapitel Speicher- und Konsistenzmodelle sowie Modelle für transparente Speicherzugriffe in verteilten Systemumgebungen behandelt. Bei der in Kapitel 3 vorgestellten und im Rahmen dieser Arbeit entstandenen Spezifikation einer hybriden Programmierumgebung werden die hier betrachteten Modelle bzw. Konzepte zum Teil verwendet, ohne daß eine gesonderte Beschreibung erfolgt.

Am Ende des Kapitels wird auf einige Aspekte der Programmiersprache Java und deren Laufzeitumgebung eingegangen, da hiermit die Prototypimplementierung der vorgestellten hybriden Programmierumgebung (Kapitel 4 ff.) und die verschiedenen Testanwendungen für diesen Prototyp realisiert wurden.

2.1 Architekturen

Zur allgemeinen Einteilung von Rechnersystemen wird oft die Definition von Flynn [37] herangezogen. Diese erlaubt die grobe Klassifizierung von Architekturen anhand der Anzahl von vorhandenen Befehls- (*instruction streams*) und Datenströmen (*data streams*). Dabei wird ein breites Spektrum an möglichen Architekturen abgedeckt und es werden insgesamt vier Klassen definiert:

SISD *Single Instruction Single Data*

In diese Gruppe lassen sich die klassischen Einprozessor-Rechner einordnen, welche alle Aufgaben sequentiell abarbeiten und nach der Von-Neumann¹- oder der Harvard-Architektur² aufgebaut sind.

¹Programme und Daten im gleichen Speicher

²Programme und Daten in jeweils getrennten Speichern

SIMD *Single Instruction Multiple Data*

Zu dieser Klasse kann die Familie der Vektorrechner gezählt werden. Solche Systeme wenden einen einzelnen Befehl gleichzeitig auf mehrere Datenströme an, weshalb sie auch oftmals für die Verarbeitung von Multimediadaten eingesetzt werden. Eine neuere Unterklasse sind **SIMT** (*Single Instruction Multiple Threads*) Systeme. Der wesentliche Unterschied zu klassischen SIMD Systemen ist die Tatsache, daß die Anzahl der parallelen Vektoreinheiten vorher nicht festgelegt werden muß bzw. kann [85].

MISD *Multiple Instruction Single Data*

Rechner dieses Typs wenden verschiedene Instruktionen gleichzeitig auf ein Datum an und manipulieren dieses. Das Vorgehen erscheint wenig sinnvoll und praktikabel. Teilweise werden fehlertolerante Systeme aber als *MISD* klassifiziert. Hierbei bezieht man sich auf die Tatsache, daß solche Systeme alle Berechnungen redundant vornehmen und bei einem Ausfall einer Teilkomponente keine Datenverluste entstehen. Ob die vom klassischen Modell vorgesehene *Multiple Instruction* Eigenschaft hiermit erfüllt ist, bleibt von der jeweiligen Sichtweise abhängig.

MIMD *Multiple Instruction Multiple Data*

Dieses Modell ist heute am weitesten verbreitet. Es gibt hier mehrere Verarbeitungseinheiten, die auf mehrere Programmspeicher und Datenspeicher zugreifen können. In jedem Zyklus verwendet eine Berechnungseinheit eine eventuell unterschiedliche Instruktion auf ein beliebiges Datum an.

2.2 Speicherorganisation

Neben der Art und Weise wie Berechnungen in einem Rechnersystem ausgeführt werden, ist die Organisation des Hauptspeichers und dessen Verfügbarkeit für die einzelnen Berechnungseinheiten eine weitere Möglichkeit zur Unterscheidung und Klassifizierung. Prinzipiell lassen sich zwei große Gruppen unterscheiden: gemeinsamer (*shared*) und verteilter (*distributed*) Speicher. Zur Beschleunigung von Hauptspeicherzugriffen existieren aber in beiden Ansätzen schnelle Zwischenspeicher, sog. Caches oder Cache-Speicher, auf die hier zuerst eingegangen werden soll.

In Kapitel 3 und Kapitel 4 werden die hier beschriebenen Konzepte zum Teil wieder aufgegriffen und verwendet.

2.2.1 Cache-Speicher

Die Zugriffszeiten auf den Hauptspeicher sind im Vergleich zur Prozessorgeschwindigkeit sehr hoch. Daher kommt es bei direkten Hauptspeicherzugriffen oftmals zu Wartezyklen des Prozessors. Häufige Speicherzugriffe führen deshalb zu einer hohen Zahl an unerwünschten Wartezyklen und zu einem Ansteigen der Programmausführungszeiten. Neben der Überdeckung dieser Wartezyklen mit anderen unabhängigen Berechnungen

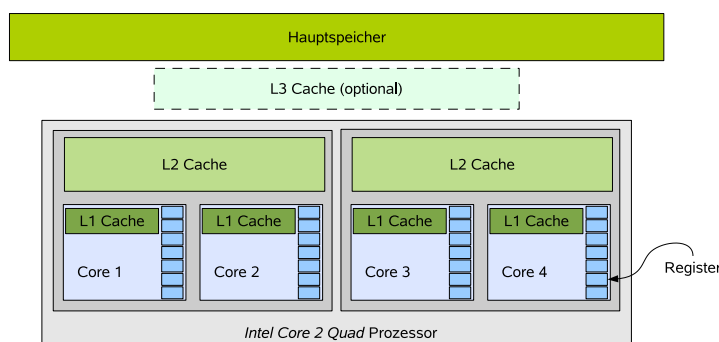


Abbildung 2.1: Schematische Darstellung der Speicherhierarchie aus Prozessorregistern, Cache-Speichern und Hauptspeicher am Beispiel des *Intel Core 2 Quad* Prozessors [66].

des Prozessors versucht man, die Anzahl der Wartezyklen zu minimieren. Auf der Hardwareebene haben sich Cache-Speicher (*Caches*) zur Reduzierung der Speicherzugriffszeiten etabliert. Im Vergleich zum normalen Hauptspeicher verfügen diese Cache-Speicher über eine um ein Vielfaches höhere Zugriffsgeschwindigkeit, bieten aber aufgrund höherer Hardwarekosten sehr viel kleinere Kapazitäten. Vom Prozessor häufig benutzte Daten werden in einem Cache zwischengespeichert und zu einem späteren Zeitpunkt mit dem Hauptspeicher abgeglichen. Mehrere hintereinander geschachtelte Cache-Speichermodule (z.B. Level 1 und Level 2 Caches) bilden eine Cache-Hierarchie (siehe *Abbildung 2.1*).

Cache-Fehlzugriff

Ein Cache-Speicher arbeitet transparent im Hintergrund, ohne daß er explizit durch den Programmierer angesprochen werden muß. Da die Kapazität des Cache-Speichers weitaus geringer als die des Hauptspeichers ist, kann nur eine begrenzte Menge an Daten zwischengespeichert werden. Ein Cache muß deshalb eine Zuordnung von Hauptspeicheradressen zu den aktuell im Cache-Speicher vorhandenen Einträgen vornehmen. Typischerweise enthält ein Eintrag im Cache mehrere aufeinanderfolgende Bytes des Hauptspeichers (*Cache-Zeile*). Bei einer Speicheranfrage an den Hauptspeicher wird zuerst im Cache nachgesehen, ob sich die angeforderten Daten dort befinden. Ist dies der Fall, so spricht man von einem *Cache-Treffer* und die Anfrage kann sofort durch den Cache beantwortet werden. Für den Fall, daß die angefragten Daten nicht vorhanden oder ungültig sind, wird ein *Cache-Fehlzugriff* erzeugt. Je nach Charakteristik des aufgetretenen Fehlers kann dieser in verschiedene Gruppen eingeteilt und eventuell auch unterschiedlich behandelt werden. Manche der *Cache-Fehlzugriffe* sind vermeidbar und das Laufzeitsystem bzw. der Compiler sollten bestrebt sein, diese zu minimieren. Bei Systemen mit einem Prozessor(-kern) unterscheidet man zwischen: Kapazitäts- (*capacity miss*), Kaltstart- (*compulsory miss*) und Konflikt-Fehlzugriff (*conflict miss*) [55]. Sind mehrere Prozessoren bzw. Kerne beteiligt, können noch zusätzliche Kohärenz-Fehlzugriffe auftreten, wenn die Prozessoren auf gemeinsame Daten konkurrierend zugreifen. Diese Daten existieren dann mit sehr hoher Wahrscheinlichkeit jeweils repliziert in den einzel-

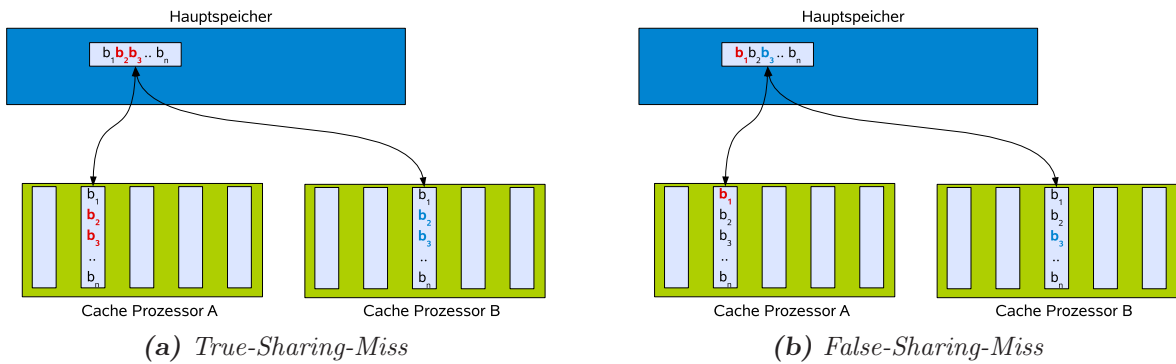


Abbildung 2.2: Illustration von *True-Sharing-Miss* und *False-Sharing-Miss*.

nen Prozessorcaches.

Die möglichen Kohärenz-Fehlzugriffe lassen sich in *True-Sharing-Miss* und *False-Sharing-Miss* unterteilen [34]. In beiden Fällen wird davon ausgegangen, daß ein Zugriff auf Daten des Hauptspeichers erfolgt, welche in allen oder mehreren Cache-Speichern der beteiligten Prozessoren vorhanden sind (siehe *Abbildung 2.2*).

True-Sharing-Miss: Nachdem ein Prozessor *A* Teile der Daten (b_2b_3) einer *Cache-Zeile* verändert hat, greift ein anderer Prozessor *B* genau auf diese Daten (b_2b_3) lesend zu. In Folge dessen wird ein Cache-Fehlzugriff erzeugt und die gesamte *Cache-Zeile* im Cache von Prozessor *B* wird erneuert.

False-Sharing-Miss: In diesem Fall modifiziert wiederum der Prozessor *A* einen Teil der Daten (b_1) einer *Cache-Zeile*. Der Zugriff des Prozessors *B* erfolgt jedoch auf ein anderes Datum (b_3), welches sich in der gleichen *Cache-Zeile* wie das von Prozessor *A* veränderte Datum (b_1) befindet. Da eine *Cache-Zeile* einen zusammenhängenden Block aus mehreren Bytes des Hauptspeichers enthält, muß der Inhalt im Cache des Prozessors *B* erneuert werden, obwohl die angeforderten Daten nicht verändert wurden. Das System kann nicht erkennen, daß es sich hierbei um einen unabhängigen Zugriff handelt und markiert deshalb die *Cache-Zeile* im Cache von Prozessor *B* als ungültig. Der so erzeugte Datentransfer ist nicht zwingend erforderlich und verursacht einen unnötigen Overhead.

Die genauen Laufzeiten eines beliebigen Programmes sind bei Benutzung von Caches nur schwer voraussagbar und hängen stark vom erzeugten Zugriffsmuster auf den Hauptspeicher ab.

Cache-Kohärenz

Um zu verhindern, daß die Cache-Speicher (der verschiedenen Prozessoren bzw. Kerne) in einer Multiprozessorumgebung zueinander abweichende Werte für eine Speicheradresse liefern, können verschiedene Verfahren zur Vermeidung solcher Inkonsistenzen eingesetzt

werden. Im Grunde gibt es zwei große Klassen in die sich solche Verfahren, die auch als Kohärenz-Protokoll bezeichnet werden, einteilen lassen:

Verzeichnisdienst: In einer zentralen Liste wird der Status aller in den Caches befindlichen Datenblöcke (Cache-Zeile) geführt. Ausgehend von diesen Statusangaben kann entschieden werden, ob die angeforderte Speicheroperation (Lesen, Schreiben) sofort durchgeführt werden kann oder ob erst noch Interaktionen mit anderen Cache-Speichern durchgeführt werden müssen (Invalidierung von Kopien).

Bus-Snooping: Diese Verfahren gehen von einem gemeinsamen Übertragungsmedium (Bus) für alle Speicheroperationen aus. Ein Cache-Speicher kann somit alle Speicheranfragen der angeschlossenen Teilnehmer lesen und seinen eigenen Inhalt entsprechend anpassen. Bei einer Schreiboperation können z.B. die im Cache enthaltenen Daten entfernt oder aktualisiert werden.

Mit zunehmender Anzahl von angeschlossenen Cache-Speichern wächst in aller Regel auch die Zahl der zu verarbeitenden Nachrichten. Deshalb ist das verwendete Bussystem der limitierende Faktor für die Skalierung solcher Protokolle, da immer nur eine begrenzte Anzahl von Nachrichten in einer bestimmten Zeitspanne übertragen werden können.

2.2.2 Gemeinsamer Speicher

Systeme mit gemeinsamen Speicher stellen einen globalen Adreßraum bereit, der von allen Prozessoren im System gleichermaßen benutzt werden kann. Die letztendliche Realisierung als Hardware muß nicht zwangsläufig aus einer einzigen Speichereinheit bestehen, sondern kann sich über verschiedene Knoten mit jeweils eigenen Speicherbausteinen erstrecken. Dieser eigentlich verteilt vorliegende Speicher wird dann durch die Hardware zusammengefaßt. Hieraus ergibt sich die folgende Unterteilung:

UMA (*uniform memory access*): Hier greifen alle Prozessoren gleichermaßen auf den gemeinsamen Speicher zu und die Zugriffszeiten sind für alle Prozessoren nahezu identisch.

NUMA (*non uniform memory access*): Zugriffe auf die Daten sind hier abhängig vom jeweiligen Speicherort und können in Abhängigkeit vom anfragenden Prozessor variieren. Die einzelnen Speichermodule sind physikalisch auf die Prozessoren verteilt. Speicherzugriffe auf Daten, die sich nicht im lokalen Speichermodul des anfragenden Prozessors befinden, benötigen in aller Regel mehr Zeit als direkte Zugriffe auf ein lokales Speichermodul, da eine Kommunikation zwischen den Speichermodulen notwendig wird.

Nimmt man die bereits erwähnten Cache-Speicher in die Betrachtungen mit auf, kann man das NUMA Modell noch in *CC-NUMA*, *COMA* und *NCC-NUMA* unterteilen.

CC-NUMA (*cache coherent - NUMA*): Bei Systemen dieses Types ist die Kohärenz der einzelnen Cache-Speicher sichergestellt. Ein Zugriff auf ein entferntes

Speicherwort löst einen Cache-Fehlzugriff³ aus und der gesamte Cacheblock des entfernten Speichers wird in den eigenen lokalen Cache übertragen. Dieser Vorgang basiert häufig auf der Grundlage von verzeichnisbasierten Kohärenzprotokollen⁴ [70].

COMA (*cache only memory architecture*): Das COMA Modell ist ein Spezialfall von CC-NUMA, wobei die lokalen Cache-Speicher den physikalisch verteilten Speicher darstellen. Der gesamte Hauptspeicher des Rechners besitzt keine herkömmlichen Speichermodule sondern besteht nur aus Caches.

NCC-NUMA (*non cache coherent - NUMA*): Solche Systeme unterscheiden zwischen lokalen Zugriffen auf den Cache und globalen Speicherzugriffen auf den Hauptspeicher. Globale Zugriffe werden ohne Benutzung des Cache durchgeführt. Die Sicherstellung der Cache-Kohärenz muß durch eine Softwareschicht übernommen werden.

Ein noch im aktiven Forschungsstadium befindliches Hauptspeicherkonzept ist ein als **Transactional Memory** bezeichneter Ansatz. Die Synchronisation und Koordinaten von parallelen Berechnungen soll hierbei (fast) vollständig vom Compiler oder der vorhandenen Hardware übernommen werden. Man unterscheidet zwischen softwarebasierten und hardwarebasierten Implementierungen. Ausführliche Informationen sind z.B. in [21, 51, 78] zu finden.

Architekturen für Parallelrechner und deren Prozessoren

Bereits in den Anfängen des Computerzeitalters wurden Rechnersysteme mit mehreren Prozessoren entworfen und eingesetzt (UNIVAC LARC⁵). Durch mehrfach vorhandene Prozessoren wird eine parallele Abarbeitung der notwendigen Berechnungen einer Anwendung ermöglicht und im Allgemeinen sinkt die Ausführungszeit im Vergleich zur Ausführung einer äquivalenten Anwendung in einem sequentiellen System.

Der nun folgende Abschnitt wird einen kurzen Einblick in die bis heute entstandene Vielfalt von parallelen Architekturansätzen geben. Die angesprochenen Varianten beziehen sich dabei auf Systeme mit gemeinsamem Speicher.

Multiprozessorsysteme Unter einem Multiprozessorsystem versteht man ein Computersystem mit mehr als einem Prozessor. Jeder in einem solchen System vorhandene Prozessor ist dabei eigenständig und verfügt z.B. über eigene Speicheranbindungen. Je nach Struktur und Verwendung der einzelnen Prozessoren im Gesamtsystem kann man eine Aufteilung in *asymmetrische* und *symmetrische* Multiprozessorsysteme vornehmen. **Asymmetrische Multiprozessorsysteme** bestehen oft aus unterschiedlichen Prozessoren. Jedem der Prozessoren werden hierbei eine oder mehrere feste Aufgaben zugeteilt.

³siehe Kapitel 2.2.1

⁴siehe auch Kapitel 2.2.1

⁵Vom UNIVAC LARC System (*Livermore Advanced Research Computer*) wurden 1960 zwei Exemplare gebaut, die jeweils zwei CPUs für arithmetische Berechnungen besaßen [134].

Deshalb können die Prozessoren voneinander abweichen, da z.B. die Prozessoren auf die ihnen zugeordneten Aufgaben spezialisiert sein können (I/O-Prozessor). Solche Systeme haben ihren Ursprung in der Anfangszeit der Parallelrechner, da sie neben anderen Aspekten⁶ auch die Handhabung und Programmierung im Betriebssystem (OpenVMS von DEC oder IBM System/360 Operating System) vereinfachen. Heute spielen asymmetrische Multiprozessorsysteme kaum noch eine Rolle.

Ein **symmetrisches Multiprozessorsystem** (SMP) besitzt zwei oder mehr identische Prozessoren, die alle auf einen gemeinsamen Adreßraum zugreifen können. Jeder der Prozessoren ist gleichberechtigt, d.h. alle laufenden Berechnungen können dynamisch auf die verfügbaren Prozessoren verteilt werden. Probleme entstehen, wenn diese Berechnungen fortlaufend einem anderen Prozessor zugewiesen werden. In diesen Fällen kommt es zu ständigen Cache-Fehlzugriffen und die Daten müssen aus dem relativ langsamen Hauptspeicher nachgeladen werden. Unnötige Wartezeiten der Prozessoren sind dann die Folge. Diese können aber durch entsprechende Vorkehrungen des Betriebssystems vermieden werden. Darüber hinaus führt die architekturbedingte Anbindung der Prozessoren über ein Bussystem an den Hauptspeicher zu einer begrenzten Skalierbarkeit bzw. Erweiterbarkeit mit zusätzlichen Prozessoren.

Vektorrechner Parallelrechner, welche den *SIMD*-Ansatz verfolgen, werden als Vektorrechner bezeichnet. Die hier eingesetzten Prozessoren wenden eine Operation gleichzeitig auf mehrere Daten an. Bereits ein einzelner Prozessor ist somit für die parallele Abarbeitung geeignet. Um das parallele Verarbeitungspotential weiter zu steigern, ist ein Rechnersystem nicht auf einen Vektorprozessor beschränkt und kann aus tausend oder mehr Einzelprozessoren bestehen.

Vektorrechner waren lange Zeit eine der dominierenden Architekturen im Bereich des *High Performance Computing* (HPC) [89]. Ein bekannter Vertreter solcher Systeme ist der japanische *Earth Simulator*, der einige Jahre lang die Liste der leistungsfähigsten Parallelrechner⁷ anführte. Abgesehen von diesem Anwendungsfeld finden sich heute nahezu in jedem PC Vektoreinheiten in Form von Grafikprozessoren (*GPUs*) [85].

Prozessoren mit mehreren Kernen Mit der Einführung der als **Hyper-Threading** bezeichneten Technologie begann der Halbleiterkonzern Intel ein Simultaneous Multi-threading (SMT) Konzept [129] in seinen Prozessoren einzusetzen. Hiermit können Wartezeiten eines Prozessors effizienter überdeckt werden. Beim *Hyper-Threading* existiert eine hardwareseitige Unterstützung für Threads, die durch mehrere vollständige Registersätze realisiert wird. Zwei Threads können im Idealfall parallel ausgeführt werden, da in den Wartezeiten des einen Threads automatisch auf den anderen Thread umgeschaltet wird. Ein Prozessor simuliert auf diese Weise zwei oder mehr logische Prozessoren [88, 109].

Mit dem Erscheinen der ersten *DualCore* Prozessoren von Intel verschwand das *Hyper-*

⁶Verringerung der Komplexität eines Prozessors

⁷www.top500.org

Threading zunächst, wird aber in den aktuellen *Core i7* Prozessoren⁸ wieder unterstützt⁹, da es eine relativ einfache Methode zur Beschleunigung von multithreaded Anwendungen ist. Untersuchungen hierzu haben gezeigt, daß solche Anwendungen durch den Einsatz von *Hyper-Threading* im Schnitt um ca. 15% beschleunigt werden können [65, 88].

Heutige Prozessoren von Intel oder anderen Prozessorherstellern sind meist sogenannte **Multicore**-Prozessoren. Die oder der in einem System vorhandene *Multicore*-Prozessor besitzt dabei mehrere identische Kerne auf seiner Chipfläche. Jeder der Kerne ist dabei ein vollwertiger Prozessor, lediglich einige Funktionseinheiten, wie z.B. Schnittstellen nach Außen oder Caches¹⁰, werden gemeinsam verwendet.

Analog hierzu gibt es auch einige Multicore-Prozessoren mit verschiedenen Kernen (*asymmetric multicore*). Die einzelnen Kerne übernehmen dabei unterschiedliche Aufgaben und sind meist nur für bestimmte Problemstellungen geeignet. Der von IBM angebotene *Cell* Prozessor ist ein Vertreter dieser Klasse von Prozessoren. Er besitzt einen vollwertigen Hauptprozessor (PPE) auf der Basis der IBM PowerPC Architektur und meist¹¹ 8 weitere, untergeordnete SIMD-Kerne (SPE) [50, 136].

Manycore-Architekturen besitzen eine wesentlich höhere Anzahl von Kernen auf einem einzelnen Prozessorchip und sind der nächste logische Schritt nach den Multicore-Prozessoren. Eine genaue Abgrenzung zwischen Manycore und Multicore ist schwierig. Allgemein spricht man heute von Manycore, wenn mehrere 10 oder 100 Kerne vorhanden sind.

Zukünftige *Manycore*-Modelle von Intel könnten außer dem Cache-Speicher noch ein zusätzliches Speichermodul auf dem Prozessor enthalten. Dort sollen die einzelnen Kerne z.B. Zwischenergebnisse puffern oder mit benachbarten Kernen schnell austauschen können [104, 113].

2.2.3 verteilter Speicher

Computersysteme mit verteiltem Speicher bestehen aus mehreren selbständigen Berechnungseinheiten und sind über ein Netzwerk miteinander verbunden. Einzelne Berechnungseinheiten werden dabei meist als Knoten bezeichnet und verfügen über einen eigenen Hauptspeicher, auf den ein anderer Knoten keinen direkten Zugriff hat. Der Austausch von Daten zwischen den Knoten ist deshalb nur durch explizite Kommunikation möglich. Die für diese Kommunikation verwendeten Verbindungsnetzwerke können sehr unterschiedlich ausfallen. Die Möglichkeiten reichen von einem spezialisierten und optimierten Bussystem bis hin zum heute weit verbreiteten TCP/IP-Netz. Auch für die Knoten gibt es hier vielfältige Varianten, da prinzipiell alle im Abschnitt 2.2.2 erwähnten

⁸Nehalem Architektur, Ende 2008

⁹z.T. auch unter der Bezeichnung *Simultaneous Multi-Threading* (SMT)

¹⁰IBM Power4 Prozessoren verwenden sog. Shared Caches.

¹¹Die in der Sony Playstation 3 eingesetzten Cell Prozessoren verwenden/besitzen nur 7 SPEs. Toshiba vermarktet seine Qosmio Notebooks mit zusätzlich eingebautem *Quad Core HD Prozessor*, welcher ein Cell Prozessor mit 4 SPEs ist. <http://en.cellusersgroup.com/modules/spursengine/>

Architekturen bzw. Prozessoren kombiniert werden können. In der Regel verwendet man jedoch homogene Knoten, d.h. die verwendeten Prozessoren und die jeweilige Anzahl sind auf allen Knoten identisch. Der Spitzenreiter der Top500-Liste¹² zu Beginn des Jahres 2009, das IBM RoadRunner System im Los Alamos National Laboratory, ist ein hybrider InfiniBand-Cluster. Jeder Knoten besitzt zwei DualCore AMD Opteron Prozessoren, die jeweils zwei zusätzliche PowerXCell-8i Prozessoren ansprechen können¹³.

Verteilte Systeme

Verteilte Systeme lassen sich ebenfalls als Systeme mit verteiltem Speicher klassifizieren. Sie unterscheiden sich aber in einigen Aspekten von parallelen Systemen mit verteiltem Speicher, welche im Regelfall als Einzelsystem konzipiert sind (z.B. *Cray T3E*). Nach Tanenbaum [125] ist ein *verteilt System* definiert als:

A distributed system is a collection of independent computers that appears to its users as a single coherent system.

Dem Benutzer soll, gemäß dieser Definition, eine einheitliche und kohärente Sicht auf ein System aus mehreren unabhängigen Computerknoten präsentiert werden. Anhand des Aufbaus aus einzelnen Computerknoten läßt sich zwischen *homogenen* und *heterogenen* Systemen unterscheiden. Diese Aufteilung ist sehr allgemein und erlaubt eine sehr freizügige Einordnung in die Klasse der *heterogenen* Systeme. An dieser Stelle soll deshalb die Klassifizierung etwas abgeändert und eine neue Gruppe *semi-heterogen* eingeführt werden.

homogen Alle Knoten sind weitestgehend identisch, insbesondere sind alle verwendeten Prozessoren gleich. Die Anzahl der Prozessoren auf einem Knoten unterscheidet sich nicht. Der auf den Knoten verfügbare Hauptspeicher kann eine unterschiedliche Größe aufweisen.

semi-heterogen Die Prozessoren der einzelnen Knoten können voneinander abweichen. Hierbei darf z.B. die Taktfrequenz oder die Anzahl der Kerne variieren. Alle Prozessoren müssen die gleiche Architektur aufweisen (z.B. *x86*).

heterogen Die einzelnen Knoten können jeweils eine andere Prozessorarchitektur besitzen. Alle sonstigen Systemparameter sind variabel.

Oftmals wird beim Einsatz *verteilter Systeme* auch der Begriff **Grid-Computing** verwendet. In der Definition von Ian Foster und Carl Kesselman [40] ist ein *Grid* charakterisiert als:

„... a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities ...“.

¹²www.top500.org

¹³http://www.lanl.gov/discover/world_s_fastest_computer

Der Schwerpunkt dieser Auslegung liegt in der standardisierten Bereitstellung von Ressourcen für rechenintensive Anwendungen. Allerdings müssen sich diese Ressourcen nicht zwangsläufig auf die Rechenkapazitäten im *Grid* beziehen. Vielmehr umfaßt dieser Begriff eine Vielzahl von Möglichkeiten, wie z.B. die Bereitstellung von Speicherplatz oder von spezialisierten Berechnungsverfahren. Das zugrundeliegende Prinzip folgt dabei der *Open Grid Service Architecture* (OGSA). Hierbei stellen die einzelnen Computersysteme in einem verteilten System verschiedene Service-Dienste bereit, die von einem Nutzer der Grid-Infrastruktur verwendet werden können [39].

2.3 Programmiermodelle und Laufzeitsysteme

Dieser Abschnitt gibt einen Überblick über Programmiermodelle und bezieht die dafür notwendigen Laufzeitsysteme mit ein. Dabei werden die Programmiermodelle für Systeme mit verteiltem Speicher und Systeme mit gemeinsamen Speicher besprochen. An dieser Stelle erfolgt auch die Beschreibung von Konzepten, die für die Koordination von parallelen Kontrollflüssen notwendig sind. Diese Konzepte und die vorher beschriebenen Programmiermodelle bilden die Grundlage für die im Rahmen dieser Arbeit entstandene Programmierumgebung.

Weiterhin geht dieser Abschnitt auf Hauptspeichersysteme ein, welche in einer Umgebung mit verteiltem Speicher dem Programmierer den Zugriff auf einen virtuellen gemeinsamen Speicher erlauben und nennt einige Beispiele für Umsetzungen solcher Systeme. Ein auf dem Konzept des virtuellen Hauptspeichers aufbauendes Konzept bzw. die Unterstützung hierfür ist auch in der entstandenen Programmierumgebung (*vgl. dazu Kapitel 3 (S.57)*) integriert.

Für die Definition von parallelen Berechnungen als Programmcode dominieren zwei prinzipielle Vorgehensweisen.

Datenparallel Durch eine oder mehrere Berechnungseinheiten werden gleiche Instruktionen auf unterschiedliche Daten parallel angewendet.

Taskparallel Zunächst voneinander unabhängige Instruktionsfolgen (Tasks) werden mit Hilfe von einer oder mehreren Berechnungseinheiten parallel ausgeführt. Ein Task kann selbst wieder parallelisierbar sein und dann eventuell *datenparallel* berechnet werden. In diesen Fällen spricht man auch von *gemischt-parallel*. Sind Tasks voneinander abhängig, müssen diese getrennt und nacheinander ausgeführt werden. Gibt es mehrere Tasks mit Abhängigkeiten, kann man die Zusammenhänge in einem Taskgraphen modellieren und versuchen, eine möglichst effektive Abarbeitungsreihenfolge zu finden [62].

Der Einsatz von *datenparalleler*, *taskparalleler* oder *gemischt-paralleler* Programmierung richtet sich nach dem Anforderungsprofil der jeweiligen Aufgabenstellung und der vorhandenen Computerhardware. Auf die jeweiligen Vorzüge und Einsatzfelder soll hier nicht weiter eingegangen werden. Eine ausführlichere Betrachtung ist z.B. in [109] zu finden.

Eigenschaften des Programmiermodelles und der vorhandenen Systemumgebung (Hardware, Betriebssystem) können bei der Ausführung eines Programmes mit mehreren Kontrollflüssen ungewollte Effekte verursachen, die bei der Programmierung mit nur einem Kontrollfluß nicht in Erscheinung treten. Bei der parallelen Programmierung müssen solche Eigenschaften dem Programmierer bekannt sein, um Fehler vermeiden zu können. Ein Beispiel hierfür ist u.a. die Sichtbarkeit von Datenmanipulationen im Speicher, d.h. zu welchem(n) Zeitpunkt(en) besitzen die vorhandenen Kontrollflüsse eine identische Sicht auf den gemeinsamen Hauptspeicher und durch welche Mechanismen wird diese einheitliche Sicht generiert und sichergestellt.

Auch die Spezifikation von Java [45], die als Programmiersprache für die untersuchte Prototypimplementierung der in dieser Arbeit beschriebenen Programmierumgebung eingesetzt wurde, enthält solche Definitionen. Deshalb sollen an dieser Stelle einige Speicher- und Konsistenzmodelle vorgestellt werden. In Kapitel 2.4.2 wird dann auf die Programmiersprache Java und das zur Verfügung gestellte Laufzeitsystem näher eingegangen.

2.3.1 Speicher- und Konsistenzmodelle

Im Speichermodell (*memory model*) einer Programmiersprache wird die Art und Weise von Speicherzugriffen aller Art definiert. Darunter fällt auch die Behandlung von Variablen, die normalerweise im Hauptspeicher abgelegt sind, zur Bearbeitung aber in Prozessorregister geladen werden müssen.

Alle Hochsprachen, wie C, Cobol oder Fortran, wandeln den jeweiligen Sourcecode in eine für Maschinen lesbare Form um. Für den Anwender ist der gerade erwähnte Vorgang des Ladens von Variableninhalten in Register nicht direkt sichtbar, muß aber von den Compilern durchgeführt werden. Zur Optimierung der Programmlaufzeiten werden oft möglichst viele Register benutzt und bereits aus dem Hauptspeicher geladene Inhalte nicht aktualisiert. Teilweise kommt es auch zu einer Umsortierung der im eigentlichen Programmcode vorkommenden Operationen oder Variableninhalte werden im Voraus geladen [3, 30]. Solange bestimmte Regeln eingehalten werden, funktioniert dieses Vorgehen sehr gut. Dieser Regeln werden im Speichermodell (*memory model*) der Programmiersprache festgelegt und oftmals auch als Konsistenzmodell bezeichnet. Die so definierten Regeln können z.B. verlangen, daß sich die berechneten Ergebnisse mit und ohne Compilereingriff zu keinem Zeitpunkt unterscheiden dürfen. Für die bis vor einigen Jahren üblichen Architekturen mit nur einem Prozessor und den darauf ausgeführten sequentiellen Programmen ist dies noch relativ einfach zu realisieren. Sobald mehrere Berechnungsflüsse hinzu kommen, müssen die Compilereingriffe vorsichtiger gestaltet werden. Eine andere Möglichkeit ist die Abschwächung und Veränderung dieses strikten Konsistenzmodelles. Hierbei wird die Sicherstellung der Korrektheit der durchgeführten Berechnungen dem Programmierer übertragen. Im folgenden findet sich eine kurze Beschreibung verbreiteter Konsistenzmodelle.

strikt: Das am einfachsten umzusetzende Konsistenzmodell [1] ist die *strikte Konsistenz*. In diesem Modell werden alle Speicheroperationen nach dem „*first come - first serve*“-Prinzip abgearbeitet, so daß jede Leseoperation den zuletzt geschriebenen Wert

zurück gibt. Jeder Programmdurchlauf mit mehreren Kontrollflüssen kann aber unter Umständen andere Ergebnisse produzieren, da die Inhalte des gemeinsamen Speichers vom genauen zeitlichen Auftreten der durchgeführten Operationen der einzelnen Kontrollflüsse abhängig sind.

sequentiell: Die sequentielle Konsistenz geht zurück auf Lamport [77]. Verlangt wird eine identische Sicht aller parallelen Kontrollflüsse auf die Daten. Dabei dürfen sich die Datenmanipulationen eines einzelnen Kontrollflusses nicht überholen, bei der Betrachtung von Operationen verschiedener Kontrollflüsse ist dies jedoch möglich. Deshalb spielt das genaue zeitliche Auftreten der Operationen keine so starke Rolle wie im *strikten* Konsistenzmodell. Ein Compiler kann die Operationen eines Kontrollflusses zusammenfassen und erst zu einem späteren Zeitpunkt in den Speicher übertragen.

weak ordering: Beim *weak ordering* werden spezielle Synchronisationspunkte eingefügt. Unmittelbar nach einem solchen Punkt haben alle Teilnehmer eine identische Sicht auf das Speichersystem. Nach dem Absetzen neuer Operationen wird diese einheitliche Sicht nicht mehr garantiert.

release: Unter *release consistency* werden komplexere Konsistenzmodelle zusammengefaßt, die ebenfalls Synchronisationspunkte anbieten, aber zum Teil über mehrere Synchronisationsoperationen verfügen. Hierdurch ist eine differenziertere Steuerung möglich und die Performance kann verbessert werden.

Neben den genannten Modellen finden sich in der Literatur [1, 69] noch eine Vielzahl weiterer Ansätze. Diese sind dann oftmals für eine spezielle Aufgabenstellung optimiert und unterscheiden sich nur durch geringfügige Änderungen.

Die Konsistenzmodelle gewährleisten eine feste und für den Programmierer nachvollziehbare Ordnung der abgesetzten Speicherzugriffe bzw. stellen eine einheitliche Sicht auf den Speicher her. Findet ein konkurrierender Zugriff auf gemeinsame Datenstrukturen statt, muß trotzdem eine Synchronisation durch geeignete Kontrollmechanismen erfolgen.

2.3.2 Nachrichtenbasierte Programmierung - Message Passing

Das *Message Passing* Programmiermodell bietet die Sicht auf einen Parallelrechner mit verteiltem Speicher und ermöglicht die Kommunikation zwischen parallelen Kontrollflüssen. In diesem Abschnitt soll *MPI* als ein Vertreter dieses Programmiermodelles kurz vorgestellt werden.

Der **MPI** (*Message Passing Interface*) Standard [48, 116] ermöglicht die Kommunikation zwischen parallelen Kontrollflüssen auf Systemen mit verteiltem Speicher. Derartige Programme können aber auch auf *shared memory* Rechnern ausgeführt werden. Eine MPI-Anwendung besteht aus mehreren unabhängigen und untereinander kommunizierenden Prozessen. Diese Prozesse werden in der Regel beim Programmstart auf allen

Knoten zur gleichen Zeit gestartet und führen ein identisches Programm aus. Zur Identifikation wird jedem Prozess eine eindeutige ID zugeordnet. Diese ID wird zur gezielten Kommunikation verwendet, kann aber auch vom Programmierer für die Aufteilung der Berechnungen benutzt werden¹⁴. Für die Kommunikation stehen Punkt-zu-Punkt und globale Operationen zur Verfügung. Viele der zur Verfügung gestellten Kommunikationsoperationen sind blockierend, d.h. nach einem Aufruf einer solchen Operation kehrt die Kontrolle erst nach der Ausführung an den Aufrufer zurück. Man spricht daher oft davon, daß die Bereitschaft von Sender und Empfänger vorausgesetzt wird. Es existieren aber auch nichtblockierende Kommunikationsoperationen oder Funktionen zum Erfragen der Bereitschaft der Gegenseite.

Der MPI-Standard unterstützt keine Threads. Dies bedeutet, daß keine gesonderten MPI-Routinen für den Umgang mit Threads existieren. Es gibt für den Gebrauch von Threads aber einige Empfehlungen und Hinweise, welche von der Implementierung einer MPI-Laufzeitbibliothek angewendet werden sollten [48]. Unter anderen wird dort der *thread-sichere* Zugriff auf die bereitgestellten Methoden der Bibliothek empfohlen. Da diese Empfehlungen nicht fest im Standard verankert sind, muß der Programmierer bei der Verwendung von Threads z.B. die Kommunikation durch *Mutex*-Variablen schützen, sofern beliebige MPI-Implementierungen abgedeckt werden sollen. Unterbleibt diese Maßnahme, können sich Nachrichten überholen und es kann zu Programmfehlern kommen.

Message Passing Programme sind oftmals sehr effektiv und die benötigte Laufzeit sinkt mit zunehmender Anzahl von eingesetzten Prozessoren. Allerdings eignen sich nicht alle Algorithmen für das von MPI angebotene Programmiermodell, da die Lage der benötigten Daten bekannt bzw. berechenbar sein muß. Hierunter fallen z.B. irreguläre Algorithmen oder manche Zeitschrittverfahren, welche nach einer festen Anzahl von Rechenoperationen größere Datenmengen zwischen den Kontrollflüssen austauschen müssen, ehe mit weiteren Berechnungen fortgesetzt werden kann [56]. Neben klassischen Implementierungen für Systeme mit verteiltem Speicher existiert auch eine ganze Reihe von spezialisierten Umsetzungen. So handelt es sich z.B. bei GridMPI [123] und MPICH-G2 [72] um MPI-Varianten, welche für Grid-Umgebungen [40] konzipiert sind und deshalb auch die Nutzung von MPI in lose gekoppelten sowie heterogenen Systemumgebungen zulassen.

PVM (*Parallel Virtual Machine*)¹⁵ ist eine weitere Programmierumgebung für *Message Passing*. Ein paralleles PVM Programm wird hierbei durch Tasks beschrieben, welche untereinander kommunizieren und Daten austauschen können. Das Erzeugen zusätzlicher Tasks zur Laufzeit ist ebenfalls möglich und erlaubt eine dynamische Abarbeitung. Eine Einführung und Beschreibung der wichtigsten Konzepte ist z.B. in [109] zu finden.

¹⁴z.B. Master Slave Ansatz

¹⁵<http://www.csm.ornl.gov/pvm/>

2.3.3 Multithreading

Das Multithreading Programmiermodell findet in Systemen mit gemeinsamem Speicher Anwendung und basiert auf der Erzeugung von Threads. Hierfür wird das Prozessmodell dahingehend erweitert, daß ein Prozeß aus mehreren Threads bestehen kann. Ein Thread ist dabei ein eigenständiger Kontrollfluß, der separate Befehlszähler, Register für lokale Variablen und einen Keller besitzt. Den zur Verfügung stehenden Speicher für die Ablage von Daten teilt sich ein Thread mit allen anderen Threads des zugehörigen Prozesses. Ein Thread kann deshalb auf alle Daten anderer Threads im eigenen Prozeß zugreifen, was eine aufwendige Kommunikation über Betriebssystemroutinen erspart. Darüber hinaus teilen sich alle Threads eventuell geöffnete Dateien und andere Ressourcen des Prozesses. Deshalb wird auch die Bezeichnung *leichtgewichtiger Prozeß*¹⁶ als Synonym verwendet. Ein Standard, der heute nahezu von jedem Betriebssystem unterstützt wird, sind *POSIX-Threads* [22]. Dort wird ein einfaches *Application Programming Interface* (API) zur Erzeugung und zum Umgang mit Threads durch das Betriebssystem beschrieben. Häufig ist auch von der *Pthreads*-Bibliothek die Rede. Hierbei ist dann oft eine entsprechende Implementierung von *POSIX-Threads* für ein bestimmtes System gemeint. Ein recht genauer Überblick zur gesamten Thematik des Multithreadings ist in [5, 124] zu finden.

Synchronisations- und Kontrollstrukturen im Threadmodell

Diese Abschnitt nennt und beschreibt einige der weit verbreiteten Kontrollkonstrukte, die im Zusammenhang mit dem Multithreading benutzt werden [127]. Im späteren Verlauf der Arbeit werden diese Verfahren verwendet, ohne nochmals genauer darauf einzugehen.

Barrier Unter einer *Barrier* wird ein Konstrukt verstanden, welches dazu dient, die beteiligten Kontrollflüsse an einem bestimmten Punkt der Programmausführung zu synchronisieren. Ein Thread, der eine Barrierenanweisung erreicht, wartet bis alle beteiligten Threads ebenfalls diesen Punkt der Programmausführung erreicht haben. Erst danach wird die Ausführung der Threads fortgesetzt.

Bei Berechnungen, die zu einem Zeitpunkt alle Teilergebnisse zusammentragen und als Ausgang für die weitere Arbeit nehmen, ist es meistens vorteilhaft, eine Barrier einzusetzen. Das Konzept der Barrier steht für Programmiermodelle mit verteiltem und für Programmiermodelle mit gemeinsamen Speicher gleichermaßen zur Verfügung.

Mutex Der Begriff Mutex steht für **M**utual **E**xclusion bzw. gegenseitigen Ausschluß und wird meist mit speziellen Variablen realisiert. Mit dieser Methode kann ein kritischer Bereich des Programmcodes zu einer Zeit t von maximal einem Thread ausgeführt werden. Um dieses Verhalten sicherzustellen, müssen die konkurrierenden Threads folgende Regeln einhalten: Bevor ein Thread einen geschützten kritischen Bereich betritt, versucht er die zugehörige Mutex-Variable zu sperren (*lock()*). Gelingt ihm dies, ist er der Eigentümer der Mutex-Variable und kann den kritischen Bereich betreten. Nach Verlassen des Bereiches gibt der Thread die Sperre wieder frei (*unlock()*). Kann ein Thread

¹⁶oder *lightweight process* bzw. *LWP*

nicht die Kontrolle über die Mutex-Variable erlangen, so wird er gesperrt, d.h. ihm wird keine Rechenzeit mehr zugeteilt, bis die Mutex-Variable wieder freigegeben wurde [109].

Monitor Ein Monitor stellt einen speziellen Mutex-Mechanismus auf Datenebene bereit und ist oftmals an ein Datenobjekt gebunden [57]. Die Funktionalität ist meist in der Programmiersprache selbst integriert (Ada, Modula, Java), so daß der Programmierer keine expliziten Funktionsaufrufe benutzen muß¹⁷.

Semaphore Semaphore werden ebenfalls zur Realisierung eines wechselseitigen Ausschlusses eingesetzt. Ein Semaphor ist in der Regel ein Konstrukt aus einer Integer Variable s sowie zwei zugehörigen atomaren Operationen $P(s)$ und $V(s)$ zur Manipulation von s . Die Bezeichnung der Methoden stammt von den holländischen Begriffen *passeren* (passieren) und *vrijgeven* (freigeben). Kann die Zählervariable s nur die Werte 0 und 1 annehmen, wird das Konstrukt als *binärer Semaphor* bezeichnet. Im anderen Fall handelt es sich um einen *zählenden Semaphor*.

Die Operation $P(s)$ blockiert den aufrufenden Thread bis der Wert von s größer als 0 ist, dekrementiert dann dessen Wert und erlaubt die weitere Ausführung. Im Gegensatz dazu erhöht die Operation $V(s)$ den Wert von s um 1 und aktiviert einen blockierten Thread [100, 109].

Programmierungsumgebungen für Threads

Dieser Abschnitt beschreibt den *OpenMP*-Standard als eine Umsetzung des Thread-Programmiermodelles auf Compilerenebene und geht kurz auf zwei Alternativen im Java-Umfeld, *JOMP* und *Hydra PPS*, ein.

OpenMP OpenMP ist eine Erweiterung der Programmiersprachen C/C++ und Fortran um Compilerdirektiven, durch die sich parallele Bereiche in einem Programm kennzeichnen lassen. Ein OpenMP-Compiler ist dann in der Lage, ein threadparalleles Programm zu erzeugen. Entwicklungsziel war eine einfache, aber dennoch mächtige Schnittstelle für die Programmierung von Systemen mit gemeinsamem Adreßraum. Die entstandene OpenMP-Spezifikation besteht zu einem Großteil aus Compilerdirektiven, enthält aber auch Laufzeitroutinen. Diese Laufzeitroutinen dienen in erster Linie zur Informationsbeschaffung während der Laufzeit. Andere Routinen stellen z.B. einfache *Lock*-Operationen bereit, mit denen der Zugriff auf Variablen geschützt werden kann.

Der überwiegende Anteil der Funktionalität von OpenMP ist in Kommentarbereichen der jeweiligen Programmiersprache untergebracht und wird als Direktive bezeichnet. Damit soll auch die Übersetzung des Programmes mit nicht-OpenMP-fähigen Compilern ermöglicht werden, welche dann sequentielle Programme generieren. Wird ein OpenMP-fähiger Compiler verwendet, entsteht ein threadparalleles Programm [26, 102].

¹⁷Eine Kennzeichnung der Bereiche mit Schlüsselwörtern der Sprache ist aber notwendig.

Das parallele Ausführungsmodell von OpenMP basiert auf einem *fork-join* Modell. Die Ausführung eines OpenMP-Programmes beginnt mit einem einzelnen Thread, der als *Master Thread* bezeichnet wird. Dieser arbeitet das Programm so lange sequentiell ab, bis er auf ein paralleles Konstrukt stößt. An dieser Stelle generiert der *Master-Thread* ein Team aus Threads und wird selbst zum *Master* dieses Teams. Die Anzahl der erzeugten Threads im Team hängt vom Laufzeitsystem und verschiedenen, vom Programmierer zu bestimmenden, Parametern ab.

Das Team führt gemeinsam und konkurrierend den mit einer Direktive gekennzeichneten Block von Anweisungen aus. Danach werden alle Threads bis auf den *Master-Thread* beendet. Dieser kann dann seine Arbeit am Hauptprogramm fortsetzen.

Eine OpenMP-Version für Java ist **JOMP** (*Java OpenMP*). Der OpenMP-Standard wird bis auf einige Ausnahmen¹⁸ fast vollständig unterstützt und implementiert [71]. Zur Übersetzung von Programmen muß der mitgelieferte Compiler benutzt werden. Dieser enthält einen Präprozessor, welcher die *Direktiven*, die ebenfalls in Kommentaren untergebracht sind, in entsprechende Quellcodekonstrukte umwandelt. Danach wird automatisch ein beliebiger Java-Compiler aufgerufen und der vom Präprozessor erzeugte Quellcode übersetzt. Die eigentliche Funktionalität ist in einer separaten Bibliothek enthalten, welche ein Laufzeitsystem enthält. Die Aufgabe des Präprozessors ist deshalb die Transformation der Direktiven in entsprechende Methodenaufrufe. Das Vorgehen wurde so gewählt, um die Umsetzung einiger Direktiven¹⁹ zu vereinfachen. Ein weiterer Vorteil ist eine problemlose Erweiterung der Funktionalität durch den einfachen Austausch der verwendeten Bibliothek, ohne eine Neuübersetzung der Programme.

Hydra PPS Das *Hydra Parallel Programming System* wendet eine ähnliche Philosophie wie OpenMP an. Hydra Programme können von jedem Java-Compiler übersetzt werden und sind auch in jeder virtuellen Java-Maschine lauffähig. Um jedoch die angebotene Funktionalität von *Hydra PPS* nutzen zu können, muß ein Programm in der speziellen virtuellen Maschine der Hydra Laufzeitumgebung (*HVM - Hydra Virtual Machine*) gestartet werden. Die *HVM* wertet die im ursprünglichen Quellcode enthaltenen und vom Compiler in den Bytecode übernommenen Metadaten²⁰ aus und startet mehrere parallel arbeitende Threads. Die Metadaten, deren Syntax und Semantik durch Hydra beschrieben werden, enthalten den genauen parallelen Programmablauf [105].

Das Programmiermodell von Hydra PPS basiert, im Gegensatz zu OpenMP, auf Ereignissen und deren Beziehungen untereinander. Diese Ereignisse werden zur Laufzeit auf Threads abgebildet. Es kann daher eher als taskorientierter Ansatz verstanden werden.

¹⁸Es existiert aber beispielsweise keine `atomic` Direktive.

¹⁹wie z.B. `for`

²⁰Seit Java 5 ist es möglich, sog. Annotations im Quellcode zu hinterlassen, die auch in der übersetzten Version vorhanden sind und zur Laufzeit ausgelesen werden können.

2.3.4 DSM und VSM Modelle

Als *virtuelle Hauptspeichersysteme* bezeichnet man (*NUMA*-) Systeme, welche den Prozessoren eines Multiprozessorsystems mit verteiltem Speicher den Zugriff auf einen (physikalisch nicht als solchen vorhandenen) logisch gemeinsamen Speicher ermöglichen [69]. Je nach Art der Umsetzung (Hardware, Software oder kombiniert) lassen sich verschiedene Formen unterscheiden. Allgemein hat sich die 1989 von Li und Hudak [25, 82, 84] eingeführte Bezeichnung *Distributed Shared Memory (DSM)* als zentraler Oberbegriff für solche Systeme durchgesetzt. Manchmal wird, im gleichen Kontext, auch von einem *Virtual Shared Memory (VSM)* gesprochen. Eine Einteilung nach Raina [108] (*vgl. dazu Abbildung 2.3*) gliedert mögliche Systeme in:

VSM (*Virtual Shared Memory*) Solche Systeme stellen mittels Hardwareunterstützung einen kohärenten gemeinsamen Adreßraum bereit. Die adressierbaren Speichereinheiten sind relativ klein und liegen etwa in der Größenordnung von Cacheeinträgen. Ein Beispiel für diese Gruppe ist der an der Stanford University entwickelte DASH Prototyp [81, 108]. Dieser war Gegenstand für eine Vielzahl von Untersuchungen. Unter anderen basieren die Programmsammlungen SPLASH und SPLASH-2 auf Erfahrungen mit dieser Maschine und enthalten Teile, welche auf die spezielle Architektur des DASH Systems abzielen [137].

SVM (*Shared Virtual Memory*) Als SVM werden seitenbasierte Systeme bezeichnet, die meist ohne Hardwareunterstützung auskommen. Der gemeinsame Speicher und insbesondere die Kohärenz der Daten wird durch eine Softwareschicht realisiert. Der zur Verfügung stehende gemeinsame Speicher setzt sich aus den von jedem Knoten einzeln zur Verfügung gestellten virtuellen Speicherbereichen (siehe Abbildung 2.3) zusammen. Da ein Knoten nicht zwangsläufig über einen gemeinsamen Speicher verfügen muß, kann die Art und Weise der Bereitstellung des virtuellen Speichers eines Knotens variieren. Beispiele hierfür sind die weiter unten nochmals näher beschriebenen IVY [83] und KOAN [76] Systeme.

DSM (*Distributed Shared Memory*) DSM-Systeme können als Hardware-, Software- oder Hybridlösung implementiert werden. Gemeinsames Merkmal ist jeweils ein fester Speicherplatz für ein Datum, d.h. es erfolgt keine Replikation des Datums auf mehreren Knoten. Falls das angeforderte Datum nicht lokal gespeichert ist, muß deshalb immer ein Fernzugriff erfolgen. Die direkte Unterstützung der Datenkohärenz ist nicht vorgesehen und muß in den höheren Schichten einer Anwendung erfolgen. Anhand der verwalteten Speichereinheiten unterscheidet man noch zwischen seitenbasierten und objektbasierten DSM-Systemen.

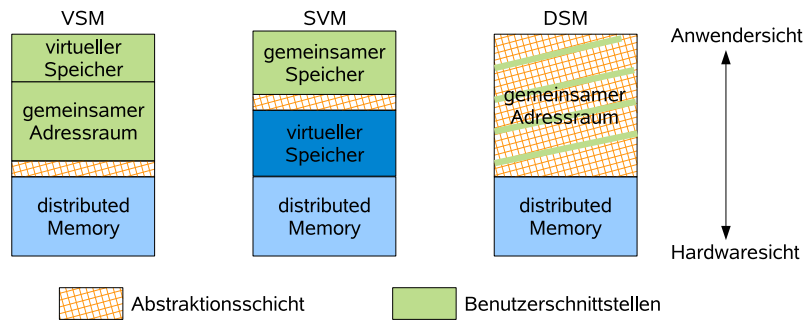


Abbildung 2.3: Einteilung virtueller Speichermodelle nach Raina [108].

RDMA

Mit dem von RDMA (*Remote Direct Memory Access*) verfolgtem Ansatz ist es möglich, Daten in den Hauptspeicher eines anderen Knotens zu übertragen oder von dort zu lesen, ohne das Betriebssystem oder den Prozess auf der Gegenseite zu involvieren. RDMA beschreibt hierbei nur das Verhalten, die genaue Implementierung und die zu benutzende Hardware ist variabel. Durch die zur Verfügung gestellten Hardwaredienste kann es aber als Basis für die Umsetzung eines *virtuellen Hauptspeichersystems* dienen. Mit *iWARP* (bzw. *RDMA over TCP/IP*) gibt es bereits eine Lösung, die für den Einsatz in den meisten Computernetzen geeignet ist [110]. Ebenfalls einen RDMA Mode sieht der InfiniBand-Standard vor. Zusätzlich werden noch zwei Sperrmechanismen: *Atomic CompareSwap* sowie *Atomic FetchAdd* zur Verfügung gestellt. Mit deren Hilfe können z.B. Mutexvariablen oder Semaphore implementiert werden [63].

Softwarebasierte Implementierungen

Es gibt eine ganze Reihe von experimentellen virtuellen Hauptspeichersystemen. An dieser Stelle soll auf softwarebasierte Systeme eingegangen werden und ein kurzer Überblick über einige dieser Systeme erfolgen. Ausführlichere Informationen sind z.B. in [25, 69, 75, 106] zu finden. Die hier beschriebene Auswahl bezieht sich auf Systeme, die zum Teil bei Neuentwicklungen als Referenz bzw. Basisimplementierung dienen und somit die Entwicklung maßgeblich beeinflusst haben.

IVY (*Integrated Shared Virtual Memory at Yale*) [83] ist eine der ersten Prototypimplementierungen für ein SVM-System nach [108]. Teile des Systems wurden direkt in die Speicherverwaltung eines Aegis Betriebssystems²¹ integriert, indem bei Seitenfehlern ein spezieller IVY Handler aufgerufen wird. Dadurch ist die Vergrößerung des Adreßraumes und die transparente Handhabung von Speicherseiten möglich. Durch die Benutzung von Speicherseiten kann es zu *False Sharing* Effekten kommen, da die Seiten von verschie-

²¹Aegis ist ein Unix-ähnliches Betriebssystem und wurde für Workstations der Firma Apollo Computer etwa zwischen 1980 und 1989 eingesetzt. Später wurde es in Domain/OS umbenannt, wird aber heute nicht mehr weiterentwickelt.

denen Prozessen beschrieben werden können. Zur Sicherstellung der Kohärenz wird ein *write-invalidate* Protokoll verwendet.

TreadMarks [7, 73] ist ein seitenbasiertes Software-DSM-System, welches an der Rice University ab 1990 entwickelt wurde. Eines der Ziele war es, die Portabilität von Anwendungen sicher zu stellen. Es sind daher keine Eingriffe ins Betriebssystem notwendig und alle notwendigen Prozesse laufen im Userspace. Deshalb ist TreadMarks auf einer Vielzahl von Systemen, wie z.B. IBM, DEC, SUN, HP, AMD, INTEL und SGI einsetzbar. Der verwaltete Speicher wird, in Form einzelner Speicherseiten, vom System auf den physikalisch vorhandenen Hauptspeicher verteilt. Mit der von TreadMarks zur Verfügung gestellten Bibliotheksroutine `Tmk_malloc()` kann ein beliebig großer Speicherblock reserviert und dann benutzt werden. Für die Synchronisation und Zugriffskontrolle werden *Barriers* und *exclusive Locks* angeboten.

Munin [17, 140] ist ein DSM-System, welches als eines der Ersten mehrere Kohärenzprotokolle unterstützte. Die Auswahl des gewünschten Protokolles erfolgt im Sourcecode durch Kennzeichnung der entsprechenden Datenobjekte. Methoden für verteilte Locks und Barriers sind ebenfalls enthalten.

Die **JDSM** [117] Implementierung eines DSM-Systems stellt eine Laufzeitbibliothek bereit und ermöglicht so den verteilten Zugriff auf die im virtuellen Speicher abgelegten Objekte. Es werden mehrere Schichten mit klar getrennten Aufgabengebieten definiert. Hervorzuheben ist dabei die Möglichkeit zur Benutzung unterschiedlicher Implementierungen für die Kommunikationsschicht. Der interne Mechanismus von Java zur Objektserialisierung wird jedoch immer benutzt.

Das **JuxMem** Projekt stellt einen gemeinsamen Adreßraum mit transparentem Zugriff bereit. Zielplattformen sind Grid und Peer-to-Peer Umgebungen [8]. In diesen Anwendungsfällen ist es denkbar, daß einzelne Knoten unzuverlässig arbeiten oder die Verbindungen einfach abbrechen. Da die Persistenz der Daten sichergestellt werden muß, können konventionelle Verfahren nicht angewendet werden. Weiterhin erfolgt die Kommunikation der einzelnen Hostsysteme einer solchen Zielplattform normalerweise über potentiell unsichere Netze (Internet). Daher müssen zusätzliche Vorkehrungen zur Sicherung der übertragenen Daten getroffen werden. All diese Besonderheiten erhöhen den Verwaltungsaufwand, der sich auf die Geschwindigkeit des Laufzeitsystems auswirkt. Aus diesem Grund sprechen die Autoren auch nur von einem DSM-ähnlichen System [9]. Aktuell sind zwei Implementierungen für *C* und *Java* verfügbar²².

Jackal [132] setzt sich aus der Kombination eines Compilers für Java-Sourcecode und einer Laufzeitumgebung zur Bereitstellung eines DSM-Bereiches zusammen. Der Jackal Compiler sorgt dafür, daß gemeinsam verwendete Daten im DSM abgelegt werden und fügt zusätzliche Zugriffstests, zur Sicherstellung der Datenkonsistenz, ein.

²²<http://ralyx.inria.fr/2007/Raweb/paris/uid57.html>

2.3.5 Linda TupleSpace

Linda wurde als Programmiersprache für die verteilte Programmierung von David Gelernter an der Yale-Universität entwickelt [4, 43]. Zentraler Bestandteil ist ein für die gemeinsame Kommunikation benutzter Tupelraum. Verschiedene Prozesse eines verteilten Programms kommunizieren über den gemeinsamen Tupelraum dadurch, daß jeder dieser Prozesse beliebig viele Tupel in den Tupelraum hinzufügen oder daraus entfernen kann. Für die Nachbildung eines klassischen Send/Receive-Vorganges ist bei Linda eine Operation zum Einfügen und eine Operation zum Entfernen des Kommunikationstupels notwendig. Die hieraus resultierende zeitliche Entkoppelung ermöglicht flexible und robuste Programme, die auch auf mögliche Teilausfälle des Systems reagieren können [24].

Tupelraum-Implementierungen

Es gibt hier wieder eine ganze Reihe von Implementierungen, welche das Konzept des Tupelraumes aufgreifen und nach eigenen Zielsetzungen interpretieren. An dieser Stelle sollen einige dieser Systeme vorgestellt werden.

Die **TSpaces**-Bibliothek von IBM stellt einen Tupelraum bereit, unterstützt alle in Linda definierten Operationen und stellt einige zusätzliche Operationen, wie z.B. `delete` zum Löschen einzelner Tupel, bereit. Vorteil der Implementierung ist eine sehr einfache Handhabung, da nur die TSpaces-Bibliothek eingebunden und ein Server gestartet werden muß. Weitere Merkmale sind ein „*Verfallsdatum*“ für Tupel²³ und die Unterstützung von Transaktionen [38, 80].

JavaSpaces von Sun Microsystems unterstützt ebenfalls alle in Linda definierten Tupelraumoperationen. Das gesamte System ist relativ komplex und enthält plattformabhängige Teile. Auf JavaSpaces basierende Anwendungen sind portabel, die Installation des Laufzeitsystems kann jedoch einigen Aufwand erfordern. Die meisten Komponenten von JavaSpaces bauen auf der Jini Technologie²⁴ auf. Für die Kommunikation wird eine leicht abgeänderte RMI-Variante verwendet, welche eine eigene Objektserialisierung benutzt, die nicht kompatibel mit dem Standard von Java ist [10, 41, 135].

Die kommerzielle **GigaSpaces** Implementierung²⁵ basiert auf der von Sun Microsystems für *JavaSpaces/Jini* veröffentlichten Spezifikation. Es ist weitestgehend abwärtskompatibel zu *JavaSpaces* und enthält einige zusätzliche Operationen für die Arbeit mit Tupeln. Außerdem existieren verteilte Implementierungen für die im *Java Collections Framework*²⁶ definierten Schnittstellen `List`, `Set` und `Map` [135].

Neben vielen weiteren Implementierungen existiert z.B. für die Programmiersprache C++ die *CppLinda*²⁷ Implementierung. Mit *Rinda* ist eine *Linda* Implementierung direkt in der Klassenbibliothek²⁸ von Ruby enthalten.

²³Die Tupel werden nach Ablauf einer bestimmten Zeitspanne aus dem Tupelraum entfernt.

²⁴<http://www.jini.org>

²⁵<http://www.gigaspace.com>

²⁶<http://java.sun.com/javase/6/docs/technotes/guides/collections/index.html>

²⁷<http://sourceforge.net/projects/cpp linda/>

²⁸<http://www.ruby-doc.org/stdlib/>

2.3.6 weitere parallele Programmiermodelle und Umgebungen

Eine auf *C* basierende Programmierplattform für CPUs, GPUs und DSPs bietet die *Open Computing Language (OpenCL)*²⁹. Der Sprachumfang von *ANSI C* wurde hierbei um Datentypen und Funktionen zur parallelen Verarbeitung erweitert. Ein mögliches Zielsystem der resultierenden Programme setzt sich aus einem sogenannten *Host* und einem oder mehreren *Compute Devices* zusammen. Jedes dieser *Compute Devices* kann wieder aus einer oder mehreren Einheiten (*Compute Units*) bestehen, die ihrerseits in ein oder mehrere Berechnungseinheiten (*Processing Elements*) untergliedert sind (Abbildung 2.4). Programme bzw. Programmteile (*Kernel*) werden zur Laufzeit durch den Host auf die vorhandenen *Devices* verteilt. Eine Besonderheit des Systems ist die dynamische Übersetzung bestimmter Teile (*Kernel*) zur Laufzeit. Dazu werden zwei Arten von *Kernel* unterschieden: einem *OpenCL Kernel*, bei dem eine dynamische Übersetzung möglich ist, und einem *Native Kernel*. Der letztgenannte Kerneleyp ist hardware- und implementierungsspezifisch, wodurch eine dynamische Übersetzung verhindert wird [91]. Die genaue Systemkonfiguration muß zum Zeitpunkt der Erstellung nicht bekannt sein und es können noch nachträgliche Optimierungen durch das Laufzeitsystem eingefügt werden.

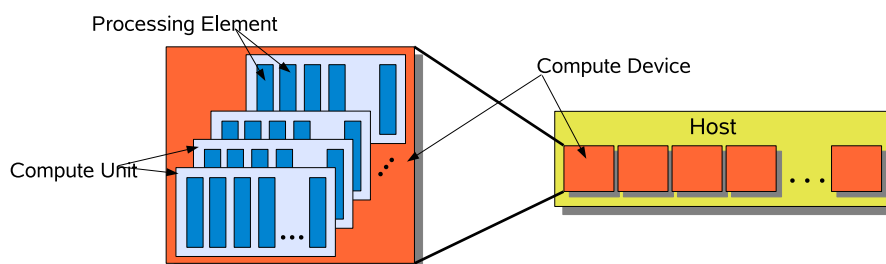


Abbildung 2.4: Aufbau eines OpenCL Zielsystemes aus *Host*, *Compute Devices*, *Compute Units* und *Processing Elements*.

JavaParty³⁰ ist eine Spracherweiterung der Java-Programmiersprache für die verteilte Ausführung von Programmen. Die zugehörige Laufzeitumgebung faßt dabei verteilt auf verschiedenen Hostsystemen laufende Java-Maschinen (JVMs) zu einer einzigen virtuellen JVM mit gemeinsamem Adreßraum zusammen. Ein paralleles Java-Programm mit mehreren Kontrollflüssen läuft transparent in dieser Umgebung ab. Der in JavaParty enthaltene Compiler wandelt JavaParty Sourcecode in normalen Java-Sourcecode um, indem die mittels des neu eingeführten Schlüsselwortes `remote` gekennzeichneten Klassen in Sourcecode mit RMI Aufrufen transformiert werden. Dadurch wird ein Objekt dieser Klasse entfernt referenzierbar und seine Methoden entfernt aufrufbar [53]. Die Ausführung von einzelnen Klassenmethoden kann durch dieses Vorgehen auf einem entfernten

²⁹<http://www.khronos.org/registry/cl/>

³⁰<http://www.ipd.uka.de/Tichy/projects.php?id=9>

System geschehen. Ein direkter Vergleich mit einem DSM-System ist nicht möglich, da JavaParty das Konzept von entfernten Unterfunktionsaufrufen (RPC) aufgreift und dieses konsequent umsetzt.

Orca ist eine prozedurale Programmiersprache für verteilte Umgebungen, die sich an Modula-2 orientiert. Parallele Kontrollflüsse werden durch einen `fork` Aufruf gestartet und können einen gemeinsamen Speicher verwenden. Eigene Datenstrukturen, die dort abgelegt werden, bestehen aus einem Spezifikations- und einem Implementationsteil [12]. Es sind bereits einige Datenstrukturen wie z.B. Records, Unions, dynamische Felder und Mengen vorhanden. Zeiger werden aus Sicherheitsgründen nicht unterstützt.

UPC³¹ (*Unified Parallel C*) [23, 36, 131] ist ebenfalls eine parallele Programmiersprache, die an der University of California in Berkeley entstanden ist. Es ist eine direkte Erweiterung der *C* Programmiersprache. Der Programmierer kann einen „*single shared, partitioned address space*“ verwenden, aus dem jeder Prozessor direkt schreiben und lesen kann. Die Daten sind aber physikalisch einem einzelnen Prozessor/System zugeordnet. Weiterhin verfügt jeder Thread über einen privaten Speicherbereich, der nicht global verfügbar ist. Im privaten Speicherbereich können lokale Daten bzw. Zwischenergebnisse gepuffert werden. Zu einem späteren Zeitpunkt kann dann schnell auf die abgelegten Daten zugegriffen werden, da in diesem Speicherbereich auf Synchronisationsoperationen verzichtet werden kann.

UPC arbeitet mit einem *Single Program Multiple Data* (SPMD) Modell, in dem der Grad der Parallelität beim Programmstart, z.B. durch die feste Zuordnung von Threads auf die vorhandenen Prozessoren, vorgegeben ist.

Ein weiteres, im Umfeld von *UPC*, entstandenes Projekt ist z.B. **Titanium**, welches als Java-Variante der in *UPC* verwendeten Konzepte betrachtet werden kann [139]. Das **GASNet** Projekt³² (*Global-Address Space Networking*) stellt eine sprachunabhängige und netzwerkunabhängige Abstraktionsschicht zur Kommunikation bereit, die speziell auf die Bedürfnisse von „*parallel global address space SPMD*“ Programmiersprachen³³, wie *UPC* und *Titanium*, abgestimmt ist [20].

Beim **Global Arrays** [95, 97] Toolkit³⁴ handelt es sich um eine auf MPI basierende Bibliothekslösung für homogene Systemumgebungen, welche den transparenten Zugriff auf verteilte Feld-Datenstrukturen unterstützt. Der Anwender kann verteilte, multidimensionale Felder erzeugen und deren physikalische Verteilung auf die Knoten der Ausführungsumgebung beeinflussen [96]. Das System setzt ausdrücklich eine *NUMA*-Architektur für den über das Feld zugreifbaren gemeinsamen Speicher um, enthält aber auch Funktionen, um die auf dem lokalen Hostsystem des Aufrufers gespeicherten Bereiche des gemeinsamen verteilten Feldes zu bestimmen. Dadurch ergibt sich eine zusätzliche Op-

³¹<http://upc.lbl.gov/>

³²<http://gasnet.cs.berkeley.edu/>

³³Abgekürzt auch als *PGAS* Programmiermodell bezeichnet. Auf diesem Modell basieren unter anderen noch die Programmiersprachen *Fortress* und *Co-array Fortran*.

³⁴<http://www.emsl.pnl.gov/docs/global/>

timierungsmöglichkeit für den Programmierer, da z.B. diese lokal vorhandenen Bereiche gezielt beschrieben können und somit eine Kommunikation mit anderen Hostsystemen entfällt.

Das Programmiermodell erlaubt die Mischung von *MPI* Instruktionen mit Zugriffen auf die *verteilte Feld-Datenstruktur*, allerdings ist das ausgeführte Programm an die Beschränkungen von *MPI* gebunden³⁵.

*ObjectSpace Voyager Core Technology*³⁶, oder kurz **Voyager**, ist ein Object Request Broker (ORB) für Java. Das kommerziell verfügbare Programmpaket besteht aus Compiler, Klassenbibliothek und einem Server. Im Grunde erweitert es das von RMI³⁷ bekannte Verfahren um einige komfortable Eigenschaften, beruht aber im wesentlichen auf einer Verteilung von Objekten.

Der mitgelieferte Compiler erzeugt aus Klassen (im Java `class` Format oder als Quellcode) neue Wrapperklassen, die dann als entfernte Objekte eingesetzt werden können. Für die interne Umsetzung wird nicht RMI, sondern eine eigene Implementierung verwendet. Eine Erweiterung des *Voyager Systems* ist das Forschungsprojekt **Dejay** der Universität Hamburg [19]. Es führt sogenannte virtuelle Prozessoren ein. Mit diesem Konzept ist es möglich, eine beliebige virtuelle Rechner- und Prozessorhierarchie aufzubauen.

2.4 Java

Die Prototypimplementierung der in dieser Arbeit beschriebenen Programmierumgebung (Kapitel 3 (S.57)) wurde mit der Programmiersprache Java erstellt. Deshalb werden an dieser Stelle einige Aspekte der Java-Programmiersprache und der Laufzeitumgebung besprochen.

2.4.1 Kritische Bereiche

Ein kritischer Bereich ist ein Programmabschnitt, in dem Daten von mehreren Threads geändert oder zugegriffen werden. Viele der dabei durchgeführten Operationen benötigen mehrere Berechnungsschritte. Da die vorhandenen Threads einer parallelen Berechnung gegenseitig in Konkurrenz stehen, muß durch geeignete Methoden sichergestellt werden, daß die Datenmanipulation eines Threads nicht durch einen anderen unterbrochen wird. Anderenfalls kann es zu fehlerhaften Berechnungen kommen (*vgl. hierzu auch Kapitel 2.3.3 (S.38)*). Das Threadmodell wurde in den Sprachumfang von Java direkt integriert. Zur Unterstützung des Programmierers bei der Koordination der Threads stehen in Java die nachfolgend beschriebenen Konstrukte und Methoden zur Verfügung.

³⁵Es werden semi-heterogene Systemumgebungen, z.B. wegen einheitlichen Wordgrößen benötigt, Objekte werden nicht unterstützt.

³⁶<http://www.recursionsw.com/products/voyager/voyager.asp>

³⁷ist aber vollkommen eigenständig

synchronized Mit dem `synchronized` Schlüsselwort kann ein bestimmter Block von Anweisungen oder eine ganze Methode geschützt werden. Auf diese Art wird sichergestellt, daß höchstens ein Thread einen so markierten Bereich betreten kann.

In der Urklasse aller Java-Objekte (`Object`) existiert ein implizites Sperrflag. Bevor ein mit `synchronized` geschützter Bereich betreten werden kann, muß diese Sperre aktiviert werden. Ist diese bereits durch einen anderen Thread gesetzt, so wird der aufrufende Thread blockiert und bei der weiteren Verteilung von Rechenzeit nicht mehr berücksichtigt. Erst nach der Freigabe der Sperre kann der Thread seine Berechnungen fortsetzen. Es gibt verschiedene Möglichkeiten, `synchronized` einzusetzen :

1. Einsatz als Block: das angegebene Objekt `schutzObjekt` wird zum Sperren benutzt.

```
synchronized(Object schutzObjekt)
{
    /* kritischer Bereich */
}
```

2. Alternativ dazu können auch ganze Methoden als `synchronized` deklariert werden.

```
public synchronized foo()
{
    /* kritischer Bereich */
}
```

Dieser in Java verwirklichte Ansatz wird auch als *Monitor* bezeichnet [58]. Zu jedem Zeitpunkt ist das Manipulieren eines Objektes nur einem Thread erlaubt [109]. Der wechselseitige Ausschluß bezüglich aller durch `synchronized` geschützten Programmbereiche wird automatisch durch die Laufzeitumgebung sichergestellt. Methodenaufrufe aus einem geschützten Bereich, die am gleichen Objekt synchronisiert sind, können ohne weitere Unterbrechung ausgeführt werden, wenn der aufrufende Thread Besitzer des Sperrflags des Objektes (Monitores) ist [45, 130].

Meist sind nur bestimmte Teile einer Funktion kritisch. Deshalb ist es oft sinnvoller, die *Variante 1* anzuwenden. Der hierbei zu schützende Abschnitt bleibt relativ klein und unkritische Abschnitte können parallel ausgeführt werden. Bei der Anwendung von `synchronized` auf statische Methoden einer Klasse (Klassenmethoden) wird ein für diese Klasse gültiges globales Sperrflag verwendet und so der Schutz auf alle existierenden Instanzen der Klasse erweitert [79, 99].

volatile Mit Hilfe des Variablenmodifizierers `volatile` können Zugriffsoperationen auf Objekt- bzw. Klassenvariablen als *atomar* gekennzeichnet und gleichzeitig interne Optimierungsversuche der JVM auf diese Daten unterbunden werden. Mit `volatile` gekennzeichnete Daten müssen vor jedem Zugriff aus dem Speicher gelesen und unmittelbar nach Beendigung der Operation wieder in den Speicher zurück geschrieben werden. Zur

Laufzeit dürfen die Inhalte dieser Variablen nicht in den Prozessorregistern zwischengespeichert oder die im Programm vorgegebene Zugriffsreihenfolge durch den Compiler verändert werden [45, 107, 130].

Die Kennzeichnung als `volatile` bezieht sich nur auf die Zugriffe im Speicher, Manipulationen an den Daten selbst können unterbrochen werden. Deshalb ist `volatile` mehr als Hilfsmittel zu sehen, um die Variableninhalte möglichst aktuell zu halten und eventuelle Optimierungsstrategien unterdrücken zu können.

wait und notify In vielen Fällen müssen konkurrierende Threads auf das Eintreten eines bestimmten Ereignisses warten. Hierbei sollten Konstruktionen wie z.B.

```
while (Bedingung == false) ;
```

vermieden werden, da durch die ständige Auswertung der Schleifenbedingung unnötig Rechenzeit verbraucht wird (*aktives Warten*).

Die von Java zur Lösung dieser Problematik zur Verfügung gestellte Operation `wait()` entzieht einem Thread die gesamte Rechenzeit bis er durch ein `notify()` wieder geweckt wird³⁸.

Als Bezugsgrundlage dient eine Objektinstanz, die vor dem Aufruf von `wait` oder `notify` durch `synchronized` geschützt werden muß. Die Methoden `wait()` und `notify()` sind ebenfalls in der Klasse `Object` definiert und werden so an alle Objekte vererbt. Um die wartenden Threads für ein Objekt speichern zu können, verfügt jede Instanz über eine Warteliste. Der Java-Scheduler kann anhand dieser Liste einen beliebigen Thread oder alle Threads aktivieren und aus der Liste entfernen. Nach der Reaktivierung versucht der Thread die Kontrolle über das mit `synchronized` geschützte Objekt zu erhalten. Erst danach kann er mit seiner Arbeit fortfahren [79, 99].

concurrent Package Ab Java 5 gibt es eine komplette Unterstützung für viele Probleme, die mit den in der Sprachdefinition festgelegten Mitteln schwer oder nur sehr umständlich realisiert werden konnten. Hierzu wurde das Package `java.util.concurrent` in den Standard aufgenommen. Die darin enthaltenen Klassen umfassen unter anderem Implementierungen für atomare Objekte (z.B. `AtomicBoolean` oder `AtomicInteger`), Locks (`ReentrantLock`) und Threadpools (`ThreadPoolExecutor`) [44].

2.4.2 Das Java Memory Modell

Im *Java Memory Modell* (JMM) wird das Verhalten der Java-Laufzeitumgebung bei Speicherzugriffen definiert [45]. Spezifikationsziele sind Plattformunabhängigkeit und eine Verringerung der Komplexität, um möglichst viele Hardwaresysteme unterstützen zu können und die Implementierung der jeweiligen virtuellen Maschine nicht unnötig zu erschweren.

In Kapitel 2.3.1 (S.35) wurden bereits einige Konsistenzmodelle vorgestellt. Für das in Java verwendete Modell orientierte man sich an der *sequentiellen Konsistenz*, da diese

³⁸`notify()` weckt einen beliebigen und `notifyAll()` alle wartenden Threads

intuitiv zu verstehen und deshalb auch weniger fehleranfällig ist [44, 45]. Allerdings hätte man bei einer konsequenten Umsetzung der *sequentiellen Konsistenz* die Optimierungsmöglichkeiten des Laufzeitsystems stark eingeschränkt, weil dort gefordert wird, daß alle Speichervorgänge in der vom Programm vorgegebenen Reihenfolge stattfinden müssen. Dadurch sind viele Compilertransformationen nur noch begrenzt durchführbar. Das in Java verwendete Modell wurde daher leicht abgeändert und erlaubt Modifikationen solange diese für den Programmierer nicht sichtbar sind [45].

Die ursprüngliche Beschreibung des Modelles war an einigen Stellen noch etwas ungenau und ließ Interpretationsspielraum. Beispielsweise konnten sich die sichtbaren Werte von `final` Variablen ändern, obwohl per Definition die darin gespeicherten Werte unveränderlich sind. Deshalb wurde im *Java Specification Request 133* [107] eine Überarbeitung des Java-Speichermodelles vorgenommen und mit Java 5 eingeführt. Bisher existierende Unstimmigkeiten wurden beseitigt und eine weitere Präzisierung vorgenommen. Für detailliertere Ausführungen sei auf [45, 107] verwiesen³⁹.

2.4.3 Der Garbage Collector

Dieser Abschnitt beschreibt das von der Java-Laufzeitumgebung eingesetzte Verfahren zur automatischen Speicherbereinigung. Die Kenntnis der Funktionsweise ist u.a. für die Interpretation der Ergebnisse der in Kapitel 5.2.3 durchgeführten Untersuchungen vorteilhaft. Insbesondere lassen sich der notwendige Speicherbedarf und einige Laufzeit-schwankungen der Testanwendungen auf das Speichermanagement zurückführen.

Java übernimmt vollständig das Speichermanagement von Programmen und erlaubt kaum Eingriffe des Programmierers in diese Vorgänge. Für die Freigabe von Speicherbereichen, die von nicht mehr referenzierten Objekten belegt werden, kümmert sich ein *Garbage Collector* [27, 33, 120]. An dieser Stelle soll der in der *Java HotSpot* Laufzeitumgebung von Sun Microsystems enthaltene *Garbage Collector* und dessen Funktionsweise beschrieben werden. Für die Arbeitsweise einer solchen automatischen Speicherbereinigung gibt es wieder verschiedene Verfahren mit Vor- und Nachteilen. Bevor einige allgemeine Verfahren vorgestellt werden, muß kurz auf den Begriff der *Erreichbarkeit* von Objekten eingegangen werden.

