RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE AACHEN
(RWTH AACHEN)

**Lehrstuhl für Hochleistungsrechnen**
Prof. C. Bischof, Ph.D.

# Shared-Memory Parallelisierung von C++ Programmen

**Diplomarbeit**

Christian Terboven
Matr.-Nr. 227633

28. Januar 2006

Erstgutachter: Prof. Christian H. Bischof, Ph.D.

Zweitgutachter: Prof. Dr. Dr. Thomas Lippert

Betreuer: Dipl.-Math. Dieter an Mey

# Erklärung

Ich versichere hiermit, die vorliegende Arbeit selbständig und ohne Verwendung anderer als der angegebenen Hilfsmittel angefertigt zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Aachen, im Januar 2006.

_____

(Christian Terboven)

# Danksagung

Mein Dank gilt Herrn Prof. C. Bischof, Ph.D. für die Vergabe dieses Diplomthemas und Herrn Prof. Dr. Dr. Thomas Lippert für sein Zweitgutachten.

Für die hervorragende Betreuung, bei der er sich immer Zeit genommen hat um Fragen und Ideen zu diskutieren und zu beantworten, bedanke ich mich bei Herrn Dipl.-Math. Dieter an Mey.

Danke sage ich auch Herrn Andre Hegerath und Herrn Dipl.-Inform. Thomas Deselaers für das Korrekturlesen, meinen Eltern, die mir das Studium der Informatik und so viel mehr ermöglicht haben, und meiner Freundin Sabrina für die Unterstützung in dieser Zeit.

# Inhaltsverzeichnis

# 1 Einleitung

Zur Verkürzung der Laufzeit von rechenintensiven Anwendungen besonders im technisch-wissenschaftlichen Bereich werden Programme zur Nutzung von mehreren Prozessoren vorbereitet, d.h. parallelisiert. Während zur parallelen Nutzung von Rechnern mit eigenem Hauptspeicher das Message-Passing über ein schnelles Netzwerk im Hochleistungsrechnen (HPC) populär ist, gewinnt die Shared-Memory Parallelisierung mit der steigenden Verfügbarkeit von Parallelrechnern, deren Prozessoren Zugriff auf einen gemeinsamen Speicher haben, an Bedeutung. Anhand der sich in jüngster Zeit abzeichnenden Tendenz, mehrere Prozessoren auf einem Chip unterzubringen und mehrere Threads damit gleichzeitig in einem Prozessor zu bedienen, lässt sich absehen, dass diese Vorgehensweise an Bedeutung zunehmen wird.

Während viele Anwendungen im Hochleistungsrechnen noch in der Programmiersprache FORTRAN programmiert sind, ist in den letzten Jahren die Einsatzhäufigkeit der Programmiersprache C++ auch im HPC-Umfeld stetig gewachsen. Durch Fortschritte bei den Fähigkeiten von C++ Compilern zur Generierung hoch-performanten Codes ist es damit möglich, die Stärken desr objektorientierten Programmierung insbesondere im Hinblick auf die Programmentwicklung mit der Rechenleistung von parallelen Computern zu verbinden.

Diese Arbeit beschäftigt sich mit verschiedenen Aspekten der Parallelisierung von C++ Programmen für Shared-Memory Systeme. Anhand von drei in C++ geschriebenen Anwendungsprogrammen sowie ausgewählter Benchmarks wird untersucht, welche Strategien sich zur Shared-Memory Parallelisierung von modernen C++ Programmen anbieten. Bei den Anwendungsprogrammen handelt es sich um den Navier-Stokes Solver DROPS, der am Institut für Geometrie und Praktische Mathematik (IGPM) der RWTH Aachen entwickelt wird, das Image-Retrieval Systems FIRE, welches am Lehrstuhl für Informatik 6 der RWTH Aachen entwickelt wird, sowie um den Poisson-Solver ADI, der am NeuroInformatics Center der University of Oregon entwickelt wird.

Zur Parallelisierung von C++ Programmen für Shared-Memory Architekturen bieten sich die drei Parallelisierungstechniken OpenMP, Posix-Threads und UPC an. Jede dieser drei Techniken verfolgt einen unterschiedlichen Ansatz zur Parallelisierung von Programmen, dennoch sind die eingesetzten Konzepte in weiten Teilen identisch. Es soll untersucht werden, welche der drei Techniken sich am besten für die Parallelisierung der betrachten Programme eignet.

