Universität Augsburg
Institut für Informatik
Programmierung verteilter Systeme

Entwicklung parallel ablaufender Programme

WS 2007/2008

Seminarband

Bernhard Bauer
Viviane Schöbel

Vorwort

Im Sommersemester 2008 wurde von der Gruppe *Programmierung verteilter Systeme* der Universität Augsburg das Seminar 'Programmierung parallel ablaufender Programme' veranstaltet. Dieser Seminarband umfasst alle studentischen Arbeiten, die im Rahmen des Seminars erstellt wurden.

Den Beginn des Bandes bildet die Arbeit von *Hadi Deniz*, welcher eine umfassende Einführung in das Forschungsgebiet der 'Parallelen Programmierung' bietet. Im Anschluss daran geht *Veysel Sercan Aksu* in seiner Arbeit detailliert auf die Konzepte und Mechanismen zur Synchronierung ein. Für das Verständnis eines parallelen Systems können meist geeignete Modellierungssprachen für die Darstellung des Ablaufs und der Zusammenhänge helfen. Ein Vertreter dieser Sprachen, 'cmUML', wird von *Tadas Sugintas* vorgestellt.

Da die Art des verwendeten Adressraums die eingesetzten Konzepte, Techniken und Sprachen bei der parallelen Programmierung maßgeblich beeinflusst, stellen *Sven Mentl* und *Thomas Eisenbarth* in ihren Arbeiten die Programmierung im gemeinsamen wie auch im verteilten Adressraum detailliert vor. In diesem Rahmen werden verschiedene Vertreter von Programmiersprachen beider Arten beleuchtet. Dieser Band wird mit der Arbeit von *Dominik Bösl* abgerundet, der sich mit dem Debugging paralleler Programme befasst. Er behandelt neben den Grundlagen des Debuggings und deren Einsatzmöglichkeiten im parallelen Umfeld, einen aktuellen parallelen Debugger aus der Eclipse Parallel Tools Platform.

Die in diesem Seminarband vorgestellten Themen spiegeln nur einen Teil der relevanten Themen im Umfeld der parallelen Programmierung wieder. Trotz seines langjährigen Bestehens hat dieser Bereich noch immer Forschungsbedarf, da die Lösung vieler noch bestehender Probleme durch den vermehrte Einsatz von Mehrkern-Prozessoren wieder an Relevanz zunimmt. Demnach ist in der Zukunft hoffentlich mit neuen und interessanten Forschungsresultaten zu rechnen.

Juli 2008 Die Editoren

Inhaltsverzeichnis

Einführung in die Parallele Programmierung

Hadi, Deniz

Universität Augsburg
hadideniz@yahoo.com

Zusammenfassung Parallelität findet in verschiedenen Ebenen eines Computers die Anwendung. In vielen von dieser Ebenen sollten sich die Programmierer keine Gedanken um die Parallelität machen, Nachdem Rechner mit mehreren Prozessoren auf dem Markt sind, sollen sie sich mehr Gedanken über die Programmierung mit verteilten Systemen machen.

1 Einleitung

Da der Umfang und die Komplexität von Problemen, die man nur mit dem Rechner lösen kann, immer größer werden, muss auch entsprechend die Rechenleistung von Rechnern vergrößert werden. Um die Rechenleistung von Rechnern zu verbessern haben sich die Herstellern eine Zeit lang immer mehr Transistoren unter einem Prozessor zusammengestellt, aber seit zwei Jahren statt sind die Computer statt mit einem Prozesskern mit mehreren Prozessorkernen auf dem Markt. Der Grund dafür ist das die Realisierung von noch stärkeren einzelnen Prozessoren aufwändig und schwer ist, aber stattdessen kann man die gleiche sogar bessere Leistung mit Rechnern die mehrere Prozessorkerne haben erreichen. Da die Prozessoren mit vielen Prozessorkernen immer mehr eingesetzt werden, gewinnt auch die Parallelität an Popularität. Programmierer können dies nicht mehr außer Acht lassen. Normalerweise wendet man Parallelität seit 30 Jahren in verschiedenen Ebenen der Computer Technologie an. Parallele Rechner oder parallele Prozessoren wurden bis jetzt nur bei wissenschaftlicher Arbeit eingesetzt da es teuer war, aber heute kann man diese Systeme mit mehreren Prozessorkern günstig kaufen und nach einer Zeit werden nur noch diese Systeme eingesetzt. Daraus kann man schließen das Parallelität in der Zukunft ein zentrales Thema spielen wird.

2 Parallelisierung

Die Idee hier ist, dass man die Ausführungszeit einer Aufgabe(eines Programms) dadurch reduziert, dass mehrere Berechnungsströme erzeugt werden, die auf verschiedenen Prozessoren eines Parallelrechners gleichzeitig zur Ausführung gelangen und durch koordinierte Zusammenarbeit die gewünschte Aufgabe erledigen. Das bedeutet dass eine Aufgabe in kleinere Teilaufgaben geteilt wird und diese Teilaufgaben werden von verschiedenen Prozessoren erledigt. Zur tatsächlichen

Abarbeitung werden diese Teilaufgaben auf physikalischen Prozessoren abgebildet das nennt man Mapping, dies kann entweder statisch oder dynamisch stattfinden.

Die Bearbeitung von Teilaufgaben ist möglicherweise nicht vollkommen unabhängig voneinander, sondern kann durch Daten und Kontrollabhängigkeiten miteinander gekoppelt sein. Der Grund dafür ist das manche Teilaufgaben nicht ausgeführt werden, bevor andere Teilaufgaben benötigte Daten oder Informationen für den nachfolgenden Kontrollfluss bereitgestellt haben. Deswegen wird für die korrekte Abarbeitung die Ausführung von Synchronisation- und Kommunikationsanweisungen notwendig. Durch diese Anweisungen warten alle beteiligten Prozessoren aufeinander um miteinander Informationen auszutauschen.

Ausführungszeit einer Aufgabe setzt sich aus der Rechenzeit jedes beteiligten Prozessors und die Zeit für den Informationsaustausch unter diese Prozessoren zusammen. Um eine optimale parallele Laufzeit zu bekommen, sollte eine möglichst gleichmäßige Lastverteilung auf die Prozessoren angestrebt werden, sodass ein Lastgleichgewicht entsteht. Was hier auch beachtet werden sollte ist, dass der Informationsfluss zwischen den Prozessoren so gering wie möglich gehalten werden sollte, damit die Prozessoren nicht lange aufeinander warten müssen. Der Entscheidungsvorgang in welcher Reihenfolge, welche Teilaufgabe, auf welchem Prozessor abgearbeitet werden, wird *scheduling* genannt.

2.1 Historie von Parallelität im Rechner

1972: Slotnick entwickelt Illiac IV (erster SIMD-Computer mit 64-Bit PEs in Gitter-Topologie)
1973: zur Programmierung des ILLIAC IV wird der FORTRAN Dialekt IV-TRAN mit parallelem DO-Konstrukt angeboten
1976: Cray Research installiert ersten Vektorrechner Cray-1 mit einer Leistung von 100 Flop/s ??
1982: Fujitsu installiertVP-200 Vektorrechner mit 500 MFlop/s
1985: Thinking Machines stellt Connection Machine namens CM-1 vor (SIMD-Computer mit 64k 1-Bit PEs)
1986: erster SMP: Sequent Balance 8000, 8 CPUs 1987: die datenparallelen Programmiersprachen LISP* und C* werden für die Connection Machine verfügbar
1988: Intel stellt iPSC/2 vor (MIMD-Rechner mit bis zu 128 in einem Hyperkubus angeordneten Intel 386-Prozessoren)
1991: Univ. von Tennessee präsentiert PVM (Parallel VirtualMachine) zur Programmierung heterogener paralleler Systeme
1993: HPF 1.0 (High Performance Fortran) wird standardisiert
1993: Cray baut MIMD-Rechner Cray T3D (bis zu 2048 DEC Alpha-Prozessoren verbunden in 3DTorus-Topologie)
1994: IBM SP2: Kopplung vieler RISCSysteme/6000 Workstations über ein schnelles, skalierbares Netzwerk MPI 1.0 wird neuer Standard einer Kommunikationsbibliothek für netzwerkgekoppelte parallele Systeme Cray präsentiert SHMEM als Bibliothek zur Programmierung speichergekoppelter paralleler Systeme

1995: DEC Alpha 21164 Prozessor mit 4-facher Superskalarität
1996: SGI Origin 2000 (erster Parallelrechner mit virtuellem gemeinsamem Speicher)
1997: OpenMP wird neuer Standard für eine direktiven-basierte parallele Programmierung speichergekoppelter Systeme neuer Standard MPI 2.0 ergänzt MPI um einige Eigenschaften, z.B. dynamische Generierung paralleler Prozesse
1997: ASCI Red mit 4536 Pentium Pro CPUs erreicht eine Leistung von mehr als 1 TFlop/s
2003: ASCI Q aus 4096 Alpha CPUs erreicht mehr als 10 TFlop/s
2005: IBM BlueGene/L mit 65536 PowerPC CPUs erreicht eine Leistung von mehr als 100 TFlop/s

3 Ebenen der Parallelität

Die Bedeutung der Parallelität ist , dass zwei Befehle(Programme ,Aktivitäten usw.) gleichzeitig verarbeitet(ausgeführt) werden. Das Hauptziel ist, dass die gegebene Aufgabe schnell und sicher erledigt werden, deswegen geschieht die Parallelität im Rechner in verschiedenen Ebenen. Manche von denen sind dem Benutzer oder dem Programmierer nicht einmal bekannt. Die Klassifizierung dieser Ebenen ist nicht einheitlich. Deshalb werde ich versuchen, dies mit verschieden Sichten zu verdeutlichen.

3.1 Sicht des Programmierers

Bei der Programmierung eines Programms wird die Parallelität entweder durch einen Programmierer oder durch den Compiler gesteuert. Bei dem Expliziten Programmierungsmodell soll ein Programmierer Parallelität an manchen Stellen selber programmieren. Zum Beispiel kann er bei dem Datenaustausch die Kommunikation oder die Synchronisation selber bestimmen. Bei dem impliziten Programmierungsmodell der Compiler bestimmt die Parallelität. Das geschieht direktivengesteuert oder automatisch, in Schleifenebene oder Anweisungsebene. Im Kapitel parallele Programmierungsmodelle werden wir nochmals auf die implizite und explizite Programmierung eingehen.

3.2 Sicht des Systems

Eine echte parallele Ausführung kann natürlich nur auf einem Rechner, der mit mehreren Prozessoren ausgestattet ist oder auf mehreren miteinander vernetzten Rechnern realisiert werden. Ansonsten kann die Nebenläufigkeit von Programmen nur simuliert werden, indem der Prozessor während der Laufzeit zwischen diesen wechselt. Für den Benutzer entsteht der Eindruck einer gleichzeitigen Ausführung. Man spricht von Quasiparallelität. Das Betriebsystem steuert mehrere Ebenen von Parallelität in Quasi-oder Echteparallelität, Zum Beispiel können in der Programmebene verschiedene unabhängige Programme gleichzeitig laufen oder in Prozessebene(Taskebene). Diese Prozesse

können sogar miteinander kooperieren, was auch von Programmierer explizit programmiert werden sollte. Ein Paar Begriffe sind hier wichtig: Ein Prozess ist eine (Software-) Einheit, die durch das Betriebssystem auf dem realen Prozessor zur Ausführung gebracht werden kann. Ein Prozess umfasst dabei seinen Programmcode (inkl. Daten) und einen Kontext, in dem der Code ausgeführt wird(`http://www.stud.informatik.uni-goettingen.de/os/ws2007/folien/prozesse1.pdf`). Mehrere gleichzeitig laufende Prozessen müssen auf physikalische Prozessoren abgebildet werden, was man Mapping nennt, wenn es mehr Prozesse als Prozessoren gibt, dann müssen Prozessoren mehrere Aufgaben gleichzeitig erledigen. Die Aufgabe vom Betriebsystem ist den verschiedenen Prozessen die Prozessorzeit zuzuteilen(scheduling). Hier fängt die Täuschung des Systems an. Jeder Prozess hat einen eigenen Adressraum, was sein Spielraum ist. Ein Task ist die kleinste ausführbare Einheit der Parallelität, die in verschiedenen Ebenen auftauchen können. Es gibt auch leichtgewichtige Prozessen, die man Thread nennt. Thread ist ein Ausführungsfaden innerhalb eines Prozess. Ein Prozess kann aus einer Thread oder mehrere Threads bestehen. Threads haben auch einen eigenen Adressraum vom Adressraum des Prozesses. Threads können explizit oder implizit programmiert werden. Mit der Threads kann die Parallelität in Blockebene dargestellt werden, das wird durch den Compiler automatisch erzeugt. Als Beispiel kann man die Schleife geben.

3.3 Sicht des Prozessors

In diesem Kapitel wird die Parallelität innerhalb des Prozessors behandelt, also in der Anweisungsebene. Das Ziel ist mehrere unabhängige Befehle gleichzeitig auszuführen. Als Beispiel kann eine Pipeline gegeben werden. Hier werden verschiedene Befehle parallel ausgeführt. Ein anderer Beispiel sind die Superscaler und Vliw Prozessoren bei denen mehrere Befehle in einem Wort zusammengeführt und dann ausgeführt werden, damit wird durch die Zusammenführung Parallelität erreicht. Schudelung wird durch Hardware oder Compiler ausgeführt.

4 Architekturen

Es gibt verschiedene Wege zur Klassifizierung der parallelen Rechnern. Eine der weit verbreiteten Klassifikation, die seit 1966 im Einsatz ist, heißt Flynn's Taxonomie. Diese Klassifizierung können wir nochmal nach Speicherorganisation klassifizieren. Als erstes werden wir die Speicherorganisationen anschauen und dann die flynische Klassifikation.

4.1 Klassifikation nach Art der Speicheranordnung

4.1.1 Gemeinsamme Speicher (shared memory) Ein Rechner mit mehreren Prozessoren, die zusammen auf einen globalen Speicher zugreifen nennt man Shared Memory Maschiene(als SMM bezeichnet). Alle Prozessoren sind gleichberechtigt. Sie können unabhängig voneinander auf dem selben Speicher

Daten schreiben oder lesen. Sie können sich auch denken, das ganze Adressräume ihm gehören. Der Speicher besteht aus kleinen Speichermodulen, die gemeinsam ein Adressraum darstellen. Die Prozessoren können kooperieren durch gemeinsame Variablen, dieser Art der Kommunikation ist schneller als bei den Distributed Memory Machine(DMM) Systeme. Bei dem Zugriff auf gemeinsamen Variablen ohne Kooperation können Kollisionen ausbrechen. SMM Maschinen können nach unterschiedlichen Modelle auf den Speicher zugreifen. **UMA:** Unified Memory Access Modell : alle Prozessoren greifen in gleicher Weise auf den gemeinsamen Speicher zu, insbesondere die Zugriffzeit ist bei allen gleich. Jeder Prozessor kann ein eigenes Cache haben. Ein Vertreter dieses Modells sind die symmetrischen Multiprozessoren (SMP) . Alle Prozessoren können auf die gleiche Weise auf den Betriebsmittel zugreifen und gleichberechtigte Systemdienste ausführen. NUMA:Non Uniform Memory Access:alle Prozessoren können auf den gemeinsamen Speicher zugreifen. Die Zugriffszeiten variieren aber, je nach dem Ort, an dem sich der adressierte Speicherblock physisch befindet. Die Speichermodule sind auf die Knoten verteilt. Der Zugriff auf ein lokales Speichermodell ist schneller. Für die Benutzer ist praktisch kein Unterschied zu SMP. Typische Vertreter der Klasse:Distributed Shared Memory-Systeme (DSM) Vorteile: *Ist für Programmierer einfacher und freuendlicher. *Ist schneller als verteilte Speicher. Nachteile: *Da alle Prozessoren gleiche Verbindungsnetzwerke benutzen, könnte es bei großerer Anzahl von Prozessoren schwerer realisierbar sein. * Wenn Prozessoren auf gleiche Module zugreifen sollen, dann muss der Programmierer Synchronisation selber programmieren *Systeme sind teuer.

4.1.2 Verteilte Speicher(distributed memory) Rechner dieser Katagorie haben mehrere Prozessoren, die keinen gemeinsamen Speicher besitzen, sondern eigene lokale bzw. private Speicher haben. Solche Systeme nennt man Distributed Memory Maschine (DMM). Private Speicher können nur von dem Prozessor benutzt werden, dem er gehört. Ein Prozessor kann normalerweise nicht auf einen Speicher zugreifen, der ihm nicht gehört. Wenn aber irgendwie ein Prozessor Daten von einem solchen Speicher braucht, kann er das durch Kommunikation machen. Bei diesen Systemen erfolgt die Kommunikation durch Nachrichtenaustausch. Deswegen brauchen die DMM ein Verbindungsnetzwerk, über die Daten übertragen werden können. Wann und wie die Kommunikation(Synchronisation) aufgebaut wird, ist meistens die Aufgabe des Programmierers. Vorteile: Da jeder Prozessor einen eigenen Speicher hat, können sie sehr schnell arbeiten. Aber bei der Zusammenarbeit ist sehr viel Kommunikation nötig. Dann sind DMM langsamer als SMM. Der Speicher ist skalierbar mit der Anzahl der Prozessoren. Wenn die Anzahl der Prozessoren und die Größe des Speichers erhöht wird, steigt der Proportionalität. Nachteile: Der Programmierer ist verantwortlich für viele der Einzelheiten im Zusammenhang mit den Daten der Kommunikation zwischen Prozessoren. Es könnte schwierig sein, bestehende Datenstrukturen, die für gemeinsamen Speicher gedacht sind, auf dem DMM abzubilden.

4.1.3 Hyprite Speicher (Hybrid Distributed-Shared Memory) Diese Art von Rechnern ist die Mischung aus einem verteiltem und gemeinsamen Speicher. Es sind Systeme, die aus mehrere SMP bestehen. Die Prozessoren aus denselben SMP's haben einen gemeinsamen(globalen) Speicher. Der verteilte Speicher-Komponente ist die Vernetzung von mehreren SMP. Die SMP's haben nur Informationen über ihren eigenen Speicher - nicht über den Speicher von einem anderen SMP. Deshalb ist eine Netzwerk-Kommunikation erforderlich, damit Daten von einem SMP zu einem anderen übertragen werden.

4.2 Klassifikation von Rechnerarchitekturen nach Flynn

Flynn's Taxonomie unterscheidet Multi-Prozessor-Computer-Architekturen in vier mögliche Klassen.

Unterscheidungskriterien:

* wieviele Befehlsströme bearbeitet ein Rechner zu einem gegebenen Zeitpunkt (einen, mehrere)?
* wieviele Datenströme bearbeitet ein Rechner zu einem gegebenen Zeitpunkt (einen, mehrere)?

4.2.1 SISD: Single Instruction stream, Single Data stream Rechner dieser Klasse sind ganz normale also Einprozessorrechner. Einen Datenstrom wird von einem Prozessor in einer bestimmten Zeit abgearbeitet. Hier kann man Quasiparallelität realisieren.

4.2.2 MISD: Multiple Instruction streams, Single Data stream Hier geht es darum, dass mehrere Prozessoren auf einem Datenstrom gleichzeitig arbeiten. Das ist nicht sinnvoll. Das könnte man so beschreiben. Es gibt eine Arbeit, die nur für eine Person bestimmt ist, wird aber von mehreren Personen erledigt, obwohl es gar nicht nötig wäre. Die folgenden Klassen sind für den Parallelrechner sehr wichtig.

4.2.3 SIMD: Single Instruction stream, Multiple Data streams Ein SIMD-Rechner hat einen Instruktionstrom aber mehrere Recheneinheiten (Prozessoren), die dieser Anweisungsstrom parallel bearbeiten könnten. Die Prozessoren können einen eigenen lokalen Speicher besitzen, also sie könnten unabhängig von anderen Verarbeitungseinheiten auf den privaten Speicher zugreifen. Wird aber eine Information von einem anderen Speicher benötigt, dann wird es durch Verbindungskanäle, Daten und Signale ausgetauscht. Solche sind wie wir vorher gesehen haben Distributed-Memory-Rechner.

Die vorbereitete Ausführung eines SIMD- Rechner hat einen gemeinsamen Speicher. Dieser Rechner wird dann als Shared- Memory- Rechner bezeichnet. Da jeder Prozessor überall auf einem Speicher schreiben oder lesen darf, muss das irgendwie Hardwaremäßig oder Softwaremäßig organisiert werden, um eventuelle Zugriffskonflikte zu vermeiden. Rechner dieser Art in den Siebzigern und

Achtsieger wurden auch Vectorrechner genannt. Heute werden dieser Systeme mit Workstations oder PCs die mehrere Prozessoren besitzen realisiert.

4.2.4 MIMD: Multiple Instruction streams, Multiple Data streams

Rechner dieser Klasse besitzen im Vergleich zu SIMD eine allgemeinere Struktur und arbeiten stets asynchron. Jeder Prozessor hat einen eigenen Anweisungsstrom und verarbeitet unabhängig von anderen parallel. Diese Systeme können entweder einen gemeinsamen Speicher oder einen verteilten Speicher besitzen. MIMD Rechner mit einem gemeinsamen Speicher nennt man auch eng gekoppelt. Datenaustausch erfolg über Speicherbereiche, auf die verschiene Prozessoren koordiniert zugreifen können. Systeme ohne gemeinsamen Speicher sind lose gekoppelt und die Kommunikation wird über den Verbindungsnetzwerk realisiert. Deswegen ist es die aufwändigere Art als die Andere. Einzelne Computer oder Workstations, die in einem Netzwerk verschaltet sind, gehören auch zu dem MIMD Rechner Klasse.

5 Paralleles Programmiermodell

Wenn man von paralleler Programmierung spricht, dann tauchen manche technische Begriffe auf. Das sond die Wörter Partitionierung(Daten für parallele Bearbeitung vorbereiten), Mapping, Kommunikation und Synchronisation. Alle diese Begriffe müssen nach ihrer Bedeutungen, während der Programmierung organisiert werden. Diese Organisation kann automatisch von Compiler, aber auch teilweise durch Programmierer oder ganz durch Programmierer erfolgen.

5.1 Explizite Parallelität

In diesem Modell kümmert sich der Programmierer um die Parallelität. Er übernimmt die Zuordnung der Tasks an die Prozessoren, sowie Kommunikation und Synchronisation. Die Aufteilung der Aufgaben sollte explizit angegeben werden. Ein Beispiel dieser Klasse ist das BSP- Modell (Bulk Synchronous Parallelism Model). Dieses Modell kann auch in zwei Gruppen aufgeteilt werden. Explizite Parallelität mit implizierter Zerlegung. Hier gibt der Programmierer an, wo im Programm Potential für eine parallele Bearbeitung vorliegt, etwa eine parallele Schleife, die Kodierung in Threads oder Prozesse, aber nicht vornehmen muss. Bei der expliziten Zerlegung muss sich der Programmierer über alle Einzelheiten der Parallelisierung kümmern Zerlegung, Verteilung, Kommunikation und Synchronisation. MPI ist der Vertreter dieser Modelle.

5.2 Implizite Parallelität

Hier geschieht die Parallelität automatisch. Der Programmierer programmiert einfach sequentiel, die nötige Aufteilung wird von Compiler übernimmt. Programmierung mit diesem Modell ist am einfachsten zu verstehen, da man keine

explizite Darstellung der Parallelität im Programm braucht. Es gibt zwei Arten davon: **Parallelisierende Compiler:** Unter Compilerparallelisierung versteht man die Zerlegung von Programmen in klassischen Sprachen wie C und FORTRAN in Teilaufgaben, die so unabhängig voneinander sind, dass sie gleichzeitig ausgeführt werden können. Deswegen ist diese Technik auch praktisch die einzige Wahl, wenn man vor der Aufgabe steht, ein bereits existierendes sequentielles Programm zu parallelisieren.

5.3 MPI

In diesem Modell läuft die Kommunikation der Prozessoren über den Nachrichtenaustausch und das sollte vom Entwickler selbst programmiert werden . Zwei oder mehrere Prozessoren können als Sender und Empfänger miteinander kommunizieren. Dabei sollte zwischen beiden eine Kooperation vorliegen z.B. eine Sendeoperation sollte ein entsprechend ein Empfängeroperation haben. Ein Sender kann eine Nachricht an einen oder mehreren Empfänger schicken bzw. mehrere Sender können eine Nachricht an eine oder mehrere Empfänger schicken. Der Nachrichtenaustausch kann sowohl innerhalb eines Rechners mit mehreren Prozessoren als auch innerhalb eines Netzes mit mehreren Rechnern erfolgen. Die Prozessoren sollen aufeinander warten, wenn sie voneinander abhängige Prozesse ausführen. Deswegen es ist sehr wichtig, dass man immer minimierende Nachrichtenaustausch vorzieht, während der Aufgabenverteilung. Die Prozesse, die viel Kommunikation brauchen müssen zu demselben Prozessor zugeteilt werden. Um die Message Passing unabhängig von Hardware und Betriebsystem realisieren zu können, wurden verschiedene Standarte durchgeführt: Message Passing Interface und Parallel Virtual Machiene

5.4 Shared Memory

Bei diesem Modell kommunizieren Sender und Empfänger miteinander über einen gemeinsamen Adressraum. Der Sender speichert die Werte in einem Modull und der Empfänger liest es dort ab. Das sieht man besonders bei Threads Programming: Alle Threads haben zugriff auf alle Daten. Dieses Modell ist für die Programmierer einfach, da es sich nicht explizit um die Kommunikation zwischen verschiedene Prozessen kümmern soll.

5.5 Hypride Systeme und andere Modelle

hier geht es um die Mischung von verschiedenen Modelle: Z.B. MPI und openMP

6 Grundkonzepte der parallelen Programmierung

6.1 Syncronisation

Bei gemeinsamen Speicher- Modellen gibt es kritische Abschnitte, wo verschiedene Prozesse gemeinsame Variablen haben. Diese Prozesse dürfen nicht gleichzeitig die kritische Bereiche antreten und die Anweisungen bearbeiten. Das ist nicht

zugelassen, um eventuelle Probleme zu vermeiden. Wenn sich zwei oder mehrere Prozesse auf eine Resource in kritischen Abschnitten interessieren, müssen sie sich gegenseitig ausschließen damit sie sich gegenseitig nicht stören. Das nennt man Synchronisation. Bei der Synchronisation sollen die Prozesse abwechselnd auf die kritische Variable zugreifen. In der gleiche Zeit kann sich höchstens ein Prozess im kritischen Bereich befinden.

6.2 kominikation

Wenn verschiedene Prozesse bei Bearbeitung eines Problems zusammenarbeiten, dann müssen sie auch kommunizieren können. Besonders bei abhängigen Aufgaben. Kommunikation zwischen zwei parallelen Prozessen wird im Programmiermodell Shared Memory genannt durch den Zugriffe auf gemeinsame, globale Variablen erfolgen. Bei den verteilte Systeme wird es über Verbindungsnetz durch Nachrichtenaustausch realisiert.

7 Ausblick

Die Leistungserhöhung von Rechnern mit einem Prozessor ist schwer zu realisieren. Die Hauptprobleme sind Stromverbrauch, Aufwärmung und langsame Speicherzugriffe. Deswegen kann man sagen, dass die Zukunft den Multicoresystemen gehört. Bis jetzt hatten wir auf dem Markt Multicoresysteme mit zwei oder vier Kernen. Es wird immer mehr Kerne auf einem zusammengebracht. Laut einem Intelsprecher werden ab Jahresende 2009 multicoresysteme mit 12 und 16 Kernen auf dem Markt sein. Die Entwickler von Programmierungssprachen werden neue Schnittstellen(Bibliotheken) entwickeln um mit multicoresysteme gut umzugehen. Jetzt ist die Frage wie soll ein Programmierer erfolgreich parallel programmieren? Er soll zuerst parallel denken, also nicht einfach sequentieler Programme parallel programmieren, sondern wissen um worum es wirklich bei Parallelismus geht. Er soll eine Strategie haben, das Konzept schrittweise einführen. Zweitens er soll nicht die Anzahl der vorhandenen Prozessoren zu programmieren, sondern er soll sich aufgabenbezogen, an Teile des Parallelismus und an Programmierung-Umgebung denken. Die entwickelte Programme sollten skalierbar sein damit in die Zukunft auch angewandt wird[1].

Literatur

[1] J. Reinders. *Gehört Multi-Core-Prozessoren die Zukunft.*

Konzepte und Mechanismen zur Synchronisierung

Veysel Sercan Aksu

Universität Augsburg
aksu_veysel@hotmail.de

1 Einleitung

Seit längerer Zeit beschäftigt man sich mit der Aufgabe, die Rechnerleistung immer wieder zu steigern. Der Einsatz von paralleler Programmierung bietet hier die Möglichkeit, eine höhere Leistung zu nutzen als sequenziell bearbeitende Rechner, was auch wichtig ist um mit den neu entwickelten Multiprozessoren zu handeln. Dabei muss man beachten, dass parallel laufende Prozesse[1] mit einer Koordination arbeiten müssen, weil sonst unerwünschte Probleme entstehen können.

Um diese unerwünschten Probleme zu vermeiden, wird zuerst im ersten Kapitel die Probleme motiviert, demnächst findet man im zweiten Kapitel einige grundlegende Begriffe, die im Thema Synchronisation behilflich sein können und einige Lösungen dieser Probleme. Im dritten Kapitel werden die wichtigen Synchronisationsmechanismen vorgestellt, wie Semaphor,Monitor,Barriere und Locks. Am Ende findet man einen kurzen Ausblick und eine Zusammenfassung.

Motivation
Prozesse sind voneinander abhängig, wenn sie in einer bestimmten Reihenfolge ausgeführt werden müssen. Bei einer gegebenen Gesamtaufgabe übernehmen die Prozesse jeweils bestimmte Teilaufgaben. Bei einem sequenziellen Rechner werden die Prozesse von einem Prozessor übernommen und sequenziell ausgeführt, bei Multiprozessoren können sie dahingegen aber auch gleichzeitig ausgeführt werden.

Bei der parallelen Ausführung von Prozessen können allgemein 2 Probleme auftreten.

1. Zwei Prozesse wollen Daten austauschen.(*Kommunikation*) Beim Erzeuger-/Verbraucher-System nimmt ein Prozess(*Erzeuger*) Daten auf, was der andere(*Verbraucher*) erzeugt hat. Beim Auftraggeber-Auftragnehmer-System nimmt ein Prozess nicht nur Daten auf, sondern liefert sie später wieder zurück. Es ist eine Art von Kooperation und es kann noch andere Formen von Kooperation definiert werden.
   Nehmen wir an, dass zwei Prozesse Daten über einen gemeinsamen Pufferbereich austauschen. Ein Prozess erzeugt Daten und legt sie im Puffer ab, der andere Prozess liest Daten aus dem Puffer und verarbeitet sie weiter.

[1] Definition [10] :Ein Prozess ist der definierte Ablauf von Zuständen eines Systems.

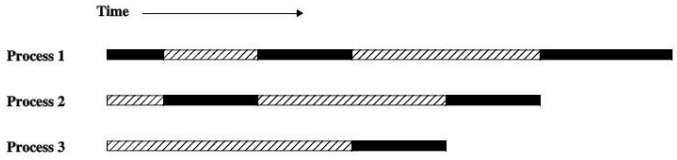

Abbildung 1. Ein Prozessor mit mehreren Prozessen.[16]

Abbildung 2. Multiprozessor mit mehreren Prozessen.[16]

Abbildung 3. Erzeuger-/Verbraucher Problem : [2]

- Falls der Erzeuger aber schneller ist als der Verbraucher :
  Der Erzeuger kann die Daten überschreiben, bevor sie vom Verbraucher verarbeitet werden.
- Falls der Verbraucher schneller ist als der Erzeuger :
  Die gleichen Daten werden vom Verbraucher mehrfach gelesen.

Diese Arten von Probleme können durch Sychronisationsmechanismen gelöst werden. Im folgenden Kapitel wird anhand eines Beispiels gezeigt, wie das Erzeuger-/Verbraucher Problem gelöst werden kann.

2. Falls ein gemeinsamer Speicher vorhanden ist, muss der gleichzeitige Zugriff mehrerer Prozesse auf den gleichen Datenbereich koordiniert bzw. verhindert werden. Also müssen Speicher und Lese Zugriffe sequentialisiert werden. Beispiel: Kontostandsänderung

Prozess A

```
x=readAccount(account);
x=x+500;
writeAccount(x,account);
```

Prozess B

```
y=readAccount(account);
y=y-200;
writeAccount(y,account);
```

Abbildung 4. BeispielCode [10]

Während Prozess A den Kontostand erhöht, erniedrigt Prozess B den Kontostand und abhängig von der Ausführungsreihenfolge kann sich der Kontostand varieren. Aus diesem Grund hat man kein deterministisches Ergebnis. In so einem Fall können die Prozesse Konkurrenz machen, dass heißt, ein Ablauf eines Prozesses kann den Ablauf eines anderen verändern oder aber auch verhindern.

Im dritten Kapitel findet man auch einige Lock und Synhronisationsmechanismen, damit man den gleichzeitigen Zugriff auf den gleichen Datenbereich koordinieren kann.

2 Grundlagen der Synchronisation

In diesem Kapitel wird hier der Begriff *Synchronisation* vorgestellt.Es ist eines der wichtigsten Teilbereiche der parallelen Programmierung. Hier werden zuerst der Begriff Synchronisation definiert und einige grundlegende und wichtige Begriffe erklärt.

2.1 Definition von Synchronisation

Unter *Synchronisation*[2] versteht man die Koordination der Kooperation und Konkurrenz zwischen Prozessen. Eine Synchronisationsoperation bringt die Aktivitäten verschiedener Prozesse in eine Reihenfolge, indem er Prozesse für die anderen warten lässt, oder erlaubt den Prozessen, die auf andere warten, weiterzulaufen. Synchronisation verhindert also die gleichzeitige Ausführung von Prozessen, indem er andere Prozesse warten lässt. Ein Prozess wartet auf den anderen und dieses Verhalten von Prozessen ist ähnlich wie der sequenziell bearbeitende Prozessoren. Wegen dieses Verhaltens verzögert der Beginn der Durchführung von anderen Prozessen.

Prozesse können voneinander wissen, wenn ihre Koordination durch Kommunikation(Datenaustausch) ergänzt ist. Kommunikation und Synchronisation arbeiten Hand in Hand. Zum Abstimmen zweier Aktivitäten können Prozesse Informationen austauschen.

Es gibt 2 Arten von Synchronisation.

1. ***Einseitige Synchronisation***:
   Wie bei dem Erzeuger-Verbraucher Problem muss der Erzeuger zuerst die Daten erzeugen, damit der Verbraucher die Daten abarbeiten kann. Der erste Prozess bietet die Voraussetzung für den zweiten Prozess.

