Parallelität durch Abstraktion

Andreas Wilhelm

Hauptseminar - Multicore-Programmierung - WS 2010
Lehrstuhl für Programmierung

9. November 2010

Zusammenfassung

Die IT-Industrie benötigt dringende Hilfe durch neue Verfahren, um einen Großteil der Entwickler für Parallelprogrammierung zu begeistern. Nebenläufige Systeme sind mittlerweile für jeden verfügbar, doch die theoretische Leistung wird selten genutzt, weil die Entwicklung von Software dafür noch zu komplex ist. Programmierer müssen sich mit vielen unangenehmen Herausforderungen wie *Deadlocks*, *Race Conditions* oder aufwändigen Sperrmechanismen auseinandersetzen um Software für Parallelcomputer zu implementieren.
In dieser Arbeit wird eine Auswahl an unterschiedlichen Möglichkeiten gezeigt, um dem Programmierer die Hürden der Parallelität besser meistern zu lassen.

Inhaltsverzeichnis

1 Einführung

Der rasche Wechsel zu parallelen Multicore-Prozessoren in den letzten Jahren ist ein klares Anzeichen für einen Wandel in der Informatik. Traditionelle Verfahren zur Steigerung der Rechengeschwindigkeit durch mehr Transistoren pro Chip und die Erhöhung der Taktraten haben eine Grenze erreicht: Physikalische Naturgesetze fordern unakzeptablen Stromverbrauch und hohe Temperaturen [2].

Als Lösung und nächsten Evolutionsschritt nutzen Prozessorhersteller die technischen Möglichkeiten, um mehrere Kerne auf einem Prozessorchip zu integrieren. Die dadurch gewonnene Rechenkapazität wird allerdings bisher nur in seltenen Fällen ausgenutzt. Dafür ist neben zusätzlichem Koordinationsaufwand vor allem der Mangel an geeigneter Software verantwortlich. Diese wurde bisher fast ausschließlich für Bereiche wie Hochleistungsrechnen, Server, Grafikbeschleuniger und Eingebettete Systeme entwickelt. Der Durchbruch im Client-Sektor blieb jedoch aus. Die Industrie hatte aufgrund der ständigen Beschleunigung von Programmen durch steigende Taktraten keinen Bedarf, die bewährte sequentielle Softwarearchitektur zu verändern [13]. Um die Vorteile von neuen Multicore-Systemen für bestehende Anwendungen zu nutzen, müssen unzählige Quellcodezeilen überarbeitet werden. Nebenläufige Software ist nicht nur schwieriger zu warten, Threads, Synchronisation, Sperren, Race Conditions oder Deadlocks machen Qualitätssicherung äußerst unsicher. Die Portierung paralleler Anwendungen auf unterschiedliche Hardware-Architekturen ist ebenfalls nicht trivial. Die Software muss eine beliebige Anzahl von Kernen, verschiedene Speichermodelle und spezielle Instruktionen bzw. Optimierungen berücksichtigen, um von der theoretischen Rechenleistung zu profitieren. Die Folge ist eine immer größer werdende Kluft zwischen Hardware- und Software-Herstellern. Während die Anzahl an Kernen je Prozessor stets neue Rekordmarken erreicht, können nur wenige Anwendungen das Potential der Hardware-Performanz ausnutzen.

Multithread-Microprozessoren, Multicore-CPUs, Multiprozessor-PCs, Cluster oder parallele Spielekonsolen, in naher Zukunft wird nahezu jeder Computer mit paralleler Hardware ausgestattet sein. Damit diese mit voller Leistung betrieben werden können, müssen jedoch die Anwendungen parallel ausgeführt werden. Aber wer erstellt diese Programme?

1.1 Ziele

Die vorliegende Arbeit wurde im Rahmen des Hauptseminars *Multicore-Programmierung* erstellt. Es werden darin verschiedene Ansätze vorgestellt, welche den einfachen Zugang zur Erstellung nebenläufiger Systeme durch Abstraktion ermöglichen sollen. Die Auswahl der Themen sowie die Erläuterung stellt weder den Anspruch vollständig zu sein, noch bietet sie konkrete Anwendungsleitfäden. Das Ziel ist vielmehr, einen Einblick in aktuelle Forschungsschwerpunkte in dem Bereich zu erhalten. Der Leser soll die Notwendigkeit von Abstraktion im Bezug auf Parallelität erkennen und sich durch verschiedene Ansätze eine eigene Meinung über die zukünftige Parallelprogrammierung bilden können. Bei der Ausarbeitung wurde besonders auf eine möglichst breite Auswahl der Themen Wert gelegt. Es soll deutlich werden, dass eine mögliche Zukunft von parallelen Systemen mit hoher Wahrscheinlichkeit Innovationen in allen Bereichen der Informatik erfordert. Software- sowie Hardware-Spezialisten sind aufgefordert, miteinander solche Lösungen zu suchen. Die Arbeit soll auch dazu ermuntern.

1.2 Abstraktion

"[...] general programmers, especially professional programmers who "have lives", ignore parallel computers" [16]

Nach wie vor muss sich ein Software-Entwickler großen Herausforderungen stellen, um effiziente nebenläufige Softwaresysteme zu erstellen. Neben den semantischen Aspekten ähnlich zur sequentiellen Programmierung müssen zusätzlich Punkte wie z.B. Abhängigkeiten, Synchronisation, Deadlocks oder Race Conditions berücksichtigt werden. Um Parallelisierung in allen

Bereichen universal verwenden zu können, dürfen solche fehleranfälligen Aspekte keine Rolle mehr spielen. Es werden Werkzeuge benötigt, welche die Möglichkeiten für Fehler reduzieren und gleichzeitig die Leistungsfähigkeit paralleler Hardware nutzbar machen. Das Thema dieser Arbeit beschäftigt sich mit solchen Werkzeugen. Abstraktion von Parallelität soll all den genannten Problemen entgegenwirken und folgende Punkte ermöglichen:

- Reduzierung von Komplexität
- Steigerung der Effektivität
- Erhöhung von Portabilität
- Bessere Skalierbarkeit

2 Zugänge zur Abstraktion

Seit der Verwendung der ersten parallelen Rechner existiert der Wunsch, die Koordination von "gleichzeitigen" Aufgaben so einfach wie möglich zu gestalten. Es wurden viele unterschiedliche Verfahren verfolgt, wie z.B. Hardware-Abstraktion, implizit parallele Programmiersprachen oder portable Bibliotheken, die Kommunikation über Nachrichtenaustausch unterstützen [16]. Dennoch blieb der Durchbruch aus, sodass der Großteil derzeitiger Softwareentwicklung im Client-Bereich in traditionell sequentieller Form abläuft.

Seitdem klar ist, dass das "free lunch"[1] vorbei ist, hat die Computer-Industrie die Notwendigkeit neuer Modelle zum Entwurf von nebenläufigen Softwaresystemen erkannt. Daraus erwächst eine völlig neue Ausgangssituation für die parallele Programmierlandschaft. Aktuelle Initiativen, wie das *MARC*-Programm [2] oder die *UPCRC*-Initiative [3] der Firma Intel mit Zusammenschluss mehrerer renommierter Firmen und Universitäten, zeigen das hohe Interesse an neuen Ideen, um das Potential von Multicore-Systemen bequem nutzen zu können.

Es mangelt nicht an Forschungsfragen, jedoch ist bisher nicht absehbar wohin der Weg auf der Suche nach einem Standard für die Parallelprogrammierung führt. Daher ist es nicht verwunderlich, dass sich viele Forschungsgruppen in diesem Bereich möglichst breit positionieren wollen (Abbildung 1).

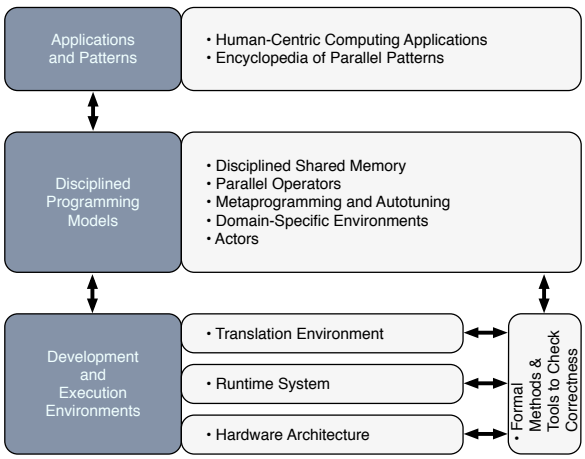

Abbildung 1: Überblick über die Forschungsagenda der UPCRC/Illinois [13]

[1] The Free Lunch Is Over, Herb Sutter (`http://www.gotw.ca/publications/concurrency-ddj.htm`)
[2] *MARC*-Programm (`http://communities.intel.com/community/marc`)
[3] *UPCRC*-Initiative (`http://www.intel.com/pressroom/kits/upcrc/`)

2.1 Anwendungen und Muster

Parallelprogrammierung hat eine lange Geschichte in Bereichen wie Hochleistungsrechnen oder Eingebettete Systeme. Die Software-Entwickler dieser Sparten akzeptieren die notwendigen Programmierparadigmen und nutzen die gewonnene Expertise zur Erstellung leistungsfähiger Systeme. Leider fehlt diese notwendige Akzeptanz bisher im Client-Bereich. Das liegt natürlich zu einem bestimmten Grad an den verfügbaren technischen Möglichkeiten, weshalb der Bedarf an nebenläufiger Software nicht allzu hoch war. Spätestens die Ära der Multicore-Prozessoren ändert diese Tatsache jedoch grundlegend. Es sind Innovationen notwendig, um die Entwicklung paralleler Client-Software komfortabel zu ermöglichen. Das größte Hindernis dabei ist die unbeantwortete Frage, wie sich Parallelität am Besten ausdrücken lässt. Bevor diese Frage nicht geklärt ist, wäre es nicht sinnvoll die Ausrichtung von Forschung und Industrie von bestehendem Quellcode abhängig zu machen. Vielmehr sollten Argumente auf hohen Abstraktionsstufen bezüglich Anwendungsanforderungen verwendet werden [2].
In diesem Kapitel werden zwei mögliche Ansätze erläutert, die durch Abstraktion einen einfachen Zugang zu Parallelität mit sich bringen. Domänenspezifische Umgebungen tragen dabei zur vorher erwähnten Akzeptanzsteigerung in bestimmten Anwendungsgebieten bei. Parallele Entwurfsmuster und parallele Skelette hingegen haben einen Einfluss auf sämtliche Abstraktionsstufen von nebenläufiger Programmierung.

2.1.1 Domänenspezifische Umgebungen

Durch Abstraktion in der Parallelprogrammierung sollte dem Entwickler die Hürde anspruchsvoller Koordination von parallelen Tasks abgenommen werden. Innerhalb bestimmter Anwendungsgebiete, in denen domänenspezifische Umgebungen (DSEs) existieren, kann die Parallelisierung sogar vollkommen vor dem Anwender versteckt werden. DSEs sind hierbei eine Sammlung von domänenspezifischen Werkzeugen, welche für typische Operationen des Anwendungsbereiches verwendet werden. Das Fachwissen in den entsprechenden Bereichen kann zur Erzeugung mächtiger Operatoren verwendet werden. Gerade bei kleinen Umgebungen wird dadurch ein hoher Grad an Ausdrucksstärke und Effektivität erreicht [9].

Auch wenn domänenspezifische Umgebungen und insbesondere ihre domänenspezifischen Sprachen (DSLs) nicht neu sind, so erfahren sie aufgrund der aktuellen Bewegung hin zu Multicore-Prozessoren einen enormen Aufschwung. Das Ziel vieler DSEs ist idealerweise, dass Anwendungsentwickler keine expliziten Methoden zur Parallelprogrammierung lernen müssen. Durch die Verwendung der Umgebungen sollen die erstellten Programme automatisch auf mehreren Kernen ausgeführt werden können. Damit dies möglich ist, muss die Nebenläufigkeit implizit im Hintergrund stattfinden. Die schwierigen Aufgaben können somit von wenigen Experten mit domänenspezifischen Fachwissen und Erfahrung in Parallelprogrammierung übernommen werden. Mittlerweile gibt es einige DSEs, welche die parallele Ausführung ihrer Operationen unterstützen. Beispiele sind Bibliotheken wie Apple's *Core Audio*, Google's *MapReduce* oder die Sprache MATLAB®. Letzteres wird im Folgenden als Beispiel verwendet, um einige Sprachkonstrukte zur Erläuterung vorzustellen.