Als *erreichbar* gilt ein Objekt, wenn es vom Programm manipuliert und so den weiteren Ablauf beeinflussen kann. Es muß also innerhalb des Programmlaufes noch mindestens eine gültige Referenz auf das Objekt geben. Referenzen können sich auf dem Programmstack, in globalen Variablen oder in Registern befinden. Weiterhin sind alle über diese Referenzen direkt oder indirekt referenzierten Objekte erreichbar. Man unterscheidet deshalb hier zwischen direkt und indirekt erreichbaren Objekten. Ist ein Objekt nicht mehr *erreichbar*, kann es als unbenutzt klassifiziert und gegebenenfalls aus dem Speicher entfernt werden.

³⁹oder die Seite <http://gee.cs.oswego.edu/dl/jmm/cookbook.html> von Doug Lea

den eigentlichen Algorithmus. Da eine Implementierung meist rekursiv erfolgt, kann es bei knappen Systemressourcen zu Fehlern kommen, da für neue Unterfunktionsaufrufe kein Speicher auf dem Stack mehr reserviert werden kann. Eine iterative Variante ist der *Deutsch-Schorr-Waite* Algorithmus (DSW). Dieser benötigt erheblich weniger Speicher, weil die benötigten Daten platzsparender gespeichert werden können [47, 112].

Mark and Compact

Die bisherigen Verfahren konnten eine Fragmentierung des verwalteten Hauptspeichers nicht verhindern, so daß es im späteren Programmablauf zu Situationen kommen kann, in welchen für große Objekte kein zusammenhängender Speicherbereich gefunden werden kann und diese Objekte nicht erzeugt werden können. Der *Mark and Compact* Algorithmus setzt sich ebenfalls aus zwei Phasen zusammen, defragmentiert aber gleichzeitig den Speicher.

Mark Analog zum *Mark and Sweep* Verfahren erfolgt hier das Markieren der erreichbaren Objekte.

Compact Löscht die unbenutzten Objekte und führt anschließend eine Defragmentierung des zum Programm gehörenden Datenspeichers in drei Schritten durch:

- Berechnung der Zieladresse für jedes Objekt
- Aktualisierung alle Referenzen durch die neu berechneten Adressen
- Verschieben der Objekte an die neue Adresse

Nachteilig ist das ständige Kopieren der Objekte, auch wenn sich deren Zustand nie geändert hat. Dadurch wird ein relativ großer Overhead erzeugt, den die nachfolgenden Algorithmen zu verhindern versuchen.

Stop and Copy

Das *Stop and Copy* Verfahren unterteilt den *Heap* in die zwei Bereiche **Teil A** und **Teil B**. Beim Erzeugen neuer Objekte werden diese zunächst immer in **A** abgelegt, der **Teil B** bleibt unberührt. Ist dieser Bereich vollständig gefüllt, d.h. es können dort keine neuen Objekte mehr eingeordnet werden, beginnt die *Garbage Collection*. Analog zum mehrfach verwendeten *Mark* Schritt, der im Vorfeld beschriebenen Algorithmen, werden wieder alle erreichbaren Objekte gesucht. Diesmal erfolgt außer dem Markieren noch ein Kopieren der erreichbaren Objekte in den **Teil B** des Heaps. Nach diesem Schritt befinden sich nur benutzte Objekte in **B** und die dortigen Referenzen müssen in einem erneuten Durchlauf über alle enthaltenen Objekte angepaßt werden. Für die weitere Arbeit wird nur noch der **Teil B** benutzt, bis dieser ebenfalls voll ist. Dann beginnt der Vorgang erneut, allerdings werden die Objekte jetzt von **B** nach **A** kopiert.

Ein Vorteil gegenüber dem *Mark and Compact* Verfahren ist eine geringere Einzellaufzeit, da der verwendete Speicherbereich kleiner dimensioniert ist. Die Gesamtlaufzeiten⁴¹ des

⁴¹gemeint sind hier alle während der Programmausführung durchgeführten Speicherbereinigungen

Mark and Compact und des *Stop and Copy* Verfahrens weichen asymptotisch betrachtet nicht voneinander ab, da eine Speicherbereinigung bei der Anwendung des *Stop and Copy* Verfahrens öfters durchgeführt werden muß. Allerdings verteilt sich diese Laufzeit nun gleichmäßiger über die gesamte Laufzeit des Programmes.

Generationell

Die *generationelle Speicherbereinigung* nutzt die Situation aus, daß in der Praxis die Lebensdauer von Objekten meist sehr unterschiedlich ist. Einerseits existieren langlebige Objekte die fast die gesamte Programmlaufzeit gültig sind. Bei den bisherigen Verfahren wurden diese Objekte immer wieder hin und her kopiert, was im Hinblick auf die Lebensdauer ein unnötiger Vorgang ist. Andererseits ist die so genannte „*Infant Mortality*“ sehr hoch. Es gibt viele Objekte, die bereits kurz nach der Speicherzuweisung nicht mehr erreichbar sind. Ein Beispiel hierfür sind Iterator-Objekte einer Schleife, die oft nur einen Durchlauf lang aktiv sind.

Aus diesem Grund wird der Heap in verschiedene Altersklassen oder Speicherbereiche eingeteilt (Abbildung 2.6). Jeder dieser Bereiche wird durch ein *Stop and Copy* Verfahren verwaltet und splittet sich deshalb wiederum in zwei Teile. Alle Objekte werden zunächst in der jüngsten Altersklasse angelegt und erst nach einer bestimmten Zeitspanne⁴² in die nächst höhere Altersstufe verschoben.

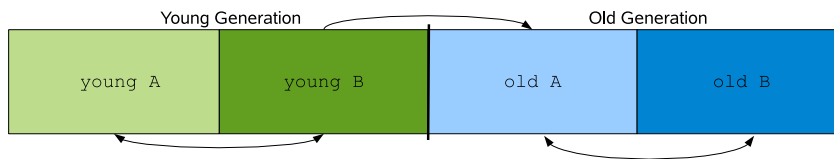


Abbildung 2.6: Ein Beispiel für die Struktur einer *generationellen Speicherbereinigung* mit zwei Generationen. Die Pfeile symbolisieren die möglichen Verschieberichtungen der Objekte.

Java Garbage Collection

Die verschiedenen Java-Versionen benutzen unterschiedliche Speicherbereinigungsverfahren und wurden im Laufe der Zeit und von Version zu Version verbessert. Hier sollen die bei der Laufzeitumgebung von Sun Microsystems verwendeten Techniken im Ansatz beschrieben werden. Da die genaue Implementierung des *Garbage Collector* nicht Bestandteil der *Java Language Specification* [45] ist, können andere Laufzeitumgebungen⁴³ hiervon abweichende Strategien verwenden.

Prinzipiell wird bei den Laufzeitumgebungen von Sun ein generationeller Ansatz verwendet, wobei in jeder Generation ein anderes Verfahren eingesetzt werden kann. Neuere

⁴²oder nach einer festen Anzahl von überlebten *Garbage Collector* Durchläufen der jeweiligen Stufe

⁴³z.B. JRockit, Kaffe, GNU Java, IBM JVM

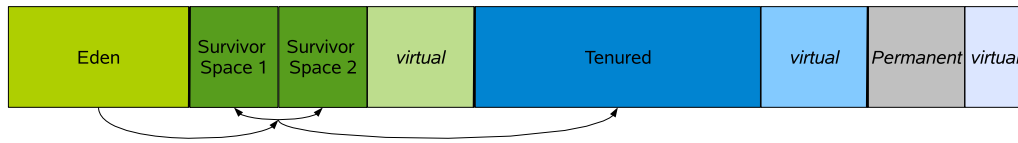


Abbildung 2.7: Schematische Aufteilung des von Java verwalteten Speichers. Die mit *virtual* gekennzeichneten Teile sind von der Laufzeitumgebung reserviert, aber noch nicht in die Speicherverwaltung mit einbezogen. Bei Bedarf können die einzelnen Bereiche durch diesen *virtual* Speicher vergrößert werden. Die Pfeile kennzeichnen wieder mögliche Verschieberichtungen der Objekte.

Versionen der aktuellen *HotSpot JVM* verfügen über mehrere Alternativen und erlauben dem Anwender die Auswahl der Verfahren.

Beim Start der virtuellen Maschine wird ein maximaler virtueller Adreßbereich reserviert und normalerweise erst dann im physischen Speicher belegt, wenn er benötigt wird. Der so belegte Speicher teilt sich wieder in zwei Generationen auf. Die Generation „*Young*“ besteht aus einem als *Eden* bezeichneten Bereich und zwei *Survivor Spaces* (siehe *Abbildung 2.7*). Die Erzeugung von Objekten findet immer im *Eden* statt. Es ist immer ein *Survivor Space* frei, der beim nächsten Durchlauf der Speicherbereinigung zur Aufnahme der Objekte aus *Eden* und dem anderen *Survivor Space* dient. Alle jungen oder gerade erst erzeugten Objekte werden auf diese Weise so lange zwischen den *Survivor Spaces* hin und her verschoben, bis sie alt genug sind, um in die nächste Generation (*Tenured*) verschoben zu werden. In diesem Bereich wendet die JVM standardmäßig einen *Mark and Compact* Algorithmus zur Reinigung an.

Eine dritte, eng mit der Generation *Tenured* verbundene, Generation ist die Generation *Permanent*. Diese Generation nimmt eine Sonderstellung ein, da sie Daten enthält, die von der Laufzeitumgebung benötigt werden, um Objekte zu beschreiben, für die es auf der Java-Sprachebene keine Entsprechung gibt. So werden zum Beispiel Objekte, die Klassen und Methoden beschreiben, in der Generation *Permanent* gespeichert.

2.5 Zusammenfassung

In diesem Kapitel wurden einige der grundlegendsten parallelen Hardwarearchitekturen und Modelle kurz vorgestellt. Hierbei wurde vor allem auf Systemarchitekturen mit einem gemeinsamen Hauptspeicher eingegangen. Solche Einzelsysteme können aber auch in einem Netzwerk miteinander verbunden sein und erhöhen so das vorhandene Potential für parallel durchführbare Berechnungen.

Für den Programmierer eines parallelen Programmes ergeben sich eine Reihe von Möglichkeiten, um die Parallelität solcher Computersysteme effizient auszunutzen. Meist muß man sich dann aber auf ein Programmiermodell festlegen. Zur Unterstützung des Programmierers gibt es deshalb eine Fülle von erweiterten Programmiermodellen und Laufzeitsystemen. Einige davon wurden in diesem Kapitel kurz besprochen. Ein weiterer

Schwerpunkt dieses Kapitels lag in der Beschreibung von internen Funktionsweisen der Java-Laufzeitumgebung. Die im Rahmen dieser Arbeit entstandene verteilte Laufzeitumgebung wurde mit der Programmiersprache Java erstellt. Deshalb sind diese hier vorgestellten Mechanismen für die in den nachfolgenden Kapiteln besprochenen Sachverhalte relevant.

Kapitel 3

Das DEE Programmiermodell

I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.

(Tony Hoare)

Dieses Kapitel umfaßt die allgemeine Beschreibung des im Rahmen dieser Arbeit entwickelten parallelen verteilten Programmiermodelles und die Definition einer Menge von notwendigen Funktionen, die eine zugehörige Laufzeitumgebung enthalten muß. Detailliert sollen hier zunächst das Systemverhalten, der Funktionsumfang und die abstrakten Schnittstellen dokumentiert werden. Zusammenfassend werden dann diese Spezifikationen immer als *DEE* (*Distributed Execution Environment*) bezeichnet. Eine Prototypimplementierung für *DEE* wird im nachfolgenden Kapitel beschrieben.

Wie oben erwähnt, handelt es sich bei der *DEE* Umgebung um die Spezifikation eines Programmiermodelles und des dazugehörigen Laufzeitsystems. Bei der Festlegung der einzelnen Funktionen und Verhaltensweisen von *DEE* stand eine möglichst hohe Flexibilität bei der Umsetzung als Laufzeitumgebung im Vordergrund. Dies erkennt man z.B. daran, daß keine expliziten Bedingungen an die Struktur der Laufzeitumgebung einer letztendlichen Implementierung gestellt werden. So wird u.a. die Erweiterung der Programmiersprache in Form von Schlüsselwörtern¹ zur Kennzeichnung bestimmter Mechanismen vermieden. Dies ermöglicht die Umsetzung als reine Bibliothek. Denkbar sind aber auch Versionen mit Nutzung einer speziellen virtuellen Laufzeitmaschine (VM). Die Beschreibung der Funktionen erfolgt allgemein ohne Bezug zu einer Programmiersprache. Hierdurch sollen verschiedene Programmiersprachen unterstützt werden. Für mögliche Java-Implementierungen des *DEE* existiert eine Schnittstellensammlung in Form einer Bibliothek. Diese enthält alle beschriebenen Funktionen, welche von einer Laufzeitumgebung realisiert werden müssen. Die im nachfolgenden Kapitel 4 beschriebene Prototypimplementierung der *DEE* ist als Java-Bibliothek realisiert.

Das von *DEE* angebotene hybride Programmiermodell (*vgl. dazu Kapitel 3.2*) erweitert das Konzept der Threads um eine Ausdehnung über einzelne Hostsysteme hinweg auf

¹analog zu JavaParty, *vgl. dazu Kapitel 2.3.6*

verteilte Ausführungsumgebungen. Die parallelen Kontrollflüsse in einem *DEE* Programm können, wie im Threadmodell, über einen gemeinsamen Speicher Daten austauschen, obwohl keine Ausführungsplattform mit gemeinsamem Speicher vorausgesetzt wird. Eine abstrakte Schicht des *DEE* (siehe Abbildung 3.1) sorgt für die Bereitstellung dieses virtuellen Speichers, der den im Threadmodell möglichen Zugriff auf gemeinsame Daten nachbildet. Gleichzeitig ist auch das Versenden bzw. Empfangen von Nachrichten zwischen den Kontrollflüssen erlaubt und ermöglicht so die gemischte Nutzung von threadbasierter und nachrichtenbasierter Programmierung.

Für den Benutzer ist die genaue Art und Weise der Ausführung verborgen. Das auszuführende Programm ist auf einem System mit nur einem Prozessor genauso lauffähig wie in einem Multi-Cluster-System mit einer Vielzahl unterschiedlicher Multiprozessorknoten. Zum Zeitpunkt der Erstellung eines *DEE* Programmes müssen keine Annahmen über die Struktur der Ausführungsplattform getroffen werden.

Das *DEE* System besteht aus vier Schichten mit Kernkomponenten, welche unterschiedliche Aufgaben abstrahieren. Jede der Kernkomponenten enthält ein oder mehrere Module, die vollständig ausgetauscht oder an die jeweiligen Bedürfnisse angepaßt werden können. Lediglich die Schnittstellen und das Verhalten der einzelnen Teile sind fest definiert und können nicht verändert werden. Kernkomponenten ab der Schicht 3 müssen nicht zwingend vorhanden sein, d.h. eine Implementierung kann auch aus einer Teilmenge der definierten Kernkomponenten bestehen. Aus diesem Grund und für die weitere klare Abgrenzung dürfen keine horizontalen Abhängigkeiten zwischen Kernkomponenten einer Schicht existieren. Vielmehr greifen Kernkomponenten nur auf die Funktionen der direkt darunter liegenden Schicht zu (vertikaler Zugriff). Einige der definierten Kernkomponenten erstrecken sich über mehrere Schichten. In diesen Fällen sind *horizontale* Zugriffe erlaubt.

Abbildung 3.1 zeigt den vereinfachten schematischen Aufbau des *DEE* -Systems und das Zusammenspiel der einzelnen Module und Kernkomponenten. Dort wird ersichtlich, daß die oberste mit *DEE-Service* bezeichnete Kernkomponente sowohl Abhängigkeiten zum *DTM* als auch zum *System-Core* enthalten darf, da sie sich über mehrere Schichten erstreckt. Abhängigkeiten von der *DTM*-Kernkomponente zur *DSM*-Kernkomponente sind allerdings nicht erlaubt, weil dies einer horizontalen Abhängigkeit entspricht.

Die in der Abbildung 3.1 gezeigten Kernkomponenten und einige Module sind in der folgenden Übersicht mit jeweils kurzen Erläuterungen zusammengefaßt.

System-Core: Diese Kernkomponente enthält grundlegende Funktionalitäten und einige Module für die Arbeit des Systems. Das **Kommunikationsmodul** bietet Methoden zum Versenden und Empfangen von Nachrichten zwischen den einzelnen Hostsystemen. Beim **Namespace** handelt es sich um ein Modul für die Bereitstellung eines einheitlichen Namensraumes für die einzelnen Hosts. Die Namen sind dabei unabhängig von der verwendeten I/O Schicht und somit vom Verbindungsnetzwerk.

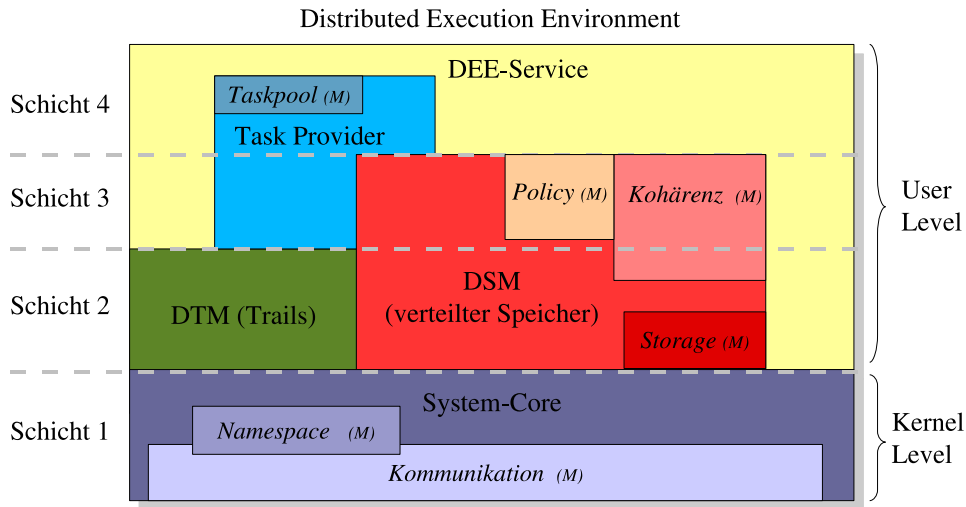


Abbildung 3.1: Übersicht der im *DEE* System definierten Schichten und Kernkomponenten mit einigen Modulen. Die Module sind hier mit dem Zusatz *M* versehen.

DTM: Das *Distributed Trail Model* definiert ein einfaches Modell für die Abarbeitung von parallelen und verteilten Berechnungen. Da der gemeinsame Speicher nicht zwingend benutzt werden muß, kann die Programmierung mittels Nachrichtenaustausch erfolgen. Hierdurch ergeben sich meist effiziente Programme.

DSM: Die als *Distributed Shared Memory* bezeichnete Kernkomponente verwaltet einen gemeinsamen globalen Speicher. Bei der Benutzung, der dort zur Verfügung gestellten Funktionen, können die parallelen Kontrollflüsse direkt Daten verändern, ohne deren genauen Speicherort kennen zu müssen. Für den Programmierer ergeben sich einige Erleichterungen und die Erstellung von Programmen wird vereinfacht.

Der gesamte DSM gliedert sich in mehrere Abstraktionsebenen mit Modulen zur Sicherstellung der Datenkonsistenz im Speicher, sowie Modulen mit Datenverteilungsstrategien für die Aufteilung der Nutzerdaten in einzelne Speicherblöcke².

DEE-Service: Hierbei handelt es sich um eine Serviceschicht für den Anwender mit einigen Standardroutinen für den Benutzer. Insbesondere kann das Benutzerprogramm über solche Routinen die Verfügbarkeit von Kernkomponenten erfragen und somit feststellen, ob es selbst auf der Laufzeitumgebung ablauffähig ist.

Task-Provider: Die *Task-Provider*-Kernkomponente unterstützt ein Taskkonzept (siehe Kapitel 3.4.1).

Auf die Funktionalität und das Verhalten der verschiedenen Kernkomponenten soll im folgenden genauer eingegangen werden.

²Im eigentlichen Sinn handelt es sich nicht um Speicherblöcke, sondern um Datenzellen, die als Objekte realisiert sind.

3.1 Kernel Level

Die *System-Core*-Kernkomponente sowie die dort enthaltenen Module *Kommunikation* und *Namespace* können als Low-Level-Schicht mit überwiegend autonomen Funktionen zusammenfassend betrachtet werden. Zunächst soll hier der Funktionsumfang der *System-Core*-Kernkomponente beschrieben werden. Diese Kernkomponente übernimmt die Verwaltung von Ressourcen und die Bereitstellung von Standardinformationen des Laufzeitsystemes. In der nachfolgenden Auflistung sind diese Aufgaben zusammengefaßt.

- Einlesen und Überprüfen der Konfiguration
Eine Konfiguration beschreibt die genaue Zusammensetzung des Laufzeitsystems, insbesondere die zu verwendenden Kernkomponenten. Beispielsweise kann hier die Benutzung einer spezieller DSM Implementierung durch den Benutzer angefordert werden. Weiterhin enthält die Konfiguration Angaben zur Zusammensetzung des Rechnersystems, d.h. die Anzahl der benötigten Knoten bzw. die Anzahl von erforderlichen Prozessoren.
- Bereitstellung von Informationen zum aktuellen Laufzeitsystem
Hierzu zählen z.B. die Anzahl der verfügbaren Prozessoren/Kerne oder die relative Leistungsfähigkeit des Hostsystems.
- Initialisierung aller Systemkomponenten der Laufzeitumgebung
Die Kernkomponenten und Module müssen in einer bestimmten Reihenfolge gestartet werden, da zum Teil einige Abhängigkeiten bestehen (z.B. das DSM-Subsystem und die Kommunikationskomponente).
- Startroutinen der Laufzeitumgebung
In einer verteilten Laufzeitkonfiguration (Zusammensetzung aus mehreren unabhängigen Hostsystemen) muß erst auf die Anmeldung aller Hostsysteme gewartet werden, ehe ein auszuführendes Programm verteilt und gestartet werden kann.

Namespace-Modul Für die Generierung von eindeutigen Namen (*deeHostName*) für jedes zur Laufzeit in eine *DEE* Umgebung eingebundene Hostsystem ist das *Namespace*-Modul verantwortlich. Insbesondere lassen sich durch die von diesem Modul bereitgestellte Funktionalität verschiedene Namensräume anlegen und dadurch unterschiedliche Subnetze schaffen. Die vom *Namespace*-Modul generierten *deeHostName*-Adressen sind unabhängig von den bei der Kommunikation verwendeten physikalischen Adressen bzw. den Namen der Hostsysteme (d.h. kein Bezug auf IP-Adressen). Vom gesamten *DEE* System wird immer dieser eindeutige *deeHostName* zur Adressierung verwendet. Das für die Kommunikation zuständige *Kommunikationsmodul* löst solche *deeHostName* Angaben dann in entsprechende Adressen der verwendeten Kommunikationstechnologie auf. Dieser Vorgang läuft transparent ab und ist für den Programmierer nicht sichtbar.

Kommunikationsmodul Die allgemeine Schnittstellendefinition des *Kommunikationsmodules* stellt Basisdienste für die direkte Kommunikation zwischen zwei Hostsystemen in einem *DEE* Laufzeitsystem zur Verfügung. Diese Dienste umfassen im wesentlichen nicht blockierende Sendeoperationen und eine blockierende Operation für das Empfangen von Nachrichten zwischen einzelnen Hostsystemen. Die Sendeoperationen gliedern sich in eine Einzel- sowie in eine Broadcastoperation und versuchen die Daten unmittelbar zu versenden. Der Sendevorgang kann dabei von einem eigenen Thread durchgeführt werden, so daß es zu keiner Verzögerung kommt und die Kontrolle sofort an den Aufrufer zurückkehrt. Die Empfangsoperation wartet hingegen, bis entsprechende Daten vorhanden sind. Es ist dabei unerheblich, ob es sich beim Sendevorgang um eine Einzeloperation oder um einen Broadcast handelt. Das *Kommunikationsmodul* muß sicherstellen, daß verschiedene Nachrichten eines Senders an einen Empfänger sich nicht überholen und in der Sendereihenfolge des Absenders auch beim Empfänger eintreffen. Bei unterschiedlichen Sendern darf sich die Reihenfolge zwischen den Nachrichten ändern.

Für die Bezeichnung der Empfänger- (*receiverHostName*) und Senderadressen (*senderHostName*) werden die vom *DEE Namespace*-Modul generierten eindeutigen Namen (*deHostName*) der beteiligten Hostsysteme verwendet³. Die genaue Funktionsweise der durchgeführten Kommunikation ist für die höheren *DEE* Schichten (*siehe Abbildung 3.1*) nicht sichtbar. Auf die Definition von komplexen Operationen, wie z.B. die Gatter/Scatter Befehle aus dem MPI-Standard, wird verzichtet. Damit ist eine große Menge von möglichen Implementierungen auf der Basis von RMI, UDP oder nativen Lösungen⁴ denkbar und besser realisierbar.

Die Operationen sind hier als Pseudocode dargestellt:

`send(data, receiverHostName)`: Sendet die Daten `<data>` an den angegebenen Empfänger `<receiverHostName>`.

`broadcast(data)`: Sendet die Daten `<data>` an alle bekannten Hostsysteme.

`data receive(senderHostName)`: Wartet auf den Empfang der Daten `<data>` vom angegebenen Absender `<senderHostName>`.

Die *DEE* Umgebung ist nicht auf Hostsysteme in einem einzelnen Netzwerk beschränkt, sondern kann diverse Kommunikationstechnologien nutzen. Nachrichten, die zwischen verschiedenen Netzen gesendet werden, müssen über *Gateway-Server* geroutet werden. In *Abbildung 3.2* ist eine Systemkonfiguration mit vier unterschiedlichen Kommunikationstechnologien skizziert. Ein Gateway, d.h. die *Kommunikationskomponente* eines Hostsystems, muß dabei wenigstens zwei Netzwerktypen gleichzeitig unterstützen, damit Nachrichten weitervermittelt werden können. Die in der *Abbildung 3.2* dargestellten

³Die Generierung des Namens geschieht automatisch beim Anmelden des Hostrechners am System.

⁴Beispielsweise High-speed Networking Stacks (InfiniBand, iWARP, Myrinet), deren Hardware bzw. Treiber direkt mit dem JNI programmiert werden könnten.

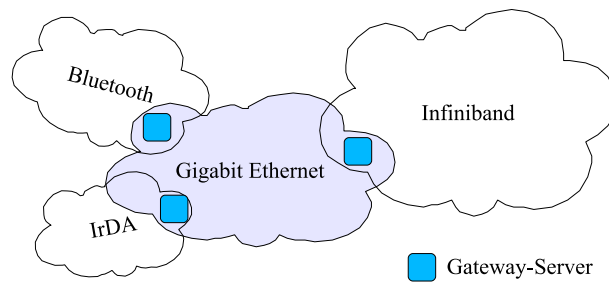


Abbildung 3.2: Abgebildet ist eine mögliche Konfiguration, die unterschiedliche Übertragungstechniken nutzt.

Bluetooth- und IrDA-Netze sind aufgrund ihrer geringen Bandbreite nicht für einen intensiven Nachrichtenaustausch geeignet. Ihr Einsatzfeld ist eher im Monitoring oder beim Anstoßen von Berechnungen angesiedelt.

3.2 DTM: Programmiermodell und Trails

Jedes Programm, welches das *DEE* Modell benutzt, besteht aus einer Menge eigenständiger Kontrollflüsse, den sog. **Trails**. Jeder dieser Trails wird auf einen eigenen Thread abgebildet, wobei meist nur eine begrenzte Anzahl der erzeugten Trails auf einem einzelnen Hostsystem abgearbeitet wird. Die Trails werden vielmehr auf die zur Verfügung stehenden Hostsysteme verteilt und können daher zunächst nur auf Daten anderer Trails zugreifen, die sich ebenfalls auf diesem Host befinden (lokaler MM). Wird der vom *DEE* Laufzeitsystem angebotene DSM benutzt, fällt diese Einschränkung weg und alle Trails haben den Zugriff auf einen gemeinsamen globalen Speicher. Damit gleicht die Programmierung weitestgehend dem des Threadmodelles. Zusätzlich können die Trails aber auch durch Nachrichtenaustausch kommunizieren. Dies bietet sich vor allem bei der verteilten Ausführung an.

Für einen zu implementierenden Algorithmus kann der Anwender die Strategie wählen, die ihm am geeignetsten erscheint. Insbesondere können *distributed* und *shared memory* Konzepte gleichzeitig verwendet werden.

Zum Starten der Trails bietet die Laufzeitumgebung zwei Befehle an:

`parallel(runningCode)`: Die genaue Anzahl von zu startenden Trails und eventuell weitere notwendige Parameter werden vom Laufzeitsystem automatisch bestimmt.

`parallel(runningCode, parameter)`: Der Benutzer kann mit Hilfe des Übergabeparameters `<parameter>` diverse Eigenschaften selbst bestimmen. Hierzu gehört u.a. die Anzahl der zu startenden Trails. Die restlichen Parameter und deren Bedeutung sind von der Implementierung abhängig.

Alle durch einen Aufruf von `parallel` erzeugten Trails bilden ein Trailteam. Der auszuführende Code wird im Parameter `runningCode` übergeben, mit dessen Abarbeitung unmittelbar nach der Erzeugung begonnen wird. Die Kontrolle kehrt erst nach Beendigung aller Trails an den Aufrufer zurück. Es ist auch möglich, aus einem Trail heraus weitere `parallel` Aufrufe abzusetzen, wodurch neue Trailteams erzeugt werden. Zur eindeutigen Identifikation einzelner Trails werden diese bei der Teamerzeugung von 0 bis n fortlaufend durchnummeriert.

3.2.1 Ausführungsmodell

Ähnlich dem OpenMP Standard [102] (*siehe Kapitel 2.3.3*) entscheidet die Laufzeitumgebung ohne Zutun des Programmierers, wie und auf welchen Knoten im Netzwerk die einzelnen Trails gestartet werden. Falls erwünscht, kann der Programmierer auch die Anzahl und die bevorzugte Verteilung der Trails (`parameter`) vorgeben. Die Anzahl der Trails in einem Trailteam bleibt während dessen Lebensdauer normalerweise konstant, weil dies die Programmierung erleichtert. Denkbar ist aber auch eine dynamische Veränderung der Anzahl der Trails in einem Trailteam. Dieser Fall kann beispielsweise eintreten, wenn die beteiligten Hostsysteme nur temporär Ressourcen zur Verfügung stellen (Grid).

Es lassen sich drei Varianten unterscheiden, wovon mindestens eine dieser Varianten von einer *DEE* Implementierung bereitgestellt werden muß.

statisches Modell Die Anzahl der maximal erzeugbaren Trails sowie deren Verteilung auf die physikalisch vorhandenen Ressourcen (CPUs, Hostsysteme) wird fest vom System vorgegeben und ändert sich während der gesamten Ausführung nicht.

dynamisches Modell Bei der Erstellung des Trailteams wird anhand der aktuell am Laufzeitsystem angemeldeten Hostsysteme die Anzahl der zu erzeugenden Trails ermittelt. Diese Anzahl und die Verteilung der Trails ist danach fest und kann wie im statischen Modell nicht mehr verändert werden. Je nach Auslastung der beteiligten Hostsysteme können sich so bei jeder Generierung eines neuen Trailteams unterschiedliche Verteilungen der Trails auf die Hostsysteme ergeben.

grid Modell Wie im dynamischen Modell wird hierbei die Anzahl der erzeugbaren Trails erst beim Programmstart ermittelt. Die Anzahl der Trails sowie deren Verteilung ist jedoch nicht fest und kann während der Ausführung variieren. Dieses Modell erfordert neben der Migration von Trails⁵ auf andere Hostsysteme auch eine dynamische Verwaltung der zu berechnenden Daten. Der Benutzer muß mit einer ständig schwankenden Anzahl von Trails rechnen und geeignete Algorithmen einsetzen.

Gibt der Nutzer eine Anzahl von zu startenden Trails beim Aufruf von `parallel` vor, wird versucht, die gewünschte Anzahl von Trails zu erzeugen. Je nach eingesetztem Modell kann dann aber die Verteilung dieser Trails auf die Hostsysteme unterschiedlich ausfallen.

⁵Die Verteilung kann variieren, ggf. können deshalb Trails auf ein anderes Hostsystem verschoben werden.

3.2.2 Speichermodell

Die Abbildung 3.3 zeigt den Aufbau des von *DEE* vorgegebenen Speichermodelles. Der verfügbare Speicher eines Trails kann in die drei Gruppen *privat*, *lokal* und *global* aufgeteilt werden. Die Unterschiede sind hier im einzelnen kurz zusammengefaßt.

- Im *privaten Speicher* kann nur der Eigentümertrail Manipulationen vornehmen.
- Der *lokale Speicher* ist ein Speicherbereich, der allen Trails auf dem lokalen Hostsystem (vgl. Abbildung 3.3) zur Verfügung steht. Da die Trails direkt auf Threads abgebildet werden, haben diese Zugriff auf den gemeinsamen Speicher des Hostsystems und können somit Datenstrukturen teilen. Zur Nutzung kann der Anwender entweder eigene Mechanismen implementieren, z.B. feste Speicherbereiche, oder die vom *DEE* System angebotenen Funktionen verwenden.
- Auf den *globalen Speicher* haben alle Trails Zugriff. Die Verwaltung wird von der DSM-Schicht übernommen.

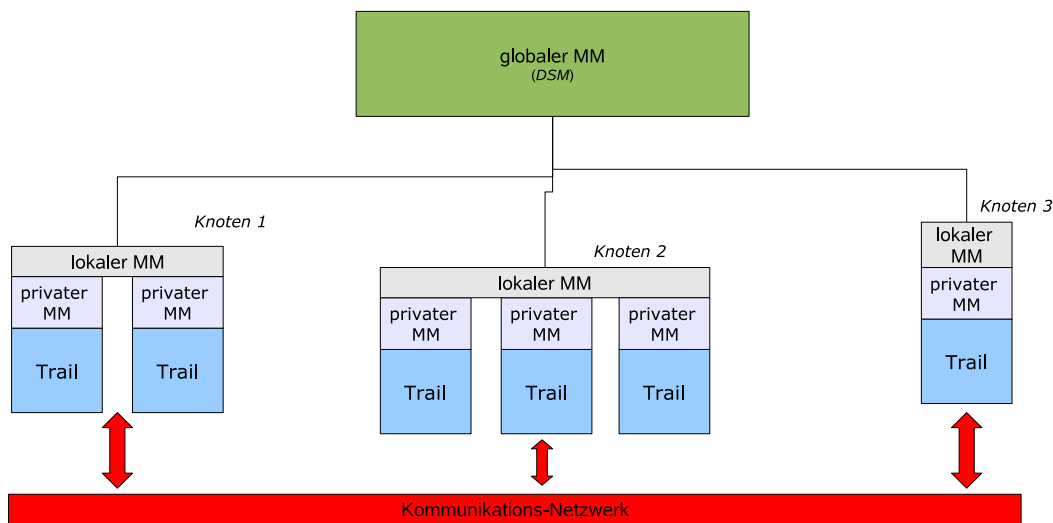


Abbildung 3.3: Trail- und Speichermodell : 3-stufige Speicherhierarchie, welche sich in den privaten Speicher (*privater MM*) eines Trails, den gemeinsamen Speicher (*lokaler MM*) aller Trails eines Host-Systems (Knoten) und einen gemeinsamen globalen (*globaler MM*) Speicher für alle Trails aufteilen läßt.

Der Zugriff auf den *privaten* oder *lokalen* Speicher ist im allgemeinen schneller als auf den *globalen* Speicher, da hierbei kein zusätzlicher Overhead für die Kommunikation mit anderen Hostsystemen notwendig ist. Beim *globalen* Speicher liegen die Daten eventuell im Speichersystem eines anderen Hostsystems und müssen deshalb erst angefragt werden.

3.2.3 Kommunikation und Steuerung

Für die parallele Steuerung und Synchronisation einzelner oder aller Trails werden die aus dem Threadmodell bekannten Mechanismen (Mutex, Barrier und Semaphor) unterstützt. Der Programmierer kann mit deren Hilfe die Granularität der parallelen Ausführung sehr genau und effizient steuern. Für den Informationsaustausch zwischen verschiedenen Trails gibt es in einer *DEE* Laufzeitumgebung zwei generelle Vorgehensweisen, die sich kurz durch die Schlagwörter *Effizienz* und *Einfachheit* charakterisieren lassen.

Effizienz - Nachrichtenaustausch

Für den Nachrichtenaustausch zwischen den Trails sind die folgenden Operationen im DTM vorgesehen:

`send(data, receiverTrailID)`: Sendet die Daten `<data>` an den angegebenen Empfängertrail `<receiverTrailID>`.

`broadcast(data)`: Sendet die Daten `<data>` an alle im Team vorhandenen Trails.

`data receive(senderTrailID)`: Wartet auf den Empfang der Daten `<data>` vom angegebenen Absender `<senderTrailID>`.

Aufbauend auf den Funktionen der *System-Core*-Kernkomponente und des dort angesiedelten *Kommunikationsmodules* erlauben diese Methoden das Empfangen bzw. Versenden von beliebigen Objekten an einzelne (`receiverTrailID`) oder alle Trails eines Trailteams. Als Empfänger- und Absenderadresse werden die eindeutigen Trailnummern $0 \dots n - 1$ benutzt. Von der DTM-Kernkomponente wird die Zuordnung der Trails auf die Hostsysteme vorgenommen und die Nachrichten werden dann entsprechend verteilt. Durch den gezielten Einsatz von Nachrichten kann eine implizite Optimierung des Programmablaufes durch den Programmierer erfolgen. Das resultierende Programm ist dann in verteilten Ausführungsumgebungen oftmals um ein Vielfaches schneller als vergleichbare Programme mit DSM-Nutzung.

Einfachheit - gemeinsamer Speicherbereich

Verwenden die Trails den globalen Speicher, der in der DSM-Kernkomponente bereitgestellt wird, dann genügt es, für die Kommunikation einen gemeinsamen Speicherbereich zu definieren und Daten dort abzulegen bzw. Daten von dort zu lesen. Abgesehen von der Zugriffssynchronisation auf diesen Speicherbereich ist diese Art der Programmierung oftmals für den Anwender einfacher, d.h. der Zeitaufwand für die Erstellung solcher Programme sinkt.

Die genaue Funktionsweise des globalen Speichers soll in den nun folgenden Abschnitten erläutert werden.

3.3 Distributed Shared Memory

Die DSM (oder *Distributed Shared Memory*) Kernkomponente stellt einen virtuellen gemeinsamen aber strukturell verteilten Speicher für die Ablage von Objekten dar. Dieser Speicher entspricht dem globalem Speicher des Trailmodelles (DTM, Abbildung 3.3). Die Ablage und der Zugriff auf Daten in diesem Speicher geschieht über sogenannte Speicherzellen. Diese Speicherzellen sind die kleinste Speichereinheit im DSM und können zur Laufzeit auf verschiedenen Hostsystemen abgelegt sein. Der Zugriff auf eine dieser Speicherzellen geschieht dabei transparent, d.h. dem Programmierer bleiben die Art der Ablage und der genaue Speicherort verborgen. Gemäß der in Kapitel 2.3.4 eingeführten Hierarchie handelt es sich um einen DSM-Objektspeicher.

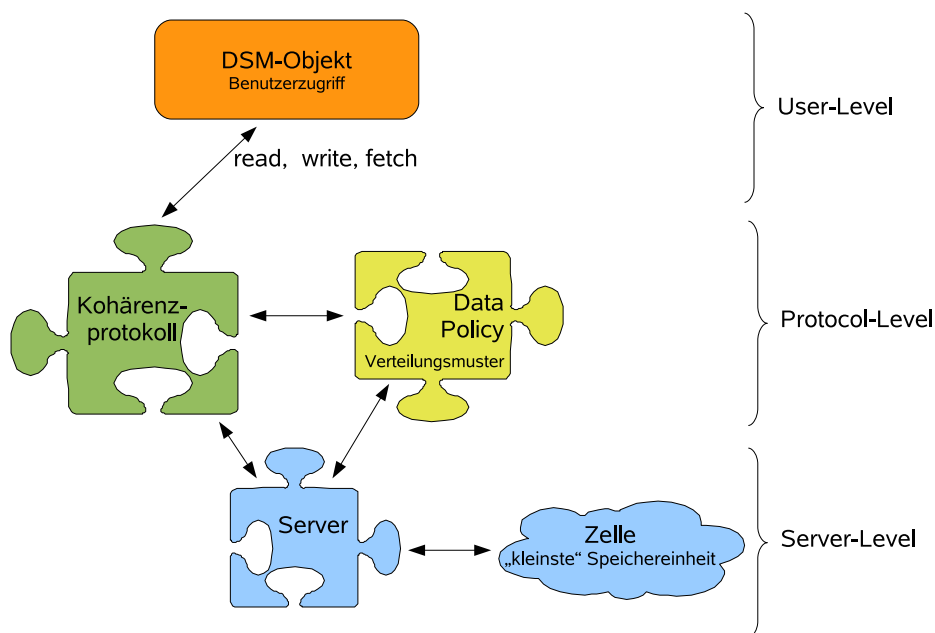


Abbildung 3.4: Aufbau und Zusammenspiel der einzelnen Komponenten/Module des DSM.

Die DSM-Kernkomponente⁶ besteht aus den 5 eigenständigen Modulen: *Zelle*, *Server*, *Data-Policy*, *Kohärenz* und *DSM-Objekt*. Diese lassen sich in drei Stufen: *Server-Level*, *Protocol-Level* und *User-Level* mit unterschiedlichen Abstraktionsebenen anordnen. Die Zuordnung der Module und der Aufbau des DSM-Subsystems sind in Abbildung 3.4 schematisch dargestellt.

Alle Module der DSM-Kernkomponente können direkt durch den Programmierer benutzt werden. Beispielsweise bieten die Module des *Server-Levels* einfache Schreib- und Leseoperationen auf eine DSM-Zelle an. Nutzt ein Programmierer diese Funktionen, kann er die Verwaltung der Daten direkt beeinflussen und gezielt Manipulationen vornehmen.

⁶alternativ wird auch vom DSM-Subsystem gesprochen

Eine Sicherstellung der Konsistenz der Daten kann dann aber nicht garantiert werden, da diese Funktionalität erst in den höheren Schichten des DSM-Subsystems angeboten wird. Der Anwender ist in diesen Fällen selbst für die Korrektheit seiner Daten verantwortlich.

3.3.1 Server-Level

Das *Server-Level* beinhaltet die Module *Server* und *Zelle*⁷. Beide Module sind eng miteinander verknüpft, da eine Serverimplementierung alle Zellen des DSM verwaltet. Zur Vereinfachung und aufgrund dieses engen Zusammenspiels werden im folgenden die beiden Module gelegentlich zusammengefaßt und es wird dann vom *Storage-Modul* oder *DSM-Server-Level* gesprochen. Zunächst folgt jedoch die Beschreibung der Module im Einzelnen.

DSM-Zelle

Eine DSM-Zelle ist die kleinste Speichereinheit im DSM, sie enthält die eigentlichen Daten. Die verfügbare Funktionalität beschränkt sich auf das *Lesen* und *Schreiben* dieser Daten. Hierfür sind die folgenden Operationen auf eine DSM-Zelle definiert:

- data read()* Anfordern einer aktuellen Kopie der Zelle aus dem DSM und Rückgabe dieser Daten in *<data>*.
- write()* Den aktuellen Inhalt der Zelle in den DSM übertragen.
- data get()* Die Daten *<data>* der lokal vorhandenen Kopie der Zelle liefern, ohne eine Leseanfrage an den DSM zu stellen.
- set(data)* Die Daten *<data>* in der lokalen Zelle verändern. Es erfolgt keine Übertragung in den DSM.

Der Zugriff auf die Daten einer Speicherzelle geschieht durch das sog. *Getter/Setter Pattern*, d.h. die Speicherzelle muß *get* bzw. *set* Methoden zur Arbeit mit den Daten anbieten [42]. Das Setzen der Daten durch *set* führt nicht zwangsläufig zu einem Schreiben der Speicherzelle in den DSM. Die Daten werden vielmehr nur in der lokalen Instanz der Speicherzelle verändert und sind nur auf dem lokalen Hostsystem sichtbar. Erst durch einen Aufruf der *write* Operation werden die Daten in den globalen DSM geschrieben.

DSM-Server

Der gesamte virtuelle Speicher⁸ wird von einem oder mehreren DSM-Servern angeboten. Jeder dieser DSM-Server kann den gesamten virtuellen Speicher oder nur ein Teil davon enthalten. Bei der Ablage von Speicherzellen im DSM entscheidet das DSM-Subsystem

⁷Im nachfolgenden werden auch die Begriffe *DSM-Zelle* und *Speicherzelle* als Synonym für das DSM-Modul *Zelle* verwendet, um eine klare Abgrenzung zu erhalten und eventuelle Mehrdeutigkeiten zu vermeiden.

⁸d.h. alle DSM-Zellen

selbständig, welcher DSM-Server verwendet wird. Der Benutzer hat im allgemeinen hierauf keinen Einfluß.

Die resultierende Speicherstruktur des DSM-Subsystems reicht von zentral⁹ bis komplett dezentral¹⁰ und ist stark von der verwendeten Implementierung des/der DSM-Server und deren Anzahl abhängig. Nachfolgend sind mögliche Strukturen aufgeführt:

1. **zentrales Verzeichnis:** Alle Knoten tauschen hierbei nur Informationen mit einem einzigem Masterserver aus. Eine Kommunikation zwischen zwei beliebigen Knoten ist nicht vorgesehen. Nur der Masterserver hat das Recht, neue Zellen anzulegen. Im einfachsten Fall halten die einzelnen Knoten selbst keine lokalen Daten, sondern leiten jede Anfrage an den Masterserver weiter. Da nur ein Server vorhanden ist, kann diese Architektur in einer Laufzeitkonfiguration mit mehreren Hostsystemen relativ schnell zu Performanceengpässen führen und die Leistung des DSM-Subsystems stark beeinträchtigen. Ein wesentlicher Vorteil ist die einfache Implementierung eines solchen Systems.
2. **repliziert:** In dieser Variante ist der gesamte Speicher auf allen Hostsystemen gleichzeitig vorhanden und die Veränderung einer Speicherzelle wird sofort weiter propagiert. Eine Leseanfrage kann sehr schnell beantwortet werden, da keine Netzwerkkommunikation benötigt wird und die Daten auf jedem Hostsystem lokal vorhanden sind.
3. **verteilt:** Dieser Fall erlaubt zwei oder mehr Server, welche die Speicherung und Verwaltung des DSM übernehmen. Dabei ist jeder Server nur für einen Teilbereich der im DSM bereit gehaltenen Zellen verantwortlich. Alle Server sind gleichberechtigt und tauschen auf Anfrage die entsprechenden Zellen untereinander aus. Diese Variante ist aufwendig zu implementieren, verspricht aber auf der anderen Seite eine höhere Leistung, da die Beschränkung auf einen Server entfällt.

Die Zugriffe des Nutzers auf die Daten im DSM erfolgen transparent mit Hilfe der von der DSM-Schicht angebotenen Schnittstellen. Dies bedeutet, daß nach einem Austausch der DSM-Serverkomponente keine Änderungen am Benutzerprogramm oder Laufzeitsystem durchgeführt werden müssen. Das Laufzeitverhalten des Benutzerprogrammes kann sich in solchen Fällen jedoch ändern, da je nach verwendetem DSM-Server die Zugriffszeiten auf die Daten im DSM variieren können.

In Abbildung 3.5 ist eine mögliche Laufzeitkonfiguration für ein DSM-Subsystem mit zwei DSM-Servern abgebildet. Die Speicherung der Daten (X, Y, Z) erfolgt auf den Systemen *Host 1* und *Host 2*. Für den Nutzer ist die Verteilung der Daten nicht sichtbar, die Latenzzeiten für den Zugriff hängen stark vom Speicherort und vom Ort der Anfrage ab. Ein Zugriff von *Host 1* auf X ist wesentlich schneller als ein Zugriff von *Host 3* auf X, da hier im allgemeinen eine Kommunikation über das Netzwerk notwendig ist¹¹. Die

⁹nur ein zentraler DSM-Server

¹⁰jedes Hostsystem ist gleichzeitig auch ein DSM-Server

¹¹Numa Architektur, siehe Kapitel 2.2.2

Aufgabe der höheren DSM-Schichten (*Protocol- und User-Level*) ist es, diese Abhängigkeiten zu erkennen und Optimierungen vorzunehmen.

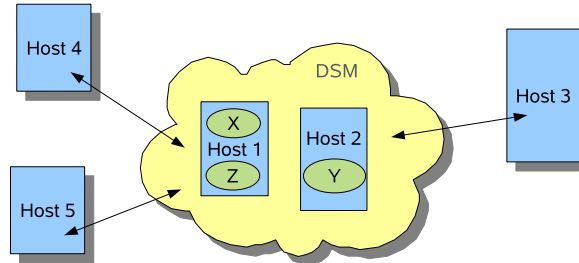


Abbildung 3.5: Laufzeitkonfiguration eines DSM mit 5 Hostsystemen und 3 Speicherzellen (X,Y,Z).

Die Hauptaufgabe der als *DSM-Server-Level* bezeichneten Schicht im DSM-Subsystem (siehe *Abbildung 3.4*) ist die Bereitstellung bzw. Verwaltung von Datenzellen. Diese Funktionalitäten erstrecken sich im wesentlichen auf das *Suchen* und *Erzeugen* von DSM-Zellen. Ein weiteres Merkmal ist die Austauschbarkeit dieser Schicht durch verschiedene Implementierungen. Neben der Erstellung komplett neuer Implementierungen soll es auch möglich sein, bereits erprobte und stabile Bibliotheken mit ähnlicher Funktionalität direkt ins Laufzeitsystem zu integrieren. Für das DSM-Subsystem bieten sich hier z.B. TupleSpace-Lösungen an. Um eine Nutzung im DSM-Subsystem zu ermöglichen, müssen hierfür dann sogenannte *Wrapper* erstellt werden, welche die notwendige Funktionalität eines DSM-Servers auf die Funktionen der entsprechenden Bibliothek abbildet. Ein Nachteil beim Einsatz solcher *Wrapper* können eventuell nicht direkt zugängliche Kontrollmechanismen sein, die zum Teil von den höheren Schichten des DSM-Subsystems (*Protocol-Level*) benötigt werden. Insbesondere ist eine Kontrolle des Speicherortes (Hostsystem) für die Speicherzellen meist nicht frei wählbar. Diese Funktion ist jedoch erforderlich, um eine Lokalisationsoptimierung bei Datenzugriffen durchführen zu können.

Zusätzlich zu den gerade beschriebenen Eigenschaften ist ein *Exclusive Mode*, zur Steuerung des exklusiven Zugriffs auf die Daten, für jede Zelle oder wenigstens für die gesamte *DSM-Server* Infrastruktur notwendig. Eine solche Funktion wird von den höheren Schichten des DSM-Subsystems benutzt, um z.B. Locks für einzelne Zellen zu implementieren. Ohne die Möglichkeit zum gegenseitigen Ausschluß bei DSM-Speicheroperationen kann die Datenintegrität von parallelen *DEE* Programmen nicht sichergestellt werden.

3.3.2 Protocol-Level

Das *Protocol-Level* setzt sich aus einem *Kohärenz-Modul* und einem *Data-Policy-Modul* zusammen. Durch beide Module wird versucht, die Zugriffslatenzen auf Speicherzellen zu reduzieren.

Kohärenz

Die Speicherung der Daten eines DSM-Systems erfolgt in den verteilt existierenden Hauptspeichersystemen der einzelnen Knoten. Speicheranfragen auf nicht lokal vorhandene Daten erfolgen in der Regel über verhältnismäßig langsame Netzwerkverbindungen. Für DSM-Systeme existieren verschiedene Optimierungsstrategien, um solche langsameren Netzwerkzugriffe zu vermeiden oder zu minimieren [133]. Bei *DEE* Implementierungen können allerdings, je nach Umsetzung, nicht alle Strategien verwirklicht werden. Für eine Bibliotheksimplementierung ist z.B. ein Prefetching der Daten schwer realisierbar, während dies für eine Implementierung als virtuelle Laufzeitmaschine besser umzusetzen ist. Die universellste Variante zur Optimierung von Zugriffslatenzen ist deshalb der Einsatz von Kohärenzmechanismen. Deren Aufgabe ist es, häufig verwendete Daten auf den lokalen Knoten zwischenspeichern und für jeden Teilnehmer eine konsistente Sicht auf das Speichersystem zu gewährleisten. Die benötigten Daten müssen dann nicht bei jeder Anfrage von dem oder den DSM-Servern geholt werden.

Die Kohärenzschicht der *DEE* DSM-Kernkomponente enthält für diese Aufgabenstellung eine Schnittstellendefinition. Die Implementierung einer solchen Schnittstelle wird im DSM-Subsystem als Kohärenzprotokoll bezeichnet. Der Aufbau der Kohärenzschicht ist modular und es können verschiedene solcher Kohärenzprotokolle gleichzeitig verwendet werden¹². Durch die Benutzung eines Protokolles kann der Nutzer eine für ihn günstige Strategie bei der Speichernutzung wählen, da jedes Protokoll eine bestimmte Menge von festen Eigenschaften definiert und deren Gültigkeit sicherstellt. Eine Zielsetzung für ein Protokoll kann die oben bereits angesprochene, möglichst schnelle Beantwortung von Leseanfragen sein. Durch Replikation aller Daten des DSM auf jedem Hostsystem läßt sich ein entsprechendes Kohärenzprotokoll implementieren. Bei der Verwendung von großen Datenmengen wird dieses Vorgehen aber unter Umständen durch eine Speicherplatzbegrenzung auf einzelnen (heterogenen) Knoten erschwert oder unmöglich. Aus diesem Grund ist die Implementierung eines allgemeinen Kohärenzprotokolles, welches in allen denkbaren Anwendungsfällen gute Ergebnisse liefert, nicht möglich (*siehe Kapitel 3.3.4*).

Ein Kohärenzprotokoll baut auf den von einer DSM-Zelle bereitgestellten Operationen (*siehe Kapitel 3.3.1 (S.67)*) auf. Für die Sicherstellung der Datenkonsistenz muß der Zugriff des Benutzer auf die DSM-Zellen über die vom Kohärenzprotokoll angebotenen Routinen erfolgen. Hierfür stehen die folgenden Methoden zur Auswahl:

- data read(cell)*: Daten aus der DSM-Zelle *<cell>* lesen.
- write(cell)*: Daten in die DSM-Zelle *<cell>* schreiben, die Änderungen müssen nicht sofort im DSM sichtbar sein.
- data fetch(cell)*: Die Daten aus der DSM-Zelle *<cell>* lesen und dabei möglichst keinen Aufruf in das DSM-Subsystem absetzen.
- flush()*: Ein Schreiben aller gepufferten Daten (DSM-Zellen) in den DSM erzwingen.

¹²Die Protokolle werden mit DSM-Objekten verknüpft. Unterschiedliche DSM-Objekte können deshalb durch andere Protokolle verwaltet werden.

- `clear()`: Löscht einen eventuell vorhandenen Zwischenspeicher, welcher von der Protokollimplementierung zur Pufferung von DSM-Zellen verwendet werden kann. Dieser Zwischenspeicher muß vom Protokoll selbst verwaltet werden.
- `lock(cell, type)`: Der Aufruf liefert eine Lock-Datenstruktur, mit deren Hilfe der konkurrierende Zugriff auf die Zelle `<cell>` durch den Programmierer gesteuert werden kann. Der Parameter `<type>` ermöglicht die Anforderung unterschiedlicher Varianten von *Locks* (*s.u.*).

Die Zugriffskontrolle kann vom Programmierer durch den vom Protokoll verwalteten *Lock* gesteuert werden. Es werden zwei Typen von Locks unterschieden: ein **Readlock** und ein **Writelock**. Ein *Readlock* erlaubt gleichzeitige Zugriffe mehrerer Trails bis ein *Writelock* angefordert wird. Ein solcher *Writelock* erlaubt zu jedem Zeitpunkt nur einem einzigem Trail den Zugriff auf die geschützte Datenstruktur. Alle weiteren Anfragen (*Readlocks* und *Writelocks*) blockieren, bis dieser *Writelock* wieder aufgehoben wird [28, 60].

Data-Policy

Das als *Data-Policy* bezeichnete Modul ist für die Partitionierung von Benutzerdaten zuständig. Eine dynamische Aufteilung größerer Datenmengen in kleinere Datenblöcke bzw. DSM-Zellen oder umgekehrt ist hierdurch möglich. Als Funktionen werden die Methoden *merge* und *split* definiert. Ausgehend von Zugriffsstatistiken des DSM, manipulieren beide Mechanismen den DSM direkt, indem neue Speicherzellen erzeugt oder mehrere Zellen zusammengefaßt werden. Geeignet ist diese Funktionalität z.B. für große Felder, die nur partiell von einigen Trails gelesen oder geschrieben werden. Allerdings müssen für eine sinnvolle Arbeitsweise einige Angaben zur übergeordneten Datenstruktur bekannt sein. Im soeben angesprochenen Beispiel eines verteilten Feldes kennt das Modul die Datenstruktur des Feldes sowie die Zuordnung der einzelnen Feldelemente auf die Speicherzellen. Deshalb gibt es eine enge Kopplung zwischen *Data-Policy* und dem letztendlichen *DSM-Objekt*, welches die Daten für den Benutzer verwaltet. Beim Zugriff auf das DSM-Objekt ist für den Benutzer die *Data-Policy* nicht sichtbar. Der Nutzer kann die zu verwendende *Data-Policy* beim Anlegen eines DSM-Objektes wählen, falls die Implementierung des betreffenden DSM-Objektes dies unterstützt. Eine nachträgliche Änderung während des Lebenszyklus des DSM-Objektes ist nicht vorgesehen.

3.3.3 User-Level und DSM-Objekt

Auf der höchsten Benutzerebene des DSM sind sog. DSM-Objekte angesiedelt. Jedes dieser DSM-Objekte ist im allgemeinen eine Implementierung oder speziell optimierte Variante einer **verteilten Datenstruktur**. Es kann aus einer Vielzahl von DSM-Zellen bestehen und die Daten unterschiedlich verwalten. Abbildung 3.6 skizziert die prinzipielle Funktionsweise am Beispiel eines verteilten Feldes `Feld`. Im Bezug auf die gerade eingeführte Definition ist das *verteilte Feld* `Feld` ein DSM-Objekt. Es besteht aus einer für den Nutzer nicht sichtbaren Menge von DSM-Zellen, die im DSM gespeichert sind

und keine zusammenhängenden Speicherbereiche belegen müssen. Der Zugriff durch den Nutzer geschieht ausschließlich durch die vom DSM-Objekt `Feld` angebotenen Routinen. Dadurch wird die Komplexität der Zugriffe verborgen und die Benutzung des DSM weiter vereinfacht.

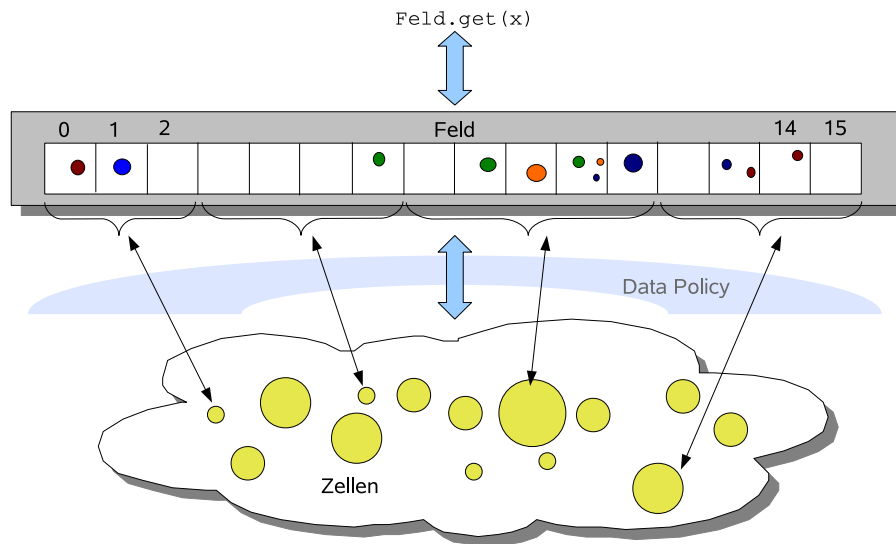


Abbildung 3.6: Dargestellt ist die Zuordnung von Feldeinträge auf einzelne DSM-Zellen des DSM-Objektes `Feld`. Eine zwischengeschaltete *Data-Policy* aus dem *Protocol-Level* des DSM-Subsystems kann die Zuordnung der DSM-Zellen zu den Feldeinträgen dynamisch zur Laufzeit ändern.

3.3.4 Erweiterte Funktionen des DSM-Subsystems

Je nach Anforderungsprofil der zu erstellenden Anwendung gibt es unterschiedliche Optimierungsziele für das DSM-Subsystem, die zum Teil gegenläufig sein können. Beispielsweise könnte die in der *Data-Policy* verankerte Strategie versuchen, immer eine relativ große Datenmenge innerhalb einer DSM-Zelle zu speichern, um die Anzahl der DSM-Speicheranfragen zu reduzieren. Dies könnte man mit dem Designziel einer möglichst groben Granularität verallgemeinern. Auf der anderen Seite ist eine feine Granularität der Daten und Speicherzellen für die Kohärenzprotokolle vorteilhafter, da ein Schreibzugriff dann nur einen relativ kleinen Datenbestand verändert. Dies verringert wiederum die Wahrscheinlichkeit für Kollisionen mit Speicheranfragen von anderen Trails¹³.

Eine allgemeine Strategie zur Lösung solcher Probleme kann nicht vorgegeben werden und muß für jede Problemstellung neu evaluiert werden. Durch den modularen Aufbau der Laufzeitumgebung können allerdings entsprechende Profile vorgegeben und an eigene Bedürfnisse angepaßt werden.

¹³siehe hierzu auch *False-Sharing* Kapitel 2.2.1 (S.28)

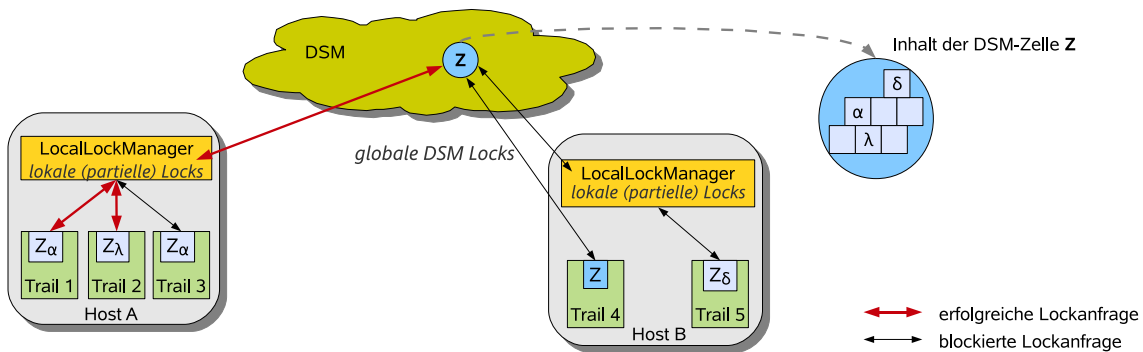


Abbildung 3.7: Die Abbildung skizziert die Funktionsweise des hierarchischen Lockings beim Zugriff mehrerer Trails auf eine DSM-Zelle Z .

Dieser Abschnitt soll noch kurz auf einen weiteren Mechanismus des DSM-Subsystems eingehen, welcher den Zugriff auf gemeinsam verwendete Daten reglementiert und als *LocalLockManager* bezeichnet wird.

Die von der Kohärenzschicht angebotenen *Read-* und *Writelocks* erlauben nur das globale Setzen und Entfernen einzelner Sperren (*Locks*) auf Datenzellen. Falls mehrere Trails eines Hostsystems auf identische DSM-Zellen zugreifen, diese aber nicht vom lokalen DSM-Server verwaltet werden, müssen für jeden dieser Zugriffe aufwendige Netzwerkoperationen abgesetzt werden. Für solche Fälle sieht das Laufzeitsystem die Benutzung eines lokalen Locking-Mechanismus vor, der auf dem Prinzip von hierarchischen Locks basiert [31, 32, 87]. Zur Optimierung der Zugriffe werden dabei globale Locks vom DSM-Subsystem angefordert und für das gesamte Hostsystem verwaltet. Falls mehrere Trails eines Hostsystemes unabhängige Bereiche einer Datenzelle manipulieren, so kann dies nun gleichzeitig erfolgen (Abbildung 3.7). Der *LocalLockManager* stellt die folgenden Methoden zur Verfügung:

- `lock(cell, part)`: Es wird versucht den Teil $\langle part \rangle$ der DSM-Zelle $\langle cell \rangle$ zu sperren. Der aufrufende Trail wird blockiert, bis dieser Vorgang erfolgreich durchgeführt werden konnte.
- `unlock(cell, part)`: Der Lock für den Teil $\langle part \rangle$ der DSM-Zelle $\langle cell \rangle$ wird aufgehoben. Alle bezüglich dieses Locks blockierten Trails werden aufgeweckt und versuchen sofort die Kontrolle über den Lock zu erlangen.

Der Anwender kann diese Methoden verwenden, um Zugriffe auf Datenzellen feingranularer zu steuern. Alternativ ist aber weiterhin der direkte Zugriff über die Locks des *Protocol Levels* (siehe Kapitel 3.3.2) möglich. Beide Mechanismen können gleichzeitig angewendet werden.

3.4 Weitere Kernkomponenten des DEE

Neben den im bisherigen Verlauf beschriebenen *System-Core*-, DTM- und DSM-Kernkomponenten enthält das Schichtenmodell der *DEE* Spezifikation noch die *Task-Provider*- und *DEE-Service*-Kernkomponenten. Auf diese Kernkomponenten soll im nachfolgenden näher eingegangen werden.

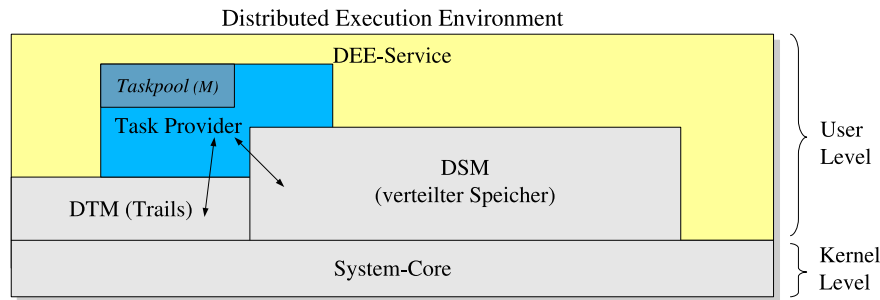


Abbildung 3.8: Das *DEE* Schichtenmodell mit den Kernkomponenten. Die in Kapitel 3.4 besprochenen Kernkomponenten sind dabei farblich hervorgehoben.

3.4.1 Task-Provider

Im *Task-Provider* wird die Generierung und die Berechnung von Tasks unterstützt. Ein Task kann selbst Tasks (Untertasks) erzeugen und in einen Taskpool einfügen. Auftretende Abhängigkeiten zwischen den Tasks, welche die Berechnung blockieren können, müssen vom Programmierer erkannt und gesondert behandelt werden. Das gesamte Thema ist Gegenstand vielfältiger Untersuchungen und hat eine hohe Komplexität [2, 35, 115]. Aufbauend auf den DTM- und DSM-Kernkomponenten erfolgt die (verteilte) Berechnung der im *Task-Provider* verwalteten Tasks durch mehrere Trails. Die Kernkomponente enthält hierfür ein austauschbares *Taskpool*-Modul, in welchem Tasks zur Laufzeit vom Benutzer abgelegt werden können. Das Laufzeitsystem des *Task-Providers* beginnt unverzüglich, die in diesen Taskpool eingestellten Tasks abzuarbeiten¹⁴. Der genaue Ablauf einer solchen taskbasierten Abarbeitung ist z.B. in [59] erläutert. Das DSM-Subsystem ist für die transparente Speicherung von gemeinsam verwendeten Datenstrukturen vorgesehen.

3.4.2 DEE-Service

Die in der *DEE-Service* Kernkomponente enthaltenen Routinen fassen im wesentlichen Standardaufgaben zusammen oder liefern für den Programmierer nützliche Systeminformationen. Dazu zählt z.B. die Initialisierung oder das Beenden aller notwendigen Dien-

¹⁴sofern keine Abhängigkeiten zu anderen Tasks bestehen

ste¹⁵ einer *DEE* Laufzeitumgebung. Die zwingend erforderlichen Funktionen der *DEE-Service*-Schicht sind in der folgenden kurzen Auflistung zusammengestellt.

- Initialisieren und Beenden der Laufzeitumgebung
- Starten von *DEE* Programmen
- Bereitstellen von *Profilinformationen* (s.u.) der *DEE* Laufzeitumgebung

Als Profil wird in diesem Kontext die Zusammensetzung einer *DEE* Laufzeitumgebung aus Kernkomponenten verstanden. Der Anwender kann zu jeder im *DEE* Modell definierten Kernkomponente des *User-Levels*¹⁶ die Existenz und die verfügbaren Versionsinformationen erfragen. Dies ist notwendig, da eine *DEE* Laufzeitumgebung nicht alle beschriebenen *User-Level*-Kernkomponenten und Module enthalten muß. Deshalb ist eine *DEE* Laufzeitumgebung denkbar, welche beispielsweise nur die Implementierung einer DTM-Kernkomponente¹⁷ enthält. Eine Anwendung, die zusätzlich noch auf die Funktionen des *DSM* zurückgreift, kann somit in einer solchen Laufzeitumgebung nicht ausgeführt werden.

Weitere, von der *DEE-Service*-Kernkomponente zu implementierende Benutzerfunktionen werden nicht verlangt. Es ist aber möglich, daß die Implementierung einer Laufzeitumgebung darüber hinausgehende Funktionen bereitstellt. Die im Kapitel 4 beschriebene Prototypimplementierung besitzt z.B. eine interaktive Benutzershell für die Steuerung der Laufzeitumgebung und das Starten von Programmen. Daneben werden von dieser Shell noch zahlreiche Routinen zur Überwachung des Laufzeitsystems angeboten (*vgl. dazu Kapitel 4.4 (S.129)*).

3.5 Schlußbemerkungen

In diesem Kapitel wurde ein allgemeines Programmiermodell für die Ausführung von parallelen verteilten Programmen beschrieben. Kernpunkte des Modelles sind das *Distributed Trail Model (DTM)* und ein *Distributed Shared Memory (DSM)*. Beide Komponenten unterstützen die transparente Ausführung von parallelen Kontrollflüssen in einer verteilten Ausführungsumgebung mit mehreren unabhängigen Hostsystemen. Im Unterschied zu den in Kapitel 2.3 vorgestellten Programmiersystemen ist mit dem *DEE* Programmiermodell die Erstellung von plattformunabhängigen parallelen Programmen möglich, welche thread- und nachrichtenbasierte Konzepte gleichzeitig enthalten. Wird die *DSM*-Kernkomponente bei der Programmierung verwendet, dann ähneln die resultierenden trailparallelen Programme sehr stark ihren threadparallelen Vorbildern. Allerdings ist mit solchen trailparallelen Programmen auch eine verteilte Ausführung möglich.

¹⁵z.B. Hintergrundthreads für die Verarbeitung von Systemnachrichten

¹⁶siehe Kapitel 3 (S.57) und Abbildung 3.1 bzw. Abbildung 3.8

¹⁷Die *System-Core*-Kernkomponente befindet sich im *Kernel-Level* und muß deshalb in jedem *DEE* Laufzeitsystem vorhanden sein.

Das *DEE* System übernimmt einige Mechanismen aus bereits existierenden Programmierumgebungen und erlaubt den flexiblen Austausch bestehender Systemkomponenten im Laufzeitsystem. So wird z.B. die aus OpenMP bekannte automatische Bestimmung der Anzahl von zu startenden Threads beim Programmstart übernommen und ein analoger austauschbarer Mechanismus in der *DTM*-Kernkomponente für die Trails angeboten. Da sich diese Austauschbarkeit über nahezu alle Bereiche der *DEE* Umgebung erstreckt, können maßgeschneiderte Laufzeitumgebungen für die Problemstellungen des Benutzers erstellt werden, ohne die existierenden *DEE* Anwendungsprogramme abändern zu müssen. Es ist sogar denkbar, daß verschiedene Aspekte der in Abschnitt 2.3.6 vorgestellten Programmiersysteme mit einem *DEE* Laufzeitsystem nachgebildet werden oder Teile dieser Systeme in ein *DEE* Laufzeitsystem integriert werden. Die verteilten Felder des *Global Arrays* Toolkits könnten beispielsweise als Grundlage für eine plattformabhängige DSM-Implementierung dienen.

Im folgenden Kapitel wird das hier beschriebene Programmiermodell in einer Prototypimplementierung als Laufzeitsystem umgesetzt und dokumentiert.

Kapitel 4

Die DEE Prototypimplementierung

Beware of bugs in the above code; I have only proved it correct, not tried it.

(Donald E. Knuth)

Ausgehend von der im vorangegangenen Kapitel 3 erfolgten Beschreibung von *DEE* als abstrakte Ausführungsplattform für die Durchführung verteilter paralleler Berechnungen, wurde eine Prototypimplementierung erstellt. Diese Implementierung basiert vollständig auf einer Java-Bibliothek und ist deshalb ohne zusätzliche Anforderungen auf jedem Hostsystem mit Java SE lauffähig. Um die im Vorfeld beschriebene allgemeine Spezifikation und die hier diskutierte Implementierung besser voneinander abgrenzen zu können, wird im folgenden die Bezeichnung *DEE* immer für einen Bezug auf das zugrundeliegende Programmiermodell mit der zugehöriger Beschreibung der Laufzeitumgebung und der Name des Prototypes *Beluga* für die Teile, welche die Implementierung betreffen, verwendet.

In diesem Kapitel werden die internen Mechanismen und die Funktionsweise der Prototypimplementierung *Beluga* beschrieben, welche notwendig sind, um das durch *DEE* vorgegebene Verhalten zu erreichen. Dabei werden die durch *DEE* definierten Schichten und Kernkomponenten jeweils getrennt betrachtet und deren konkrete Implementierung beschrieben. Die Dokumentation der für den Anwendungsprogrammierer relevanten Schnittstellen und Routinen erfolgt hierbei meist am Anfang dieser Betrachtungen. Eine zusätzliche Auflistung von Klassen und Schnittstellen der *Beluga* Prototypimplementierung findet sich im Anhang B.1.

Die primäre Zielplattform des *Beluga* Prototypes sind z.B. Computersysteme in einem Unternehmensnetzwerk oder Clustersysteme. In solchen Umgebungen kann meist von einem sicheren Übertragungsmedium und stabilen, ausfallsicheren Einzelsystemen ausgegangen werden. Durch diese Anforderung wird die Komplexität der Implementierung zunächst etwas verringert. Durch den Austausch einzelner Kernkomponenten ist aber auch der Einsatz in anderen Umgebungen denkbar. Hier können dann Merkmale wie die Persistenz von Daten oder eine geschützte und manipulationsgesicherte Übertragung einen größeren Stellenwert einnehmen, wodurch mit hoher Wahrscheinlichkeit die Performance der Laufzeitumgebung sinkt.

4.1 DTM-Kernkomponente

Die DTM-Kernkomponente erzeugt und verwaltet die im *DEE* Modell beschriebenen Trails. Der Programmierer kann durch einen `parallel` Aufruf (vgl. dazu Kapitel 3.2) die Generierung eines Trailteams, welches aus mehreren Trails bestehen kann, anstoßen. Ein Trail wird dabei immer auf einen Thread abgebildet. Alle durch das Laufzeitsystem erzeugten Threads bzw. Trails eines Trailteams müssen nicht auf einem einzigen Hostsystem ausgeführt werden und können auf die zur Laufzeit verfügbaren Hostsysteme verteilt werden.

Der folgende Abschnitt bietet zunächst einen Überblick über die für den Anwender zur Verfügung stehenden Schnittstellen der DTM-Kernkomponente. Diese werden anhand einiger Beispiele näher erläutert. Im Anschluß daran werden die interne Realisierung und die Vorgänge innerhalb der DTM-Komponente beschrieben.

4.1.1 Benutzerschnittstellen des DTM

Das *Distributed Trail Model* (DTM) ist für die Verwaltung und Erzeugung von Trails zuständig. Die Benutzerschnittstellen dieser *DEE* Kernkomponente sind im *Beluga* Package `DTM.runtime` zu finden. Jedes *Beluga* Programm ist bis zum Auftauchen der ersten `parallel` Anweisung im Programmcode ein übliches singlethreaded Programm. Die von der Implementierung vorgegebenen Routinen lauten:

```
DIM.runtime.parallel(TExecuteCode runningCode) ;

DIM.runtime.parallel(TExecuteCode runningCode ,
                    int numTrails ,
                    DTMPParams params) ;
```

Listing 4.1: Varianten der `parallel` Funktion

Im Aufrufparameter `runningCode` wird der von jedem erzeugten Trail auszuführende Code übergeben. Das Objekt `runningCode` muß von der abstrakten Klasse `TExecuteCode` abgeleitet sein und die Methode `Code()` implementiert. Die optionalen Parameter (`numTrails` und `params`) können zum Abändern von Standardparametern des DTM-Laufzeitsystems benutzt werden. Zusätzliche Daten für die Trails, wie z.B. Inputdaten, müssen vor dem Aufruf von `parallel` an das `TExecuteCode`-Objekt übergeben werden. Das *Beluga* Laufzeitsystem schreibt hierfür keine spezielle Vorgehensweise vor. Der Benutzer nimmt die Implementierung einer abgeleiteten Klasse von `TExecuteCode` selbst vor. Dadurch können zur Übergabe zusätzlicher Daten an die Trails individuelle Mechanismen vorgesehen werden.

Durch den Aufruf von `parallel(...)` versucht die Laufzeitumgebung, `numTrails` Trails zu erzeugen und zu starten. Die so erzeugten Trails beginnen unmittelbar nach ihrer Erstellung mit der Abarbeitung der in `TExecuteCode.Code()` enthaltenen Anweisungen

und werden danach sofort beendet. Der Aufrufer von `parallel` gehört nicht zum generierten Trailteam. Er wird solange blockiert, bis alle Trails des Trailteams ihre Berechnung durchgeführt haben und beendet wurden. Erst danach kann er seine weiteren Berechnungen fortsetzen. Die möglichen Argumente eines `parallel` Aufrufes sind im nachfolgenden kurz aufgelistet.

int numTrails

Enthält die Anzahl der vom Nutzer angeforderten Trails.

Für die Übergabe `numTrails = -1` wird die Anzahl der Trails von der Laufzeitumgebung automatisch bestimmt. Dafür wird jeder verfügbaren CPU bzw. jedem verfügbarem CPU-Kern ein Trail zugeordnet, d.h. einem SMP System mit 4 Prozessoren werden 4 Trails zugeordnet. Der Benutzer kann diese Automatik überschreiben, indem er einen eigenen Handler (abgeleitet von `TTeamBuilderLogic`) im XML-Konfigurationsfile der Laufzeitumgebung registriert. In diesem Fall wird für die automatische Bestimmung die im Handler implementierte Logik benutzt.

boolean params.shared

Erzwingt die Benutzung des *shared* oder *private* Datenmodelles für das mittels `parallel` generierte Team. Auf diese Datenmodelle wird in Abschnitt 4.1.4 näher eingegangen.

int params.hosts

Gibt die Anzahl der Hostsysteme an, die zur Ausführung des Trailteams benutzt werden sollen. Der Anwender hat keinen direkten Einfluß auf die Wahl dieser Hostsysteme (siehe `numTrails`).

Die Verteilung der Trails ist im *dynamischen* Ausführungsmodell (vgl. dazu *Kapitel 3.2.1 und Kapitel 4.1.2*) abhängig von den verfügbaren Hostsystemen. Es wird aber durch die Standardlogik der DTM-Kernkomponente (`TSimpleTeamBuilder`) versucht, den Trail mit der Nummer 0 auf dem Hostsystem des Aufrufers zu starten. Eine Änderung dieser Strategie kann durch das Implementieren der Schnittstelle `TTeamBuilderLogic` aus dem Package `dee.domp.team` und der Registrierung in der *Beluga* Konfiguration erfolgen (siehe *Kapitel 4.4*). Beim *statischen* Ausführungsmodell ist die Verteilung der Trails fest vorgegeben.

