**Hochschule**
**Bonn-Rhein-Sieg**
*University of Applied Sciences*

**Fachbereich Informatik**
*Department of Computer Science*

# Master Thesis

im Studiengang
Master of Science in Computer Science

## Vergleich paralleler Programmieransätze

***von***
***Christian Hornschuh***

| | |
|---|---|
| Erstbetreuer: | Prof. Dr. Rudolf Berrendorf |
| Zweitbetreuer: | Prof. Dr. Peter Becker |
| Eingereicht am: | 06.09.2010 |

## Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbst angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

_____

## Urheberrecht

# Abstract

Parallelität ist heute an vielen Stellen in Rechnerarchitekturen vorhanden. Damit sie jedoch effizient genutzt werden kann, ist eine manuelle Spezifizierung durch den Programmierer derzeit noch unerlässlich. Mit dem Aufkommen von Mehrkernprozessoren und General Purpose Graphics Processing Units (GPGPUs) wurde die parallele Programmierung auch für den Massenmarkt interessant, und es haben sich zahlreiche Programmiersprachen, parallele Bibliotheken und eigene Programmiermodelle entwickelt, mit denen Parallelität durch den Programmierer spezifiziert werden kann.

Diese Master Thesis beschäftigt sich mit einer umfassenden Sichtung, Klassifikation und Evaluation von Ansätzen zur parallelen Programmierung. Die Untersuchungen werden sich hauptsächlich auf solche Ansätze beschränken, die einen gemeinsamen Adressraum voraussetzen, einen gemeinsamen Adressraum selbst schaffen (in Hardware, Betriebssystem, Compiler, Bibliothek, …) oder die die parallele Rechenleistung aktueller Grafikkarten ausnutzen können. Die Sichtung versucht alle praxisrelevanten bzw. derzeit interessanten Ansätze zu erfassen und gemäß einer bestimmten Methodik zu klassifizieren. Die Evaluation erfolgt für einige ausgewählte Ansätze sowohl vom Umfang als von der auch Komplexität überschaubarer, anwendungsorientierter Programme bzw. Programmkerne, die unterschiedlicher Natur sind (Form der Parallelität, Zugriffsmuster, ...) und demzufolge als Repräsentanten ganzer Anwendungsklassen anzusehen sind.

# Inhaltsverzeichnis

# 1 Einführung

## 1.1 Motivation

Innerhalb der letzten Jahre zeichnete sich ein Trend bei den Prozessorherstellern ab, bei dem man bei der Entwicklung vom klassischen Prozessor mit nur einem Kern zu Prozessoren mit mehreren Kernen übergegangen ist. Solche Mehrkern- bzw. Multi-Core-Prozessoren wurden im Jahr 2005 von den beiden großen Prozessorherstellern AMD und Intel in Form von Dual-Cores[1] für den Endkundenmarkt vorgestellt. In den folgenden Jahren konnte die Anzahl an Kernen pro Prozessorchip weiter gesteigert werden, so dass Kunden bzw. Hersteller von Serversystemen aktuell auf Prozessoren mit bis zu 12 Kernen zurückgreifen können.

Physikalische Grenzen, welche die maximale Taktrate eines Prozessors beschränken bzw. die eine ausreichende Kühlung der Prozessoren verhindern, sind der Grund für diese Trendwende. Die Prozessorgeschwindigkeit lässt sich, neben der Taktrate, auch durch eine Erhöhung der Effizienz (Instructions per Cycle, IPC) steigern, die aber ebenfalls gewissen Grenzen beim Chipdesign unterliegt. Deshalb sind die Hersteller davon abgewichen, immer schnellere Prozessoren nur auf Basis von höheren Taktraten und einer besseren Effizienz zu produzieren, sondern erzielen eine höhere Geschwindigkeit nun mittels einer gesteigerten Arbeitsparallelität durch die Verwendung von vielen Kernen auf einem Prozessor. [Rau08] Dieser Wandel macht es erforderlich, Anwendungen auf Basis von neueren Ansätzen zu entwickeln, die den Entwicklern die Ausnutzung der parallelen Verarbeitungsleistung auf einfache Weise ermöglichen.

Parallel zum Wandel im Prozessormarkt wurden Möglichkeiten gesucht, um die hohe Leistungsfähigkeit aktueller Grafikkarten auch außerhalb von Computerspielen nutzen zu können. Diese kommt aufgrund des Aufbaus der Grafikprozessoren zustande, in denen zwar nur simple, aber dafür sehr viele Recheneinheiten untergebracht sind. Durch die Einführung der Vertex- bzw. Pixel-Shader wurde es möglich, diese Einheiten nach eigenen Wünschen programmieren zu können. [Pla09]

[1] Prozessoren mit zwei Kernen

Dafür stellten AMD/ATI und Nvidia im Jahr 2006 ihre jeweils eigenen, herstellerspezifischen Schnittstellen in Form von Stream bzw. CUDA vor. [GPG10] Mit OpenCL wurde drei Jahre später ein einheitlicher Standard definiert, mit dem sich ATI- und Nvidia-Grafikkarten gleichermaßen programmieren lassen. Durch die hohe Anzahl an Recheneinheiten aufgrund der parallel ausgelegten Architektur, unterstützen alle Ansätze einen hohen Grad an Parallelität.

Die stetig wachsende Bedeutung der parallelen Programmierung, sowohl von Prozessoren als auch von Grafikkarten, sowie der Wunsch der Entwickler nach einfach verwendbaren und effizienten Ansätzen, bilden die Grundlage und Motivation für diese Arbeit.

## 1.2 Zielsetzung

Das Ziel dieser Master Thesis ist es, eine möglichst vollständige Sammlung an praxisrelevanten bzw. interessanten parallelen Programmiermodellen für programmierbare Grafikkarten und Mehrprozessor- bzw. Mehrkern-Systeme zu schaffen, die auf einem gemeinsamen Speicher basieren bzw. diesen selbst schaffen (mittels Hardware, Betriebssystem, Bibliothek, Compiler usw.). Bei der Sammlung sollen gewisse Aspekte anhand ausgewählter Kriterien für die Modelle herausgearbeitet und in einem kleineren Umfang Leistungsuntersuchungen durchgeführt werden, mit denen am Ende dieser Arbeit eine übersichtliche Bewertung der verschiedenen Modelle erfolgen kann.

## 1.3 Aufbau

Nach der Einführung wird das zweite Kapitel eine kurze Definition darüber geben, was parallele Programmiermodelle sind und Besonderheiten aufzeigen, anhand derer sich die Modelle signifikant voneinander unterscheiden lassen. Um im vierten Kapitel die verschiedenen parallelen Programmiermodelle in geeigneter Weise sichten zu können, werden zuvor die Anforderungen herausgearbeitet, die ein Modell erfüllen sollte und anhand derer letztendlich auch die Bewertung erfolgen kann. Für einige ausgewählte Modelle werden im fünften Kapitel Laufzeitmessungen von repräsentativen Anwendungen durchgeführt, die, zusammen mit den Ergebnissen aus den vorherigen Kapiteln, die Evaluation bilden. Ein Fazit wird die Thesis zum Abschluss bringen, in dem die wichtigsten Erkenntnisse nochmals komprimiert aufbereitet werden.

# 2 Eigenschaften von parallelen Programmiermodellen

Dieses Kapitel soll parallele Programmiermodelle erklären und Kriterien aufzeigen, anhand derer sich solche Modelle voneinander unterscheiden lassen. Im weiteren Verlauf dieser Arbeit werden die Begriffe Programmiermodell und Programmieransatz synonym verwendet.

## 2.1 Definition

Parallele Programmiermodelle haben die Aufgabe, eine Verbindung zu schaffen zwischen dem, was der Programmierer entwickeln möchte (die Anwendung) und dem, wie er das auf der ihm vorliegenden Hardware erreichen kann (die Implementierung). Das Programmiermodell ist maßgeblich für den Aufwand verantwortlich, der für die parallele Entwicklung und der dabei möglichen Effizienz betrieben werden muss. [Asa06]

Im Gegensatz zu Maschinen- und Architekturmodellen sind Programmiermodelle auf einer höheren Abstraktionsebene angesiedelt. Maschinen- und Architekturmodelle beschreiben die grundlegenden Eigenschaften eines Rechensystems wie Prozessortyp, Registerzahl usw., auf denen die (parallelen) Programmiermodelle aufbauen. Durch die Wahl eines Programmiermodells wird somit einerseits die Hardware festgelegt, auf der die mit diesem Modell entwickelten Programme lauffähig sind und andererseits liefert es dem Programmierer gewisse Möglichkeiten in Form von Compilern, Laufzeitbibliotheken und Funktionen. Diese Funktionen können bei der Implementierung genutzt werden und sollen, im Falle von parallelen Programmiermodellen, die Entwicklung von Programmen mit einem hohen Grad an Verarbeitungsparallelität ermöglichen bzw. vereinfachen. [Rau07]

Anhand bestimmter Kriterien lassen sich die verschiedenen Modelle zur parallelen Programmierung voneinander unterscheiden, die im Folgenden vorgestellt werden.

## 2.2 Art der Systemarchitektur

Die grundlegendste Differenzierung von parallelen Programmiermodellen ist durch die dem System zugrunde liegende Rechnerarchitektur gegeben. Dadurch lassen sich die verschiedenen Ansätze bereits grob klassifizieren.

### 2.2.1 Klassifizierung nach Flynn

Eine sehr weit verbreitete Methode zur Klassifizierung paralleler Systeme ist die von Michael J. Flynn. Diese Klassifizierung basiert auf dem Konzept von *Streams*. Ein Stream ist eine Folge von Daten bzw. Instruktionen, die von einem Prozessor ausgeführt werden. Damit konnte Flynn insgesamt vier Klassen von Systemen definieren [Fly72]:

- SISD (Single Instruction Single Data): Entspricht dem klassischen Von-Neumann Rechner, bei dem eine Anweisung auf einem Datenelement ausgeführt werden kann.
- SIMD (Single Instruction Multiple Data): Vektor-Prozessoren, die eine Anweisung auf mehrere Daten anwenden können. Beispiele dafür sind die SSE-Einheiten[2] aktueller Prozessoren.
- MISD (Multiple Instruction Single Data): Mehrere Instruktionen können pipelineartig auf einem Datenelement ausgeführt werden.
- MIMD (Multiple Instruction Multiple Data): Mehrprozessor-Systeme, die mehrere Anweisungen zeitgleich auf mehreren Daten durchführen können.

Das Problem an dieser Einteilung besteht darin, dass viele der heutigen Computer in die MIMD-Klasse fallen. [Rau08] Daher werden weitere Kriterien zur Klassifizierung aufgrund der Systemarchitektur herangezogen.

### 2.2.2 Systeme mit gemeinsamem Speicher

Bei Systemen mit einem gemeinsamen Speicher *(shared memory)* sind alle Prozessoren dieses Systems an einen einzigen Speicherbereich angeschlossen. Dies ermöglicht den einfachen Austausch von Daten über gemeinsame Variablen, fügt aber auch zugleich Komplexität bei gleichzeitigen Schreib- und Lesezugriffen auf diese Variablen hinzu. [Wil99]

[2] Streaming SIMD Extensions

Außerdem muss durch Kohärenz-Protokolle dafür gesorgt werden, dass sich die verschiedenen Caches aller Prozessoren immer in einem widerspruchsfreien Zustand zum gemeinsamen Speicher befinden. [Bau06] Abbildung 2-1 stellt den schematischen Aufbau eines solchen Systems dar:

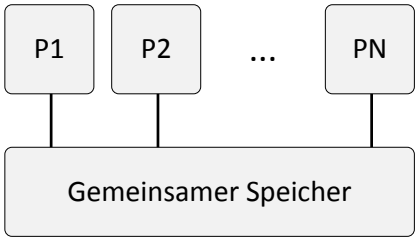



**Abbildung 2-1: Skizzierung der Architektur mit einem gemeinsamen Speicher [Cha01]**

Eine Beschränkung dieser Systeme ergibt sich aus der maximalen Anzahl an Prozessoren, die physisch und wirtschaftlich an einen einzelnen Speicherbereich angebunden werden können. Somit sind Systeme auf Basis eines gemeinsamen Speichers nicht beliebig erweiterbar und finden keine Verwendung bei massiv parallelisierten Anwendungen. [Bau06]

### 2.2.3 Systeme mit verteiltem Speicher

Systeme mit verteiltem Speicher bestehen aus mehreren, für sich eigenständigen Knoten (z.B. ganz normale Computer mit lokalem Speicher), die über ein schnelles Kommunikationsnetz[3] miteinander verbunden sind.

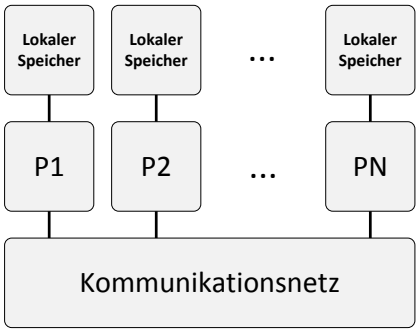



**Abbildung 2-2: Skizzierung der Architektur mit einem verteilten Speicher [Cha01]**

Wie Abbildung 2-2 zeigt, hat jeder Knoten lediglich Zugriff auf seinen lokalen Speicherbereich. Der Zugriff auf Daten anderer Knoten kann nur durch Kommunikation geschehen. Der Vorteil dieses Ansatzes ist, dass man relativ einfach weitere Knoten dem Gesamtsystem hin-

[3] Wie z.B. Myrinet-2000, Infiniband oder QsNet.

zufügen kann. Dadurch eignen sich solche Systeme für Anwendungen, die von einer massiven Parallelität profitieren können. Der Nachteil dieser Systeme ist die Geschwindigkeit des Verbindungsnetzes, über das gemeinsame Daten ausgetauscht werden müssen sowie der damit verbundene programmiertechnische Aufwand. [Bau06]

Diese Art von Systemen bildet nicht den Schwerpunkt dieser Arbeit. Deshalb soll an dieser Stelle nur in verkürzter Form auf ein wesentliches Programmiermodell dieser Klasse eingegangen werden. Das *Message Passing Interface* (MPI) ist ein sehr bekannter und weit verbreiteter Ansatz zur Programmierung von Systemen mit einem verteilten Speicher. MPI bietet in der ersten Version Unterstützung für die Programmiersprachen FORTRAN 77 und C, die in Version 2 im April 1997 mit der Unterstützung für FORTRAN 90 und C++ erweitert wurde. [Qui03] Neben kommerziellen Bibliotheken steht mit MPICH2 auch eine frei verwendbare der Öffentlichkeit zur Verfügung. MPICH2 ist eine quelloffene Implementierung des MPI-1 und MPI-2 Standards und kann unter Linux, Mac OS X, Solaris und Windows eingesetzt werden. [Arg10] Für weitere Informationen über MPICH2 und die Programmierung des MPI sei auf die Literatur verwiesen, wie z.B. [Arg101], [Bau06], [Gra03], [Gro99], [Gro09], [Qui03] und [Rau07].

### 2.2.4 Systeme mit verteiltem gemeinsamen Speicher

Systeme mit verteiltem gemeinsamen Speicher, auch DSM[4]-Systeme genannt, versuchen die Vorteile beider bisher vorgestellter Architekturen zu kombinieren. Abbildung 2-3 soll diesen Zusammenhang verdeutlichen:

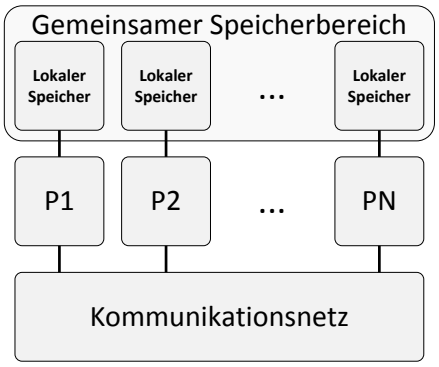



**Abbildung 2-3: Skizzierung der Architektur eines DSM-Systems [Wil99]**

[4] Distributed Shared Memory

DSM-Systeme bilden einen Cluster analog zu Systemen mit verteiltem Speicher. Sie präsentieren dem Programmierer jedoch einen (logischen) gemeinsamen Adressraum. Zugriffe auf entfernte Speicherbereiche erfolgen weiterhin durch Kommunikationsoperationen über ein Netzwerk, jedoch mit dem Unterschied, dass sie für den Programmierer transparent sind. Ein solches System kann in Hardware, Software oder in einer Mischform realisiert werden. [Wil99] Die konkrete Umsetzung ist im weiteren Verlauf dieser Arbeit jedoch nicht von Bedeutung. Wichtiger hingegen ist die Unterstützung durch Programmiermodelle, die einen Nutzen aus der Speicherorganisation ziehen können. Solche Programmiermodelle werden unter den PGAS[5]-Sprachen zusammengefasst und berücksichtigen die Architektur durch Assoziation der Daten mit bestimmten Threads bzw. Knoten. [Rau07] Der Begriff PGAS setzt sich zum einen aus dem globalen Speicherbereich zusammen, der den Zugriff auf Speicherbereiche anderer Knoten ermöglicht, und zum anderen aus der Partitionierung, bei der lokale Speicherzugriffe schneller erfolgen können als globale Zugriffe (aufgrund des zusätzlichen Kommunikationsaufwandes). [Yel06]

### 2.2.5 Systeme mit programmierbaren Grafikkarten

Wurden Grafikkarten bis vor ein paar Jahren ausschließlich zur beschleunigten Darstellung zwei- und dreidimensionaler Objekte genutzt, können sie heute auch für allgemeinere Aufgaben *(general purpose)* verwendet werden. Dies prägte den Begriff der GPGPU[6].

Entsprechende Grafikkarten können zur Beschleunigung bestimmter Operationen sowohl auf Systemen mit gemeinsamem als auch verteiltem Speicher ergänzend genutzt werden. Die Architektur einer Graphics Processing Unit (GPU) unterscheidet sich dabei wesentlich von der einer CPU. Vereinfacht ausgedrückt sind CPUs dafür konzipiert worden, einen sequentiellen Anweisungsstrom möglichst schnell und effizient zu bearbeiten. Dabei helfen Technologien wie bspw. Sprungvorhersagen oder Vektoreinheiten. Bei Grafikprozessoren kommen sehr einfache Recheneinheiten zum Einsatz, von denen aufgrund ihrer geringeren Komplexität erheblich mehr verwendet werden können. Das Ziel bei vielen Recheneinheiten ist es, möglichst viele Pixel (oder Daten), die in aktuellen Computerspielen in großem Umfang vorhanden sind, gleichzeitig zu berechnen. [Pla09]

[5] Partitioned Global Address Space
[6] General Purpose Computation on Graphics Processing Unit

Abbildung 2-4 stellt einen schematischen Aufbau aktueller Grafikprozessoren dar:

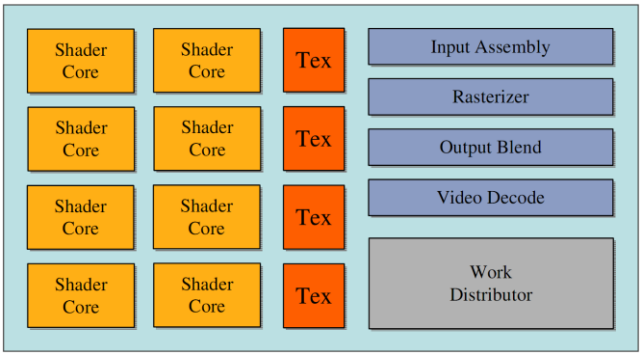



**Abbildung 2-4: Schematischer Aufbau einer programmierbaren GPU [Str09]**

Die für den Programmierer interessantesten Komponenten sind die *Shader Cores*, die letztendlich die Ausführung des Codes übernehmen. Die Anzahl der Shader Cores ist je nach Grafikchip-Architektur unterschiedlich und beträgt bspw. 10 bei einer *ATI Radeon HD 4890* und 30 einer *Nvidia Geforce GTX 280*. Jeder Shader Core verfügt über eine bestimmte Anzahl programmierbarer SIMD-Einheiten, die darüber hinaus mehrere ALUs[7] enthalten können und die Ausführung der Operationen übernehmen. Auch hier ist die Anzahl an vorhandenen SIMD-Einheiten pro Core und die darin enthaltenen ALUs von der jeweiligen Architektur abhängig[8]. Die Arbeitsweise ist ähnlich zu den Vektoreinheiten aktueller Prozessoren, nur dass die Operationen auf mehr Daten angewendet werden können und die Hardware die Vektorisierung übernimmt[9]. [Str09] Ein weiterer Unterschied zwischen (Multi-Core)-CPUs und GPUs betrifft die simultane Ausführung von Threads. Bei einer CPU mit mehreren Kernen kann jeder Kern einen unterschiedlichen Thread ausführen. In GPUs werden mehrere Threads zu Thread-Gruppen zusammengefasst. ATI nennt diese Gruppen *Wavefronts*, bei Nvidia heißen sie *Warps*. Alle Threads innerhalb einer Gruppe müssen alle Anweisungen aufgrund der vereinfachten Recheneinheiten gemeinsam ausführen. Dies wird in der Literatur auch mit *Single Instruction Multiple Threads* (SIMT) bezeichnet. [Pla09]

Aufgrund des unterschiedlichen Aufbaus einer GPU, sind zu deren Programmierung ebenfalls andere Ansätze erforderlich. Diese werden in einem späteren Kapitel gesondert vorgestellt.

[7] Arithmetical Logical Unit – kann in Grafikprozessoren Ganz- und Fließkommazahlen verarbeiten
[8] HD 4890: 16 SIMD-Einheiten mit jeweils 5 ALUs / GTX 280: 8 SIMD-Einheiten mit jeweils 2 ALUs
[9] Im Gegensatz zu CPUs, wo dies vom Compiler geschehen muss.

## 2.3 Ebenen der Parallelität

Bei der Parallelisierung eines Programmes mit einem parallelen Programmiermodell ist eine Betrachtung bestimmter Bereiche im Programm erforderlich, die sich für eine Parallelisierung eignen könnten. Die Betrachtung kann auf unterschiedlichen Ebenen erfolgen, die sich in ihrer Granularität unterscheiden. [Rau07] Somit lassen sich die Programmiermodelle weiterhin dahingehend klassifizieren, für welche Ebenen die Modelle geeignete Methoden und Funktionen zur Parallelisierung bereitstellen. Diese Ebenen sollen im Folgenden beschrieben werden.

### 2.3.1 Instruktionsparallelität

Bei der Instruktionsparallelität wird zwischen einer Parallelität unterschieden, die automatisch durch die Hardware durchgeführt wird (Instruktionspipelining) und somit transparent für den Programmierer ist und einer, auf die mittels SIMD Prozessoren und deren Erweiterungen wie MMX, SSE usw. in geringem Umfang Einfluss genommen werden kann. Diese Form der Parallelität zielt darauf ab, sequentielle Anweisungen in einem Programm zu erkennen und, sofern keine Abhängigkeiten zwischen diesen Anweisungen existieren, sie zu parallelisieren. [Gra03] Wann und wie dies durchgeführt wird ist von der eingesetzten Hardware bzw. dem Compiler abhängig und kann unabhängig vom gewählten parallelen Programmiermodell durchgeführt werden. Aus diesem Grund wird auf die Instruktionsparallelität in dieser Arbeit nicht weiter eingegangen.

### 2.3.2 Datenparallelität

Die Datenparallelität betrachtet die Bereiche in einem Programm, bei denen gleiche oder sehr ähnliche, unabhängige Operationen auf einer großen Menge an Daten durchgeführt werden. [Qui03] Eine Möglichkeit zur beschleunigten Ausführung ist die Verwendung von Vektorprozessoren in Kombination mit Vektoranweisungen. Dies entspricht dem SIMD-Modell, das unter der Systemarchitektur bereits vorgestellt wurde. [Rau07] Eine andere und in der Praxis häufiger eingesetzte Variante ist der Einsatz von parallelen Schleifen. [Bau06]

### 2.3.3 Taskparallelität

Anders als bei der Datenparallelität werden bei der Taskparallelität die Bereiche in einem Programm betrachtet, die unabhängig voneinander sind und deshalb parallel ausgeführt werden können. Zum Beispiel lassen sich auf diese Weise unterschiedliche Aufgaben auf einer Menge an Daten parallel durchführen. Solche unabhängigen Aufgaben werden auch Tasks genannt. [Rau07]

### 2.3.4 Pipelining

Pipelining bei der Anwendungsentwicklung ist sehr ähnlich zum Instruktionspipelining, welches in aktuellen Prozessoren eingesetzt wird. Ausgangspunkt ist eine sequentielle Aufgabe, die in viele kleinere Aufgaben (Tasks) zerlegt werden kann. Die entstandenen Tasks sind jedoch voneinander abhängig, so dass ein Task erst dann mit der Bearbeitung beginnen kann, wenn sein Vorgänger abgeschlossen ist. Solche Verfahren werden auch als Producer-Consumer Verfahren bezeichnet, bei denen die Ergebnisse in den verschiedenen Pipelinestufen schrittweise verfeinert werden. [Gra03] Der Vorteil an diesem Verfahren entsteht dann, wenn die Berechnungen auf mehreren Daten durchgeführt werden sollen. Man stelle sich eine Pipeline mit insgesamt fünf Berechnungsstufen vor und in jedem Schritt kann eine Stufe ausgeführt werden. Ist die Pipeline leer, so dauert es insgesamt fünf Schritte, bis das Ergebnis für das erste Datenelement vorliegt. Die nachfolgenden Ergebnisse kommen nun schrittweise zustande, da die Pipeline gefüllt ist und pro Schritt ein Ergebnis fertig wird. [Qui03]

Pipelining-Verfahren lassen sich effizient implementieren, wenn parallele Programmiermodelle spezielle Konstrukte, z.B. in Form von Agenten und Methoden zum Nachrichtenaustausch, bereitstellen.

## 2.4 Abstraktionsgrad

Der Abstraktionsgrad eines parallelen Programmiermodells gibt an, in welchem Umfang sich der Programmierer mit der Parallelisierung von Programmteilen beschäftigen muss. Dabei kann grundsätzlich zwischen zwei Arten von Modellen unterschieden werden [Rau07]:

- Modelle für implizite Parallelität
- Modelle für explizite Parallelität

### 2.4.1 Implizite Parallelität

Programmiermodelle für implizite Parallelität verfügen über einen hohen Abstraktionsgrad und sind für den Programmierer am leichtesten zu handhaben. Sie bedürfen keinerlei Angaben zur Parallelität. Die einzige Aufgabe des Programmierers ist es, den Zweck des Programms zu spezifizieren ohne angeben zu müssen, wie dies erreicht werden soll. Dadurch gestaltet sich die Entwicklung ähnlich der eines sequentiellen Programms. [Ski97]

Solche Ansätze erleichtern einerseits die Entwicklung paralleler Anwendungen, setzen aber andererseits sehr fortgeschrittene Compiler voraus, die eine effiziente Parallelisierung der Programmteile ermöglichen. Die Compiler müssen dafür komplexe Abhängigkeitsanalysen durchführen, um parallelisierbare Bereiche zu erkennen. In der Praxis liefern solche Compiler derweil noch unbefriedigende Ergebnisse. [Rau08]

Neben parallelisierenden Compilern kann eine implizite Parallelität aber auch durch parallele Programmiersprachen ermöglicht werden. [Sil99] Mitte der 90er Jahre erreichte die Forschung in diesem Bereich ihren Höhepunkt, und es entstanden parallele Sprachen wie Occam oder Linda. Dass sich diese Sprachen nicht weiter durchgesetzt haben, ist womöglich auf die bereits damals hohe Verbreitung und Popularität von Sprachen wie C oder FORTRAN zurückzuführen. [Bau06] Mit Fortress steht eine aktuelle Sprache zur Verfügung, die erneut Aspekte der impliziten Parallelität aufgreift [Rau08] und die im Verlauf dieser Arbeit vorgestellt wird.

### 2.4.2 Explizite Parallelität

Aufgrund des Mangels an parallelen Programmiersprachen und den nicht zufriedenstellenden Ergebnissen parallelisierender Compiler wird vorwiegend die Angabe expliziter Parallelität genutzt. Dadurch können simplere Compiler verwendet werden, da der Programmierer nun dafür verantwortlich ist, parallelisierbare Stellen zu erkennen und geeignete Funktionen und Methoden zu deren Parallelisierung einzusetzen. Dies erfordert jedoch einen höheren Programmieraufwand. [Rau07]

Die explizite Parallelität kann darüber hinaus noch weiter unterschieden werden, indem der zu bewältigende Aufwand bei einer Parallelisierung genauer betrachtet wird. Auf Basis der Veröffentlichungen von [Kas08] und [Ski97] werden die folgenden Punkte zur weiteren Differenzierung genutzt:

- Zerlegung der Aufgaben
- Verwaltung der Parallelität
- Abbildung auf Systemressourcen
- Kommunikation
- Synchronisation

Mit Zerlegung der Aufgaben ist gemeint, inwiefern die zu bewältigende Arbeit vom Programmierer für eine parallele Ausführung und für eine bestimmte Anzahl an Threads aufgeteilt werden muss. Bei der Taskparallelität und beim Pipelining ist die Zerlegung immer explizit, da der Programmierer bereits durch den Aufbau des Programms dafür gesorgt hat, dass bestimmte Teile parallel zueinander ausgeführt werden können. Diese Bereiche benötigen anschließend, je nach Ansatz, nur eine explizite Kennzeichnung. Bei der Datenparallelität wird von einer impliziten Zerlegung gesprochen, sofern der Ansatz die Partitionierung der Daten und die Zuweisung auf die Threads übernimmt. Muss der Programmierer hingegen selber für eine geeignete Aufteilung sorgen, spricht man von einer expliziten Zerlegung. [Ski97]

Die Verwaltung der Parallelität und die Abbildung auf die Systemressourcen beschreiben Aspekte eines Modells, die sich mit der konkreten Umsetzung der Parallelität auf einem bestimmten System beschäftigen. Muss sich der Programmierer innerhalb der Anwendung nicht um die Erzeugung, die Verwaltung und das Löschen von Threads kümmern, so handelt es sich um eine implizite Verwaltung. Andernfalls ist die Verwaltung explizit. Wird dem Programmierer weiterhin die Aufgabe abgenommen, die Threads in geeigneter Weise auf die vorhandenen Systemressourcen, wie z.B. Prozessoren, abzubilden, so ist auch die Abbildung implizit. [Kas08,Ski97]

Kommunikation findet in der Anwendung implizit statt, falls keine speziellen Methoden zum Datenaustausch zwischen mehreren Threads oder verschiedenen Knoten genutzt werden

müssen. Ist hingegen die Verwendung von bestimmten Methoden zum Nachrichtenaustausch erforderlich, so erfolgt die Kommunikation explizit.

Der Begriff der Synchronisation fasst bestimmte Methoden zusammen, um die Arbeit der verschiedenen Threads bzw. Tasks zu koordinieren. Bspw. sind Synchronisationsmethoden immer dann nötig, wenn der Lese- und Schreibzugriff auf gemeinsame Variablen kontrolliert oder auf die Fertigstellung aller Threads einer parallelen Schleife gewartet werden soll. Muss der Programmierer in allen Fällen selber die Koordination zwischen den verschiedenen Threads übernehmen, ist die Synchronisation explizit durchzuführen. Ist kein weiteres Eingreifen durch den Programmierer erforderlich, so geschieht die Synchronisation implizit. [Kas08] Auch sind Mischformen möglich, in denen gewisse Operationen implizit synchronisiert werden, während andere eine explizite Synchronisierung benötigen.

## 2.5 Erscheinungsform

Ein paralleles Programmiermodell kann auf unterschiedliche Weise spezifiziert bzw. dem Programmierer zur Verfügung gestellt werden. Die Literatur unterscheidet hier zwischen den folgenden vier Arten [Rei07,Sil99]:

- Automatische Parallelisierung durch den Compiler
- Spracherweiterungen
- Parallele Bibliotheken
- Parallele Sprachen

Die Reihenfolge, die für die Auflistung der unterschiedlichen Erscheinungsformen gewählt wurde, spiegelt den für Programmierer verbundenen Aufwand wieder. Somit ist der zuvor angesprochene Abstraktionsgrad zum Teil von der jeweiligen Erscheinungsform des Ansatzes abhängig. Der größte Vorteil an den ersten drei Varianten ist, dass sie, bei entsprechender Unterstützung, mit bekannten Programmiersprachen verwendet werden können (mit ansteigendem Aufwand). Dies ist auch bei größer angelegten und kommerziellen Projekten, die bereits über eine große Codebasis verfügen, von hoher Bedeutung. Die Verwendung von parallelen Sprachen benötigt hingegen eine höhere Einarbeitungszeit und erfordert bei existierenden Projekten ein Neuschreiben der Codebasis. [Rei07]

# 3 Anforderungen an parallele Programmiermodelle

Für eine umfangreiche Sichtung und Evaluation paralleler Ansätze ist, neben grundlegenden Eigenschaften zur Klassifikation, eine Sammlung an Anforderungen erforderlich, anhand derer die Ansätze weiterführend untersucht und bewertet werden können.

Zur Festlegung der Anforderungen werden sowohl die bereits bekannten Eigenschaften und Klassifikationskriterien paralleler Programmiermodelle herangezogen als auch solche, die nicht-technischer Natur sind und beispielsweise auf Aspekte der Verständlichkeit und Einfachheit dieser Ansätze eingehen. Viele der im Folgenden gesammelten Punkte beruhen auf den Veröffentlichungen von [Asa06], [Kas08] und [Ski97].

Dabei sei angemerkt, dass die Kriterien, die für die Sichtung und Bewertung der Programmieransätze herangezogen werden, auf Basis einer ausgiebigen Literaturrecherche und subjektiven Vorstellungen ausgewählt wurden, jedoch keinen Anspruch auf Vollständigkeit erheben.

## 3.1 Aufwand bei der Programmierung

Der bei der Verwendung eines bestimmten parallelen Programmiermodells einhergehende Aufwand entspricht dem Anteil an Quellcode, der durch die Nutzung paralleler Methoden und Funktionen verändert bzw. eingefügt werden muss, um die Anwendung zu parallelisieren. [Sil99]

Dieser Aufwand wird sowohl durch den Abstraktionsgrad des zugrundeliegenden Modells als auch durch dessen Erscheinungsform festgelegt. Der Abstraktionsgrad bestimmt den Umfang, inwieweit der Programmierer selber für Parallelität sorgen muss (explizit) oder ob gar keine weiteren Angaben erforderlich sind (implizit). [Sil99] Neben dieser groben Einstufung wird eine feinere Einteilung anhand von folgenden Merkmalen durchgeführt: Zerlegung der Aufgaben, Verwaltung der Parallelität, Abbildung auf Systemressourcen und Synchronisation. [Ski97] Auf Aspekte der Kommunikation wird nur bei der Untersuchung der Ansätze für Systeme mit programmierbaren Grafikkarten und Systemen mit einem gemeinsamen verteil-

ten Speicher eingegangen. Bei den Ansätzen für Systeme mit einem gemeinsamen Speicher, kann die Kommunikation direkt über den Hauptspeicher und ohne spezielle Kommunikationsoperatoren erfolgen. [Rau07]

## 3.2 Funktionaler Umfang

Der funktionale Umfang soll hier die unterschiedlichen Ebenen umfassen, die mit Hilfe des Programmiermodells parallelisiert werden können. Konkret wird in dieser Arbeit die Unterstützung für Datenparallelität, Taskparallelität und Pipelining betrachtet.

Vorab lässt sich festhalten, dass manche der später vorgestellten Ansätze auf einer sehr systemnahen Ebene arbeiten und einen sehr hohen Grad an expliziter Parallelität aufweisen. Wiederum andere Ansätze sind dagegen auf einer etwas abstrakteren Ebene positioniert und greifen ihrerseits auf die systemnahen Ansätze zurück, um dem Programmierer komfortablere Möglichkeiten, wie z.B. Datenparallelität durch parallele Schleifenkonstrukte, bieten zu können. Aus diesem Grund könnte argumentiert werden, dass mit jedem Ansatz, durch mehr oder weniger Aufwand, eine bestimmte Ebene der Parallelität erreicht werden kann. Bei der Betrachtung und auch bei der Bewertung wird daher einem Ansatz nur dann Unterstützung für eine bestimmte Ebene der Parallelität zugeschrieben, wenn dieser über die dafür vorgesehenen Methoden verfügt. Beispielsweise verfügen Pthreads mit dem bereitgestellten Spektrum an Funktionen nicht über die Möglichkeit der Datenparallelität in Form von parallelen Schleifen. Zwar könnten Schleifen mit Hilfe der von Pthreads zur Verfügung gestellten Threads manuell parallelisiert werden, jedoch wird diese Vorgehensweise, zu Gunsten einer besseren Vergleichbarkeit bzgl. des Funktionsumfanges, hier nicht berücksichtigt.

## 3.3 Verständlichkeit

Wie einfach ein Programmiermodell zu erlernen und zu benutzen ist entscheidet letztendlich darüber, ob es von der breiten Masse an Entwicklern akzeptiert wird. [Ski97] Um das zu erreichen, müssen neuere Modelle von der reinen Hardware- und Anwendungssicht zu einer benutzerorientierten Methodik übergehen. [Asa06] Gerade durch den Wandel, bei dem Multi-Core-Prozessoren Einzug in den Massenmarkt halten, sind einfach zu benutzende Modelle

von sehr großer Wichtigkeit. Sie sollen Entwicklern, die noch über keine oder nur geringe Erfahrung im Bereich der parallelen Programmierung verfügen, dabei helfen, den steigenden Grad an Verarbeitungsparallelität aktueller Prozessoren ausnutzen zu können.

Schwierig gestaltet sich jedoch die Bewertung aufgrund der doch eher subjektiven Charakteristik dieser Größe. Des Weiteren gibt es im Bereich der parallelen Programmierung nur wenig Literatur, die diese Aspekte umfassend behandelt. [Hoc05] Feldstudien im Bereich der *Usability* sind beispielsweise in den Werken von [Hoc05] und [Sza96] zu finden. Aus diesem Grund werden, neben einer subjektiven Einschätzung der Verständlichkeit eines parallelen Programmiermodells durch den Autor, die Ergebnisse von [Shn78] und [Sza96] für spezifischere Messgrößen herangezogen. Als wichtige Kenngrößen werden in den Dokumenten u.a.: Lernbarkeit, integrierte Maßnahmen zur Fehlervermeidung und Hilfsmittel für das Debugging, Profiling und Testen von Anwendungen genannt. Während die Lernbarkeit beispielsweise gut anhand einer verfügbaren Spezifikation und Dokumentation beschrieben werden kann, hängen die Fähigkeiten für das Debugging oder Testen auch von den verwendeten Entwicklungswerkzeugen ab. Die Werkzeuge können dabei unabhängig gestaltet sein und sich für viele Programmieransätze eignen, wie z.B. Eclipse, Sun Studio oder der GNU Debugger. Aus diesem Grund werden die Aspekte der Lernbarkeit vorranging untersucht und Fähigkeiten für das Debugging, Profiling und Testen nur dann gesondert herausgestellt, wenn diese ansatzspezifisch sind. Zur Beurteilung der Lernbarkeit wird geprüft, in welchem Umfang Dokumentationen für den zu untersuchenden Ansatz zur Verfügung stehen. Eine detaillierte Überprüfung der Qualität kann hier aus zeitlichen Gründen nicht erfolgen; deshalb wird eine generelle Verfügbarkeit der folgenden Punkte geprüft:

- Spezifikation / API-Beschreibung
- Literatur (gedruckt / online)

## 3.4 Portabilität

Im Verlauf dieser Arbeit soll eine weitere Anforderung an parallele Programmiermodelle wie die Portabilität der betrachteten Ansätze untersucht werden. Diese lässt sich dahingehend unterscheiden, ob Aspekte der Hardware (z.B. die Architektur) oder die der Software eines Systems (z.B. das Betriebssystem) betrachtet werden sollen.

Parallele Programmiermodelle sollten weitestgehend unabhängig von der Hardware sein, damit ein mit dem Modell erstelltes Programm bei der Ausführung nicht nur auf eine bestimmte Computer-Architektur beschränkt ist. [Ski97] Bei der Entwicklung von sehr großen Computerclustern sind die Hersteller davon abgerückt, solche Systeme auf spezieller Hardware aufzubauen. Stattdessen wird auf „Standard-Hardware“ aufgrund der geringeren Kosten und des geringeren Programmieraufwandes zurückgegriffen. [Bau06] Der größte Teil der derzeit schnellsten Supercomputer basiert auf einer AMD x86_64 oder einer Intel 64 (EM64T) Prozessor-Architektur [Top10], die auch in Systemen für Endverbraucher zum Einsatz kommt. Abbildung 3-1 stellt die prozentuale Verteilung der genutzten Prozessorarchitekturen für die 500 schnellsten Supercomputer dar[10].

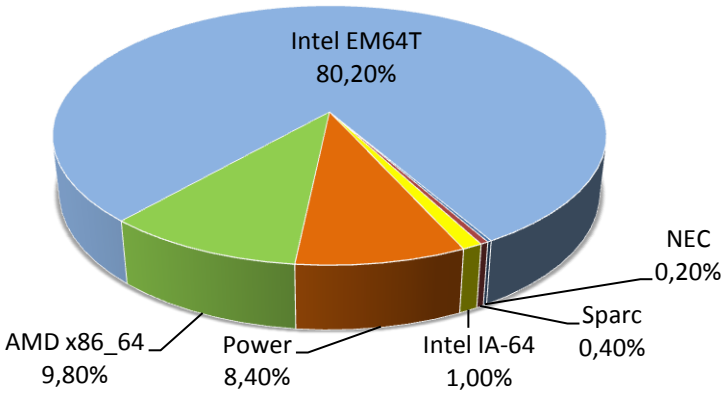



**Abbildung 3-1: Prozentuale Verteilung der Prozessor-Architektur in den Top 500 [Top10]**

Aufgrund dieses Trends und der Ausrichtung der Arbeit auf Systeme mit einem gemeinsamen Adressraum, wird auf Aspekte der Architektur nur bei den Ansätzen für programmierbare Grafikkarten genauer eingegangen.

[10] Stand: Juni 2010

Die Portabilität soll sich daher schwerpunktmäßig auf die Software-Aspekte beschränken und prüfen, für welche Betriebssysteme Anwendungen mit den Ansätzen entwickelt werden können. Als Referenz dienen die drei am häufigsten eingesetzten Betriebssysteme auf Basis von [Net10]:

- Windows[11]
- Mac OS X
- Linux

## 3.5 Leistung

Die Leistung von Anwendungen, die mit Hilfe paralleler Programmiermodelle parallelisiert wurden, galt lange Zeit als einziges Kriterium, um den Erfolg eines solchen Modells zu bewerten. Mittlerweile werden auch andere Aspekte betrachtet, die bereits in dieser Arbeit genannt wurden. [Hoc05] Dennoch nimmt die Leistungsfähigkeit weiterhin einen hohen Stellenwert ein [Rau07] und soll im Rahmen dieser Arbeit für einige ausgewählte Ansätze untersucht werden. Die Leistungsanalyse erfolgt dabei unabhängig von der Sichtung und wird im Kapitel 5 ‚*Evaluation paralleler Programmiermodelle*', gesondert durchgeführt.

Die Skalierbarkeit parallelisierter Anwendungen ist ein weiteres wichtiges Leistungskriterium aktueller Programmiermodelle. [Asa06] Eine gute Skalierbarkeit sorgt, ähnlich wie vor ein paar Jahren noch die Taktrate, für eine automatische Geschwindigkeitszunahme parallelisierter Anwendungen im Zuge neuer Prozessorgenerationen mit einer steigenden Anzahl an Kernen. Diese Eigenschaft kann längerfristig den Erfolg eines Programmiermodells bestimmen und wird bei den Leistungsuntersuchungen ebenfalls näher betrachtet.

## 3.6 Kosten für die Verwendung

Im Punkt Kosten wird geprüft, ob ein Programmiermodell kostenfrei verwendet werden kann, oder ob für die Nutzung bestimmte Lizenzen erforderlich sind, aus denen zusätzliche Kosten entstehen. Dieser Punkt wird nur der Vollständigkeit halber aufgeführt und nicht signifikant in die Bewertung mit einfließen, da wirtschaftliche Aspekte in dieser Arbeit nicht im Vordergrund stehen.

[11] Beinhaltet Windows-basierte Betriebssysteme ab Windows XP.

# 4 Sichtung paralleler Programmiermodelle

In dem folgenden Kapitel werden die zurzeit existierenden und einsetzbaren parallelen Programmieransätze zusammengestellt und beschrieben. Jede Darstellung enthält eine kurze Einführung über die Entstehung und die wesentlichen Merkmale des Ansatzes. Darüber hinaus wird in kompakter Form beschrieben, welche Funktionen und Methoden dem Programmierer zur Parallelisierung zur Verfügung stehen und wie sie verwendet werden können.

Innerhalb der Ausführungen wird auf die im vorherigen Kapitel definierten Eigenschaften, Kriterien und Anforderungen Bezug genommen. Aus den Beschreibungen wird hervorgehen, wie die Ansätze zu klassifizieren sind und in welchem Umfang die Anforderungen durch die Ansätze abgedeckt bzw. nicht abgedeckt werden. Eine zusammenfassende Darstellung der Eigenschaften und Kriterien erfolgt bei der Bewertung nochmals in tabellarischer Form.