Beispiel MATLAB ist eine Sprache (und gleichzeitig eine Entwicklungsumgebung) für den Einsatz in vielen technischen Gebieten wie z.B. Bild- und Signalverarbeitung, Finanzmathematik oder Bioinformatik. MATLAB kann leicht durch fachspezifische Routinen in Form von Add-Ons, genannt "Toolboxes", erweitert werden. Durch den Bedarf an Software für Multicore-Hardware wurde im Jahr 2004 eine Erweiterung für MATLAB präsentiert: die *Parallel Computing Toolbox*™. Die Toolbox erlaubt dem Benutzer bis zu 8 *Worker*-Objekte für die nebenläufige Verarbeitung einer Aufgabe zu verwenden. Falls der Desktop-Rechner, auf dem MATLAB ausgeführt wird, mehrere Prozessoren (bzw. Prozessorkerne) beinhaltet, so aktivieren und verwenden die *Worker* diese. Eine zusätzliche Erweiterung durch die *Parallel Computing Toolbox* stellt der MATLAB *Distributed Computing Server* (DCS) dar. Falls der Client Zugriff auf einen Rechner-Cluster besitzt, so lassen sich dadurch auch Aufgaben auf *Worker*-Objekten des Clusters ausführen. Die Verteilung der Aufgaben übernimmt ein Scheduler des *Distributed*

Computing Server.
Je nach Art der Parallelität (lokal oder auf Cluster-Rechnern) werden unterschiedliche Kommunikationstechniken verwendet. Bei lokaler Ausführung erfolgt diese über gemeinsamen Speicher ähnlich zu *OpenMP*; bei Verteilung der Aufgaben über den Scheduler wird eine *MPI*-Version (MPICH2) verwendet.

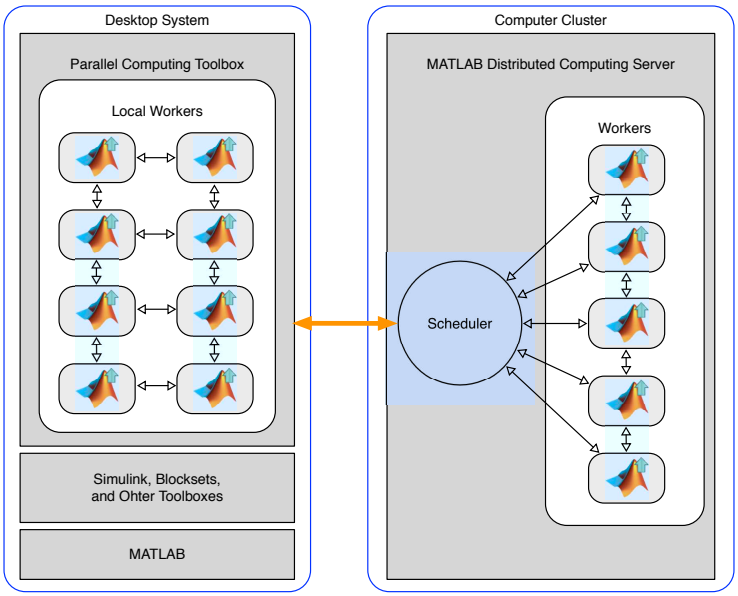

Abbildung 2: MATLAB - Parallel Computing Toolbox

Die *Parallel Computing Toolbox* ermöglicht neben expliziten Techniken zur Parallelisierung, wie z.B. taskübergreifende Nachrichtenübertragung oder parallele for-Schleifen, auch implizites Multithreading. Im Folgenden werden zwei Möglichkeiten vorgestellt, wie man MATLAB-Programme nebenläufig ausführen kann, ohne Änderungen am sequentiellen Programmcode vornehmen zu müssen.

Worker-Methode Durch die Einführung der *Worker*-Objekte lassen sich erstellte MATLAB-Funktionen (in Form von `.m`-Dateien) automatisch auf mehrere Hardware-Threads verteilen. Zur parallelen Ausführungen müssen drei Schritte durchgeführt werden:

1. Eine Anzahl an (lokalen) Workern anfordern (max. 8).
2. Die gewünschte MATLAB-Funktion wie gewohnt aufrufen. Bei Bedarf werden Aufgaben auf die verfügbaren Worker verteilt.
3. Freigeben der Worker.

Eine beispielhafte Implementierung wird in Listing 1 präsentiert. Es wird ein lokaler Pool von vier *Workern* angefordert. Der hier verwendete Parameter `local` legt die Konfiguration der *Parallel Computing Toolbox* fest (Standard: `local`). Durch Änderung der Konfiguration kann z.B. die parallele Berechnung auf Cluster-Computern erfolgen.
MATLAB stellt sicher, dass das Resultat der Funktion identisch mit der sequentiellen Ausführung ist. Die Berechnungen werden bei Bedarf auf mehrere Kerne verteilt, ohne explizite Steuerung des Benutzers. Je nach Aufbau von MATLAB-Funktionen wird die Laufzeit der Ausführung gegenüber der sequentiellen Variante deutlich sinken. Dafür wird der Quellcode der Funktionen analysiert und dabei Variablen auf mögliche Parallelisierung hin untersucht. Dies ist für den Benutzer sehr komfortabel, leider können einige parallelisierbaren Situationen dabei nicht

erkannt werden[4]. Falls beispielsweise Arrays in der Funktion verarbeitet werden, so verteilt MATLAB die Arbeitspakete nur dann auf mehrere Threads, falls bei jeder Iteration fest definierte, unabhängige Teile des Arrays verarbeitet werden.

```
matlabpool open local 4
  calculate
matlabpool close
```

Listing 1: Worker-Methode

SPMD-Methode SPMD steht für **S**ingle **P**rogram, **M**ultiple **Data** und erlaubt dem Benutzer die nebenläufige Bearbeitung einer Menge von Daten durch ein ein einzelnes Programm. MATLAB stellt dazu bereits über 150 Operatoren für die Verwendung auf Arrays zur Verfügung (siehe Abbildung 3).
Es ist auch möglich, eigene Operationen auf einer Menge von Daten zu definieren. Der Entwickler muss sich weder um das Scheduling, noch um eine geeignete Technik für die Parallelisierung kümmern. Die Aufgaben erledigt die *Parallel Processing Toolbox*, indem sie zusätzliche Syntax zur Verfügung stellt. In dem Beispiel aus Listing 2 wird eine neue Funktion `calcspmd` erstellt, welche eine existierende Rechenfunktion `MyFunction` auf ein Array anwendet und das Resultat zurückliefert. Das Schlüsselwort `spmd` leitet einen Ausführungsbereich ein, der auf einzelnen *Worker*-Objekten ausgeführt wird. Der Parameter `n` legt die Anzahl der zu verwendenden *Worker* fest. Die Aufteilung der Arbeitspakete kann mit Hilfe der der Variable `labindex` berücksichtigt werden, welche die Nummer des aktiven Workers beinhaltet. Da die einzelnen Threads unterschiedliche Speicherbereiche besitzen, müssen die Rückgabewerte über Shared-Memory gesammelt werden. Das erledigt die Funktion `gplus`, indem sie die Rückgabewerte der *Worker* addiert.
Der Benutzer der Operationen kann, ohne zusätzliche Syntax zu verwenden, Berechnungen auf einer Menge von Daten parallel ausführen. Die Komplexität wird in den Aufgabenbereich des Operations-Entwicklers verschoben.

```
function value = calcspmd (n, x)
  value = 0
  spmd (n)
    len = length(x) / n
    labdata = x((labindex - 1) * len: labindex * len - 1)
    value = value + gplus( MyFunction(labdata) )
  end
end
```

Listing 2: SPMD-Methode

Fazit Abstraktion von Parallelisierung innerhalb domänenspezifischer Umgebungen wird hauptsächlich über zwei Techniken realisiert. Zum Einen können spezielle Compiler verwendet werden, um mögliche Stellen für gleichzeitige Ausführungen zu identifizieren. Gegenüber den entsprechenden Komponenten üblicher Umgebungen, wie z.B. C++ oder Java, besitzen sie einen entscheidenden Vorteil: Sie kennen ihr genaues Einsatzgebiet und damit die möglichen Optimierungen hinsichtlich Nebenläufigkeit. Ein Compiler kann zum Beispiel Programme mit höherer Effektivität erzeugen, falls er im Voraus weiß, dass zwei verschiedene Vektoren stets orthogonal sind oder dass bestimmte Operationen kommutativ sind.
Zum Anderen können mächtige Operatoren (Bsp.: Array-Operator `*` bei Matlab), verwendet werden um Parallelität durch Abstraktion zu erreichen. Auch hierbei dient domänenspezifisches Wissen zur Auswahl und Optimierung solcher Funktionen. Für den Entwickler ist es eine der komfortabelsten Möglichkeiten, seine Multicore-Hardware zu nutzen. Allerdings bieten die

[4]MATLAB Parallel Computing, John Burkardt (`http://people.sc.fsu.edu/~burkardt/presentations/fdimatlab2009.pdf`)

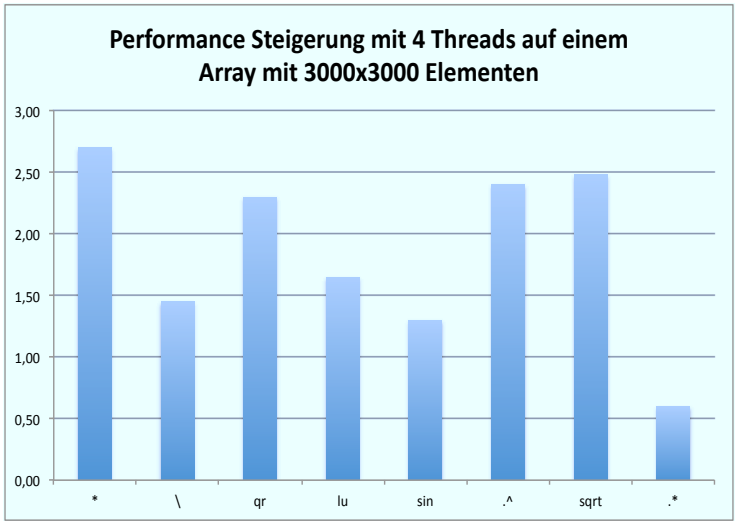

Abbildung 3: MATLAB SPMD-Performanzsteigerung mit vier Threads

Operationen für den Verwender meist wenig Gelegenheit zur Steuerung der Parallelität.

Domänenspezifische Umgebungen ermöglichen aufgrund des beschränkten Einsatzgebietes einen einfachen Zugang zu nebenläufiger Ausführung. Wie am Beispiel von MATLAB ersichtlich wird, kann die Parallelität sogar über mehrere Rechnercluster erfolgen. Idealerweise wird die Schnittstelle einer DSE nicht verändert, falls sie Parallelität unterstützt. Nicht jede Umgebung kann dies ohne jegliche Erweiterung der Syntax ermöglichen. Die Beschränkung des Sprachumfangs ist auch der entscheidende Nachteil für die Verallgemeinerung der Parallelisierungs-Techniken. Nach Boyle's Gesetz für domänenspezifische Sprachen wird die Ausdrucksstärke der jeweiligen Sprache umso größer, je kleiner der Umfang ist [9]. Damit ist auch der Effektivität von impliziter Parallelisierung bei DSEs eine Grenze gesetzt.

2.1.2 Parallele Muster

Um Parallelprogrammierung zu vereinfachen, reichen nicht allein neue Technologien aus. Es ist eine andere Denkweise im Bezug auf nebenläufiger Ausführung notwendig [13]. Entwickler haben viele Jahre Erfahrung sammeln können und erkannt, dass automatische Parallelisierung lediglich bei low-level Scheduling gut funktioniert. Für abstraktere Stufen ist häufig Wissen von erfahrenen Experten erforderlich, welche in anderer Art und Weise über parallelen Entwurf denken als Anfänger. Sie sehen Relationen zwischen verschiedenen Programmteilen und verstehen die Probleme, sowie Ansätze, diese zu lösen. In vielen Büchern kann man über parallele Algorithmen bzw. spezifische Spracherweiterungen wie *MPI* oder *OpenMP* lesen. Allerdings gibt es vergleichsweise wenig Informationen über sinnvolle Software-Strukturen nebenläufiger Programme oder Entwurfsmuster auf höheren Abstraktionsebenen.