In Listing 4.2 ist ein kurzes *Hello World* Programm aufgelistet. Jeder Trail ermittelt dort aktuelle Laufzeitparameter und gibt diese auf der Konsole aus. Da die Trails eines *DEE* Programmes auf verschiedenen Hostsystemen ausgeführt werden, existieren globale und lokale IDs (*rank*) für einen Trail. Die vom Laufzeitsystem vergebenen IDs sind innerhalb eines Trailteams eindeutig. Die Vergabe der Werte startet bei 0 und ist fortlaufend. In einem Trailteam mit vier Trails werden deshalb die globalen IDs 0, 1, 2 und 3 vergeben. Bei den lokalen IDs wird analog verfahren, die genaue Belegung richtet sich nach der

Anzahl der an der Ausführung des Trailteams beteiligten Hostsysteme. Die Anzahl dieser ausführenden Hostsysteme läßt sich ebenfalls durch einen Funktionsaufruf im Laufzeitsystem ermitteln (Zeile 9). Die Zeilen 27 bis 32 zeigen das Hauptprogramm. Dort wird in der Zeile 29 die Laufzeitumgebung gestartet, d.h. alle notwendigen Initialisierungen werden durchgeführt und die erforderlichen Komponenten des Laufzeitsystemes¹ werden gestartet. Danach wird eine parallele Ausführung von `HelloWorld` mit der von der Laufzeitumgebung automatisch bestimmten Anzahl von Trails angestoßen. Die Kontrolle kehrt erst nach der vollständigen Abarbeitung und Beendigung aller abgesetzter Trails an den Aufrufer im Hauptprogramm zurück. In der Zeile 30 wird die Laufzeitumgebung heruntergefahren und das Programm beendet.

```

1 public class HelloWorld extends TExecuteCode
2 {
3     public void Code()
4     {           // Trail number
5         int rank = DIM.runtime.getRank() ;
6             // number of Trails
7         int members = DIM.runtime.getTeamSize() ;
8             // number of Hosts
9         int hosts = DIM.runtime.getHostCount() ;
10
11             // local Trail number
12         int localRank = DIM.runtime.getLocalRank() ;
13             // number of local Trails
14         int localMembers = DIM.runtime.getLocalTeamCount() ;
15             // Host id
16         int hostID =
17             TComponentFactory.settings.getMyName().getHostTag() ;
18
19         System.out.println("Hello World from TRAIL "+rank
20                             +" of " +members) ;
21         if (localRank == 0)
22             System.out.println("There are " +localMembers
23                                 +" local Trails and " +hosts
24                                 +" Hosts, i'm running @" +hostID) ;
25     }
26
27     public static void main(String args [])
28     {
29         Beluga.init(); // Laufzeitumgebung initialisieren
30         DIM.runtime.parallel( new HelloWorld() ) ;
31         Beluga.exit(); // Laufzeitumgebung herunterfahren
32     }
33 }

```

Listing 4.2: *Beluga Hello World* Programm mit einigen Statusausgaben

¹z.B. der Kommunikationskomponente

Kommunikation durch Nachrichtenaustausch

Die DTM-Kernkomponente stellt für den Nachrichtenaustausch zwischen den Trails mehrere Kommunikationsoperationen bereit. Die Adressierung der Nachrichten erfolgt dabei über die eindeutigen globalen IDs der Trails.

void send(int receiverID, Object msg)

Sendet die Nachricht `msg` an den Trail mit der globalen ID `receiverID`. Wird dieser Trail auf dem Hostsystem des Senders ausgeführt, erfolgt keine Netzwerkoperation. Stattdessen wird die Nachricht direkt in den internen Empfangspuffer des Empfängers gestellt. Falls der Empfängertrail auf einem anderen Hostsystem ausgeführt wird, ist eine Netzwerkoperation notwendig.

Die Kontrolle kehrt an der Aufrufer der `send` Funktion zurück, sobald die Nachricht an die Kommunikationskomponente der *Beluga* Laufzeitumgebung übergeben wurde. Eventuell auftretende Wartezeiten beim Sendevorgang können somit durch Berechnungen des sendenden Trails überdeckt werden.

Object receive(int senderID)

Blockiert den Aufrufer bis zum Eintreffen einer Nachricht vom Trail mit der globalen ID `senderID`.

void broadcast(Object msg)

Schickt an alle Trails im Team die Nachricht `msg`. Für den Empfang der Nachricht muß auf der Empfängerseite ein `receive` durchgeführt werden.

In Listing 4.3 ist ein kurzer Ausschnitt eines Trailprogrammes zu sehen, welches die Verwendung dieser gerade vorgestellten Kommunikationsoperationen zeigt. Der Mastertrail sendet dort einen Integerwert (7) an alle Trails des Teams (Zeile 7). Diese empfangen diese Nachricht und speichern den Wert in der Variablen `data` (Zeile 10).

```

1  public final void Code()
2  {
3      int rank = DIM.runtime.getRank() ;
4      ...
5      // Mastertrail verteilt Daten
6      if (rank == 0)
7          DIM.runtime.broadcast( 7 ) ;
8
9      else // alle Trails empfangen vom Mastertrail
10         Integer data = (Integer) DIM.runtime.receive( 0 ) ;
11         ...
12     }

```

Listing 4.3: Beispiel Quelltext zur Dokumentation der DTM-Kommunikationsoperationen.

Leistungsindex

In heterogenen Ausführungsumgebungen kann davon ausgegangen werden, daß bei einer gleichmäßigen Aufteilung der Berechnungen auf die vorhandenen Trails die Ausführungszeiten der einzelnen Trails voneinander abweichen. Es gibt in solchen Fällen Trails, welche Teilberechnungen viel früher als andere Trails des Teams abschließen können und eventuell relativ lange Wartezyklen besitzen. Für eine Optimierung der Ausführungszeiten, d.h. für eine verbesserte Lastbalancierung, müssen solche Wartezyklen auf ein Minimum reduziert werden.

Zur Unterstützung des Programmierers bei dieser Aufgabenstellung bietet die *Beluga* Laufzeitumgebung einen sogenannten *Leistungsindex* an. Hierfür wird für jedes Hostsystem ein relativer Leistungswert L_w durch den *System-Core*² berechnet. Dieser Wert kann in ähnlicher Form wie eine MFlop-Angabe interpretiert werden. Aus den einzelnen L_w -Werten aller zur Laufzeit an der *Beluga* Umgebung angemeldeten Hostsysteme und der Verteilung der Trails auf diese, wird für jeden Trail der Anteil (*quota*) an den Gesamtberechnungen bestimmt. Der Programmierer kann die so ermittelten Werte durch die nachfolgend beschriebenen Funktionen erfragen.

```
double DTM.runtime.getPerformance()
```

Liefert den Leistungswert L_w^{pt} des Hostsystems p , auf welchem der aufrufende Trail t ausgeführt wird.

```
double DTM.runtime.getQuota()
```

Liefert einen relativen Wert q (mit $0 \leq q \leq 1$) für den aufrufenden Trail t , welcher die relative Performance dieses Trails im Vergleich zu allen anderen Trails angibt. Der Wert ergibt sich aus dem Verhältnis des absoluten L_w^{pt} Wertes des Hostsystems p und der summierten Gesamtleistung aller Hostsysteme auf denen das Trailteam (mit $n + 1$ Trails) des aufrufenden Trails t ausgeführt wird.

$$q = \frac{L_w^{pt}}{L_w^{p_0} + L_w^{p_1} + \dots + L_w^{p_n}}$$

```
double DTM.runtime.getQuota(trailNum)
```

Liefert den Rechenanteil q für den Trail mit der globalen Trailnummer `trail-Num`.

4.1.2 Ausführungsmodell

Das *Beluga* Laufzeitsystem unterstützt ein *statisches* und ein *dynamisches* Ausführungsmodell (siehe Kapitel 3.2.1). In beiden Modellen ist die Anzahl der zu startenden Trails bei der Generierung des Trailteams bekannt bzw. wird durch die Laufzeitumgebung be-

² siehe Kapitel 4.3.1

stimmt. Die Anzahl der Trails in diesem Trailteam bleibt dann während der gesamten Ausführung konstant und ändert sich nicht mehr. Auf die Bereitstellung eines *grid* Ausführungsmodelles wurde verzichtet.

Durch den in *DEE* definierten und in der *Beluga* Laufzeitumgebung umgesetzten `parallel` Aufruf werden mehrere Trails erzeugt und der aufrufende Kontrollfluß wird solange blockiert, bis alle von diesem Aufruf erzeugten Trails wieder beendet sind. Die Vorgänge beim Erzeugen und Starten eines Trailteams werden unter dem Begriff *Startsequenz* im anschließenden Abschnitt erläutert.

Startsequenz

Zentraler Bestandteil eines *Beluga* Programmes ist die Klasse `TExecuteCode`. Die in dieser Klasse definierte abstrakte Methode `Code()` muß vom Programmierer mit den für die Trails vorgesehenen Anweisungen überschrieben werden. Die Laufzeitumgebung erwartet eine Instanz dieser Klasse als Parameter des `parallel` Aufrufes und führt die in der Methode `Code()` enthaltenen Anweisungen mit möglichst vielen Trails parallel und verteilt aus. Der hierfür notwendig Initialisierungsprozeß für eine verteilte Ausführung setzt sich aus drei Phasen zusammen (Abbildung 4.1).

Zunächst wird in der **1.Phase** eine *Team Request* Nachricht vom Initiator an alle anderen Hostsysteme gesendet. In dieser Nachricht sind bereits einige Ausführungsparameter wie die Anzahl der gewünschten Trails enthalten. Jeder Host entscheidet anhand der empfangenen Informationen und lokaler Gegebenheiten³, ob er sich beteiligen möchte. Ist dies der Fall, sendet er in der **2.Phase** ein *Team Reply* mit verschiedenen aktuellen Hostinformationen an den Initiator zurück. Nach Ablauf einer bestimmten Zeitspanne oder wenn genügend Hostsysteme eine Antwort geschickt haben, beginnt die **3.Phase**. Der Initiator wertet dabei die empfangenen Hostinformationen aus und kann anhand dieser Daten eine Zuordnung von Trails auf die ausführungsbereiten Hosts vornehmen. Im Standardfall wird so jedem gemeldeten Prozessorkern ein Trail zugeordnet. Für den Fall, daß nicht genügend Hostsysteme bzw. Prozessorkerne gefunden werden können, werden die Trails auf die verfügbaren Hostsysteme gleichmäßig aufgeteilt. In diesen Fällen ist es dann möglich, daß mehrere Trails auf einem Prozessor(-kern) ausgeführt werden. Ein anderes Vorgehen bei der Zuordnung von Trails auf die Prozessorkerne und Hostsysteme ist möglich, da diese Funktionalität ebenfalls als Modul implementiert ist. Der Programmierer kann durch Erstellung einer abgeleiteten Klasse von `TTeamBuilderLogic` und Registrierung im System eine eigene Verteilungslogik definieren.

Nachdem für jeden Host die Anzahl der Trails berechnet wurde, werden *Team Init* Nachrichten (*Startnachricht*) mit allen notwendigen Informationen und einer Instanz des ursprünglich übergebenen `TExecuteCode` Objektes verschickt. Unmittelbar nach dem Erhalt solcher Nachrichten generieren die Hosts lokale Threads und starten diese. Jeder der lokal gestarteten Threads führt genau einen Trail aus. Die Threads sind von der *Beluga* Klasse `TdtmThread` abgeleitet und basieren direkt auf der in der Programmiersprache Ja-

³z.B. aktuelle Systemauslastung

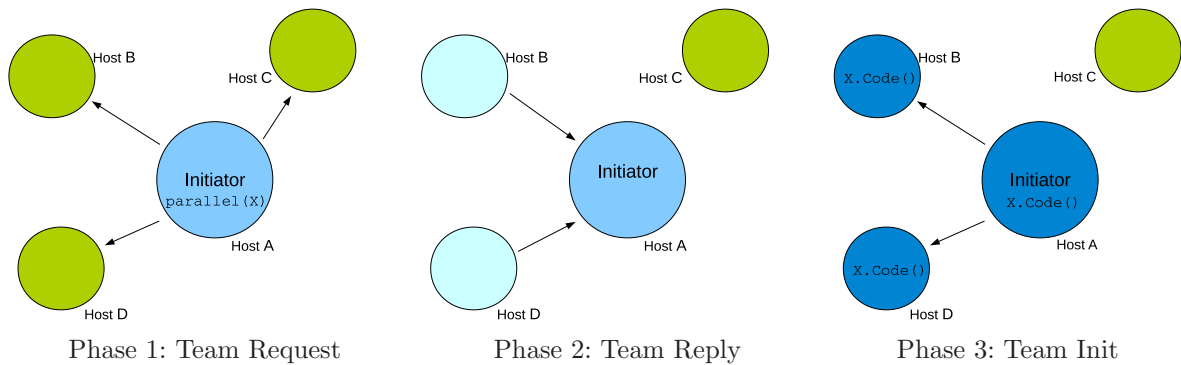


Abbildung 4.1: Die 3 Phasen zur Generierung eines Trailteams.

va vorhandenen `Thread`-Klasse. Die Einführung von `TdtmThread` wurde notwendig, um z.B. den Threads eindeutige Nummern zuzuweisen.

Startnachricht

Die Startnachricht (Abbildung 4.1) wird in der Phase 3 der Trailteamerzeugung (Team Init) generiert und an alle beteiligten Hostsysteme gesendet. Diese Nachricht ist vom Typ `TTeamInitMessage` und enthält alle relevanten Informationen für die Erzeugung von Trails auf dem Zielsystem.

```
public class TTeamInitMessage implements java.io.Serializable
{
    public TdtmTeamInfo teamInfo ;
    public TExecuteCode runObj ;
}
```

Listing 4.4: Die Startnachricht ist ein Objekt vom Typ `TTeamInitMessage`

Außer einem `TExecuteCode` Objekt, mit den parallel auszuführenden Anweisungen, ist in der Startnachricht noch ein Objekt vom Typ `TdtmTeamInfo` mit Status- und Verwaltungsinformationen enthalten. Das Listing 4.5 enthält einen Ausschnitt der Java-Deklaration dieser Klasse. Neben den Angaben zur Anzahl der Trails auf den Hostsystemen (`teamGlobalMembers`, `teamLocalMembers`) enthält die Klasse `TdtmTeamInfo` noch einige Informationen für die Zuordnung einzelner Trails auf die jeweiligen Hostsysteme (`threadMap[]` und `teamHostNames`). Diese Angaben werden z.B. bei einer Kommunikationsanweisung wie:

```
DIM.runtime.send(2, msg) ;
```

für die Auflösung von Hostnamen benötigt. In dem hier vorliegenden Fall soll eine Nachricht `msg` an den Trail mit der *Trailnummer* 2 im Trailteam gesendet werden. Die DTM-Kernkomponente kann anhand der im `TdtmTeamInfo` Objekt vorliegenden Informationen das Hostsystem identifizieren, auf welchem der Trail 2 ausgeführt wird und eine Nachricht an dieses senden.

```

public class TdtmTeamInfo implements java.io.Serializable
{
    /** eindeutige ID des Teams */
    private int teamID ;

    /** Anzahl aller Trails im Team */
    private int teamGlobalMembers ;

    /** Anzahl der Trails auf dem Zielsystem */
    private int teamLocalMembers ;

    /** alle beteiligten Hostsysteme */
    private THostNamesSet teamHostNames ;

    /** Zuordnung jedes Trails auf einen Host */
    private int threadMap [] ;

    /** summierte Leistungswerte aller Hostsysteme */
    private long maxTeamPerformance ;

    /** shared oder private Modell */
    private boolean sharedDataModel = true ;

    ..
}

```

Listing 4.5: Ausschnitt aus der Klasse TdtmTeamInfo

4.1.3 Verteilte Synchronisation

Bei der Programmierung mit den von der DTM-Kernkomponente bereitgestellten Trails und der Beschränkung auf ein lokales Hostsystem kann der Programmierer für verschiedene Synchronisationsaufgaben die in der Java-Programmierungsumgebung enthaltenen Funktionen nutzen (*synchronized*, *Concurrent Package*, siehe Kapitel 2.4.1 (S.47)). Daneben bietet die *Beluga* Laufzeitumgebung eigene Klassen für die lokale Synchronisation⁴ mit Hilfe von *Mutex*-Variablen, *Semaphore*, *Locks* und *Barriers*. Diese lokalen Implementierungen orientieren sich im wesentlichen an den in [109] enthaltenen Beispielen und sollen deshalb nicht weiter besprochen werden.

Das *Beluga* Laufzeitsystem erlaubt die verteilte Ausführung von Trails auf verschiedenen Hostsystemen. Für diese Anwendungsfälle sind die oben genannten Lösungen nicht einsetzbar. Die in diesem Abschnitt beschriebene DTM-Kernkomponente stellt deshalb einen verteilten Barrier-Mechanismus bereit, damit der Programmierer die ausgeführten

⁴gemeint ist hier die Beschränkung der Funktionalität auf ein einzelnes Hostsystem

Trails an bestimmten Punkten im Programmablauf synchronisieren kann. Da die DTM-Kernkomponente keinen gemeinsamen Speicher (DSM-Subsystem) voraussetzt, existieren im DTM keine weiteren Funktionen, welche z.B. die Sicherstellung des gegenseitigen Ausschlusses beim Zugriff auf gemeinsame Daten betreffen. Eine mögliche Lösung für Problemstellungen dieser Art ist als *verteilter Lock* in der DSM-Kernkomponente implementiert (siehe 4.2.3).

Verteilte Barriers

Dieser Abschnitt beschreibt die Funktionsweise der in der *Beluga* DTM Kernkomponente implementierten *verteilten Barrier* (siehe Kapitel 2.3.3).

Eine einfache Variante einer Barrier für den verteilten Fall ließe sich durch die Nutzung des DSM-Subsystems und der Verwendung von Zählern im DSM erstellen. Dieses Vorgehen verstößt aber gegen das in Kapitel 3 (S.57) beschriebene Prinzip der vertikalen Zugriffe zwischen den Kernkomponenten, insbesondere sollen Module einer Ebene soweit wie irgend möglich unabhängig voneinander sein (siehe Abbildung 3.1).

Deshalb verwendet *Beluga* eine einfache Kombination aus lokalen Barriers und verteilter Synchronisation. Um die diversen Probleme, die für eine korrekte Arbeitsweise der *Barrier* zu lösen sind [14, 49], etwas einzugrenzen, wird der Funktionsumfang etwas beschränkt und eine Parametrisierung der Barrierbedingung (Angabe einer Anzahl von Trails durch den Benutzer) nicht unterstützt. Die implementierte Barrier der DTM gilt immer für das gesamte Team und alle Trails dieses Teams müssen die Barrier erreicht haben, ehe weitere Anweisungen ausgeführt werden.

Das Listing 4.6 skizziert das prinzipielle Vorgehen der implementierten verteilten Barrier.

```

1 // warten bis alle Trails dieses Hostsystem hier sind
2 localBarrier( localTrails ) ;
3
4 if (localRank == 0)
5 {
6     // Nachricht, daß das Hostsystem bereit ist, senden
7     broadcast_host_ready() ;
8
9     // von jedem Host eine Bereitschaftsmeldung empfangen
10    for (t = 0 ; t < numHosts ; t++)
11    {
12        receive_host_ready( t ) ;
13    }
14 }
15
16 localBarrier( localTrails ) ;

```

Listing 4.6: Pseudocode einer verteilten Barrier

In Listing 4.6 wird zunächst auf das Eintreffen aller lokalen Trails am Synchronisationspunkt `localBarrier` in Zeile 1 gewartet. Danach wird **eine** Bereitschaftsmeldung an alle beteiligten Hostsysteme gesendet und von allen Hostsystemen, die Trails dieses Teams

ausführen, eine Nachricht empfangen. Für die Durchführung dieser Kommunikation wird immer ein lokaler Trail ausgewählt (Zeile 3), alle anderen Trails (dieses Hostsystemes) warten in einer erneuten lokalen Barrier (Zeile 14) auf das Eintreffen aller Nachrichten.

Die verteilte Barrier der *Beluga* DTM-Kernkomponente kann über einen Aufruf von `DTM.runtime.barrier()` durch die Trails verwendet werden. Jeder Trail muß einen solchen Aufruf absetzen und wird hierdurch solange blockiert, bis alle Trails des Trailteams die *Barrier* auf diese Weise betreten haben.

4.1.4 Speicher- und Datenmodell

Der für einen Trail verfügbare Speicher (Speichermodell) läßt sich in *privaten*, *lokalen* und *globalen* Speicher unterteilen⁵. Für den Zugriff auf den *lokalen* Speicherbereich enthält die *Beluga* DTM-Komponente einen einfachen Mechanismus zur Ablage von Objekten. Dieser ist unabhängig von der verwendeten DSM-Kernkomponente und bietet einen vereinfachten Zugriff auf den Speicher eines einzelnen Knotens. Folgende Methoden stehen zur Verfügung.

`Object DTM.runtime.getLocalData(String name)`

Ein im *lokalen* Speicher (des Hostsystems) abgelegtes Datum lesen. Die Daten werden durch den angegebenen Parameter `name` eindeutig referenziert.

`DTM.runtime.putLocalData(String name, Object data)`

Die Daten `data` im lokalen Speicher unter dem Namen `name` ablegen.

`DTM.runtime.clearLocalData()`

Den gesamten lokalen Speicherbereich löschen.

Die *Beluga* Laufzeitumgebung bietet zusätzlich zum Speichermodell noch einen weiteren Unterstützungsmechanismus, um Daten zwischen Trails eines Knotens austauschen zu können. Dieser wird als **Datenmodell** bezeichnet und soll im folgenden beschrieben werden. Das Datenmodell ist unabhängig von dem in der *DEE* Spezifikation beschriebenen Speichermodell (*private*, *lokal*, *global*) und ist eine Besonderheit der *Beluga* Laufzeitumgebung. Andere Implementierungen der *DEE* müssen dieses Datenmodell nicht unterstützen.

shared und private Datenmodelle

Beim Starten eines Trailteams mit `parallel` kann durch die Angabe von zusätzlichen Parametern ein von der Laufzeitumgebung zu benutzendes Datenmodell für die Trails

⁵siehe hierzu auch Kapitel 3.2.2 (S.64)

vorgeben werden. Zur Auswahl stehen ein *shared* und ein *private* Modell. Die Unterschiede beider Modelle sollen anhand eines kurzen Beispiels (Listing 4.7) veranschaulicht werden.

```

1 public class DataModelTest extends TExecuteCode
2 {
3     /** Hilfsklasse zur Aufnahme von Statusinformationen */
4     public TDMcontainer data = new TDMcontainer() ;
5
6     public class TDMcontainer extends Sheltered
7     {
8         public int i = 0;
9
10        public String toString()
11        {
12            return "value=" + i + " id=" + hashCode() ;
13        }
14    }
15
16    public void Code()
17    {
18        int rank = DIM.runtime.getRank();
19
20        data.i += rank;
21
22        DIM.runtime.barrier();
23
24        System.out.println("rank <" + rank + "> " + data);
25    }
26 }

```

Listing 4.7: Testprogramm zur Demonstration der Unterschiede von *shared* und *private* Datenmodell

In Listing 4.7 manipuliert jeder Trail unmittelbar nach dem Starten das Feld `data` (Zeile 20). Danach wird mittels einer Barrier auf die Beendigung aller Trails gewartet, um noch eine abschließende Ausgabe des Ergebnisses vornehmen zu können. Die Ausgaben der einzelnen Trails (Zeile 24) unterscheiden sich je nach verwendetem DTM-Datenmodell.

shared Datenmodell Bei der Erzeugung von Trails wird deren Codebasis⁶ in einem Objekt vom Typ `TExecuteCode` übergeben. Im vorliegenden Beispiel könnte ein Aufruf so aussehen:

```
DIM.runtime.parallel( new DataModelTest() );
```

um eine beliebige Anzahl von Trails zu starten. Da das *shared* Modell als Standard durch die DTM benutzt wird, brauchen keine weiteren Parameter (`DTMPParams`) gesetzt werden.

⁶die auszuführende Folge von Anweisungen

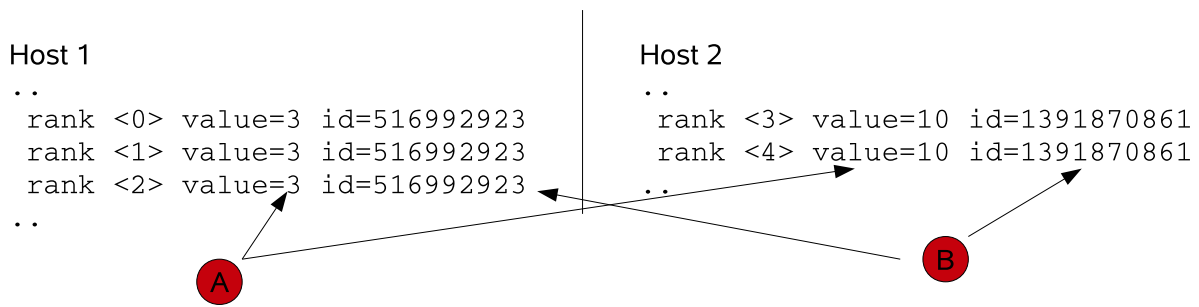


Abbildung 4.2: Host 1 mit 3 Trails und Host 2 mit 2 Trails im *shared* Datenmodell

Im *shared* Modell wird die übergebene Instanz von `DataModelTest` (Codebasis) durch das Laufzeitsystem nicht kopiert (*call by reference*). Alle Trails eines Hostsystemes arbeiten somit auf der gleichen Kopie, was in manchen Situationen die Programmierung der Trails vereinfachen kann.

Eine Beispielausgabe des Programmes, das auf einer Beluga Laufzeitkonfiguration mit 5 Trails auf zwei Hostsystemen ausgeführt wurde, ist in Abbildung 4.2 dargestellt. Die Ausgabe von `id` (mit B gekennzeichnet) zeigt, daß jeweils nur eine Instanz von `data` pro Hostsystem existiert und dort der Inhalt des Zählers `i` (mit A gekennzeichnet) für die Trails identisch ist. Bei genauerer Analyse fällt noch auf, daß die Werte auf Host 1 (`value=10`) vom erwarteten⁷ Ergebnis (`value=7`) abweichen. Der Grund hierfür liegt in einer unterschiedlich schnellen Abarbeitung der Trails. Die Ausführung wird auf Host 1 angestoßen, wodurch die lokalen Trails (`rank 0`, `rank 1`, `rank 2`,) unmittelbar gestartet werden (vgl. Abschnitt 4.1.2). Für Host 2 muß erst noch eine Startnachricht generiert und zum Versand vorbereitet werden. Diese Zeit reicht offenbar den lokalen Trails, um die Manipulation der Daten abzuschließen. Danach erfolgt das Verschicken der Startnachricht an Host 2 mit den bereits veränderten Werten. Durch Einfügen einer `Barrier` am Start der Methode `Code()` (Zeile 17) kann dieses Verhalten unterbunden werden.

private Datenmodell Das *private* Datenmodell kann von der Laufzeitumgebung nur unter bestimmten Voraussetzungen angeboten werden (*s.u.*). Sind diese erfüllt, dann kopiert die Laufzeitumgebung alle Datenfelder der übergebenen Instanz von `TExecuteCode` und stellt somit jedem Trail eine eigene Instanz (*Clone*) zur Verfügung (*call by value*). Da das Speichermodell nicht als Standard vom DTM verwendet wird, muß es explizit durch einen Parameter im `parallel` Aufruf angefordert werden.

```
DTM.runtime.parallel( new DataModelTest(),
                    -1,
                    new DTMPParams(-1, false));
```

⁷Man könnte hierbei von der Annahme ausgehen, daß das Objekt `data` nach dem Erzeugen den Wert `data.i = 0` besitzt $\rightarrow 7 = 3 + 4$.

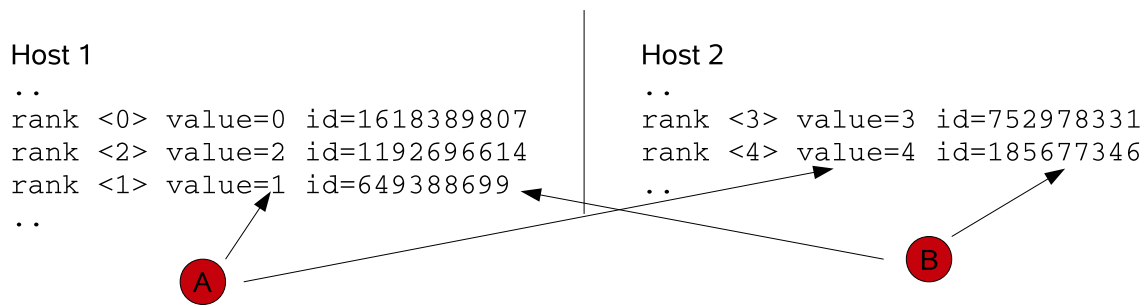


Abbildung 4.3: Host 1 mit 3 Trails und Host 2 mit 2 Trails im *private* Datenmodell

Die mit B gekennzeichnete Ausgabe in Abbildung 4.3 zeigt, daß jeder Trail eine andere Instanz der Objektvariable *data* zugeteilt bekommen hat und der Inhalt immer mit der ID des Trails übereinstimmt.

Die beschriebene Funktionalität des *private* Datenmodelles kann nur unter den folgenden Voraussetzungen durch die DTM garantiert werden. Bei der internen Realisierung wird das Java *Reflection API*⁸ benutzt, um ein beliebiges Objekt zur Laufzeit vervielfältigen zu können. Hierdurch ist es möglich Methoden und Felder einer, zum Zeitpunkt der Entwicklung unbekanntem Klasse, zur Laufzeit zu bestimmen und zu manipulieren. Allerdings ist dies nur für als *public* definierte Bezeichner problemlos möglich. Für mit *private* oder *protected* versehene Definitionen ist dieser Vorgang nicht durchführbar, da diese für das *Reflection API* nicht sichtbar sind. Aus diesem Grund können nur die *public* Datenfelder einer Klasse kopiert werden. In Listing 4.7 würde ein Abändern der Deklarationen in den Zeilen 3 und 16 von *public* auf *private* die Erstellung von Kopien der Objektvariablen *data* verhindern. Das *private* Datenmodell kann dann für diesen Fall nicht umgesetzt werden. Aufgrund dieser Faktoren wurde das Modell nicht zum Standarddatenmodell der DTM gewählt.

Der nun folgende Abschnitt wird auf den im *DEE* Programmiermodell beschriebenen globalen Speicher für die Trails eingehen und die Umsetzung in der *Beluga* Laufzeitumgebung beschreiben.

4.2 DSM-Kernkomponente

Das *Beluga* Laufzeitsystem enthält eine DSM-Kernkomponente, welche für einen transparenten Zugriff auf die gemeinsam von den Trails verwendeten Daten genutzt werden kann. Durch das DSM-Subsystem wird dabei ein globaler gemeinsamer Datenspeicher (*globaler MM*) zur Verfügung gestellt.

⁸<http://java.sun.com/docs/books/tutorial/reflect/index.html>

In der Abbildung 4.4 ist ein vereinfachtes DSM-Modell dargestellt, welches von einer *DEE* Umgebung bereitgestellt werden kann (DSM-Kernkomponente, *siehe Kapitel 3.3*) und in der hier beschriebenen *Beluga* Laufzeitumgebung umgesetzt ist. Die für den Anwender vorhandenen Schnittstellen des *Beluga* DSM-Systems und deren Benutzung werden im nachfolgenden Abschnitt beschrieben. Im Anschluß daran erfolgt, beginnend mit der Beschreibung des *DSM-Server-Level*, eine Betrachtung der einzelnen Implementierungen der im DSM-Subsystem vorhandenen Schichten und Module.

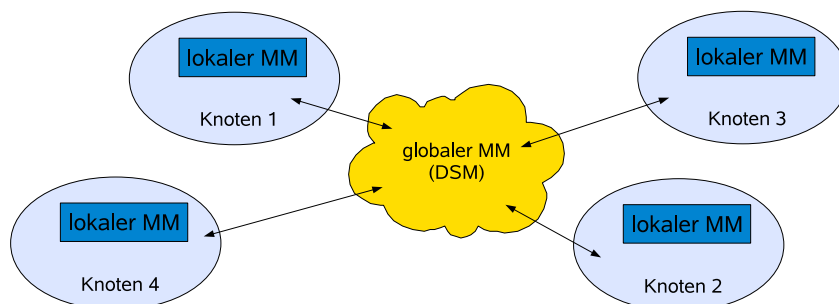


Abbildung 4.4: Vereinfachte Darstellung des *DEE* Speichermodells in einer Laufzeitkonfiguration mit vier Hostsystemen (Knoten). Der lokale Datenspeicher ist nur innerhalb eines Knotens von allen lokalen Trails zugreifbar.

4.2.1 Benutzerschnittstellen des DSM

Im Kapitel 3.3 wurden das DSM-Modell einer *DEE* Umgebung und die Aufgaben der darin enthaltenen Schichten beschrieben. Die zentralen Benutzerfunktionen des in der *Beluga* Prototypimplementierung zur Verfügung stehenden DSM, sind in der Klasse `beluga.DSM` zu finden. Bevor auf die Methoden dieser Anwenderklasse näher eingegangen wird, erfolgt noch eine kurze Funktionsbeschreibung der Klassen bzw. Schnittstellen `DSMCell`, `Data-Type` und `TCoherenceInterface`, da diese in den nachfolgenden Ausführungen häufig verwendet werden. Detailliertere Beschreibungen hierzu finden sich im weiteren Verlauf dieses Abschnittes.

DSMCell Ein Objekt vom Typ `DSMCell` wird als Referenz auf eine Zelle im DSM benutzt. Die Inhalte der referenzierten DSM-Zelle können direkt ausgelesen oder geschrieben werden. Solche DSM-Zellen befinden sich im *Server-Level* der *DEE* Spezifikation. Verwendet der Benutzer die von `DSMCell` zur Verfügung gestellten Methoden, müssen konkurrierende Zugriffe von verschiedenen Trails auf diese Zelle selbst koordiniert werden. Nur durch die Verwendung der Mechanismen im höher angesiedelten *Protocol-Level*, können diese Problemstellungen durch das DSM-Subsystem automatisch berücksichtigt werden.

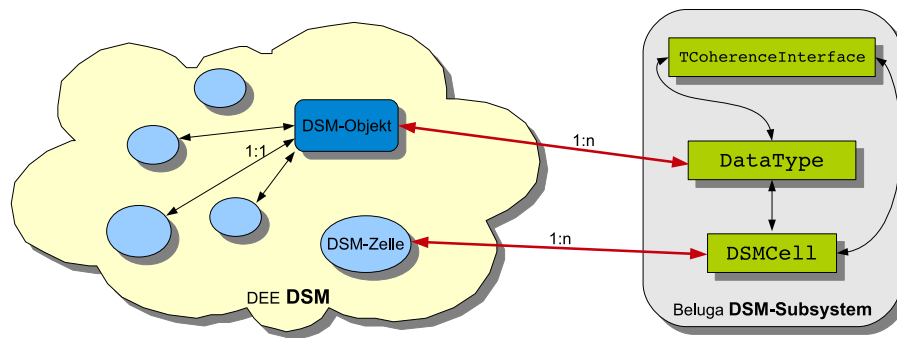


Abbildung 4.5: Gegenüberstellung einiger DSM-Funktionseinheiten aus der *DEE* Spezifikation und den in der *Beluga* Implementierung dafür vorgesehenen Klassen bzw. Schnittstellen.

DataType Für ein allgemeines DSM-Objekt, welches aus mehreren DSM-Zellen bestehen kann, wird das *Interface* `DataType` im *Beluga* Prototyp verwendet. Jedes DSM-Objekt bzw. jeder `DataType` verwaltet Daten im DSM selbstständig und kann unterschiedliche Zugriffe darauf anbieten. Dabei ist die Art der Abspeicherung im DSM (eine oder mehrere DSM-Zellen) für den Benutzer nicht sichtbar. Die `DataType` Schnittstelle im `dee.dsm.datatype` Package dient als Gerüst für die Definition neuer DSM-Objekte durch den Programmierer (*siehe auch Abschnitt 4.2.1*).

Der Typ `DataType` kann auch als Kontrollstruktur für ein DSM-Objekt im DSM interpretiert werden, da durch `DataType` die Zugriffe auf dieses DSM-Objekt koordiniert werden. In einem *Beluga* Laufzeitsystem existieren i.A. immer mehrere `DataType`-Instanzen für ein DSM-Objekt. Der Zugriff mehrerer Hostsysteme auf ein DSM-Objekt wäre sonst nicht möglich.

TCoherenceInterface Beim `TCoherenceInterface` handelt es sich um die Instanz eines Kohärenzprotokolles aus dem *Protocol-Level*. Dieses kann vom Anwender bei konkurrierenden Zugriffen verwendet werden, um inkonsistente Daten im DSM oder auf den einzelnen Knoten des Laufzeitsystems zu vermeiden (*siehe Kapitel 3.3.2 und Kapitel 4.2.1 (S.96)*).

Die *Beluga* Schnittstellen `DSMCell` und `DataType` dienen jeweils als Referenz- bzw. Kontrollstruktur auf die abstrakten DSM-Elemente `DSM-Zelle` und `DSM-Objekt` der *DEE* DSM-Spezifikation (Abbildung 4.5). Eine genauere Unterscheidung zwischen diesen Elementen wird an einigen Stellen der nachfolgenden Ausführungen notwendig, da von jeder `DSM-Zelle` und von jedem `DSM-Objekt` nur genau eine Instanz im DSM existiert. Von den *Beluga* Varianten `DSMCell` und `DataType` können jedoch mehrere Instanzen vorhanden sein, welche jeweils eine identische `DSM-Zelle` oder ein identisches `DSM-Objekt` referenzieren.

Verwaltung des DSM und die `beluga.DSM-Klasse`

Mit den in der Klasse `DSM` aus dem Package `beluga` existierenden Routinen kann auf verschiedene Schichten des DSM-Subsystems zugegriffen werden. Dabei sind diese Methoden so definiert, daß die Benutzung unabhängig von dem zur Laufzeit im DSM-Subsystem verwendeten DSM-Server ist. Der Austausch einzelner Module ist deshalb ohne Eingriff in den Quellcode des Anwenderprogrammes möglich.

Zur Erzeugung von DSM-Zellen wird die Methode:

```
DSMCell DSM.space.create( String name )
```

angeboten. Diese Routine liefert ein Objekt vom Typ `DSMCell`, welches genau eine Zelle im DSM repräsentiert. Existiert zum Zeitpunkt des Aufrufes bereits eine Zelle mit gleichem Namen im DSM, so wird keine neue Datenzelle erzeugt, sondern die bereits existierende Zelle zurückgegeben. Durch dieses Verhalten werden Mehrfachkopien im DSM vermieden.

Nach dem Anlegen einer DSM-Zelle muß diese durch einen einmaligen Aufruf⁹ der Funktion `DSMCell.init(...)` initialisiert werden. Alle bereits abgesetzten Lese- oder Schreibzugriffe blockieren, bis ein solcher Initialisierungsvorgang abgeschlossen wurde. Nachfolgende Initialisierungsoperationen haben keinen Einfluß. In Abschnitt 4.2.2 wird auf die Vorgänge beim Erzeugen von DSM-Zellen genauer eingegangen und der Grund für dieses Verhalten erläutert.

Eine DSM-Zelle enthält immer eine Kopie der eigentlichen, im DSM gespeicherten Daten. Für jede im DSM existierende DSM-Zelle können mehrere solcher lokalen Instanzen vom Typ `DSMCell` existieren, wobei die darin enthaltenen Kopien nicht identisch sein

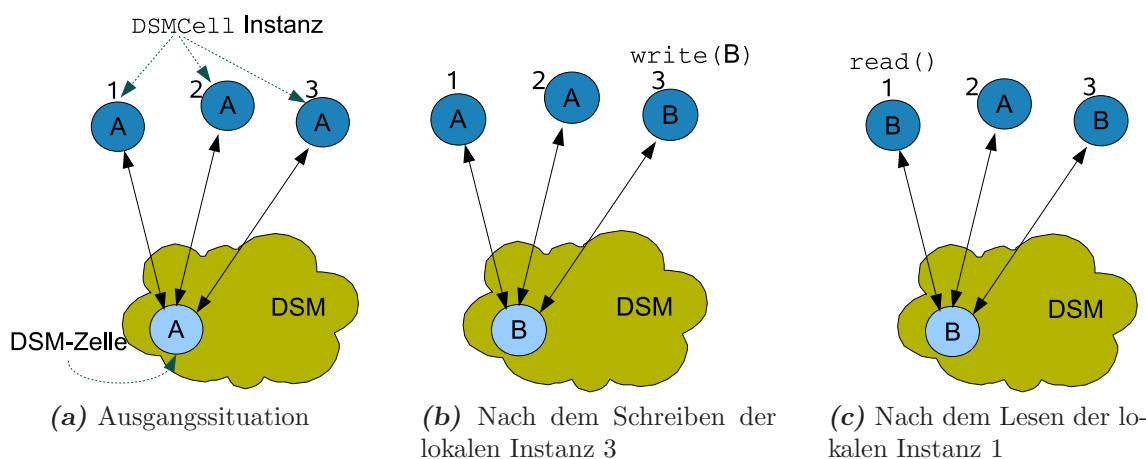


Abbildung 4.6: Wechselwirkung zwischen einer DSM-Zelle und drei lokalen Referenzen vom Typ `DSMCell` bei der Durchführung mehrerer Operationen auf die DSM-Zelle.

⁹eines beliebigen Trails

müssen. Der Programmierer muß diese Kopien durch ein explizites `read` mit den aktuellen Daten im DSM abgleichen. Deshalb enthält jede `DSMCell` immer die Daten der referenzierten DSM-Zelle vom letzten Lesevorgang. In Abbildung 4.6 ist ein mögliches Szenario für drei existierende `DSMCell` Instanzen und der Durchführung von mehreren Operationen dargestellt. Die `DSMCell` Instanzen 1 bis 3 müssen nicht auf einem einzelnen Hostsystem existieren sondern können auf verschiedene Hosts verteilt sein.

Auf die lokale Kopie bzw. auf den Inhalt einer lokalen Datenzelle kann durch `get/set` Operationen zugegriffen werden. Diese und weitere Methoden der `DSMCell`-Klasse sind hier aufgeführt.

void `init(Object data)`

Initialisiert die Datenzelle mit den übergebenen Daten `data`.

Object `read()`

Stößt einen Lesevorgang im DSM an und aktualisiert die Daten der lokalen Instanz (Referenz) mit den Daten der DSM-Zelle im DSM.

void `write()`

Schreibt die in der lokalen Instanz enthaltenen Daten in den DSM.

void `write(Object newValue)`

Füllt den Inhalt der lokalen Instanz mit den übergebenen Daten `data` und schreibt diese danach in den DSM.

Object `getValue()`

Liest die Daten der lokalen Instanz, ohne Interaktion mit dem DSM.

void `setValue(Object newValue)`

Überschreibt die Daten der lokalen Instanz der Datenzelle mit dem übergebenen Wert von `newValue`.

Die Klasse `beluga.DSM` Die Routinen der Klasse `beluga.DSM` umfassen neben dem Suchen und Erzeugen von DSM-Zellen auch die Generierung neuer Instanzen von DSM-Objekten. Weiterhin werden Methoden für den Umgang mit den verfügbaren Kohärenzprotokollen angeboten.

Für den Zugriff auf die Funktionen der Klasse `beluga.DSM` existiert eine Klassenvariable `space`, die beim Start der Laufzeitumgebung erzeugt und initialisiert wird¹⁰. Das Anlegen weiterer Instanzen dieser Klasse ist nicht möglich. Somit ist sichergestellt, daß nur

¹⁰Singleton Pattern [42]

eine einziges Kontrollobjekt für das DSM-Subsystem zur Laufzeit vorhanden ist.

DSMCell `get(String userCellName)`

Sucht nach der durch den Parameter `userCellName` bestimmten Speicherzelle im DSM. Falls diese Zelle nicht gefunden werden kann, wird ein leerer Wert, d.h. `null` geliefert.

DSMCell `create(String userCellName)`

Erzeugt eine Speicherzelle mit dem Namen `userCellName` im DSM und liefert eine lokale Kontrollinstanz als Referenz auf diese DSM-Zelle zurück. Dieser Aufruf wurde bereits am Beginn dieses Abschnittes detaillierter besprochen.

`void clear()`

Löscht alle Einträge im gesamten DSM-System.

TCoherecenceInterface `getCoherecenceProtokoll(String dataTypeName)`

Liefert das Kohärenzprotokoll zurück, welches für das über den Parameter `dataTypeName` spezifizierte DSM-Objekt registriert ist. Kann kein entsprechendes Protokoll zur Laufzeit gefunden werden, wird das als Standard definierte Protokoll zurückgegeben. Der Anwender kann alle Zugriffe auf Daten in den DSM-Zellen durch dieses Protokoll durchführen lassen.

TCoherecenceInterface `getCoherecenceProtokollbyName(String protocolName)`

Liefert das durch den Parameter `protocolName` angegebene Kohärenzprotokoll. In der im Rahmen dieser Arbeit verwendeten *Beluga* Laufzeitumgebung sind die Protokolle `simple`, `writeThrough` und `writeBack` verfügbar. Eine Beschreibung erfolgt in Kapitel 4.2.3 (*S.108*).

DataType `createInstance(String name, String dataTypeName)`

Erstellt und liefert eine Instanz des durch den Parameter `dataTypeName` angegebenen DSM-Objektes. Das DSM-Objekt wird im DSM-System unter dem Namen `name` abgelegt und kann somit von allen Trails eindeutig referenziert werden.

Ein einfaches Anwendungsbeispiel

In Listing 4.8 ist ein Beispielquelltext für ein Trailprogramm mit DSM-Nutzung dargestellt. Vom Mastertrail mit der *ID* 0 wird dort eine DSM-Zelle `testzelle` angelegt (Zeile 6) und von allen Trails im weiteren Verlauf gelesen (Zeile 16). In der Zeile 6 erfolgt das Zurückschreiben der gelesenen Daten, wobei der Wert um 1 erhöht wird. Da kein Zugriffsschutz beim Schreiben und Lesen der Datenzelle erfolgt, ist der Inhalt dieser

Datenzelle nach dem Beenden aller Trails nicht vorhersehbar. Auf die Verwendung von Kohärenzmechanismen (*Protocol-Level*) der *Beluga* Laufzeitumgebung wird im nachfolgenden Abschnitt eingegangen.

```

public void Code()
{
    if (rank == 0)
    {
        // Zelle im DSM anlegen
        DSMCell cell = DSM.space.create("testzelle") ;

        // Initialisierung
        cell.init(new Integer(0)) ;
    }

    // erst starten, wenn die Zellen initialisiert wurde
    DIM.runtime.barrier() ;

    // Zelle aus dem DSM lesen
    DSMCell cell = DSM.space.get("testzelle") ;

    // Datum aus der Zelle lesen
    int i = ((Integer) cell.read()).intValue() ;

    // erhaltenen Wert erhöhen und schreiben
    cell.write( new Integer(i+1) ) ;
}

```

Listing 4.8: Beispiel für die DSM-Benutzung

Protocol-Level: Datenkohärenz

Zur automatischen Sicherstellung der Datenkohärenz bei der Manipulation von Datenzellen durch konkurrierende Trails können verschiedene Kohärenzprotokolle verwendet werden. Jedes dieser Protokolle stellt bestimmte Eigenschaften sicher oder implementiert ein ausgewähltes Verfahren. Eine Kontrollinstanz eines solchen Protokolles erhält der Anwender durch den Aufruf einer der Methoden

```

TCoherenceInterface getCoherenceProtokollbyName(String protocolName)
oder

```

```

TCoherenceInterface getStandardProtocol()

```

aus der Klasse `DataTypeFactory`. Ein auf diesem Wege von der Laufzeitumgebung geliefertes Kohärenzprotokoll bietet die folgenden Methoden an, welche von den Trails beim konkurrierende Zugriff auf eine DSM-Zelle verwendet werden können.

TCoherenceHandle `getHandle(String cellName)`

Liefert ein *Handle* einer existierenden Zelle im DSM. Solch ein *Handle* wird von den Manipulationsoperationen (*read*, *write* ...) des Protokolles verwendet und enthält interne Verwaltungsdaten, welche für den Benutzer nicht sichtbar sind. Jeder Aufruf von `getHandle` kann eine neue Instanz eines *TCoherenceHandle* Objektes liefern.

TCoherenceHandle `getHandle(DSMCell cell)`

Liefert ebenfalls ein *Handle* für die weitere Benutzung.

TCoherenceHandle `performInit(DSMCell cell, Object data)`

Initialisiert eine Zelle `cell` im DSM mit den im Parameter `data` angegebenen Daten. Zurückgeliefert wird ein *Handle*, welches der Benutzer mit den Manipulationsoperationen des Protokolles (*read*, *write* ...) verwenden kann.

TCoherenceHandle `performInit(String cellName, Object data)`

Initialisiert die, durch den Namen `cellName` referenzierte, Zelle im DSM. Existiert noch keine DSM-Zelle mit dem angegebenen Namen, so wird eine DSM-Zelle durch diese Methode erstellt. Es wird ebenfalls ein *Handle* zurückgegeben.

Lock `getLock(TCoherenceHandle handle, int type)`

Liefert für die durch `handle` referenzierte DSM-Zelle eine *Lock*-Datenstruktur. Mit den von *Lock* bereitgestellten Funktionen `lock()` und `unlock()` kann der Anwender exklusiven Zugriff anfordern bzw. freigeben. Durch den Parameter `type` der `getLock()` Methode kann der Programmierer bestimmen, ob es sich um einen *Lese-Lock* oder einen *Schreib-Lock* handeln soll. Bei *Lese-Locks* können eventuell andere Trails ebenfalls lesend auf die durch den *Lock* geschützten Daten zugreifen. Im Falle eines *Schreib-Locks* erhält nur ein Trail die volle Kontrolle.

Object `performWrite(TCoherenceHandle handle, Object data)`

Schreibt die übergebenen Daten `data` in die DSM-Zelle. Dabei wird davon ausgegangen, daß der Benutzer die DSM-Zelle – im Gegensatz zu `performRead()` – bereits durch einen `Lock.lock()` Aufruf geschützt hat. Die Daten müssen nicht unmittelbar in den DSM geschrieben werden, sondern können vom Protokoll zurückgehalten werden und zu einem späteren Zeitpunkt in den DSM übertragen werden.

Object `performWrite(TCoherenceHandle handle)`

Überträgt die durch `handle` referenzierten, in der lokalen Kopie der Speicherzelle enthaltenen Daten, in den DSM. Dieser Vorgang kann ebenfalls durch das Protokoll auf einen späteren Zeitpunkt verschoben werden.

Object performRead(TCoherenceHandle handle)

Liest den Inhalt der durch das *Handle handle* angegebenen DSM-Zelle. Das Protokoll entscheidet, ob eine Leseanfrage an den DSM gestellt werden muß oder die Daten aus einem internen Zwischenspeicher des Protokolles gelesen werden können. Der Aufruf kann je nach verwendetem DSM-Server immer zu einer DSM-Anfrage führen, da die Aktualität der Daten u.U. nicht sichergestellt werden kann.

Object fetch(TCoherenceHandle handle)

Liefert die Daten der durch das *Handle handle* angegebenen DSM-Zelle. Hierbei sollen Leseanfragen an den DSM vermieden werden und die Daten können aus einem internen Zwischenspeicher des Protokolles gelesen werden. Falls Daten bereits lokal verfügbar sind, führt das meist zu keiner Interaktion mit dem DSM-Server und die benötigte Zeit für die Beantwortung dieser Anfrage sinkt.

void performFlush()

Erzwingt das Schreiben aller durch das Protokoll zurückgehaltenen Schreiboperationen.

void clearCache()

Leert den internen Zwischenspeicher.

Die Benutzung der soeben beschriebenen Methoden wird im nächsten Abschnitt anhand mehrerer Beispiele erläutert.

User-Level: DSM-Objekt

Um die Verwendung der vorhandenen Schnittstellen und Mechanismen des DSM zu vereinfachen und deren Komplexität soweit wie möglich zu verbergen, existiert im DSM-Subsystem das *User-Level*. Dort werden DSM-Objekte definiert, die helfen sollen, die Verwaltung mehrerer DSM-Zellen als gemeinsame Datenstruktur (**DataType**) abstrahieren zu können. Ein *DSM-Objekt* kann deshalb auch als *verteilte Datenstruktur* bezeichnet werden. Beide Begriffe werden hier als Synonym füreinander benutzt, wobei die Benutzung von *DSM-Objekt* immer einen Bezug zur *Beluga* Laufzeitumgebung herstellen soll.

Das in Listing 4.9 dargestellte Vorgehen zeigt die Verwendung eines Kohärenzprotokolles aus dem *Protocol-Level* für die Erzeugung und die nachfolgende Manipulation einer einzelnen DSM-Zelle. Es soll die Vereinfachung durch die im *User-Level* verfügbaren Mechanismen dokumentieren, indem zunächst die Programmierung mit den im *Protocol-Level* zur Verfügung stehenden Methoden aufgezeigt wird.

```

1  TCoherenceInterface manager =
2      DataTypeFactory.getStandardProtocol() ;
3  ..
4
5  // Zelle anlegen
6  manager.performInit("c_name", new Integer(5)) ;
7  ..
8
9  // Handle der Zelle im DSM holen
10 TCoherenceHandle handle = manager.getHandle("c_name") ;
11
12 // Schreibzugriff
13 Lock lock = manager.getLock(handle, Lock.LOCK_WRITE) ;
14 lock.lock() ;
15 manager.performWrite(handle, new Integer(12)) ;
16 lock.unlock() ;
17 ..

```

Listing 4.9: Benutzung des DSM-Subsystems im *Protocol-Level*

In Zeile 1 muß zunächst ein Protokoll von der Laufzeitumgebung angefordert werden. Danach kann mit Hilfe dieses Protokoll eine Zelle `c_name` erzeugt und initialisiert werden (Zeile 6). Im späteren Verlauf soll der Inhalt dieser Speicherzelle verändert werden (Zeilen 13-16). Dazu ist es erforderlich, den Zugriff mit einem Lock zu schützen; erst danach kann der Inhalt der Zelle neu geschrieben werden.

Das in Listing 4.9 beschriebene Vorgehen mit direkter Nutzung von Routinen des *Protocol-Levels* wird durch die im *User-Level* existierende Klasse `ObjectCell` abstrahiert. Das obige Beispiel läßt sich so auf das in Listing 4.10 gezeigte Vorgehen reduzieren¹¹.

```

1  ObjectCell oCell = (ObjectCell)
2      DSM.space.createInstance("c_name", ObjectCell.class) ;
3  ..
4  oCell.init(5) ;
5  ..
6  oCell.write(12) ;

```

Listing 4.10: `ObjectCell` DSM-Objekt

Bei der Klasse `ObjectCell` handelt es sich um ein DSM-Objekt für einfach strukturierte Daten im DSM. Es wird z.B. im `QueensTest`¹² für das Aufsammeln (Summieren) von berechneten Teilergebnissen verwendet. Das `ObjectCell` DSM-Objekt unterstützt keine Verteilung der Daten auf mehrere DSM-Zellen oder macht Annahmen über die Struktur der Daten. Vor dem Programmierer werden die Zugriffe auf das *Protocol-Level* weitestgehend verborgen und vereinfachen somit die Benutzung des DSM. Insbesondere sind

¹¹Es wird gleichzeitig noch das seit Java 5 vorhandene *Autounboxing* angewendet. Dabei erfolgt eine automatische Konvertierung von primitiven Datentypen (hier `int`) in die korrespondierende Wrapper-Klasse (hier `Integer`).

¹²siehe hierzu auch Kapitel 5.3.2 (S.172)

die Verwendung eines Kohärenzprotokolles und die notwendigen Funktionsaufrufe nicht sichtbar.

Durch Konfigurationseinstellungen der Laufzeitumgebung kann ein Standardprotokoll und für jedes registrierte DSM-Objekt (*User-Level*) genau ein als Standard zu verwendendes Kohärenzprotokoll definiert werden. Diese Bindung der Protokolle an die DSM-Objekte ist für die gesamte Lebensdauer eines DSM-Objektes (Instanz) fest. Dies bedeutet, daß spätestens beim Anlegen eines DSM-Objektes das Kohärenzprotokoll für diese Instanz des DSM-Objektes festgelegt werden muß. Mehrere Kohärenzprotokolle können aber gleichzeitig benutzt werden, da andere Instanzen nicht dieses Kohärenzprotokoll nutzen müssen, sondern mit einem anderen Protokoll arbeiten können. Das Ändern der Standardzuordnung von Kohärenzprotokoll und DSM-Objekt wird durch die Methode

```
setProtocolForType(String dataTypeName, String protocolName)
```

während der Laufzeit ermöglicht. Diese Methode ist in der Klasse `DataTypeFactory` des Packages `beluga.dsm.datatype` definiert. Dort sind auch noch einige weitere Funktionen für ähnliche Aufgaben vorhanden, auf die hier nicht näher eingegangen werden soll¹³.

Erzeugung von DSM-Objekten In der *Beluga* Laufzeitumgebung sind bereits die DSM-Objekte `ObjectCell`, `Feld`, `Raum` und `Octtree` vordefiniert. Der `ObjectCell` Typ wurde bereits verwendet und kurz vorgestellt. Bei den letztgenannten DSM-Objekten handelt es sich um speziell erstellte verteilte Datenstrukturen für die in den nachfolgenden Kapiteln 6, 7 und 8 durchgeführten Untersuchungen.

Zur Erzeugung eines im Laufzeitsystem registrierten DSM-Objektes wird die Methode:

```
DSM.space.createInstance( String name, String dataTypeName)
```

angeboten (*siehe Kapitel 4.2.1 (S.94)*). Zurückgegeben wird eine Objektinstanz vom Typ `DataType`, welches dem aufrufenden Trail den Zugriff auf die durch den Parameter `dataTypeName` spezifizierte verteilte Datenstruktur erlaubt. Damit es möglich ist, mehrere solcher Datenstrukturen gleichzeitig im DSM zu verwalten, wird der Parameter `name` zur eindeutigen Referenzierung des DSM-Objektes verwendet. Analog zum Verhalten von DSM-Zellen erzeugt jeder Aufruf von `createInstance` eine neue Kontrollstruktur für das DSM-Objekt.

Für die Benutzung eines DSM-Objektes wird das im nachfolgenden erläuterte *create-init-use* Vorgehen von der *Beluga* Laufzeitumgebung vorgeschrieben. Die drei Phasen dieser Vorgehensweise müssen von einer DSM-Objekt-Implementierung umgesetzt werden. Da die Initialisierung eines verteilten Feldes andere Parameter als beispielsweise die Initialisierung eines `ObjectCell` DSM-Objektes erfordert, existiert keine vorgegebene Zuordnung dieser Phasen auf feste Methodenaufrufe des DSM-Objektes. Dies bedeutet, daß z.B. die Methode `A.init(1, 2, 3)` des DSM-Objektes `A` und die Methode `B.make("x")`

¹³Beispiele hierfür sind die Registrierung von neuen Kohärenzprotokollen, von DSM-Objekten (bzw. DSM-Datentypen) oder die Festlegung des Standardprotokolles.

des DSM-Objektes B jeweils für die Initialisierungsphase eines DSM-Objektes stehen können. Die Dokumentation eines DSM-Objektes muß die Methoden für die einzelnen Phasen beschreiben und dem Anwender die Durchführung des *create-init-use* Vorgehens ermöglichen. Das Listing 4.10 zeigt ein Minimalbeispiel für die Umsetzung dieser Regelung in einem DSM-Objekt. In der nun folgenden Beschreibung der einzelnen Phasen wird auf die dort verwendeten Methoden Bezug genommen. Für andere DSM-Objekte können die zur Verfügung gestellten Methoden abweichen.

- create** Erzeugung (Listing 4.10 *Zeile 2*)
Zunächst wird eine lokale Kontrollstruktur (`oCell` vom Typ `DataType`) für das DSM-Objekt erzeugt. Ziel dieses Vorganges ist es, einen eindeutigen Zugriffspunkt im DSM zu generieren, den alle nachfolgenden Operationen verwenden können. Dabei versucht die DSM-Schicht, Initialisierungen und Zugriffe auf den DSM weitestgehend zu vermeiden. Es wird allerdings mindestens eine DSM-Zugriffoperation abgesetzt, um die anzulegende DSM-Datenstruktur eindeutig identifizieren zu können und um festzustellen, ob bereits Einträge zu diesem DSM-Objekt existieren. Hierfür wird eine einzelne DSM-Zelle benutzt, die über einen vom Benutzer angegebenen Namen ("`c_name`") angesprochen wird. Existiert diese Zelle bereits, so kann davon ausgegangen werden, daß das zu erstellende Datenobjekt (DSM-Objekt) bereits von einem anderen Trail erzeugt wurde und keine weiteren Operationen notwendig sind. Die verwendete DSM-Zelle kann später z.B. für die Aufnahme von Verwaltungsdaten oder direkt für die Datenspeicherung des DSM-Objektes benutzt werden.
- init** Initialisierung der Datenstruktur im DSM (Listing 4.10 *Zeile 4*)
Durch den im *create* Vorgang durchlaufenen Prozeß wurde die Kontrollstruktur `oCell` für das DSM-Objekt erstellt. Ehe die Datenstruktur (DSM-Objekt) und die durch dieses DSM-Objekt verwalteten Zelle(n) durch den Anwender verwendet werden können, müssen diese DSM-Zellen im DSM angelegt und initialisiert werden (siehe Kapitel 4.2.2 (*S.104*)). Dieser Vorgang wird von der *init* Phase des DSM-Objektes durchgeführt, dabei ist der genaue Initialisierungsprozeß und die Anzahl der erzeugten DSM-Zellen vom DSM-Objekt abhängig. Ein verteiltes Feld erstellt z.B. bei der Initialisierung mehrere DSM-Zellen für die Abspeicherung einzelner Teilfelder. Die Anzahl der DSM-Zellen hängt von der Feldgröße und der verwendeten *Data-Policy* ab (Feldeinträge pro DSM-Zelle).
Die Initialisierung eines DSM-Objektes darf nur einmal erfolgen, alle nachfolgenden Aufrufe (d.h. ein erneuter Initialisierungsaufruf) haben keine Auswirkungen und werden ignoriert.
- use** Verwendung (Listing 4.10 *Zeile 6*)
Nach dem Anlegen und Initialisieren kann die Datenstruktur durch den Anwender benutzt werden, d.h es können Daten gelesen oder geschrieben werden.

Erstellung benutzereigener Datentypen Um die Definition eigener Datenstrukturen für den DSM (DSM-Objekte) durch den Programmierer zu erleichtern und zu vereinheitlichen, werden zwei Java-Interface-Definitionen vorgegeben. Die Schnittstelle `DataType` dient dabei als Beschreibung des neuen Datentypes und `DataTypeAccess` für die Regelung des Zugriffes auf Teile der Daten im DSM. Hierbei wird von der Sicht des Anwenders auf die abgelegten Daten im DSM-Objekt ausgegangen. Die Verwendung des `DataTypeAccess` Interfaces muß nicht erfolgen, es dient als Vorlage für weitere Benutzerimplementierungen und wird intern nicht von der *Beluga* Laufzeitumgebung verwendet. Im Gegensatz hierzu, wird das Interface `DataType` vom DSM-Subsystem benutzt und muß verwendet werden, da die von dieser Schnittstellendefinition geforderten Methoden zum Zugriff auf die Daten im DSM dienen.

Im nachfolgenden Listing 4.11 ist ein Auszug aus der Implementierung des in den letzten Abschnitten bereits mehrfach beschriebenen DSM-Objektes `ObjectCell` zu sehen. In den Zeilen 5 bis 14 erfolgt dort die Implementierung der vom Interface `DataType` verlangten Methoden. Auf das von der Methode `getDataSegmentation()` gelieferte `DataSegmentation` Objekt wird im später Verlauf in Abschnitt 4.2.3 (*S.108*) näher eingegangen. Bei den Zeilen 17 bis 49 handelt es sich um die für ein DSM-Objekt spezifischen Methoden zur Umsetzung des für den Umgang mit diesen verteilten DSM-Datentypen geforderten *create-init-use* Zyklus. Für andere DSM-Objekte können abweichende Methoden existieren, da die genaue Methodensignatur nicht vorgegeben ist. Die Klasse `ObjectCell` verwendet für die Verwaltung und Abspeicherung von Daten nur eine DSM-Zelle, deshalb ist die Implementierung dieser Methoden sehr übersichtlich. Andere DSM-Objekte, wie z.B. ein verteiltes Feld, verwalten ihre Daten in verschiedenen DSM-Zellen und müssen deshalb erst die jeweilige DSM-Zelle lokalisieren. Dieser Vorgang soll für den Benutzer jedoch vollkommen transparent sein.

```

1 public class ObjectCell<TYPE> implements DataType
2 {
3     ...
4     /** Aufteilung aller Daten auf DSM-Zellen */
5     public DataSegmentation getDataSegmentation()
6     {
7         return segment ;
8     }
9
10    /** vom Benutzer vorgegebener Name, Variablenname */
11    public String getName()
12    {
13        return headName ;
14    }
15
16    /** einfache Initialisierung */
17    public void init(TYPE data)
18    {

```

```

19     protocol.performInit(headName, data ) ;
20 }
21
22 /** auf die Initialisierung warten */
23 public void waitForInit()
24 {
25     // ein Lesezugriff blockiert bis zur Initialisierung
26     read() ;
27 }
28
29 /** Daten lesen */
30 public TYPE read()
31 {
32     TCoherenceHandle handle = protocol.getHandle(headName) ;
33
34     // eventuell ist ein ungültiger Cache-Eintrag vorhanden
35     // deshalb Lesen aus dem DSM erzwingen
36     handle.setRefreshFlag() ;
37     return (TYPE) protocol.performRead(handle) ;
38 }
39
40 /** Daten schreiben */
41 public void write(TYPE data)
42 {
43     TCoherenceHandle handle = protocol.getHandle(headName) ;
44     Lock lock = protocol.getLock(handle, Lock.LOCK_WRITE) ;
45
46     lock.lock() ;
47     protocol.performWrite(handle, data);
48     lock.unlock();
49 }
50     ...
51 }

```

Listing 4.11: Auszug aus der Implementierung des ObjectCell DSM-Objektes

Um ein so erstelltes DSM-Objekt dem Laufzeitsystem zur Verfügung zu stellen, muß dieses zur Laufzeit durch den Aufruf von:

```

DataTypeFactory.registerDataType(
    TAbstractDataTypeProducer producer) ;

```

aus dem Package `beluga.dsm.datatype` registriert werden. Die Methode erwartet ein Objekt vom Typ `TAbstractDataTypeProducer`. Der Programmierer muß deshalb zusätzlich noch eine abgeleitete Klasse von `TAbstractDataTypeProducer` erstellen und als *Producer* mit dem oben genannten Aufruf registrieren. Das *Beluga* Laufzeitsystem verwendet dann diesen *Producer* für die Generierung entsprechender DSM-Objekte.

```
public abstract DataType produce(  
    DSMCell zelle ,  
    TCoherenceInterface coho ) ;
```

In den nachfolgenden Abschnitten werden die interne Realisierung des DSM-Subsystemes und die vorhandenen DSM-Serverimplementierungen (*Storage-Modul*) näher beschrieben.

4.2.2 DSM-Server-Level

Wie im allgemeinen Funktionsüberblick (Kapitel 3.3) dargelegt, enthält die *Server-Level* Schicht einen DSM-Server und die Implementierung von dazugehörigen DSM-Zellen als kleinste Speichereinheit. Beide Funktionseinheiten sind sehr eng miteinander verknüpft und werden mitunter als *Storage-Modul* bezeichnet. Der oder die DSM-Server des Laufzeitsystems verwalten jeweils alle oder nur eine Teilmenge der im *DSM* vorhandenen DSM-Zellen. Die genaue Verteilung der Zellen hängt hierbei von der verwendeten Implementierung des DSM-Servers ab. Der einfachste Fall ist die Verwaltung aller DSM-Zellen durch einen einzigen Server. Dies bringt allerdings den Nachteil mit sich, daß alle Anfragen an ein einziges System gerichtet werden müssen und dieses zwangsläufig zum limitierenden Faktor bezüglich der Performance der Speicheranfragen wird.

Einen anderen Aspekt, der an dieser Stelle erläutern werden soll, ist die Aufgabenabgrenzung zwischen dem Servermodul und der Kohärenzschicht. Für eine DSM-Zelle gibt es immer nur einen Speicherort, d.h. einen einzigen zuständigen DSM-Server. In einer Umgebung mit mehreren DSM-Servern ist es denkbar, daß Zellen zwischen den existierenden Servern verschoben werden. Durch diesen Vorgang ändert sich aber auch der für die verschobene DSM-Zelle zuständige DSM-Server. Die Kohärenzprotokolle können hingegen mehrere Kopien einer Zelle auf den verschiedenen Hostsystemen halten und z.B. mehrere, aufeinanderfolgende Schreiboperationen zusammenfassen. Die Serverschicht ist davon nicht betroffen und im allgemeinen ändert sich nichts an der Zugehörigkeit der Zelle zum DSM-Server.

Zellen anlegen

Bei der Erzeugung einer Zelle durch das DSM-Subsystem wird diese als *nicht initialisiert* gekennzeichnet. Alle Zugriffe darauf blockieren, bis diese Markierung durch einen separaten Initialisierungsaufwurf entfernt wird. Dieses Vorgehen ist notwendig, um einen gültigen Anfangszustand zu erhalten und die Zelle mit einem Anfangswert zu belegen. Andernfalls müßten alle Trails die Zelle mit einem identischen Wert initialisieren, da bei einem gleichzeitigen Zugriff mit unterschiedlichen Werten nicht festgelegt werden kann, welcher davon in der Zelle erhalten bleibt (Abbildung 4.7).

Exclusive Mode

Um Kohärenzprotokollen eine Möglichkeit zur Interaktion zu geben, müssen die Routinen `requestExclusive` und `leaveExclusive` durch den DSM-Server angeboten werden. Durch diese Routinen soll eine sehr einfache Zugriffskontrolle für einzelne Zellen auf

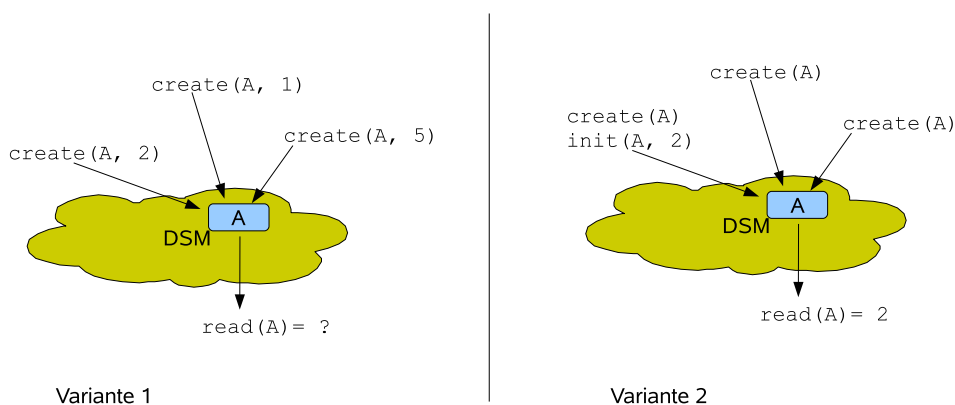


Abbildung 4.7: Veranschaulichung von Varianten für das Vorgehen beim Erzeugen von DSM-Zellen. Bei der Variante 1 kann der Inhalt der Zelle nicht vorhergesagt werden. Die Variante 2 wird von der *Beluga* Implementierung benutzt.

der Serverebene ermöglicht werden. Ein Aufruf von `requestExclusive` sequenzialisiert die Aufrufsreihenfolge auf eine Zelle, indem alle nachfolgenden Aufrufe von `requestExclusive` solange blockieren bis `leaveExclusive` aufgerufen wird. Im weitesten Sinne kann man diese Funktionalität mit dem *Mutex* Ansatz vergleichen. Allerdings wird kein Eigentümerhandlung verlangt, was wiederum bedeutet, daß `leaveExclusive` von einem beliebigen Trail aufgerufen werden kann. Die Benutzung der *Exclusive* Funktionalität hat keinerlei Einfluß auf das Verhalten der `read` oder `write` Operationen der entsprechenden Zelle. Die Bereitstellung einer erweiterten Zugriffskontrolle für den Benutzer erfolgt in den höheren Schichten des DSM-Subsystems, die Basisfunktionen des *Exclusive Mode* sind hierfür aber eine Voraussetzung.

Verfügbare Implementierungen

Im Rahmen dieser Arbeit wurden drei *Storage-Module* erstellt und verwendet. Ein solches *Storage-Modul* besteht jeweils aus einem DSM-Server und der zugehörigen Implementierung einer DSM-Zelle. Das unterstützte Konsistenzmodell ist bei allen Varianten mit dem Modell der *sequentiellen Konsistenz* identisch. Dies bedeutet, daß keine Umordnung der Reihenfolge von Speicherzugriffen eines einzelnen Trails erfolgt. Die zeitliche Reihenfolge von Zugriffen zwischen den einzelnen Trails kann allerdings variieren (siehe Kapitel 2.3.1 (S.36)).

- Central** Eine einfache Basisversion eines DSM-Servers. Die Datenzellen werden nur von einem einzigen System (zentraler Server) verwaltet.
- Spread** Die auf dem *Central*-Server basierende Implementierung, allerdings können die Zellen auf andere Systeme ausgelagert werden.
- TSpaces** Die Umsetzung eines DSM-Servers mit Hilfe der *TSpaces*-Bibliothek von IBM, welche einen Tupelraum zur Verfügung stellt [80].

Weitere Implementierungen auf Basis der *Java Temporary Caching API* (JSR 107) und *JuxMem*¹⁴ wurden kurz evaluiert, werden aber im Rahmen dieser Arbeit nicht weiter betrachtet. Die Persistenz der gespeicherten Daten ist in diesen Projekten ein ausdrückliches Ziel und nimmt deshalb einen sehr großen Stellenwert in den bereitgestellten Laufzeitumgebungen ein. Allerdings geht dieses Verhalten oftmals zu Lasten der Performance und ist für die hier beschriebene Aufgabenstellung eher ungeeignet. Im weiteren Verlauf werden nun die für die *Beluga* Laufzeitumgebung implementierten *Storage-Module* besprochen.

Central Das Storage-Modul *Central* ist die Umsetzung des in Abschnitt 3.3.1 angesprochenen Konzeptes eines *zentralen Verzeichnisses*. In der Konfiguration des Laufzeitsystems wird ein einziges Hostsystem (*DSM-Rootserver*) bestimmt, das alle Einträge des DSM verwaltet und speichert. Für Zugriffe anderer Hostsysteme (*DSM-Client*) auf eine oder mehrere DSM-Zellen werden Nachrichten generiert und über das Netzwerk verschickt. Wie in der Abbildung 4.8a skizziert, läuft die gesamte Kommunikation über den

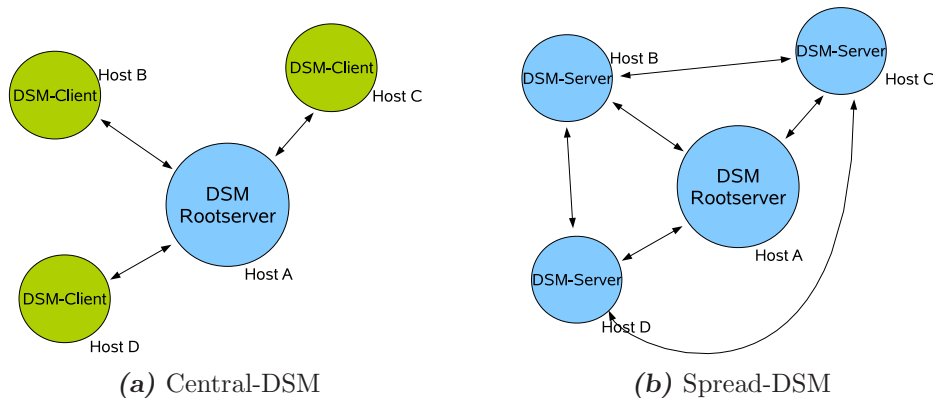


Abbildung 4.8: Veranschaulichung der Client-Server-Struktur für *Central*- und *Spread*-DSM.

zentralen DSM-Rootserver, d.h. es findet keine Kommunikation zwischen den einzelnen DSM-Clients statt. Für die Optimierung der Zugriffe besitzt jeder DSM-Client ein lokales Verzeichnis, in welchem alle temporär bekannten DSM-Zellen mit einigen zusätzlichen Angaben vermerkt sind.

TSpaces Die *TSpaces*-Implementierung von IBM stellt einen einfachen Tupelraum zur Verfügung. Im Gegensatz zur Implementierung von Sun (Java Spaces) sind keine zusätzlichen, hardwareabhängigen Bibliotheken notwendig (vgl. Kapitel 2.3.5). Der *Beluga TSpaces*-Wrapper bildet alle DSM-Zellen 1 : 1 im Tupelraum ab.

Das prinzipielle Vorgehen für die Bereitstellung des Tupelraumes ähnelt dem des *Central*-DSM. Bevor der Tupelraum benutzt werden kann, muß der *TSpaces*-Server auf einem der beteiligten Hostsysteme gestartet werden (*DSM-Rootserver*). Hostsysteme, die diesen Tupelraum benutzen wollen, können dann Tuplespace Kontrollinstanzen erstellen,

¹⁴<http://juxmem.gforge.inria.fr/>

die sich mit diesem *TSpaces*-Server verbinden. Auf diese Weise können mehrere Tupelräume parallel benutzt werden. Die Benutzung ist weitestgehend transparent und für den Benutzer gibt es keine Steuerungsmöglichkeiten, um Tupel auf verschiedene Hostsysteme auszulagern. Die Bibliothek ist nicht threadsicher implementiert¹⁵, deshalb muß bei der Benutzung durch mehrere Threads für jeden Thread eine eigene Kontrollinstanz erstellt werden. Dieses Implementierungsdetail wird durch den DSM-Wrapper verborgen und ist deshalb für das Laufzeitsystem und den Benutzer nicht mehr relevant.

Ursprünglich wurde die TSpaces 2.12 Implementierung von IBM verwendet. Im frei verfügbaren Projekt *OptimalGrid*¹⁶ von IBM wurde eine aktuellere Implementierung mit der Versionsnummer 3.1.04 veröffentlicht. Beide *TSpaces*-Versionen sind nicht quellcodekompatibel, auch eine gleichzeitige Nutzung mit verschiedenen angepaßten Wrapperimplementierungen ist nicht möglich. Die Versionen werden in Kapitel 5.2.2 (S.147) kurz gegenübergestellt und im späteren Verlauf wird ausschließlich die Version 3 verwendet.

Spread Der *Spread*-DSM ist eine Weiterentwicklung des bereits kurz beschriebenen *Central*-DSM. Im Unterschied zu diesem können jedoch existierende Zellen auf andere Hostsystem ausgelagert werden und die Kommunikation kann auch ohne Beteiligung des DSM-Rootservers stattfinden (*siehe Abbildung 4.8b*). Der DSM-Rootserver ist aber für das Anlegen neuer Zellen notwendig und dient zusätzlich als zentrale Anlaufstation für alle anderen DSM-Server im System, um DSM-Zellen ausfindig zu machen. Zu diesem Zweck enthält er immer eine vollständige Liste aller existierender DSM-Zellen und deren Verwaltungsdaten. Wird eine auf dem DSM-Rootserver verwaltete DSM-Zelle auf einen anderen DSM-Server verschoben, so bekommt der Eintrag im zentralen Register einen Verweis auf diesen DSM-Server. Alle eingehenden Anfragen bezüglich dieser DSM-Zelle werden ab diesem Zeitpunkt immer an diese Adresse weitergeleitet.

```

1  if (owner_since > OWNERTIME)
2  {
3      if (last_local_access > ACESSTIME)
4      {
5          if ( external_requests > MIN_REQUESTS)
6          {
7              return true ;
8          }
9      }
10 }
11 return false ;

```

Listing 4.12: Auslagerungsheuristik des Spread-DSM

Die Auslagerung von Zellen auf andere Hostsysteme basiert auf einem speziellen Verfahren, das Zugriff auf interne Statistiken des DSM hat und anhand dieser Daten Entschei-

¹⁵IBM TSpaces Version 3: Client Programmer's Guide

<http://www.almaden.ibm.com/cs/TSpaces/Version3/ClientProgrGuide.html>

¹⁶<http://www.alphaworks.ibm.com/tech/optimalgrid/>

dungen treffen kann. Der Auslagerungsprozeß wird aber im Falle des *Spread*-DSM nur dann angestoßen, wenn Schreibzugriffe erfolgen. Die zur Aktivierung dieses Vorganges verwendete Heuristik ist austauschbar und kann wieder in der *Beluga* Konfigurationsdatei definiert werden.

Das von der Laufzeitumgebung verwendete Standardauslagerungsverfahren beruht auf Beobachtungen des Laufzeitverhaltens einiger Testprogramme und ist in Listing 4.12 als Pseudocode kurz dargestellt. Der Grundgedanke geht von der Annahme aus, daß eine Zelle nach einer gewissen Zeit ohne lokale Zugriffe (`ACCESSTIME`, Zeile 3) ausgelagert werden darf. Um jedoch ein ständiges hin und her zwischen einzelnen Hostsystemen bei konkurrierenden Zugriffen zu vermeiden, muß sich die Zelle bereits seit einiger Zeit im Besitz des jeweiligen Hosts befinden (`OWNERTIME`, Zeile 1) und es sollte nicht die erste Anfrage eines externen Hosts sein. Die Anzahl der mindestens erforderlichen Anfragen `MIN_REQUESTS` - ehe eine Auslagerung erfolgen darf - ist auf den Wert 4 festgelegt¹⁷ (Zeile 5).

4.2.3 DSM-Protocol-Level und Datenkohärenz

Wie im Kapitel 3.3.2 (S.70) beschrieben, definiert das *Protocol-Level* eine Kohärenzschnittstelle, welche in erster Linie ein sog. Kohärenzprotokoll beinhaltet. Zur korrekten Funktionsweise sind daneben aber noch einige Mechanismen wie lokale Caches und Locks notwendig. Diese erfüllen standardisierte Aufgaben und müssen nicht durch jede Protokollimplementierung neu erstellt werden. Das DSM-Subsystem bietet bereits einige dieser Funktionen an. Ein Beispiel hierfür ist der weiter unten nochmals angesprochene *lokale Cache*.

Bei der Zugriffskontrolle auf DSM-Zellen unterscheidet die *DEE* Beschreibung zwischen *Read-* und *Writelocks*. Diese Trennung soll gleichzeitige Leseoperationen auf gemeinsame Daten erlauben. In den verfügbaren Protokoll-Implementierungen für *Beluga* wird nur ein Lock-Typ verwendet, d.h. es wird nicht zwischen *Read* und *Write* unterschieden. Dieses Vorgehen reduziert die Komplexität der notwendigen Vorgänge in der *Beluga* Prototypimplementierung. Für den Benutzer ist dieses Verhalten nicht direkt sichtbar, da in der Benutzerschicht die Locks trotzdem immer mit dem entsprechenden Lock-Typ angefordert werden müssen. Zukünftige Kohärenzprotokoll-Implementierungen können mehrere Lock-Typen umsetzen und bereits bestehende Programme können so direkt profitieren.

Für die konkrete Umsetzung eines Kohärenzprotokolles gibt es eine Vielzahl von unterschiedlichen Herangehensweisen und Lösungen [18, 109, 126], damit die geforderte Datenkonsistenz sichergestellt werden kann. Ein Kohärenzprotokoll für die *Beluga* Laufzeitumgebung muß das Interface `TCoherenceInterface` implementieren und kann in der Klasse `DataTypeFactory` des Packages `beluga.dsm.datatype` registriert werden. Die

¹⁷Bei späteren Tests im Rahmen dieser Arbeit wird dieser Wert zur Bestimmung einiger Eigenschaften als Konstante verwendet, weshalb er hier explizit erwähnt wird.