## 4.1 Ansätze für Systeme mit gemeinsamem Speicher

### 4.1.1 Pthreads

Pthreads, oder POSIX[12] Threads, ist ein standardisiertes Programmierinterface zur Threadprogrammierung für die Programmiersprache C, definiert vom IEEE[13] im Standard 1003.1c-1995. [Nic96] Bevor mit Pthreads ein einheitlicher Standard zur Threadprogrammierung eingeführt wurde, gab es für jedes Betriebssystem eigene Konzepte zur Realisierung von Threads. [Gra03] Dies erhöhte einerseits den Aufwand für die Programmierer, da sie sich mit mehreren Threadbibliotheken befassen mussten, und andererseits erschwerte es die Portierbarkeit von threadbasierten Anwendungen. Primär wurde Pthreads für UNIX basierte Systeme entworfen [Ben08] und steht somit auch unter Mac OS X zur Verfügung. Unter Linux hat man die ursprüngliche Thread-API *Linux Threads* zugunsten von Pthreads verworfen und auch für Windows existieren, wenn auch zum Teil mit eingeschränktem Funktionsumfang, Pthreads-Implementierungen. [Bin09,Joh10]

[12] Portable Operating System Interface
[13] Institute of Electrical and Electronics Engineers

Um die Pthreads-Bibliothek mit ihren Funktionen, Prozeduren usw. verwenden zu können, muss die Header-Datei `pthread.h` und beim Kompilieren bzw. Linken die Pthreads-Bibliothek eingebunden werden. Wie die Bibliothek eingebunden wird, hängt vom verwendeten Compiler ab. Beim GNU C Compiler erfolgt dies durch die Angabe von `-pthreads`. [Bar10]

Eine kurze Beschreibung der Funktionsweise und die Möglichkeiten von Pthreads erfolgt anhand von drei wesentlichen Merkmalen threadbasierter Systeme gemäß [But02]:

- Ausführungskontext
- Scheduling
- Synchronisierung

### *Ausführungskontext*

Die Grundfunktion für Threadbibliotheken umfasst die Erstellung, Verwaltung und auch Löschung von Ausführungskontexten (Threads). In Pthreads wird ein neuer Thread mit der Funktion

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start)(void *), void *arg);
```

erzeugt. [But02] Der Zeiger vom Typ `pthread_t` dient als Identifikator, mit dem ein Pthread im Programm eindeutig identifiziert werden kann. Dieser wird z.B. dann benötigt, wenn auf die Beendigung eines Threads gewartet werden soll. Mit Hilfe des Zeigers auf die Struktur `pthread_attr_t` lassen sich Optionen für den Thread angeben, die sich z.B. auf das Scheduling beziehen. Mit dem dritten Argument wird auf die auszuführende Funktion verwiesen, die nur ein einziges Objekt vom Typ `void*` aufnehmen und ein solches zurückliefern kann. Dieses Objekt wird im vierten Parameter übergeben. [Rau07] Wie bei allen Pthreads-Funktionen liefert `pthread_create` eine 0 zurück, falls die Operation erfolgreich abgeschlossen werden konnte und -1 im Falle eines Fehlers. [But02]

Mittels `join` kann auf die Beendigung eines Threads durch Angabe seines Identifikators und eines Speicherbereichs für seinen Rückgabewert blockierend gewartet werden. [But02]

```
int pthread_join(pthread_t thread, void **result_ptr)
```

### *Scheduling*

Auf das Scheduling kann, wie bereits im Ausführungskontext angesprochen, bei der Threaderstellung Einfluss genommen werden. Ebenfalls ist es möglich, dass Scheduling von aktiven Threads zu verändern. Die Steuerung erfolgt über die im Folgenden aufgelisteten Parameter [Nic96]:

- Scheduling Priority
- Scheduling Policy

Mit Hilfe der *Scheduling Priority* können Prioritäten für Threads festgelegt werden, die bestimmen, welche Threads die CPU vorrangig nutzen dürfen. Die *Scheduling Policy* legt die Art und Weise fest, wie die CPU Threads gleicher Priorität zugewiesen wird. Hierbei kann der Programmierer zwischen drei verschiedenen Möglichkeiten wählen [Nic96]:

- FIFO (`SCHED_FIFO`)
- Round Robin (`SCHED_RR`)
- `SCHED_OTHER`

Beim *First-In-First-Out*-Scheduling erhält ein Thread solange die CPU, bis der Thread fertig ist, blockiert oder ein Thread mit einer höheren Priorität hinzustößt. *Round Robin* gewährt jedem gleichpriorisierten Thread eine gewisse Zeit Zugriff auf die CPU. Was bei `SCHED_OTHER` passiert, ist nicht immer eindeutig und abhängig von der jeweiligen Pthreads-Implementierung. So kann sich `SCHED_OTHER` wie FIFO, Round Robin oder wie eine andere Strategie verhalten. [But02]

### *Synchronisierung*

Synchronisation muss immer dann erfolgen, wenn mehrere Threads auf gemeinsamen Daten arbeiten und mindestens ein Thread schreibend zugreift. Die Bereiche im Code, in denen

dies geschieht, werden *kritische Bereiche* genannt. Zur Manipulation von Daten in kritischen Bereichen stellt Pthreads zwei verschiedene Mechanismen bereit [Zah06]:

- Mutexe
- Bedingungsvariablen *(condition variables)*

Eine Mutex-Variable in Pthreads ist vom Typ `pthread_mutex_t` und kann mit den Funktionen

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

manipuliert werden. Mit `lock` bzw. `unlock` erhält man exklusiven Zugriff auf die Mutex-Variable bzw. gibt diese frei. `trylock` blockiert im Gegensatz zu `lock` nicht, sollte ein anderer Thread bereits exklusiven Zugriff auf die Variable haben und weist anhand des Rückgabewertes auf eine Blockierung hin. [But02]

Bedingungsvariablen sind vom Typ `pthread_cond_t` und können nur in Kombination mit einer Mutex-Variablen verwendet werden. Bedingungsvariablen haben die Aufgabe, Threads unter bestimmten Bedingungen zu benachrichtigen, damit sie ihre Arbeit anschließend fortsetzen können[14]. Der Vorteil ist, dass so unnötiges Polling, oder auch aktives Warten genannt, auf Variablen seitens der Threads vermieden wird. Wartet ein Thread auf das Eintreten einer Bedingung, verbraucht dieser für diese Zeit keine CPU-Ressourcen. Im Gegensatz dazu müssen die Threads beim Polling den Zustand, z.B. mittels einer Endlosschleife, eigenständig prüfen. [Rau07]

Die Verwendung einer Bedingungsvariablen sieht im Allgemeinen wie folgt aus [Rau07]:

```
pthread_mutex_lock(&mutex);
while(!Bedingung)
    pthread_cond_wait(&cond_var, &mutex);
pthread_mutex_unlock(&mutex);
```

[14] Z.B. wenn eine Variable einen bestimmten Wert erreicht hat.

Andere Threads können wartende Threads mit den Funktionen

```
int pthread_cond_signal(pthread_cond_t *cond_var)
int pthread_cond_broadcast(pthread_cond_t *cond_var)
```

aufwecken. Die erste Funktion weckt nur einen wartenden Thread auf, wohingegen die zweite Funktion alle wartenden Threads aufweckt. [Nic96]

#### *Zusammenfassung*

Durch die reine Beschränkung auf Threads und Methoden zu deren Verwaltung, Synchronisation und Scheduling, steht nur die Ebene der Taskparallelität zur Verfügung. Weitere Ebenen müssen eigenhändig durch den Programmierer geschaffen werden. Dadurch befindet sich Pthreads auf einem sehr niedrigen Abstraktionslevel, bei dem, bis auf die Abbildung auf Systemressourcen, alles in expliziter Form vom Programmierer vorgegeben werden muss. Dies ist auch ein Grund, warum Pthreads in der Anwendungsentwicklung nur wenig Verwendung findet und Programmierer stattdessen bevorzugt auf abstraktere Ansätze, wie z.B. OpenMP, zurückgreifen. [Gra03]

Eine Einarbeitung in Pthreads ist aufgrund einer großen Auswahl an Literatur, z.B. [Bar10], [But02], [Nic96], [UAZ98], [Zah06], um nur einige zu nennen, relativ schnell möglich. Beim Debugging ist man auf die Fähigkeiten verfügbarer Debugger, wie z.B. GDB[15], angewiesen, da im Pthreads-Standard keine Möglichkeiten für das Debugging spezifiziert wurden. GDB ermöglicht u.a. eine Auflistung aller aktiven Threads der Anwendung sowie das Setzen thread-spezifischer Breakpoints[16]. [Sta10] Für die Durchführung von Tests steht die Open POSIX Test Suite[17] zur Verfügung.

### 4.1.2 Java Threads / Concurrency Utilities

Threads sind bereits ein fester Bestandteil der Sprache Java und müssen nicht über zusätzliche Bibliotheken eingebunden werden. Dadurch konnte sich die Threadfunktionalität tief in der Sprache verankern lassen, wie später bei den Synchronisationsmethoden noch deutlich

[15] http://sourceware.org/gdb/
[16] Breakpoints bezeichnen vom Programmierer gesetzte Punkte in einem Programm, die beim Erreichen für eine Unterbrechung der Ausführung sorgen.
[17] http://posixtest.sourceforge.net/

wird. Mit der Version 5 von Java kamen durch die *Concurrency Utilities* weitere Klassen hinzu, welche die Entwicklung von parallelen Anwendungen weiter vereinfachen sollten. [Oec07] Aufgrund des Konzeptes der virtuellen Maschine sind Java-Anwendungen auf jedem System lauffähig, für das eine solche virtuelle Maschine bereitsteht. Aktuell stellt Oracle die Java Virtual Machine (JVM) für Windows, Linux und Solaris zur Verfügung. Versionen für Mac OS X werden von Apple eigenständig verwaltet. [Ora10]

Dieser Abschnitt soll als eine kurze Einführung in die zentralen Punkte der Threadentwicklung unter Java dienen und orientiert sich dabei ebenfalls an den von [But02] definierten Merkmalen threadbasierter Systeme.

### *Ausführungskontext*

Um in Java einen Thread zu erstellen, gibt es zwei Möglichkeiten. Die beiden Varianten haben gemeinsam, dass sich der auszuführende Code des Threads innerhalb der Methode `run()` befinden muss. Der Unterschied liegt in der Klasse, welche die Methode beinhaltet. [Oec07]

- Ableiten von `Thread`

```
// Thread-Klasse, abgeleitet von Thread
public class MyThread extends Thread {
    public void run() {
          // Anweisungen für den Thread
    }
}
public static void main(String[] args) {
    // Thread-Objekt initialisieren und starten
    MyThread thread = new MyThread();
    thread.start();
}
```

Die Methode `start()` ist in der Klasse `Thread` vorhanden und steht durch Vererbung ebenfalls in der eigenen Threadklasse `MyThread` zur Verfügung. Durch den Aufruf wird ein neuer Thread gestartet, der intern die Methode `run()` aufruft.

- Implementierung des `Runnable`-Interfaces

Neben Vererbung kann auch eine Schnittstelle implementiert werden, wie das folgende Beispiel verdeutlichen soll:

```
// Thread-Klasse, implementiert Runnable
public class MyRunnable implements Runnable {
    public void run() {
          // Anweisungen für den Thread
    }
}
public static void main(String[] args) {
    MyRunnable run = new MyRunnable();
    Thread thread = new Thread(run);
    thread.start();
}
```

Der wesentliche Vorteil bei der Implementierung eines Interfaces ggü. der Ableitung einer Klasse ergibt sich durch die eingeschränkte Mehrfachvererbung in Java. Somit kann durch die Implementierung einer Schnittstelle die Klasse weiterhin von einer anderen abgeleitet werden. Wird bereits von `Thread` abgeleitet, hat man diese Möglichkeit nicht. [Oec07]

Eine Neuerung in Java 5 stellt die Implementierung der `Callable<T>`-Schnittstellte dar, in der die generische Funktion `call()` zu implementieren ist. Dadurch können Threads direkt ein Ergebnis in Form eines `Futures<T>`-Objekts zurückliefern. Eine Implementierung des `Futures` ist der `FutureTask<T>`, der eine von `Callable<T>` implementierte Klasse übergeben wird. Die Ausführung wird, wie bei einem normalen Thread, mit `start()` begonnen. Zusätzlich kann über die `get()`-Methode ein Ergebnis abgefragt werden. Liegt dieses zum Abfragezeitpunkt noch nicht vor, blockiert die Methode. Darüber hinaus stehen weitere Möglichkeiten zur Prüfung auf Fertigstellung oder zum Abbrechen ganzer Threads bereit. [Sun041]

```
public class MyCallable implements Callable<T> {
    public T call() {
          return DoWork(); // Muss Wert vom Typ T zurückliefern
    }
}
public static void main(String[] args) throws Exception {
```

```
    FutureTask<T> future = new FutureTask<T>(new MyCallable());
    // Callable ausführen und Ergebnis vom Typ T anzeigen
    future.run();
    ShowResult(future.get());
}
```

Eine weitere Neuerung in Java 5 ist die Möglichkeit, zur Ausführung von Aufgaben auf einen Thread-Pool zurückgreifen zu können. Der Vorteil bei einem Thread-Pool besteht in einem effizienteren Umgang mit bereits erzeugten Threads. Dadurch können Threads nach Beendigung ihrer Aufgabe dem Pool zurückgeführt und wiederverwendet werden. Zur Verwendung steht das `Executor`-Interface zur Verfügung, das z.B. vom `ThreadPoolExecutor` implementiert wird. Wie der `Executor` im Detail verwendet wird, kann bspw. den Ausführungen von [Goe06], [Oec07] oder[Ull09] entnommen werden.

### *Scheduling*

Auf die Scheduling Policy hat der Programmierer, anders als bei Pthreads, unter Java keinen Einfluss. Es können lediglich die Prioritäten für Threads verändert werden, die für die JVM allerdings auch nur Hinweise darstellen. Die JVM definiert insgesamt 10 Prioritätsstufen, die sie auf die jeweiligen Threadprioritäten des zugrundeliegenden Betriebssystems abbildet. Die Abbildung ist plattformspezifisch und hängt von der Anzahl der vom Betriebssystem unterstützten Prioritäten ab. [Goe06]

### *Synchronisierung*

Zur Synchronisierung steht in Java das Schlüsselwort `synchronized` zur Verfügung. Damit lassen sich ganze Methoden oder auch einzelne Blöcke innerhalb von Methoden kennzeichnen. Bei den gekennzeichneten Methoden bzw. Blöcken sorgt die JVM dafür, diese Codeabschnitte nicht von mehreren Threads gleichzeitig betreten werden können. Bei genauerer Betrachtung der Funktionsweise wird die zuvor angesprochene tiefe Integration des Threadkonzeptes in die Sprache deutlich. Zur Realisierung wird beim Aufruf einer *synchronized*-Methode die Sperre einer Klasse gesetzt. Dafür verfügt jedes Objekt über einen Monitor[18], der beim Aufruf einer gekennzeichneten Methode seinen *Lock* setzt und diesen beim Verlas-

[18] Der Begriff des Monitors geht auf C.A.R. Hoare zurück, der dieses Konzept 1978 erstmalig veröffentlichte.

sen wieder freigibt. Dadurch wird sichergestellt, dass sich immer nur ein Thread in einer solchen Methode befinden kann. Bei der Verwendung von synchronized-Blöcken, muss bei der Definition ein Synchronisationsobjekt angegeben werden. [Ull09] Dies schafft mehr Flexibilität, da so mehrere Threads in kritische Bereiche eintreten können, sofern sie über ein unterschiedliches Synchronisationsobjekt verfügen.

Zur atomaren Modifikation einzelner Variablen kann auf effizientere Methoden zurückgegriffen werden, die Java 5 in dem Namensraum `java.util.concurrent.atomic` bereitstellt. In diesem Paket werden für die Variablentypen Boolean, Integer, Long und Referenzen sowie für Felder von Integer, Long usw. spezielle Klassen bereitgestellt, mit denen atomar Werte verändert werden können. [Oec07] So bietet `AtomicInteger` u.a. die atomaren Methoden `compareAndSet(int expect, int update)` oder `addAndGet(int delta)` an. [Sun042]

Ebenfalls fester Bestandteil der Basisklasse `Object` sind die Methoden: `wait`, `notify` und `notifyAll`. Der Hintergrund bei der Verwendung dieser Methoden ist analog zum Konzept der Bedingungsvariablen von Pthreads. Mitunter müssen bestimmte Invarianten eingehalten werden, auf die man entweder aktiv warten kann oder sich beim Eintreffen benachrichtigen lässt. Ist eine Bedingung nicht erfüllt, kann der Thread mittels `wait` warten, bis ihn ein anderer irgendwann mit `notify` (oder alle mit `notifyAll`) beim Erreichen der Bedingung aufweckt. [Oec07] Das Problem bei der Verwendung dieser Methoden ist, dass alle Methoden das ihnen zugrundeliegende Objekt als Bezugspunkt nutzen. Dadurch kann nicht auf unterschiedliche Bedingungen gewartet werden. Diese Problematik hat man ab Java 5 durch die zusätzlichen Schnittstellen `Lock` und `Condition` gelöst. Ein Lock funktioniert dabei genau wie in Pthreads und stellt mit `lock`, `unlock` und `tryLock` ähnliche Methoden bereit. Eine Bedingungsvariable kann in Java ebenfalls nur in Verbindung mit einem Lock-Objekt verwendet und auch nur mit Hilfe eines Lock-Objektes erzeugt werden.

```
ReentrantLock lock = new ReentrantLock();
Condition cond = lock.newCondition();
```

Eine konkrete Implementierung eines Locks ist der `ReentrantLock`, der im Konstruktor optional durch eine bool'sche Variable angewiesen werden kann, eine faire Bedienreihenfol-

ge zu gewährleisten. Eine weitere Implementierung ist der `ReentrantReadWriteLock`, mit dem zwei Lock-Objekte für Lese- und Schreib-Operationen bereitstehen. Lesesperren können von mehreren Threads zur gleichen Zeit angefordert werden, eine Schreibsperre dagegen nur von einem Thread. [Oec07]

#### *Zusammenfassung*

Die Ebenen der Parallelität und der Abstraktionsgrad von Java entsprechen in etwa denen von Pthreads. In beiden Ansätzen müssen die Aufgaben manuell zerlegt, Threads erzeugt und verwaltet werden. Des Weiteren muss an entsprechenden Stellen Synchronisation erfolgen. Die Abbildung auf Systemressourcen übernimmt die JVM. Durch das Executor-Konzept kann die Verwaltung der Threads auch implizit durch einen Thread-Pool erfolgen und so zusätzlich die Wiederverwendung von Threads ermöglichen. Aufgrund der festen Integration in Java lassen sich die threadbezogenen Methoden leicht verwenden. Die gute API-Beschreibung von Oracle und eine umfassende Auswahl an Fachliteratur ermöglichen einen schnellen Einstieg. Weitere Informationen sind bspw. in den Büchern von [Lea99] und [Mag06] zu finden. Zusätzlich steht mit VisualVM[19] ein kostenloses Profiling-Tool für Java-Anwendungen zur Verfügung.

### 4.1.3 OpenMP

OpenMP entstand 1997 in Zusammenarbeit diverser Unternehmen als Spracherweiterung für C/C++ und FORTRAN und gilt bis heute als die erfolgreichste Spracherweiterung zur parallelen Programmierung von Systemen mit einem gemeinsamen Speicher. [Rei07] Portierbarkeit stand bei der Entwicklung im Vordergrund, so dass OpenMP sowohl auf Systemen mit UNIX-Basis als auch auf Windows-basierten Systemen eingesetzt werden kann. [OMP10] Da es sich um eine Compiler-bezogene Spracherweiterung handelt, muss der verwendete Compiler diese unterstützen. Unter anderem bieten die Compiler von Intel, Microsoft und der GNU Compiler OpenMP Unterstützung an[20]. Mit der Version 3.0 vom Mai 2008 liegt die zurzeit aktuellste Version der OpenMP-Spezifikation vor, auf die sich dieser Teil der Arbeit stützen wird. Codebeispiele werden für die Programmiersprache C/C++ angegeben.

[19] http://visualvm.dev.java.net
[20] Für eine vollständige Liste siehe [OMP101]

OpenMP-Anweisungen erfolgen im Quellcode durch Direktiven, die bei C/C++ durch das `#pragma`-Schlüsselwort eingeleitet werden. Dies hat den Vorteil, dass auch Compiler ohne OpenMP-Unterstützung das Programm interpretieren können und ein dennoch gültiges, sequentielles Programm produzieren. Diese Herangehensweise wird auch *inkrementelle Parallelisierung* genannt, bei der das sequentielle Programm schrittweise parallelisiert wird. [Cha01,Qui03]

Das grundlegende Konzept von OpenMP basiert auf einem fork/join-Modell, welches threadbasiert anstatt prozessbasiert arbeitet und in Abbildung 4-1 konzeptuell dargestellt ist. Mit der Direktive `#pragma omp parallel` wird eine parallele Region eingeleitet, die den Master-Thread von OpenMP anweist, eine geeignete Anzahl an Threads zu erstellen und die Anweisungen in der parallelen Region vom jedem Thread ausführen zu lassen. [Wil99]

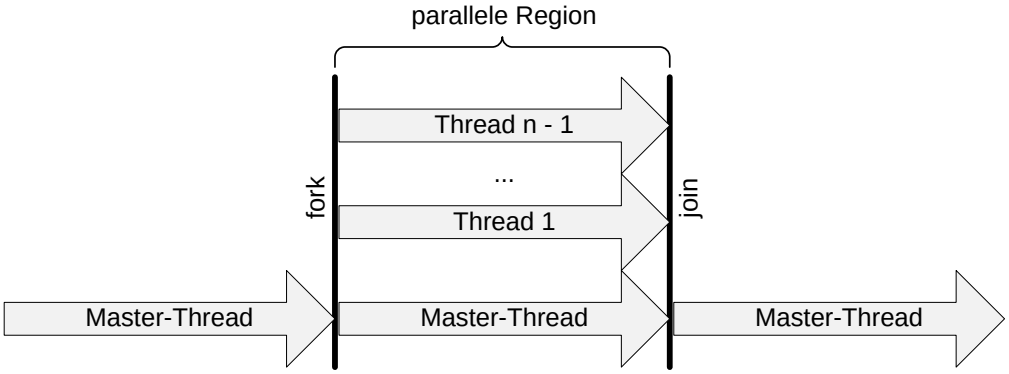



**Abbildung 4-1: fork/join-Konzept von OpenMP [Qui03]**

Die Ebenen der Parallelität umfassen sowohl die Task- als auch die Datenparallelität, anhand der die in OpenMP zur Verfügung stehenden Direktiven vorgestellt werden.

### *Datenparallelität*

Datenparallelität wird in OpenMP mit Hilfe von parallelen Schleifen geschaffen. Dazu stehen die Direktiven `#pragma omp parallel for` und `#pragma omp for` zur Verfügung. Beim ersten Aufruf wird automatisch eine parallele Region geöffnet, im zweiten Aufruf muss eine parallele Region bereits vorhanden sein. An die Direktive muss direkt eine For-Schleife anknüpfen, die anschließend von OpenMP parallelisiert wird.

Damit dies effizient geschehen kann, sind an For-Schleifen die folgenden Bedingungen gebunden [Cha01]:

- Der Bereich, über den iteriert werden soll, muss vorher feststehen und darf während der Ausführung nicht verändert werden.
- Die Iterationsschritte müssen gleich bleiben.
- Es dürfen keine Befehle verwendet werden, die ein vorzeitiges Abbrechen der Schleife zur Folge hätten (`break`, Auslösen von Exceptions).

Das folgende Beispiel soll die Verwendung verdeutlichen:

```
#pragma omp parallel for
    for(int i = 0; i < 1000; i++)
          // Code der parallelen Schleife
```

In OpenMP kann jeder Thread, der von einer parallelen Region erstellt wird, Daten über gemeinsame Variablen austauschen. OpenMP definiert dafür insgesamt drei verschiedene Gültigkeitsbereiche für Variablen [Cha01]:

- shared
- private
- reduction

Standardmäßig sind alle Variablen, die innerhalb eines parallelen Konstrukts genutzt werden und nicht weiter gekennzeichnet sind, *shared*. Jeder Thread kann die Inhalte der Variablen lesen und auch schreiben. Die Probleme, die beim Schreiben gemeinsamer Variablen entstehen, werden später aufgegriffen. Ausnahmen bilden Zählervariablen von parallelen Schleifen und lokale Variablen von Funktionsaufrufen, die automatisch *private* sind. Ebenfalls sind alle Variablen *private*, die ein Thread innerhalb der parallelen Region während der Ausführung erzeugt. [Cha01]

Soll manuell Einfluss auf den Gültigkeitsbereich genommen werden, so kann dies bei der Angabe der Direktiven erfolgen. Durch die optionalen Schlüsselwörter `shared()` und `private()`, denen eine Liste an Variablen übergeben wird, kann der Gültigkeitsbereich vom Programmierer vorgegeben werden.

Eine Sonderrolle nimmt der Typ *reduction* ein. Eine Reduktionsvariable ist sowohl *private* als auch *shared* und muss bei der Definition an einen Operator gebunden werden. Unter C/C++ stehen dafür die Operatoren: $+, *, -, \&, |, \hat{}, \&\&$ und || für Integer- und Float-Variablen zur Verfügung. Reduktionsoperationen ermöglichen die effiziente Berechnung eines gemeinsamen Ergebnisses durch viele Threads, ohne kritische Bereiche definieren zu müssen. Dafür arbeitet jeder Thread auf einer lokalen Kopie der gemeinsamen Variablen, die am Ende seiner Berechnungen auf die gemeinsame Variable gemäß der spezifizierten Operation reduziert wird. [Cha01]

```
    // Integer-Array a[n] definiert und gefüllt
    int sum;
#pragma omp parallel for reduction(+ : sum)
    for(int i = 0; i < n; i++)
          sum += a[i];
```

Würde man das obige Beispiel ohne Reduktionsvariable programmieren wollen, so müssen Synchronisationsmethoden verwendet werden, um den Schreibzugriff auf die gemeinsame Variable `sum` zu koordinieren. Dafür stellt OpenMP die folgenden Möglichkeiten bereit:

- Mutexe
- Kritische Bereiche
- Atomare Operationen

Das Konzept von Mutex-Variablen wurde bereits beim Pthreads-Ansatz beschrieben, so dass hier nur noch auf die zwei letzten Möglichkeiten eingegangen wird. Ein kritischer Bereich wird wie folgt definiert [OMP08]:

```
#pragma omp critical [(name)]
    {
          Anweisungsblock
    }
```

Innerhalb einer parallelen Region sorgt OpenMP dafür, dass immer nur ein Thread in einen kritischen Bereich eintreten kann. Die optionale Möglichkeit, einen Namen für einen kritischen Bereich anzugeben, hat den Vorteil, dass Threads in mehrere kritische Bereiche mit unterschiedlichem Namen eintreten können. [Cha01]

```
#pragma omp atomic
    Anweisung
```

`Atomic` unterscheidet sich insofern von `Critical`, dass der Direktive kein Block von Anweisungen folgen darf, sondern nur eine einzige Anweisung. `Atomic` greift zur Implementierung auf integrierte Mechanismen aktueller Prozessoren zurück, um einen Speicherbereich atomar manipulieren zu können. [Cha01] Durch die systemnahe Umsetzung sind zusätzliche Restriktionen an die Art der Anweisung gebunden. Hier sei an dieser Stelle auf die Spezifikation von OpenMP [OMP08] verwiesen.

Weitere Synchronisationsmethoden sind [Wil99]:

- `#pragma omp barrier` – Alle Threads müssen den Punkt erreichen, bevor die Ausführung fortgesetzt wird (implizit nach einer parallelen for-Schleife).
- `#pragma omp flush [(Variablen)]` – Sorgt für einen konsistenten Zustand gemeinsamer Variablen (Lese-/Schreiboperationen werden abgeschlossen).
- `#pragma omp ordered` – Sorgt innerhalb einer Schleife dafür, dass die Reihenfolge der Operationen dem sequentiellen Ablauf entspricht.

Als einen letzten Punkt bei der Datenparallelität soll noch kurz auf das Schleifen-Scheduling eingegangen werden, das der Programmierer mittels bestimmter Parameter ebenfalls beeinflussen kann. Bei der Ausführung einer parallelen Schleife weist OpenMP in der Standardeinstellung jedem Thread annähernd gleich viele Iterationen zu (statisches Scheduling). Ist jede Iteration vom benötigten Aufwand her ähnlich, ist dies aufgrund des geringen Overheads die beste Variante. Problematisch wird es jedoch, wenn der Aufwand je nach Iteration variiert (z.B. zunehmend oder abnehmend). Dies hätte zur Folge, dass gewisse Threads viel früher fertig werden als andere. Dafür bietet OpenMP dynamisches Scheduling an, bei dem keine feste Aufteilung bereits zu Beginn erfolgen muss. Dies wird dadurch realisiert, in dem den Threads zu Anfang nur eine bestimmte Anzahl an Iterationen zugeordnet wird und sie nach Abarbeitung ihrer zugeteilten Iterationen weitere anfordern müssen. Die Größe der Iterationen wird mit *chunk size* bezeichnet. Durch das Schlüsselwort `schedule(type[,chunk size])` kann die Art des Schedulings festgelegt werden. Insgesamt existieren folgende Möglichkeiten [Qui03]:

- `schedule(static[,chunk size])` – Statisches Scheduling; ohne Angabe der *chunk size* werden die Iterationen wie in der Standardeinstellung zugewiesen.
- `schedule(dynamic[,chunk size])` – Dynamisches Scheduling; keine Angabe der *chunk size* entspricht `schedule(dynamic,1)`.
- `scheduling(guided[,chunk])` – Beim Guided-Scheduling startet jeder Thread mit einer großen *chunk size*, die pro Abfrage neuer Iterationen schrittweise bis zu `chunk` verringert wird. Wird *chunk* nicht angegeben, entspricht dies `schedule(guided,1)`.
- `scheduling(runtime)` – Wendet das Scheduling-Verfahren an, welches global in der Umgebungsvariable `OMP_SCHEDULE` definiert wurde.

### *Taskparallelität*

Seit der Version 3.0 unterstützt OpenMP Taskparallelität in Form von Task-Direktiven. Zwar bestand auch schon in OpenMP Versionen < 3.0 die Möglichkeit, Aufgaben parallel zueinander ausführen zu lassen (z.B. mit `#pragma omp sections`), jedoch eignen sich solche Aufrufe nur unzureichend für rekursive Methoden. Der Grund dafür ist, dass bei rekursiven Aufrufen die *Sections* immer durch eine neue parallele Region erzeugt werden müssen. Dies benötigt einerseits zusätzlichen Overhead und andererseits läuft man Gefahr, mehr Ressourcen (Prozessoren) anzufordern, als dem System zur Verfügung stehen. [Ayg07]

Mit den in OpenMP 3.0 verfügbaren Tasks lässt sich dies mit der Direktive `#pragma omp task` vermeiden. Die Ausführung eines Tasks kann ein beliebiger Thread der parallelen Region übernehmen, die allerdings nur einmal beim ersten Aufruf durch eine Kombination von `#pragma omp parallel` und `#pragma omp single` definiert werden muss. Die Single-Anweisung sorgt dafür, dass nur ein (beliebiger) Thread die Ausführung der folgenden Anweisungen übernimmt. Soll explizit auf die Beendigung eines Tasks gewartet werden, steht die Direktive `#pragma omp taskwait` zur Verfügung. [OMP08]

Zur Erläuterung der Verwendung soll ein Ausschnitt von Quicksort dienen:

```
void quicksort (long n, data_t* a, long l, long r) {
    if(l < r) {
          long m = partition(n, a, l, r);
          // Linke Seite
#pragma omp task
          quicksort (n, a, l, m-1);
          // Rechte Seite (nicht als Task auslagern)
          quicksort (n, a, m+1, r);
#pragma omp taskwait // Auf Beendigung des erzeugten Tasks warten
}    }
int main(...) {
    [...]
#pragma omp parallel
    #pragma omp single
          quicksort(n, a, 0, n - 1);
}
```

### *Zusammenfassung*

Zusammenfassend betrachtet ist OpenMP ein abstraktes Programmiermodell, das nur eine explizite Angabe der Parallelität vom Programmierer sowie an bestimmten Stellen eine explizite Synchronisation erfordert. Die Abbildung auf Systemressourcen, die Zerlegung der Aufgaben (bei der Datenparallelität) und die Verwaltung der entstandenen Threads wird komplett von OpenMP übernommen. Die ausführliche Spezifikation sowie eine umfassende Literatur, auf die zum Teil referenziert wurde, tragen dazu bei, dass mit OpenMP schnell Resultate hervorgebracht werden können. Weitere Literaturquellen sind auf der offiziellen OpenMP-Webseite[21] angegeben. Zur Durchführung von detaillierten Performance-Analysen kann auf das kostenlose ompP[22] oder auf kostenpflichtige Werkzeuge von Intel[23] oder PGI[24] zurückgegriffen werden.

[21] http://openmp.org/wp/resources
[22] http://www.cs.utk.edu/~karl/ompp.html
[23] Intel VTune Performance Analyzer: http://software.intel.com/en-us/intel-vtune/
[24] PGPROF Graphical Performance Profiler: http://www.pgroup.com/products/pgprof.htm

### 4.1.4 Intel Thread Building Blocks

Intel Thread Building Blocks (TBB) wurde erstmals in der Version 1.0 im Jahr 2006 als eine kommerzielle Bibliothek für C++ veröffentlicht. Ein Jahr später folgte die Version 2.0, in der Intel ebenfalls den Quellcode zu TBB unter der GPLv2 Lizenz bereitstellte. Intel entschied sich zu diesem Schritt, um TBB als quelloffene Lösung zu einer größeren Verbreitung zu verhelfen, welche unter einer ausschließlich proprietären Lizenz nicht möglich gewesen wäre. [Rei10] Ab der Version 2.0 vertreibt Intel sowohl eine kommerzielle als auch eine kostenfreie Version. Dabei gibt es keinerlei Unterschiede in der Software, sondern nur in der Lizenzierung und beim Support. Aktuell liegt die gerade veröffentlichte Version 3.0 vor, auf die sich auch die Arbeit stützen wird. [Neu10]

Ein großer Vorteil der Bibliothek ist ihre Portabilität. Es existieren Implementierungen für Windows, Linux, Solaris und Mac OS X, so dass alle gängigen Plattformen abgedeckt sind. Darüber hinaus bietet TBB Unterstützung für Microsofts Spielekonsole XBOX 360. [Wer09]

Zur Verwendung der Bibliothek sind bestimmte Header-Dateien zu inkludieren sowie Verweise auf Namensräume anzugeben. Auf eine manuelle Initiierung des Task-Scheduler kann seit Version 2.2 verzichtet werden. [Int10]

```
#include "tbb/tbb.h"
using namespace tbb;

int main() {
    [...]
}
```

In TBB stehen die Ebenen der Datenparallelität, Taskparallelität und Pipelining zur Verfügung, anhand derer die grundlegende Funktionsweise von TBB dargestellt wird.

#### *Datenparallelität*

Analog zu OpenMP stehen in TBB parallele For-Schleifen (`parallel_for`) und Reduktions-Schleifen (`parallel_reduce`) zur Verfügung. `parallel_reduce` ist jedoch nicht auf bestimmte Datentypen beschränkt, sondern generisch einsetzbar. Zusätzlich können Do-

Schleifen (`parallel_do`) parallelisiert werden [Rei07], was in OpenMP aufgrund der vorher bestimmbaren Iterationsanzahl nicht möglich ist.

Bevor Lambda-Funktionen in C++ mit dem kommenden C++0x-Standard eingeführt wurden, musste der Programmierer etwas mehr Aufwand betreiben, um eine simple Schleife parallelisieren zu können. Dafür war es nötig, den Schleifenkörper in einer Klasse unter Verwendung eines STL[25]-Funktionsobjekts unterzubringen, die anschließend von der `parallel_for`-Methode genutzt wird. Durch Verwendung von Lambda-Funktionen verringert sich der Aufwand erheblich, da der zu parallelisierende Code direkt als Lambda-Ausdruck der parallelen Schleifenmethode übergeben werden kann. [Rei09] Der nachfolgende Abschnitt wird sich jedoch nicht mit den neuen Methoden in Verbindung mit Lambda-Ausdrücken beschäftigen, sondern auf die komplexere, aber auch flexiblere Variante eingehen. Die Verwendung von Lambda-Ausdrücken wird im Kapitel *4.1.5 - Visual C++ 2010 Concurrency Runtime* im Detail behandelt. Die dort beschriebenen Funktionen sind größtenteils kompatibel zu den neuen Funktionen in TBB 3.0

Als Beispiel soll die folgende Schleife dienen, die im weiteren Verlauf mittels TBB parallelisiert wird:

```
for(int i = 0; i < n; i++)
    DoWork(i);
```

Zu Anfang muss der Schleifenkörper in die `operator`-Methode des STL-Funktionsobjektes transformiert werden. Dies erfordert das Anlegen einer ganzen Klasse, die im Beispiel `STLDoWork` genannt wird:

```
class STLDoWork {
public:
    void operator() (const blocked_range<size_t>& r ) const {
          for(size_t i=r.begin(); i!=r.end(); ++i)
                DoWork(i);
    }
};
```

[25] Standard Template Library

Der `operator`-Methode wird ein `Range`-Objekt eines bestimmten Typs (im Beispiel: `size_t`) übergeben, das für die Beschreibung der einzelnen Iterationen verantwortlich ist. Innerhalb der TBB wird beim Aufruf einer parallelen Schleife für jeden einzelnen Thread ein solches Objekt erzeugt, mit dem jeder Thread über die Methoden `begin()` und `end()` den von ihm zu bearbeitenden Iterationsbereich abfragen kann. [Rei07] Um eine parallele For-Schleife letztendlich nutzen zu können ist neben der Klasse, die den Schleifenkörper kapselt, der Aufruf der `parallel_for`-Methode erforderlich:

```
parallel_for(blocked_range<size_t>(0, n [, chunk_size]), STLDoWork());
```

Zusätzlich zur Angabe der Schleifengrenzen und der zuvor definierten Klasse kann Einfluss auf die *chunk size* genommen werden. Die Bedeutung dieser optionalen Größe wurde bereits im Kapitel 4.1.3 bei OpenMP behandelt. Intel empfiehlt für diesen Wert mindestens 10.000 bis 100.000 anzugeben. [Rei07] Dies ist auch der wesentliche Vorteil ggü. der Verwendung von Lambda-Funktionen, die zwar den Aufruf erheblich vereinfachen aber dem Programmierer keine Kontrolle über die *chunk size* oder die Partitionierung ermöglichen. Darüber hinaus lassen sich die Funktionen `parallel_reduce` und `parallel_do` ebenfalls nur mit der komplexeren Variante verwenden. Für weitere Informationen sei an dieser Stelle auf die Literatur [Rei07] verwiesen.

### *Taskparallelität*

Taskparallelität wird in TBB über Task-Objekte erzielt, die durch den Task-Scheduler verwaltet und auf die Systemressourcen abgebildet werden. Um einen Task zu erzeugen, sind in TBB insgesamt drei Schritte sowie die Einbindung der Header-Datei `task.h` notwendig.

1. Task in einem Objekt kapseln, das von dem Objekt `task` abgeleitet wurde

```
class DoWorkTask : public task {
public:
    task* execute() {
          // Code des Tasks
          return NULL;
    }
};
```

In dem `task`-Objekt ist die Methode `execute()` definiert, die von dem eigenen Objekt überschrieben werden muss und die die eigentliche Arbeit enthält. Als Rückgabewert ist entweder NULL oder ein Pointer auf den anschließend auszuführenden Task zurückzugeben. [Rei07]

2. Task-Objekt initialisieren

```
DoWorkTask& tsk = *new(task::allocate_root()) DoWorkTask();
```

Bei der Initialisierung kann zwischen `allocate_root()` und `allocate_child()` differenziert werden. `root` bedeutet, dass der Task der erste in einem Taskgraphen ist und über keinen Parent-Task verfügt, wohingegen ein `child` einen Parent-Task besitzt. Dies ist für die TBB insofern von Bedeutung, weil sie die Tasks intern als Graphen darstellt. Der Taskgraph wird anschließend von der TBB genutzt, um ein möglichst effizientes Scheduling durchführen zu können. Hier wird zwischen zwei Arten der Abarbeitung des Graphen unterschieden, die je nach Art des Graphen zur Ausführung genutzt werden [Rei07]:

- Tiefensuche – geeignet für endliche Graphen, geringer Speicherverbrauch
- Breitensuche – erzeugt mehrere aktive Tasks zugleich; potentiell höhere Parallelität bei sehr vielen verfügbaren Threads

3. Task ausführen

```
task::spawn_root_and_wait(tsk);
```

Mittels `spawn` werden Tasks gestartet. Die obige Methode kombiniert das Starten und das Warten auf die Beendigung des Tasks. Wurden in einem Parent-Task mehrere Child-Tasks erzeugt und gestartet, kann mit `task::wait_for_all` auf die Beendigung aller Child-Tasks gewartet werden. [Rei07]

Seit Version 3.0 der TBB-Bibliothek ist auch das Erzeugen von Tasks mittels Task-Gruppen und Lambda-Ausdrücken möglich. Die Vorgehensweise ist identisch zur Visual C++ 2010 Concurrency Runtime, so dass an dieser Stelle auf das Kapitel 4.1.5 verwiesen wird.

### *Pipelining*

TBB realisiert Pipelining durch zwei Komponenten: Die Pipeline und eine Menge von Filtern. Filter stehen für die jeweiligen Stufen in einer Pipeline, die die Daten in bestimmter Weise verarbeiten bzw. manipulieren und sie zum nächsten Filter weiterreichen. Für jeden Filter ist eine abgeleitete Klasse von `filter` zu erstellen, in der die Methode:

```
void operator(void *item);
```

überschrieben werden muss. Per Definition muss die Methode einen Zeiger auf den nächsten anzuwendenden Filter zurückliefern oder NULL, falls es der letzte Filter in der Pipeline ist. Für den ersten Filter in der Pipeline ist das zu übergebene `item` ebenfalls NULL, da dieser für die Erzeugung der Daten zuständig ist. [Rei07]

```
// Pipeline erzeugen
tbb::pipeline pipe;
// Filter hinzufügen
pipe.add_filter(filter_1);
…
pipe.add_filter(filter_n);
// Pipeline starten
pipe.run(max_token);
// Filter entfernen
pipe.clear();
```

Neben der grundsätzlichen Parallelität durch die verschiedenen Filter kann zusätzliche Parallelität geschaffen werden, indem gewisse Filter mehrere `items` parallel verarbeiten können. Um den Grad der Parallelität zu kontrollieren, nimmt die `run()`-Methode einer Pipeline einen Parameter auf (`max_token` im vorherigen Beispiel), der bestimmt, wie viele Daten zu einem Zeitpunkt maximal durch die Pipeline laufen dürfen. [Rei07]

### *Zusammenfassung*

Aufgrund des sehr großen Funktionsumfangs von TBB konnte nicht auf jeden Aspekt der Bibliothek eingegangen werden, so dass es bei dieser kurzen Einführung bleiben soll. Eine detaillierte Beschreibung aller Funktionen und Strukturen ist in den Dokumentationen von Intel ([Int10], [Int101], [Int102]) und in [Rei07] zu finden. Trotz der guten Ressourcen, die Intel

online zu Verfügung stellt, ist der Umfang an gedruckter Literatur mit nur einem Buch gering[26]. Durch den teilweise höheren Aufwand zur Parallelisierung macht die TBB, trotz des gleichen Abstraktionsniveaus wie z.B. OpenMP, einen komplexeren Eindruck und ist mit einem höheren Programmieraufwand verbunden. Dieser konnte mit der Einführung von Lambda-Funktionen und der Version 3.0 zum Teil drastisch reduziert werden.

### 4.1.5 Visual C++ 2010 Concurrency Runtime

Die Concurrency Runtime (CRT) ist eine Erweiterung in Visual C++ 2010 von Microsoft im Bereich der parallelen Programmierung, die auf der Windows-API aufbaut. Sie bietet darüber hinaus eine Kombination aus einem präemptiven[27] und einem kooperativen[28] Task Scheduler, sowie einen *Work Stealing* Algorithmus als Erweiterung der bekannten FIFO Scheduling-Strategie zur Steuerung von Tasks. [Mic1015] Zur Ausführung von Programmen, die die Methoden der CRT nutzen, wird die *Visual C++ 2010 Runtime* auf dem jeweiligen Zielsystem benötigt. Microsoft stellt die Runtime nur für windowsbasierte Betriebssysteme ab Windows XP SP3 bereit, so dass auch nur windowsbasierte Anwendungen von diesen Erweiterungen profitieren können. Eine detaillierte Behandlung der CRT erfolgte bereits in [Hor10], so dass hier nur in verkürzter Form auf diese Technologien eingegangen wird.

Die folgende Darstellung der Funktionsweise orientiert sich an den vier Komponenten, aus denen die CRT aufgebaut ist.

#### *Parallel Pattern Library*

Die Parallel Pattern Library (PPL) stellt die Ebenen der Daten- und Taskparallelität mit Hilfe von parallelen Schleifenkonstrukten und Task-Objekten bereit. Um die PPL in einer Anwendung verwenden zu können, ist die Header-Datei `ppl.h` zu inkludieren sowie ein Verweis auf den Namensraum `Concurrency` zu tätigen. [Mic1016]

[26] Quelle: Amazon.de, Stand: 05.05.2010

[27] Ein präemptives Scheduling ermöglicht den Threads, nach einem bestimmten Schema (z.B. Round-Robin) und für eine beschränkte Zeitspanne, Zugriff auf die CPU

[28] Ein kooperatives Scheduling ermöglicht den Threads die CPU so lange zu belegen, bis dass sie ihre Aufgabe beendet haben oder die Ressourcen eigenständig freigeben

Bei den Schleifenkonstrukten steht, neben der parallelen For-Schleife, auch eine parallele Foreach-Schleife zur Verfügung. Beide Schleifenmethoden werden mit Hilfe von Lambda-Funktionen aufgerufen und stehen in gleicher Weise auch in Intels TBB 3.0 zur Verfügung.

Folgender Aufruf ist mit Hilfe der Lambda-Funktionen möglich [Mic10]:

```
parallel_for(Index First, Index Last [, Index Step],
    const _Function& Func);
```

Die Parameter `First` und `Last` legen die Schranke der Schleife fest. Mit `Step` kann optional Einfluss auf die Schritte der Iterationen genommen werden. `Func` ist eine Lambda-Funktion, welcher der zu parallelisierende Schleifenkörper übergeben wird. Das folgende Beispiel soll die Verwendung verdeutlichen:

```
parallel_for(0, n, [=](int i) {
    // Schleifenkörper – Zugriff per Kopie auf alle äußeren Variablen
});
```

Eine Lambda-Funktion wird durch die zwei Klammer-Ausdrücke `[ ]( )` eingeleitet, wobei in den eckigen Klammern angegeben wird, wie der Compiler die im Programm zugänglichen Variablen der Lambda-Funktion zur Verfügung stellen soll. `[&]` bedeutet die Übergabe als Referenz, wohingegen mit `[=]` der Compiler angewiesen wird, der Lambda-Funktion nur Kopien bereitzustellen und Änderungen somit keine Auswirkungen auf die ursprünglichen Variablen haben. Effizienter ist jedoch die Definition einer Liste, mit der nur die benötigten Variablen bereitgestellt werden können, wie z.B. `[&counter,=tmp]`. Nach der eckigen Klammer folgt in den runden Klammern die Definition der Variablen, die eine Lambda-Funktion beim Aufruf entgegennimmt. Im Falle einer parallelen For-Schleife entspricht dies der Iterationsvariablen. Anschließend folgt in geschweiften Klammern die eigentliche Funktion. [Rei091]

Für die parallele Foreach-Schleife gestaltet sich der Aufruf wie folgt [Mic101]:

```
parallel_for_each(Iterator First, Iterator Last,
    const _Function& Func);
```

Einfluss auf die Partitionierung einer Schleife kann mit der PPL nicht genommen werden, genauso wenig existieren spezielle Schleifenvarianten zur Reduktion. Jedoch steht mit der `combinable<T>`-Klasse eine generische Implementierung für Reduktionsoperationen zur Verfügung. Bei einer gemeinsamen Verwendung einer solchen Variablen arbeiten alle Threads auf lokalen Kopien, die sich am Ende durch eine benutzerdefinierte Lambda-Funktion auf einen Wert reduzieren lassen. [Ker09]

```
combinable<int> AddReduce;
// Jeder Thread reduiziert lokal mit AddReduce.local() += … ;
// Anschließend werden die lokalen Ergebnisse kombiniert
int sum = AddReduce.combine([](int l, int r){ return l + r; });
```

Die PPL realisiert Taskparallelität mit Hilfe von Task-Gruppen, die mehrere Tasks aufnehmen können. Dabei wird zwischen den folgenden Gruppen unterschieden [Mic1013]:

- Strukturierte Task-Gruppen (`structured_task_group`)
- Unstrukturierte Task-Gruppen (`task_group`)

Der Unterschied zwischen strukturierten und unstrukturierten Task-Gruppen besteht darin, dass Tasks in strukturierten Task-Gruppen komplett abgearbeitet werden müssen, bevor der aufrufende Kontext verlassen werden kann. Ein Parent-Task ist somit erst dann beendet, wenn alle durch ihn erzeugten Child-Tasks beendet sind. Bei unstrukturierten Task-Gruppen gibt es diese Bedingung nicht. Der Programmierer kann Tasks mit der Hilfsfunktion `make_task()` erzeugen, die einen Lambda-Ausdruck aufnimmt. Anschließend ist die Zuweisung zu einer Task-Gruppe erforderlich, unter der der Task ausgeführt wird. [Mic1013]

```
// Task erzeugen
auto tsk = make_task([]{ printf("Hallo Welt!\n"); });
// Task-Gruppe erzeugen
structured_task_group stg; // Alternativ: task_group für unstrukturier-
te Gruppen
// Task hinzufügen und starten
stg.run(tsk);
```

Muss im weiteren Verlauf kein Einfluss mehr auf die Tasks ausgeübt werden, lassen sich zwei bis zehn Tasks parallel mit der Methode `parallel_invoke()` starten. Die Methode ist blockierend und wartet auf die Beendigung aller gestarteten Tasks. [Mic102]

```
parallel_invoke(const _Function& Func1, const _Function& Func2, …)
```

### *Asynchronous Agents Library*

Die Asynchronous Agents Library (AAL) ermöglicht die Entwicklung von Anwendungen gemäß Datenfluss-Modellen *(dataflow models)*. Solche Modelle ermöglichen – im Gegensatz zu Kontrollflussmodellen, bei denen Operationen immer in einer festen Reihenfolge ausgeführt werden – die Bearbeitung der Daten, sobald diese bereitliegen. Die Kommunikation erfolgt explizit durch entsprechende Methoden für den Nachrichtenaustausch. Dadurch wird Pipelining mit Hilfe von grobförmiger Parallelität ermöglicht. Zur Nutzung der AAL ist die Header-Datei `agents.h` einzubinden. [Mic1014]

Die AAL setzt sich insgesamt aus drei Komponenten zusammen [Mic1014]:

- Asynchrone Agenten

Mit Hilfe von Agenten wird grobe Parallelität geschaffen, indem komplexe Aufgaben in einzelne Teilaufgaben zerlegt und den verschiedenen Agenten zugewiesen werden. Ein Agent ist vergleichbar mit einem Task, der eine bestimmte Aufgabe ausführt. Im Gegensatz zu einem Task kann der Agent seine Ergebnisse mit anderen Agenten austauschen und abhängig davon weitere Aufgaben durchführen. [Mic1017]

- Asynchrone Nachrichtenblöcke

Die Kommunikation zwischen Agenten findet über asynchrone Nachrichteblöcke statt. Dies sind bestimmte Strukturen, die threadsicher sind und so von mehreren Tasks (Agenten) gleichzeitig manipuliert werden können. Ein Beispiel für solch einen Nachrichtenblock ist der `unbounded_buffer`, der Daten von mehreren Agenten aufnehmen und sie mehreren Agenten zur Verfügung stellen kann. [Mic1018]

- Funktionen für den Nachrichtenaustausch

Die Nachrichtenstrukturen werden von den Agenten jedoch nicht unmittelbar benutzt, sondern indirekt durch spezielle Funktionen. Mit den Methoden `send` und `receive` kann unter Angabe eines Nachrichtenblocks synchron in diesen geschrieben bzw. aus ihm gelesen werden. Nicht-blockierende Methoden stehen mit `asend` und `try_receive` zur Verfügung. [Mic1019]

### *Task Scheduler*

Mit Hilfe eines benutzerdefinierten Task Schedulers lässt sich das Scheduling der CRT beeinflussen. Grundsätzlich benutzt die CRT eine Mischung aus präemptiven und kooperierenden Scheduling, um die Systemressourcen möglichst effizient ausnutzen zu können. [Mic1015] Wird ein eigener Task Scheduler durch den Programmierer definiert, so kann er die folgenden Eigenschaften festlegen [Mic1020]:

- Minimale bzw. maximale Anzahl an Threads
- Gezielte Überbelegung (mehr Threads als Prozessoren)
- Priorität der Threads
- Nutzung von Win32-Threads oder UMSThreads[29]
- Tasks innerhalb einer Gruppe bevorzugen oder gruppenübergreifend auswählen

### *Resource Manager*

Der Resource Manager (RM) hat die Aufgabe, die Tasks der verschiedenen Task Scheduler auf die vorhandenen Systemressourcen unter Beachtung der definierten Prioritäten und Eigenschaften abzubilden[30]. Auf den RM hat der Programmierer im Gegensatz zum Task Scheduler jedoch nur wenig Einfluss. Deshalb wird er an dieser Stelle nicht näher behandelt. [Gun09]

[29] User Mode Scheduler Threads: Verfügbar in Windows 7; ermöglichen der Concurrency Runtime die vollständige Verwaltung der Threads, ohne den Kernel damit beschäftigen zu müssen.
[30] Es können mehrere Task Scheduler mit unterschiedlichen Strategien definiert werden.

***Zusammenfassung***

Die Concurrency Runtime ist vom grundlegenden Funktionsumfang vergleichbar mit Intels Thread Building Blocks. Sie bietet Unterstützung für die Daten- und Taskparallelität sowie Pipelining. Zur Parallelisierung ist eine explizite Angabe erforderlich. Eine Zerlegung erfolgt bei der Datenparallelität implizit, bei der Taskparallelität und beim Pipelining explizit. Um eine Zuordnung und Verwaltung der dadurch entstehenden Threads braucht sich der Programmierer nicht zu kümmern. Synchronisation muss an erforderlichen Stellen explizit erfolgen. In kritischen Bereichen lässt sie sich durch die Nutzung threadsicherer Datenstrukturen, wie z.B. `concurrent_queue`, `concurrent_vector`, `combinable` oder atomaren Operationen, vermeiden.

Der Umfang an Literatur ist aufgrund der Neuheit der Bibliothek noch beschränkt. Jedoch steht mit dem Microsoft Developer Network (MSDN) ein umfangreiches Informationsportal zur Verfügung. Zur Entwicklung kann einerseits auf das kostenpflichtige Visual Studio 2010 oder auf die kostenlose Express Variante[31] zurückgegriffen werden, die jedoch gewissen Einschränkungen innerhalb der Entwicklungsumgebung unterliegt. Wird die kostenpflichtige Variante eingesetzt, stehen dem Entwickler komfortable Funktionalitäten im Bereich des Testens, Debuggens und Profiling zur Verfügung.

### 4.1.6 .NET 4.0 Parallel Extensions

Mit den Parallel Extensions erweiterte Microsoft, analog zur Concurrency Runtime, .NET-basierte Sprachen ebenfalls um verbesserte Parallelisierungsmöglichkeiten. Diese Erweiterungen halten im .NET Framework 4.0 Einzug und stehen somit für die Sprachen: Visual C#, Visual F# und Visual Basic.NET bereit. Zur Ausführung von .NET-basierten Anwendungen ist das .NET Framework erforderlich, das Microsoft ebenfalls nur für Windows-Systeme ab Windows XP SP3 zur Verfügung stellt. Mit dem Mono-Projekt[32] existiert eine quelloffene und plattformübergreifende Lösung, basierend auf den ECMA-Standards für C# und der *Common Language Infrastructure* (CLI), die eine Ausführung von .NET-Anwendungen auch unter Linux und Mac OS X ermöglicht. [Mon10] Die derzeit aktuellste Version 2.6 unterstützt das komplette Funktionsspektrum von .NET 3.5, bis auf die Windows Presentation Foundation (WPF)

[31] Download unter: http://www.microsoft.com/germany/express/products/windows.aspx
[32] http://www.mono-project.com

und die Windows Workflow Foundation (WF). An einer Unterstützung für .NET 4.0 wird gearbeitet. [Mon101]

Wie bereits bei der Concurrency Runtime erfolgte eine detaillierte Behandlung der Parallel Extensions in [Hor10], weshalb hier nur in verkürzter Form auf die Erweiterungen eingegangen wird. Die Beschreibung orientiert sich an den zwei Bestandteilen der Parallel Extensions: Die Task Parallel Library und PLINQ.

### *Task Parallel Library*

Durch die Task Parallel Library (TPL) werden Daten- und Taskparallelität ermöglicht. Der Aufbau und die Funktionsweise sind sehr ähnlich zur Parallel Pattern Library der Concurrency Runtime und erlauben ebenfalls die Nutzung von parallelen Schleifen sowie Tasks. [Mic105,Mic106] Damit die Methoden und Objekte der TPL in einer .NET-Anwendung genutzt werden können, sind Verweise auf die Namensräume `System.Threading` und `System.Threading.Tasks` anzugeben. [Mic104]

Die beiden Schleifenkonstrukte `Parallel.For` sowie `Parallel.ForEach` sind mehrfach überladen. Der einfachste Konstruktor der beiden Methoden sieht wie folgt aus [Mic107,Mic108]:

```
public static ParallelLoopResult For(
    int fromInclusive, int toExclusive, Action<int> body)
public static ParallelLoopResult ForEach<T>(
    IEnumerable<T> source, Action<T> body)
```

Beide Methoden erwarten als `body` ein `Action`-Objekt, das in .NET ebenfalls durch einen Lambda-Ausdruck dargestellt wird. Der Lambda-Ausdruck muss bei der For-Schleife eine Integer-Zählvariable aufnehmen können, bei der Foreach-Schleife das Objekt der aktuellen Iteration vom Typ `T`. Das folgende Beispiel soll die Verwendung verdeutlichen:

```
using System.Threading;
using System.Threading.Tasks;
[…]
// Liste erstellen, über die iteriert werden kann
List<int> myList = Enumerable.Range(0, MAX_COUNT).ToList<int>();
// For-Schleife – Zugriff auf Element mit Zählervariable
```

```
Parallel.For(0, MAX_COUNT, (int i) => {
    DoWork(myList[i]);
});
// Foreach-Schleife – Zugriff über aktuelles Iterationsobjekt
Parallel.ForEach<int>(myList, (int current_item) => {
    DoWork(current_item);
});
```

In weiteren Aufrufvarianten kann Einfluss auf die Partitionierung (nur bei Foreach), den Task Scheduler sowie die Abbruchsteuerung genommen werden. Außerdem existieren erweiterte Varianten der parallelen For- und Foreach-Schleife zur Durchführung von Reduktionsoperationen. [Mic105]

Mit den seit .NET 4.0 hinzugekommenen Tasks lässt sich Taskparallelität nun auf einer einfacheren und abstrakteren Weise erzeugen, die einerseits die Systemressourcen besser ausnutzen und andererseits dem Programmierer eine erhöhte Kontrolle bieten sollen. [Mic106] Vor .NET 4.0 existierten dazu lediglich Threads. In der einfachsten Variante verlangt der Konstruktor eines neuen Task-Objektes nur ein Argument in Form eines Lambda-Ausdrucks [Mic103]:

```
public Task(Action action)
```

Ebenfalls gibt es spezielle Task-Objekte, die ein Ergebnis zurückliefern können und den `Futures` aus Java ähnlich sind [Mic109]:

```
public Task(Func<TResult> function)
```

`TResult` legt den Typ des zurückzugebenen Ergebnisses fest, auf das durch die `Result`-Eigenschaft des Task-Objekts zugegriffen werden kann. Wird `Result` vor Fertigstellung des Tasks aufgerufen, blockiert die Methode bis das Ergebnis vorliegt. [Mic1010]

```
// Task erzeugen, der Ergebnis vom Typ string zurückliefert
Task<string> tsk = new Task<string>(() => {return "Hallo Welt!";});
tsk.Start();
// Ergebnis auslesen und anzeigen
Console.WriteLine(tsk.Result);
```

Neben einem Action-Objekt kann ein Task weitere Argumente bei der Erzeugung aufnehmen, mit denen sich das Scheduling beeinflussen oder eine Abbruchsteuerung definieren lässt. Außerdem ist es möglich, mehrere Tasks über Bedingungen miteinander zu verknüpfen. So können unterschiedliche Tasks in Abhängigkeit davon ausgeführt werden, ob der vorangegangene Task erfolgreich beendet wurde oder nicht. Produzieren Tasks Ergebnisse, lassen sich diese ebenfalls an anknüpfende Tasks weiterreichen. [Mic106]

Analog zur Concurrency Runtime gib es in der PPL die `Parallel.Invoke(Action[] actions)`-Methode, die jedoch nicht auf maximal zehn Tasks beschränkt ist, sondern eine beliebige Anzahl von Tasks parallel zueinander ausführen kann. [Mic106]

### *Parallel Language Integrated Query (PLINQ)*

Die mit dem .NET-Framework 3.5 hinzugekommene Abfragesprache LINQ erfährt mit Version 4.0 ebenfalls Erweiterungen im Bereich der Parallelität. Die Idee hinter LINQ ist es, datenbasierte Abfragen grundsätzlich auch auf Objekten zu ermöglichen und die Syntax zu vereinheitlichen. Dadurch kann für Abfragen auf Objekten, XML-Daten, SQL-Daten und ADO .NET Entitäten[33] die gleiche, SQL-ähnliche, Syntax genutzt werden. [Wes07]

PLINQ stellt dafür parallelisierte Methoden dieser Abfragesprache bereit, die sich jedoch nur auf parallele Abfragen von Objekten und XML-Daten beschränken. Als Grund für diese vermeintliche Einschränkung gibt Microsoft an, dass SQL-Server sowie ADO .NET ihrerseits selber für Parallelität sorgen und dies somit nicht durch PLINQ erfolgen muss. [Duf07]

Um eine PLINQ-Anfrage zu schreiben, ist das Quellobjekt der bisherigen LINQ-Anfrage lediglich um `.AsParallel()` zu erweitern, wie das folgende Beispiel zeigt [Mic1011]:

```
using System.Linq;
[…]
// Liste von Integer erstellen
List<int> myList = Enumerable.Range(0, MAX_COUNT).ToList<int>();
// PLINQ-Anfrage, die alle ungeraden Werte in uneven_numbers speichert
ParallelQuery<int> uneven_numbers = from item in myList.AsParallel()
    where item % 2 != 0 select item;
```

[33] Relationale Datenbanken, die im Arbeitsspeicher vorgehalten werden.

Der Zugriff auf die in einem generischen `ParallelQuery`-Objekt gespeicherten Ergebnisse kann durch eine Foreach-Schleife erfolgen. Ist eine parallele Auswertung der Ergebnisse möglich, bietet sich dazu entweder die parallele Foreach-Schleife an oder die im `ParallelQuery`-Objekt integrierte Methode [Mic1011]:

```
ForAll<T>(Action<T> action)
```

Durch Verwendung der `ForAll`-Methode können die Teilergebnisse, die bei PLINQ aufgrund der vielen Threads entstehen, direkt in `ForAll` weiterverarbeitet werden und ersparen so die sonst nötige Reduzierung auf eine sequentielle Struktur. Dies macht die parallele Auswertung mit `ForAll` effizienter als mit einer parallelen Foreach-Schleife. [Tou09] Analog zu `Parallel.ForEach` hat der Programmierer auch bei PLINQ durch zusätzliche Parameter die Möglichkeit, die Partitionierung zu beeinflussen oder eine Abbruchsteuerung zu implementieren. [Mic1011]

### *Zusammenfassung*

Bei der Entwicklung der parallelen Erweiterungen für Visual C++ und .NET war Microsoft bemüht, für beide Sprachen ähnliche Konstrukte bereitzustellen. Daher ist es auch nicht verwunderlich, dass die Parallel Extensions die gleichen parallelen Eigenschaften wie die Concurrency Runtime aufweisen. Jedoch hat man bei der TPL auf die Unterstützung von Datenfluss-Modellen und somit Pipelining verzichtet, die die CRT mit der Asynchronous Agents Library bietet. Auch ist die parallelisierte Abfragesprache PLINQ nur unter .NET verfügbar, was auf die generelle Beschränkung von LINQ auf .NET-Anwendungen zurückzuführen ist. Durch die objektorientierte und natürliche Integration in die Sprache lassen sich die parallelen Erweiterungen schnell erlernen und leicht verwenden.

Obwohl .NET 4.0 in der finalen Version erst im April dieses Jahres veröffentlicht wurde, existieren bereits einige Bücher, die sich mit den neuen Funktionen des Frameworks beschäftigen (z.B. [Dob10], [Tro10]). Weitere Bücher sind angekündigt und sollen in der nächsten Zeit erscheinen[34].

[34] Quelle: Amazon.de, Stand: 12.08.2010

### 4.1.7 Intel Cilk++

Cilk wurde am MIT[35] Computer Science and Artificial Intelligence Laboratory als Spracherweiterung für C zur parallelen Programmierung entwickelt. Mit der Gründung des Startup-Unternehmens Cilk Arts Inc. wollten die Entwickler im kommerziellen Bereich an den Erfolg anknüpfen und erweiterten Cilk zu Cilk++. Dadurch kann Cilk++ in C++-basierten Anwendungen eingesetzt werden und erfuhr zudem weitere Verbesserungen, wie z.B. eine parallele For-Schleife. [MIT07] Intel kaufte Cilk Arts im August 2009 auf und stellt Cilk++ aktuell als Software Development Kit (SDK) für C++ kostenlos für den GCC-Compiler unter Linux und den Visual C++-Compiler unter Windows zur Verfügung. [Int103] Bestandteil des SDKs ist ein eigener Compiler, der beim Aufruf bestimmte Vorarbeiten und Transformationen auf dem Cilk-Quellcode ausführt und anschließend den GCC bzw. Visual C++ Compiler startet. [Int09]

Das Besondere an Cilk++ ist die sehr einfache Verwendung der Spracherweiterung, die sich auf lediglich drei Befehle beschränkt [Lei09]:

- `cilk_for`
- `cilk_spawn`
- `cilk_sync`

Mittels dieser drei Befehle kann sowohl Daten- als auch Taskparallelität in einem Programm erzielt werden. Zusätzlich stellt Cilk++ sogenannte *Hyperobjects* bereit, die in kritischen Bereichen Daten threadsicher manipulieren können. Anhand dieser Punkte erfolgt im weiteren Verlauf die Beschreibung der Befehle. [Lei09] Zuvor muss jedoch etwas genauer auf den Sprachkontext eingegangen werden, bei dem zwischen einem C/C++- und einem Cilk++-Kontext unterschieden wird.

#### *Sprachkontext*

Innerhalb des Cilk++ Sprachkontextes können Cilk-spezifische Funktionen und Objekte sowie, durch eine gesonderte Kennzeichnung, bereits vorhandene C/C++ Funktionen verwendet werden. Der andere Fall ist jedoch nicht ohne weiteres möglich (s.u.). Um Anweisungen in einem bestimmten Sprachkontext zu definieren, wird das `extern`-Schlüsselwort, gefolgt

[35] Massachusetts Institute of Technology

vom gewünschten Kontext („Cilk++“, „C“ oder „C++“), verwendet. Ausnahmen bilden die Funktionen `main()` und `cilk_main()`, die immer im C- bzw. Cilk++-Kontext gestartet werden. Die Sprachbindung betrifft demnach Funktionen, aber auch strukturierte Datentypen, Klassen und Unions. Elementare Typen, Enumerationen und `typedef` sind nicht auf einen speziellen Sprachkontext beschränkt und ermöglichen eine übergreifende Verwendung. [Int09]

Um aus einem C++-Kontext heraus Cilk++-Anweisungen ausführen zu können, ist die Header-Datei `cilk.h` zu inkludieren und die Verwendung von speziellen Wrapper-Funktionen erforderlich, die beim Aufruf einen Cilk++-Kontext erzeugen. Diese sind [Int09]:

- `cilk::context::run` bzw. in verkürzter Form auch
- `cilk::run`

Von diesen Möglichkeiten sollte allerdings nur spärlich Gebrauch gemacht werden, da das Starten eines Cilk++-Kontextes für zusätzlichen Overhead sorgt. Diese Vorgehensweise bietet sich daher bei einer bereits vorhandenen C/C++-Codebasis an, in der einzelne Stellen inkrementell mit Cilk++ parallelisiert oder wenn andere Threading-Konzepte, wie z.B. Pthreads, mit Cilk++ kombiniert verwendet werden sollen. Andernfalls bietet sich der direkte Start der Anwendung im Cilk++-Kontext an. Dies wird durch die Endung *.cilk im Dateinamen und durch Verwendung der Funktion `cilk_main()` gekennzeichnet. [Int09]

### *Datenparallelität*

Zur Erzielung von Datenparallelität gibt es in Cilk++ die `cilk_for`-Methode, die eine parallelisierte Variante der for-Schleife aus C/C++ darstellt. Eine mögliche Verwendung von `cilk_for` kann wie folgt aussehen [Int09]:

```
cilk_for (int i = start; i < end; i += step) {
    // Parallele Arbeit
}
```

Wie bei den parallelen For-Schleifen in OpenMP sind an die Cilk++-Schleifen ebenfalls gewisse Bedingungen geknüpft, die bei der Verwendung zu beachten sind. Analog zu OpenMP dürfen innerhalb des Schleifenkörpers die Zählvariable und der Iterationsbereich nicht ver-

ändert und keine `break`- oder `return`-Anweisungen aufgerufen werden. Zusätzlich muss die Deklaration der Zählvariablen im Schleifenkopf stattfinden. [Int09]

Auf die Scheduling-Strategie hat der Programmierer allerdings keinen Einfluss. Nur ein manuelles Einstellen der *chunk size* (hier als *grainsize* bezeichnet) ist mit

```
#pragma cilk_grainsize = expression
```

optional möglich.

Expression kann ein atomarer Wert sein (z.B. 1) oder auch eine Formel, die die *chunk size* bspw. anhand der Anzahl an Iterationen berechnet. Wird keine *chunk size* angegeben, berechnet Cilk++ diese mittels der Formel

$$\min\left(512, \frac{n}{8 * p}\right)$$

automatisch. $n$ steht für die Anzahl auszuführender Iterationen und $p$ für die zu verwendenden Threads. [Int09]

### *Taskparallelität*

Mit der Funktion `cilk_spawn` lässt sich Taskparallelität erzeugen, indem die Cilk++-Laufzeitumgebung, unter Angabe einer Prozedur oder Funktion, dafür sorgt, dass der Aufruf in einem separaten Thread ausgelagert wird. `cilk_spawn` ist kein Garant für eine parallele Ausführung, sondern eher als Hinweis zu betrachten. Ob der Aufruf letztendlich parallelisiert wird, entscheidet die Laufzeitumgebung. Die aufzurufende Prozedur bzw. Funktion muss sich dafür in einem Cilk++-Sprachkontext befinden. [Int09]

```
// Rückgabeloser Aufruf
cilk_spawn DoWorkProc();
```

Dieser Aufruf von `cilk_spawn` ist vergleichbar mit der Task-Direktive `#pragma omp task` von OpenMP. Die beiden Varianten unterscheiden sich dadurch, dass Cilk++ die Ergebnisse der Tasks als Rückgabewert in einer Variablen speichern kann.

```
// Aufruf produziert einen Rückgabewert
int result;
result = cilk_spawn DoWorkFunc();
```

Der aufrufende Thread kehrt nach dem Starten des Tasks sofort zurück, kann aber erst dann auf das Ergebnis des erzeugten Tasks (sicher) zugreifen, wenn dieser vollständig beendet ist. Mit `cilk_sync` kann auf alle Tasks gewartet werden, die innerhalb einer Methode erzeugt wurden. Ein implizites `cilk_sync` erfolgt beim Verlassen einer Methode automatisch. Dies hat zur Folge, dass Tasks in Cilk++ niemals länger aktiv sein können, als die Methode, in der sie erzeugt wurden. [Lei09] Die Entwickler begründen diese Designentscheidung damit, dass somit einerseits die vorhandenen Systemressourcen besser kontrolliert werden können und andererseits dieses Verhalten mehr dem Muster eines sequentiellen Programms entspricht, da potentielle Seiteneffekte bei weiterhin aktiven Tasks vermieden werden. [Car09]

Durch den sehr geringen Ressourcenverbrauch der `cilk_spawn`-Methode[36] und der Garantie, bei einer Ausführung auf $p$ Prozessoren maximal den $p$-fachen Platz auf dem Ausführungsstapel *(Call Stack)* zu belegen, eignet sich Cilk++ optimal für „Divide and Conquer“ Algorithmen. [Car09,Int09] Dies macht das Zurückfallen auf eine sequentielle Arbeitsweise ab einer gewissen Rekursionstiefe überflüssig, da die von `cilk_spawn` benötigten Ressourcen vernachlässigbar gering sind.

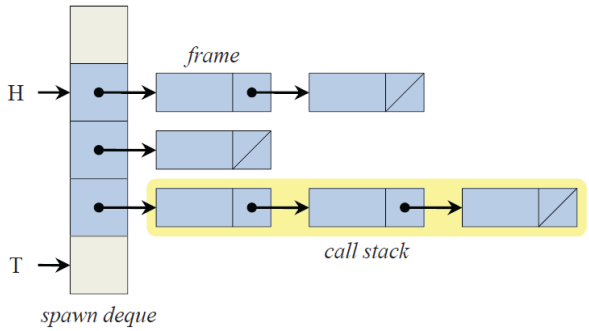



**Abbildung 4-2: Datenstruktur eines Workers in Cilk++ [Fri09]**

Während der Ausführung von parallelen Tasks benutzt die Laufzeitumgebung von Cilk++ *Work Stealing*, um alle Ressourcen möglichst gleichmäßig auszulasten. Zu Beginn erzeugt die Laufzeitumgebung so viele *Worker-Threads*, wie Prozessoren bzw. Prozessorkerne vorhan-

[36] Der Aufwand ist um den Faktor 400 geringer als bei der Erzeugung eines Pthreads.

den sind[37]. Jeder *Worker* verfügt über eine Liste, auf die sowohl von unten (T) als auch von oben (H) zugegriffen werden kann (siehe Abbildung 4-2). [Lei09] Beim Aufruf einer Methode mittels `cilk_spawn` wird in diese Liste ein sogenannter *Frame* eingefügt, der einem Call Stack ähnelt und alle benötigten Informationen zur Ausführung der Methode aufnimmt. Die Worker greifen von unten auf die Elemente der Liste zu und löschen den Frame, sobald die Ausführung der dort definierten Methode beendet ist. Sollte es im Verlauf der Ausführung dazu kommen, dass ein Worker über eine leere Liste verfügt, dann schaut er in die Liste eines zufällig ausgewählten anderen Workers und „stiehlt" Frames von dem oberen Ende dieser Liste. [Fri09]

### *Hyperobjects*

Die Idee hinter Hyperobjects ist es, mehrere Zugriffe auf ein Objekt durch verschiedene Threads zu koordinieren. Dafür wird jedem Thread eine private Sichtweise für den Zugriff auf das Objekt ermöglicht. In Cilk++ gibt es drei konkrete Hyperobjects: Reducer, Holder und Splitter, von denen hier aus zeitlichen Gründen nur auf die Reducer eingegangen werden soll. Eine detaillierte Darstellung aller Hyperobjects und ihrer Implementierung ist unter [Fri09] zu finden.

Reducer werden in Cilk++ für Reduktionsoperationen verwendet. Dies ist ein Grund ist, warum keine expliziten Schleifenkonstrukte für diese Operationen existieren (ähnlich wie bei der Concurrency Runtime). Dadurch sind Reduktionsoperationen nicht nur auf parallele Schleifen beschränkt, sondern lassen sich auch bei taskparallelen Anwendungen nutzen. Ein Reducer ist eine globale Variable, die von mehreren Threads gleichzeitig verwendet werden kann, ohne dafür auf Mechanismen der Synchronisation (z.B. Locks) zurückgreifen zu müssen. Sobald die verschiedenen Threads ihre parallele Ausführung beendet haben, werden ihre lokalen Ergebnisse zu einem Gemeinsamen zusammengeführt. Hierbei stellt Cilk++ sicher, dass ihre sequentielle Semantik erhalten bleibt. Bspw. kann das Resultat eines List-Reducers nicht zwischen einem parallelen und einem sequentiellen Hinzufügen von Einträgen unterschieden werden. [Lei09] Cilk++ liefert eine Reihe von vorgefertigten Reducern mit,

[37] Kann beeinflusst werden durch das Kommandozeilenargument: `-cilk_set_worker_count`

die unter [Int09] detailliert aufgelistet sind. Das folgende Beispiel zeigt die Verwendung eines Additions-Reducers:

```
#include <reducer_opadd.h>
[…]
cilk::reducer_opadd<int> sum;

cilk_for(int i = 0; i < 100000; i++)
    sum+= i;

printf("Summe: %d\n", sum.get_value());
```

Sollten die mitgelieferten Reducer nicht ausreichen, ist auch die Definition eines eigenen Reducers möglich. Die Vorgehensweise für das Erstellen eigener Reducer kann in den Veröffentlichungen von [Fri09] und [Int09] nachgelesen werden.

#### *Zusammenfassung*

Mit Cilk++ hat Intel neben einer parallelen Bibliothek nun auch eine Spracherweiterung für C++ im Portfolio. Cilk++ bietet Unterstützung für die Daten- und Taskparallelität auf einem Abstraktionsniveau ähnlich TBB oder der CRT. Da Cilk++ mit nur drei Schlüsselwörtern auskommt, ist die Funktionsweise schnell erlernbar. Da stellt auch der zurzeit noch eher geringe Umfang an Literatur kein Hindernis dar, die derweil nur in Form von Spezifikationen und Handbüchern zur Verfügung steht und auf die im Laufe dieses Kapitels bereits referenziert wurde. Das Debugging kann mit dem Werkzeug *Cilkscreen* ebenfalls erleichtert werden, indem der Code nach potentiellen Race Conditions[38] untersucht wird. Eine genaue Erläuterung der Algorithmen zum Finden der Race Conditions ist in [Che98] dargestellt.

### 4.1.8 Intel Ct

Intel Ct (C for Throughput Computing) ist ein sich in Entwicklung befindliches Programmiermodell für Datenparallelität basierend auf der Syntax von C++ und steht einigen ausgewählten Testern als Beta-Version zur Verfügung. Der Grund für die Entwicklung ist der zunehmende Trend hin zu Many-Core Systemen, bei denen die parallele Verarbeitung von Daten einen immer höheren Stellenwert einnimmt. Ein weiterer Grund für eine eigene datenparal-

[38] Eine Race Condition entsteht, wenn mehrere parallele Anweisungen auf eine gemeinsame Variable zugreifen und in mind. einer Anweisung in die Variable ohne Synchronisation geschrieben wird.

lele Spracherweiterung ist der Larrabee-Prozessor von Intel, der für Anfang 2010 angekündigt und dann auf unbestimmte Zeit verschoben wurde. Mit Ct könnte Intel neben Nvidia mit CUDA und AMD/ATI mit Stream sein eigenes datenparalleles Programmiermodell anbieten. [Fel09]

Anwendungen basierend auf Ct lassen sich auf einem sehr viel abstrakteren Niveau schreiben als bspw. Anwendungen für CUDA oder OpenCL, auf die im weiteren Verlauf dieser Arbeit noch eingegangen wird. Der hohe Abstraktionsgrad ermöglicht eine wissenschaftlichere Darstellung von dem, was das Programm letztendlich berechnen soll[39]. [Fel09] Der Programmierer muss sich dabei nicht um die Zerlegung der Daten und die Abbildung auf Systemressourcen kümmern, da dies alles von Ct übernommen wird. Ein weiteres Ziel bei der Entwicklung war es, eine sehr hohe Portabilität für unterschiedliche Architekturen basierend auf Multi- und Many-Core Systemen zu ermöglichen. Abbildung 4-3 skizziert die grundsätzlichen Komponenten von Ct, die im Folgenden näher erläutert werden.

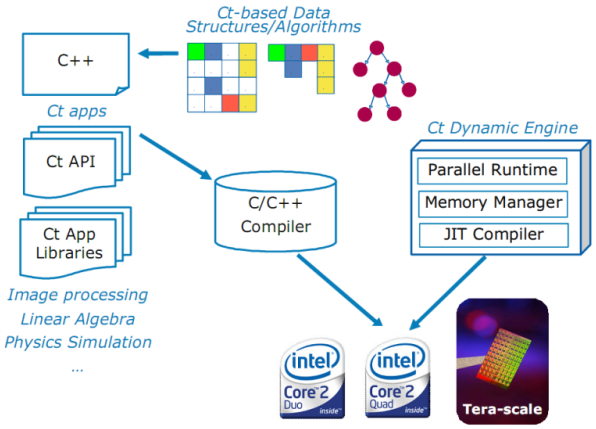



**Abbildung 4-3: Aufbau und Verwendung von Intel Ct [Ghu07]**

Die Basis einer Ct-Anwendung bilden die Ct-Datenstrukturen, auf die an späterer Stelle eingegangen wird, sowie spezielle Bibliotheken, mit denen spezifische Aufgaben aus den Bereichen der Bildverarbeitung oder der linearen Algebra durchgeführt werden können. [Ghu07] Zur Verwendung der Bibliotheken ist die Header-Datei `ct.h` einzubinden. Das resultierende Programm kann anschließend mit gewöhnlichen Compilern wie Visual C++ oder dem GNU C/C++ Compiler kompiliert werden. Innerhalb der Ct-Anwendung kann der Programmierer

[39] Damit ist die Darstellung von Arbeitsschritten gemeint, die so auch im wissenschaftlichen / mathematischen Umfeld erfolgen würde. Bspw. eine direkte Multiplikation zweier Matrizen ohne die Verwendung von Schleifen.

sowohl C++-Code, als auch spezifische Ct-Instruktionen verwenden. Die Ausführung der Ct-Anweisungen übernimmt die *Ct Dynamic Engine*, die sich aus insgesamt drei Teilen zusammensetzt [Ghu07]:

- Threading Runtime

Die Threading Runtime (TRT) erzeugt anhand des zugrundeliegenden Zielsystems bzw. der Architektur eine geeignete Anzahl an Threads. [Ghu10]

- Memory Manager

Der Memory Manager ist für die Trennung der Speicherbereiche der C++- und der Ct-Datenstrukturen verantwortlich. Zusammen mit der TRT sorgt der Memory Manager für eine Zerlegung der Daten, die dann auf die von der TRT erstellten Threads abgebildet werden. Außerdem gehört die *Garbage Collection* zu seinen weiteren Aufgaben, die nicht weiter benötigte Datenstrukturen entfernt. [Ghu10]

- Compiler

Der Compiler ist aufgrund seines dynamischen Konzeptes von besonderer Bedeutung. Trifft das Programm während seiner Ausführung auf eine Ct-Anweisung, sorgt dies für die Intervention der Ct Dynamic Engine. Die Dynamic Engine erzeugt daraufhin optimierten Zwischencode, der für eine Wiederverwendung auch zwischengespeichert werden kann. Die Erzeugung des Zwischencodes geschieht in drei Phasen [Ghu07,Ghu10]:

    - High-Level Optimizer – Führt architekturunabhängige Optimierungen durch (z.B. Dekomposition, Umwandlungen in Skalarform u.a.).
    - Low-Level Optimizer – Generiert parallelisierten Code mittels der TRT, der anschließend vektorisiert und in eine architekturunabhängige Darstellungsform gebracht wird (VIP).
    - Virtual Intel Platform (VIP) Code Generator – Führt Optimierungen zur Laufzeit entsprechend der verwendeten Architektur durch, indem die architekturunabhängigen skalaren Operationen auf architekturabhängige Operationen der konkreten Architektur abgebildet werden.

Durch die hohe Dynamik des VIP Code Generators erzielt die gleiche Anwendung auf unterschiedlichen Architekturen immer eine hohe Effizienz, ohne von einem Compiler explizit für eine bestimmte Architektur optimiert zu werden. Dies geschieht bei Ct zur Laufzeit. [Ghu07]

Im Folgenden soll kurz gezeigt werden, wie sich Parallelität mittels Ct ausdrücken lässt. Weiter oben wurde erwähnt, dass die Darstellungsweise von Ct eher wissenschaftlicher Natur ist. Dies liegt daran, dass Operationen in Ct nicht elementweise mit Schleifen, sondern aggregiert basierend auf Containern durchgeführt werden. Container bilden die zentrale Struktur, um Parallelität zu definieren. Es wird zwischen den zwei Containern `dense` und `nested` unterschieden. Erstere sind vergleichbar mit einem eindimensionalen Array, wohingegen `nested`-Container in bis zu maximal drei Dimensionen geschaltet sein können. Bei der Definition ist die Art des Containers anzugeben (`dense`/`nested`) sowie die zu speichernde Datenstruktur[40]. [Ghu10]

```
dense<i32> my_dense_ct_container = {4711, 4712, 4713, 4714};
nested<i32> my_nested_ct_container = { {4711, 4712}, {4713, 4714} };
```

Neben den bereits von Ct vordefinierten skalaren Typen können auch eigene Datenstrukturen genutzt werden (z.B. `structs`). Eine Bedingung ist jedoch, dass sämtliche Datenstrukturen innerhalb der Struktur die vordefinierten Datentypen von Ct benutzen. [Ghu10]

Um mit diesen Datenstrukturen Parallelität zu erzielen, stellt Ct skalare Operationen für Vektoren, kollektive Operationen (Reduktionen und Scans / Prefix-Summen) und Permutations-Operationen (Gather, Scatter) zur Verfügung. Auch ist die Definition von benutzerdefinierten Operationen möglich. [Ghu07] Damit die Operationen aus einem C/C++-Kontext heraus verwendet werden können, wird der *C++-Invoker* benötigt. Der C++-Invoker hat die Aufgabe, die Ct-spezifischen Funktionen aufzurufen und die Ausführung dann der Ct Dynamic Engine zu übergeben. Das Schlüsselwort `call` ruft den Invoker auf [Ghu10]:

```
const int N = 5;
double A[N], B[N], C[N]; // Arrays A und B bereits gefüllt
// C/C++ Code
int main() {
    for(int i = 0; i < N; i++)
```

[40] Integer mit 8-, 16-, 32- und 64-Bit, Floating-Point mit 32- oder 64-Bit oder Boolean

```
            C[i] = A[i] * B[i];
}
// Ct Code
#include <ct.h>
using namespace ct;
void calc(dense<f64> ctA, dense<f64> ctB, dense<f64> &ctC) {
    ctC = ctA * ctB;
}
int main() {
    dense<f64> ctA(A, N), ctB(B, N), ctC(C, N);
    call(calc)(ctA, ctB, ctC);
}
```

Mittels `call` werden zu Anfang die Daten aus den Datenstrukturen A, B und C von C/C++ in die Ct-Datenstrukturen `ctA`, `ctB` und `ctC` kopiert. Danach wird die Funktion `calc` vom Invoker aufgerufen, der die Daten wieder in die ursprünglichen Strukturen zurückkopiert, sobald die Funktion beendet ist. Soll das Ergebnis nicht als Referenz in einer Variablen, sondern als Rückgabewert der Funktion übertragen werden, steht dafür die Methode `map` bereit. [Ghu10]

### *Zusammenfassung*

Mit Ct entwickelt Intel ein sehr interessantes Programmiermodell für Datenparallelität, bei dem sich der Programmierer durch den sehr hohen Abstraktionsgrad lediglich darum kümmern muss, Ct-Anweisungen an den geeigneten Stellen einzusetzen. Eine manuelle Zerlegung der Aufgaben und die Abbildung auf Systemressourcen sind nicht erforderlich. Ebenfalls kann auf Methoden zur Synchronisation verzichtet werden, da Ct ein deterministisches Modell zugrunde liegt. Das bedeutet, dass sich alle Operationen so verhalten, als würde man sie auf einer sequentiellen Maschine ausführen. [Ghu07]

Da Ct noch nicht von Intel veröffentlicht wurde und nur als geschlossene Beta-Version einigen Testern zur Verfügung steht, ist bisher nur sehr wenig Dokumentation vorhanden. Eine Übersicht der verfügbaren Literatur ist auf der Intel-Seite von Ct zu finden[41]. Ebenfalls bleibt noch abzuwarten, welches Lizensierungsmodell Ct zugrunde liegen wird. Hier könnte sich Intel, wie bei TBB, dazu entschließen, Ct in einer kommerziellen und einer kostenfreien Ver-

[41] http://software.intel.com/en-us/data-parallel/

sion zur Verfügung zu stellen, da CUDA und OpenCL ebenfalls frei verwendet werden können.

### 4.1.9 Grand Central Dispatch

Apple hat in seiner aktuellsten Version 10.6 von Mac OS X einige neue Technologien eingeführt. Eine davon wird mit *Grand Central Dispatch* (GCD) bezeichnet und erweitert die Sprachen C, C++ und Objective-C um Funktionen, mit denen Programmierer die steigende Anzahl an Prozessoren bzw. Prozessorkernen in Mac-basierten Systemen besser ausnutzen können. Damit GCD effizient arbeiten kann, wurde die Technologie tief im Systemkern von Mac OS X integriert und kümmert sich dort um die Erzeugung und Zuweisung von Threads in Abhängigkeit der vorhandenen Hardware und Auslastung. Dafür wird zu Beginn ein Thread-Pool erzeugt, dessen Charakteristiken durch das verwendete System festgelegt werden (z.B. Anzahl an Threads). Anwendungen, die GCD nutzen, können ihre Arbeit einem sogenannten *Dispatcher* übergeben, der sie einem Thread zuordnet oder, wenn alle Threads beschäftigt sind, einer Warteschlange hinzufügt. Der Aufwand für das Hinzufügen zu einer Warteschlange wird mit 15 Instruktionen angegeben, wohingegen das manuelle Erstellen eines Threads mit dem 50-fachen an Aufwand verbunden wäre. Warteschlangen bilden die Kernkomponenten von GCD, mit denen Arbeit dem Dispatcher übergeben werden kann. Diese kann sowohl Daten- als auch Taskparallelität umfassen. [App09] Damit die Funktionen von GCD in einer Anwendung genutzt werden können, muss ein Verweis auf die Header-Datei `dispatch.h` erfolgen.

#### *Warteschlangen*

In Warteschlangen können Tasks platziert werden, die entweder parallel oder hintereinander auszuführen sind. Die Spezifikation und Übergabe eines Tasks erfolgt entweder durch die Angabe einer Prozedur oder durch einen *Block*. Blöcke sind eine Erweiterung von Apple für die Sprachen C, C++ und Objective-C. Mit ihnen wird die Kapselung von (anonymen) Anweisungen innerhalb von Variablen ähnlich wie mit Lambda-Ausdrücken ermöglicht. [App091] Weitere Informationen über die Syntax und die Verwendung von Blöcken ist in der zuvor referenzierten Quelle zu finden.

Zur Aufnahme der Tasks stellt GCD drei verschiedene Warteschlangen *(Queues)* zur Verfügung [App092]:

- Concurrent Queues
- Serial Queues
- Main Dispatch Queues

Allen Queues gemeinsam ist das FIFO-Prinzip, bei dem zuerst die Tasks aus der Queue entnommen werden, die als erstes eingefügt wurden. Dies gilt auch für *Concurrent Queues*, denen so viele Tasks entnommen werden können, wie Threads zur Verfügung stehen. Letztere können somit parallel zueinander arbeiten. Mit der Funktion

```
dispatch_get_global_queue(QUEUE_PRIORITY, 0)
```

wird auf die Concurrent-Queue zugegriffen. Insgesamt gibt es drei dieser Warteschlangen, die sich durch ihre Priorität unterscheiden[42]. Der zweite Parameter wurde für zukünftige Zwecke reserviert und sollte bei der jetzigen Implementierung immer mit 0 belegt sein. [App092]

GCD stellt *Serial Queues* nicht automatisch zur Verfügung. Sie müssen von der Anwendung selber erzeugt werden und ermöglichen die sequentielle Ausführung von Tasks gemäß FIFO. Dies kann bei Tasks mit vielen Abhängigkeiten unter gewissen Umständen effektiver sein als die Verwendung von Locks. Innerhalb der Anwendung können beliebig viele Serial Queues erstellt werden, wobei Tasks in unterschiedlichen Serial Queues weiterhin parallel zueinander ausgeführt werden. Mit Hilfe der folgenden Anweisung wird eine solche Queue erstellt [App092]:

```
dispatch_queue_t sQueue;
sQueue = dispatch_queue_create("Name der Queue", NULL);
```

Programmierer können Serial Queues Namen geben, um sie beim Debugging leichter voneinander unterscheiden zu können. Der zweite Parameter ist ebenfalls für zukünftige Zwecke reserviert und sollte mit NULL initialisiert werden. [App092]

[42] `DISPATCH_QUEUE_PRIORITY_LOW/DEFAULT/HIGH`

Die letzte Warteschlange ist die *Main Dispatcher Queue*, die Tasks explizit dem Thread zuordnet, unter dem auch die Anwendung gestartet wurde. Die Ausführung der Tasks innerhalb dieser Warteschlange erfolgt ebenfalls sequentiell. Sie bietet sich bspw. für Synchronisationsoperationen an oder für Anwendungen, in denen eine Manipulation der Benutzeroberfläche notwendig ist[43]. Zugriff auf die Main Dispatcher Queue erhält man durch die parameterlose Funktion `dispatch_get_main_queue()`. [App092]

### *Synchronisierung*

Zur Synchronisation stellt GCD Semaphore bereit, denen bei der Erzeugung die Anzahl an verfügbaren Ressourcen übergeben wird und die in ihrer Funktionsweise einem Lock ähneln. Die Anzahl bestimmt, wie oft das Semaphor angefordert werden kann, bevor der Thread blockiert. Der Vorteil an Semaphoren in GCD besteht darin, dass nur dann der Kernel in die Blockierung eines Threads involviert werden muss, wenn keine Ressourcen mehr verfügbar sind. [App092] Mit der Funktion

```
dispatch_semaphore_t dispatch_semaphore_create(long resources);
```

wird ein Semaphor erstellt [App10]. Mit den zwei Funktionen

```
long dispatch_semaphore_wait(dispatch_semaphore_t semaphor,
    dispatch_time_t timeout);
long dispatch_semaphore_signal(dispatch_semaphore_t semaphor);
```

wird das Semaphor angefordert bzw. freigegeben. [App10]

### *Datenparallelität*

Mit Hilfe der Funktion `dispatch_apply` bzw. `dispatch_apply_f` können Schleifen parallelisiert werden. Beim Aufruf wird die Anzahl der Iterationen angegeben (0 bis Iterationen – 1) sowie der auszuführende Block bzw. die auszuführende Funktion. Die Ausführung übernimmt die Concurrent Queue und ist dieser dementsprechend zu übergeben. [App092] Das nachfolgende Beispiel soll die Schritte demonstrieren [App092]:

[43] Bei Cocoa-Anwendungen (einer API zur Entwicklung von Anwendungen mit einer GUI) kann nur vom Main-Thread aus auf die Elemente der Benutzeroberfläche zugegriffen werden.

```
int max_iter = 1000; // Anzahl Iterationen
dispatch_queue_t cQueue = dispatch_get_global_queue(DISPATCH_QUEUE
    _PRIORITY_DEFAULT, 0); // Concurrent-Queue beziehen
// Parallele Schleife ausführen
// Entspricht for(int i = 0; i < max_iter; i++) { ... }
dispatch_apply(max_iter, cQueue, ^(int i) {
   // Zu erledigende Arbeit im Schleifen-Körper
});
```

### *Taskparallelität*

Damit Tasks parallel ausgeführt werden können, müssen sie einer (Concurrent) Queue hinzugefügt werden. Für das Hinzufügen stellt GCD sowohl eine synchrone als auch eine asynchrone Variante zur Verfügung. Während die asynchrone Variante sofort zurückkehrt, wird bei der synchronen Variante gewartet, bis alle Tasks innerhalb der Queue abgearbeitet sind.

Die zwei Methoden sind folgendermaßen definiert [App10]:

```
// Asynchrone Ausführung
void dispatch_async(dispatch_queue_t queue, dispatch_block_t block);
// Synchrone Ausführung
void dispatch_sync(dispatch_queue_t queue, dispatch_block_t block);
```

Der erste Parameter gibt die gewünschte Queue an, mit dem zweiten wird der auszuführende Block übergeben. Zur Ausführung einer Funktion anstelle eines Blockes können `dispatch_async_f` bzw. `dispatch_sync_f` verwendet werden. [App10] Das Hinzufügen und Entfernen von Elementen der Warteschlange geschieht innerhalb von GCD atomar durch hardwaregestützte Operationen. [App09]

Soll nur auf die Beendigung bestimmter Tasks gewartet werden (und nicht auf alle in der Warteschlange), existieren für diesen Zweck Task-Gruppen. Tasks werden mit `dispatch_group_async(_f)` unter Angabe einer Queue sowie einer Gruppe gestartet. Für das Warten auf Tasks der Gruppe steht der Befehl `dispatch_group_wait` zur Verfügung, dem die gewünschte Gruppe sowie eine maximal zu wartende Zeitspanne übergeben wird. [App092]

***Zusammenfassung***

Der große Vorteil von Grand Central Dispatch liegt an der tiefen Integration in das System. Dadurch können aktuell zwar nur Anwendungen für Mac OS X von den Möglichkeiten profitieren, jedoch stehen sie allen Anwendungen unabhängig vom Compiler zur Verfügung. Die einzige Bedingung an den Compiler ist die Unterstützung von Blocks. Da Apple den Quellcode zu GCD veröffentlich hat, besteht die Möglichkeit, dass das Konzept auch in anderen Betriebssystemen übernommen wird und ein größerer Kreis an Anwendungen diese Konzepte nutzen kann.

GCD ermöglicht Daten- und Taskparallelität, wobei der Funktionsumfang bei der Datenparallelität aufgrund fehlender Strukturen zur Reduktion geringer ausfällt als bspw. bei OpenMP oder TBB. Aufgrund des identischen Abstraktionsgrades braucht sich der Programmierer, neben der Angabe der Parallelität, nur um die Zerlegung (bei der Taskparallelität) sowie an geeigneten Stellen um die Synchronisation zu kümmern. Die gute Integration in die Sprachen C, C++ und Objective-C durch Blocks sowie eine sehr ausführliche Referenz sorgen dafür, dass die Funktionen von GCD schnell erlernt und einfach in Anwendungen integriert werden können. Neben den bereits referenzierten Quellen bietet [Sir09] eine ausführliche Einführung in die Neuerungen von Mac OS X 10.6 und insbesondere auch GCD.

## 4.2 Ansätze für Systeme mit programmierbaren Grafikkarten

### 4.2.1 ATI Stream

ATI bzw. AMD stellt mit dem Stream SDK Entwicklungswerkzeuge zur Verfügung, mit denen aktuelle ATI-Grafikkarten in Kooperation zu der (oder den) CPU(s) für die Berechnung verschiedener Aufgaben genutzt werden können. In der Version 1.x besteht das SDK aus den folgenden Komponenten [AMD09]:

- Compiler
- Profiling-Werkzeuge
- AMD Core Math Library (ACML)

Als Entwicklungssprache setzte ATI auf die High-Level Programmiersprache Brook+. Sie entspricht einer für ATI-Hardware optimierten Version von BrookGPU, die an der Stanford Uni-

versität als quelloffener Compiler mit Laufzeitumgebung als eine Erweiterung für C zur Programmierung von Grafikprozessoren entwickelt wurde[44]. [AMD07] Der `brcc`-Compiler wandelt den geschrieben Brook+-Code sowohl in IL[45]-Code für die Grafikkarte als auch in C-Code für die CPU um. Zur Realisierung von Brook+ hat ATI eine weitere Zwischenschicht in Form des CAL[46] eingeführt. CAL ist eine Bibliothek im Treiber der Grafikkarte und bildet die Schnittstelle zu den dortigen Shader Cores bzw. Stream Prozessoren. [AMD09] Eine vereinfachte Darstellung des Erzeugungsprozesses sowie der beteiligten Komponenten liefert Abbildung 4-4.

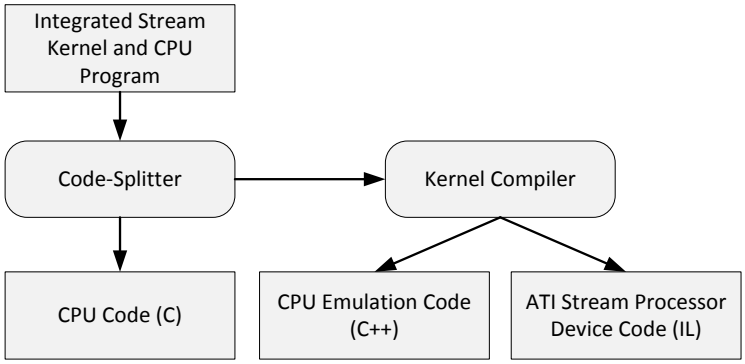



**Abbildung 4-4: Brook+-Komponenten und Erzeugungsablauf [AMD09]**

Profiling wird durch den Stream Kernel Analyzer ermöglicht, mit dessen Hilfe Stream-Programme hinsichtlich Fehler untersucht und anhand bestimmter Messgrößen[47] in Bezug zu ihrer Performance evaluiert werden können. Die AMD Core Math Library enthält für AMD-Architekturen optimierte, mathematische Funktionen; u.a. für die lineare Algebra oder FFT[48]. Die sogenannte ACML-GPU Version dieser Bibliothek enthält eine modifizierte Variante der GEMM[49]-Funktionen für einfache und doppelte Genauigkeit. Wird eine solche Funktion aufgerufen, prüft die Bibliothek den damit verbundenen Aufwand und entscheidet in Echtzeit, ob die Berechnungen auf der CPU oder auf der GPU ausgeführt werden sollen. [AMD09]

[44] http://graphics.stanford.edu/projects/brookgpu/index.html
[45] Intermediate Language
[46] Compute Abstraction Layer
[47] Z.B. die Anzahl an Speicherzugriffen, Verzweigungen usw.
[48] Schnelle Fourier Transformation (Fast Fourier Transform)
[49] General Matrix Multiply

Ab Version 2.x des Stream SDKs wurde die Unterstützung für Brook+ zu Gunsten von OpenCL aufgegeben, dessen Spezifikation Ende 2008 veröffentlicht wurde. Daher wird an dieser Stelle nicht weiter auf Brook+ und die restlichen Komponenten des Stream SDKs 1.x eingegangen. Zum Zeitpunkt dieser Arbeit steht die Version 2.1 des Stream SDKs als kostenloser Download für Windows und Linux zur Verfügung. Für eine Liste der unterstützten Grafikkarten siehe [AMD10].

Da es sich bei OpenCL um einen plattformunabhängigen Standard handelt und dieser somit nicht nur auf die Verwendung von ATI-Karten in Verbindung mit dem Stream SDK 2.x beschränkt ist [AMD091], wird in dem Abschnitt 4.2.3 gezielt auf diese Programmierschnittstelle eingegangen und auf weitere Ausführung an dieser Stelle verzichtet.

### 4.2.2 Nvidia CUDA

Nvidias *Compute Unified Device Architecture* (CUDA) ermöglicht Programmierern den hohen Grad an parallelen Recheneinheiten aktueller Nvidia-Grafikkarten auch außerhalb von Computerspielen nutzen zu können. Abbildung 4-5 stellt die Architektur von CUDA dar:

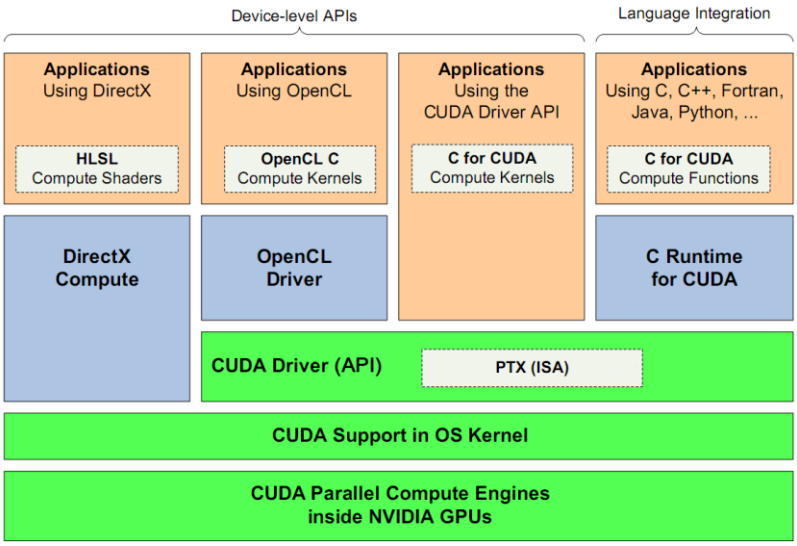



**Abbildung 4-5: Architektur von CUDA [Nvi09]**

Wie in Abbildung 4-5 zu erkennen ist, ermöglicht die CUDA-Architektur mit Hilfe der *Device-Level APIs* auch die Programmierung mit OpenCL- und DirectX-Compute, auf die anschließend noch genauer eingegangen wird. Für die Programmierung von CUDA kann auf die *C Runtime for CUDA* oder die *CUDA Driver API* zurückgegriffen werden. Die CUDA Driver API

bildet die Grundlage für die C Runtime for CUDA und bietet dem Programmierer mehr Kontrolle und Flexibilität auf Kosten eines höheren Programmieraufwandes. Bei der C Runtime for CUDA geschieht vieles automatisch, wie z.B. die Initialisierung der Grafikkarte oder die Parameterübergabe für Kernel-Prozeduren. Deshalb ist sie die bevorzugte Variante zur Programmierung von CUDA-Anwendungen. [Nvi102] Aus diesem Grund wird die CUDA Runtime API auch den Schwerpunkt dieses Kapitels bilden und hinsichtlich ihrer grundsätzlichen Methoden zur Auslagerung von Berechnungen auf die Grafikkarte beschrieben werden. Die Ansteuerung ist direkt aus C- bzw. C++-Anwendungen möglich, indem in der Anwendung sogenannte *Kernel*-Funktionen definiert und diese explizit auf der Grafikkarte ausgeführt werden. Höherwertige Sprachen, wie z.B. Java[50] [51], Python[52] oder .NET[53], können CUDA mit Hilfe der CUDA Driver API und entsprechenden Wrapper-Klassen ebenfalls nutzen. [Nvi09,Nvi102]

Vor der Verwendung von CUDA in einer C- oder C++-Anwendung ist die Installation des CUDA Toolkits erforderlich. Das CUDA Toolkit liegt zum Zeitpunkt dieser Arbeit in Version 3.0 vor und wird von Nvidia kostenlos für Windows, Linux und Mac OS X als Download angeboten. [Nvi10] Nach der Installation stehen die folgenden Komponenten bereit [Nvi10]:

- C for CUDA Compiler (nvcc) und CUDA Debugger (cudagdb)
- CUDA Visual Profiler (cudaprof)
- GPU-beschleunigte Routinen der linearen Algebra (CUBLAS)
- GPU-beschleunigte Routinen für schnelle Fourier Transformationen (CUFFT)
- Dokumentationen und Beispielanwendungen

Zusätzlich muss auf dem System ein C/C++ Compiler vorhanden sein, mit dem die Teile des Programms kompiliert werden, die nicht als Device Code[54] vorliegen. Die Vorgehensweise ist in Abbildung 4-6 dargestellt und ist ähnlich wie bei ATI Stream.

Jede Quellcode-Datei, in der Funktionen der CUDA API genutzt werden, ist vom CUDA-Compiler `nvcc` zu kompilieren. Diese Dateien haben üblicherweise die Endung *.cu. Beim

[50] http://www.jcuda.org/
[51] http://www.hoopoe-cloud.com/Solutions/jCUDA/
[52] http://mathema.tician.de/software/pycuda
[53] http://www.hoopoe-cloud.com/Solutions/CUDA.NET/
[54] Bezeichnet Code, der nur von der Grafikkarte ausgeführt werden kann.

Kompilieren wird nur der Device Code von `nvcc` in die Zwischensprache PTX[55] für die Grafikkarte übersetzt. CUDA nutzt an dieser Stelle ebenfalls eine Zwischensprache, ähnlich dem CAL, um möglichst viele GPU-Generationen unterstützen und Optimierungen entsprechend der verwendeten Generation durchführen zu können. [Nvi091] Der übrige Code wird vom vorhandenen System-Compiler für die CPU kompiliert.

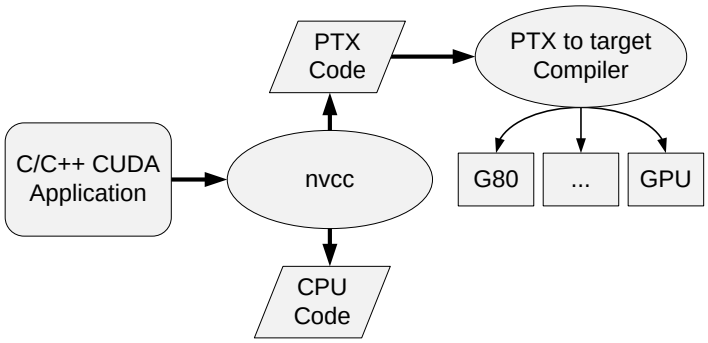



**Abbildung 4-6: Vorgehensweise bei der CUDA Kompilierung [Nvi08]**

Bevor anhand von Beispielen gezeigt wird, wie Daten zwischen Host und GPU kopiert, Kernel geschrieben und anschließend ausgeführt werden, muss zuerst auf die Art und Weise eingegangen werden, wie Threads in CUDA organisiert sind.

### *Threadorganisation*

Threads werden in CUDA in mehreren Stufen organisiert und auf die Systemressourcen abgebildet. Die oberste Ebene bilden die sogenannten *Grids*. Ein Grid beinhaltet mehrere Thread-Blöcke *(Blocks)* und kann sowohl ein- als auch zweidimensional sein. Die Dimensionsgrößen können zwischen 1 und 65.536 liegen. Jeder Thread-Block verfügt über eine bestimmte Anzahl an Threads, die jedoch 512 nicht überschreiten darf (1024 bei Grafikkarten basierend auf der Fermi-Architektur). Die Threads können in maximal drei Dimensionen organisiert werden, wodurch sich Operationen auf Vektoren, Matrizen oder Körpern effizient abbilden lassen. [Nvi101] Abbildung 4-7 soll die Struktur veranschaulichen, in der sowohl das Grid als auch die Thread-Blöcke zweidimensional sind.

[55] Parallel Thread Execution

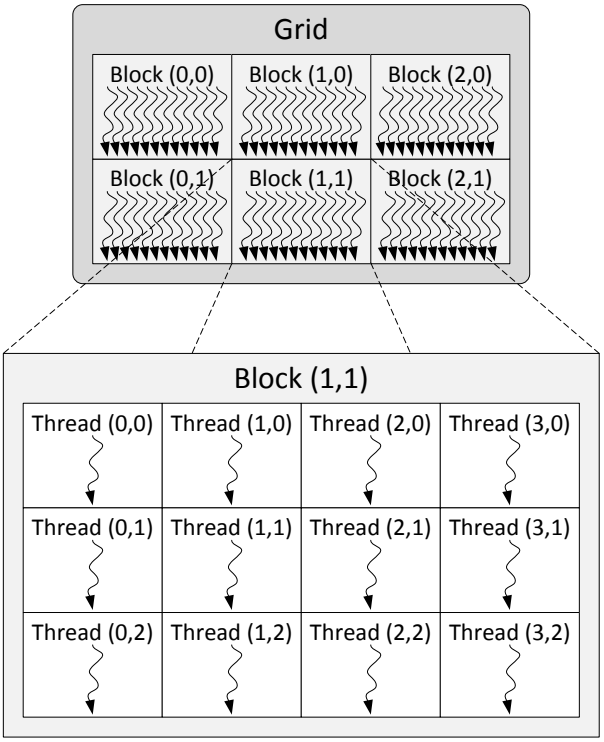



**Abbildung 4-7: Threadorganisation in CUDA [Nvi101]**

Innerhalb eines Kernels stellt die Laufzeitumgebung die Variablen `threadIdx`, `blockIdx` und `blockDim` bereit, mit denen die jeweilige Position des Threads im Grid und im Thread-Block bestimmt werden kann. Diese Informationen können genutzt werden, um bspw. die zu bearbeitenden Elemente einer Matrix zu bestimmen. Der folgende Code stellt die allgemeine Vorgehensweise dar, mit der sich die zu lesende Spalte und Zeile einer zweidimensionalen Matrix errechnen lassen [Nvi101]:

```
// Zugehörige Zeile
int row = blockIdx.y * blockDim.y + threadIdx.y;
// Zugehörige Spalte
int col = blockIdx.x * blockDim.x + threadIdx.x;
```

### *Kernel*

Die auf der Grafikkarte auszuführenden Berechnungen werden in Kernel-Funktionen definiert, die das Host-Programm mittels bestimmter Methoden anstoßen kann. In CUDA wird eine Kernel-Prozedur mit `__global__` gekennzeichnet. Zusätzlich gibt es noch Prozeduren oder Funktionen, die nur von einer Kernel-Funktion aufgerufen werden können. Die Kennzeichnung hierfür ist `__device__`. Der in den Methoden gekapselte Code wird bei beiden

Varianten als *Device Code* bezeichnet, da er nur von der Grafikkarte ausgeführt werden kann. Analog dazu wird der Code, der nur von der CPU ausgeführt wird, als *Host Code* bezeichnet. [Nvi101]

Für den Device Code gibt es zusätzliche Restriktionen, die es bei der Programmierung zu beachten gilt. So dürfen z.B. keine Rekursionen verwendet werden, die Deklaration statischer Variablen wird nicht unterstützt und die Anzahl an Parametern, die einer solchen Methode übergeben wird, darf nicht variabel sein. Neben diesen wichtigsten Einschränkungen kann eine detaillierte Aufzählung aller Limitierungen in [Nvi101] eigesehen werden.

Um einen Kernel vom Host-Programm aufrufen zu können, muss der Name der Prozedur, gefolgt von Angaben zur Grid- und Blockanzahl und den eigentlichen Parametern für die Prozedur, angegeben werden:

```
Kernel_Prozedur<<<GRID, THREADS_PER_BLOCK>>>(...);
```

Sollen sowohl die Grid- als auch die Thread-Blöcke eindimensional sein, können für die Platzhalter direkt die gewünschten Werte angegeben werden. Ist jedoch mehr als eine Dimension erforderlich, wird die Größe unter Verwendung des integrierten Vektor-Typs `dim3` spezifiziert. Fehlt die Angabe einer Dimension, so wird sie mit 1 initialisiert.