[2] Definition aus dem Buch von Hertwich-Hommel [8] und von Theo Ungerer [19].

Beispiel : Entladung von Waren.

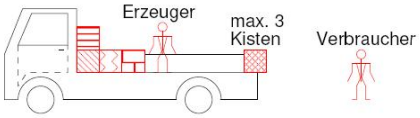

Abbildung 5. Erzeuger Verbraucher.[6]

Der Erzeuger und Verbraucher müssen deswegen synchronisiert werden. Falls auf der Ladezone keine Ware steht, muss der Verbraucher warten. Falls auf der Ladezone schon 3 Kisten sind , muss der Erzeuger auf den Verbraucher warten. Somit ist ihre Durchführung beschränkt.

2. ***Mehrseitige Synchronisation***:
Bei der mehrseitige Synchronisation haben zwei oder mehrere Prozesse gleiche Zugriffsrechte. Falls ein Prozess mit seiner Durchführung angefangen hat, müssen die anderen Prozesse warten, bis dieser Prozess zum Abschluss kommt.Das kann mit dem folgenden Beispiel[3] besser veranschaulicht werden.

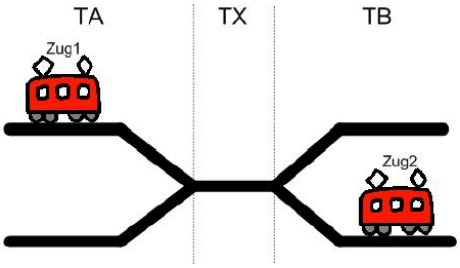

Abbildung 6. Mehrseitige Synchronisation

Zwei Züge haben eine gemeinsame Bahnstrecke TX.Falls Zug1 zuerst die Strecke TX erreicht hat, muss Zug2 warten bis er diese Strecke verlässt, und falls Zug2 zuerst die Strecke TX erreicht hat, muss diesmal Zug1 auf Zug2 warten, damit kein Unfall passiert.

2.2 Grundlegende Begriffe

Bevor hier Synchronisationsmechanismen erklärt werden, werden zuerst einige grundlegende Begriffe klar gemacht, die dabei behilflich sein können, das The-

[3] Bild ist aus der Vorlesung Informatik II(SS07) Übungsaufgabe von Übungsblatt 3.

ma besser zu verstehen. Es gibt verschiedene Kontrollprobleme für das Nutzen von gemeinsamen Ressourcen, die berücksichtigt werden müssen.Dazu zählen wechselseitige Ausschluß, deadlock, livelock und starvation.

Typisch für eine mehrseitige Synchronisation ist der *wechselseitige Ausschluß* (Mutual Exclusion). Man spricht von einem *wechselseitigem Ausschluß*[4], wenn zu jeden Zeitpunkt nie mehr als ein Prozess auf ein Objekt zugreifen kann. Ein Beispiel dafür ist die eingeleisige Bahnstrecke.

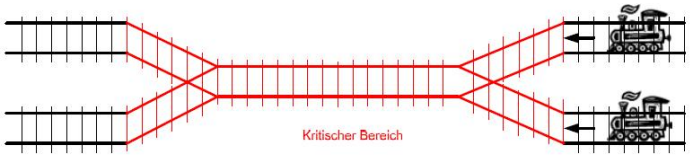

Abbildung 7. Wechselseitiger Ausschluß und kritische Bereiche [10]

Wenn ein Zug den so genannten kritischen Bereich erreicht, darf kein anderer Zug auf diesen Bereich eintreten. Abschnitte, in denen mehrere Prozesse um gemeinsame Ressourcen konkurrieren, bezeichnet man als *kritische Abschnitte* oder *kritische Bereiche*.

Durch Synchronisierung werden parallel laufende Prozesse verzögert und bleiben in einem verzögerten Zustand. Dabei können Verklemmungssituationen entstehen, bei denen Prozesse gegenseitig sich aufeinander warten und selbst nicht in der Lage sind, jemals aus diesem verzögerten Zustand herauszukommen. Diese Situation nennt man ***Deadlock***[5].

Es gibt vier verschiedene Bedingungen[6], die Deadlock beschreiben und wegen knappen Betriebsmitteln entstehen können.

1. Die beteiligten Prozesse wollen alleinigen (exklusiven) Zugriff auf Betriebsmittel erhalten(wechselseitiger Ausschluß)
2. Die Betriebsmittel, die von den Prozessen bereits belegt werden, können nicht kurzfistig entzogen werden.
3. Prozesse besitzen bereits Betriebsmittel, während sie auf den Zugriff auf andere Betriebsmittel warten.
4. Es findet sich eine geschlossene Kette von Prozessen, die Betriebsmittel besitzen und gleichzeitig auf Betriebsmittel warten, die die jeweiligen Vorgänger in dieser Kette besitzen.

Verklemmungsprobleme kann man auf drei verschieden Arten vermeiden.

– Man kann versuchen, die Verklemmung zu vermeiden, indem man darauf achtet, dass immer mindestens eine der obigen vier Bedingungen nicht erfüllt ist

[4] Definition: wechselseitiger Ausschluß [20].
[5] Die Definition von Deadlock [1] und [19]
[6] Diese Bedingungen : [1]

– Man kann versuchen, die Verklemmung zu vermeiden, indem man die zukünftigen Betriebsmittelanforderungen der Prozesse analysiert und Zustände verbietet, die zu Verklemmungen führen können.
– Man kann versuchen, eine Verklemmung festzustellen und sollte sie eingetreten sein, dann zu beseitigen.

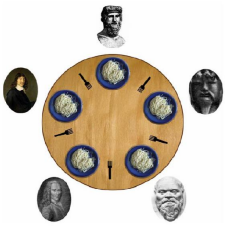

Abbildung 8. Dining Philosophers

Ein klassisches Problem aus der Informatik auch bekannt Dining Philosophers[7]. Fünf Philosophen sitzen an einem Tisch. Vor ihnen stehen jeweils ein Teller mit Essen und auf der linken und rechten Seite Gabeln. Sie benötigen zum Essen zwei Gabeln (Betriebsmittel).

Sie diskutieren, wenn ein Philosoph hungrig ist, nimmt er zuerst die linke Gabel und danach die rechte Gabel wenn sie frei sind. Falls die rechte Gabel nicht frei ist, wartet er ohne die linke Gabel zurückzulegen, bis die rechte frei wird. Nach dem Essen legt er die Gabeln wieder zurück. Das Problem ist, wenn alle Philosphen gleichzeitig hungrig sind. Alle nehmen zuerst die linke Gabel in die Hand und warten dann gleichzeitig auf die rechte Gabel, weil keine mehr verfügbar ist.

Livelock[8] : zwei oder mehrere Threads sind zwar nicht blockiert, sind aber nicht mehr lebendig, etwa weil sie immer wieder wechselseitig aufeinander reagieren und keinen Berechnungsfortschritt mehr machen. Ein einfaches Beispiel ist die Situation, dass 2 Personen durch einen engen Korridor gegenseitig laufen.Wenn sie aufeinander treffen, müssen sie ausweichen, aber beide wählen immer gleichzeitig die selbe Richtung und sie kommen nicht mehr weiter.

Starvation[9] : ein Thread kommt nicht mehr zur Ausführung, etwa weil ein anderer Prozess eine Ressource dauernd belegt. Beispiel wie bei Dining-Philosophers.

[7] Bild ist aus der Internetseite `http://www.computerbase.de/lexikon/Philosophenproblem`
[8] Definition von Livelock : [7]
[9] Definition von Starvation : [7]

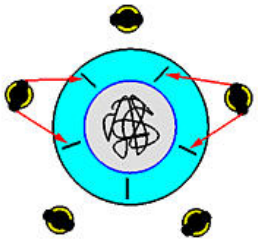

Abbildung 9. Starvation [18]

Nehmen wir an, die auf dem Abbildung gezeichneten Personen essen und reden sehr schnell.Weil sie so schnell essen greifen sie sofort auf Gabeln zu und essen. Nachdem sie gegessen haben, holen sie aber schnell wieder ihre Gabel, bevor die anderen drei sie holen. In dem Fall sitzen die anderen drei Personen am Tisch und warten sehr lange und sie haben keine Chance mehr etwas zu essen. Diese Situation nennt man Starvation.

2.3 Race Conditions

Für die Prozessoren ist es wichtig, so schnell wie möglich ihre Zugriffsoperationen auszuführen, und dabei achten sie nicht darauf, mit welcher Reihenfolge die Zugriffe(Schreibe-/Lese) erfolgen. Wenn der Zugriff bei parallelen Programmen auf die gemeinsamen Variablen nicht richtig synchronisiert wird(also in Konkurenz stehende Prozesse), kann ein zeitlich abhängiger Fehler verursacht werden. ***Race Condition***(Wettrennen). Man sagt die Prozessoren liefern sich ein Wettrennen. Egal wie die Zugrifffolge ist, muss das Programm determinierte Ergebnisse liefern.Es soll unabhängig von dem Wettrennen sein. Hier ermöglicht die Synchronisation die Vermeidung der Fehler, die durch RaceConditions entstehen können.

Es gibt zwei wesentliche Arten von Races.*General Races* und *Data Races*, von denen Data Races öfters vorkommt.

Data Races tritt auf, wenn mehrere Prozesse auf eine gemeinsame Adresse zugreifen wollen, aber nur eine davon hat den Schreibzugriffsrecht auf die gemeinsame Adresse.Wenn zwei Prozesse in Konflikt stehen, so dass weder Prozess eins vor Prozess zwei zugreift noch Prozess zwei vor Prozess eins, dann hat das Programm ein Data Race. Data Race Problem ist meistens unberechenbar und könnte auch in hunderten von Ablaufen von Programmen passieren und ist somit auch schwer mit Debug Modus herauszufinden. Data Race Problem kann falsche Werte im kritischen Bereiche produzieren und verwirrt den Programmierer über den Programmablauf.

Das General Race Problem verursacht nicht deterministische Ausführung und sind Fehler in Programmen, die deterministisch wirken. Dieses Problem ist schwerer herauszufinden als Data Races. Obwohl die kritischen Bereiche gesichert sind und Data Races nicht auftreten können, können General Race Pro-

bleme trotzdem auftreten. Bis jetzt ist aber kein konkreter Unterschied zwischen den beiden Problemen bekannt. In einem Article [10]aus Sun Microsystems wurde gesagt, dass Data Races eine einfache Art von General Races sind, bzw. Race Conditions, die keine Data Races sind, sind General Races .

2.4 Hardware- und Software-Lösungen

Das Synchronisationsproblem kann auf zwei Arten, Hardware- und Software-Lösungen gelöst werden.

2.4.1 Softwarelösungen : Petterson und Silberschatz[11] beschreiben eine Lösung des Synchronisationsproblems für Systeme mit gemeinsamen Speicher, die ohne spezielle Hardware auskommt. Das Programm besteht dabei aus zwei parallelen Prozessen, die auf einen gemeinsamen Datenbereich zugreifen wollen. Die beiden Prozesse P1 und P2 werden im Hauptprogramm nacheinander gestartet und parallel ausgeführt.

```
...
start(P1);
start(P2);
...
```

1. **Versuch** [12]: Die Reihenfolge der Zugriffe von P1 und P2 sind festgelegt.

   var turn: 1..2
   Initialisierung: turn = 1

P1	P2

Loop
while (turn!=1) do (nichts) end;
<kritischer Abschnitt>
turn =2;
<sonstige Anweisungen>
end
	Loop
	While turn !=2 do (nichts) end;
	<kritischer Abschnitt>
	turn =2;
	<sonstige Anweisungen>
	end

   In dieser Lösung ist die Reihenfolge der Zugriffe festgelegt und ist auch gesichert, dass nur ein Prozess auf den kritischen Bereich zugreifen darf. Die Prozesse warten aufeinander in einer busy-wait Schleife. Am Anfang ist der turn Variable mit 1 initialisiert und somit fängt zuerst P1 mit der

[10] Article aus der Internetseite von Sun Microsystems siehe Quelle [4]
[11] Petterson und Silberschatz : [3]
[12] Programmbeispiele und Versuche : [3]

Ausführung an und P2 wartet in der busy-wait Schleife, solange, bis P1 nach seiner Ausführung das turn Variable auf 2 setzt. Danach fängt P2 mit seiner Ausführung an. Es ist aber keine Gute Idee, da wir ein Prozess nicht mehrmals hintereinander ausführen können und deswegen ihre Ausführung beschränkt ist.

2. **Versuch**: var flag: array [1..2] of BOOLEAN
Initialisierung flag[1]= false; flag[2] = false;

P1	P2

Loop
while (flag[2]) do (nichts) end;
flag[1] = true;
<kritischer Abschnitt>
flag[1] =false;
<sonstige Anweisungen>
end
	Loop
	While flag[1] do (nichts) end;
	flag[2] = true;
	<kritischer Abschnitt>
	Flag[2]=false;
	<sonstige Anweisungen>
	end

Bei dem zweiten Versuch wurde das Betreten des kritischen Abschnittes nicht gesichert.Wenn beide flags mit false initialisiert ist, ist es möglich, dass beide Prozesse gleichzeitig ihre while-Schleife verlassen und in den kritischen Bereich eintreten und das muss in dem nächsten Versuch verhindert werden.

3. **Versuch**: Da der zweite Versuch nicht ausreichend war, wird es hier modifiziert.
var flag : array[1..2] of BOOLEAN;
Initialisierung: flag[1] = false; flag [2] = false;

P1	P2

Loop
flag[1] = true;
while (flag[2]) do (nichts) end;
<kritischer Abschnitt>
flag[1] =false;
<sonstige Anweisungen>
end
	Loop
	flag[2] = true;
	While flag[1] do (nichts) end;
	<kritischer Abschnitt>
	flag[2]=false;
	<sonstige Anweisungen>
	end

Weil jetzt der Wert des flags vor der while-Schleife gesetzt wurde, ist das gleichzeitige Eintreten von beiden Prozessen in den kritischen Abschnitt verhindert.Nun ist das Problem aber, dass beide Prozesse gleichzeitig ihre flags

auf true setzen und in der while-Schleife warten. Somit geht keiner von beiden Prozessen aus der Schleife raus und entsteht eine Verklemmung. ***Livelock***

4. **Versuch**:Beide Ideen werden zusammengefasst.
   var turn:1..2; flag: array[1..2] of BOOLEAN
   Initialisierung: turn :=1;(beliebig) flag[1] = false; flag[2] = false;

P1	P2

Loop
flag[1] = true;
turn=2;
while (flag[2] && turn=2) do (nichts)
end;
<kritischer Abschnitt>
flag[1] =false;
<sonstige Anweisungen>
end
	Loop
	flag[2] = true;
	turn =1;
	While (flag[1] && turn =1) do
	(nichts) end;
	<kritischer Abschnitt>
	flag[2]=false;
	<sonstige Anweisungen>
	end

Nach der letzten Verbesserung tritt keine Verklemmung auf und der gleichzeitige Zugriff auf den kritischen Abschnitt wurde verhindert.
Ausserdem existieren noch andere Algorithmen, die bekannt als Dekker oder von Eisenberg und McGuire. Noch effizientere Lösungen werden im Kapitel drei beschrieben.

2.4.2 Hardwarelösungen : Obwohl das Synchronisationsproblem mit Software gelöst werden kann, verwenden die meisten Rechner eine Hardwarelösung. Bei einem einfachen Ein-Prozessor-System kann dies durch das Sperren aller Interrupts geschehen. Da nun kein Zeit-Interrupt mehr möglich ist, kann kein Prozesswechsel stattfinden und der ausführende Prozess kann ohne ein Hindernis den kritischen Abschnitt ausführen. Besser geeignet ist es, die so genannte *Test und Set* Operation zu verwenden.Diese Operation besteht aus zwei Teiloperationen:

1. Lesen eines booleschen Variablenwertes und
 	
2. Überschreiben dieses Variablenwertes mit „true“
	Procedure test_and_set
	(var lock:BOOLEAN):BOOLEAN;
	Var mem:BOOLEAN;
	beginn(*unteilbar*)
	mem = lock;
	lock = true;
	return(mem)
	end;(*unteilbar*)

Abbildung 10. Beispiel [3]

Diese beiden Teiloperationen müssen direkt hintereinander als unteilbare Operation ausgeführt werden, so dass kein anderer Prozess die Möglichkeit hat dazwischen einen Zugriff zu machen. Es geschieht dadurch, dass ein Prozessor, der Test-and-Set ausführt, den Speicherbus sperrt um einen Zugriff von anderen Prozessoren zu verhindern. Unter Verwendung einer Test-and-Set-Hardwarelösung kann das Synchronisationsproblem für beliebig viele Prozesse einfacher gelöst werden. Das Zurücksetzen des Variablenwertes auf false kann als einfache Zuweisung durchgeführt werden und benötigt keine spezielle Hardwareunterstützung.

var lock: BOOLEAN;
Initialisierung:lock = false;
Prozess i

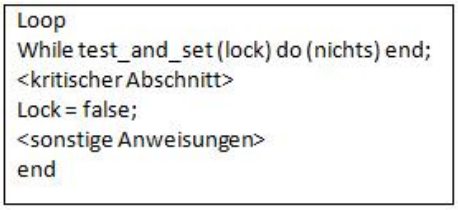

Abbildung 11. Beispiel TestundSet [3]

Diese zwei Lösungsarten des Synchronisationsproblems ist wegen der *busy-wait* Schleifen sehr ineffizient. Im folgenden Kapitel werden effizientere Synchronisationsmechanismen betrachtet.

3 Synchronisationsmechanismen

Im obigen Kapiteln wurden bis jetzt mögliche Synchronisationsprobleme, einige grundlegende Begriffe und einige ineffiziente Synchronisationslösungen bekanntgemacht. In diesem Abschnitt werden mehr die effiziente Synchronisationsmechanismen betrachtet.Bevor mit diesem Thema angefangen wird, werden hier die Unterschiede zwischen Synchronisation in gemeinsamen und verteiltem Adressräumen erklärt. Unterteilung der Synchronisationsmechanismen können unter diesen zwei Begriffen erfolgen.

Bei einem gemeinsamen Adressraum geschieht der Zugriff über gemeinsame Variablen, die für Informationsaustausch nützlich sind. Diese einfache Methode von Informationsaustausch (Komminikation) geschieht meistens durch Threads bezeichneten Berechnungsströme. Mit Hilfe von Threads werden die konkurrierende Datenzugriffe koordiniert.

Bei einem verteilten Adressraum werden dahingegen die Daten vom Parallelprogramm in privaten Adressbereichen abgelegt und nur ein bestimmter Prozess hat darauf Zugriffsrechte. Die Kommunikation erfolgt über explizite Anweisungen, mit denen ein Prozess die Daten seines privaten Adressbereiches an einen anderen Prozess senden kann. Um diese Prozesse besser zu koordinieren, eignet sich die Synchronisationsform *Barrieren*, die später detailliert beschrieben werden.

3.1 Wechselseitiger Ausschluß(Mutual Exclusion)

In Kapitel 2 wurde der Begriff des wechselseitigen Auschlusses definiert. Kurzgefasst darf auf einer Variable im kritischen Bereich nur ein Prozess zugreifen, damit man einen korrekten Ergebnis haben kann. Wechselseitiger Ausschluss sollte folgende Anforderungen[13] haben:

1. Nur ein Prozess darf zu einem Zeitpunkt in den kritischen Bereich kommen, um auf die gemeinsame Variable oder auf die gemeinsamen Betriebsmittel zu zugreifen.Um diesen Zugriff zu kontrollieren wird meistens diese Programmstruktur genutzt.
   ....
   entercritical();
   /*criticalsection*/
   exitcritical();
   ...
   Beide operationen gehören im System zusammen, und sie garantieren den Zugriff auf gemeinsame Ressourcen.
2. Wenn kein anderer Prozess zu einem Zeitpunkt im kritischen Abschnitt ist und ein Prozess ein Zugang zu diesem Bereich haben will, wird der Zugang ohne Verzögerung erlaubt.
3. Ein Prozess darf im kritischen Bereich für eine endliche Zeit bleiben.
4. Es sollte keine Annahmen über die Geschwindigkeiten eines Prozesses oder die Anzahl der Prozessoren geben. Oben genannte Methoden sollten genug sein, um die Kontrollprobleme zu lösen.

3.1.1 Semaphor : Ein ***Semaphor***[14] ist eine nicht-negative Zählvariable, auf die alle Zugriffe als unteilbare Operationen ausgeführt werden. Semaphore werden häufig dazu verwendet, den Zugriff auf Ressourcen zu koordinieren. Semaphore wurde von *Dijkstra* eingeführt und als Hilfsmittel zur Synchronisation bezeichnet.

Ein Semaphor besteht aus zwei Operationen, die mit P(s) *passieren* und V(s) *verlassen* bezeichnet werden. Sie können für die Sicherung kritischer Abschnitte aber auch allgemein für das Belegen und Freigeben von Betriebsmitteln verwendet werden, und sind ein sehr effizientes Synchronisationsverfahren.

[13] Diese Anforderungen sind von der Quelle [16]
[14] Definition von Semaphor : [19]

Implementierung von P(s) und V(s):

```
void P(Semaphor S) {
  S.zähler--;
  if (S.zähler < 0) {
    block(S.liste);
    assign();   // nächsten laufbereiten Thread
                // auf den Prozessor holen
  }
}

void V(Semaphor S) {
  if (S.zähler < 0) {
    ready(S.liste);
  }
  S.zähler++;
}
```

Abbildung 12. Beispiel Code von P(s) und V(s) [5]

Beim Aufruf der Funktion P(s) wird der Wert der Variable dekrementiert und der Prozess kann mit der Ausführung fortfahren.Dann wird geprüft, ob der Semaphore-Variable kleiner als null wird.Falls er kleiner als null wird, wird der Zugriff auf den kritischen Bereich verweigert. Beim Aufruf der Funktion V(s) wird der Wert der Variable s um eins inkrementiert und damit sind die Ressourcen wieder freigegeben. Ein beliebiger wartender Prozess wird dann erweckt. Die wartenden Prozesse können in einer Warteschlange bewahrt werden, und sie können mit dem FIFO-Prinzip bearbeitet werden, damit sie dann in fairer Reihenfolge abgearbeitet werden.

Semaphore können in zwei Gruppen[15] unterteilt werden. Semaphore die nur die Werte Null und Eins haben, werden als binäre Semaphore bezeichnet. Wenn Semaphore beliebige nichtnegative, ganzzahlige Werte annehmen können, heißen sie allgemeine Semaphore.

* **Erzeuger-Verbraucher Problemlösung mit Semaphore** *

Es wurde im vorherigen Kapitel das Problem Erzeuger-Verbraucher bekannt gemacht. Dieses Problem kann mit Hilfe von Semaphore gelöst werden.

[15] Definitionen und Unterteilung von Gruppen : [19]

```
int       buffer[N];                    // Puffer
int       in=0, out=0;                  // Pufferzeiger
semaphore   used=0, free=N, mutex=1;

void Erzeuger() {                       // n Erzeuger
    int item_w;
    for (;;) { produce(&item_w);
          P(free);   P(mutex);          // noch Platz im Puffer? // Eintritt in krit. Abschnitt
          buffer[in]=item_w;  in=(in+1) % N;
          V(mutex); V(used);            // ein weiterer Eintrag belegt
}  }
void Verbraucher(){                     // n Verbraucher
    int item_r;
    for (;;) {
          P(used); P(mutex);            // Daten im Puffer? // Eintritt in krit. Abschnitt
          item_r=buffer[out]; out=(out+1)%N;
          V(mutex); V(free);            // ein Eintrag wird frei
          consume(item_r);
}  }
```

Abbildung 13. Beispielcode für Erzeuger-Verbraucher Problem [17].

Das Erzeuger-Verbraucherproblem wird im obigen Beispiel mit drei Semaphoren gelöst und diese Art von Erzeuger-Verbraucher Problem wird auch als Bounded-Buffer Problem genannt.

1. Der erste Semaphor , der *mutex* heißt, wird für die Sicherung der kritischen Abschnitte benutzt und ist ein binärer Semaphor.
2. Der zweite Semaphore *free* ist ein allgemeiner Semaphor und zählt die freien Plätze im Puffer herunter und falls ein Prozess im vollen Puffer schreiben will, wird er blockiert bis es wieder einen freien Platz gibt.
3. Der letzte Semaphor *used* zählt die belegten Plätze im Puffer herauf und blockiert den Prozess, der von dem leeren Puffer lesen will, und ist auch ein allgemeiner Semaphor.

Somit kann man mit Hilfe von Semaphoren das Erzeuger-Verbraucherproblem effizient lösen. Es könnte aber abei der Benutzung von Semaphoren einige Probleme auftreten.

- Aus dem kritischen Bereich austreten, ohne den mutex-Semaphor wieder freizugeben
- In den kritischen Bereich eintreten, ohne den mutex-Semaphor zu setzen.
- Vertauschungen von Semaphoren zum Schutz von kritischen Abschnitten und Semaphoren, die vorhandene Betriebsmittel zählen.

Eine bessere Lösung bieten Monitore an.

3.1.2 Monitoren : Ein weiterer Synchronisationsmechanismus sind die so genannten Monitore: Sie wurden von Hoare und Brinch Hansen als ein weiteres Synchronisationskonzept eingeführt.

Ein ***Monitor***[16] basiert auf die Idee der Objektorientierung, ist ein Software-Modul, bestehend aus

- einer Anzahl von Monitorvariablen.
- einer Anzahl von Prozeduren oder Funktionen
- einem Monitorkörper, der dazu dient, Initialisierung der Monitorvariablen direkt nach dem Programmstart.

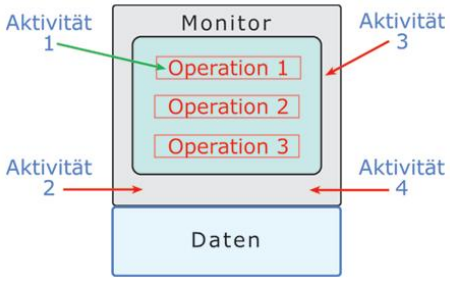

Abbildung 14. veranschaulicht Struktur eines Monitors [11]

Der Vorteil eines Monitors ist, dass beliebig viele Aktivitäten(Prozesse) Monitoroperationen gleichzeitig aufrufen können, ohne dass man Synchronisationsoperationen wie bei der Semaphore einsetzen muss. Ein vergessener P(s) oder V(s) Operation kann sich im ganzen System schlecht auswirken. Ein Monitor sorgt selbst dafür, dass in seinen Operationen zu einem bestimmten Zeitpunkt immer nur ein Prozess aktiv ist und alle anderen blockiert werden.Um das zu erhalten, benötigt man innerhalb eines Monitors Synchronisationsoperationen. Da das Zählen durch Integer Variablen im geschützten Bereich des Monitors möglich ist, muss man sich nur darum Sorgen machen, dass Prozesse blockiert und aufgeweckt wird.Das geschieht mit Hilfe von zwei Operationen : *wait(c)* und *signal(c)* arbeiten mit einer Bedingungsvariable. c ist hier die Bedingungsvariable.

- wait (c): Der aufrufende Prozess wartet auf die Bedingung c, bis er eintritt und ist in einer internen Warteschlange von Prozessen eingeordnet, danach wird der Monitor sofort wieder freigegeben.
- signal(c): Falls die Warteschlange von c nicht leer ist, dann wird der Prozess am Kopf der internen Warteschlange von c erweckt.

[16] Definition von Monitor [19]

```
monitor PrintMonitor;
var     printerBusy: Boolean;
        printerFree: Condition;

   procedure ReservePrinter;
   begin    if printerBusy then wait(printerFree);
            printerBusy := true
   end { ReservePrinter };

   procedure ReleasePrinter;
   begin    printerBusy := false;
            signal( printerFree );
   end { ReleasePrinter };

begin  printerBusy := false
end { PrintMonitor }.
```

Abbildung 15. Codebeispiel [17]

Im obigen Beispiel zu sehen sind die Monitorvariablen, eine Bedingungsvariable ***printerFree*** und die boole'sche Variable ***printerBusy***. Es gibt ausserdem noch zwei Prozeduren ReservePrinter und ReleasePrinter. Im Prozedur ReservePrinter wartet der aufrufende Prozess solange, bis die Bedingungsvariable(printerFree) auf frei gesetzt wird und er wird in einer internen Warteschlange gesetzt. Danach wird der Monitor wieder freigegeben.In der Prozedur ReleasePrinter wird das Signal zum Wecken des Prozesses an dem Warteschlangekopf gegeben.Unten am Monitorkörper wird die Variable printerBusy initialisiert.

3.2 Barrieren :

Eine ***Barriere***[17] ist ein Synchronisationspunkt für mehrere Prozesse. Bei der Barrieren-Synchronisation warten alle Prozesse, welche die Barrieren-Synchronisation aufrufen, so lange, bis auch der letzte einer Gruppe von Prozessen an der Barriere angekommen ist.

Für die Barrieren-Synchronisation wird eine Variable von Typ Barriere angelegt.Die Anzahl der Prozesse, die an Barrieren-Synchronisation teilnehmen, müssen vor der Ausführung initialisiert werden, damit, wenn ein Prozess oder Prozesse auf eine Barriere erreicht hat, muss dieser solange warten, bis alle anderen auch die Barriere erreicht haben. Nachdem alle Prozesse die Barriere erreicht haben, muss die Barriere wieder auf den Initialwert zurückgesetzt werden. Mit der Beispielimplementierung einer Barriere unten wird es veranschaulicht.

[17] Definition von Barriere [19]

Listing 2.1. Ein Codebeispiel für Bariere [9]

```
class Barrier
{ private int processes         //Die Anzahl von Prozessen , die
                                 synchronisiert werden
              arrived =0;        //Die Anzahl von Prozessen , die Barriere
                                 erreicht haben.

    public Barrier(int procs)
       { processes = procs;   }
    synchronized public barrier()
       {arrived++;
        if (arrived < processes)
                try { wait();} catch(Interrupted Exception e) {}
        else
            { arrived =0;        //Variablenwert zurücksetzen
              notifyAll();       // alle wartende Prozesse arbeiten weiter

} } }
```

Für die Prozesse, die zusammen mit den gleichen Berechnungsschritten Daten austauschen, sind Barrieren eine Möglichkeit, um alle Prozesse zusammen abzuarbeiten. Es gibt noch andere Implementierungstechniken der Barrieren-Synchronisation. Einige davon sind *verteilte Barriere als Baum* , *Symmetrische, verteilte Barriere (Dissemination)*.Weitere Informationen zu diesem Thema kann im Internet[18] erreicht werden.

3.3 Locks

Eine andere Art von Synchronisierung von Prozessen sind die so genannten ***Locks***. Sie werden für das Aufsperren und Freigeben von Prozessen zu den kritischen Abschnitten benutzt. Zu jedem Zeitpunkt darf nur ein Prozess zum Aufsperren erwerben und darf nur ein Prozess im kritischen Abschnitt sein. Ins deutsche übersetzt heißen sie Schlossvariable.

Wenn ein Prozess den kritischen Abschnitt betreten will, wartet dieser so lange, bis das Schloss offen ist. Nachdem das Schloss geöffnet worden ist, tritt er in den kritischen Abschnitt ein, und schließt das Schloss ab, damit kein anderer Prozess auf den kritischen Abschitt zugreifen kann. Diese Operation nennt man lock[19]. Nachdem der Prozess mit seiner Ausführung im kritischen Abschnitt fertig ist, öffnet er den Schloss wieder und gibt diesen für die anderen Prozesse, die darauf warten, wieder frei.Diese Operation nennt man unlock. Bei der Implementierung von lock und unlock Operationen muss man beachten, dass sie nicht unterbrochen werden können.

[18] Emfehlende Internetseite im Literaturverzeichnis [19]
[19] Definition aus der Quelle [19]

```
/* Funktion zum Zuschließen */
lock (boolean Besetzt){
    while (Besetzt = TRUE){
    /* Nichts */
    }
    Besetzt = TRUE;
}
```

```
/* Funkt. zum Aufschließen */
unlock (boolean Besetzt){
    Besetzt = FALSE;
}
```

Abbildung 16. Beispielimpelementierung von Lock und Unlock [15].

Verwendung von Lock und Unlock Operationen in einem Rechenprozess :

```
lockvar s;                  /* s ist die Schloßvariable (Datentyp lockvar) */
rechenprozess_RP1() {
    . . .
    lock (s);               /* Verschließen des kritischen Abschnittes */
    . . .                   /* Anweisungen des kritischen Abschnittes */
    unlock (s);             /* Aufschließen des kritischen Abschnittes */
    . . .
}
```

Abbildung 17. Verwendung von Lock und Unlock [15].

Locks, die mittels eines solchen aktiven Wartens implementiert sind, werden auch als ***Spin locks***[20] bezeichnet. Wegen des aktiven Wartens ist der Prozessor und der Systembus unnötig und stark belastet. Die beiden Operationen lock und unlock müssen atomar[21] sein.Atomarität kann mit Hilfe von Interrupts oder mit Hardwareunterstützung, die im Kapitel zwei bei den Hardwarelösungen Test-and-Set behandelt werden, geschehen.

Mit Hilfe von Interrupts können die Operationen atomar gemacht werden. *Disable Interrupt* und *Enable Interrupt* helfen, dass die Operationen ungestört ausgeführt werden.

[20] Definition von Spinlock : [19]
[21] Definition :eine Funktion, die keinen Prozesswechsel zulässt : [15]

```
DISABLE_INTERRUPTS();
LOCK(lock);
...
  // critical
...
UNLOCK(lock);
ENABLE_INTERRUPTS();
```

Abbildung 18. Verwendung von Interrupts [13].

Aber Verwendung von Interrupts hat einige Nachteile[22] :

1. Sie ist nur für Einprozessorsysteme geignet.
2. Kritische Bereiche, die mit Interrupts behandelt werden, sollten möglichst kurz sein, damit auch die Unterbrechungen möglichst mit kurzen Verzögerungen erfolgen können.
3. Die beiden Befehle *Disable Interrupt* und *Enable Interrupt* sollten zusammen benutzt werden.

Eine andere Lösung bietet sich die Hardware. Die Nachteile von Interrupts können hier als Vorteil erscheinen, da sich Test-and-Set auch für Mehrprozessor Systeme geeignet sind. Die Verwendung von Test-and-Set wurde bereits im Kapitel zwei unter Hardwarelösungen erklärt.

Eine Mutex-Implementierung wird als *Suspend lock*[23] bezeichnet, wenn das Betriebssystem die wartenden Prozesse oder Threads verwaltet und erst dann wieder einen der wartenden Prozesse oder Threads aktiviert, wenn der kritische Bereich freigegeben wird.