Benutzung des Protokolles geschieht über sog. *Handle* Datenstrukturen vom Typ *TCoherecenceHandle*, die selbst nur Verwaltungsdaten enthalten, aber für den Benutzer nicht sichtbar sind. Alle von einem Protokoll zu implementierenden Funktionen wurden bereits in Abschnitt 4.2.1 bei der Besprechung der Benutzerschnittstellen dokumentiert. An dieser Stelle sollen die in der *Beluga* Laufzeitumgebung enthaltenen Kohärenzprotokolle beschrieben werden.

Simple Dieses Protokoll ist eine Wrapperimplementierung, bei der alle Operationen direkt auf der Zelle durchgeführt werden. Kohärenz und Caching sind nicht gewährleistet. Insbesondere erfolgt keine Zugriffskontrolle und Schreib- bzw. Leseoperationen können sich überlagern.

WriteThrough Ein einfaches WriteThrough-Protokoll, d.h. ein Schreibzugriff wirkt direkt auf den DSM. Lesezugriffe kommen aus einem internen Cache, solange die Zelle noch als gültig markiert ist.

WriteBack Beim WriteBack-Protokoll werden alle Schreibzugriffe auf eine Zelle solange zurückgehalten (d.h. nicht in den DSM geschrieben), bis das Lock auf die Zelle entfernt wird.

Für die Zwischenspeicherung ihrer Daten verwenden die *WriteThrough*- und *WriteBack*-Protokolle einen lokalen Cache auf jedem Hostsystem. Dieser Cache wird vom DSM-Subsystem bereitgestellt und hat eine beschränkte Kapazität. Intern werden alle Zugriffe auf die enthaltenen Einträge protokolliert, um unbenutzte Einträge entfernen und Speicherplatz freigeben zu können. Außerdem enthalten die Cacheeinträge einige Verwaltungsdaten, die Änderungen an den gehaltenen Daten markieren oder deren Gültigkeit betreffen. Die aktuelle Implementierung benutzt eine einfach verkettete Liste zur Speicherung, kann aber durch eigene Implementierungen an die jeweiligen Bedürfnisse angepaßt und ausgetauscht werden.

Data-Policy

Die in Kapitel 3.3.2 (S.71) beschriebene Funktionalität des *Data-Policy*-Modules hilft in einem DSM-Subsystem bei der automatischen Partitionierung von Benutzerdaten eines DSM-Objektes. Im Einzelnen sorgt das *Data-Policy*-Modul für die korrekte Zuordnung von den in einem DSM-Objekt gespeicherten Daten auf die DSM-Zellen. Hierfür muß die Struktur der abzulegenden Daten bekannt sein. Deshalb ist eine *Data-Policy* immer eng an das im *User-Level* existierende DSM-Objekt gebunden. Je nach Anforderung können in einem Laufzeitsystem auch mehrere *Data-Policy*-Module definiert werden und mit verschiedenen DSM-Objekten benutzt werden. Für den Benutzer der DSM-Datenstruktur¹⁸ ist die *Data-Policy* nicht direkt sichtbar.

¹⁸Ein DSM-Objekt verwaltet eine Datenstruktur im DSM.

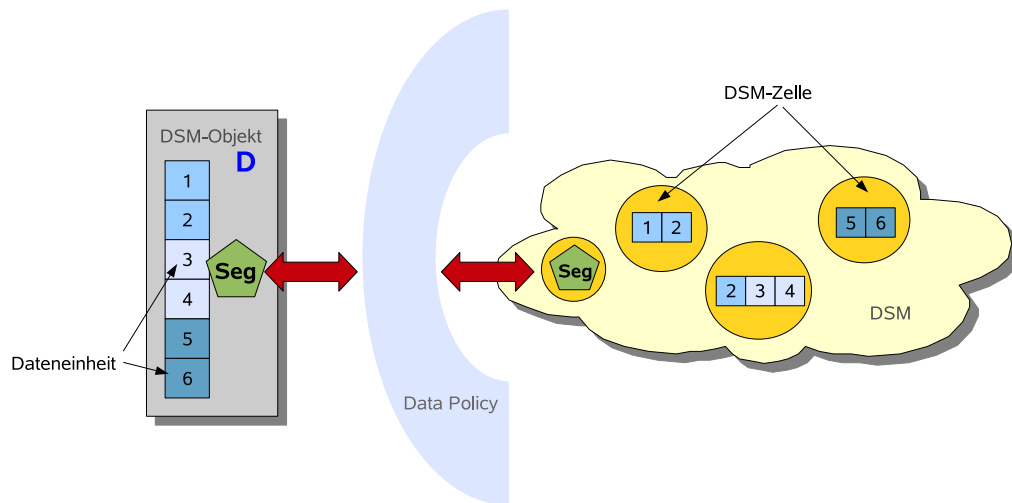


Abbildung 4.9: Schematische Darstellung der Arbeitsweise einer *Data-Policy*. Das in einer DSM-Zelle gespeicherte Objekt **Seg** vom Typ `DataSegmentation` enthält Angaben zur Verteilung der Dateneinheiten 1...6 des DSM-Objektes **D**. Mit den von der *Data-Policy*-Implementierung bereitgestellten Funktionen kann für jede der Dateneinheiten die DSM-Zelle (oder die DSM-Zellen) ermittelt werden, in welcher die betreffende Dateneinheit abgelegt ist. Die Dateneinheit 2 wird in diesem Beispiel in zwei DSM-Zellen redundant gespeichert.

Die in *Beluga* zur Verfügung stehenden *Data-Policy*-Varianten sind statisch, d.h. eine Aufteilung der Daten eines DSM-Objektes auf DSM-Zellen wird nur einmal vorgenommen und ist fest vorgegeben. Im System selbst ist aber eine prinzipielle Unterstützung für einen dynamischen Mechanismus in Form der *DEE* Schnittstellendefinition `DataSegmentation` vorhanden. Die *Beluga* Laufzeitumgebung verwendet intern dieses *Interface* und verlangt von allen DSM-Objekten die Bereitstellung einer Implementierung. Eine nachträgliche Ergänzung mit einer dynamischen *Data-Policy* ist deshalb möglich.

Das für eine Implementierung einer *Data-Policy* notwendige *Interface* `DataSegmentation` enthält Angaben zur Verteilung der Daten auf die DSM-Zellen und ermöglicht so deren Zuordnung. Es wird davon ausgegangen, daß die Daten, welche in einer DSM-Zelle gespeichert werden sollen, in kleine Dateneinheiten zerlegbar sind und von der *Data-Policy* geteilt bzw. zusammengefügt werden können. Um die Abspeicherung in einer DSM-Zelle zu ermöglichen, müssen diese Dateneinheiten als Objektinstanz vorliegen. Die interne Struktur dieses Objektes ist für die *Data-Policy* unerheblich.

Jede der Dateneinheiten wird über einen eindeutigen Index referenziert. Analog hierzu kann man sich ein Feld oder eine Liste **D** vorstellen, dieses würde dann dem DSM-Objekt entsprechen (vgl. dazu *Abbildung 4.9*). Die Elemente von **D** repräsentieren dabei die Dateneinheiten. Jede Dateneinheit kann mindestens einer Zelle zugeordnet werden, d.h. die mehrfache Ablage in verschiedenen DSM-Zellen als Kopie ist möglich. Allerdings muß in diesen Fällen das DSM-Objekt die existierenden Kopien solcher Dateneinheiten in verschiedenen DSM-Zellen selbst verwalten und die Konsistenz sicherstellen.

`int getSize()`

Die Methode liefert die Anzahl der insgesamt verwalteten Dateneinheiten.

`int numberOfSegments()`

Es wird die Anzahl der insgesamt vorhanden DSM-Zellen geliefert, die für die Verwaltung der Dateneinheiten verwendet werden.

`SegmentInfo getSegmentInfo(int index)`

Das zurückgegebene Objekt `SegmentInfo` beinhaltet Angaben zur DSM-Zelle, welche die durch `index` referenzierte Dateneinheit enthält.

`Enumeration<SegmentInfo> getAllLeafs()`

Es wird eine Aufzählung aller Segmentinformationen (Angaben zu den DSM-Zellen) für die verwalteten Dateneinheiten geliefert.

`split(int index, int len)`

Der Aufruf teilt die DSM-Zelle, welche die mit `index` referenzierte Dateneinheit enthält, in zwei DSM-Zellen. Dabei sollte mindestens eine der entstehenden DSM-Zellen `len` Dateneinheiten enthalten.

`merge(int index1, int index2)`

Die DSM-Zellen, welche die Dateneinheiten `index1` und `index2` enthalten, werden zu einer DSM-Zelle verschmolzen.

LocalLockManager

Das DSM-Subsystem der *Beluga* Laufzeitumgebung stellt einen *LocalLockManager*-Mechanismus (vgl. dazu Kapitel 3.3.4) zur Verfügung, um den gleichzeitigen Zugriff mehrerer lokaler Trails auf eine durch das lokale Hostsystem gesperrte (`lock`) DSM-Zelle zu erlauben. Hierbei handelt es sich um die direkte Umsetzung, der in der *DEE* Spezifikation beschriebenen Funktionalität. Die Methoden des *Beluga LocalLockManager* sind hier aufgeführt.

`lock(String cellName, int partID)`

Sperrt die durch den Namen `cellName` referenzierte DSM-Zelle. Der gleichzeitige Zugriff mehrerer Trails auf diese DSM-Zelle ist möglich. Zur Kennzeichnung unterschiedlicher Datenbereiche wird der Parameter `partID` verwendet. Der Zugriff von zwei Trails auf eine DSM-Zelle mit identischer `partID` ist nicht möglich. Die Zuordnung von Datenbereichen der DSM-Zelle auf diese `partID` wird vom *LocalLockManager* nicht vorgenommen.

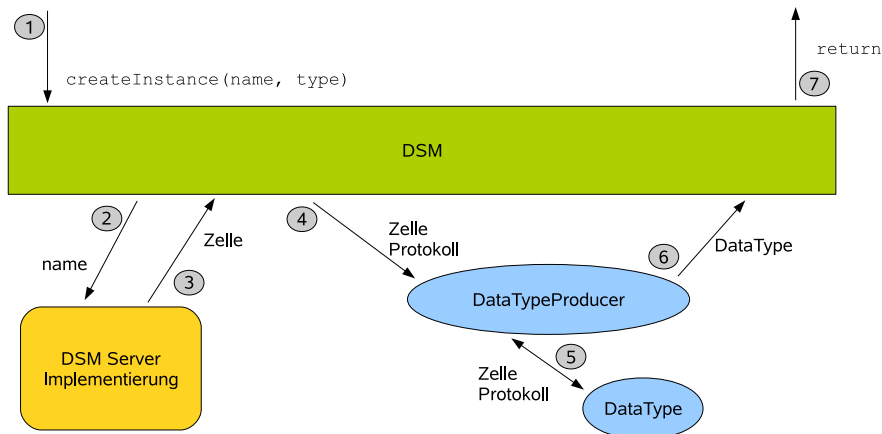


Abbildung 4.10: Schematischer Ablauf einer *create* Phase zum Erzeugen einer lokalen Kontrollstruktur.

`unlock(String cellName, int partID)`

Gibt einen durch `lock` gesperrten Datenbereich (`partID`) der DSM-Zelle (`cellName`) wieder frei. Sind keine weiteren *partiellen* Locks auf diese DSM-Zelle aktiv, wird der globale DSM-Lock auf diese DSM-Zelle entfernt und andere Hostsysteme können die Kontrolle über die DSM-Zelle erlangen.

Im Benutzerprogramm kann eine Instanz des *LocalLockManagers* durch den Aufruf der Methode `getLockManager(TCoherenceInterface protocol)` aus der Klasse *LockManagerFactory* erzeugt und verwendet werden. Mit dem Methodenparameter `protocol` wird das zu verwendende Kohärenzprotokoll übergeben.

4.2.4 DSM-User-Level

Im *User-Level* des DSM-Subsystems sind bereits einige DSM-Objekte vordefiniert (z.B. *Object-Cell*, *Feld* ...). Es können aber auch durch den Programmierer neue DSM-Objekte erstellt und dort registriert werden. In den vorangegangenen Abschnitten dieses Kapitels wurden solche DSM-Objekte bereits mehrfach angesprochen und das *create-init-use* Vorgehen beim Umgang mit solchen Objekten erläutert. Hierbei wurden vor allem die für den Benutzer dieser DSM-Datenstrukturen relevanten Funktionen beschrieben. In diesem Abschnitt soll die *create* Phase eingehender erläutert und dabei vor allem die interne Realisierung betrachtet werden. Für den Programmierer ist die Kenntnis dieser Vorgänge für die Erstellung eigener DSM-Objekte wichtig.

Ein DSM-Objekt ist die Implementierung einer Datenstruktur für den DSM, die auf mehrere DSM-Zellen verteilt sein kann (*siehe Kapitel 3.3.3*). Das Zusammenspiel der einzelnen Klassen und Komponenten bei der Erzeugung (*create*) eines DSM-Objektes und

einer dazugehörigen Kontrollstruktur vom Typ `DataType` ist in Abbildung 4.10 dargestellt. Die einzelnen Abarbeitungsschritte sind hier kurz erläutert:

1. Der Benutzer initiiert den Erzeugungsprozeß eines DSM-Objektes durch den Aufruf von `DSM.space.createInstance(name, type)` (vgl. dazu Kapitel 4.2.1 (S.100)). Der Parameter `name` dient zur Zuweisung eines eindeutigen Namens an die zu erzeugende Datenstruktur. Der genaue Typ des DSM-Objektes (*DataType*) wird durch den Parameter `type` spezifiziert.
2. Anhand des übergebenen Namens `name` wird beim *DSM-Server* eine gleichnamige DSM-Zelle angefordert. Darin finden sich in der Regel einige Verwaltungsinformationen, welche für das weitere Vorgehen notwendig sind.
3. Der *DSM-Server* liefert eine vorhandene Zelle oder erzeugt eine neue leere Zelle mit dem übergebenen Namen im DSM.
4. Durch den im Schritt 1 angegebenen Datentyp `type` kann das zu verwendende Kohärenzprotokoll ermittelt werden. Der *DataTypeProducer* zum Erzeugen der eigentlichen Datenstruktur *DataType* wird ebenfalls aus dem Datentyp abgeleitet und aufgerufen (siehe Kapitel 4.2.1 (S.103)).
5. Der *DataTypeProducer* generiert die gewünschte Instanz. Während der Erstellung können im Konstruktor des DSM-Objektes (bzw. Datentypes) noch weitere **lokale** Initialisierungen vorgenommen werden. Von der DSM-Kernkomponente der *Beluga* Laufzeitumgebung wird hierfür keine feste Methodensignatur vorgegeben. Deshalb ist dieser Erzeugungsvorgang vom jeweiligen DSM-Objekt abhängig und wird vom *DataTypeProducer* übernommen. Die beim Aufruf übergebene DSM-Zelle wird nicht gelesen oder beschrieben, sondern lediglich für die späteren Operationen gespeichert.
6. Ein gültiges DSM-Objekt vom Typ *DataType* wird an die DSM Schicht übergeben. Außer einer Verwaltungszelle wurden (noch) keine weiteren Zellen zur Ablage von Daten im DSM erzeugt. Die Initialisierung der verteilten Datenstruktur (DSM-Objekt), d.h. das Erzeugen der eigentlichen Benutzerdaten, muß durch eine explizite `init` Operation des DSM-Objektes geschehen.
7. Rückgabe der Kontrolle an den Benutzer.

4.3 Kernel Level Implementierung

In diesem Abschnitt werden die für die Laufzeitumgebung maßgeblichen Service- und Verwaltungsroutinen der *Kernel*-Schicht beschrieben. Diese Schicht ist in der untersten Ebene einer *DEE* Umgebung zu finden und wird von den höheren Schichten benötigt. Für den Benutzer ist kein direkter Zugriff vorgesehen.

4.3.1 System-Core-Kernkomponente

Die *System-Core*-Kernkomponente übernimmt das Starten der Laufzeitumgebung. Dazu gehören neben dem Einlesen der Konfiguration und dem Erzeugen bzw. Starten der benötigten Module auch die Ermittlung von Systemwerten und die Bestimmung eines Leistungsindex. Das Kommunikations- und das Namespace-Modul gehören ebenfalls zu dieser Kernkomponente. Diese beiden Module werden im späteren Verlauf dieses Abschnittes detaillierter betrachtet.

Zur Initiierung des Startvorganges der Laufzeitumgebung durch den Benutzer, steht die Routine `Beluga.init()` aus der DEE-Service-Kernkomponente des *User-Levels* zur Verfügung. Diese Funktion übernimmt alle notwendigen Arbeiten (Konfiguration, Initialisierung, ...). Auf die Automatisierung des Vorganges und ein Verschieben in den Startvorgang der JVM wurde verzichtet, um dem Anwender mehr Flexibilität zu ermöglichen. Insbesondere kann so der Zeitpunkt des Startens der *Beluga* Laufzeitumgebung vom Benutzer selbst bestimmt werden.

Startvorgang der Laufzeitumgebung

Beim Starten der *Beluga* Laufzeitumgebung wird die Existenz eines zentralen *Registry-Servers* vorausgesetzt. Dieser hat die Aufgabe, den sich am System anmeldenden Hostsystemen einen eindeutigen Namen zuzuordnen. Anhand dieses Namens können die Hostsysteme dann eindeutig im System identifiziert werden. Außerdem kann beim *Registry-Server* eine Liste mit allen momentan bekannten Hostsystemen erfragt werden. Anhand dieser Liste können die Hostsysteme dann direkt miteinander kommunizieren. In Abbildung 4.11 ist dieser Vorgang für vier Hostsysteme illustriert. Die Definition des *Registry-Servers* erfolgt in der Konfigurationsdatei der *Beluga* Laufzeitumgebung (siehe Abschnitt 4.4). In dieser Konfigurationsdatei können u.a. auch Eigenschaften gesetzt werden, welche erfüllt sein müssen, ehe der Startvorgang abgeschlossen werden kann. Die *Beluga* Lauf-

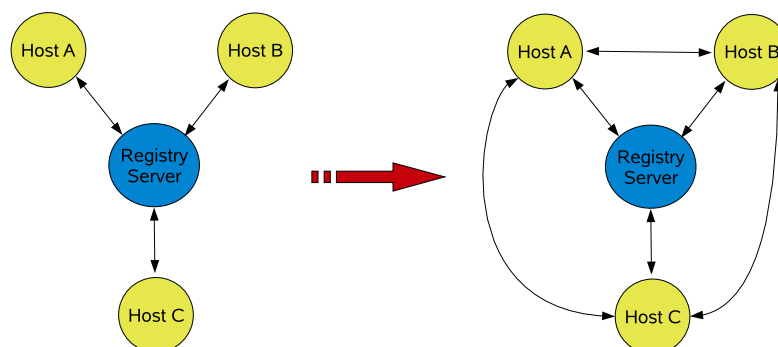


Abbildung 4.11: Anmeldevorgang mehrerer Hosts beim Registry-Server. Der *Registry-Server* wird beim Start benötigt.

zeitumgebung kennt hierfür die Parameter `minhost` und `timeout`. Durch `minhost` kann die Mindestanzahl von vorhandenen Hostsystemen angegeben werden. Die Standardvorgabe für `minhosts` ist 1, deshalb kann durch das Abändern des Wertes die Verteilung auf mehrere Hostsysteme für ein Benutzerprogramm erzwungen werden. Ist diese definierte Bedingung nach der Zeitspanne `timeout` nicht erfüllt, bricht die Laufzeitumgebung mit einer Fehlermeldung ab.

Leistungsindex

Für den Vergleich der beteiligten Hostsysteme untereinander ermittelt das Laufzeitsystem einen numerischen Wert, der Rückschlüsse auf eine einzelne Prozessorleistung zulassen soll. Dieser kann z.B. später für die Verteilung der Trails auf die beteiligten Hostsysteme benutzt werden, um eine möglichst hohe Rechenleistung zu erreichen. Hierbei orientiert sich die Implementierung am *LinPack Benchmark*¹⁹ und berechnet einen ähnlichen Wert anhand von durchgeführten Operationen und der benötigten Zeit einer Benchmarkroutine. Im Gegensatz zu *LinPack* wird hierfür eine Matrixmultiplikation verwendet. Der verwendete Algorithmus hat eine asymptotische Laufzeit von $O(n^3)$. Ebenfalls notwendige Operationen für die Erhöhung von Schleifenzählern oder Speicherzugriffe fließen nicht in die Berechnung des Leistungswertes mit ein.

```

1 int computePerformanceValue( int size )
2 {
3     // Zeit in s
4     double zeit = bench( size ) ;
5
6     // falls die Zeit zu klein ist, wiederholen
7     while( zeit < 0.5)
8     {
9         size = 2 * size ;
10        zeit = bench( size ) ;
11    }
12
13    return calculate(zeit , size) ;
14 }
```

Listing 4.13: Bestimmung des relativen Leistungsindex

Die Bestimmung des relativen *Leistungsindex* wird automatisch beim Starten der Laufzeitumgebung durchgeführt. Zur Laufzeit kann eine Neubestimmung durch den Benutzer mit dem Aufruf von `computeLocalHostPerformance()` aus der Klasse `TSystemSettings` angestoßen werden. Um die Startzeit der Umgebung durch die Ermittlung des *Leistungsindex* nicht unnötig zu verlängern, aber gleichzeitig zuverlässige Ergebnisse zu erzielen, wird das in Listing 4.13 dargestellte Verfahren benutzt. Dort werden zunächst in der Routine `bench` (Zeile 4) die eigentlichen Benchmarkberechnungen ausgeführt und

¹⁹www.top500.org

eventuell mit einer höheren Problemgröße `size` erneut angestoßen, falls die benötigte Zeit unter einer festen, vorgegebenen Grenze liegt (*Zeile 4*). Hierdurch sollen Meßfehler reduziert werden.

Die von der Laufzeitumgebung mit dem in Listing 4.13 dargestellten Verfahren ermittelten Werte können mit Hilfe der beiden in Abschnitt 4.1.1 genauer dokumentierten Methoden `DTM.runtime.getPerformance()` und `DTM.runtime.getQuota()` abgefragt werden. Der von `DTM.runtime.getQuota()` gelieferte Wert kann vom Programmierer benutzt werden, um den Trails dynamisch zur Laufzeit unterschiedlich große Berechnungsanteile zuzuweisen. Dies ist sinnvoll, da in heterogenen Umgebungen die Leistung (MFlops) der einzelnen Prozessoren mit hoher Wahrscheinlichkeit nicht identisch ist und mitunter stark voneinander abweichen kann. Daher sollten die durchzuführenden Berechnungen einer parallelen Anwendung nicht gleichmäßig auf die Hostsysteme/Trails verteilt werden. Zur Veranschaulichung sind in der Abbildung 4.12 die ermittelten L_w -Werte aller im Rahmen dieser Arbeit verwendeter Hostsysteme dargestellt.

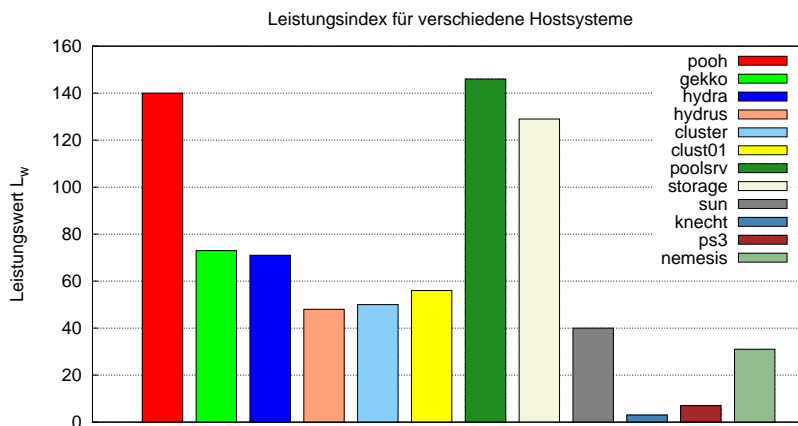


Abbildung 4.12: Von der *Beluga* Laufzeitumgebung ermittelte Leistungswerte L_w .

4.3.2 Namespace-Modul

Das Kommunikationsmodul ermöglicht den Nachrichtenaustausch zwischen verschiedenen Hostsystemen. Das hierfür benutzte Verbindungsnetzwerk ist für die höheren Schichten der Umgebung transparent und kann aus verschiedenen Teilnetzen mit jeweils unterschiedlicher Übertragungstechnologie aufgebaut sein (Abbildung 3.2). Die von diesen Teilnetzen verwendeten Adressen sind mitunter nicht zueinander kompatibel (Ethernet, Bluetooth). Deshalb wird von einem *DEE* System ein eigener, unabhängiger Adressierungsmechanismus (*Namespace*) bereitgestellt, um einzelne Hostsysteme eindeutig identifizieren zu können.

Die Vergabe und die Struktur solcher Adressen orientiert sich am *IP* Protokoll [54]. Jedes Hostsystem bekommt während des Startvorganges der Laufzeitumgebung eine vom *Namespace-Modul* generierte Adresse zugeordnet. Diese Adresse ist ein Objekt vom Typ *THostName* und enthält alle notwendigen Informationen, welche sich in *HostID*, *SubNetID_1*, *SubNetID_2* unterteilen. Zusätzlich enthält jede dieser Adressen noch eine Liste *nativeAdrList* mit allen zugehörigen spezifischen Adressen des benutzen Kommunikationsmodules (vgl. dazu *Listing 4.14*).

```
public class THostName
{
    private int hostID ;
    private int subNetID_1 ;
    private int subNetID_2 ;

    private List nativeAdrList ;
}
```

Listing 4.14: Datenfelder der *THostName*-Klasse

Der in der *Beluga* Laufzeitumgebung verwendete *TCP-Server* ermittelt alle vorhandenen lokalen Netzwerkgeräte und die dazugehörigen IP-Adressen. Diese IP-Adressen werden dann in die Liste *nativeAdrList* eingetragen. Da diese geräteabhängigen Adressen²⁰ Bestandteil der *Beluga* Adresse sind, können diese z.B. von der Kommunikationssimplementierung benutzt werden, um Verbindungen mit anderen, bisher noch nicht verbundenen - also unbekanntem, Hostsystemen aufzunehmen.

Für die Vergabe von Adressen existiert im *Beluga* Prototyp ein zentraler *Registry-Server*. Beim Starten der Laufzeitumgebung versucht jedes Hostsystem, mit diesem Server Kontakt aufzunehmen und eine Adresse zugeordnet zu bekommen. Der *Registry-Server* ist notwendig, um jedem Hostsystem eine eindeutige Adresse zuzuordnen zu können und Adreßüberschneidungen zu vermeiden.

In der aktuellen *Beluga* Implementierung werden keine Subnetze oder *Gateway Server* unterstützt. Es ist nur ein zentraler *Registry-Server* vorgesehen, so daß kein Routing notwendig ist. Eine Erweiterung auf die in Kapitel 3.1 (*S.61*) angesprochene Struktur ist jedoch ohne großen Aufwand möglich. Die Benutzung mehrerer *Registry-Server* kann z.B. durch eine einfache Zuordnung von zu verwaltenden Adreßbereichen geschehen.

4.3.3 Kommunikationsmodul

Für den Nachrichtenaustausch zwischen verschiedenen Hostsystemen verlangt die *DEE* Beschreibung ein Modul für die Bereitstellung einer einfachen Punkt-zu-Punkt Kommunikation. Das verwendete Verbindungsnetzwerk ist dabei für die höheren Schichten transparent, zur Adressierung der Hostsysteme werden die vom *Namespace-Modul* erzeugten Adressen verwendet.

²⁰z.B. Netzwerkgerät *eth0* mit IP-Adresse 132.180.193.32

Die zur Verfügung stehenden Mechanismen unterstützen den Nachrichtenaustausch zwischen Kernkomponenten verschiedener Hostsysteme. Bei der Benutzung wird deshalb die Kenntnis des internen Aufbaues und der Funktionsweise des Laufzeitsystemes vorausgesetzt. Daher existiert keine direkte Unterstützung für Zugriffe des Benutzer auf dieses Modul, hierfür sind die Kernkomponenten des *DEE User-Level* vorgesehen.

In der *Beluga* Prototypimplementierung wird das Kommunikationsmodul als *Beluga I/O-Schicht* bezeichnet. Der Aufbau dieses Modules aus *Bus-* und *Server-*Einheit ist schematisch in Abbildung 4.13 dargestellt. Die Funktionen der einzelnen Einheiten sind hier kurz zusammengefaßt und werden im weiteren Verlauf dieses Abschnittes näher erläutert.

1. **Bus** Der in der *DEE* Spezifikation beschriebene Funktionsumfang des Kommunikationsmodules (*send, receive, broadcast*) wird von der *Bus-*Einheit zur Verfügung gestellt. Diese dient als abstrakte Schnittstelle für den Zugriff auf die Funktionalität der *Server-Einheit*.
2. **Server** Die Server-Einheit ist austauschbar und setzt den notwendigen Funktionsumfang (Kommunikation) für eine spezifische Kommunikationstechnologie (z.B. TCP/IP Sockets) um.

Um einen direkten Datenaustausch - mit eventuell blockierenden Methodenaufrufen - zwischen *Bus-* und *Server-Einheit* zu vermeiden²¹, werden die zu sendenden/empfangenden Daten nicht direkt übergeben. Für den Datenfluß zwischen beiden Einheiten werden Datenpuffer (bzw. Schlangen) verwendet und damit ein *Produzenten/Konsumenten* Konzept [109] zwischen *Bus-* und *Server-Einheit* umgesetzt. Hierbei wartet die Serverimplementierung, bis Daten in diesen Datenpuffer gestellt werden und wird erst dann aktiv. Im umgekehrten Fall blockiert ein Leseaufruf auf den *Beluga I/O-Bus*, bis entsprechende Daten im Datenpuffer vorhanden sind, welche von der *Server-Einheit* dort eingestellt werden. Das Warten wird in beiden Fällen durch die in Java verankerten `wait()` Aufrufe realisiert, welche die weitere Ausführung des aufrufenden Threads unterbinden. Durch einen `notify` Aufruf lassen sich solche Threads wieder aktivieren (*siehe Kapitel 2.4.1 (S.49)*).

Beluga I/O-Bus

Die *Beluga I/O-Bus*-Einheit stellt einige standardisierte Zugriffsmethoden für die höheren Schichten der Laufzeitumgebung bereit. Durch diese Trennung wird es möglich, den Entwicklungsaufwand für die Implementierung neuer *Server-Einheiten* zu reduzieren, weil Routineaufgaben und Servicemethoden vom *Beluga I/O-Bus* des Kommunikationsmodules durchgeführt werden. Die Schnittstelle `TServerInterface` (*siehe Abbildung 4.13*) beschreibt hierbei eine Menge von Funktionen, welche von jedem *Beluga I/O-Server* implementiert werden müssen. In den Klassen `TMessageGenerator` und `TServerQueueHandler` sind die oben angesprochenen Routineaufgaben implementiert. Dabei wird auf

²¹Eine Realisierung mit verschiedenen Threads soll für den Benutzer nicht sichtbar sein.

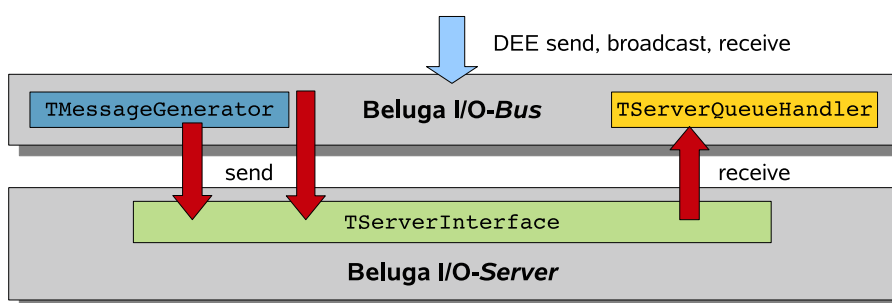


Abbildung 4.13: Aufbau des *Beluga I/O* -[Modules] aus *Bus*- und *Server*-Einheit. *TMessageGenerator*, *TServerQueueHandler* und *TServerInterface* sind Klassen bzw. Schnittstellen mit genau vorgegebenen Funktionsumfang, welche die Implementierung neuer Server-Einheiten vereinfachen sollen.

die Methoden der abstrakten *TServerInterface* Schicht zugegriffen. Die nachfolgende Übersicht faßt die wichtigsten Methoden aus der Schnittstelle *TServerInterface* kurz zusammen.

boolean dialUp(THostName receiver)

Initiiert den Verbindungsaufbau zu dem mit *receiver* angegebenen Hostsystem. Die Rückgabe des Aufrufes ist *false*, wenn beim Verbindungsaufbau Fehler aufgetreten sind.

void send(int type, int tag, THostName receiver, Object msg, boolean compressed)

Sendet die Nachricht *msg* an den Empfänger *receiver*. Die Parameter *type* und *tag* enthaltenen zusätzliche Informationen, an welche Systemkomponente im *Beluga* Laufzeitsystem (DTM, DSM, System-Core ...) die Nachricht gerichtet ist. Durch den Parameter *compressed* kann das Komprimieren der Nachricht in einem *GZIP-Stream* erzwungen werden. Bei den in dieser Arbeit besprochenen Tests werden jedoch immer unkomprimierte Datenströme (*compressed = false*) verwendet. Die Kontrolle kehrt unmittelbar an den Aufrufer zurück, d.h. der Sendevorgang muß noch nicht abgeschlossen sein.

void broadcast(TBroadcastMessage msg)

Führt eine *Broadcastoperation* durch, indem die Nachricht *msg* an alle zur Laufzeit in die Laufzeitumgebung eingebundenen Hostsysteme gesendet wird. Die Kontrolle kehrt unmittelbar an den Aufrufer zurück, d.h. der Sendevorgang muß noch nicht abgeschlossen sein.

TServerQueueHandler getQueueHandler()

Liefert einen Handler für die Behandlung von eintreffenden Nachrichten (s.u.) zurück.

Die Verwendung der Klassen `TMessageGenerator` und `TServerQueueHandler` aus der *Beluga I/O-Bus*-Einheit, welche den direkten Datenaustausch mit der Server-Einheit vermeiden, soll anhand zweier Fallbeispiele, dem Versenden und dem Empfangen einer Nachricht, dokumentiert werden.

Empfang von Nachrichten Für den Empfang von Nachrichten muß sich jede Kernkomponente mit einer eindeutigen Adresse (hier: `NetConstant.ID_DSM_SERVER`) und einer Eingabeschlange, zum Einstellen empfangener Nachrichten, beim *TServerQueueHandler* der *Beluga I/O-Bus*-Einheit registrieren (Listing 4.15 Zeile 8). Anschließend können ankommende Nachrichten aus dieser Eingabeschlange gelesen werden (Zeile 13).

```

1
2 /** Kommunikationsserver und Nachrichtenhandler holen */
3 TServerInterface commServer = ... ;
4 TServerQueueHandler handler = commServer.getQueueHandler() ;
5
6 /** Schlange für eingehende Nachrichten registrieren */
7 TInQueue schlange = new TInQueue() ;
8 handler.registerInQueue(NetConstant.ID_DSM_SERVER, schlange) ;
9
10 ...
11
12 /** Nachricht empfangen */
13 TInputMessage msg = schlange.get() ;

```

Listing 4.15: Empfangen von Nachrichten

Versenden von Nachrichten Zum Versenden von Nachrichten können die Routinen der von `TServerInterface` abgeleiteten TCP-Server-Implementierung verwendet werden (siehe Listing 4.16).

```

/** Implementierung des Kommunikationsservers holen */
TServerInterface commServer =
    TComponentFactory.settings.getIOServer() ;

commServer.send( NetConstant.ID_REGISTRY_SERVER,
                tag, toHost, message, false );

```

Listing 4.16: Senden einer Nachricht an den *Registry-Server*

Als Alternative hierzu bietet die Klasse `TMessageGenerator` eine Vielzahl von vorgefertigten Standardnachrichten und Sendeoperationen an. Der in Listing 4.16 dargestellte Vorgang läßt sich so durch den Aufruf von

```
TMessageGenerator.sendToRegistry(tag, toHost, message) ;
```


vereinfachen. Die Parameter `toHost` und `message` geben den Empfängerhost bzw. die zu übermittelnde Nachricht an. Der Parameter `tag` wird intern vom Laufzeitsystem benutzt und enthält die eindeutige Adresse der Laufzeitkomponenten, welche die Nachricht zugestellt bekommen soll. Diese Adresse `tag` ist identisch mit der eindeutigen Adresse beim Empfang von Nachrichten.

Beluga I/O-Server

Die im Rahmen dieser Arbeit entwickelte Server-Einheit basiert auf den in der Standard Java Klassenbibliothek vorhandenen Klassen für TCP-Sockets und baut darauf auf. Der TCP-Standard ist ein im OSI-Modell [54] auf Schicht 4 angesiedeltes verbindungsorientiertes Protokoll für Netzwerke, insbesondere wird die Zustellung von einmal abgesetzten Nachrichten garantiert. Durch die weite Verbreitung dieses Protokolls gibt es nahezu für jeden Netzwerktyp eine entsprechende Implementierung, z.B. *TCP over InfiniBand*.

Als weitere Implementierungsalternativen kamen im Vorfeld noch Implementierungsvarianten mit RMI oder MPI als *Beluga I/O-Server* in Frage. Gegenüber einer reinen Realisierung mit TCP-Sockets ergeben sich allerdings einige Nachteile. Die wichtigsten Gründe sollen hier kurz genannt werden.

- **RMI** ist als entfernter Unterfunktionsaufruf realisiert (remote procedure call) bzw. eine erweitert dieses Konzept. Die vorhandenen Protokolle bauen auf der bestehenden Netzwerkinfrastruktur (meist TCP) auf und bieten bereits Mechanismen zur einfachen Registrierung von Clients [86]. Operationen für die Kommunikation sind in der Benutzerschicht von RMI nicht vorgesehen. Kommunikationsroutinen müssen daher mit den Mitteln von RMI nachgebildet werden. Deshalb erzeugt die Verwendung von RMI für den reinen Nachrichtenaustausch einen zusätzlichen und unnötigen Overhead.
- **MPI**-Realisierungen für Java benutzen entweder existierende Bibliotheken über die JNI-Schnittstelle oder setzen auf die im `java.net` Package definierten Netzwerkclassen auf.
Die von der *DEE* Beschreibung vorgegebene Funktionalität läßt sich nur mit erhöhten Aufwand realisieren, da beim Versenden von Nachrichten durch MPI immer der Sender und der Empfänger auf beiden Seiten bekannt sein muß. Für eine Laufzeitumgebung, die neben den Programmnachrichten noch Nachrichten zur Systemverwaltung übertragen muß, ist dieser Verhalten eher hinderlich.

Aus diesen Gründen wurde die TCP-Variante als *Beluga I/O-Server* bevorzugt. Dennoch ist es denkbar, die Kommunikation auf RMI bzw. MPI aufzubauen, da für spezielle Anwendungsszenarien die oben als nachteilig aufgeführten Punkte nicht oder nur teilweise greifen. Ein Beispiel wäre z.B. die Ausführung eines *DEE* Programmes, welches ohne DSM-Benutzung auskommt und ausschließlich durch Nachrichtenaustausch kommuniziert.

Bevor nun weitere Details und Designentscheidungen des TCP-Servers besprochen werden, sollen noch einige Hintergründe erläutern werden, um darauf aufbauend die Eigenschaften des Kommunikationsmodulles erläutern zu können.

Java-Sockets und Streams Das Versenden und Empfangen von Daten wird durch die verwendeten Java-TCP-Sockets mittels Datenströmen, den sog. Streams, realisiert. In diese Streams können Daten byteweise hineingeschrieben bzw. ausgelesen werden. Standardmäßig können keine Objekte übertragen werden, dies wird erst möglich, wenn ein `ObjectStream` benutzt und mit dem vom Socket bereitgestelltem Stream verbunden wird :

```
ObjectInputStream inStream = new ObjectInputStream(  
                                socket.getInputStream() ) ;
```

Aus diesem Datenstrom können nun alle über die TCP-Verbindung gesendeten Java-Objekte mit der vom `ObjectInputStream` bereitgestellten Funktion `readObject()` gelesen werden. Voraussetzung ist, daß die behandelten Objekte vom Interface `Serializable` aus dem Package `java.io` abgeleitet und auf der Senderseite in einen `ObjectOutputStream` geschrieben wurden. Hierdurch übernimmt Java automatisch das Ein- und Auspacken der Objekte. Der Programmierer braucht sich nicht darum zu kümmern, wie Felder, Listen oder andere Datentypen in einzelne Bytes zerlegt (serialisiert) und später wieder zusammengesetzt werden. Standardmäßig werden alle Objektattribute serialisiert. Man kann Attribute eines Objektes vom Serialisieren ausschließen, indem man sie mit dem Schlüsselwort `transient` deklariert. Der resultierende Datenstrom ist relativ umfangreich und enthält für jede Klasse die vollständigen Klasseninformationen.

Um Speicher zu sparen bzw. Übertragungsraten zu verbessern, werden mehrfach referenzierte Objekte auf Sender und Empfängerseite automatisch durch die Laufzeitumgebung zwischengespeichert. Beim nächsten Sendevorgang muß nicht mehr das gesamte Objekt übertragen werden, stattdessen genügt ein kurze ID, mit der das Objekt beim Empfänger eindeutig referenziert werden kann. Bei großen Objekten, wie z.B. Bäumen, kann es aber schnell zu Speicherüberläufen der virtuellen Maschine kommen, da der Serialisierungsvorgang im allgemeinen rekursiv durchgeführt wird. Solche Fehler werden durch einen Abbruch der JVM quittiert und lassen sich relativ gut zurückverfolgen und lokalisieren.

Ein weitaus schwieriger zu identifizierendes Fehlverhalten tritt in diesem Zusammenhang beim mehrfachen Versenden von Objekten auf, deren Zustand sich zwischen diesen Sendevorgängen verändert. Die oben beschriebene ID wird nur einmal beim Erzeugen des Objektes erstellt und bleibt während der gesamten Lebensdauer identisch - auch wenn sich die intern enthaltenen Daten verändert haben. Da bereits beim 2. Sendevorgang nur noch eine kurze Referenzinformation in den Sendestrom geschrieben wird, generiert der Empfänger ein Objekt mit veralteten Daten. Durch diese Strategie werden zwar Übertragungskapazitäten gespart, es kann aber auch schnell zu Fehlern in Anwendungen führen, wenn diese Eigenheiten nicht beachtet werden. Die Motivation zur Einführung dieses Mechanismus als Standardverhalten der JVM von Sun Microsystems beruht auf verschiedenen Untersuchungen. Demnach wird die Mehrzahl von Objekten, in den un-

tersuchten Netzwerkanwendungen, nach der Erzeugung und Initialisierung nicht mehr verändert [45].

Dieses Problem kann umgangen werden, indem vom Programmierer verlangt wird:

- A** jedes Objekt selbst zu Serialisieren²² oder
- B** die Streams nach jedem Sendevorgang in den Ausgangszustand zu versetzen (Reset).

Die Variante **A** ist wenig praktikabel, da sie vom Anwender verlangt, die Kodierung der Objekte im Datenstrom selbst zu erledigen. Die zweite Variante **B** vergrößert den Overhead in Form von Datenmenge und Rechenzeit, erlaubt aber die einfache Verwendung von Objekten. Die Implementierung des *Beluga* TCP-Servers benutzt diese Form und setzt die Datenströme nach jedem Sende/Empfangsvorgang zurück. Falls eine effiziente Übertragungsmethode gewünscht wird, besteht noch die Möglichkeit der Kodierung und Zerlegung der Objektdaten in einem Bytefeld. Dabei kann der Programmierer die Daten des Objektes byteweise ablegen und gegebenenfalls Optimierungen vornehmen. Insbesondere müssen bei dieser Form keine Informationen zu Datentypen übertragen werden, da die Reihenfolge und Art der Daten dem Programmierer für die Dekodierung bekannt sind. Dieses Vorgehen reduziert die Menge der Daten im Vergleich zum standardisierten Vorgehen der Java-Laufzeitumgebung.

Dies bedeutet, daß durch die Variante **B** die Objekte sowohl automatisch als auch vom Programmierer selbst verpackt werden können. Weiterhin ist sichergestellt, daß die Objekte korrekt, d.h. mit aktuellen Daten, beim Empfänger ankommen. Die TCP-Implementierung der *Beluga* Kommunikationskomponente verwendet deshalb dieses Verfahren.

Die Effekte bei der Benutzung bzw. Nichtbenutzung des oben beschriebenen Resetmechanismus eines Streams sind in Abbildung 4.14 dargestellt. Hierzu wurde für jeden Test ein `double` Feld fester Größe angelegt und mehrfach zwischen zwei Netzknoten hin- und her gesendet. Da Felder in Java selbst wieder Objekte sind, muß die JVM hier eine Serialisierung vornehmen.

Falls nach jedem Sendevorgang²³ der Stream zurückgesetzt wurde, fiel die Übertragungsrate aufgrund der bei jedem Sendevorgang notwendigen Objektserialisierung ab. Vergleicht man die tatsächlich versendeten Daten (Abbildung 4.14), erkennt man erhebliche Unterschiede zwischen beiden Varianten. In diesem Beispiel werden viermal mehr Daten gesendet, wenn keine Zwischenspeicherung verwendet und die Sockets nach jedem Vorgang zurückgesetzt werden (*Reset*)²⁴. Eine genauere Analyse der Performance erfolgt in Kapitel 5.1.1 (*S.136*).

Für eine Steigerung der Übertragungsraten bieten sich einige Strategien zur Reduzierung der Größe des Datenstromes an. Eine einfache Methode ist die Benutzung der in

²²Interface `Externalizable`

²³Ein Reset kann nur nach dem Schreiben eines Objektes durchgeführt werden.

²⁴Diese Aussage gilt nur wenn die Objekte mehrmals gesendet werden.

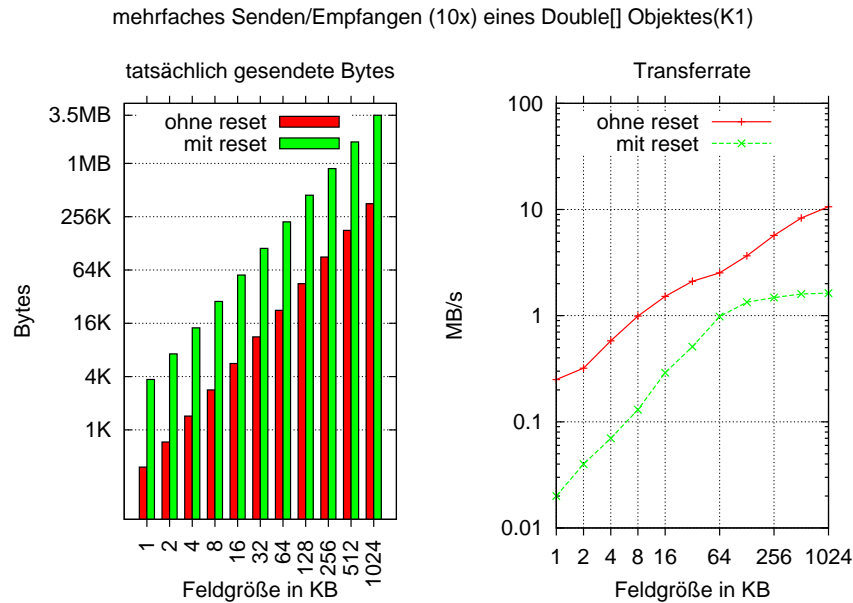


Abbildung 4.14: 10 Ping-Pong Zyklen zwischen 2 Clusterknoten (*clust01*, *clust03*), d.h. auf jedem Knoten wurden insgesamt 10 Lese- und 10 Schreiboperationen durchgeführt. Die resultierenden Messwerte wurden auf einen Lese/Schreibzyklus (*RoundTrip*) normiert, die angegebene Feldgröße bezieht sich auf die Größe eines `double` Feldes in KB.

der Java-Klassenbibliothek vorhandenen GZIP-Streams, d.h. die Daten der Ströme werden komprimiert. Ein Nachteil dieser Methode ist, daß die verwendeten GZIP-Streams nicht wiederverwendet werden können, da der Kompressionsstrom geschlossen werden muß, um den Sendevorgang abschließen zu können²⁵. Eine weitere Variante ist die Benutzung von alternativen Serialisierern. Als ein Beispiel hierfür soll eine XML-Serialisierung anhand der `XStream` Bibliothek²⁶ untersucht werden, da diese am ehesten eine Leistungssteigerung verspricht. Diese Bibliothek wandelt Objekte in eine XML-Beschreibung um, welche dann in einen Datenstrom geschrieben werden kann. Die `XStream` Bibliothek wird dabei als stabil und performant beschrieben [15].

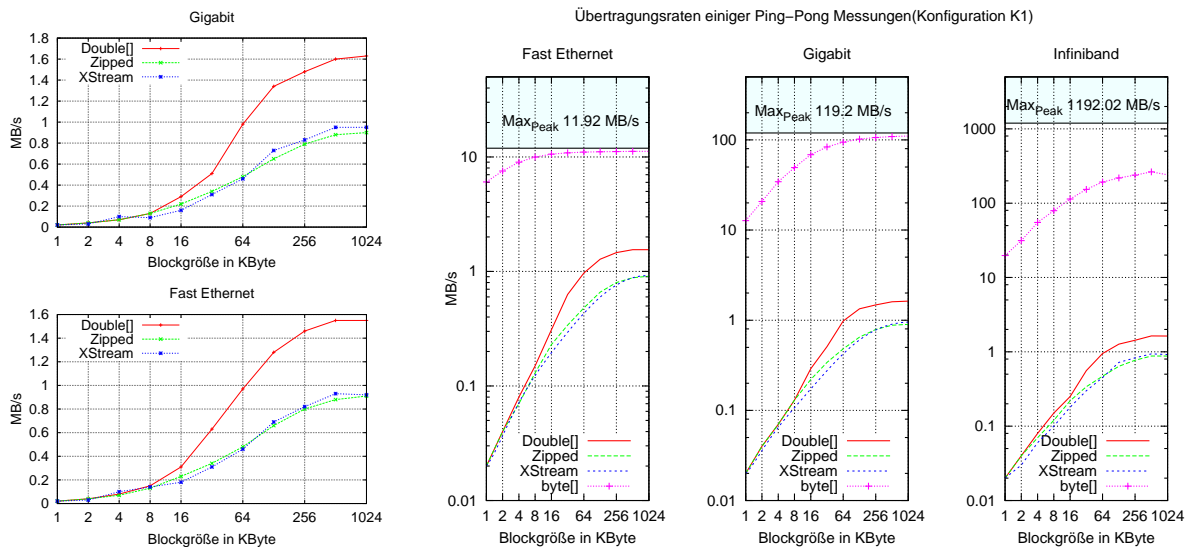
In Abbildung 4.15a sind die Ergebnisse der beschriebenen Varianten für ein Fast-Ethernet und ein Gigabit-Ethernet Netzwerk dargestellt. Vergleicht man diese Messungen, so erkennt man, daß die Standardserialisierung von Java (`Double[]`) im Vergleich mit den hier noch untersuchten Alternativen (`Zippped`, `XStream`) am effektivsten arbeitet. Der bei den Tests verwendete Netzwerktyp hat auf die erzielten Übertragungsraten der Serialisierer keinen nennenswerten Einfluß. Die gemessenen Raten liegen etwa in der Größenordnung von 1 MB/s (für Blockgrößen $\geq 64K$) für alle Netzwerktypen. Abbildung 4.15b zeigt neben den in der Abbildung 4.15a dargestellten Testergebnissen

²⁵Hierbei können durchaus mehrere Objekte versendet werden.

²⁶<http://xstream.codehaus.org/>

auch die Resultate für ein InfiniBand Netzwerk. Auch hier weichen die Ergebnisse kaum von denen der Ethernet Netzwerke ab. Zusätzlich sind in der Abbildung 4.15b auch die Ergebnisse für die Übertragungsraten dargestellt, die erreicht werden, wenn auf die automatische Serialisierung verzichtet wird und der Programmierer einen einfachen Bytestrom nutzt (`byte[]`). Die dabei erzielten Resultate erreichen fast die theoretischen Maximalwerte für die jeweils benutzten Netzwerktypen. Zusammenfassend läßt sich so schlußfolgern, daß die Serialisierung einen erhöhten Berechnungsaufwand beim Kodieren/Dekodieren der Nachrichten erfordert, welcher die Latenzzeiten wesentlich erhöht. Dieser Aufwand ist so hoch, daß eine Erhöhung der Übertragungsleistung (Netzwerktyp) keinen Einfluß auf die erreichbare Übertragungsrate hat.

Der TCP-Kommunikationsserver benutzt zur Serialisierung von Objekten den Standardmechanismus von Java (hier `Double[]`).



(a) direkte Gegenüberstellung der Serialisierer

(b) Vergleich aller Varianten

Abbildung 4.15: Gegenüberstellung verschiedener Übertragungsvarianten in unterschiedlichen Netzwerken. Bei den mit *Zipped* und *XStream* gekennzeichneten Reihen wurden ebenfalls `double` Felder versendet. Die als `byte[]` bezeichneten Meßreihen wurden ohne Objektserialisierung durchgeführt. Alle angegebenen Blockgrößen beziehen sich auf die übertragenen Nutzdaten in Kilobyte (siehe Kapitel 5.1.1).

Der theoretische Maximalwert der Bandbreite des jeweiligen verwendeten Netzwerkes (Fast-Ethernet, Gigabit-Ethernet und Infiniband) ist in der in Abbildung 4.15b mit *MaxPeak* gekennzeichnet (vgl. dazu Kapitel 5.2.3 (S.155)).

SocketFactory Die JVM von Sun erlaubt die Ersetzung der standardmäßig verwendeten Socket-Implementierungen. Hierzu muß von der im `javax.net` Package existierenden abstrakten Klasse `SocketFactory` geerbt und die darin enthaltenen Methoden müssen

implementiert bzw. überschrieben werden. Eine so erstellte Alternative läßt sich direkt durch die Routinen der `SocketFactory` benutzen oder mittels *Java Reflection API* ohne Änderungen am bestehenden Sourcecode in existierende Projekte einbinden. Exemplarisch hierfür wird im späteren Verlauf²⁷ die *Java Fast Sockets* Bibliothek (oder kurz *JFS*) untersucht, welche eine Optimierung der Sende- und Empfangsleistung verspricht [121, 122].

TCP-Server Wie am Beginn dieses Abschnittes (Kapitel 4.3.3 (S.122)) beschrieben, basiert der in der System-Core-Kernkomponente enthaltene TCP-Server (*Beluga I/O-Server*) auf den von Java bereitgestellten Sockets. Java-Sockets folgen einem Client-Server-Modell, d.h für den Aufbau der Kommunikation muß ein Teilnehmer die Rolle eines Servers übernehmen und auf eingehende Verbindungen warten. Der andere Teilnehmer übernimmt die Rolle eines Clients und öffnet eine Verbindung zum Server, danach können beide beliebige Nachrichten austauschen.

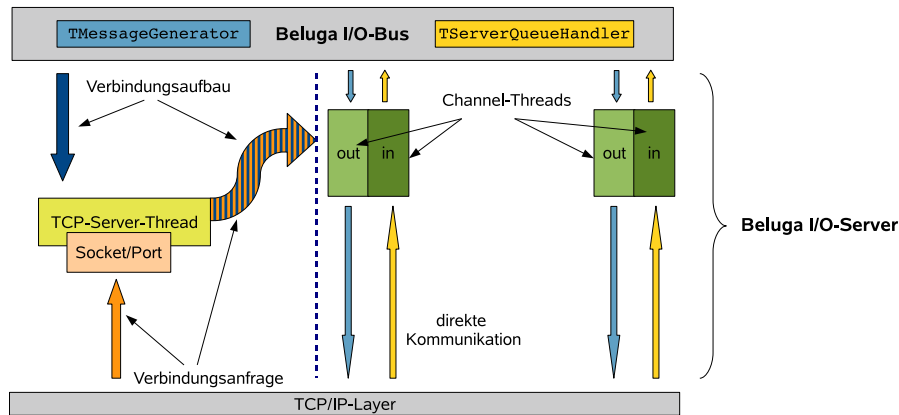


Abbildung 4.16: Aufbau des Kommunikationsmoduls aus *Beluga I/O-Server* und *Beluga I/O-Bus*.

Die Grundzüge des hier beschriebenen TCP-Servers sind in Abbildung 4.16 kurz skizziert. Ein Merkmal dieser Implementierung ist, daß jedes teilnehmende Hostsystem sowohl Server als auch Client ist. Aus diesem Grund muß ein TCP-Server alle eingehenden Verbindungen überwachen. Hierzu wird auf jedem Rechner ein eigener Thread (*TCP-Server-Thread*) gestartet, welcher an einem vereinbarten TCP-Port gebunden ist und nur aktiv wird, wenn eine eingehende Verbindung aufgebaut werden soll. Nach einem erfolgreichen Verbindungsaufbau leitet der Server die Kontrolle an sog. *Channel-Threads* weiter und wartet auf neue eingehende Verbindungsanfragen (siehe Abbildung 4.16). Die Channel-Threads sind notwendig, da die Socketoperationen im allgemeinen blockierend sind und der TCP-Server mehrere Verbindungen gleichzeitig verwalten soll. Ohne die zusätzlichen Channel-Threads müßte der *TCP-Server-Thread* nach dem Aufbauen der

²⁷siehe hierzu Kapitel 5.1.2 (S.141)

Verbindung das Senden und Empfangen von Nachrichten übernehmen und wäre durch diese Aufgabe solange blockiert, bis die Gegenseite ebenfalls die entsprechenden Operationen durchgeführt hat. Neue Verbindungen oder die Beantwortung von Anfragen anderer Systeme könnten deshalb während dieser Zeit nicht durchgeführt werden. Für das *DEE* System wäre ein solches Verhalten aber nicht akzeptabel, weil z.B. Systemnachrichten abgesetzt und gleichzeitig Nachrichten aus anderen Kernkomponenten verarbeitet werden. Deshalb werden für jede Verbindung zwei eigenständige Channel-Threads erzeugt²⁸. Diese Threads sind die meiste Zeit inaktiv, wachen jedoch in bestimmten Zeitintervallen auf und überprüfen, ob Daten vorhanden sind und verarbeiten diese gegebenenfalls. Für den Fall daß eine Verbindung für längere Zeit nicht aktiv war, wird diese geschlossen und die Channel-Threads werden beendet. Bei einem erneuten Verbindungsaufbau werden vom *TCP-Server-Thread* wieder neue Channel-Threads erzeugt.

Die Zeitspanne, nach der es zu einer automatischen Beendigung der Channel-Threads bzw. dem Verbindungsabbau kommt, ist großzügig ausgelegt, damit zusätzlicher Overhead zum erneuten Erzeugen der Sockets, Streams und Threads vermieden wird. In den hier getesteten *DEE* Programmen kommt es daher während des Programmablaufes meist zu keinem automatischen Beenden einer Verbindung. Die Zeitspanne ist jedoch parametrisiert und kann durch einen Konfigurationsparameter gesetzt werden.

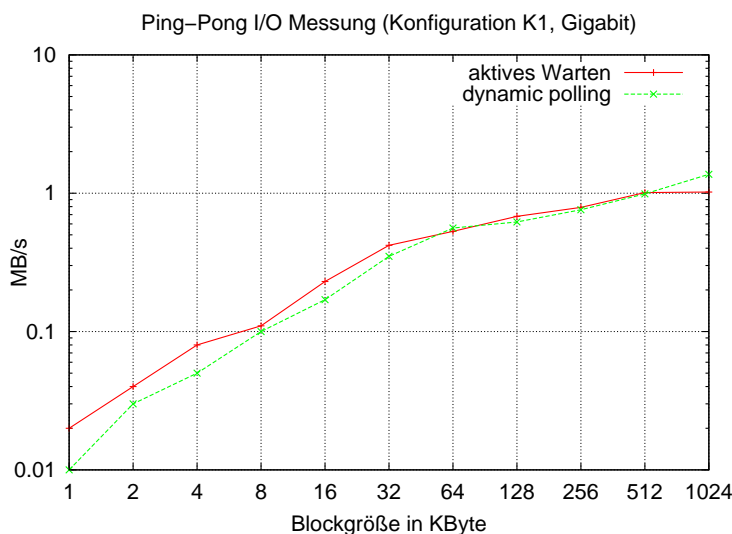


Abbildung 4.17: Übertragungsrate eines PingPong-Tests: Die Verzögerungen durch regelmäßige Suspendierung (Schlafen) des Empfangsthreads sind minimal oder nicht meßbar.

Polling Nach dem Aufbau einer Verbindung wird die Kontrolle an separate Threads weitergegeben (Channel-Threads). Da ein Lesen aus einem Stream des Verbindungssockets

²⁸Je ein Thread zum Empfangen und ein Thread zum Versenden von Nachrichten.

so lange blockiert, bis Daten zum Lesen vorhanden sind, erlangt ein Aufrufer die Kontrolle erst durch Aktivität auf der Senderseite zurück. Durch diesen Umstand kann ein blockierter Empfangsthread nicht mehr beendet werden oder auf lokale Ereignisse reagieren. Deshalb überprüfen die in dieser Implementierung vorhandenen Empfangsthreads vor jedem Lesevorgang die Anzahl der im Eingabestrom vorhandenen Daten und führen die Leseoperation erst aus, wenn eine gewisse Menge an Daten im Stream vorhanden ist. Dieser Vorgang der ständigen Überprüfung wird in einer Schleife stetig wiederholt und würde zu einer übermäßigen Belastung der CPU (aktives Warten) führen. In einem realen *DEE* Programm soll aber die Rechenleistung für die Trails bereit stehen und nicht durch das System verbraucht werden. Aus diesem Grund verfährt jeder Empfangsthread nach dem in Listing 4.17 dargestellten Verfahren (dynamic polling) zur dynamischen Adaption der Wartezeit.

```
while ( true )
{
    // Daten im Eingabestrom
    if ( stream.hasInput() )
    {
        // Leseanweisungen
        // ....

        // Zähler sleepTime zurücksetzen
        sleepTime = MIN_WAIT_TIME ;
    }
    else // keine Daten im Eingabestrom
    {
        sleep( sleepTime ) ;
        if ( sleepTime < MAX_WAIT_TIME )
        {
            // nächste Schlafphase verlängern
            sleepTime += x ;
        }
    }
}
```

Listing 4.17: Adaption der Wartezeit (dynamic polling)

Die Werte der verwendeten Konstanten `MIN_WAIT_TIME` und `MAX_WAIT_TIME` (Listing 4.17) liegen im Bereich zwischen *10ms* und *500ms*. Der genaue Wert kann über Konfigurationseinstellungen bestimmt werden. Dieses Vorgehen führt zu einem recht guten Reaktionsverhalten der Threads, ohne die CPU(s) zu belasten (*siehe Abbildung 4.17*). Die Überprüfung eines mit dieser Heuristik arbeitenden *Beluga* Programmes im Leerlauf zeigte 0% CPU Auslastung²⁹ auf allen getesteten Systemen. Beim Einsatz einer Variante mit aktiv wartenden Threads lag die Auslastung jeweils bei 100%.

²⁹Anzeige des Unix Tools `top`

4.4 DEE-Service-Kernkomponente

Die in der *DEE-Service*-Kernkomponente enthaltenen Routinen dienen zur Analyse und zur Steuerung der Laufzeitumgebung. Hier soll kurz auf die Steuerungsmöglichkeiten des Benutzers (interaktive Shell) und auf die Konfigurationsmöglichkeiten eingegangen werden.

Benutzerschnittstellen der Laufzeitumgebung

Die direkte Ausführung von Programmen nach dem Starten der Laufzeitumgebung ist möglich und soll anhand der nachfolgenden Kommandozeile erläutert werden:

```
java -jar beluga.jar --job automatic -1 test.queens.QueensDSM 10
```

Die Bedeutung der in der Kommandozeile übergebenen Parameter ist hier kurz zusammengefaßt.

<code>--job automatic</code>	Starten der nachfolgend angegebenen Klasse als Benutzerprogramm unmittelbar nach Abschluß des Startvorganges.
<code>-1</code>	Automatisches Bestimmen der Trailanzahl.
<code>test.queens.QueensDSM</code>	Vollständiger Klassenname des auszuführenden Programmes. Die Klasse muß sich im Suchpfad der Java-Umgebung befinden
<code>10</code>	Ein zusätzlicher Parameter, der vom Benutzerprogramm ausgelesen wird.

Neben dem Starten von *Beluga* Programmen direkt von der Kommandozeile existiert eine interaktive Shell. Dort können durch den Benutzer Programmstarts initiiert oder der aktuelle Status des Systems kontrolliert werden. Der Aufruf der Laufzeitumgebung mit dieser Shell erfolgt durch:

```
java -jar beluga.jar --shell
```

Zur Unterstützung des Benutzers beherrscht die Shell eine automatische Vervollständigung für die eingebetteten Kommandos. Eine kurze Übersicht von verfügbaren Kommandos ist in Tabelle 4.1 zu finden.

Konfiguration der Laufzeitumgebung

Die zentralen Parameter der Laufzeitumgebung werden in der `beluga.xml` Datei definiert (siehe auch Tabelle 4.2 und Kapitel B.1.1 (*S.226*) für eine detailliertere Beschreibung). Diese Datei muß im Startverzeichnis der Anwendung stehen und umfaßt im Regelfall nur einige wenige Einträge. Für nicht definierte Parameter werden Standardeinstellungen gesetzt. Die einzig zwingende Angabe ist die Definition eines zentralen *Registry-Servers*. Dieser Server wird von allen Hostsystemen benutzt, um sich am *Beluga* Laufzeitsystem anzumelden und aktuelle Systeminformationen zu erfragen.

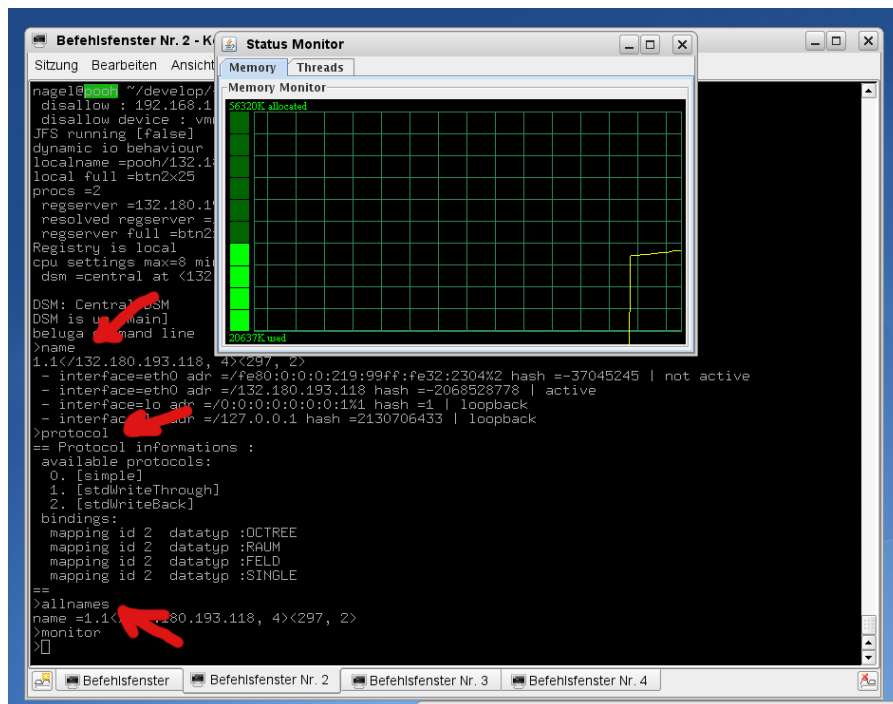


Abbildung 4.18: *Beluga*-Shell. Zu sehen sind neben der grafischen Statusanzeige (monitor) noch die Ausgaben der Kommandos `name`, `protocol` und `allnames`.

```

1 <?xml version="1.0" encoding="iso-8859-1" ?>
2 <beluga>
3
4     <connection port="3012" ipv6="false" device="eth1">
5         <disallow adr="192.168.1.1" />
6     </connection>
7
8     <registry name="132.180.193.118" />
9
10    <boot minhosts="2" timeout="30" />
11
12 </beluga>
    
```

Listing 4.18: Beispiel für eine typische Konfigurationsdatei

Die Parameter für die TCP-Kommunikationskomponente sind in den *Zeile 4* bis *Zeile 6* des Listings 4.18 enthalten. Hierzu zählen z.B. die Auswahl des bevorzugten Netzwerkes (`device`) oder die Definition des Netzwerkports (`port`). Die genauen Angaben in diesem Abschnitt sind abhängig von der verwendeten Kommunikationskomponente. Der zentrale *Registry-Server* wird in der *Zeile 8* durch Angabe einer IP-Adresse definiert. Jedes Hostsystem, welches sich am *Beluga* System anmeldet, versucht mit diesem Server in Verbindung zu treten.

In der *Zeile 10* werden Parameter für den Systemstart gesetzt. Die Belegung des Attributes `minhosts="2"` bedeutet, daß sich noch mindestens ein zusätzliches Hostsystem beim *Registry-Server* anmelden muß, ehe der Startvorgang abgeschlossen wird. Während des Startvorganges werden von der Laufzeitumgebung keine Benutzerprogramme ausgeführt oder angestoßen. Erst nach dem erfolgreichen Abschluß dieser Phase können Programme gestartet werden. Durch den Parameter `timeout` kann die maximale Dauer des Bootvorganges bestimmt werden. Sind nach dieser Zeitspanne die erforderlichen Eigenschaften (hier `minhosts="2"`) nicht erfüllt, bricht die Laufzeitumgebung mit einer Fehlermeldung ab (*siehe Kapitel 4.3.1 (S.114)*).

4.5 Schlußbemerkungen

In diesem Kapitel wurde die Prototypimplementierung *Beluga* beschrieben. Diese besteht aus einer Java-Laufzeitbibliothek und setzt den von *DEE* geforderten Funktionsumfang um. Die einzelnen Kernkomponenten sind austauschbar und lassen sich über eine Konfigurationsdatei (XML-Dialekt) anpassen. Für die DSM-Kernkomponente wurden mehrere *Storage-Module* (*Central*, *Spread* und *TSpaces*) sowie mehrere Kohärenzprotokolle entwickelt. In den nachfolgenden Kapiteln soll diese Prototypimplementierung mit verschiedenen Konfigurationsvarianten des DSM-Subsystems (Storage-Modul, Kohärenzprotokoll) und mit Hilfe verschiedener Testanwendungen auf unterschiedlichen Ausführungsplattformen näher untersucht werden.

<code>name</code>	Liefert den zugewiesenen Namen des Hostsystems in der <i>Beluga</i> Laufzeitumgebung.
<code>allnames</code>	Listet alle angemeldeten und dem aktuellen System bekannten Hostsysteme auf.
<code>cpus</code>	Zeigt die Anzahl der vom System benutzten CPUs an oder oder ändern diese ab.
<code>shared</code>	Erfragt oder ändert das aktuelles Datenmodell.
<code>monitor</code>	Öffnet ein grafisches Dialogfenster mit Statusinformationen zum aktuellen Speicherverbrauch und den Threads der JVM.
<code>ifconfig</code>	Listet alle bekannten Netzwerkadressen der Kommunikationsschicht auf.
<code>protocol</code>	Liefert eine Liste mit allen verfügbaren Kohärenzprotokollen des DSM und der registrierten DSM-Objekte. Zusätzlich wird noch die Zuordnung zwischen diesen angezeigt.
<code>run</code>	Startet ein <i>Beluga</i> Programm und führt es aus.
<code>close</code>	Beendet die gesamte Laufzeitumgebung auf allen angemeldeten Hostsystemen.

Tabelle 4.1: Einige verfügbare Kommandos der *Beluga* Shell.

Parameterklasse	xml Tag	mögliche Attribute und Beschreibung
Kommunikation Registry-Server Bootvorgang	<code>connection</code> <code>registry</code> <code>boot</code>	TCP/IP Port, Netzwerk, Ausschluß von Adressen IP-Adresse des Registry-Servers erforderliche Anzahl von Hosts, maximale Wartezeit bis der Bootvorgang abgebrochen wird, wenn die erforderlichen Ressourcen nicht verfügbar sind
Hostsysteme	<code>hostlist</code>	Angabe von minimaler/maximaler Anzahl von Prozessoren für alle oder einzelne Hostsysteme.
DTM	<code>dtm</code>	Datenmodell <i>shared/private</i>
DSM	<code>dsm</code>	verwendeter DSM-Server, Zuordnung von Kohärenzprotokollen auf DSM-Objekte

Tabelle 4.2: Übersicht über mögliche Parameter der *Beluga* Konfigurationsdatei.

Kapitel 5

Experimentelle Bewertung der Prototypimplementierung

All generalizations are false, including this one.

(Mark Twain)

Zur Überprüfung der korrekten Funktionsweise und zur Einschätzung der Leistungsfähigkeit des *Beluga* Systems wurden verschiedene Laufzeittests durchgeführt. Dieses Kapitel beschreibt die dazu erstellten Programme und diskutiert die erzielten Ergebnisse.

Zunächst werden einzelne Schichten bzw. Kernkomponenten der Laufzeitumgebung getrennt betrachtet und untersucht. Diese Laufzeittests sollen oftmals nur bestimmte Teilbereiche der Implementierung ansprechen. Deshalb wurden hierfür spezielle synthetische Applikationen erstellt. Deren Aufgabe ist neben der Simulation eines fest vorgegebenen Verhaltens, das für die gewünschte Generierung von Leistungsdaten der Laufzeitumgebung notwendig ist, auch die Überprüfung der Korrektheit der durchgeführten Operationen anhand eines nachvollziehbaren Berechnungsergebnisses am Programmende¹.

Im Anschluß an die Benchmarktests einzelner Komponenten wurde das Verhalten des gesamten Laufzeitsystems betrachtet. Ein Ziel dieser Untersuchungen, für welche unterschiedliche Programmvarianten zur Lösung von Matrixmultiplikation und N-Damen-Problem dienen, soll auch die Überprüfung der in Kapitel 1.1 definierten Zielsetzungen sein.

Falls keine anderen Angaben zu den einzelnen Systemkonfigurationen gemacht werden, ist von den nachfolgend beschriebenen Annahmen auszugehen. Generell werden **8 Trails** verwendet, deren genaue Verteilung sich nach der jeweiligen Konfiguration richtet und gesondert angegeben ist. Der Festlegung auf 8 Trails liegen die Überlegungen zugrunde, daß die Verwendung von 4 oder weniger Trails ein zu geringes Potential zur Verteilung der Trails auf mögliche Hostsysteme liefert. Höhere Werte als 8, wie z.B. 16, für die Anzahl der Trails sind mit den verfügbaren Computersystemen ebenfalls realisierbar, vergrößern aber auch die Anzahl von möglichen Verteilungsvarianten der Trails auf die einzelnen Hostsysteme in den betrachteten Konfigurationen. Ein Wert von 8 erscheint deshalb als guter Kompromiß.

¹Insbesondere bei den Untersuchungen zum DSM-Subsystem.

Im Anhang B.2.2 ist ein kompletter Überblick zu allen verwendeten Konfigurationen und deren Zusammensetzung zu finden. In diesem Kapitel werden die Konfigurationen K1, K4 und K5 verwendet. Diese setzen sich jeweils aus homogenen Knoten zusammen und ermöglichen somit eine bessere Vergleichsgrundlage für die untersuchten Fragestellungen.

K1: 4 Clusterknoten, jeweils 2 Prozessorkerne verwendet,
Verbindungsnetzwerk: Gigabit

K4: hydra, 8 Prozessorkerne verwendet

K5: 8 Clusterknoten, jeweils 1 Prozessorkern verwendet,
Verbindungsnetzwerk: Gigabit

Die beschriebenen Laufzeittests sind auch mit heterogenen Konfigurationen durchführbar. In diesen Fällen können sich, bedingt durch die Heterogenität der einzelnen Knoten, Effekte² einstellen, die sich auf die erzielten Ergebnisse auswirken und deren Interpretation erschweren. Deshalb konzentrieren sich die Betrachtungen in diesem Kapitel auf die bereits genannten homogenen Konfigurationen.

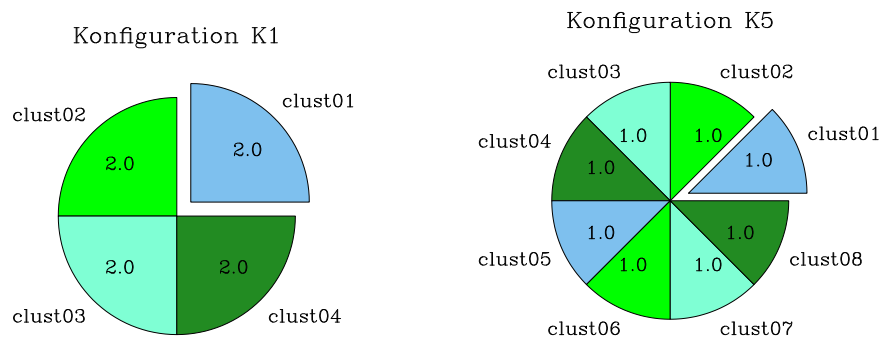


Abbildung 5.1: Zuordnung von 8 Trails auf die einzelnen Hostsysteme in den verteilten Konfigurationen K1 und K5. Die hervorgehobenen Systeme beherbergen zusätzlich den DSM-Server.

Die folgenden Konstanten sind in den nachstehenden Abschnitten ohne explizite Erläuterung zu finden und sollen an dieser Stelle kurz beschrieben werden.

- `#numTrails` bezeichnet die Anzahl der vorhandenen Trails
- `#numHosts` die Anzahl der beteiligten Hostsysteme.

²Ein Beispiel hierfür ist eine unterschiedliche Laufzeiten der Trails für die durchzuführenden Berechnungen in Abhängigkeit von der relativen Systemleistung der Hostsysteme.

5.1 Kommunikationsschicht

In diesem Abschnitt werden die Laufzeittests beschrieben, welche erstellt wurden, um die Kommunikationsschnittstelle der *Beluga* Laufzeitumgebung bewerten und einordnen zu können. Zunächst sollen aber die dazu verwendeten Kenngrößen und Begriffe kurz besprochen werden.

Die **Bandbreite** oder die maximale Übertragungsrate eines Netzwerkes gibt an, wieviele Bytes pro Zeiteinheit über dieses Netzwerk zwischen zwei Knoten gesendet werden können und wird heute meist mit *Mbit/s* angegeben. Dieser Wert weicht eigentlich immer von den real erzielten Werten ab, weil diese Angaben rein auf das physikalische Medium bezogen sind und die Übertragungsdaten einer Anwendung z.T. mehrere Softwareschichten³ durchlaufen müssen [54]. Diese Softwareschichten fügen zu den eigentlichen Sendedaten noch Verwaltungsdaten hinzu. Daraus ergibt sich zum einen eine höhere Anzahl von zu sendenden Bytes und zum anderen kommt die benötigte Verarbeitungszeit zur eigentlichen Laufzeit hinzu.

In diesem Zusammenhang wird in dieser Arbeit auch der Begriff **Nutzdaten** verwendet, um die Anzahl der vom Anwender zu sendenden Bytes zu klassifizieren und um eine klare Abgrenzung von der Anzahl der wirklich gesendeten Bytes zu erhalten. Für einige Netzwerktypen sind in der nachfolgenden Tabelle⁴ die jeweiligen physikalischen Bandbreiten aufgelistet.

Netzwerk	max. Bandbreite	
Ethernet	10 Mbit/s	1.19 MB/s
Fast-Ethernet	100 Mbit/s	11.92 MB/s
Gigabit-Ethernet	1 Gbit/s	119.20 MB/s
InfiniBand	10 Gbit/s	1192.09 MB/s

Der **Durchsatz** wird zur Bezeichnung der Netzwerkbandbreite benutzt, die bei einer bestimmten Anwendung erzielt wird [109]. Da in der Regel verschiedene Nachrichten mit unterschiedlichen Größen, aber auch redundante Informationen versendet werden, kann der Durchsatz erheblich von der durchschnittlichen Bandbreite abweichen. Diese durchschnittlich erzielte Bandbreite kann mit synthetischen Benchmarkprogrammen ermittelt werden und erlaubt den Vergleich verschiedener Kommunikationsnetze.

Als **Latenz** wird die Zeitspanne bezeichnet, die zwischen dem Absetzen der Nachricht beim Sender und dem Empfangen der gesamten Nachricht (Gesamtlatenz) auf der Empfängerseite vergeht [109]. Die Latenz ist abhängig von der Größe der Nachricht, da das Netzwerk durch die Bandbreite begrenzt ist und kleinere Nachrichten entsprechend

³Protokollstack

⁴Die Umrechnung wurden auf der Basis von 1024, statt 1000, durchgeführt, um die *Mbit/s* Angaben auf das für die weiteren Betrachtungen verwendete *MB/s* Format zu bringen. Die angegebenen InfiniBand Werte bezieht sich auf Dual 10Gb/s 4X InfiniBand [63].

schneller versendet werden können. Weiterhin durchläuft die Nachricht den Protokollstack beim Versenden und beim Empfangen. Allgemein kann man die Latenz durch die Formel

$$T_{Latenz} = T_{Overhead} + \frac{Nachrichtengröße}{Bandbreite}$$

vereinfacht ausdrücken.

Die Latenz einer einzelnen Übertragung ist unter realen Bedingungen normalerweise schlecht meßbar, da hierbei die Uhren der beteiligten Systeme absolut synchron laufen müssen. Deshalb behilft man sich hier oftmals mit der *RoundTrip-Zeit*. Mit dieser Zeit ist die Gesamtzeit, die zwischen dem Senden einer Nachricht, deren sofortiger Rücksendung durch den Empfänger und abschließenden Empfangen beim ursprünglichen Initiator vergeht, gemeint. Entsprechend wird der Vorgang allein als **RoundTrip** bezeichnet.