```
// Maximal 512 Threads pro Block, 8 * 8 * 8 = 512
dim3 threadsPerBlock(8, 8, 8);
// Angabe der Grids in Abhängigkeit zu den benötigten Threads
dim3 blocksPerGrid(64, 64);
// Ergibt: 64 * 64 * 512 = 2097152 Threads
```

Die Abarbeitung des Kernels geschieht asynchron, so dass das Host-Programm direkt mit den nachfolgenden Anweisungen fortfahren kann. Die Synchronisation erfolgt implizit, wenn Variablen zwischen Host und Grafikkarte kopiert oder wenn der Befehl `cudaThreadSynchronize()` genutzt wird. [Nvi102] Zur Synchronisation innerhalb einer Kernel-Prozedur kann der Befehl `__syncthreads` genutzt werden. Die Synchronisation erfolgt allerdings nur für die Threads des jeweiligen Thread-Blocks. Eine globale Synchronisation aller Threads ist innerhalb einer Kernel-Prozedur nicht möglich. [Nvi101]

### *Speicherverwaltung*

Analog zur Differenzierung zwischen Host und Device Code kann auch zwischen Host und Device Memory unterschieden werden. Device Code kann nur auf Device Memory zurückgreifen, so dass benötigter Speicher zuvor vom Host-Programm in der Grafikkarte allokiert und bei vorhandenen Daten auch kopiert werden muss. Hierbei gilt es, die geringe Bandbreite zu berücksichtigen, die zwischen Host und Device zur Verfügung steht und in aktuellen Systemen mit PCIe 2.0 und einer 16x-Anbindung 8 GB/s in eine Richtung beträgt. [PCI10]

Die Allokation von globalem Speicher bzw. die Freigabe geschieht durch die zwei Methoden `cudaMalloc()` und `cudaFree()`, die in ihrer Funktionsweise der `malloc()`-Funktion von C/C++ nachempfunden wurden. Um Speicher zwischen Host und Device Memory kopieren zu können, gibt es die Funktion `cudaMemcpy()`, die insgesamt vier Parameter erwartet. Die ersten zwei Parameter spezifizieren die Ziel- und Quellvariablen, der dritte Parameter gibt die Größe des zu kopierenden Speicherbereiches an, und mit dem vierten Parameter wird festgelegt, in welche Richtung der Kopiervorgang erfolgen soll: Vom Host zur Grafikkarte oder von der Grafikkarte zum Host. Das folgende Beispiel soll die Verwendung der oben beschriebenen Methoden demonstrieren [Nvi101]:

```
int n = 1024;
int size = n * sizeof(float);
// Speicher im Host allokieren
float *h_mem = (float*)malloc(size);
float *h_result_mem = (float*)malloc(size);
// Speicher in der Grafikkarte allokieren
float *d_mem, *d_result_mem;
cudaMalloc((void**)&d_mem, size);
cudaMalloc((void**)&d_result_mem, size);
// Speicher kopieren (Ziel, Quelle, Größe, Richtung)
cudaMemcpy(d_mem, h_mem, size, cudaMemcpyHostToDevice);
// Kernel ausführen...
kernel_procedure<<<2, 512>>>(d_mem, d_result_mem);
// Ergebnisse zurückkopieren
cudaMemcpy(h_result_mem, d_result_mem, size, cudaMemcpyDeviceToHost);
// Speicher freigeben
cudaFree(d_mem); cudaFree(d_result_mem);
```

Der Zugriff auf den globalen Grafikkartenspeicher lässt sich durch die Verwendung von Shared Memory noch weiter beschleunigen. Shared Memory ist ein kleiner Speicherbereich in der Größenordnung von 16 bis 48 KB, der allen Threads eines Thread-Blocks als gemeinsamer Speicher bereitsteht. Innerhalb des Device Codes wird er durch das Voranstellen von `__shared__` bei einer Variablendeklaration definiert. [Nvi101]

#### *Zusammenfassung*

Obwohl ATI und Nvidia ähnliche Konzepte zur Programmierung von Grafikkarten zur Verfügung stellen, erfuhr nur CUDA eine breite Akzeptanz bei den Entwicklern. Viele Anwendungen im Videobereich kommen bereits mit Unterstützung für CUDA, wie z.B. die Creative Suite 5 von Adobe. [Gau10] Ein Grund für den Erfolg von CUDA ist der geringe Aufwand und die Einfachheit, mit der Anwendungen für Grafikkarten erstellt werden können. Die Anlehnung an C ermöglicht eine schnelle Einarbeitung, zu der auch die ausführliche Dokumentation von Nvidia beiträgt und auf die bereits referenziert wurde. Gedruckte Literatur ist in geringem Umfang unter [Kir10] und [San10] zu finden. Mit dem CUDA Debugger und Profiler sowie Parallel Nsight[56] stehen den Entwicklern weitere Hilfsmittel zur Verfügung, mit denen CUDA-Anwendungen auf Fehler und Performance-Engpässe hin untersucht werden können.

Die Ebenen der Parallelität umfassen sowohl die Daten- als auch die Taskparallelität, wobei der Fokus durch die sehr hohe Anzahl an Recheneinheiten und der Beschränkungen hinsichtlich unterschiedlicher Threads innerhalb eines Warps[57] auf massive Datenparallelität gelegt wurde. Der Abstraktionsgrad ist geringer als bei anderen Modellen, da nur die Verwaltung der Threads und die Abbildung auf die Systemressourcen implizit erfolgen. Die Angabe der Parallelität geschieht explizit durch die Kernel sowie die Zerlegung der Aufgaben und die Synchronisation (teilweise auch implizit). Durch das Konzept des Host und Device Memorys muss die Kommunikation ebenfalls explizit erfolgen.

### 4.2.3 Open Computing Language (OpenCL)

Die bisherigen Ansätze zur parallelen Programmierung auf Grafikprozessoren können zwar einen Nutzen aus der sehr hohen Leistungsfähigkeit dieser Geräte ziehen, sind aber immer

[56] http://developer.nvidia.com/object/nsight.html
[57] Als Warp bezeichnet Nvidia zu einer Gruppe zusammengefasste Threads, siehe S. 10.

auf einen Hersteller beschränkt und somit abhängig von der verwendeten Hardware[58]. Aus diesem Grund hat Apple in Zusammenarbeit mit Firmen wie AMD, Intel und Nvidia, den OpenCL Standard entworfen, der die Programmierung heterogener Hardware (CPUs, GPUs, DSPs, Cell) unter einem einheitlichen Programmiermodell ermöglichen soll. [Khr09]

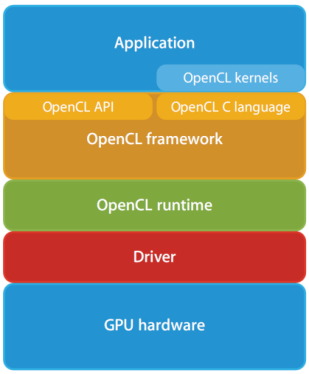



**Abbildung 4-8: OpenCL Architektur [App093]**

Damit OpenCL auch von vielen Herstellern genutzt werden kann, hat Apple den Standard der Khronos Group[59] übergeben, die ihn Ende 2008 in der Version 1.0 als frei zugänglichen Standard veröffentlichte. [Gol08] Bei der Definition des Standards hat man sich nicht nur auf die Spezifikation einer Sprache beschränkt, sondern wollte ein komplettes Framework bestehend aus einer Sprache, einer API, Bibliotheken und einer Laufzeitumgebung schaffen. [Khr09] Die OpenCL Spezifikation lässt sich grob in drei Teile gliedern, die sich auch in Abbildung 4-8 wiederfinden [AMD101]:

- OpenCL C Programming Language

Innerhalb einer Anwendung werden, wie auch bei CUDA, Kernel genutzt, um die parallel auszuführende Arbeit zu spezifizieren. Die Kernel werden in OpenCL in einer modifizierten C-Variante geschrieben, die auf dem C99 Standard aufbaut und ihn bspw. um Datentypen für Vektoren und Funktionen zur Threadsynchronisation erweitert und in Bereichen der Rekursion, statischer Variablen und Funktionszeigern einschränkt. [Khr09]

[58] ATI Stream SDK 1.x nur verwendbar mit ATI-Grafikkarten; CUDA nur mit Nvidia-Grafikkarten.
[59] http://www.khronos.org/opencl/

- OpenCL Platform Layer

Das Host Programm verwendet die OpenCL API, um die auf dem jeweiligen System verfügbaren, OpenCL-fähigen, Geräte abzufragen, auf ihnen Speicher zu allokieren und die Kernel-Prozeduren zu konfigurieren und zu starten. [AMD101]

- OpenCL Runtime

Mit der OpenCL Runtime werden die Kernel auf der entsprechenden Hardware ausgeführt. Ein Unterschied zu CUDA ist, dass bei OpenCL die Kernel nicht direkt bei der Erzeugung des Programmes kompiliert werden[60], sondern erst bei der Ausführung für die verwendete Hardware *(online)*. [App093,Khr09] Diese Vorgehensweise ist nötig, um weitestgehend unabhängig von der Hardware zu bleiben.

Damit OpenCL in einer C/C++-Anwendung verwendet werden kann, müssen die benötigten Header-Dateien und Bibliotheken auf dem Zielsystem vorhanden sein. Bei der Installation des ATI Stream SDKs 2.x bzw. des CUDA Toolkits werden die Dateien automatisch installiert. Alternativ können die Header-Dateien auch von der Khronos Group[61] bezogen werden. Die für das Linken benötigte *OpenCL.lib* liegt entweder dem Grafikkarten-Treiber bei oder ist in den Paketen von ATI bzw. Nvidia enthalten. Die OpenCL-Anwendung kann anschließend mit einem gewöhnlichen Compiler kompiliert werden. Für höherwertige Sprachen existieren ebenfalls Bibliotheken, die eine Verwendung von OpenCL ermöglichen. Beispiele dafür sind JOCL[62] für Java sowie OpenTK[63] und OpenCL.NET[64] für .NET-basierte Anwendungen.

### *Threadorganisation*

Threads werden in OpenCL als *Work-Items* bezeichnet und unterliegen ebenfalls einer Hierarchie. OpenCL definiert dafür *Work-Groups*, die den Thread-Blocks aus CUDA gleichen und ebenfalls in einer, zwei oder drei Dimensionen Work-Items beinhalten können. Anstelle eines Grids gibt es den Index-Bereich `NDRange`, wobei N für die Dimension steht. Anders als bei CUDA kann `NDRange` auch dreidimensional sein. [Khr09]

[60] Unter Angabe der Zielhardware ist dies aber ebenfalls möglich und wird als *offline* bezeichnet.
[61] http://www.khronos.org/registry/cl/
[62] http://www.jocl.org/
[63] http://www.opentk.com/
[64] http://www.hoopoe-cloud.com/Solutions/OpenCL.NET/

Damit verschiedene Work-Items voneinander unterschieden werden können, wird ihnen eine lokale ID zugewiesen, die innerhalb einer Work-Group gültig ist. Zusätzlich gibt es eine global gültige ID, die sich aus der lokalen ID, der Work-Group-ID sowie der Größe der Work-Group zusammensetzt. Dies erspart das in CUDA notwendige Errechnen des globalen Index mit Hilfe von `threadIdx`, `blockIdx` und `blockDim`. [Nvi092] Die Methoden sind folgendermaßen definiert [Khr09]:

```
// Bestimmt die lokal-gültige ID für die angegebene Dimension
size_t get_local_id (uint dimindx)
// Bestimmt die Work-Group, in der sich das Work-Item befindet
size_t get_group_id (uint dimindx)
// Bestimmt die global-gültige ID für die angegebene Dimension
size_t get_global_id (uint dimindx)
```

### *Ausführungskontext*

Der erste Schritt zur Ausführung eines OpenCL Kernels besteht in der Erstellung eines Ausführungskontextes. Bei der Erzeugung muss das gewünschte Zielgerät (Device) angegeben werden, wie z.B. die CPU, GPU, spezielle OpenCL-Beschleuniger (z.B. Cell) oder alle verfügbaren OpenCL-fähigen Geräte des Systems. [Khr09]

```
cl_platform_id platform;
cl_device_id device;
cl_context context;
// Liefert eine OpenCL Plattform
clGetPlatformIDs(1, &platform, NULL);
// Liefert ein OpenCL-Device vom Typ GPU zurück
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
// Erstellt einen Kontext für ein Device, ohne weitere Eigenschaften
context = clCreateContext(0, 1, &device, NULL, NULL, 0);
```

Nachdem der Kontext erstellt wurde, ist eine `CommandQueue` zu erzeugen. Sämtliche Operationen wie Kopieroptionen oder Kernelprozeduren werden der `CommandQueue` hinzugefügt und anschließend ausgeführt. Pro Device können mehrere Warteschlangen existieren. [Khr09]

```
cl_command_queue queue;
queue = clCreateCommandQueue(context, device, 0, NULL);
```

### *Speicherverwaltung*

Der nächste Schritt besteht darin, den benötigten Speicher auf dem OpenCL-Gerät zu allokieren und ggf. vorhandene Daten zu kopieren. Bei CUDA wurden die bereits aus C/C++ bekannten Funktionen zur Speicherverwaltung nachgebildet, wohingegen OpenCL auf *Buffer Objects* setzt. [Nvi092]

```
// Host-Variablen h_mem, h_result_mem und size anlegen ...
cl_mem d_mem, d_result_mem;
// Speicher allokieren, aus dem nur gelesen wird.
    Inhalt der Host-Variablen h_mem wird nach d_mem kopiert
d_mem = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, size, h_mem, 0);
// Speicher allokieren, der nur geschrieben wird.
d_result_mem = clCreateBuffer(context, CL_MEM_WRITE_ONLY, size, 0, 0)
```

Bei der Erzeugung eines Buffer-Objektes kann spezifiziert werden, ob der Zugriff nur lesend, nur schreibend oder sowohl lesend als auch schreiben erfolgen darf. Zusätzlich lässt sich ein Verweis auf eine Host-Variable angeben, deren Inhalte automatisch in den Speicherbereich auf dem OpenCL-Gerät übertragen werden. [Khr09] OpenCL ermöglicht dies mit nur einer Anweisung anstelle von zwei wie bei CUDA.

Insgesamt unterstützt OpenCL vier verschiedene Arten von Speicherbereichen. Diese sind: globaler, konstanter, lokaler und privater Speicher. Globaler und konstanter Speicher stehen immer allen Work-Items aus allen Work-Groups zur Verfügung; lokaler Speicher entspricht dem Shared Memory aus CUDA und privater Speicher kann innerhalb der Work-Items unabhängig von anderen genutzt werden. [Khr09]

### *Kernel erzeugen*

Kernel-Funktionen sind in OpenCL nicht mit `__global__`, sondern mit `__kernel` gekennzeichnet. Auf eine zusätzliche Kennzeichnung mit `__device__` für Funktionen, die nur vom Device Code aufgerufen werden können, wurde verzichtet. Dafür können ebenfalls die mit `__kernel` markierten Funktionen genutzt werden. [Khr09]

Einen weiteren Unterschied gibt es bei der Angabe der Funktionen. Während bei CUDA auf im Quellcode vorhandene und mit `__global__` gekennzeichnete Funktionen verwiesen

werden kann, erwartet OpenCL die Übergabe in Form einer Zeichenfolge (online) mittels `clCreateProgramWithSource()` oder offline als vorkompilierte Binärdatei mit `clCreateProgramWithBinary()`. Idealerweise werden die beiden Methoden kombiniert, so dass ein Kernel auf einem System beim ersten Aufruf kompiliert und bei späteren direkt in vorkompilierter Form genutzt werden kann. [Khr09]

Mit den folgenden Anweisungen wird ein, in der Zeichenfolge `program` definierter Kernel erstellt und seine Argumente belegt [Nvi092]:

```
cl_program prog;
cl_kernel kernel;
// Programm aus der Zeichenfolge program erzeugen
prog = clCreateProgramWithSource(context, 1,
    (const char**)&program, NULL, NULL);
// Programm kompilieren
clBuildProgram(prog, 0, NULL, NULL, NULL, NULL);
// Kernel erstellen
kernel = clCreateKernel(prog, "Kernel_Name", NULL);
// Argumente belegen
clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_mem);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_result_mem);
```

### *Kernel ausführen*

Der letzte Schritt zur Ausführung eines OpenCL-Programmes besteht darin, die Dimensionen und Größen der Work-Groups festzulegen und damit den Kernel zu starten. Da die zulässigen Größen je nach Hardware variieren können, stellt OpenCL die maximalen Werte in den Konstanten `CL_DEVICE_MAX_WORK_ITEM_SIZES` und `CL_DEVICE_MAX_WORK_GROUP_SIZE` bereit. [Khr09] Alternativ kann für die Größe der Work Groups auch NULL übergeben werden und OpenCL berechnet selbst eine geeignete Dimensionierung.

```
// Anzahl und Größe der Work-Groups definieren
size_t globalWorkSize[] = { ELEMENTS, ELEMENTS };
size_t localWorkSize [] = { ELEMENTS / BLOCK_SIZE,
    ELEMENTS / BLOCK_SIZE }; // Alternativ mit NULL übergeben
// Kernel ausführen
clEnqueueNDRangeKernel(cmd_queue, kernel, 2, NULL, globalWorkSize,
    localWorkSize, 0, 0, 0);
// Ergebnisse zum Host kopieren
```

```
clEnqueueReadBuffer(cmd_queue, d_result_mem, CL_TRUE, 0, size,
    h_result_mem, 0, 0, 0);
// Device Memory freigeben
clReleaseMemObject(d_mem); clReleaseMemObject(d_result_mem);
```

Die letzten Anweisungen kopieren die Ergebnisse zurück in den Host-Speicher und geben den Grafikkartenspeicher wieder frei.

#### *Zusammenfassung*

Abschließend betrachtet, ist das stärkste Argument zur Nutzung von OpenCL die Hardwareunabhängigkeit. OpenCL kann auf allen Grafikkarten eingesetzt werden, für die entsprechende Unterstützung durch den Hersteller vorliegt. Sowohl ATI als auch Nvidia bieten für ihre aktuellen Modelle OpenCL Unterstützung an. Die gewonnene Unabhängigkeit macht sich jedoch durch einen höheren Programmieraufwand bemerkbar, der bei CUDA in Verbindung mit der C Runtime for CUDA um ein Vielfaches geringer ausfällt. Da es keinen gesonderten Compiler für OpenCL-Anwendungen gibt, sind Fehler schwerer zu lokalisieren und zu beheben. Mit dem gDEBugger CL[65] bietet die Firma Graphic Remedy eine kostenpflichte Sammlung an Entwicklungswerkzeugen bestehend aus einem Debugger, Profiler und eines Speicher-Visualisierers an.

Der Abstraktionsgrad und auch die Ebenen der Parallelität entsprechen denen von CUDA, wobei auch OpenCL den Schwerpunkt auf Datenparallelität legt. Für detailliertere Informationen zu OpenCL sei auf die Spezifikation [Khr09], auf die Seiten von ATI[66] und Nvidia[67] oder auf [Kir10] verwiesen.

### 4.2.4 Direct Compute

Direct Compute ist eine Programmierschnittstelle von Microsoft, mit der die Shader Cores aktueller Grafikkarten ebenfalls für allgemeinere Berechnungen, ähnlich CUDA und OpenCL, verwendet werden können. Um Direct Compute zu nutzen, wird DirectX (DX) in der Version 11 benötigt. DirectX 11 steht aktuell nur für Windows Vista und Windows 7 zur Verfügung. Weiterhin wird eine Grafikkarte mit Unterstützung für das Shader Model 5.0 benötigt (DX

[65] http://www.gremedy.com/gDEBuggerCL.php
[66] http://developer.amd.com/gpu/ATIStreamSDK/
[67] http://developer.nvidia.com/object/opencl.html

11). Grafikkarten mit Unterstützung für das Shader Model 4.x (DX 10.x) lassen sich mit gewissen Beschränkungen ebenfalls für Direct Compute einsetzen. [Thi09]

### *Vorgehensweise*

Damit die Compute Shader in einer C/C++ Anwendung zur Berechnung verwendet werden können, muss das aktuelle DirectX SDK von Microsoft auf dem System vorhanden sein; ferner sind Verweise auf diverse Header-Dateien, wie z.B. `D3DX11.h`, zu tätigen. Der benötigte Aufwand zur Initialisierung der Grafikkarte und zur Erzeugung des Kontextes ist umfangreicher als bei den bisher kennengelernten Ansätzen. Dadurch ist der resultierende Quelltext, im Vergleich zu OpenCL, größer und weniger gut lesbar. Aus diesem Grund soll an dieser Stelle auf eine detaillierte Beschreibung der einzelnen Schritte unter C/C++ verzichtet werden. Beispiele zur Verwendung bieten [Nvi093], [Ope10] und [gam08] sowie Beispielanwendungen im Nvidia Toolkit [Nvi10].

Die nötigen Schritte zur Nutzung von Direct Compute sind im Folgenden prozessartig zusammengefasst [Ope10]:

1. Gerät und Kontext initialisieren
2. Shader Code aus Datei laden und kompilieren
3. Speicher für den Compute Shader bereitstellen
4. Compute Shader ausführen
5. Ergebnisse auslesen

### *Gerät und Kontext initialisieren*

Der erste Schritt zur Erzeugung und Ausführung eines Compute Shaders ist die Initialisierung des Gerätes und des Kontextes. Bei der Erzeugung wird das gewünschte Feature Level angegeben (bspw. `D3D_FEATURE_LEVEL_11_0` für DX 11- oder `D3D_FEATURE_LEVEL_10_0` für DX 10.0-Hardware), das von der Hardware auch unterstützt werden muss. [Ope10]

### *Shader Code aus Datei laden und kompilieren*

Shader Code wird unter DirectX in der High Level Shading Language (HLSL) geschrieben, die Ähnlichkeiten zu C aufweist. Ursprünglich wurde die HLSL dafür genutzt, um Vertex-, Pixel-

und Geometrie-Shader für die Grafikdarstellung zu programmieren. Mit DirectX 11 kommen die Compute Shader hinzu, die die Ausführung von allgemeinen Berechnungen ermöglichen. Innerhalb eines Compute Shaders wird angegeben, wie viele Threads einer Gruppe den Shader abarbeiten sollen. [Mic1012,Nvi093]

```
[numthreads(x, y, z)]
```

Für die Angabe der Threads müssen die Gleichungen $x * y * z \leq 1024$ und $z \leq 64$ gelten. Diese Angaben entsprechen der Größe eines Thread-Blocks in CUDA bzw. einer Work-Group in OpenCL. Innerhalb des Shader Codes lässt sich der Thread über die folgenden Vektoren identifizieren: `SV_GroupID`, `SV_GroupThreadID` und `SV_DispatchThreadID`. [Thi09]

### *Speicher für den Compute Shader bereitstellen*

Speicher wird durch *Structured Buffers* bereitgestellt, die von allen Threads gelesen und geschrieben werden können. Um einen Buffer zu erzeugen, verfügt das Objekt des DirectX-Geräts über die Funktion `CreateBuffer`, die wie folgt aufgerufen wird:

```
dx11_device->CreateBuffer(const D3D11_BUFFER_DESC *pDesc, const
    D3D11_SUBRESOURCE_DATA *pInitialData, ID3D11Buffer, **ppBuffer)
```

Der erste Parameter ist ein Beschreibungs-Objekt für den zu erzeugenden Buffer, der Angaben zur Größe sowie Lese- und Schreib-Berechtigungen enthält. Über die Zugriffsrechte lässt sich ebenfalls festlegen, ob die Buffer-Objekte auch von der CPU gelesen oder geschrieben werden können. Der zweite Parameter enthält einen Verweis auf den in das Buffer-Objekt zu kopierenden Speicherbereich (alternativ NULL, dann wird kein Speicher kopiert). Der letzte Parameter dient als Referenz auf das Objekt. [San09]

Analog zu CUDA und OpenCL gibt es in Direct Compute die Möglichkeit, einen gruppenlokalen Speicher zu erstellen, den alle Threads einer Gruppe manipulieren können. Der *Thread Local Storage* ist in seiner Größe auf maximal 32 KB beschränkt. [Thi09]

#### *Compute Shader ausführen*

Compute Shader werden von einem DirectX-fähigen Gerät mit der Methode `Dispatch()` gestartet. Beim Aufruf wird zudem die endgültige Anzahl an Threads bestimmt:

```
dx11_device->Dispatch(nX, nY, nZ);
```

Die Parameter `nX`, `nY` und `nZ` legen die Anzahl der Work-Groups in bis zu drei Dimensionen fest, mit denen der Compute Shader ausgeführt wird. [Thi09] Die Größe der einzelnen Work-Groups wurde zuvor durch die Anweisung `numthreads` bestimmt.

#### *Ergebnisse auslesen*

Das Kopieren der Ergebnisse erfolgt mit der Funktion `Map`, die beim Erzeugen des Kontextes im Kontext-Objekt zur Verfügung steht. `Map` erwartet u.a. ein Objekt vom Typ `D3D11_MAPPED_SUBRESOURCE` sowie ein Buffer-Objekt mit CPU-Zugriffsrechten, aus dem die Ergebnisse in das `Subresource`-Objekt übertragen werden. Das `Subresource`-Objekt lässt sich anschließend im Host-Programm wie eine normale Variable verwenden. [San09]

#### *Zusammenfassung*

Direct Compute erweitert die bereits in der Spiele-Entwicklung massiv genutzte DirectX-API um Fähigkeiten für GPGPU, ohne eine neue API dafür bereitstellen zu müssen [Nvi093]. Jedoch ist der Aufwand unter C/C++ zur Erzeugung von Compute Shadern im Vergleich zu anderen Konzepten unverhältnismäßig hoch. Entwickler, die bereits mit der DirectX-API vertraut sind, mag der Einstieg womöglich etwas leichter fallen. Andernfalls erschwert der geringe Umfang an Dokumentation seitens Microsoft den Einstieg. Technisch gesehen ist Direct Compute für datenparallele Aufgaben ausgelegt. [Boy08] Der Abstraktionsgrad ist identisch zu CUDA oder OpenCL, obwohl in Direct Compute generell mehr Code benötigt wird.

### 4.2.5 PGI Accelerator

Anhand der vorangegangenen Abschnitte wurde deutlich, dass die Ausnutzung der parallelen Rechenleistung aktueller GPUs mit einigem Aufwand verbunden ist. Sowohl bei CUDA als auch bei OpenCL ist eine Auslagerung der zu parallelisierenden Routinen in Kernel notwen-

dig und die benötigten Daten müssen explizit vom Programmierer zwischen Host und GPU transferiert werden. Bei OpenCL ist zudem eine Initialisierung des zu verwendenden Gerätes erforderlich.

Die Portland Group[68] bietet mit dem PGI Accelerator einen Compiler für C99 und FORTRAN an, der mittels ähnlicher Direktiven wie OpenMP den gekennzeichneten Quellcode automatisch auf eine CUDA-fähige Grafikkarte auslagert (sofern möglich) und eigenständig die benötigten Variablen zwischen Host und Device kopiert. Der PGI Accelerator Compiler steht für Linux, Mac OS X und Windows zur Verfügung. Die Verwendung unterliegt allerdings einem Lizenzmodell und ist nicht kostenlos[69], weshalb an dieser Stelle nur in verkürzter Form auf diesen Ansatz eingegangen wird. [TPG10]

Nach der Installation des PGI Accelerator Compilers stehen auf dem Zielsystem der `pgcc`-Compiler für C und der `pgfortran`-Compiler für FORTRAN bereit, mit denen CUDA-beschleunigte Programme kompiliert werden müssen. Die OpenMP-ähnlichen Direktiven haben in C die folgende Syntax [TPG101]:

```
#pragma acc directive-name [clause [,clause]…]
```

Die wohl wichtigste Direktive ist die der *Region*, mit der sich Schleifen auf die Grafikkarte auslagern lassen [Wol09]:

```
    // Variablen erzeugen
    int i, n = 100000;
    float *restrict src = (float*)malloc(n * sizeof(float));
    float *restrict dst = (float*)malloc(n * sizeof(float));
    // Array src befüllen ...
    // Code auf der Grafikkarte ausführen
#pragma acc region
    {
          for(i = 0; i < n; i++)
                dst[i] = src[i] * 2.0f;
    }
```

[68] http://www.pgroup.com
[69] Es kann eine zweiwöchige Testlizenz nach Registrierung kostenlos bezogen werden.

Anschließend muss das Programm mit dem PGI C-Compiler durch Hinzufügen des Arguments `-ta=nvidia` kompiliert werden. Zusätzliche Informationen liefert der Compiler durch Angabe des Parameters `-Minfo`.

Beim Kompilieren erstellt der Compiler den Kernel für die Region und sorgt dafür, dass alle benötigten Variablen (`src` und `dst`) im Speicher der Grafikkarte allokiert bzw. kopiert (`src`) und die Ergebnisse (`dst`) am Ende der Ausführung wieder zurück zum Host übertragen werden. [Wol09] Ähnlich wie bei CUDA und OpenCL unterliegen die Regions bestimmten Beschränkungen. Sie dürfen bspw. nicht verschachtelt werden, maximal eine `if`-Anweisung enthalten und keine Methoden außerhalb der Region ansteuern. Das `restrict`-Schlüsselwort dient bei der Variablendeklaration dazu, unabhängige Speicherbereiche zu kennzeichnen. Dies ist für den Compiler bei der Analyse zu parallelisierender Regionen von Bedeutung. [TPG101]

Bei der Definition der Region kann der Programmierer über optionale Parameter manuellen Einfluss auf die zu kopierenden Variablen oder die Abbildung der Schleife auf die Grafikkarte nehmen. Eine detaillierte Beschreibung aller möglichen Optionen ist unter [TPG101] sowie weitere Literatur- und Informationsquellen unter [TPG10] zu finden.

### *Zusammenfassung*

Der PGI Accelerator Compiler ist ein vielversprechendes Konzept für den vereinfachten Umgang mit einer programmierbaren Grafikkarte von Nvidia, welcher den Schwerpunkt bei der Programmierung wieder auf das eigentlich zu lösende Problem legt. Aktuell können nur datenparallele Anweisungen in Form von Schleifen von dem PGI Compiler auf die Grafikkarte ausgelagert werden. Taskparallelität ist somit noch nicht möglich. Durch den höheren Anteil der Automatisierung fällt der Abstraktionsgrad beim PGI Compiler, im Gegensatz zu CUDA oder OpenCL, entsprechend höher aus. Aspekte der Kommunikation, Synchronisation und Zerlegung der Aufgaben erfolgen implizit und müssen nicht vom Programmierer bewerkstelligt werden. Der Umfang an bereitstehender Literatur ist zurzeit noch begrenzt und beschränkt sich auf eine Spezifikation und auf kleinere Einführungen, die auf den Seiten des Herstellers unter [TPG10] zu finden sind.

## 4.3 Ansätze für Systeme mit verteiltem gemeinsamen Speicher

### 4.3.1 Unified Parallel C

Unified Parallel C (UPC) ist eine Erweiterung des C99-Standards und eignet sich insbesondere, aufgrund der hohen Kontrolle an Lokalität, für Systeme mit einem verteilten gemeinsamen Speicher. Die erste Veröffentlichung von UPC erfolgte in der Version 0.9 im Jahr 1999 als technischer Report. Mittlerweile liegt UPC in Version 1.2 vor, an deren Entwicklung sich Unternehmen (Cray, HP, IBM u.a.), Behörden (die U.S. Energiebehörde u.a.) und Universitäten (George Washington University, University of California at Berkeley u.a.) beteiligen. [UPC05,UPC06]

Für UPC existieren sowohl kommerzielle als auch quelloffene und damit kostenfreie Compiler. Auch in Bezug zu den unterstützten Plattformen wird mit Linux, Mac OS X und Windows ein breites Spektrum abgedeckt. Bspw. ist GCC UPC[70] für Linux und Mac OS X ein kostenfreier Compiler; für Windows stellt die University of California (Berkeley) geeignete Versionen bereit[71]. Zur Erzeugung von UPC-Anwendungen muss die Header-Datei `upc.h` eingebunden und die Quellcode-Datei anschließend mit dem UPC-Compiler kompiliert werden.

#### *Ausführungsmodell*

Parallelität ist in UPC bereits implizit durch das SPMD-Prinzip[72] vorgegeben, die jedoch in expliziter Weise im Programm ausgenutzt werden muss. [Rau07] Die Anzahl an Threads wird entweder beim Kompilieren oder beim Starten der Anwendung festgelegt, woraufhin alle Threads die `main()`-Funktion aufrufen. [Car99] Threads in UPC sind allerdings nicht direkt mit den Threads der bisherigen Ansätze vergleichbar, da sie sich nicht zwangsläufig auf einem System befinden müssen. Vielmehr können sie sich auch auf anderen physischen Systemen befinden, die über ein schnelles Kommunikationsnetz miteinander verbunden sind. Die Ausführung auf verteilten Systemen übernimmt die UPC Runtime, die dafür sorgt, dass die Anwendung auf den entfernten Knoten gestartet wird. [LBL10] Innerhalb des Programms

[70] http://www.gccupc.org
[71] http://upc.lbl.gov/download/binaries.shtml
[72] Single Program Multiple Data: Bezeichnet einen Ansatz, in dem ein Programm parallel auf mehreren Prozessoren ausgeführt wird. Die Prozessoren können durch ihre ID unterschieden werden und dadurch verschiedene Berechnungen ausführen.

stellt UPC die Variablen `THREADS` und `MYTHREAD` bereit, mit denen die gesamte Anzahl an Threads sowie der Index des jeweiligen Threads abgefragt werden kann. [Cha05]

### *Speichermodell*

UPC ist eine PGAS-Sprache, die die logische Sicht auf den gemeinsamen Speicher gemäß der Threads partitioniert. Jedem Thread stehen ein privater und ein gemeinsam genutzter Speicherbereich zur Verfügung. Der globale gemeinsame Speicher wird durch die einzelnen Speicherbereiche der Threads aufgespannt, wobei jeder Thread mit seinem Bereich assoziiert ist. [Cha05,Rau07] Zur Veranschaulichung soll Abbildung 4-9 dienen:

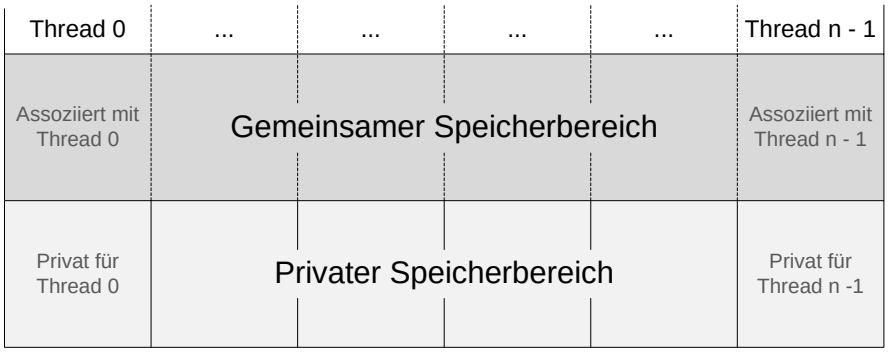



**Abbildung 4-9: UPC Speichermodell [Cha05]**

Damit Daten in dem gemeinsamen Speicherbereich allokiert werden können, führt UPC das Schlüsselwort `shared` ein. Eine mit `shared` gekennzeichnete (skalare) Variable wird immer in dem gemeinsamen Speicherbreich von Thread 0 allokiert. Mit `shared` gekennzeichnete Arrays verteilt UPC über den gesamten Speicherbereich. [Cha05]

```
shared int a; // Allokiert bei Thread 0
shared int b[THREADS]; // Pro Thread ein Element
```

Bei der Deklaration eines gemeinsamen Arrays wird standardmäßig eine Blockgröße von 1 angenommen. Das sorgt dafür, dass ein Thread im Abstand von `THREADS` Daten zugewiesen bekommt. Mit

```
shared [x] int c[x*THREADS];
```

kann der Programmierer die Blockgröße manuell beeinflussen. In dem vorangegangenen Beispiel erhält jeder Thread einen kontinuierlichen Block von `x` Elementen. [Cha05]

### *Datenparallelität*

Operationen auf gemeinsamen Daten lassen sich in UPC effizient mit der parallelen `upc_forall`-Schleife durchführen [Cha05]:

```
upc_forall(expression; expression; expression; affinity)
    statement;
```

Die ersten drei Parameter haben die gleiche Bedeutung wie bei einer normalen For-Schleife in C/C++. Der hinzugekommene vierte Parameter `affinity` bestimmt, welcher Thread die Iteration durchführen soll. Hier gibt es zwei verschiedene Möglichkeiten der Angabe [Cha05]:

- Angabe eines Integer-Wertes

Bei der Angabe eins Integer-Wertes wird pro Iteration implizit der Ausdruck: `MYTHREAD == (affinity % THREADS)` ausgewertet und der Schleifenkörper nur von dem Thread ausgeführt, der die Gleichung erfüllt.

- Angabe einer Speicheradresse

Bei Angabe einer Speicheradresse, wie bspw. `&a[i]`, wird die Iteration nur von dem Thread ausgeführt, der mit dieser Speicheradresse assoziiert ist bzw. dessen gemeinsamer Speicherbereich dieses Element beinhaltet.