Die NUMA-Architektur der Sun Fire V40z System begrenzt bei den Programmen DROPS und ADI die Skalierbarkeit, da sie bisher nicht für die Beachtung der Datenlokalität optimiert wurden. Es soll festgestellt werden, mit welchen Mitteln auf der NUMA-Architektur der Sun Fire V40z Systeme die Datenplatzierung beeinflusst werden kann, um den Speedup zu verbessern. Dazu werden sowohl Mittel der Programmiersprache C++ als auch Fähigkeiten der eingesetzten Betriebssysteme betrachtet. Darüber hinaus wird gezeigt, wie auf verschiedenen Plattformen das Speichermanagement von STL Datentypen verbessert und wie bei der Parallelisierung von objekt-

orientierten Codes und von Schleifen in nicht kanonischer Form vorgegangen werden kann.

Als weiteres Anwendungsprogramm wird an FIRE untersucht, wie die objektorientierte Programmierung mit OpenMP verbunden werden kann und wie durch den Einsatz von geschachteltem OpenMP die Flexibilität und Skalierbarkeit erhöht werden kann.

Die Arbeit ist wie folgt gegliedert:

Zunächst wird in Kapitel 2 auf die Grundlagen der verwendeten Hardware-Architekturen sowie auf wichtige Aspekte der Parallelisierung eingegangen.

Daran anschließend werden in Kapitel 3 die drei Programme DROPS, FIRE und ADI und die synthetischen Benchmarks soweit vorgestellt, wie es für die an ihnen durchgeführten Untersuchungen im Rahmen dieser Diplomarbeit notwendig ist. Für diese drei Anwendungsprogramme wird ebenfalls ihre Parallelisierung beschrieben.

In Kapitel 4 werden drei Parallelisierungstechniken miteinander verglichen, die hardwarenahe Programmierung mit den Posix-Threads, die direktivengesteuerte Programmierung mit OpenMP und die noch recht junge Programmiersprachenerweiterung UPC. Alle drei Techniken eignen sich zur parallelen Programmierung von Shared-Memory Systemen, wobei UPC eine gewisse Sonderstellung einnimmt. Es wird untersucht, welche Technik sich hinsichtlich erreichbarer Performance und Benutzerfreundlichkeit optimal zur Parallelisierung von Programmen aus dem HPC-Umfeld eignet.

Im Kapitel 5 werden Werkzeuge zur effizienten Umsetzung von C++ Sprachkonstrukten im Zusammenhang mit OpenMP behandelt. Es wird untersucht, was bei der Parallelisierung von C++ Programmen, besonders beim Einsatz von OpenMP, zum Erreichen einer hohen Performance beachtet werden muss.

Im Anhang wird auf ausgesuchte technische Aspekte der Implementierungen eingegangen. Dabei werden die Parallelisierungen der DROPS Programmkerne vorgestellt sowie die Implementierung des Thread-Pools für die Posix-Threads erläutert.

# 2 Grundlagen

In diesem Kapitel werden einige ausgewählte Grundlagen und Begriffe erläutert sowie die eingesetzte Hardware und ihre wesentlichen Eigenschaften vorgestellt.

## 2.1 Rechnerarchitekturen

In der Literatur findet man eine Vielzahl von Klassifikationssystemen für Computer bzw. Rechnerarchitekturen. Eine einfache Klassifikation schlug Flynn schon 1966 in [Flynn 66] vor, sie ist aber auch heute noch gültig und wird oft verwendet. Sie ist z.B. in [Hennessy & Patterson 03] detailliert beschrieben. Nach Flynn werden vier grundlegende Rechnerarchitekturen anhand der Art der Instruktions- und Datenströme unterschieden:

- *SISD* steht für *single instruction, single data*. Diese Kategorie stellt die Einzelprozessorsysteme dar.
- *SIMD* steht für *single instruction, multiple data*. Eine Instruktion wird von mehreren Prozessoren auf verschiedene Datenströme angewendet. Zu dieser Kategorie zählen vor allem die sog. Vektorrechner. Aber auch die auf modernen Desktop-CPUs zu findenden Multimedia-Erweiterungen kann man zu dieser Kategorie zählen.
- *MISD* steht für *multiple instruction, single data*. Aus dieser Kategorie findet man bisher eigentlich keine kommerziellen CPUs, außer Stream-Prozessoren für spezielle Anwendungen. Nach der Kategorienbezeichnung werden auf einen Datenstrom hintereinanderfolgend mehrere Funktionen angewendet.
- *MIMD* steht für *multiple instruction, multiple data*. Bei Systemen aus dieser Kategorie hat jeder Prozessor seinen eigenen Instruktionsstrom und seinen eigenen Datenstrom, mit denen er arbeitet. Beinahe alle parallelen Architekturen sind nach diesem Schema aufgebaut, da es sowohl bezüglich der darauf ausführbaren Programme eine hohe Flexibilität bietet, als auch kosteneffizient aus mehreren einzelnen CPUs zusammengebaut werden kann.