5.1.1 Verwaltungsmehraufwand

Bei der Ermittlung des Verwaltungsmehraufwandes soll der entstehende Overhead durch die *Beluga* Kommunikationsschicht im Vergleich zu den Standardmethoden der Java-Laufzeitumgebung bestimmt werden. Dabei kann man zwischen einem *Datenoverhead* und einem *Zeitoverhead* unterscheiden. Der Datenoverhead resultiert aus dem Anfügen zusätzlicher Daten an die zu sendende Nachricht. Der Zeitoverhead entsteht durch die notwendige Verarbeitung dieser angefügten Zusatzdaten beim Senden und Empfangen. Zur Bestimmung werden zunächst die Werte für die Kommunikation ohne und anschließend mit Beteiligung der *Beluga* Laufzeitumgebung ermittelt. Aus der Differenz der so erhaltenen Meßwerte kann der gesuchte Mehraufwand berechnet werden.

Beschreibung des Testverfahrens

Zur Bestimmung der notwendigen Kenngrößen (*Bandbreite* und *Latenz*) wurde ein einfacher PingPong-Test verwendet. Dieser baut zwischen zwei Hostsystemen eine Verbindung auf und sendet ein Datenpaket mit vorher fest definierter Blockgröße (*blocksize*) mehrfach hin und her. Der Verbindungsaufbau findet im allgemeinen am Anfang jedes Tests genau einmal statt, da dieser Vorgang Betriebssystemressourcen reserviert und somit einen unerwünschten Mehraufwand verursacht. Aus der gemessenen RoundTrip-Zeit und der Anzahl der durchgeführten Tests kann dann ein Mittelwert für die Latenz bestimmt werden. Die Bandbreite läßt sich ebenfalls anhand der RoundTrip-Zeiten berechnen.

Für die nachfolgenden Untersuchungen wurden mehrere Varianten der Testanwendung (Ping-Pong) mit unterschiedlichen Zielsetzungen erstellt. Bei allen Varianten ist der Parameter *blocksize* erforderlich, welcher die Anzahl der zu sendenden Bytes in einer Iteration angibt. Dies bedeutet, daß bei der Verwendung von Java-Bytestreams genau *blocksize* Byte pro Sendevorgang abgeschickt werden. Für die Benutzung der Objektserialisierung werden in den hier beschriebenen Fällen immer Felder vom Typ `Double` verwendet. Deshalb ergibt sich *blocksize* aus:

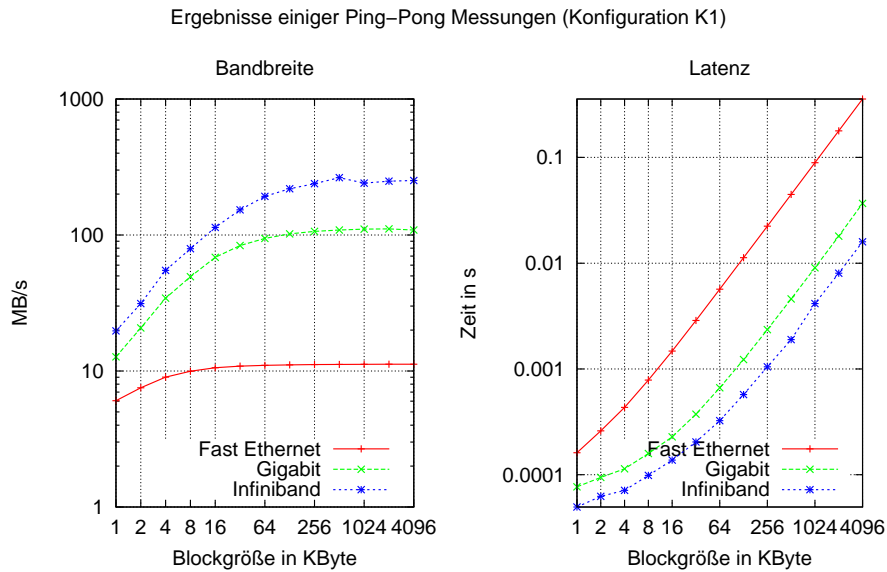


Abbildung 5.2: Die Ergebnisse mehrerer Ping-Pong-Tests (ohne Objektserialisierung) zwischen den Clusterknoten *clust01* und *clust04* für verschiedene Netzwerktypen (K1).

```
blocksize = Double[].length * 8
```

Durch die Verwendung der Objektserialisierung werden allerdings noch weitere Daten automatisch durch die Java-Laufzeitumgebung erzeugt und verschickt. Dieser Datenanteil wird in der Blockgröße (*blocksize*) nicht erfaßt. In den weiteren Betrachtungen wird daher oft der Begriff **Nutzdaten** verwendet, um zu kennzeichnen, daß es sich um reine Anwendungsdaten ohne diese zusätzlichen Verwaltungsinformationen handelt.

Basisgrößen

Zunächst wurden die oben besprochenen Kenngrößen (*Bandbreite* und *Latenz*) für die verschiedenen Netzwerke ermittelt. Die hierfür verwendeten Testprogramme benutzen ausschließlich die in der Java-Umgebung vorhandenen TCP-Sockets ohne Objektserialisierung. Die *Beluga* Laufzeitumgebung ist nicht involviert.

In Abbildung 5.2 sind die ermittelten Bandbreiten und Latenzen dargestellt. Die beiden verwendeten Ethernet-Netze erreichen bereits bei kleinen Nachrichtengröße nahezu die jeweils maximalen Bandbreiten. Für das InfiniBand-Netzwerk mußte eine zusätzliche *InfiniBand Over IP* Softwareschicht benutzt werden, da dieses Netzwerk nicht direkt das IP-Protokoll unterstützt und normalerweise eigene Treiber bereitgestellt werden. Die ermittelten Bandbreiten liegen dennoch deutlich über denen des Gigabit-Netzes.

Anhand der in Kapitel 4.3.3 (*S.122*) gewonnenen Erkenntnisse kann man bei der *Beluga* Kommunikationskomponente von einer Steigerung der Latenzzeit und einem Absinken

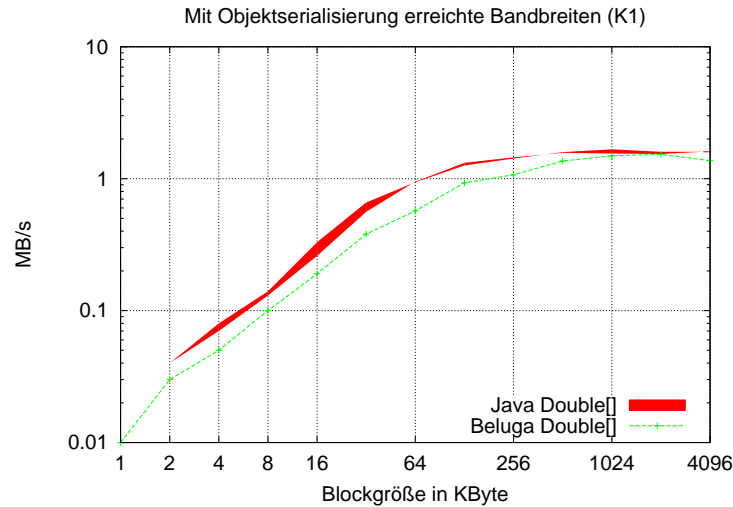


Abbildung 5.3: Erzielte Bandbreiten für Übertragungen mit Objektserialisierung in einer Gigabit-Konfiguration. Die gemessenen Bandbreiten der Java-Variante mit Fast-Ethernet weichen kaum von denen in einem Gigabit-Netzwerk ab (vgl. dazu Kapitel 4.3.3). Zur Illustration wurde deshalb der Bereich zwischen beiden Meßreihen (Gigabit und Fast-Ethernet) aufgetragen und als Java Double[] gekennzeichnet. Bei Beluga Double[] handelt es sich um die erzielten Ergebnisse der *Beluga* Laufzeitumgebung.

der Bandbreiten gegenüber⁵ den Standardsockets von Java ausgehen. In Abbildung 5.3 sind einige Bandbreiten für die Kommunikation mit Objektserialisierung dargestellt. Dort ist erkennbar, daß die Bandbreite der *Beluga* Laufzeitumgebung nur geringfügig gegenüber der erzielten JVM-Bandbreite abweicht. Der entstehende Mehraufwand durch den Einsatz der *Beluga* TCP-Kommunikationskomponente ist deshalb als relativ klein einzuschätzen. Im folgenden soll dieser Overhead genauer untersucht werden.

Datenoverhead

Zur Bestimmung des resultierenden Datenoverheads bei Verwendung der *Beluga* Kommunikationsschicht ist die Kenntnis über die Größe der insgesamt versendeten bzw. empfangenen Daten in Byte notwendig.

Zu deren Bestimmung wurden spezielle `InputStream`- und `OutputStream`-Klassen erstellt. Die einzige Aufgabe dieser Klassen ist das Zählen der insgesamt versendeten bzw. empfangenen Daten. Durch Überschreiben der in den Basisklassen existierenden `read` und `write` Methoden können die Sende- und Empfangsdaten abgefangen und byteweise gezählt werden. Der modulare Aufbau von Streams in Java ermöglicht dann ein Einhängen dieser *Zählströme* zwischen den vom Socket gelieferten Stream und den vom TCP-Server benutzten Objektstream. Für den `InputStream` sieht der resultierende Javacode dann wie folgt aus:

⁵siehe dazu auch Abbildung 4.15

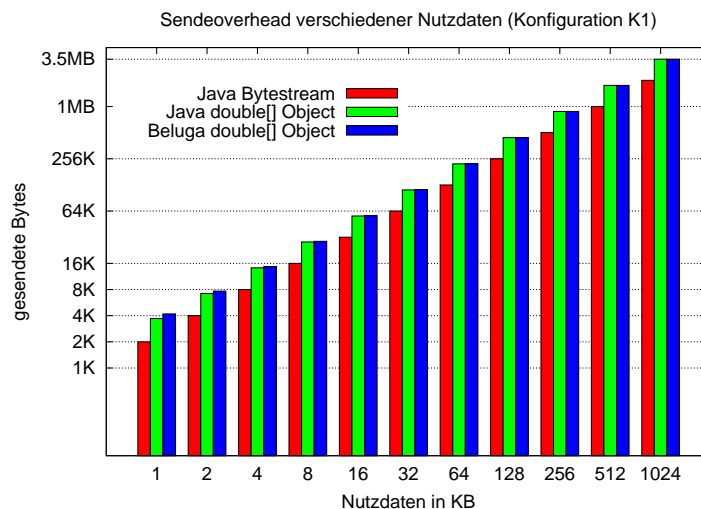


Abbildung 5.4: Vergleich der gesendeten Bytes für einen *RoundTrip*. Die aufgetragenen Daten ergeben sich jeweils aus den beim Absender **und** den beim Empfänger gesendeten Daten (deshalb verdoppeln sich die gesendeten Bytes). Für die Serialisierung des `Double` Feldes wird gegenüber dem Versenden im *Bytestream* annähernd die doppelte Datenmenge erzeugt.

```
new ObjectInputStream (
    new CountInputStream (
        socket.getInputStream() ) ) ;
```

Die mit dieser Methode ermittelten Daten sind eine gute Näherung für die Anzahl der insgesamt gesendeten/empfangenen Bytes. Eine genaue Größe läßt sich mit diesem Vorgehen nicht bestimmen, da noch diverse Protokoll Daten durch die nachfolgenden Schichten eingefügt werden. Aufgrund der Beobachtungen bei der Bestimmung der Basisgrößen (siehe *Abbildung 5.2*) kann man aber davon ausgehen, daß der Anteil dieser zusätzlich eingefügten Daten vernachlässigbar ist. Bei den im Vorfeld durchgeführten Laufzeittests erreichten die verwendeten Java-Socket-Implementierungen nahezu die theoretisch möglichen Maximalwerte für die Bandbreiten. Solche Ergebnisse wären beim Einfügen vieler Zusatzdaten durch die nachfolgenden Protokolle nicht denkbar.

Die Benutzung einer externen Lösung zur Protokollierung der Netzwerkaktivitäten⁶ ist ebenfalls möglich. Es erfordert jedoch einen höheren Aufwand die gewünschten Daten zu extrahieren, wobei der erzielte Nutzen in etwa gleich bleibt.

In *Abbildung 5.4* ist der ermittelte Datenoverhead für verschiedene Blockgrößen (Nutzdaten) abgebildet. Erwartungsgemäß fallen für das direkte Senden mittels eines `Bytestreams` keine zusätzlichen Verwaltungsdaten⁷ an. Das Serialisieren von Objekten erzeugt erheblich mehr Daten. Zum Teil kann bei den vorliegenden Daten eine Verdopplung

⁶z.B. WireShark (vormals Ethereal) www.wireshark.org

⁷Die Anzahl der gesendeten Daten ergibt sich aus den gesendeten Bytes des Absenders (ping) und der Rücksendung des Empfängers (pong).

der Datenmenge beobachtet werden. Der zusätzliche Overhead, der durch die *Beluga* Schichten erzeugt wird, ist dagegen vernachlässigbar. Dieser Overhead ist konstant, da nur wenige Verwaltungsinformationen eingefügt werden, die nicht von der Nachrichtengröße abhängen.

Zeitoverhead

Die *Beluga* Kommunikationskomponente basiert auf den in Java angebotenen Sockets und verwendet deren automatische Objektserialisierung. Somit setzt sich die resultierende Latenzzeit für die Benutzung der *Beluga* Kommunikationsroutinen aus der ohnehin notwendigen Latenz der Socketkommunikation T_{Socket} und den Zeiten für das Serialisieren bzw. Deserialisieren der Objekte $T_{Serializable}$ sowie dem Zeitaufwand der *Beluga* Schichten T_{Beluga} zusammen.

$$T_{Latenz} = T_{Socket} + T_{Serializable} + T_{Beluga}$$

Diese Zeiten (T_{Socket} , $T_{Serializable}$, T_{Beluga}) wurden jeweils getrennt durch entsprechend angepaßte Varianten des bereits beschriebenen *PingPong*-Testprogrammes ermittelt.

In Abbildung 5.5 sind die Anteile der *Beluga* I/O-Schicht an der gesamten Kommunikation dargestellt. Zur besseren Übersicht wurden dort die Meßdaten für insgesamt 10 Roundtrip Zyklen dargestellt. Der *Beluga* Overhead ist für kleine Blockgrößen ($\leq 32K$) kaum meßbar (Abbildung 5.5a). Für Blockgrößen ab $\geq 64K$ ist die Gesamtdauer des gemessenen *Beluga* Overheads nahezu konstant. Dagegen wächst der notwendige Aufwand zum Serialisieren der Objekte sehr stark.

Die in den Messungen (Abbildung 5.5b) erkennbaren prozentualen Schwankungen des *Beluga* Overheads können auf das Multithreading im Kommunikationsmodul zurückgeführt werden. Nachrichten bis zu einer Größen von 32K können oft ohne Zeitverzug abgearbeitet werden. Bei größeren Nachrichten ($\geq 64K$) benötigt die Übertragung unter Umständen etwas mehr Zeit, weshalb der Empfangsthread für eine geringe Zeitspanne deaktiviert wird (siehe Polling Kapitel 4.3.3 (S.127)). Im ungünstigsten Fall wird ein Empfangsthread in dem Moment deaktiviert, in welchem eine Nachricht vollständig eintrifft. Da es sich bei den hier verwendeten Meßdaten um gemittelte Zeiten aus mehreren unmittelbar nacheinander durchgeführten Testläufen handelt, beeinflussen diese Effekte die Resultate stärker als in vergleichbaren Einzeltests. Verschiedene Laufzeittests mit veränderten Parametern (z.B. `MAX_WAIT_TIME` beim Polling des TCP-Servers) führten zu ähnlichen Ergebnissen. Diese sollen an dieser Stelle aber nicht weiter diskutiert werden, da auch dort der Hauptteil der benötigten Zeit ($\geq 80\%$) für die Objektserialisierung aufgewendet werden muß.

Zwischen zwei *Beluga* Hostsystemen ist die Größe typischer Systemnachrichten in der Regel $\leq 1K$. Der Overhead der Kommunikationskomponente ist hier deshalb vernachlässigbar. Die Nachrichtengrößen des DSM-Subsystems oder die Größe von Benutzer- nachrichten läßt sich nicht voraussagen und ist von den Anforderungen der jeweiligen Applikation abhängig.

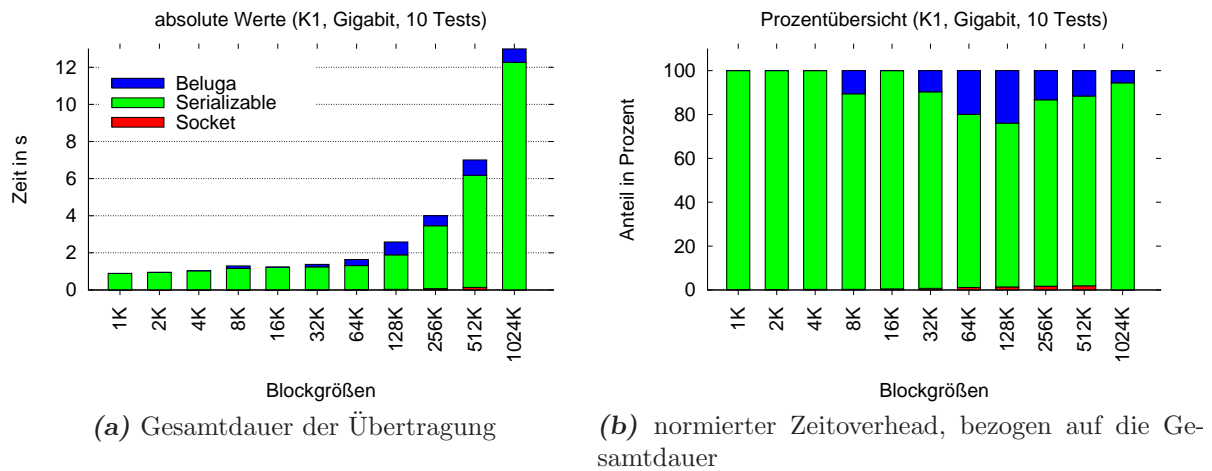


Abbildung 5.5: Zeitlicher Overhead der *Beluga* I/O Schicht, bezogen auf jeweils 10 Ping-Pong Tests. Mit zunehmender Nachrichtengröße wächst der Zeitbedarf für die Serialisierung der Objekte, wobei die eigentliche Übertragung (*Socket*) nur einen Bruchteil der gesamten Zeit aus macht.

5.1.2 Erweiterung durch Java Fast Sockets

Exemplarisch für die in Kapitel 4.3.3 angesprochene Verwendung alternativer Socket-Implementierungen durch die TCP-Server-Einheit des Kommunikationsmoduls soll an dieser Stelle das *Java Fast Sockets* (JFS) Projekt⁸ betrachtet werden [122].

Da *JFS* keine reine Java-Implementierung ist, sondern kritische Teile in eine plattformabhängige Bibliothek auslagert, enthält die *Beluga* Laufzeitumgebung verschiedene spezifische Routinen für den Umgang mit *JFS*. Diese Routinen sind in der Klasse `beluga.comm.io.jfsJFSLoader` definiert und laden eine entsprechende plattformabhängige Bibliothek von *JFS* dynamisch bei Bedarf nach. Das beschriebene Vorgehen wurde erforderlich, da die *Beluga* Laufzeitumgebung in verschiedenen Systemumgebungen getestet werden sollte und hierfür jeweils eine angepasste bzw. für die Plattform übersetzte *JFS* Bibliothek erforderlich ist. Das Listing 5.1 zeigt das prinzipielle Vorgehen für die Benutzung von *JFS* durch Registrierung des alternativen *JFS*-Sockets.

```
SocketImplFactory fab = new jfs.net.FastSocketImplFactory() ;
Socket.setSocketImplFactory(fab) ;
ServerSocket.setSocketFactory(fab) ;
```

Listing 5.1: Registrierung der JFS SocketFactory im Java-System

Die Benutzung von *JFS* kann in der *Beluga* Konfigurationsdatei als Parameter⁹ für den TCP-Server angegeben werden.

⁸<http://jfs.des.udc.es/>

⁹`<parameter jfs="true|false"/>` siehe auch Anhang B.1.1

In Abbildung 5.6 sind die Ergebnisse des bereits mehrfach verwendeten PingPong-Tests mit eingebundener JFS-Implementierung dargestellt. Hierbei handelt es sich um einen Test ohne Serialisierung (5.6a), welcher zum Senden den direkt vom TCP-Socket erhaltenen Bytestream benutzt. Ein weiterer Test (5.6b) benutzt die *Beluga* Kommunikationsschnittstelle und versendet `Double` Felder, was einer Kommunikation mit Serialisierung entspricht. In beiden Fällen sind zum Vergleich die Meßergebnisse äquivalenter Tests mit Benutzung von Java-Sockets ohne JFS aufgetragen.

Die Tests zeigen, daß die Verwendung von JFS zu geringeren Übertragungsraten in den untersuchten Testfällen führt. Für die Kommunikation ohne Objektserialisierung liegt die durch die reine Java-Implementierung erreichte Übertragungsrate bereits sehr nahe am Maximalwert für das benutzte Gigabit-Netzwerk (vgl. auch Abbildung 5.2).

Da *JFS* selbst keine Objektserialisierung bietet, ist die resultierende Übertragungsrate in Teilabbildung 5.6b stark von der Performance der Sockets abhängig. Zusammenfassend läßt sich kein Vorteil durch die Benutzung von *JFS* feststellen.

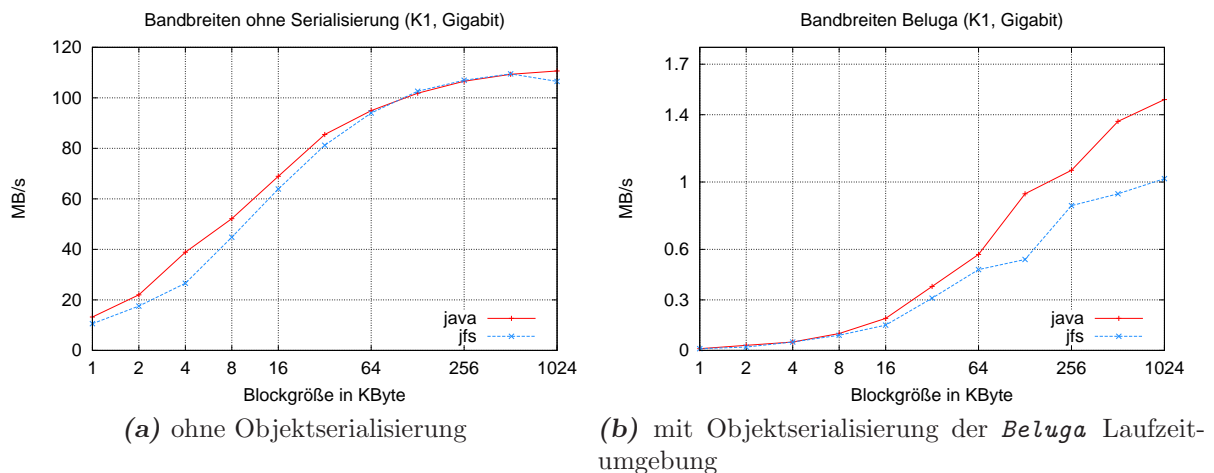


Abbildung 5.6: Erzielte Bandbreiten jeweils mit und ohne die Benutzung der Java Fast Sockets Implementierung (JFS).

5.1.3 Fazit

Der limitierender Faktor bei der Übertragung von Datenobjekten in Java ist die Serialisierung, d.h. das automatische Einpacken bzw. Auspacken der Objekte kostet viel Zeit. Gut erkennbar ist in Abbildung 5.3, daß die Bandbreiten für Fast- und Gigabit-Ethernet nahezu identisch sind. Die maximal ermittelte Bandbreite liegt bei 1,61 MB/s, was die theoretisch erreichbaren Maximalwerte für die beiden Ethernet-Netze nicht annähernd erreicht. Auch durch die Benutzung des schnelleren InfiniBand-Netzwerkes kann deshalb kein zusätzlicher Vorteil bei der Kommunikation erreicht werden¹⁰.

¹⁰Die Meßwerte hierfür wurden, zur besseren Übersicht, in Abbildung 5.3 nicht mit aufgenommen.

Insgesamt betrachtet (Abbildung 5.4 und Abbildung 5.5a) ist der von der *Beluga* Kommunikationskomponente erzeugte Overhead gering und kann im Vergleich mit der von Java durchgeführten Objektserialisierung vernachlässigt werden.

Ein guter Ansatzpunkt für eine Verbesserung der Übertragungsleistung der *Beluga* Kommunikationsschicht wäre z.B. die Verwendung eines eigenen Compilers. Dieser könnte beim Übersetzungsvorgang versuchen, alle Kommunikationsoperationen mit Einzelobjekten durch eine Folge von Anweisungen zu ersetzen, die direkt einen Bytestrom erzeugen. Für die hier erfolgte Nutzung der Laufzeitumgebung als experimentelle Plattform ist die erreichte Leistungsfähigkeit aber durchaus akzeptabel.

5.2 DSM

In diesem Abschnitt soll eine Leistungsanalyse des DSM-Subsystems erfolgen. Das weitere Vorgehen orientiert sich dabei am Aufbau¹¹ des DSM und beginnt mit der Analyse des *Server-Level*. Im Anschluß daran folgen Tests für das *Protocol-Level* sowie eine Bandbreitenmessung für die Beurteilung der Gesamtleistung der verschiedenen DSM-Server-Implementierungen (bzw. der Storage-Module).

Ein weiteres Ziel der meisten Tests war neben der erwähnten Leistungsmessung auch die Evaluierung des Funktionsumfanges der eingesetzten Implementierungen. Bei Neuimplementierungen von Teilen des DSM-Subsystems, z.B. eines DSM-Servers, können deshalb diese Tests eingesetzt werden, um die korrekte Funktionsweise des DSM zu überprüfen. Die Testbeschreibungen enthalten aus diesem Grund auch immer einen Hinweis auf die genaue Zielsetzung.

Falls nicht anders beschrieben, umfaßt ein hier durchgeführter DSM-Tests sowohl die Erzeugung der Zellen, den eigentlichen Test als auch die notwendige Zeit für das Warten auf die Beendigung aller Trails. In Listing 5.2 ist der prinzipielle Aufbau der DSM-Tests als Pseudocode dargestellt. Ersichtlich ist, daß in die ermittelten Zeiten sowohl die Erzeugung von Zellen (Zeile 7) als auch einige Synchronisationsanweisungen (Zeile 11, Zeile 17) mit einfließen.

```

1  timer.start() ;
2
3  // meist erzeugt nur ein ausgezeichneter Trail alle Zellen
4  if (me == master)
5  {
6      dsmCreateCells() ;
7  }
8
9  // Warten auf Beendigung der Zellerzeugung
10 barrier() ;
```

¹¹siehe Kapitel 3.3 (S.66)

```

11
12     // eigentlichen Test durchführen
13     performTest() ;
14
15     // Warten auf die Beendigung aller Trails
16     barrier() ;
17
18     timer.stop() ;

```

Listing 5.2: Zeitmessung für die DSM-Tests

5.2.1 Server-Level: direkter DSM-Serverzugriff

Die hier beschriebenen Tests greifen auf die unterste Ebene des in Kapitel 3.3 (S.66) beschriebenen DSM-Subsystems zu. Durch die Umgehung der höheren Schichten des DSM-Subsystems werden die angebotenen Caching- und Optimierungsmodule umgangen, insbesondere werden keine Daten auf den einzelnen Knoten zwischengespeichert¹². Jeder DSM-Zugriff wird daher direkt in eine Serveranfrage umgewandelt, ohne von den höheren Schichten abgefangen zu werden. Durch dieses Vorgehen kann die Leistungsfähigkeit der als *Server-Level* bezeichneten Schicht direkt bestimmt werden.

Beschreibung der durchgeführten Tests

CellTest Ziel dieses Tests ist die Erzeugung einer kleinen Menge von Zellen durch die verschiedenen Trails. Nach dem Start legt jeder Trail t eine einzelne Zelle im DSM an und beschreibt diese mit seiner eindeutigen ID¹³. Danach werden von jedem Trail alle Zellen einzeln angefordert, gelesen und deren Inhalt ausgegeben. Als Ergebnis sollten `#numTrails` Zellen¹⁴ generiert und jeder Trail als Ausgabe `1...#numTrails` liefern.

WriteTest Beim *WriteTest* (Listing 5.3) soll die DSM-Implementierung unter simulierter Vollast mit vielen konkurrierenden Zugriffen getestet werden. Dazu werden η Zellen durch einen einzigen ausgezeichneten Trail im DSM angelegt. Die Zellen enthalten einen Zähler, dessen Wert bei der Erzeugung mit 0 initialisiert wird. Jeder Trail liest nun in zufälliger Reihenfolge eine Zelle und inkrementiert den darin enthaltenen Zähler. Dieser Schritt wird von jedem Trail genau τ mal ausgeführt (*Zeile 3* bis *Zeile 15*). Abschließend liest jeder Trail i nochmals alle Zellen, summiert deren Inhalt in einer lokalen Variable ρ_i und gibt diesen Wert zur Kontrolle aus (*Zeile 20* bis *Zeile 25*). Es werden insgesamt

$$\kappa = \#numTrails \cdot \tau$$

¹²Dies bezieht sich nur auf ein Caching von Zellen im Rahmen einer Optimierung. Bei der Verwendung des Spread-Servers werden einzelne Zellen u.U. auf unterschiedlichen Hostsystemen gespeichert.

¹³ID im Trail-Team, fortlaufende Nummer ≥ 0

¹⁴wobei `#numTrails` identisch mit der Anzahl der laufenden Trails ist

Schreiboperationen auf die verschiedenen DSM-Zellen durchgeführt. Da die Schreiboperationen konkurrierend ohne Zugriffsschutz erfolgen, können ein oder mehrere Inkrementierungsvorgänge auf die Zähler in den DSM-Zellen verloren gehen. Deshalb gilt für diesen Test $\rho_i \leq \kappa$.

```

1 public void performTest()
2 {
3     for ( t = 1... $\tau$ )
4     {
5         // zufällig eine Zelle auswählen
6         int id = getRandom() ;
7
8         // DSM-Zelle aus dem DSM lesen
9         DSMCell cell = DSM.space.get( Integer.toString(id) ) ;
10
11        // Inhalt der DSM-Zelle inkrementieren
12        IntegerContainer cont = (IntegerContainer) cell.read() ;
13        cont.inc() ;
14        cell.write(cont) ;
15    }
16
17     $p_i = 0$  ;
18
19    // alle Zellen lesen und lokale Summe berechnen
20    for ( t = 1... $\eta$ )
21    {
22        DSMCell cell = DSM.space.get( Integer.toString(t) ) ;
23
24         $p_i += ((IntegerContainer) cell.read()).get()$  ;
25    }
26
27    print  $p_i$  ;
28 }

```

Listing 5.3: performTest Methode des WriteTest als Java-Pseudocode

ExclusiveTest Der *ExclusiveTest* baut auf dem *WriteTest* auf. Allerdings wird der im DSM-Server bereitgestellte *Exclusive*-Mechanismus benutzt, um konkurrierende Zugriffe auf die DSM-Zellen zu verhindern (Zeile 12 bis Zeile 14 in Listing 5.3 wurden hierfür entsprechend abgeändert). Die Ausgabe der einzelnen Trails ρ_i weicht in der Praxis kaum von den im *WriteTest* erhaltenen Werten ab. Der letzte sich beendende Trail liefert den Wert κ . Abweichende Ergebnisse anderer Trails sind darauf zurückzuführen, daß bei deren Beendigung noch nicht alle Trails mit der Abarbeitung der Write-Operationen fertig sind und deshalb im lokalen Zähler ρ_i des Trails i nicht alle Schreibvorgänge erfaßt wurden.

Auswertung

Die Laufzeiten für verschiedene Testläufe sind in Abbildung 5.7 dargestellt, dabei sind die Meßreihen für die beiden *TSpaces*-DSM-Implementierungen (*TSpacesA* (Version 2.12) und *TSpacesB* (Version 3.1)) getrennt erfaßt. Aufgrund der besseren Ergebnisse wird in allen noch folgenden Betrachtungen die *TSpacesB*-Variante verwendet und mit *TSpaces* referenziert.

Betrachtet man die Ergebnisse des *CellTest*, so wird deutlich, daß die *TSpaces*-Varianten etwas mehr Zeit für die Initialisierung der Zellen und den einmaligen Zugriff darauf benötigen. Die Verteilung auf mehrere Hostsysteme scheint darauf keinen Einfluß zu haben. Erwartungsgemäß steigt der Zeitbedarf aber bei der Verwendung mehrerer Hostsysteme für die *Central*- und *Spread*-Varianten wegen der dann notwendigen Kommunikation. Der *WriteTest* und der darauf aufbauende *ExclusiveTest* verursachen aufgrund der Vielzahl der durchgeführten Schreibzugriffe einen erhöhten Kommunikationsbedarf. Dieser wirkt sich direkt auf die ermittelten Laufzeiten aus. Die Vorteile einer dynamischen Verteilung von DSM-Zellen durch den *Spread*-DSM können nicht genutzt werden, da die DSM-Zugriffe zu einem großen Teil zufällig, d.h. wahllos erfolgen. Der Overhead gegenüber dem *Central*-DSM ist in diesen Tests gut erkennbar.

Ein eher unerwartetes Verhalten ist bei der *TSpacesB*-Variante zu beobachten, da die Laufzeiten mit nur einem Host um ein Vielfaches höher sind als im verteilten Fall (4 Hosts). Aufgrund dieser Beobachtung wurde in weiteren Tests das Verhalten der *TSpacesB*-Variante für den *WriteTest* genauer untersucht. Die Ergebnisse sind in Ab-

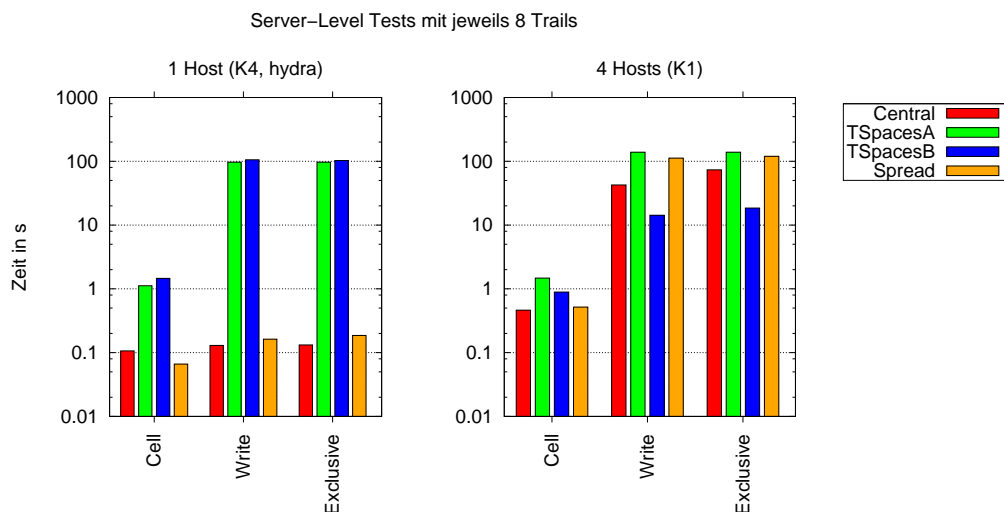


Abbildung 5.7: Ergebnisse der Server-Level Tests mit 8 Trails für die Aufteilung auf ein oder vier Hostsysteme. Als Parameter für die Write- und Exklusivetest wurden $\eta = 100$ Zellen mit jeweils $\tau = 500$ Zugriffen pro Trail gewählt. Die mit *TSpacesA* gekennzeichneten Messungen verwenden IBM *TSpaces* in der Version 2.12, *TSpacesB* bezieht sich auf die Version 3.1.08

bildung 5.8 dargestellt. Dort wird der Trend erkennbar, daß die Laufzeit mit zunehmender Anzahl von Hostsystemen sinkt. Ab ca. 4 benutzten Systemen stagniert diese Entwicklung und bleibt dann in etwa auf diesem Niveau. Für die Testläufe, die insgesamt mehr DSM-Zellen ($\eta = 500$) mit konstant bleibender Gesamtanzahl von Zugriffen darauf ($\tau = 500$ pro Trail) generieren, kann man eine Verringerung der Laufzeit erwarten. Da in diesen Fällen weniger Zugriffskonflikte aufgrund der höheren Anzahl von verfügbaren Datenzellen im DSM auftreten. Dies ist allerdings nicht der Fall, die Laufzeit verdoppelt sich nahezu. Gründe hierfür könnten in einem höheren Verwaltungsaufwand und höheren Kosten für die Initialisierung liegen.

Da mit den bisher durchgeführten Tests keine ausreichende Begründung für das Verhalten der *TSpaces*-Variante gefunden werden konnte, sollen an dieser Stelle weitere Laufzeit-tests durchgeführt werden.

Erzeugung von Zellen Die Laufzeiten der zusätzlich durchgeführten *WriteTests* (Abbildung 5.8) lassen den Schluß zu, daß das Ablegen bzw. das Erzeugen von Daten im Tupelraum der *TSpaces*-Implementierung eine nicht zu vernachlässigende Größe darstellt. Zur Bestimmung des resultierenden Zeitbedarfes zum Anlegen neuer DSM-Zellen wurde daher eine zusätzliche Testanwendung erstellt, die eine feste Anzahl η von DSM-Zellen im DSM durch einen einzigen Mastertrail t_M erzeugt. Um ungewollte Einflüsse durch Netzwerkkommunikation ausschließen zu können, wird zunächst nur ein Hostsystem benutzt. Alle notwendigen Operationen können somit lokal von einem Kontrollfluß durchgeführt werden. Die Laufzeiten der durchgeführten Tests sollten deshalb sehr klein sein. Für die Erzeugung von Zellen auf einem entfernten System wird, bedingt durch die Architektur der verwendeten DSM-Systeme, eine spürbare Erhöhung der Laufzeiten erwartet.

Nach der Analyse der erhaltenen Messdaten (Abbildung 5.9) aus einigen Testläufen, kann die aus den bisherigen Tests abgeleitete Vermutung bestätigt werden. Für die Erzeugung und Initialisierung von Daten benötigt die *TSpaces*-Variante sehr viel Zeit. Dabei spielt es keine Rolle, wo die Zellen erzeugt werden. Die *Central*- und *Spread*-Implementierungen hingegen verhalten sich wie erwartet, d.h. die Erzeugung von DSM-Zellen auf einem lokalen Hostsystem ist um ein Vielfaches schneller als eine entfernte Erzeugung (`clust1` und `clust2`). Die Laufzeiten dieses verteilten Falles sind dabei höher als die Laufzeiten der *TSpaces*-Versionen. Da in den späteren Anwendungsbeispielen eine sehr große Zahl von Daten und Zellen im DSM abgelegt werden soll, ist es empfehlenswert, diese von einem lokalen Trail des DSM-Rootsystems durchführen zu lassen.

5.2.2 Protocol-Level: Cache und Kohärenz

Die bis jetzt beschriebenen Programme hatten das primäre Ziel, die korrekte Funktionsweise der Server-Implementierung zu überprüfen. Normalerweise zwischengeschaltete Kohärenz- und Data-Policy-Mechanismen wurden umgangen. Die folgenden Tests widmen sich deshalb der Aufgabe, ein Kohärenzprotokoll mit zugehöriger Cache-Implementierung zu testen. Prinzipiell ist hierfür auch der *ExclusiveTest* geeignet, allerdings greift

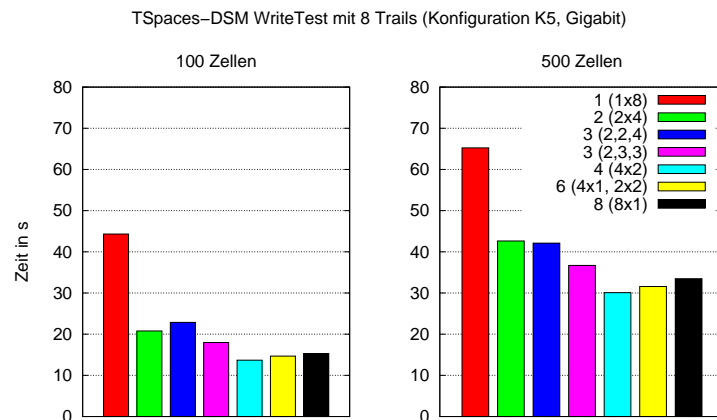


Abbildung 5.8: Laufzeiten des WriteTest mit *TSpaces*-DSM für $\eta = 100$ und $\eta = 500$. Der Parameter $\tau = 500$ und die Anzahl der Trails $\#numTrails=8$ sind jeweils fest gewählt. Allerdings ändert sich die Anzahl der Hostsysteme und damit die Verteilung der Trails. Die Angaben für diese Verteilung sind wie folgt zu interpretieren: $\#hosts (trails)$. Wobei 6 (4x1, 2x2) für 6 Hosts steht, deren 8 Trails sich auf 4 Hosts mit jeweils einem Trail und 2 Hosts mit jeweils zwei Trails aufteilen.

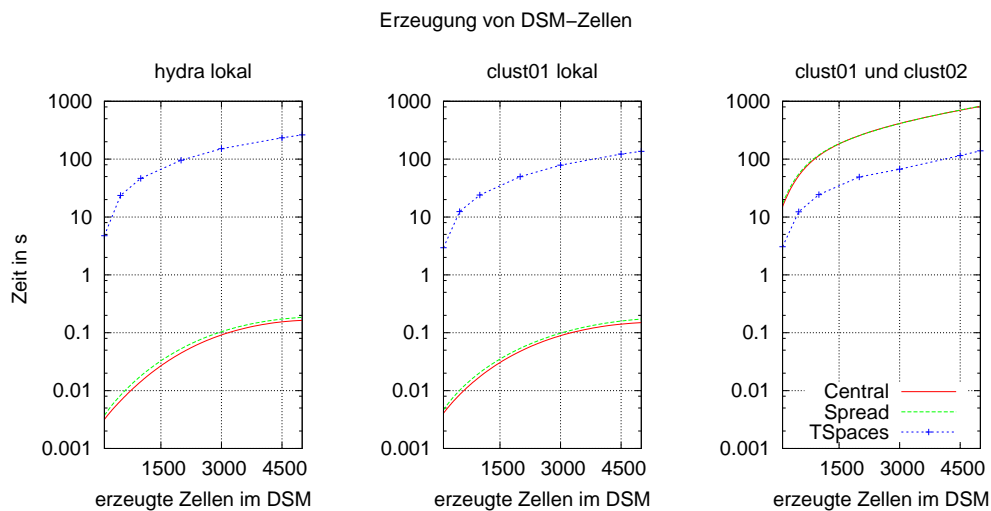


Abbildung 5.9: Erzeugung und Initialisierung von DSM-Zellen. In den mit *lokal* bezeichneten Abbildungen wurden die Zellen auf dem Host des DSM-Servers erzeugt, weitere Hostsysteme waren nicht beteiligt. *Central* und *Spread*-DSM besitzen ein ähnliches Verhalten, die Laufzeiten für den verteilten Fall sind erheblich höher als für die lokalen Testfälle. Beim *TSpaces*-DSM sind keine Unterschiede zwischen verteiltem und lokalem Test feststellbar.

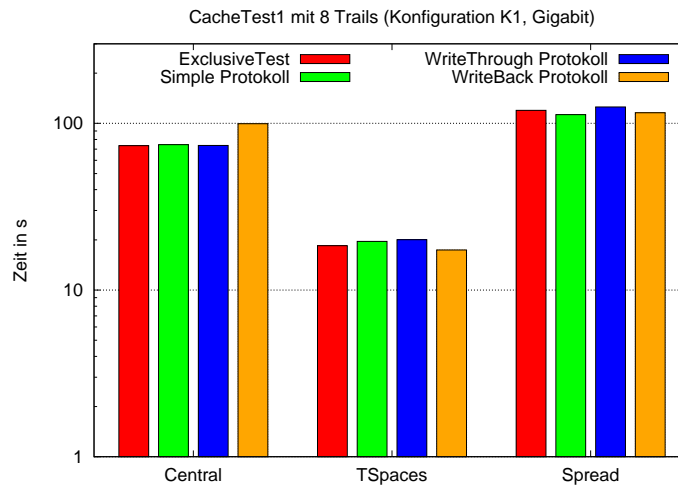


Abbildung 5.10: Dargestellt sind die Laufzeiten des *CacheTest1* mit $\eta = 100$ und $\tau = 500$ für die verschiedenen Kombinationen aus Kohärenzprotokoll und DSM-Server. Zum Vergleich sind noch die Zeiten des *ExclusiveTest* enthalten, der sich nur in der Art des verwendeten Lockmechanismus unterscheidet.

dieser Test wahllos auf einzelne Zellen zu und manipuliert diese. Längere Zugriffssequenzen auf eine oder eine begrenzte Anzahl von Zellen werden nicht generiert, was den Wirkungsgrad des Caches begrenzt.

Beschreibung der durchgeführten Tests

CacheTest1 Der *CacheTest1* unterscheidet sich vom *ExclusiveTest* nur durch die Verwendung eines Kohärenzprotokolles anstatt die Zelloperationen direkt in den DSM abzusetzen. Es werden wieder η Zellen vom Mastertrail angelegt, danach manipuliert jeder Trail genau τ zufällig ausgewählte Zellen. Jede Schreib-/Leseoperation wird über das Kohärenzprotokoll abgewickelt, wobei der vom Protokoll zur Verfügung gestellte *Lock* zur Steuerung von konkurrierenden Zugriffen benutzt wird.

Dieser Test dient zur Überprüfung der korrekten Funktionsweise der Protokoll-Locks. Er kann aber auch im direkten Vergleich mit dem *ExclusiveTest* für die Ermittlung von Lock-Overhead bzw. zur Beurteilung der Gesamtleistung des verwendeten Kohärenzprotokolles herangezogen werden.

In Abbildung 5.10 sind die Ergebnisse des Cachetest für die Protokolle *Simple*, *WriteThrough* und *WriteBack* abgebildet (siehe Kapitel 4.2.3). Da dieser Test ebenfalls keine einheitlichen Zugriffsmuster erzeugt, können kaum Performancegewinne durch den Einsatz der Protokolle im verteilten Fall erzielt werden. Dies zeigt sich am deutlichsten beim Meßwert für das *WriteBack*-Protokoll in Verbindung mit dem *Central*-DSM. In diesem Fall kann der Overhead direkt abgelesen werden. Für die Werte der *TSpaces*- und *Spread*-Implementierungen kann man diese Aussage nicht direkt übernehmen. Dort ver-

ursachen die internen Mechanismen zur möglichst effektiven Verteilung der Speicherorte von Zellen wahrscheinlich bereits einen gewissen Overhead, der dann nicht mehr durch das entsprechende Protokoll ausgeglichen werden kann. Wie erwartet, führt keines der untersuchten Protokolle (mit dieser Testanwendung) zu einer spürbaren Verringerung der Zugriffszeiten.

CacheTest2 Dieser Test baut wieder auf dem in Listing 5.3 dargestellten Verfahren auf und generiert η Zellen im DSM, welche jeweils ein Feld F der Größe s_j mit Integerwerten enthalten. Im Gegensatz zu allen bisher beschriebenen Tests, greifen die Trails eines Hosts pro Schritt t auf genau **eine** der η Zellen zu. Diese in Schritt t verwendete DSM-Zelle wird von einem ausgezeichneten Trail¹⁵ (**master**) zufällig bestimmt. Im Anschluß daran fordern alle Trails des Hostsystems diese DSM-Zelle aus dem DSM an und inkrementieren einen beliebigen Eintrag des in dieser Zelle enthaltenen Zählerfeldes F . Dieser Test erzeugt eine Vielzahl von konkurrierenden Zugriffen auf eine Zelle, die alle vom gleichen Hostsystem generiert werden (Datenlokalität). Da die Kohärenzunterstützung des DSM-Subsystems benutzt wird, sollten die Zugriffe auf den DSM verringert werden. Der gleichzeitige Zugriff auf verschiedene Bereiche des Zählerfeldes ist allerdings nicht möglich, d.h. die Zugriffe auf eine Zelle werden sequenzialisiert.

CacheTest3 Um lokale Zugriffe auf unterschiedliche Bereiche einer Speicherzelle parallel durchführen zu können, stellt das DSM-Subsystem eine Schnittstelle bereit. Dieser als *LocalLockManager* bezeichnete Mechanismus (siehe Kapitel 4.2.3) erlaubt den konkurrierenden Zugriff auf eine Zelle, die z.B ein Feld enthalten kann. Zur Überprüfung der korrekten Funktionalität wurde der *CacheTest3* erstellt. Dieser Test baut wieder auf dem vorangegangenen *CacheTest2* auf. Im Unterschied zu diesem Test wird der von *Beluga* bereitgestellte *LocalLockManager* benutzt, damit die Zugriffe auf verschiedene Feldeinträge innerhalb einer DSM-Zelle gleichzeitig durch die lokalen Trails durchgeführt werden können.

CacheTest4 Da die bisher erstellten Tests weitestgehend irregulär auf die gespeicherten Daten zugreifen, die Entwicklungsziele für den *Spread*-DSM und die erweiterten Protokolle (*WriteThrough* und *WriteBack*) jedoch auf eine Optimierung von Mehrfachzugriffen abzielen, wurde der *CacheTest4* erstellt.

Dieser Test erzeugt wie der *CacheTest2* η Zellen im DSM, wobei alle Zellen wieder Zählerfelder enthalten, die von den Trails manipuliert werden. Für die Zugriffskontrolle wird der für dieses Anwendungsbeispiel relative grobe Lockmechanismus der Kohärenzschicht (Kapitel 4.2.3) benutzt¹⁶. Der bisherige wahllose Zugriff auf beliebige Zellen im DSM wird nun durch eine Beschränkung auf Hostebene ersetzt. Dies bedeutet, daß alle Trails eines Hostsystems nur auf eine feste Teilmenge aller Zellen im DSM zugreifen (*Bereiche*).

¹⁵pro Hostsystem gibt es einen **master** Trail

¹⁶Dadurch ist eine direkte Vergleichbarkeit mit dem *CacheTest2* gegeben.

Dabei sind die von den Hostsystemen verwendeten Teilmengen zu einem Großteil disjunkt, d.h. pro Teilmenge gibt es ein bis zwei DSM-Zellen, welche noch zu einer anderen Teilmenge gehören (Abbildung 5.11). Mit dieser Verteilung sollen kleinere Abhängigkeiten zwischen einzelnen Hostsystemen modelliert werden. Dieses Vorgehen generiert viele ähnliche Zugriffe eines Hostsystems, die besser vom DSM-Subsystems optimiert werden können.

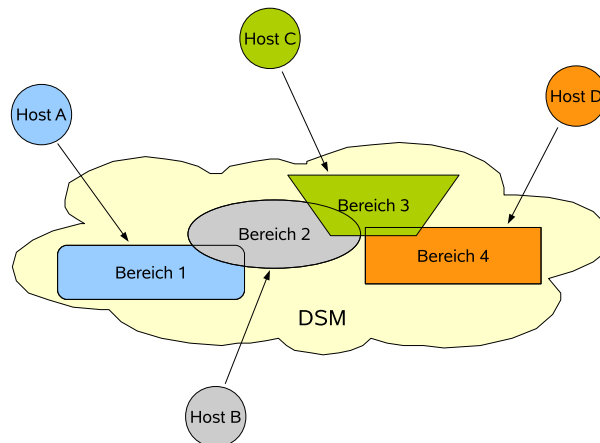


Abbildung 5.11: Schematische Darstellung des Zugriffsverhaltens der beteiligten Hostsysteme auf den DSM beim CacheTest4.

Auswertung

Bei den durchgeführten Tests (vgl. dazu Abbildung 5.12) zeigt sich, daß der *TSpaces*-DSM eine gute Performance bietet. Die DSM-Implementierungen *Central* und *Spread* weisen kaum nennenswerte Unterschiede in ihrem Verhalten auf. Bei vielen ähnlichen Zugriffen¹⁷ eines Trails (*CacheTest4*) kann aber die *Spread*-Implementierung gegenüber der *Central*-Variante eine gute Geschwindigkeitssteigerung erzielen.

Ein etwas anderes Bild ergibt sich bei der Betrachtung der in den Tests verwendeten Kohärenzprotokolle. Die Leistung kann durch deren Verwendung kaum beeinflußt werden und die Laufzeit steigt in den meisten Fällen sogar noch an. Ein Grund für dieses Verhalten liegt in der Art und Weise der durchgeführten synthetischen Tests. Die erweiterten Protokolle (*WriteBack*, *WriteThrough*) gehen von der Annahme aus, daß Zugriffe zwischengespeichert werden können, was in den hier durchgeführten Testszenarien oft nicht der Fall war.

Für die *TSpaces*-Implementierung treten diese Laufzeitunterschiede nicht in diesem Maße auf. Die verwendete *TSpaces*-Bibliothek ist bereits zu einem hohem Grad optimiert und kann die Zugriffe sehr performant beantworten. Es existieren allerdings keine

¹⁷Die Zugriffe beschränken sich meist auf eine kleine Anzahl von unterschiedlichen Zellen im DSM.

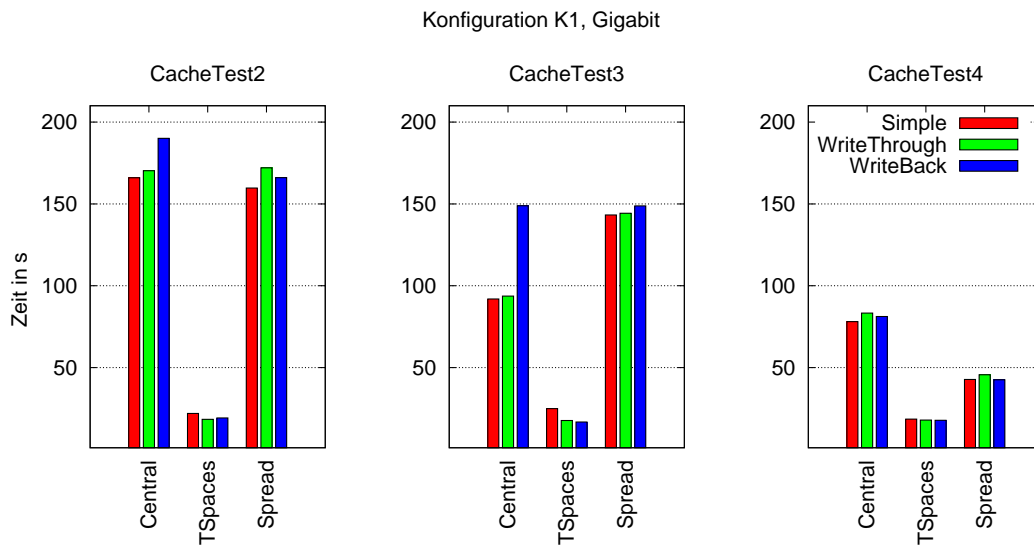


Abbildung 5.12: Ergebnisse der Cache-Benchmarks mit $\eta = 100$ und $\tau = 500$.

aussagekräftigen Veröffentlichungen, welche die Funktionsweise der Bibliothek detailliert beschreiben. Deshalb basieren die hier getroffenen Aussagen meist auf der Analyse der durchgeführten Tests. In [80] beschreiben die Autoren *TSpaces* selbst als:

„... *network communications buffer with database capabilities*...“.

Diese Beschreibung trifft sehr gut auf die hier gemachten Beobachtungen zu, da die Schreib- und Lesezugriffe mit guter Performance durchgeführt werden. Beim Erzeugen neuer Daten entstehen allerdings sehr hohe Kosten.

Ein weiterer Punkt, der bei den bisherigen Betrachtungen noch keine Erwähnung fand, sind simultane Zugriffsprobleme der *TSpaces*-Bibliothek. In einigen wenigen Fällen können Tupel¹⁸ verloren gehen, d.h. die Bibliothek scheint fehlerhaft zu sein. Dies hat zur Folge, daß Zugriffe auf diese Tupel blockieren und somit die gesamte Ausführung stoppt. Da der Quellcode für die Bibliothek nicht verfügbar ist, kann diese Vermutung nicht eindeutig belegt werden und stützt sich nur auf selten reproduzierbare Beobachtungen.

5.2.3 Bandbreite

Ein Kriterium für die Bewertung von physikalischem Hauptspeicher ist wie bei den Netzwerken die *Bandbreite*, als Maß für die Anzahl der Bytes, die in einem bestimmten Zeitraum vom Speichermodul zum anfragenden Prozessor übertragen werden können. Daneben gibt es auch wieder die Latenzzeit für die vergangene Zeit zwischen dem Absetzen einer Speicheranfrage an das Speichersystem und der Antwort durch dieses (*vgl. dazu Kapitel 5.1*). Hier soll nur auf die Bandbreite weiter eingegangen werden und zunächst die

¹⁸Jeder Eintrag im Tupelraum wird als Tupel bezeichnet. Die DSM-Wrapperimplementierung bildet die DSM-Zellen 1 : 1 auf die Tupel ab.

Bandbreiten einiger Hauptspeichermodule betrachtet werden. Im Anschluß daran folgen Untersuchungen zur Bandbreite des DSM-Subsystems.

Die theoretische maximale Bandbreite $\Phi_{mem_{max}}$ kann aus dem Takt der Speichermodule ν , der Bitbreite b der Speicheranbindung und der Anzahl der benutzten Taktflanken k pro Takt bestimmt werden.

$$\Phi_{mem_{max}} = \frac{\nu \cdot b \cdot k}{8 \frac{Bit}{Byte}}$$

Im Falle von *DDR-SDRAM* können 64 Bit gleichzeitig übertragen werden. Die verschiedenen Entwicklungsstufen dieses Speichermodultypes benutzen pro Taktsignal z.T. mehrere Abschnitte¹⁹ des Signales, so daß pro Takt mehrere Übertragungen stattfinden können. Bei *DDR* sind es 2, bei *DDR2* 4 und bei *DDR3* bereits 8 Taktflanken [67, 68]. Für ein *DDR-400* Modul mit 200 MHz ergibt sich somit:

$$\Phi_{mem_{max}} = \frac{\nu \cdot b \cdot k}{8 \frac{Bit}{Byte}} = \frac{200 \cdot 10^6 Hz \cdot 64 Bit \cdot 2}{8 \frac{Bit}{Byte}} = 3200 \cdot 10^6 Byte/s$$

In Tabelle 5.1 sind einige Eckdaten für momentan aktuelle Speichermodule aufgelistet. Im praktischen Anwendungsfall unterscheidet man zusätzlich noch zwischen Lese- und Schreibfragen, wobei im Regelfall ein lesender Zugriff wesentlich schneller ist als ein Schreibender. Die erreichbare Bandbreite des Hauptspeichers hängt noch von vielen weiteren Faktoren, wie z.B. der gesamten Hardwareausstattung²⁰ eines Systemes ab. Oftmals läßt sich deshalb keine genaue Voraussage der zu erwartenden Bandbreite eines Computersystems treffen, wenn nur die verwendeten Speichermodule bekannt sind. In diesem Abschnitt sollen die real erreichbaren Datenraten der verfügbaren DSM-Implementierungen ermitteln werden. Für die Verwaltung dieses verteilten Speichers ist im allgemeinen eine Netzwerkkommunikation notwendig. Deshalb spielen die soeben genannten Faktoren eine eher untergeordnete Rolle, da die Bandbreiten für den Netzwerkverkehr wesentlich geringer sind.

Ausgehend von diesen Überlegungen kann man die maximale Bandbreite des DSM Φ_{DSM} durch die Bandbreite der Kommunikationsschicht Φ_{comm} abschätzen. Dieser Wert bildet gewissermaßen eine untere Schranke, wobei die real erreichbaren Bandbreiten durch den Einsatz von Kohärenzprotokollen und anderen Caching Mechanismen gesteigert werden können. Als oberer Schranke kann die Bandbreite des zugehörigen Hauptspeichers $\Phi_{mem_{max}}$ verwendet werden. Es gilt:

$$\Phi_{comm} \leq \Phi_{DSM} \leq \Phi_{mem_{max}}$$

Legt man die in Kapitel 5.1.3 (*S.142*) ermittelten Bandbreiten für die Kommunikationsschicht und die theoretischen Hauptspeicherbandbreiten²¹ für *DDR2 SDRAM 800* aus Tabelle 5.1 zugrunde, ergibt sich:

$$2,0 MB/s \leq \Phi_{DSM} \leq 6103,5 MB/s \tag{5.1}$$

¹⁹absteigende bzw. aufsteigende Flanken

²⁰Chipsatz des Mainboards

²¹Dieser Wert entspricht dem „schnellsten“ zur Verfügung stehenden Testsystem: pooh.

Chip/Speichermodul	Taktrate ν	max. Übertragungsrate $\Phi_{mem_{max}}$
DDR SDRAM 200	100 MHz	1525,8 MB/s
DDR SDRAM 266	133 MHz	2002,7 MB/s
RDRAM (Rambus)	600 MHz	2288,8 MB/s
DDR SDRAM 400	200 MHz	3051,7 MB/s
DDR2 SDRAM 400	200 MHz	3051,7 MB/s
DDR2 SDRAM 667	333 MHz	5054,4 MB/s
DDR2 SDRAM 800	400 MHz	6103,5 MB/s
DDR2 SDRAM 1066	533 MHz	8106,2 MB/s
DDR3 SDRAM 1600	800 MHz	12207,0 MB/s

Tabelle 5.1: Parameter heute gebräuchlicher Speichermodule, bezogen auf ein Speichermodul. Da die verwendeten Speicherkontroller mehrere solcher Module parallel betreiben können, können auch höhere Raten erzielt werden (Dual Channel). *Quelle: www.jedec.org*

Um die so erhaltene grobe Abschätzung weiter einzugrenzen, sollen deshalb zunächst reale Bandbreiten auf den benutzten Hostsystemen als Ersatz für die zunächst verwendeten theoretischen Hauptspeicherbandbreiten ermittelt werden. Neben den Speichermodulen hat der Aufbau des gesamten Speichersystems²² und das Zugriffsmuster eines Programmes einen ebenfalls nicht unerheblichen Einfluß auf die erreichte Speicherbandbreite. Das dafür anzuwendende Testverfahren soll mittlere Zugriffsraten bestimmen, da maximale Zugriffsraten in der Praxis selten erreicht werden und deshalb für die hier durchgeführten Untersuchungen nicht geeignet sind.

Testverfahren Für die Bestimmung der erforderlichen Zugriffsraten wurde ein kurzer Java-Benchmark erstellt. Dieser Benchmarktest generiert ein Feld aus n `double` Einträgen mit einer Gesamtgröße $size$ und iteriert anschließend mehrmals über das gesamte Feld. Dabei wird auf die Feldelemente in sequentieller Reihenfolge lesend bzw. schreibend zugegriffen. Dieses Vorgehen wurde gewählt, um lokale Speicherzugriffe nachzubilden, die durch das Speichersystem meist schneller beantwortet werden können (Caching). Damit nicht nur Daten aus dem Cache gelesen oder geschrieben werden und des öfteren Cache-Fehlzugriffe bei den Zugriffen auftreten, muß das verwendete Feld größer als die im System verfügbaren Cache-Speicher sein. Diese Größe kann als Parameter an den Test übergeben werden und ist hier auf 100 MB festgesetzt. Um die so erzielten Ergebnisse hinsichtlich ihrer Aussagekraft überprüfen zu können, wurde das Tool `bandwidth`²³ verwendet. Dieses ermittelt auf ähnliche Weise einige Hauptspeicher- und Cachewerte. Im späteren Verlauf wird auf das Vorgehen des hier erstellten **Java-Benchmark-Tests** noch mehrmals Bezug genommen. An diesen Stellen wird dann kurz vom **JB-Test** gesprochen.

²²Controller und Cache

²³siehe auch Kapitel B.2.3 (S.232)

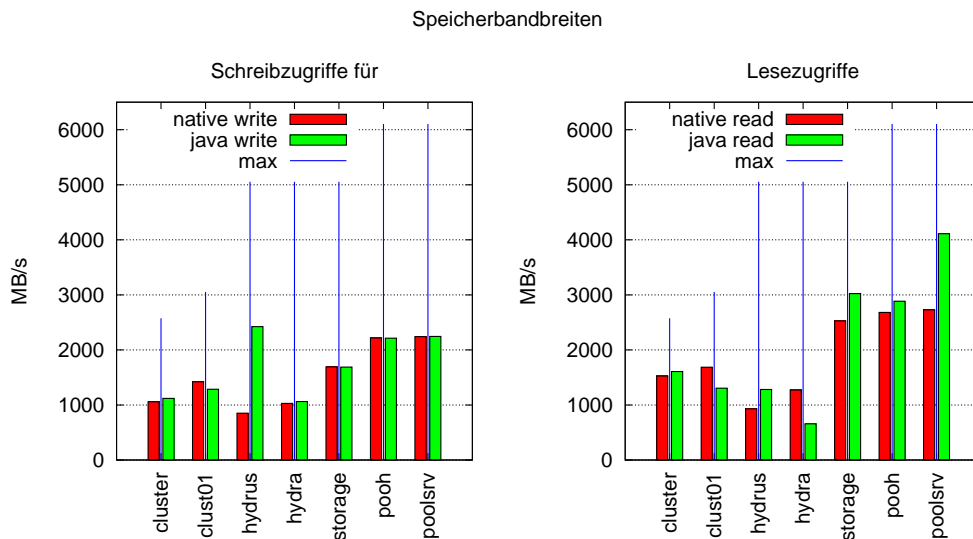


Abbildung 5.13: Ermittelte Hauptspeicherbandbreiten für sequentielle Schreib/Lesezugriffe. Die Größe der benutzten Felder im Java-Benchmark lag jeweils bei 100 MB. Zusätzlich sind noch die theoretischen Maximalwerte (**max**) der jeweiligen Speichermodule aufgetragen.

Ergebnisse In Abbildung 5.13 sind die mit den beschriebenen Testverfahren ermittelten realen Bandbreiten Φ_{mem} dargestellt. Bis auf wenige Ausnahmen stimmen die Größenordnungen der Meßwerte des **bandwidth** Tools und des Java-Benchmarks (*JB-Test*) für die durchgeführten Schreibzugriffe überein. Erkennbar ist dort eine etwas höhere Bandbreite der JB-Laufzeittests. Gründe hierfür können Optimierungen durch Blockzugriffe und Caching der geschriebenen Werte durch die virtuelle Maschine sein. Möglichkeiten zur Verhinderung dieser Effekte sind nicht vorhanden, da diese im Normalfall erwünscht sind.

Die ermittelten Leseraten weichen deutlicher voneinander ab, es können aber ähnliche Schlußfolgerungen gezogen werden. Insgesamt wird deutlich, daß in den praktischen Tests kaum mehr als 50% der theoretisch erreichbaren Bandbreite erzielt werden konnte. Die bisherige Abschätzung (5.1) läßt sich aber für die eingesetzten Testsysteme weiter auf

$$2 MB/s \leq \Phi_{DSM} \leq 4100 MB/s$$

verfeinern. Hierbei wurde die maximal erzielte Leseratte (Abbildung 5.13) als obere Grenze gewählt.

DSM-Bandbreite

Zur weiteren Eingrenzung von Φ_{DSM} werden im folgenden einige Bandbreitentests für das DSM-Subsystem durchgeführt. Diese gliedern sich in die Bestimmung von lokalen Bandbreiten (ohne Netzwerkkommunikation) und in verteilte Bandbreitentest.

Beschreibung des Verfahrens Das für die Bestimmung der DSM-Bandbreiten eingesetzte Verfahren ist mit dem Vorgehen im vorangegangenen Java-Benchmark (*JB-Test*) vergleichbar. An dieser Stelle soll das Verfahren (Listing 5.4) kurz beschrieben und auf einige Unterschiede zum Java-Benchmark hingewiesen werden.

Statt eines Feldes werden von diesem Test nun einzelne DSM-Zellen mit `double` Werten verwendet. Das Erzeugen (*Zeile 1*) dieser Speicherzellen im DSM erledigt der Trail `master`, welcher sich immer auf dem Hostsystem des DSM-Servers befindet.²⁴ Die eigentliche Messung gliedert sich in drei Phasen und wird vom Trail `tester` durchgeführt.

Dazu wird zuerst ein *WarmUp* durchgeführt, d.h. jede Zelle wird einmal gelesen. Durch diesen Vorgang ist sichergestellt, daß Verwaltungseinträge für alle Zellen existieren und nicht erst angelegt werden müssen. Dies ist für die Fälle von Interesse, bei denen `master` und `tester` nicht auf dem gleichen Hostsystem ausgeführt werden. Die im Rahmen dieser Arbeit erstellten *Central*- und *Spread*-DSM-Server halten auf jedem Hostsystem ein Verzeichnis mit Verwaltungsinformationen zu allen angefragten Zellen. Im Falle eines Zugriffs existieren dort Informationen, die das Auffinden erleichtern.

```

1   if ( me == master )
2   {
3       dsmCreateCells () ;
4   }
5
6   // ausgewählter Trail
7   if (me == tester)
8   {
9       // 1.Phase: WarmUp, alle Zellen 1x lesen
10      readAllCells () ;
11
12      // 2.Phase: mehrere Lesezyklen
13      readTimer.start () ;
14      for ( int t = 0 ; t < loops ; t++)
15      {
16          readAllCells () ;
17      }
18      readTimer.stop () ;
19
20      //3.Phase: mehrere Schreibzyklen
21      writeTimer.start () ;
22      for ( int t = 0 ; t < loops ; t++)
23      {
24          writeAllCells () ;
25      }
26      writeTimer.stop () ;
27  }
```

Listing 5.4: DSM-Bandbreitentest

²⁴gemeint ist hier der Rootserver, bei dem neue Anfragen eingehen.

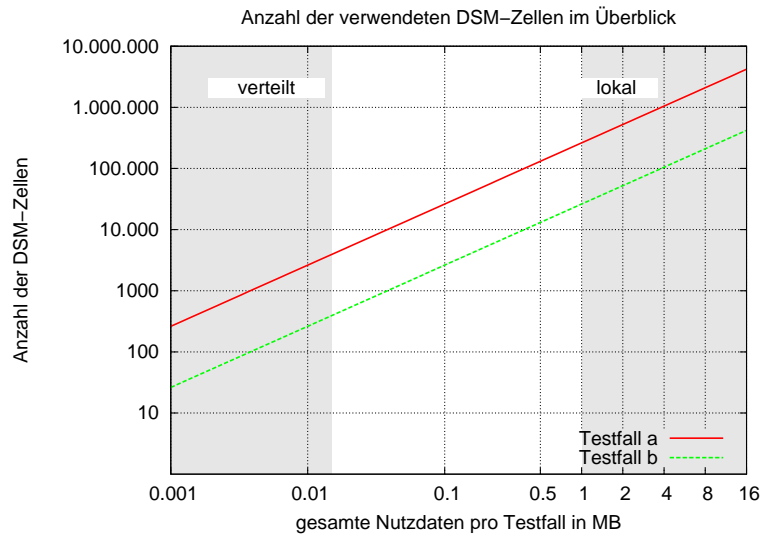


Abbildung 5.14: Anzahl der verwendeten DSM-Zellen und der Speicherbedarf der reinen Nutzdaten für verschiedene Testfälle. Die mit *verteilt* und *lokal* gekennzeichneten Abschnitte sind die benutzten Wertebereiche für die entsprechenden Tests.

Für die jeweilige Bestimmung der Lese- und Schreibraten in den Phasen 2 und 3 wird mehrmals über alle Zellen iteriert. Um ungewollte Nebeneffekte minimieren zu können, wird nur das *Server-Level* des DSM-Subsystems verwendet und die *Protocol-* und *User-Level* werden umgangen. Insbesondere wird für die Schreibzugriffe kein *Locking* verwendet.

Die durchgeführten Test lassen sich durch die Wahl des Trails `tester` leicht in zwei Gruppen aufteilen: *lokal* und *verteilt*.

Testfälle Nachdem das Verfahren zur Bestimmung der DSM-Bandbreiten erläutert wurde, werden an dieser Stelle die untersuchten Testfälle beschrieben. Wie im Falle des *JB-Tests*, zur Bestimmung der Hauptspeicherbandbreiten, sollten die in jedem Lese/Schreibvorgang adressierten (Nutz-)Daten nur einige wenige Byte groß sein²⁵. Die Gesamtgröße des verwendeten Feldes betrug beim *JB-Test* einige *MB*, so daß eine hohe Anzahl von einzelnen Einträgen vorhanden war. Bei den DSM-Bandbreitentests werden die Daten in einzelnen DSM-Zellen abgelegt. Dafür wurden zwei Testfälle **a** und **b** mit jeweils unterschiedlicher Nutzdatengröße in den DSM-Zellen generiert. Im ersten *Testfall a* enthalten die zu erzeugenden Zellen einen einzelnen Integerwert aus 4 Byte. Der zweite *Testfall b* erzeugt jeweils Zellen mit 40 Byte an Nutzdaten bzw. 10 Integerwerten. Die insgesamt durch die Tests verwendete (Nutz-)Datenmenge sollte wieder einige *MB* betragen, damit die im System vorhandenen Cache-Speicher die Messungen nicht zu stark beeinflussen. Durch die Wahl der Anzahl der zu erzeugenden DSM-Zellen für die Laufzeittests kann diese Datenmenge beeinflusst werden. In Abbildung 5.14 sind die

²⁵Beim *JB-Test* wurden `double` Werte, also jeweils 8 Byte gelesen.

benötigten DSM-Zellen der Testfälle *a* und *b* für verschiedene Nutzdaten dargestellt. Der gesamte Speicherbedarf des DSM-Subsystem wird im späteren Verlauf noch detaillierter betrachtet und beschrieben.

In der Abbildung 5.14 sind weiterhin die verwendeten Wertebereiche der Nutzdaten für die *lokalen* und *verteilten* Varianten der durchzuführenden Laufzeittests skizziert. Eine Aufteilung in *lokal* und *verteilt* ist notwendig, um aussagekräftige Meßergebnisse zu erhalten. Da die benötigte Zeit für die Erzeugung einer DSM-Zelle im verteilten Fall stark ansteigt (vgl. dazu Kapitel 5.2.1 (S.147)), können in vertretbarer Zeit nur relativ wenige DSM-Zelle (≈ 5000) betrachtet werden. Im *lokalen* Testfall sollten wiederum Nutzdatenumenge von > 1 MB betrachtet werden. Dies erfordert die Benutzung von mehr als 10.000 DSM-Zellen.

Lokale Bandbreite Als *lokale Bandbreite* werden hier die Zugriffe auf das DSM-System verstanden, in denen nur ein Hostsystem involviert ist und deshalb keine Netzwerkkommunikation mit DSM-Komponenten auf anderen Hostsystemen notwendig wird. Im bereits beschriebenen Verfahren (Listing 5.4) führt deshalb der Mastertrail alle Zugriffe aus (`master == tester`). In den beiden Testfällen *a* und *b* ist keine Kommunikation mit anderen Hostsystemen notwendig und die Meßwerte der Bandbreiten sollten über dem in Kapitel 5.2.3 (S.153) ermittelten Wert Φ_{comm} liegen.

Die Anzahl der Nutzdaten wird für die durchzuführenden Messungen auf Werte zwischen 1MB und 16MB festgelegt. Im Testfall *b* müssen deshalb mindestens 26.000 Datenzellen im DSM erzeugt werden (siehe Abbildung 5.14). Für den Testfall *a* werden für 16MB Nutzdaten mehr als 4.000.000 DSM-Zellen erzeugt. Geht man von einem realen Speicherbedarf von etwa 100Byte pro DSM-Zelle aus, so ergeben sich ≥ 380 MB Speicherbedarf für diesen Test.

Bei der Durchführung der Tests erwies sich der bereits erwähnte hohe Zeitbedarf der *TSpaces*-Variante zur Erzeugung einer DSM-Zelle als erschwerendes Element, weshalb die praktische Durchführung der vorgesehenen Tests nahezu unmöglich wurde. In diesen Testfällen war die resultierende Laufzeit für die Erzeugung von mindestens 26.000 Zellen im *TSpaces*-DSM nicht mehr akzeptabel.²⁶ Deshalb sind in den Abbildungen 5.15 und 5.16 nur die Bandbreiten für *Central*- und *Spread*-DSM aufgetragen. Eine Untersuchung des *TSpaces*-DSM erfolgt gesondert im Abschnitt 5.2.3.

In den Abbildungen 5.15 und 5.16 läßt sich eine bessere Performance der Beantwortung von Anfragen an den *Central*-DSM feststellen. Dieser erzeugt im Vergleich zum *Spread*-DSM weniger Verwaltungsoverhead, da die verwendeten Datenstrukturen kleiner sind und weniger Systemnachrichten, im hier betrachteten lokalen Fall, generiert werden.

Bei den Lesezugriffen erkennt man für beide DSM-Server relativ konstante Raten, während die ermittelten Schreibraten kontinuierlich mit steigender Nutzdatengröße sinken. In beiden Fällen läßt sich das Absinken der Bandbreiten bei der Erhöhung der im Test verwendeten Nutzdaten, auf die Anzahl der verwalteten DSM-Zellen, den allokierten Hauptspeicher der JVM und den *Garbage Collector* zurückführen. Für die problemlose

²⁶siehe auch Abbildung 5.9

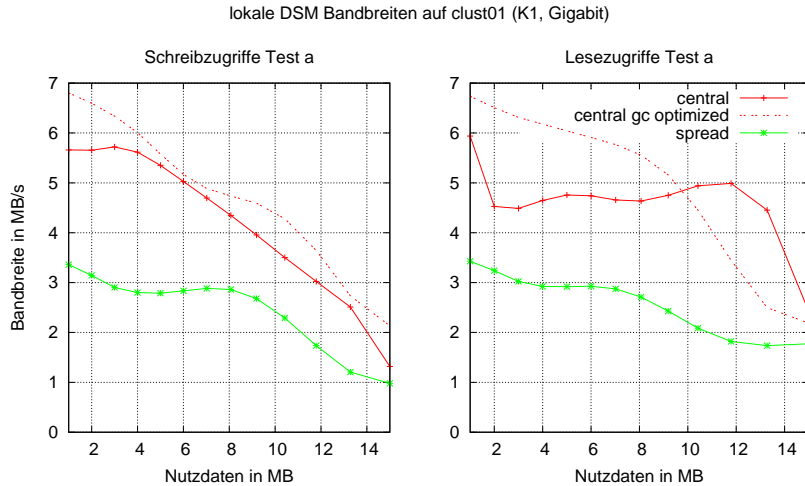


Abbildung 5.15: Ermittelte Bandbreiten von lokalen Zugriffen (*master = tester*) für den *Testfall a* (getrennt nach Schreib- und Leseanfragen). Die Datenzellen enthielten einzelne *int* Werte. Zusätzlich ist noch eine Meßreihe für Tests mit erweiterten *Garbage Collector* Parametern enthalten (*central gc optimized*).

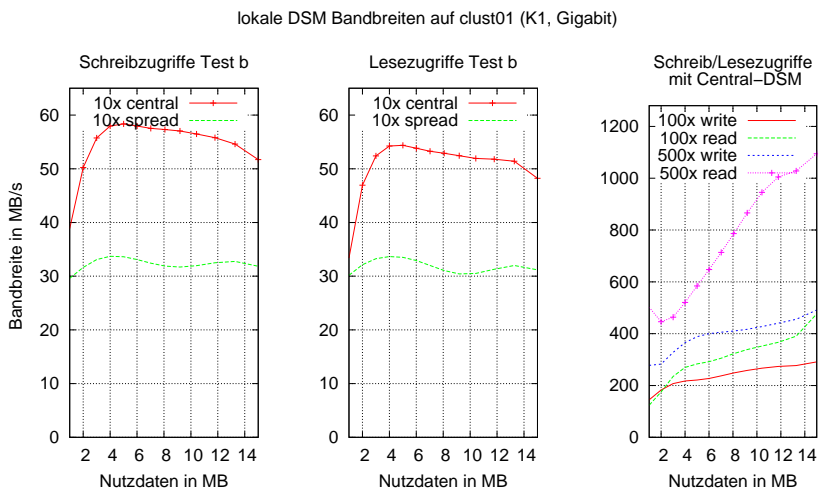


Abbildung 5.16: Ermittelte Bandbreiten von lokalen Zugriffen für den *Testfall b* (10x) mit 10 Integerwerten pro Zelle. Zusätzlich sind noch die Schreibraten von einigen durchgeführten Testläufen mit einer weiteren Anhebung der Nutzdaten in den Zellen dargestellt. Die Anzahl der pro Zelle enthaltenen Nutzdaten kann wie folgt ermittelt werden: 500x steht für 500 Integerwerte, d.h. $500 \cdot 4 = 2000$ Byte.

Durchführung der vorgesehenen Tests wurden der Java-Umgebung generell 2 GB Arbeitsspeicher auf allen beteiligten Hostsystemen zur Verfügung gestellt. Bei kleineren Werten wurde der *Garbage Collector* sehr oft aufgerufen und konnte zum Teil die vorgesehene Speicherbereinigung nicht vollständig durchführen. In diesen Fällen brachte der *Garbage Collector* die gesamte Laufzeitumgebung zum Stillstand²⁷. Auf den Einfluß des *Garbage Collectors* wird auch im nachfolgenden Abschnitt nochmals eingegangen.

Bei einer Erhöhung der in den Zellen enthaltenen Nutzdaten (Testfall *b*) steigt die resultierende Bandbreite (Abbildung 5.16). Da für die gleiche Menge an Nutzdaten dann weniger DSM-Zugriffe notwendig sind, ist dieses Verhalten zu erwarten. Eine weitere Erhöhung der Nutzdaten in einer Zelle führt zu einem weiteren Anstieg der Bandbreite. Aus den ermittelten Bandbreiten für die Schreibzugriffe läßt sich Ähnliches ableiten. Weil sich diese Zugriffe aus einem Lesezugriff für die Verwaltungsinformationen der DSM-Zelle und dem Schreiben der Daten zusammensetzen, entsteht ein etwas größerer Aufwand. Die Schreibraten sind daher meist geringer als die Leseraten. Außerdem erzeugt jeder Schreibvorgang ein neues Objekt, ohne daß der belegte Speicherbereich der alten Daten sofort wieder frei gegeben wird. Mit zunehmender Laufzeit muß die Java-Umgebung deshalb den allokierten Speicher erhöhen oder einen *Garbage Collector* Lauf durchführen. Diese Maßnahmen nehmen einen nicht unerheblichen Anteil der gemessenen Zeiten ein, welche in die Berechnung der Bandbreite einfließen. Zusätzlich erhöht sich der Einfluß dieses Effektes mit steigender Datenmenge im DSM.