```
shared int a[THREADS];
upc_forall(i = 0; i < THREADS; i++; i) // Alternativ auch &a[i] mögl.
    a[i] = ...;
```

### *Synchronisation*

Die Synchronisation der Threads geschieht in UPC nicht implizit (auch nicht nach einer parallelen Schleife). Für die explizite Synchronisierung stehen die folgenden Methoden zur Verfügung [UPC05]:

- `upc_barrier()` – müssen alle Threads erreichen; blockierend.

- `upc_notify()`, `upc_wait()` – notify und wait müssen von allen Threads aufgerufen werden; nur `upc_wait()` ist blockierend (2-Phasen Barrier).
- `upc_lock(upc_lock_t *lock)`, `upc_unlock(upc_lock_t *lock)`, `upc_lock_attempt(upc_lock_t *lock)` – Lock-Konzept.

Mit der Anweisung `upc_fence()` wird dafür gesorgt, dass alle Zugriffe auf gemeinsame Variablen abgeschlossen werden, bevor die Ausführung des Programms fortgesetzt wird. [UPC05]

### *Konsistenz*

UPC bietet Unterstützung für zwei verschiedene Konsistenzmodelle, die das Zugriffsverhalten auf den gemeinsamen Speicher beeinflussen [Cha05]:

- Strikte Konsistenz

Bei der strikten Konsistenz (`strict`) darf der Compiler mögliche Speicheranweisungen zugunsten einer effizienteren Ausführung nicht vertauschen. Alle Zugriffe auf den gemeinsamen Speicher erfolgen außerdem synchronisiert, so dass sie für andere Threads in der Reihenfolge sichtbar sind, in der sie ausgeführt wurden.

- Schwache Konsistenz

Bei der schwachen Konsistenz (`relaxed`) gibt es die oben genannten Restriktionen nicht, wodurch weniger Overhead erzeugt wird. Auch erfolgt keine Synchronisation der Speicherzugriffe zwischen den Threads, die der Programmierer jedoch bei Bedarf explizit durch `upc_fence()` durchführen kann.

Das zu nutzende Konsistenzmodell kann sehr granular auf insgesamt drei verschiedenen Ebenen festgelegt werden. Soll ein Modell für das ganze Programm gelten, erreicht man dies durch das Inkludieren der Header-Dateien `upc_strict.h` bzw. `upc_relaxed.h`. Eine feinere Festlegung kann mit der Direktive `#pragma upc strict/relaxed` erfolgen, deren folgender Block gemäß des spezifizierten Modells ausgeführt wird. Die feinste Steuerung betrifft die Variablen-Ebene, in der bei der Deklaration durch Voranstellen der Schlüsselworte `strict` oder `relaxed` das gewünschte Modell nur für diese Variable spezifiziert

wird. Eine feinere Festlegung wird ggü. einer gröberen vorranging behandelt. [Rau07] Welches Konsistenzmodell standardmäßig benutzt wird, ist von der jeweiligen UPC-Implementierung abhängig. [UPC05]

#### *Zusammenfassung*

UPC kombiniert die Vorteile von Ansätzen basierend auf verteiltem und gemeinsamem Speicher, ohne dass sich der Programmierer mit Kommunikationsaspekten beschäftigen muss. Durch den hohen Grad an Kontrolle, wie der Speicher für die verschiedenen Threads verteilt wird, kann die Lokalität bei DSM-Systemen auf Kosten eines höheren Programmieraufwandes gut ausgenutzt werden. Datenparallelität ist aufgrund des SPMD-Ansatzes generell gegeben und wird durch die in UPC enthaltene parallele For-Schleife weiter optimiert. Taskparallelität steht in dem Sinne nicht bereit und muss vom Programmierer über die Thread-Indizes eigenständig geschaffen werden.

Der grundlegende Umgang mit UPC lässt sich schnell erlernen. Dies liegt einerseits an der Integration in C auf Basis weniger Methoden und Schlüsselwörter und andererseits an dem großen Angebot an Literatur. Eine übersichtliche Liste weiterer Ressourcen findet sich unter [HPC10] und [HPC101]. Bzgl. des Abstraktionsgrades muss der Programmierer, neben einer Spezifikation der zu parallelisierenden Bereiche an bestimmten Stellen (z.B. bei parallelen Schleifen), für eine Verteilung der Daten sorgen und sich um die Synchronisation kümmern.

### 4.3.2 Co-Array Fortran

Co-Arrays stellen eine Erweiterung der Programmiersprache FORTRAN 95 dar und wurden im Jahr 1998 das erste Mal in [Num98] vorgestellt. Das Ziel bei der Erweiterung war es, FORTRAN mit möglichst wenigen Veränderungen zu einer parallelen Sprache zu erweitern. [Num98] Im Jahr 2008 wurden Co-Arrays in abgewandelter Form offiziell in FORTRAN 2008 übernommen. Ein quelloffener und zum Teil kostenloser Compiler mit Co-Array-Unterstützung ist der G95-Compiler[73] für Linux- und Windows-Systeme. Eine experimentelle und zur Zeit auf einen Knoten beschränkte Co-Array Implementierung existiert in der Version 4.6 des GCC FORTRAN Compilers, der für Linux, Mac OS X und Windows angeboten wird. [Fre10]

[73] http://www.g95.org

### *Ausführungsmodell*

Co-Array FORTRAN (CAF) fällt ebenfalls unter die Kategorie der SPMD-Programmiermodelle, die ein Programm auf mehreren Prozessoren bzw. Systemen ausführen. Möchte man ein CAF-Programm lediglich auf einem System mit mehreren Prozessoren bzw. Prozessorkernen ausführen, kann es mit dem G95-Compiler und dem Parameter `--g95 image=x`, unter Angabe der gewünschten Anzahl an Images[74], kompiliert werden. Wird ein Cluster-System benutzt, dessen Knoten über ein Netzwerk miteinander verbunden sind, muss die Netzwerk-Version des G95-Compilers verwendet werden. Mit Hilfe der darin enthaltenen *Co-Array Console* lassen sich die Anwendungen auf die entfernten Systeme übertragen und dort ausführen. Die Netzwerk-Version des G95-Compilers ist allerdings auf maximal fünf Images beschränkt, so dass bei weiteren Images eine Lizenz benötigt wird. [TGP10]

Innerhalb einer CAF-Anwendung kann, ähnlich wie bei UPC, mit der Funktion `THIS_IMAGE()` auf den eigenen Index des Images und mit `NUM_IMAGES()` auf die gesamte Anzahl an Images zugegriffen werden. [Vau08]

### *Speichermodell*

CAF zählt ebenfalls zu den PGAS-Sprachen, bei denen die Lokalität von Daten Berücksichtigung findet. Anders als bei UPC erfolgt der Zugriff auf gemeinsame Daten nicht implizit[75], sondern durch die explizite Angabe des gewünschten Images. Der Vorteil dieser Variante ist eine höhere Transparenz für den Programmierer, der sofort sehen kann, wenn (kostenintensivere) Zugriffe auf entfernte Speicherbereiche erfolgen. Der etwas höhere Programmieraufwand ist jedoch von Nachteil. [Num98]

```
integer :: a[*] ! Erzeugt eine Variable auf jedem Image
integer, dimension(n)[*] :: b ! Erzeugt ein Array auf jedem Image
```

Co-Arrays werden, im Gegensatz zu normalen Arrays, durch eckige Klammern `[ ]` definiert und angesteuert. Die erste Anweisung sorgt dafür, dass auf jedem Image die Variable `a` erzeugt wird. Mit `a[IMAGE]` erfolgt der Zugriff auf die Variable des angegebenen Images.

[74] In CAF wird ein Thread als Image bezeichnet
[75] Bei UPC erfolgt der Zugriff auf verteilten Speicher ohne eine zusätzliche Kennzeichnung.

Eine Angabe ohne eckige Klammern ist ebenfalls möglich und bezeichnet immer die lokale Instanz. Die zweite Anweisung erzeugt ein n-dimensionales Array auf jedem Image. Anders als bei UPC wird das Array nicht auf alle Threads bzw. Images verteilt, sondern pro Image erzeugt. [Num98]

Die Co-Arrays aus dem vorangegangenen Beispiel hatten eine Dimension *(co-rank)* von 1, jedoch ist auch eine Definition mit mehreren Dimensionen möglich. Die Angabe und der Zugriff auf ein mehrdimensionales Co-Array geschehen analog zu den normalen Arrays in FORTRAN. [Vau08]

#### *Synchronisation*

Synchronisation muss in CAF ebenfalls vom Programmierer explizit durchgeführt werden. Dafür stehen globale Schranken (`SYNC_ALL`, `NOFITY` / `QUERY`), gezielte Schranken (`SYNC_IMAGES`) und eine Speichersynchronisation (`SYNC_MEMORY`) zur Verfügung. Bis auf die gezielten Schranken sollten die Konstrukte in ähnlicher Form bereits von UPC bekannt sein. `SYNC_IMAGES` ermöglicht die gezielte Synchronisation mit einem oder einer Menge von Images. [Vau08]

#### *Zusammenfassung*

CAF verfolgt die gleichen Ziele wie UPC, bei denen der Programmierer ohne Angabe von expliziten Kommunikationsfunktionen auf Speicherbereiche anderer Images zugreifen kann. Durch die Bemühungen einer sehr simplen und einfachen Integration in die Sprache sollten die Erweiterungen von FORTRAN-Entwicklern schnell zu erlernen und zu verwenden sein. Im Gegensatz zu UPC muss die Angabe der Parallelität durch die Co-Array Notation jedoch explizit erfolgen und auch Datenparallelität ist in CAF in Abhängigkeit des Image-Indexes manuell zu erzeugen. Detailliertere Informationen zu CAF können den bereits referenzierten Quellen sowie [Wal10] entnommen werden.

### 4.3.3 Titanium

Titanium entstand 1995 an der University of California (Berkeley) als eine explizite parallele Programmiersprache in Anlehnung an Java. Es wurde sich für eine Erweiterung der Sprache Java aus Gründen der Objektorientierung, der guten Verständlichkeit, der hohen Verbreitung

und der Typsicherheit entschieden, um letztendlich eine performante, sichere und ausdrucksstarke Sprache zu schaffen. Dabei reiht sich Titanium in die PGAS-Sprachen ein, die auf dem SPMD-Ausführungsmodell basieren. [Yel98,Yel061]

Titanium-Programme nutzen Java 1.4 als Grundlage (mit Beschränkungen in Bereichen der Java-Threads und Numerics) und werden vom Titanium-Compiler (`titaniumc`) nach C übersetzt. [UCB09,Yel98] Die weitere Übersetzung des C-Programms kann durch einen beliebigen Compiler erfolgen. Dies macht Titanium-Programme zum einen von der Java Virtual Machine (JVM) unabhängig und zum anderen können weitere Optimierungen durch den C-Compiler durchgeführt werden. [Yel061] Entsprechende Compiler existieren sowohl für Linux als auch für Windows und sind auf der offiziellen Homepage als Download erhältlich. [UCB09] Seit 2005 stagniert die weitere Entwicklung von Titanium, da seit diesem Zeitpunkt keine Änderungen mehr an der Sprache bzw. am Compiler durchgeführt wurden. Aus diesem Grund wird an dieser Stelle nicht weiter auf Titanium eingegangen. Für detailliertere Ausführungen sei auf die Literatur verwiesen, wie z.B. [Hil06], [UCB09], [Yel061] und [Yel98].

### 4.3.4 Chapel

Der Supercomputer-Hersteller Cray arbeitet mit Chapel an einer komplett neuen Programmiersprache im Rahmen des HPCS[76]-Programmes der DARPA[77], die für erfahrene C, FORTRAN oder Java-Programmierer leicht zu erlernen sein soll. [Cha07] Neben Cray nehmen auch IBM mit X10 und Sun bzw. Oracle mit Fortress an dem HPCS-Programm teil, auf die im weiteren Verlauf dieses Kapitels eingegangen wird. Das Ziel des Programmes ist die Schaffung einer neuen Generation von hoch produktiven Computersystemen, die folgende Aspekte erfüllen [Cha09]:

- Verbesserung der Programmierbarkeit paralleler Systeme
- Performance aktueller Programmiermodelle erzielen oder überschreiten
- Bessere Portabilität als aktuelle Programmiermodelle
- Erhöhte Robustheit von parallelem Code

[76] High Productivity Computer Systems
[77] Defense Advanced Research Projects Agency

Das Vorhaben umfasst sowohl Hardware- als auch Software-Aspekte, wobei Chapel zu der letzten Kategorie gehört. Aktuell steht Chapel in der Version 1.1 für Linux, Mac OS X und Windows zur Verfügung und kann auf der offiziellen Homepage[78] kostenlos heruntergeladen werden.

### *Ausführungsmodell*

Zur Erzielung der Parallelität benutzt Chapel ein sogenanntes *Multithreaded Execution Model*, das sich aus einer globalen Sicht auf die Datenstrukturen *(global-view data structures)* und einer globalen Sicht auf den Kontrollfluss *(global-view of control)* zusammensetzt. Das Konzept der globalen Datenstrukturen entspricht dem der Datenverteilung in PGAS-Sprachen, bei denen die Daten global definiert und verteilt gespeichert werden, ohne dass der Programmierer die Verteilung selber durchführen muss. Ein globaler Kontrollfluss steht im Kontrast zu den bisher kennengelernten Sprachen, basierend auf einem SPMD-Modell, bei denen zwar auch nur ein Programm, aber parallel auf mehreren Knoten ausgeführt wird. In Chapel gibt es nur eine Programminstanz, in der Arbeit mittels bestimmter Anweisungen auf andere Knoten ausgelagert wird. Konkret bietet Chapel Unterstützung für Daten- und Taskparallelität. [Cra10]

### *Speichermodell*

Die Abbildung zwischen Daten und den darauf stattfindenden Berechnungen erfolgt in Chapel durch sorgenannte *Locales*. Ein Locale bezeichnet eine Teilkomponente eines parallelen Systems, die über einen eigenen Speicherbereich verfügt und Berechnungen ausführen kann. Locales werden für eine vorgegebene Architektur vom Compiler erzeugt oder können beim Start einer Chapel-Anwendung durch den Parameter `-nl=locales` vorgegeben werden. Bspw. ist für ein Multi-Core- oder SMP-System[79] nur ein Locale zu definieren, wohingegen für ein Cluster-System für jeden Knoten ein Locale zu erzeugen wäre. Threads innerhalb eines Locales können unter gleichen Bedingungen auf den Speicher des Locales zugreifen, wohingegen der Zugriff auf Speicherbereiche anderer Locales mehr Zeit benötigt. Ein Programm wird immer auf dem Locale mit der ID 0 gestartet. [Cha09]

[78] http://chapel.cray.com

[79] Symmetrisches Multiprozessor-System, in dem mehrere (Multi-Core-) Prozessoren durch zwei oder mehr physische Sockel an einen gemeinsamen Speicherbereich angebunden werden.

Damit Daten nicht nur auf einem Locale, sondern auf mehreren verteilt angelegt werden, gibt es *Domain Maps*. Standardmäßig bietet Chapel eine blockweise und eine zyklische Verteilung an. Daneben ist auch die Definition einer benutzerdefinierten Verteilung möglich. Domains dienen dazu, die Form und Größe (und auch die Verteilung) von Arrays festzulegen. [Cha07] Das folgende Beispiel definiert eine zweidimensionale Domain, die anschließend zur Erzeugung eines Arrays genutzt wird [Cha09]:

```
config const rows = 4, cols = 8;
// Domain mit 2 Dimensionen und gewünschter Größe definieren
var D: domain(2) = [1..rows, 1..cols];
var A: [D] int; // Integer-Array auf Basis der Domain D erzeugen
```

Um eine Domain über mehrere Locales hinweg zu verteilen und ggf. Lokalität ausnutzen zu können, muss eine Domain Map spezifiziert werden, der die gewünschte Verteilung übergeben wird [Cho10]:

```
const Distrib = new dmap(new Block(boundingBox=[1..rows, 1..cols]));
var D: domain(2) dmapped Distrib = [1..rows, 1..cols];
```

Die erste Anweisung erzeugt die Domain Map `Distrib`, die für eine blockweise Verteilung für die angegebenen Größen sorgt. In der zweiten Anweisung wird die Verteilung der Domain `D` zugewiesen. Bei vier Zeilen und acht Spalten würde die Verteilung bei acht Locales folgendermaßen aussehen:

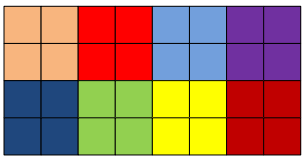


**Abbildung 4-10: Blockweise Verteilung mit acht Locales in Chapel [Cho10]**

Alternativ kann eine zyklische Verteilung wie folgt erzeugt und dargestellt werden [Cho10]:

```
const Distrib = new dmap(new Cyclic(startIdx=(1,1)));
var D: domain(2) dmapped Distrib = [1..rows, 1..cols];
```

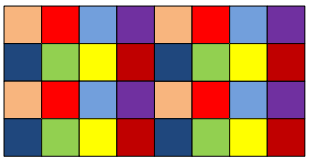


**Abbildung 4-11: Zyklische Verteilung mit acht Locales in Chapel [Cho10]**

Anhand des Startattributes `startIdx` verteilt Chapel nach einem Round-Robin Verfahren die Elemente auf die verschiedenen Locales. [Cho10] Die exakte Vorgehensweise zur Zuordnung der Indizes durch eine Verteilung kann der Spezifikation [Cra10] entnommen werden.

### *Datenparallelität*

Chapel kann Datenparallelität entweder explizit durch die parallele `forall`-Schleife erzeugen oder durch Ausdrücke, die implizit für Parallelität sorgen (Array-Zuweisungen, Reduktions- und Scan-Operationen). [Cra10]

Das nachfolgende Listing stellt die allgemeinste Form der `forall`-Schleife dar [Cra10]:

```
forall index-var-declaration in iterator-expression do statement
```

Bei der Ausführung einer parallelen Schleife muss differenziert werden, auf welcher Art von Domain gearbeitet wird. Bei nicht-verteilten Domains wird anhand der *iterator-expression* die Anzahl an Tasks bestimmt, die alle auf dem aktuellen Locale ausgeführt werden. Ist die Domain verteilt startet Chapel mehrere `forall`-Schleifen auf den verschiedenen Locales, welche die Iterationen auf ihren lokalen Daten ausführen. [Cra10]

```
forall (i,j,k) in [1..N, 1..N, 1..N] do
    C(i,j) += A(i,k) * B(k,j) // Parallele Matrix-Multiplikation
```

Bei der Ausführung einer parallelen Schleife kann der Programmierer manuellen Einfluss darauf nehmen, auf welchem Locale eine `forall`-Schleife und mit wie vielen Threads sie ausgeführt werden soll. Standardmäßig ist die Anzahl an maximalen Threads pro Locale durch die vorhandene Anzahl an Kernen begrenzt. [Cho10]

Reduktions- und Scan-Operationen haben in Chapel die folgende Syntax [Cra10]:

```
reduce-scan-operator [reduce|scan] expression
```

Der erste Teil einer Reduce/Scan-Anweisung besteht aus dem anzuwendenden *Operator*. Standardmäßig unterstützt Chapel die Operatoren: $+, *, \&\&, ||, \&, |, \hat{}, min, max, minloc, maxloc$. Darüber hinaus lassen sich auch eigene Reduktionsoperationen definieren. *Expression* ist ein Platzhalter für jeden Typ, über den iteriert und auf den der Reduktionsoperator angewendet werden kann (z.B. ein Array). [Cra10]

```
var A: [1..5] int = 1; // Initialisiert jedes Element mit 1
var B: [1..5] int;
B = (+ scan A); // Liefert B = 1 2 3 4 5
```

Aktuelle Versuche zeigen, dass der Chapel-Compiler Scan-Operationen immer sequentiell durchführt. In der dem Compiler beiliegenden Status-Datei wird als Grund dafür eine noch unfertige Implementierung angeführt.

### *Taskparallelität*

Analog zur Parallel Pattern Library aus Visual C++ 2010 gibt es in Chapel ebenfalls strukturierte und unstrukturierte Taskparallelität. Unstrukturierte Parallelität wird mit dem Schlüsselwort `begin` erzeugt, dessen folgender Ausdruck in einem eigenen Task ausgeführt wird. Nach Beendigung des Ausdrucks erfolgt keine implizite Synchronisation, wodurch die Anweisung, die `begin` aufruft, sofort zurückkehrt. [Cra10]

```
begin statement
```

Strukturierte Parallelität wird mit den Schlüsselwörtern `cobegin` und `coforall` erzeugt.

```
cobegin { statement-list }
```

Jede Anweisung innerhalb des `cobegin`-Blockes wird als eigener Task gestartet. Anders als bei `begin` erfolgt am Ende des Blockes eine implizite Synchronisation, die auf die Beendigung aller Tasks wartet. [Cra10]

```
coforall index-var-declaration in iterator-expression do statement
```

Mit der `coforall`-Schleife lässt sich die Erzeugung mehrerer Tasks vereinfachen. Der Unterschied zur `forall`-Schleife besteht darin, dass bei `coforall` eine parallele Ausführung

der Iterationen garantiert wird, wohingegen bei `forall` in Abhängigkeit des verwendeten Iterators auch eine sequentielle Ausführung bevorzugt werden kann. Wie bei `cobegin` und `forall` wird die Ausführung des ursprünglichen Programmablaufes erst dann fortgesetzt, wenn alle Iterationen abgeschlossen sind. [Cra10]

### *Synchronisation*

Chapel stellt zur Synchronisierung verschiedene Konstrukte und Variablen bereit, um die Ausführung paralleler Tasks und Schleifen koordinieren zu können [Cra10]:

- Variablenkennzeichnung: `sync`

Synchronisationsvariablen werden mit dem Schlüsselwort `sync` vor der Typdefinition markiert und funktionieren ähnlich wie Locks. Dazu definiert Chapel zwei Zustände: *leer* und *voll*. Ist eine Synchronisationsvariable leer, kann nur dann von ihr gelesen werden, wenn sie durch einen vorangegangen Schreibzugriff gefüllt wurde. Im anderen Fall kann eine volle Synchronisationsvariable nur dann beschrieben werden, wenn sie durch einen vorangegangenen Lesezugriff geleert wurde.

- Variablenkennzeichnung: `single`

Eine `single`-Variable ähnelt einer Synchronisationsvariablen mit dem Unterschied, dass sie nur einmal beschrieben werden kann. Wird auf eine `single`-Variable zugegriffen bevor sie beschrieben wurde, so blockiert die Anweisung.

- Schlüsselwort: `sync` *`statement`*

Das Voranstellen des Schlüsselwortes `sync` vor eine Anweisung oder einen Block von Anweisungen sorgt dafür, dass auf die Beendigung sämtlicher unstrukturierter Tasks, die innerhalb der Anweisungen erzeugt wurden, gewartet wird.

- Schlüsselwort: `serial` *`expression`* `do` *`statement`*

Mit `serial` kann, unter Angabe einer Bedingung, eine parallele Ausführung der angegebenen Anweisungen verhindert werden. Eine parallele Ausführung wird nur dann ermöglicht, wenn die Bedingung nicht erfüllt ist.

- Schlüsselwort: `atomic` *`statement`*

Die mit `atomic` gekennzeichneten Anweisungen bzw. Blöcke können nur von einem Task gleichzeitig ausgeführt bzw. betreten werden.

#### *Zusammenfassung*

Mit Chapel versucht Cray, die Vorteile von Programmiersprachen für Systeme mit gemeinsamem Speicher mit denen von den bisher kennengelernten PGAS-Sprachen zu kombinieren. Dabei ist ein Programmiermodell mit Sicht auf globale Daten und einer globalen Kontrollstruktur entstanden, das trotz einer komplett neuen Entwicklung leicht zu erlernen und zu verstehen ist. Dazu tragen ebenfalls eine verständliche Spezifikation und hilfreiche Tutorials bei, auf die bereits referenziert wurde.

Sowohl Daten- als auch Taskparallelität werden auf einem hohen Abstraktionsniveau unterstützt. An manchen Stellen erfolgt die Parallelität implizit (Array-Operationen, Reduktionen, Scans[80]), an anderen Stellen ist eine explizite Kennzeichnung erforderlich (Schleifen, Tasks). Gleiches gilt für die Zerlegung der Aufgaben, die sich nach der Ebene der Parallelität richtet. Der Programmierer muss sich nicht um die Abbildung auf die Systemressourcen, die Verwaltung der Threads und Kommunikation zwischen den verschiedenen Locales kümmern. Synchronisation kann durch die vorgestellten Mittel manuell vorgenommen werden, bei der strukturierten Parallelität ist sie ebenfalls implizit.

### 4.3.5 Fortress

Fortress wurde im Rahmen des HPCS-Programmes der DARPA von der Firma Sun entwickelt. Im Jahr 2006 entschied sich die DARPA jedoch gegen Fortress und förderte in der letzten Phase nur noch Cray und IBM. Dadurch war die Zukunft von Fortress ungewiss, bis sich Sun im Jahr 2007 dazu entschloss, sämtlichen Quellcode zu veröffentlichen und das Projekt der Allgemeinheit zu übergeben. Dies sicherte die Weiterentwicklung, an der sich viele Universitäten wie die University of Tokyo, University of Virginia, University of Aarhus und die Rice University beteiligten. [Fel08] Dadurch konnte Ende März 2008 die Fortress-Spezifikation und Implementierung in der Version 1.0 fertigstellt werden. Um die Programmierung wis-

[80] In der aktuellen Implementierung erfolgt die Ausführung noch sequentiell

senschaftlicher Anwendungen durch die Sprache natürlicher zu gestalten, wurde bei der Entwicklung von Fortress besonders großer Wert auf die Unterstützung einer mathematischen Notation gelegt. Weiterhin stand eine gute Programmierbarkeit bei der Spezifikation der Sprache im Vordergrund. [Ste08]

Da Fortress in quelloffener Form vorliegt, kann es kostenlos auf der offiziellen Homepage heruntergeladen werden[81]. Die Implementierung des Fortress-Compilers und des Interpreters basieren auf Java 1.6 und setzen zur Ausführung eine JVM voraus. Dadurch lässt sich Fortress auf allen Betriebssystemen einsetzen, für die eine JVM zur Verfügung steht. Darunter zählen u.a. Linux, Mac OS X und Windows. Der Interpreter ermöglicht die direkte Ausführung einer Fortress-Anwendung mittels des Quellcodes. Die Verwendung des Compilers hat ggü. dem Interpreter aufgrund einer höheren Robustheit und Geschwindigkeit der Anwendung Vorteile, da statische Fehler vor der Ausführung erkannt und weitere Optimierungen durchgeführt werden können. [All08]

### *Ausführungsmodell*

Fortress setzt, wie auch Chapel, auf ein *Multithreaded Execution Model*, das sowohl für Multi-Core- und SMP-Systeme als auch für große Cluster-Systeme ausgelegt sein soll. Parallelität ist in der Sprache an vielen Punkten bereits implizit integriert. So werden sämtliche Schleifen standardmäßig parallel ausgeführt, und auch die Berechnung von Argumenten für eine Funktion erfolgt parallel, sofern sich die Argumente aus Funktionen ergeben. Die Auslagerungen auf die vorhandenen Prozessoren übernimmt die Laufzeitumgebung und ist somit transparent für den Entwickler. [Fel08]

### *Speichermodell*

Das Speichermodell ist ähnlich aufgebaut wie in Chapel, jedoch mit dem Unterschied, dass die Locales in Fortress *Regions* heißen und in einer Baumstruktur organisiert sind. Durch eine Hierarchie der Regions soll sich Lokalität besser kontrollieren lassen. Dafür wird jeder Thread und jedes Objekt einer bestimmten Region zugeordnet. [All08]

[81] http://projectfortress.sun.com

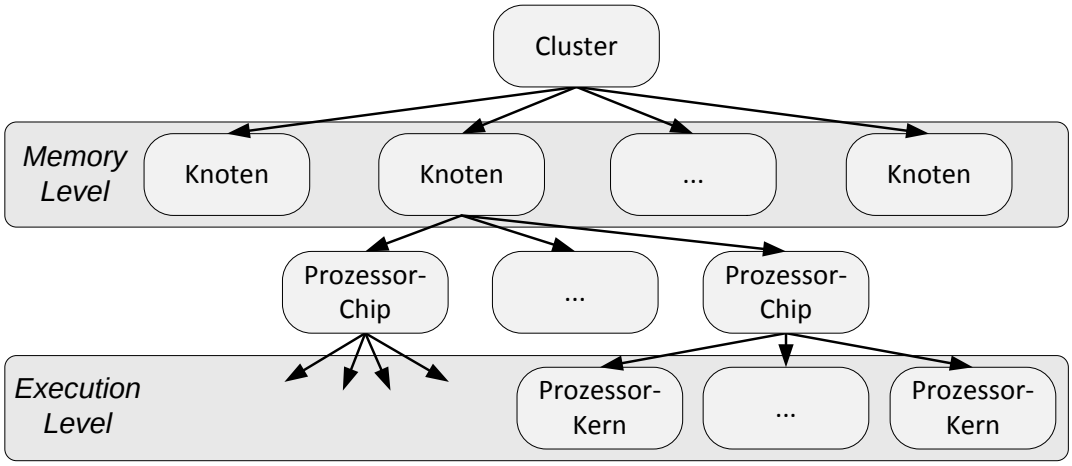



**Abbildung 4-12: Speichermodell von Fortress [Ste06]**

Auf der untersten Ebene der Baumstruktur befindet sich der *Execution Level*, in dem Threads den verschiedenen Regions zugeordnet werden. Eine weitere Ebene ist der *Memory Level*, in dem bestimmt wird, in welcher Region Objekte zu speichern sind. Bei Systemen mit einem gemeinsamen Speicher ist der Memory Level nicht weiter von Bedeutung. [All08]

Jedes Objekt verfügt über Zugriffsrechte die festlegen, ob nur lokaler Zugriff auf das Objekt durch einen Thread möglich ist *(local)*, oder ob auch globaler Zugriff von anderen Threads gestattet wird *(shared)*. Dabei muss ein lokales Objekt sich nicht zwangsläufig in derselben Region befinden wie der Thread, zu dem es gehört. In der aktuellen Implementierung sind alle Objekte nach ihrer Erzeugung shared. In Abhängigkeit von der standardmäßig festgelegten Verteilung, der Größe des Arrays und der Charakteristik des Systems werden Arrays bei der Erzeugung in Fortress über das gesamte System verteilt. Ein Überschreiben der Standardverteilung ist möglich. [All08]

### *Datenparallelität*

Datenparallelität ist in Fortress bereits implizit vorhanden, indem Ausdrücke, Schleifen, Reduktionen und Auswertungen automatisch parallel erfolgen und keiner besonderen Kennzeichnung bedürfen. Zur Realisierung orientierte sich Fortress an Cilk++ und nutzt Work Stealing, um alle Prozessoren möglichst lange gleichmäßig auszulasten. [Sun08] Da keine besonderen Funktionen zur Erzielung der Datenparallelität erforderlich sind, wird auf eine detaillierte Auflistung und Beschreibung aller parallelen Konstrukte an dieser Stelle verzichtet und auf die Spezifikation [All08] verwiesen.

### *Taskparallelität*

Explizite Tasks können in Fortress mit dem Schlüsselwort `spawn` erzeugt werden, das bereits von Cilk++ bekannt ist. Anders als bei Cilk++ erhält man in Fortress als Rückgabewert von `spawn` ein Thread-Objekt, mit dem auf die Beendigung gewartet, die Ausführung abgebrochen oder der Rückgabewert ausgelesen werden kann. [Ben08]

```
T₁ = spawn DoWork()
T₂ = spawn DoOtherWork()
T₁.Wait(); T₂.Wait();
result: ℤ32 := T₁.Value()
```

Alternativ lassen sich parallele Blöcke mit dem Schlüsselwort `also do` nutzen:

```
do
    DoWork()
also do
    DoOtherWork()
end
```

Sämtliche Threads des Blockes bilden eine Gruppe, auf deren gemeinsame Beendigung am Ende des Blockes gewartet wird. [All08] Die Entwickler von Fortress weisen jedoch darauf hin, explizite Methoden zur Parallelisierung nur selten zu verwenden und stattdessen, aufgrund der höheren Flexibilität, auf die implizite Parallelisierung zurückzugreifen. [Ste08]

### *Synchronisation*

Atomare Ausdrücke bilden die einzigen Möglichkeiten zur Synchronisierung in Fortress. Sie können als Block geschrieben werden, wie z.B.

```
atomic do
    x += 1
end
```

oder als einzeiliger Ausdruck, wie z.B. `atomic x += 1`. [All08] Fortress führt atomare Anweisungen durch ein softwarebasiertes, transaktionales Speichermodell auf Basis der in Java geschriebenen DSTM2-Bibliothek aus. Eine detaillierte Beschreibung von DSTM2 ist in [Her06] zu finden.

***Zusammenfassung***

Ein Ziel bei der Entwicklung von Fortress war es, für FORTRAN das zu erreichen, was Java für C geschafft hat. [Fel08] Dennoch ist Fortress keine Erweiterung von FORTRAN, sondern eine komplett neue Programmiersprache. Die mathematische Notation lenkt Fortress in den wissenschaftlichen Bereich und gestaltet die Verwendung, zumindest für Programmierer aus dem C- oder Java-Umfeld, jedoch komplizierter als bspw. Chapel. Der Umfang an Literatur ist ähnlich zu Chapel und beschränkt sich auf eine Spezifikation der Sprache und kurzen Einführungen über die wesentlichen Punkte von Fortress. Auf die entsprechenden Quellen wurde in diesem Abschnitt bereits verwiesen.

Der hohe Abstraktionsgrad durch die implizite Parallelität, die eine Kennzeichnung für datenparallele Operationen überflüssig macht, stellt eine besondere Stärke dieses Ansatzes dar. Die einzigen Aspekte, um die sich der Programmierer selber kümmern muss, sind Synchronisation in kritischen Bereichen sowie die Zerlegung der Aufgaben bei der Taskparallelität.

Es bleibt abzuwarten, ob und inwieweit sich Fortress ggü. den weiterhin von der DARPA unterstützten Sprachen Chapel und X10 etablieren kann.

### 4.3.6 X10

Als letzter Ansatz, sowohl im HPCS-Programm als auch in dieser Arbeit, wird X10 von IBM vorgestellt. Bei der Entwicklung wurde sich sehr stark an der Sprache Java orientiert, die als Grundlage für den sequentiellen Kern von X10 diente. [Cha051] Das Ziel sollte eine neue Programmiersprache sein, mit der sich auf produktive Weise hoch-performante Anwendungen für High-End Systeme schreiben lassen. Der Einsatzbereich von X10 ist aber nicht nur auf große Cluster-Systeme beschränkt, sondern umfasst auch Ein- und Mehrprozessor- bzw. Multi-Core Systeme. [Sar10]

Zum Zeitpunkt dieser Arbeit liegt X10 in der Version 2.0.4 vor und steht auf der offiziellen Homepage kostenlos zum Download bereit[82]. Dort werden bereits vorkompilierte Binärdateien u.a. für Linux, Mac OS X und Windows angeboten. Sollte die gewünschte Plattform

[82] http://x10.codehaus.org

nicht aufgelistet sein, kann X10 mit Hilfe des dort ebenfalls verfügbaren Quellcodes selbst kompiliert werden.

Nach dem Herunterladen bzw. Kompilieren steht neben einem X10-Compiler mit C++-Backend (`x10c++`) auch ein Compiler mit Java-Backend (`x10c`) zur Verfügung. Je nach verwendetem Compiler wird der X10-Quellcode in C++- bzw. Java-Code transformiert und anschließend einem weiteren Compiler (`g++`/`javac`) übergeben. Die mit `x10c++` kompilierten Anwendungen können nativ auf dem System ausgeführt werden, wohingegen die mit `x10c` kompilierten Anwendungen nur innerhalb einer JVM lauffähig sind. [Gro10]

Um X10-Anwendungen auf einem System zu starten, werden die Programme `runx10` für C++- bzw. `x10` für Java-basierte Anwendungen benötigt, denen der Programmname und die Parameter der eigentlichen Anwendung zu übergeben sind. Möchte man eine X10-Anwendung auf mehrere Knoten verteilt starten, kann entweder `mpirun` oder die mitgelieferte Job-Verwaltung, bestehend aus einem *Manager* und einem *Launcher*, genutzt werden. Beide Varianten sind auf native X10-Anwendungen beschränkt. [TXP10]

### *Ausführungsmodell*

X10 nutzt ein ähnliches Ausführungsmodell wie Chapel und Fortress, welches eine höhere Flexibilität im Kontrollfluss, im Gegensatz zu SPMD-Modellen, ermöglicht. Unterstützt werden sowohl Daten- als auch Taskparallelität. Taskparallelität wird durch asynchrone *Activities* erreicht, weshalb X10 in der Literatur teilweise auch als ein *Asynchronous PGAS* (APGAS)-Programmiermodell bezeichnet wird. [Sar10]

### *Speichermodell*

Das bei X10 verwendete Speichermodell basiert auf sogenannten *Places* und hat große Ähnlichkeiten zu den Locales aus Chapel. Places sind Abstraktionen von Computersystemen[83], die Berechnungen ausführen können und über eine gewisse Menge an Speicher verfügen. Auf den Speicher kann von allen Threads des Systems in gleicher Weise zugegriffen werden. Die Anzahl an vorhandenen Places wird beim Start eines X10-Programmes festgelegt und kann intern mit `Place.MAX_PLACES` abgefragt werden. [Sar10]

[83] Single-Core, Multi-Core oder SMP Systeme.

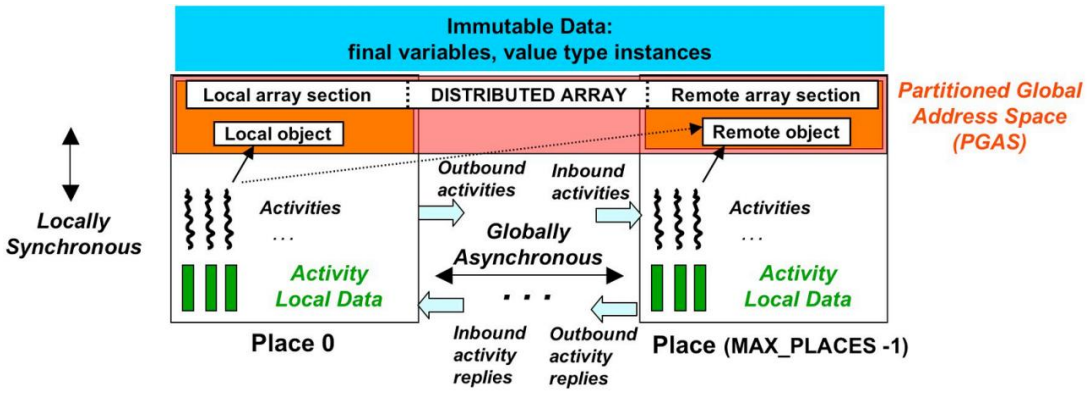



**Abbildung 4-13: Speichermodell und Activities in X10 [Cha051]**

Bei X10 hat sich IBM für ein abstraktes aber explizites Kommunikationsmodell entschieden, um Stellen im Quellcode, an denen Kommunikation erfolgt, besser herauszustellen und dem Programmierer das Finden potentieller Engpässe zu erleichtern. Dies hat zur Folge, dass der Zugriff auf den globalen Speicher eines Places (in Abbildung 4-13 orange dargestellt) nur von einer lokalen Activity aus möglich ist. Soll aus einer Activity auf den Speicher eines anderen Places zugegriffen werden, muss die Activity zuvor in den entfernten Place verschoben werden. Zu diesem Zweck gibt es den Befehl `at`, dem die Angabe des gewünschten Places sowie die dort auszuführende Anweisung folgen. Nach Beendigung der Anweisung wird die Activity mit dem Ergebnis wieder zu dem ursprünglichen Place zurückkopiert. Dadurch kann der Speicherzugriff innerhalb eines Places in einer sequentiellen Weise erfolgen, bei der Lese- und Schreibzugriffe immer komplett und unteilbar ausgeführt werden. Wird fälschlicherweise dennoch auf Objekte anderer Places zugegriffen, sorgt die Laufzeitumgebung von X10 für das Auslösen einer `BadPlaceException`. Oftmals können solche Fehler bereits vom Compiler erkannt werden. [Cha051]

Auf Lokalität ist besonders dann zu achten, wenn Arrays über die gesamten Places hinweg verteilt erzeugt werden. Verschiedene Verteilungsstrategien stehen in X10 in der Klasse `x10.lang.Dist` zur Verfügung und umfassen eine blockweise, zyklische, blockzyklische, zufällige, konstante und eindeutige[84] Verteilung. Die Definition einer benutzerdefinierten Verteilung ist noch nicht möglich. Das folgende Beispiel soll die Erzeugung eines verteilten Arrays mit anschließendem Zugriff verdeutlichen [Sar10]:

[84] Bei der eindeutigen Verteilung wird jedes Element in einem anderen Place abgelegt

```
val N:Int = 1000;
val R = 1..N;
val D = Dist.makeBlock(R);
val A = DistArray.make[Int](D);

for(i in D.region)
    at(A(i)) A(i) = …;
```

### *Datenparallelität*

Datenparallelität lässt sich auf zwei verschiedene Arten schaffen, die sich bzgl. der Lokalität der zu bearbeitenden Daten unterscheiden [Sar10]:

- `foreach`

`foreach` ist eine parallele Variante der normalen `for`-Schleife, die die Iterationen parallel auf dem aktuellen Place ausführt.

- `ateach`

`ateach` ist ebenfalls eine parallele Schleife, erzeugt zur Abarbeitung aber mehrere Activities auf verschiedenen Places gemäß der Verteilung der Daten. Beim Aufruf `ateach(p in D)` muss `D` deshalb entweder eine Verteilung oder ein Array sein, woraus sich die Verteilung der Elemente ableiten lässt. [Cha051,Sar10]

Neben parallelen Schleifen stellen Arrays in X10 parallele kollektive Funktionen bereit. Beispiele dafür sind Reduktionen und Scans. Bei Reduktionen garantiert X10, dass pro Place nur jeweils ein Element kommuniziert wird. [Sar10]

### *Taskparallelität*

Threads werden in X10 durch asynchrone Activities repräsentiert, deren Erzeugung mit geringerem Aufwand und Overhead verbunden ist. Eine Activity wird mit dem Befehl `async`, gefolgt von einer optionalen Angabe des Places und der auszuführenden Anweisung gestartet. [Sar10]

```
async (P) Statement
```

`async` ist nicht blockierend, so dass die Activity nach dem Erzeugen der neuen Activity sofort zurückkehrt. Soll auf die Beendigung einer (oder mehrerer Activities) gewartet werden, steht dafür `finish { }` zur Verfügung. `finish` wartet auf die Beendigung aller in dem Block erzeugten Activities. [Sar10]

Ähnlich wie bei Java bietet X10 neben Activities auch Futures an, die ein Ergebnis zurückliefern. Auf das Ergebnis kann entweder mit der Methode `force` blockierend gewartet oder nicht-blockierend mit der Methode `forced` auf Fertigstellung getestet werden. [Sar10]

```
val F = future DoWork();
Console.OUT.println("Result: " + F.force());
```

### *Synchronisation*

Zur Synchronisierung von Activities existieren, außer den bereits vorgestellten Methoden `finish` und `force`, atomare und bedingte atomare Bereiche sowie Clocks.

- Schlüsselwort: `atomic`

Mit `atomic` lassen sich ganze Methoden oder Blöcke kennzeichnen, die innerhalb eines Places nur von einer Activity ausgeführt werden dürfen. Die Anweisungen innerhalb eines atomaren Blocks müssen vom Compiler analysierbar sein, so dass Lese- und Schreibzugriffe bereits zur Compilezeit feststehen müssen. Auch dürfen innerhalb eines atomaren Blocks keine weiteren Activities erzeugt oder Methoden aufgerufen werden, die eine mögliche Blockierung zur Folge hätten (z.B. `finish`, `force`). Weiterhin müssen sich alle Speicherstellen, auf die innerhalb eines atomaren Blockes zugegriffen wird, innerhalb des jeweiligen Places befinden. [Sar10]

- Schlüsselwort: `when`

Mit dem Schlüsselwort `when` werden bedingte atomare Blöcke definiert. Diese haben die gleichen Eigenschaften und Voraussetzungen wie normale atomare Blöcke, sind aber zusätzlich noch an eine Bedingung gebunden:

```
when (Expression) Statement
```

X10 kann die Anweisungen eines bedingten atomaren Blocks erst dann ausführen, wenn die Bedingung wahr ist. Andernfalls blockiert die Activity an dieser Stelle. Für die Bedingung ist es wichtig, dass sie eine gewisse Stabilität aufweist und nicht nur kurzfristig wahr wird. Bei instabilen Bedingungen ist eine Ausführung in X10 nicht garantiert. [Sar10]

- Clocks

Clocks wurden eingeführt, um Barrier-ähnliche Synchronisationen auf einem feingranulareren Level durchzuführen. Clock-Objekte müssen vor ihrer Verwendung erzeugt werden und lassen sich anschließend Methoden wie `async`, `foreach` oder `ateach` übergeben. Dadurch können unterschiedliche Clock-Objekte eingesetzt werden, womit sich mehrere Gruppen von Activities unabhängig voneinander synchronisieren lassen. Die eigentliche Synchronisation erfolgt innerhalb einer Activity mit der Methode `next()` des Clock-Objektes. Neben `next()` existieren noch weitere Methoden, wie z.B. `drop()`, um das Clock-Objekt von der Activity zu lösen. [Sar10]