Ein Entwurfsmuster beschreibt eine "gute" Lösung für ein häufig auftretendes Problem in einem speziellen Kontext [16]. Die Muster sind in einem bestimmten Format definiert, mit dem Namen des Entwurfsmusters, einer Beschreibung des Kontextes, der Ziele, den Bedingungen sowie der Lösungsmethode. Die Idee dahinter ist, Erfahrungen von Experten für alle zur Verfügung zu stellen, um ähnliche Probleme lösen zu können. Informatiker sind es gewohnt, in wohldefinierten Formalismen zu denken. Die Sprache der Entwurfsmuster fördert eine weniger formale, mehr assoziative Weise, über Probleme nachzudenken. Eine Muster-Sprache erzwingt keine festen Methoden; sie fördert kreative Problemlösung durch bereitstellen geprüfter Architekturent-

scheidungen [2].

Die Erkenntnis, dass Entwurfsmuster für die Software-Entwicklung nützlich sind, ist nicht neu. In der Objektorientierung werden sie gerne verwendet, um die Wiederverwendbarkeit von Klassen und Objekten zu unterstützen [7]. Die Arbeit an parallelen Entwurfsmustern wird ebenfalls seit einigen Jahren durchgeführt (Veröffentlichungen solcher Muster werden regelmäßig in *Pattern Language of Programs(PLoP)*-Konferenzen präsentiert). Manche Bücher behandeln Parallelprogrammierung auf unteren Abstraktionsebenen, wie das von Doug Lea [15] im Kontext von Java Threads. Das Buch von Mattson et. al [16] dagegen beschreibt Entwurfsmuster, die durch hohe Abstraktion Technologie-unabhängig sind.
Dass großes Interesse an einer "Pattern-Sprache" existiert, zeigen auch aktuelle Forschungsprojekte namhafter Universitäten. Das *UPCRC*-Team aus Berkeley erstellt momentan eine Liste an "dwarfs", welche als nebenläufige Berechnungsmuster verstanden werden können[5]. Sie werden für die Berechnung häufig auftretender mathematischer und algorithmischer Herausforderungen verwendet.

Musterbibliothek von Illinois Die Universität von Illinois hat eine große Geschichte im Bereich der Parallelprogrammierung. Am dortigen *Department of Computer Science* läuft das *UPCRC (Universal Parallel Computing Research Center)* Programm mit Beteiligung der Firmen Intel und Microsoft. Eine Kernaufgabe besteht in der Erstellung einer Bibliothek für parallele Entwurfsmuster. Ziel des Projekts ist die Dokumentation und Lehre dieser Muster,

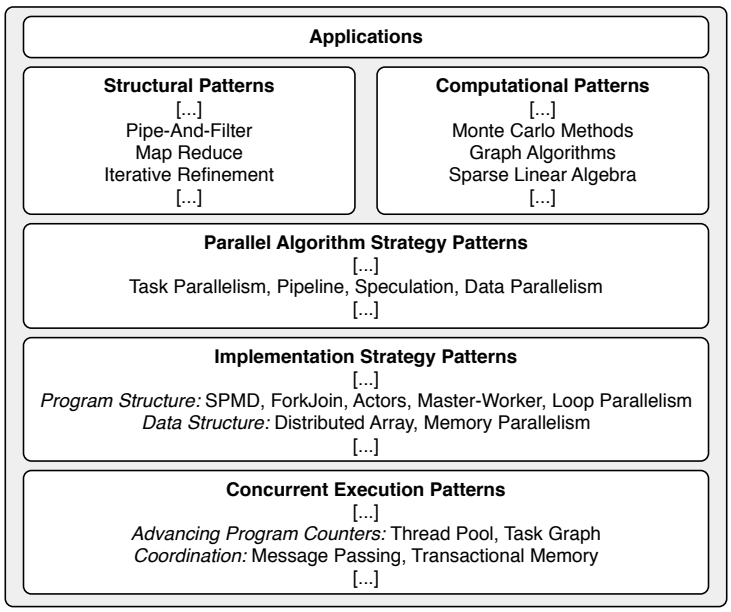

Abbildung 4: UPCRC/Illinois - Entwurfsmuster-Bibliothek[13]

um möglichst viele Entwickler zu Experten im Bereich paralleler Programmierung auszubilden. Außerdem soll die Weiterentwicklung existierender Programmiersprachen angeregt werden. Die implizite Unterstützung von parallelen Entwurfsmustern ermöglicht eine neue Stufe von Abstraktion.
Obwohl die Arbeit noch nicht abgeschlossen ist und einige wichtige Punkte fehlen, so gibt die Bibliothek (Abbildung 4) bereits einen guten Überblick über gängige parallele Entwurfsmuster. Diese wurden in unterschiedliche Kategorien eingeteilt, sodass eine schnelle Suche nach

[5]Dwarfs (`http://parlab.eecs.berkeley.edu/research/193`)

passenden Lösungen möglich ist. Es existieren folgende Kategorien:

- Strukturelle Muster (Structural Patterns)
  Behandeln den strukturellen Aufbau von nebenläufigen Anwendung
- Muster für parallele Berechnungen (Computational Patterns)
  Werden für die Berechnung spezieller mathematischer oder algorithmischer Probleme eingesetzt (vgl. "dwarfs")
- Strategie-Muster für parallele Algorithmen (Parallel Algorithm Strategy Patterns)
  Definiert generelle Vorgehensmuster für das parallele Design von Anwendungen
- Strategie-Muster für parallele Implementierungen (Implementation Strategy Patterns)
  Behandelt wichtige Entwurfsentscheidungen, welche die Implementierung beeinflussen
- Parallelausführungs-Muster (Concurrent Execution Patterns)
  Legt die Art und Weise paralleler Ausführung fest

Beispiel: Pipeline-Entwurfsmuster Im folgenden wird das Pipeline-Entwurfsmuster verwendet, um dem Leser ein anschauliches Beispiel zu geben. Es soll deutlich werden, dass Muster einen großen Beitrag zur Abstraktion in der Parallelisierung leisten.
Eine *Pipeline* stellt eine gute Möglichkeit dar, Anwendungen ohne den fehleranfälligen Einsatz von aufwändiger Synchronisation zu parallelisieren. Das Verfahren beruht auf der Verarbeitung einer Menge von Datensätzen, wobei die Berechnungen in mehreren Stationen geschieht. Als analoges Beispiel kann man sich ein Fließband einer Fabrik vorstellen, bei dem mehrere Bearbeitungsschritte an Werkstücken durchgeführt werden. Jede Bearbeitungseinheit kann nur ein Werkstück zur selben Zeit bearbeiten. Die Nebenläufigkeit liegt somit nicht bei der gleichzeitigen Verarbeitung durch eine Bearbeitungseinheit. Vielmehr wird durch die Vielzahl der Werkstücke eine parallele Auslastung der Bearbeitungsschritte ermöglicht.
Gute Beispiele lassen sich auch in vielen Ebenen der Computer-Hardware finden:

- Befehls-Pipeline moderner CPUs: Die Ebenen (Anweisungen holen, dekodieren, ausführen, usw.) werden in einer *Pipeline* verarbeitet. Während die CPU eine Anweisung dekodiert, wird die Vorgängeranweisung verarbeitet und der Nachfolger geholt.
- Signal Verarbeitung: Eine Menge an Sensorwerten von Echtzeitsensoren wird durch eine Sequenz von Filtern gesendet. Diese Sequenz stellt ebenfalls eine *Pipeline*-Architektur dar.
- Shell-Programme unter UNIX: Zum Beispiel das Kommando
  `cat sampleFile | grep 'word' | we`.

Die Beispiele sowie die Fließband-Analogie haben etwas gemeinsam. Sie wenden eine Folge von Operationen auf jedes Element der zu verarbeitenden Menge an. Obwohl häufig eine definierte Reihenfolge beachtet werden muss, ist es möglich, mehrere Operationen auf verschiedenen Daten anzuwenden. Die entscheidende Idee des Pipeline-Musters ist, dass unterschiedliche Verarbeitungsschritte auf beliebigen Hardware-Threads ausgeführt werden können. Abbildung 5 zeigt vier Pipeline-Ebenen, welche die einzelnen Verarbeitungsschritte darstellen. Die Reihenfolge, in der die Bearbeitungseinheiten `B1,...,B5` von den Ebenen abgearbeitet werden ist aufsteigend vorgegeben. Falls bei unterschiedlichen Operationen die Reihenfolge der Verarbeitung keine Rolle spielt, so kann der Entwurf auch von der rein linearen Ausführung abweichen. Solche Operationen können beispielsweise mit dem "Task-Parallelism"-Entwurfsmuster gleichzeitig auf mehrere Hardware-Threads verteilt werden, wodurch zusätzliche Effizienz erreicht wird.
Bei der Verwendung des Pipeline-Musters muss auf folgende Dinge geachtet werden:

- Das Ausmaß der Parallelität ist durch die Anzahl der Bearbeitungsschritte limitiert. Eine höhere Anzahl erlaubt mehr gleichzeitige Bearbeitungen. Allerdings müssen die zu bearbeitenden Daten stets zwischen den Operationen übergeben werden, was zusätzlichen Rechenaufwand bedeutet. Eine Abwägung zwischen wenigen und vielen Bearbeitungsebenen ist somit erforderlich.
- Das Entwurfsmuster ist effektiver, falls die einzelnen Operationen ähnliche Ausführungszeiten besitzen. Falls die Zeiten sehr variieren, so stellt die langsamste Bearbeitungseinheit einen Flaschenhals des Systems dar.

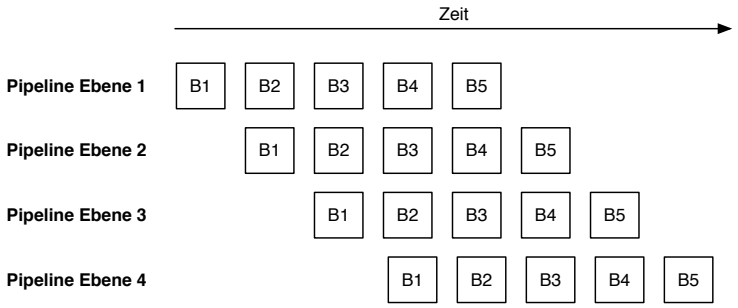

Abbildung 5: Pipeline Entwurfsmuster

Die einfachste Art, Hardware-Threads an das Pipeline-Entwurfsmuster zu verteilen, ist eine direkte Zuweisung zu einzelnen Operationen. Falls weniger solcher Threads als Bearbeitungsschritte vorhanden sind, so müssen sich mehrere von diesen eine Einheit teilen. Hierbei ist zu beachten, dass lokale Ressourcen in ausreichender Weise zur Verfügung stehen. Falls mehr Hardware-Threads als Operationen verfügbar sind, so kann über die Verwendung eines Entwurfsmuster der Algorithmen-Klasse (siehe Abbildung 4) entschieden werden. Damit können mehrere Threads die Ausführung einer Operation durchführen und ggf. Flaschenhälse beseitigt werden.

Das Pipeline Entwurfsmuster harmoniert mit dem Aktor-Modell, welches in Kapitel 2.2.1 vorgestellt wird. Hierbei stellen Aktoren die unterschiedlichen Bearbeitungsschritte dar, wodurch die Parallelisierung implizit ohne die fehleranfällige Erzeugung von Threads ermöglicht wird.