Speicherbedarf In diesem Abschnitt soll der Datenoverhead des DSM-Subsystems, d.h. die Größe der Verwaltungsinformationen einer DSM-Zelle, ermittelt werden. Prinzipiell läßt sich dieser Overhead aus einer Analyse des Quellcodes des eingesetzten *Storage-Module* bestimmen. Für das untersuchte *Beluga* Laufzeitsystem ergeben sich zwei Einschränkungen:

1. Im Fall der verwendeten *TSpaces*-Bibliothek ist der Quellcode nicht frei zugänglich.
2. Der reale Speicherbedarf von Objekten kann in jeder virtuellen Java-Maschine unterschiedlich ausfallen und ist von der Version bzw. Implementierung der JVM abhängig.

Daher wird an dieser Stelle eine experimentelle Bestimmung des Datenoverheads vorgenommen und gleichzeitig der Einfluß des *Garbage Collectors* bei intensiver Speichernutzung untersucht.

Zur Bestimmung des entstehenden Datenoverhead, welcher durch die Abspeicherung der Nutzdaten in den DSM-Zellen entsteht, werden die bisherigen Bandbreitentests verwendet und der hierfür von der JVM belegte Speicher ermittelt. Aus den gewonnenen Werten kann der Speicherbedarf für eine DSM-Zelle berechnet werden.

²⁷Da die Objekte zwischen den einzelnen Generationen der Speicherverwaltung kopiert werden, muß zusätzlicher Speicher bereit stehen (*siehe Kapitel 2.4.3 (S.53)*)

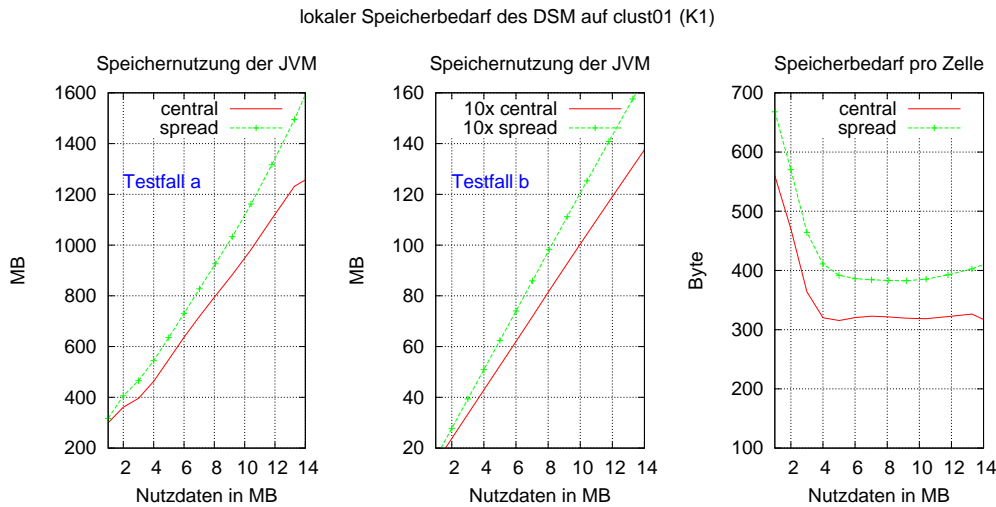


Abbildung 5.17: Die Speicherbelegung der JVM und der gemittelte Speicherbedarf pro Zelle für die besprochenen Testfälle *a* und *b*.

In Abbildung 5.17 sind der gesamte Speicherbedarf für die einzelnen Bandbreitentests und die daraus ermittelten Zellengrößen dargestellt. Da für die Speicherung eines Integerwertes 4 Byte verwendet werden, die Tests aber einige *MB* an Nutzdaten generieren, ist die Anzahl der verwendeten Zellen sehr hoch. Hieraus resultiert ein hoher Speicherbedarf²⁸ der virtuellen Maschine (JVM). Pro DSM-Zelle liegt der tatsächliche Speicherbedarf etwa zwischen 300 und 400 Byte. Diese Werte lassen sich aus den Beträgen für den belegten Speicher der JVM vor und nach dem Anlegen der Zellen ermitteln. In den Meßwerten spiegelt sich allerdings der gesamte von der virtuelle Maschine als belegt gekennzeichnete Speicher wieder. Enthalten sind deshalb auch Anteile, die durch den *Garbage Collector* noch nicht wieder frei gegeben wurden²⁹ (siehe Kapitel 2.4.3).

Um den Einfluß des *Garbage Collectors* auf alle durchgeführten Tests soweit wie möglich einzuschränken, wurde der JVM beim Start immer eine ausreichende Menge³⁰ an Hauptspeicher zur Verfügung gestellt und per Kommandozeilenparameter aufgefordert, diesen Speicher sofort beim Start zu allokiieren. Durch dieses Vorgehen wird die dynamische Erhöhung des für ein Java-Programm zur Verfügung stehenden Hauptspeichers verhindert und es kommt zu keinen Unterbrechungen während der Programmausführung aufgrund von Speichermangel.

Die Java-Referenzimplementierung von Sun bietet zusätzlich noch einige Möglichkeiten zur Anpassung der benutzten Algorithmen und Strategien für die Speicherbereinigung.

²⁸im Vergleich zur gespeicherten Nutzdatenmenge

²⁹Vor dem Auslesen der Speicherwerte wurden mehrere *Garbage Collector* Läufe mit einem expliziten Aufruf von `System.gc()` angefordert. Im Speicher der JVM befinden sich auch Klassen und Objekte die vom Laufzeitsystem benutzt und deshalb durch die Speicherbereinigung nicht entfernt werden können.

³⁰von mindestens 2 GB

Diese Möglichkeiten variieren und sind abhängig von der Version der eingesetzten virtuellen Maschine³¹. Exemplarisch für diese Form der Optimierung wurden einige Testläufe mit den Parametern `-XX:-UseParallelGC -XX:+AggressiveHeap` durchgeführt. Durch diese Parameter werden vom *Garbage Collector* mehrere Threads eingesetzt und es wird versucht, soviel Hauptspeicher wie möglich für die JVM zu verwenden[119, 120].

Der Einfluß dieser Parameter auf das Laufzeitverhalten ist in der Abbildung 5.15 als `central gc optimized` für die bereits besprochenen lokalen Bandbreitentest dargestellt. Dort zeigt sich eine leichte Steigerung für die ermittelten Schreibraten. Für die Leserate läßt sich nur eine Verbesserung bis etwa 10 MB Nutzdaten erkennen. Hieraus läßt sich ableiten, daß der *Garbage Collector* und der zur Verfügung stehende Hauptspeicher einen wesentlichen Einfluß auf das Laufzeitverhalten von speicherintensiven Java-Anwendungen haben. Die Kombination verschiedener JVM-Parameter kann zu einer Reduzierung der Laufzeit eines konkreten Anwendungsprogrammes führen.

Verteilte Bandbreite Neben der soeben besprochenen *lokalen Bandbreite* ist die Untersuchung der resultierenden Bandbreiten einer verteilte Ausführung ein weiterer Punkt bei der Analyse des allgemeinen Verhaltens der vorhandenen DSM-Server. Für diese Fälle wurde die Laufzeitumgebung so konfiguriert, daß die Trails `master` und `tester` (siehe Listing 5.4) auf unterschiedlichen Hostsystemen ausgeführt wurden. Hierdurch wird die Benutzung der Kommunikationsschnittstellen erzwungen und die resultierenden Bandbreiten sollten deutlich abfallen. Anhand der Erkenntnisse aus Kapitel 5.2.1 (*S.147*) wird die Anzahl der zu erzeugenden Zellen im DSM nun deutlich verringert, da die Gesamtlaufzeiten der Tests sonst sehr stark ansteigt. Zur Sicherstellung der Vergleichbarkeit mit den vorangegangenen lokalen Tests bleiben aber die in den Zellen enthaltenen Nutzdaten konstant. Als Folge dessen reduziert sich die Gesamtzahl der im DSM abgelegten Nutzdaten erheblich (siehe Abbildung 5.14) und liegt nun zwischen 1KB und 16KB bei maximal 4000 zu erzeugenden Zellen.

Die Ergebnisse für den Testfall *a* sind in der Abbildung 5.18 dargestellt. Hier zeigt sich ein deutlicher Abfall der gemessenen Raten gegenüber den lokalen Tests. Im Fall des lesenden Zugriffs können nur noch einige Bytes an Nutzdaten pro Sekunde übertragen werden. Die Schreibraten sind allerdings erheblich höher. Dies liegt vor allem am nicht blockierenden Charakter der Schreiboperationen, d.h. die Kontrolle kehrt an den Aufrufer zurück, sobald die Operationen ans DSM-System abgesetzt wurden.

Wird die Datenmenge in den einzelnen DSM-Zelle erhöht (Testfall *b*) steigen die resultierenden Bandbreiten für die Lese- und Schreibzugriffe des Testprogrammes (Abbildung 5.19). Durch das vom *Spread-DSM* verwendete Auslagerungsverfahren und der damit verbundenen Verminderung der Kommunikation, weichen die Leserate zwischen *Central-* und *Spread-DSM* für Nutzdaten ≥ 6 KB kaum voneinander ab. Der notwendige Mehraufwand der Verwaltung im *Spread-DSM* wird in diesen Fällen durch eine schnellere Zugriffscharakteristik ausgeglichen.

³¹Ein Überblick findet sich z.B. auf

<http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp>

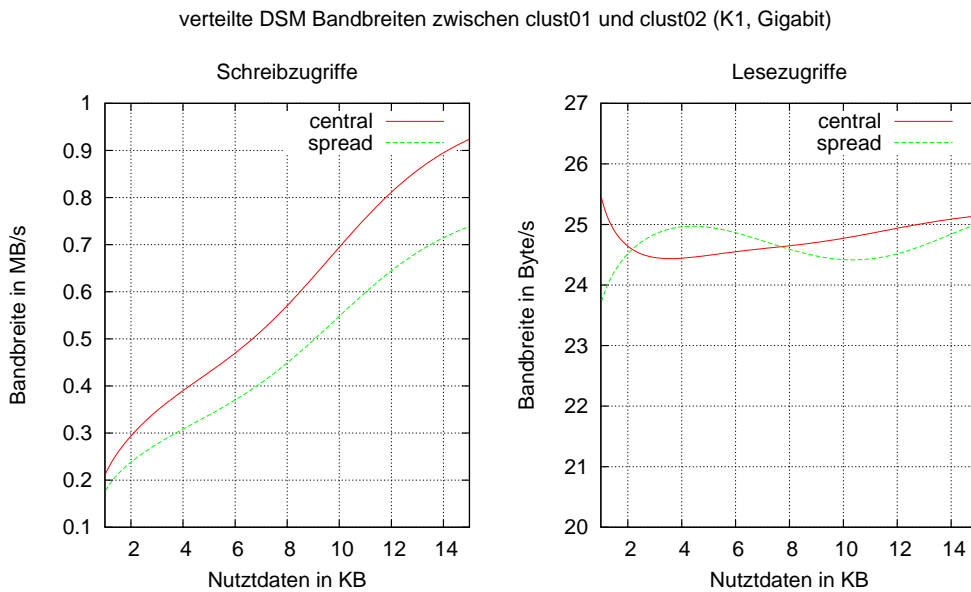


Abbildung 5.18: Ermittelte Bandbreiten für verteilte Zugriffe auf Datenzellen mit einzelnen int Werten für *Testfall a*. Die DSM-Zellen wurden jeweils auf clust01 angelegt und auf clust02 bearbeitet.

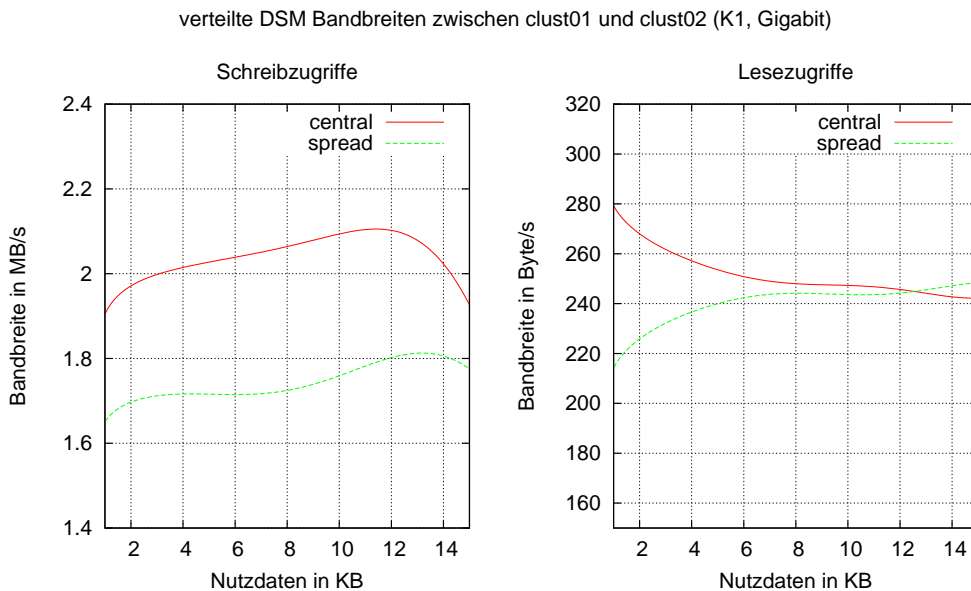


Abbildung 5.19: Ermittelte Bandbreiten für verteilte Zugriffe auf Datenzellen für den *Testfall b*.

TSpaces Der *TSpaces*-DSM wurde getrennt untersucht. Die Ergebnisse der durchgeführten Laufzeittests sind in der Abbildung 5.20 dargestellt. Gravierende Unterschiede zwischen den ermittelten lokalen und verteilten Bandbreiten lassen sich nicht feststellen. Eine Erhöhung der Nutzdaten (*Testfall b*) resultiert, wie bei den Tests mit der *Central*- und der *Spread*-Variante, in einer Steigerung der erzielten Transferraten. Auffallend ist, daß im Gegensatz zu den bisherigen Tests mit *Central*- und *Spread*-DSM, kein nennenswerter Unterschied zwischen lesenden und schreibenden Zugriffen zu erkennen ist. Verteilte Lesezugriffe können durch den *TSpaces*-DSM mit guter Performance durchgeführt werden. Bei Schreibzugriffen schneiden die *Central*- und *Spread*-DSM-Implementierungen mit einer höheren Bandbreite etwas besser ab.

Die Speichernutzung und der Speicherbedarf pro DSM-Zelle sind in der Abbildung 5.21 dargestellt. Der belegte Speicher wächst erwartungsgemäß linear mit der Anzahl der Zellen. Für die Ablage einer Zelle im DSM verwendet die Implementierung ca. 1100 Bytes, dies ist etwa das Dreifache der *Central*-Implementierung.

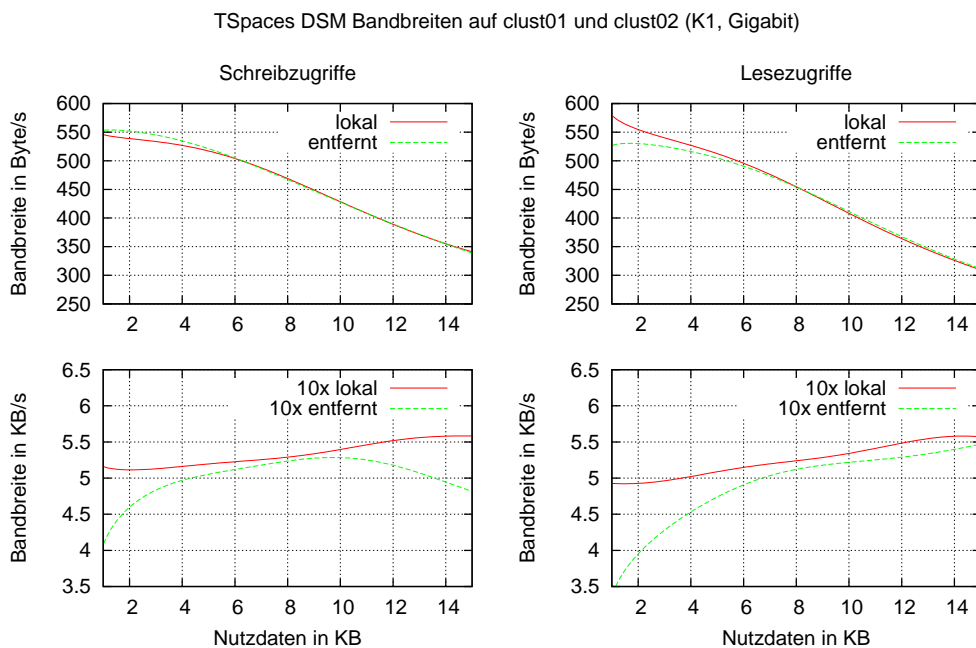


Abbildung 5.20: Ergebnisse der Bandbreitentests für die besprochenen *TSpaces*-Testfälle (jeweils lokal und entfernt).

DSM-Bandbreite und Locking

Bei den bisherigen Tests wurde der notwendige Aufwand für die Schutzmechanismen zum gegenseitigen Ausschluß (*Locking*) beim Schreiben nicht berücksichtigt. Deshalb wurde das bisherige Testverfahren um Lock-Operationen für die Schreibzugriffe erweitert und einige der Laufzeittests wiederholt.

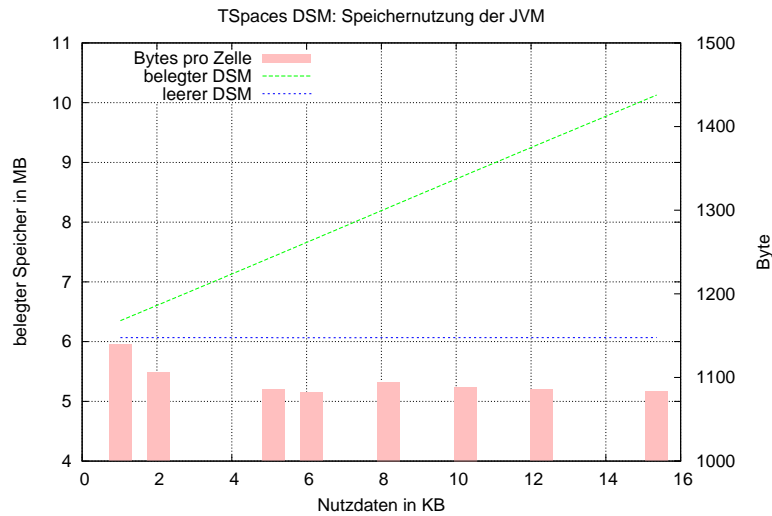


Abbildung 5.21: Speicherbelegung und rechnerisch ermittelte Zellgrößen des *TSpaces*-DSM für den *Testfall a*.

Zum besserem Verständnis ist der Ablauf dieser Tests kurz als Pseudocode im nachfolgenden Listing 5.5 dargestellt. Es wird hierbei vorausgesetzt, daß die Daten bereits im DSM erzeugt wurden.

```

1 // Warmup Phase
2 for(int t = 0 ; t < MIN_REQUESTS ; t++)
3 {
4     forAll cells
5     {
6         lock(cell) ;
7         cell.write() ;
8         unlock(cell) ;
9     }
10 }
11
12 // Work Phase
13 for(int t = 0 ; t < maxTests ; t++)
14 {
15     forAll cells
16     {
17         lock(cell) ;
18         cell.write() ;
19         unlock(cell) ;
20     }
21 }
    
```

Listing 5.5: Pseudocode des DSM-Bandbreitentests mit aktiven *Write Locks*

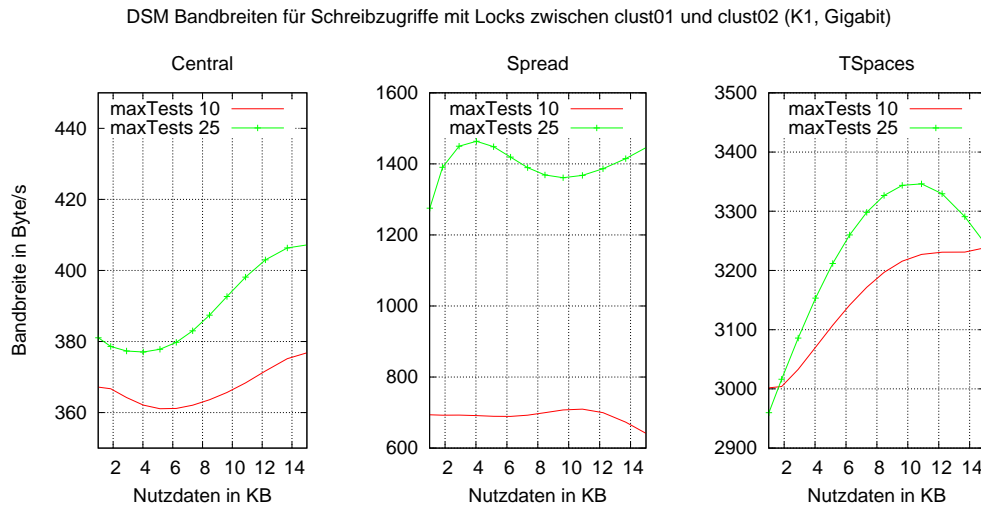


Abbildung 5.22: Ermittelte Gesamtraten für die untersuchten DSM-Implementierungen mit jeweils 4 *Warmup* und 10 bzw. 25 *Work* Schreibzugriffen. Je höher die Anzahl der Durchläufe (*maxTests*) in der *Work* Phase, desto höher wird der Einfluß der Auslagerung von Zellen beim *Spread*-DSM.

Der auf Listing 5.5 basierende Laufzeittest gliedert sich in eine *Warmup Phase* und eine *Work Phase*. In der *Warmup Phase* sollten durch das im *Spread*-DSM eingesetzte Verfahren alle geschriebenen DSM-Zellen verschoben und dadurch die Zugriffe in der *Work Phase* erheblich beschleunigt werden. Wird keine Optimierung durch die *Warmup Phase* erreicht, sollten die Werte von *Warmup* und *Work* nahezu identisch sein. Die Anzahl der Durchläufe in der *Warmup Phase* (Zeile 2) basiert auf der Konstante `MIN_REQUESTS` des *Spread*-DSM-Servers³².

Die Ergebnisse der Laufzeittests sind in Abbildung 5.22 dargestellt. Durch die Benutzung der *Locks* und dem damit verbundenen erhöhtem Aufwand sinken die Bandbreiten der untersuchten Schreibzugriffe gegenüber den in den früheren Tests ermittelten Werten (siehe z.B. Abbildung 5.19) erheblich. Erkennbar ist bereits jetzt das bessere Abschneiden des *Spread*-DSM, der bisher immer aufgrund eines höheren Overheads bei der Verwaltung der Zellen eine geringere Rate als der *Central*-DSM erzielte. Noch deutlicher wird dieser Unterschied bei der Gegenüberstellung der gemessenen Schreibraten in der *Work Phase* (Abbildung 5.23). Insbesondere sind die dort dargestellten und in den Laufzeittests ermittelten Bandbreiten des *Spread*-DSM höher als die des *TSpaces*-DSM.

5.2.4 Zusammenfassende Auswertung

Die Gegenüberstellung der vorhandenen DSM-Subsysteme hat gezeigt, daß die Performance stark von der Art und Weise der Zugriffe auf den DSM abhängt. Da die Speicherzugriffszeiten bzw. Bandbreiten eines DSM wesentlich geringer als die von herkömmlichen

³² siehe Kapitel 4.2.2 (S.107)

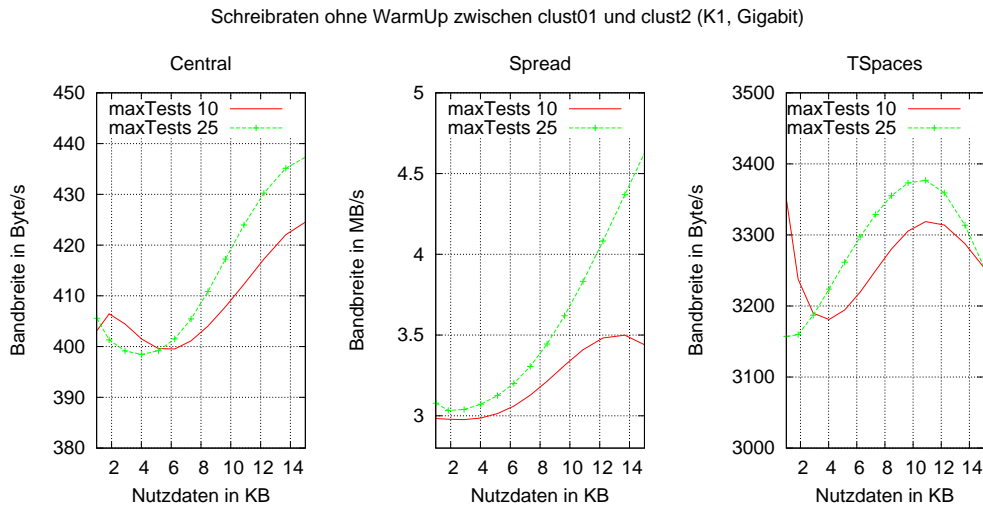


Abbildung 5.23: Ermittelte Werte für die Schreibzugriffe (10 bzw. 25) in der *Work* Phase (ohne *Warmup*).

Speichersystemen sind, hat die Reduzierung der Anzahl von Zugriffen oberste Priorität. Aufgrund der ermittelten Faktoren kann man sagen, daß sogar auf den ersten Blick eher ineffiziente Methoden, wie z.B. die Durchführung von redundanten Berechnungen zur Vermeidung von DSM-Zugriffen, Laufzeitvorteile erzielen können.

Insgesamt ist mit einer signifikanten Erhöhung der Programmlaufzeiten zu rechnen, wenn das auszuführende Programm eine größere Anzahl von Daten im DSM für die Berechnungen hält und hierauf viele Zugriffe erfolgen. Das genaue Verhalten ist jedoch von der Anzahl bzw. der Art der Zugriffe abhängig und läßt sich deshalb nur schwer verallgemeinern.

Bei einem Vergleich der einzelnen Implementierungen untereinander läßt sich feststellen, daß die *TSpaces*-DSM-Variante gute Ergebnisse erzielt. Allerdings ist die benötigte Zeit für die Erzeugung vom Daten im DSM nicht akzeptabel. Dies ist auf die wahrscheinlich erfolgte Replikation der Daten des Tupelraumes auf allen Hostsystemen zurückzuführen. Bezogen auf die Anzahl der verwaltbaren Einträge haben die *Central*- und *Spread*-Implementierungen gezeigt, daß mit ihnen weit über eine Million Einträge verwaltbar sind. Für die *TSpaces*-Implementierung konnten derartige Untersuchungen mit einer so hohen Zahl an DSM-Zellen nicht durchgeführt werden, da dies durch die hohen Laufzeiten der praktischen Testanwendungen verhindert wurde.

5.3 Laufzeitsystem

In diesem Abschnitt werden einige Untersuchungen beschrieben, welche das Verhalten der Laufzeitumgebung mit realen Anwendungen dokumentieren. Umfangreichere Laufzeitbetrachtungen der Prototypimplementierung *Beluga* mit größeren Simulationsanwendungen erfolgen in den Kapiteln 7 und 8.

5.3.1 Matrixmultiplikation

Für die verteilte Berechnung einer Matrixmultiplikation

$$A \cdot B = C \quad \text{mit } c_{ij} = \sum_{k=1}^m a_{ik} \cdot b_{kj}$$

wurden zwei Varianten implementiert. Die erste Variante setzt die Multiplikation einer Matrix mit Hilfe des expliziten Nachrichtenaustausches zwischen allen Trails um, während die zweite Variante auf dem vorhandenen DSM aufbaut. Beide Implementierungen verwenden einen *Master-Slave* Ansatz, bei welchem der *Master-Trail* alle benötigten Daten (Matrizen) anlegt, die Verteilung einzelner Berechnungsstücke für die Trails initiiert und im Anschluß an die Berechnungen das Ergebnis zusammenfügt. Hier sollen zunächst die beiden Varianten in einer homogenen Systemkonfiguration (identische Cluster-Knoten) untersucht werden. Im Anschluß hieran wird die nachrichtenbasierte Variante in einer heterogenen Systemkonfiguration untersucht.

Nachrichtenbasierte Variante

Bei der nachrichtenbasierten Variante versendet der *Master-Trail* zunächst die komplette Matrix B . Von der Matrix A werden jeweils nur einige Zeilen an die Trails verteilt³³. Danach erfolgt die parallele Berechnung des Ergebnisses, indem jeder Trail nur eine Teilmenge von Zeilen aus C bestimmt. Nach Beendigung dieses Schrittes werden die Teilergebnisse vom *Master-Trail* eingesammelt und zu einer Gesamtmatrix C zusammengesetzt.

Abbildung 5.24 zeigt einige Ergebnisse von durchgeführten Laufzeittests der nachrichtenbasierten Variante mit 4, 8 und 16 Trails. Eine Erhöhung der Trailanzahl durch den Einsatz weiterer Hostsysteme (Abbildung 5.24a) resultiert in einer deutlichen Steigerung der erzielten *MFlops*. Die Benutzung zusätzlicher Trails pro Host (Abbildung 5.24b) liefert nur bis zu einem gewissen Punkt einen Zuwachs, da sich dann mehrere lokale Trails einen Prozessor teilen müssen.

DSM-Variante

Die Testimplementierung mit DSM-Nutzung verwendet **eine** globale Speicherzelle für die Bereitstellung der Matrizen A und B . Da keine Schreiboperationen auf diese Speicherzelle durchgeführt werden, müssen diese Daten nicht bei jedem Zugriff erneuert werden und können stattdessen im Cache des DSM-Subsystems zwischengespeichert werden. Zur Speicherung der Teilergebnisse wird eine separate DSM-Zelle für die gesamte Matrix C benutzt, wobei jeder Trail die durch ihn berechneten Elemente ergänzt.

Neben der Möglichkeit der effizienten Umsetzung enthalten die Zielsetzungen im *DEE* Entwurf auch eine für den Anwender leicht zu handhabende Umgebung. Genau dies

³³Eine weitere blockweise Aufteilung der Matrizen erfolgt hier nicht. Für eine Reduzierung des notwendigen Datenvolumens der Kommunikation könnte die Matrix B ebenfalls in (Spalten-)Blöcke zerlegt werden. Hierauf wurde aufgrund einer besseren Vergleichbarkeit mit der DSM-Variante verzichtet.

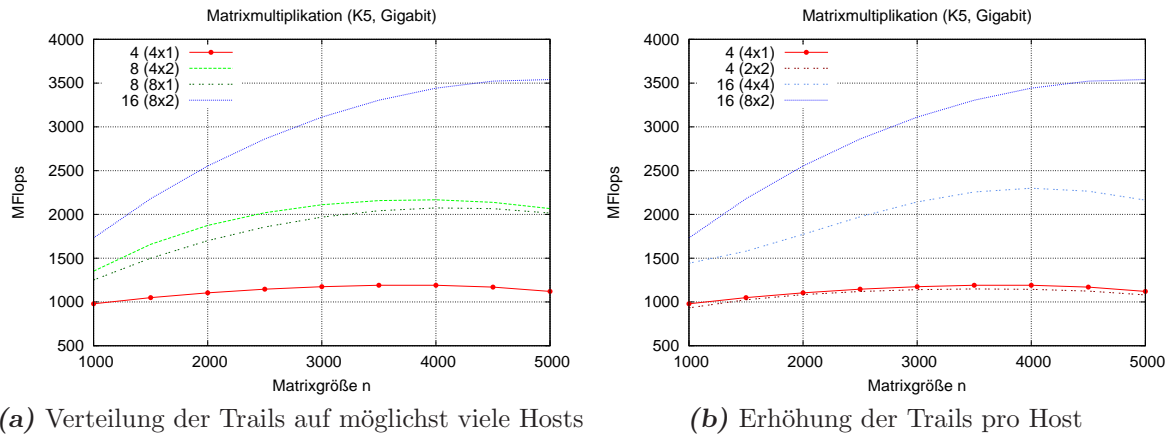


Abbildung 5.24: Testergebnisse für die beschriebene Variante einer verteilten Matrixmultiplikation durch Nachrichtenaustausch. Die verwendeten Knoten der Konfiguration K5 hatten jeweils 2 Prozessoren. Die Angaben in der Legende sind jeweils als Trails (Hostsysteme \times lokale Trails) zu interpretieren. 16(4x4) steht z.B. für 16 gestartete Trails, die sich auf 4 Hostsysteme mit je 4 Trails verteilen.

sollte durch das Testprogramm unter Beweis gestellt werden, denn die Umsetzung der DSM-Variante ist direkt mit dem Vorgehen in einem threadparallelen Programm vergleichbar. Eine effizientere Implementierung könnte die Matrix B in einer eigenen DSM-Zelle speichern sowie die Matrix A anhand der sich durch die blockweise Berechnung ergebenden Zugriffsmuster in mehrere Teile aufsplitten und diese dann in getrennten DSM-Zellen ablegen. Konkurrierende Zugriffe werden hierdurch vermieden, aber gleichzeitig die Komplexität der Implementierung erhöht.

Die Ergebnisse dieses Tests sind in Abbildung 5.25 aufgetragen. Erwartungsgemäß wird die Leistung der *nachrichtenbasierten* Variante nicht erreicht. Aufgrund der bisher gemachten Beobachtungen kann man bei einer weiteren Reduzierung der DSM-Zugriffe im Programm (wie oben beschrieben) von einer Steigerung der erhaltenen *MFlop*-Werte ausgehen. Solche Optimierungen sollen aber an dieser Stelle nicht weiter verfolgt werden. Zu sehen ist in der Abbildung 5.25 weiterhin ein Unterschied zwischen den verwendeten DSM-Implementierungen. Der *Spread*-DSM liefert für 8 Trails leicht höhere *MFlop*-Werte als die *Central*-Variante. Bei der Erhöhung der Trails und der Hostsysteme vergrößert sich dieser Unterschied, aber auch die absolut erzielten Werte sind für große Problemgrößen n höher³⁴. Die *TSpaces*-Variante kann dagegen nur einen Bruchteil der *MFlops*-Werte der beiden anderen Versionen (*Central*, *Spread*) erreichen und benötigt erheblich mehr *Java-Heap-Speicher*³⁵.

³⁴1800 zu 2000 *MFlops* für $n = 5000$. Für Werte $n \leq 4500$ ist der notwendige Overhead für die Verteilung der Daten größer als der Leistungsgewinn, der durch die parallelen Berechnungen erzielt wird.

³⁵Für die Berechnung von Matrizen der Größe $n = 5000$ mußten z.B. bei allen Knoten 3 GB Speicher reserviert werden. Im Vergleich hierzu fiel der Bedarf der beiden anderen Varianten mit etwa 1.2 GB etwas moderater aus. Der rechnerische Speicherbedarf liegt bei $\approx 570\text{MB}$ für die drei Matrizen.

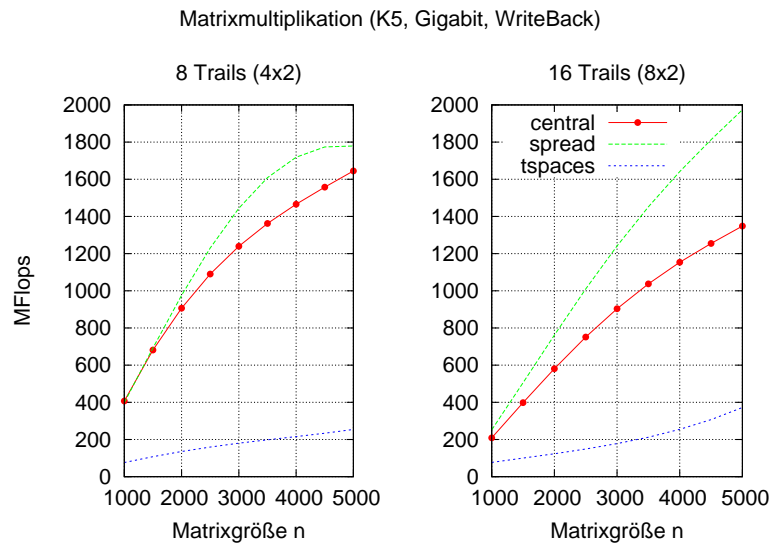


Abbildung 5.25: Testergebnisse für die beschriebenen Varianten der verteilten DSM-Matrixmultiplikation in der Konfiguration K5. Die Angaben zur Trailverteilung sind analog zu Abbildung 5.24 zu interpretieren.

Nachrichtenbasierte Variante in einer heterogenen Systemumgebung

Für die Untersuchung der nachrichtenbasierten Matrixmultiplikation in einer heterogenen Systemumgebung wurde die Konfiguration **K3** gewählt³⁶. Diese Systemkonfiguration setzt sich aus mehreren Knoten mit unterschiedlichen Prozessorarchitekturen zusammen. Unter anderen enthält K3 jeweils ein Hostsystem mit Itanium (**nemesis**), Cell (Playstation **ps3**) und UltraSPARC (**sun**) Prozessoren.

Abbildung 5.26 zeigt die unterschiedlichen Verteilungsvarianten von 12 bzw. 16 Trails, welche mit dieser Systemkonfiguration untersucht wurden. Den Knoten **hydrus** und **nemesis** wird in diesen Konfigurationen nicht die maximale Anzahl von möglichen Trails zugeordnet, welche sich anhand der verfügbaren Prozessoren(-kerne) ergibt. Betrachtet man einen solchen Fall, bei dem für jeden Prozessorkern ein Trail erstellt wird, würden in der Konfiguration K3 26 Trails generiert werden und davon allein 16 Trails auf das **hydrus** Hostsystem entfallen. Eine solche Bündelung von Trails auf nur einem Hostsystem ist aber für die nachfolgende Untersuchung eher nachteilig. Deshalb werden dem **hydrus** Hostsystem maximal 8 Trails zugeordnet. Gleiches gilt für den **nemesis** Knoten mit 4 Itanium-Prozessoren.

Die einzelnen Knoten bzw. Prozessorkerne der Systemkonfiguration K3 besitzen eine sehr unterschiedliche Leistungsfähigkeit in Bezug auf die Anzahl der durchführbaren Berechnungsoperationen pro Zeiteinheit (MFlops, *siehe Abbildung 4.12*). Dies wird auch in der Abbildung 5.27a deutlich. Bei den dort dargestellten Werten handelt es sich um die Ergebnisse der in diesem Abschnitt betrachteten nachrichtenbasierten Matrixmultiplika-

³⁶siehe Kapitel B.2.2 (S.232)

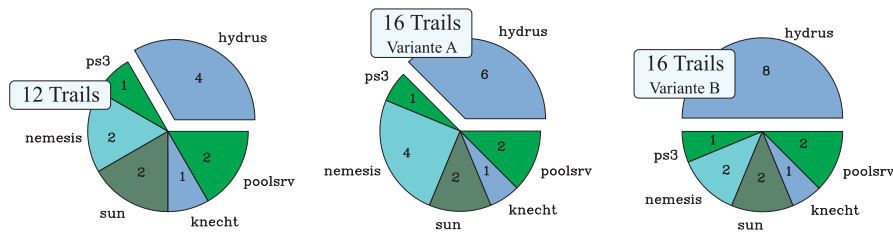


Abbildung 5.26: Aufteilung der Trails in der Systemkonfiguration K3 auf die einzelnen Hostsysteme für die untersuchte Matrixmultiplikation.

tion in der Systemkonfiguration K3. Dieser als *statisch* bezeichneten Test ordnet jedem Trail³⁷ die gleiche Anzahl von Matrixelementen zu. Die im Vergleich zu den anderen Knoten von K3 relativ langsamen Hostsysteme **ps3**, **knecht** und **sun** bestimmen hier das Resultat der Messungen, da erst nach dem Abschluß der Berechnungen aller Trails das Gesamtergebnis vom Mastertrail generiert werden kann.

Wird die Anzahl der Trails erhöht (16 Trails), so ist der Anteil jedes Trails an den Gesamtberechnungen kleiner und die Berechnungen können schneller abgeschlossen werden. Eine Variation der Trailverteilung (*Variante A* und *Variante B*) hat dagegen kaum Einfluß auf die erzielten *MFlop*-Werte. Dieses Verhalten läßt sich darauf zurückführen, daß die Anzahl der Trails nur auf den Hostsystemen **hydrus** und **nemesis** leicht abgeändert wird, aber diese Hostsysteme höhere Leistungswerte L_w als z.B. das **ps3** Hostsystem besitzen.

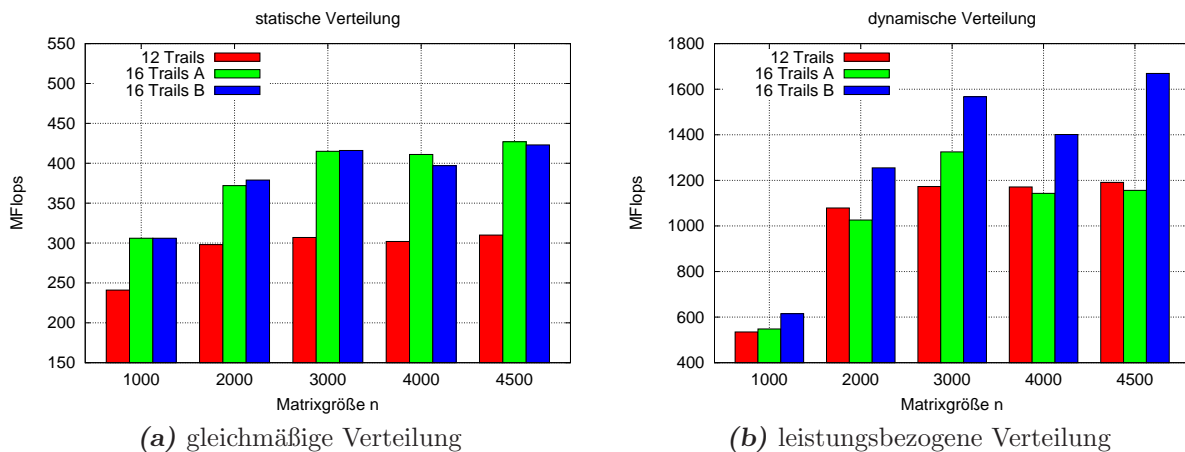


Abbildung 5.27: Ergebnisse der nachrichtenbasierten Matrixmultiplikation in einer heterogenen Systemkonfiguration **K3**.

³⁷und damit jedem Prozessorkern

Um eine bessere Ausnutzung der bestehenden Ressourcen zu erreichen, wurde eine Variante der nachrichtenbasierten Matrixmultiplikation erstellt, die keine gleichmäßige Verteilung der Berechnungen auf die Trails vornimmt. Die Implementierung nutzt hierfür die vom Laufzeitsystem angebotene Methode `DTM.runtime.getQuota()`, um jedem Trail einen an die Leistung des ausführenden Hostsystems angepaßten Berechnungsblock zuweisen zu können. Die erzielten Ergebnisse sind in der Abbildung 5.27b dargestellt. Es wurden dabei wieder die gleichen Trailverteilungen wie bei den Tests mit einer statischen Verteilung verwendet. Im direkten Vergleich mit diesen Tests (Abbildung 5.27a) ist jeweils eine deutliche Verbesserung der erzielten Berechnungsgeschwindigkeiten erkennbar. Vergleicht man die Tests der *dynamischen* Verteilung untereinander, so fällt auf, daß bei der Erhöhung der Trailanzahl von 12 auf 16 Trails (*Variante A*) zunächst keine signifikante Zunahme der MFlops Werte verzeichnet werden kann. Erst beim Verschieben einiger Trails vom `nemesis` Hostsystem auf das `hydrus` Hostsystem (*Variante B*) kann man wieder eine solche Steigerung erkennen. Zurückzuführen ist dies auf bessere Leistungswerte L_w der Prozessorkerne im `hydrus` Hostsystem. Die im Vorfeld erfolgte Erhöhung der Trailanzahl von 12 auf 16 Trails (*Variante A*) erzielt auch eine solche Verbesserung, diese wird aber durch den notwendigen Mehraufwand bei der Kommunikation (mehr Nachrichten) überdeckt.

5.3.2 N-Damen-Problem

Beim ursprünglich definierten *8 Damen-Problem* geht es um die Frage, wie 8 Damen auf einem Schachbrett angeordnet werden müssen, ohne daß diese sich nach den geltenden Schachregeln gegenseitig *schlagen* können. Es geht auf den bayerischen Schachmeister Max Bezzel im Jahre 1848 zurück. Später wurde es auf eine Schachbrettgröße von N mit N Damen verallgemeinert [52, 93].

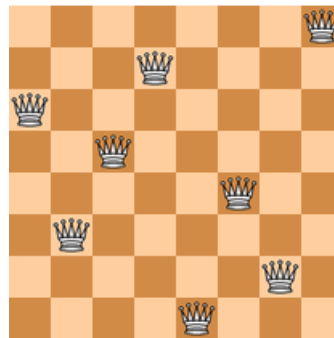


Abbildung 5.28: Eine mögliche Lösung für das 8 Damen-Problem.

Quelle: de.wikipedia.org

Das Problem besitzt eine exponentielle Komplexität, da es auf einem $N \times N$ Spielfeld

$$\alpha = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot (n - N + 1) < n^N \quad \text{mit } n = N \cdot N$$

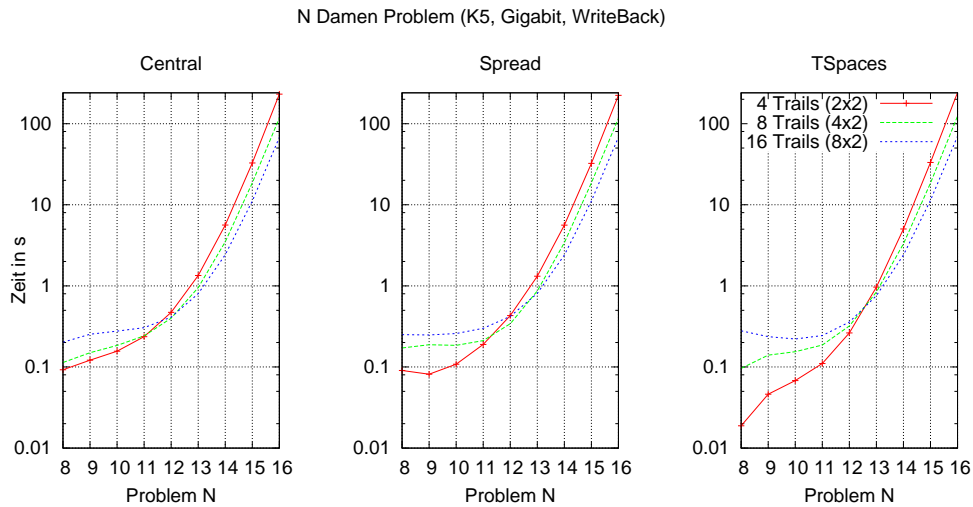


Abbildung 5.29: Abgebildet sind die Laufzeiten der Implementierung zur Lösung des N Damen Problems. Aufgrund der geringen Anzahl von DSM-Zugriffen unterscheiden sich die Ergebnisse für einzelne DSM-Varianten nicht. Die in den Klammern stehenden Angaben der Legende geben Auskunft über die genaue Verteilung der Trails und sind als (Hostsysteme \times lokale Trails) zu interpretieren.

mögliche Positionierungen der N Damen gibt³⁸. Diese Abschätzung kann man noch weiter eingrenzen. Geht man davon aus, daß in jeder Reihe nur eine Dame vorkommen darf, dann reduziert sich die Anzahl der Möglichkeiten α auf:

$$2^N < \alpha = N \cdot (N - 1) \cdot \dots \cdot 2 \cdot 1 < N^N \quad \forall N > 2$$

Der zu untersuchende Lösungsraum ist also bereits für relativ kleine N sehr groß³⁹ und wird von den meisten Lösungsverfahren komplett durchsucht. Die Anzahl der darin enthaltenen, gültigen Anordnungen $g(N)$ kann man durch die relativ grobe Abschätzung

$$g(N) = N! \cdot c^N \quad \text{mit } c \approx 0.39$$

bestimmen, aber auch hier ist mit einem exponentiellen Wachstum zu rechnen [111]. Die verwendete Implementierung gibt in Abhängigkeit der verfügbaren Trails einige feste Anfangspermutationen mit x Damen⁴⁰ für jeden einzelnen Trail vor. Jeder der Trails berechnet dann mögliche Lösungen, indem die freien Positionen systematisch variiert werden. Nach Beendigung der Suche addiert jeder Trail die Anzahl seiner gefundenen Lösungen auf den in einer DSM-Zelle abgelegten globalen Zähler. Sind alle Trails beendet, enthält dieser Zähler die Anzahl der insgesamt möglichen Lösungen.

³⁸Permutationen werden eventuell mehrfach gezählt

³⁹Aktuell sind nur die Lösungen bis $N = 25$ bekannt.

⁴⁰ $x < N$

Auswertung Das erstellte Testprogramm wurde mit allen verfügbaren DSM-Varianten und verschiedenen Verteilungen von Trails untersucht. Einige der Ergebnisse sind getrennt nach der DSM-Implementierung in Abbildung 5.29 aufgetragen. Da sehr wenige Zugriffe auf den DSM generiert werden, beeinflußt dessen Performance die Laufzeiten kaum. Die Gesamtlaufzeit steigt wie erwartet bereits bei kleinen Problemgrößen sehr stark an.

5.3.3 Bewertung des Laufzeitsystems

Bei den untersuchten Anwendungsbeispielen hat sich gezeigt, daß threadparallele Programme gut auf die *Beluga* Laufzeitumgebung zu portieren sind. Die erzielten Ergebnisse sind auch ohne zusätzliche Optimierungen akzeptabel. Dennoch müssen u.U. speziell für den DSM zugeschnittene Strategien verwendet werden, da die Zugriffe auf den DSM einen entscheidenden Einfluß auf die Laufzeiteigenschaften der Anwendung besitzen.

5.3.4 Laufzeitvorhersage

Dieser Abschnitt wird sich mit den bis jetzt gewonnenen Aussagen auseinandersetzen und einige allgemeine Abschätzungen über die zu erwartenden Laufzeiten von verteilt laufenden *DEE* Anwendungen liefern.

Für die späteren Betrachtungen und beim Vergleich der implementierten Algorithmen wird häufig der **Speedup** s benutzt. Dabei handelt es sich um einen rechnerischen Wert, welcher die relative Laufzeitverbesserung eines parallelen Programmes gegenüber seinem sequentiellen Pendant angibt.

$$s = \frac{T_s}{T_p}$$

Der Wert von T_s steht für die Laufzeit einer sequentiellen Variante und der Wert von T_p gibt die Laufzeit der betrachteten parallelen Version an.

Die in dieser Arbeit benutzten Speedup-Werte beziehen sich immer⁴¹ auf die Laufzeiten einer *Beluga* Version des gerade betrachteten Algorithmus, die mit einem Trail erzielt wurden.

Gene Amdahl formulierte 1967 eine Abschätzung für den Speedup, die als **Amdahlsches Gesetz** (5.2) bekannt ist [6]. Es bezieht den sequentiell auszuführenden Teil eines parallelen Programmes ein und liefert eine obere Grenze des zu erwartenden Speedups bei der Parallelisierung. Da verschiedene Einflußfaktoren nicht berücksichtigt werden, können in der Praxis dennoch höhere (superlineare) Speedup-Werte erreicht werden. Gleichwohl kann mit dem *Gesetz von Amdahl* das Potential einer parallelen Implementierung bei Benutzung von p Kontrollflüssen und einem sequentiellen Anteil α_s bestimmt werden.

$$s = \frac{1}{\alpha_s + \frac{1-\alpha_s}{p}} \leq \frac{1}{\alpha_s} \quad (5.2)$$

⁴¹Abweichungen werden angegeben. Bei Betrachtungen threadparalleler Implementierungen wird der *Speedup* auf die entsprechende serielle Variante bezogen.

	Schreibzugriffe max	Ω Referenz 1000 MB/s	Lesezugriffe max	Ω Referenz 1000 MB/s
Central	2.1MB/s	≈ 500	280 Byte/s	$\approx 4 \cdot 10^6$
Spread	1.8MB/s	≈ 500	250 Byte/s	$\approx 4 \cdot 10^6$
TSpaces	5000Byte/s	$\approx 2 \cdot 10^5$	5500 Byte/s	$\approx 2 \cdot 10^5$

Tabelle 5.2: Ermittelte Ω Werte für die Tests aus Abschnitt 5.2.3 Als Referenz für die Bestimmung von Ω wurde ein Vergleichsmedium mit 1000 MB/s verwendet. Dieser Wert entspricht in etwa den ermittelten Hauptspeicherbandbreiten des `hydra` Hostsystems.

Bei einem sequentiellen Anteil $\alpha_s = 20\%$ beträgt der maximal erreichbare Speedup 5 und kann durch den Einsatz zusätzlicher paralleler Kontrollflüsse nicht weiter verbessert werden [109].

Im Falle einer parallelen Abarbeitung ist häufig eine Kommunikation zwischen den Kontrollflüssen notwendig. Im Threadmodell findet diese Kommunikation implizit über direkte Hauptspeicherzugriffe statt. Für die Programmierung von Maschinen mit verteiltem Speicher ist das Versenden von Nachrichten notwendig. Läßt man diese Überlegungen in die Formel 5.2 einfließen, erhält man Formel 5.3:

$$s = \frac{1}{\alpha_s + K(p) \cdot \frac{1-\alpha_s}{p}} \quad \text{mit} \quad 1 \leq K(p) \quad (5.3)$$

Die Funktion $K(p)$ (Formel 5.4) gibt dabei einen Faktor für die notwendige parallele Kommunikation an. Diese Funktion ist zum einen abhängig vom Kommunikationsverhalten γ_p des Verfahrens. Dieser Parameter beschreibt die Menge der bei der gesamten Kommunikation übertragenen Daten und die hierfür notwendigen Operationen. Zum anderen hängt $K(p)$ von der Übertragungsleistung des benutzten Kommunikationsmediums Ω ab. Der Wert von Ω ist relativ und bezieht sich immer auf ein Vergleichsmedium.

$$K(p) = \Omega \cdot \gamma_p \quad \text{mit} \quad 0 < \Omega, \gamma_p \quad (5.4)$$

Der Wert von γ_p (5.5) läßt sich aus den Formeln 5.3 und 5.4 herleiten.

$$\gamma_p = p \cdot \frac{1 - s \cdot \alpha_s}{\Omega \cdot (1 - \alpha_s)} \quad (5.5)$$

In der Tabelle 5.2 sind die ermittelten Ω Werte für die gemessenen Übertragungsraten der DSM-Varianten enthalten. Als Grundlage wurden die in Kapitel 5.2.3 ermittelten Hauptspeicherbandbreiten des `hydrus` Hostsystems verwendet. Die Werte für Ω schwanken relativ stark ($500 < \Omega < 4 \cdot 10^6$), somit können sich für einen untersuchten Algorithmus erhebliche Unterschiede ergeben. Allerdings ist der letztendlich resultierende Speedup noch vom Kommunikationsanteil γ_p abhängig. Bei einem wenig kommunizierenden Verfahren, wie z.B. dem N-Queens-Problemlöser, besitzt deshalb die Leistung des Kommunikationsmediums Ω kaum Einfluß auf den erreichbaren Speedup.

5.3.5 Zusammenfassung

Für die Beurteilung der Prototyp-Laufzeitumgebung wurden mehrere Varianten einer Matrixmultiplikation und ein Verfahren zur Lösung des N-Damen-Problems implementiert. Bei den zur Untersuchung dieser Programmvarianten durchgeführten Laufzeittests konnte bereits gute Ergebnisse für die verteilten Ausführungsfälle erzielt werden.

Für die Gegenüberstellung von Programmausführungszeiten unterschiedlicher (paralleler) Varianten einer Anwendung eignet sich der Speedup. Mit einer alternativen Berechnungsvorschrift für den Speedup, dem Ahmdahlschen Gesetz, können Voraussagen zur Laufzeit eines implementierten Verfahrens getroffen werden, wenn bestimmte Parameter dieser Implementierung bekannt sind. Erweitert man diese Sichtweise und läßt z.B. als weitere Faktoren einige Parameter des verwendeten Kommunikationsmediums (Netzwerk, Hauptspeicher) in die Betrachtungen mit einfließen, so können zusätzliche Aussagen zum Laufzeitverhalten einer untersuchten parallelen Anwendung gewonnen werden. Eine genaue Bestimmung der Einflußfaktoren ist für eine allgemeine Analyse schwierig und muß sich am gegebenen Problem bzw. dem implementierten Verfahren orientieren. Generell kann man jedoch feststellen, daß ein wesentlicher Anteil an der erreichbaren Performance durch das Kommunikationsverhalten der untersuchten Anwendung bestimmt wird.

5.4 Schlußbemerkungen

In diesem Kapitel wurden einzelne Kernkomponenten der *Beluga* Prototypimplementierung sowie die gesamte Laufzeitumgebung mit Hilfe verschiedener Testverfahren und Testanwendungen eingehender untersucht. Im Mittelpunkt der Untersuchungen stand zunächst die verwendete *Beluga* Kommunikationsschicht. Die Beobachtungsergebnisse für diesen Teil der Untersuchungen lassen den Schluß zu, daß die verwendete Objektserialisierung einen Großteil der Kommunikationszeit einnimmt. Zukünftige Erweiterungen könnten deshalb gezielt diese Kernkomponente optimieren, um eine Verbesserung der Übertragungsleistungen zu erreichen.

Einen weiteren Schwerpunkt dieses Kapitels bildeten die Untersuchungen des vorhandenen DSM-Subsystems. Hierbei wurden die Unterschiede der einzelnen DSM-Serverimplementierungen deutlich. Insbesondere ergibt sich ein differenziertes Bild für den *TSpaces*-DSM. Zum Abschluß wurde das Laufzeitverhalten einiger Testanwendungen betrachtet. Neben einer Laufzeitanalyse, wurden hierbei auch praktische Fragestellungen, wie z.B. die Portierbarkeit threadparalleler Programme in trailparallele Varianten kurz angesprochen.

Aufbauend auf den in diesem Kapitel gewonnenen Erkenntnissen, sollen in den nun nachfolgenden Kapiteln weitere Testanwendungen für die *Beluga* Laufzeitumgebung implementiert und vertiefend untersucht werden.

Kapitel 6

Numerische Simulation

There's no sense in being precise when you don't even know what you're talking about.

(John von Neumann)

Der nachfolgende Teil der Arbeit widmet sich dem praktischen Einsatz der erstellten *DEE* Prototypimplementierung und gliedert sich in mehrere Kapitel. In diesem Kapitel wird zunächst in die Thematik von numerischen Simulationsalgorithmen eingeführt und es werden einige ausgewählte Verfahren kurz vorgestellt. Die sich dann anschließenden Kapitel 7 und Kapitel 8 greifen die *Linked-Cell*-Methode sowie den *Barnes-Hut*-Algorithmus auf und beschreiben verschiedene Implementierungsvarianten, welche in der *Beluga* Laufzeitumgebung untersucht werden. Das Hauptaugenmerk bei diesen Untersuchungen liegt zunächst in den Anforderungen an die jeweiligen verteilten Datenstrukturen der Verfahren. Optimale Laufzeiteigenschaften und Skalierbarkeit der Implementierungen spielen dabei eine eher untergeordnete Rolle. Vorrangiges Ziel ist die Untersuchung der prinzipiellen Eignung der *Beluga* Laufzeitumgebung für die Erstellung von portablen parallelen Anwendungen.

6.1 Simulationsverfahren

Allgemein versteht man heute unter der *Numerischen Simulation* eine Computersimulation, welche auf der Grundlage von numerischen Methoden durchgeführt wird. In der letzten Zeit hat sich hierfür auch die Bezeichnung *wissenschaftliches Rechnen* durchgesetzt, die manchmal bereits als eigenständige Disziplin geführt wird [46, 74]. Diese Formulierung sorgt für eine klare Abgrenzung von der nicht rechnergestützten numerischen Simulation, die für die meisten Anwendungsfälle ohnehin nicht mehr durchführbar ist.

Zwei des öfteren benutzte Begriffe sollen zunächst genannt und voneinander abgegrenzt werden. Im Falle der **Simulationszeit** t_{end} ist die im Simulationsmodell vergangene Zeit gemeint, während sich die Angabe einer **Laufzeit** immer auf die real vergangene Zeit für die Durchführung der Simulationsberechnungen in einem Computersystem bezieht.

6.1.1 Simulationsraum

Viele numerischen Simulationsverfahren basieren auf einer Aufteilung des Simulationsgebietes in kleinere Unterräume. Diese so erzeugten Teilräume können einzeln berechnet werden, besitzen aber meist Abhängigkeiten zu anderen Teilräumen. Die genauen Wechselwirkungen zwischen diesen Simulationsgebieten hängen vom gewählten Verfahren ab und wirken einer effektiven parallelen Implementierung meist entgegen.

Generell kann man zwei Strategien für die Aufteilung des Simulationsraumes unterscheiden:

äquidistante Aufteilung (uniform) Das Simulationsgebiet wird in Teilgebiete gleicher Größe und Form eingeteilt. Jeder Berechnungseinheit¹ kann dabei eine unterschiedliche Anzahl dieser Teilgebiete zur Bearbeitung zugeordnet werden.

nicht äquidistante Aufteilung (non-uniform) Das Simulationsgebiet wird meist rekursiv in Teile zerlegt, die eine unterschiedliche Größe besitzen können. Der Aufwand für die Speicherung der einzelnen Gebiete ist dabei oft höher als bei einer uniformen Zerlegung, da sich im allgemeinen die Anzahl dieser Gebiete nicht im Voraus festlegen läßt.

6.1.2 Verfahren zur Partikel- und Strömungssimulation

Zwei große Teilbereiche der numerischen Simulation sind die Partikelsimulation und die Strömungssimulation². Für diese beiden Klassen gibt es eine Vielzahl von Methoden und Algorithmen, um unterschiedliche Anforderungen zu erfüllen.

Partikelsimulation

Partikelsimulationen beruhen auf der Berechnung von Wechselwirkungen zwischen einzelnen Partikeln und ermöglichen so die Nachbildung von natürlichen Prozessen. Das Verhalten von Molekülen, Gasmischen, Flüssigkeiten, Kristallen, Festkörpern bis hin zu Galaxien kann so simuliert werden. Zur Beschreibung von Partikelbeziehungen und wirkenden Kräften gibt es für die unterschiedlichen Anforderungen, die von einem speziellen Problem vorgegeben werden, wieder vielfältige Möglichkeiten und Varianten. Generell kann man jedoch zwei große Gruppen unterscheiden: Verfahren für **kurzreichweitige** und für **langreichweitige** Potentiale. Bei *kurzreichweitigen* Potentialen nimmt die Kraft zwischen zwei Partikeln mit zunehmendem Abstand zueinander sehr stark ab, so daß ab einer bestimmten Entfernung die Wechselwirkung dieser Partikeln vernachlässigt werden kann. Beispiele für solche Potentiale sind z.B. das *Lennard-Jones-Potential* oder das *Finnis-Sinclair-Potential* [46]. Als Berechnungsmethode kann das *Linked-Cell-Verfahren* eingesetzt werden. Bei *langreichweitigen* Potentialen (Gravitations- oder elektrostatische Kräfte) müssen immer alle Partikel bzw. eine sehr große Anzahl von Partikeln des Simulationsgebietes betrachtet werden. Die Beschränkung auf eine kleine Partikelmenge pro

¹damit sind parallel arbeitende Prozesse, Threads bzw. Trails gemeint

²Weitere Simulationen sind Wetter- und Klimaprognosen sowie Crashtests.

Berechnung ist deshalb nicht möglich und die allgemeine Laufzeit solcher Simulationen ist quadratisch. Läßt man wieder eine Toleranz in den Berechnungen zu oder nimmt gewisse Ungenauigkeiten in Kauf, existieren z.B. Baumverfahren, die wesentlich bessere Laufzeiteigenschaften vorweisen können. Neben dem später noch näher beschriebenen *Barnes-Hut*-Algorithmus (Komplexität $\mathcal{O}(n \log n)$) gibt es u.a. die Fast-Multipole-Methode mit einer Komplexität von $\mathcal{O}(n)$.

Strömungssimulation

Als Alternative zu aufwendigen Windkanal-Versuchen bieten sich Strömungssimulationen an. Die verbreitetsten Verfahren stützen sich meist auf Navier-Stokes-Gleichungen, die ein System von nichtlinearen partiellen Differentialgleichungen 2.Ordnung bilden und Strömungen in newtonschen Flüssigkeiten und Gasen sehr umfassend beschreiben. Alternativ können auch Euler-Gleichungen oder Gleichungen für Potentialströmungen verwendet werden [92].

Als zentralen Oberbegriff für diese Klasse von Simulationen verwendet man häufig die Abkürzung *CFD* (für *Computational Fluid Dynamics*). Darunter gibt es wieder eine große Fülle von spezialisierten Verfahren, von denen nur einige wie z.B. Large-Eddy-Simulation (*LES*), Smoothed-Particle-Hydrodynamics (*SPH*), Spektralmethode (*SEM*) oder die Lattice-Boltzmann-Methode (*LBM*) an dieser Stelle genannt werden sollen.

6.2 Kurzbetrachtung ausgesuchter Verfahren

Dieser Abschnitt stellt drei Verfahren zur Lösung unterschiedlicher Probleme vor. Die hier beschriebenen *Linked-Cell*- und *Barnes-Hut*-Verfahren werden in den nachfolgenden Kapiteln implementiert und vertiefend untersucht.

Als Drittes werden kurz einige Grundzüge der Lattice-Boltzmann-Methode (*LBM*) als ein Verfahren zur Strömungssimulation skizziert. Die Analyse einer entsprechenden Anwendung für die *Beluga* Laufzeitumgebung erfolgt an dieser Stelle nicht. Das *LBM*-Verfahren wird aber dennoch kurz besprochen, um Gemeinsamkeiten zu den betrachteten *Linked-Cell*- und *Barnes-Hut*-Verfahren aufzuzeigen und eine Verallgemeinerung der dort erhaltenen Resultate zu erleichtern.

6.2.1 Linked-Cell-Verfahren

Das *Linked-Cell*-Verfahren wird zur approximativen Auswertung der Kräfte und Energien für schnell abfallende (kurzreichweitige) Potentiale verwendet [46], d.h. eventuelle Wechselwirkungen zwischen Partikeln lassen sich auf die nächsten geometrischen Nachbarn beschränken. Je nach Anforderungsprofil können für die direkte Berechnung dieser Wechselwirkungen verschiedene numerische Verfahren eingesetzt werden, einige Beispiele hierfür sind z.B. das Lennard-Jones-Potential oder das EAM-Potential [46].

Beim *Linked-Cell*-Verfahren wird nun der Simulationsraum in uniforme Teilgebiete (Zellen) zerlegt, welche jeweils getrennt voneinander betrachtet und berechnet werden können. Jede diese Zellen enthält eine endliche Anzahl von Partikeln. Für die Berechnungen

der Wechselwirkungen eines Partikels müssen alle Partikel der jeweiligen Zelle und der direkten Nachbarzellen berücksichtigt werden. Nach einem oder mehreren Berechnungsschritten ist im allgemeinen ein Korrekturschritt notwendig, der Interaktionen zwischen benachbarten Zellen auflöst, die z.B. beim Übergang von Partikeln einer Simulationszelle in eine andere Simulationszelle auftreten.

Während der Berechnung der neuen Raumkoordinaten von Partikeln in einer Zelle kann auf diese Partikel ein lesender Zugriff erfolgen. Solche Zugriffe können z.B. bei der parallelen Berechnung von zwei benachbarten Zellen auftreten.

6.2.2 Barnes-Hut-Algorithmus

Der *Barnes-Hut*-Algorithmus wurde ursprünglich für astrophysikalische Anwendungen entwickelt und geht auf Barnes und Hut zurück [16]. Er gehört zur Gruppe der Baumverfahren für langreichweitige Potentiale [46] und erreicht eine Reduzierung der Laufzeit, indem mehrere Partikel hierarchisch zu einzelnen Pseudopartikeln zusammengefaßt werden. Bei der Berechnung eines einzelnen Partikels müssen dann nicht mehr alle anderen Partikel betrachtet werden, sondern es können ganze Partikelmengen durch einen einzelnen Pseudopartikel ersetzt werden. Voraussetzung dafür ist, daß der Pseudopartikel und der zu berechnende Partikel weit genug voneinander entfernt sind. Der Simulationsraum

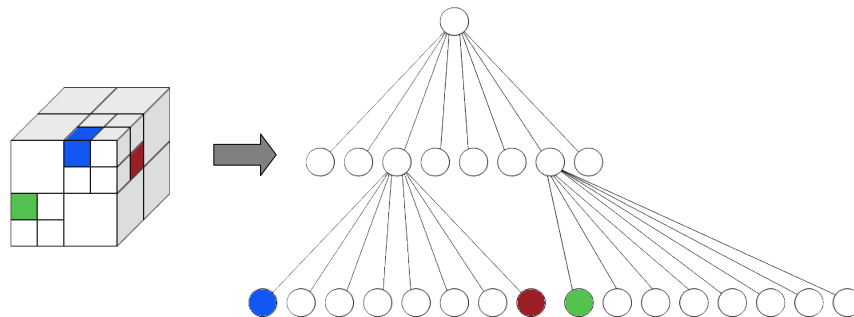


Abbildung 6.1: Darstellung und Aufteilung eines Simulationsgebietes als Octree. Zur besseren Übersicht sind einige Teilbereiche farblich markiert.

wird beim *Barnes-Hut*-Verfahren in einem Octree verwaltet und enthält in jedem Blatt höchstens einen Partikel. Die Tiefe des Baumes hängt deshalb von der Partikelverteilung ab, kann aber mit $\mathcal{O}(\log n)$ abgeschätzt werden (Abbildung 6.1). Jeder innere Knoten im Baum enthält einen Pseudopartikel und repräsentiert gleichzeitig einen Teilbereich des Simulationsraumes in Form eines Würfels mit fester Kantenlänge. Die Kantenlänge wird in jeder Stufe des Baumes halbiert, weshalb im dreidimensionalen Fall immer 8 Unterwürfel entstehen. Für einen Simulationsraum mit einer Kantenlänge von 1000 Längeneinheiten (LE) existieren deshalb in einer Tiefe³ von 2 maximal 64 Blattknoten mit einer Kantenlänge von jeweils $250LE$.

³Die Wurzel hat die Tiefe 0.

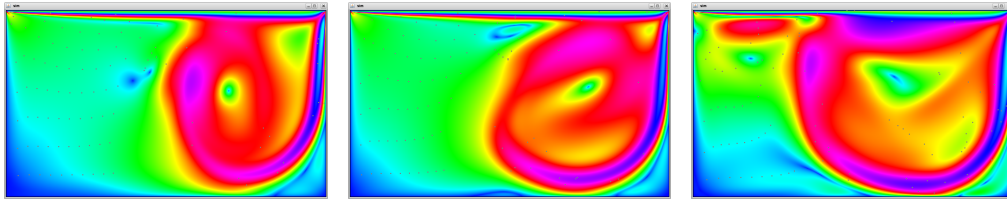


Abbildung 6.2: Darstellung der Geschwindigkeitsverteilung einer Lattice-Boltzmann-Simulation aus mehreren Zeitabschnitten.

6.2.3 Lattice-Boltzmann-Methode

Die Lattice-Boltzmann-Methode ist ein hinreichend [29, 94, 118] untersuchtes Verfahren zur Simulation von inkompressiblen Strömungen (Abbildung 6.2). Es basiert auf einer vereinfachten Teilchen-Mikrodynamik, wobei eigentlich keine Teilchen sondern kontinuierliche Teilchendichten verwendet werden. Die zu simulierenden Fluide setzen sich aber aus mikroskopischer Sicht aus einzelnen Teilchen zusammen, welche sich nach den Gesetzen der Mechanik durch den Raum bewegen und miteinander wechselwirken. Diese Wechselwirkungen werden durch die Boltzmann-Gleichung beschrieben. Bei den Berechnungen wird der Simulationsraum wieder anhand eines uniformen Gitters in einzelne Zellen unterteilt. Für jede dieser Zellen kann für die Simulationsberechnungen die Anzahl der benötigten (Nachbar-)Zellen eingegrenzt werden. Hierdurch wird wie beim *Linked-Cell-Verfahren* eine Reduktion der insgesamt notwendigen Berechnungen erreicht.

6.3 Zusammenfassung

In diesem Kapitel wurde kurz in die Thematik der *numerischen Simulation* eingeführt und stellvertretend für die Vielzahl der existierenden Algorithmen eine kleine Auswahl von Verfahren zur Partikel- und Strömungssimulation vorgestellt. Die besprochenen Verfahren verwenden eine Aufteilung des Simulationsraumes in kleinere Teilräume. Die Wechselwirkungen der in diesen Teilräumen enthaltenen Partikel können dann oft mit geringen Abhängigkeiten zu anderen Teilräumen parallel berechnet werden. Zur Verwaltung der Teilräume kommen unterschiedliche Datenstrukturen zum Einsatz. Der *Barnes-Hut*-Algorithmus verwendet eine Baumdatenstruktur, während beim *Linked-Cell-Verfahren* und der *Lattice-Boltzmann-Methode* lineare Felder eingesetzt werden können. Die effiziente Ablage dieser Datenstrukturen im DSM-Subsystem einer *DEE* Programmierumgebung ist für die Laufzeit trailparalleler Programme ein entscheidender Faktor und wird neben anderen Aspekten in den nachfolgenden Kapiteln genauer untersucht.

Kapitel 7

Linked-Cell-Verfahren

When in doubt, use brute force.

(Ken Thompson)

Dieses Kapitel wird verschiedene Implementierungen des *Linked-Cell*-Verfahrens beschreiben und deren Laufzeitverhalten analysieren. Zunächst werden das Verfahren und die für eine Implementierung notwendigen Zusammenhänge erläutert. Ausgehend von diesen Betrachtungen erfolgt die Implementierung einer sequentiellen sowie einer threadparallelen Variante des Verfahrens. Beide Implementierungen werden kurz untersucht und aus den daraus gewonnenen Erkenntnissen eine trailparallele Variante für das *Beluga* Laufzeitsystem entwickelt, welche im Anschluß detailliert in verschiedenen verteilten Laufzeitkonfigurationen untersucht wird.

7.1 Beschreibung des Verfahrens

Das *Linked-Cell*-Verfahren wird für Partikelsimulationen mit kurzreichweitigen Potentialen eingesetzt. Durch das Verfahren wird die uniforme Aufteilung des Simulationsraumes in mehrere Teilräume (Raumzellen¹) beschrieben und die Rahmenbedingungen für die Bestimmung der Kräfte zwischen den darin befindlichen Partikel werden definiert. Für die durchzuführenden Berechnungen, welche für die Simulation der Wechselwirkung zwischen den einzelnen Partikeln notwendig sind, können verschiedene austauschbare Methoden verwendet werden. Die Raumzellen mit den darin befindlichen Partikel können hierbei getrennt voneinander betrachtet und berechnet werden. Abhängigkeiten existieren jeweils nur zwischen benachbarten Raumzellen. Aufgrund dieser Eigenschaft ist die parallele Berechnung einzelner Raumzellen möglich.

7.1.1 Grundlagen und Berechnungsvorschriften

Bevor auf das eigentliche Verfahren eingegangen wird, sollen die notwendigen und benutzten Berechnungsvorschriften erläutert werden.

¹sog. Raumzellen für den dreidimensionalen Fall

Diskretisierung

Um ein gegebenes Modell berechnen zu können, muß das beschreibende kontinuierliche mathematische Problem aus unendlich vielen Bezugspunkten in ein Problem mit endlich vielen Punkten umgewandelt werden. Diesen Vorgang umschreibt man in der Numerik als Diskretisierung. Bei der Diskretisierung von Differentialgleichungen werden diese z.B. in ein Gleichungssystem umgewandelt, dessen Lösung sich der zugrundeliegenden Differentialgleichung nur an einigen ausgewählten Punkten annähert. Durch diesen Vorgang wird anhand von Zusatzinformationen oder bewußten Einschränkungen die Komplexität verringert und es kann meist mit vertretbarem Aufwand eine Näherungslösung gefunden werden.

Vor dem hier betrachteten Hintergrund wird eine Zeitdiskretisierung zur Berechnung der neuen Raumkoordinaten und der neuen Geschwindigkeit des Partikels i aus dem *alten* Positionswert (\hat{x}_i^n), der Geschwindigkeit (\hat{v}_i^n), der Masse (m_i) und der zugehörigen Krafteinwirkung (\hat{F}_i^n) aus dem vorangegangenen Zeitschritt n vorgenommen. Für solche Zeitdiskretisierungen gibt es wieder ein Vielzahl von Möglichkeiten. Hier soll das *Störmer-Verlet*-Verfahren bzw. dessen Variante, das *Geschwindigkeits-Störmer-Verlet*-Verfahren, benutzt und kurz beschrieben werden. Es basiert auf der Zeitdiskretisierung der Newtonschen Bewegungsgleichung $\hat{F} = m\hat{a}$ und liefert neue Werte² für die Position (7.1) und die Geschwindigkeit (7.2) eines Partikels i . Hierbei wird das Zeitintervall $[0, t_{end}] \subset \mathbb{R}$ der Simulation ($t_{end} = \text{Simulationszeit}$) in I Teilintervalle gleicher Größe $\delta t = t_{end}/I$ zerlegt. Eine ausführliche Herleitung des Verfahrens ist z.B. in [46] zu finden.

$$\hat{x}_i^{n+1} = \hat{x}_i^n + \delta t \cdot \hat{v}_i^n + \frac{\hat{F}_i^n \cdot \delta t^2}{2m_i} \quad (7.1)$$

$$\hat{v}_i^{n+1} = \hat{v}_i^n + \frac{(\hat{F}_i^n + \hat{F}_i^{n+1})\delta t}{2m_i} \quad (7.2)$$

Potentiale

Für das Diskretisierungsverfahren müssen die Werte aus der vorangegangenen Iteration, die Partikelmasse und die aktuell einwirkende Kraft \hat{F}_i^{n+1} bekannt sein. Diese Kraftwirkung wird durch ein Potential bzw. eine Potentialfunktion U beschrieben, welche anhand der konkreten Problemstellung ausgewählt werden kann. Für Wechselwirkungen zwischen zwei Partikeln i und j mit dem Abstand³ r_{ij} eignen sich je nach Anforderungsprofil z.B. das *Lennard-Jones-Potential* oder das *EAM-Potential* [46, 128]. Bei der erstellten Implementierung wurde als Potentialfunktion $U(r_{ij})$ das *Lennard-Jones-Potential* verwendet, dessen allgemeine Form in der Formel (7.3) zu finden ist.

$$U(r_{ij}) = \underbrace{\frac{1}{n-m} \left(\frac{n^n}{m^m} \right)^{\frac{1}{n-m}}}_{\alpha} \cdot \varepsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^n - \left(\frac{\sigma}{r_{ij}} \right)^m \right] \quad (7.3)$$

²Bei \hat{F} , \hat{a} , \hat{v} und \hat{x} handelt es sich jeweils um Vektoren der Dimension d .

³ $r_{ij} = \|x_j - x_i\|$

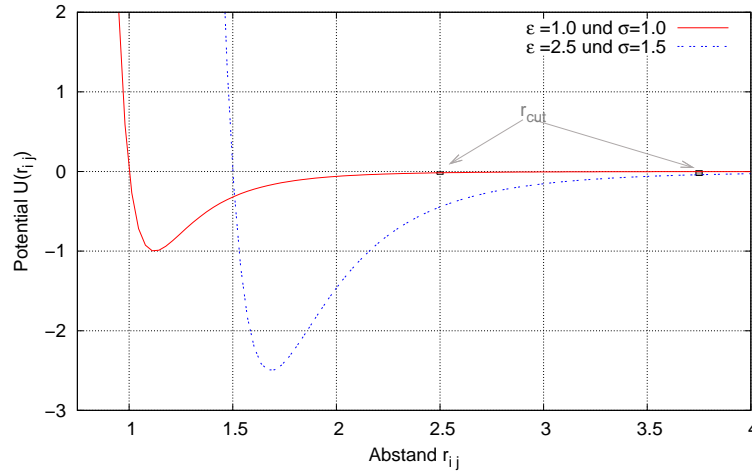


Abbildung 7.1: Lennard-Jones-(12,6)-Potential für verschiedene Parameterwerte σ und ε . Gut zu sehen ist der Einfluß der Parameter auf das Kurvenverhalten. Der Nulldurchgang, d.h der Übergang von der Abstoßung zur Anziehung ist jeweils bei $r_{ij} = \sigma$ zu finden. Als max. Anziehung wird $-\varepsilon$ erreicht, diese fällt dann recht schnell gegen 0.

Der Parameter σ gibt den Nulldurchgang des Potentials an. Mit ε werden die Bindungen zwischen den Partikeln (Festigkeit) des simulierten Materials beschrieben. Eine Erhöhung führt hierbei zu einem härteren Material bzw. zu stärkeren Bindungen (vgl. dazu *Abbildung 7.1*). Durch die Wahl der weiteren Konstanten $m = 6$ und $n = 12$, die sich an [46] orientieren, läßt sich (7.3) weiter vereinfachen und der Wert von α ergibt 4. Diese Form (7.4) ist teilweise auch als *Lennard-Jones-(12,6)-Potential* zu finden.

$$U(r_{ij}) = 4 \cdot \varepsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^6 \right] \quad (7.4)$$

Abschneideradius

Die im Vorfeld erwähnte lokale Betrachtung der Wechselwirkungen von Partikeln mit deren unmittelbaren Nachbarn zur Beschleunigung der Berechnungen, wird durch den sog. *Abschneideradius* r_{cut} ausgedrückt. Dieser Radius ist abhängig vom verwendeten Potential und gibt an, bis zu welchem Abstand r_{ij} ein Partikel j noch einen wesentlichen Beitrag zur Potentialbestimmung von i besitzt. Der Abstand r_{cut} wird oft mit 2.5σ gewählt. Alle Partikel j mit einem größeren Abstand zu i werden dann nicht mehr betrachtet. Konsequenterweise muß man jetzt erwähnen, daß die Gesamtenergie des Systems nicht mehr exakt erhalten bleibt. Dem kann man durch die Vergrößerung von r_{cut} entgegen wirken, nimmt dann aber wieder eine Erhöhung der zu betrachtenden Partikel und damit eine höhere Laufzeit in Kauf.