```
val clk = Clock.make();
finish {
    async clocked(clk) { DoWorkA(); clk.next(); DoOtherWorkA(); }
    async clocked(clk) { DoWorkB(); clk.next(); DoOtherWorkB(); }
}
```

### *Zusammenfassung*

Von bereits erfahrenen Programmierern kann sich, aufgrund einer Java-ähnlichen Syntax, in X10 leichter eingearbeitet werden als bspw. in Fortress. Auf der Homepage von X10 sind, neben der Spezifikation, noch weitere hilfreiche Quellen, Tutorials und eine komplette API-Beschreibung zu Finden, die einen Einstieg in die Sprache weiter erleichten. Unterstützung bietet X10 sowohl für Daten- als auch für Taskparallelität. Zum Teil geschieht dies implizit durch kollektive Operationen (Reduktionen, Scans) oder explizit durch parallele Schleifen oder durch das Erzeugen von Activities. Bei der Taskparallelität muss sich der Programmierer selber um eine Zerlegung der Aufgaben und die Synchronisation kümmern. Kommunikation ist in X10 per Definition explizit und findet auf einer sehr abstrakten Ebene statt. Dazu sind Activities beim Zugriff auf Daten entfernter Places auf diese zu migrieren. Die Verwaltung der Parallelität und die Abbildung auf die Systemressourcen erfolgen implizit.

# 5 Evaluation paralleler Programmiermodelle

Beststandteil der Evaluation ist die beispielhafte Implementierung bestimmter Anwendungsklassen für einige ausgewählte Ansätze, deren Ergebnisse im Bezug zur Laufzeit und Skalierbarkeit in diesem Kapitel näher untersucht werden. Diese Resultate, zusammen mit den Erkenntnissen der bisherigen Untersuchungen, dienen als Grundlage für eine abschließende Bewertung der Ansätze. Dadurch sollen sich konkrete Aussagen über ihre Einsatzmöglichkeiten und Fähigkeiten tätigen lassen.

## 5.1 Auswahl exemplarischer Anwendungen

Die Auswahl der zu implementierenden Anwendungen wurde in Absprache mit dem betreuenden Dozenten dieser Arbeit durchgeführt. Die ausgewählten Anwendungen sollten möglichst unterschiedliche Bereiche der Parallelisierung abdecken, als auch in der Praxis von Bedeutung sein. Grundsätzlich kann zwischen datenparallelen und taskparallelen Anwendungen unterschieden werden und inwieweit die zu bearbeitenden Daten Abhängigkeiten aufweisen. Tabelle 5-1 stellt eine Übersicht der implementierten Anwendungen mit ihren wesentlichen Eigenschaften dar:

| Name des Benchmarks | Ebene der Parallelität | Abhängigkeiten |
|---|---|---|
| Mandelbrot | Datenparallelität | Nein |
| MST Berechnung mittels Prim | Datenparallelität | Ja |
| Heat | Datenparallelität | Indirekt |
| Matrix-Vektor Multiplikation | Datenparallelität | Nein |
| Quicksort | Taskparallelität | Ja |
| CG aus den NAS Parallel Benchmarks | Datenparallelität | Indirekt |

**Tabelle 5-1: Ausgewählte Anwendungen**

Heat sowie die Minimalgerüst-Bestimmung (MST) eines Graphen nach dem Algorithmus von Prim sind Beispiele für gitterbasierte Anwendungen. Abhängigkeiten werden in Heat durch eine alternierende Verwendung von zwei Datensets vermieden. Bei der MST-Berechnung

lassen sich diese durch den iterativen Ansatz des Prim-Algorithmus nicht vermeiden, da jede Iteration von ihrem Vorgänger abhängig ist. Dadurch wird eine Parallelisierung erschwert.

Die beiden Anwendungen Mandelbrot und Matrix-Vektor Multiplikation sind relativ simple Beispiele für Datenparallelität. In beiden Fällen gibt es keine Abhängigkeiten zu berücksichtigen. Die Matrix-Vektor Multiplikation wird mit spärlich besetzten Matrizen im Compressed Storage Row (CSR) Format unter Verwendung von beispielhaften Matrizen von Matrix Market[85] und der University of Florida[86] durchgeführt.

Quicksort ist ein bekannter Suchalgorithmus, der wegen seiner durchschnittlich guten Laufzeit von $O(n * \log n)$ in der Praxis häufig eingesetzt wird. [Wil99] Von Quicksort werden insgesamt zwei Varianten implementiert, auf die in dem entsprechenden Abschnitt genauer eingegangen wird.

Als letzter Benchmark wird mit Conjugate Gradient ein bereits existierender Benchmark aus den *NASA Advanced Supercomputing* (NAS) Parallel Benchmarks genutzt[87].

Der Quellcode sämtlicher hier genannter Anwendungen befindet sich auf der dieser Arbeit beiliegenden DVD.

## 5.2 Auswahl zu messender Programmiermodelle

Der für diese Arbeit zur Verfügung stehende zeitliche Rahmen macht es unmöglich, alle exemplarischen Anwendungen für jedes vorgestellte Programmiermodell zu implementieren. Dadurch ist eine Beschränkung auf ein paar bestimmte Ansätze erforderlich.

Den Schwerpunkt dieser Arbeit bilden Ansätze für Systeme mit einem gemeinsamen Speicher, so dass aus diesem Bereich die meisten Ansätze herangezogen werden. Diese sind:

- OpenMP
- Intel Thread Building Blocks
- Die Microsoft Concurrency Runtime
- Intel Cilk++

[85] http://math.nist.gov/MatrixMarket/
[86] http://www.cise.ufl.edu/research/sparse/matrices/index.html
[87] http://www.nas.nasa.gov/Resources/Software/npb.html

Auf eine detaillierte Untersuchung der Thread-Bibliotheken Pthreads und Java-Threads wurde verzichtet, da beide Bibliotheken auf einem sehr niedrigen Abstraktionsniveau angesiedelt sind und eine effiziente Implementierung der Benchmarks sehr viel Zeit in Anspruch nehmen würde. Dies ist ein generelles Problem mit threadbasierten Bibliotheken, da diese nicht gut bzw. gar nicht skalieren. [Rei07] Sämtliche Threads sind vom Programmierer manuell und in geeigneter Anzahl für das verwendete System zu erzeugen. Die parallelen Erweiterungen für .NET 4.0 wurden bereits ausführlich in [Hor10] untersucht, so dass an dieser Stelle auf weitere Messungen verzichtet wird. Ebenfalls muss auf eine Untersuchung von Apples Grand Central Dispatch verzichtet werden, da die dafür benötigte Apple Hard- und Software nicht zur Verfügung steht.

Zur Evaluation von Ansätzen mit programmierbaren Grafikkarten steht dieser Arbeit ein leistungsfähiges Grafikkarten-Modell von ATI zur Verfügung, welches die möglichen Ansätze folglich auf OpenCL und Direct Compute beschränkt. Da es sich bei OpenCL um einen offenen Standard handelt und dieser bereits auf vielen Plattformen eingesetzt werden kann, fiel die Entscheidung zugunsten von OpenCL.

Systeme mit einem gemeinsamen verteilten Speicher bilden die letzte Klasse der in dieser Arbeit behandelten Programmiermodelle, bei der ich mich für die beiden Ansätze Unified Parallel C und Chapel entschieden habe. Da UPC im wissenschaftlichen Bereich bereits eine hohe Verbreitung hat, sollte dieser Ansatz in die Untersuchungen mit einfließen. Weiterhin sollten die Untersuchungen eine der neu entwickelten Sprachen im Rahmen des HPCS Programmes der DARPA umfassen. Da eine komplett neue Sprache eine gewisse Einarbeitungszeit benötigt, beschränke ich mich mit Chapel auf eine Sprache. Chapel wurde den anderen Sprachen vorgezogen, da die verfügbaren Materialen den Einstieg am leichtesten gestalten und eine Implementierung im zeitlichen Rahmen dieser Arbeit ermöglichen.

## 5.3 Testsystem und Umgebung

Für die Laufzeituntersuchungen stand dieser Arbeit ein sehr leistungsstarkes Multi-Core SMP-System, bestehend aus zwei Intel Xeon E5530 Quad-Core Prozessoren mit jeweils 2,4 GHz sowie 12 GB DDR3 Arbeitsspeicher zur Verfügung. Die Architektur des Speichersystems

entspricht der NUMA[88], so dass jeder Prozessor direkt auf 6 GB, angebunden durch drei Speicherkanäle, zurückgreifen kann. Der Zugriff auf die anderen 6 GB erfolgt indirekt durch den Intel QPI-Link[89], der die zwei Prozessoren mit bis zu 25,6 GB/s untereinander verbindet. [Int091] Für präzise Messungen wurden die Prozessor-Technologien: Hyper-Threading, TurboBoost und Speedstep während der Tests deaktiviert. Dadurch können die Anwendungen durchgehend auf acht Kerne bei konstanten 2,4 GHz zurückgreifen. Als programmierbare Grafikkarte kam eine ATI Radeon HD 4890 zum Einsatz, die über zehn Shader Cores und insgesamt 800 Streamprozessoren[90] sowie 1 GB Arbeitsspeicher verfügt.

Soweit es die Ansätze ermöglichen, werden alle Messungen unter dem Betriebssystem OpenSUSE 11.2 in der 64-Bit Variante durchgeführt. Lediglich bei den Messungen zur Visual C++ 2010 Concurrency Runtime wird auf Windows 7 Professional 64-Bit zurückgegriffen. Ebenfalls wurde versucht, für alle Ansätze den gleichen Compiler und die gleichen Optimierungseinstellungen zu verwenden. Die folgende Tabelle enthält alle Ansätze mit den jeweils verwendeten Compilern und Parametern:

| Ansatz | Betriebssystem | Compiler | Parameter |
|---|---|---|---|
| OpenMP | OpenSUSE 11.2 x64 | g++ 4.5 | -O3 |
| Intel Thread Building Blocks | OpenSUSE 11.2 x64 | g++ 4.5 | -O3 -std=c++0x |
| Visual C++ 2010 CRT | Windows 7 Prof. x64 | Visual C++ 2010 | /Ox |
| Intel Cilk++ | OpenSUSE 11.2 x64 | cilk++ 4.2.4 | -O3 |
| UPC | OpenSUSE 11.2 x64 | upcc 2.10.2 | -O -Wc,-O3 |
| Chapel | OpenSUSE 11.2 x64 | chpl 1.1 | -O --fast |
| OpenCL | OpenSUSE 11.2 x64 | g++ 4.5 | -O3 |

**Tabelle 5-2: Verwendete Compiler und Parameter**

Bei Chapel weisen die Entwickler darauf hin, dass es sich um eine noch in Entwicklung befindliche Sprache handelt, bei der die Geschwindigkeit an einigen Stellen noch nicht ausreichend optimiert wurde. Weiterhin wird darauf hingewiesen, dass die Sprache im jetzigen Zustand nicht für Geschwindigkeitsuntersuchungen geeignet ist, so dass die in dieser Arbeit

[88] Non-Uniform Memory Architecture
[89] QuickPath Interconnect
[90] 10 Shader Cores * 16 SIMD Einheiten * 5 ALUs pro Einheit = 800 Streamprozessoren

gewonnenen Erkenntnisse bzgl. der Laufzeit eher als richtungsweisend und nicht als endgültig betrachtet werden sollten. [Cra101]

## 5.4 Laufzeit-Messungen

Die im weiteren Verlauf dargestellten Laufzeitmessungen und Skalierbarkeitsuntersuchungen wurden mit einer bestimmten Methodik und unter gewissen Rahmenbedingungen durchgeführt, auf die an dieser Stelle kurz eingegangen werden soll.

### *Verwendete Datenmengen*

Die weiter oben vorgestellten Anwendungen erwarten bei ihrer Ausführung die Übergabe bestimmter Parameter, die den Umfang der Berechnungen beeinflussen. Bei Mandelbrot wird die Anzahl der Iterationen festgelegt, die zur Bestimmung der Ergebnisse herangezogen werden. Quicksort erwartet als Angabe die Anzahl der zu sortierenden Elemente. Für Heat und der MST-Berechnung wird die Größe des zu bearbeitenden Gitters bzw. Graphen (durch Angabe der Anzahl an Knoten) festgelegt. Bei der Matrix-Vektor Multiplikation wird mit dem Parameter eine von drei Matrizen ausgewählt, die beim Start des Programmes geladen wird. Die drei Matrizen unterscheiden sich hauptsächlich durch ihre Größe und betragen im Matrix Market-Format 114 MB, 618 MB und 2,15 GB. Die NAS Parallel Benchmarks stellen unterschiedliche Datenmengen mit Hilfe von Klassen bereit, die beim CG-Benchmark in unterschiedlichen Array-Größen resultieren und beim Kompilieren anzugeben sind.

| Name des Benchmarks | Messreihe 1 | Messreihe 2 | Messreihe 3 |
|---|---|---|---|
| Mandelbrot | 50.000 Iter. | 100.000 Iter. | 200.000 Iter. |
| MST Berechnung mittels Prim | 11.584 Knoten | 25.000 Knoten | 46.000 Knoten |
| Heat | 1024 * 1024 Pkt. | 2048 * 2048 Pkt. | 4096 * 4096 Pkt. |
| Matrix-Vektor Multiplikation | s3dkq4m2.mtx[91] | Freescale1.mtx[92] | nlpkkt160.mtx[93] |
| Quicksort | 100.000.000 Elem. | 250.000.000 Elem. | 500.000.000 Elem. |
| CG aus den NPB | Klasse A | Klasse B | Klasse C |

**Tabelle 5-3: Verwendete Messreihen und Messgrößen**

[91] http://math.nist.gov/MatrixMarket/data/misc/cylshell/s3dkq4m2.html
[92] http://www.cise.ufl.edu/research/sparse/matrices/Freescale/Freescale1.html
[93] http://www.cise.ufl.edu/research/sparse/matrices/Schenk/nlpkkt160.html

Für die Messungen in dieser Arbeit habe ich mich für drei Messreihen mit steigendem Arbeitsaufwand entschieden, die mit ihren Parametern in Tabelle 5-3 dargestellt sind. Sollte ein Ansatz mit einer Größe nicht ausgeführt werden können (z.B. weil nicht genügend Speicher bereitsteht), so wird diese Messung mit einem entsprechenden Hinweis ausgelassen.

Um kleinen Ungenauigkeiten bei den Messungen entgegenzuwirken, wird jede Messung insgesamt fünf Mal durchgeführt und anschließend das arithmetische Mittel gemäß der Formel

$$\bar{x} = \frac{1}{5}\sum_{i=1}^{5} x_i$$

gebildet. Diese Vorgehensweise erhebt nicht den Anspruch auf eine statistische Genauigkeit, jedoch war eine größere Anzahl an Messungen aus zeitlichen Gründen nicht möglich.

### *Initialisierung der Daten*

Während die durchschnittliche Prozessorgeschwindigkeit pro Jahr um ca. 40 Prozent zunimmt, steigt die durchschnittliche Geschwindigkeit des Speichersystems nur um ca. 10 Prozent. Um diesem Ungleichgewicht entgegenzuwirken, entwickelten die Prozessor-Hersteller mehrere Techniken, mit denen sich der Engpass in gewissem Maße kompensieren lässt. Dazu dienen einerseits die in die Prozessoren integrierten Zwischenspeicher *(Caches)*, andererseits entstanden andere Speicherarchitekturen, wie z.B. NUMA, die weiterhin akzeptable Zugriffsgeschwindigkeiten bei sehr vielen Prozessoren ermöglichen sollen. Daher spielt besonders in NUMA-Systemen die Verteilung der Daten eine wichtige Rolle, da der Zugriff auf die Speicherbänke anderer NUMA-Knoten i.d.R. mit höheren Kosten verbunden ist. [Gra03] Dies ist besonders bei den Anwendungen von Bedeutung, die auf einer großen Menge an Daten arbeiten (alle betrachteten Anwendungen außer Mandelbrot). Damit die Vorteile von Caches ausgenutzt und der Zugriff auf Speicherbänke anderer Prozessoren verringert wird, können mehrere Techniken eingesetzt werden.

- Parallele Initialisierung der Daten

Eine parallele Initialisierung der Daten sorgt dafür, dass die Daten, die ein bestimmter Prozessor initialisiert, automatisch seinem Speicherbereich zugeordnet werden und je nach Größe bereits in seinem Cache für den schnellen Zugriff bereit liegen. Wird an späterer Stelle im Programm unter einer ähnlichen Aufteilung der Prozessoren auf die Daten zugegriffen, lassen sich die Vorteile durch Caches und NUMA ausnutzen.

- Verwenden einer Bibliothek zur Anordnung der Daten

Für diese Zwecke steht unter Linux die NUMA-Bibliothek bereit, mit der explizit Speicherbereiche auf einzelnen NUMA-Knoten allokiert werden können. Auch ist eine zyklische Verteilung möglich, die bei solchen Anwendungen hilfreich ist, bei denen das Zugriffsmuster der Prozessoren im Voraus nicht bekannt ist.

Für die Anwendungen in dieser Arbeit wird eine parallele Initialisierung der Daten benutzt, da sie bei den verwendeten Anwendungen zu leicht besseren Ergebnissen führte, als eine zyklische Initialisierung mit der NUMA-Bibliothek. Lediglich bei der Matrix-Vektor Multiplikation wird für die Ansätze: OpenMP, Intel TBB und Intel Cilk++ auf die NUMA-Bibliothek zurückgegriffen, da das Konvertieren der Matrix in das CSR-Format nicht an allen Stellen parallel erfolgt und somit eine parallele Initialisierung nicht gewährleistet ist.

OpenCL-Anwendungen können bzgl. des Speicherzugriffes ebenfalls beschleunigt werden, indem der lokale Speicher der Work-Groups *(shared memory)* ausgenutzt wird. Dies ist jedoch mit einem erheblichen Mehraufwand verbunden und im zeitlichen Rahmen dieser Arbeit so nicht durchzuführen. Daher besteht bei allen OpenCL-Anwendungen, die auf einer großen Menge an Daten arbeiten, noch potentieller Optimierungsspielraum.

### *Prozessoranzahl*

Für Skalierbarkeitsuntersuchungen ist es erforderlich, dass die Laufzeitmessungen der Anwendungen mit einer steigenden Anzahl an Prozessoren durchgeführt werden. Daher wird jede Messung für jede Messreihe unter Verwendung von 1, 2, …, 8 Prozessoren bzw. Prozes-

sorkernen ausgeführt. Bei nur einem Prozessor erfolgt die Messung mit der sequentiellen Variante der Anwendung[94].

Zur Beschränkung der maximalen Anzahl an Prozessoren wurde nach Möglichkeit auf Methoden und Funktionen der jeweiligen Ansätze zurückgegriffen. Bei OpenMP steht dafür die Umgebungsvariable `OMP_NUM_THREADS` zur Verfügung, die auf die gewünschte Anzahl an Prozessoren gesetzt wird. Bei Intel Cilk++ und Chapel lässt sich die maximale Anzahl durch einen Parameter beim Programmstart festlegen (`cilk_set_worker_count` bzw. `maxThreadsPerLocale`). Für UPC wurde jede Anwendung explizit für die entsprechende Prozessoranzahl kompiliert (`-T`). Intel TBB unterstützt die Angabe der maximal zu nutzenden Prozessoren innerhalb des Initialisierungsausdruckes der Bibliothek, jedoch funktionierte diese Beschränkung nicht zuverlässig. Gleiches gilt auch für die Concurrency Runtime, so dass bei diesen Ansätzen die maximale Anzahl mit dem Linux-Befehl `taskset` unter Angabe der zu nutzenden Prozessoren bzw. in Windows durch ein globales Deaktivieren von CPU-Kernen beschränkt wurde.

### *Sonstige Anmerkungen*

Ein paar Besonderheiten gilt es bei den Messungen der mit OpenCL parallelisierten Anwendungen zu beachten. Bei OpenCL-Anwendungen entsteht zusätzlicher Overhead durch die Initialisierung der Geräte und die Kompilierung der Kernel-Funktionen. Im Gegensatz zu C oder C++-Anwendungen, bei denen die Kompilierung vor ihrer Ausführung durchgeführt wird, erfolgt diese bei OpenCL zur Laufzeit (online). Da dieser Aufwand einmalig ist und sich bei bereits vorkompilierten Kernel-Funktionen teilweise auch vermeiden lässt *(offline)*, wird er nicht in die Gesamtlaufzeit mit einbezogen. Lediglich die zusätzlichen Speicheroperationen, die notwendig sind, um Daten vom Host- in den Grafikkarten-Speicher zu kopieren bzw. die Ergebnisse wieder zurück zum Host zu übertragen, werden berücksichtigt.

[94] Alle sequentiellen Anwendungen wurden mit dem g++-4.5 Compiler und der Optimierungsstufe –O3 bzw. dem Visual C++ 2010 Compiler und der Optimierungsstufe /Ox erzeugt.

### *Darstellung der Ergebnisse*

Die Darstellung der gewonnenen Ergebnisse erfolgt in den folgenden Abschnitten durch Tabellen. Pro Messreihe existiert eine Tabelle, in der sowohl die Laufzeiten aller Ansätze ($T_P$) in Sekunden sowie die Speedup-Werte in Bezug zur sequentiellen Laufzeit ($T_S$) gemäß der Formel

$$S(p) = \frac{T_S(n)}{T_P(n)}$$

in Abhängigkeit zu den verwendeten Prozessoren $p$ und der Problemgröße $n$ aufgeführt sind. $n$ wird durch die jeweilige Messreihe festgelegt.

Im Verlauf der Messungen wurde festgestellt, dass die sequentiellen Ergebnisse unter Windows 7 und OpenSUSE 11.2 um bis zu 24 Prozent voneinander differieren. Damit sich diese Differenz nicht negativ auf den Speedup auswirkt, wird zusätzlich die sequentielle Laufzeit unter Windows, für Ansätze basierend auf der Visual C++ 2010 Concurrency Runtime, als Bezugsgröße herangezogen und in der Tabelle dargestellt.

Besonderer Betrachtung bedürfen darüber hinaus die Messergebnisse von Chapel aufgrund des noch nicht durchgängig optimierten Compilers sowie die Messergebnisse von OpenCL. Trotz der unterschiedlichen Architektur von Grafikkarte und Prozessor werden die Messungen von OpenCL in ähnlicher Weise in den Tabellen dargestellt. Die Auflistung der Ergebnisse von OpenCL erfolgt unter der höchsten Prozessor-Anzahl, dessen Überschrift aus diesem Grund mit einem * gekennzeichnet ist. Das * steht stellvertretend für die maximale Anzahl an Prozessorkernen des Systems und beträgt bei der verwendeten ATI Radeon HD 4890 Grafikkarte 800 und bei den zwei Intel Xeon Quad-Core-Prozessoren acht. Die jeweils besten und schlechtesten Werte werden pro Prozessoranzahl farblich hervorgehoben.

Als Basis für die Messwerte der Tabellen dienen die Durchschnittswerte der Messungen. Alle Messergebnisse können im Detail der beiliegenden DVD entnommen werden. Weiterhin finden sich dort grafische Darstellungen der Tabellen in Form von Säulendiagrammen. Aus Gründen der Übersichtlichkeit wurde auf eine zusätzliche Abbildung in den folgenden Abschnitten verzichtet.

### 5.4.1 Mandelbrot

Die Anwendung Mandelbrot berechnet iterativ die sogenannte *Mandelbrot-Menge*, bei der es sich um eine Menge von Punkten in der gaußschen Zahlenebene handelt. Ein Punkt gehört zu der Menge, falls sein Wert „quasi-stabil“ ist, d.h. wenn er eine gewisse Schranke nicht überschreitet. Zur iterativen Berechnung wird die Funktion

$$z_{k+1} = z_k^2 + c$$

genutzt, mit $z = a + bi,\ i = \sqrt{-1}$ und einem Punkt $c$ aus der gaußschen Zahlenebene. Die Iterationen werden solange durchgeführt, bis entweder eine gewisse Anzahl erreicht ist, oder $z > 2$ gilt ($z$ würde somit weiter unbeschränkt wachsen). [Wil99] Dies stellt eine Herausforderung an die Ansätze dar, da die Berechnung der Punkte zwar parallel erfolgen kann, aber je nach Punkt unterschiedlich viele Iterationen benötigt. Bis auf OpenMP nutzten alle Ansätze für die parallele Schleife ihre standardmäßige Scheduling-Strategie. Da OpenMP ohne manuelles Eingreifen ein statisches Scheduling vornimmt, aber dies aufgrund des variierenden Aufwandes nicht ideal ist, wurde auf ein dynamisches Scheduling zurückgegriffen.

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 64,9396 | | | | | | | |
| | OpenMP | | 32,4714 | 21,6429 | 16,2322 | 13,0064 | 10,8612 | 9,3389 | 8,1698 |
| | Intel TBB | | 32,8434 | 22,9490 | 17,2298 | 13,5860 | 12,0709 | 10,5798 | 8,8857 |
| | MS CRT | 69,4685 | 34,9181 | 23,3949 | 17,7998 | 14,4536 | 12,0082 | 10,4871 | 9,2175 |
| | Intel Cilk++ | | 36,9355 | 23,0496 | 18,2719 | 13,8417 | 11,7688 | 10,2628 | 9,1066 |
| | UPC | | 32,4696 | 21,6853 | 16,2442 | 12,9872 | 10,8432 | 9,3088 | 8,1274 |
| | Chapel | | 37,3802 | 26,1409 | 21,2684 | 20,5950 | 20,5970 | 20,5899 | 20,5866 |
| | OpenCL | | | | | | | | 2,9028 |
| Speedup | OpenMP | | 1,9999 | 3,0005 | 4,0007 | 4,9929 | 5,9790 | 6,9537 | 7,9488 |
| | Intel TBB | | 1,9772 | 2,8297 | 3,7690 | 4,7799 | 5,3799 | 6,1380 | 7,3083 |
| | MS CRT | | 1,9895 | 2,9694 | 3,9028 | 4,8063 | 5,7851 | 6,6242 | 7,5366 |
| | Intel Cilk++ | | 1,7582 | 2,8174 | 3,5541 | 4,6916 | 5,5180 | 6,3277 | 7,1310 |
| | UPC | | 2,0000 | 2,9946 | 3,9977 | 5,0003 | 5,9890 | 6,9761 | 7,9902 |
| | Chapel | | 1,7373 | 2,4842 | 3,0533 | 3,1532 | 3,1529 | 3,1540 | 3,1545 |
| | OpenCL | | | | | | | | 22,3741 |

**Tabelle 5-4: Messergebnisse für Mandelbrot (Messreihe 1)**

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 129,860 | | | | | | | |
| | OpenMP | | 64,9187 | 43,2775 | 32,4578 | 26,0138 | 21,6658 | 18,6301 | 16,2751 |
| | Intel TBB | | 66,8668 | 44,6031 | 34,1240 | 27,1193 | 23,9041 | 21,1656 | 17,9333 |
| | MS CRT | 138,895 | 69,8315 | 46,7697 | 35,5806 | 28,9771 | 24,0663 | 20,8963 | 18,5024 |
| | Intel Cilk++ | | 73,8393 | 45,6847 | 35,9801 | 27,7177 | 23,0806 | 20,2199 | 17,9091 |
| | UPC | | 64,9386 | 43,3518 | 32,4837 | 25,9626 | 21,6852 | 18,6141 | 16,2525 |
| | Chapel | | 74,7600 | 52,2890 | 42,5420 | 41,1735 | 41,1770 | 41,1758 | 41,1757 |
| | OpenCL | | | | | | | | 5,7712 |
| Speedup | OpenMP | | 2,0003 | 3,0006 | 4,0009 | 4,9920 | 5,9938 | 6,9704 | 7,9790 |
| | Intel TBB | | 1,9421 | 2,9114 | 3,8055 | 4,7885 | 5,4325 | 6,1354 | 7,2412 |
| | MS CRT | | 1,9890 | 2,9698 | 3,9037 | 4,7933 | 5,7713 | 6,6469 | 7,5069 |
| | Intel Cilk++ | | 1,7587 | 2,8425 | 3,6092 | 4,6851 | 5,6264 | 6,4224 | 7,2511 |
| | UPC | | 1,9997 | 2,9955 | 3,9977 | 5,0018 | 5,9884 | 6,9764 | 7,9901 |
| | Chapel | | 1,7370 | 2,4835 | 3,0525 | 3,1540 | 3,1537 | 3,1538 | 3,1538 |
| | OpenCL | | | | | | | | 22,5012 |

**Tabelle 5-5: Messergebnisse für Mandelbrot (Messreihe 2)**

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 259,630 | | | | | | | |
| | OpenMP | | 129,803 | 86,529 | 64,927 | 51,973 | 43,296 | 37,162 | 32,531 |
| | Intel TBB | | 131,386 | 89,776 | 67,451 | 55,945 | 46,432 | 39,869 | 36,008 |
| | MS CRT | 277,747 | 139,617 | 93,411 | 71,082 | 57,516 | 47,963 | 41,854 | 37,151 |
| | Intel Cilk++ | | 147,641 | 91,353 | 71,989 | 55,394 | 45,818 | 40,460 | 35,749 |
| | UPC | | 129,824 | 86,675 | 64,968 | 51,903 | 43,367 | 37,223 | 32,507 |
| | Chapel | | 149,487 | 104,535 | 85,057 | 82,338 | 82,343 | 82,335 | 82,343 |
| | OpenCL | | | | | | | | 11,506 |
| Speedup | OpenMP | | 2,0002 | 3,0005 | 3,9988 | 4,9955 | 5,9967 | 6,9865 | 7,9810 |
| | Intel TBB | | 1,9761 | 2,8920 | 3,8492 | 4,6408 | 5,5917 | 6,5122 | 7,2103 |
| | MS CRT | | 1,9893 | 2,9734 | 3,9074 | 4,8290 | 5,7908 | 6,6361 | 7,4762 |
| | Intel Cilk++ | | 1,7585 | 2,8421 | 3,6065 | 4,6870 | 5,6665 | 6,4169 | 7,2627 |
| | UPC | | 1,9999 | 2,9954 | 3,9963 | 5,0022 | 5,9868 | 6,9749 | 7,9870 |
| | Chapel | | 1,7368 | 2,4837 | 3,0524 | 3,1532 | 3,1530 | 3,1533 | 3,1530 |
| | OpenCL | | | | | | | | 22,565 |

**Tabelle 5-6: Messergebnisse für Mandelbrot (Messreihe 3)**

Zur Parallelisierung konnte das sequentielle Programm größtenteils übernommen werden. Lediglich zur Berechnung der Prüfsumme, die sich aus der tatsächlichen Anzahl durchgeführter Iterationen ergibt, wurde auf spezielle Reduktionskonzepte der jeweiligen Ansätze zurückgegriffen. Anhand der Messergebnisse ist zu erkennen, dass sich Mandelbrot sehr gut parallelisieren lässt. Der ansteigende Arbeitsaufwand führt zu keinen signifikanten Veränderungen beim Speedup, so dass die automatische Lastbalancierung der Ansätze ebenfalls sehr gut funktioniert. Der Speedup von OpenMP und UPC wächst linear mit der Anzahl an Prozes-

soren, die Werte von Intel TBB, der CRT und Intel Cilk++ liegen leicht darunter. Eine mögliche Ursache dafür kann ein höherer Overhead der Reduktionsoperationen sein. Chapel zieht keinen weiteren Nutzen aus mehr als vier Prozessoren. Dieses Bild wird sich auch teilweise bei den noch folgenden Anwendungen zeigen, so dass das Problem vermutlich auf die noch unausgereifte Implementierung zurückzuführen ist.

Das beste Resultat ließe sich mit OpenCL erzielen, das den sehr hohen Grad an Datenparallelität durch die vielen Recheneinheiten am besten ausnutzen konnte. Das OpenCL-Programm brauchte durchschnittlich nur 4,4 Prozent der Zeit des sequentiellen Algorithmus und nur 35 Prozent ggü. dem schnellsten parallelen Ansatz. Bei der Umsetzung stellte es sich als effizienter heraus, die Reduktion der Prüfsumme von der CPU berechnen zu lassen. Jede Recheneinheit hinterlegt dazu in einem gesonderten Array ihre lokale Reduktion, das anschließend in den Host-Speicher übertragen und dort von der CPU sequentiell reduziert wird.

### 5.4.2 MST Berechnung mittels Prim

Bei der Berechnung des Minimum Spanning Trees, auch kleinster spannender Baum oder Minimalgerüst genannt, wird in einem gewichteten, ungerichteten Graphen $G$ der Untergraphen $G'$ gesucht, der alle Knoten von $G$ als Baum enthält und die Summe an Kantengewichten minimiert. [Gra03] Abbildung 5-1 ist ein Beispiel für ein Minimalgerüst.

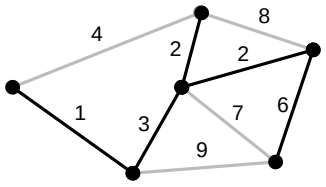



**Abbildung 5-1: Beispiel eines Minimalgerüstes**

Beim Programmstart wird, in Abhängigkeit der übergebenen Größe $n$, ein vollständiger Graph mit $n$ Knoten erzeugt. Durch die Vollständigkeit wird sichergestellt, dass der Graph zusammenhängend ist und ein gemeinsames Minimalgerüst bestimmt werden kann. Die Gewichtung der Kanten erfolgt zufällig. Das Minimalgerüst wird anschließend mit Hilfe des Prim-Algorithmus erzeugt.

Die Schwierigkeit bei der Parallelisierung des Prim-Algorithmus wird durch dessen iterative Vorgehensweise bestimmt. Es wird mit einem beliebigen Startknoten begonnen, von dem

aus ein Knoten aus der Liste der noch nicht besuchten Knoten hinzugefügt wird, zu dem das Kantengewicht minimal ist. Anschließend werden die Distanzen, mit denen die restlichen Knoten nun erreicht werden können, aktualisiert. Diese Vorgehensweise wird so lange wiederholt bis alle Knoten besucht wurden. [Gra03]

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 2,3347 | | | | | | | |
| | OpenMP | | 1,3719 | 1,0118 | 0,8025 | 0,6784 | 0,6001 | 0,9523 | 0,5001 |
| | Intel TBB | | 8,0458 | 4,2128 | 4,3047 | 2,5308 | 2,5483 | 1,9432 | 1,1112 |
| | MS CRT | 2,1474 | 5,6941 | 4,0197 | 3,5895 | 3,5818 | 3,8022 | 2,6949 | 3,8807 |
| | Intel Cilk++ | | 2,2499 | 1,8143 | 1,4355 | 1,3227 | 1,2134 | 1,1362 | 1,0566 |
| | UPC | | 4,5012 | 5,0170 | 3,5629 | 3,9437 | 3,4516 | 3,3264 | 2,3263 |
| | Chapel | | 5,2787 | 5,0104 | 5,6397 | 5,0487 | 5,4992 | 4,6301 | 5,4980 |
| | OpenCL | | | | | | | | 4,4337 |
| Speedup | OpenMP | | 1,7019 | 2,3075 | 2,9094 | 3,4416 | 3,8908 | 2,4517 | 4,6690 |
| | Intel TBB | | 0,2902 | 0,5542 | 0,5424 | 0,9225 | 0,9162 | 1,2015 | 2,1011 |
| | MS CRT | | 0,3771 | 0,5342 | 0,5982 | 0,5995 | 0,5648 | 0,7968 | 0,5533 |
| | Intel Cilk++ | | 1,0377 | 1,2868 | 1,6264 | 1,7652 | 1,9242 | 2,0548 | 2,2096 |
| | UPC | | 0,5187 | 0,4654 | 0,6553 | 0,5920 | 0,6764 | 0,7019 | 1,0036 |
| | Chapel | | 0,4423 | 0,4660 | 0,4140 | 0,4624 | 0,4246 | 0,5042 | 0,4246 |
| | OpenCL | | | | | | | | 0,5266 |

**Tabelle 5-7: Messergebnisse für die MST Berechnung (Messreihe 1)**

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 11,9686 | | | | | | | |
| | OpenMP | | 7,1693 | 5,0328 | 3,9222 | 3,2864 | 2,8517 | 2,5635 | 2,3124 |
| | Intel TBB | | 30,9379 | 13,1764 | 9,8652 | 7,8448 | 9,1717 | 7,6141 | 4,3873 |
| | MS CRT | 11,6542 | 26,8460 | 18,5663 | 16,1539 | 13,7381 | 20,0360 | 16,6395 | 19,8077 |
| | Intel Cilk++ | | 11,3774 | 8,1140 | 6,3155 | 5,4564 | 4,8601 | 4,3818 | 3,9865 |
| | UPC | | 22,7713 | 24,6652 | 17,2318 | 19,1047 | 16,5596 | 16,0206 | 10,7069 |
| | Chapel | | 17,3732 | 16,9575 | 16,9530 | 17,3251 | 17,0838 | 17,3393 | 16,9889 |
| | OpenCL | | | | | | | | ---------- |
| Speedup | OpenMP | | 1,6694 | 2,3781 | 3,0515 | 3,6419 | 4,1970 | 4,6689 | 5,1760 |
| | Intel TBB | | 0,3869 | 0,9083 | 1,2132 | 1,5257 | 1,3050 | 1,5719 | 2,7280 |
| | MS CRT | | 0,4341 | 0,6277 | 0,7214 | 0,8483 | 0,5817 | 0,7004 | 0,5884 |
| | Intel Cilk++ | | 1,0520 | 1,4751 | 1,8951 | 2,1935 | 2,4626 | 2,7314 | 3,0023 |
| | UPC | | 0,5256 | 0,4852 | 0,6946 | 0,6265 | 0,7228 | 0,7471 | 1,1178 |
| | Chapel | | 0,6889 | 0,7058 | 0,7060 | 0,6908 | 0,7006 | 0,6903 | 0,7045 |
| | OpenCL | | | | | | | | ---------- |

**Tabelle 5-8: Messergebnisse für die MST Berechnung (Messreihe 2)**

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 60,6787 | | | | | | | |
| | OpenMP | | 34,5866 | 24,3127 | 18,2937 | 15,2345 | 13,1605 | 12,1211 | 11,0746 |
| | Intel TBB | | 92,3734 | 46,7292 | 34,7267 | 29,1550 | 28,5495 | 22,0591 | 16,6125 |
| | MS CRT | 55,7612 | 122,604 | 84,0044 | 70,2606 | 54,7228 | 70,2009 | 68,1432 | 82,8914 |
| | Intel Cilk++ | | 46,0898 | 32,2068 | 24,7116 | 21,1630 | 18,3838 | 16,4874 | 14,7734 |
| | UPC | | 86,9020 | 93,3384 | 66,4524 | 72,6268 | 62,7257 | 59,6324 | 41,6369 |
| | Chapel | | 56,1021 | 55,8240 | 54,2696 | 55,8956 | 56,5334 | 56,6412 | 56,7720 |
| | OpenCL | | | | | | | | ---------- |
| Speedup | OpenMP | | 1,7544 | 2,4958 | 3,3169 | 3,9830 | 4,6107 | 5,0060 | 5,4791 |
| | Intel TBB | | 0,6569 | 1,2985 | 1,7473 | 2,0812 | 2,1254 | 2,7507 | 3,6526 |
| | MS CRT | | 0,4548 | 0,6638 | 0,7936 | 1,0190 | 0,7943 | 0,8183 | 0,6727 |
| | Intel Cilk++ | | 1,3165 | 1,8840 | 2,4555 | 2,8672 | 3,3007 | 3,6803 | 4,1073 |
| | UPC | | 0,6982 | 0,6501 | 0,9131 | 0,8355 | 0,9674 | 1,0175 | 1,4573 |
| | Chapel | | 1,0816 | 1,0870 | 1,1181 | 1,0856 | 1,0733 | 1,0713 | 1,0688 |
| | OpenCL | | | | | | | | ---------- |

**Tabelle 5-9: Messergebnisse für die MST Berechnung (Messreihe 3)**

Der iterative Ansatz verbietet die parallele Auswahl und das parallele Hinzufügen mehrerer Knoten zum Minimalgerüst. Dies schließt eine triviale Parallelisierung aus, so dass lediglich das Aktualisieren des Distanzvektors sowie das Finden des Minimums parallelisiert werden können. [Gra03]

Allen Ansätzen gemeinsam ist die Tatsache, dass die zu bewältigende Arbeit in den drei Messreihen noch zu klein ist, um den parallelen Overhead zu kompensieren. Die Speicherbandbreite stellt ebenfalls einen weiteren limitierenden Faktor dar.

Der Speedup verbessert sich mit wachsender Knotenanzahl, erreicht aber auch mit 46.000 Knoten noch keine idealen Werte. Die besten Resultate konnte OpenMP erzielen, gefolgt von Intel Cilk++ und Intel TBB. Obwohl die parallelen Konstrukte von TBB und der CRT größtenteils identisch und kompatibel zueinander sind, kann die CRT überhaupt keinen Nutzen aus mehreren Prozessoren ziehen. Im Gegenteil: Die Laufzeit wird durch den Einsatz mehrerer Prozessoren verschlechtert und erreicht, bis auf eine Ausnahme, zu keiner Zeit die Laufzeit der sequentiellen Variante. Ein ähnliches Bild zeigt sich auch bei Chapel. Hier konnte ebenfalls kein Nutzen aus mehreren Prozessoren gezogen werden.

Das relativ schlechte Abschneiden der CRT und Chapel führe ich auf die Form des Algorithmus zurück, bei dem im Körper einer sequentiellen while-Schleife pro Iteration zwei parallele

For-Schleifen gestartet werden (zum Finden des Minimums und zur Aktualisierung des Distanzvektors). Dadurch muss der Ansatz für jede Iteration zwei Mal eine geeignete Anzahl an Threads erstellen und die Arbeit entsprechend aufteilen. Dadurch lässt sich festhalten, dass die Kosten zur Erzeugung der Parallelität bei OpenMP, Cilk++ und TBB geringer ausfallen als bei der CRT und Chapel. Meine Vermutung liegt hier an der Integration in die Sprache, die dem Programmierer zwar mehr Komfort bietet, aber mit zusätzlichem Overhead verbunden ist.

UPC kann, im Gegensatz zu der CRT und Chapel, die Laufzeiten mit zunehmender Anzahl an Prozessoren verringern. Dennoch liegen sie um mehr als das Doppelte über denen von OpenMP. Ähnliche Ergebnisse zeigen sich auch bei den folgenden Benchmarks, bei denen zwischen den verschiedenen Threads Daten ausgetauscht werden müssen. Bei der MST-Berechnung muss bspw. ein globales Minimum für den nächsten hinzuzufügenden Knoten ermittelt werden, an deren Berechnung jeder Thread beteiligt ist. Alle anderen Berechnungen finden immer auf den jeweils lokalen Daten der Threads statt. Demnach verursachen die threadübergreifende Kommunikation sowie der allgemeine Zugriff auf den verteilten Speicher einen nicht zu vernachlässigenden Anteil an Overhead. Das sich manche Messungen nicht mit UPC durchführen ließen, liegt an einer Beschränkung der Größe des verteilten Speichers in Abhängigkeit zu der Anzahl verwendeter Prozessoren. Zwar kann die Größe mit dem Parameter `-shared-heap=xMB` manuell erweitert werden, jedoch brachte dies für bestimmte Prozessorkonfigurationen keinen Erfolg.

Aufgrund einer Speicherbeschränkung im Treiber von AMD stehen der Grafikkarte unter OpenCL lediglich 256 MB ihrer 1024 MB für Berechnungen zur Verfügung. Deshalb konnte nur die erste Messreihe mit dem OpenCL-Ansatz ausgeführt werden. Bereits anhand diesen Ergebnissen ist zu erkennen, dass sich keine überzeugenden Resultate erzielen lassen. Für die Messungen wurde die Anwendung auf zwei verschiedene Weisen in OpenCL parallelisiert, von denen die Ergebnisse der schnelleren Variante in den Tabellen aufgeführt sind. Bei der schnelleren Variante wird nur die Aktualisierung des Distanzvektors von der Grafikkarte übernommen. Die Berechnung des Minimums erfolgt durch die CPU. Dadurch muss der Distanzvektor in jeder Iteration aus dem Speicher der Grafikkarte zum Host kopiert werden. Die andere Variante vermeidet diesen zusätzlichen Overhead, indem sie sämtliche Berechnun-

gen auf der Grafikkarte ausführt. Dementsprechend wird auch das Finden des minimalsten Kantengewichtes von der Grafikkarte übernommen, was sich bei den Untersuchungen jedoch als weniger effizient herausstellte und 12 Prozent langsamer war. Zur Implementierung von Reduktionen orientierte ich mich an dem Reduktions-Beispiel des ATI Stream SDKs 2.1.

### 5.4.3 Heat

Heat simuliert, für eine vorgegebene Gittergröße und einer bestimmten Anzahl an Iterationen, eine einfache Hitzeverteilung, bei der jeder Punkt nur durch seine vier direkten Nachbarn beeinflusst wird. Um Abhängigkeiten zu vermeiden, nutzt das Programm zwei Gitter, die je nach Iteration alternierend verwendet werden.

Demnach ergibt sich die Formel zur Berechnung der Temperaturwerte für die Iteration $k+1$ wie folgt:

$$heat_{k+1}[i,j] = \frac{1}{4} * (heat_k[i-1,j] + heat_k[i+1,j] + heat_k[i,j-1] + heat_k[i,j+1])$$

Abbildung 5-2 stellt die Vorgehensweise nochmals grafisch dar.

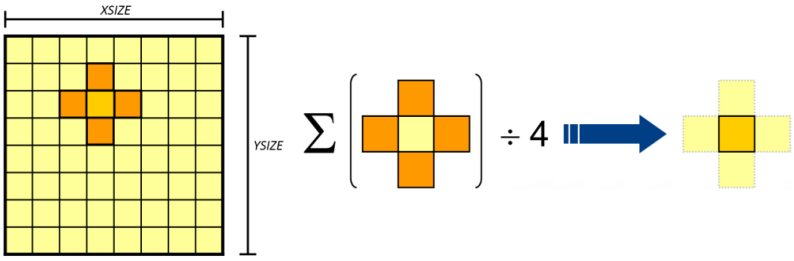



**Abbildung 5-2: Grafische Darstellung des Heat-Transfers [Cra08]**

Für die folgenden Messungen wurde eine konstante Iterationsanzahl von 5.000 bei wachsender Gittergröße pro Messreihe gewählt.

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 26,9452 | | | | | | | |
| | OpenMP | | 12,8720 | 12,6759 | 8,1364 | 8,3437 | 6,2351 | 6,6335 | 5,5714 |
| | Intel TBB | | 19,3714 | 13,0534 | 10,7066 | 10,5531 | 9,5687 | 10,1154 | 9,8527 |
| | MS CRT | 32,0253 | 18,4540 | 15,6541 | 15,3527 | 11,6433 | 13,4886 | 12,3299 | 12,3508 |
| | Intel Cilk++ | | 16,3726 | 13,3205 | 11,3040 | 10,2360 | 9,5396 | 8,6972 | 8,2393 |
| | UPC | | 148,252 | 133,775 | 97,0757 | 78,1373 | 72,4986 | 70,0399 | 48,6966 |
| | Chapel | | 16,9196 | 15,2501 | 11,3028 | 12,1612 | 11,7256 | 12,4099 | 9,8837 |
| | OpenCL | | | | | | | | 3,6713 |
| Speedup | OpenMP | | 2,0933 | 2,1257 | 3,3117 | 3,2294 | 4,3215 | 4,0620 | 4,8363 |
| | Intel TBB | | 1,3910 | 2,0642 | 2,5167 | 2,5533 | 2,8160 | 2,6638 | 2,7348 |
| | MS CRT | | 1,7354 | 2,0458 | 2,0860 | 2,7505 | 2,3742 | 2,5974 | 2,5930 |
| | Intel Cilk++ | | 1,6458 | 2,0228 | 2,3837 | 2,6324 | 2,8246 | 3,0981 | 3,2703 |
| | UPC | | 0,1818 | 0,2014 | 0,2776 | 0,3448 | 0,3717 | 0,3847 | 0,5533 |
| | Chapel | | 1,5925 | 1,7669 | 2,3840 | 2,2157 | 2,2980 | 2,1713 | 2,7262 |
| | OpenCL | | | | | | | | 7,3395 |

**Tabelle 5-10: Messergebnisse für Heat (Messreihe 1)**

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 109,381 | | | | | | | |
| | OpenMP | | 71,4371 | 63,9493 | 55,8920 | 50,1279 | 39,2329 | 39,9235 | 33,5848 |
| | Intel TBB | | 77,3750 | 57,2779 | 50,2672 | 50,2317 | 51,2738 | 48,7527 | 55,7255 |
| | MS CRT | 132,190 | 74,0290 | 62,6155 | 60,0644 | 50,9658 | 55,5772 | 56,4036 | 53,1649 |
| | Intel Cilk++ | | 78,8403 | 62,3715 | 54,6300 | 51,0507 | 45,9613 | 42,9234 | 41,0524 |
| | UPC | | 671,262 | 527,518 | 337,698 | 316,310 | 272,058 | 257,272 | 168,397 |
| | Chapel | | 86,8882 | 71,2643 | 69,9242 | 65,5513 | 60,8521 | 64,7424 | 72,3027 |
| | OpenCL | | | | | | | | 13,6210 |
| Speedup | OpenMP | | 1,5312 | 1,7104 | 1,9570 | 2,1820 | 2,7880 | 2,7398 | 3,2569 |
| | Intel TBB | | 1,4136 | 1,9097 | 2,1760 | 2,1775 | 2,1333 | 2,2436 | 1,9629 |
| | MS CRT | | 1,7856 | 2,1111 | 2,2008 | 2,5937 | 2,3785 | 2,3436 | 2,4864 |
| | Intel Cilk++ | | 1,3874 | 1,7537 | 2,0022 | 2,1426 | 2,3798 | 2,5483 | 2,6644 |
| | UPC | | 0,1629 | 0,2074 | 0,3239 | 0,3458 | 0,4021 | 0,4252 | 0,6495 |
| | Chapel | | 1,2589 | 1,5349 | 1,5643 | 1,6686 | 1,7975 | 1,6895 | 1,5128 |
| | OpenCL | | | | | | | | 8,0303 |

**Tabelle 5-11: Messergebnisse für Heat (Messreihe 2)**

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 439,331 | | | | | | | |
| | OpenMP | | 308,141 | 246,529 | 223,055 | 187,366 | 162,239 | 150,153 | 135,565 |
| | Intel TBB | | 291,251 | 229,548 | 206,227 | 192,815 | 198,462 | 202,907 | 204,793 |
| | MS CRT | 481,943 | 296,866 | 251,310 | 236,714 | 191,620 | 190,389 | 185,955 | 185,875 |
| | Intel Cilk++ | | 316,770 | 245,188 | 218,827 | 198,205 | 182,080 | 173,838 | 165,335 |
| | UPC | | 2632,33 | 2123,34 | 1322,24 | 1244,95 | 1088,04 | 1015,04 | 664,169 |
| | Chapel | | 347,659 | 266,677 | 267,902 | 251,221 | 242,674 | 254,476 | 291,785 |
| | OpenCL | | | | | | | | 52,938 |
| Speedup | OpenMP | | 1,4257 | 1,7821 | 1,9696 | 2,3448 | 2,7079 | 2,9259 | 3,2407 |
| | Intel TBB | | 1,5084 | 1,9139 | 2,1303 | 2,2785 | 2,2137 | 2,1652 | 2,1452 |
| | MS CRT | | 1,6234 | 1,9177 | 2,0360 | 2,5151 | 2,5314 | 2,5917 | 2,5928 |
| | Intel Cilk++ | | 1,3869 | 1,7918 | 2,0077 | 2,2165 | 2,4128 | 2,5272 | 2,6572 |
| | UPC | | 0,1669 | 0,2069 | 0,3323 | 0,3529 | 0,4038 | 0,4328 | 0,6615 |
| | Chapel | | 1,2637 | 1,6474 | 1,6399 | 1,7488 | 1,8104 | 1,7264 | 1,5057 |
| | OpenCL | | | | | | | | 8,299 |

**Tabelle 5-12: Messergebnisse für Heat (Messreihe 3)**

Anhand der Ergebnisse von Heat lassen sich in Bezug auf diesen Algorithmus insgesamt zwei Beobachtungen tätigen:

- Der Speedup pro Prozessoranzahl sinkt mit zunehmender Gittergröße

Heat ist eine sehr speicherlastige Anwendung, bei der das enorme Ungleichgewicht zwischen Prozessorleistung und Speichergeschwindigkeit deutlich wird. Durch die wachsende Gittergröße können die benötigten Daten nicht mehr komplett im Cache der Prozessoren gehalten werden, sondern erfordern in jeder Iteration ein erneutes Einlesen aus dem Hauptspeicher. Würde eine Initialisierung der Daten nicht, wie in der Einführung beschrieben, parallel erfolgen, wären die Ergebnisse noch schlechter aufgrund der höheren Zugriffskosten auf Speicherbänke anderer NUMA-Knoten.

- Der Speedup steigt nur langsam an bzw. stagniert

Alle Ansätze skalieren nur unzureichend mit einer wachsenden Anzahl an Prozessoren bzw. Prozessorkernen. Wird OpenCL gesondert betrachtet, dann schneidet OpenMP in allen Messgrößen mit einem maximalen Speedup von 4,8 am besten ab, gefolgt von Intel Cilk++ und der CRT von Microsoft. Ähnlich wie bei der Minimalgerüstbestimmung ist, neben der Limitierung durch den Speicher, die Form des Algorithmus ein weiterer Grund für den Schlechten Speedup. In Heat existieren drei parallele For-Schleifen innerhalb einer sequenti-

ellen For-Schleife mit insgesamt 5.000 Iterationen. Dadurch müssen insgesamt 15.000-Mal Threads erstellt und die Arbeit aufgeteilt werden. Bei OpenMP ließe sich die Anzahl auf 10.000 verringern, da zwei aufeinanderfolgende For-Schleifen innerhalb einer parallelen Region parallelisiert wurden.

Bei der Betrachtung der Ergebnisse von UPC fallen ungewöhnlich hohe Laufzeiten auf. Es wurde bei der Erzeugung der Daten darauf geachtet, dass jedem Thread ein kontinuierlicher Daten-Block zugeordnet wird, auf dem er, ohne viel mit anderen Threads zu kommunizieren, arbeiten kann. Lediglich an Randbereichen eines Blocks ist zur Bestimmung des Hitzewertes die Kommunikation mit dem Nachbarblock erforderlich. Trotz der Ausnutzung von Lokalität an entsprechenden Stellen erzeugt die threadübergreifende Kommunikation, wie auch bei der MST-Berechnung, sehr viel Overhead.

Die insgesamt beste Leistung lieferte, wie auch schon bei Mandelbrot, OpenCL. Durch den hohen Grad an Datenparallelität und der höheren Speicherbandbreite stieg der Speedup im Vergleich zu den anderen Ansätzen bei wachsender Gittergröße sogar weiter an. Trotz der höheren Speicheranbindung im Vergleich zur CPU stellt diese auch hier den limitierenden Faktor dar. Die Laufzeit ließe sich durch Ausnutzung des lokalen Speichers der Grafikkarte weiter verbessern. Dies konnte jedoch aufgrund des höheren zeitlichen Aufwandes nicht durchgeführt werden.

### 5.4.4 Matrix-Vektor Multiplikation

Bei der Matrix-Vektor Multiplikation wird eine spärlich besetzte Matrix mit einem Vektor multipliziert. Die Matrizen für die Messreihen stammen von Matrix Market und der University of Florida, deren wesentliche Eigenschaften in Tabelle 5-13 aufgelistet sind. Der Vektor wird durch das Programm erzeugt und enthält als Wert den jeweiligen Index der Zeile.

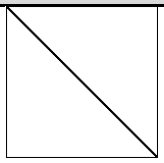
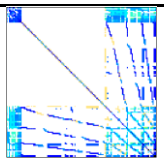
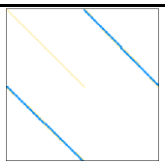
| Name | Dimension | Nicht-Null-Elemente | Datentyp | Plot |
|---|---|---|---|---|
| s3dkq4m2.mtx | 90.449 * 90.449 | 2.455.670 | Double |  |
| Freescale1.mtx | 3.428.755 * 3.428.755 | 17.052.626 | Double |  |
| nlpkkt160.mtx | 8.345.600 * 8.345.600 | 225.422.112 | Double |  |

**Tabelle 5-13: Charakteristiken der genutzten Matrizen**

Für das Einlesen der Matrizen wurde eine eigene Bibliothek geschrieben, die Matrizen mit Hilfe der Matrix Market I/O-Bibliothek[95] einliest und anschließend in das CSR-Format konvertiert. Das CSR-Format wurde gewählt, weil es sich für spärlich besetzte Matrizen im wissenschaftlichen Bereich durchgesetzt hat. [Bul09] Da die Matrix Market I/O-Bibliothek nur für C, FORTRAN und Matlab angeboten wird, musste auf Messungen mit Chapel an dieser Stelle verzichtet werden.