Fazit Parallele Entwurfsmuster spielen eine große Rolle, um Parallelprogrammierung einfacher zu gestalten als bisher. Das wird vor allem durch eine Änderung der Denkweisen erreicht, die während der Verwendung der Muster entsteht. Ähnlich zur objektorientierten Programmierung, bei der Entwurfsmuster für mehr Wiederverwendbarkeit existierender Lösungen sorgen [7], sollte eine neue Sprachebene für parallele Lösungsansätze entstehen. Die gewonnene Abstraktion hilft dabei, die Architektur nebenläufiger Anwendungen ohne aufwändige Synchronisation entwerfen zu können. Um eine hohe Effektivität zu erreichen, müssen Entwurfsmuster auf hoher sowie niedriger Abstraktionsebene eine Hilfe für die Entwickler darstellen. Es sollen Möglichkeiten aufgezeigt werden, Strategie-Entscheidungen sinnvoll zu treffen.

Bis eine möglichst einheitliche "Sprache" von Entwurfsmustern entstehen kann, ist es allerdings noch ein weiter Weg. Die ersten Forschergruppen sammeln und dokumentieren derzeit gängige Muster (siehe Abbildung 4). Doch auch die Lehre hat eine wichtige Rolle, um die Denkweise für Parallelprogrammierung durch Entwurfsmuster zu verändern[6].

2.1.3 Parallele Skelette

Ein zu Entwurfsmustern ähnliches Verfahren, um Nebenläufigkeit durch Abstraktion zu erreichen, ist die Verwendung von Skeletten. Diese stellen eine Menge von generischen Programmierschablonen dar, welche bekannte algorithmische Muster für Berechnungen und Kommunikation unterstützen. Jedes Skelett beinhaltet eine mitgelieferte, parallele Implementierung, die für den Benutzer transparent ist. Ein großer Vorteil bei der Verwendung ist, dass die die Nebenläufigkeit innerhalb der Skelette abläuft. Entwickler müssen somit nicht selbst für die Synchronisation zwischen den Komponenten sorgen. Dies ist weit weniger fehleranfällig als

[6] Multicore Programming Education - Speakers and Abstracts (`http://www.cs.tau.ac.il/~shanir/Abstracts%20and%20Speakers%20Multicore%20Programming%20Education.htm`)

traditionelle Verfahren, wie z.B. Threads oder *MPI*. Es existieren einige Bibliotheken, welche algorithmische Skelette anbieten. Dabei werden verschiedene Techniken verwendet, wie z.B. funktionale, imperative oder objektorientierte Programmierung.

Mit Hilfe von Skeletten können parallele Anwendungen erstellt werden, indem sie anwendungsspezifische Daten oder Code (meist in Form von polymorphen Funktionen höherer Ordnung) als Parameter übergeben. Die Implementierung der Skelette kann für die Zielplattform optimiert sein, sodass das Potential verschiedener paralleler Architekturen (z.B. verteilter- oder gemeinsamer Speicher, Multithreading, usw.) ausgenutzt werden kann. Für gewöhnlich werden Skelette in zwei Kategorien eingeteilt [17].

- *Datenparallel*: Datenparallele Skelette sollen die Parallelisierung von Berechnungen auf strukturierten Daten ermöglichen, indem Teile der Daten durch mehrere Prozesse simultan bearbeitet werden. Beispiele dafür sind *map*, *reduce*, *filter* oder *scan*.
- *Taskparallel*: Sogenannte taskparallele Skelette beschreiben verschiedene Muster, um nebenläufige Aufgaben zu organisieren. Das Skelett erzeugt dazu ein System von Kommunikationsprozessen, durch das Parallelität erzeugt wird. Taskparallele Skelette werden in der Regel in Kombination mit datenparallelen Skeletten verwendet. Beispiele sind *pipeline*, *farm*, *branch-and-bound* oder *divide-and-conquer*.

Beispiele in *Scala* *Scala* eignet sich, aufgrund der Kombination aus funktionalen sowie imperativen Sprachelementen, hervorragend für die Implementierung sowie Verwendung von Skeletten. Vieler solche Skelette sind bereits in den Sprachumfang der *Scala*-Bibliotheken integriert und können implizit verwendet werden. In diesem Abschnitt werden zwei Skelette zur Veranschaulichung implementiert. Für die Parallelisierung wird die Klasse `Future` verwendet, welche durch Aktoren Ausführung von Methoden auf mehrere Threads verteilt (siehe Kapitel 2.2.1).

***map*-Skelett** Die *map*-Funktion gehört zu den datenparallelen Skeletten und bietet für alle Funktionen auf Listenelementen, $f \in A \to B$, eine Funktion auf Listen, $map\, f \in [A] \to [B]$, welche die Funktion f für alle Elemente der Argumentliste $[a_1, ..., a_m]$ aufruft:

$$map\, f\, [a_1, \ldots, a_m] = [f\, a_1, \ldots f\, a_m] \tag{1}$$

Für die Realisierung des Skeletts wird zwecks Übersichtlichkeit eine pragmatischer Ansatz

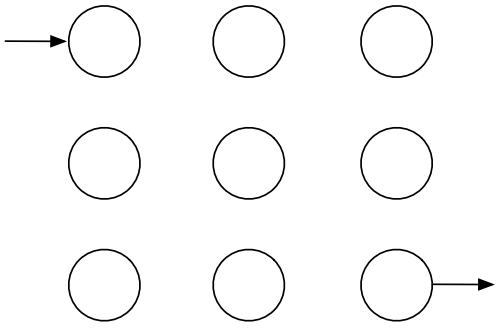

Abbildung 6: *map*-Skelett

verwendet. Jede Berechnung von Funktionswerten geschieht in einem eigenen Thread. Dies stellt offensichtlich nicht die effizienteste Lösung dar, für die Veranschaulichung ist es jedoch ausreichend. Listing 3 zeigt eine mögliche Implementierung des *map*-Skeletts in *Scala*. Die Methode `mapP` erwartet als Parameter eine Funktion, welche einen Wert vom generischen Typ `A` in einen Wert vom generischen Typ `B` überführt. Der Typ `A` ist dabei von der Klasse vorgegeben.

```
val aMinute = 1000 * 60

def mapP[B](f: (A) => B): List[Any] = {
  val futures: List[Future[Any]] = myList map {
    x => future( f(x) )
  }
  awaitAll(aMinute, futures: _*)
}
```

Listing 3: *map*-Skelett in *Scala*

`myList` ist das Listenattribut der Klasse vom Typ `List[A]`. Die Methode `mapP` liefert eine Liste vom Typ `B` zurück, welche die berechneten Funktionswerte darstellen.
Für die Berechnungen werden alle Einträge der Liste `myList` (über die sequentielle Methode `map`) an ein `future`-Objekt übergeben. Dieses führt die übergebene Methode `f` in einem Aktor (mit je einem Thread) aus. Die Methode `awaitAll` (vom Objekt `Future`) wartet bis alle Aktoren das Ergebnis berechnet und zurückgeliefert haben. Dafür blockiert die Methode entweder solange bis die Berechnungen fertig sind, oder bis die übergebene Zeit (Konstante `aMinute`) abgelaufen ist.

***farm*-Skelett** Das *farm*-Skelett besteht aus Farmer und Worker Prozessen. Der Farmer-Prozess teilt eine Datenmenge auf und gibt sie an die Worker-Prozesse weiter. Es gibt verschiedene Verfahren, um Worker zu realisieren. In dem hier vorgestellten Beispiel kann jeder Worker unterschiedliche Funktionen ausführen.

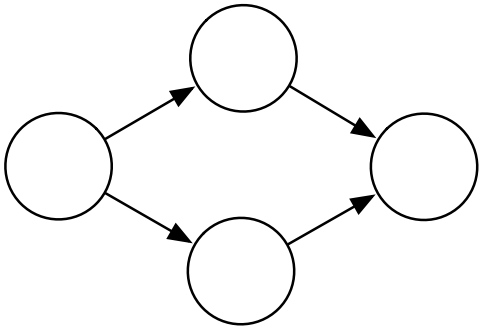

Abbildung 7: *farm*-Skelett

```
val aMinute = 1000 * 60
type func[A, B] = A => B

def forkAll[A, B](functions: func[A, B]*)(value: A): List[Any] = {
  val results = functions.toList map {
    function: func[A, B] => future(function(value))
  }
  awaitAll(aMinute, results: _*)
}
```

Listing 4: *farm*-Skelett in *Scala*

Die beispielhafte Implementierung des *farm*-Skeletts aus Listing 4 kann eine beliebige Datenmenge an verschiedene Funktionen übergeben. Dazu wird die Methode `forkAll` verwendet, welche eine beliebige Anzahl an Funktionen als Parameter entgegennimmt, die Werte vom generischen Typ `A` in Werte vom generischen Typ `B` überführt. `forkAll` verwendet *Currying*, um die Datenmenge (`A`) zu übergeben. Im Gegensatz zu Listing 3 wird hier über sämtliche Funktionen iteriert. In jedem Schritt erhält ein Aktor (über `future`) den Auftrag zur Ausführung einer Funktion mit den übergebenen Daten. Auch hier wartet die Methode *awaitAll*, bis sämtliche Berechnungen fertig sind, oder bis die vorgegebene Zeit abgelaufen ist.

Fazit Die Verwendung von Skeletten bringt ähnlich zu Entwurfsmustern eine erhöhte Wiederverwendbarkeit von paralleler Software mit sich. Es ist eine effektive Art, Synchronisations- und Kommunikationsdetails vor dem Entwickler zu verbergen. Im Gegensatz zu den Entwurfsmustern, bieten Skelette allerdings eine formale Möglichkeit, um algorithmische Probleme mit Hilfe mehrerer Prozesse zu lösen. Ziel ist es, dass Anwendungsprogramme als sequentielle Zusammenstellung von parallelen Ebenen repräsentiert werden können [8]. Der Anwendungsentwickler erstellt dafür lediglich die sequentiellen Algorithmen, meist in Form von Funktionen höherer Ordnung. Diese werden an Skelette übergeben, in welchen die Parallelität versteckt ist. Es muss lediglich spezifiziert werden, *was* nebenläufig berechnet werden soll, nicht aber *wie* das geschehen soll. Die Art und Weise der Nebenläufigkeit wird in den Implementierungen mitgeliefert. Die Sprache der Implementierungen muss nicht mit der Programmiersprache der Anwendung übereinstimmen. Somit wird noch mehr ermöglicht, dass die Implementation von anderen Personen durchgeführt werden kann, als die spätere Anwendung der Skelette. Spezialisten bezüglich paralleler Programmierung und domänenspezifischem Wissen können dadurch mächtige Funktionen für Anwendungsentwickler erstellen.
Es existieren mehrere Bibliotheken von Skeletten, die für eine Realisierung von effizienten Programmen auf unterschiedlichen Zielplattformen verwendet werden können. Jedoch muss dabei auf die geeignete Komposition von Skeletten geachtet werden. Höchste Performanz wird nur erreicht, falls eine Zusammenstellung verwendet wird, die für die entsprechende Anwendung und Zielplattform geeignet ist.

2.2 Programmiermodelle

Um die Vorteile von Parallelprogrammierung komfortabel nutzen zu können, sind auch die verwendeten Programmiermodelle von entscheidender Bedeutung. Solche Modelle müssen dem Programmierer erlauben, die Balance zwischen Produktivität und Effizienz der Implementierung zu erreichen. Noch ist nicht klar, wie und auf welcher Ebene solche Modelle benötigt werden. Sicher ist jedoch, dass der Entwickler Möglichkeiten zur Erzeugung deterministischer Parallelprogramme benötigt. Die in den folgenden Abschnitten beschriebenen Verfahren erreichen dies über unterschiedliche Ansätze auf hoher Abstraktionsstufe bzw. durch feingranulare Metaprogrammierung.

2.2.1 Aktoren

Das Aktor-Modell ist ein ereignisbasiertes Programmiermodell, das bereits vor über 25 Jahren entwickelt wurde. Die Konzeption war "motiviert durch die Erwartung von hochparallelen Rechenmaschinen, welche aus Dutzenden, Hunderten oder Tausenden von unabhängigen Mikroprozessoren bestehen, jeder mit eigenem lokalen Speicher und Kommunikationsprozessor, kommunizierend über ein hochperformantes Kommunikationsnetzwerk" [4]. Trotz einer hervorragenden Eignung für die nebenläufige Programmierung, fand es fast ausschließlich im akademischen Bereich Verwendung. Durch die steigende Verfügbarkeit von Mehrkernprozessoren in der letzten Zeit ist das Aktor-Konzept jedoch eine interessante Alternative auf der Suche nach effektiven und effizienten Programmiermodellen für parallele Architekturen [20]. Neben vielen Framework-Lösungen, welche das Aktor-Modell in gängige Programmiersprachen integrieren, gibt es immer mehr Sprachen, bei denen Aktoren zum Sprachumfang gehören. Bekannte Beispiele sind hierbei *Scala* oder *Erlang*.