Kraftberechnung

Die wirkende Kraft zwischen den Partikeln i und j ergibt sich durch Gradientenbildung⁴ und liefert (7.5).

$$\hat{F}_{ij} = 24 \cdot \varepsilon \cdot \underbrace{\frac{1}{r_{ij}^2} \cdot \left(\frac{\sigma}{r_{ij}}\right)^6 \cdot \left(1 - 2 \cdot \left(\frac{\sigma}{r_{ij}}\right)^6\right)}_{K(i,j)} \cdot \hat{r}_{ij} \quad (7.5)$$

Für die gesamte Kraft \hat{F}_i , die auf i wirkt, müssen dann alle dafür in Betracht kommenden Partikel einbezogen werden (7.6). Da nicht alle Partikel des Systems in diese Berechnungen mit einfließen, ist das Ergebnis nur eine Näherung.

$$\hat{F}_i \approx 24 \cdot \varepsilon \sum_{\substack{j=1, j \neq i \\ 0 < r_{ij} \leq r_{cut}}}^N K(i, j) \quad (7.6)$$

7.1.2 Allgemeines Vorgehen

Nachdem nun die wesentlichsten Berechnungsvorschriften besprochen wurden, soll zunächst ein allgemeines Störmer-Verlet-Schema zur Bestimmung der Werte in mehreren Zeitschritten angegeben werden. Es ist in Listing 7.1 dargestellt.

```
computeForce(MP)           // Kraftberechnung (7.6)

while (currentTime < timeEnd)
{
    currentTime += δt       // Dauer eines Zeitschrittes
    computeX(MP, δt)       // neue Positionen (7.1)
    computeForce(MP)      // Kraftberechnung (7.6)
    computeV(MP, δt)      // Geschwindigkeiten (7.2)
}
```

Listing 7.1: allgemeines Störmer-Verlet-Verfahren

Die in Listing 7.1 verwendeten Funktionen `computeForce`, `computeX` und `computeV` basieren auf den entsprechenden Vorschriften (7.1, 7.2, 7.6) und berechnen die Werte für alle Partikel des Simulationsraumes. Hierfür wird die Menge der Partikel M_P als Parameter übergeben. Vorausgesetzt wird, daß der gesamte Simulationsraum S bereits in Teilräume P_l untergliedert wurde. Jeder dieser Teilräume P_l ist im Normalfall ein Würfel mit der Kantenlänge r_{cut} . Dadurch kann gewährleistet werden, daß nur Partikel der direkten Nachbarräume mit Partikeln aus P_l interagieren.

Da die Methode `computeForce` den laufzeitkritischen Teil des Verfahrens enthält⁵ und

⁴nach \hat{x} mit Hilfe des Nabla-Operators [46]

⁵Die Methoden `computeX` und `computeV` iterieren jeweils einmal über die gesamte Partikelmenge M_P und besitzen eine asymptotische Laufzeit von $\mathcal{O}(n)$. Die Laufzeit der Methode `computeForce` ist

deshalb einen guten Ansatzpunkt für mögliche Laufzeitoptimierungen bildet, ist diese in Listing 7.2 skizziert. Zunächst wird dort über alle konstruierten Teilräume P_l des Simulationsraumes S iteriert. Für alle darin enthaltenen Partikel i wird dann die aktuelle Kraft berechnet, indem die Partikel des Teilraumes und aller Nachbarräume ($Nachbar(P_l)$) betrachtet werden.

```

ForAll  $P_l \in S$ 
{
  ForAll  $i \in P_l$ 
  {
     $\hat{F}_i = \vec{0}$  // Kraftvektor auf 0
    ForAll  $P_k \in \{P_l \cup Nachbar(P_l)\}$ 
    {
      ForAll  $j \in P_k$ 
      {
         $\hat{F}_i += K(i,j)$  // Potential siehe 7.5
      }
    }
     $\hat{F}_i = 24 \cdot \varepsilon \cdot \hat{F}_i$ 
  }
}

```

Listing 7.2: computeForce Methode

Das allgemeine Schema aus Listing 7.1 muß noch etwas an die Bedingungen des *Linked-Cell*-Verfahrens angepaßt werden. Durch die Ortsveränderungen, die mit der Methode `computeX` bestimmt werden, kommt es u.U. zum Übertreten von Partikeln in andere Teilräume. Deshalb müssen entweder in der Methode `computeX` oder durch Erweiterung des Schemas diese Übergänge beachtet und Partikel zwischen einzelnen Unterräumen verschoben werden.

7.2 Sequentielle und threadparallele Implementierungen

In diesem Abschnitt werden zunächst einige grundlegende Überlegungen für die Implementierung einer sequentiellen und einer threadparallelen Variante des *Linked-Cell*-Verfahrens besprochen. Im Anschluß hieran erfolgt deren Implementierung und die Auswertung von Laufzeittests mit diesen Varianten. In Abbildung 7.2 sind einige der berechneten Simulationsergebnisse abgebildet. Die dort abgebildete Szene wird als Eingabeproblem für die durchgeführten Laufzeittests verwendet.

abhängig von der Partikelverteilung und den gewählten Parametern. Im *worst case* Fall beträgt deren Laufzeit $\mathcal{O}(n^2)$. Geht man von einer Gleichverteilung der Partikel aus, beträgt der Aufwand nur noch $\mathcal{O}(n)$.

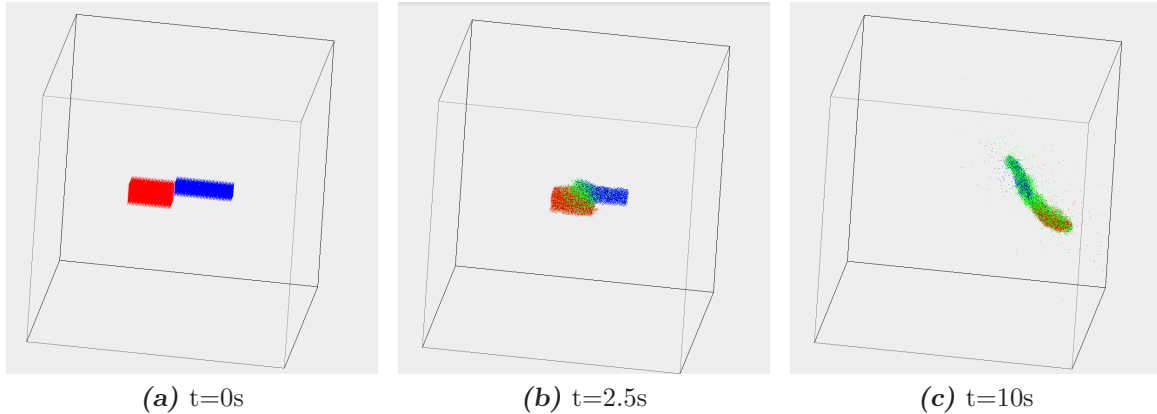


Abbildung 7.2: Simulation des Zusammenstoßes zweier Körper (150000 Partikel) mit dem *Linked-Cell*-Verfahren. Ein Zeitschritt betrug $\delta t = 0.00005s$. Die weiteren Konstanten: $\varepsilon = 5$, $\sigma = 1$ und $r_{cut} = 2.5$.

7.2.1 Berechnung der Nachbarräume

Für den Algorithmus ist die Berechnung aller Nachbarräume für einen beliebigen Teilraum notwendig. Dazu muß eine eindeutige Numerierung der einzelnen Teilräume gefunden werden. Eine solche Numerierung ist in der Abbildung 7.3 angedeutet. Der gesamte Simulationsraum S wird gleichmäßig in Würfel mit der Kantenlänge r_{cut} unterteilt. Um die Implementierung etwas zu vereinfachen, wird vorausgesetzt, daß der Simulationsraum selbst die Gestalt eines Würfels mit der Kantenlänge $l_s = \nu \cdot r_{cut}$ besitzt. Insgesamt enthält S somit ν^3 Würfel, ν Ebenen in jeder Dimension und ν^2 Elemente pro Ebene. Alle Elemente einer Ebene E_k besitzen eine ab 0 beginnende fortlaufende Numerierung α_L . Aus dieser lokalen Nummer und der Nummer der Ebene k kann ein globaler Index α_G errechnet werden.

$$\alpha_G = \alpha_L + k \cdot \nu^2 \quad (7.7)$$

Ist der Index α_G bekannt, können die Nummern der direkten Nachbarräume berechnet werden, indem analog zur Abbildung 7.3 addiert oder subtrahiert wird. Die Indizes der Würfel in der davor und in der dahinter liegenden Ebene unterscheiden sich jeweils um ν^2 . Deren Bestimmung ist daher ebenfalls leicht möglich.

An den Rändern des Simulationsgebietes gibt es entsprechend weniger Nachbarräume, was bei der Berechnung zu beachten ist. Eine andere Möglichkeit ist die Benutzung von sogenannten periodischen Randbedingungen. Hierbei werden gegenüberliegende Würfel benutzt und als direkter Nachbar klassifiziert. Konkret würde dies für den Würfel $\alpha_L = 2$ aus Abbildung 7.3 bedeuten, daß ein Nachbar der Würfel $\alpha_L = 22$ ist und ein Partikel, welcher $\alpha_L = 2$ verläßt, in $\alpha_L = 22$ wieder auftaucht. Diese Randbedingungen können im *Linked-Cell*-Verfahren beliebig definiert werden. Bei den hier betrachteten Implementierungen gelten keine periodischen Randbedingungen und aus dem Simulationsraum austretende Partikel gehen dem Gesamtsystem verloren.

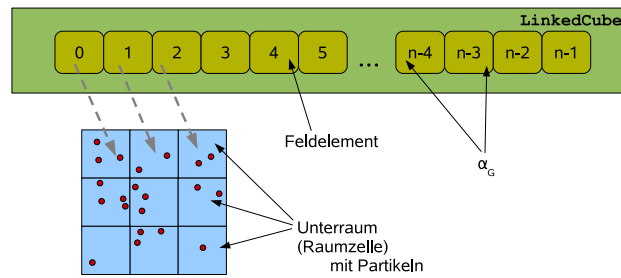


Abbildung 7.4: Skizzenhafte Darstellung der verwendeten Datenstruktur. Die Klasse `LinkedCube` enthält ein Feld mit allen Raumzellen des Simulationsgebietes. Jede dieser Raumzellen enthält einzelne Partikel.

sätzlich erfolgt am Ende jeder Iteration eine Barriersynchronisation.

In Abbildung 7.5 sind die Ergebnisse einiger Testläufe aufgetragen. Das zu simulierende Szenario ist mit dem in Abbildung 7.2 identisch. Da die Partikel nicht gleichmäßig im Simulationsraum verteilt sind, hat jeder Thread eine unterschiedliche Anzahl von Partikeln zu berechnen oder eventuell nur leere Teilräume zugeordnet bekommen. Deshalb skalieren die Tests bei diesen Anwendungsfällen nicht optimal bei Erhöhung der Threadanzahl. Der in Abbildung 7.5b für 8 Threads sichtbare Rückgang des Speedups kann auf diesen Faktor zurückgeführt werden. Mit zunehmender Anzahl von durchgeführten Iterationen ändert sich auch die Verteilung der Partikel im Raum. Im Verlauf der Simulation wird so einem oder mehreren Threads der Großteil der Berechnungen zugeordnet, da die meisten Partikel dann in einem kleinen Bereich liegen. Durch die am Ende jedes Zeitschrittes durchgeführte Barriersynchronisation, kommt es zu Wartezeiten der anderen Threads. Dies erhöht insgesamt die Ausführungszeit. Deutlich wird dieses Verhalten auch in den in Abbildung 7.6 dargestellten Speedup-Ergebnissen von Laufzeittests mit > 1000 Simulationsschritten. Für 8 Threads (Abbildung 7.6a) sind dort, nach einem starken Anstieg des Speedups (≤ 2500 Simulationsschritte), starke Schwankungen erkennbar. Erst ab etwa 40000 Simulationsschritten stabilisiert sich der Speedup Wert. Bei den anderen in dieser Abbildung dargestellten Meßreihen ist dieses Verhalten zum Teil weniger ausgeprägt.

Entgegenwirken kann man dem beschriebenen Effekt mit einer dynamischen Verteilung der Berechnungen auf die Threads. Ein solches Verfahren kann ein besseres Laufzeitverhalten der parallelen *Linked-Cell*-Anwendung erreichen, indem Raumzellen z.B. anhand der darin enthaltenen Partikel den Threads zugeordnet werden oder ein Taskpool basierter Ansatz ([59]) zur Verteilung der Raumzellen gewählt wird. Ziel dieser Verfahren ist jeweils die Minimierung von Wartezeiten einzelner Threads. An dieser Stelle sollen solche Strategien nicht weiter verfolgt werden, da diese die Untersuchung⁶ und Implementierung der nachfolgenden verteilten *Beluga* Variante erschweren, welche auf der Basis der threadparallelen Umsetzung erstellt werden soll.

⁶insbesondere die Vergleichbarkeit der Varianten

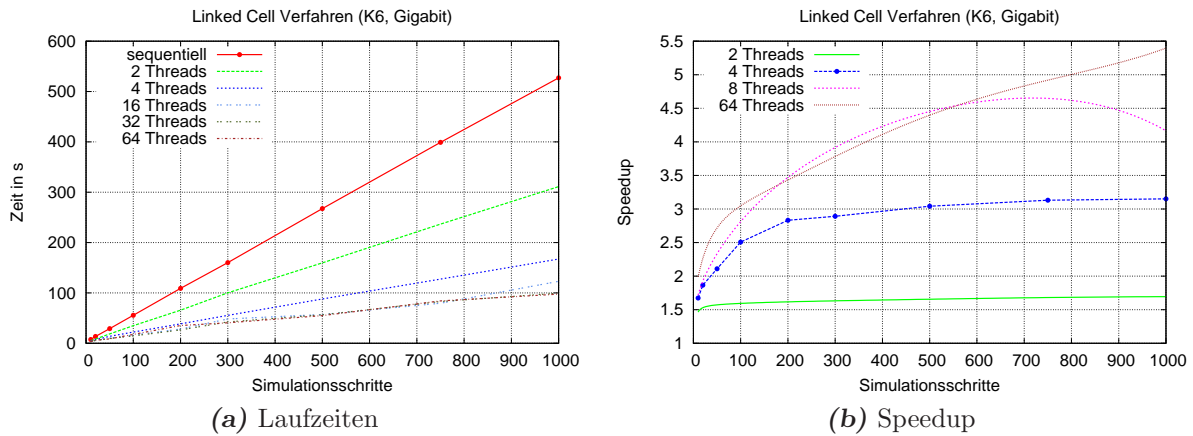


Abbildung 7.5: Laufzeiten und Speedup für eine *Linked-Cell*-Simulation. In der verwendeten Konfiguration wurde ein Knoten (hydrus) mit 4 Quadcore Prozessoren verwendet.

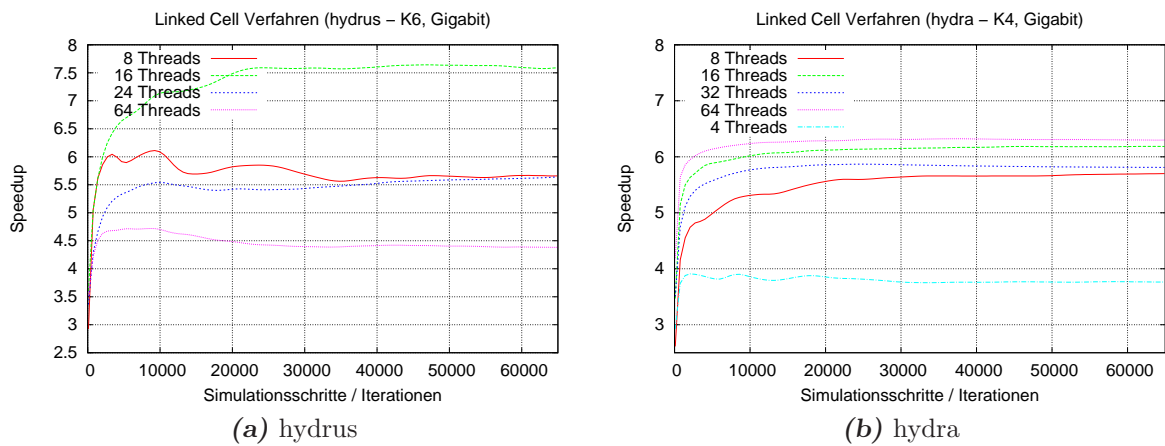


Abbildung 7.6: Ermittelte Speedupkurven für verschiedene Testläufe mit jeweils unterschiedlicher Anzahl von Berechnungsthreads. Die diesen Grafiken zugrundeliegenden Meßpunkte haben einen Abstand von jeweils 100 Iterationen.

7.3 DEE Implementierung

Aufgrund der Struktur der bisher erstellten Varianten des *Linked-Cell*-Verfahrens konnte eine verteilte *Beluga* Implementierung direkt daraus abgeleitet werden. Die verwendete Datenstruktur zur Abspeicherung der Partikel im Raum wurde durch eine DSM-Variante ersetzt. Außerdem wurden an einigen Stellen explizite Locks gesetzt, um dem DSM mitzuteilen, welche Daten erwünscht sind. Neben diesen Modifikationen waren keine tiefergehenden Änderungen notwendig. Die Datenstruktur für den DSM wird im nachfolgenden Abschnitt erläutert.

7.3.1 DSM-Datenstruktur Raum

Das für das *Beluga* DSM-Subsystem erstellte Datenobjekt (DSM-Objekt) `Raum` bildet die Funktionalität der bisher benutzten Klasse `LinkedCube` (Kapitel 7.2.2) nach. Der in ν^3 Unterräume (Würfel, Raumzellen) zerlegte Simulationsraum wird nun im DSM-Objekt `Raum` verteilt im DSM abgespeichert. Hierfür enthält `Raum` intern ein verteiltes Feld. Dort werden die einzelnen Unterräume des Simulationsgebietes abgelegt und können durch die in der Berechnungsvorschrift 7.7 vorgegebene Numerierung leicht referenziert werden. In der Abbildung 7.7 ist die Funktionsweise der DSM-Datenstruktur `Raum` skizziert. Auf einige Details soll hier kurz eingegangen werden.

Bei ihren Berechnungen greifen alle Trails gemeinsam auf das interne Feld in `Raum` zu und manipulieren nur bestimmte Bereiche, andere Bereiche des Feldes werden nur gelesen. Deshalb wird das Feld nicht in einer einzelnen DSM-Zelle abgelegt, sondern jeweils einige zusammenhängende Feldelemente als Block in einer DSM-Zelle. Dadurch müssen bei Leseanfragen nicht die gesamten Daten übertragen werden. Notwendig wird durch diese Art der Abspeicherung im DSM ein Zuordnungsmechanismus, welcher zu einem gegebenen Feldindex die entsprechende DSM-Zelle (A-D, siehe Abbildung 7.7) liefert.

Wählt man eine statische Zerlegung mit festen Blockgrößen, dann kann eine Referenz auf die benötigte DSM-Zelle leicht errechnet werden. Hier wurde aber eine dynamische Zuordnung gewählt, d.h. die Größe der Blöcke kann variieren und kann sich während der Laufzeit ändern. Um diese Funktionalität zu erreichen, wird die Aufteilung in einer weiteren DSM-Zelle (E) gespeichert. Diese enthält einen binären Baum. In den Blättern dieses Baumes sind die jeweiligen globalen Indizes eines Blockes mit der zugehörigen DSM-Zelle der Feldelemente vermerkt. Die inneren Knoten des Baumes enthalten immer Angaben zum gesamten Block im aufspannenden Unterbaum. Jedes Kind eines inneren Knotens halbiert den Block⁷. Dadurch ist der Baum immer ausbalanciert und bietet eine Zugriffszeit zum Auffinden einer DSM-Zelle von $\mathcal{O}(\log n)$.

7.3.2 Auswertung

In diesem Abschnitt sollen die durchgeführten Laufzeittests diskutiert werden. Für diese Betrachtungen werden meist Messungen der homogenen Laufzeitkonfiguration *K5* mit

⁷sofern es kein Blatt ist

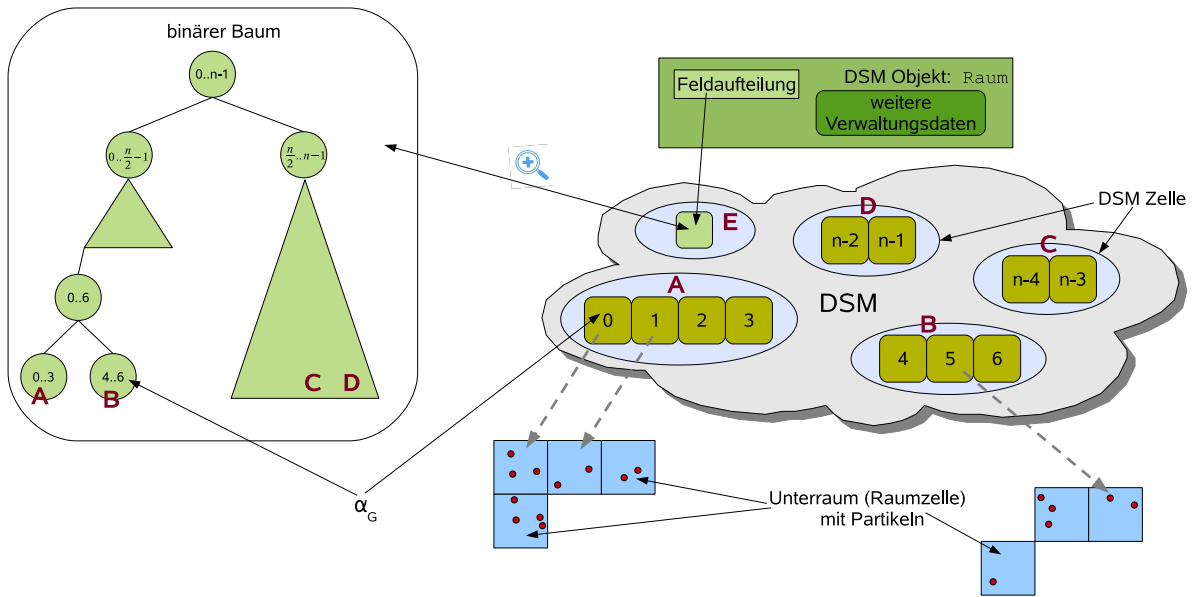


Abbildung 7.7: Funktionsweise des DSM-Objektes Raum für das *Linked-Cell*-Verfahren. Dieses benutzt ein verteiltes Feld zur Abspeicherung der Partikel und der Teilräume des Simulationsgebietes. Zum Auffinden einzelner Feldelemente ist eine zusätzliche Speicherung der *Feldaufteilung* notwendig. Im Fall des Raum Objektes geschieht dies mit einem binären Baum, dessen Blätter die Namen der entsprechenden DSM-Zellen enthalten.

Spread-DSM und *WriteBack*-Protokoll herangezogen. Abweichungen hiervon werden entsprechend dokumentiert. Als Eingabeproblem diente hier wieder das in Abbildung 7.2 dargestellte Szenario mit 15000 Partikeln.

Lokale Konfiguration

In Abbildung 7.8 sind Ergebnisse von Laufzeittests auf Konfigurationen mit jeweils nur einem Hostsystem⁸ dargestellt. Alle Trails werden in diesen Konfigurationen auf einem einzelnen Hostsystem ausgeführt und eine Netzwerkkommunikation beim Zugriff auf den DSM entfällt.

Die Zeit für eine Iteration wurde aus der Gesamtlaufzeit und den insgesamt durchgeführten Simulationsschritten ermittelt. Da am Beginn einer Simulation einige Datenstrukturen im DSM initialisiert werden, ist eine leichte Abnahme der einzelnen Iterationszeiten bei zunehmender Anzahl von durchgeführten Simulationsschritten erkennbar. Eine Parallelisierung durch die Benutzung mehrerer Trails führt zu einer Erhöhung der Laufzeiten. In diesen Fällen muß der DSM konkurrierende Zugriffe auf die lokalen Daten verhindern und steuern. Dies führt zu einer Sequentialisierung von großen Teilen der Berechnungen. Der so entstehende Overhead ist viel höher als der Zeitgewinn, der durch die parallele Verarbeitung entsteht. Dies schlägt sich direkt im Laufzeitverhalten der Tests nieder.

⁸also lokalem DSM

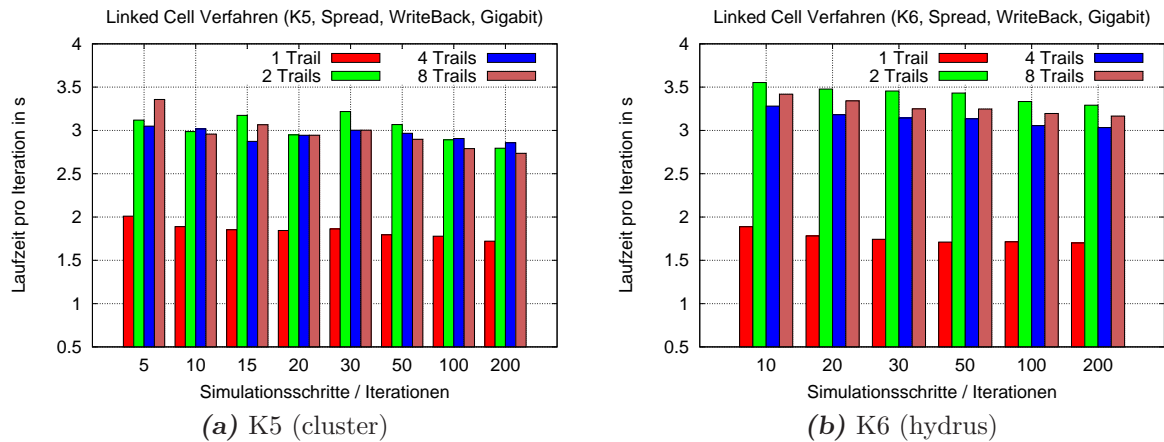


Abbildung 7.8: Dargestellt sind die Laufzeiten pro Iterationsschritt (gemittelt) für Konfigurationen mit nur einem Knoten. Erkennbar ist, daß mit zunehmender Anzahl von Iterationen die durchschnittlich benötigte Zeit sinkt.

Verteilte Konfiguration

Einige Ergebnisse von Laufzeittests, unter Verwendung mehrerer Hostsysteme der Konfiguration *K5* und *Spread*-DSM, sind in Abbildung 7.9 zu finden. Für einen besseren Vergleich ist dort jeweils auch eine Meßreihe mit nur einem Trail enthalten. Erkennbar wird auch hier, daß die Erhöhung der Anzahl der lokalen Trails auf einem System keine Laufzeitvorteile bringt. Verteilt man die Trails hingegen auf mehrere Hosts, erhält man akzeptable Speedup Werte (Abbildung 7.10b) und die benötigte Zeit pro Iteration sinkt (Abbildung 7.9b).

In Abbildung 7.10a sind die Laufzeiten von Simulationsläufen (200 Iterationen) mit den verschiedenen DSM-Implementierungen dargestellt. Außerdem enthält diese Abbildung noch die Laufzeiten vergleichbarer threadparalleler Tests mit einem oder vier Threads (vgl. dazu Abbildung 7.5).

Für lokale Berechnungen mit einem Trail sind die ermittelten Laufzeiten der *Central*- und *Spread*-Varianten in etwa in der gleichen Größenordnung wie die Laufzeiten einer äquivalenten sequentiellen Simulation (1 Thread). Die Variante für den *Central*-DSM weist aufgrund eines geringeren Overheads⁹ die kleinste lokale Laufzeit der betrachteten DSM-Varianten auf und wird deshalb als Grundlage für alle nachfolgenden Berechnungen des Speedups (siehe Kapitel 5.3.4 (S.174)) verwendet.

Im verteilten Ausführungsfall mit vier Trails, welche auf zwei Hostsysteme aufgeteilt werden, erreicht die Variante mit *Spread*-DSM nahezu die identischen Laufzeiten wie bei den im Vorfeld durchgeführten Laufzeittests mit einem Trail. Die *Central*- und *TSpaces*-Varianten können diese Ergebnisse nicht erzielen. Beim *Spread*-DSM lassen sich die Re-

⁹bei der lokalen Betrachtung

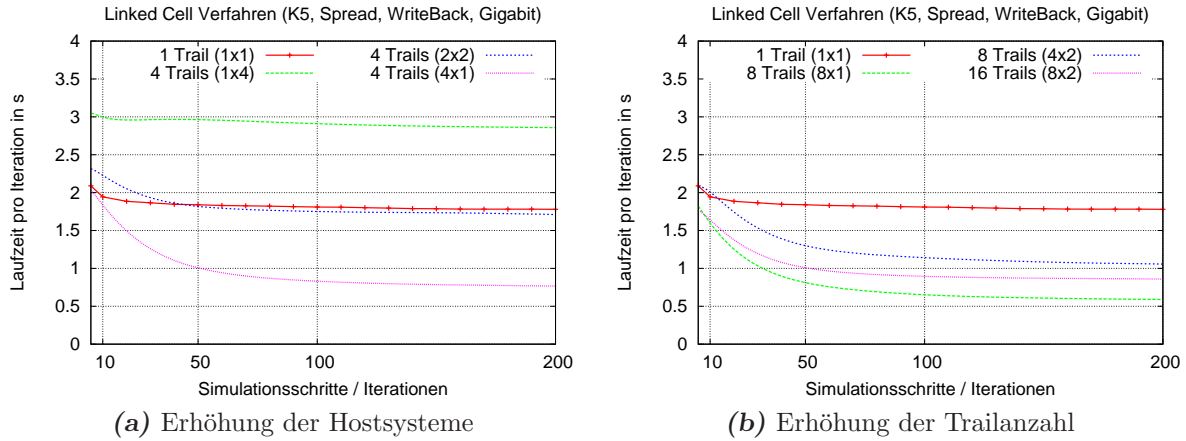


Abbildung 7.9: Ergebnisse der Konfiguration *K5* mit jeweils unterschiedlicher Aufteilung der Trails auf die Hostsysteme. Die in den Klammern stehenden Angaben der Legende geben Auskunft über die genaue Verteilung der Trails und sind als (Hostsysteme x lokale Trails) zu lesen.

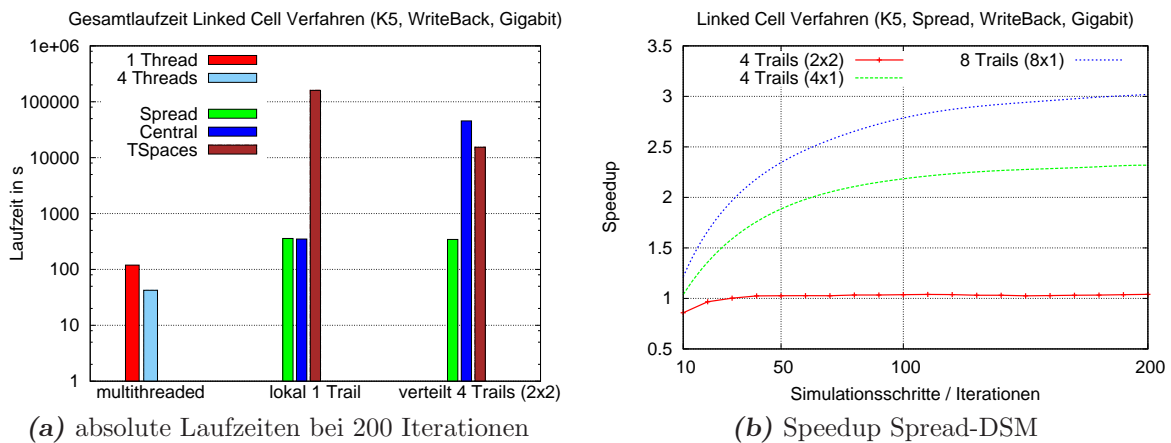


Abbildung 7.10: Ausgewählte Speedups der Laufzeittests mit *Spread*-DSM und Gegenüberstellung der ermittelten Laufzeiten mit unterschiedlichem DSM-Subsystem. Die Angaben zur Aufteilung der Trails können wieder als (Hosts x lokale Trails) interpretiert werden.

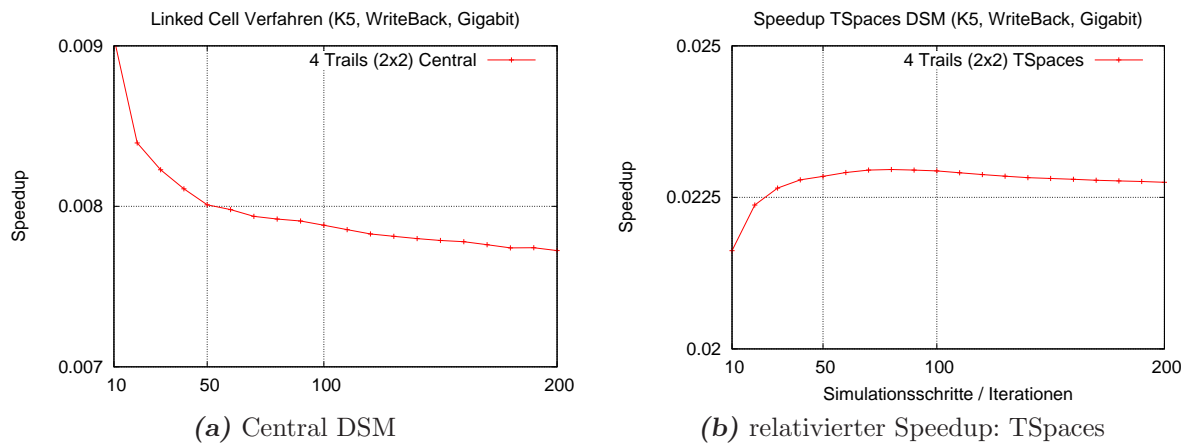


Abbildung 7.11: Dargestellt sind die ermittelten Speedups des implementierten verteilten Linked-Cell-Verfahrens mit vier Trails auf zwei Hostsystemen. Als Grundlage der Berechnung dienten die Ergebnisse des lokalen *Central*-DSM-Laufzeittests mit einem Trail (Abbildung 7.10a). Aufgrund der hohen Laufzeiten wurde nur eine geringe Zahl von Laufzeittests für diese Konfigurationen durchgeführt.

sultate vor allem auf die Verteilung der DSM-Zellen auf die einzelnen Knoten, verbunden mit dem weitestgehend konfliktfreien Zugriffsmuster des *Linked-Cell*-Verfahrens, zurückzuführen. Meist werden die verschiedenen Raumzellen des Simulationsraumes nur von einem Hostsystem (bzw. den darauf ausgeführten Trails) benötigt und liegen im Falle des *Spread*-DSM dann oftmals lokal vor. Nur an den Ränder des jeweiligen Simulationsgebietes eines Trails müssen nach jedem Simulationsschritt die entsprechenden Daten ausgetauscht werden. Einige Speedup Werte der *Spread*-DSM-Variante mit einer Verteilung der Trails auf bis zu acht Hostsysteme sind in der Abbildung 7.10b dargestellt. Bei der Verteilung der Berechnungen auf acht Hostsysteme (mit jeweils einem Trail) wird in diesem Fall ein Speedup von drei erreicht.

Die ermittelten Speedups der *Central*- und *TSpaces*-Varianten sind in der Abbildung 7.11 dargestellt. Aufgrund der hohen Laufzeitunterschiede zwischen den lokalen und verteilten Laufzeiten¹⁰ sind diese Werte sehr niedrig und in diesen Fälle empfiehlt sich die lokale Durchführung der Simulationsberechnungen.

7.4 Fazit

Das *Linked-Cell*-Verfahren für die *DEE* Umgebung konnte direkt aus einer sequentiellen bzw. threadparallelen Variante adaptiert werden. Neben einigen kleineren Veränderungen am Quellcode, wurde nur die Erstellung eines DSM-Objektes zur Ablage der Daten notwendig. Der Aufwand für die Portierung der threadparallelen Implementierung in eine

¹⁰Als sequentielle Laufzeit wird die Laufzeit des lokalen (1 Trail) *Central*-Tests verwendet.

trailparallele Implementierung war daher gering.

Die gemessenen Laufzeiten für verteilte Berechnungen dieser trailparallelen Implementierung liefern bereits akzeptable Speedup-Werte, sofern der *Spread*-DSM verwendet wird. Die *Spread*-DSM-Variante unterstützt das dynamische Auslagern von häufig verwendeten Daten auf andere Hostsysteme, wodurch die mittleren Zugriffszeiten auf diese Daten sinken und die Berechnungen insgesamt schneller abgeschlossen werden können.

Eine weitere Optimierung der für die *Beluga* Laufzeitumgebung erstellten Variante des *Linked-Cell*-Verfahrens ist denkbar. Dabei sind in erster Linie Verbesserungen beim Zugriff auf die im DSM abgelegten Daten zu nennen. Das DSM-Objekt *Raum* könnte dahingehend verbessert werden, daß die von mehreren Trails benötigten Unterräume¹¹, in gesonderte DSM-Zellen ausgelagert werden. Alle anderen DSM-Zellen würden dann Partikel bzw. Teilräume enthalten, die nur von einzelnen Trails eines Hostsystemes benötigt werden. Solche DSM-Zellen können dann z.B. vom *Spread*-DSM auf das lokale Hostsystem verschoben werden und die resultierenden Zeiten beim Zugriff auf diese DSM-Zellen fallen entsprechend.

Aufgrund der bisherigen Beobachtungen verspricht eine Senkung der durchschnittlichen Antwortzeiten des DSM-Subsystems ebenfalls gute Erfolgsaussichten. Hierbei könnte z.B. eine verfeinerte Auslagerungsstrategie des *Spread*-DSM zum Einsatz kommen, womit diese Laufzeitverbesserungen möglich wären.

¹¹Gemeint sind dabei jene Unterräume, die sich im Grenzbereich der Simulationsgebiete zweier Trails befinden.

Kapitel 8

Barnes-Hut-Algorithmus

Die Körper wären nicht schön, wenn sie sich nicht bewegten.

(Johannes Kepler)

In diesem Kapitel werden die im Rahmen dieser Arbeit erstellten Implementierungen des *Barnes-Hut*-Verfahrens beschrieben und in verschiedenen Laufzeitkonfigurationen untersucht. Das Kapitel gliedert sich dabei wie folgt: Nach einer kurzen Einführung und Beschreibung des *Barnes-Hut*-Algorithmus wird zunächst die hierzu erstellte sequentielle Variante besprochen. Dabei werden die einzelnen Phasen des Verfahrens näher erläutert und deren Umsetzung beschrieben. Eine, ausgehend von der sequentiellen Variante des Verfahrens, erstellte threadparallele Implementierung wird im Anschluß hieran in ihrem Laufzeitverhalten untersucht und in verschiedenen Tests gegenübergestellt.

Diese Betrachtungen und die daraus gewonnenen Erkenntnisse bilden die Grundlage der dann implementierten und untersuchten trailparallelen Varianten für die *Beluga* Laufzeitumgebung. Einen hohen Stellenwert nimmt hierbei die Betrachtung der verteilten Baum-Datenstruktur ein, welche für die effiziente Umsetzung des *Barnes-Hut*-Verfahrens sehr wichtig ist und als DSM-Objekt implementiert wurde. Den Abschluß dieses Kapitels bilden verschiedene Untersuchungen zur Optimierung der erstellten trailparallelen *Barnes-Hut*-Variante und eine Analyse des Laufzeitverhaltens.

8.1 Beschreibung des Verfahrens

Wie in Kapitel 6.2.2 (*S.180*) beschrieben, ist der *Barnes-Hut*-Algorithmus ein Baumverfahren zur Berechnung von langreichweitigen Potentialen. Sein Ursprung liegt in der Simulation von Gravitationskräften in astrophysikalischen Modellen, die mehrere Millionen oder Milliarden Partikel enthalten können. Für eine exakte Simulation müssen bei der Berechnung eines einzelnen Partikels alle anderen Partikel des Modelles betrachtet werden, was zu einer asymptotischen Gesamtlaufzeit von $\mathcal{O}(n^2)$ für einen Simulationsschritt führt.

Beim *Barnes-Hut*-Verfahren wird eine Näherung berechnet, indem räumlich nahe beieinander liegende Partikel zusammengefaßt werden. Dieses Vorgehen beruht auf der Tatsache, daß die Wechselwirkungen zwischen den betrachteten Partikeln mit steigender

Entfernung immer schwächer werden und ab einem gewissen Punkt durch Approximationen ersetzt werden können. Der dabei entstehende Fehler ist vernachlässigbar [16]. Beispielsweise können die Sterne, Planeten und sonstigen Teilchen einer Galaxie zusammengefaßt und im weiteren Verlauf der Berechnungen als einzelnes Pseudopartikel verwendet werden (Abbildung 8.1). Laufzeittechnisch ergeben sich so enorme Einsparpotentiale und die asymptotische Laufzeit des Verfahrens sinkt auf $\mathcal{O}(n \log n)$.

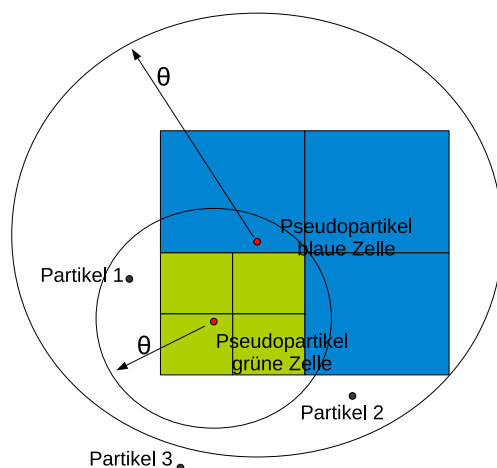


Abbildung 8.1: Liegt ein Partikel innerhalb des Einflußgebietes θ (siehe Kapitel 8.2.1 (S.204)) eines Partikels oder Pseudopartikels, müssen alle in den Zellen enthaltenen Teilchen betrachtet werden. Für den Fall, daß ein Partikel weiter entfernt ist, können die Berechnungen mit Hilfe des jeweiligen Pseudopartikel approximiert werden. Bei der Berechnung von *Partikel 3* wird demnach der Pseudopartikel der blauen Zellen benutzt. Während für den *Partikel 2* der Pseudopartikel der grüne Zelle genügt, müssen für *Partikel 1* alle Partikel der grünen Zellen verwendet oder diese weiter verfeinert werden.

Für die Speicherung des Simulationsmodelles im Rechner wird eine hierarchische Baumstruktur verwendet, deren Wurzel den gesamten Simulationsraum enthält. Jeder Kindknoten (oder jede Zelle) enthält einen Teilraum des Vaterknotens. Im dreidimensionalen Fall existieren immer 8 Kindknoten, da jede Dimension halbiert wird. Die Verfeinerung wird erst beendet, wenn in jedem Blattknoten höchstens ein Partikel existiert. Die Eigenschaften der inneren Knoten bzw. Pseudopartikel eines Baumes ergeben sich aus der Menge der im aufspannenden Unterbaum enthaltenen Partikel. Beispielsweise kann die Lage \vec{p} des Pseudopartikels im Raum als *gewichtetes Massezentrum* aufgefaßt und mit der Formel 8.1 aus den Daten der acht Kindknoten (z.B. Masse m) berechnet werden.

$$\vec{p} = \frac{1}{m_g} \cdot \sum_{j=0}^7 \vec{p}_{sohn_j} \cdot m_{sohn_j} \quad \text{mit } m_g = \sum_{j=0}^7 m_{sohn_j} \quad \text{und } \vec{p} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad (8.1)$$

Hierdurch wird erreicht, daß die Position eines Pseudopartikels nicht direkt mit dem

Mittelpunkt des untergliederten Teilraumes übereinstimmt und sich nach der Lage der Mehrzahl der Partikel dieses Teilraumes richtet (Abbildung 8.1). Falls keine Kinder existieren, der Knoten also ein Blatt repräsentiert, werden die Eigenschaften direkt vom enthaltenen Partikel geerbt.

Der von *J. Barnes* und *P. Hut* vorgestellte Algorithmus [16] läßt sich grob in 3 Phasen aufteilen, welche jeweils iterativ durchlaufen werden. Die **1.Phase** kann kurz als Baumaufbau bzw. Baumaktualisierung charakterisiert werden. Der Baumaufbau ist nur beim Starten des Verfahrens notwendig. Hierbei wird der Simulationsraum in kleinere Unterräume aufgeteilt und alle Partikel in den Baum eingefügt. Im weiteren Verlauf wird dieser Baumaufbau durch eine Baumaktualisierung ersetzt, die nur einzelne Partikel im Baum verschiebt und eventuell einzelne Unterbäume entfernt oder neu erstellt¹. In der **2.Phase** des *Barnes-Hut*-Algorithmus werden in einem *bottom-up* Lauf über den Baum alle inneren Knoten des Baumes berechnet bzw. aktualisiert, d.h. es wird aus der Summe der Massen aller enthaltenen Partikel das Massezentrum für den Knoten ermitteln. Anschließend können in der **3.Phase** die einwirkenden Kräfte auf jedes einzelne Partikel in einem *top-down* Lauf über den Baum berechnet werden. Aus der so ermittelten Krafteinwirkung auf das Partikel können dessen neue Position und die Geschwindigkeit im aktuellen Simulationsschritt bestimmt werden.

Die soeben beschriebenen drei Phasen des *Barnes-Hut*-Algorithmus werden im nachfolgenden Abschnitt detaillierter betrachtet.

8.2 Sequentielle und threadparallele Varianten

Abbildung 8.2 zeigt einige Ergebnisse einer Simulation, welche mit der nachfolgend beschriebenen sequentiellen Implementierung berechnet wurden. Wie man leicht sieht, sind selbst noch nach drei simulierten Tagen (dies entspricht etwa 26.000.000 Simulationsschritten) die Raumstrukturen der Ausgangssituation erkennbar. Bei den späteren Betrachtungen wird diese Erkenntnis bei den Überlegungen zur Lastbalancierung wieder aufgegriffen². Zunächst soll noch einmal auf einige Details des *Barnes-Hut*-Algorithmus eingegangen werden und die einzelnen Phasen im Zusammenhang mit der Erstellung der sequentiellen Variante näher erläutert werden.

8.2.1 sequentielle Implementierung

Bei der Implementierung einer sequentiellen Variante kann man sich direkt an der ursprünglichen Beschreibung des Algorithmus orientieren [16]. Die hier vorgestellte sequentielle Variante weicht jedoch in einigen Details von dieser Originalvorgabe ab. Zunächst sollen diese Änderungen kurz erwähnt und das allgemeine Vorgehen erläutert werden.

¹Das Erstellen oder Entfernen von Unterbäumen ist immer dann notwendig, wenn Partikel in Bereiche des Simulationsraumes eintreten, welche sich in einem anderen Ast des Baumes befinden.

²Für die in dieser Arbeit durchgeführten Betrachtungen ist eine implizite Lastbalancierung durch Konditionierung des Eingabeproblems durchaus sinnvoll.

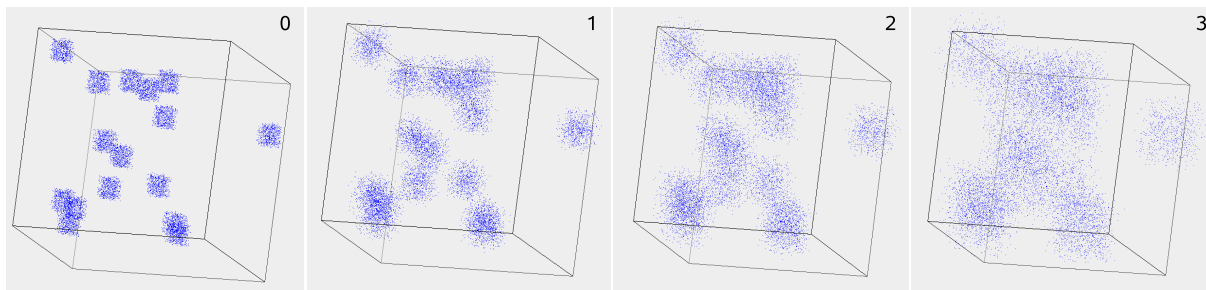


Abbildung 8.2: Dargestellt sind die Ergebnisse einer Barnes-Hut-Simulation mit 10.000 Partikeln für die Ausgangssituation (0) sowie nach ca. einem Tag (1), 2 Tagen (2) und 3 Tagen (3) Simulationszeit. Mit den verwendeten Simulationsparametern ($\delta t = 0,01s$) sind 8.640.000 Schritte pro simuliertem Tag notwendig.

Eine Beschreibung von relevanten Zusammenhängen erfolgt jeweils im Rahmen nachfolgender Einzelbetrachtungen.

Die Phase 1 (Baumaufbau) und die Phase 2 (Berechnung der Pseudopartikel) wurden bei der sequentiellen Implementierung zusammengefaßt. Dies bedeutet, daß nun in jeder Iterationen der notwendige Baum neu erzeugt wird und keine Baumaktualisierung erfolgt.

```

1  while (currentTime < endTime)
2  {
3      buildTree() ; // Phase 1 + 2
4
5      computeForce() ; // Phase 3
6
7      currentTime += dt ;
8  }
```

Listing 8.1: Hauptprogramm Barnes-Hut

In Listing 8.1 ist die Hauptschleife der sequentiellen *Barnes-Hut*-Implementierung kurz skizziert. In jedem Iterationsschritt wird zuerst der Baum neu erstellt (`buildTree`, Zeile 3) und berechnet. Dieses Vorgehen hat einen geringen Einfluß auf die resultierende Gesamtlaufzeit, da der überwiegende Teil der Simulationsberechnungen in der Methode `computeForce()` (Phase 3) durchgeführt (siehe dazu auch Abbildung 8.4) wird. Diese Routine bestimmt iterativ für jeden Partikel die darauf einwirkende Kraft. Um aus dieser Größe die neuen Geschwindigkeiten und Positionen der Partikel im Raum zu erhalten, ist in `computeForce` noch eine Zeitintegration mittels des Störmer-Verlet Verfahrens (vgl. dazu Kapitel 7.1) enthalten.

Kombinierter Baufbau

Der in Listing 8.2 abgebildete Pseudocode zeigt nochmals die prinzipielle Arbeitsweise der Baufbauphase (Phasen 1 und 2). Hierzu werden, ausgehend von der Wurzel des Baumes, alle Partikel in die entsprechenden Kindknoten der Wurzel einsortiert. Jedes dieser Kinder repräsentiert hierbei einen Teilraum (Raumzelle) des im Wurzelknoten verwalteten gesamten Simulationsraumes. Gleichzeitig kann beim Einfügen eines Partikels in einen Kindknoten dessen Gesamtmasse berechnet werden. Nachdem alle Partikel zugeordnet wurden, werden alle Blätter des Baumes solange unterteilt, bis jeder Blattknoten höchstens einen Partikel enthält (*Zeile 11*). Hierzu werden für den aktuell betrachteten Knoten `node` acht Kindknoten erzeugt und die im Knoten `node` enthaltenen Partikel auf diese Kinder verteilt (*Zeile 13 ff.*). Bei der Zuordnung von Partikeln (Einfügen) zu einem der Kindknoten können wieder gleichzeitig die Gesamtmasse und das Massezentrum des Ursprungsknotens `node` aktualisiert werden (*Zeile 21*).

```

1  buildTree()
2  {
3      stack.push( root with all particle )
4
5      while (stack.hasElements)
6      {
7          TreeNode node = stack.pop ;
8
9          // Falls mehr als 1 Partikel im Knoten existiert ,
10         // den Baum weiter verfeinern .
11         if (node.numberOfParticle > 1)
12         {
13             forAll (Particle p in node)
14             {
15                 // Anhand der Partikelkoordinaten das entsprechende
16                 // Kind auswählen .
17                 TreeNode child = node.getChildFor( p.centre ) ;
18
19                 // Masse und Massenmittelpunkt der Kindknotens
20                 // aktualisieren .
21                 child.compute( p ) ;
22
23                 // Partikel in die Liste des Kindknotens einfügen .
24                 child.add( p ) ;
25
26                 stack.push( child ) ;
27             }
28         }
29     }
30 }
```

Listing 8.2: Baufbau und Berechnung der Pseudoknoten im Barnes-Hut-Algorithmus

Asymptotisch betrachtet beträgt die Laufzeit dieser Baumaufbauphase $\mathcal{O}(n \cdot \log n)$, da für jede Baumstufe (Tiefe: $\log n$) maximal n Partikel betrachtet werden müssen.

Kraftberechnung

Bei der Kraftberechnung wird die Wechselwirkung jedes Partikels mit allen anderen Partikeln ermittelt. Der im vorangegangenen Bearbeitungsschritt erzeugte Baum wird deshalb für jeden Partikel (ausgehend von der Wurzel) solange hinabgestiegen, bis ein Pseudoknoten hinreichend weit entfernt ist oder ein Blatt erreicht wurde.

Dieses auf der Grundlage eines Abstandes zu berechnende Abbruchkriterium wird im Pseudocode (Listing 8.3) durch die Methode `breakCriteria(node, p)` bestimmt (Zeile 13). Als Grundlage für diese Berechnungen wird im allgemeinen die Formel:

$$\frac{diam}{r} \geq \theta$$

verwendet. Der Parameter *diam* ist hierbei als Maß für die Ausdehnung des Raumbereiches des betrachteten Pseudopartikels *node* und der Parameter *r* als Maß für den Abstand zwischen dem Partikel *p* und dem Pseudopartikel *node* zu interpretieren. Zur Bestimmung der Parameter *diam* und *r* können die in Abbildung 8.3 dargestellten Varianten benutzt werden. In den nachfolgenden Tests wird stets die dort skizzierte *Variante 2* benutzt und der Parameter θ auf 0.5 gesetzt (analog zu [46]). Außerdem wurde für eine bessere Übersicht das zuvor erwähnte Störmer-Verlet-Verfahren nicht im Pseudocode dargestellt. In den Implementierungen ist es, wie in Kapitel 7.1 (S.183) beschrieben, enthalten.

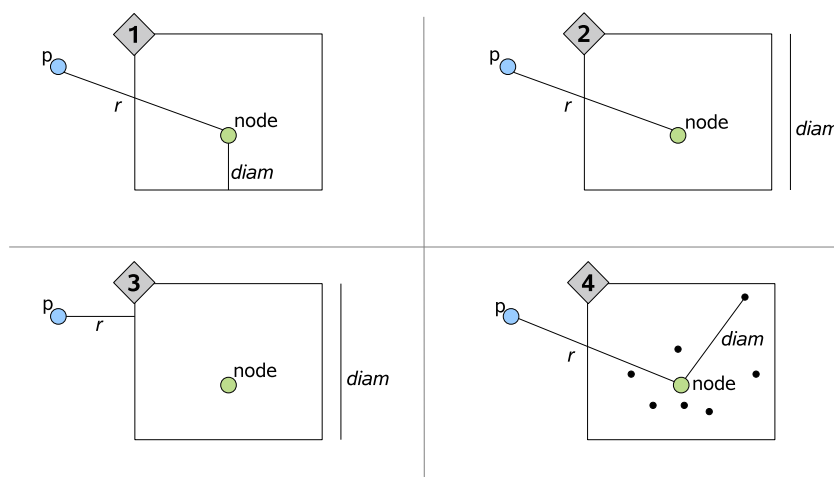


Abbildung 8.3: Mögliche Varianten zur Bestimmung der Werte von *diam* und *r*. Zur Berechnung des Abstandes *r* wird jeweils der Massenmittelpunkt des Pseudoknotens *node* gewählt. In der Veröffentlichung von Barnes und Hut [16] wird die Variante 2 mit $\theta \sim 1$ vorgeschlagen.

```

1  forAll ( Particle p )
2  {
3      // mit der Wurzel starten
4      stack.push( root ) ;
5      while ( !stack.isEmpty() )
6      {
7          TreeNode node = stack.pop() ;
8          if ( node == Leaf )
9          {
10             // Wechselwirkung zwischen aktuellem Partikel und Blatt
11             p.computeForce( node ) ;
12         }
13         else if ( breakCriteria( node, p ) )
14         {
15             // Wechselwirkung aktueller Partikel <-> Pseudopartikel
16             p.computeForce( node ) ;
17         }
18         else // nicht weit genug entfernt
19         {
20             // alle Kinder des Knotens auf den Stack
21             forAll ( Child c of node )
22             {
23                 stack.push( c ) ;
24             }
25         }
26     }
27 }

```

Listing 8.3: Kraftberechnung im Barnes-Hut-Verfahren

8.2.2 Threadparallele Variante

Ausgehend von der im vorangegangenen Abschnitt erstellten sequentiellen Variante des *Barnes-Hut*-Algorithmus wurde eine threadparallele Versionen des Verfahrens implementiert. Die grundlegende Methode des parallelen Vorgehens unterscheidet sich dabei kaum von der sequentiellen Umsetzung. Zur Beschreibung der wesentlichen Teile soll hier wieder mit dem Hauptprogramm begonnen werden.

```

1  while ( currentTime < endTime )
2  {
3      buildTree() ; // Phase 1 + 2
4      barrier() ;
5      computeForce() ; // Phase 3
6      barrier() ;
7      currentTime += dt ;
8  }

```

Listing 8.4: Hauptprogramm des threadparallelen Barnes-Hut-Verfahrens

Den einzigen Unterschied zur sequentiellen Variante in Listing 8.1 bilden die Zeilen 4 und 6, da nach jeder Phase eine Synchronisation der Threads notwendig ist. Anderenfalls könnte es vorkommen, daß einige Threads bereits mit der Kraftberechnung beginnen, obwohl der Baumaufbau noch nicht vollständig abgeschlossen wurde.

Da die beiden Methoden `buildTree` und `computeForce` aus der sequentiellen Variante einen *Stack* für die Iteration über die Elemente verwenden (*vgl. dazu Listing 8.2 und Listing 8.3*), können diese Methoden ohne größere Änderungen übertragen werden. Die beteiligten Threads teilen sich in diesen Fällen die Datenstruktur des *Stacks* und entnehmen bzw. fügen dort die noch nicht berechneten Knoten ein. Änderungen wurde nur bezüglich der nun notwendigen Synchronisation und Koordination notwendig, insbesondere mußten die Zugriffe auf den verwendeten *Stack* geschützt³ werden. Daher wurde eine veränderte Variante `safeStack` mit implizitem `synchronized` Schutz verwendet.

```

buildTree ()
{
    // ein ausgezeichneter Thread initialisiert den Stack
    if (me == master)
        safeStack.push( root with all particle )

    barrier () ;

    while (safeStack.hasElements)
    {
        TreeNode node = safeStack.pop ;

        // mehr als 1 Partikel -> Baum weiter verfeinern
        if (node.numberOfParticle > 1)
        {
            forAll (Particle p in node)
            {
                // anhand der Partikelkoordinaten das entsprechenden
                // Kind auswählen
                TreeNode child = node.getChildFor( p centre ) ;

                // Masse und Massenmittelpunkt des Kindknotens
                // berechnen und Partikel p einfügen
                child.computeAndAdd( p ) ;

                safeStack.push( child ) ;
            }
        }
    }
}

```

Listing 8.5: verteilter Baumaufbau mit Berechnung der Pseudoknoten im Barnes-Hut

³siehe hierzu auch Kapitel A.2 (S.222)

Erkennbar ist in Listing 8.5, daß es keine feste Zuordnung von Partikeln oder Knoten auf die einzelnen Threads gibt. Vielmehr wird durch die Benutzung des *Stacks* eine ausgewogene Lastbalancierung erreicht. Allerdings muß man nun davon ausgehen, daß ein einzelner Thread in jeder Iteration immer andere Partikel berechnet. Daraus resultieren schwer vorhersagbare Speicherzugriffe, die auf der Hardwareebene zu längeren Wartezeiten aufgrund von vielen Cache-Fehlzugriffen führen können. Eine Verbesserung könnte deshalb für jeden Thread einen eigenen *Stack* benutzen und erst wenn dieser lokale *Stack* keine Daten mehr enthält, mittels einer Taskstealing-Strategie [59] auf die Daten anderer Threads zugreifen. Auf die Anwendung solcher Praktiken wird verzichtet, da hierdurch die Komplexität der Implementierung erhöht und deshalb die direkte Portierung in eine trailparallele Variante erschwert wird.

Für die Methode `computeForce` gelten ähnliche Aussagen und die erforderlichen Änderungen am threadparallelen Sourcecode sind vergleichbar (*barrier*, *safeStack*).

Laufzeitmessungen

In Abbildung 8.4 sind einige Meßergebnisse der sequentiellen und threadparallelen Implementierungen für 10 Iterationsschritte dargestellt. Während sich die im oberen Teil der Grafik dargestellte Gesamtlaufzeit auf alle Threads bezieht, wurden die Ergebnisse für die einzelnen Phasen nur für einen Thread erfaßt. Der mit *Rest* bezeichnete Anteil enthält u.a. auch Wartezeiten die bei der Benutzung von mehreren Threads für die Synchronisation notwendig werden. Diese Restzeit ist zum Teil sehr hoch, was aber mit Sicherheit auf eine zu geringe Anzahl von Partikeln und somit nicht ausreichender Lastbalancierung zurückzuführen ist⁴. Gut sichtbar ist der Umstand, daß die Phase 3 (Kraftberechnung,

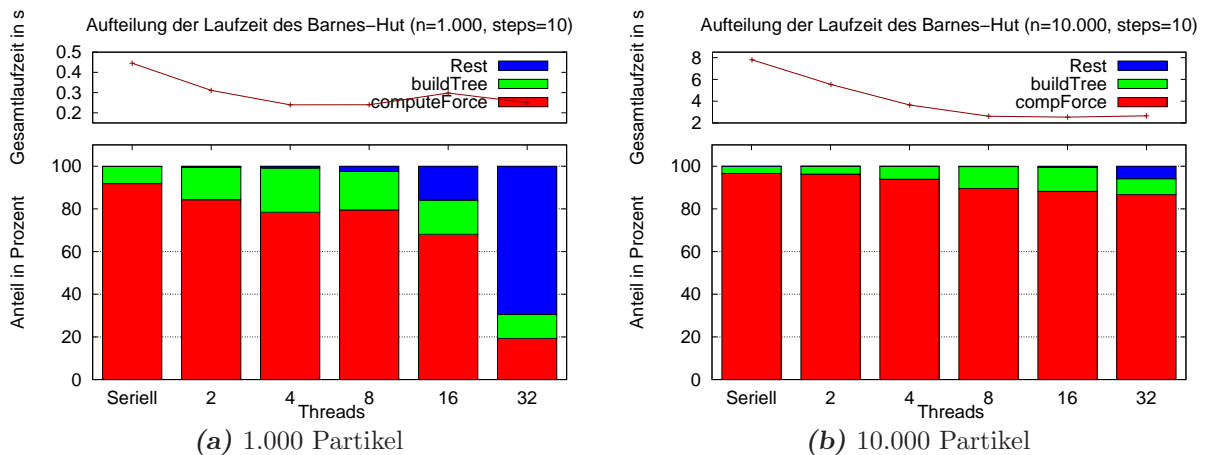


Abbildung 8.4: Verteilung der Rechenzeit eines Threads auf die einzelnen Phasen des *Barnes-Hut*-Algorithmus für verschiedene Durchläufe auf *Hydra* (16 Prozessorkerne) mit jeweils 10 Iterationen.

⁴Sehr deutlich ist dies in Abbildung 8.4a für 32 Threads zu erkennen.

`computeForce()` den größten Teil der Laufzeit benötigt. Für die weitere Arbeit erscheint es deshalb sinnvoll, Optimierungen speziell für diesen Teil des Verfahrens auszulegen.

8.3 DEE Variante

Aus den in der sequentiellen und der threadparallelen Variante gewonnenen Erkenntnissen wurde zunächst ein Programm unter Verwendung des *Beluga* DSM erstellt. Da das Programm prinzipiell auf verschiedenen Knoten, d.h. verteilt ausgeführt wird, ergeben sich einige neue Fragestellungen, die im wesentlichen auf eine bessere Lastbalancierung und Reduzierung der DSM-Zugriffe hinauslaufen. Diese Thematik und die im Vorfeld getroffenen Entscheidungen der Implementierung sollen im nun folgenden Abschnitt besprochen werden.

8.3.1 Vorüberlegungen

Jeder Trail sollte nur für einen Teil der Partikel und einen Ast des Octree zuständig sein. Idealerweise sollten sich diese Partikel auch im zugehörigen Ast befinden, damit Phase 2 und Phase 3 des Verfahrens möglichst identische DSM-Zugriffe produzieren. Die Partikelvergabe, d.h. die Aufteilung der Partikel auf die einzelnen Trails, muß sich deshalb an der räumlichen Verteilung der Partikel im Simulationsraum orientieren. Bei einer statischen und partikelbasierten Aufteilung mit fester Blockgröße wie etwa:

$$\begin{aligned} \text{Trail 0} &= \text{Partikel 1-100} \\ \text{Trail 1} &= \text{Partikel 101-200} \end{aligned}$$

kann die oben definierte Forderung nicht ohne zusätzlichen Aufwand erfüllt werden. Durch die Bewegung der Partikel im Raum ändert sich zwangsläufig auch deren Lage und somit die Speicherung im Octree. Es kann daher im Laufe der Simulation nicht ohne zusätzliche Vorkehrungen sichergestellt werden, daß sich die Partikel 1 – 100 nach x Iterationen des Verfahrens alle im gleichen Ast der Baumdatenstruktur befinden. Deshalb werden die nachfolgenden Implementierungen alle in den Unterräumen der zugeordneten Simulationsgebiete befindlichen Partikel vor jedem Simulationsschritt aufsammeln. Daraus ergibt sich, im Gegensatz zu den untersuchten threadparallelen Varianten, eine dynamische Partikelzuordnung mit einer u.U. weniger ausgewogenen Lastbalancierung. Diese entsteht bei einer Zusammenballung von Partikeln in einem Teilbereich des Simulationsraumes. Ein entscheidender Vorteil bei dem gewählten Vorgehen ist jedoch der koordinierte Zugriff von Trails auf einen begrenzten Teil der Baumdatenstruktur. Es kann somit sichergestellt werden, daß die Trails nicht wahllos die Unterbäume während ihrer Berechnungen durchlaufen müssen.

Bei der Berechnung nahe beieinander liegender Gebiete müssen die Trails gegebenenfalls auf Baumteile anderer Trails zugreifen, die nicht auf dem lokalen System liegen. In solchen Fällen steigen die Zeiten zum Lesen der erforderlichen Daten entsprechend an. Aus diesem Grund ist es wiederum günstig, solche räumlich eng beieinander liegenden Äste des Baumes den Trails eines einzelnen Hostsystems zuzuordnen. Die Umsetzung dieser

Forderung ist aber stark vom Problem und der zur Verfügung stehenden Laufzeitkonfiguration (Hostsysteme) abhängig.

Zusammenfassend kann man sagen, daß die zu erwartende Effizienz von der Partikelverteilung im Problem abhängig ist und die benötigte Laufzeit für einen Simulationsschritt aus den oben genannten Gründen im Laufe der Simulation stark variieren kann.

Um dennoch eine möglichst homogene Verteilung der Partikel auf die einzelnen Trails zu erreichen, werden die hier zu berechnenden Anfangsprobleme entsprechend konditioniert, d.h. es existieren zumindest beim Start der Simulation die gewünschten Voraussetzungen. Das im weiteren Verlauf der Simulation zu erwartende Ungleichgewicht in der Partikelverteilung soll hier nicht weiter betrachtet werden, da die Simulationszeit⁵ der durchgeführten Laufzeittests relativ gering ist und das Hauptinteresse der Untersuchungen auf den Datenstrukturen liegt. Die verwendeten Problemstellungen der im weiteren Verlauf dargestellten Laufzeitexperimente lassen sich mit der in Abbildung 8.2 dargestellten Partikelmenge vergleichen. Prinzipiell werden immer Partikelcluster erzeugt, bei denen die Gesamtanzahl der Partikel verändert wird.

Algorithmisches Gerüst für eine verteilte Berechnung

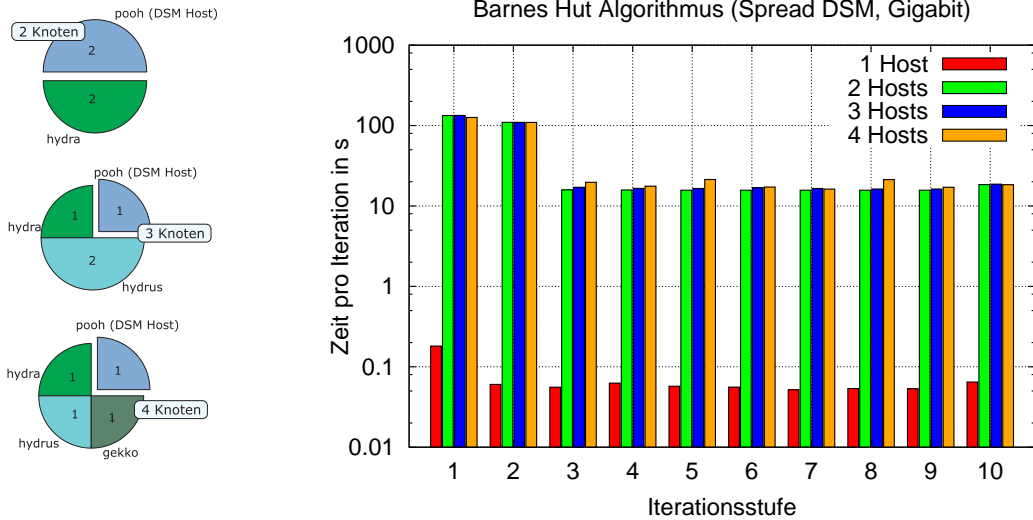
Primäres Ziel für die Implementierung muß die Reduzierung der DSM-Zugriffe sein. Deshalb ist es erforderlich, das bisherige Vorgehen in mehreren Punkten etwas abzuändern.

Der Baumaufbau wird nun nur beim Start der Berechnungen von einem einzigen Trail durchgeführt. Ein verteilter Zugriff auf den Baum wird so vermieden und der DSM kann die Daten lokal auf dem Hostsystem des ausführenden Trails bereithalten. Nach erfolgter Erstellung des kompletten Baumes wird jedem der Trails ein Teilbaum zugeordnet. Durch diese Zuordnung kann der DSM im späteren Verlauf einzelne Knoten oder eventuell ganze Teilbäume anhand der Zugriffe auf die entsprechenden Hostsysteme auslagern, wodurch zeitaufwendige Netzwerkoperationen verringert werden.

In den nachfolgenden Iterationen erfolgt kein erneuter Baumaufbau wie in der seriellen oder der threadparallelen Variante. Stattdessen modifiziert jeder Trail seinen Teilbaum. Partikel, die verschoben werden müssen, d.h. nicht mehr im jeweiligen Raumwürfel des bisherigen Blattes sind, werden an der Wurzel wieder eingefügt und können dadurch auch anderen Trails zugeordnet werden.

Die Phase 2 des Algorithmus, d.h. die Aktualisierung der Massezentren, erfolgt jetzt nach der Baumaktualisierung als eigenständiger *bottom-up* Lauf. Da davon ausgegangen werden muß, daß sich die Lage aller Partikel verändert hat, ist die Aktualisierung aller Pseudoknoten notwendig. Für die Berechnungen der 3. Phase sind keine strukturellen Veränderungen des bisherigen Vorgehens notwendig.

⁵Mit der Simulationszeit t_{end} ist hier die vergangene Zeit im Modell gemeint und umfaßt im allgemeinen mehrere Simulationsschritte, deren Laufzeit jeweils einige Sekunden betragen kann. Bei den betrachteten Problemen ist die Lastbalancierung aufgrund der relativ geringen Anzahl von Simulationsschritten (*meist* < 100) immer noch ausreichend.



(a) Verteilung der verwendeten 4 Trails

(b) Laufzeit für eine Iteration

Abbildung 8.5: Laufzeiten einzelner Iterationsschritte für Konfigurationen mit jeweils 4 ausgeführten Trails (abgeleitet aus der Konfiguration K2). Der Iterationsschritt 0 steht hier für die benötigte Zeit der Initialisierungsphase (Baumaufbau und erste Berechnung). Als DSM-Backend wurde hier immer die *Spread*-Implementierung mit WriteBack-Protokoll verwendet.

Abbildung 8.5 zeigt die gemessenen Laufzeiten der ersten 10 Iterationen einer Simulation mit *Spread*-DSM. Bedingt durch den Baumaufbau mit nur einem Trail am Start der Simulation (in der Abbildung 8.5a jeweils als *DSM-Host* markiert) werden die Daten in den mit Iterationsstufe 1 und 2 bezeichneten Schritten implizit auf die jeweiligen Hostsysteme der zugreifenden Trails verteilt. Die Zeiten für die nachfolgenden Iterationen verkürzen sich entsprechend, da dann einige Netzwerkoperationen entfallen.

Auf dem Hostsystem *pooh* wurde, zum Vergleich mit den Multihost-Konfiguration, ein lokaler Laufzeittest (1 Host) mit vier Trails durchgeführt. Die hierbei ermittelten Laufzeiten weichen stark von den Laufzeiten für mehrere Hostsysteme ab, da keine Netzwerkkommunikation notwendig ist. Ab der 2. Iterationsstufe, d.h. nach dem Baumaufbau und der Initialisierung des DSM-Subsystems, sinken dort ebenfalls die Laufzeiten pro Iteration. Für die verschiedenen Multihost-Konfigurationen gibt es keine wesentlichen Laufzeitunterschiede. Eine leichte Zunahme der Laufzeiten pro Iteration kann man allerdings bei der Erhöhung der Anzahl der Hostsysteme beobachten.

8.3.2 Datenstruktur

Ausgehend von den bisherigen Betrachtungen erscheint es sinnvoll, ein eigenständiges DSM-Feld mit allen Partikeln und eine DSM-Datenstruktur zur Abspeicherung des Baumes mit Verweisen auf die Partikel im Feld zu benutzen.

Octree

Wie in Kapitel 6.2.2 (S.180) dargestellt, besitzen die hier benutzten Bäume immer 8 Kinder. Die Darstellung im DSM wird durch eine 1:1 Abbildung der Knoten und DSM-Zellen realisiert, d.h. jedem Knoten des Baumes kann genau eine DSM-Zelle zugeordnet werden. Bei der Generierung von Namen für die DSM-Zellen muß deshalb sichergestellt werden, daß es nicht zu Überschneidungen kommt. Vorteilhaft ist weiterhin die einfache Berechnung des Vaterknotens für einen gegebenen Knoten. Da der Baumaufbau im *Barnes-Hut*-Verfahren einer adaptiven Grundidee folgt, kann sich schon bei kleinsten Lageveränderungen einzelner Partikel eine völlig andere Baumstruktur ergeben. Für die Benennung der DSM-Zellen wird das folgende Schema benutzt:

$$\text{cellName} := \text{'name'} + \text{nodeID}$$

Hierbei ist `name` eine beliebige, aber für alle Knoten des Baumes feste Zeichenkette, die für die Abgrenzung verschiedener Bäume im DSM benutzt wird. Der mit `nodeID` gekennzeichnete Anteil ist ein numerischer Wert, der leicht berechnet werden kann.

Berechnung der nodeID Für die eindeutige `nodeID` jedes Knotens wird die fortlaufende Numerierung eines vollständigen Octree benutzt. Zunächst ist hierfür die Tiefe des Vaterknotens t_v mit der Nummer k_v im Baum notwendig (Formel 8.3). Auf einige Details dieser Berechnungen wird nochmals in Anhang A.1 eingegangen. Deshalb wird hier auf eine Herleitung der genannten Berechnungsvorschriften verzichtet.

$$t_v = \lfloor \log_8(7 \cdot k_v + 1) \rfloor \quad (8.2)$$

Die Berechnung der ID des i ten Kindes eines Vaterknotens k_v kann dann anhand der Formel

$$id = 8 \cdot \underbrace{\left(k_v - \frac{8^{t_v} - 1}{7} \right)}_{\text{Knoten bis zum Vater}} + \underbrace{\frac{8^{t_v+1} - 1}{7}}_{\text{1. Knoten Kindlevel } t_v + 1} + i \quad (8.3)$$

erfolgen. Analog erhält man durch Umstellen von Formel 8.3 die Formel 8.4 zur Bestimmung der `nodeID` des Vaters k_v aus der Nummer des Kindes id .

$$k_v = \left\lfloor \frac{id - 1}{8} \right\rfloor \quad (8.4)$$

In der Implementierung wurde für die `nodeID` eine Variable vom Typ `long` benutzt, die in Java als 64-Bit Integer definiert ist. Als theoretische Extremwerte ergeben sich somit für die maximal im Baum verwaltbaren Knoten $max_k = 2^{64} - 1$ und als maximale Baumtiefe ein Wert von 21.

$$2^{63} = 2^{3 \cdot 21} = (2^3)^{21} = 8^{21} \Rightarrow \text{Tiefe } 21$$

Diese Werte entsprechen in etwa einer Simulation mit $9 \cdot 10^{18}$ Partikeln. Setzt man hier einen sehr optimistischen Speicherbedarf von 25 Byte pro Partikel⁶ an, sind mehr als

⁶Allein für die Speicherung der wesentlichen Daten werden 20 Byte benötigt. Diese setzen sich aus dem Speicherbedarf für die gerade besprochene ID (64Bit = 8Byte) und den 3D Koordinaten als `float` Vektor ($3 * 4\text{Byte} = 12\text{Byte}$) zusammen.

200 Exabyte allein für die Partikelmenge notwendig. Für die hier durchgeführten Untersuchungen werden sehr viel kleinere Partikelmengen betrachtet, allerdings ist die Verteilung wesentlich inhomogener, was zu tieferen Baumstrukturen führt. Durch eine Vergrößerung der vorhandenen Bitmaske können diese Grenzen beliebig ausgedehnt werden. Für die hier benutzten Problemstellungen reicht der durch `long` erhaltene Wertebereich aus.

8.3.3 Implementierung

Der verteilte Algorithmus für das *Beluga* Laufzeitsystem (Listing 8.6) enthält wieder die 3 Phasen: Baumaufbau, Berechnung der Pseudoknoten und abschließende Kraftberechnung.

```

// Baum erstellen
if (me == master)
    buildTree() ;

barrier() ;

while (currentTime < endTime)
{
    // Werte der Pseudopartikel
    compPseudo(root of subTree) ;

    barrier() ;

    // Kraft der Partikel im Unterbaum berechnen
    computeForce() ;

    barrier() ;

    updateTree( root of subTree ) ;

    barrier() ;

    currentTime += dt ;
}

```

Listing 8.6: Beluga Hauptprogramm Barnes-Hut

Da der Baumaufbau nur einmal beim Start der Simulation erfolgt und danach durch ein verteiltes Baumupdate ersetzt wird, ist die Reihenfolge der 3 Phasen in der `while` Schleife etwas umgeordnet. Dies hat zur Folge, daß ein Update der Baumstruktur im letzten Zeitschritt der Simulation durchgeführt wird, welches für das Ergebnis aber nicht mehr notwendig ist. Die benötigte Laufzeit dieses Schrittes ist jedoch vernachlässigbar und letztendlich gestaltet sich durch diese Aufteilung der Pseudocode lesbarer.

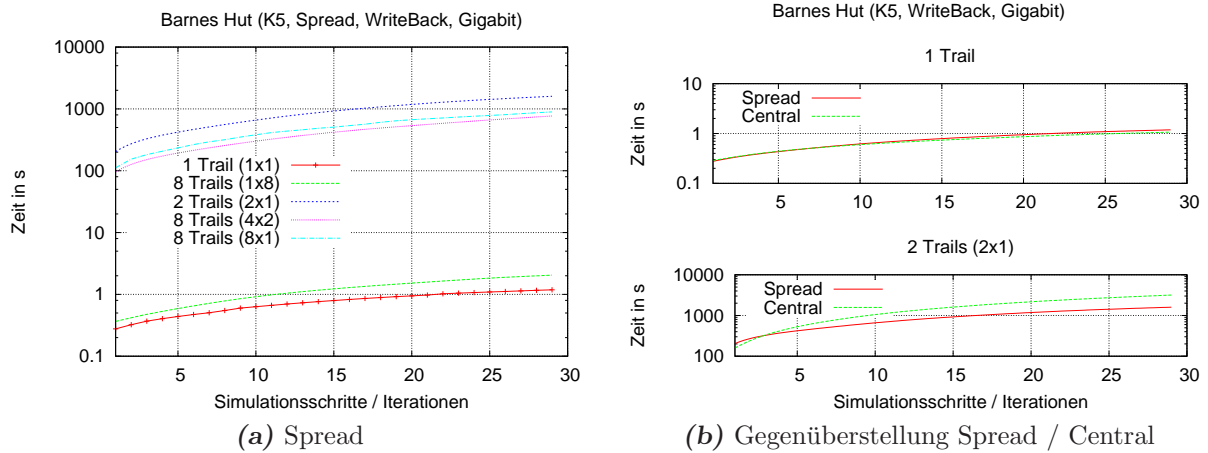


Abbildung 8.6: Ermittelte Laufzeiten für Testläufe mit unterschiedlicher Anzahl von Trails. In (b) sind Zeiten der *Spread* und *Central*-DSM gegenübergestellt. Die Angaben zur Aufteilung der Trails können wieder als (Hostsysteme x lokale Trails) interpretiert werden.

Zuordnung der Unterbäume

Bevor die ersten Testergebnisse präsentiert werden, soll noch kurz auf die verwendete Vorschrift für die Zuordnung von Trails und Unterbäumen eingegangen werden. Ausgehend von der Anzahl der vorhandenen Trails t wird die Tiefe λ im Octree bestimmt, welche mindestens so viele Knoten besitzt, wie Trails vorhanden sind.

$$\lambda = \lceil \log_8 t \rceil$$

Jedem Trail kann dann mindestens ein Knoten bzw. ein Unterbaum aus diesem Level zugeordnet werden. Genauer gesagt, werden alle Knoten dieser Tiefe gleichmäßig auf die vorhandenen Trails verteilt.