Neben den insgesamt fünf Messdurchläufen werden pro Durchgang je nach Größe der Matrix: 5.000, 500 und 50 Iterationen der Multiplikation zur Steigerung der Messgenauigkeit durchgeführt und anschließend der Mittelwert gebildet.

[95] http://math.nist.gov/MatrixMarket/mmio-c.html

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 0,0052 | | | | | | | |
| | OpenMP | | 0,0033 | 0,0025 | 0,0020 | 0,0017 | 0,0015 | 0,0014 | 0,0013 |
| | Intel TBB | | 0,0038 | 0,0026 | 0,0021 | 0,0018 | 0,0016 | 0,0015 | 0,0013 |
| | MS CRT | 0,0050 | 0,0052 | 0,0035 | 0,0031 | 0,0026 | 0,0032 | 0,0030 | 0,0031 |
| | Intel Cilk++ | | 0,0036 | 0,0026 | 0,0020 | 0,0018 | 0,0016 | 0,0014 | 0,0013 |
| | UPC | | 0,0142 | 0,0145 | 0,0093 | 0,0105 | 0,0091 | 0,0089 | 0,0059 |
| | OpenCL | | | | | | | | 0,0040 |
| Speedup | OpenMP | | 1,5742 | 2,0755 | 2,6695 | 2,9958 | 3,5454 | 3,6782 | 4,1180 |
| | Intel TBB | | 1,3651 | 1,9886 | 2,5162 | 2,9557 | 3,2007 | 3,4672 | 3,9546 |
| | MS CRT | | 0,9616 | 1,4036 | 1,6159 | 1,9126 | 1,5537 | 1,6547 | 1,6076 |
| | Intel Cilk++ | | 1,4572 | 2,0240 | 2,5753 | 2,9682 | 3,3536 | 3,6327 | 3,9131 |
| | UPC | | 0,3667 | 0,3600 | 0,5609 | 0,4990 | 0,5738 | 0,5845 | 0,8781 |
| | OpenCL | | | | | | | | 1,3193 |

**Tabelle 5-14: Messergebnisse für die Matrix-Vektor Multiplikation (Messreihe 1)**

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 0,0688 | | | | | | | |
| | OpenMP | | 0,0495 | 0,0415 | 0,0346 | 0,0305 | 0,0274 | 0,0256 | 0,0247 |
| | Intel TBB | | 0,0444 | 0,0332 | 0,0262 | 0,0227 | 0,0204 | 0,0187 | 0,0175 |
| | MS CRT | 0,0633 | 0,0835 | 0,0565 | 0,0432 | 0,0378 | 0,0331 | 0,0304 | 0,0278 |
| | Intel Cilk++ | | 0,0439 | 0,0320 | 0,0256 | 0,0220 | 0,0195 | 0,0180 | 0,0169 |
| | UPC | | 0,1488 | 0,1673 | 0,1065 | 0,1323 | 0,1225 | 0,1318 | 0,1146 |
| | OpenCL | | | | | | | | 0,3641 |
| Speedup | OpenMP | | 1,3903 | 1,6573 | 1,9865 | 2,2538 | 2,5132 | 2,6895 | 2,7828 |
| | Intel TBB | | 1,5490 | 2,0714 | 2,6290 | 3,0287 | 3,3719 | 3,6716 | 3,9277 |
| | MS CRT | | 0,7588 | 1,1218 | 1,4646 | 1,6735 | 1,9153 | 2,0824 | 2,2784 |
| | Intel Cilk++ | | 1,5665 | 2,1463 | 2,6884 | 3,1249 | 3,5284 | 3,8185 | 4,0777 |
| | UPC | | 0,4620 | 0,4110 | 0,6455 | 0,5198 | 0,5615 | 0,5218 | 0,6001 |
| | OpenCL | | | | | | | | 0,1889 |

**Tabelle 5-15: Messergebnisse für die Matrix-Vektor Multiplikation (Messreihe 2)**

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 0,2875 | | | | | | | |
| | OpenMP | | 0,3097 | 0,2134 | 0,1820 | 0,1438 | 0,1260 | 0,1131 | 0,1008 |
| | Intel TBB | | 0,1858 | 0,1388 | 0,1116 | 0,0974 | 0,0874 | 0,0806 | 0,0729 |
| | MS CRT | 0,2693 | 0,3323 | 0,2166 | 0,1809 | 0,1474 | 0,1226 | 0,1039 | 0,0986 |
| | Intel Cilk++ | | 0,1810 | 0,1327 | 0,1068 | 0,0918 | 0,0811 | 0,0754 | 0,0707 |
| | UPC | | 0,7076 | 0,7775 | 0,5185 | 0,8646 | 7,5114 | ---------- | ---------- |
| | OpenCL | | | | | | | | 2,1181 |
| Speedup | OpenMP | | 0,9283 | 1,3473 | 1,5793 | 1,9992 | 2,2824 | 2,5426 | 2,8534 |
| | Intel TBB | | 1,5478 | 2,0720 | 2,5767 | 2,9524 | 3,2887 | 3,5649 | 3,9411 |
| | MS CRT | | 0,8106 | 1,2436 | 1,4889 | 1,8276 | 2,1973 | 2,5923 | 2,7325 |
| | Intel Cilk++ | | 1,5883 | 2,1667 | 2,6926 | 3,1333 | 3,5471 | 3,8113 | 4,0683 |
| | UPC | | 0,4063 | 0,3698 | 0,5545 | 0,3325 | 0,0383 | ---------- | ---------- |
| | OpenCL | | | | | | | | 0,1357 |

**Tabelle 5-16: Messergebnisse für die Matrix-Vektor Multiplikation (Messreihe 3)**

Die Matrix-Vektor Multiplikation ist, ähnlich wie Heat, sehr von der Speicherbandbreite und einer guten Verteilung der Daten abhängig. Im Vorfeld wurde beschrieben, dass für die Ansätze: OpenMP, Intel TBB und Intel Cilk++ auf Funktionen der NUMA-Bibliothek zur Allokation der Daten zurückgegriffen wurde, da das Konvertieren der Matrix in das CSR-Format nicht parallel erfolgt, und eine parallele Initialisierung deshalb nicht gewährleistet ist. Aufgrund der Beschränkung der NUMA-Bibliothek auf Linux-Systeme können anhand der Messungen unter Linux und Windows sehr gut die Größenordnungen festgehalten werden, die eine optimierte Speicherverteilung mit sich bringt. Beim direkten Vergleich zwischen Intel TBB (mit NUMA) und der CRT (ohne NUMA) reichen die Geschwindigkeitsvorteile von 13 bis hin zu 131 Prozent. Obwohl auch OpenMP von dieser optimierten Speicherverteilung profitieren müsste, trifft dies nur auf die Ergebnisse der ersten Messreihe zu. Bei zunehmender Größe der Matrix verschlechtern sich die Laufzeiten verhältnismäßig, so dass OpenMP bei der dritten Messreihe sogar hinter die CRT zurückfällt. OpenMP kann demnach nicht in allen Situationen Vorteile aus der zyklischen Datenverteilung durch die NUMA-Bibliothek ziehen. Ich vermute an dieser Stelle eine nicht optimale Zuteilung der Iterationen auf die beteiligten Prozessoren bzw. Prozessorkernen. Dies bestätigt auch eine Analyse mit dem Profiling-Tool ompP für 8 Kerne. Bei der größten Matrix sind die Threads 1-4 bereits nach 0,6 Sekunden mit den 50 Multiplikationsoperationen fertig, wohingegen die Thread 5-8 insgesamt 4,6 Sekunden benötigen. Wird der von ompP errechnete Overhead von ca. 26 Prozent von der gesamten Laufzeit von 0,1008 Sekunden abgezogen, ist das Ergebnis vergleichbar mit dem von TBB

und Cilk++. Bei diesen Ansätzen wird ein besserer Speedup zusätzlich durch das geringe Arbeitsvolumen verhindert, bei dem sich der parallele Overhead zu stark bemerkbar macht.

Die Ergebnisse von UPC und OpenCL bedürfen weiterer Erläuterung, da bei diesen beiden Ansätzen aufgrund ihrer Natur eine etwas andere Vorgehensweise zur Berechnung genutzt wird. Aufgrund des SPMD-Konzeptes von UPC muss jeder Thread dafür sorgen, die entsprechenden Matrizen einzulesen. Ein Einlesen durch den Master-Thread mit anschließender Verteilung auf die anderen Threads dauerte zu lange, da Zugriffe auf Speicherbereiche anderer Threads mit höheren Kosten verbunden sind. Aus diesem Grund hält jeder Thread eine Repräsentation der Matrix im Speicher vor. Lediglich bei dem zu multiplizierenden Vektor und dem Ergebnisvektor erfolgt die Speicherung verteilt. Die Erzeugung bzw. Berechnung der Vektoren übernimmt anschließend eine parallele `upc_forall`-Schleife. Der Nachteil dieser Vorgehensweise macht sich in einem sehr hohen Speicherverbrauch bemerkbar, so dass unter Verwendung der größten Matrix und einer bestimmten Anzahl an Prozessoren das System auf den Swap-Speicher ausweichen muss. Dies resultiert in erheblich höheren Laufzeiten (siehe 6 Prozessoren für Messreihe 3). Deshalb wurde auf weitere Messungen mit mehr Prozessoren verzichtet. Bei den restlichen Messungen liegen die Laufzeiten häufig über denen der anderen Ansätze, was ich mit generell höheren Kosten beim Zugriff auf Elemente des verteilten Speichers begründe[96].

Die schlechten Ergebnisse von OpenCL lassen sich auf folgende Punkte zurückführen:

- Maximal zur Verfügung stehender Speicher von 256 MB
- Speicherbandbreite / Latenz
- Overhead durch Kopieroperationen zwischen Host und Grafikkarte

Die Speicherbeschränkung von ATI-Grafikkarten unter OpenCL auf 256 MB macht sich bei den Messreihen 2 und 3 bemerkbar. Bei diesen Messläufen lassen sich die Matrizen nicht komplett im Speicher der Grafikkarte vorhalten, sondern werden dynamisch durch den Host nachgeladen. Dies resultiert in einem höheren Berechnungs- und Kopieraufwand.

[96] Auch beim Zugriff auf lokale Speicherbereiche eines verteilten Feldes entsteht zusätzlicher Overhead durch notwendige Überprüfungen, ob der Speicherbereich wirklich dem jeweiligen Thread zugeordnet ist.

Die Matrix aus der ersten Messreihe passt hingegen komplett in den Speicher. Durch die insgesamt 5.000 Iterationen verschwindet auch der Einfluss durch die zwei notw. Kopierphasen[97].

| Einbeziehung des Kopieraufwandes | Laufzeit bei 1. Iteration (Messreihe 1) |
|---|---|
| **Ja** | 0,069237 Sekunden |
| **Nein** | 0,034681 Sekunden |

**Tabelle 5-17: Overhead durch Speicheroperationen bei OpenCL**

Tabelle 5-17 enthält deshalb Messungen mit nur einer Iteration, die die benötigte Zeit für die Kopieroperationen verdeutlichen soll. Dabei ist zu erkennen, dass die Kopieraktionen dieselbe Zeit benötigten wie die eigentliche Berechnung, so dass sich dieser zusätzliche Aufwand erst bei aufwendigeren Algorithmen auszahlt. Weiterhin verringerte sich die Laufzeit bei mehrmaliger Ausführung um den Faktor 8,6 auf 0,004 Sekunden. Dies konnte in der Größenordnung nur bei OpenCL beobachtet werden. Bei den anderen Ansätzen sorgten die Iterationen nur für minimale Verbesserungen und trugen zur Stabilisierung des Messwertes bei. Daraus lässt sich schließen, dass weiterer Overhead mit der initialen Ausführung einer Kernel-Funktion[98] verbunden ist und nachfolgende Aufrufe schneller ausgeführt werden können.

Analog zu Heat lässt sich auch bei der Matrix-Vektor Multiplikation die Laufzeit mit entsprechendem Aufwand durch Ausnutzung des lokalen Speichers weiter verbessern.

### 5.4.5 Quicksort

Der Sortieralgorithmus Quicksort zählt zu den „Divide and Conquer“ Verfahren, bei denen die zu bearbeitenden Elemente in zwei Mengen aufgeteilt und der Algorithmus rekursiv auf diesen beiden Mengen angewendet wird. Der Teilschritt erfolgt durch eine Partitionierung, bei der ein Element (das *Pivot-Element*) nach einem bestimmten Verfahren ausgewählt wird und welche die Elemente der Menge auf Basis des Pivot-Elements sortiert. Alle Elemente kleiner als das Pivot-Element kommen in die erste Menge und alle restlichen in die zweite Menge. Anschließend wird der Algorithmus rekursiv auf den beiden Mengen angewendet bis am Ende nur noch ein Element pro Menge übrig ist. [Gra03]

[97] 1 Phase: Matrix & Vektor zur Grafikkarte übertragen; 2 Phase: Ergebnisvektor zum Host übertragen
[98] Als Kernel werden die auf der Grafikkarte ausgeführten Prozeduren bezeichnet.

Quicksort wurde im Rahmen dieser Arbeit auf zwei verschiedene Arten implementiert. Zu Beginn sollen hier die Ergebnisse der simpleren Variante betrachtet werden, bei der der Partitionierungsschritt zu Anfang sequentiell erfolgt und die rekursiven Aufrufe durch parallele Tasks realisiert sind. Der Nachteil an dieser Vorgehensweise ist, dass sich sämtliche Prozessoren erst mit zunehmender Iterationstiefe auslasten lassen. Aufgrund der Eigenschaften und Beschränkungen von UPC und OpenCL konnte eine Implementierung in diesen Ansätzen nicht ohne umfassende Änderungen durchgeführt werden, so dass auf Messungen mit der simplen Variante verzichtet und stattdessen die erweiterte Variante genutzt wurde.

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 16,1346 | | | | | | | |
| | OpenMP | | 8,5199 | 5,9411 | 4,6576 | 3,9675 | 3,5395 | 3,2382 | 3,0341 |
| | Intel TBB | | 8,8413 | 6,2862 | 4,8015 | 4,1165 | 3,6330 | 3,3414 | 3,0691 |
| | MS CRT | 14,8696 | 7,8837 | 5,4665 | 4,3349 | 3,5862 | 3,2083 | 2,8845 | 2,7023 |
| | Intel Cilk++ | | 9,3956 | 6,5327 | 5,2608 | 4,4683 | 3,9854 | 3,6327 | 3,3639 |
| | Chapel | | 113,692 | 73,4160 | 68,1665 | 63,8575 | 58,4871 | 42,1736 | 45,5224 |
| Speedup | OpenMP | | 1,8937 | 2,7158 | 3,4641 | 4,0666 | 4,5585 | 4,9825 | 5,3178 |
| | Intel TBB | | 1,8249 | 2,5667 | 3,3603 | 3,9195 | 4,4411 | 4,8287 | 5,2571 |
| | MS CRT | | 1,8861 | 2,7201 | 3,4302 | 4,1464 | 4,6347 | 5,1550 | 5,5026 |
| | Intel Cilk++ | | 1,7172 | 2,4698 | 3,0670 | 3,6109 | 4,0484 | 4,4415 | 4,7964 |
| | Chapel | | 0,1419 | 0,2198 | 0,2367 | 0,2527 | 0,2759 | 0,3826 | 0,3544 |

**Tabelle 5-18: Messergebnisse für Quicksort (Messreihe 1)**

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 41,9907 | | | | | | | |
| | OpenMP | | 21,9084 | 15,0994 | 11,7278 | 9,8118 | 8,6108 | 8,0908 | 7,5653 |
| | Intel TBB | | 22,0937 | 15,5705 | 11,7860 | 9,8939 | 8,4958 | 7,5485 | 6,9206 |
| | MS CRT | 39,4590 | 20,9825 | 14,4641 | 11,2809 | 9,5319 | 8,2901 | 7,3763 | 6,6809 |
| | Intel Cilk++ | | 24,2162 | 16,8519 | 13,0530 | 10,8050 | 9,4180 | 8,5291 | 7,6813 |
| | Chapel | | 295,983 | 233,246 | 179,157 | 189,138 | 123,380 | 115,920 | 101,529 |
| Speedup | OpenMP | | 1,9166 | 2,7810 | 3,5805 | 4,2796 | 4,8765 | 5,1899 | 5,5504 |
| | Intel TBB | | 1,9006 | 2,6968 | 3,5628 | 4,2441 | 4,9425 | 5,5628 | 6,0675 |
| | MS CRT | | 1,8806 | 2,7281 | 3,4979 | 4,1397 | 4,7598 | 5,3494 | 5,9063 |
| | Intel Cilk++ | | 1,7340 | 2,4917 | 3,2170 | 3,8862 | 4,4585 | 4,9232 | 5,4666 |
| | Chapel | | 0,1419 | 0,1800 | 0,2344 | 0,2220 | 0,3403 | 0,3622 | 0,4136 |

**Tabelle 5-19: Messergebnisse für Quicksort (Messreihe 2)**

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 88,2963 | | | | | | | |
| | OpenMP | | 48,4926 | 31,3590 | 24,7788 | 21,1617 | 18,7238 | 17,1439 | 15,9102 |
| | Intel TBB | | 47,7450 | 33,8690 | 25,1868 | 20,3433 | 18,7478 | 17,0379 | 15,2135 |
| | MS CRT | 81,7737 | 43,0468 | 29,2613 | 22,4931 | 18,8901 | 16,6439 | 15,2772 | 14,1402 |
| | Intel Cilk++ | | 56,2343 | 34,9890 | 26,9080 | 22,7384 | 19,6652 | 17,4838 | 16,2906 |
| | Chapel | | 599,045 | 482,216 | 413,391 | 273,992 | 230,741 | 211,680 | 205,917 |
| Speedup | OpenMP | | 1,8208 | 2,8157 | 3,5634 | 4,1724 | 4,7157 | 5,1503 | 5,5497 |
| | Intel TBB | | 1,8493 | 2,6070 | 3,5057 | 4,3403 | 4,7097 | 5,1824 | 5,8038 |
| | MS CRT | | 1,8996 | 2,7946 | 3,6355 | 4,3289 | 4,9131 | 5,3527 | 5,7830 |
| | Intel Cilk++ | | 1,5702 | 2,5235 | 3,2814 | 3,8831 | 4,4900 | 5,0502 | 5,4201 |
| | Chapel | | 0,1474 | 0,1831 | 0,2136 | 0,3223 | 0,3827 | 0,4171 | 0,4288 |

**Tabelle 5-20: Messergebnisse für Quicksort (Messreihe 3)**

Quicksort ist die erste Anwendung, in denen die Taskparallelität der ausgewählten Ansätze geprüft wird. Unter Berücksichtigung des zu Anfangs sequentiellen Anteils durch die Partitionierung erzielen alle Ansätze, mit Ausnahme von Chapel, gute Werte im Speedup. Weitere Steigerungen sind durch den relativ hohen sequentiellen Anteil nicht mehr zu erwarten. Bei Chapel besteht im Bereich der Taskparallelität noch sehr großer Optimierungsbedarf, wie anhand der Laufzeiten deutlich wird.

Unter Windows lief der Algorithmus generell etwas schneller. Deshalb verfügt die CRT von Microsoft durchgängig über die kürzesten Laufzeiten. Beim Speedup wechseln sich OpenMP, Intel TBB und die CRT je nach Messreihe ab. Am häufigsten erzielt jedoch auch hier die CRT die besten Speedup-Werte, gefolgt von OpenMP und Intel TBB.

Obwohl die CRT für diese Anwendung am besten abschneidet und die Tasks somit am effizientesten umsetzen kann, muss angemerkt werden, dass sich der Algorithmus mit der CRT ohne Änderungen nicht hätte ausführen lassen können. Das Gleiche gilt auch für TBB von Intel. Bei beiden Ansätzen war die Einführung eines Grenzwertes für die Anzahl an Elementen erforderlich, ab dem die Sortierung nicht weiter parallel mittels Tasks, sondern sequentiell erfolgt. Ohne diese Änderung stürzten die Anwendungen beider Ansätze ab einer gewissen Rekursionstiefe aufgrund eines *Stack-Overflows* ab. Die restlichen Ansätze hatten bzgl. der Anzahl an aktiven Tasks keine Probleme. Für TBB musste der Grenzwert bei der größten Messreihe auf 4096 Elemente, bei der CRT sogar auf 32768 festgesetzt werden[99]. Dies zeigt,

[99] Mittels eines Try & Error Verfahrens wurde sich in Schritten von Zweierpotenzen an die Werte herangetastet.

dass sowohl die Tasks der CRT als auch die der TBB mehr Overhead produzieren als bspw. die von OpenMP oder Cilk++. Das führt zu einer Beschränkung der maximalen Anzahl aktiver Tasks, welche bei der CRT am stärksten ausgeprägt ist. Da ein solcher Grenzwert aufgrund der geringeren Anzahl der zu erzeugenden Tasks für weniger Overhead sorgt, wurde die Grenze von 32768 Elementen zugunsten einer besseren Vergleichbarkeit für alle Ansätze übernommen.

Im Folgenden sollen die Ergebnisse der komplexeren Quicksort Variante vorgestellt werden, die auch die Partitionierung in mehreren Schritten parallel durchführen kann. Dazu wird zu Anfang ein globales Pivot-Element bestimmt. Jeder Thread sortiert anschließend den ihm zugeordneten Bereich in Elemente kleiner und größer gleich dem Pivot-Element. Daraufhin bildet der Algorithmus Prefixsummen, mit denen sich das gesamte Feld anschließend parallel umordnen lässt, so dass sich im ersten Teil ($S$) alle Elemente kleiner als das Pivot-Element und im zweiten Teil ($L$) alle Elemente größer oder gleich dem Pivot-Element befinden. In Abhängigkeit der Größe der Mengen $S$ und $L$ wird mit der Formel

$$p_S = \left\lfloor \frac{|S| * p}{|S| + |L|} + 0{,}5 \right\rfloor$$

die Anzahl an Prozessoren für den Block $S$ bestimmt. Entsprechend ist die Anzahl an Prozessoren für den Block $L$:

$$p_L = p - p_S$$

Der Algorithmus wird anschließend auf den Blöcken $S$ und $L$ mit der errechneten Anzahl an Prozessoren $p_S$ und $p_L$ rekursiv angewendet. Sinkt für einen der beiden Blöcke die Anzahl der Elemente unter $n/p$, wird dieser Block sequentiell sortiert und der Algorithmus ist für diesen Prozessor zu Ende. $n$ steht für die Anzahl an zu sortierenden Elementen.

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 16,1346 | | | | | | | |
| | OpenMP | | 9,9789 | 10,4917 | 10,0588 | 9,9677 | 10,0695 | 9,8650 | 9,9089 |
| | Intel TBB | | 10,2220 | 9,8749 | 7,4172 | 5,4980 | 4,5245 | 4,0738 | 3,6711 |
| | MS CRT | 14,8696 | 11,4087 | 6,9484 | 5,9821 | 5,1197 | 5,1012 | 4,4361 | 3,9282 |
| | Intel Cilk++ | | 10,5887 | 7,7055 | 5,5748 | 4,8198 | 4,4866 | 3,8164 | 3,4500 |
| | UPC | | 46,3658 | 50,3038 | 35,3566 | 41,4382 | 36,2731 | 31,0825 | 17,5973 |
| | Chapel | | 35,4692 | 28,2596 | 20,4251 | 18,2788 | 16,0575 | 13,4713 | 11,1795 |
| | OpenCL | | | | | | | | ---------- |
| Speedup | OpenMP | | 1,6169 | 1,5378 | 1,6040 | 1,6187 | 1,6023 | 1,6355 | 1,6283 |
| | Intel TBB | | 1,5784 | 1,6339 | 2,1753 | 2,9346 | 3,5661 | 3,9606 | 4,3950 |
| | MS CRT | | 1,3034 | 2,1400 | 2,4857 | 2,9044 | 2,9149 | 3,3520 | 3,7853 |
| | Intel Cilk++ | | 1,5238 | 2,0939 | 2,8942 | 3,3476 | 3,5962 | 4,2277 | 4,6766 |
| | UPC | | 0,3480 | 0,3207 | 0,4563 | 0,3894 | 0,4448 | 0,5191 | 0,9169 |
| | Chapel | | 0,4549 | 0,5709 | 0,7899 | 0,8827 | 1,0048 | 1,1977 | 1,4432 |
| | OpenCL | | | | | | | | ---------- |

**Tabelle 5-21: Messergebnisse für Parallel Quicksort (Messreihe 1)**

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 41,9907 | | | | | | | |
| | OpenMP | | 25,3041 | 24,6888 | 25,1023 | 26,5063 | 25,8938 | 28,5617 | 25,4051 |
| | Intel TBB | | 28,9144 | 22,7108 | 15,1677 | 14,6487 | 14,2674 | 10,8312 | 9,2543 |
| | MS CRT | 39,4590 | 28,6763 | 24,3137 | 17,0954 | 14,1318 | 12,2585 | 12,3438 | 11,8667 |
| | Intel Cilk++ | | 26,5997 | 20,7401 | 15,8966 | 12,4891 | 11,6138 | 9,6418 | 9,5424 |
| | UPC | | 141,118 | 121,455 | 87,4270 | 104,958 | 95,3047 | 89,1616 | 46,2445 |
| | Chapel | | 90,3341 | 67,9547 | 50,4683 | 44,1735 | 39,8358 | 35,0836 | 28,8580 |
| | OpenCL | | | | | | | | ---------- |
| Speedup | OpenMP | | 1,6594 | 1,7008 | 1,6728 | 1,5842 | 1,6217 | 1,4702 | 1,6528 |
| | Intel TBB | | 1,4522 | 1,8489 | 2,7684 | 2,8665 | 2,9431 | 3,8768 | 4,5374 |
| | MS CRT | | 1,3760 | 1,6229 | 2,3082 | 2,7922 | 3,2189 | 3,1967 | 3,3252 |
| | Intel Cilk++ | | 1,5786 | 2,0246 | 2,6415 | 3,3622 | 3,6156 | 4,3551 | 4,4004 |
| | UPC | | 0,2976 | 0,3457 | 0,4803 | 0,4001 | 0,4406 | 0,4710 | 0,9080 |
| | Chapel | | 0,4648 | 0,6179 | 0,8320 | 0,9506 | 1,0541 | 1,1969 | 1,4551 |
| | OpenCL | | | | | | | | ---------- |

**Tabelle 5-22: Messergebnisse für Parallel Quicksort (Messreihe 2)**

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 88,2963 | | | | | | | |
| | OpenMP | | 55,2911 | 52,9579 | 50,4252 | 56,6377 | 56,5122 | 52,1547 | 54,2023 |
| | Intel TBB | | 59,5235 | 45,3908 | 36,1218 | 36,7068 | 27,9812 | 21,5649 | 19,4900 |
| | MS CRT | 81,7737 | 69,5927 | 42,5094 | 37,5923 | 28,2195 | 23,4374 | 24,3143 | 19,9115 |
| | Intel Cilk++ | | 52,5729 | 43,9244 | 29,3490 | 26,3384 | 24,6887 | 21,6713 | 19,7489 |
| | UPC[100] | | ---------- | ---------- | 189,436 | 135,139 | 212,842 | 190,154 | 94,6845 |
| | Chapel | | 181,618 | 145,911 | 105,584 | 94,3561 | 81,6407 | 75,7076 | 70,9443 |
| | OpenCL | | | | | | | | ---------- |
| Speedup | OpenMP | | 1,5969 | 1,6673 | 1,7510 | 1,5590 | 1,5624 | 1,6930 | 1,6290 |
| | Intel TBB | | 1,4834 | 1,9452 | 2,4444 | 2,4054 | 3,1556 | 4,0944 | 4,5303 |
| | MS CRT | | 1,1750 | 1,9237 | 2,1753 | 2,8978 | 3,4890 | 3,3632 | 4,1068 |
| | Intel Cilk++ | | 1,6795 | 2,0102 | 3,0085 | 3,3524 | 3,5764 | 4,0743 | 4,4709 |
| | UPC[100] | | ---------- | ---------- | 0,4661 | 0,6534 | 0,4148 | 0,4643 | 0,9325 |
| | Chapel | | 0,4862 | 0,6051 | 0,8363 | 0,9358 | 1,0815 | 1,1663 | 1,2446 |
| | OpenCL | | | | | | | | ---------- |

**Tabelle 5-23: Messergebnisse für Parallel Quicksort (Messreihe 3)**

Als Bezugsgröße für den Speedup wurde die sequentielle Laufzeit der simpleren Quicksort-Variante herangezogen. Anhand der Messwerte ist zu erkennen, dass lediglich Chapel von dem komplexeren Algorithmus profitiert. Hier konnten die Laufzeiten im Vergleich zur simpleren Variante verringert werden. Bei den Ansätzen OpenMP, Intel TBB, der CRT und Intel Cilk++ verschlechterte sich die Laufzeit und auch der Speedup. Dies führe ich auf die allgemein höhere Komplexität des Algorithmus zurück, bei dem sich der Vorteil durch die parallele Partitionierung durch den höheren Aufwand nicht amortisiert. Weiterhin scheint das Task-Modell von OpenMP für diese Art von Algorithmus nicht geeignet zu sein. Hier muss es an dieser Stelle bei einer Vermutung bleiben, da eine genauere Untersuchung dieses Problems durch das Profiling-Tool ompP nicht möglich war. Die zwingend benötigte Unterstützung zur Untersuchung von verschachtelter Parallelität mittels Tasks ist in der verwendeten Version 0.7.1 noch nicht gegeben.

Auch bei UPC lassen sich ungewöhnlich hohe Laufzeiten beobachten, was ich zum Teil auf die nur rudimentär vorhandene Unterstützung für die Entwicklung taskparalleler Anwendungen aufgrund des SPMD-Konzepts zurückführe. Die Synchronisation stellte sich dabei als das größte Problem heraus, da mit UPC nur globale Barrier-Operationen möglich sind. Somit

[100] Betroffene Messungen waren aufgrund einer Beschränkung der Größe des verteilten Speichers nicht durchführbar. Eine manuelle Vergrößerung durch den Parameter `-shared-heap=xMB` brachte ebenfalls keinen Erfolg.

können Teilmengen von Threads nicht mit den zur Verfügung gestellten Mitteln synchronisiert werden. Ein weiterer Grund für die hohen Laufzeiten ist mit den hohen Kosten für die threadübergreifende Kommunikation verbunden. Da bei Quicksort vorher nicht bekannt ist, welchem Thread welche Bereiche des zu sortierenden Arrays zugeordnet werden, erfolgte die Verteilung der Elemente auf die Threads in zyklischer Form.

Die Implementierung in OpenCL stellte sich als eine große Herausforderung dar, da die Hardware-Architektur sich wesentlich von der bisherigen unterscheidet und die Möglichkeiten bzgl. des Debuggings zurzeit noch sehr begrenzt sind. Dies spiegelt sich auch in den erreichten Laufzeiten wieder, die in Tabelle 5-24 dargestellt sind. Eine gesonderte Darstellung wurde deshalb gewählt, weil alle Datenmengen ggü. denen der eigentlichen Messreihen um den Faktor 100 geringer sind. Messungen mit den ursprünglich ausgewählten Größen wurden aufgrund der erwarteten hohen Laufzeiten nicht durchgeführt.

| 1.000.000 Elemente | 2.500.000 Elemente | 5.000.000 Elemente |
|---|---|---|
| 22,4673 Sek. | 55,7808 Sek. | 121,9263 Sek. |

**Tabelle 5-24: Gesonderte Messungen von Parallel Quicksort mit OpenCL**

Zum Vergleich: Die Cilk++-Implementierung benötigt für 5.000.000 Elemente lediglich 0,14 Sekunden. Dies zeigt, dass selbst die Programmierung bekannter Suchalgorithmen auf Grafikkarten eine ganz andere Herangehensweise erfordert und eine, wenn auch nicht triviale Portierung eines für CPUs ausgerichteten Algorithmus, nicht die gewünschten Ergebnisse liefert. Ein weiterer Grund für das schlechte Abschneiden kann dem SIMT-Konzept[101] in Kombination mit der Taskparallelität geschuldet sein, bei dem die Anweisungen der Threads innerhalb einer Wavefront[102] immer gemeinsam ausgeführt werden müssen. Eine tiefere Einarbeitung hätte an dieser Stelle jedoch zu viel Zeit gekostet. Deshalb wird auf die entsprechende Literatur verwiesen. Bspw. beschreibt [Sat09] die Implementierung des Merge Sort Algorithmus für Grafikkarten, mit dem 12.000.000 Elemente in ca. 0,25 Sekunden sortiert werden konnten[103]. Die Quicksort bzw. Parallel Quicksort Implementierung benötigt mit Cilk++ für die gleiche Anzahl von Elementen 0,34 bzw. 0,37 Sekunden. GPU-ABiSort ist ein

[101] Single Instruction Multiple Threads
[102] Als Wavefront bezeichnet ATI zu einer Gruppe zusammengefasste Threads, siehe S. 10.
[103] Unter Verwendung einer Geforce GTX 280 und CUDA – ohne Berücksichtigung des Kopieraufwandes.

weiteres Sortierverfahren für Grafikkarten, das in [Gre06] beschrieben ist und auf einem bitonischen Sortieransatz aufbaut.

### 5.4.6 NPB-CG

Als letzte Anwendung wird in diesem Abschnitt der Conjugate Gradient (CG) Benchmark aus den *NAS Parallel Benchmarks* (NPB) betrachtet. CG berechnet eine approximative Lösung des Gleichungssystems

$$Az = x$$

für die spärlich besetzte Matrix $A$ und den Vektor $x$ mit $x = (1,1, \dots ,1)^T$. [Bai94] Die genaue Vorgehensweise zur Approximation kann [Bai94] entnommen werden.

Der Aufwand der Berechnung hängt von der verwendeten Klasse ab. Für die folgenden Untersuchungen wurden die Klassen A, B und C herangezogen, die unter anderem die Größe der Matrix $A$ sowie die durchzuführende Anzahl an Iterationen festlegen (siehe Tabelle 5-25).

| Klasse | Größe der Matrix | Iterationen |
|---|---|---|
| A | 1.400 * 1.400 | 15 |
| B | 75.000 * 75.000 | 75 |
| C | 150.000 * 150.000 | 75 |

**Tabelle 5-25: Klassen von CG**

Als Basis für die Implementierungen in OpenMP, Intel TBB, der CRT, Intel Cilk++, Chapel sowie für die sequentielle Variante dient der Quellcode des CG-Benchmarks aus dem Omni OpenMP Compiler Project[104]. Das Omni Project basiert auf der Version 2.3 der NPB-Suite und hat den Vorteil, dass sämtliche Benchmarks für die Sprache C mit OpenMP-Erweiterungen vorliegen[105]. Dies ermöglicht eine leichtere Portierung für die oben genannten Ansätze. UPC nutzt eine Implementierung der George Washington University, die ebenfalls auf der OpenMP Version des Omni Projects basiert, allerdings die Änderungen von Version 2.4 der originalen NPB-Suite übernommen hat[106]. Bis auf eine zusätzliche Klasse D gibt

[104] http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp/download/download-benchmarks.html
[105] Der CG-Benchmark aus der originalen NAS Parallel Benchmark Suite ist in FORTRAN geschrieben.
[106] http://threads.hpcl.gwu.edu/sites/npb-upc

es für den CG-Benchmark jedoch keine weiteren Änderungen zwischen Version 2.3 und 2.4. [NAS10] Eine Implementierung für OpenCL konnte aufgrund von Fehlern im Treiber bzw. den OpenCL-Bibliotheken von AMD nicht durchgeführt werden. Hier verursachte die von AMD noch als experimentell bezeichnete Unterstützung für Floating-Point Werte mit doppelter Genauigkeit reproduzierbar Fehler im System-Kernel, die das gesamte System zum Absturz brachten. Dieses Verhalten konnte auch bei Mandelbrot und Heat beobachten werden, jedoch traten bei diesen Anwendungen die Abstürze erheblich seltener auf.

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 1,786 | | | | | | | |
| | OpenMP | | 0,986 | 0,678 | 0,606 | 0,592 | 0,506 | 0,492 | 0,496 |
| | Intel TBB | | 2,106 | 1,03 | 0,842 | 0,826 | 0,732 | 0,916 | 0,494 |
| | MS CRT | 1,92 | 1,39 | 1,006 | 0,926 | 1,058 | 1,022 | 1,224 | 1,166 |
| | Intel Cilk++ | | 1,052 | 0,788 | 0,664 | 0,598 | 0,544 | 0,536 | 0,49 |
| | UPC | | 0,896 | ---------- | 0,584 | ---------- | ---------- | ---------- | 0,362 |
| | Chapel | | 3,0175 | 2,3068 | 1,6829 | 1,7354 | 1,6625 | 2,0788 | 1,4771 |
| Speedup | OpenMP | | 1,8114 | 2,6342 | 2,9472 | 3,0169 | 3,5296 | 3,6301 | 3,6008 |
| | Intel TBB | | 0,8481 | 1,7340 | 2,1211 | 2,1622 | 2,4399 | 1,9498 | 3,6154 |
| | MS CRT | | 1,3813 | 1,9085 | 2,0734 | 1,8147 | 1,8787 | 1,5686 | 1,6467 |
| | Intel Cilk++ | | 1,6977 | 2,2665 | 2,6898 | 2,9866 | 3,2831 | 3,3321 | 3,6449 |
| | UPC | | 1,9933 | ---------- | 3,0582 | ---------- | ---------- | ---------- | 4,9337 |
| | Chapel | | 0,5919 | 0,7742 | 1,0612 | 1,0292 | 1,0743 | 0,8592 | 1,2092 |

**Tabelle 5-26: Messergebnisse für CG (Messreihe 1)**

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 108,592 | | | | | | | |
| | OpenMP | | 66,618 | 44,618 | 35,796 | 30,494 | 27,008 | 25,136 | 23,408 |
| | Intel TBB | | 76,288 | 49,438 | 36,636 | 32,188 | 26,53 | 25,042 | 23,488 |
| | MS CRT | 115,86 | 62,706 | 45,6 | 40,242 | 38,838 | 40,674 | 38,008 | 43,368 |
| | Intel Cilk++ | | 59,994 | 41,85 | 32,968 | 28,494 | 24,886 | 22,97 | 21,67 |
| | UPC | | 50,722 | ---------- | 32,162 | ---------- | ---------- | ---------- | 16,812 |
| | Chapel | | 121,839 | 90,0351 | 63,0142 | 62,0301 | 56,0768 | 56,0383 | 35,6449 |
| Speedup | OpenMP | | 1,6301 | 2,4338 | 3,0336 | 3,5611 | 4,0207 | 4,3202 | 4,6391 |
| | Intel TBB | | 1,4234 | 2,1965 | 2,9641 | 3,3737 | 4,0932 | 4,3364 | 4,6233 |
| | MS CRT | | 1,8477 | 2,5408 | 2,8791 | 2,9832 | 2,8485 | 3,0483 | 2,6716 |
| | Intel Cilk++ | | 1,8100 | 2,5948 | 3,2939 | 3,8110 | 4,3636 | 4,7276 | 5,0112 |
| | UPC | | 2,1409 | ---------- | 3,3764 | ---------- | ---------- | ---------- | 6,4592 |
| | Chapel | | 0,8913 | 1,2061 | 1,7233 | 1,7506 | 1,9365 | 1,9378 | 3,0465 |

**Tabelle 5-27: Messergebnisse für CG (Messreihe 2)**

| Ansatz \ Prozessoren | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | * |
|---|---|---|---|---|---|---|---|---|---|
| Laufzeiten | Sequentiell | 282,942 | | | | | | | |
| | OpenMP | | 196,402 | 134,036 | 105,548 | 87,236 | 76,298 | 68,94 | 64,954 |
| | Intel TBB | | 185,926 | 132,028 | 103,746 | 83,986 | 72,18 | 68,624 | 64,418 |
| | MS CRT | 327,98 | 170,05 | 125,642 | 112,254 | 100,226 | 92,524 | 86,766 | 90,466 |
| | Intel Cilk++ | | 168,954 | 115,026 | 91,302 | 78,736 | 68,564 | 64,286 | 60,262 |
| | UPC | | 150,792 | ---------- | 85,116 | ---------- | ---------- | ---------- | 47,732 |
| | Chapel | | 309,791 | 204,586 | 158,992 | 140,816 | 134,497 | 135,161 | 89,1202 |
| Speedup | OpenMP | | 1,4406 | 2,1109 | 2,6807 | 3,2434 | 3,7084 | 4,1042 | 4,3560 |
| | Intel TBB | | 1,5218 | 2,1430 | 2,7273 | 3,3689 | 3,9200 | 4,1231 | 4,3923 |
| | MS CRT | | 1,9287 | 2,6104 | 2,9218 | 3,2724 | 3,5448 | 3,7801 | 3,6255 |
| | Intel Cilk++ | | 1,6747 | 2,4598 | 3,0990 | 3,5936 | 4,1267 | 4,4013 | 4,6952 |
| | UPC | | 1,8764 | ---------- | 3,3242 | ---------- | ---------- | ---------- | 5,9277 |
| | Chapel | | 0,9133 | 1,3830 | 1,7796 | 2,0093 | 2,1037 | 2,0934 | 3,1748 |

**Tabelle 5-28: Messergebnisse für CG (Messreihe 3)**

Ausgangspunkt des CG-Benchmarks ist eine Matrix, so dass auch in diesem Fall die Speicherbandbreite und die Datenverteilung eine große Bedeutung für die erreichten Laufzeiten und Speedup-Werte haben. In allen Ansätzen werden die Daten auf gleiche Weise initialisiert. Bei UPC erfolgt die Initialisierung sequentiell auf allen Threads, die ihre Daten anschließend in verteilte Datenstrukturen kopieren. Bei den restlichen Ansätzen erfolgt die Initialisierung an allen möglichen Stellen parallel. Der eigentliche Benchmark besteht aus mehreren parallelen For-Schleifen und Reduktionsoperationen, die ansatzspezifisch umgesetzt wurden. Eine sequentielle äußere Schleife ruft entsprechend der verwendeten Klasse die Approximationsfunktion 15- oder 75-Mal auf.

Bis auf die CRT konnten alle Ansätze ihre Ausführungszeiten mit zunehmender Prozessoranzahl verringern. Ein linearer Speedup kann jedoch bei keinem Ansatz auch nur annähernd erreicht werden. In Bezug zu den drei Messreihen wird deutlich, dass der erreichte Speedup bei Messreihe 2 ggü. Messreihe 1 allgemein besser ist und bei Messreihe 3 wieder abfällt. Bei der ersten Messreihe ist das Arbeitsvolumen noch zu gering, um einen Nutzen aus den, mit zusätzlichem Overhead verbundenen, parallelen Methoden zu ziehen. Dass erneute Sinken des Speedups bei Messreihe 3 liegt an der Größe der verwendeten Matrix und der Limitierung durch die Speicherbandbreite. Dadurch können weniger Daten im Cache der Prozessoren vorgehalten werden und machen ein häufigeres Einlesen aus dem Arbeitsspeicher erforderlich.

Neben der Limitierung durch die Speicherbandbreite ist die immer nur kurz vorhandene Parallelität ein weiterer Grund für den schlechten Speedup. Aufgrund von Abhängigkeiten ist immer nur eine Parallelisierung einzelner Abschnitte möglich. Dies macht ein häufiges Erstellen von Threads mit anschließender Partitionierung erforderlich. Bei UPC fällt dieser Aufwand entsprechend geringer aus, da bereits von Beginn an Parallelität durch das SPMD-Konzept vorliegt und ein Erzeugen der Threads somit nicht notwendig ist. Besonders gut lässt sich dies bei der CRT beobachten. Da für die erste Messreihe nur sehr wenig Arbeit vorliegt, macht sich der zusätzliche Overhead bei mehr Prozessoren besonders deutlich bemerkbar. Mit zunehmender Größe der Daten wird der Speedup der CRT langsam besser, da der Overhead nicht mehr so stark ins Gewicht fällt. Alle anderen Ansätze weisen ein solches Verhalten nicht auf, so dass ich die Ursache nicht primär auf mögliche Engpässe beim Speicherzugriff, sondern auf den mit parallelen For-Schleifen und Reduktionen verbundenen Overhead innerhalb der CRT zurückführen würde. Ähnliche Ergebnisse kamen bereits bei der Berechnung des Minimalgerüstes zustande.

Die UPC-Implementierung der George Washington University funktioniert nach eigenen Versuchen nur mit einer Zweierpotenz als Prozessoranzahl. Das Problem liegt hier an der Aufteilung der Matrix in kleinere Bereiche mit anschließender Zuweisung auf die Threads, die bei einer Prozessoranzahl ungleich einer Zweierpotenz falsche Aufteilungen liefert. Aus diesem Grund musste auf die Messungen mit 3, 5, 6 und 7 Prozessoren verzichtet werden.

## 5.5 Bewertung der Ansätze

Den Abschluss bildet an dieser Stelle eine Bewertung der Ansätze. Dazu werden die Ergebnisse der bisherigen Untersuchungen in komprimierter Form tabellarisch dargestellt und unter Berücksichtigung der in Abschnitt 5.4 durchgeführten Laufzeitmessungen bewertet.

Die Kurzübersicht wird die wesentlichen Punkte abdecken, die unter den Anforderungen ausgearbeitet und in der Sichtung behandelt wurden. Diese umfassen den Abstraktionsgrad und die Erscheinungsform, die verfügbaren Ebenen der Parallelität, die Verständlichkeit, die Portabilität und die Kosten zur Verwendung eines Ansatzes. Die Definition der verschiedenen Punkte kann in den Kapiteln: *Eigenschaften von parallelen Programmiermodellen* (ab S. 3) und *Anforderungen an parallele Programmiermodelle* (ab S. 14) nachgelesen werden.

Zur Beurteilung der Literatur sowie zur Einschätzung der Verständlichkeit wird ein Punktesystem mit einer Skala von 1 – 4 Sternen genutzt. Die Einschätzung ist eine subjektive Messgröße und umfasst Aspekte der Verständlichkeit eines Ansatzes, die während der Einarbeitung und der Implementierung (sofern durchgeführt) gewonnen wurden. Je mehr Punkte vergeben werden, desto leichter fällt die Einarbeitung in einen Ansatz und seine Verwendung. Nach einem ähnlichen Schema wird auch die Verfügbarkeit an Literatur beurteilt. Hier richtet sich die Bewertung nach den gefundenen Quellen, die für die Sichtung der Ansätze herangezogen wurden. Mehr Punkte stehen für eine größere Anzahl an Quellen.

### 5.5.1 Ansätze für Systeme mit gemeinsamem Speicher

| Kriterium \ Ansatz | | Pthr. | Java | OMP | TBB | CRT | .NET | Cilk++ | Ct | GCD |
|---|---|---|---|---|---|---|---|---|---|---|
| Erscheinungsform | | Bibl. | Erw. | Erw. | Bibl. | Erw. | Erw. | Erw. | Erw. | Erw. |
| Abstraktionsgrad (Explizit / Implizit) | Angabe der Par. | E | E | E | E | E | E | E | E | E |
| | Zerlegung | E | E | E / I | E / I | E / I | E / I | E / I | I | E / I |
| | Verwaltung | E | E / I | I | I | I | I | I | I | I |
| | Abbildung | I | I | I | I | I | I | I | I | I |
| | Synchr. | E | E | E / I | E / I | E / I | E / I | E / I | I | E / I |
| Funkt. Umfang | Datenpar. | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Taskpar. | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ |
| | Pipelining | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| Verständlichkeit | Spezifikation öfftl. | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ |
| | Literatur (gedr.) | **** | **** | **** | ** | * | ** | * | * | * |
| | Literatur (online) | ** | *** | **** | *** | *** | *** | ** | * | *** |
| | Bes. Mögl. des Debug.,Testens, … | ✓ | ✓ | ✓ | ✗ | ~ [107] | ~ [107] | ✓ | ✗ | ✗ |
| | Bew. d. Autors | ** | *** | **** | ** | *** | *** | **** | ** | *** |
| Port. | Linux | ✓ | ✓ | ✓ | ✓ | ✗ | ~ [108] | ✓ | ✓ | ✗ |
| | Mac OS X | ✓ | ✓ | ✓ | ✓ | ✗ | ~ [108] | ✗ | ✗ | ✓ |
| | Windows | ~ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ |
| Ansatz frei verfügbar | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ |

**Legende:** *E: Explizit, I: Implizit, ✓: Verfügbar, ✗: Nicht verfügbar, ~: Unter Einschränkungen verfügbar*

**Tabelle 5-29: Gegenüberstellung der Ansätze für Systeme mit einem gemeinsamen Speicher**

Bei der Betrachtung der rein funktionalen Aspekte sind Intel TBB und die CRT von Microsoft am besten aufgestellt, da sie die meisten Parallelisierungsebenen abdecken. Darüber hinaus verfügen beide Ansätze über einen relativ hohen Grad an impliziter Parallelität, der nur von Intel Ct noch übertroffen wird. Bei den neueren Ansätzen ist Literatur generell eher im digitalen Bereich zu finden und weniger in gedruckter Form, was zum Teil auch dem hohen Entwicklungstempo neuerer Ansätze geschuldet ist. Hier sind die bereits etwas länger vorhandenen Ansätze wie Pthreads, Java Threads oder OpenMP stärker vertreten.

[107] Voller Funktionsumfang ist mit dem kostenpflichtigen Visual Studio 2010 verfügbar.
[108] An einer Portierung wird im Rahmen des Mono-Projektes bereits gearbeitet.

Mit Ausnahme von GCD, der CRT und .NET 4.0 können die meisten vorgestellten Ansätze auf mindestens zwei der drei hier betrachteten Plattformen eingesetzt werden. Bei .NET wird im Rahmen des Mono-Projekts bereits an einer neuen Version gearbeitet, welche die neuen Funktionen von .NET 4.0 unterstützen soll. Aufgrund des offenen Quellcodes existiert bei GCD ebenfalls die Möglichkeit, dass dieses Konzept noch in anderen Systemen übernommen wird. Die größten Chancen bestehen hier meiner Meinung nach für Linux-Systeme, da Microsoft eher auf die eigenen Erweiterungen in Form der CRT und .NET 4.0 setzen wird. Aktuell ist GCD jedoch ausschließlich auf Mac OS X einsetzbar.

OpenMP konnte leistungsmäßig am besten abschneiden, da es am häufigsten die höchsten Speedup-Werte erreichte. Anschließend folgen Intel Cilk++ und Intel TBB. Bei alleiniger Betrachtung der objektiven Kriterien, können sowohl OpenMP und Cilk++ aufgrund ihres hohen Abstraktionsgrades und der hohen Leistungsfähigkeit als auch Intel TBB aufgrund des großen funktionalen Umfangs und der guten Portabilität am meisten überzeugen.

Dieses Ergebnis bestätigt ebenfalls meinen subjektiven Eindruck, bei dem die Ansätze: OpenMP und Intel Cilk++ am besten abschneiden konnten. Trotz eines identischen Abstraktionsgrades im Vergleich zu Intel TBB lassen sie sich am schnellsten erlernen und am leichtesten verwenden. Ebenfalls konnten beide Ansätze sehr gute Leistungswerte für die hier untersuchten Anwendungen erreichen. Intel TBB ist dann eine ideale Alternative, wenn ein etwas größerer Funktionsumfang, z.B. in Form von parallelen Do- bzw. Foreach-Schleifen oder Pipelining, benötigt wird.

Bei reinen Mac OS X Anwendungen stellt GCD aus meiner Sicht die interessanteste Alternative dar, da aufgrund der tiefen Integration in das System die vermutlich besten Ergebnisse erzielt werden können. Besonders bei mehreren parallelisierten Anwendungen kann dies einen entscheidenden Vorteil bringen, da das System die Ressourcen zentral und optimal verteilen kann.

Die beiden Threadbibliotheken Pthreads und Java Threads verfügen hingegen nur über einen sehr geringen Abstraktionsgrad und sind daher mit viel Aufwand für den Programmierer verbunden. Weiterhin stellt die Skalierbarkeit von threadbasierten Bibliotheken ein Problem

dar, so dass sich diese Ansätze nicht für stark zu parallelisierende Anwendungen eignen. Vielmehr können sie als Basis für abstraktere Bibliotheken dienen.

### 5.5.2 Ansätze für Systeme mit programmierbaren Grafikkarten

| Kriterium \ Ansatz | | CUDA | OpenCL | Direct Compute | PGI Accelerator |
|---|---|---|---|---|---|
| **Erscheinungsform** | | Bibliothek | Bibliothek | Bibliothek | Erweiterung |
| **Abstraktionsgrad (Explizit / Implizit)** | **Angabe der Par.** | E | E | E | E |
| | **Zerlegung** | E | E | E | I |
| | **Verwaltung** | I | I | I | I |
| | **Abbildung** | I | I | I | I |
| | **Synchronisation** | E / I | E / I | E / I | I |
| | **Kommunikation** | E | E | E | I |
| **Funkt. Umfang** | **Datenparallelität** | ✓ | ✓ | ✓ | ✓ |
| | **Taskparallelität** | ✓ | ✓ | × | × |
| | **Pipelining** | × | × | × | × |
| **Verständlichkeit** | **Spezifikation öfftl.** | ✓ | ✓ | × | ✓ |
| | **Literatur (gedruckt)** | ** | * | * | * |
| | **Literatur (online)** | **** | ** | * | ** |
| | **Bes. Mögl. des Debug.,Testens, …** | ✓ | ✓ | ~ [109] | × |
| | **Bewertung des Autors** | *** | ** | * | **** |
| **Port.** | **Benötigte Architektur** | Nvidia | ATI/Nvidia/Cell/CPU | DX 10.x / 11 | Nvidia |
| | **Linux** | ✓ | ✓ | × | ✓ |
| | **Mac OS X** | ✓ | ✓ | × | ✓ |
| | **Windows** | ✓ | ✓ | ✓ [110] | ✓ |
| **Ansatz frei verfügbar** | | ✓ | ✓ | ✓ | × |

**Legende:** *E: Explizit, I: Implizit, ✓: Verfügbar, ×: Nicht verfügbar, ~: Unter Einschränkungen verfügbar*

**Tabelle 5-30: Gegenüberstellung der Ansätze für Systeme mit programmierbaren Grafikkarten**

Eine abschließende Bewertung fällt bei den Ansätzen für Grafikkarten nicht leicht, da sich ihre Vor- und Nachteile größtenteils gegenseitig aufheben. Beim Abstraktionsgrad unterscheiden sich CUDA, OpenCL und Direct Compute nicht voneinander, wobei der Program-

[109] Nvidia Parallel Nsight ermöglicht Debugging von CUDA und Direct Compute Anwendungen als Erweiterung für Visual Studio 2008; die professionelle Edition ist kostenpflichtig und ermöglicht zusätzlich Profiling.
[110] Nur Windows Vista und Windows 7

mieraufwand bei CUDA dennoch am geringsten ausfällt. Besser schneidet an dieser Stelle der PGI Accelerator ab, der dem Programmierer durch den hohen Grad an impliziter Parallelität viel Arbeit abnimmt. Aufgrund des kostenpflichtigen Compilers werden vermutlich nur Projekte mit entsprechendem Budget die Verwendung dieses Programmiermodells in Erwägung ziehen.

Den größten funktionalen Umfang bieten die Ansätze CUDA und OpenCL, die neben der Unterstützung für Datenparallelität auch Taskparallelität ermöglichen. Fraglich ist jedoch, inwiefern sich Taskparallelität durch das SIMT-Konzept effizient und sinnvoll umsetzen lässt.

Im Bezug zur Portabilität gibt es bei den vorgestellten Ansätzen die größten Unterschiede zu verzeichnen. Während CUDA und der PGI Accelerator zwar auf allen betrachteten Plattformen eingesetzt werden können, setzen beide eine Nvidia-Grafikkarte voraus. Ebenfalls unterliegt Direct Compute einigen Einschränkungen. So muss sich einerseits eine Grafikkarte mit DirectX 10 Unterstützung oder besser im System befinden und andererseits muss zwangsweise Windows Vista oder Windows 7 eingesetzt werden. Die größte Flexibilität bietet OpenCL, das, aufgrund der Unterstützung durch viele namhafte Hersteller, bereits auf einem breiten Hardwarespektrum und allen betrachteten Plattformen eingesetzt werden kann. Portabilität ist demzufolge das entscheidende Kriterium, mit dem sich OpenCL von CUDA, Direct Compute und dem PGI Accelerator absetzt.