Definition Aktoren sind reaktive Entitäten mit eindeutigem Namen, welche ausschließlich per Nachrichten an ihre Eingangs-Mailbox angesprochen werden können [1]. Sie können:

1. Eine endliche Menge an Nachrichten an andere bekannte Aktoren senden.
2. Eine endliche Anzahl an neuen Aktoren erstellen.
3. Definieren, wie auf eingehende reagiert werden soll.

Die Kommunikation zwischen Aktoren findet i.A. asynchron statt. Somit müssen die Aktoren zum Sendezeitpunkt auch nicht empfangsbereit sein; die Nachrichten werden in der Mailbox gespeichert, bis sie nacheinander verarbeitet werden. Die Reihenfolge, in der ein Aktor-Objekt Nachrichten erhält und verarbeitet, spielt keine Rolle[7]. Ein Aktor kann Nachrichten intern behandeln, oder aber auch an einen anderen bzw. neu erstellten Aktor zur Bearbeitung weitergeben.
Aktoren teilen keinen gemeinsamen Zustand; Informationen werden rein per Nachrichtenkommunikation ausgetauscht. Das bedeutet, dass eine fehleranfällige Synchronisation, wie z.B. durch Sperren für Zustandsvariablen entfällt. Um die Kommunikation performant durchzuführen, werden optimierte und effiziente Methoden benötigt.

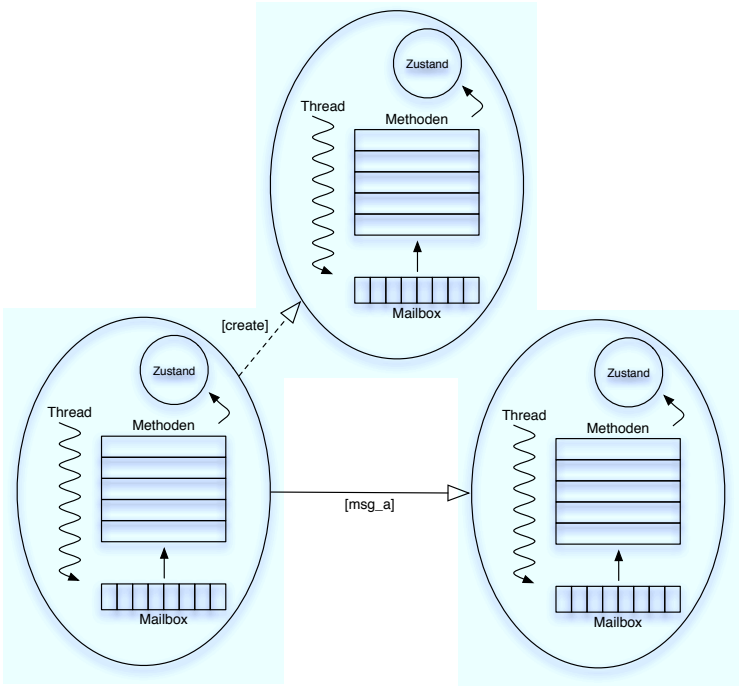

Abbildung 8: Komponenten des Aktor-Modells

Beispiel Um dem Leser einen besseren Einblick in den praktischen Umgang von Parallelität durch Aktoren zu geben, wird ein Beispiel anhand der Programmiersprache *Scala* erläutert. Es werden keine Vorkenntnisse erwartet. Durch die hohe Ausdrucksstärke von *Scala* werden die Konzepte des Aktor-Modells leicht ersichtlich.
Ein Aktor wird in *Scala* als ereignisbasierter Thread realisiert. Die einfachste Form, um einen Aktor zu erzeugen, ist die Methode `actor()` des `Actor`-Objekts. Beim Aufruf wird der Aktor implizit gestartet. Es gibt auch eine explizite Variante, diese ist allerdings nicht Teil dieses Beispiels.

[7] Manche Implementierungen von Aktoren, wie z.B. in *Scala*, verwenden geordneten Nachrichteneingang.

Falls man eine Nachricht an den Aktor senden möchte, kann man die Methode `!(msg: Any)`[8] verwenden. Der Empfang einer Nachricht kann durch die Methode `receive()` erfolgen. Diese erwartet ebenfalls eine Funktion als Parameter, in welcher die einkommenden Nachrichten verarbeitet werden können. Der Einsatz von *Pattern-Matching* fördert hier die Lesbarkeit von Aktor-Code, indem verschiedene Nachrichten bzw. Nachrichtentypen unterschiedlich behandelt werden können.

```
val fussyActor = actor {
  loop {
    receive {
      case "quit" => println("ByeBye"); exit
      case s: String => println("I got a String: " + s)
      case i: Int => println("I got an Int: " + i.toString)
      case _ => println("I have no idea what I just got")
    }
  }
}
fussyActor ! "hi there"
fussyActor ! 23
fussyActor ! 3.33
fussyActor ! "quit"
```

Listing 5: Aktoren in *Scala*

Im Beispielcode von Listing 3 wird zunächst über `val` ein unveränderbarer Aktor mit dem Namen `fussyActor` erzeugt. Dieser wird unverzüglich nach dem Aufruf der `actor()`-Methode gestartet und wartet kontinuierlich auf neue Nachrichten. Die `receive()`-Methode wird aufgerufen, sobald Nachrichten in der Mailbox vorhanden sind. Sollten keine Nachrichten vorhanden sein, so blockiert `receive` bis zum Eingang einer Nachricht. Um einen Aktor zu stoppen, muss dies explizit über eine Nachricht erfolgen (im Beispiel: ``quit''). Pattern Matching sorgt für die Verarbeitung der unterschiedlichen Nachrichtentypen, wobei "_" als Regelfall verwendet wird. Die Anwendung liefert folgende Ausgabe:

```
I got a String: hi there
I got an Int: 23
I have no idea what I just got
ByeBye
```

Wie man sieht wurden durch die `!`-Methode Nachrichten an den Aktor gesendet, welche je nach Typ unterschiedlich verarbeitet wurden. Die letzte Nachricht hat den Aktor über das Schlüsselwort `exit` beendet.

Das Beispiel zeigt nur einen kleinen Ausschnitt der Leistungsfähigkeit von Aktoren unter *Scala*. Es wird eine Vielzahl an Möglichkeiten bereitgestellt, um Nebenläufigkeit ohne schwierige Synchronisation durchzuführen. Die hohe Abstraktionsstufe hilft dabei, sich auf die wesentlichen Dinge zu beschränken, die Sprache kümmert sich um Details wie das Erstellen von Threads oder Zustellgarantien von Nachrichten. Das Resultat ist präziser Code und geringe Fehleranfälligkeit [18].

Fazit Das Aktor-Modell ist ein flexibles Verfahren für die Erstellung paralleler Systeme. Im Gegensatz zur Parallelisierung durch Threads, wobei die Kontrolle der Ausführung extern gesteuert wird, sind autonome Aktoren ein natürlicher Ansatz. Die asynchrone Verarbeitung erlaubt die Modellierung von nicht-deterministischen reaktiven Systemen, sowie flexible Platzierung und Scheduling von Aufgaben auf verschiedenen Kernen [3]. Es ist somit nicht

[8] In *Scala* können auch einzelne Zeichen wie `!` als Namen für Methoden verwendet werden.

verwunderlich, dass gerade in den Bereichen von Peer-To-Peer-Systemen, Webservices und Multicore Prozessorarchitekturen ein großes Interesse an Programmiersprachen besteht, welche Aktoren unterstützen.
Dennoch gibt es noch einige Herausforderungen, die momentan von Forschergruppen bearbeitet werden [13]. Der wichtigste Aspekt ist hierbei die Nachrichtenübertragung zwischen Aktoren. Falls keine Referenzen auf Nachrichteninhalten verwendet werden, müssen Kopien der entsprechenden Speicherbereiche erstellt werden. Dies hat erhebliche Auswirkungen auf die Performanz des Systems. Zum Anderen bewirkt ein exzessiver Austausch von Nachrichten einen erhöhten Energiebedarf.

2.2.2 Autotuner vs. traditionelle Compiler

Damit Anwendungen den theoretischen Gewinn an Performanz durch die Mehrzahl von Prozessorkernen optimal nutzen können, ist die effiziente Verwendung von Ressourcen besonders wichtig. Der Compiler spielt hier eine wichtige Rolle. Er entscheidet welche Optimierungen verwendet werden und wählt entsprechende Parameter dafür. Der resultierende Spielraum an Alternativen ist allerdings sehr groß. Moderne Compiler sind bereits heute aus vielen Millionen Codezeilen aufgebaut. Neue Optimierungen erfordern daher häufig fundamentale Änderungen an den internen Strukturen.
Da Compilern wenig semantische Information übergeben wird, müssen (für höchste Performanz) viele Optimierungen durch manuelle Eingriffe gemacht werden. Gerade bei Parallelprogrammierung enthalten die meisten skalierbaren Anwendungen Synchronisationsmethoden, um Abläufe zu steuern. Solche hardwarenahen Techniken sind arbeitsaufwändig, fehleranfällig und meist nicht auf andere Plattformen portierbar. Heterogene Architekturen, in denen mehrere verschiedenartige Prozessoren verwendet werden, erschweren die Optimierungen zusätzlich.

Autotuner können diese Probleme lösen, indem sie automatisch den Optimierungs-Raum einer Applikation auf einer speziellen Hardware-Architektur untersuchen. Dadurch kann der Autotuner die beste Kombination an Algorithmen, Implementierungen und Datenstrukturen für die Eingabedaten bestimmen [21]. Durch dieses Vorgehen kann eine *Performanz-Portabilität* erreicht werden, die Fähigkeit ein Programm ein einziges Mal zu erstellen und dennoch gute Performanz auf allen aktuellen sowie zukünftigen Multicore-Computern zu erreichen. Abbildung 9 vergleicht den traditionellen Ansatz über Compiler mit dem Autotuning Verfahren.
Die linke Darstellung zeigt ein Alberto Sangiovanni-Vincentelli (ASV) Dreieck [2]. Ein Entwickler beginnt mit einer abstrakten Operation, die er implementieren möchte. Dafür existieren viele mögliche Implementierungen, welche alle die selbe Funktionalität liefern. Dennoch entscheidet er sich für eine spezielle Repräsentation seiner Semantik. Durch diesen Schritt werden semantische Informationen vom Compiler fern gehalten. Dieser verwendet die gewählte Repräsentation und untersucht alle sicheren Transformationen anhand der geringen Informationen. Das Resultat ist ein einzelnes Binary.
Die rechte Abbildung zeigt den Autotuning Ansatz. Der Entwickler implementiert hier einen Autotuner, der anstatt einer einzigen Quellcode-Repräsentation einige hundert oder tausend von diesen erzeugt. Die Hoffnung besteht darin, dass durch diese Vielzahl die abstrakten Ideen bei der gemeinsamen Untersuchung erhalten bleiben. Der Compiler optimiert anschließend die Repräsentationen individuell und erstellt die Binaries. Der Autotuner vergleicht zuletzt die erstellten Maschinen-Code-Repräsentationen im Kontext des aktuellen Datensatzes bzw. Systems und wählt die beste Alternative.