Ein anderes Lockmechanismus sind die so genannten Read-/Write Locks, mit denen der Zugriff auf ein Objekt geschützt werden können. RW-Locks ermöglichen mehrere gleichzeitige Lesezugriffe auf einem Objekt, oder es kann nur einen Schreibzugriff geben. Mit einem Beispiel kann die Verwendung von RW-Locks besser veranschaulicht werden und in dem Beispiel wird auch als Synchronisation im Java behandelt.

Listing 2.2. Ein Codebeispiel [12]

```
1  public class IntStack {
2   private final int[] array;
    private volatile int cnt = 0;
4   public IntStack (int sz) { array = new int[sz]; }

6   synchronized public void push (int elm) {
      if (cnt < array.length) array[cnt++] = elm;
8     else throw new IndexOutOfBoundsException();
    }
10
    synchronized public int pop () {
12    if (cnt > 0) return(array[--cnt]);
      else throw new IndexOutOfBoundsException();
```

[22] Nachteile : [14]

[23] Definition von Suspendlock :[19]

```
14  }

16  synchronized public int peek() {
      if (cnt > 0) return(array[cnt-1]);
18    else throw new IndexOutOfBoundsException();
    }
20
    public int size() { return cnt; }
22
    public int capacity() { return (array.length); }
24  }
```

Im Java können die Funktionen *synchronized* deklariert werden und der gleichzeitige Zugriff wurde verhindert. Hier im diesem Beispiel wird peek() synchronized deklariert und verhindert, dass es sowohl mit push() als auch mit pop() gleichzeitig ausgeführt wird. Klar ist es notwendig, dass die Variable cnt und array nicht gleichzeitig geschrieben und gelesen werden. Aber damit ist auch verhindert, dass auf peek() mehrmals gleichzeitig zugegriffen wird. Hier hilft die RWLocks, dass es möglich gehalten wird. Unten kann man die Version mit RW-Locks sehen.

Listing 2.3. Ein Codebeispiel [12]

```
1     public class IntStackWithRWLock {
2   private volatile int cnt = 0;
    private final ReadWriteLock myLocks = new ReentrantReadWriteLock();
4   public IntStack (int sz) { array = new int[sz]; }

6   public void push (int elm) {
      myLocks.writeLock().lock();
8     try {
        if (cnt < array.length) array[cnt++] = elm;
10      else throw new IndexOutOfBoundsException();
      } finally {
12      myLocks.writeLock().unlock();
      }
14  }

16  public int pop () {
      myLock.writeLock().lock();
18    try {
      if (cnt > 0) return(array[--cnt]);
20    else throw new IndexOutOfBoundsException();
      } finally {
22      myLocks.writeLock().unlock();
      }
24  }

26  public int peek() {
      myLock.readLock().lock();
28    try {
      if (cnt > 0) return(array[cnt-1]);
30    else throw new IndexOutOfBoundsException();
      } finally {
32      myLocks.writeLock().unlock();
      }
34  }

36  public int size() { return cnt; }

38  public int capacity() { return (array.length); }
    }
```

Hier werden die Funktionen push() und pop() mit writelock().lock absolut gesichert, dass man gleichzeitig auf cnt und array zugreifen kann. Mit readlock() dahingegen ist es möglich, dass man mehrmals gleichzeitig auf peek() zugreift.Dabei kann nichts geschrieben werden, als etwas vom Stack gelesen wird. Damit ist das Problem der ersten Version abgeschafft und man kann gleichzeitig mehrmals vom Stack lesen.

Es gibt noch viele verschiedene Arten von Locks. Hier wurden kurz einige davon behandelt.

4 Zusammenfassung und Ausblick

Synchronisierung spielt bei der Entwicklung von parallelen Programmen eine wichtige Rolle. Diese Parallelität von Programmen müssen eine Koordination haben, damit keine unerwünschte Probleme entstehen. Datenaustausch zwischen Prozessen und Prozessoren und gemeinsamen Ressourcen bringen einige Probleme mit sich. Durch ungeschicktes Verteilen von Prozessen auf Prozessoren können erhebliche Effizienzverluste auftreten, die man gerade beim Einsatz eines Parallelrechners natürlich vermeiden möchte. Ohne richtige Koordination kann sich die Programmierer leicht verwirren. Diese Koordination geschieht mit wichtigen Synchronisationsmechanismen. Diese Seminararbeit gibt einem Überblick über die wichtigen Grundlagen und Konzepte der Synchronisierung. Es wurde grundlegende Begriffe, sowie der wechselseitige Ausschluss, einige Verklemmungsituationen, die bei der parallelen Programmierung entstehen können, sowie das Verhalten von Prozessoren erklärt. Es stellt ausserdem effizientere Sychronisationsmechanismen, wie Semaphor, Monitor, Barriere und Locks vor, die eine sichere Ausführung von Programmen ermöglichen, und der gemeinsamen Ressourcen auf Prozessen und Prozessoren richtig aufgeteilt wird.

Mit den neu entwickelten Multiprozessoren ist der Bedarf an parallele Programmierung gestiegen. Die Effizienz bei sequenziell bearbeitenden Rechner hat langsam seine Grenzen erreicht. Man benötigt sowohl jetzt als auch in der Zukunft mit der Neuigkeiten in der Technologie in der Prozessorentwicklung immer mehr Kentnisse im parallelen Programmierung, um die Rechnerleistung zu erhöhen. Aber die Leistung ohne ein sicheres System ist einfach nicht denkbar. Synchronisation wird auch in der Zukunft seine Wichtigkeit schützen und als wichtiges Bestandteil der parallen Programmierung bleiben.

Literatur

[1] M. Ben-Ari. *Grundlagen der Parallelprogrammierung.* Hanser, 1985.
[2] H. Brosenne. Prozess synchronisation. *Betriebssysteme*, 2007/2008. `http://www.stud.informatik.uni-goettingen.de/os/ws2007/folien/synchronisation2-2x2.pdf`.
[3] T. Bräunl. *Parallele Programmierung :Eine Einführung.* vieWeg, 1993.
[4] L. T. Chen. The challenge of race conditions in parallel programming. *Sun Microsystems*, July 21, 2006. `http://developers.sun.com/solaris/articles/raceconditions.html`.

[5] P. C. Freiling. Betriebssysteme. *Lehrstuhl für Praktische Informatik 1 Universität Mannheim*, Herbstsemester 2007. http://pi1.informatik.uni-mannheim.de/filepool/teaching/bs-2007/BS_20071107.pdf.

[6] P. D.-I. habil. Armin Zimmermann. Pdv und robotik. *Informatik 4*, SS2007. http://pdv.cs.tu-berlin.de/Info4-SS2007/Tutorium4Handout-4.pdf.

[7] D. U. Hoffmann. Verteilte systeme. *FH Wedel*, 2007/2008. http://www.fh-wedel.de/uploads/media/VS_02_Threads_Prozesse.pdf.

[8] H. Hommel. *Nebenläufige Programme*. Springer-Lehrbuch, 1994.

[9] P. D. U. Kastens. Vorlesung parallele programmierung in java. *Vorlesung SS*, 2000. http://ag-kastens.upb.de/lehre/material/ppjava/folien/comment38-48.2.pdf.

[10] P. Knoll. Einführung in die informatik. *Lehrstuhl VI Robotics and Embedded Systems*, 2006. http://atknoll1.informatik.tu-muenchen.de:8080/tum6/lectures/courses/ss06/info2/.

[11] P. D.-I. B. J. Krämer. Synchronisation. *Fachbereich Elektrotechnik und Informationstechnik*, 2004. http://mlearning.dvt.fernuni-hagen.de/html_pda/ke/ke3/html/FuX_d78e1633.html.

[12] A. Langer. Java multithread support. *Explicit Locks in JDK 5.0*, 2008. http://www.angelikalanger.com/Articles/JavaSpektrum/14.ExplicitLocks/14.ExplicitLocks.html.

[13] P. K.-P. Löhr. Betriebssysteme. *Freie Universität Berlin, Fachbereich Mathematik und Informatik*, WS 2004. http://www.inf.fu-berlin.de/lehre/WS04/BS/folien/bs-3.4.pdf.

[14] P. D. C. Linnhoff-Popien. Einführung in die informatik iii. *LMU- Mobile and Distributed Systems Group*, Wintersemester 98/99. www.mobile.ifi.lmu.de/Vorlesungen/ws9899/info3/Skript/info3_k12.ps.gz.

[15] D.-I. F. D. M.Sc. Grundlagen verteilter automatisierungssysteme. *Universität Siegen*, Wintersemester 2004/05. http://www.bs.informatik.uni-siegen.de/www/lehre/ws0405/automat/ws0405_automat.pdf.

[16] J. Niu. Concurrency:mutual exclusion and synchronisation - part 1. *Operating System*, 2003. http://www.sci.brooklyn.cuny.edu/~jniu/teaching/csc33200/files/1007-Concurrency01.pdf.

[17] P. D. P. Schulthess. Betriebssysteme. *Vorlesung im Hauptstudium*, SS 2004. http://www-vs.informatik.uni-ulm.de/teach/ss04/bs/BS_SS04_Kap06_Synchronisierung.pdf.

[18] D. C.-K. Shene. Multithreaded programming with threadmentor. *Department of Computer Science Michigan Technological University*, 2001. http://www.cs.mtu.edu/~shene/NSF-3/e-Book/MUTEX/TM-example-philos-1.html.

[19] P. Ungerer. *Parallelrechner und parallele Programmierung*. Spektrum Akademischer Verlag, 1997.

[20] M. Wirsing. Nebenläufige programmierung. *Informatik II*, SS06. http://www.pst.informatik.uni-muenchen.de/lehre/SS06/infoII/folien/Folien15Threads1SS06.pdf.

Graphische Modellierung eines parallelen Systems

Tadas Sugintas

Universität Augsburg
tadas.sugintas@student.uni-augsburg.de

Zusammenfassung UML ist als Quasi-Standard für die Spezifizierung von komplexen Hardware- und Softwaresystemen etabliert. Jedoch unterstüzt sie nur sehr gering die Modellierung von parallelen Systemen. Dieser Mangel wird mit Hilfe von UML Profilen beheben. Die vorliegende Arbeit beschreibt, welche UML-Konstrukten für die Spezifizierung von parallelen Systemen benutzt werden können, gibt eine kurze Überblick über den bekanntesten UML-Profilen, die sich für die parallelen Systemen eignen. Weiter wird der Aufbau des cmUML Frameworks erläutert. Am Ende wird ein Beispiel gegeben, wie cmUML für die Modellierung eines Aufzuges eingesetzt werden kann.

1 Einleitung

Laut [7] „Die Unified Modelling Language (UML) ist heute die verbreiteste Notation, um Softwaresysteme zu analysieren und zu entwerfen."Die UML ist breit anwendbar, von Tools als auch von der Industrie unterstützt und gut standardisiert. Die UML dient zur Modellierung, Dokumentation, Visualisierung, Performance Modelling, Simulation komplexer Softwaresysteme, unabhängig von deren Fach- und Realisierungsgebiet. Jedoch ist die UML nicht perfekt und nicht vollständig. Die Sprache hat keine formale Semantik und die informale Beschreibung ist teilweise nicht eindeutig. Die Autoren von UML haben versucht die Sprache möglichst breit einsetzbar zu machen und an manchen Stellen semantische Lücken mit Absicht gelassen. Solche Lücken (*semantic variation points*) sollten von der Erweiterungsmechanismen der UML geschloßen werden.

Die Modellierung von parallelen Systemen ist ein der Gebieten, wo die UML große Mangeln hat. Generell sind alle UML-Abläufe sequenziell und solche wichtige Konzepte wie Shared Resources, Synchronization oder parallele Abläufe lassen sich nur mit den Konstrukten des niedrigen Grades (z.B. Semaphoren) modellieren. Um diesen Mangel zu beheben einige UML-Profile wurden von der Industrie sowie von den Akademikern vorgeschlagen, jedoch eignen sie sich mehr für Modellierung der real-time und eingebetteten Systemen und nur in geringer Sinne für Spezifizierung der parallelen Abläufe.

2 UML und die Profile für parallelen Systeme

Ein Merkmal von UML ist multi-view Sicht, wenn unterschiedliche Eigenschaften eines Systemes mit der Hilfe von unterschiedlichen Diagrammtypen darge-

stellt werden (z.B. Klassendiagramme, Aktivitätsdiagramme, Zustandsdiagramme). Einige Diagrammentypen der UML können direkt und ohne Anpassung für die Modellierung von parallelen Systemen benutzt werden, wie z.B. der Verteilungsdiagramm. In diesem Kapitel werden die von der UML bereitgestellten Konstrukten, die für die Modellierung von parallelen Systemen benutzt werden können, beschrieben.

2.1 Modellierung des parallelen Systems in UML

Um ein nebenläufiges System zu modellieren, drei Arten von Konstrukten sind nötig : aktive und passive Objekte, ein Mechanismus für den Nachrichtenaustausch und die Zustandsautomaten. Basierend auf [2] weiter wird ein Überblick von UML-Konstrukten gegeben, die für solche Modellierung benutzt werden können. Die grösten Probleme und Inkonsistenzen werden auch erläutert.

2.1.1 Aktive und passive Objekte Die Klassen in UML können entweder *active* oder *passive* sein. Das Attribut *isActive* zeigt, ob das Objekt einer Klasse sein eigenes Thread hat und parallel mit den anderen aktiven Objekten ausgeführt wird. Falls das Attribut auf den Wert *false* gesetzt wird, dann wird die Operation in dem Adressraum und unter der Kontrolle von einem anderen aktiven Objekt ausgefürt. Ein aktives UML Objekt hat nur einen Kontrollthread, also die aktive Objekte sind innerlich sequenziall. Jedoch die UML-Semantik sagt nichts über eine Situation, wenn mehrere nebenläufige Aufrufe auf denselben Aktiven Objekt ausgeführt werden. Da es nur ein Ausruf jeweils behandelt sein kann, wird ein Mechanismus für das Einreihen der Nachrichten benötigt. Es wird nicht klar, ob dieser Mechanism existiert und wie es funktioniert.

Falls ein Objekt *passive* ist, kann jede Operation entweder *sequenziall*, *guarded* oder *concurrent* sein. Das wird von dem Attribut *concurrency* bestimmt.

2.1.2 Nachrichten Die Kommunikation zwischen den Objekten in UML wird von den *messages* und *signals* unterstützt. Das Senden einer Nachricht wird im UML-Metamodel *CallAction* genannt. Ein *CallAction* kann einen Atribut *mode* haben, der auf *synchronous* oder auf *asynchronous* gesetzt werden kann.

Die UML gibt keine Einschränkungen für den Emfänger des asynchronen Aufrufes. Das kann problematisch sein, weil die passive Objekte ihre Methoden auf den aufgerufenen (called) Thread ausführen. Falls der angerufene (calling) Thread auf die Beendung der Operation nicht wartet, sondern weiterführt, bleibt für die aufgerufenen Operation keinen Thread für die Ausfürung übrig. Darüber hinaus, die asynchrone *request-reply* Kommunikation ist von der UML nicht unterstützt.

2.1.3 Zustandsautomaten UML definiert, dass falls ein Objekt einen Zustandsautomat hat, alle zu dem Objekt gesendeten Anfragen durch diesen Zustandsautomat fließen. Die Zustandsautomaten antworten auf die *SignalEvents*

und *CallEvents*. Die Anfragen werden in sequenziellen Ordnung bearbeitet, die nach der Einreihungsweise bestimmt wird. Aus der UML Semantik is es nicht klar, ob die Methoden eines Objektes von dem Zustandsautomat ausgefürt werden, oder ob der Zustandsautomat von den Methoden manipuliert wird.

2.1.4 Aktivitätsdiagramme Die UML bietet die Möglichkeit, ABläufe in mehreren Flüssen zu parallelisieren oder einen Fluss synchron zusammenzuführen. Dazu dienen

- Parallelisierungsknoten (*fork node*) und
- Synchronisationsknoten (*join node*)

Ein anderes Konstrukt, das für die Visualisierung von parallelen Prozessen nützlich sein kann, ist der Aktivitätsbereich (*swimline*). Jedoch beide Konstrukten haben nur sehr geringe Möglichkiten die Abläufe zu synchronisieren.

2.2 UML Profile für parallele Systeme

Die UML hat laut [7] zwei Möglichkeiten für die Anpassung an den speziellen Anforderungen des Benutzers:

- Leichtgewichtige Erweiterung durch Profile;
- Schwergewichtige Erweiterung durch Metamodellveränderung.

Die Metamodellveränderung ist das mächtigere Vorgehen und lässt die UML flexibel an den ungewönlichsten Projektbedingungen anpassen. Jedoch muß der Modellierer in das Metamodell tief gehende Kenntnisse besitzen. Die Aufgabe wird noch schwerer, falls solche Anpassungen im Rahmen eines Projekts stattfinden, denn sie sollen den anderen Projektbeteiligten kommuniziert werden. Auch die Anpassung von Modellierungstools muß berücksichtigt werden. „Daher wird man in der Praxis häufig festellen, dass der Aufwand für solche Eingriffe in das Metamodell den Nutzen übersteigt.“[7]

Im Gegensatz zur Erweiterung des Metamodells wird bei leichtgewichtige Erweiterung das Metamodell nur um bestimmte Elemente erweitert. Die bestehende Metaklassen werden erhalten und das Hinzufügen oder Entfernen von Metaklassen oder Assoziationen ist nicht erlaubt. [7] listet folgende Vorteile von Profilen auf:

- Viele UML-Basiskonzepte bleiben unangetastet und müssen nicht neu erlernt werden.
- Die Basis-UML-Notation kann genutzt werden.
- Besseren Möglichkeiten für die Wiederverwendung eines Profils.
- Das verwendete Modellierungstool muss keine Funktionalität zur Veränderung des Metamodells anbieten.
- Die Erstellung eines Profils ist einfacher als die Anpassung oder die Neuerstellung eines Metamodells.
- Modelle und Profile können per XMI einfach ausgetauscht werden.

– Profil-Anwendung ist einfacher als ein Modell-Merge.

Aus Modellierungssicht ist ein Profil ein Paket, das zusammengehörige Stereotype und Randbedingungen (*constraints*) gruppiert und auf andere Pakete angewendet werden kann. Es gibt in UML kein spezieller Diagrammtyp für Profile, die graphische Notation für Elemente werden in Paketdiagrammen als auch in Klassendiagrammen benutzt.

Weiter wird ein kurzer Überblick über die bekanntesten UML-Profilen, die sich für die Modellierung von parallelen Systemengut eignen, gegeben.

2.2.1 UML-RT UML-RT ist ein real-time UML-Profile, das von der Firma 'Rational Software' entwickelt wurde. Im Gegensatz zu den meisten UML-Profilen, UML-RT ist mehr als nur eine Menge von Stereotypen und Tags, sondern eine eigene Modellierungssprache [1]. UML-RT ermöglicht die Entwicklung von komplexen, ereignisgesteuerten, und möglicherweise verteilten Echtzeitsystemen.

Strukturmodellierung wird mit der Hilfe von den Kapseln (*Capsule*, die Erweiterung des UML-Elements "Class") ermoglicht. Die Kapseln sind komplexe aktive Objekte, die sich kommunizieren können. Durch mehrere Schnittstellen (*Ports*, die Erweiterung des UML-Elements "Class") werden die Nachrichten ins Umfeld gesendet und empfangen. Außerdem kann eine Kapsel andere Kapseln enthalten und so kann ein hierarchisches und komplexes Struktur gebildet werden. Eine Kapsel kann höchstens einem Zustandsautomat zugeordnet sein.

Die Kommunikation zwischen den *Ports* der Kapseln werden mit den Konnektoren (*Connector*, eine Erweiterung des UML-Elements "AssociationClass") modelliert.

Das Protokoll (Erweiterung des UML-Elements "Collaboration") spezifiziert eine verschloßsene Gruppe von Teilnehmern, die kommunizieren und die Aufgaben erledigen.

Das **Verhalten** eines Systems wird mit dem erweiterten Zustandsautomat modelliert. Wie die Kapseln, die Zustandsautomaten können auch hierarchisch sein. Der Empfang einer Nachricht löst eine Transition im Zustandsautomat aus.

UML-RT hat keine formalen Grundlagen.

2.2.2 UML Based Modeling of Performance Oriented Parallel and Distributed Applications In [6] wird ein UML Profil beschrieben, das für die Modellierung des parallelen Systems gut geeignet ist. Die Autoren konzentrieren sich auf eine Teilmenge von UML Diagrammen, nämlich Klassen-, Aktivitäts- und Kommunikationsdiagrammen. Mit Hilfe von Stereotypen und Tags werden die in der UML fehlende Konstrukte für *Distributed Memory*, *Message Passing*, *Work-Sharing*, *Parallel Region* und *Synchronisation* definiert.

2.2.3 UML/SPT *UML Profile for Schedulability, Performance and Time* ist ein Framework für die Modellierung von Servicequalität, Ressourcen, Zeit und Nebenläufigkeit. Der Modellierer kann eine Menge von Stereotypen und Tags

benutzen um das Model zu annotieren. Auf solchen Model kann die quantitative Analyse durchgeführt werden.

Das Profil ist in folgende Sub-Profile aufgeteilt:

- Das *General Resource Model* Paket ist der Kern von UML/SPT. Es wird weiter in folgende Pakete aufgeteilt:
  - *RTResourceModeling* für wesentliche Konzepte der Servicequalität.
  - *RTConcurrencyModeling* für nebenläufige Modellierung. Dieses Sub-Profil ist besonders interessant, denn hier werden solche abstrakte Konzepten wie *ConcurrentUnit*, *Scenario*, *ActionExecution* definiert, die von den Autoren von cmUML benutzt und erweitert wurden.
  - *RTtimeModeling* für Zeit und mit der Zeit verbundene Modellierung. Das Sub-Profil enthält die Stereotypen wie ≪RTtime≫, ≪RTclock≫, ≪RTtimer≫, ≪RTtimeout≫oder Rahmenbedingungen wie ≪RTdelay≫und ≪RTintervale≫, mit denen z.B. die Sensoren modelliert werden können.
- *Analysis Modeling Package* definiert die Sub-Profile für die Analyse, darunter:
  - *PAprofile* für die Leistungsanalyse.
  - *SAprofile* für die Zeitplannungsanalyse.

2.2.4 MARTE (Modeling and Analysis of Real-Time and Embedded Systems) ist der Nachfolger des UML/SPT Profils [5]. Dieses Profil ist während der Phasen von Spezifikation, Design und Überprüfung einsetzbar. Marte bietet folgende Nutzen an:

- Weiterverbreitete Methode für die Modellierung von real-time und eingebetteten Hardware- und Softwaresystemen, was die bessere Kommunikation zwischen den Entwicklern ermöglicht.
- Erlauben für das Zusammenspiel zwischen Modellierung- und Kodierungtools, die für die Spezifikation, Design, Überprüfung, Code-Generierung usw. benutzt werden können.
- Konstruktion von Modellen, die sich für Simulation und quantitative Prognose von den Hardware- und Softwaresystemen eignen.

3 cmUML

Die meisten UML-Profilen für parallelen Systemen basieren auf die Semantik eines Zustandsautomats und mangelt an der Beschreibung anderen Aspekten des parallelen Verhaltens wie Daten- und Kontrollfluss, Gleichzeitigkeit oder Synchronisierung. Zudem keiner der Profilen unterstützt die Konzepten des höheres Grades der Parallelität wie mehrfache Operationsaufruf oder präzise Synchronisationssemantik.

In [3] wird ein neuer Ansatz für Spezifizierung von parallelen Systemen eingeführt. Das cmUML Framework bietet architektonische Abstraktionen des

höheren Grades, die präzise operationale Semantik haben und die Aktionssemantik, aktiven/passiven Objekte und die Diagrammen des höheren Abstraktionsgrades integrieren. Das Framework wurde von der in [4] beschriebenen Transition Axiom Methode (TAM) inspiriert. Da die TAM abstrakt und von der Implementierung unabhängig ist, haben die Autoren der cmUML die wichtigste Konzepte in das Framework integriert, wie z.B. das Trennen von Lebendigkeits- und Sicherheitsbedingungen. Das Vorige beschreibt, welche Aktion mu's durchgeführt werden, wobei die Sicherheitsbedingungen beschreiben, welche Aktion nicht erlaubt ist. cmUML beschreibt eine Menge von Modulen (Komponenten genannt), die nach der Lamport's Empfehlung präzisen Schnittstellen und interne Spezifikation haben.

3.1 Aufbau der cmUML

cmUML erweitert die Konzepte des SPT-Profiles mit der präzisen Semantik und baut auf diesen Konzepten der UML-RT auf: Kapseln, Sub-Kapseln und Schnittstellen (Ports).

In cmUML ist ein *Component* eine allgemeine Einheit mit den spezifischen Funktionalitäten und Verhalten. Ein *Component* kann entweder *concurrent* oder *sequential* sein, was von dem internen Nebenlaäufigkeit abgestimmt wird. Die Nebenläufigkeit kann sich aus dem überlappenden oder abwechselnden Ablauf ergeben. Basierend auf die Funktionalität, die Komponenten werden weiter in *system*, *state*, *port*, *service* und *resource* eingeordnet.

Eine **System** Komponente enthält andere Komponenten und ist für ihre Initialisierung verantwortlich. Sie hat die Subkomponenten vom Typ 'port', 'state' und 'service'. Die Komponente 'port' definiert ihre Schnittstelle wobei die 'state' und 'service' Komponenten ihre interne Spezifikation beschreiben. Die 'port' und 'state' Komponenten sind statisch während die 'service' Komponenten dynamisch sind und auf die externe Anfragen antworten. Eine 'system' Komponente kann auch eine Komponente vom Typ 'resource' enthalten um die gesicherten und geteilten Ressourcen darzustellen.

Eine **Resource** Komponente stellt eine 'einfache' gesicherte und geteilte Ressource dar. Die Ressource kann z.B. die Daten oder Hardware sein. Die hat die Methoden 'acquire()', 'release()', 'read()' und 'write()'. Eine 'resource' Instanz wird explizit und atomar 'acquired' und 'released'. Die Ressourcen mit dem komplexen internen Verhalten können als 'system' Komponenten spezifiziert werden.

Ein **Service** ist eine dynamische Komponente, die nach dem Empfang einer externen Anfrage instanziiert wird. Ein 'Service' hat eine Schnittstelle, die von dem 'ServiceType' definiert wird. Ein 'Service' wird als ein Aktivitätsdiagramm dargestellt und mit dem Daten- und Kontrollefluss spezifiziert. Die Tags 'ServiceType' und 'ServiceKind' legen die Attributen der Konkurrenz zwischen mehreren 'ServiceTypes' fest. Die Nachrichten 'start' und 'end' werden entsprechend dem Ablauf des Services generiert (falls eine Ausnahme ausgelöst und das Service abgebrochen wird, wird die 'end' Nachricht nicht generiert). Diese Nachrichten werden an allen 'state' Komponenten, die sich im gleichen Gültischkeistsbereich

befinden, geleitet. Ein 'service' kann konkurrierend mit sich selbst oder anderen Services laufen.

Ein **Port** spezifiziert das konkurente und reaktive Verhalten, dass extern beobachtbar ist. Wie es im [4] empfohlen wurde, kann die Schnittstelle mit der präzisen operationellen Semantik beschrieben werden. Ein 'port' exportiert eine mit den Tags annotierte Kollektion von 'ServiceTypes', die die Gleichläufigkeit spezifizieren. Ein 'port' erzwingt die Vorbedingungen für die 'ServiceTypes', wobei ihre interne Spezifikation die Nachbedingungen garantiert. Eine 'port' Komponente bearbeitet auch die Kommunikation zwischen den Komponenten. Das dazugehörende 'AccessOrder' Verhalten bestimmt die Ordnung der Aufrufe des Services (z.B. temporale Abhängigkeit zwischen beschriebenen Services).

Ein **State** bestimmt die reaktiven und koordinierenden Aspekten sowie die Ausnahmefälle einer 'system' Komponente. Das begleitende 'Reactive' Verhalten läuft asynchron mit seinen Services und wird von einem Zustandsautomat beschrieben. Also, eine mit dem 'state' Verhalten assoziierte 'system' Komponente entspricht einen abstrakten Monitor mit den gleichzeitläufenden Kontrollethreads. Obwohl eine 'state' Komponente entspricht einer 'AccessOrder' Spezifikation von einer korrespondierenden 'port' Komponente, sie kann zusätzliche abstrakte Zustande, Transitionen und Aktivitäten enthalten. Die Methoden *wait* und *notify* ermöglichen die Synchronization von Services. Überdies empfängt eine 'state' Komponente die den Ablauf der Services entsprechende Events '*start*' und '*end*'.

ScenarioContext: ein 'ScenarioContext' repräsentiert das mit jeder UseCase verbundene Zusammenspiel von internen Komponenten. Diese Kontexten spezifizieren den Nachrichten- und Eventsaustausch sowie die Koordination als Reaktion auf externen Anreizen wie z.B. Events oder Aufrufe. Für die Modellierung von 'ScenarioContext' werden die Sequenzdiagrammen mit der Semantik der Lebendigkeit verwendet. Sie unterstützen auch die Überprüfung von der Eigenschaften einer Komponente. Diese Kontexte beschreiben das prinzipielle Systemsverhalten ohne das Fehlerszenario, das wegen falschen Vorbedingungen oder Guard Expressions auftreten kann, zu berücksichtigen.

ActivityExecution ist die Generalization von der in dem SPT-Profile beschriebener ActionExecution. Die Aktivitäten sind von der höheren Granularität und entsprechen einen Service. Ein Service ist mit einem run-time Handler 'ServiceHandler' aus dem 'port' Komponente verbunden, so kann man über den Zähler der Inkarnationen von Serviceinstanzen erfahren. Diese Information kann für die Spezifizierung von komplexen Synchronisationsstrukturen in Form von globalen Invarianten benutzt werden und so die Sicherheitsbedingungen bestimmen. Es gibt Übersetzungen von den globalen Invarianten in den Synchronisationscode, der die Konstrukten des niedriges Grades wie Semaphoren, Monitors usw. benutzt.

GuardedAction hilft die gleichläufigen Aspekten zu beschreiben. Mit den 'GuardedActions' kann die präzise Semantik für die Evaluierung von Guards spezifiziert werden. Die 'GuardedAction' Komponente beschreibt die Synchronisation und die Bearbeitung von Ausnahmen. Die Ausnahmen werden von der

entsprechenden 'state' Komponente bearbeitet oder in den 'state' Komponenten der höheren Ebene geworfen (wie im Java *try-catch* Block). Also, 'GuardedAction' bietet die Konstruktoren für die Synchronisation und Ausnahmebehandlung von sequenziellen Abläufen in der nebenläufige Umgebung. Die externe Kommunikation von den cmUML Komponenten basiert auf die Nachrichten, wobei die interne Kommunikation auf die nachrichten und auf verteilten Ressourcen basiert.

Die formale Beschreibung des cmUML-Profiles folgt in der Tabelle 1.

Stereotype (UML, *SPT Element*)	**Tags[tagtyp](Multiplizität); Spezialisierung/Generalisierung;** **– Einschränkungen; (Beschreibung)**
Component (Class, *Descriptor*, *ConcurrentUnit)*	(Abstract) spec[Behaviour](*); root[≪system≫]; concurrencyKind= {*concurrent, sequential*}, evBuffer[≪resource≫] **Spezialisierung** : system, port, state, service, resource
≪system≫	port[≪port≫]; state[≪state≫]; service[≪service≫](*); – port, state sind nicht *null*
≪port≫	interface[≪serviceType≫](*); spec[≪AccessOrder≫]; handles[≪serviceHandler≫](*); policy={*FIFO, Priority*} – concurrencyKind='sequential'; port, state sind *null*
≪state≫	spec[≪Reactive≫]; – concurrencyKind='sequential'; port, state sind *null*
≪service≫ (*Scenario*)	spec[≪Flow≫]; (dynamisches *Component* eines Systems) – concurrencyKind='sequential' – evBuffer ist *null* **Generalisierung**: ActivityExecution
≪resource≫ (*ProtectedResource*)	(geschützte Einheit, die muss für die *read* und *write* Zugriff (atomar) erworben und freigegeben werden. Ressourcen mit komplexem Verhalten können als ≪system≫Komponenten spezifiziert werden)
ServiceType (Operation)	max[integer]; serviceKind={*read, write*}; parService[≪serviceType≫](*) params[string]l (ein String kann mit der BNF-ähnlichen Grammatik interpretiert werden)
ServiceHandler	execs[≪servie≫](*); in(integer); out(integer);

Fortsetzung auf der nächsten Seite

Fortsetzung	
Stereotype (UML, *SPT Element*)	**Tags[tagtyp](Multiplizität); Spezialisierung/Generalisierung;** **– Einschränkungen; (Beschreibung)**
(Classifier)	– service executions corresponding to a service handler are of same serviceType
GuardedAction (Action)	guard[bool]; isDelay[bool]; isHot[bool]; exception[≪exception≫]; isAtomic[bool] **Generalisierung** : GuardedActivity
ActivityExecution (*Action Execution*)	(Abstract) ('Activity' ist eine Folge von atomaren Aktionen, die eine partielle Ordnung haben kann)
Exception (*Stimulus*)	(service execution containing exception raising action terminates)
MessageAction	synchKind={*send, accept, return*} (entspricht den asynchronen Aufruf, synchronen Aufruf bis die Nachricht akzeptiert wird, synchronen Aufruf bis das Resultat bekommen oder der Service beendet wird)
ScenarioEv (*EventOccurence*)	eventKind={*start, end*} (entspricht den Start und das Ende der Serviceausführung)
Exception (*Stimulus*)	(service execution containing exception raising action terminates)
AccessOrder (BehaviourStateMachine)	scope: {*local, global*} ('scope' legt fest, ob die Zugrifsordnung global oder pro Client anwendbar ist)
Reactive (BehaviourStateMachine)	Repräsentiert das reaktive Verhalten eines≪Systems≫, das asynchron mit der ≪service≫Komponente läuft
Flow (Activity)	Repräsentiert den Daten- und Kontrollefluss einer ≪service≫Komponenten
ScenarioContext (Sequence)	(specification of behaviour service interactions in response to external requests with liveness semantics)
Assertion, Invariant (Constraint)	(Assertion - eine Einschränkung auf den lokalen Daten Invariant - eine Einschränkung auf globalen Daten z.B. zält wieviel Mal ein Service aufgerufen wurde)

Tabelle 1. cmUML Profil (Quelle: [3])

4 Beispiel der cmUML-Nützung

In diesem Kapitel wird die Methodologie aus [3] für die Spezifizierung eines parallelen Systemes anhand eines Beispiels erläutert. Im Beispiel handelt es sich um einen vereinfachten Aufzug. Um das System im beherrschbaren Umfang zu halten, nehmen wir an, dass der Anzug zu einem Zeitpunkt nur von einem Benutzer gerufen werden kann.