Aus meiner Sicht ist OpenCL ebenfalls als der erfolgversprechendste Ansatz einzuschätzen. Zwar lassen sich CUDA und der PGI Accelerator leichter verwenden und auch steht bei CUDA ein breiteres Spektrum an gedruckter und elektronischer Literatur zur Verfügung, jedoch ist der Punkt der Portabilität meiner Meinung nach stärker zu gewichten. Besonders wenn Anwendungen im Massenmarkt von einer beschleunigten Ausführung durch Grafikkarten profitieren sollen, ist es sinnvoll, sich auf das Programmiermodell zu konzentrieren, für das die meiste Unterstützung existiert. Die Messergebnisse zeigen, dass OpenCL-basierte Anwendungen bei vielen datenparallelen Operationen ihre Stärken ausspielen können. Sind die Berechnungen nur kurz oder mit einem hohen Kommunikationsaufwand verbunden, lohnt sich OpenCL aufgrund des zusätzlichen Overheads nur bedingt.

Der bei der Entwicklung von OpenCL-Anwendungen entstehende Mehraufwand darf dabei jedoch nicht unterschätzt werden. Hier besteht grundsätzlich noch Verbesserungsbedarf, wie auch in den OpenCL-Treibern von AMD unter Linux. Die experimentelle Unterstützung für Double-Werte ließ das System teilweise komplett abstürzen und verhinderte im Falle des CG-Benchmarks sogar die Ausführung. Auch ist nicht ersichtlich, weshalb, sowohl unter Linux als auch unter Windows, nur 256 MB des Grafikkartenspeichers für OpenCL-Anwendungen genutzt werden können. Ein weiteres Problem besteht bei der Ausführung über SSH-Sitzungen. AMD stellt zur Lösung ein PDF[111] bereit, in dem die nötigen Schritte erklärt sind, um eine Ausführung dennoch zu ermöglichen. Trotz dieses Dokuments war es unter OpenSUSE 11.2 nicht möglich, OpenCL -basierte Anwendungen über eine SSH-Sitzung zu starten.

Weiterhin wird es interessant sein zu beobachten, welcher Ansatz sich in naher Zukunft durchsetzen wird. Aufgrund der bereits großen Akzeptanz von CUDA und dessen Ausgereiftheit, wird Nvidia CUDA zugunsten von OpenCL kaum aufgeben wollen, zumal CUDA als exklusives Feature nur auf Nvidia-Grafikkarten angeboten wird und somit als weiteres Kaufargument dienen kann. Auch ist es denkbar, dass der PGI Accelerator in absehbarer Zeit nicht mehr nur auf CUDA beschränkt bleibt, sondern ebenfalls eine Unterstützung für OpenCL erfährt.

[111] http://developer.amd.com/gpu_assets/App_Note-Running_ATI_Stream_Apps_Remotely.pdf

### 5.5.3 Ansätze für Systeme mit verteiltem gemeinsamen Speicher

| Kriterium \ Ansatz | | UPC | CAF | Chapel | Fortress | X10 |
|---|---|---|---|---|---|---|
| Erscheinungsform | | Erw. | Erw. | Par. Spr. | Par. Spr. | Par. Spr. |
| Abstraktionsgrad (Explizit / Implizit) | Angabe der Par. | E / I | E / I | E / I | E / I | E / I |
| | Zerlegung | E | E | E / I | E / I | E / I |
| | Verwaltung | I | I | I | I | I |
| | Abbildung | I | I | I | I | I |
| | Synchronisation | E | E | E / I | E / I | E / I |
| | Kommunikation | I | I | I | I | E |
| Funkt. Umfang | Datenparallelität | ✓ | × | ✓ | ✓ | ✓ |
| | Taskparallelität | × | × | ✓ | ✓ | ✓ |
| | Pipelining | × | × | × | × | × |
| Verständlichkeit | Spezifikation öffentlich | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Literatur (gedruckt) | ** | * | * | * | * |
| | Literatur (online) | **** | *** | ** | ** | ** |
| | Bes. Mögl. des Debug.,Testens, … | × | × | × | × | × |
| | Bewertung des Autors | ** | ** | **** | ** | *** |
| Port. | Linux | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Mac OS X | ✓ | × | ✓ | ✓ | ✓ |
| | Windows | ✓ | ✓ | ✓ | ✓ | ✓ |
| Ansatz frei verfügbar | | ✓ | ~ [112] | ✓ | ✓ | ✓ |

**Legende:** *E: Explizit, I: Implizit, ✓: Verfügbar, ×: Nicht verfügbar, ~: Unter Einschränkungen verfügbar*

**Tabelle 5-31: Gegenüberstellung der Ansätze für Systeme mit verteiltem gemeinsamen Speicher**

UPC und CAF sind Vertreter des PGAS-Programmiermodells der ersten Generation. Beide Ansätze haben einen geringeren Abstraktionsgrad und Funktionsumfang als die neueren Sprachen aus dem HPCS Programm. Da sich der Programmierer bei CAF um alle Aspekte der Parallelität selber kümmern muss, ist dieser Ansatz so eigentlich kaum noch praktikabel. UPC bietet dem Entwickler an dieser Stelle etwas mehr Komfort durch eine parallelisierte Schleife. Über weitere Möglichkeiten verfügen Chapel, Fortress und X10 z.B. durch implizite Parallelität mit kollektiven Operationen oder auch im Bereich der Taskparallelität. Sie erlauben die Programmierung auf eine Art und Weise, die eine sehr hohe Ähnlichkeit zu der Pro-

[112] Die Netzwerk-Version des Compilers ist in der kostenlosen Variante auf 5 Images beschränkt

grammierung von Systemen mit einem gemeinsamen Speicher aufweist. Nur IBM hat sich bei X10 dafür entschieden, Kommunikation durch abstrakte aber explizite Operationen im Quellcode zu kennzeichnen. Potentielle Engpässe lassen sich dadurch zwar schneller finden, erfordern aber auch einen höheren Programmieraufwand.

Literatur ist bei den neueren Ansätzen nur in geringem Umfang vorhanden und beschränkt sich größtenteils auf elektronische Quellen. UPC und CAF können an dieser Stelle auf eine größere Auswahl zurückgreifen, die allerdings auch mehr im elektronischen Bereich zu finden ist.

Bei den Performance-Messungen zu UPC fiel auf, dass nur dann sehr gute Laufzeiten erzielt werden konnten, wenn man die Kommunikation zwischen den Threads minimiert oder keine benötigt. Sobald Daten threadübergreifend auszutauschen sind, wird unverhältnismäßig viel Overhead produziert. Daher ist UPC besonders für eine solche Art von Aufgaben geeignet, bei denen die Datenverteilung im Voraus bekannt ist und die Kommunikation gering ausfällt. Konkrete Aussagen zur Leistungsfähigkeit von Chapel können auf Basis der hier durchgeführten Messungen, aufgrund des unzureichend optimierten Compilers, nicht getroffen werden. Es lässt sich jedoch festhalten, dass Chapel noch starken Aufholbedarf auf allen Ebenen der Parallelität hat. Chapel belegte fast durchgängig den letzten Platz und konnte auch beim Speedup im Vergleich zu anderen Ansätzen nicht überzeugen. Teilweise entstand überhaupt kein Mehrwert durch die Nutzung zusätzlicher Prozessoren.

Auf Basis der hier gesammelten Erkenntnisse und aufgrund der Tatsache, dass die DARPA in der letzten Phase die weitere Unterstützung für Fortress eingestellt hat, sind Chapel und X10 die Ansätze mit dem größten Potential. Ob sich die Sprachen letztendlich ggü. einem bereits etablierten Ansatz wie UPC durchsetzen können, bleibt abzuwarten. Hier wird es auf die Vorteile ankommen, die sich durch den Einsatz dieser Sprachen ergeben. Der Programmieraufwand kann jedenfalls mit beiden Sprachen verringert werden. Ebenfalls ist eine moderne Programmierweise, z.B. durch Unterstützung für die Objektorientierung, möglich. Die wirkliche Leistungsfähigkeit von Chapel kann spätestens mit einer für Benchmarks freigegeben Version des Chapel-Compilers genau geprüft werden. Dann werden auch Vergleiche zwischen Chapel und X10 möglich sein und darüber entscheiden, welche der beiden Sprachen

die Gunst der Entwickler gewinnen wird. Vom Aufwand und der Verständlichkeit bewegen sich beide Sprachen auf einem ähnlichen Niveau, meiner Meinung nach mit leichten Vorteilen für Chapel.

# 6 Fazit

Werden lediglich die Ansätze für Systeme mit einem gemeinsamen Speicher betrachtet, so lassen sich in diesem Bereich kaum wirkliche Neuerungen verzeichnen. Vielmehr wurden die Ansätze dahingehend überarbeitet, um ein möglichst großes Spektrum an Parallelität unterstützen zu können. So kam mit der Version 3.0 von OpenMP die Unterstützung für Taskparallelität hinzu, Cilk wurde zu Cilk++ und beherrscht nun auch parallele For-Schleifen. Ganz neue Ansätze bzw. Erweiterungen wiederum decken von Anfang an die Daten- und Taskparallelität (.NET 4.0, GCD) sowie Pipelining (CRT) ab. Dadurch ist man bei der Entwicklung nicht mehr auf reine Compiler-Erweiterungen wie OpenMP angewiesen, sondern konnte die parallelen Konstrukte tief in einer Sprache (.NET 4.0, CRT, Cilk++, Ct) oder sogar im System integrieren (GCD).

Die Vorteile stecken für den Entwickler daher im Detail, der in seiner gewohnten Programmiersprache nun auf integrierte Konstrukte zur Parallelisierung zurückgreifen kann. Threadsichere Datentypen, abstrakte Task-Objekte sowie Reduktionsobjekte sind nur ein paar Beispiele der Funktionen neuerer Ansätze. Andere Ansätze wiederum können mit Hilfe von Spracherweiterungen, wie z.B. Lambda-Funktionen, ihre Verwendung erheblich vereinfachen (Intel TBB). Dieser Komfort ist allerdings nicht ganz umsonst, sondern bringt zusätzlichen Overhead mit und konnte auch bei den Laufzeituntersuchungen beobachtet werden. Hier lieferte ein nur wenig in die Sprache integrierter und mit wenigen Funktionen ausgestatteter Ansatz in Form von OpenMP oftmals die besten Ergebnisse, die z.B. die komfortable und hoch integrierte CRT aufgrund des höheren Overheads so nicht erreichen konnte.

Ein ähnlicher Trend hin zu abstrakteren Programmieransätzen vollzieht sich auch bei der Programmierung von Systemen mit einem verteilten gemeinsamen Speicher. Hier wollen die Behörden bzw. die Industrie ebenfalls weg von klassischen Ansätzen wie MPI oder UPC und schufen mit dem HPCS-Programm eine Initiative, in der komplett neue Sprachen mit Ausrichtung auf eine einfache Verwendbarkeit bei gleichbleibender oder besserer Leistungsfähigkeit entwickelt werden. Teilweise konnte dies auch bereits mit Chapel und X10 umgesetzt werden, die Aspekte moderner Programmiersprachen aufgreifen (Objektorientierung, Paralleli-

sierung, …) und sich dadurch immer weniger von Programmieransätzen für Systeme mit einem gemeinsamen Speicher unterscheiden. Inwiefern auch das zweite Ziel in Anbetracht des höheren Overheads erreicht werden kann, konnte in dieser Arbeit nicht endgültig beantwortet werden. Hier gibt es, jedenfalls bei Chapel, noch viel Optimierungsbedarf.

Durch das Aufkommen programmierbarer Grafikkarten in Form von CUDA und ATI Stream bzw. OpenCL entstanden komplett neue Ansätze. In allen Fällen soll die enorme Rechenleistung aktueller Grafikkarten auch außerhalb von Computerspielen Verwendung finden. Bei den Untersuchungen im Rahmen dieser Arbeit konnte die Laufzeit bestimmter Anwendungen teilweise erheblich verringert werden. Dies trifft vor allem auf solche Anwendungen zu, die über einen sehr hohen Grad an Datenparallelität verfügen und deren einzelne Berechnungen mit einem gewissen Aufwand verbunden sind. Das größte Problem dieser Ansätze ist der benötigte Aufwand, um einerseits das Programm zu schreiben und andererseits zu optimieren. Dieser fällt bei architekturunabhängigen Ansätzen, wie z.B. bei OpenCL, besonders hoch aus[113]. Für zukünftige Versionen von OpenCL wäre ein höherer Abstraktionsgrad wünschenswert, wie ihn bspw. der PGI Accelerator für CUDA bietet. Dies kann durch abstrakte Bibliotheken oder ebenfalls durch Compiler-Erweiterungen geschehen.

Abschließend lässt sich festhalten, dass die Parallele Programmierung von Systemen mit einem gemeinsamen Speicher durch die Erweiterungen bestehender sowie komplett neuerer Ansätze vereinheitlich wurde und dass sich die Programmierung von Systemen mit einem verteilten gemeinsamen Speicher nicht mehr gravierend von Systemen mit einem gemeinsamen Speicher unterscheidet. Aufgrund der noch relativ neuen Ansätze im Bereich der programmierbaren Grafikkarten fällt der Programmieraufwand, wie auch anfangs bei der parallelen Programmierung, höher aus. Hier wird es noch einige Zeit dauern, bis die Ansätze ähnlich leicht zu verwenden sind wie Ansätze für die CPU-Programmierung oder sich sogar irgendwann aufgrund neuerer Architekturen nicht mehr von diesen unterscheiden. Ein Ziel aller Ansätze sollte es demnach sein, den Aufwand für den Programmierer weiter zu verringern und eine effiziente Umsetzung für das jeweilige System ohne gesonderte Anpassungen durch den Programmierer zu ermöglichen.

[113] Aufgrund der Notwendigkeit zur Initialisierung der Geräte, dem Erstellen von Kernel-Funktionen und dem manuellen Allokieren und Kopieren von benötigtem Speicher.

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Abkürzungsverzeichnis

**A**

| | |
|---|---|
| AAL | Asynchronous Agents Library |
| ACML | AMD Core Math Library |
| ADO | ActiveX Data Objects |
| ALU | Arithmetic Logic Unit |
| AMD | Advanced Micro Devices |
| APGAS | Asynchronous PGAS |
| API | Advanced Programmers Interface |

**B**

| | |
|---|---|
| BLAS | Basic Linear Algebra Subprograms |

**C**

| | |
|---|---|
| C99 | Erweiterungen des C-Standards aus dem Jahr 1999 |
| CAF | Co-Array FORTRAN |
| CAL | Compute Abstraction Layer |
| Cell | Prozessorserie von IBM |
| CG | Conjugate Gradient |
| CLI | Common Language Infrastructure |
| CPU | Central Processing Unit |
| CRT | Concurrency Runtime |
| CSR | Compressed Sparse Row |
| Ct | C for Throughput Computing |
| CUDA | Compute Unified Device Architecture |

**D**

| | |
|---|---|
| DARPA | Defense Advanced Research Projects Agency |
| DSM | Distributed Shared Memory |
| DSP | Digital Signal Processor |
| DSTM2 | Dynamic Software Transactional Memory Library 2.0 |
| DX | DirectX |

**E**

| | |
|---|---|
| ECMA | European Computer Manufacturers Association |
| EM64T | Extended Memory 64 Technology |

**F**

| | |
|---|---|
| FFT | Fast Fourier Transform |
| FIFO | First In First Out |

**G**

| | |
|---|---|
| GCC | GNU Compiler Collection |
| GCD | Grand Central Dispatch |
| GDB | GNU Debugger |
| GEMM | General Matrix Multiply |
| GNU | GNU is Not Unix |
| GPGPU | General Purpose Computation on Graphics Processing Unit |
| GPLv2 | GNU General Public License Version 2 |
| GPU | Graphics Processing Unit |

**H**

| | |
|---|---|
| HLSL | High Level Shading Language |
| HPCS | High Productivity Computer Systems |

**I**

| | |
|---|---|
| IEEE | Institute of Electrical and Electronics Engineers |
| IL | Intermediate Language |
| IPC | Instructions per Cycle |
| ISA | Instruction Set Architecture |

**J**

| | |
|---|---|
| JVM | Java Virtual Machine |

**L**

| | |
|---|---|
| LINQ | Language Integrated Query |

**M**

| | |
|---|---|
| MIMD | Multiple Instruction Multiple Data |
| MISD | Multiple Instruction Single Data |
| MIT | Massachusetts Institute of Technology |
| MMX | Multi Media Extension |
| MPI | Message Passing Interface |
| MPICH | Message Passing Interface Chameleon |
| MSDN | Microsoft Developer Network |
| MST | Minimum Spanning Tree |

**N**

| | |
|---|---|
| NAS | NASA Advanced Supercomputing |
| NASA | National Aeronautics and Space Administration |
| NEC | Nippon Electric Company |
| NPB | NAS Parallel Benchmarks |
| NUMA | Non-Uniform Memory Architecture |

**O**

| | |
|---|---|
| OpenCL | Open Computing Language |

**P**

| | |
|---|---|
| PGAS | Partitioned Global Address Space |
| PLINQ | Parallel LINQ |
| PPL | Parallel Pattern Library |
| PTX | Parallel Thread Execution |
| POSIX | Portable Operating System Interface |

**Q**

| | |
|---|---|
| QPI | QuickPath Interconnect |

**R**

| | |
|---|---|
| RM | Resource Manager |

**S**

| | |
|---|---|
| SDK | Software Development Kit |
| SIMD | Single Instruction Multiple Data |
| SIMT | Single Instruction Multiple Threads |
| SISD | Single Instruction Single Data |
| SMP | Symmetric Multiprocessing |
| SPMD | Single Program Multiple Data |
| SQL | Structured Query Language |
| SSE | Streaming SIMD Extensions |
| SSH | Secure Shell |
| STL | Standard Template Library |

**T**

| | |
|---|---|
| TBB | Thread Building Blocks |
| TPL | Task Parallel Library |
| TRT | Threading Runtime |

**U**

| | |
|---|---|
| UPC | Unified Parallel C |

**V**

| | |
|---|---|
| VIP | Virtual Intel Plattform |

**W**

| | |
|---|---|
| WPF | Windows Presentation Foundation |
| WF | Windows Workflow Foundation |

**X**

| | |
|---|---|
| XML | Extensible Markup Language |

# Literaturverzeichnis

[All08] E. Allen, et al.: *The Fortress Language Specification Version 1.0*, März 2008. [Online] http://labs.oracle.com/projects/plrg/fortress.pdf (Stand: 22. Apr. 2010).

[AMD07] Advanced Micro Devices: *Brook+*, Nov. 2007. [Online] http://developer.amd.com/gpu_assets/AMD-Brookplus.pdf (Stand: 19. Mai 2010).

[AMD09] Advanced Micro Devices: *Technical Overview - ATI Stream Computing*, 2009. [Online] http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf (Stand: 19. Mai 2010).

[AMD091] Advanced Micro Devices: *OpenCL™ and the ATI Stream SDK v2.0*, Nov. 2009. [Online] http://developer.amd.com/documentation/articles/pages/OpenCL-and-the-ATI-Stream-v2.0-Beta.aspx (Stand: 19. Mai 2010).

[AMD10] Advanced Micro Devices: *ATI Stream Software Development Kit (SDK) v2.1*. [Online] http://developer.amd.com/gpu/atistreamsdk/pages/default.aspx (Stand: 19. Mai 2010).

[AMD101] Advanced Micro Devices: *An Introduction to OpenCL*. [Online] http://ati.amd.com/technology/streamcomputing/intro_opencl.html (Stand: 26. Mai 2010).

[App09] Apple Inc.: *Grand Central Dispatch - Technological Brief*, Aug. 2009. [Online] http://images.apple.com/macosx/technology/docs/GrandCentral_TB_brief_20090903.pdf (Stand: 17. Mai 2010).

[App091] Apple Inc.: *Introducing Blocks and Grand Central Dispatch*, Sept. 2009. [Online] http://developer.apple.com/mac/articles/cocoa/introblocksgcd.html (Stand: 17. Mai 2010).

[App092] Apple Inc.: *Dispatch Queues*, Aug. 2009. [Online] http://developer.apple.com/mac/library/documentation/General/Conceptual/ConcurrencyProgrammingGuide/OperationQueues/OperationQueues.html#//apple_ref/doc/uid/TP40008091-CH102-SW1 (Stand: 17. Mai 2010).

[App093] Apple Inc.: *OpenCL*, Aug. 2009. [Online] http://images.apple.com/euro/macosx/technology/docs/OpenCL_TB_brief_20090608.pdf (Stand: 14. Mai 2010).

[App10] Apple Inc.: *Grand Central Dispatch (GCD) Reference*, Feb. 2010. [Online] http://developer.apple.com/mac/library/documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html (Stand: 17. Mai 2010).

[Arg10] Argonne National Laboratory: *About MPICH2*. [Online] http://www.mcs.anl.gov/research/projects/mpich2/about/index.php?s=about (Stand: 16. Apr. 2010).

[Arg101] Argonne National Laboratory: *MPICH2*. [Online] http://www.mcs.anl.gov/research/projects/mpich2/index.php (Stand: 16. Apr. 2010).

[Asa06] K. Asanovic, et al.: *The Landscape of Parallel Computing Research: A View from Berkeley*, Dez. 2006. [Online] http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf (Stand: 20. Apr. 2010).

[Ayg07] E. Ayguade, et al.: "A Proposal for Task Parallelism in OpenMP" in *Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing*, S. 1-12, Berlin Heidelberg. Springer-Verlag, 2007.

[Bai94] D. Bailey, et al.: *The NAS Parallel Benchmarks*, März 1994. [Online] http://www.nas.nasa.gov/News/Techreports/1994/PDF/RNR-94-007.pdf (Stand: 28. Juli 2010).

[Bar10] B. Barney: *POSIX Threads Programming*, Jan. 2010. [Online] https://computing.llnl.gov/tutorials/pthreads/ (Stand: 28. Apr. 2010).

[Bau06] H. Bauke und S. Mertens: *Cluster Computing*. Berlin Heidelberg: Springer, 2006.

[Ben08] G. Bengel, C. Baun, M. Kunze und K.-U. Stucky: *Masterkurs Parallele und Verteilte Systeme*. Wiesbaden: Vieweg+Teubner, 2008.

[Bin09] A. Binstock: *Threading Models for High-Performance Computing: Pthreads or OpenMP?*, Okt. 2009. [Online] http://software.intel.com/en-us/articles/threading-models-for-high-performance-computing-pthreads-or-openmp/ (Stand: 28. Apr. 2010).

[Boy08] C. Boyd: *The Direct3D 11 Compute Shader*, 2008. [Online] http://s08.idav.ucdavis.edu/boyd-dx11-compute-shader.pdf (Stand: 31. Mai 2010).

[Bul09] A. Buluc, J. T. Fineman, M. Frigo, J. R. Gilbert und C. E. Leiserson: "Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks" in *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, S. 233-244, Calgary (Kanada). ACM Press, 2009.

[But02] D. R. Butenhof: *Programming with POSIX Threads*. Indianapolis (USA): Addison-Wesley, 2002.

[Car09] J. Carr: *The Power of Well-Structured Parallelism (answering a FAQ about Cilk++)*, Okt. 2009. [Online] http://software.intel.com/en-us/articles/The-Power-of-Well-Structured-Parallelism-answering-a-FAQ-about-Cilk/ (Stand: 13. Mai 2010).

[Car99] W. W. Carlson, et al.: *Introduction to UPC and Language Specification*, Mai 1999. [Online] http://upc.lbl.gov/publications/upctr.pdf (Stand: 01. Juni 2010).

[Cha01] R. Chandra, et al.: *Parallel Programming in OpenMP*. San Diego (USA): Academic Press, 2001.

[Cha05] S. Chauvin, P. Saha, F. Cantonnet, S. Annareddy und T. El-Ghazawi: *UPC Manual*, Mai 2005. [Online] http://upc.gwu.edu/downloads/Manual-1.2.pdf (Stand: 01. Juni 2010).

[Cha051] P. Charles, et al.: "X10: An Object-Orientated Approach to Non-Uniform Cluster Computing" in *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, S. 519-538, San Diego 2005.

[Cha07] B. L. Chamberlain, D. Callahan und H. P. Zima: "Parallel Programmability and the Chapel Language", *International Journal of High Performance Computing Applications*, Band 21, Ausg. 3, S. 291-312. Aug. 2007.

[Cha09] B. L. Chamberlain: *Chapel: the Cascade High Productivity Language*, Okt. 2009. [Online] http://chapel.cray.com/presentations/ChapelForBSC-presented.pdf (Stand: 15. Juni 2010).

[Che98] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall und A. F. Stark: "Detecting Data Races in Cilk Programs that Use Locks" in *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, S. 298-309, Puerto Vallarta 1998.

[Cho10] S.-E. Choi und S. Deitz: *Introduction to Chapel - A Next-Generation HPC Language*, Apr. 2010. [Online] http://chapel.cray.com/tutorials/DC2010/DC-tutorial.tar.gz (Stand: 15. Juni 2010).

[Cra08] Cray Inc.: *Heat Transfer in Chapel*, Nov. 2008. [Online] http://chapel.cray.com/tutorials/SC08/PGAS/SC08ChapelHeatTransfer.pdf (Stand: 14. Juni 2010).

[Cra10] Cray Inc.: *Chapel Language Specification 0.975*, Apr. 2010. [Online] http://chapel.cray.com/spec/spec-0.795.pdf (Stand: 14. Juni 2010).

[Cra101] Cray Inc.: *Chapel Release Notes 1.1*, Apr. 2010. [Online] https://sourceforge.net/projects/chapel/files/chapel/1.1/chapel-relnotes-1.1.txt/view (Stand: 27. Juli 2010).

[Dob10] W. Doberenz und T. Gewinnus: *Visual C# 2010 - Grundlagen und Profiwissen*. München: Carl Hanser Verlag, 2010.

[Duf07] J. Duffy und E. Essey: *Running Queries On Multi-Core Processors*, Okt. 2007. [Online] http://msdn.microsoft.com/en-us/magazine/cc163329.aspx (Stand: 13. Okt. 2009).

[Fel08] M. Feldman: *Sun's Fortress Language: Parallelism by Default*, Juli 2008. [Online] http://www.hpcwire.com/features/Suns_Fortress_Language_Parallelism_by_Default.html (Stand: 16. Juni 2010).

[Fel09] M. Feldman: *Intel Gets Ready to Push Ct Out of the Lab*, Apr. 2009. [Online] http://www.hpcwire.com/features/Intel-Gets-Ready-to-Push-Ct-Out-of-the-Lab-42634332.html (Stand: 11. Mai 2010).

[Fly72] M. J. Flynn: "Some Computer Organizations and Their Effectiveness", *IEEE Transactions on Computers*, Band 21, Ausg. 9, S. 948-960. IEEE Computer Society Press, Okt. 1972.

[Fre10] Free Software Foundation Inc.: *gfortran — the GNU Fortran compiler, part of GCC*, Juli 2010. [Online] http://gcc.gnu.org/wiki/GFortran (Stand: 13. Aug. 2010).

[Fri09] M. Frigo, P. Halpern, C. E. Leiserson und S. Lewin-Berlin: "Reducers and Other Cilk++ Hyperobjects" in *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, S. 79-90, Calgary (Kanada). ACM Press, 2009.

[gam08] gamedev.net: *DirectX 11 Compute Shader tutorial*, 2008. [Online] http://www.gamedev.net/community/forums/topic.asp?topic_id=516043 (Stand: 28. Mai 2010).

[Gau10] M. Gause, et al.: "Sanfte Evolution - Adobe Creative Suite 5", *c't*, Ausg. 13, S. 98-103. Hannover: Heise Zeitschriften Verlag, Juni 2010.

[Ghu07] A. Ghuloum, et al.: "Future-Proof Data Parallel Algorithms and Software on Intel® Multi-Core Architecture", *Intel Technology Journal*, Band 11, Ausg. 4, S. 333-348. Nov. 2007.

[Ghu10] A. Ghuloum, et al.: *Ct Technology: A Flexible Parallel Programming Model for Multicore and Many-Core Architectures*, März 2010. [Online] http://www.drdobbs.com/go-parallel/article/showArticle.jhtml;jsessionid=BOMDIPNZAP53RQE1GHRSKH4ATMY32JVN?articleID=224200207 (Stand: 11. Mai 2010).

[Goe06] B. Goetz, et al.: *Java Concurrency in Practice*. Stoughton (USA): Pearson Education, 2006.

[Gol08] Golem: *OpenCL 1.0 und OpenGL 3.0 sind da*, Dez. 2008. [Online] http://www.golem.de/0812/64015.html (Stand: 26. Mai 2010).

[GPG10] GPGPU.org: *GPGPU Developer Resources*. [Online] http://gpgpu.org/developer (Stand: 14. Apr. 2010).

[Gra03] A. Gramma, A. Gupta, G. Karypis und V. Kumar: *Introduction to Parallel Computing*. Essex (England): Pearson Education, 2003.

[Gre06] A. Greß und G. Zachmann: "GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures" in *Proc. of the 20th IEEE Int. Parallel and Distributed Processing Symposium*, S. 45-54, Rhodes Island (Griechenland) 2006.

[Gro09] W. Gropp, et al.: *MPICH2 User's Guide*, Nov. 2009. [Online] http://www.mcs.anl.gov/research/projects/mpich2/documentation/files/mpich2-1.2.1-userguide.pdf (Stand: 16. Apr. 2010).

[Gro10] D. Grove: *X10 Compiler Overview*, Jan. 2010. [Online] http://docs.codehaus.org/display/XTENLANG/X10+Compiler+Overview (Stand: 21. Juni 2010).

[Gro99] W. Gropp: *Using MPI-2*. Massachusetts (USA): MIT Press, 1999.

[Gun09] A. Gunal: *Resource Management in the Concurrency Runtime (MSDN Blogs)*, Mai 2009. [Online] http://blogs.msdn.com/b/nativeconcurrency/archive/2009/03/10/resource-management-in-the-concurrency-runtime-part-1.aspx (Stand: 16. Aug. 2010).

[Her06] M. Herlihy, V. Luchangco und M. Moir: "A Flexible Framework for Implementing Software Transactional Memory" in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, S. 253-262, Portland (USA). ACM Press, 2006.

[Hil06] P. Hilfinger, et al.: *Titanium Language Reference Manual Version 2.20*, Aug. 2006. [Online] http://titanium.cs.berkeley.edu/doc/lang-ref.pdf (Stand: 14. Juni 2010).

[Hoc05] L. Hochsein, et al.: "Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers" in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, S. 35-43, Washington (USA). IEEE Computer Society Press, 2005.

[Hor10] C. Hornschuh: "Zusammenstellung und Bewertung existierender Ansätze von Microsoft zur Programmierung von Multi-Core Prozessoren", Hochschule Bonn-Rhein-Sieg, Sankt Augustin 2010.

[HPC10] High Performance Computing Laboratory: *UPC Tutorials*. [Online] http://upc.gwu.edu/tutorials.html (Stand: 02. Juni 2010).

[HPC101] High Performance Computing Laboratory: *UPC Documentation*, Apr. 2010. [Online] http://upc.gwu.edu/documentation.html (Stand: 02. Juni 2010).

[Int09] Intel Corporation: *Intel® Cilk++ SDK Programmer's Guide*, Okt. 2009. [Online] http://software.intel.com/file/23634 (Stand: 09. Apr. 2010).

[Int091] Intel Corporation: *An Introduction to the Intel QuickPath Interconnect*, Jan. 2009. [Online] http://www.intel.com/technology/quickpath/introduction.pdf (Stand: 29. Juli 2010).

[Int10] Intel Corporation: *Intel® Threading Building Blocks Tutorial*, Apr. 2010. [Online] http://www.threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Tutorial.pdf (Stand: 04. Mai 2010).

[Int101] Intel Corporation: *Intel® Threading Building Blocks Reference Manual*, Apr. 2010. [Online] http://www.threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Reference.pdf (Stand: 04. Mai 2010).

[Int102] Intel Corporation: *Intel® Threading Building Blocks Getting Started Guide*, März 2010. [Online] http://www.threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Getting_Started.pdf (Stand: 09. Mai 2010).

[Int103] Intel Corporation: *Intel® Cilk++ Software Development Kit*, Feb. 2010. [Online] http://software.intel.com/en-us/articles/intel-cilk/ (Stand: 13. Mai 2010).

[Joh10] R. Johnson: *POSIX Threads (pthreads) for Win32*. [Online] http://sourceware.org/pthreads-win32/ (Stand: 28. Apr. 2010).

[Kas08] H. Kasim, V. March, R. Zhang und S. See: "Survey on Parallel Programming Model", *Lecture Notes in Computer Science*, Ausg. 5245, S. 266-275. Berlin Heidelberg: Springer-Verlag, 2008.

[Ker09] K. Kerr: *Visual C++ 2010 und die Parallel Patterns Library*, Feb. 2009. [Online] http://msdn.microsoft.com/de-de/magazine/dd434652.aspx (Stand: 25. Feb. 2010).

[Khr09] Khronos OpenCL Working Group: *The OpenCL Specification*, Juni 2009. [Online] http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf (Stand: 25. Mai 2010).

[Kir10] D. B. Kirk und W.-m. W. Hwu: *Programming Massively Parallel Processors: A Hands-on Approach*. Burlington (USA): Elsevier, 2010.

[LBL10] Lawrence Berkeley National Laboratory: *Berkeley UPC User's Guide Version 2.10.2*, Mai 2010. [Online] http://upc.lbl.gov/docs/user/index.shtml (Stand: 02. Juni 2010).

[Lea99] D. Lea: *Concurrent Programming in Java*, 2. Ausg. Massachusetts (USA): Addison Wesley Longman Inc., 1999.

[Lei09] C. E. Leiserson: "The Cilk++ concurrency platform" in *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, vol. 51, S. 522-527, San Francisco. ACM Press, März 2009.

[Mag06] J. Magee und J. Kramer: *Concurrency - State Models & Java Programming*. West Sussex (England): John Wiley & Sons Ltd, 2006.

[Mic10] Microsoft Corporation: *MSDN - parallel_for Function*, 2010. [Online] http://msdn.microsoft.com/en-us/library/dd505035.aspx (Stand: 07. Mai 2010).

[Mic101] Microsoft Corporation: *MSDN - parallel_for_each Function*, 2010. [Online] http://msdn.microsoft.com/en-us/library/dd492857.aspx (Stand: 07. Mai 2010).

[Mic1010] Microsoft Corporation: *MSDN - Task(TResult).Result Property* , 2010. [Online] http://msdn.microsoft.com/en-us/library/dd321468.aspx (Stand: 11. Mai 2010).

[Mic1011] Microsoft Corporation: *MSDN - Introduction to PLINQ*, 2010. [Online] http://msdn.microsoft.com/en-us/library/dd997425.aspx (Stand: 11. Mai 2010).

[Mic1012] Microsoft Corporation: *MSDN - HLSL*, Mai 2010. [Online] http://msdn.microsoft.com/en-us/library/bb509561%28VS.85%29.aspx (Stand: 31. Mai 2010).

[Mic1013] Microsoft Corporation: *MSDN - Task Parallelism (Concurrency Runtime)*, Juli 2010. [Online] http://msdn.microsoft.com/en-us/library/dd492427.aspx (Stand: 16. Aug. 2010).

[Mic1014] Microsoft Corporation: *MSDN - Asynchronous Agents Library*, 2010. [Online] http://msdn.microsoft.com/en-us/library/dd492627.aspx (Stand: 16. Aug. 2010).

[Mic1015] Microsoft Corporation: *MSDN - Comparing the Concurrency Runtime to Other Concurrency Models*, Juli 2010. [Online] http://msdn.microsoft.com/en-us/library/dd998048.aspx (Stand: 16. Aug. 2010).

[Mic1016] Microsoft Corporation: *MSDN - Parallel Patterns Library (PPL)*, 2010. [Online] http://msdn.microsoft.com/en-us/library/dd492418.aspx (Stand: 16. Aug. 2010).

[Mic1017] Microsoft Corporation: *MSDN - Asynchronous Agents*, 2010. [Online] http://msdn.microsoft.com/en-us/library/dd551463.aspx (Stand: 16. Aug. 2010).

[Mic1018] Microsoft Corporation: *MSDN - Asynchronous Message Blocks*, 2010. [Online] http://msdn.microsoft.com/en-us/library/dd504833.aspx (Stand: 16. Aug. 2010).

[Mic1019] Microsoft Corporation: *MSDN - Message Passing Functions*, 2010. [Online] http://msdn.microsoft.com/en-us/library/dd492424.aspx (Stand: 16. Aug. 2010).

[Mic102] Microsoft Corporation: *MSDN - parallel_invoke Function*, 2010. [Online] http://msdn.microsoft.com/en-us/library/dd504887.aspx (Stand: 07. Mai 2010).

[Mic1020] Microsoft Corporation: *MSDN - Scheduler Policies*, Juli 2010. [Online] http://msdn.microsoft.com/en-us/library/ff829269.aspx (Stand: 16. Aug. 2010).

[Mic103] Microsoft Corporation: *MSDN - Task Constructor*, 2010. [Online] http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.task.aspx (Stand: 11. Mai 2010).

[Mic104] Microsoft Corporation: *MSDN - Task Parallel Library*, 2010. [Online] http://msdn.microsoft.com/en-us/library/dd460717.aspx (Stand: 11. Mai 2010).

[Mic105] Microsoft Corporation: *MSDN - Data Parallelism (Task Parallel Library)*, 2010. [Online] http://msdn.microsoft.com/en-us/library/dd537608.aspx (Stand: 11. Mai 2010).

[Mic106] Microsoft Corporation: *MSDN - Task Parallelism (Task Parallel Library)*, 2010. [Online] http://msdn.microsoft.com/en-us/library/dd537609.aspx (Stand: 11. Mai 2010).

[Mic107] Microsoft Corporation: *MSDN - Parallel.For Method*, 2010. [Online] http://msdn.microsoft.com/en-us/library/system.threading.tasks.parallel.for.aspx (Stand: 11. Mai 2010).

[Mic108] Microsoft Corporation: *MSDN - Parallel.ForEach Method* , 2010. [Online] http://msdn.microsoft.com/en-us/library/system.threading.tasks.parallel.foreach.aspx (Stand: 11. Mai 2010).

[Mic109] Microsoft Corporation: *MSDN - Task(TResult) Constructor*, 2010. [Online] http://msdn.microsoft.com/en-us/library/dd321477.aspx (Stand: 11. Mai 2010).

[MIT07] MIT CSAIL Supercomputing Technologies Group: *The Cilk Project*, Okt. 2007. [Online] http://supertech.csail.mit.edu/cilk/ (Stand: 13. Mai 2010).

[Mon10] Mono: *What is Mono*. [Online] http://www.mono-project.com/What_is_Mono (Stand: 10. Mai 2010).

[Mon101] Mono: *Compatibility*. [Online] http://www.mono-project.com/Compatibility (Stand: 10. Mai 2010).

[NAS10] NASA Ames Research Center: *NAS Parallel Benchmark Changes*, Mai 2010. [Online] http://www.nas.nasa.gov/Resources/Software/npb_changes.html#npb2 (Stand: 29. Juli 2010).

[Net10] Net Applications: *Operating System Market Share*, März 2010. [Online] http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8 (Stand: 22. Apr. 2010).

[Neu10] A. Neumann: *Intels C++-Bibliothek Threading Building Blocks 3.0 veröffentlicht*, Mai 2010. [Online] http://www.heise.de/newsticker/meldung/Intels-C-Bibliothek-Threading-Building-Blocks-3-0-veroeffentlicht-993385.html (Stand: 05. Mai 2010).

[Nic96] B. Nichols, D. Buttlar und J. P. Farrell: *Pthreads Programming*. Sebastopol (USA): O'Reilly, 1996.

[Num98] R. W. Numrich und J. Reid: "Co-Array Fortran for parallel programming", *SIGPLAN Fortran Forum*, Band 17, Ausg. 2, S. 1-31. ACM Press, 1998.

[Nvi08] Nvidia: *CUDA Technical Training*, 2008. [Online] http://www.nvidia.com/docs/IO/47904/VolumeI.pdf (Stand: 20. Mai 2010).

[Nvi09] Nvidia: *NVIDIA CUDA Architecture*, Apr. 2009. [Online] http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf (Stand: 20. Mai 2010).

[Nvi091] Nvidia: *PTX: Parallel Thread Execution*, März 2009. [Online] http://www.nvidia.com/content/CUDA-ptx_isa_1.4.pdf (Stand: 25. Mai 2010).

[Nvi092] Nvidia: *NVIDIA OpenCL JumpStart Guide*, Apr. 2009. [Online] http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf (Stand: 25. Mai 2010).

[Nvi093] Nvidia: DirectCompute Programming Guide Version 2.3., Aug. 2009.

[Nvi10] Nvidia: *CUDA 3.0 Downloads*, Mai 2010. [Online] http://developer.nvidia.com/object/cuda_3_0_downloads.html (Stand: 20. Mai 2010).

[Nvi101] Nvidia: *NVIDIA CUDA Programming Guide Version 3.0*, Feb. 2010. [Online] http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf (Stand: 22. Apr. 2010).

[Nvi102] Nvidia: *NVIDIA CUDA Best Practices Guide*, Apr. 2010. [Online] http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide.pdf (Stand: 19. Mai 2010).

[Oec07] R. Oechsle: *Parallele und verteilte Anwendungen in JAVA*. München: Hanser, 2007.

[OMP08] OpenMP Architecture Review Board: *OpenMP Application Program Interface*, Mai 2008. [Online] http://www.openmp.org/mp-documents/spec30.pdf (Stand: 02. Apr. 2010).

[OMP10] OpenMP: *About OpenMP and OpenMP.org*, Apr. 2010. [Online] http://openmp.org/wp/about-openmp/ (Stand: 30. 2010).

[OMP101] OpenMP: *OpenMP Compilers*, März 2010. [Online] http://openmp.org/wp/openmp-compilers/ (Stand: 30. Apr. 2010).

[Ope10] OpenVIDIA: *DirectCompute*, Mai 2010. [Online] http://openvidia.sourceforge.net/index.php/DirectCompute (Stand: 31. Mai 2010).

[Ora10] Oracle: *Java-Downloads für alle Betriebssysteme*. [Online] http://java.com/de/download/manual.jsp (Stand: 06. Mai 2010).

[PCI10] PCI-SIG : *PCIe® Base 2.1 Specification*. [Online] http://www.pcisig.com/specifications/pciexpress/base2/ (Stand: 26. Mai 2010).

[Pla09] Planet 3DNow: *GPGPU Computing - ein Überblick für Anfänger und Fortgeschrittene*, Mai 2009. [Online] http://www.planet3dnow.de/vbulletin/showthread.php?t=362621 (Stand: 14. Apr. 2010).

[Qui03] M. J. Quinn: *Parallel Programming in C with MPI and OpenMP*. New York (USA): McGraw-Hill, 2003.

[Rau07] T. Rauber und G. Rünger: *Parallele Programmierung*. Heidelberg: Springer, 2007.

[Rau08] T. Rauber und G. Rünger: *Multicore: Parallele Programmierung*. Heidelberg: Springer, 2008.

[Rei07] J. Reinders: *Intel Threading Building Blocks*. Sebastopol (USA): O'Reilly, 2007.

[Rei09] J. Reinders: *parallel_for is easier with lambdas, Intel Threading Building Blocks*, Aug. 2009. [Online] http://software.intel.com/en-us/blogs/2009/08/03/parallel_for-is-easier-with-lambdas-intel-threading-building-blocks/ (Stand: 04. Mai 2010).

[Rei091] J. Reinders: *"Hello Lambdas" C++ 0x, a quick guide to Lambdas in C++*, Aug. 2009. [Online] http://software.intel.com/en-us/blogs/2009/08/03/hello-lambdas-c-0x-a-quick-guide-to-lambdas-in-c/ (Stand: 04. Mai 2010).

[Rei10] J. Reinders: *TBB 3.0: New (today) Version of Intel Threading Building Blocks*, Mai 2010. [Online] http://software.intel.com/en-us/blogs/2010/05/04/tbb-30-new-today-version-of-intel-threading-building-blocks/ (Stand: 04. Mai 2010).

[San09] M. Sandy: *Data-Parallel Programming with DirectCompute*, Nov. 2009. [Online] http://code.msdn.microsoft.com/directcomputehol

[San10] J. Sanders und E. Kandrot: *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Boston (USA): Addison-Wesley, 2010.

[Sar10] V. Saraswat und B. Bloom: *Report on the Programming Language X10 Version 2.0.4*, Juni 2010. [Online] http://dist.codehaus.org/x10/documentation/languagespec/x10-latest.pdf (Stand: 18. Juni 2010).

[Sat09] N. Satish, M. Harris und M. Garland: "Designing Efficient Sorting Algorithms for Manycore GPUs" in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, Rom (Italien). IEEE Computer Society Press, 2009.

[Shn78] B. Shneiderman und R. Mayer: "Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results", *International Journal of Parallel Programming*, Band 8, Ausg. 3, S. 219-238. Springer Netherlands, 1978.

[Sil99] L. M. Silva und R. Buyya: *Parallel Programming Models and Paradigms*. New Jersey (USA): Prentice Hall, 1999.

[Sir09] J. Siracusa: *Mac OS X 10.6 Snow Leopard: the Ars Technica review*, Aug. 2009. [Online] http://arstechnica.com/apple/reviews/2009/08/mac-os-x-10-6.ars/ (Stand: 17. Mai 2010).

[Ski97] D. B. Skillicorn und D. Talia: "Models and Languages for Parallel Computation", *ACM Computing Surveys*, Band 30, Ausg. 2, S. 123-169. ACM Press, 1997.

[Sta10] R. Stallman, R. Pesch und S. Shebs: *Debugging with GDB*. Boston (USA): Free Software Foundation, 2010.

[Ste06] G. L. Steele und J.-W. Maessen: *Fortress Programming Language Tutorial*, Juni 2006. [Online] http://labs.oracle.com/projects/plrg/PLDITutorialSlides9Jun2006.pdf (Stand: 16. Juni 2010).

[Ste08] G. L. Steele: *What's Cool about Fortress*, Mai 2008. [Online] http://labs.oracle.com/projects/plrg/Publications/2008-0210.pdf (Stand: 16. Juni 2010).

[Str09] R. Strzodka, D. Göddeke und D. Behr: *GPU Architecture*, Sept. 2009. [Online] http://gpgpu.org/wp/wp-content/uploads/2009/09/A3-Architecture.pdf (Stand: 18. Mai 2010).

[Sun041] Sun Microsystems: *Interface Future<V>*, 2004. [Online] http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/Future.html (Stand: 06. Mai 2010).

[Sun042] Sun Microsystems: *Class AtomicInteger*, 2004. [Online] http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/atomic/AtomicInteger.html (Stand: 06. Mai 2010).

[Sun08] Sun Microsystems: *Implicit Parallelism in Fortress*, Mai 2008. [Online] http://projectfortress.sun.com/Projects/Community/wiki/ImplicitParallelismInFortress (Stand: 18. Juni 2010).

[Sza96] D. Szafron und J. Schaeffer: "An Experiment to Measure the Usability of Parallel Programming Systems", University of Alberta, Edmonton (Kanada) 1996.

[TGP10] The G95 Project: *Quick Overview of Coarrays*. [Online] http://www.g95.org/coarray.shtml (Stand: 03. Juni 2010).

[Thi09] N. Thibieroz: *Shader Model 5.0 and Compute Shader*, März 2009. [Online] http://developer.amd.com/gpu_assets/Shader%20Model%205-0%20and%20Compute%20Shader.pps (Stand: 31. Mai 2010).

[Top10] Top500: *Processor Family share for 06/2010*, Juni 2010. [Online] http://www.top500.org/stats/list/35/procfam (Stand: 22. Juni 2010).

[Tou09] S. Toub: *PATTERNS OF PARALLEL PROGRAMMING*, 2009. [Online] http://www.microsoft.com/downloads/details.aspx?FamilyID=86b3d32b-ad26-4bb8-a3ae-c1637026c3ee&displaylang=en (Stand: 12. Nov. 2009).

[TPG10] The Portland Group: *PGI Accelerator Compilers*. [Online] http://www.pgroup.com/resources/accel.htm (Stand: 28. Mai 2010).

[TPG101] The Portland Group: *PGI Fortran & C Accelerator Programming Model*, März 2010. [Online] http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.2.pdf (Stand: 28. Mai 2010).

[Tro10] A. Troelsen: *Pro C# 2010 and the .Net 4 Platform*, 5. Ausg. New York (Springer-Verlag), 2010.

[TXP10] The X10 Project: *X10 2.0.4 Release*. [Online] http://dist.codehaus.org/x10/binaryReleases/2.0.4/INSTALL.txt (Stand: 22. Juni 2010).

[UAZ98] University of Arizona - Software Interest Group: *POSIX Threads Tutorial*, Mai 1998. [Online] http://math.arizona.edu/~swig/documentation/pthreads/ (Stand: 28. Apr. 2010).

[UCB09] Computer Science Division, University of California at Berkeley: *Titanium Software and Documentation*, Nov. 2009. [Online] http://titanium.cs.berkeley.edu/ (Stand: 14. Juni 2010).

[Ull09] C. Ullenboom: *Java ist auch eine Insel*. Bonn: Galileo Press, 2009.

[UPC05] UPC Consortium: *UPC Language Specifications V1.2*, Mai 2005. [Online] http://www.gwu.edu/~upc/publications/LBNL-59208.pdf (Stand: 22. Apr. 2010).

[UPC06] UPC Consortium: *Institutions Participating in the UPC Consortium*, Okt. 2006. [Online] https://upc-wiki.lbl.gov/index.php/Institutions (Stand: 12. Juni 2010).

[Vau08] A. Vaught: *The Complete Compendium on Cooperative Computing using Coarrays*, Okt. 2008. [Online] http://www.g95.org/compendium.pdf (Stand: 3. Juni 2010).

[Wal10] A. Wallcraft: *Co-Array Fortran*. [Online] http://www.co-array.org/ (Stand: 3. Juni 2010).

[Wer09] B. Werth: *Optimizing Game Architectures with Intel® Threading Building Blocks*, Juni 2009. [Online] http://software.intel.com/en-us/articles/optimizing-game-architectures-with-intel-threading-building-blocks/ (Stand: 04. Mai 2010).

[Wes07] R. Westphal und C. Weyer: *.NET 3.0 kompakt*. München: Spektrum Akademischer Verlag, 2007.

[Wil99] B. Wilkinson und M. Allen: *Parallel Programming*. New Jersey (USA): Prentice Hall, 1999.

[Wol09] M. Wolfe: *The PGI Accelerator Programming Model on NVIDIA GPUs Part 1*, Juni 2009. [Online] http://www.pgroup.com/lit/articles/insider/v1n1a1.htm (Stand: 28. Mai 2010).

[Yel06] K. Yelick: *Performance and Productivity Opportunities using Global Address Space Programming Models*, 2006. [Online] http://www.sdsc.edu/pmac/ workshops/geo2006/pubs/Yelick.pdf (Stand: 1. Juni 2010).

[Yel061] K. Yelick, et al.: *Parallel Languages and Compilers: Perspective from the Titanium Experience*, Juni 2007. [Online] http://www.cs.berkeley.edu/~bonachea/ti/ti_experience.pdf (Stand: 14. Juni 2010).

[Yel98] K. Yelick, et al.: *Titanium: A High-Performance Java Dialect*, 1998. [Online] http://www.cs.umd.edu/class/fall2005/cmsc714/Readings/titanium-hpj98.pdf (Stand: 14. Juni 2010).

[Zah06] M. Zahn: *Unix-Netzwerkprogrammierung mit Threads, Sockets und SSL*. Berlin Heidelberg: Springer-Verlag, 2006.

# Anhang

## A. Inhaltsverzeichnis der CD/DVD