Bei Autotunern werden drei entscheidende Schritte durchgeführt: die Suche nach möglichen Implementierungen, Code-Generierung und Vergleich. Zunächst werden mehrere mögliche Implementierungen spezifiziert. Anschließend erstellt der Code-Generator Quellcode-Repräsentationen für die zu optimierenden Kernel. Zuletzt iteriert der Autotuner mit Benchmarks über die Repräsentationen. Das Resultat ist das Binary mit der besten Performanz aller Varianten.
Es gibt mehrere Versionen von Autotunern. Die einfachste Version untersucht lediglich Optimierungen auf niedriger Abstraktionsebene, wie z.B. Umordnungen, Restrukturierung von Schleifen,

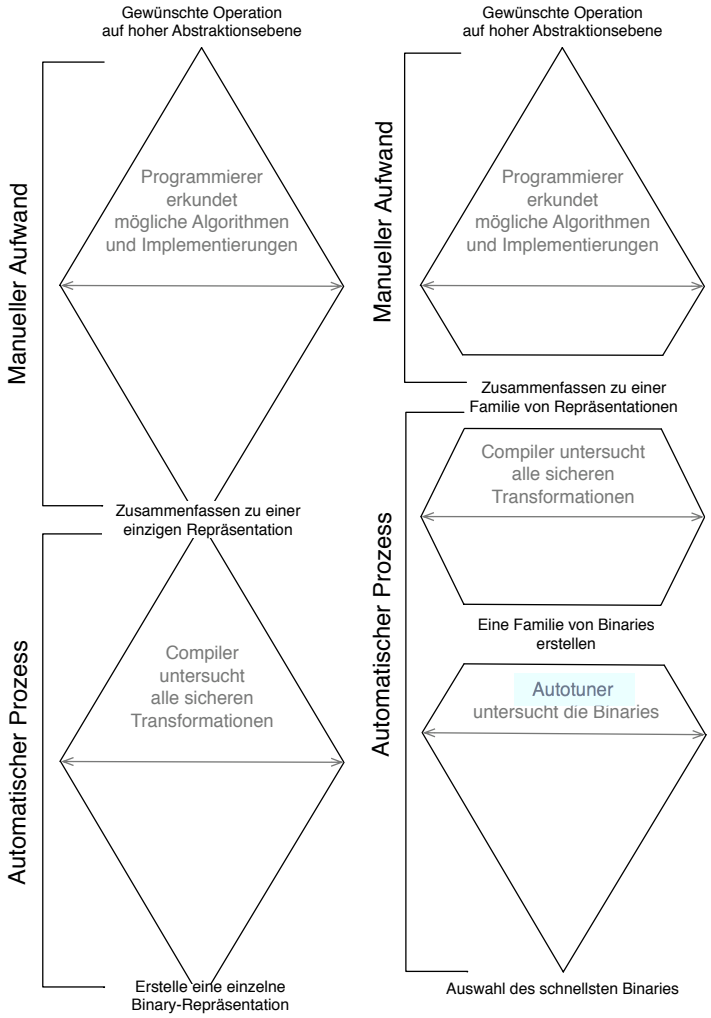

Abbildung 9: ASV-Dreiecke für den konventionellen Ansatz sowie Autotuner

Entfernung von Sprüngen, Übersetzung expliziter SIMD[9]-Anweisungen oder Cache-Umgehungs-Mechanismen. Solche Verfahren können theoretisch auch anspruchsvolle Compiler erledigen, doch häufig sind sie aufgrund fehlender Informationen dazu nicht in der Lage. Autotuner mit erweiterten Möglichkeiten können auch unterschiedliche Datentypen, Datenlayouts oder Datenstrukturen untersuchen. Traditionelle Compiler können dies nicht. Die anspruchsvollsten Autotuner untersuchen verschiedene Algorithmen, welche identische Resultate für das abstrakte Problem darstellen. Diese Autotuner wecken derzeit großes Interesse im Forschungsumfeld [21]. Als nächstes folgt die Code-Generierung. Die einfachste Strategie stellt ein automatisches Skript dar, das eine Menge von möglichen Binaries zu den Quellcode-Repräsentationen erstellt. Ein intelligentes Verfahren verwendet Hintergrundinformationen über die parallele Architektur und generiert alle gültigen Optimierungen über einen Satz von entsprechenden Transformationen [14]. Dies widerspricht dem Verfahren von Compilern in zweifacher Hinsicht: Zum Einen kann ein Compiler keine Optimierungen verwenden, bei denen die Sicherheit nicht für alle Fälle prüfbar ist. Zweitens erstellt diese Version viele Binaries, wohingegen ein Compiler lediglich

[9]Single-Instruction-Multiple-Data ist ähnlich zu SPMD

ein einziges erzeugt.
Der dritte Schritt von Autotunern ist die Untersuchung der Binaries. Die am häufigsten verwendete Technik ist eine aufwändige Suche aller Parameter für sämtliche Optimierungen. Für jede Optimierung wird der Reihe nach ein geeigneter Parameter ausgewählt und die Laufzeit des Binaries gemessen. Falls die Performanz besser als die des Vorgängers ist, wird das Binary als beste Lösung gesetzt. Offensichtlich kann diese Lösung bei einigen tausend Kombinationen äußerst aufwändig sein. Deshalb existieren viele Heuristiken um passende Parameter auszuwählen [21]. Diese Techniken sind jedoch nicht Bestandteil dieser Arbeit.

Fazit Durch Autotuner allein werden Anwendungen der Entwickler nicht zu perfekt skalierenden Systemen. Falls jedoch Architektur-Richtlinien und Entwurfsmuster (siehe Kapitel 2.1.2) auf hoher Abstraktionsebene eingehalten werden, so bietet Autotuning viele Vorteile. Durch die Verwendung von statischen sowie semantischen Informationen wird die optimale Effizienz im Bezug auf Laufzeit und Stromverbrauch erreicht. Zudem haben Autotuner einen entscheidenden Vorteil, sie ermöglichen Portabilität bezüglich unterschiedlicher Hardware-Architekturen, welche sogar heterogen aufgebaut sein können. Das alles ist eine enorme Komplexitätsreduzierung für Entwickler von nebenläufigen Anwendungen. Daher wird gerade im Bereich der Multicore-Programmierung erwartet, dass Autotuner zum Mittel der Wahl für die Erzeugung von Anwendungen werden [13].
Es ist jedoch zu beachten, dass die Verwendung von Autotunern erheblich aufwändiger ist als die von traditionellen Compilern. Selbst beim Einsatz von intelligenten Vergleichsheuristiken erfordert ein Durchlauf auf einer speziellen Hardware-Architektur viel Rechenaufwand. In dem Forschungsbereich für Autotuner gibt es noch viele offene Punkte die geklärt werden müssen, bevor der Einsatz für die Allgemeinheit ermöglicht werden kann.

2.3 Ausführungsumgebungen

Neben geeigneten Programmiermodellen und Sprachen, sollten auch die Plattformen für parallele Systeme Abstraktion ermöglichen. Das Ziel ist neben hoher Performanz vor allem auch die einfache Programmierbarkeit. Im Folgenden wird ein Blick in vielversprechende Technologien gegeben.

2.3.1 Transaktionsspeicher

Nach wie vor müssen beim Umgang mit Nebenläufigkeit meist aufwändige Synchronisations-Primitiven verwendet werden. Es ist allgemein bekannt, dass diese Werkzeuge viele Probleme mit sich bringen. Sperren sind gerade für unerfahrene Programmierer eine ständige Fehlerquelle. Unerwartete Prioritätsumkehr passiert beispielsweise, wenn ein Thread mit niedriger Priorität eine Sperre auf eine Ressource hält und somit einen Thread mit hoher Priorität nicht starten lässt. Convoying beschreibt einen Zustand, in dem Prozesse nicht fortfahren können weil ein Scheduler einen anderen Prozess verzögert, der eine benötigte Sperre hält. Deadlocks können entstehen, falls mehrere Threads eine Sperre auf Objekte haben und auf die jeweilige Freigabe warten. Bei Livelocks verharren mehrere Prozesse in einem Zustand. Starvation kann entstehen, falls ein Prozess zu keinem Zeitpunkt den Zugriff auf gemeinsame Ressourcen erhält (z.B. aufgrund der Priorisierung im Scheduler).
Das wahre Problem liegt in der Tatsache, dass niemand wirklich weiß, wie große Systeme in Bezug auf Sperren organisiert werden müssen [12].

Transaktion Eine Transaktion ist eine endliche Sequenz von Instruktionen, die von einem einzelnen Thread ausgeführt werden. Transaktionen sind atomar; jede Transaktion wird entweder erfolgreich ausgeführt oder wird abgebrochen. Transaktionen sind linearisierbar; sie werden schrittweise ausgeführt. Folgende Eigenschaften werden von Transaktion erwartet:

- **Atomar**
  Transaktionen werden vollständig oder gar nicht ausgeführt.

- **Konsistent**
  Eine Transaktion führt ein System von einem konsistenten Zustand in einen anderen konsistenten Zustand.
- **Isoliert**
  Während der Bearbeitung einer Transaktion haben andere Transaktionen keinen Einfluss.

Transaktionsspeicher bieten ein Programmiermodell, bei dem jeder Thread eine Transaktion starten kann. Die Transaktionen führen eine Sequenz von Operationen auf gemeinsamen Speicher aus, ohne die Änderungen sofort abzuspeichern. Nach Abschluss der Transaktion wird versucht, eine *Commit*-Anweisung auszuführen, um die geänderten Daten zu speichern. Sollte diese Anweisung erfolgreich sein, so werden die Modifikationen der Transaktion wirksam. Andernfalls werden sie verworfen.

Hardware-Transaktionsspeicher Die meisten Ansätze für Hardware-Transaktionsspeicher basieren auf Modifikationen üblicher Cache-Kohärenz-Protokolle von Multiprozessor-Systemen [10]. Falls ein Thread über eine Transaktion auf einen Speicherbereich zugreifen möchte, wird der zugehörige Cache-Eintrag als transaktional gekennzeichnet. Solange die Transaktion aktiv ist, werden Schreibzugriffe im Cache gesammelt, aber nicht in den gemeinsamen Speicher geschrieben. Falls während der Ausführung ein Daten-Konflikt mit einem anderen Thread entsteht, so wird die Transaktion abgebrochen und erneut gestartet. Falls die Transaktion ohne Konflikt ausgeführt werden kann, so werden die Cache-Einträge als gültig gekennzeichnet und in den gemeinsamen Speicher geschrieben.
Hardware-Transaktionsspeicher besitzen eine Einschränkung. Es können nicht beliebig viele Transaktionen ausgeführt werden, da die Cache-Größe der Systeme begrenzt ist. Somit können auch nur eine bestimmte Anzahl an Operationen in transaktionalen Cache-Einträgen gehalten werden. Unterschiedliche Plattformen mit verschiedenen Cache-Größen machen es für Entwickler schwierig, generische Transaktionen für Hardware-Transaktionsspeicher zu erstellen.

Software-Transaktionsspeicher Eine Alternative zu Hardware-Transaktionsspeichern ist die Realisierung in Software. Dabei ergeben sich allerdings viele Fragen bezüglich der Semantik und der Implementierung.
Zunächst muss geklärt werden, wie Transaktionen die Ausführung von nicht-transaktionalen Operationen behandeln. Eine Möglichkeit ist hierbei die *starke Isolation* (wie bei Hardware-Transaktionsspeicher). Diese garantiert, dass Transaktionen atomar bezüglich nicht-transaktionalem Zugriff sind. Leider ist dieses Vorgehen für Software-Transaktionsspeicher zu langsam [12]. Die Alternative dazu ist die *schwache Isolation*, welche diese Garantie nicht gibt.
Es existieren zwei grundsätzliche Verfahren für die Organisation von transaktionalen Daten. In einem System, das *eager-update* verwendet, werden Daten direkt im gemeinsamen Speicherbereich geändert. Die Transaktionen führen zusätzlich ein Änderungs-Protokoll, mit dessen Hilfe Änderungen bei Abbruch zurückgesetzt werden können. Bei dem zweiten Verfahren (*lazy-update*) werden die Daten optimistisch im lokalen Speicher der Transaktion berechnet und geändert. Bei erfolgreicher Ausführung werden die Daten in den gemeinsamen Speicher übertragen, ansonsten werden sie verworfen. Ein System mit *eager-update* ist effizienter als ein System mit *lazy-udpate*, jedoch ist damit der Zugriff auf konsistente Zustände schwieriger. Außerdem wird meist ein Konflikt-Manager benötigt, um Lösungsanweisungen bei Konflikten zu erhalten.