Auswertung

Im Gegensatz zum *Linked-Cell*-Verfahren kann beim Einsatz mehrerer Trails mit der *Beluga* Implementierung des *Barnes-Hut*-Algorithmus keine Verringerung der Laufzeit erreicht werden (Abbildung 8.6). Beim Einsatz des *Spread*-DSM entsteht ein geringer Laufzeitvorteil gegenüber dem *Central*-DSM, trotzdem ist der Laufzeitunterschied zur Ausführung mit nur einem Trail sehr hoch (Abbildung 8.6b). Mögliche Ursachen können im Zugriffsverhalten auf den DSM gesucht werden. Durch die mehrfache Baumtraversierung sind sehr häufig DSM-Operationen notwendig.

8.3.4 Optimierungsansätze

Für eine Verbesserung der Skalierungseigenschaften bietet sich in erster Linie die Reduzierung bzw. Optimierung der DSM-Zugriffe an. Da ein Großteil der Rechenzeit auf die

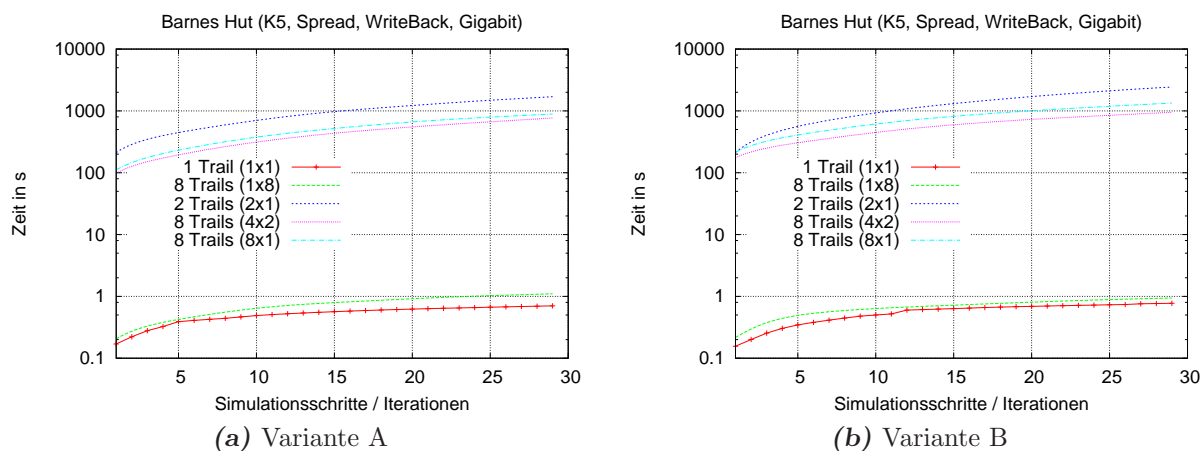


Abbildung 8.7: Meßergebnisse für die beschriebenen Optimierungsvarianten **A** und **B** mit verschiedenen Trailkonfigurationen (Hostsysteme x lokale Trails) und *Spread*-DSM.

Phase 3 (`computeForce`) entfällt, sollen hier mögliche Ansätze überprüft werden.

Variante A: gleichzeitige Baumtraversierung

Betrachtet man die DSM-Zugriffe, so läßt sich deren Anzahl für einen Baum b mit σ_b Knoten und n Partikel abschätzen durch

$$\mathcal{O}(\sigma_b \cdot n).$$

Da für jeden Partikel ein *top-down* Lauf über den Baum erforderlich ist, müssen im schlechtesten Fall alle Knoten σ_b des Baumes b betrachtet werden. Alternativ könnte man auch nur einen solchen *top-down* Lauf durchführen und gleichzeitig alle n Partikel aktualisieren. Pro Schritt ist dann nur ein DSM-Zugriff auf den aktuellen Knoten notwendig, dieser kann dann für die anschließende Iteration über alle Partikel lokal gespeichert werden. Zum Erhalt der asymptotischen Laufzeit müssen noch zusätzliche Vorkehrungen getroffen werden, damit ein Baumdurchlauf abgebrochen werden kann, sobald alle Partikel berechnet sind⁷. Insgesamt wird der gesamte Berechnungsaufwand durch die zusätzlichen Bedingungen etwas erhöht, die Anzahl der **DSM-Zugriffe** reduziert sich damit aber auf

$$\mathcal{O}(\sigma_b).$$

⁷Dies kann z.B. mit einfachen Zählern realisiert werden, indem für jeden Pseudoknoten die Anzahl der durch diesen approximierten Partikel ermittelt und mit der Gesamtanzahl verglichen wird. Die weitere Betrachtung des Teilbaumes entfällt, wenn der Zähler des Pseudoknotens gleich der Anzahl der betrachteten Partikel ist.

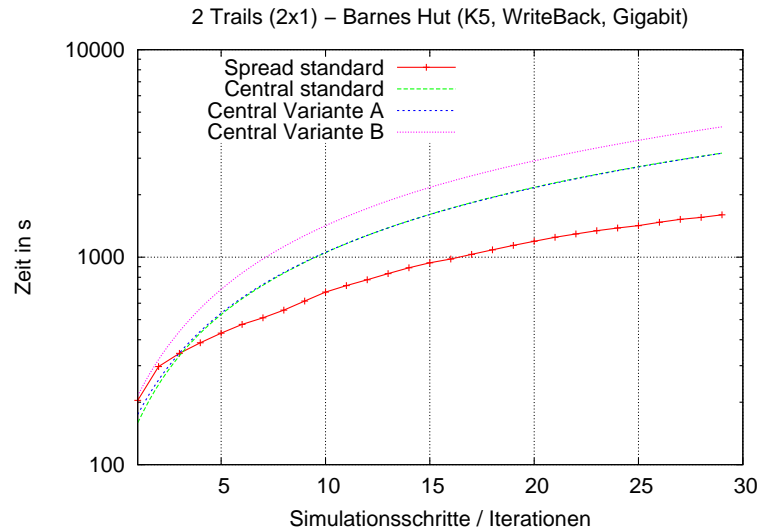


Abbildung 8.8: Gegenüberstellung von Laufzeiten der Barnes-Hut Optimierungsvarianten und *Central*-DSM. Zum Vergleich sind die Laufzeiten der ursprünglichen Implementierung (*standard*) für *Spread*- und *Central*-DSM enthalten. Die Meßreihen der Laufzeittests von *Central standard* und *Central Variante A* sind deckungsgleich.

Variante B: CachedOtree

Ein anderer Optimierungsgedanke zur Reduzierung der DSM-Zugriffe ist die gezielte Zwischenspeicherung aller in der aktuellen Iteration angefragten Knoten. Ziel ist es, Mehrfachanfragen ans DSM-System zu vermeiden und bereits bekannte Werte zu benutzen. Da sich die Daten nach jeder Iteration verändern, müssen nach jedem Zeitschritt der Zwischenspeicher gelöscht und eventuelle Veränderungen in den DSM übertragen werden, was den Aufwand insgesamt etwas erhöht.

Bewertung

In Abbildung 8.7 sind die Laufzeiten, der in Abbildung 8.6 benutzten Trailverteilungen, für die soeben erläuterten Varianten **A** und **B** aufgetragen. Für den *Spread*-DSM ist keine Veränderung im Laufzeitverhalten erkennbar (vgl. auch Abbildung 8.6). Als Ursachen kommen mehrere Gegebenheiten in Betracht:

1. Die *Spread*-DSM-Implementierung und der im DSM verankerte Cache-Speicher optimieren die Zugriffe bereits.
2. Für die Berechnungen ist das Lesen aller bzw. einer großen Mehrheit der DSM-Zellen notwendig.

Zur Eingrenzung der möglichen Ursachen wurden deshalb die Tests für die Varianten A und B mit einer Konfiguration von 2 Trails auf 2 Hostsystemen (*2x1*) und dem *Central*-DSM wiederholt (Abbildung 8.8). In dieser Konfiguration müssen alle DSM-Anfragen

zentral bearbeitet werden und das Verschieben von einzelnen Datenblöcken auf andere Hostsysteme ist nicht möglich.

Bei der Analyse der Meßdaten zeigt sich, daß die Variante B einen höheren Laufzeitbedarf hat. Mehrfache Zugriffe auf Zellen innerhalb einer Iteration sollten hierbei vermieden und durch einen lokalen Zwischenspeicher beantwortet werden. Der eingesetzte zusätzliche Zwischenspeicher muß am Ende jeder Iteration bereinigt und veränderte Daten in den DSM geschrieben werden. Da dieser Vorgang von allen Trails zu einem festen Zeitpunkt durchgeführt wird, kann es zu Verzögerungen beim zentralen *Rootserver* des *Central*-DSM kommen.

Die ermittelten Laufzeiten der Variante A sind deckungsgleich mit denen der ursprünglichen Variante. Schlußfolgernd kann man deshalb festhalten, daß die Zugriffe auf den DSM hier nicht vermieden werden konnten. Ursächlich kann dies an einem für die DSM-Varianten zu klein dimensionierten Problem liegen. Eine Vergrößerung ist aber nur begrenzt möglich, da die benötigten Laufzeiten bereits sehr hoch sind.

8.4 Zusammenfassung

In diesem Kapitel wurden verschiedene Implementierungen des *Barnes-Hut* Algorithmus untersucht. Eine trailparallele Implementierung für das *Beluga* Laufzeitsystem wurde direkt aus der zuvor implementierten threadparallelen Variante entwickelt. Aufgrund einiger Besonderheiten des *Barnes-Hut*-Verfahrens waren hierbei umfangreichere Änderungen als bei dem im Vorfeld beschriebenen *Linked-Cell*-Verfahren notwendig. Hierbei flossen allerdings die bisher gewonnenen Erkenntnisse ein, d.h. bei der Implementierung wurde versucht, Zugriffe auf den DSM zu reduzieren. Zusammenfassend betrachtet konnte die trailparallele Variante gut aus der threadparallelen Variante abgeleitet werden.

Bei der Analyse wurde deutlich, daß die Laufzeit der *Barnes-Hut*-Varianten für das *Beluga* Laufzeitsystem nicht durch den Einsatz mehrerer Trails gesenkt werden kann. Dieses Verhalten ist vor allem auf die Zugriffe der Trails auf den DSM zurückzuführen. Durch eine Veränderung dieser Zugriffsmuster und einer optimierten Abspeicherung der Daten in einem verteiltem Baum (DSM-Objekt) kann mit Sicherheit eine Verbesserung erreicht werden. Einige erste Anhaltspunkte hierfür konnten im Abschnitt 8.3.4 gefunden werden. Für die Laufzeitoptimierung ist eine weitaus genauere Betrachtung, Evaluierung und Implementierung von möglichen Strategien notwendig. Die hier durchgeführten Arbeiten hatten jedoch das Ziel, die Eignung der *DEE* Plattform für direkte Portierungen von threadparallelen Implementierungen zu untersuchen. Beim *Barnes-Hut*-Algorithmus ist dies nicht so uneingeschränkt wie bei dem im vorangegangenen Kapitel betrachteten *Linked-Cell*-Verfahren möglich.

Kapitel 9

Zusammenfassung und Ausblick

Ziel dieser Arbeit war die Definition einer flexiblen Programmierumgebung für die Ausführung von verteilten Berechnungen in einem Computernetzwerk. Die einzelnen Knoten in einem solchen Netzwerk können dabei unterschiedliche Charakteristiken aufweisen und die konventionelle Programmierung erschweren. Es gibt bereits eine Vielzahl von Programmiersystemen, um parallel arbeitende Programme für solche verteilten heterogenen Systemumgebungen zu erstellen. Oftmals ist das zugrundeliegende Programmiermodell dieser Programmiersysteme aber komplex und erlaubt die Erstellung von effizienten Programmen nur für eine spezielle Zielsetzung. Die hier vorgestellte Programmierumgebung bzw. das bereitgestellte Programmiermodell sollten sich daher am weit verbreiteten Threadmodell orientieren und so die Programmierung verteilter Systeme erleichtern. Weitere Zielsetzungen waren die Anpaßbarkeit der Umgebung an geänderte Anforderungen und die Möglichkeit zur Einbettung bereits existierender Programmierlösungen.

Die entstandene *DEE* Spezifikation und die Prototypimplementierung *Beluga* verfügen über eine hohe Modularität. Einzelne Funktionseinheiten der Laufzeitumgebung lassen sich einfach ersetzen, da die benötigten Funktionen in einem Schichtenmodell mit verschiedenen Kernkomponenten verzahnt sind. Diese Austauschbarkeit einzelner Systemteile wurde intensiv bei den Untersuchungen der verschiedenen *Storage-Modul*-Implementierungen des DSM-Subsystems angewendet. In den hierfür durchgeführten Laufzeittests konnten die jeweils betrachteten DSM-Varianten durch ein einfaches Abändern von Konfigurationsparametern der *Beluga* Prototypimplementierung schnell in das Laufzeitsystem integriert werden - ohne eine Neuübersetzung der Testanwendungen vorzunehmen. Die *Beluga* Kommunikationsschicht und die Kohärenzprotokolle der DSM-Kernkomponente sind ebenfalls Beispiele, in denen ein Austausch einzelner Teile vorgenommen wurde.

Da sich das entstandene Laufzeitsystem am Threadmodell orientiert, aber gleichzeitig auch das Versenden von Nachrichten zwischen den einzelnen parallelen Kontrollflüssen, den sogenannten Trails erlaubt, können bestehende parallele Anwendungen relativ schnell portiert werden. Dies zeigte sich z.B. bei den im Rahmen der experimentellen Evaluierung der Laufzeitumgebung erstellten *Beluga* Implementierungen einer Matrixmultiplikation und eines N-Queens Problemlösers. Falls der vom Laufzeitsystem zur Verfügung gestellte, gemeinsame Speicher (DSM) vom Anwendungsprogramm verwendet wird, hängt die erreichte Performance vom Zugriffsverhalten auf diesen DSM und von der verwendeten

DSM-Implementierung ab. Für das detailliert untersuchte *Linked-Cell*-Verfahren konnten, ohne größere Änderungen am Quellcode, gute Speedup-Werte für Berechnungen in einer verteilten Laufzeitkonfiguration erzielt werden. Beim *Barnes-Hut*-Algorithmus gelang dies nicht. Es wurden aber erste Anhaltspunkte für mögliche Gründe gefunden und somit ist auch hier eine besser skalierende Implementierung denkbar.

Für den Programmierer kann sich durch die gleichzeitige Unterstützung von Threadmodell und nachrichtenbasierter Programmierung eine Verminderung des Programmieraufwandes ergeben. Die meist bessere Performance von Programmen, die auf dem Austausch von Nachrichten basieren, läßt sich auf die direkte Kommunikation und indirekte Optimierung durch den Programmierer zurückführen. Allerdings ist der Zeitaufwand für die Erstellung einer solchen Implementierung und deren Fehlerpotential oftmals höher, als für ein vergleichbares sequentielles oder threadparalleles Programm. Wird eine *DEE* Umgebung benutzt, kann in der Frühphase der Umsetzung eines Algorithmus die Funktionalität des DSM benutzt und somit eine erste lauffähige Variante erstellt werden. In einer solchen Variante (*proof of concept*) können bereits erste Schwachstellen lokalisiert und eventuelle Fehlerquellen beseitigt werden. Bei einer nachfolgenden Überarbeitung, mit dem Ziel einer möglichst effizienten parallelen Abarbeitung, können dann die als besonders zeitkritisch lokalisierten Abschnitte optimiert und eventuell durch nachrichtenbasierte Operationen ausgetauscht werden. Für zeitlich unbedeutende Abschnitte kann allerdings der für den Programmierer deutlich komfortablere DSM-Zugriff beibehalten werden.

Im Verlauf der Arbeit wurden drei DSM-Varianten *Central*, *Spread* und *TSpaces* erstellt und untersucht. Jede der Implementierungen verfügt über spezifische Eigenschaften und ist damit für unterschiedliche Anwendungsszenarien mehr oder weniger geeignet. Der *Central*-DSM eignet sich aufgrund seines zentralen Ansatzes und des geringen Verwaltungsaufwandes für Anwendungen, bei denen einzelne Ergebnisse in einer DSM-Zelle gesammelt werden. Beim *TSpaces*-DSM stehen Aspekte wie Sicherheit der Kommunikation und Bereitstellung wenig beschriebener statischer Daten im Vordergrund. Im Fall des *Spread*-DSM werden Anwendungen unterstützt, deren Kontrollflüsse auf unterschiedliche Bereiche des Speichers zugreifen.

Bei der Verwendung der Funktionalitäten des DSM sind aber auch deutliche Laufzeiteinbußen durch die in der Kommunikationsschicht verwendete Objektserialisierung erkennbar. Die benutzten Netzwerke können dadurch nur einen Bruchteil der möglichen Übertragungsleistungen erzielen. Dadurch steigt die Zugriffszeit auf den DSM deutlich an und wirkt einer effizienten verteilten Ausführung entgegen.

Weiterführende Arbeiten könnten sich mit einer Verbesserung der Serialisierung von Objekten befassen und z.B. die in der Neuimplementierung des RMI Protokolles *KaRMI* aus JavaParty gewonnenen Erkenntnisse [53, 103] anwenden. Die bereits angesprochene Reimplementierung des *Barnes-Hut*-Algorithmus für *Beluga* ist ebenfalls ein mögliches Aufgabengebiet.

Weitaus umfangreicher könnte die Erstellung einer weiteren *DEE* Umgebung in Form einer

Kombination aus Compiler und Laufzeitsystem¹ sein. Mit Hilfe verschiedener Analysen könnte ein solcher Compiler z.B. spezielle Bibliotheksaufrufe in das Ausgabeprogramm integrieren oder anhand gesammelter Informationen aus mehreren möglichen Bibliotheksfunktionen eine geeignete auswählen. Hierdurch könnten die erzeugten trailparallelen Programme weiter optimiert werden.

¹Zusätzlich zu einer *Beluga*-ähnlichen Bibliothek könnte noch eine virtuelle Maschine implementiert werden.

Anhang A

Ergänzungen

Im Kapitel 8 wurden einige Aussage bei der Berechnung von Knotennummern in einem *Octree* verwendet, welche in diesem Abschnitt aufgegriffen und kurz erläutert werden sollen.

A.1 Berechnungen für k -näre Bäume

Tiefe

Bei der Berechnung der Knotentiefe t wird ein vollständiger Baum mit jeweils q Kindern vorausgesetzt. Die Knotennummer k läßt sich dann darstellen als:

$$k = q^0 + q^1 + \dots + q^{t-1} + x$$

Der verwendete Parameter x gibt die Anzahl der Vorgängerknoten auf dem aktuellen Level bzw. der aktuellen Tiefe t an (siehe auch Abbildung A.1). Die darin enthaltene *geometrische Zahlenfolge* kann man zusammenfassen und erhält:

$$k = \frac{q^t - 1}{q - 1} + x$$

Da x einen variablen lokalen Index zwischen 0 und $q^t - 1$ auf dem untersuchten Knotenlevel angibt und gerade diese Tiefe ermittelt werden soll, reicht es den ersten Knoten zu betrachten¹ und entsprechend umzustellen:

$$t = \log_q ((q - 1) \cdot k + 1) \tag{A.1}$$

Kinder

Anhand der Informationen zur Tiefe t und Knotennummer k eines Knotens können die Knotennummern der Kinder berechnet werden. Die Nummer des ersten Kindes k_{c_1} eines Knotens setzt sich zusammen aus den Kindern aller Vorgängerknoten x im Level des Vaters und der Summer aller Knoten bis zum Level des Kindes $t + 1$.

$$k_{c_1} = q \cdot x + \underbrace{\frac{q^{t+1} - 1}{q - 1} - 1}_{\text{Knoten bis } t + 1} \underbrace{+ 1}_{\text{1. Kind}}$$

¹ $x = 0$

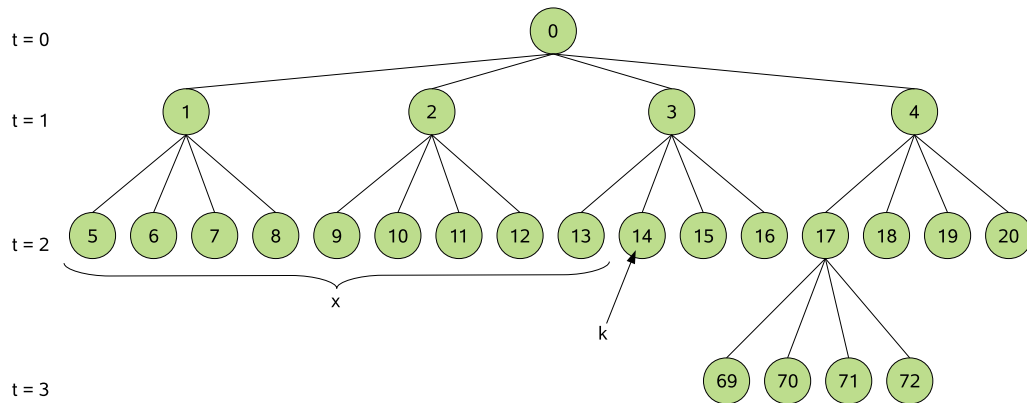


Abbildung A.1: Illustration zu den Formeln A.1 und A.2 am Beispiel eines Quadtrees.

Den Wert von x kann man berechnen, indem man von der Nummer des Vaters k die Nummer des ersten Knotens auf dem Vaterlevel subtrahiert.

$$x = k - \frac{q^t - 1}{q - 1}$$

Faßt man wieder zusammen erhält man die Formel A.2.

$$k_{c_1} = q \cdot \left(k - \frac{q^t - 1}{q - 1} \right) + \frac{q^{t+1} - 1}{q - 1} \tag{A.2}$$

A.2 Java-Optimierungen

An dieser Stelle sollen nur kurz einige Techniken und Strategien zur Laufzeitoptimierung von Java-Programmen genannt werden. Ausführlichere Informationen hierzu sind z.B. in [114] oder auf <http://www.javaperformancetuning.com> zu finden.

- Benutzung von `ArrayList` statt `Vector`
 Der Zugriff auf die Klasse `Vector` ist durch `synchronized` geschützt. Im Falle eines konkurrierenden Zugriffes durch mehrere Threads ist dies durchaus praktikabel. Falls nur ein Thread existiert bzw. auf diese Daten zugreift, entsteht ein unnötiger Overhead. Indirekt wirkt sich dieser Effekt auch auf die Benutzung der Klasse `Stack` aus. Diese ist direkt von `Vector` abgeleitet und erbt deshalb dessen Verhalten.
- Definition von `final` Methoden und Variablen
 Als `final` definierte Methoden oder Variablen dürfen nach der Initialisierung nicht mehr verändert werden. Deshalb können verschiedene Überprüfungen der Java-Laufzeitumgebung entfallen.
- Benutzung von `public` Modifizierern
 Auf mit `public` gekennzeichnete Methoden oder Variablen können ebenfalls einige Überprüfungen zur Laufzeit entfallen.

Anhang B

Laufzeitsystem und Testumgebungen

B.1 Beluga - Benutzerschnittstellen und Klassen

Der folgende Abschnitt enthält eine Auflistung von Klassen und Schnittstellen der *Beluga* Laufzeitumgebung. Die Auswahl ist dabei auf die für den Anwender relevanten Klassen beschränkt. Das gesamte *Beluga* System und die zusätzlichen Schnittstellen der *DEE* Beschreibung umfassen mehr als 360 einzelne Klassen und Schnittstellen.

`beluga.Beluga` Benutzerrouninen der *Beluga* Serviceschicht.

`beluga.Beluga.BelugaProfile` Das von *Beluga* implementierte *DEE* Profil.

`beluga.DSM` Benutzerschnittstellen des DSM-Subsystems.

`beluga.DTM` Routinen des *Distributed Trail Models*.

`beluga.DTMParames` Parameter beim Starten eines Trailteams.

`beluga.comm.io.jfs.JFSLoader` Laderoutinen für die optionale JFS Bibliothek.

`beluga.comm.names.registry.TRegistryServerInterface` Die Schnittstellendefinition des *Registry-Servers*.

`beluga.comm.names.registry.TRegistryServer` Implementierung eines vom Laufzeitsystem benötigten *Registry-Servers*.

`beluga.comm.names.HostIDs` „*Leichtgewichtige*“ Klasse für Hostnamen der Laufzeitumgebung.

`beluga.comm.names.THostNamesFactory` Factory zum Erzeugen von `THostNamesImpl` Objekten.

`beluga.comm.names.THostNamesImpl` Implementierung eines abstrakten *Hostname* Objektes. Enthält neben den `HostIDs` auch eine Liste mit den Adressen der Kommunikationskomponente (IP-Adressen).

`beluga.domp.TdtmThread` Jeder Trail wird auf einen dieser Threads abgebildet.

- `beluga.domp.TExecuteInfo` Diese Klasse enthält Status- und Verwaltungsinformationen der ausführenden Threads.
- `beluga.domp.TExecuteCode` Enthält die ausführbaren Anweisungen der Trails, hierfür muß die abstrakte Methode `Code()` überschrieben werden.
- `beluga.domp.TLocalBarrier` Implementierung einer lokalen Barrier.
- `beluga.domp.TMutex` Implementierung einer Mutex-Variable.
- `beluga.domp.TRWMutex` Ein einfacher Mutex-Mechanismus, aber mit Unterscheidung zwischen Lese- und Schreibzugriff.
- `beluga.domp.team.TSimpleTeamBuilder` Implementiert das Vorgehen zum Erzeugen eines Trailteams. Die allgemeine Schnittstelle hierfür ist durch das *Interface* `dee.domp.team.TTeamBuilderLogic` vorgegeben.
- `beluga.dsm.coherence.StandardWriteBack` Kohärenzprotokoll
- `beluga.dsm.coherence.StandardWriteThrough` Kohärenzprotokoll
- `beluga.dsm.coherence.simple.SimpleCoherence` Das *Simple*-Kohärenzprotokoll
- `beluga.dsm.coherence.simple.SimpleLock` Implementierung eines Locks für das *Simple*-Kohärenzprotokoll.
- `beluga.dsm.locks.LocalLockManager` Schnittstellendefinition für einen hierarchischen Lockmechanismus.
- `beluga.dsm.locks.LLMsimple` Einfache Implementierung der Schnittstelle `LocalLockManager`.
- `beluga.dsm.locks.LocalManagerFactory` Factory zum Erzeugen von Instanzen eines *LocalLockManagers*.
- `beluga.dsm.datatype.cube.Raum` Eine verteilte Datenstruktur zur Abspeicherung von Partikeln in einem Simulationsraum für das *Linked-Cell*-Verfahren.
- `beluga.dsm.datatype.cube.RaumProducer` Der zugehörige und im Laufzeitsystem zu registrierende *Producer* für das Raum DSM-Objekt.
- `beluga.dsm.datatype.feld.Feld` DSM-Objekt eines verteilten Feldes.
- `beluga.dsm.datatype.feld.FeldProducer` Der zugehörige und im Laufzeitsystem zu registrierende *Producer* für das Feld DSM-Objekt.
- `beluga.dsm.datatype.octree.OctreeCube` Dieses DSM-Objekt verwaltet eine verteilte Baumdatenstruktur für den untersuchten *Barnes-Hut*-Algorithmus.

`beluga.dsm.datatype.octree.OctreeCubeProducer` Der zugehörige und im Laufzeitsystem zu registrierende *Producer* für das `OctreeCube` DSM-Objekt.

`beluga.dsm.datatype.single.ObjectCell` Ein einfaches DSM-Objekt zur Verwaltung von Daten in nur einer DSM-Zelle.

`beluga.dsm.datatype.single.ObjectCellProducer` Der zugehörige und im Laufzeitsystem zu registrierende *Producer* für das `ObjectCell` DSM-Objekt.

`beluga.system.bench.TSysBenchData` Die Klasse enthält allgemeine Angaben zur Systemleistung und zur Konfiguration der Laufzeitumgebung.

B.1.1 Konfiguration

Das Listing B.1 zeigt die Konfigurationsdatei der *Beluga* Prototypimplementierung. Diese Datei enthält verschiedene XML-Elemente (Tags) und ist in verschiedene Abschnitte unterteilt. Jede Kernkomponente des Laufzeitsystems hat Zugriff auf diese Konfiguration und kann die notwendigen Parameter auslesen. Die in der Konfigurationsdatei angegebenen zentralen Parameter sind deshalb abhängig von den verwendeten Kernkomponenten, da z.B. die TCP Kommunikationskomponente speziell für TCP angepaßte Einstellungen enthält, welche für ein anderes Kommunikationsmodul nicht relevant sein können.

```

1 <?xml version="1.0" encoding="iso-8859-1" ?>
2
3 <beluga>
4   <connection port="3012" ipv6="false" device="eth0">
5     <disallow adr="192.168.1.1"/>
6     <disallow device="vmnet8"/>
7     <server>
8       <parameter dynamic="true"
9         jfs="false"
10        measure="false"/>
11     </server>
12   </connection>
13
14   <registry name="132.180.193.118"/>
15
16   <boot minhosts="1" timeout="30" />
17
18   <hostlist maxcpus="4" mincpus="1">
19     <host name="132.180.193.64" cpus="3"/>
20   </hostlist>
21
22   <dtm shared="true" />
23
24   <dsm rootServer="132.180.193.39"
25     implementation="spread"
26     coherenceProtocol="stdWriteBack">
27   </dsm>
28 </beluga>

```

Listing B.1: Konfigurationsdatei *beluga.xml*

beluga Dieses Element ist das Wurzelement der Konfiguration und enthält alle weiteren Elemente.

connection Das `connection` Tag erlaubt die Konfiguration der Kommunikationskomponente. Als Attribut sind

port Verbindungsport für die Kommunikation

ipv6 Verwendung des IPv6 Protokolles

device zu benutzendes Netzwerkgerät

definiert. Weiterhin ist die Einbettung der Elemente `disallow` und `server` erlaubt.

disallow Mit `disallow` können Netzwerkgeräte (`device`) und Netzwerkadressen (`adr`) von der Benutzung durch die *Beluga* TCP Kommunikationskomponente ausgeschlossen werden.

server Weitere Angaben für die TCP Komponente.

registry Das `registry` definiert durch das Attribut `name` den *Registry-Server* des Laufzeitsystems.

boot Angaben zum Bootvorgang der Laufzeitumgebung können mit diesem Element vorgegeben werden. Hierzu zählen die Anzahl der verfügbaren Hostsysteme (`minhosts`) und die maximale Wartezeit (`timeout`), die bis zum Abbruch des Bootvorganges vergehen muß.

hostlist Mit dem `hostlist` Tag können Angaben zur Prozessorverwendung auf allen oder einzelnen Hostsystemen gemacht werden. Hierdurch kann indirekt die Verteilung von Trails gesteuert werden, da das Laufzeitsystem beispielsweise versucht, auf einem Hostsystem mit drei Prozessoren auch drei Trails zu starten.

host Angaben zu einem einzelnen Hostsystem. Dieses Element ist in ein `hostlist` Tag eingebettet.

dtm Laufzeitparameter für die DTM-Kernkomponente.

dsm Laufzeitparameter für die DSM-Kernkomponente. Hierzu zählen die zu verwendende Implementierung (`implementation`), der zentrale Rootserver (`rootServer`) und das Standardprotokoll für die Kohärenzfunktionen (`coherenceProtocol`).

B.2 verwendete Rechnersysteme und Systemkonfigurationen

An dieser Stelle sollen die verwendeten Rechnersysteme und die für die Laufzeittests zusammengestellten Laufzeitkonfigurationen beschrieben werden. Weiterhin sind in diesem Abschnitt einige Informationen zu Toolprogrammen, welche zur Bestimmung von Leistungsmerkmalen der Testsysteme benutzt wurden, zu finden.

B.2.1 Übersicht verwendeter Rechnersysteme

Dieser Abschnitt dokumentiert die Hardwareausstattung der für die Laufzeittests eingesetzten Rechnersysteme. Neben einer Vielzahl unterschiedlicher Computersysteme stand am *Lehrstuhl für Angewandte Informatik 2* der *Universität Bayreuth* ein Rechencluster aus 32 homogenen Knoten zur Verfügung. In der nachfolgenden Auflistung ist das Hostsystem `clust01` stellvertretend für alle Knoten dieses Rechenclusters beschrieben.

pooh

Prozessor	Intel Core™ 2 Duo (<i>E8400 Wolfdale</i>), 3GHz
Anzahl Prozessoren	1
Kerne pro Prozessor / Σ	2 / 2
Daten-Cache (L1/L2)	2x 32KB / 6144KB
Hauptspeicher	4GB (2x 2GB DDR2-800)
Netzwerke	Gigabit
verwendete JVM	Sun Java 1.6, 64Bit

gekko

Prozessor	Intel Pentium 4 <i>HT</i> (<i>Northwood</i>), 3.2GHz
Anzahl Prozessoren	1
Kerne pro Prozessor / Σ	1 / 1
Daten-Cache (L1/L2)	8KB / 512KB
Hauptspeicher	2GB (4x 512MB DDR PC3200 400MHz)
Netzwerke	Gigabit
verwendete JVM	Sun Java 1.5

hydra

Prozessor	Intel Xeon Quad-Core (<i>E7330 Tigerton</i>), 2,4GHz
Anzahl Prozessoren	4
Kerne pro Prozessor / Σ	4 / 16
Daten-Cache (L1/L2)	4x32KB / 2x 3072KB
Hauptspeicher	16GB (8x 2GB DDR-2 PC5300 667MHz)
Netzwerke	Gigabit
verwendete JVM	Sun Java 1.6, 64Bit

hydrus

Prozessor	AMD Opteron Quad-Core (<i>8350 Barcelona</i>), 2GHz
Anzahl Prozessoren	4
Kerne pro Prozessor / Σ	4 / 16
Daten-Cache (L1/L2)	4x64KB / 4x512KB
Hauptspeicher	32GB (16x 2GB DDR-2 PC5300 667MHz)
Netzwerke	Gigabit
verwendete JVM	Sun Java 1.6, 64Bit

cluster (Frontendknoten des Clusters)

Prozessor	AMD Opteron (<i>244 Troy</i>), 1,8GHz
Anzahl Prozessoren	2
Kerne pro Prozessor / Σ	1 / 2
Daten-Cache (L1/L2)	64KB / 1024KB
Hauptspeicher	4GB (4x 256MB DDR PC333)
Netzwerke	Fast Ethernet, Gigabit, InfiniBand (10GBit/s)
verwendete JVM	Sun Java 1.6, 64Bit

clust01 (ein einzelner Clusterknoten)

Prozessor	AMD Opteron (<i>246 Troy</i>), 2GHz
Anzahl Prozessoren	2
Kerne pro Prozessor / Σ	1 / 2
Daten-Cache (L1/L2)	64KB / 1024KB
Hauptspeicher	4GB (4x 1GB DDR PC400)
Netzwerke	Fast Ethernet, Gigabit, InfiniBand (10GBit/s)
verwendete JVM	Sun Java 1.6, 64Bit

poolsrv

Prozessor	Intel Core TM 2 Duo (<i>E8400 Wolfdale</i>), 3GHz
Anzahl Prozessoren	1
Kerne pro Prozessor / Σ	2 / 2
Daten-Cache (L1/L2)	2x 32KB / 6144KB
Hauptspeicher	4GB (2x 2GB DDR2-800)
Netzwerke	Gigabit
verwendete JVM	Sun Java 1.6, 64Bit

storage

Prozessor	Intel Core™ 2 Duo (<i>E6600 Conroe</i>), 2,4GHz
Anzahl Prozessoren	1
Kerne pro Prozessor / Σ	2 / 2
Daten-Cache (L1/L2)	2x 32KB / 4096KB
Hauptspeicher	4GB, (2x 1GB DDR2-667 ECC Samsung)
Netzwerke	Gigabit
verwendete JVM	Sun Java 1.6, 64Bit

sun

Prozessor	UltraSPARC III Cu, 1050 MHz
Anzahl Prozessoren	2
Kerne pro Prozessor / Σ	1 / 2
Daten-Cache (L1/L2)	64KB / 8192KB
Hauptspeicher	3GB
Netzwerke	Gigabit
verwendete JVM	Sun Java 1.5

knecht (Wyse ThinClient)

Prozessor	AMD K6-III, 400MHz
Anzahl Prozessoren	1
Kerne pro Prozessor / Σ	1 / 1
Daten-Cache (L1/L2)	32KB / 256KB
Hauptspeicher	256MB
Netzwerke	Fast Ethernet
verwendete JVM	Sun Java 1.6

ps3 (Sony Playstation 3)

Prozessor	IBM Cell, 3,2GHz
Anzahl Prozessoren	1
Kerne pro Prozessor / Σ	1x PPE, 7x SPE / 8 nur die PPE ist mit Java nutzbar
Daten-Cache (L1/L2)	64KB / 8192KB
Hauptspeicher	256MB
Netzwerke	Gigabit
verwendete JVM	IBM Java 1.6, 32Bit

nemesis (HP Integrity rx5670)

Prozessor	Intel Itanium-2 (<i>Madison</i>), 1.5GHz
Anzahl Prozessoren	4
Kerne pro Prozessor / Σ	1 / 4
Daten-Cache (L1/L2)	16KB / 256KB
Hauptspeicher	2GB (4x 512MB DDR PC3200 400MHz)
Netzwerke	Gigabit
verwendete JVM	Bea JRockit Java 1.5

B.2.2 Laufzeitkonfigurationen

Die Tabelle B.1 zeigt verschiedene Konfigurationen, welche bei der Durchführung der im Rahmen dieser Arbeit durchgeführten Laufzeittests verwendet wurden. Als Konfiguration wird dabei die Zusammensetzung einer *Beluga* Laufzeitumgebung aus einzelnen Computersystemen bezeichnet. Jede dieser Konfigurationen besitzt eine Vorgabe für die Anzahl von maximal zu startenden Trails und deren Verteilung auf die vorhandenen Hostsysteme. Diese Vorgaben wurden als Standard bei den durchgeführten Tests verwendet, Abweichungen hiervon sind jeweils dokumentiert.

B.2.3 Bestimmung von Leistungsmerkmalen

Zur Bestimmung der Leistungsfähigkeit eines Computers oder bestimmter darin enthaltener Baugruppen reicht es oftmals nicht aus, die technischen Spezifikationen dieser Bauteile zu kennen. Das Zusammenspiel mehrerer Komponenten entscheidet oftmals über die im praktischen Anwendungsfall erzielte Performance. Aus diesem Grund gibt es eine Fülle von Benchmarkprogrammen, welche solche Daten auf einem Computersystem bestimmen.

Im Rahmen dieser Arbeit wurde die Hauptspeicherbandbreiten der verwendeten Testsysteme ermittelt. Die hierfür eingesetzten Benchmarkprogramme sollen deshalb an dieser Stelle kurz vorgestellt werden.

bandwidth Tool

Mit dem Tool `bandwidth`¹ wurden die in der Arbeit verwendeten Hauptspeicherbandbreiten ermittelt. Die Routinen des Tools sind in Assembler geschrieben, um möglichst wenig Overhead zu erzeugen und direkte Kontrolle über die erzeugte Codesequenz zu bekommen. Allerdings unterstützt der zur Verfügung stehende *NASM* Assembler noch keine 64Bit-Architekturen. Deshalb wurden alle Test mit einer 32Bit-Version durchgeführt.

Stream und Stream2 Tools

Der *Stream* Benchmark² und der weiterentwickelte *Stream2* Benchmark³ ermitteln ebenfalls Hauptspeicherbandbreiten. Die eingesetzten Verfahren sind jedoch wesentlich komplexer als das Verfahren des `bandwidth` Tools. Um eine bessere Vergleichbarkeit mit dem erstellten Java-Benchmark zu gewährleisten, wurde das einfachere `bandwidth` Tool als Referenzmethode zur Bestimmung der Bandbreiten verwendet. Die *Stream* Benchmarks dienen zur Überprüfung der so gewonnenen Meßdaten.

¹Benutzt wurde die Version 0.15, <http://home.comcast.net/~fbui/bandwidth.html>

²<http://www.cs.virginia.edu/stream/>

³<http://www.cs.virginia.edu/stream/stream2/>

ANHANG B. LAUFZEITSYSTEM UND TESTUMGEBUNGEN

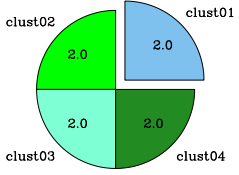
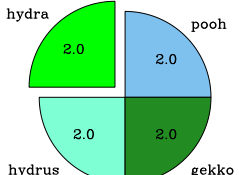
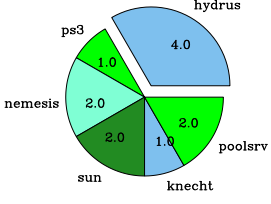
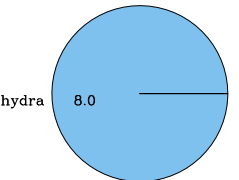
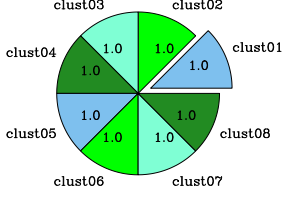
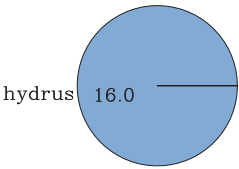
Konfiguration	Systeme	Bemerkungen	Trailverteilung
K1	clust01 clust02 clust03 clust04	Prozessorkerne: 8 Trails: 8 Registry: clust01	
K2	hydra hydrus pooh gekko	Prozessorkerne: 35 Trails: 8 Registry: pooh	
K3	hydrus ps3 nemesis sun knecht poolsrv	Prozessorkerne: 26 Trails: 12 Registry: hydrus	
K4	hydra	Prozessorkerne: 16 Trails: 8 Registry: hydra	
K5	clust01 clust02 clust03 clust04 clust05 clust06 clust07 clust08	Prozessorkerne: 16 Trails: 8 Registry: clust01	
K6	hydrus	Prozessorkerne: 16 Trails: 16 Registry: hydrus	

Tabelle B.1: Auflistung aller Konfigurationen

Literaturverzeichnis

- [1] ADVE, S. V. ; GHARACHORLOO, K.: Shared Memory Consistency Models: A Tutorial. In: *IEEE Computer* 29 (1996), Nr. 12, S. 66–76. – URL citeseer.ist.psu.edu/adve95shared.html
- [2] AHMADINIA, Ali ; BOBDA, Christophe ; KOCH, Dirk ; MAJER, Mateusz ; TEICH, Jürgen: Task Scheduling for Heterogeneous Reconfigurable Computers. In: *Proceedings of the 17th Symposium on Integrated Circuits and Systems Design (SBCCI)*. Pernambuco, Brazil : ACM Press, September 2004, S. 22–27
- [3] AHO, Alfred V. ; LAM, Monica S. ; SETHI, Ravi ; ULLMAN, Jeffrey D.: *Compilers: Principles, Techniques, and Tools*. 2nd Edition. Addison Wesley, August 2006. – 1000 S. – ISBN 978-0321486811
- [4] AHUJA, Sudhir ; CARRIERO, Nicholas ; GELERNTER, David: Linda and Friends. In: *Computer* 19 (1986), August, Nr. 8, S. 26–34
- [5] AKHTER, Shameem ; ROBERTS, Jason: *Multi-Core Programming: Increasing Performance through Software Multithreading*. Intel Press, April 2006. – 336 S. – ISBN 978-0976-48324-3
- [6] AMDAHL, Gene M.: Validity of the single processor approach to achieving large scale computing capabilities. In: *Proceedings of the spring joint computer conference, AFIPS '67 (Spring)*. New York, NY : ACM Press, April 1967, S. 483–485
- [7] AMZA, Christiana ; COX, Alan L. ; DWARKADAS, Hya ; KELEHER, Pete ; LU, Honghui ; RAJAMONY, Ramakrishnan ; YU, Weimin ; ZWAENEOEL, Willy: Treadmarks: Shared memory computing on networks of workstations. In: *IEEE Computer* 29 (1996), S. 18–28
- [8] ANTONIU, Gabriel ; BOUGÉ, Luc ; JAN, Mathieu: Peer-to-Peer Distributed Shared Memory? In: *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT 2003)*. New Orleans, LA : IEEE Computer Society Press, September 2003, S. 1–6. – URL <http://hal.inria.fr/inria-00000981/en>. – Work in Progress Session
- [9] ANTONIU, Gabriel ; BOUGÉ, Luc ; JAN, Mathieu: JuxMem: An Adaptive Supportive Platform for Data Sharing on the Grid. In: *Scalable Computing: Practice and Experience* 6 (2005), September, Nr. 3, S. 45–55. – URL <http://hal.inria.fr/inria-00000984/en>

-
- [10] ARNOLD, Ken ; SCHEIFLER, Robert ; WALDO, Jim ; O'SULLIVAN, Bryan ; WOLLRATH, Ann: *The JINI Specification*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., 1999. – ISBN 0201616343
- [11] ASANOVIC, Krste ; BODIK, Ras ; CATANZARO, Bryan C. ; GEBIS, Joseph J. ; HUSBANDS, Parry ; KEUTZER, Kurt ; PATTERSON, David A. ; PLISHKER, William L. ; SHALF, John ; WILLIAMS, Samuel W. ; YELICK, Katherine A.: The Landscape of Parallel Computing Research: A View from Berkeley / EECS Department, University of California, Berkeley. URL <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>, Dezember 2006 (UCB/EECS-2006-183). – Forschungsbericht
- [12] BAL, Henri E. ; KAASHOEK, M. F. ; TANENBAUM, Andrew S.: Orca: A language for parallel programming of distributed systems. In: *IEEE Transactions on Software Engineering* 18 (1992), März, Nr. 3, S. 190–205. – URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=126768. – ISSN 0098-5589
- [13] BALAJI, Pavan ; BUNTINAS, Darius ; GOODELL, David ; GROPP, William ; THAKUR, Rajeev: Toward Efficient Support for Multithreaded MPI Communication. In: *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Dublin, Ireland : Springer Verlag, September 2008 (Lecture Notes In Computer Science), S. 120–129. – ISBN 978-3-540-87474-4
- [14] BALL, Carwyn ; BULL, Mark: Barrier Synchronisation in Java / UKHEC. URL <http://www.ukhec.ac.uk/publications/>, 2003. – Forschungsbericht
- [15] BANGALORE, Rajiv: *Use XStream to serialize Java objects into XML*. IBM developerWorks. Juli 2008. – URL <http://www.ibm.com/developerworks/xml/library/x-xstream/>
- [16] BARNES, Joshua E. ; HUT, Piet: A hierarchical $O(N \log N)$ force-calculation algorithm. In: *Nature* 324 (1986), Dezember, Nr. 6096, S. 446–449. – URL <http://dx.doi.org/10.1038/324446a0>
- [17] BENNETT, John K. ; CARTER, John B. ; ZWAENEPOEL, Willy: *The Cache Coherence Problem in Shared-Memory Multiprocessors: Software Solutions*. Kap. Adaptive software cache management for distributed shared memory architectures, S. 212–222, Wiley-IEEE Computer Society Press, Februar 1996. – ISBN 978-0-8186-7096-1
- [18] BLOCK, Aaron ; LEONTYEV, Hennadiy ; BRANDENBURG, Björn B. ; ANDERSON, James H.: A Flexible Real-Time Locking Protocol for Multiprocessors. In: *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*. Daegu, Korea : IEEE Computer Society Press, August 2007, S. 47–56

-
- [19] BOGER, Marko ; WIENBERG, Frank ; LAMERSDORF, Winfried: Dejay: Concepts for a Distributed Java. In: *Proceedings of Distributed Computing on the Web (DCW99)* Universität Rostock (Veranst.), Juni 1999, S. 8
- [20] BONACHEA, Dan: Proposal for Extending the UPC Memory Copy Library Functions and Supporting Extensions to GASNet, v2.0 / Lawrence Berkeley National Lab. URL <http://upc.lbl.gov/publications/>, März 2007 (LBNL-56495). – Forschungsbericht
- [21] BRITO, Andrey ; FETZER, Christof ; STURZREHM, Heiko ; FELBER, Pascal: Speculative out-of-order event processing with software transaction memory. In: *Proceedings of the 2nd International Conference on Distributed Event-Based Systems*. Rome, Italy : ACM Press, Juli 2008 (ACM International Conference Proceeding Series), S. 265–275
- [22] BUTENHOF, David R.: *Programming with POSIX Threads*. Addison Wesley, Mai 1997 (Professional Computing Series). – 400 S. – ISBN 978-0201633924
- [23] CARLSON, William W. ; DRAPER, Jesse M. ; CULLER, David ; YELICK, Kathy ; BROOKS, Eugene ; WARREN, Karren: Introduction to UPC and Language Specification / Center for Computing Sciences, IDA. Bowie, MD, Mai 1999 (CCS-TR-99-157). – Forschungsbericht
- [24] CARRIERO, Nicholas J. ; GELERNTER, David ; MATTSON, Timothy G. ; SHERMAN, Andrew H.: The Linda alternative to message-passing systems. In: *Parallel Computing* 20 (1994), April, Nr. 4, S. 633–655. – URL <http://www.sciencedirect.com/science/article/B6V12-4998WMW-57/2/d87327b92772d53e0347229609bd570b>
- [25] CARTER, J.B. ; KHANDEKAR, D. ; KAMB, L.: Distributed shared memory: where we are and where we should beheaded. In: *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*. Orcas Island, WA, USA, Mai 1995, S. 119–122. – ISBN 0-8186-7081-9
- [26] CHANDRA, Robit ; DAGUM, Leonardo ; KOHR, Dave ; MAYDAN, Dror ; MCDONALD, Jeff ; MENON, Ramesh: *Parallel programming in OpenMP*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2001. – ISBN 1-55860-671-8
- [27] COOMES, John ; PRINTEZIS, Antonios: Performance Through Parallelism: Java HotSpot Virtual Machine Garbage Collection Improvements. In: *JavaOne Conference*. San Francisco, CA, Mai 2006
- [28] COURTOIS, P. J. ; HEYMANS, F. ; PARNAS, D. L.: Concurrent control with “readers” and “writers”. In: *Communications of the ACM* 14 (1971), Oktober, Nr. 10, S. 667–668. – ISSN 0001-0782

-
- [29] CROUSE, Bernd: *Lattice-Boltzmann Strömungssimulationen auf Baumdatenstrukturen*, Lehrstuhl für Bauinformatik Fakultät für Bauingenieur- und Vermessungswesen Technische Universität München, Dissertation, 2003. – URL <http://tumb1.biblio.tu-muenchen.de/publ/diss/bv/2003/crouse.pdf>
- [30] CULLER, David E. ; SINGH, Jaswinder P. ; GUPTA, Anoop: *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, August 1998 (The Morgan Kaufmann Series in Computer Architecture and Design). – 1100 S. – ISBN 1-55860-343-3
- [31] DESAI, Nirmitt ; MUELLER, Frank: Scalable Distributed Concurrency Services for Hierarchical Locking. In: *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003)*. Providence, RI : IEEE Computer Society Press, Mai 2003, S. 530–537. – ISBN 0-7695-1920-2
- [32] DESAI, Nirmitt ; MUELLER, Frank: Scalable Hierarchical Locking for Distributed Systems. In: *Journal of Parallel and Distributed Computing* 64 (2004), Juni, Nr. 6, S. 708–724. – Special Issue on Middleware. – ISSN 0743-7315
- [33] DEVER, Steve ; DETLEFS, David ; GOLDMAN, Steve ; RUSSELL, Kenneth: New Compiler Optimizations in the Java HotSpot Virtual Machine. In: *JavaOne Conference*. San Francisco, CA, Mai 2006. – URL <http://developers.sun.com/learning/javaoneonline/j1sessn.jsp?sessn=TS-3412&yr=2006&track=coreplatform>
- [34] DUBOIS, Michel ; RICCIULLI, Livio ; RAMAMURTHY, Krishnan ; STENSTRÖM, Per: The Detection and Elimination of Useless Misses in Multiprocessors. In: *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*. San Diego, CA : IEEE Computer Society Press, Mai 1993, S. 88–97
- [35] DÜMMLER, Jörg ; RAUBER, Thomas ; RÜNGER, Gudula: Mapping Algorithms for Multiprocessor Tasks on Multi-Core Clusters. In: *Proceedings of the 37th International Conference on Parallel Processing (ICPP 2008)*. Portland, OR : IEEE Computer Society Press, September 2008, S. 141–148. – ISBN 978-0-7695-3374-2
- [36] EL-GHAZAWI, Tarek ; CARLSON, William ; STERLING, Thomas ; YELICK, Katherine ; ZOMAYA, Albert Y. (Hrsg.): *UPC: Distributed Shared Memory Programming*. Hoboken, New Jersey : John Wiley & Sons, Juni 2005 (Wiley Series on Parallel and Distributed Computing). – 252 S. – URL <http://dmmsclick.wiley.com/click.asp?s=680227&m=1617&u=17365>. – ISBN 978-0-471-22048-5
- [37] FLYNN, Micheal J.: Some Computer Organizations and Their Effectiveness. In: *IEEE Transactions on Computing* 9 (1972), September, Nr. 9, S. 948–960
- [38] FONTOURA, Marcus ; LEHMAN, Tobin J. ; NELSON, Dwayne ; TRUONG, Thomas ; XIONG, Yuhong: TSpaces Services Suite: Automating the Development and Management of Web Services. In: *Proceedings of the Twelfth International World Wide*

- Web Conference - Alternate Paper Tracks (WWW2003)*. Budapest, Hungary : ACM Press, Mai 2003
- [39] FOSTER, I. ; KISHIMOTO, H. ; SAVVA, A. ; BERRY, D. ; DJAOUI, A. ; GRIMSHAW, A. ; HORN, B. ; MACIEL, F. ; SIEBENLIST, F. ; SUBRAMANIAM, R. ; TREADWELL, J. ; REICH, J. v.: *The Open Grid Services Architecture, Version 1.0*. Global Grid Forum (GGF). Januar 2005. – URL <http://www.ggf.org>
- [40] FOSTER, Ian (Hrsg.) ; KESSELMAN, Carl (Hrsg.): *The Grid: Blueprint for a New Computing Infrastructure*. 2. Auflage. San Francisco, CA : Morgan Kaufmann Publishers, 2004 (The Elsevier Series in Grid Computing). – 748 S. – ISBN 978-1-55860-933-4
- [41] FREEMAN, Eric ; HUPFER, Susanne ; ARNOLD, Ken: *Javaspace Principles, Patterns, and Practice*. Addison-Wesley Longman, Juni 1999 (The Jini Technology Series). – ISBN 978-0201309553
- [42] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John M.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA : Addison-Wesley Longman Publishing Co., Inc., Oktober 1995. – 416 S. – ISBN 987-02016-3361-2
- [43] GELERTER, David: Generative Communication in Linda. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7 (1985), Januar, Nr. 1, S. 80–112. – URL <http://doi.acm.org/10.1145/2363.2433>
- [44] GOETZ, Brian ; PEIERLS, Tim ; BLOCH, Joshua ; BOWBEER, Joseph ; HOLMES, David ; LEA, Doug: *Java concurrency in practice*. Addison-Wesley, Juli 2007. – 403 S. – ISBN 978-0321-34960-6
- [45] GOSLING, James ; JOY, Bill ; STEELE, Guy ; BRACHA, Gilad: *The Java Language Specification*. 3. Auflage. Addison-Wesley, Juni 2005 (The Java Series). – 668 S. – URL <http://java.sun.com/docs/books/jls/index.html>
- [46] GRIEBEL, M. ; KNAPEK, S. ; ZUMBUSCH, G. ; CAGLAR, A.: *Numerische Simulation in der Moleküldynamik. Numerik, Algorithmen, Parallelisierung, Anwendungen*. Berlin, Heidelberg : Springer, 2004. – ISBN 3-540-41856-3
- [47] GRIES, David: The Schorr-Waite graph marking algorithm. In: *Acta Informatica* 11 (1979), September, Nr. 3, S. 223–232. – URL <http://dx.doi.org/10.1007/BF00289068>
- [48] GROPP, William ; HUSS-LEDERMAN, Steven ; LUMSDAINE, Andrew ; LUSK, Ewing ; NITZBERG, Bill ; SAPHIR, William ; SNIR, Marc: *MPI – The Complete Reference: Volume 2, The MPI-2 Extensions*. 2. Auflage. MIT Press, 1999. – URL <http://mitpress.mit.edu/book-home.tcl?isbn=0262571234>. – ISBN 978-0-262-57123-4

- [49] GRUNWALD, Dirk ; VAJRACHARYA, Suvas: Efficient barriers for distributed shared memory computers. In: *Proceedings of the 8th International Symposium on Parallel Processing (IPPS-94)*. Cancún, Mexico : IEEE Computer Society Press, April 1994, S. 604–608. – ISBN 0-8186-5602-6
- [50] GSCHWIND, Michael: The cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor. In: *International Journal of Parallel Programming* 35 (2007), Juni, Nr. 3, S. 233–262. – URL [domino.research.ibm.com/library/cyberdig.nsf/papers/1B2480A9DBF5B9538525723D0051A8C1/\\$File/rc24128.pdf](http://domino.research.ibm.com/library/cyberdig.nsf/papers/1B2480A9DBF5B9538525723D0051A8C1/$File/rc24128.pdf). – ISSN 0885-7458
- [51] HAMMOND, Lance ; WONG, Vicky ; CHEN, Mike ; CARLSTROM, Brian D. ; DAVIS, John D. ; HERTZBERG, Ben ; PRABHU, Manohar K. ; WIJAYA, Honggo ; KOZYRAKIS, Christos ; OLUKOTUN, Kunle: Transactional memory coherence and consistency. In: *Proceeding of the 31st International Symposium on Computer Architecture (ISCA 2004)*. München : IEEE Computer Society Press, Juni 2004, S. 102–113. – ISBN 0-7695-2143-6
- [52] HANSCHKE, B. ; VUCENIC, W.: On the n -queens problem. In: *Notices of the American Mathematical Society* 20 (1973), S. 568
- [53] HAUMACHER, Bernhard: *Parallel Programming Environment for Computer Clusters*, IPD Tichy, University of Karlsruhe, Germany, Dissertation, Dezember 2005
- [54] HEIN, Mathias: *TCP/ IP. Internet- Protokolle im professionellen Einsatz*. 4. Auflage. International Thomson Publishing, 1998
- [55] HILL, Mark D.: *Aspects of cache memory and instruction buffer performance*, University of California, Berkeley, Dissertation, 1987. – Chairman-Alan Jay Smith
- [56] HIPPOLD, Judith ; RÜNGER, Gudula: A Communication API for Implementing Irregular Algorithms on SMP Clusters. In: *Proceedings of the 10th European PV-M/MPI* Bd. 2840. Venedig, Italien : Springer Verlag, September 2003, S. 455–463. – ISBN 3-540-20149-1
- [57] HOARE, Charles Antony R.: Monitors: An Operating System Structuring Concept. In: *Communications of the ACM* 17 (1974), Oktober, Nr. 10, S. 549–557. – ISSN 0001-0782
- [58] HOARE, Charles Antony R.: Communicating sequential processes. In: *Communications of the ACM* 26 (1983), Nr. 1, S. 100–106. – ISSN 0001-0782
- [59] HOFFMANN, Ralf ; RAUBER, Thomas: Fine-grained task scheduling using adaptive data structures. In: *Proceedings of Euro-Par 2008*, Springer, 2008 (LNCS)
- [60] HSIEH, Wilson C. ; WEIHL, William E.: Scalable Reader-Writer Locks for Parallel Systems. In: *Proceedings of the 6th International Parallel Processing Symposium*

- (*IPPS-92*). Washington, DC : IEEE Computer Society Press, März 1992, S. 656–659. – ISBN 0-8186-2672-0
- [61] HUNOLD, Sascha ; RAUBER, Thomas: Reducing the Overhead of Intra-Node Communication in Clusters of SMPs. In: *Proceedings of the 3rd International Symposium on Parallel and Distributed Processing and Applications (ISPA 2005)*. Nanjing, China : Springer Verlag, November 2005
- [62] HUNOLD, Sascha ; RAUBER, Thomas ; RÜNGER, Gudula: Dynamic Scheduling of Multi-Processor Tasks on Clusters of Clusters. In: *Proceedings of the Sixth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (Heteropar'07)*, Austin, TX, IEEE Computer Society Press, 2007
- [63] INFINIBAND TRADE ASSOCIATION: *InfiniBand Architecture Specification Volume 1 Release 1.2.1*. November 2007. – URL www.infinibandta.org
- [64] INTEL CORPORATION: *Intel Pentium 4 Processor 6x1 Sequence Datasheet*. Dezember 2006. – URL <http://www.intel.com/design/Pentium4/documentation.htm>
- [65] INTEL CORPORATION: First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem) / Intel Corporation. URL www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf, Juli 2008. – White Paper
- [66] INTEL CORPORATION: Intel 64 and IA-32 Architectures Software Developers Manual, Volume 1: Basic Architecture / Intel Corporation. URL <http://www.intel.com/products/processor/manuals/index.htm>, November 2008 (Order Number: 253665-029US). – Manual
- [67] JEDEC: *Double Data Rate (DDR) SDRAM Specifications (JESD79D)*. JEDEC Solid State Technology Association. Januar 2004. – URL www.jedec.org
- [68] JEDEC: *DDR3 SDRAM (JESD79-3C)*. JEDEC Solid State Technology Association. November 2008. – URL www.jedec.org
- [69] JUDGE, Alan ; NIXON, Paddy A. ; CAHILL, Vinny J. ; TANGNEY, Brendan ; WEBER, Stefan: Overview of distributed shared memory / Department of Computer Science, Trinity College. Dublin, Ireland, Oktober 1998 (TCD-CS-1998-24). – Forschungsbericht. – URL citeseer.ist.psu.edu/judge98overview.html
- [70] KAELI, D. R. ; FONG, L. L. ; BOOTH, R. C. ; IMMING, K. C. ; WEIGEL, J. P.: Performance analysis on a CC-NUMA prototype. In: *IBM Journal of Research and Development* 41 (1997), Nr. 3, S. 205–214. – ISSN 0018-8646
- [71] KAMBITES, M. E. ; OBDRŽÁLEK, J. ; BULL, J. M.: An OpenMP-like interface for parallel programming in Java. In: *Concurrency and Computation: Practice and Experience* 13 (2001), Nr. 8–9, S. 793–814

-
- [72] KARONIS, Nicholas T. ; TOONEN, Brian ; FOSTER, Ian: MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. In: *Journal of Parallel and Distributed Computing (JPDC)* 63 (2003), Mai, Nr. 5, S. 551–563. – ISSN 0743-7315
- [73] KELEHER, Pete ; COX, Alan L. ; DWARKADAS, Hya ; ZWAENEPOEL, Willy: TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In: *Proceedings of the USENIX Winter 1994 Technical Conference*. San Francisco, CA : USENIX Association, Berkeley, Januar 1994, S. 115–132
- [74] KREMER, Kurt (Hrsg.) ; MACHO, Volker (Hrsg.): *Forschung und wissenschaftliches Rechnen: Beiträge zum Heinz-Billing-Preis 2006*. Göttingen : Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen, 2007. (GWDG-Bericht 72). – URL <http://www.billingpreis.mpg.de/>
- [75] KÜHN, Eva (Hrsg.): *Virtual shared memory for distributed architectures*. Huntington, NY : Nova Science Publishers, 2001. – 112 S. – ISBN 1-59033-101-X
- [76] LAHJOMRI, Zakaria ; PRIOL, Thierry: *Lecture Notes in Computer Science*. Bd. 634: *Parallel Processing: CONPAR 92-VAPP V*. Kap. KOAN: A Shared Virtual Memory for the iPSC/2 hypercube, S. 441–452. Berlin/Heidelberg : Springer Verlag, 1992. – ISBN 978-3-540-55895-8
- [77] LAMPORT, Leslie: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. In: *IEEE Transactions on Computers* 28 (1979), September, Nr. 9, S. 690–691. – ISSN 0018-9340
- [78] LARUS, James R. ; LARUS, Jim ; RAJWAR, Ravi ; HILL, Mark D. (Hrsg.): *Transactional memory*. Morgan & Claypool Publishers, 2007 (Synthesis lectures on computer architecture). – 211 S
- [79] LEA, Doug: *Concurrent programming in Java: Design Principles and Pattern*. 2. Boston : Addison-Wesley, 2002 (The Java Series). – 411 S
- [80] LEHMAN, Tobin J. ; MCLAUGHRY, Stephen W. ; WYCKOFF, Peter: TSpaces: The Next Wave. In: *32nd Annual Hawaii International Conference on System Sciences HICSS-32*. Maui, Hawaii : IEEE Computer Society Press, Januar 1999. – URL citeseer.ist.psu.edu/lehman99spaces.html
- [81] LENOSKI, Daniel ; LAUDON, James ; GHARACHORLOO, Kouros ; WEBER, Wolf-Dietrich ; GUPTA, Anoop ; HENNESSY, John ; HOROWITZ, Mark ; LAM, Monica S.: The Stanford Dash Multiprocessor. In: *Computer* 25 (1992), Nr. 3, S. 63–79. – ISSN 0018-9162
- [82] LI, Kai: *Shared Virtual Memory on Loosely Coupled Multiprocessors*, Yale University, Dissertation, 1986

- [83] LI, Kai: IVY: A Prototype Shared Virtual Memory System for Parallel Computing. In: *Proceedings of the 1988 International Conference on Parallel Computing (ICPP '88)* Bd. 2 Software. The Pennsylvania State University, University Park, PA : IEEE Computer Society Press, August 1988, S. 94–101. – ISBN 0-271-00654-4
- [84] LI, Kai ; HUDAK, Paul: Memory coherence in shared virtual memory systems. In: *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing (PODC)*. New York, NY : ACM Press, 1989, S. 229–239. – URL citeseer.ist.psu.edu/li89memory.html
- [85] LINDHOLM, E. ; NICKOLLS, J. ; OBERMAN, S. ; MONTRYM, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. In: *IEEE Micro* 28 (2008), März, Nr. 2, S. 39–55. – ISSN 0272-1732
- [86] LINDSEY, Clark S. ; TOLLIVER, Johnny S. ; LINDBLAD, Thomas: *JavaTech, an Introduction to Scientific and Technical Computing with Java*. New York, NY : Cambridge University Press, 2005. – 708 S. – ISBN 978-0-521-82113-1
- [87] LUCHANGCO, Victor ; NUSSBAUM, Daniel ; SHAVIT, Nir: A Hierarchical CLH Queue Lock. In: *Proceedings of the 12th International Euro-Par Conference for Parallel Processing (EuroPar 2006)*. Dresden : Springer Verlag, September 2006 (Lecture Notes in Computer Science Bd. 4128), S. 801–810. – ISBN 3-540-37783-2
- [88] MARR, Deborah T. ; BINNS, Frank ; HILL, David L. ; HINTON, Glenn ; KOUFATY, David A. ; MILLER, J. A. ; UPTON, Micheal: Hyper-Threading Technology Architecture and Microarchitecture. In: *Intel Technology Journal* 6 (2002), 8, Nr. 1, S. 4–16. – URL <http://www.intel.com/technology/itj/archive/2002.htm>
- [89] MEUER, Hans W.: The TOP500 Project: Looking Back Over 15 Years of Supercomputing Experience. In: *Informatik-Spektrum* 31 (2008), Juni, Nr. 3, S. 203–222. – URL <http://dx.doi.org/10.1007/s00287-008-0240-6>. – ISSN 0170-6012
- [90] MOORE, Gordon E.: Cramming more components onto integrated circuits. In: *Electronics* 38 (1965), April, Nr. 8, S. 114–117. – URL http://www.intel.com/museum/archives/history_docs/mooreslaw.htm
- [91] MUNSHI, Aaftab: *OpenCL Core API Specification*. Februar 2009. – URL <http://www.khronos.org/opencv/>
- [92] MUNSON, Bruce R. ; YOUNG, Donald F. ; OKIISHI, Theodore H. ; HUEBSCH, Wade W.: *Fundamentals of Fluid Mechanics*. 6th. New York, NY : John Wiley & Sons, Januar 2009. – 776 S. – ISBN 978-0470262849
- [93] NAUCK, Franz: Schach. In: *Illustrierter Zeitung* 361 (1850), S. 352
- [94] NEUHIERL, Barbara: *Strömungsakustik und Fluid-Struktur-Kopplung mit der Lattice-Boltzmann- und der Finite-Element-Methode*, Lehrstuhl für Bauinformatik

- Fakultät für Bauingenieur- und Vermessungswesen Technische Universität München, Dissertation, 2008. – URL <http://mediatum2.ub.tum.de/doc/631373/631373.pdf>. – Zugriffsdatum: 26.08.2008
- [95] NIEPLOCHA, Jarek ; HARRISON, Robert J. ; LITTLEFIELD, Richard J.: Global arrays: A nonuniform memory access programming model for high-performance computers. In: *The Journal of Supercomputing* 10 (1996), Juni, Nr. 2, S. 169–189. – URL <http://dx.doi.org/10.1007/BF00130708>
- [96] NIEPLOCHA, Jarek ; KRISHNAN, Manojkumar ; TIPPARAJU, Vinod ; PALMER, Bruce: *Global Arrays User Manual*. Pacific Northwest National Laboratory, Richland, WA: . – URL <http://www.emsl.pnl.gov/docs/global/>
- [97] NIEPLOCHA, Jarek ; TIPPARAJU, Vinod ; KRISHNAN, Manojkumar ; PANDA, D. K.: High Performance Remote Memory Access Communication: The Armc1 Approach. In: *International Journal of High Performance Computing Applications* 20 (2006), Mai, Nr. 2, S. 233–253. – URL <http://hpc.sagepub.com/cgi/content/abstract/20/2/233>
- [98] NIEUWPOORT, Rob V. van ; KIELMANN, Thilo ; BAL, Henri E.: Satin: Efficient parallel divide-and-conquer in java. In: *Proceedings of the 6th International Euro-Par Conference for Parallel Processing (EuroPar 2000)*. München : Springer Verlag, September 2000 (Lecture Notes in Computer Science Bd. 1900), S. 690–699. – ISBN 3-540-67956-1
- [99] OAKS, Scott ; WONG, Henry ; LOUKIDES, Mike (Hrsg.): *Java Threads*. 3. O'Reilly, September 2004. – ISBN 978-0596-00782-9
- [100] OECHSLE, Rainer: *Parallele Programmierung mit Java Threads*. Fachbuchverlag Leipzig, 2002
- [101] OLUKOTUN, Kunle ; HAMMOND, Lance: The Future of Microprocessors. In: *QUEUE* 3 (2005), September, Nr. 7, S. 26–29. – ISSN 1542-7730
- [102] OPENMP ARCHITECTURE REVIEW BOARD: *OpenMP Application Program Interface Version 3.0*. Mai 2008. – URL www.openmp.org
- [103] PHILIPPSEN, Michael ; HAUMACHER, Bernhard ; NESTER, Christian: More Efficient Serialization and RMI for Java. In: *Concurrency: Practice and Experience* 12 (2000), May, Nr. 7, S. 495–518
- [104] POLKA, Lesley A. ; KALYANAM, Huthasana ; HU, Grace ; KRISHNAMOORTHY, Satish: Package Technology to Address the Memory Bandwidth Challenge for Tera-scale Computing. In: *Intel Technology Journal* 11 (2007), August, Nr. 3, S. 197–206. – URL <http://www.intel.com/technology/itj/2007/v11i3/3-bandwidth/1-abstract.htm>

-
- [105] POWERS, Franklin E. J. ; ALAGHBAND, Gita: The Hydra Parallel Programming System: Research Articles. In: *Concurrency and Computation: Practice & Experience* 20 (2008), Nr. 1, S. 1–27. – ISSN 1532-0626
- [106] PROTIĆ, Jelica (Hrsg.) ; TOMAŠEVIĆ, Milo (Hrsg.) ; MILUTINOVIĆ, Veljko (Hrsg.): *Distributed Shared Memory: Concepts and Systems*. Los Alamitos, CA : IEEE Computer Society Press, 1998. – 365 S. – ISBN 0-8186-7737-6
- [107] PUGH, William ; ADVE, Sarita ; LEA, Doug: *JSR 133: Java Memory Model and Thread Specification Revision*. September 2004. – URL <http://jcp.org/aboutJava/communityprocess/final/jsr133/index.html>
- [108] RAINA, Sanjay: Virtual Shared Memory: A Survey of Techniques and Systems / Department of Computer Science, University of Bristol. URL <http://www.cs.bris.ac.uk/Publications/Papers/1000011.pdf>, Dezember 1992 (CSTR-92-36). – Forschungsbericht
- [109] RAUBER, Thomas ; RÜNGER, Gudula: *Parallele Programmierung*. 2. neu bearbeitete und erweiterte Auflage. Springer Verlag, Februar 2007. – 485 S. – ISBN 978-3540465492
- [110] RECIO, R. ; CULLEY, P. ; GARCIA, D. ; HILLAND, J.: *An RDMA Protocol Specification 1.0*. Oktober 2002. – URL www.rdmaconsortium.org
- [111] RIVIN, I. ; VARDI, I. ; ZIMMERMANN, P.: The n -queens problem. In: *The American Mathematical Monthly* 101 (1994), S. 629–639
- [112] SCHORR, H. ; WAITE, W. M.: An efficient machine-independent procedure for garbage collection in various list structures. In: *Communications of the ACM* 10 (1967), August, Nr. 8, S. 501–506. – ISSN 0001-0782
- [113] SEILER, Larry ; CARMEAN, Doug ; SPRANGLE, Eric ; FORSYTH, Tom ; ABRASH, Michael ; DUBEY, Pradeep ; JUNKINS, Stephen ; LAKE, Adam ; SUGERMAN, Jeremy ; CAVIN, Robert ; ESPASA, Roger ; GROCHOWSKI, Ed ; JUAN, Toni ; HANRAHAN, Pat: Larrabee: a many-core x86 architecture for visual computing. In: *Proceedings of the ACM SIGGRAPH 2008 (SIGGRAPH '08)*. New York, NY, USA : ACM, 2008, S. 1–15. – URL <http://dx.doi.org/10.1145/1399504.1360617>. – ISSN 0730-0301
- [114] SHIRAZI, Jack: *Java Performance Tuning*. 2nd Edition. Sebastopol, CA : O'Reilly Media, Inc., Januar 2003. – 592 S. – ISBN 978-0-596-00377-7
- [115] SINNEN, Oliver ; SOUSA, Leonel A.: Communication contention in task scheduling. In: *IEEE Transactions on Parallel and Distributed Systems* 16 (2005), S. 503–515. – ISSN 1045-9219

-
- [116] SNIR, Marc ; OTTO, Steve ; HUSS-LEDERMAN, Steven ; WALKER, David ; DONGARRA, Jack: *MPI – The Complete Reference: Volume 1, The MPI Core.* 2. Cambridge, MA, USA : MIT Press, 1998. – ISBN 978-0-262-69215-1
- [117] SOHDA, Yukihiro ; NAKADA, Hidemoto ; MATSUOKA, Satoshi ; OGAWA, Hirota: Implementation of a portable software DSM in Java. In: *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, ACM Press, 2001, S. 163–172. – URL citeseer.ist.psu.edu/sohda01implementation.html. – ISBN 1-58113-359-6
- [118] SUCCI, Sauro: *The Lattice Boltzmann Equation - For Fluid Dynamics and Beyond.* Oxford University Press, August 2001. – URL http://www.oup.com/us/catalog/general/subject/Physics/Mechanics/?view=usa&ci=9780198503989#Product_Details. – Zugriffsdatum: 26.08.2008. – ISBN 978-0-19-850398-9
- [119] SUN MICROSYSTEMS: *Java SE 6 HotSpot Virtual Machine Garbage Collection Tuning.* 2006. – URL http://java.sun.com/javase/technologies/hotspot/gc/gc_tuning_6.html. – Zugriffsdatum: 12.06.2008
- [120] SUN MICROSYSTEMS: *White Paper: The Java HotSpot Performance Engine Architecture.* 2008. – URL <http://java.sun.com/products/hotspot/whitepaper.html>
- [121] TABOADA, G. L. ; TOURIÑO, J. ; DOALLO, R.: Efficient Java Communication Protocols on High-speed Cluster Interconnects. In: *Proceedings of the 31st IEEE Conference on Local Computer Networks (LCN'06)*. Tampa, FL : IEEE Computer Society, 2006, S. 264–271
- [122] TABOADA, Guillermo L. ; TOURIÑO, Juan ; DOALLO, Ramón: Java Fast Sockets: Enabling high-speed Java communications on high performance clusters. In: *Computer Communications* 31 (2008), November, Nr. 17, S. 4049–4059. – URL <http://www.sciencedirect.com/science/article/B6TYP-4T9VP7J-1/2/5e966946105c6f1a07284a887cfbb1bd>
- [123] TAKANO, Ryousei ; MATSUDA, Motohiko ; KUDOH, Tomohiro ; KODAMA, Yuet-su ; OKAZAKI, Fumihiro ; ISHIKAWA, Yutaka: Effects of packet pacing for MPI programs in a Grid environment. In: *Proceedings of the 2007 IEEE International Conference on Cluster Computing (Cluster07)*. Austin, TX : IEEE Computer Society Press, September 2007, S. 382–391
- [124] TANENBAUM, Andrew S.: *Moderne Betriebssysteme.* 2. Pearson Education Deutschland GmbH (Prentice Hall), 2002. – ISBN 3-8273-7019-1
- [125] TANENBAUM, Andrew S. ; STEEN, Maarten V.: *Distributed Systems: Principles and Paradigms.* Upper Saddle River, NJ : Prentice Hall, 2002. – 803 S. – ISBN 0-13-088893-1

- [126] TAO, Jie ; KUNZE, Marcel ; NOWAK, Fabian ; BUCHTY, Rainer ; KARL, Wolfgang: Performance Advantage of Reconfigurable Cache Design on Multicore Processor Systems. In: *International Journal of Parallel Programming* 36 (2008), Nr. 3, S. 347–360
- [127] TAUBENFELD, Gadi: *Synchronization Algorithms and Concurrent Programming*. Pearson/Prentice Hall, Mai 2006. – 433 S. – ISBN 0-131-97259-6
- [128] TREBIN, H. R. ; BROMMER, P. ; ENGEL, M. ; GÄHLER, F. ; HOCKER, S. ; RÖSCH, F. ; ROTH, J.: Simulating structure and physical properties of complex metallic alloys. In: *2nd European School in Materials Science*, World Scientific, May 2007
- [129] TULLSEN, Dean M. ; EGGERS, Susan J. ; LEVY, Henry M.: Simultaneous Multi-threading: Maximizing On-Chip Parallelism. In: *Proceedings of the 22rd Annual International Symposium on Computer Architecture*. S. Margherita Ligure, Italy : ACM Press, Juni 1995, S. 392–403. – ISBN 0-89791-698-0
- [130] ULLENBOOM, Christian: *Java ist auch eine Insel*. 6., aktualisierte und erweiterte Auflage. Bonn : Galileo Computing, 2007. – URL <http://www.galileocomputing.de/openbook/javainsel6/>
- [131] UPC CONSORTIUM: *UPC Language Specifications, v1.2*. Mai 2005. – URL <http://upc.lbl.gov/publications/>
- [132] VELDEMA, R. ; HOFMAN, R. F. H. ; BHOEDJANG, R. A. F. ; JACOBS, C. J. H. ; BAL, H. E.: Source-level global optimizations for fine-grain distributed shared memory systems. In: *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, ACM Press, 2001, S. 83–92. – ISBN 1-58113-346-4
- [133] WANG, Hsiao-Hsi ; LI, Kuan-Ching ; LU, Ssu-Hsuan ; YANG, Chun-Chieh ; GAUDIOT, Jean-Luc: Design and Implementation of an Agent Home Scheme Strategy for Prefetch-Based DSM Systems. In: *International Journal of Parallel Programming* 36 (2008), Dezember, Nr. 6, S. 521–542. – URL <http://dx.doi.org/10.1007/s10766-007-0069-7>
- [134] WEIK, Martin H.: UNIVAC LARC. In: *A Third Survey of Domestic Electronic Digital Computing Systems*. Aberdeen Proving Ground, ML : Ballistic Research Laboratories, März 1961 (Report No. 1115), S. 958–962. – URL <http://ed-thelen.org/comp-hist/BRL61.html>
- [135] WELLS, G. C. ; CHALMERS, A. G. ; CLAYTON, P. G.: Linda implementations in Java for concurrent systems. In: *Concurrency and Computation: Practice and Experience* 16 (2004), Juni, Nr. 10, S. 1005–1022. – URL <http://dx.doi.org/10.1002/cpe.794>

-
- [136] WILLIAMS, Samuel ; SHALF, John ; OLIKER, Leonid ; KAMIL, Shoaib ; HUSBANDS, Parry ; YELICK, Katherine: The potential of the cell processor for scientific computing. In: *Proceedings of the 3rd conference on Computing frontiers (CF '06)*. New York, NY : ACM Press, 2006, S. 9–20. – URL <http://dx.doi.org/10.1145/1128022.1128027>. – ISBN 1595933026
- [137] WOO, Steven C. ; OHARA, Moriyoshi ; TORRIE, Evan ; SINGH, Jaswinder P. ; GUPTA, Anoop: The SPLASH-2 programs: characterization and methodological considerations. In: *SIGARCH Computer Architecture News* 23 (1995), Nr. 2, S. 24–36. – ISSN 0163-5964
- [138] WRZESINSKA, Gosia ; MAASSEN, Jason ; VERSTOEP, Kees ; BAL, Henri E.: Sartin++: Divide-and-Share on the Grid. In: *Second IEEE International Conference on e-Science and Grid Technologies (e-Science'06)*. Amsterdam : IEEE Computer Society Press, Dezember 2006, S. 61. – URL <http://www.cs.vu.nl/ibis/papers.html>. – ISBN 0-7695-2734-5
- [139] YELICK, Katherine A. ; SEMENZATO, Luigi ; PIKE, Geoff ; MIYAMOTO, Carleton ; LIBLIT, Ben ; KRISHNAMURTHY, Arvind ; HILFINGER, Paul N. ; GRAHAM, Susan L. ; GAY, David ; COLELLA, Phillip ; AIKEN, Alexander: Titanium: A High-performance Java Dialect. In: *Concurrency - Practice and Experience* 10 (1998), September, Nr. 11-13, S. 825–836. – URL <http://titanium.cs.berkeley.edu/>. – An earlier version was presented at the Workshop on Java for High-Performance Network Computing, Palo Alto, CA, Feb. 1998
- [140] ZWAENEPOEL, Willy ; BENNETT, John K. ; CARTER, John B. ; KELEHER, Pete: Munin: distributed shared memory using multi-protocol release consistency. In: *IEEE Computer Society Technical Committee Newsletter on Operating Systems and Application Environments* 5 (1991), Nr. 4, S. 11