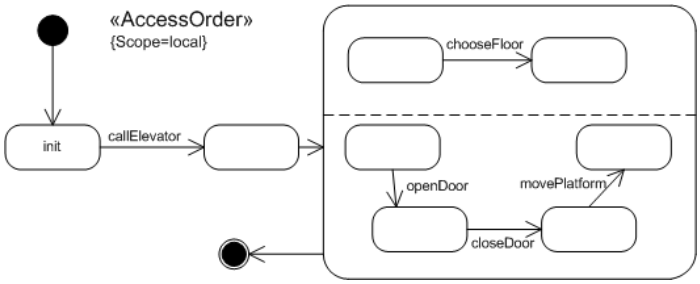

Abbildung 1. ≪port≫Spezifikation für ein **Elevator** System

Im **1. Schritt** soll die Schnittstelle der Komponente identifiziert werden, d.h. die Interaktion mit der Umgebung (z.B. Benutzer) betrachtet wird. Die Information kann von den Anwendungsfällen oder der Problemstellung stammen.

Im Fall eines Aufzugs drückt der Benutzer die Taste im Gang um den Fahrstuhl zu rufen (die Aktion 'callElevator'). Der Aufzug öffnet die Tür ('openDoor'), wartet bis der Benutzer die Zieletage wählt ('chooseFloor'), schließst die Tür ('closeDoor') und bewegt den Fahrstuhl ('movePlatform'). Siehe Abbildung 1 für die graphische Darstellung.

Im **2. Schritt** wird die mögliche Nebenläufigkeit und temporale Abhängigkeiten zwischen den Services einer Komponente gefunden. Hier hilf die externe Analyse der angebotenen Services. Die Zusammenhängen werden als das 'AccessOrder' Verhalten der entsprechenden 'port' Komponente spezifiziert. 'AccessOrder' ist das Zustandsautomat des Verhaltens. Da wir im Beispiel auf einem Benutzer eingeschränkt haben, 'scope' wird auf 'local' gesetzt. In Realität mit mehreren Benutzern sollte 'scope' auf 'global' gesetzt werden.

Im **3. Schritt** wird die Systemsdekomposition durchgeführt. Man findet nebenläufige und sequenzielle Regione. Wie man in der Abbildung 1 sieht, die Aktionen 'chooseFloor' und ('openDoor' - 'closeDoor' - 'movePlatform') können gleichzeitig stattfinden und diese Tatsache wird mit der gestrichelten Linie abgebildet. Die Aktion 'chooseFloor' wird von der Komponente 'AufrufTaste' und die Aktionen ('openDoor' - 'closeDoor' - 'movePlatform') von der Komponente 'Fahrstuhl' ausgeführt. Die Kompositionsstruktur wird mit dem Komponentendiagramm (siehe Abbildung 2) modeliert.

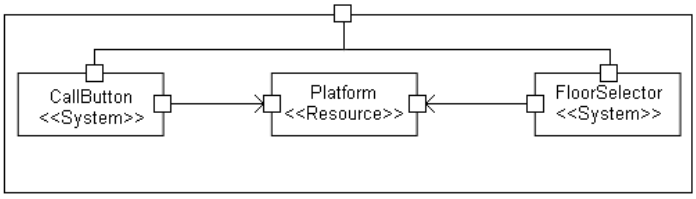

Abbildung 2. ≪port≫Spezifikation für ein **Elevator** System

Die temporale Abhängigkeiten, die als Teil der Schnittstelle spezifiziert wurden, sollen auch in der interne Spezifikation erhalten werden. Dazu dient die ≪state≫Komponente (siehe Abbildung 3).

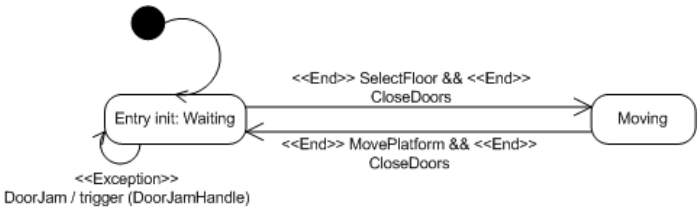

Abbildung 3. ≪port≫Spezifikation für ein **Elevator** System

4. Schritt. Für jeden *ScenarioContext* (Use Case) wird ein oder mehrere Sequenzdiagrammen erstellt, die die Interaktion zwischen 'system', 'service', 'port' 'state' und 'resource' Komponenten darstellen. Die Events sollen eindeutig eingeordnet werden. Wir nehmen an, dass 'Env' alle Hardware-Schnittstellen (Anrufftaste, Taste für das Wählen der Etage, Notrufsstaste) enthält.

Im Beispiel (siehe Abbildung 4) bilden alle solide Linien (Lebenslinien, Nachrichten) obligatorisches Verhalten ab, wobei die gestrichelte Linien das optionale Verhalten darstellt.

Im **5. Schritt** werden die Services der 'system' Komponente als UML Aktivitätsdiagramm spezifiziert und die zu implementierende Funktionen identifiziert. Die 'Guard'-Aktionen sollen auch festgelegt werden. Das Au'sen-Guard darf sich nicht während des Ablaufs der Funktion SelectFloor ändern.

Im **6. Schritt** werden die 'service', 'state', 'ScenarioContext' Spezifikationen weiter verfeinert. Die Synchronizations- und Ausnahmenhandlungsfälle zwischen parallel laufenden 'services' und 'state' Komponenten werden berücksichtigt und passende Invariante als 'guards' definiert. In diesem Schritt werden auch die interne und externe Events als auch die Antworte auf die Anfragen identifiziert.

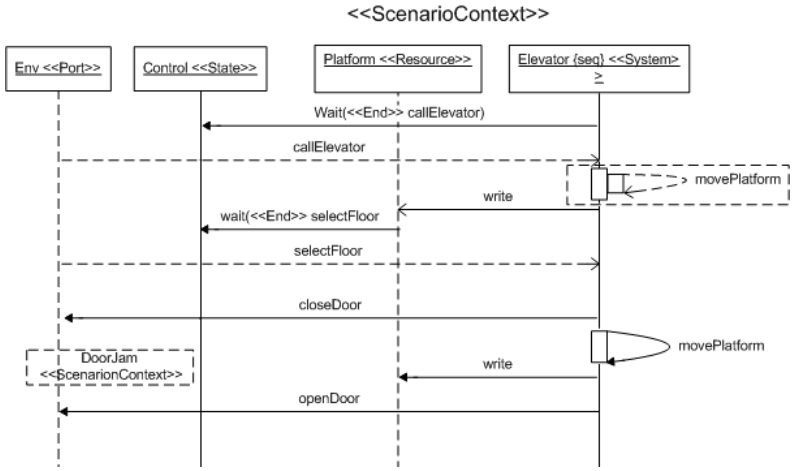

Abbildung 4. ≪port≫Spezifikation für ein **Elevator** System

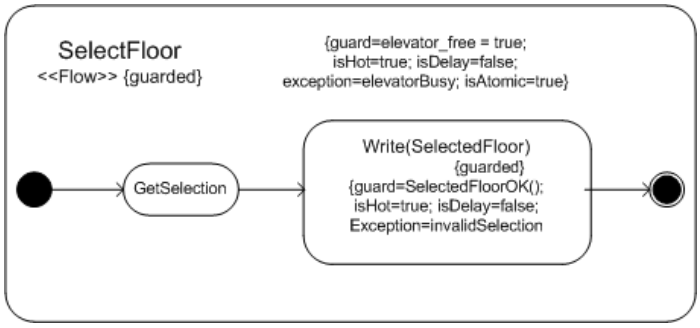

Abbildung 5. ≪port≫Spezifikation für ein **Elevator** System

Im **7. Schritt** werden alle vorige Schritte rekursiv auf die in dem 2. Schritt identifizierten Subkomponenten ausgeführt.

5 Ausblick

Die UML hat keine formale Semantik und mangelt an Konstrukten für die Modellierung der parallelen Systemen. Die Mechanismen für die Erweiterung der UML lassen diese Lücke teilweise schließen. In vorliegender Arbeit wurden einige UML-Profile beschrieben, die sich für die Spezifizierung von parallelen Systemen eignen. Weiter wurde das cmUML Framework beschrieben, das speziell für diesen Zweck entwickelt wurde. Mit Hilfe eines einfaches Beispiels wurde die Methodologie für die Nützung des cmUML erläutert.

Literatur

[1] A. Gherbi and F. Khendek. Uml Profiles for Real-Time Systems and their Applications. *Journal of Object Technology*, 5(4):149–169, May-June 2006.
[2] I. S. Iulian Ober. On the Concurrent Object Model of uml. formal/07-02-05, February 2005.
[3] S. R. Jagadish Suryadevara, Lawrence Chung. A UML Based Framework for Formal Specification of Concurren, Reactive Systems. *Journal of Object Technology*, 4(3):149–169, May-June 2005.
[4] L. Lamport. A Simple Approach to Specifying Concurrent Systems. *Communications of the ACM*, 32(1):32–45, Januar 1989.
[5] OMG. A UML Profile for MARTE, Beta 1. formal/07-08-04, February 2007.
[6] S. Pllana and T. Fahringer. Uml Based Modeling of Performance Oriented Parallel and Distributed Applications. In *Proceedings of the 2002 Winter Simulation Conference*, pages 497–505, 2002.
[7] C. Rupp, S. Queins, and B. Zengler. *UML 2 Glasklar*. Hanser Verlag München, 2007.

Programmierung mit gemeinsamem Adressraum

Sven Mentl

Universität Augsburg
sven.mentl@web.de

Zusammenfassung Die technische sowie die preisliche Entwicklung von Multicore-Prozessoren führt unweigerlich dazu, dass die Entwicklung paralleler Programme in Zukunft eine wesentlich wichtigere Rolle als bisher spielen wird. Diese Seminararbeit beschäftigt sich mit der Entwicklung paralleler Programme, unter der Annahme eines gemeinsamen Adressraums. Hierzu werden zuerst Konzepte und Probleme, die dabei auftreten, näher erläutert und schließlich Java Threads, POSIX Threads und OpenMP als APIs zur Entwicklung Thread-basierter paralleler Programme vorgestellt.

1 Einleitung

> Eines der wichtigsten Ziele der Parallelverarbeitung besteht darin, Aufgaben in einer kürzeren Ausführungszeit zu erledigen, als dies durch eine Ausführung auf einem sequentiellen Rechner möglich wäre. ([7], Seite 1)

Die Verwendung von paralleler Verarbeitung dient dazu, die Verarbeitungsgeschwindigkeit zu steigern. Hohe Verarbeitungsgeschwindigkeit sowie die Bewältigung großer Datenmengen sind Kennzeichen des High Performance Computing ([11]), die nur durch Verwendung von Parallelität erreicht werden können. Durch den Trend der Prozessorhersteller, immer mehr Kerne auf einen Prozessor zu packen, wird es nun aber auch für Programmierer, die nicht im Spezialbereich des High Performance Computing tätig sind, interessant und notwendig, sich mit der parallelen Programmierung auseinander zusetzen. Auf die technologische Entwicklung, die die parallele Programmierung zu einem sehr aktuellen Thema macht, wird im Anhang A näher eingegangen.

Parallele Programme bestehen aus mehreren Arbeitssträngen, die dann parallel abgearbeitet werden können. Diese Arbeitsstränge werden durch die Konzepte Prozesse und Threads repräsentiert, die in Kapitel 2 eingeführt werden. Es existieren verschiedene Ansätze, parallele Programme zu entwickeln, eine grobe Unterscheidung lässt sich darin treffen, ob den parallelen Arbeitssträngen ein gemeinsamer Adressraum zur Verfügung steht oder ob jeder Arbeitsstrang seinen eigenen Speicher besitzt. Einen Überblick über die Entwicklung für einen verteilten Adressraum erhalten sie in der Seminararbeit “Programmierung für den verteilten Adressraum” von Thomas Eisenbarth. Im Normalfall sind die Arbeitsstränge eines Programms nicht voneinander unabhängig, sondern weisen Abhängigkeiten auf. Dies macht eine Kommunikation der Prozesse und Threads

untereinander notwendig, die im Falle des gemeinsamen Adressraums über gemeinsame Variablen erfolgen kann. Konzepte, Techniken und Probleme bei der Koordination und Synchronisation des Zugriffs auf gemeinsame Variablen werden in Kapitel 3 erläutert. Schließlich werden als APIs zur Programmierung paralleler Programme, denen ein gemeinsamer Adressraum zugrunde liegt, die **Java Threads** (Kapitel 5), **Posix Threads** (Kapitel 4) und **OpenMP** (Kapitel 6) vorgestellt.

2 Threads und Prozesse

In der Einleitung wurde bereits erwähnt, dass ein paralleles Programm aus mehreren Aktivitätsträgern besteht. In diesem Kapitel werden die Konzepte Prozesse und Threads, die eben solche Aktivitätsträger darstellen, definiert und voneinander abgegrenzt.

2.1 Prozesse

Ein Prozess ist definiert, als ein sich in Ausführung befindliches Programm (vgl hierzu [8], S. 40). Zur Ausführung eines Programms muss der ausführbare Code im Speicher abgelegt werden, sowie Speicher für die Daten bereitgestellt werden. Weiterhin benötigt ein Prozess eventuell vom Betriebssystem bereitgestellte Ressourcen und einen Prozessor, der den Prozess letztendlich ausführt. Den in einem System vorhandenen Prozessen muss also der Prozessor zugeteilt werden, was auch als Scheduling bezeichnet wird und Aufgabe des Betriebssystems ist. In einem Multitasking-Betriebssystem werden den vorhanden Prozessoren die Prozesse durch ein Scheduler sequentiell zugeordnet. Teilt der Scheduler den Prozessor einem anderen Prozess zu, müssen alle notwendigen Betriebsmittel (Register, Puffer, Heap, Stack,..) des suspendierten Prozesses gesichert werden, und die Betriebsmittel des nun aktuellen Prozesses aus dem Hauptspeicher geladen werden. Dieser Vorgang wird auch als **Kontextwechsel** ([8], S. 40) bezeichnet. Ein klassischer Unix-Prozess besteht aus den folgenden Bereichen (siehe Abbildung 1):. Im Text-Segment befindet sich der auszuführende Code, während im Data-Segment die initialisierten Variablen abgelegt werden. Im BSS-Segment werden alle nicht-initialisierten globalen Variablen gespeichert. Weiterhin besteht er aus einem nach oben wachsenden Heap-Segment, in dem dynamisch allokierte Speicherbereiche angelegt werden, und einem nach unten wachsenden Stack.

Im Kontext dieser Seminararbeit werden wir den Prozess nicht weiter als Umsetzung für einen Arbeitsstrang beachten, da es Prozessen nach der eben genannten Definition nicht möglich ist, einen gemeinsamen Speicher zu besitzen. Vollständigkeitshalber sei aber erwähnt, dass es die Möglichkeit gibt, ein spezielles Segment anzulegen, das so genannte Shared Memory Segement, dass als gemeinsamer Speicherbereich für Prozesse dient. Dieses Vorgehen ist zum einen zu stark vom konkreten Betriebssystem abhängig, so dass es hier nicht angemessen repräsentiert werden kann. Zum anderen handelt es sich auch nicht um die

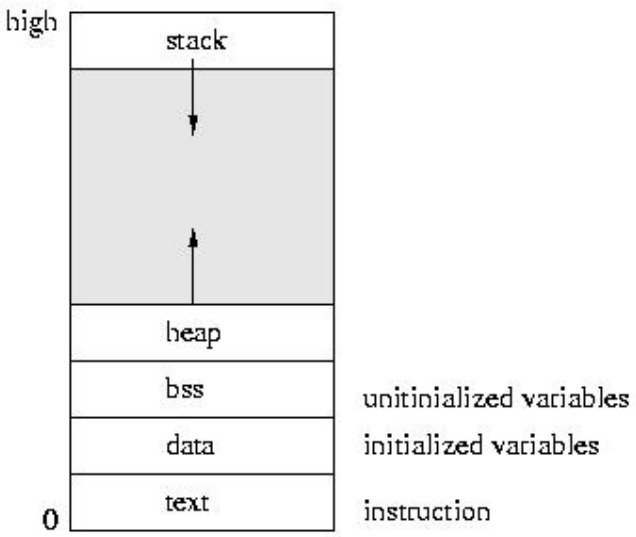

Abbildung 1. Aufbau eines klassischen Unix-Prozesses [17]

typische Vorgehensweise, um parallele Programme zu entwickeln. Informationen zur Shared Memory Segment Programmierung für Windows finden sie in [2] und für Unix in [5] .

2.2 Threads

Threads sind eine Weiterentwicklung des Prozess-Konzepts, in der Literatur ist auch oft die Bezeichnung leichtgewichtiger Prozess anzutreffen (Lightweight Process, LWP [14]). Diese Bezeichnung lässt bereits vermuten, dass der bei den Prozessen angesprochene Kontextwechsel bei Threads wechsentlich günstiger ausfällt. Ein Prozess besteht nun nicht mehr aus einem Berechnungsstrom, sondern aus mehreren Berechnungsströmen, den Threads. Diese teilen sich alle den selben Adressraum. Ein Vorteil gegenüber einem verteilten Adressraum ist, dass die Kommunikation zwischen Threads sehr einfach und günstig über gemeinsame Variablen funktioniert. Allerdings ist dieser Vorteil auch mit einem Mehraufwand verbunden, denn die Zugriffe von Threads auf gemeinsame Variablen müssen nun synchronisiert werden (vgl. hierzu 3.1). Der Aufbau ist etwas anders als der des klassischen Unix-Prozesses; betrachten wir hierzu exemplarisch den Aufbau eines Prozesses in Java. Wie in Abbildung 2 zu sehen, besitzt auch ein Java-Prozess einen Stack und einen Heap. Jeder Thread hat seinen eigenen Laufzeitstack, seinen eigenen Program Counter und seinen eigenen Native method stack, teilt sich aber alle sonstigen Resourcen mit den anderen Threads (siehe Abbildung 3). Mehr zum Aufbau der JVM findet sich in [16].

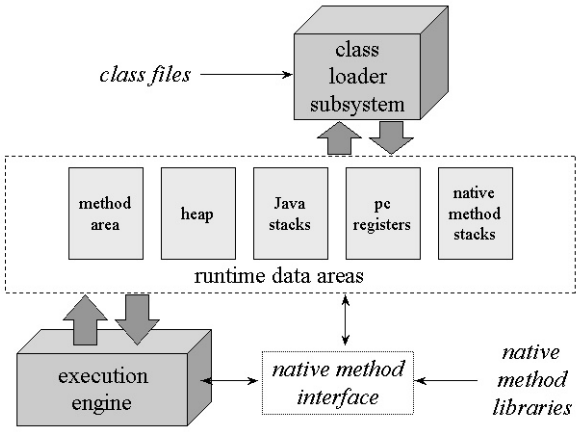

Abbildung 2. Der Prozessaufbau in der JVM (vgl. hierzu [16])

Es gibt genau genommen zwei unterschiedliche Arten von Threads, die User-Threads und die Kernel-Threads. Bei User-Threads handelt es sich um Threads, die komplett im Userspace (also ohne Kenntnis des Betriebssystems) ablaufen, das heißt, das Betriebssystem hat keine Kenntnis davon, dass ein Prozess aus mehreren Threads besteht. Dies hat Vor- und Nachteile. Zum einen kann man mit Threads auch auf Betriebssytemen arbeiten, die gar keine Threads kennen und zum anderen ist ein Wechsel zwischen zwei User Threads sehr schnell, da kein Kontextwechsel nötig ist. Ein Kernel Thread hingegen ist ein Thread der dem Betriebssystem bekannt ist und wird deshalb auch vom Betriebssystem-Scheduler verwaltet.

2.3 Thread-Prozess Mapping

Wir haben nun bereits Prozesse, User Threads und Kernel Threads kennen gelernt, und werden nun sehen, wie diese zusammen gehören. Zur Programmierung mit Threads wird entweder eine Threadbibliothek wie Pthreads oder OpenMP verwendet, oder die Threads sind integraler Bestandteil der Programmiersprache wie in Java. Kennt das Betriebssytem keine Threads, sondern nur Prozesse, wird unser Programm und alle darin enthaltenen Threads auf einen Prozess abgebildet. Es handelt sich also um eine N zu 1 Abbildung mehrerer Threads auf einen Prozess. Das Scheduling der Threads muss von der Threadbibliothek selbst erledigt werden. Der Vorteil dieser Variante ist der schnelle Wechsel zweier Threads, der Nachteil, dass es nicht möglich ist den Prozess auf mehrere Kerne zu verteilen sowie die Tatsache, dass ein blockierender Aufruf eines Threads den gesamten Prozess blockiert. Eine andere Möglichkeit besteht darin, jeden Thread

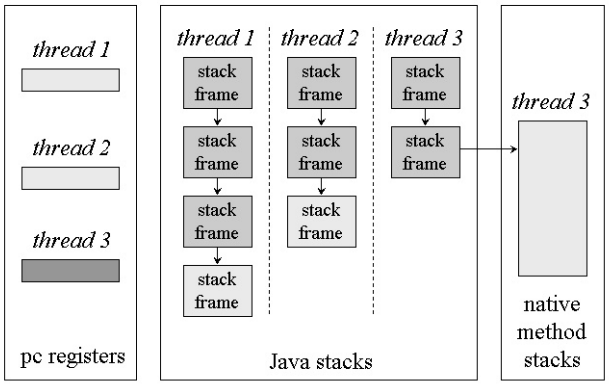

Abbildung 3. Java Threads (vgl. hierzu [16])

unseres Programms auf einen Kernel-Thread zu mappen, sofern das Betriebssystem Threads unterstützt. Diese Variante ist sehr einfach, allerdings auch sehr kostspielig. Bei der 1 zu 1 Abbildung übernimmt das Betriebssystem das Scheduling. Die Variante N User Threads auf M Kernel Threads abzubilden stellt den Kompromiss zwischen den beiden bereits genannten Möglichkeiten dar. Bei dieser Variante haben wir einen Bibliotheks- sowie einen Betriebssystem-Scheduler. Weitere Ausführungen finden sich in [8].

2.4 Scheduling von Threads

Wie bereits angesprochen ist der Scheduler dafür verantwortlich, in welcher Reihenfolge und wie lange den einzelnen Threads der Prozessor zugeteilt wird. Man unterscheidet präemptive von nicht-präemptiven, auch kooperativ genannten Verfahren. Bei einem kooperativen Scheduling-Verfahren besitzen die Threads den Prozessor so lange, bis sie ihn von sich aus wieder freigeben. Ein präemptives Verfahren hingegen kann den Prozess unterbrechen und ihm den Prozessor entziehen, um diesen nach einer bestimmten Strategie einem anderen Prozess zuzuteilen. Wie wir später noch sehen werden, sind die Schedulingverfahren meist prioritätsbasiert, was bedeutet, dass manche Threads vom Scheduler gegenüber anderen bevorzugt behandelt werden können. Mehr zum Scheduling findet sich in jedem gängigen Buch über Betriebssysteme.

2.5 Zustände eines Threads

Ein Thread kann sich in verschiedenen Zuständen befinden, nur im Zustand **Laufend** wird er tatsächlich ausgeführt. Die anderen Zustände sind **Neu Erzeugt**,

Lauffähig, **Wartend** und **Beendet**. Die Zustände eines Threads unterscheiden sich etwas von API zu API, deshalb finden sich im Anhang weitere Information zum Zustandsmodell von Java-Threads sowie POSIX-Threads.

3 Parallele Programmierung mit Threads

Die Aktivitätsträger eines parallelen Programms sind die im vorherigen Kapitel behandelten Threads. Bei der Verwendung von Threads gibt es nun verschiedene Aspekte die der Programmierer beachten muss, um ein korrektes paralleles Programm zu entwickeln. Threads eines Programms müssen bezüglich des Zugriffs auf gemeinsam genutzte Variablen synchronisiert und koordiniert werden. Da jede API hier etwas unterschiedliche Möglichkeiten bereitstellt, wird dies erst im Kapitel der jeweiligen API abgehandelt.

3.1 Race Conditions

> Abläufe, deren Endergebnis,..., von der zeitlichen Abfolge der ausgeführten Operationen abhängig ist, nennt man zeitkritische Abläufe oder Race Conditions. (Zitat aus [17])

Insbesondere beim Zugriff auf gemeinsam genutzte Variablen, spielt die zeitliche Abfolge eine entscheidende Rolle. Hierzu folgendes Beispiel, inklusive Erläuterung, aus [13].

```
static int i;                0 getstatic #19 <Field int i>
static void foo()            3 iconst_1
{                            4 iadd
i++;                         5 putstatic #19 <Field int i>
}                            8 return
```

Abbildung 4. i++ in Java Bytecode

Rufen zwei Threads A und B die Funktion foo() auf, so kann folgendes passieren:

- Thread A holt sich den Wert von i in den internen Speicher, wird dann aber unterbrochen. Er kann das um 1 erhöhte Resultat nicht wieder i zuweisen.
- Nach der Unterbrechung von A kommt Thread B an die Reihe. Auch er besorgt sich i, kann aber i + 1 berechnen und das Ergebnis in i ablegen. Dann ist B beendet, und der Scheduler beachtet Thread A.
- Jetzt steht in i das von Thread B um 1 erhöhte i. Thread A addiert nun 1 zu dem gespeicherten alten Wert von i und schreibt dann nochmals denselben Wert wie Thread B zuvor. Insgesamt wurde die Variable i nur um 1 erhöht, obwohl zweimal inkrementiert werden sollte. Jeder Thread hat für sich gesehen das korrekte Ergebnis berechnet.

Die Bereiche, in denen es zu Race Conditions kommen kann, werden auch **kritische Bereiche** genannt. Um Race Conditions zu vermeiden, muss die Anweisung i++ atomar ausgeführt werden. Welche Sprachkonstrukte die jeweiligen APIs hier zur Verfügung stellen, wird später genauer behandelt.

3.2 Nebenläufigkeit und Parallelität

Auf einem Prozessor kann zu jedem Zeitpunkt nur genau ein Thread ausgeführt werden, der den Prozessor exklusiv besitzt. Wirkliche Parallelität im Sinne von gleichzeitiger Ausführung ist nur dann möglich, wenn mehrere Recheneinheiten zur Verfügung stehen. Threads auf einem Prozessor werden nebenläufig (oder auch qausiparallel genannt) ausgeführt. Sind zwei Threads zueinander nebenläufig, so ist die Reihenfolge, mit der diese beiden Threads ausgeführt werden, für ein korrektes Ergebnis des Programms irrelevant. Durch das schnelle Wechseln der ausgeführten Threads entsteht bei der nebenläufigen Ausführung der Eindruck von Parallelität ([15], S. 805).

3.3 Threadsicherheit

Durch die Tatsache, dass sich Threads einen Adressraum teilen, kann es zu Race Conditions und somit zu nicht-deterministischen Verhalten kommen. Beim Benutzen von Sofwarekomponenten (i.e. dem Aufruf von Funktionen) ist es eine wünschenswerte Eigenschaft, dass die Softwarekomponente von mehreren Threads gleichzeitig benutzt werden kann, ohne dass solche Inkonsistenzen auftreten. Diese Eigenschaft wird als Threadsicherheit (engl. thread safety) bezeichnet. Threadsicherheit kann durch gegenseitigen Ausschluß oder durch reentrante Funktionen erreicht werden.

3.4 Reentrant

Ist der gleichzeitige Zugriff mehrerer Threads möglich, so bezeichnet man die Komponente auch als eintrittsinvariant oder reentrant. Eine reentrant programmierte Funktion erreicht diese Eigenschaft allerdings nicht durch Verwendung irgendwelcher Synchronisationsmechanismen. Im folgenden Beispiel ist die Funktion non_reentrant nicht gleichzeitig von mehreren Threads benutzbar, da sie auf einer globalen Variable operiert. Es ist leicht einzusehen, dass, wenn mehrere Threads versuchen würden, diese Funktion aufzurufen, das Ergebnis, also der Wert von g_var nicht vorhersehbar ist. Um eine Funktion reentrant zu programmieren, müssen alle Parameter in lokale Variablen kopiert werden und befinden sich dadurch im jeweils threadeigenen Speicherbereich. Hiermit wird das Problem umgangen, dass die Variable sich im gemeinsamen Adressraum befindet und es deshalb zu Inkonsistenzen kommen kann.

```
int g_var = 1;

int non_reentrant()
{
  g_var = g_var + 2;
  return g_var;
}
```

```
int reentrant(int i)
{
  int priv = i;
  priv = priv + 2;
  return priv;
}
```

Abbildung 5. Eine nicht-eintrittsinvariant programmierte Funktion und eintrittsinvariantes Pendant (vgl hierzu [13])

3.5 Deadlocks

> A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause. [12]

Ein Deadlock tritt dann auf, wenn ein Thread A auf ein Betriebsmittel oder einen Lock wartet und somit blockiert. Der Thread B, der aktuell das Betriebsmittel oder den Lock besitzt, kann diesen allerdings nicht freigeben, da er seinerseits blockiert ist, weil er auf ein von Thread A gesperrten Lock oder gesperrtes Betriebsmittel warten muss. Ist diese Situation eingetreten, gibt es kein Weiterkommen. Das Eintreten eines Deadlocks kann allerdings immer dadurch verhindert werden, dass alle Threads immer in derselben Reihenfolge Objekte sperren und freigeben.

3.6 Livelocks

Interagieren zwei Threads so miteinander, dass die Reaktion von Thread 1 auf eine Aktion von Thread 2 dazu führt, dass Thread 2 auf die Reaktion von Thread 1 wiederum in einer unglücklichen Art und Weise reagiert, kann es zu einem Livelock kommen. Das folgende Zitat veranschaulicht das Problem sehr schön:

> This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so... (siehe [10])

3.7 Starvation

Ein Thread, der nie einen Mutex oder Lock auf eine benötigte Ressource erhält, da die Ressource jedem anderen Thread zugeteilt wird, nur ihm selbst nicht, verhungert, was auch als Starvation bezeichnet wird. Starvation hängt eng mit dem Scheduling von Threads zusammen. Insbesondere beim prioritätsbasierten Scheduling kann es schnell passieren, dass Threads mit hoher Priorität, Threads

kleinerer Priorität ständig verdrängen. Ein damit verwandtes Problem ist die **Prioritätsinversion** ([8], S.114). Hier bekommt ein Thread mit hoher Priorität zwar sehr häufig den Prozessor zugeteilt, kann aber nicht fortfahren, da er auf das Ergebnis eines Threads mit wesentlich kleinerer Priorität warten muss. Wie man sieht, sollte die Vergabe von Prioritäten gut durchdacht sein.

3.8 Parallele Programmiermuster

Aus der Arbeit mit Threads haben sich im Laufe der Zeit verschiedene Programmiermuster bzw. Patterns entwickelt, die beschreiben, wie Programme, die Threads benutzen, dies auf ein sinnvolle und strukturierte Art und Weise tun können. Für jedes thread-basierte Programm müssen gewisse Designentscheidungen getroffen werden, dazu gehört zum Beispiel die Anzahl der Threads und ob diese dynamisch oder statisch erzeugt werden. Das **Fork und Join** Pattern ist eines der Älteren, welches typischerweise bei der Arbeit mit Unix-Prozessen Verwendung findet, aber genauso sinnvoll auch bei Threads verwendet werden kann. Jedes Programm startet mit einem einzigen Thread, dem Master-Thread, der sich dann an einem bestimmten Punkt gabelt (engl. fork). Der Haupt-Thread sowie der abgespaltene neue Thread können nun beide Arbeit verrichten. Der Master-Thread wartet auf das Ende des abgespaltenen Threads (i.e. Aufruf der join Funktion). Der Punkt, an dem der Master-Thread wartet, stellt einen Synchronisationspunkt dar, an dem der Master-Thread als einziger mit der Arbeit fortfährt und dabei ein eventuell vorhandenes Ergebnis des zuvor abespaltenen Threads verwendet. Das **Parbegin-Parend** Konzept ist dem Fork-Join Prinzip recht ähnlich, mit dem Unterschied, dass an einem bestimmten Punkt (Parbegin) mehrere Threads erzeugt werden, welche dann parallel arbeiten und an einem bestimmten Punkt (Parend) wieder zusammengeführt werden. Wiederum sehr ähnlich zum **Parbegin-Parend** Konzept ist das **SPMD und SIMD** Muster, welches uns später auch noch bei OpenMP (Abschnitt 6) begegnen wird. Auch hier existieren mehrere Threads, die alle den selben Programmcode, allerdings auf unterschiedlichen Daten, ausführen. Beim **Master-Slave** Pattern existieren mehrere Threads, von denen einer der Master ist und alle anderen die Slaves. Des Master-Thread's einzige Aufgabe ist es, die Arbeit an die Slaves zu verteilen sowie gewisse Kontroll- und Managementfunktionen zu übernehmen. **Pipelining** ist ein Konzept zur Effizienzsteigerung, welches man aus dem Bereich der Mikroprozessoren kennt. Es existieren mehrere Threads t_1 bis t_n, wobei die Ausgabe von t_x die Eingabe von t_{x+1} darstellt. Hat t_1 sein Ergebnis weitergegeben, kann er sofort mit der nächsten Aufgabe beginnen. **Taskpools** besitzen eine gewisse Ähnlichkeit mit dem **Master-Slave** Modell, im Gegensatz dazu existiert allerdings keine zentrale Verwaltungsinstanz, die die Arbeit verteilt. Im Taskpool sind alle vorhandenen Aufgaben (engl. tasks) zu Beginn enthalten. Ein Thread holt sich nun einen Task aus dem Taskpool und arbeitet diesen ab, der Thread kann dabei auch neue Tasks erzeugen, die dann wieder im Taskpool abgelegt werden. Sind alle Threads fertig und der Taskpool leer, so ist auch das Programm fertig abgearbeitet. Das **Produzenten-Konsumenten Modell** ist ähnlich dem Pipelining, hier haben wir Produzenten-Threads, die etwas erzeugen, das den

Konsumenten-Threads als Eingabe dient. Die von Produzenten-Threads erzeugten Elemente werden in einem gemeinsamen Puffer abgelegt, aus dem sich die Konsumenten-Threads dann bedienen können. Ausführlichere Informationen zu diesem Abschnitt finden sie in [8].

4 POSIX-Threads

Bei POSIX (Portable Operating System Interface [4]) handelt es sich um verschiedene Standards, die versuchten, die in den 90er Jahren immer weiter auseinander driftenden Unix-Derivate an gemeinsame Schnittstellen zu binden, um Interoperabilität zu bewahren. Im Rahmen der Standardisierung wurde auch ein Standard bezüglich Thread-Erweiterungen definiert(POSIX.4a/IEEE 1003.1c-1994). Die POSIX-Threads sind eine verbreitete Threadbibliothek für die Programmierung unter Unix/Linux, es existiert allerdings auch eine Portierung für Windows, die pthreads-win32 Bibliothek. Aufgrund der nachgelagerten Entwicklung von Threads, werden diese nicht wie in Java direkt von der Programmiersprache C unterstützt, deshalb muss beim Arbeiten mit pthreads die Bibliothek <pthread.h> eingebunden werden und beim Compilieren gegen die Bibliothek lpthread gelinkt werden. Dieses Kapitel stützt sich im Wesentlichen auf [17].