Fazit Transaktionsspeicher haben in den letzten Jahren viel Aufmerksamkeit seitens der Forschung erhalten. Sie sollen die Schwierigkeiten bei der Erstellung paralleler Anwendung lösen und gleichzeitig skalierbar, effizient und korrekt sein. Der Transaktionsspeicher stellt sicher, dass mehrfacher Zugriff auf gemeinsamen Speicher atomar abläuft. Entwickler müssen sich somit keine Gedanken mehr über gleichzeitige Zugriffe von anderen Threads machen. Der Benutzer spezifiziert, was atomar ausgeführt werden soll, ohne dem System sagen zu müssen, wie es dies erreicht. Damit werden Programmierer von Problemen wie Sperrbedingungen und Deadlocks befreit.

Es wurden bereits viele verschiedene Techniken zum Entwurf von Transaktionsspeichern veröffentlicht. Hardware-Transaktionsspeicher von Herlihy und Moss [10], Software-Transaktionsspeicher [11] und Ansätze, die beides kombinieren [5].

Viele Implementierungen von Hardware-Transaktionsspeicher sind aufgrund der begrenzten Cache-Größe nicht produktiv einsetzbar [6]. Ein großer Fortschritt wurde in den letzten Jahren im Entwurf von Software-Transaktionsspeichern gemacht. Allerdings erfordern solche Implementierungen erheblichen Rechenaufwand, weshalb eine gute Hardwareunterstützung erhofft wird. Neue Hybrid-Ansätze von Software- und Hardware-Transaktionsspeicher versprechen den jeweiligen Nachteilen entgegenzuwirken. Bis allerdings Erfolge in dem Bereich zu beachten sind, dürfte noch etwas Zeit bleiben [6].

2.3.2 Bulk Multicore-Architektur

Damit eine Akzeptanz von Parallelprogrammierung als Standard möglich wird, muss neben geeigneten Programmiermodellen und Entwicklungsumgebungen auch eine neue Hardware-Architektur verwendet werden [19]. Bisher wurden Prozessorarchitekturen vor allem im Bezug auf Performanz oder Energieeffizienz entworfen. Es wurden dadurch viele Fortschritte in den letzten Jahren erreicht, sodass aktuelle Recheneinheiten äußerst leistungsfähig sind. Durch den Wandel zu Multicore-Prozessoren hat sich die theoretische Performanz weiter erhöht, jedoch muss eine bessere Programmierbarkeit erreicht werden, um die Leistung nutzen zu können. Es werden dabei folgende Anforderungen an neue Architekturen gestellt:

1. Die Hardware muss in der Lage sein, hohe Effizienz zu erreichen, ohne die Entwickler vor Aufgaben auf niedrigen Abstraktionsstufen zu stellen.
2. Die Hardware muss helfen, die Wahrscheinlichkeit von (parallelen) Programmierfehlern zu minimieren.

Im Folgenden wird die *Bulk Multicore* Architektur vorgestellt, welche an der Universität Illinois [19] entwickelt wurde. Beim Entwurf wurde vor allem auf gute Programmierbarkeit sowie Skalierbarkeit geachtet. Außerdem sollte hohe Performanz bei möglichst einfacher Hardware erreicht werden. Der Schlüssel von *Bulk Multicore* besteht aus zwei Teilen.
Zum Einen wird Software als eine Serie atomarer Blöcke, bestehend aus vielen dynamischen Instruktionen (*Chunks*), ausgeführt. Die *Chunks* sind nicht sichtbar für den Entwickler oder die Laufzeitumgebung, daher müssen von den Programmierern keine besonderen Restriktionen eingehalten werden. *Chunks* werden in einzelnen Threads ausgeführt.
Zum Anderen führt *Bulk Multicore*-Hardware die Verwendung von Hardware-Adress-Signaturen ein, welche den *Chunks* zugewiesen werden. Eine Signatur ist ein Register mit 1024 Bit, welches Hardware-Adressen akkumuliert. Jeder *Chunk* enthält je zwei Signaturen. Eine W-Signatur, in welcher die Adressen der geschriebenen Daten akkumuliert werden und eine entsprechende R-Signatur für die gelesenen Daten. Zusätzlich gibt es Hardware-Operatoren, die beispielsweise die Schnittmenge von Signaturen unterschiedlicher *Chunks* berechnen können.

Durch die Hardware-Adress-Signaturen wird eine atomare und isolierte Ausführung von *Chunks* sichergestellt. Während der Bearbeitung werden keine veränderten Werte an die anderen, nebenläufigen *Chunks*, gesendet. Wenn ein *Chunk* mit seiner Ausführung fertig ist (Commit), so sendet er seine W-Signatur an andere Threads und löscht die lokalen Signaturen. Beim Empfang der fremden Signatur W_{inc} durch einen aktiven *Chunk*, wird die Schnittmenge mit den lokalen Signaturen W_{loc} und R_{loc} gebildet. Falls eine der Schnittmengen nicht leer sein sollte, so bedeutet dies, dass der *Chunk* auf bereits geänderte Daten zugreift. In dem Fall wird der *Chunk* verworfen und mit aktuellen Werten neu gestartet.

In Abbildung 10 bearbeitet Thread 0 die Variablen X, B, C und Y in einem *Chunk*. Auf B und C wird schreibend zugegriffen, sodass die entsprechenden Adressen in der Signatur W_0 enthalten sind. Der Thread 1 beginnt wenige Zeit später mit der Bearbeitung seines *Chunks*, jedoch noch bevor Thread 1 fertig ist. In der Signatur R_1 von Thread 1 befinden sich die Adressen der Variablen B und C, da auf diese lesend zugegriffen wurde. Nach dem Abschluss

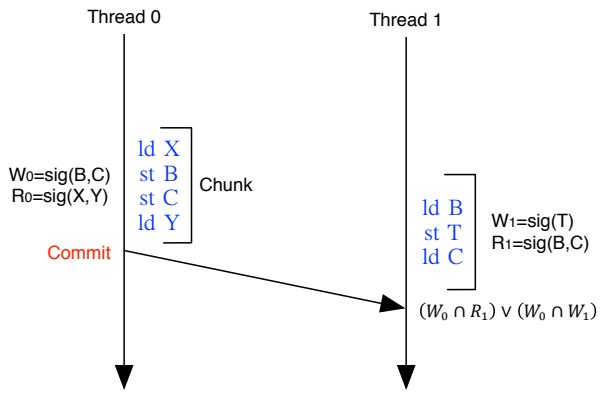

Abbildung 10: *Chunk*-Verarbeitung bei *Bulk-Multicore*-Architekturen

des *Chunks* durch Thread 1 erfolgt ein *Commit* und die Schreib-Signatur wird an Thread 1 gesendet. Dort wird die erhaltene Signaturen mit den lokalen Signaturen verglichen. Die Schnittmenge aus W_0 und R_1 $(= (B, C))$ ist nicht leer. Damit wird die Bearbeitung des *Chunks* von Thread 1 abgebrochen und neu gestartet.

Die Architektur bietet sequentielle Konsistenz auf *Chunk*-Ebene, da:

- *Chunks* atomar und isoliert ausgeführt werden.
- Commits in jedem Thread in definierter Reihenfolge ausgeführt werden.
- Eine globale Reihenfolge der *Chunks* existiert.

Damit wird die sequentielle Konsistenz auch auf Instruktions-Ebene sichergestellt. Zugleich wird eine relativ geringe Komplexität der Hardware-Implementierung erreicht, da keine komplexen Mechanismen zur Erkennung von Konsistenz-Verstößen benötigt werden. Einem Thread ist die Restrukturierung von Speicherzugriffen innerhalb eines *Chunks* erlaubt. Somit können viele Optimierungen bezüglich Laufzeit angewendet werden.

Fazit *Bulk Multicore* zeigt deutlich, dass die traditionelle *Von-Neumann*-Architektur keine Selbstverständlichkeit sein muss. Es handelt sich um einen neuartigen Aufbau von Multiprozessor-Systemen mit gemeinsamen Speicher. Die gesamte Ausführung der Instruktionen geschieht in atomaren *Chunks*. Im Gegensatz zu aktuell üblichen Architekturen, unterstützt *Bulk Multicore* sequentielle Konsistenz auf Hardware-Ebene. Dieses Verfahren kann eine signifikante Steigerung der Produktivität von Parallel-Programmierern bewirken, ohne dass die verwendeten Programmiersprachen oder Modelle verändert werden müssen. Um den produktiven Einsatz dieser Technologie zu ermöglichen, fehlt es jedoch an geeigneten Entwicklungsumgebungen [13]. Es muss zudem sichergestellt werden, dass der Einsatz von Signaturen auch bei großen Systemen mit vielen Operationen auf gemeinsamen Daten sinnvoll ist.

3 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurden unterschiedliche Ansätze vorgestellt, die ein gemeinsames Ziel verfolgen: Parallelprogrammierung soll durch Abstraktion weg von feingranularen Synchronisations- und Kommunikationsdetails führen.

Domänenspezifische Umgebungen können innerhalb ihres Anwendungsgebiets eine solche Forderung schon heute erfüllen. Durch die Verwendung fachspezifischer Informationen wird hohe Effizienz erreicht, ohne dass der Entwickler selbst für Optimierungen sorgen muss. Gerade

im Bereich domänenspezifischer Sprachen, wie z.B. das vorgestellte MATLAB®, kommen häufig *SIMD* Techniken zum Einsatz. Für den Entwickler stellen diese Befehle die wohl komfortabelste Möglichkeit dar, seine Anwendungen für die vorhandenen Multicore- Plattformen zu entwickeln. Die Beschränkung der einzelnen DSEs auf festgelegte Domänen ist allerdings auch deren größter Nachteil, um als Zukunft für Parallelität zu gelten. Statt der Entwicklung spezieller domänenspezifischer Techniken wäre es sinnvoller, generische Werkzeuge zu produzieren. Diese könnten Domänenexperten nutzen um effektive, parallele Umgebungen zu erstellen.

Technische Fortschritte im Bereich von Programmiermodellen oder Hardware-Architekturen reichen nicht aus, um Parallelität für die Massen bereitzustellen. Mindestens genauso wichtig ist die Art und Weise, wie Programmierer über parallele Synchronisation und Kommunikation denken. Damit dies verbessert werden kann, ist eine einheitliche "Muster-Sprache" wünschenswert, ähnlich zu der von objektorientierter Programmierung. Die Sammlung und Auswahl der notwendigen Entwurfsmuster kann und soll nicht Aufgabe Einzelner, sondern eine Gemeinschaftsarbeit von akademischen Gruppen sowie Praktikern der Industrie sein. "Muster-Sprachen" entwickeln sich durch eine kontinuierliche Interaktion zwischen den Autoren der Dokumentationen, verschiedenen Experten als Review-Partner, Programmierer als Verwender und Studenten, welche die Muster erlernen.
Um eine abstrakte Denkweise durch Entwurfsmuster zu fördern, wird auch implizite Unterstützung durch Programmiersprachen benötigt. Zudem muss diese Art über parallele Programmierung auch durch die Lehre unterstützt werden: *"The sequential algorithms and programming tricks that have served us so well for 50 years are the wrong way to think going forward"*[10].

Im Gegensatz zu Entwurfsmustern stellen Skelette ein formales Verfahren dar, um Wiederverwendbarkeit von parallelen Algorithmen und Datenstrukturen zu erreichen. Der Anwender von Skeletten kann auf eine Reihe von generischen Implementierungen zugreifen, um seine eigene Logik komfortabel in parallele Anwendungen zu überführen. Dafür stehen ihm Skelette unterschiedlicher Kategorien zur Verfügung. Es existieren z.B. Skelette für die Organisation von parallelen Aufgaben oder welche, die Berechnung von Daten auf mehreren Prozessen erledigen. Die größte Herausforderung für den Entwickler ist die Identifizierung von möglichen parallelisierbaren Abschnitten seiner Anwendung. Anschließend müssen geeignete Kompositionen von Skeletten ausgewählt werden, um hohe Performanz auf Zielsystemen zu erreichen. Leider sind die Identifikation von Bereichen und Komposition von Skeletten keine trivialen Aufgaben für unerfahrene Entwickler.