4.1 Erzeugen von Pthreads

Um Pthreads benutzen zu können, müssen wir zunächst einmal einen Thread erzeugen. Dies geschieht mit der pthread_create Funktion.

```
int pthread_create(pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine)(void*),
        void *arg);
int pthread_join ( pthread_t thread, void **value_ptr);
int pthread_detach ( pthread_t thread);
```

Für C typisch dient der Rückgabewert einer Funktion als Statusmeldung, ob der Aufruf der Funktion erfolgreich war oder nicht, während Ergebnisse meist an Speicherstellen zurückgeschrieben werden, die als Parameter übergeben wurden. Im Erfolgsfall liefern alle Pthreads-Funktionen den Wert 0 zurück, ein Wert ungleich 0 gibt Auskunft über die Fehlerursache. Im ersten Parameter wird der **Thread_Identifier**, kurz **TID**, abgelegt. Bei dem zweiten Parameter handelt es sich um einen Zeiger auf eine Attributsstruktur, die das Verhalten des Threads näher spezifiziert. Setzen wir hier **NULL**, so startet der Thread mit Default-Werten. Das dritte Argument ist ein Pointer auf die Funktion, die als Thread ausgeführt wird, und **arg** ist ein Zeiger auf einen Speicherbereich, welcher der start_routine übergeben wird. Ein Pthread befindet sich nach seiner Erzeugung direkt im Status Runnable und muss nur noch darauf warten, den Prozessor zugeteilt zu bekommen, um mit der Arbeit zu beginnen. Ein mit pthread_create erzeugter Thread ist ein Kind-Thread des Threads, in dem die Funktion aufgerufen

wurde. Der Kind-Thread kann nun ein Ergebnis erzeugen. Ist der Vater-Thread an diesem Ergebnis interessiert, kann er mithilfe der Funktion pthread_join auf das Ende des Kind-Threads warten und sich dass Ergebnis an der Stelle value_ptr übergeben lassen. Ist hingegen keiner am Ergebnis des Kind-Threads interessiert, kann der Thread mit pthread_detach entkoppelt werden, was dazu führt, dass die Laufzeitumgebung alle Ressourcen des Kind-Threads nach dessen Beenden freigibt.

4.2 Realisierung gegenseitigen Ausschlußes

Da wir uns beim Arbeiten mit Threads im selben Adressraum befinden kann es beim Zugriff auf gemeinsam genutzte Variablen zu Race Conditions und daraus folgend zu Inkonsistenzen kommen. Um dies zu vermeiden gibt es das Konzept des gegenseitigen Ausschlußes. Mit dessen Hilfe kann der Programmierer sicherstellen, dass ein kritischer Bereich nie von mehr als einem Thread gleichzeitig benutzt werden kann. Vor Betreten eines kritischen Bereiches muss der Thread den Mutex erlangen. Solange der Thread den Mutex nicht wieder freigibt, was er nach Abarbeiten des kritischen Abschnitts tunlichst sollte, kann kein anderer Thread den kritischen Bereich betreten. Zunächst einmal muss eine Mutexvariable angelegt werden, die verwendet werden kann um kritische Bereiche zu schützen.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init( pthread_mutex_t *mutex,
  const pthread_mutexattr_t *attr );
int pthread_mutex_destroy( pthread_mutex_t *mutex );
```

Eine Mutexvariable ist vom opaquen (dass heißt, die genaue Implementierung bleibt uns verborgen) Typ pthread_mutex_t. Vor der ersten Verwendung des Mutex muss dieser initialisiert werden. Hierfür kann der Variable der Initialwert PTHREAD_MUTEX_INITIALIZER zugewiesen werden, mit der der Mutex Standardattribute erhält. Alternativ verwendet man die pthread_mutex_init Funktion und übergibt dieser die Adresse des zu initialisierenden Mutex und die Adresse eventueller Mutexattribute. Mit pthread_mutex_destroy kann ein Mutex zerstört werden. Der nun angelegte Mutex kann dazu verwendet werden, exklusiven Zugriff auf Variablen, die im gemeinsamen Speicher liegen, zu erhalten. Hierzu muss jeder Thread, der gemeinsame Variablen benutzen will, zuvor den Mutex erlangen. Zu beachten ist, dass der Programmierer selber sicherstellen muss, dass der Mutex immer vor Betreten eines kritischen Abschnitts erlangt wird, da nicht abgefangen werden kann, ob ein Zugriff auf die gemeinsamen Variablen am Mutex vorbei stattfindet.

```
int pthread_mutex_lock( pthread_mutex_t *mutex );
int pthread_mutex_trylock( pthread_mutex_t *mutex );
int pthread_mutex_unlock( pthread_mutex_t *mutex );
```

Mithilfe der Funktion pthread_mutex_lock versucht ein Thread den Mutex zu erlangen. Der Aufruf blockiert, sofern der Mutex noch von einem anderen

Thread gesperrt ist und noch nicht wieder freigegeben wurde. Der Aufruf von pthread_mutex_trylock kehrt hingegen sofort zurück.

4.3 Kooperation von Threads

Häufig muss ein Thread auf das Eintreten einer bestimmten Bedingung (i.e. einen bestimmten Wert einer Variable) warten. Da die Bedingung von mehreren Threads gleichzeitig verändert und ausgelesen werden kann, ist es notwendig, einen Zugriff auf an der Bedingung beteiligten Variablen durch einen Mutex zu schützen. Ein Thread hat nun folgendene Möglichkeit zu überprüfen, ob die Bedingung eingetreten ist. Der Thread erlangt zunächst den Mutex und überprüft die Bedingung. Ist die Bedingung erfüllt, kann der Thread mit seiner Arbeit fortfahren, ist die Bedingung allerdings nicht erfüllt, so muss der Thread den Mutex wieder freigeben und später noch mal nachsehen. Dieses Verfahren, auch Polling genannt, ist sehr ineffizient, sofern die Bedingung längere Zeit von keinem anderen Thread geändert wird. Eine Lösung für dieses Problem stellen Bedingungsvariablen dar. Ein Thread wird während seiner Wartezeit blockiert und aufgeweckt, sobald sich die Bedingungsvariable ändert. Eine Bedingungsvariable ist vom opaquen Datentyp pthread_cond_t, Initialisierung und Zerstörung erfolgen analog zu einer Mutexvariablen.

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init( pthread_cond_t *cond,
        const pthread_condattr_t *attr );
int pthread_cond_destroy( pthread_cond_t *cond );
```

Eine Thread. der nun auf das Eintreten einer Bedingung warten möchte, ruft hierzu die Funktion pthread_cond_wait auf. Die Funktion erwartet zwei Parameter, zum Einen die Bedingungsvariable und zum Anderen eine Mutexvariable.

```
int pthread_cond_wait( pthread_cond_t *cond,
        pthread_mutex_t *mutex );
int pthread_cond_timedwait( pthread_cond_t *cond,
        pthread_mutex_t *mutex,
        const struct timespec *abstime );
```

Der wartende Thread gibt nun implizit seinen Mutex frei, da sonst keiner die Daten ändern könnte und somit die Bedingung nie wahr werden könnte und legt sich danach schlafen. Sofern ein anderer Thread nun an der Bedingung beteiligte Variablen ändert, weckt er wartende Threads auf. Dies geschieht mithilfe der beiden folgenden Funktionen, die sich lediglich in der Anzahl der Threads, die aufgeweckt werden, unterscheiden. Das Wecken bedeutet, dass die wartenden Threads ihren Status auf Runnable ändern.

```
int pthread_cond_broadcast( pthread_cond_t *cond );
int pthread_cond_signal( pthread_cond_t *cond );
```

Die aufgeweckten Threads bemühen sich, nachdem sie geweckt wurden, wieder um den Mutex. Bis einer den Mutex allerdings erhält, kann sich die Bedingung schon wieder geändert haben, was zu einem typischen Konstrukt führt.

```
pthread_mutex_lock (&mutex);
while (!Bedingung)
pthread_cond_wait (&cond, &mutex);
//do something
pthread_mutex_unlock (&mutex);
```

4.4 Scheduling von Posix Threads

Sofern das Makro _POSIX_THREAD_PRIORITY_SCHEDULING in <unistd.h> definiert ist, ist es möglich darauf Einfluß zu nehmen, wie den Threads der Prozessor zugeteilt wird. Es kann also beeinflusst werden, wie die User-Threads auf Kernel Threads abgebildet werden. Einem Thread kann dann, wie in folgendem Beispiel-Code gezeigt, eine Priorität zugewiesen werden. Die Informationen zum Scheduling werden dem Thread in der bereits angesprochenen Attributsstruktur beim create-Aufruf mit übergeben. Zu beachten ist, dass der Wert von sched_priority durch ein Maximum und Minimum beschränkt ist. Die konkreten Werte können mit sched_get_priority_max und sched_get_priority_min erfragt werden.

```
struct sched_param param;
pthread_attr_t attr;
pthread_attr_init(&attr);
param.sched_priority = 5;
pthread_attr_setschedparam(&attr,&param);
```

Es gibt drei verschiedene Scheduling-Methoden, die verwendet werden können, um festzulegen wie lange ein Thread ausgeführt wird.

SCHED_FIFO (first in, first out): Für jede Prioritätsstufe existiert eine eigene Warteschlange, in die die Threads eingereiht werden. Ausgeführt wird immer der erste Thread aus der Warteschlange, der höchsten Prioritätsstufe. Blockiert ein Thread, so fügt er sich am Ende der Warteschlange wieder ein, wohingegen ein unterbrochener Thread vorne an die Warteschlange gesetzt wird. Ein Thread beliebiger Prioritätsstufe wird immer von einem ausführungsbereiten Thread höherer Prioritätsstufe unterbrochen.

SCHED_RR (round robin): Bei diesem Verfahren bekommt jeder Thread der Warteschlange für einen gewissen Zeitslot den Prozessor zur Verfügung gestellt. Nach Ablauf der Zeitspanne reiht sich der Thread wieder am Ende ein. Ansonsten ist dieses Scheduling-Verfahren ähnlich zu SCHED_FIFO.

SCHED_OTHER: Dies ist die dritte Möglichkeit um auf das Scheduling Einfluss zu nehmen. Hierfür gibt es keine Vorgaben vom POSIX-Standard. Das Verfahren ist implementierungsabhängig, womit es möglich ist, die Scheduling-Strategie an das verwendete Betriebssystem anzupaßen.

Mit folgender Funktion kann das Scheduling-Verfahren gesetzt werden:

```
int pthread_attr_setschedpolicy ( pthread_attr_t *attr,
        int schedpolicy)
```

Die wichtigsten Funktionen zur Erstellung paralleler Programme mithilfe der Pthreads wurden angesprochen. Weitere Informationen zur Programmierung mit POSIX-Threads finden sie zum Beispiel in [17].

5 Java-Threads

Threads sind ein integraler Bestandteil der Programmiersprache Java. Dies lässt sich beispielsweise daran erkennen, dass die Klasse Object Methoden besitzt, die nur der Synchronisation von Threads dienen, sowie an thread-spezifischen Schlüsselwörtern wie synchronized. Unterstützt das Betriebssystem keine Threads, so werden die Java Threads auf einen Betriebssystemprozess abgebildet. Die JVM ist in diesem Fall alleinig für das Scheduling der Threads und deren verzahnte Ausführung verantwortlich. Sofern das Betriebssystem allerdings Threads unterstützt, werden die Java-Threads auf Betriebssystemthreads 1:1 abgebildet.

> Die 1:1-Abbildung ermöglicht eine einfache Verteilung auf Mehrprozessorsystemen, doch mit dem Nachteil, dass das Betriebssystem in den Threads auch Bibliotheksaufrufe ausführen kann, zum Beispiel, um das Ein- und Ausgabesystem zu verwenden oder um grafische Ausgaben zu machen. Damit dies ohne Probleme funktioniert, müssen diese Bibliotheken jedoch Thread-sicher sein. Da hatten die Unix-Versionen jedoch diverse Probleme, insbesondere die grafische Standardbibliothek X11 und Motif waren lange nicht Thread-sicher. Um schwerwiegenden Problemen mit grafischen Oberflächen aus dem Weg zu gehen, haben die Entwickler daher auf eine native Multithreaded-Umgebung zunächst verzichtet [13].

Wie die Abbildung nun tatsächlich erfolgt, hängt also immer von der konkreten Implementierung der JVM ab. Für die Korrektheit des Programms macht es keinen Unterschied, wie die Threads abgebildet werden.

5.1 Erzeugung von Java-Threads

Ein Thread wird in Java dadurch erzeugt, dass von der Klasse Thread, die das Interface Runnable implementiert, geerbt wird.

```
public class Thread extends Object implements Runnable
```

Hat die Klasse bereits von einer anderen Klasse geerbt, entfällt diese Möglichkeit und die Klasse muss dass Interface Runnable direkt implementieren.

```
interface java.lang.Runnable{
public void run() {}
}
```

Dieses Interface hat nur eine Methode, die run() Methode, die den sequentiellen Befehlsstrom enthält, den der Thread abarbeitet. Ein direktes Aufrufen der run Methode kommt einem normalen Methodenaufruf gleich, startet aber den

Thread nicht, denn dies geschieht durch die Methode start(). Diese Methode wird von der Klasse Thread geerbt, im Falle, dass wir das Interface Runnable direkt implementieren, müssen wir unsere eigene Klasse noch in einen Thread mithilfe des Konstruktors Thread(Runnable target) umwandeln. Durch den Aufruf der Methode start() wird der Thread initialisiert und der Laufzeitumgebung bekannt gemacht.

```
static void join()
static void join(long timeout)
static void join(long timeout, int nanos)
```

Ein Thread wird immer innerhalb eines anderen Threads gestartet. Der Vater-Thread kann durch Aufruf der Join-Methode auf das Ende des Kind-Threads warten (vgl. hierzu 3.8). Dies stellt eine erste Form der Synchronisation dar.

5.2 Realisierung gegenseitigen Ausschlußes

Die Probleme bei Java-Threads sind natürlich die selben wie bei den POSIX Threads. Gemeinsam genutzte Variablen müssen vor konkurrierendem Zugriff geschützt werden. Da es sich bei Java um eine objektorientierte Sprache handelt, ist eine logische Schlussfolgerung, dass Objekte vor einem konkurrierenden Zugriff geschützt werden. Tatsächlich ist zu jedem Objekt implizit ein Monitor assoziiert. Ein Monitor ist einem Mutex ähnlich, er ist allerdings leistungsfähiger als ein Mutex, denn der Monitor vereinigt z.B die Warten/Benachrichtigen-Funktionalität in sich. Mithilfe des Monitors kann ein gegenseitiger Ausschluß realisiert werden. Kommen wir nun dazu, mit welchen Sprachmitteln wir dies realisieren können. Seit Java 1.0 gehört das Schlüsselwort **synchronized** zum Sprachumfang. Der Modifizierer synchronized kann sich auf eine ganze Methode oder auch nur auf einen Anweisungsblock beziehen. Ein Thread, der nun einen mit synchronized markierten Anweisungsblock betritt, setzt bei Objektfunktionen implizit den Monitor des this-Objekts und bei statischen Methoden den Monitor des dazugehörigen Class Objekts. Hiermit ist sichergestellt, dass sich nur ein Thread innerhalb des Anweisungsblocks befindet.

```
synchronized <Type> method() {}
synchronized( Object o ){}
```

Monitore sind reentrant. Dies bedeutet, dass ein , der den Monitor bereits besitzt, ihn nicht neu erlangen muss, wenn er eine weitere synchronized Methode aufruft.

5.3 Koordination zwischen Threads

Auch in Java können Threads auf das Eintreten einer Bedingung warten, beziehungsweise die Veränderung an einem Objekt kommunizieren. Die Methoden, die zur Verfügung stehen, sind recht ähnlich zu den Funktionen der Pthread Bibliothek, mit dem großen Unterschied, dass wir nicht mit Bedingungsvariablen arbeiten müssen.

```
public final void wait();
public final void wait(long millis);
public final void wait(long millis, int nanos);
public final void notify();
public final void notifyAll();
```

Zu beachten ist, dass alle diese Methoden nur innerhalb eines synchronisierten Abschnitts aufgerufen werden können, da sie ansonsten eine IllegalMonitorStateException auslösen. Dies hängt damit zusammen, dass ein Objekt o1 sich, bei einem Aufruf von o2.wait(), in die Warteschlange von o2 einreiht. Es muss also garantiert sein, dass o1 den Monitor auf o2 hat.

5.4 Scheduling von Java Threads

Die JVM bietet prioritätsbasiertes Scheduling nach dem Round-Robin Prinzip. Das bedeutet, jeder Thread bekommt eine bestimmte Zeitscheibe zugeteilt, nach deren Ablauf ihm vom Scheduler der Prozessor entzogen wird. Für jeden Thread kann eine Priorität mithilfe der Methode**final void** setPriority(**int** priority) angegeben werden. Der Wert liegt zwischen Thread.MIN_PRIORITY (1) und Thread.MAX_PRIORITY (10), standardmäßig bekommt ein neuer Thread die Priorität Thread.NORM_PRIORITY (5). Wie den Threads nun der Prozessor zugeteilt wird, soll an folgendem Zitat verdeutlicht werden. Das folgende Zitat aus [13] illustriert den konkreten Ablauf des Scheduling-Verfahrens.

> Das Betriebssystem (oder die JVM) nimmt die Threads immer entsprechend der Priorität aus der Warteschlange heraus (daher Prioritätswarteschlange). Ein Thread mit der Priorität N wird vor allen Threads mit der Wichtigkeit kleiner N, aber hinter denen der Priorität größer gleich N gesetzt. Ruft nun ein kooperativer Thread mit der Priorität N die Methode yield() auf, bekommt ein Thread mit der Priorität ¡= N auch eine Chance zur Ausführung.

Beim Vergeben von Prioritäten muss mit Vorsicht gearbeitet werden. Dies hat verschiedene Gründe. Threads hoher Priorität können Threads geringerer Priorität stark ausbremsen, was die Applikation dann insgesamt nicht schneller machen muss. Sofern Java-Threads auf Betriebssystem-Threads abgebildet werden, muss mam auch beachten, das Windows XP zum Beispiel nur 7 (vgl. hierzu [3]) Prioritätsstufen kennt und hier eventuell unerwünschte Effekte auftreten können.

In diesem Kapitel konnten nicht alle Möglichkeiten angesprochen werden, die Java zum Arbeiten mit Threads anbietet. Insbesondere das neue java.util.concurrent Package wurde nicht behandelt. Weitere Informationen hierzu findet man in verschiedenen Java Büchern, die Java ab der Version 1.6 behandeln.

6 OpenMP

OpenMP ist ein offener, herstellerübergreifender Standard zur Entwicklung paralleler Programme, die sich einen gemeinsamen Adressraum teilen. OpenMP

weist jedoch einige Unterschiede im Gegensatz zu den beiden bisher kennengelernten APIs, den Posix-Threads und den Java Threads auf. Während der Programmierer bisher Threads explizit erstellen, starten und beenden musste, bleibt ihm diese Arbeit nun weitestgehend erspart. Zur Spezifizierung paralleler Bereiche verwendet OpenMP Compilerdirektiven, deren großer Vorteil darin besteht, dass der ursprüngliche, sequentielle Code nicht verändert werden muss, sondern nur angereichert wird. Ein bisher sequentielles Programm kann in OpenMP durch wenige zusätzliche Zeilen Code zu einem parallelen Programm gemacht werden. Die Tatsache allerdings, dass das Handling der Threads bei OpenMP intern erfolgt, hat auch den Nachteil, dass der Programmierer wesentlich weniger Kontrolle hat als bisher und somit die Programmierung mit OpenMP nicht so flexibel wie mit Java/POSIX-Threads ist. Dies führt uns zu folgendem Schluss, dass alles was mit OpenMP möglich ist, und was das genau ist werden wir in den folgenden Abschnitten sehen, auch mit Java oder Posix Threads entwickelt werden kann, wohingegen der Umkehrschluss nicht immer möglich ist. Das verwendete parallele Programmiermodell ist das Fork/Join-Modell, beziehungsweise das Parbegin/Parend-Modell. Ein paralleler Programmabschnitt in OpenMP wird durch die Direktive # pragma omp parallel (Fork) eingeleitet und mit #pragma end parallel (Join) beendet. Zu Beginn eines so spezifizierten Abschnitts erzeugt der Master-Thread ein Team von Threads, welches den parallelen Bereich gemeinsam abarbeitet.

6.1 Hello World

Um einen ersten Eindruck eines OpenMP Programms zu gewinnen, betrachten wird das bekannte “Hello World”-Programm aus [6]. Sehr gut zu erkennen ist bereits das Verwenden der Compilerdirektiven.

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif

int main(int argc, char* argv[])
{
#ifdef _OPENMP
        printf("Anzahl Prozessoren: %d\n",omp_get_num_procs());
#pragma omp parallel
{
        printf("Thread %d von %d sagt \"Hallo Welt!\"\n",
                omp_get_thread_num(), omp_get_num_threads());
}
#else
        printf("OpenMP wird nicht unterstützt.\n");
#endif
        return 0;
```

}

Mit #pragma omp parallel wird zunächst einmal ein paralleler Abschnitt angelegt. Um die vorhandene Arbeit innerhalb dieses Abschnitts auf die Threads des abarbeitenden Teams aufzuteilen, stellt OpenMP drei Möglichkeiten, genauer gesagt drei Direktiven, zur Verfügung:

#pragma omp for: For-Schleifen zu parallelisieren ist das Hauptanwendungsfeld von OpenMP, dies wird ausführlich in Abschnitt 6.2 dargestellt

#pragma omp section: Mit dieser Direktive ist es möglich, ganze Codeblöcke auf die verschiedenen Threads aufzuteilen. Mehr dazu im Abschnitt 6.3

#pragma omp single: Ein, mit dieser Direktive gekennzeichneter, Anweisungsblock wird innerhalb eines parallelen Abschnitts nur genau einmal ausgeführt.

6.2 Parallele Schleifen mit OpenMP

Eine der wichtigsten Anwendungsfelder von OpenMP ist das Parallelisieren von Schleifen. Die parallele Abarbeitung von Schleifen funktioniert nach dem SPMD Pattern. Jeder Thread führt das selbe Programmfragment auf unterschiedlichen Daten aus. Im Falle einer For-Schleife mit 9000 Iterationen und 3 Threads kann jeder Thread 3000 Iterationen durchführen und die Abarbeitungsgeschwindigkeit verdreifacht sich. Die Aufteilung der Arbeit auf die Threads erledigt OpenMP implizit. Dies funktioniert natürlich nur, sofern die einzelnen Iterationen voneinander unabhängig sind. Insbesondere Vektoroperation (beispielsweise $y[i] = alpha * x[i] + z[i] * beta$), wie sie im High Performance Computing oder in Multimedia Anwendungen häufig vorkommen, sind meist unabhängig. Die Direktive **#pragma** omp parallel **for** ist lediglich eine syntaktische Abkürzung für den Fall, dass der parallele Block nur eine For-Schleife enthält. Abbildung 6 verdeutlicht die Anwendung dieser Direktive.

```
int y[10000],x[10000],z[10000];
int alpha=0.7, beta =0.3;
  // Arrays y,x,z mit Werten befüllen
  #pragma omp parallel for
  for (i=0;i<N;i++){
        y[i]= alpha * x[i] + z[i] * beta;
  }
```

Abbildung 6. Parallelisierung einer Vektoroperation

Inwieweit die Threads auf gemeinsamen Daten arbeiten, oder ob die Threads Variablen in ihren privaten Speicherbereich kopieren, kann durch Parameter angegeben werden.

```
private(list_of_variables)
```

Private Variablen werden mithilfe von private angegeben. Jede in list_of_variables angegebene Variable wird als uninitialisierte Variable auf den Laufzeitstack eines jeden Threads gelegt, der als einziger auf diese Variablen zugreifen kann.

```
shared(list_of_variables)
```

Das Gegenstück zu private ist shared. Alle hier angegebenen Variablen können von allen Threads gleichzeitig zugegriffen werden

Folgendes Beispielprogramm (zu finden in [7] oder [8]) soll die Verwendung verdeutlichen.

```
int npoints, iam, np, mypoints;
double *x;

int main() {
 scanf("\%d", &npoints);
 x = (double*) malloc(npoints *sizeof(double));
 initialize();
 #pragma omp parallel shared(x, npoints) private(iam,np,mypoints)
 {
 np = omp_get_num_threads();
 iam = omp_get_thread_num();
 mypoints = npoints / np;
 compute_subdomain(x, iam, mypoints);
 }
}
```

Jeder Thread besitzt drei private Variablen, seine eigene Threadnummer, die Anzahl an Threads die den parallelen Bereich abarbeiten, sowie die Anzahl der Punkte die der Thread abarbeiten muss. X und npoints müssen shared sein, da alle Threads das selbe Array, nur verschiedene Bereiche davon, abarbeiten wollen.

6.3 Parallele Abschnitte in OpenMP

Neben der Parallelisierung von Schleifen stellt OpenMP auch die Möglichkeit zur Verfügung verschiedene Codeblöcke parallel abzuarbeiten. Der Unterschied ist nun, dass wir nicht mehr das selbe Programm auf verschiedene Daten anwenden, sondern verschiedene Programme auf verschiedene Daten. Vorstellbar ist, dass in einem parallelen Bereich, ein Array x vertauscht wird, während im anderen parallelen Bereich der Mittelwert über ein Array y gebildet wird. Die konkrete Syntax ist wie folgt:

```
#pragma omp sections [Parameter [Parameter] ... ]
{
  #pragma omp section {Anweisungsblock}
  #pragma omp section {Anweisungsblock}
   .
```

```
    .
    .
}
```

6.4 Gegenseitiger Ausschluß in OpenMP

Die parallelen Bereiche in OpenMP werden von mehreren Threads gleichzeitig ausgeführt, was eventuell wieder gegenseitigen Ausschluß notwendig machen kann. Hierfür stehen die beiden folgenden Direktiven zur Verfügung:

```
#pragma omp critical
#pragma omp atomic
```

Ein mit critical gekennzeichneter Anweisungsblock kann immer nur von einem Thread gleichzeitig ausgeführt werden. Atomic ist etwas feingranularer, denn damit können einzelne Zuweisungen, als atomare Aktionen durchgeführt werden. Erinnern wir uns an das Problem, dass es sich bei i++ um keine atomare Aktion handelt, so leuchtet der Sinn dieser Direktive recht schnell ein. Weitere Informationen zu OpenMP gibts in [8] und [7]

7 Schluss

Die Programmierung im gemeinsamen Adressraum bringt spezielle Problem mit sich, für die allerdings erprobte Konzepte existieren. Mit der fortschreitenden technischen Entwicklung und der Tatsache, dass bereits nahezu jeder aktuelle Desktop-PC mindestens einen Dual-Core Prozessor besitzt, gewinnt diese Art parallele Programme zu entwickeln zunehmend an Wichtigkeit. So stellt heute jede moderne Programmiersprache, sei es Java oder C#, eine Schnittstelle bereit um mit Threads zu arbeiten. Ein Anwendungsentwickler muss sich momentan allerdings sehr viel mit dem Threadmanagment, der Synchronisation und Koordination beschäftigen, anstatt sich um die Funktionalität der Anwendung zu kümmern. Eine Verbesserung könnten hier neue Programmiersprachen oder Frameworks bieten, die den Programmier durch eine höhere Abstraktionsebene von diesen Aufgaben befreien.

A Technologische Entwicklung

Denken wir an die klassische von Neumann Architektur zurück so besteht diese aus den folgenden Komponenten, der ALU, der Control Unit, I/O Unit, einer Memory Unit, und einem Bus der diese Komponenten verbindet. Der von Neumann Rechner arbeitete einen sequentiellen Befehlsstrom ab und operierte dabei auf Werten aus dem Speicher. Verschiedene Entwicklungen wie die Erhöhung der Wortbreite (die Anzahl der Bits die gleichzeitig verarbeitet werden können), die Erfindung der Pipeline, sowie superskalarer Prozessoren steigerten durch Ausnutzung von Parallelität auf Instruktionsebene die Verarbeitungsgeschwindigkeit (mehr hierzu in und . Das entscheidende Merkmal der Softwareentwicklung zu diesem Zeitpunkt, und ein limitierender dazu, war dass dem Übersetzer nur ein einziger Arbeitsstrang zur Verfügung gestellt wurde, so dass der Hardware gar nicht mehr wie ein Arbeitsstrang zur Verfügung stand, der ausgeführt werden konnte. Kommen wir zunächst dazu, welche hardwareseitigen Möglichkeiten entwickelt wurden, um mehr als nur einen Arbeitsstrang gleichzeitig auszuführen. Ein erste wichtige Erweiterung stellte die Memory Management Unit dar, mit deren Hilfe das Konzept von Prozessen eingeführt wurde, denn mithilfe dieser MMU war es nun möglich den physikalischen Speicher in virtuelle Speicherbereiche aufzugliederen, und diese voreinander zu schützen. Eine weiterer wichtiger Schritt war die Einführung von logischen Prozessoren, hierzu gibt es auf dem Chip zwei identische Bereiche, in den der Prozessorzustand (hauptsächlich die Registerinhalte) abgelegt werden konnte. Beim Wechseln zwischen zwei Arbeitssträngen kann nun einfach umgeschalten werden, ohne alle notwendigen Informationen des einen Arbeitsstranges zunächst im Hauptspeicher zu sichern und dann die des anderen wieder neu zu laden. Ganz aktuell ist die Entwicklung immer mehr Kerne auf einem Prozessor zu platzieren, die sich den Cache teilen. Diese Entwicklung ist dadurch bedingt, dass eine weitere Erhöhung der Taktrate aufgrund physikalischer Grenzen nicht möglich ist, und deshalb andere Maßnahmen zur Leistungssteigerung ergriffen werden müssen. Mithilfe dieser Entwicklung kann nun wirkliche Parallelität der einzelnen Arbeitsstränge erreicht werden. Ein Überblick zu den angesprochenen Technologien findet sich in [1].

B Posix Threadstates

Pthreads können vier verschiedene Zustände einnehmen (vgl. hierzu [17]).

Ready: Ein Thread in diesem Zustand wartet lediglich darauf, vom Scheduler den Prozessor zugeteilt zu bekommen.

Running: Ein Thread im Zustand Running, ist im Besitz des Prozessors und arbeitet gerade.

Blocked: Ein Thread im Zustand Blocked wartet auf Betriebsmittel, auf das Ende eines anderen Threads, einen Mutex oder eine Bedingungsvariable. Sobald dem Thread das Betriebsmittel zugeteilt wurde und alle Ressourcen vorhanden sind, so dass der Thread weiterarbeiten kann, wechselt er in den Zustand Ready.

Terminated: Ein Thread im Zustand Terminated hat sich selbst beendet, bzw seine Aufgabe erledigt oder wurde von außen abgebrochen.

C Java Threadstates

Die Zustände die ein Java Thread annehmen kann sind in der Enumeration java.lang Enum Thread.State definiert (vgl. hierzu [9]). **New:** Ein Thread in diesem Zustand wurde neu erzeugt und wartet jetzt lediglich darauf, vom Scheduler den Prozessor zugeteilt zu bekommen.

Runnable: Ein Thread im Zustand Running, ist im Besitz des Prozessors und arbeitet gerade.

Blocked: Ein Thread der auf den auf einen Monitor-Lock oder Betriebsmittel warten muss befindet sich in diesem Zustand.

Waiting: Ein Thread befindet sich nach Aufruf der Methode Thread.join() oder der Methode Object.wait() im Zustand **Waiting**.

Timed_Waiting: Diesen Zustand nimmt der Thread ein, wenn er nur eine bestimmte Zeit, durch Angabe eines Timeouts, wartet oder auch nach Aufruf der Methode Thread.sleep(**long** millis).

Terminated: Ein Thread im Zustand Terminated hat seine Aufgabe erledigt.

Literatur

[1] U. Brinkschulte and T. Ungerer. *Mikrocontroller und Mikroprozessoren (eXamen.press)*. Springer, April 2007.

[2] M. Cooperation. Creating Named Shared Memory. Website, 2008. Available online at `http://msdn.microsoft.com/en-us/library/aa366551(VS.85).aspx`; visited on June 25th 2008.

[3] M. Cooperation. Scheduling priorities. Website, 2008. Available online at `http://msdn.microsoft.com/en-us/library/ms685100.aspx`; visited on June 25th 2008.

[4] T. O. Group. Ieee std 1003.1, 2004 edition. Website, 2004. Available online at `http://www.unix.org/version3/ieee_std.html`; visited on June 25th 2008.

[5] B. B. Hall. Shared Memory Segments in unix. Website, 1997. Available online at `http://www.ecst.csuchico.edu/~beej/guide/ipc/shmem.html`; visited on June 25th 2008.

[6] R. Hoffmann, Simon und Lienhart. *OpenMP*. Springer Verlag, 2008.

[7] T. Rauber and G. Rünger. *Parallele und verteilte Programmierung - Architektur, Programmierung, Algorithmen*. Springer, 2000.

[8] T. Rauber and G. Rünger. *Multicore: Parallele Programmierung*. Springer Verlag, (Informatik Im Fokus) , 2007.

[9] I. Sun Microsystems. Java api, enum thread.state. Website, 2004. Available online at `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.State.html`; visited on June 28th 2008.

[10] I. Sun Microsystems. Starvation and Livelock. Website, 2008. Available online at `http://java.sun.com/docs/books/tutorial/essential/concurrency/starvelive.html`; visited on June 25th 2008.

[11] T-Systems. High Performance Computing. Website, 2008. Available online at `http://www.t-systems.de/tsi/de/20050/Startseite/OeffentlicherSektor/ForschungLehre/HighPerformanceComputing`; visited on June 25th 2008.
[12] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
[13] C. Ullenboom. *Java ist auch eine Insel*. Galileo Computing, Bonn, 6., aktualisierte und erweiterte auflage edition, 2007.
[14] H. und M.Sommer. *Einführung in die Informatik, 6. Auflage*. Oldenburg Wissenschaftsverlag GmbH, 2004.
[15] D. F. und Ulrike Böttcher. *Java 6 Programmierhandbuch*. entwickler.press, 2007.
[16] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill, Inc., New York, NY, USA, 1996.
[17] M. Zahn. *Unix-Netzwerkprogrammierung mit Threads, Sockets und SSL (X.systems.press)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

Programmierung mit verteiltem Adressraum

Thomas Eisenbarth

Universität Augsburg
thomas.eisenbarth@student.uni-augsburg.de

Zusammenfassung Programmierung mit verteiltem Adressraum ist im Bereich von Hochleistungscomputern eine notwendige Grundlage um hohe Rechenleistung zu erreichen. In dieser Arbeit werden neber einer grundlegenden Einführung in die Thematik die Vor- und Nachteile der Programmierung mit verteiltem Adressraum betrachtet. Ferner werden konkrete Muster für die Implementierung sowie der Standard MPI und das Softwarepaket PVM vorgestellt.

A Einleitung und Motivation

Seit Jahren bzw. Jahrzehnten unterstützen gängige Desktop- wie auch Server-Betriebssysteme sogenanntes Multitasking. Quasi-parallele Bearbeitung von Programmen auf Computern ist heutzutage gängig. Genau genommen war bereits MS-DOS 4.0 ein Multitasking-Betriebssystem. Es arbeitete mit einer präemptiven Methode, die auch in heutigen Systemen noch vorherrschend ist[11]. Das Ziel von Multitasking ist, eine quasi-parallele Bearbeitung mehrerer Programme zu ermöglichen. Dabei wird die Prozessorzeit in Zeitschlitze eingeteilt, die dann mit kurzer Frequenz an die Ressourcen-nachfragenden Prozesse zugeteilt wird.

Ein zweiter Bereich der parallelen Programmierung hinsichtlich der eingesetzten Hardware hat sich ebenfalls weiterentwickelt: Die aus dem Server-Bereich bekannte Entwicklung hin zu Mehrprozessor-Systemen setzt seit geraumer Zeit auch bei Arbeitsplatzrechnern ein. Trotz dieser rasanten Entwicklung der Verarbeitungsgeschwindigkeiten existiert eine praktisch nicht zu deckende Nachfrage an Rechenkapazität: Um aktuelle Fragen der Forschung aus den Bereichen der Materialwissenschaften, Biologie oder Plasmawissenschaften zu simulieren und zu untersuchen, ist der Bedarf an Rechenzeit prinzipiell nicht zu befriedigen.

Obwohl das mooresche Gesetz heute noch Gültigkeit besitzt, wonach sich die Anzahl der Schaltkreiskomponenten auf Prozessoren etwa alle zwei Jahre verdoppelt, kann man komplexe Anforderungen bezüglich der Berechnung umfangreicher wissenschaftlicher Probleme heute nicht mehr in einzelnen Computern beherrschen: Die Liste der schnellsten Rechner der Welt besteht überwiegend aus Systemen, die aus vielen Computern zusammengeschalten sind[15]. Der Supercomputer „JUGENE" am Jülicher Supercomputing Centre (JSC) besteht beispielsweise aus 65536 einzelnen PowerPC-CPUs, die zusammengeschaltet eine Leistung von 223 Teraflops bereitstellen. Zum Vergleich: Eine heute handelsübliche CPU mit 4 Cores kommt auf etwa 25 Gigaflops.

Um ein Zusammenarbeiten einzelner Computer zu einem großen System zu realisieren, benötigt man zusätzlich zu einem Netzwerk definierte Kommunikationsstrategien zwischen den einzelnen Systemen für eine Abstimmung der zu erledigenden Aufgaben. Diese Strategien sowie die Programmierung derartiger verteilter Programme werden in dieser Arbeit behandelt: In Kapitel 2 wird verteilter Adressraum definiert und von gemeinsamem Adressraum abgegrenzt. Im Anschluss werden allgemeine Herausforderungen von paralleler Programmierung betrachtet - speziell unter Berücksichtigung von verteiltem Adressraum. In Kapitel vier werden Vor- und Nachteile von Programmierung mit verteiltem Adressraum untersucht. Anschließenden werden Kommunikationsstrategien für die Programmierung mit verteiltem Adressraum vorgestellt. Das Kapitel sechs umfasst die verschiedenen Versionen des MPI-Standards, konkrete Implementierungen dessen sowie das Softwarepaket PVM. Das siebte und letzte Kapitel bildet als Abschluss der Arbeit einen Ausblick auf das behandelte Thema.

B Verteilter Adressraum

Unter einem Adressraum versteht man allgemein einen reproduzierbar eindeutig identifizierbaren Bereich einer Menge. In der Informatik sind verschiedenste Formen von Adressräumen bekannt. Für das Erstellen von Computerprogrammen — worum es ja letztendlich gehen soll — sind wiederum ebenfalls mehrere Adressräume bekannt: Die Festplatte dient dem Programm als persistente Speicherumgebung. Neber den Programmdaten, die die Anweisungen an den Computer bereitstellen, werden dort auch Benutzerdaten gespeichert. Der Hauptspeicher (Random Access Memory, RAM) des Rechners stellt die Laufzeitumgebung für Programme bereit. Dort werden die Maschinenbefehle zur Ausführung von der Festplatte hingeladen. Ist das Programm im ausführenden Zustand, kann es Speicher im RAM allokieren, also für sich exklusiv belegen.

Bei Programmen, die in einem einzigen Prozess und einem Thread ablaufen, also quasi nicht parallelisiert sind, gestaltet sich die Steuerung des Programmablaufs sequentiell. Es ist also keine Kommunikation zwischen möglicherweise parallel ablaufenden Programmteilen notwendig. Anders jedoch, wenn mehrere Prozesse[1] oder mehrere Threads[2] verwendet werden: Werden mehrere Threads in einem Prozess ausgeführt, können diese beispielsweise durch gemeinsame Variablen miteinander kommunizieren. Weil Prozesse aus Sicherheitsgründen vom Betriebssystem voneinander abgegrenzt werden, verfällt die Möglichkeit der Steuerung paralleler Abläufe durch gemeinsamen Speicher. Gleiches gilt bei der Programmierung mit verteiltem Adressraum, da in Betriebssystemen keine Speicherzugriffsmöglichkeiten für verteilte Rechner integriert sind. Eine grobe Einordnung der logischen und physikalischen Speicherverteilgung findet sich in Ab-

[1] Silberschatz, et al.[14] definiert einen Prozess als ein Programm, das in den Speicher geladen ist und ausgeführt wird.

[2] Threads werden häufig als leichtgewichtige Prozesse bezeichnet, da die Trennung bzgl. Adressraum/Speicher, Sockets, etc. zwischen mehreren Threads weniger strikt ist, als bei Prozessen.

bildung 1. Die Abbildung basiert auf einer Abbildung aus der Dissertation von Mairandres[9, Abb. 1.1, S. 2]. Die physikalische Komponente ist als Speicher bezeichnet: Dieser kann gemeinsam sein, auch wenn der physikalische Speicher verteilt ist.

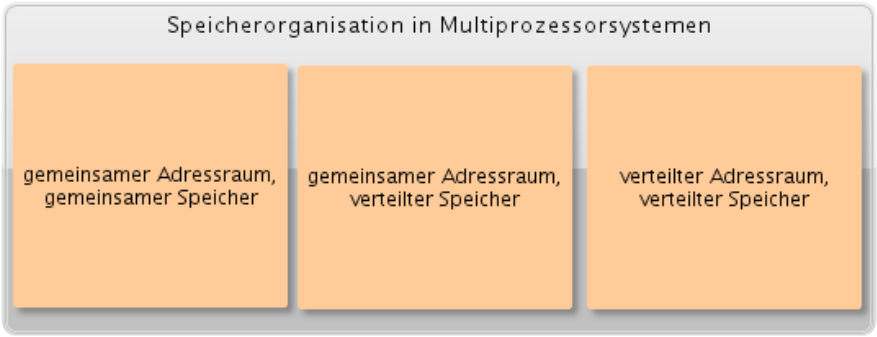

Abbildung 1. Grobe Gliederung der Speicherarchitekturen in Multiprozessorsystemen.

Ferner ist das Betriebssystem für die Vergabe von Rechenzeit an Prozesse und Threads zuständig. Da eine Reihenfolge oder sonstige Vorausbestimmung des so genannten Schedulers, der die Einteilung vornimmt, nicht möglich ist, müssen die parallel ablaufenden Teile synchronisiert werden. Dies geschieht beispielsweise mittels eines ,,Semaphor": Diese (für alle beteiligten Threads sichtbare Variable) ist die Kontrollvariable. Vor dem Eintritt in den kritischen Bereich[3] wird das Semaphor abgefragt. Ist es belegt, greift aktuell ein anderer, parallel laufender Thread auf den kritischen Bereich zu. Der kritische Bereich darf also zunächst nicht ausgeführt werden - das aufrufende Programm frägt das Semaphor entweder periodisch ab (,,POLL") oder es registriert sich beim Semaphor mit einer eindeutigen Kennzeichnung für die spätere Ausführung (,,PUSH"). Bei der Verwendung von verteiltem Adressraum, also beispielsweise physikalisch getrennten Computern, hat jede Recheneinheit voneinander unabhängige, meist physikalisch getrennte Arbeitsspeicher. Das Konzept von Semaphor oder beliebigen anderen Varianten, die auf einem zentralen, für alle Threads bzw. Prozesse gültigen Speicherplatz beruhen, haben bei verteiltem Adressraum keinerlei Grundlage und sind somit nicht anwendbar. Dennoch haben sich Methoden etabliert, die eine parallele Programmierung auf verteiltem Adressraum ermöglichen. Diese verschiedenen Varianten sowie konkrete Implementierungen werden

[3] Als kritischer Bereich wird der Abschnitt des Quellcodes bezeichnet, der exklusiven Zugriff auf bestimmte Variablen erfordert und nicht von anderen Prozessen und/oder Threads unterbrochen werden darf. [14].

in dieser Arbeit diskutiert. Im folgenden Kapitel werden zunächst die Grundlagen zur Programmierung mit verteiltem Adressraum erörtert und danach auf die resultierenden Herausforderungen eingegangen.

C Herausforderungen

Durch die Entwicklung neuer Generationen von Programmiersprachen hat sich die Programmierung mit diesen stetig vereinfacht: Nach den Lochkarten etablierten sich hardwarenahe Assemblersprachen, daraufhin entstanden als eine erste Abstraktion Sprachen wie BASIC oder C. Als nächsten Evolutionsschritt interpretiert man die Entwicklung hin zu objektorientierten Sprachen, worunter beispielsweise C++ und Java eingeordnet werden. Eines der zentralen Ziele eines jeden Schritts dieser Evolution war die Reduktion der Komplexität der Sprachen, um die Technologien besser handhabbar zu machen und potentielle Fehlerquellen auszuschließen. Betrachten wir eine weitere Entwicklung der letzten Jahre: Die Nutzung von mehr als einer CPU bzw. die Nutzung mehrerer cores auf einer CPU: Im Server-Bereich ist der Einsatz von SMP[4] schon vor Jahren üblich. Beispielsweise muss ein Web-Server parallel oder zumindest nahezu parallel mitunter viele hundert Anfragen beantworten. Die derzeitige Entwicklung des Prozessormarktes lässt erwarten, dass es eine Entwicklung hin zur Verwendung von sehr vielen Cores auch in handelsüblichen PCs geben wird: Advanced Micro Devices, Inc. (AMD) hat für 2009 eine 6-core-CPU und für 2010 einen Prozessor mit 12 Cores angekündigt[1]. Diese Entwicklung und die Nachfrage nach immer mehr Rechenzeit für die Simulation naturwissenschaftlicher Experimente führt zu einer komplizierteren Programmierung. Die Parallelisierung von Programmen erfordert nicht nur zusätzlichen Overhead in Form von Programmcode, sondern auch die prinzipielle Überlegung: Wann wird ein Programm durch Parallelisierung überhaupt schneller? Dieses Problem wird vom Amdahlschen Gesetz im folgenden Abschnitt behandelt:

C.1 Amdahlsches Gesetz

Das Amdahlsche Gesetz ist ein Modell zur Berechnung der möglichen Beschleunigung von Programmen durch den Einsatz von mehreren Prozessoren[13, S. 313],[12]. Das Modell ist nicht uneingeschränkt geeignet, um auch parallele Programmierung mit verteilten Adressräumen zu betrachten, da bei letzteren mehr Overhead mit eingeplant werden muss, als dies bei einem gemeinsamen Adressraum der Fall ist. Der Kommunikationsaufwand und der daraus resultierende zeitliche Overhead zwischen verteilten Recheneinheiten ist um ein Vielfaches höher als bei Systemen mit gemeinsamem Speicher. Dennoch eignet sich das Amdahlsche Gesetz um zu sehen, dass die Parallelisierung von Programmen

[4] SMP (engl. Symmetrical Multiprocessing) bezeichnet ein Computersystem mit geteiltem Speicher („shared-memory") bei dem alle CPUs identisch sind und jede CPU auf alle Speicherstellen zugreifen darf.[7]

nicht immer zu Performance-Verbesserung führen muss: Für ein gegebenes Programm sei p der prozentuale Anteil an parallelisierbaren Programmteilen, also solchen, die zerlegbar und parallelen ausführbar sind. Der erzielbare Geschwindigkeitsgewinn s beim Einsatz von n Prozessoren ist nach dem Amdahlschen Gesetz:

$$s = \frac{1}{(1-p) + \frac{p}{n}} \tag{1}$$

Wie man sehen kann würde der zweite Summand im Nenner beim Einsatz von beliebig vielen Prozessoren gegen Null gehen und der Anteil nicht parallelisierbaren Codes $1 - p$ wäre entscheidend für den Geschwindigkeitsgewinn s. Bei voll parallelisierbarem Code ($p = 1$) stiege s linear mit der Anzahl der Prozessoren n[13, vgl. S. 313f.]. Dieser Fall ist jedoch nicht möglich: Allein die notwendigen Kommunikationsanweisungen um die Synchronisierung der parallel ablaufenden Prozesse sind keine Operatoren, die man zerlegen und parallel ausführen kann. Generell lässt sich folgendes festhalten: Je umfangreicher die Teile des Programmes sind, die parallel ablaufen können, desto höher ist der erzielbare Geschwindigkeitsgewinn. Für verteilte Adressräume gilt dies ebenfalls, mit einer weiteren Bedingung: Die parallel auszuführenden Berechnungen sollten möglichst umfangreich hinsichtlich der Rechenzeit sein: Je kleiner die Teilberechnungen, desto häufiger müssen die Prozesse miteinander kommunizieren. Dies ist bei verteilten Adressräumen durch die hohe Latenz[5] der Netzwerkverbindungen im Vergleich zu gemeinsamen Adressräumen wie dem Arbeitsspeicher besonders teuer. Auf diese Einflussgrößen wie die Latenz wird im nächsten Kapitel eingegangen.

C.2 Räumliche Distanz

Bei der Untersuchtung der Programmierung im verteilten Adressraum spielt eine grundlegende und entscheidende Rolle, auf welchem Wege die Recheneinheiten miteinander kommunizieren. Viele Möglichkeiten zur Rechnerkommunikation sind im Laufe der Zeit entstanden. Zur Klassifizierung und Einordnung der Leistungsfähigkeit von Netzwerken, über die die Kommunikation stattfindet, existieren folgende Parameter: Die Latenz zwischen den Recheneinheiten muss so gering wie möglich sein, denn als Maßstab für diese Latenz wird die Zugriffszeit auf den Hauptspeicher in einem PC gesetzt, da ohne die Nutzung des Netzwerks und damit des verteilten Adressraums der Arbeitsspeicher der zentrale Zugriffspunkt ist. Die Datenübertragungsrate oder Bandbreite beschreibt die maximale Menge an fehlerfrei übertragbaren Daten in einem Netzwerk und wird in Kilo-, Mega- oder Gigabit angegeben.

[5] Latenz in Netzwerken beschreibt so genannte Roundtrip-Zeiten, genauer in welchen Zeitfenstern ein Packet von der Quelle am Ziel und ein Packet vom Ziel wieder bei der Quelle ist[2].

C.3 Kommunikation zwischen Rechenknoten

Ein Rechnersystem mit verteiltem Adressraum besteht aus Rechenknoten und einem Verbindungsnetzwerk, das die Knoten miteinander verbindet. Obwohl man sich als Entwickler bei der Programmierung mit verteiltem Adressraum nicht unbedingt um die Vernetzung der Rechner kümmern muss, da diese wohl in den meisten Fällen bereitstehen, sollten Grundlagen zu der möglichen Kommunikationsstruktur bekannt sein. Deshalb seien im Folgenden zunächst mögliche abstrakte Formen für diese Netzwerkstrukturen aufgezeigt. Anschließend werden konkrete und in der Realität eingesetzte Verfahren vorgestellt.

- **Punkt-zu-Punkt-Verbindungen** dienen dem Aufbau von Direktverbindungen zwischen einem Knoten und genau einem weiteren (1:1-Verbindung) oder mehreren Knoten (1:n-Verbindung). Die Realisierung dieser Konnektivität wird dabei jeweils durch physikalische Verbindungen umgesetzt. Punkt-zu-Punkt-Verbindungen waren vor allem in früheren Multicomputern eingesetzt.
- Durch die Verwendung von **Hardware-Routern** wird die Kommunikation mit anderen Knoten an diese Einheit ausgelagert. Auch sind bei diesem Modell weniger physikalische Leitungen erforderlich, da die Router entlang von Pfaden kommunizieren können.
- Als ein **Cluster** wird ein Rechensystem bezeichnet, das aus vielen Rechnern besteht, jedoch in seiner Gesamtheit als ein einziger Rechner angesprochen werden kann. Die einzelnen Komponenten des Clusters sind anonym und untereinander austauschbar. Parallel Virtual Machine (PVM) ist eine Implementierung dieser Idee, mit der eine Menge an heterogenen Rechnern zu einem virtuellen zusammengeführt werden können. PVM wird in Kapitel 6 detailiert betrachtet.
- Das regelmäßig als Netzwerk benannte **„Local Area Network" (LAN)** besteht aus mehreren eigenständigen Workstations, weswegen es auch als NOW („Network Of Workstations") oder COW („Cluster Of Workstations") bekannt ist[13, S. 24.f]. Es ist für Verbindungsnetzwerke in Parallelrechnern nicht zwangsläufig die effizienteste Vernetzungsmethode: Speziell ausgelegte Vernetzungsmethoden erreichen normalerweise eine höhere Effizienz als LANs: Der bekannte OSI-Stack, der sich teilweise im heutzutage verwendeten TCP/IP-Stack wiederfindet, abstrahiert absichtlich jede Schicht voneinander, um Interoperabilität und Auswechselbarkeit der Schichten sicherzustellen. In fein abgegrenzten Umgebungen mit hochspeziellem Einsatzzweck wie beim Supercomputing kann man darauf zu Gunsten höherer Performanz verzichten.

Eine konkrete Implementierung einer Vernetzungstechnik ist **Myrinet**[10], das vom Unternehmen Myricom für NOWs entworfen wurde. Der Vorteil von Myrinet gegenüber bekannten Vernetzungstechniken wie Ethernet ist der höhere Datendurchsatz und geringere Latenzzeiten, die von Myrinet durch optimierte Protokolle und daraus resultierenden geringerem Overhead erreicht werden. Weitere Vorteile sind eingebaute Vorkehrungen zur Behandlung von Fehlern und

Flusskontrolle im Netzwerk. Der Datendurchsatz war in der ersten Generation bei 512MBit/s brutto. Um die tatsächliche Datenrate („Netto-Datenrate") zu erhalten muss man Verwaltungs-Overhead abziehen, der beispielsweise durch Zugriffsprotokolle auf das physikalische Medium entsteht. Die neuesten Adapter bieten einen Durchsatz von 10 GBit/s. Myrinet war im Supercomputing eine sehr verbreitete Vernetzungstechnik: Nach Aussage des Herstellers setzten in der im Juni 2005 veröffentlichten TOP500-Supercomputers-Liste 141 der 500 Rechner Myrinet zur Vernetzung ein.[10] In der aktuellsten TOP500-Liste vom Juni 2008 setzten lediglich noch neun Großrechner auf Myrinet und drei auf Myrinet10G.

Eine andere, etwas bekanntere Technik ist **Infiniband**: Es benutzt einen seriellen Bus zur Übertragung der Daten und ist dabei extrem günstig hinsichtlich der Latenz. Der Bus kann in mehrere Kanäle unterteilt werden. Über einen solchen Kanal fließen theoretisch 2,5GBit/s und 5 GBit/s bei Nutzung von Double Data Rate (DDR). Durch die (transparente) Kopplung mehrerer Kanäle kann eine höhere Geschwindigkeit erreicht werden: Durch Zusammenschalten von 4 Kanälen erreicht man maximal 20 GBit/s bei Nutzung von DDR. Infiniband ist laut der TOP500-Liste deutlich häufiger eingesetzt als Myrinet: 73 der 500 Supercomputer setzen auf diese Technik und diverse weitere auf Ableger von Infiniband.

Die wohl mit Abstand bekannteste und verbreiteste Vernetzungstechnik ist **Ethernet**: Man findet es im (kabelgebundenen) Consumer-Bereich und auch bei der Vernetzung von Bürogebäuden ist Ethernet über Kupferkabel heutzutage gängig. Glasfaser wird in Core-Netzen oder Rechenzentren als physikalischer Träger benutzt, das Protokoll auf OSI-Layer 2 ist dabei immer Ethernet, das in IEEE 802.3 spezifiert ist. Übliche Bandbreiten bei Ethernet sind heute 100MBit/s und 1GBit/s. Die nächste Stufe mit 10GBit/s wird aufgrund der besseren Eigenschaften bzgl. Übertragungsqualität hauptsächlich über Glasfaserverbindungen eingesetzt werden. Derzeit wird Ethernet bei der Vernetzung von Supercomputern häufig eingesetzt: 284 der Supercomputer setzen Ethernet ein.

Es existieren neber den vorgestellten Techniken eine Vielzahl weiterer sogenannter Interconnects zur Vernetzung von Recheneinheiten: Quadrics, Multistage crossbar, Crossbar, NUMAlink und viele andere mehr.[8]

Die grundlegende Funktionsweise aller dieser Netzwerksysteme ist trotz verschiedener Standards und unterschiedlicher Implementierungen identisch.

D Vor- und Nachteile der Programmierung mit verteiltem Adressraum

Zunächst zu den Nachteilen bei der Programmierung mit verteiltem Adressraum: Bei nicht-parallelen Programmen entfallen die Programm-Anweisungen, die für die Synchronisation der parallel laufenden Anwendungsteile notwendig sind. Diese sind zwar sowohl bei gemeinsamen als auch bei verteiltem Adressraum notwendig. Allerdings sind die Bibliotheken, die diese Funktionalität bereitstellen bei verteiltem Adressraum umfangreicher. So muss bei gemeinsamen Adressraum

nicht beachtet werden, ob die Datenrepräsentation möglicherweise geändert werden muss, schließlich ist der Quell- ja auch der Zielrechner. Bei verteiltem Adressraum können die Datenrepräsentation durch abweichende Rechnerarchitekturen jedoch unterschiedlich sein, und in den Programm-Bibliotheken zusätzliche Aufgaben notwendig machen. Der Programmierer benötigt mehr Erfahrung bei der Entwicklung von parallelen Programmen und muss zudem komplexe Bibliotheken handhaben.

Eine weitere Notwendigkeit bei verteiltem Adressraum ist die Voraussetzung, dass alle beteiligten Rechnerknoten im Verbund mittels einer Vernetzung über eine Kommunikationsbasis verfügen. Diese ist in zweierlei hinsicht teuer. Zum einen fiskalisch: Es werden spezielle Netzwerk-Adapter, Kabel, Switches verwendet, die durch die geringe Verbreitung teuer sind. Zum anderen ist die Vernetzung teuer im Hinblick auf die Performance, die Rechnerverbunde mit verteilten Adressräumen erreichen. Im Vergleich zu Rechnern mit einem gemeinsamen Adressraum ist der verteilte Adressraum um bis zu Faktor 100 langsamer[13, S. 24].

Trotz dieser Nachteile macht es Sinn auf verteilte Adressräume zu setzen: Die erzielbare Geschwindigkeit ist auf einzelnen Computern begrenzt und wird dies — beispielsweise durch physikalische Grenzen der Materialien — auch bleiben. Durch den Einsatz vieler Rechner in einem Verbund kann man also die gesamt zur Verfügung gestellte Performance erhöhen. Auch hinsichtlich der Ausfallsicherheit haben verteilte Adressräume Vorteile gegenüber einem gemeinsamen Adressraum: Bei letzterem existiert durch den gemeinsamen Adressraum einsogenannter *Single Point of Failure (SPOF)*, also eine Komponente, bei deren Ausfall das gesamte System nicht mehr funktioniert. Bei verteilten Adressräumen steigt die Ausfallsicherheit, denn bei Wegfall eines Rechners können bei Einsatz geeigneter Maßnahmen die verbleibenden weiter arbeiten. Ein weiterer Vorteil liegt in der besseren Skalierbarkeit des Rechnerverbundes bei steigender Komplexität der zu lösenden Aufgaben. Durch Hinzunehmen von Rechnern kann die Performance im Vergleich zu Rechnern mit einem gemeinsamen Adressraum recht leicht erhöht werden, da die gesamte Anzahl an Prozessoren bei gemeinsamen Adressräumen limitiert sind. Bei verteilten Adressäumen ist die Anzahl der Prozessoren nicht beschränkt. Es muss lediglich die Vernetzung der Systeme sichergestellt sein.

Es existieren also sowohl Vor- als auch Nachteile bei der Programmierung mit verteiltem Adressraum. In gewissen Situationen müssen die Nachteile jedoch in Kauf genommen werden. Für High Performance Computing ist die Verwendung der Techniken für verteilte Adressräume unabdingbar, da sonst die Kapazität und Rechengeschwindigkeit nicht ausreichend ist.

E Kommunikationsanweisungen verteilter Programme

Bei der Programmierung mit verteiltem Adressraum sind die beteiligten Rechnersysteme auf explizite Kommunikationsanweisungen angewiesen. Diese geschehen durch Austausch von Nachrichten mittels in den vorherigen Kapitel vorge-

stellten Vernetzungstechniken. Nachrichtenoperationen treten dabei immer als Paar auf: Das Quellsystem ruft eine Methode zum Versenden der Nachricht auf. Entsprechend muss das Zielsystem eine Operation zum Empfangen der Nachricht aufrufen. Bei dieser Kommunikation haben sich eine Reihe von Patterns herausgebildet, die für die Programmierung verwendet werden[13, S. 151ff.]. Diese werden im Folgenden ausführlich vorgestellt. Im Anschluss folgt die konkreten Spezifikationen MPI und das Softwarepaket PVM. Alle beteiligten Prozesse müssen global - also jedem anderen Prozess - bekannt sein. Es sei n die gesamte Anzahl der registrierten, verteilten Prozesse und P_i ein eindeutig identifizierbarer Prozess mit der ID i.

- **Einzeltransfer**: Bei diesem Kommunikationsmuster sind lediglich ein Sender mit der Prozess-ID i sowie ein Empfänger $j \neq i$ beteiligt. Der Sender legt die zu verschickende Nachricht in einen Ausgangspuffer. Der Empfänger ruft eine entsprechende Empfangsmethode unter Angabe eines Empfangspuffers auf. Die Identifikation des Empfängers geschieht über die endeutige ID j. Einzeltransfers bilden die Grundlage aller weiteren Pattern. Beim Einzeltransfer wird lediglich eine einzige Nachricht versendet.
- **Einzelbroadcast**: Ein einzelner Knoten i (die ,,Quelle" oder ,,Wurzel") sendet eine Nachricht an alle registrierten Prozesse $P_{o..i}$. Der Sender ist gleichzeitig Empfänger. Der Inhalt der Nachricht ist für alle Empfänger identisch. Der Sendeknoten bestimmt den Sendepuffer mit der beinhalteten Nachricht. Alle Empfangknoten definieren den Empfangspuffer, worin die Nachricht abgespeichert wird. Bei der Einzelbroadcast-Operation werden n Nachrichten versendet.
- **Einzel-Akkumulation**: Bei einer Einzel-Akkumulation schicken alle Prozesse eine Nachricht an einen einzelnen Prozess P_i, der als *Wurzel* bezeichnet wird. Dabei werden alle Nachrichten mit einer sogenannten Reduktionsoperation bitweise miteinander verknüpft. Als Ergebnis erhält der Wurzel-Prozess nur eine Nachricht. Eine mögliche Reduktionsoperation ist beispielsweise die Addition. Es werden insgesamt n Nachrichten verschickt.
- **Gather**: Die Gather-Operation verhält sich wie die Einzel-Akkumulation, mit dem Unterschied, dass die eingesammelten Nachrichten nicht akkumuliert werden. Der Wurzelprozess P_i, der auch hier der Empfänger aller Daten ist, erhält also statt einer einzigen akkumulierten Nachricht bei der Gather-Operation n Nachrichten - so viele Nachrichten werden effektiv versendet.
- **Scatter**: Die Scatter-Operation sendet von einem Prozess ausgehend eine Nachricht an jeden anderen. Der Unterschied zum Einzelbroadcast liegt darin, dass die Nachricht an jeden Empfänger möglicherweise angepasst wird. Der Wurzelprozess legt dazu die nach der ID zugeordneten Nachrichten in einen Sendepuffer, woraus die Nachrichten verschickt werden. Bei dieser Operation werden n Nachrichten versendet.
- **Multi-Broadcast**: Bei einer Multi-Broadcast-Operation wird von jedem Prozess eine Einzel-Broadcast-Operation ausgeführt. Jeder Prozess sendet also eine Nachricht an alle anderen. Von jedem Prozess werden wiederum von jedem anderen Prozess n Nachrichten empfangen. Multi-Broadcast-Operationen sind besonders bei Berechnungen von Arrays praktisch, denn

damit lassen sich verteilt berechnete Array-Elemente einsammeln und allen Prozessen zur Verfügung stellen. Der Overhead der Operation ist größer als bei den bisher vorgestellten Operationen: Es werden n^2 Nachrichten ausgetauscht.
- **Multi-Akkumulation**: Die Multi-Akkumulation basiert, wie der Name vermuten lässt, auf der Einzel-Akkumulation: Jeder Prozess führt eine solche aus und stellt damit jedem Prozess eine unter Umständen unterschiedliche Nachricht zur Verfügung. Beim Empfangen der Nachrichten wird eine Reduktionsoperation ausgeführt. Als Resultat erhält man eine einzige Nachricht, die mit der Reduktionsoperation verknüpft wurde. Auch bei dieser Operation beträgt die Summe der versendeten Nachrichten n^2.
- **Gesamtaustausch**: Bei einer Gesamtaustausch-Operation wird von jedem Prozess eine Scatteroperation durchgeführt: Es wird demnach eine möglicherweise unterschiedliche Nachricht an jeden Empfänger versendet und auf eine Reduktionsoperation wie bei den Akkumulations-Operationen verzichtet. Es werden n^2 Nachrichten versendet.

Die vorgestellten Operationen werden im Standard MPI aufgegriffen und finden sich dadurch in den jeweiligen konkreten Implementierungen dessen wieder.

F Verteilte Programmierung durch Nachrichtenaustausch

Zunächst wird im Folgenden das allgemeine Prinzip der Programmierung durch Austausch von Nachrichten (engl. ,,Message-Passing-Programming") vorgestellt. Danach wird kurz auf die Entwicklung und Historie des Standards *Message Passing Interface* (MPI) in den Versionen 1 und 2, sowie des davon abzugrenzenden Softwarepackets *Parallel Virtual Machine* (PVM) eingegangen. Zu erstgenannter Norm existieren freie Implementierungen, die anschließend betrachtet und mit einer Beispielimplementierung eines PVM-Programms abgeschlossen wird.

F.1 Die Message-Passing-Programmierung

Die Message-Passing-Programmierung ist ein Programmiermodell für Parallelrechner mit verteiltem Speicher und Adressraum. Sie erlaubt den Prozessen durch das explizite Austauschen von Nachrichten miteinander zu kommunizieren und indirekt auf Daten jeweils anderer Prozesse zuzugreifen. Obwohl es prinzipiell vorstellbar ist, dass verschiedene Programme über Nachrichten miteinander kommunizieren (engl. *MPMD: Multiple Program Multiple Data*), verzichtet man normalerweise darauf, um die Programmierung zu vereinfachen: Es wird in der Regel dasselbe Programm auf mehreren Rechenknoten gestartet (engl. *SPMD: Single Program Multiple Data*)

F.2 MPI-1

Das *Message-Passing-Interface* (MPI) ist ein Standard für die Programmierung von Systemen nach dem Message-Passing-Modell. Dieses Modell entspricht dem

System des Nachrichtenaustausches, das in Kapitel E vorgestellt wurde. Es ist unter der Leitung des Message Passing Interface Forum (MPIF) entstanden. Erste Diskussionen fanden im Januar 1993 statt, die Version 1.0 von MPI wurde im Juni 1994 verabschiedet. Version 1.1 folgte genau ein Jahr später[3]. Die erste Implementierung von MPI-1 war *MPICH* vom Argonne National Lab und der Mississippi State University. In Version 1 wurden Schnittstellen zu den Programmiersprachen FORTRAN 77 sowie C definiert. Grundsätzlich definiert der Standard MPI lediglich die Syntax und Semantik, also die Auswirkungen der Operationen. Die konkrete Implementierung wird nicht vorgeschrieben[3]. Ein Programm nach dem MPI-1-Standard besteht aus mehreren Prozessen. Die genaue Anzahl muss jedoch beim initialen Starten des Programms angegeben werden. Es ist nicht möglich, die Anzahl der Prozesse im laufenden Betrieb zu ändern.

Bei den Nachrichtenoperationen wird grundsätzlich zwischen zwei Typen der Kommunikation unterschieden:

- blockierend: Eine Operation heißt *blockierend*, wenn die Kontrolle erst dann an den aufrufenden Programmteil zurückgegeben und dieser fortgesetzt wird, wenn die Operation komplett abgeschlossen ist, und alle Ressourcen freigegeben sind.
- nicht-blockierend: Im Gegenzug heißt eine Operation *nicht-blockierend*, wenn sie die Kontrolle sofort an den Aufrufer zurückgibt, noch bevor die Operation ausgeführt wurde. Die Effizienz von Programmen kann mit nichtblockierenden Methoden oft verbessert werden, da Kommunikation und Berechnungen parallel ablaufen können.

Da der Umfang der von MPI-1 unterstützten Operationen deutlich über die im vorigen Abschnitt hinausgeht, werden wir im Folgenden lediglich einen Ausschnitt der Operationen vorstellen, die allerdings die wichtigsten Funktionen bereitstellen.

Der Grundaufbau eines MPI-Programms entspricht dem aus Listing 5.1:

Listing 5.1. Grundaufbau von MPI-Programmen