Das Aktor-Modell, als Beispiel paralleler Programmiermodelle und Repräsentant bekannter Entwurfsmuster, bietet implizite Abstraktion von Parallelität. Aktoren ermöglichen Anwendungen eine Aufteilung in autonome, interaktive, sowie asynchrone Komponenten, welche durch Nachrichtenaustausch miteinander kommunizieren. Durch diese Aufteilung wird eine hohe Flexibilität und Skalierbarkeit erreicht. Viele Programmiersprachen und Bibliotheken integrieren bereits das Aktor-Modell. Vor allem durch die Übergabe unveränderbarer Strukturen, wie bei funktionaler Programmierung, wird deterministisches Verhalten ohne Threads, Synchronisation und Sperren erreicht. Zusätzlich ermöglicht das Aktor-Konzept eine enorm abstrakte Sicht auf Problemstellungen, indem Systeme auf einzelne reaktive Komponenten zerlegt werden. Leider benötigen Aktoren je nach verwendeter Hardware-Plattform und Größe des Systems viel Kommunikationsaufwand. Häufig ist somit eine passende Abwägung bezüglich der Granularität notwendig, um die Vorteile bezüglich Skalierbarkeit erhalten zu können.

Durch Autotuner wird den Entwicklern paralleler Software ein mächtiges Werkzeug zur Verfügung gestellt, um eine hohe Effizienz zu erreichen. Es werden dabei viele verschiedene Varianten von Implementierungen auf einer speziellen Hardware-Plattform im Bezug auf die Laufzeit untersucht und die beste Alternative gewählt. Der Suchvorgang testet unzählige Optimierungen, was bei manchen Konstellationen sehr aufwändig sein kann. Gerade im Bereich von parallelen Architekturen existieren viele zusätzliche Optimierungsmöglichkeiten. Das Resultat

[10] Guy L. Steele Jr., Sun Microsystems Laboratories

ist häufig um ein vielfaches schneller als bei Verwendung von traditionellen Compilern. Die Aufgabe der Forschung im Bereich Autotuning liegt somit vor allem in der Erkundung von Suchstrategien für passende Optimierungen.

Bisher hat sich der von vielen erhoffte Wunsch, dass Transaktionsspeicher das Ende von *Deadlocks* einläuten, noch nicht vollkommen erfüllt. Diese Multiprozessor-Architektur hat den initialen Gedanken, dass Sperr-freie Synchronisation so effizient und einfach zu nutzen ist wie konventionelle Techniken, die auf exklusiven Ausschluss basieren. Gleichzeitiger Zugriff auf Speicherbereiche durch mehrere Teilnehmer soll somit kein Problem mehr darstellen. Transaktionen führen dazu vorläufige Änderungen auf dem Speicher aus, die nur bei erfolgreichem Abschluss persistent gespeichert werden. Es existieren viele verschiedene Arten von Transaktionsspeicher, manche basieren auf einer Hardware-Realisierung, manche werden vollständig in Software umgesetzt und einige verwenden Hybrid-Ansätze. Software-TM sind zum aktuellen Zeitpunkt noch nicht performant genug, Hardware-TM besitzen deutliche Einschränkungen bezüglich Komplexität bzw. Realisierbarkeit. Eine geeignete Lösung würde viele Schwierigkeiten der parallelen Programmierung lösen und Entwickler zu abstrakteren Ansätzen bringen. Leider ist nicht abzusehen, wann solche Lösungen für den produktiven Einsatz existieren.

Auch im Bereich der Hardware-Forschung werden große Anstrengungen gemacht, um den Entwicklern das Erstellen von parallelen Anwendungen zu vereinfachen. In den letzten Jahren war vor allem die zu verwendende Speicherarchitektur im Blick der Experten. Dabei stehen noch grundsätzliche Fragen an, wie z.B. ob die einzelnen Kerne direkten Zugriff auf gemeinsame Speicherbereiche haben sollen, oder Informationen per Nachrichten austauschen. Außerdem ist nicht klar, ob die Vermeidung von *Race Conditions* eine Aufgabe der Software, der Hardware oder ein gemeinsamer Einsatz sein soll. *Hardware-Transaktionsspeicher* oder *Bulk Multicore* sind Beispiele für vielversprechende Ansätze. Ob und welcher sich davon zu praxistauglichen Technologien entwickelt, liegt allerdings nicht nur bei den entsprechenden Forschungsgruppen. Die Hardware kann nur dann erfolgreich sein, falls genügend Software-Entwickler für die Plattformen zur Verfügung stehen und damit eine hohe Akzeptanz erreicht wird. Vielleicht muss sich daher das Vorgehen beim Entwurf von Parallelcomputern ändern. Anstatt zunächst nach einer neuen, vielversprechenden Plattform zu suchen, um später Software dafür zu entwickeln, wäre ein Top-Down-Ansatz vorteilhaft. Zunächst sollte nach Anwendungen gesucht werden, die viel Spielraum an Performanzbedarf haben. Anschließend können dafür geeignete Systeme entwickelt werden. Dies hätte den Vorteil, dass die benötigte Akzeptanz von vornherein sichergestellt wäre. Die Entwicklung neuer Architekturen muss nicht zwingend auf neuen Chips aufbauen, welche bis zu 5 Jahre bis zur Vollendung benötigen können. Durch FPGAs können Forscher vollständige Prototypen erstellen, auf denen die Software schnell genug läuft, um Innovationen zu finden.

In der vorliegenden Arbeit wurde ein genereller Blick auf die aktuellen Möglichkeiten gegeben, um Parallelität durch Abstraktion einfacher zu gestalten. Parallele Systeme sollen dadurch produktiv, effizient, korrekt, portabel und skalierbar erstellt werden können. Diese Herausforderungen bieten der internationalen Forschergemeinschaft eine Chance, das schnelle Wachstum der IT-Industrie weiterhin zu gewährleisten. Es ist eine einmalige Gelegenheit für jeden Informatiker, Teile der vorhandenen Software/Hardware-Architektur neu zu erfinden und damit die großen Schwierigkeiten der vergangenen Zeit zu überwinden.

Glossar

Cache Kohärenz
Durch die Sicherstellung von Cache-Kohärenz wird bei Multiprozessorsystemen mit mehreren CPU-Caches verhindert, dass einzelne Caches für gleiche Speicheradressen (inkonsistente) Daten zurückliefern.

Core Audio
Core Audio ist eine Programmierschnittstelle, welche von der Firma Apple entwickelt wurde. Sie ermöglicht unter dem Apple Betriebssystem Mac OS X Sound zu manipulieren und basiert auf der Cross-Plattform-Bibliothek OpenAL.

Deadlock
Ein Deadlock bezeichnet einen Zustand, bei dem ein oder mehrere Prozesse auf Betriebsmittel warten, die dem Prozess selbst oder einem anderen beteiligten Prozess zugeteilt sind.

Erlang
Erlang ist eine Programmiersprache, die bei Ericsson von Joe Armstrong und anderen entwickelt wurde. Sie ist nach dem dänischen Mathematiker Agner Krarup Erlang benannt, kann aber auch für Ericsson language stehen.

FPGA
Ein Field Programmable Gate Array (FPGA) ist ein integrierter Schaltkreis, in den eine logische Schaltung programmiert werden kann.

Gegenseitiger Ausschluss
Gegenseitiger Ausschluss (engl. Mutual Exclusion) wird bei Parallelprogrammierung verwendet, um gleichzeitigen Zugriff auf gemeinsame Ressourcen zu vermeiden.

Hardware-Thread
Der Begriff Hardware-Thread stammt aus der *Hyper-Thread*-Technologie. Dabei wird es Prozessoren ermöglicht, auf einem einzigen Hardware-Kern mehrere Hardware-Threads zu verwenden. Ein aktueller Core i7 Prozessor von Intel hat beispielsweise 2 Hardware-Threads pro Kern.

Hochleistungsrechnen
Hochleistungsrechnen (englisch: *high-performance computing* – HPC) ist ein Bereich des computergestützten Rechnens. Er umfasst alle Rechenarbeiten, deren Bearbeitung einer hohen Rechenleistung oder Speicherkapazität bedarf.

Livelock
Ein Livelock bezeichnet eine Art Deadlock mehrerer Prozesse, die im Gegensatz zum Deadlocknicht in einem Zustand verharren, sondern ständig zwischen mehreren Zuständen wechseln, aus denen sie nicht entkommen können.

Map Reduce
Map Reduce ist ein von Google eingeführtes Framework für nebenläufige Berechnungen über große Datenmengen auf Computerclustern. Dieses Framework wurde durch die in der funktionalen Programmierung häufig verwendeten Funktionen map und reduce inspiriert.

MPI
Das Message Passing Interface (MPI) ist eine Programmierschnittstelle, die den Nachrichtenaustausch paralleler Berechnungen auf verteilten Computersystemen beschreibt.

OpenMP
Open Multi-Processing (OpenMP) ist eine Programmierschnittstelle, welche die Parallelisierung von Programmen auf Thread- und Schleifenebene.

Race Condition
Eine *Race Condition* ist eine Konstellation, in der das Ergebnis einer Operation vom zeitlichen Verhalten bestimmter Einzeloperationen abhängt.

Scala
Scala ist eine Programmiersprache, welche von Martin Odersky entwickelt wurde. Sie ist konzeptuell für objektorientierte sowie funktionale Programmierelemente geeignet. Der Name Scala steht für "*scalable language*".

Scheduler
Ein Scheduler ist eine Logik, welche die zeitliche Ausführung mehrerer Prozesse regelt.

Thread
Threads sind eigenständige Aktivitäten in einem Prozess, die unabhängig von anderen Prozessteilen abgewickelt werden.

Literatur

[1] G. Agha. Actors: A Model of Concurrent Computation in Distributed Systems. Technical report, MIT Artificial Intelligence Laboratory, 1985.

[2] K. Asanovic. A View of the Parallel Computing Landscape. *Communications of the ACM*, Volume 52 Issue 10, October 2009.

[3] M. Astley, D.C. Sturman, and G. Agha. Customizable Middleware for Modular Distributed Software. *Communications of the ACM*, Volume 44 Issue 5, May 2001.

[4] W. Clinger. *Foundations of Actor Semantics*. PhD thesis, MIT Artificial Intelligence Laboratory, June, 1981.

[5] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid Transactional Memory. In *Proc. 12th Symposium on Architectural Support for Programming Languages and Operating Systems*, 2006.

[6] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation, 2009.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[8] S Gorlatch and C. Lengauer. Abstraction and performance in the design of parallel programs: overview of the SAT approach. *Acta Informatica*, 36(9):761–803, 2000.

[9] J. Heering and M. Mernik. Domain Specific Languages in Perspective. Technical report, CWI Amsterdam and University of Maribor, 2008.

[10] M. Herlihy. Transactional Memory: Architectural Support for Lock-Free Data Structures. Technical report, Cambridge Research Laboratory, 1993.

[11] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software Transactional Memory for Supporting Dynamic-Sized Data Structures. In *Proc. 22th Annual ACM Symposium on Principles of Distributed Computing*, 2003.

[12] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier, 2008.

[13] W.W. Hwu and M. Snir. Parallel @Illinois - Parallel Computing Research at Illinois. Technical report, University of Illinois, Nov. 2008.

[14] S. Kamil, C. Chan, K Datta, Williams S., J. Shalf, L. Oliker, and Yelick K. In-Place Auto-tuning of Structured Grid Kernels. Technical report, University of California at Berkeley, 2008.

[15] D. Lea. *Concurrent Programming in Java(TM): Design Principles and Patterns (3rd Edition)*. Addison-Wesley, 2006.

[16] T.G. Mattson, B.A. Sanders, and B.L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004.

[17] F.A. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2002.

[18] V. Subramaniam. *Programming Scala: Tackle Multicore Complexity on the Java Virtual Machine*. Pragmatic Bookshelf, 2009.

[19] J. Torrellas, L. Ceze, J. Tuck, C. Cascaval, P. Montes, W. Ahn, and M. Prvulovic. The Bulk Multicore Architecture for Improved Programmability. *Communications of the ACM*, Volume 52 Issue 12, December 2009.

[20] D. Wampler and A. Payne. *Programming Scala*. O'Reilly, 2008.

[21] S. Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, University of California at Berkeley, 2008.