```
1 MPI_Init()
2 MPI_Comm_size()
  MPI_Comm_rank()
4  ...
  MPI_Finalize()
```

MPI unterteilt die Menge der zur Verfügung stehenden Recheneinheiten zur möglichen parallelen Ausführung in so genannte *Kommunikatoren*. Diese stellen als Untermengen der Gesamtheit der Prozesse eine Gruppierungsmöglichkeit zur Verfügung. `MPI_Init()` initialisiert den Kommunikator auf den voreingestellten Standard `MPI_COMM_WORLD`. Dieser umfasst alle zur Verfügung stehenden Prozesse.

Mittels `MPI_Comm_size()` wird die Anzahl der in einem Kommunikator befindlichen Prozesse ausgelesen, `MPI_Comm_rank()` liefert dem aufrufenden Prozess eine eindeutige ID, die dem Aufrufer zugeordnet wurde. `MPI_Finalize()` ist

der letzte Aufruf eines jeden teilnehmenden Prozesses. Dieser erledigt Aufräumarbeiten und sorgt dafür, dass durch das Beenden eines Prozesses kein Deadlock entsteht.

Für den Hauptteil der MPI-Programme, der im Pseudo-Code absichtlich weggelassen wurde, sind neben den eigentlichen Berechnungsproblemen die Nachrichtenoperatoren von MPI von Bedeutung: Bei der Übertragung von Nachrichten zwischen Prozessen wird unterschieden in **Einzeltransferoperationen**, die — wie der Name vermuten lässt — Nachrichten zwischen zwei einzelnen Prozessen austauschen, sowie Broadcast- oder **Globaltransferoperationen**. An letzteren sind alle oder eine Teilmenge aller Prozesse beteiligt.

Als Vertreter der Einzeltransferoperationen betrachten wir zunächst die einfachste Form des Datenaustausches, bei der nur ein Sender sowie ein Empfänger beteiligt sind:

```
int MPI_Send(void *smessage,
             int count,
             MPI_Datatype datatype,
             int dest,
             int tag,
             MPI_Comm comm)
```

Die Argumente dieser Methode sind im Einzelnen:

- `*smessage`: Der Zeiger auf den Sendepuffer, der die zu versendenden Daten beinhaltet.
- `count`: Die Anzahl der zu sendenden Elemente.
- `datatype`: Der Datentyp der zu sendenden Elemente.
- `dest`: Die ID des Empfängerprozesses.
- `tag`: Eine Markierung der Nachricht. Diese dient der Unterscheidung mehrerer Nachrichten des gleichen Senders.
- `comm`: Der Kommunikator für die zu sendende Nachricht.

Die auf Zielseite notwendige entsprechende Funktion zum Empfangen der Nachricht lautet:

```
int MPI_Recv(void *rmessage,
             int count,
             MPI_Datatype datatype,
             int source,
             int tag,
             MPI_Comm comm,
             MPI_Status *status)
```

Die Ähnlichkeit der Methode mit der Sendefunktion ist offensichtlich. Die Parameter folgen in ihrer Bedeutung entsprechend: `*rmessage` ist ein Zeiger auf einen Empfangspuffer, in dem die zu empfangende Nachricht abgelegt werden soll. `count` sowie `datatype` bleiben in ihrer Bedeutung gleich, `source` ist die Nummer des Prozesses, von dem die Nachricht empfangen werden soll, `tag` und

`comm` haben ebenfalls die gleiche Bedeutung wie bei der Sendeoperation. `*status` schließlich ist eine Datenstruktur, die Informationen über die tatsächlich erhaltene Nachricht enthält.

MPI-1 unterstützt folgende Datentypen:

MPI-Datentyp	entsprechender C-Datentyp
MPI_CHAR	signed char
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double

Beide vorgestellte Methoden sind blockierende Anweisungen. Sie sperren den Aufrufer also bis die Nachricht vom Empfänger abgerufen wurde (im Falle von `MPI_Send`) bzw. bis eine Nachricht empfangen wurde (im Falle von `MPI_Recv`). Als Resultat geben die Methoden einen Statuscode zurück, der den Erfolg oder Fehler anzeigt: Im Erfolgsfall wird `MPI_SUCCESS` zurückgegeben. Im Fehlerfall können folgende Fehlermeldungen zurückgegeben werden:

- MPI_ERR_COMM Die Fehlermeldung deutet auf einen ungültigen Kommunikator hin.
- MPI_ERR_COUNT Das Argument „count" ist fehlerhaft, es muss positiv sein.
- MPI_ERR_TYPE Ungültiger Datentyp.
- MPI_ERR_TAG Das Argument „tag" ist fehlerhaft, es muss ebenfalls positiv sein. Beim Empfangen von Nachrichten kann auch MPI_ANY_TAG benutzt werden, um alle eintreffenden Nachrichten entgegen zu nehmen.
- MPI_ERR_RANK Ungültige Sende- oder Empfangs-ID. Diese muss zwischen 0 und $n-1$ sein, wenn n die gesamte Anzahl der Prozesse darstellt. Bei Empfangsoperationen kann auch MPI_ANY_SOURCE benutzt werden, um Nachrichten von allen Absendern entgegenzunehmen.

Eine wichtige von MPI definierte Eigenschaft ist die Einhaltung der Sendereihenfolge. MPI stellt für den Benutzer also sicher, dass zwei Nachrichten von einem Sender, die beide auf eine Empfangsoperation passen würden, das Ziel in der richtigen Reihenfolge erreichen. Dies ist für die Betrachtung von möglichen Deadlock-Situationen wichtig. Bei 3 oder mehr Prozessen kann diese Eigenschaft allerdings nicht mehr sichergestellt werden. Die Gefahr von Deadlocks muss vom Programmierer in jedem Fall adressiert werden. Folgendes Programmfragment würde einen Deadlock erzeugen. Diese Situation ist in Listing 5.2 veranschaulicht.

Listing 5.2. Deadlock mit MPI

```
[...]
int myRank, count, tag;
[...]
MPI_Comm_rank(comm, &myRank);
if (myRank == 0) {
  MPI_Recv(recvbuff, count, MPI_INT, 1, tag, comm, &status);
  MPI_Send(sendbuff, count, MPI_INT, 1, tag, comm);
} else if (myRank == 1) {
  MPI_Recv(recvbuff, count, MPI_INT, 0, tag, comm, &status);
  MPI_Send(sendbuff, count, MPI_INT, 0, tag, comm);
}
```

Der Benutzer der MPI-Bibliothek muss beim Versenden und Empfangen von Nachrichten die Reihenfolge beachten. In Listing 5.2 warten die beiden Prozesse mit den IDs 0 und 1 jeweils auf den Empfang einer Nachricht des jeweils anderen Prozesses. Beide Prozesse würden in diesem Fall prinzipiell unendlich lange aufeinander warten.

Nachdem wir die einfachen und grundlegenden Methoden `MPI_Send()` sowie `MPI_Recv()` betrachtet haben, werden wir nun auf die Operationen eingehen, die mehrere Empfänger haben: Unter **Globaltransferoperationen** versteht man alle Funktionen, die Nachrichten an eine Gruppe oder alle Prozesse versenden. Die in Kapitel E vorgestellten Funktionen werden von MPI unterstützt. Als Beispiel für eine solche Operation wollen wir uns die Gatheranweisung im Detail ansehen. In MPI ist die Signatur der Methode wie folgt festgelegt:

```
int MPI_Gather(void *sendbuf,
               int sendcount,
               MPI_Datatype sendtype,
               void *recvbuf,
               count recvcount,
               MPI_Datatype recvtype,
               int root,
               MPI_Comm comm)
```

Die Argumente von MPI_Gather sind im Einzelnen:

- `*sendbuf`: Der Zeiger auf den Sendepuffer.
- `sendcount`: Die Anzahl der zu sendenden Elemente.
- `sendtype`: Der Datentyp der zu sendenden Elemente.
- `*recvbuf`: Der Zeiger auf den Empfangspuffer.
- `recvcount`: Die Anzahl der zu empfangenden Elemente.
- `recvtype`: Der Datentyp der zu empfangenden Elemente.
- `root`: Der Wurzelprozeß, der die Daten empfangen soll.
- `comm`: Der zu verwendende Kommunikator; semantisch identisch zum Argument `comm` bei `MPI_Send()` bzw. `MPI_Recv()`.

Damit die Methode korrekt arbeiten kann, muss der Parameter `root` bei allen beteiligten Prozessen identisch sein. Wie in Kapitel E vorgestellt, ist das Ziel der Operation, dass am Wurzelknoten Nachrichten von allen beteiligten Prozessen ankommen - die Angabe verschiedener Wurzelknoten würde zu Fehlern führen.

Ein Beispiel für eine Gatheroperation, die 10 Integer-Werte von allen Prozessen einsammeln soll, ist in Listing 5.3 dargestellt.

Listing 5.3. MPI_Gather()

```
1 [...]
2 MPI_Comm comm;
  int sendbuf[10], myRank, root = 0, gsize, *rbuf;
4 MPI_Comm_rank (comm, &myRank);
  if (myRank == root) {
6   MPI_Comm_size (comm, &gsize);
    rbuf = (int*) malloc(gsize * 10 * sizeof(int));
8 }
  MPI_Gather(sendbuf, 10, MPI_INT, rbuf, 10, MPI_INT, root, comm);
```

Wie man sehen kann, sind für den Root-Prozess zunächst die Zeilen 5 bis 8 relevant. Der benötigter Speicherplatz zum Empfangen der Daten wird dort allokiert. Die Methode `MPI_Gather()` führt schließlich sowohl die am Root-Knoten als auch die an den restlichen Prozessen benötigen Zuweisungen durch. Als Resultat werden am Root-Prozess die eingesammelten Daten in der durch `rbuf` definierten Speicherstelle abgelegt.

F.3 MPI-2

Nach der Verabschiedung von MPI-1, genauer gesagt MPI-1.1 und MPI-1.2, tauchten weitere Mängel und Verbesserungsmöglichkeiten am Standard auf, sodass 1996 eine neue Version des MPI-Standards veröffentlicht wurde: MPI-2. Zunächst wird nun kurz auf die grundlegenden Veränderungen eingegangen, die mit dem neuen Standard verabschiedet wurden. Anschließend wird auf die Zukunft von MPI eingegangen.

Eine Einschränkung von MPI-1 war, dass es keine Erzeugung von Prozessen zur Laufzeit unterstützt hat. Dieser Umstand wurde mit der Verabschiedung von MPI-2 beseitigt. Änderungen und Erweiterungen betreffen auch die Ein- und Ausgabe. Mit Version 2 wurde die Möglichkeit definiert, Ein- und Ausgaben parallel auszuführen, worauf man Performance-Verbesserungen aufbauen kann.

Ebenfalls neu ist die Möglichkeit, entfernte Speicheroperationen auszuführen, die auch als ,,einseitige Kommunikation" bezeichnet werden: Ein Prozess kann den Speicher eines zweiten Prozesses direkt modifizieren. Der Vorteil dieser Operationen gegenüber herkömmlichem Nachrichtenaustausch für die Speichersynchronisation sind spezielle Hardware-Operationen, die auf einigen Plattformen angeboten werden. Implementierungen, die für diese Plattformen entwickelt werden, können diese Funktionen nutzen und besonders performante Zugriffe bereitstellen[6]. Ferner wurden in MPI-2 u.a. weitere Datentypen eingeführt und Schnittstellen zu C++ und FORTRAN 90 definiert. Die Versionen seit MPI-1.1 sind vorwärtskompatibel. Programme nach MPI-1.1 sind gültig nach dem MPI-1.2-Standard, und diese wiederum nach dem MPI-2-Standard [4].

Zur **Zukunft von MPI**: Die Entwicklung von MPI wird immer noch vorangetrieben: Das MPI Forum plant und arbeitet an den kommenden Versionen MPI-2.1, MPI-2.2 und MPI-3. Die weitreichende Planung des MPI-Forum[6] deutet darauf hin, dass sich der Standard weiterentwickeln wird, und auch in Zukunft eine wichtige Rolle in der Programmierung in verteilten Adressräumen

[6] Für 2008 sind insgesamt sieben Meetings geplant, für 2009 bereits sechs Treffen.

spielen wird. Die Themen, die von zukünftigen MPI-Standards adressiert werden müssen, und dies laut Planung auch werden, beinhalten beispielsweise einen Fokus auf Ausfallsicherheit oder den Ausbau der direkten Speicherzugriffsfunktionen auf entfernten Prozessen.

F.4 MPI-Implementierungen

Wie bereits angedeutet existieren für den MPI-Standard mehrere Implementierungen. Open MPI und MPICH sind zwei Beispiele dafür. Beide werden in den folgenden zwei Abschnitten kurz betrachtet.

F.4.1 Open MPI Open MPI ist eine vollständige Open Source-Implementierung des MPI-1- und MPI-2-Standards. Es ist aus der Zusammenlegung diverser MPI-Implementierungs-Projekte entstanden. Unter anderen waren dies FT-MPI von der University of Tennessee, LA-MPI vom Los Alamos National Laboratory, LAM/MPI von der Indiana University und PACX-MPI, das an der Universität Stuttgart entwickelt wurde. Sun Microsystemens setzt bei ihrem Produkt ,,Sun HPC Cluster Tools" seit der Version 7.0 auf Open MPI, und stellt dem Open MPI-Team Ressourcen zur Verfügung. Durch diesen öffentlichen Einsatz von Sun gegenüber Open MPI ist zu erwarten, dass Open MPI im Supercomputing-Markt eines der am häufigsten eingesetzten Softwarepakete werden wird.

F.4.2 MPICH Die erste Implementierung, die nach der Verabschiedung des MPI-1-Standards überhaupt entstanden ist, war MPICH1. MPI-2 wurde ebenfalls implementiert und wurde unter dem Namen MPICH2 veröffentlicht. MPICH ist Open-Source verfügbar. Fokus setzt die Entwicklergruppe dabei auf die Unterstützung einer möglichst breiten Palette von Systemen von Arbeitsplatzrechnern bis zu Hochleistungscomputern sowie Vernetzungsmöglichkeiten wie 10-Gigabit-Ethernet, InfiniBand, Myrinet oder Quadrics.

F.5 PVM

Ein anderer Ansatz zur Programmierung mit verteilten Adressräumen ist PVM (Parallel Virtual Machine)[5]. Es wurde von der University of Tennessee, dem Oak Ridge National Laboratory und der Emory University konzipiert, um mehrere Recheneinheiten mit unterschiedlichen Hardware-Architekturen und/oder verschiedenen Betriebssystemen zu einer großen virtuellen Recheneinheit zusammenzuschalten. PVM stellt als Softwarepaket die Funktionen bereits zur Verfügung und ist daher kein Standard wie MPI. Das Projekt ist unter eine Open Source-Lizenz frei verfügbar und mit Fokus auf Portabilität entwickelt: Derzeit werden von PVM folgende Architekturen und Betriebssysteme unterstützt:

- Microsoft Windows 95 bis Windows Vista
- Apple MacOS
- Linux

- FreeBSD, NetBSD
- Sun Solaris
- IBM AIX 3.x, AIX 4.x
- Hewlett Packard HPUX
- diverse weitere, teils proprietäre Systeme: Cray YMP, T3D, T3E, Cray2, Convex Exemplar, IBM SP2, 3090

PVM unterstützt alle derzeit im Workstation-Bereich relevanten Betriebssysteme. Auch für Systeme, die hauptsächlich in High-Performance-Computing-Umgebungen Anwendung finden, ist Unterstützung vorgesehen.

Die von PVM implementierten Funktionen umfassen unter anderem die dynamische Erzeugung von Prozessen zur Laufzeit und Load-Balancing zum Verteilen von Prozessen. Dabei abstrahiert PVM die Hardware derart, dass dem Benutzer beziehungsweise dem PVM-Programm alle Maschinen identisch erscheinen. Falls es der Benutzer wünscht und entsprechend programmiert, kann diese Abstrahierung durchbrochen werden. Dies ist beispielsweise sinnvoll, um spezialisierten Rechnern bestimmte Aufgaben zu geben: Man könnte sich vorstellen, dass schon beim Entwurf des PVM-Programms klar ist, dass bestimmte Teilergebnisse mehr Fließkommaoperationen benötigen als andere. Dem kann man entsprechen, und die Teilaufgaben auf die Maschinen verteilen, die eine höhere Bearbeitungsgeschwindigkeit von Fließkommaoperationen aufweisen.

Spezielle Funktionen stellt PVM zur Organisation von Aufgaben bereit: Prozesse heißen im PVM-Jargon *Tasks* und entsprechen den bekannten Prozessen bei MPI. Über einen *TID* (Task Identifier) ist eine eindeutige Adressierung eines jeden Tasks möglich. Wie in MPI können auch bei PVM mehrere Tasks gruppiert werden, die vom Programmierer über eine Zeichenkette adressiert werden. Ein Task kann dabei in einer oder mehreren Gruppen vertreten sein.

Anders als bei frühen Versionen von MPI war das dynamische Erzeugen und Beenden von Tasks schon immer Bestandteil von PVM. Dadurch wird bei PVM-Programmen normal initial lediglich ein Task gestartet, und danach weitere erzeugt. Dafür sind folgende PVM-Anweisungen vorgesehen:

```
int pvm_spawn(char *task,
              char **argv,
              int flag,
              char *where,
              int ntask,
              int *tids)
```

Die Argumente von `pvm_spawn` sind im Einzelnen:

- `*task`: Der Zeiger auf den Namen des auszuführenden Task.
- `**argv`: Ein Array mit Parametern für das zu startende Programm.
- `flag`: Dieser Paramter gibt als Flag an, wie das Abspalten des Task durchgeführt werden soll.
- `*where`: Der Rechnername, der für bestimmte Besetzungen des Flags angibt, auf welchem Rechner oder auf welcher Architektur der Task gestartet werden soll.

- `ntask`: Die Anzahl der zu startenden Tasks.
- `*tids`: Ein Array mit TIDs der neu gestarteten Tasks.

Für das Flag sind unterschiedliche Möglichkeiten interessant:

Mittels `PvmTaskDefault` kann man PVM die Aufgabe komplett übergeben, einen Zielrechner für den neuen Task zu suchen. Bei `PvmTaskHost` wählt man den in `*where` spezifizierten Zielrechner, bei `PvmTaskArch` wählt man die dort spezifizierte Architektur. Andere Möglichkeiten zur Besetzung des Flags sind die Ausführung des Tasks in einem Debugger, Erzeugung von Trace-Dateien bei der Ausführung oder der Wahl der Komplementmenge von `*where`.

Das Hinzunehmen und Entfernen von Rechnern aus dem PVM-Verbund geschieht durch die Methoden `pvm_addhosts()` beziehungsweise `pvm_delhosts()`. Beide Methoden nehmen die folgenden Paramter:

- char **hosts: Ein Feld der Länge *nhost* mit den Adressen der Rechner.
- int nhost: Die Anzahl der hinzuzufügenden bzw. zu löschenden Rechner.
- int *infos: Als Integerzahl codierte Informationen zu den Rechnern, beispielsweise bezüglich der Hardware-Architektur.

Als Rückgabewert gibt `pvm_addhosts()` die Anzahl der hinzugefügten Rechner zurück. `pvm_delhosts()` gibt die Anzahl der gelöschten Rechner zurück.

Bei der Verwaltung von **Prozeßgruppen** bietet PVM einige Besonderheiten: PVM erlaubt es allen Tasks, eine Prozeßgruppe zu verlassen oder einer beizutreten, ohne die Tasks der jeweiligen Gruppe zu informieren. Das Beitreten wird über die Methode `pvm_joingroup()` unter Angabe des Namens der Prozeßgruppe realisiert. Zurückgegeben wird entweder eine positive ganze Zahl, die die Identifikationsnummer des Task darstellt oder eine negative Zahl für den Fall eines Fehlers. Im Gegenzug wird eine Gruppe mit `pvm_lvgroup()` verlassen. Nach der Erstellung einer Gruppe durch `pvm_joingroup()` können diverse Operationen auf der Gruppe ausgeführt werden:

- Als Synchronisationsanweisung steht `pvm_barrier()` zur Verfügung. Der aufrufende Task blockiert dabei so lange, bis die als Parameter übergebene Anzahl an Tasks die Synchronisationsoperation aufgerufen haben.
- `pvm_gsize()` gibt die Anzahl der Tasks in einer Gruppe zurück.
- Mit `pvm_bcast()` kann eine Broadcastoperation an eine Gruppe realisiert werden.
- Als Akkumulationsoperation dient `pvm_reduce()`.

Eine weitere Besonderheit von PVM besteht darin, dass jeder Task Nachrichten als Broadcast an eine Prozeßgruppe schicken kann, selbst wenn dieser nicht Mitglied der Gruppe ist.

Das **Versenden von Nachrichten** wird bei PVM in mehreren Schritten durchgeführt:

1. `int pvm_initsend(int encoding)` initialisiert den Nachrichtenaustausch und legt eine Kodierung fest. Der Standard ist `PvmDataDefault` und definiert eine sichere Kodierung, falls im PVM-Verbund Rechner mit verschiedenen Datenrepräsentationen sind.

2. Vor dem Versenden werden alle zu versendenden Variablen gepackt. PVM stellt für jeden Datentyp eine eigene Packanweisung zur Verfügung: Für Gleitkommazahlen ist dies `pvm_pkfloat()`. Strings werden beispielsweise mittels `pvm_pkstr()` verpackt.
3. Das eigentliche Versenden wird mit `pvm_send()` durchgeführt. Als Parameter werden das Ziel sowie eine Markierung der Nachricht übergeben.

Das **Empfangen** von Nachrichten wird in zwei Schritten durchgeführt:

Zunächst wird mit `pvm_recv()` die Nachricht entgegengenommen. Im zweiten Schritt wird das Verpacken der Nachricht beim Versenden rückgängig gemacht. Dafür ist zu jeder Verpackoperation ein Pendant zum Entpacken festgelegt. Für Float-Werte ist dies beispielsweise `pvm_upkfloat()`.

Im Listing 5.4, das leicht modifiziert dem Beispiel „Fork-Join Example" aus [5] entspricht, ist ein Beispiel zu PVM aufgeführt, das die einzelnen Methoden praktisch veranschaulicht.

Listing 5.4. Beispiel mit PVM

```
1  #include <pvm3.h> // Prototypen und Konstanten des PVM-Pakets
2
   int main(int argc, char* argv[]) {
4
     int ntask = 3;           // Anzahl der zu erstellenden Tasks
6    int info;                // Statusmeldungen der PVM-Aufrufe
     int mytid;               // Lokale Task-ID
8    int myparent;            // Task-ID des Elternprozesses (sofern vorhanden)
     int child[20];           // Array der Kinder-Task-IDs
10   int i, mydata, buf,
         len, tag, tid;
12
     mytid = pvm_mytid();     // Eigene Task-ID herausfinden
14
     if (mytid < 0) {         // Auf Fehler pruefen
16     pvm_perror(argv[0]);   // Fehler ausgeben
       return -1;
18   }

20   myparent = pvm_parent(); // Task-ID des Elternprozesses herausfinden

22   // Auf Fehler pruefen
     if ((myparent < 0) && (myparent != PvmNoParent)) {
24     pvm_perror(argv[0]);
       pvm_exit();
26     return -1;
     }
28

30   // Wenn lokaler Task keinen Elterntask besitzt, ist es die Wurzel
     if (myparent == PvmNoParent) {
32     // Anzahl der Tasks erstellen
       info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault,
34     (char*)0, ntask, child);

36     for (i = 0; i < ntask; i++)
         if (child[i] < 0) // Fehler ausgeben
38         printf(" \d", child[i]);
39       else              // Task-ID ausgeben
40         printf("t\t", child[i]);
41
42     // Beenden, falls das Task-Erstellen nicht erfolgreich war
       if (info == 0) { pvm_exit(); return -1; }
44
```

```
    // Anzahl der erfolgreich erstellen Tasks
46  ntask = info;

48  for (i = 0; i < ntask; i++) {
      buf = pvm_recv(-1, 11); // Nachricht empfangen
50    if (buf < 0) pvm_perror("Fehler bei pvm_recv");
      info = pvm_bufinfo(buf, &len, &tag, &tid);
52    if (info < 0) pvm_perror("Fehler bei pvm_bufinfo");
      info = pvm_upkint(&mydata, 1, 1);
54    if (info < 0) pvm_perror("Fehler bei pvm_upkint");
      if (mydata != tid) printf("Ohoh!\n");
56    printf("L\änge %d, Tag \d, Tid t\x\n", len, tag, tid);
57  }
58  pvm_exit();
    return 0;
60 }
}
62

  // Lokaler Task ist nicht die Wurzel, sondern ein Kind
64 info = pvm_initsend(PvmDataDefault);
  if (info < 0) {
66  pvm_perror("Fehler bei pvm_initsend"); pvm_exit(); return -1;
  }
68 info = pvm_pkint(&mytid, 1, 1);
  if (info < 0) {
70  pvm_perror("Fehler bei pvm_pkint"); pvm_exit(); return -1;
  }
72 info = pvm_send(myparent, JOINTAG);
  if (info < 0) {
74  pvm_perror("Fehler bei pvm_send"); pvm_exit(); return -1;
  }
76 pvm_exit();
  return 0;
78}
```

Im Erfolgsfall liefert Listing 5.4 beispielsweise folgende Ausgabe:

```
t10001c t40149  tc0037
Length 4, Tag 11, Tid t40149
Length 4, Tag 11, Tid tc0037
Length 4, Tag 11, Tid t10001c
```

Die Ausgabe ist hinsichtlich der Reihenfolge nicht deterministisch, da diese abhängig davon ist, wann die Nachrichten eintreffen. Dies wird in der Realität wiederum von Größen wie beispielsweise der Netzwerk-Latenz beeinflusst.

G Ausblick

Parallele Programmierung wird durch den in Kapitel C dargestellten Trend hin zu Arbeitsplatzcomputern mit vielen Cores noch wichtiger werden als es bereits ist. Obwohl man Techniken für die Programmierung mit gemeinsamen Adressraum für die dort anzuwendende Technik wägt, werden wohl auch Forschungs- und konkrete Anwendungsergebnisse aus der Welt des verteilten Adressraums von Bedeutung sein und in die Entwicklung eingehen. Durch den in der Einleitung dargestellten hohen Bedarf an Rechenleistung wird die Programmierung mit verteiltem Adressraum für Hochleistungssysteme zur Simulation und Forschung auch künftig ein wichtiges Forschungsfeld darstellen.

Literatur

[1] I. A. Advanced Micro Devices. Server roadmap update fact sheet, 05 2008. http://www.amd.com/us-en/assets/content_type/DownloadableAssets/Roadmap_Update_Fact_Sheet_Final.pdf (abgerufen am 19.05.2008).

[2] S. Cheshire. It's the latency, stupid, 05 1996. http://rescomp.stanford.edu/ cheshire/rants/Latency.html (abgerufen am 28.06.2008).

[3] M. P. I. Forum. Mpi: A message-passing interface standard, 08 1997. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html, abgerufen am 24.05.08.

[4] M. P. I. Forum. Mpi-2: Extensions to the message-passing interface, 09 2001. MPI-2: Extensions to the Message-Passing Interface, abgerufen am 22.05.08.

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. eram. *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Paralle l Computing*. MIT Press, Cambridge, Mass., 1994.

[6] F. Hoffman. One-sided communications with mpi-2, 2004. http://www.linux-mag.com/id/1793 (abgerufen am 30.05.08).

[7] Intel Corporation. Intel software network knowledge base wiki, 05 2008. http://softwarecommunity.intel.com/Wiki/GlossaryofTechnicalTerms/875.htm (abgerufen am 19.05.2008).

[8] T. I. List. http://www.top500.org/stats/list/31/conn (abgerufen am 28.06.2008).

[9] M. Mairandres. *Virtuell gemeinsamer Speicher mit integrierter Laufzeitbeobachtung*. PhD thesis, Forschungszentrum Jülich, 1996.

[10] Myricom. Myrinet overview, 06 2005. http://www.myricom.com/myrinet/overview/ (abgerufen am 28.05.2008).

[11] L. Osterman. Did you know that os/2 wasn't microsoft's first non unix multi-tasking operating system?, 03 2004. http://blogs.msdn.com/larryosterman/archive/2004/03/22/94209.aspx (abgerufen am 27.04.2008).

[12] E. B. Pitman. Amdahl's law for parallel speedup, 08 2000. http://www.math.buffalo.edu/ pitman/courses/cor501/HPC1/node11.html (abgerufen am 27.4.2008).

[13] T. Rauber and G. Rünger. *Parallele und verteilte Programmireung*. Springer, 2000.

[14] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts 7th Edition with Java 7th Edition*. John Wiley & Sons, 2006.

[15] TOP500.Org. Top 500 supercomputing sites, 11 2008. http://www.top500.org/lists/2008/06 (abgerufen am 21.06.2008).

Debugging in parallelen Programmen

Dominik Bösl

Universität Augsburg
dominik@boesl.org

Zusammenfassung In dieser Arbeit wird ein Überblick über die Probleme bei parallelem Debugging sowie deren Lösungsansätze gegeben. Darüberhinaus wird der Aufbau und die Funktionsweise der Eclipse Parallel Tool Platform und des darin enthaltenen Parallel Debuggers erläutert.

A Einleitung

Innerhalb der letzten Jahre ist der Stellenwert des parallelen Programmierens enorm gewachsen. Dies hat mehrere Gründe: Zum einen bieten seit geraumer Zeit selbst einfachste Desktopprozessoren mehrere Prozessorkerne und damit echte Multithreading-Fähigkeiten, zum anderen unterstützen auch immer mehr Programmiersprachen das Konzept der Nebenläufigkeit. Aber nicht nur die Erstellung von Softwareanwendungen muss parallele Abläufe unterstützen, sondern damit einher gehend ist nunmehr auch die Möglichkeit parallelen Debuggens gefordert. Im Verlauf dieser Arbeit soll daher auf die Schwierigkeiten beim parallelen Debuggen sowie einige Lösungsansätze daraus resultierender Problemstellungen eingegangen werden. Der zweite Teil behandelt daraufhin mit der Eclipse Parallel Tool Platform sowie dem Zugehörigen Debugger ein Toolkit, das diese Problematiken zu lösen sucht.

B Paralleles Debuggen

Das Debuggen von Software gehört an sich bereits zu den nichttrivialen Aufgaben im Softwareentwicklungszyklus. Wenn es sich dabei allerdings zusätzlich um ein Programm mit nebenläufigen Teilen handelt, eröffnen sich eine Menge weiterer möglicher Problemstellungen, die es zu Beachten gilt.

Der klassische Ansatz um sequentielle Programme zu debuggen, das zyklische Debuggen, beinhaltet das wiederholte anhalten des Programms während seiner Ausführung, die Untersuchung des aktuellen Zustands, um darauf entweder im Ablauf fortzufahren, oder den ausgeführten Programmteil erneut auszuführen um an einer früheren Stelle anhalten zu können. Unerfreulicher weise ist das Verhalten während der Ausführung paralleler Code-Abschnitte nicht immer reproduzierbar gleich; das Ergebnis kann selbst bei identischen Bedingungen und gleichen Eingaben bei jeder Ausführung abweichen. Dieses Verhalten ist durch so genannte „races“ [10] (Wettrennen) zu erklären, die Auftreten, sobald zwei

oder mehr Stränge parallel ausgeführt werden. So kann sich das Verhalten eines Prozesses, der aus einem Speicherbereich lesen möchte, in den ein anderer zu schreiben sucht, radikal ändern, je nachdem, welchem der beiden der Zugriff zuerst erlaubt wird. Eine weitere Fehlerquelle, die dazu führen kann, dass Fehler durch das zyklische Debuggen nicht gefunden werden, besteht darin, dass das unerwünschte Verhalten bei einer oder mehrerer erneuter Ausführungen nicht zwangsläufig auftreten muss und somit übersehen werden kann; abhängig von der Wahrscheinlichkeit des Fehlerfalls kann es passieren, dass dieser beim Testen niemals auftritt, dennoch ist die Möglichkeit gegeben. Dies gilt gleichermaßen für alle Versuche, mehr Wissen über das Programm zu erlangen. Dieses Verhalten wird in der Literatur auch als „Probe Effect“ [4] oder „Heisenberg'sche Unschärfe angewandt auf Software“ [9] referenziert und bezeichnet die Eigenschaft, dass in Programmen mit wetteifernden Prozessen jede zusätzliche Ausgabe oder das Einfügen von Debugging-Code zu einer Veränderung des Ablaufverhaltens und damit zu einem anderen Ausgang des Rennens führen kann, der nicht beobachtbar ist und dennoch die Wahrscheinlichkeit, ein bestimmtes Verhalten zu beobachten, entscheidend verändert. Zusätzlich erschwert wird die Vorhersagbarkeit des Ausgangs solcher races dadurch, dass das Programm selbst in den seltensten Fällen keinen Einfluss auf deren Ergebnis hat, sondern sich das Ergebnis meistens durch äußere Enflüsse wie die aktuelle Auslastung der vorhandenen CPUs, des Netzwerks oder die Schwankungen in nichtdeterministischen Kommunikationsmedien (vgl. [12] zu exponentiellen backoff-Protokollen) bedingt. Eben dieser Nichtdeterminismus ist es, der das Schreiben, Verstehen und Debuggen von parallelem Code in Vergleich zu sequentiellen Abläufen so komplex und zeitaufwendig macht. Ein weiteres Problem bei parallelen Programmen stellt das Fehlen eines globalen Gesamtzustandes dar [7], da es ohne eine allgemein verbindliche globale Uhr nicht möglich ist, eine sicher Aussage über die zeitliche Abfolge von Ereignissen zu treffen, auch wenn diese nebenläufig zueinander arbeiten.

B.1 Problemstellungen bei parallelem Debugging und deren Lösungsansätze

Im Zusammenhang mit parallelem Debuggen müssen zuerst die Begrifflichkeiten „Monitoring“ und “Debugging“ unterschieden werden. Bezugnehmend auf [5] bietet sich folgende Definition an: Beim Monitoring handelt es sich um einen Prozess des Beobachtens, in dessen Ablauf Information über die Programmausführung gesammelt wird. Debugging hingegen ist, laut dem ANSI/IEEE Standard, definiert als der Prozess, in dessen Verlauf mögliche Fehlerquellen gefunden, analysiert und gegebenen Falls behoben werden; Fehlerquellen werden als zufälliger Zustand festgelegt, der für die fehlerhafte Programmausführung verantwortlich ist. Nachfolgend soll auf verschiedene Ansätze des parallelen Debuggens und Ihre Einsatzgebiete eingegangen werden. Grob lassen sich vier Haupteinteilungen in traditionelle Debuggingverfahren, die nur auf parallele Anwendung angepasst wurden, ereignisgesteuerte Debugger, die die Programmausführung als Folge von Ereignissen betrachten, Techniken zur Darstellung von Daten- und Kontrollflüssen in traditionelle Debuggingverfahren, die nur auf

parallele Anwendung angepasst wurden, ereignisgesteuerte Debugger, die die Programmausführung als Folge von Ereignissen betrachten, Techniken zur Darstellung von Daten- und Kontrollflüssen in parallelen Programmen sowie die statische Datenflussanalyse treffen.

B.2 Überblick über Systeme für paralleles Debugging und deren genaue Lösungsstrategien

Wie bereits erwähnt, stellt die Verwendung von mehreren konventionellen Debuggern für ein paralleles System die einfachste Form eines parallelen Debuggers dar. Dabei wird jedem der parallelen Prozesse sein eigenen Debugger zugeordnet. Diesem Prinzip folgt auch immer noch ein Großteil der eingesetzten, einfacheren, Paralleldebugger. Diese Sorte unterscheidet sich von den einfachen Lösungen für singlethreaded Software nur darin, wie die einzelnen sequentiellen Teile gesteuert werden und wie deren Ausgabe zu einem sinnvollen Gesamtergebnis kombiniert werden kann. Den im vorhergehenden Kapitel diskutierten Probe Effekt vernachlässigt dieser Lösungsansatz dabei meist vollständig, wodurch Fehler durch zeitliche Abhängigkeiten nur schwer bis überhaupt nicht erkannt werden können. Alle anderen, nicht-zeitlichen Fehler lassen sich jedoch mit dieser Methode sehr gut erkennen und beheben, was zum einen dadurch bedingt ist, dass nur zeitliche Fehler vom Probe-Effekt begünstigt werden; zum anderen ist nicht gesagt, dass ein zeitkritischer Fehler überhaupt wichtige konkurrierende Programmabläufe beeinträchtigt. Zu guter Letzt kann man dem traditionellen Ansatz zudem Entgegenhalten, dass er auf eine relativ niedrigen Level der Software ansetzt, da die Schwierigkeit bei Programmen, die aus vielen Nebenläufigen Strängen bestehen, eindeutig im Verständnis der Vorgänge auf der Interprozess-Eben liegt und nicht im eigentlichen Ablauf der Threads. Um hingegen die Vorgänge auf der Instruktions- sowie Prozedurebene zu betrachten sind die traditionellen sequentiellen Debugger bestens geeignet.

Koordination verschiedener Sequentieller Debugger

Das Hauptproblem beim Einsatz mehrerer traditioneller serieller Debugger ist ihr Kommunikation untereinander, wenn sie gleichzeitig auf unterschiedlichen parallelen Prozessen agieren. Dabei müssen drei Aufgaben koordiniert und überwacht werden: Jede Aufgabe, die an einen der Debugger gesendet wird, muss an alle der Debugger gesendet werden, es sei denn, es handelt sich um eine Aktion, die nur auf dem aktuellen Thread ausgeführt werden soll, weil zum Beispiel ein Fehler gefunden wurde. Einzelne Aktionsbefehle für einen speziellen der Debugger müssen auch zu ausschließlich diesem Debugger gelangen können. Zuletzt müssen die verschiedenen Ausgaben der einzelnen Routinen sortiert und sinnvoll den einzelnen Strängen zugeordnet ausgegeben werden.

Im simpelsten Fall könnte man sich hierbei vorstellen, jedem Prozess einen eignen Debugging-Prozess zuzuweisen auf den der Anwender jeweils über eine eigene Terminmal-Session bzw. ein eigenes Fenster zugreift. Intuitiv ist aber sofort einsichtig, dass diese Methode allenfalls für Testzwecke und für eine geringer Anzahl an Prozessen denkbar ist, da alle Steuerungsaufgaben manuelle erfolgen müssen. Zudem ist eine direkte Auswirkung auf den Probe-Effekt zu erwarten, da

manuelle Befehle zum Stoppen und Fortführen eines Prozesses -ungeachtet der Laufzeitverzögerung bis zur User-Reaktion- direkt in das Scheduling eingreifen.

Für das UNIX-Umfeld bietet Sun Microsystems ein entsprechendes Tool, das dbxtool, an, das es ermöglicht, jedem Unix-Prozess eines nebenläufigen Programms einen eigenen sequentiellen Debugger zuzuordnen, wobei diese untereinander nicht koordinierbar sind. Dabei unterliegen die Prozesse verschiedenen Einschränken und dürfen zum Beispiel keine forks enthalten; ebenso können Programmressourcen, auf denen der Debugger gerade arbeitet, nicht zwischen den Prozessen geteilt werden sondern sie sind während dieser Zeit explizit für Zugriffe gesperrt.

Bei der erläuterten (zugegebener Maßen etwas suboptimalen) Variante ist die Koordination der Nachrichten zu den einzelnen Debug-Prozessen dem Fenstermanager überlassen. Falls aber auch an dieser Stelle vollständige Kontrolle über das Messaging erforderlich ist, müssen die Prozesse einzeln adressiert werden. Dabei kann ein Prozess als aktueller Prozess ausgewählt und gesteuert werden. Allerdings ist für Befehle wie etwa „stop process P531“ wiederum ein genauer Überblick über alle Prozesse erforderlich und auch in diesem Fall sollte das Timing besser außer acht gelassen werden, da sich die direkte Interaktion durch den User maßgeblich auf die Programmausführung auswirkt. Außer Anweisungen an einzelne Prozesse zu schicken, kann dies auch für alle erfolgen. Dabei erhält jeder Strang die Nachricht beinahe zur selben Zeit, was für normalkritische Programme eine ausreichende Genauigkeit darstellt. Wenn es sich bei den zu debuggenden Systemen allerdings um harte Echtzeitanwendungen handelt, bei deren Ausführung jeder Prozessortakt zählt, muss zusätzlich beachtet werden, dass zwischen dem Senden eines „stop all processes“-Befehls durchaus ein paar Takte vergehen können bis auch der letzte Thread auf dem letzten Kern angehalten wurde. Die Betrachtung komplexer Scheduling-Mechanismen zur Kompensation dieser speziellen Problematik werden im weiteren aber nicht den Focus dieser Arbeit bilden.

Breakpoints

Das Hauptmerkmal sequentieller Debugger ist ihre Fähigkeit, Breakpoints zu setzen, die dazu dienen, den Programmablauf beim Debuggen genau zu verfolgen. Alle parallelen Debugger, die auf einem sequentiellen Pendant basieren, verfügen ebenfalls über dieses Feature, wobei diese generell verschiedene Betriebsmodi anbieten, wie das Anhalten an einer zuvor spezifizierten Code-Stelle, bei einer Ausnahme, einem durch den Anwender beobachtbaren Ereignis, dem Zugriff auf einen spezielle Variable oder wenn eine vorher spezifizierte Bedingung erfüllt ist. Bei der Anwendung von Breakpoints für nebenläufige Programmteile kann zudem die Reaktion bei Erreichen eines solchen Punktes definiert werden; dabei können entweder nur der aktuell betroffene oder alle Prozesse angehalten werden, wobei der erste Fall komplexe Synchronisationsverfahren unumgänglich machen kann und unter Umständen nicht zu einem sofortigen Halt führt, wohingegen Zweiterer mit den Timeouts anderer Stränge kollidieren könnte.

Die Behandlung von Timeouts stellt die größte Herausforderung bezüglich Breakpoint in mehrfädigen Anwendungen dar, da durch das Anhalten eines Pro-

zesses Timeout-Ereignisse in anderen, davon abhängigen Vorgängen ausgelöst werden können, die zum Abbruch des Programms führen. Ein Lösungsansatz hierfür ist, eine für alle Prozesse verbindliche, logische Zeit einzuführen, die in dem Fall, dass alle Prozesse gestoppt werden, ebenfalls angehalten wird und erst nach deren Fortlauf die Prozesse wieder ausgeführt werden können. Falls durch den Breakpoint nur ein Prozess angehalten wird, laufen die logische Zeit und damit auch die Ausführung für alle anderen Prozesse unverändert weite. Sobald allerdings ein Prozess eine zeitkritische Stelle erreicht hat, die zu einem Timeout führen würde, wird auch für ihn die Zeit eingefroren und er beginnt erst wieder zu laufen, wenn der ursprünglich gestoppte Strang wieder ausgeführt wird. Somit beginnt auch erst ab diesem Zeitpunkt der Timeout-Zähler wieder zu laufen. Durch dieses Verfahren wird zwar der Probe-Effekt nicht vollständig aufgehoben, dennoch kann die Auswirkung konventionellen parallelen Debuggens auf zeitabhängige Abläufe minimiert werden.

Um adäquat über Breakpoints in mehrfädigen Systemen sprechen zu können, muss man nicht nur den aktuellen Zustand sondern auch die für den Prozess geltenden Ereignisse betrachten. Bei einem Ereignis handelt es sich dabei um eine atomare Aktion die auch über den Gültigkeitsbereich eines Prozesses hinaus existiert. Problematisch für paralleles Debuggen können vor allem die Anweisungen sein, die einen globalen Zustand innerhalb des Programms benötigen. Anhand eines Beispiels aus [10] wird dies deutlich: Ein Zusammenhang wie „Prozess A darf die Variable X niemals verändern, solange Prozess B die Variable X noch bearbeitet“ kann dem Debugger durch die Laufzeitverzögerung als nicht verletzt gemeldet werden, obwohl bereits ein unerlaubter Schreibvorgang stattgefunden hat. Zwar kann dies wiederum durch oben genannte Techniken und das einführen einer globalen Uhr verhindert werden, jedoch kann es unter Umständen dennoch nicht möglich sein, die betroffenen Prozesse zu stoppen, bevor sich der Zustand bereits geändert hat und die falschen Werte gesetzt wurden.

Event Histories

Bevor die Betrachtung von Event Histories möglich ist, muss festgehalten werden, dass jedem Debugger durch seinen Unterschiedlichen Focus und Aufgabenbereich eine andere Definition eines Ereignisses Zugrunde liegt. Bei einem RADAR-System handelt es sich dabei um gesendete oder empfangene Nachrichten, in YODA bezeichnen Ereignisse Aktivitäten von einzelnen ADA tasks, in DISDEB betrachtet man Speicherzugriffe und in wieder anderen Systemen wie TSL und EDL kann der Programmierer selbst die Ereignisse definieren. Bei Systemen die auf Interprozesskommunikation angewiesen sind, kann man den Begriff des Events in zwei Bereiche unterteilen, je nachdem ob es sich dabei um ein Ereignis handelt, dass zwischen Prozessen stattfindet oder innerhalb eines einzelnen.

Einige Debug-Systeme zeigen bisher nur die Ereignisse an, wenn sie auftreten; wünschenswert ist aber die Verwendung von Event Histories, die alle vom Programm generierten Events aufzeichnen. Dieses Protokoll kann nach der Programmausführung dann Schritt für Schritt analysiert werden, wobei hierfür einen Such- und Filterfunktion wünschenswert ist, um in dem Fülle an Infor-

mation nicht den Überblick zu verlieren. Darüberhinaus bieten die Log-Daten die Möglichkeit, anhand der Aufzeichnungen fehlerhafte Programmpfade reproduzieren zu können, da eine Sequenz der einzelnen Ablaufschritte aufgezeichnet vorliegt. Falls diese Aufzeichnungen dann zudem lückenlos und umfangreich genug vorliegen, kann ein einzelner Prozess an sich isoliert analysiert werden, wobei der gesammelte Datensatz die nötige Kommunikationsinformation liefert. Schlussendlich sind auch Verfahren denkbar, die automatisch anhand von typischen Fehlersignaturen Programmbereiche identifizieren, in denen es zu einer Fehlausführung kommen könnte oder bereits gekommen ist.

Beim eigentlichen Protokollieren des Programmablaufs wird versucht, den Debugger so wenig wie möglich in den Ablauf eingreifen zu lassen, sondern lediglich zur Laufzeit die anfallenden Daten zu sammeln. Anschließend kann die aufgezeichnete Information analysiert werden. Wie viel und wie genau die Daten gesammelt werden müssen hängt wiederum vom Einsatzzweck der Event History ab. In Aufsteigender Komplexität sind die das Durchsuchen der History, die (automatisierte) Wiederausführung und die Simulation des Programmablaufs. Das Durchsuchen der Aufzeichnung kann dabei manuell über Filterregeln bis hin zur Vollautomatischen Analyse durch spezielle Tools erfolgen; die benötigte Genauigkeit wird dabei durch den Anwendungszweck bestimmt. Bei der Wiederausführung steuert der Debugger den Programmablauf anhand der History um einen alten Ablauf mehrmals exakt reproduzieren zu könne. Somit ist es möglich, das Verhalten jedes Prozesses nacheinander einzeln zu analysieren; hierzu können Breakpoint, Zustandsanalyse und die Schrittweise Ausführung verwendet werden, ohne das Verhalten der Anwendung zu verfälschen. Bei der Simulation schließlich werden einzelne Stränge durch reine Simulation ohne einen realen Durchlauf des Programms analysiert, wobei traditionelle sequentielle Debugger zum Einsatz kommen können.

Abgesehen vom benötigten Detailgrad der Protokolle ist je nach Einsatzbereich die Grundsatzentscheidung zwischen einer linear und partiell geordneten History zu treffen. Beide Verfahren bieten Vor- und Nachteile, so ist die lineare Ordnung für den Menschen leichter verständlich und zeigt zeitliche Abfolgen genau auf; die Zuordnung von Ereignissen zu ihren Prozessen ist jedoch schwierig und es kann sogar zu einer Missinterpretation der Zusammenhänge zwischen Prozessen kommen. Um eine partiell geordnete Aufzeichnung zu erhalten, kann jeder Prozess unabhängig von den anderen beobachtet und protokolliert werden; unter Anderem wird dies vom Traveler-Debugger für Arcore Programme (eine LISP-ähnliche Sprache) so realisiert.

Durchsuchen von Protokollen

Die gesammelten Protokolle können bei umfangreichen Programmen schnell ein für den Menschen unüberschaubares Ausmaß annehmen. Daher sind Filter- und Suchmechanismen unabdingbar, falls nicht sogar graphische Aufbereitungen der Daten zur Verfügung stehen, auf die aber erst ein wenig später eingegangen werden soll. Ein wichtiger Aspekt bei der Speicherung ist zudem, dass die zeitlichen Abhängigkeiten unter den einzelnen Prozessen erhalten bleiben müssen. Dazu ist es möglich, wie in YODA, diese zeitlichen Beziehungen durch

die Verwendung der Sprache PROLOG auszudrücken, die über die Zeitbegriffe „während“, „ bevor“ und „nachdem“ verfügt. Damit können Anfragen der Form „Welchen Wert hatte die Variable X vor Ausführung des Tasks T durch den Prozess P2?“

Eine weitere Möglichkeit ist, wie in [11] beschrieben, die Ereignisse in einer relationalen Datenbank zu speichern. Zusätzlich zu den sonst gespeicherten Zusammenhänge der Ereignisse untereinander können so auch Interval-Relationen gespeichert und somit Abläufe mit einer speziellen Dauer beschrieben werden.

Replay von Event Histories

Die erneute Ausführung eines Programms unter der Steuerung durch die Ereignisprotokolle nennt sich Replay. Wenn die Prozesssteuerung vollständig umgesetzt wurde, können Programmabläufe beliebig oft exakt reproduziert werden. Dieses Verfahren ist allerdings nur sinnvoll, wenn durch die Wiederholungen neue Informationen gesammelt oder anderen Einsichten in die Programmabläufe gewonnen werden. Einerseits kann dies durch eine erneute Ausführung im Debug Mode unter Zuhilfenahme eines sequentiellen Debuggers für jeden Prozess erfolgen. Auf diese Weise ist es möglich, Einblick in die internen Status der Prozesse zu gewinnen. Des Weiteren erlauben einige Systeme auch die isolierte Betrachtung eines dedizierte Prozesses unter Simulation der anderen Prozesse anhand der Event History.

Im Gegensatz dazu kann das Programm auch anhand des von der Event History erstellen Protokolls gesteuert ausgeführt werden. Dadurch können veränderte Programmteile und Algorithmen unter exakt den Bedingungen getestet werden unter denen die vorhergehende Ausführung fehlgeschlagen ist. Ebenfalls können Rückschlüsse darauf getroffen werden, ob ein Fehler deterministisch reproduzierbar auftritt oder nicht.

Es bleibt im allgemeinen allerdings bei all diesen Methoden zu beachten, dass die angewandten Synchronisationsmechanismen die Programmausführung und -performance stark verlangsamen bzw. beeinträchtigen. Allerdings kann dieser Effekt durch entsprechende Optimierungen weitestgehend eliminiert werden ([8] geben an, den Overhead bei speziellen Problemen auf unter ein Prozent drücken zu können).

Überprüfen und Umwandeln von Ereignisprotokollen

Eine Vielzahl von Debugging-Systemen vergleichen die gesammelten Protokolldaten mit einer vorher bestimmten Menge an Merkmalen und analysieren dabei die History bereits zur Laufzeit. Eine Verletzung eines solchen vordefinierten Prädikats kann wiederum andere Debug-Maßnahmen auslösen. Zum Einsatz kommen diese Möglichkeiten etwa in DISDEB, das eine Vielzahl programmierbarer Debug-Hilfen anbietet. Es basiert auf der Definition von Events in der Form „Ein Prozess P mit entsprechender Berechtigung greift auf die Speicherstelle X zu um einen Wert V zu schreiben/lesen“. Darauf aufsetzend kann der Debugger durch Ereignisse gesteuert werden; dies geschieht durch Aufrufe der Form „ wenn ein Event E1 auftritt, dann zeige den Zähler Z an und stoppe Prozess N“ [10]. Auf dieselbe Art und Weise können Traces initialisiert oder beendet und Timer verändert werden.

Das HARD System für ADA Programme arbeitet ähnlich, wo die Prädikate in speziellen ADA-Tasks, den sogenannten D Tasks, enthalten sind. Um sie aufzurufen, werden entsprechende Anweisungen von Hand in den Code eingefügt, die Informationen über den Systemstatus und die Ausführung an die D Tasks zurückliefern, die unter deren Verwendung entsprechende Methoden aufrufen, um den aktuellen Programmstatus anzuzeigen oder zu modifizieren. Da diese speziellen Tasks komplett auf ADA basieren, muss der Programmierer keine spezielle Sprache erlernen und kann die entsprechenden Spezifikationen leicht erstellen. Anders als die bisherigen Ansätze, setzen die nun folgenden Systeme nicht auf die die Überwachung von Ereignisströmen sondern überprüfen direkt die Programmspezifikationen. Zwar muss der Entwickler in jedem Fall eine neue Sprache erlernen, jedoch stehen ihm zugleich auch Möglichkeiten zur Programmspezifikation und -verifikation zur Verfügung. Unterschiedlich behandeln die entsprechenden Lösungen lediglich die, unterschiedlich komplexe, Spezifikation von Eventformeln; zumeist geschieht dies über sequentielle und parallele Kombination von Ereignissen.

IDD [6] verwendet dabei eine intervall-logische Spezifikation des zu analysierenden Programms die mit dem tatsächlichen Verhalten des Programms verglichen wird. Bei Verletzung von Spezifikationen wird das Programm angehalten um es genauer analysieren zu können, wobei die Ausführung der Idee der Temporallogik folgend als Sequenz verschiedener Zustände interpretiert wird. Hauptoperatoren bilden Always und Eventually, mit der Bedeutung, dass eine Eigenschaft entweder immer gilt oder irgendwann einmal gelten wird. Allgemein lässt sich sagen, dass die Intervall-Logik die Temporallogik in dedizierte Zeitabschnitte „zerhackt“.

Der ECSP-Debugger von [1] hingegen arbeitet mit so genannten Verhaltens-Spezifikationen, die versuchen, das komplette erlaubte Kommunikationsverhalten eines Prozesses zu beschreiben. Dabei überprüft das System verschiedenste Kommunikationspfade und kann bestimmte Assertions für den Prozesszustand beinhalten. Im Falle eines Fehlers wird die Kontrolle an den User zurückgegeben, damit dieser (auch mit anderen Tools) den Fehler analysieren kann. Um die Ordnungs- und Reihenfolgenproblematik von Spezifikationen zu umgehen, setzt ECSP voraus, dass ich jede Spezifikation nur auf Ereignisse von einem Prozess beziehen kann.

[3] beschreiben für ihr TSL System einen Aufbau, der die Spezifikationen automatisch mit den Ereignissen, die von einem ADA-Task Programm erzeugt werden, vergleicht. Eine TSL Spezifikation besteht dabei aus einer Anweisung in der Form „wenn etwas passiert, dann ereignet sich etwas anderes vor einem weiteren Ereignis“, wobei es sich bei jedem der Teile um einen Ereignis-Formel handelt. Darüberhinaus enthält TSL Wildcards, die es erlauben, mit einer einzelnen Spezifikation mehrere Aufgaben zu behandeln. Eine weitere Abstraktion ist durch den Einsatz von Macros für Ereignis-Unterformeln erreichbar. Herausragendstes Merkmal dieser Idee ist jedoch der Einsatz der ADA Semantik, durch den auch in verteilten Systemen Ereignispaare in den Protokollen in der richtige Reihenfolge auftreten.

Zuletzt wählt die Ereignis-Beschreibungssprache (Event Description Language, EDL) einen leicht unterschiedlichen Ansatz [2], in dem sie nicht die Unterschiede zwischen Ereignissen und deren Spezifikationen erkennt, sondern eine Methode anbietet, um verschiedene Stufen von abstrakten Ereignissen zu definieren, die auf Basis der einfachen Events erstellt werden, die im Programm ablaufen. Jede komplexe Ereignisformel auf höherem Level ist basierend auf anderen, nieder-levligen Ereignissen.

Was jedoch alle diese Methoden gemein haben ist, dass sie alle über vereinfachende Restriktionen verfügen. In EDL ist dies zum Beispiel die Voraussetzung einer globalen Zeit; darüberhinaus ist das Erkennen von Ereignissen ein möglicher Flaschenhals und eine gewisse Unklarheit kann entstehen, wenn ein Event niederen Levels in mehreren Ereignissen mit höherem Level verwendet werden kann. Im Gegensatz dazu erfordert der TSL Mechanismus zur Überprüfung von Spezifikationen eine linear geordnete Abfolge von Ereignissen und stellt selbst die Schwachstelle in der Anwendung bezüglich Performance dar. Anders wiederum verhält sich IDD, bei dem Ereignisse auf Broadcasts auf z.B. einem Netzwerk beruhen.

Statische Programmanalyse In speziellen Fällen kann es vorkommen, dass der Probe-Effekt nicht durch die bisher genannten Verfahren umgangen werden kann; dem Softwareentwickler bleiben dann nur noch wenige Möglichkeiten, das Programm sinnvolle analysieren und debuggen zu können. Der Einsatz von statischen Analysetechniken scheint sich hierbei zum Aufspüren einzelner Fehlerklassen zu bewähren. Im Gegensatz zu den formalen Analysemethoden basiert dieser Ansatz auf der Idee, sicherzustellen, dass das Programm unterschiedliche, vorher fest definierte Zustände, die auf einen Programmfehler schließen lassen, nicht einnehmen kann. Insbesondere lassen sich dadurch Synchronisationsprobleme und Fehler in der (gleichzeitigen) Verwendung von Daten aufspüren. Zu den behandelbaren Synchronisationsfehlern zählen unter anderem Deadlocks und Probleme mit Zuständen die ffür die Ewigkeitäuf einen anderen Prozess warten würden; Datenzugriffsprobleme bezeichnen vor allem normale sequentielle Zugriffsfehler wie das lesen einer uninitialisierten Variable und Schreibfehler durch das gleichzeitige aktualisieren einer Variable durch zwei Prozesse. Dazu werden zwei grundlegen verschiedene Problemstellungen betrachtet. Zum einen das Anwenden von Datenflussanalysen um Fehler bei der Verwendung von Variablen und Daten zu erkennen; zum anderen festzustellen, ob zwei Aufrufe in einem parallelen Programm auch tatsächlich parallel ausgeführt werden können.

Datenflussanalyse in parallelen Programmen

Praktisch alle weiterührenden Ansätze zur Datenflussanalyse in parallelen Programmen stützen sich auf die Arbeit von Taylor und Osterweil [13], deren vorgeschlagener Algorithmus mit vier Datennutzungsgruppen agiert: gen, kill, live und avail. Darauf aufbauend spezifizieren sie einen simplen Algorithmus für ein Prozesssynchronisationsmodell, in dem ein Prozess einen anderen dazu veranlassen kann, eine Aufgabe zu starten und wiederum auf deren Ergebnis zu warten, wobei der Ausgangsprozess nicht parallel zu sich selbst ausgeführt werden kann. Diese und alle anderen Formen von rekursiven Aufrufen sind zudem

ebenfalls nicht erlaubt. Zusammen mit anderen erweiternden Algorithmen sind sie somit in der Lage, folgende Anomalien zu erkennen: Das Referenzieren einer unitialisierten Variable; Das nebenläufige referenzieren einer Variable die gleichzeitig erst definiert wird; die Einführung einer Variable die im späteren Verlauf nie referenziert wird; Variablen mit einem indeterministischen Wert; Prozesse, die auf die Terminierung eines ungetakteten Prozesses warten; Prozesse die auf einen andern Prozess warten, der unter Garantie bereits terminiert hat; und Prozesse die parallel zu sich selbst ausgeführt werden sollen.

C Eclipse Parallel Tool Platform

Nachdem nun einige Herangehensweisen an die Probleme des parallelen Debuggens erörtert worden sind soll die Betrachtung eines aktuellen Toolkits nicht fehlen.

C.1 PTP Allgemein

Die auf Eclipse aufsetzende Parallel Tool Platform hat es sich zum Ziel gesetzt, eine Opensource-Lösung auf industriell einsetzbarem Niveau zu schaffen, die eine hoch integrierte Umgebung für die Entwicklung von parallelen Anwendungen schafft. In diesem Rahmen arbeite das Projekt an einer standardisierten, portablen IDE die eine Vielzahl an parallelen Architekturen und Laufzeitumgebungen unterstützt, einem parallelen Debugger (auf den im weiteren das Hauptaugenmerk gerichtet sein soll), einer Unterstützung zur einfachen Integration weiterer Tools für die Entwicklung nebenläufiger Software und eine Umgebung, zur Vereinfachung der Userinteraktion mit parallelen Systemen.

C.2 PTP Parallel Debugger

Ein in der Eclipse Parallel Tool Platform enthaltener Bestandteil ist der Parallele Tool Platform Debugger oder auch Parallel Debugger.

C.2.1 Funktionsumfang und Aufbau Dabei handelt es sich um eine Erweiterung des bereits existierenden Platform Debuging Frameworks der PTP, dessen Ziel es ist, Funktionalitäten zum Debuggen zur Laufzeit eines mehrfädigen Programms zur Verfügung zu stellen. Im Idealfall soll es in späteren Versionen möglich sein, ein Programm entweder einfach im Debug-Modus zu starten, oder einen Debugger mit wenigen Klicks an ein sehr großes paralleles Programm (mehrere Tausend nebenläufige Prozesse) anzufügen. Dabei stützt sich der parallele Debugger auf die Dienste, die ihm die parallele Ausführungsumgebung (parallel execution environment) der PTP zur Verfügung stellt um eine parallele Anwendung zu starten oder sich an ihr anzudocken. Dazu sind eine ganze Reihe an Erweiterungen des bereits vorhandenen Debug Modells und der Debug Oberfläche notwendig, um diese Zusatzfunktionalitäten bereitstellen und bedienen zu können.

Der parallel Debugger ist dabei wie bei Eclipse Projekten gewohnt in die Eclipseoberfläche als Plugin integriert. Die traditionellen Debugging-Ansätze für parallele Programme wie B.I.S.R (das setzen eines Breakpoints, das analysieren des Fehlers -inspect-, das Ausführen des Schrittes -step-, und das Wiederholen dieses Schrittes -repeat-) stoßen bei modernen mehrfädigen Programmen mit tausenden von Prozessen an ihre Grenzen; zudem scheitert der Einsatz mit dort verwendeten Hochsprachen zumeist an der mangelnden Integration in die entsprechenden Entwicklungsumgebungen.

Diese Probleme können durch andere Debugging-Verfahren, wie das Identifizieren des Auftretens von Fehlern durch vordefinierte Prädikate und Zusicherungen sowie Nachrichten- und Daten-Patterns oder die genauere Lokalisierung der Fehlerdaten durch die Darstellung des globalen Programmzustandes und den Vergleich bzw. die Suche nach diesen Mustern in der gesamten Applikation adressiert werden. Zudem wäre eine Funktion zum beliebigen „Zurückspulen“ und der erneuten Wiedergabe des Programmablaufs wünschenswert. Diese ganzen Funktionen müssten aber auch in heterogenen Systemen mit unter dem Einsatz verschiedener Sprachen funktionieren.

Die Parallel Debugging Platform versucht diese Aspekte durch eine reichhaltige Benutzeroberfläche und die Integration der genannten Verfahren zu lösen.

C.2.2 Lösungsansätze für Probleme im Rahmen des parallel Debugging Der Parallel Debugger ist für Systeme mit zwischen zwei und 128 000 Prozessen ausgelegt, wobei er ebenfalls Architekturen berücksichtigt, die Multi-Prozesse und Strangarchitekturen vereinigen. Die Debug-Operationen können dabei auf allen, einer Teilmenge, oder einem speziellen der Prozesse ausgeführt werden, wobei die gegenseitige Abhängigkeit der Prozesse untereinander berücksichtigt ist. Ebenso kann der Debugger mit der Verwendung von mehreren Programmiersprachen wie etwa Fortran und C im selben Projekt umgehen.

Von den Programmiermodellen wird bisher nur das nachrichtenbasierte Message Passing Model unterstützt, bei dem einzelne Prozesse direkt über Nachrichten miteinander kommunizieren. Dazu sind explizite Sende- und Empfangsoperationen notwendig. Das Konzept des Shared Memory Models, in dem die Datenstrukturen zwischen den Prozessen verteilt sind, die aber spezielle Mechanismen von sich im Zugriff befindlichen Speicherstellen und atomare Operationsunterteilungen voraussetzen, sollen erst in einer späteren Version des parallel Debuggers unterstützt werden.

Zum eigentlichen Debuggen stehen Breakpoint zur Verfügung, die zur Synchronisierung über alle ablaufenden Prozesse verteilt sind. Nach Starten des Programms wird es daraufhin Schritt für Schritt abgearbeitet bis ein Breakpoint oder ein Fehler erreicht wird. Daraufhin werden alle Prozesse angehalten, da das durchlaufen eines einzelnen Prozesses normalerweise an dessen Abhängigkeiten zu anderen Prozessen scheitert. Darauffolgend werden das Speicherabbild und die gesammelten Protokolle aller oder eines Teils der Prozesse analysiert. Dabei setzt der PTP Debugger vor allem auf den Vergleich der Daten der Prozesse untereinander, die darüber Aufschluss geben können, wieso ein spezieller

Prozess einen Fehler hervorgerufen hat und ein anderer nicht. Die Datenrepräsentation finde dabei in einem architekturunabhängigen Format, dem sogenannten AIF, statt, das unabhängig von Sprachspezifika, der Wort- und Zeichengröße und anderer Restriktionen plattformübergreifend eingesetzt werden kann. Die Ausdrucksmächtigkeit leidet dabei allerdings nicht im Geringsten, das alle Datentypen und Werte sowie komplexe Konstrukte wie verkettete Listen abgebildet werden können. Zudem werden alle Typen der Sprachen C, C++ und Fortran und teilweise auch Java unterstützt.

Die Benutzeroberfläche versucht, dem Entwickler möglichst einfachen Zugriff auf alle entscheidenden Features zu bieten. Dabei bietet der sogenannte Parallel Debug View eine globale Ansicht aller Prozesse und der aggregierten Prozessmengen. Der Zugriff auf diese einzelnen Elemente wird durch das Einblenden der nötigen Information über Tooltips erleichtert. Zudem ist das An- und Abmelden eines Prozesses um ihn in der Debug-Ansicht genauer betrachten zu können über wenige Klicks möglich. Das setzen der Breakpoints ist entweder global möglich, wobei dieser Haltepunkt für jeden Prozess ungeachtet seines Aufgabentyps, deren Größe oder Priorität gilt. Alternativ kann ein Haltepunkt auch nur für eine dedizierte Auswahl an Prozessen gewählt werden, die alle zum Beispiel eine spezielle Aufgabe haben. Die Unterscheidung der einzelnen Prozesse und deren Zuordnung zu den Prozessen ist anhand von Farbmarkierungen möglich. Eine weitere optische Vereinfachung für die Arbeit mit dem Debugger-Interface stellen Markierungen für die aktuelle Zeile dar. Es sind dabei mehrere dieser Current Line Markers erlaubt, wobei unterschiedliche von Ihnen für angemeldete und unangemeldete Prozesse zur Verfügung stehen.

D Zusammenfassung

Zusammenfassend lässt sich sagen, dass mehre Ansätze existieren, um mit den Problemen des parallelen Debuggen, wie dem Probe-Effekt, umzugehen. Jedoch scheinen sich bis heute vor allem die Verwendung von Breakpoints durchgesetzt zu haben. Die Eclipse Parallele Tool Platform bietet hierzu einen leistungsstarken parallelen Debugger an, der sich nahtlos in Eclipse integriert und mit seiner durchdachten Benutzeroberfläche das Debuggen von bis zu 128 000 Prozessen über (Programmier-)Sprachgrenzen hinweg ermöglicht.

Literatur

[1] F. Baiardi, N. D. Francesco, E. Matteoli, S. Stefanini, and G. Vaglini. Development of a debugger for a concurrent language. *SIGSOFT Softw. Eng. Notes*, 8(4):98–106, 1983.

[2] P. Bates and J. C. Wileden. An approach to high-level debugging of distributed systems: preliminary draft. *SIGPLAN Not.*, 18(8):107–111, 1983.

[3] A. R. Brindle, R. N. Taylor, and D. F. Martin. A debugger for ada tasking. *IEEE Trans. Softw. Eng.*, 15(3):293–304, 1989.

[4] J. Gait. A debugger for concurrent programs. *Softw. Pract. Exper.*, 15(6):539–554, 1985.

[5] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *ACM Trans. Comput. Syst.*, 5(2):121–150, 1987.
[6] P. K. H. Jr., D. Heimbigner, and R. King. Idd: An interactive distributed debugger. In *ICDCS*, pages 498–506, 1985.
[7] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
[8] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, 1987.
[9] C. H. LeDoux and J. D. Stott Parker. Saving traces for ada debugging. In *SIGAda '85: Proceedings of the 1985 annual ACM SIGAda international conference on Ada*, pages 97–108, New York, NY, USA, 1985. Cambridge University Press.
[10] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, 1989.
[11] R. Snodgrass. Monitoring in a software development environment: A relational approach. *SIGPLAN Not.*, 19(5):124–131, 1984.
[12] A. S. Tanenbaum. *Computer networks: 2nd edition.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.
[13] R. N. Taylor and L. J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Trans. Softw. Eng.*, 6(3):265–278, 1980.