Auf einem parallelen System nach MIMD hat jeder Prozessor (CPU) seinen eigenen Instruktionsstrom. Diese abstrakte Beschreibung kann in der Praxis auf zwei Arten umgesetzt werden:

- *Prozesse*: ein Prozess ist eine ausgeführte Instanz eines Programms, wobei ein Prozess immer zu einem Programm gehört, ein Programm aber auch aus mehreren Prozessen bestehen kann (z.B. bei MPI). Zu einem Prozess gehören im Allgemeinen ein Teil des virtuellen Speichers, Ressourcen vom Betriebssystem (z.B. Dateihandles), Sicherheitsinformationen (z.B. Userid) sowie der Zustand des Prozesses (z.B. Registerwerte).

- *Threads*: Threads bieten einem Programm die Möglichkeit, sich bei der Ausführung in zwei oder mehrere unabhängige Instruktionsströme aufzuspalten. In vielen Aspekten gleichen sich Threads und Prozesse, allerdings teilen sich mehrere Threads eines Programms z.B. den virtuellen Speicher und die Ressourcen des Betriebssystems. Deshalb werden Threads oft auch leichte (light-weight) Prozesse (LWP) genannt.

Die MIMD Systeme kann man weiter nach der Art ihrer Speicherorganisation unterscheiden. Es gibt zahlreiche Quellen, welche insbesondere parallele Rechnerarchitekturen detailliert vorstellen, als Beispiel seien hier [Dowd & Severance 98] und [Hennessy & Patterson 03] genannt.

### 2.1.1 Distributed-Memory Systeme

Auch wenn die Programmierung von Distributed-Memory Systemen in dieser Diplomarbeit nicht betrachtet wird, stellen sie doch eine sehr wichtige und weit verbreitete Klasse von parallelen Computersystemen dar und sollen kurz beschrieben werden.

Bei Distributed-Memory Systemen ist der Speicher physikalisch verteilt. Die grundlegende Architektur wird in Abbildung 1 abgebildet. Jeder Knoten des Distributed-Memory Systems ist mit einem Prozessor und eigenem Speicher ausgestattet, sowie üblicherweise mit I/O-Funktionalität. Die Verbindung erfolgt über ein schnelles Netzwerk, so dass jeder Knoten auf den Speicher der anderen Knoten zugreifen kann, wobei der Remote-Speicherzugriff natürlich langsamer ist als der Zugriff auf den lokalen Speicher. Ebenfalls ist es möglich, dass mehrere Ebenen im Netzwerk existieren, so dass eine gewisse Lokalität vorhanden ist.

Weiter ist es möglich, dass jeder Knoten aus nicht nur einem Prozessor besteht, sondern aus mehreren. Die zur Zeit weit verbreiteten sog. *Cluster* sind prinzipiell Distributed-Memory Systeme, deren einzelne Knoten Shared-Memory Systeme sind.

### 2.1.2 Shared-Memory Systeme

Die Programmierung von Shared-Memory Systemen steht in dieser Arbeit im Mittelpunkt, so dass hier etwas detaillierter auf die Architektur eingegangen werden muss.

Bei Shared-Memory Systemen gibt es einen globalen Speicher auf den alle Prozessoren zugreifen können, wobei jeder Prozessor noch eine Anzahl von *Cache*-Ebenen lokal besitzen kann. Cache ist ein besonders schneller, aber dafür kleiner Speicher direkt auf oder nahe bei der CPU. Die grundlegende Architektur ist in Abbildung 2 dargestellt.

Auf sog. *Symmetrischen Multi-Prozessor Systemen* (SMP) sind zwei oder mehrere identische Prozessoren mit einem Hauptspeicher verbunden. Auf SMP-Systemen kann jeder Prozessor unabhängig von den anderen Prozessoren an einer Aufgabe arbeiten, unabhängig von der Lage der Daten im Speicher. Für jeden Prozessor ist der Zugriff auf den Speicher gleich schnell. Bei entsprechender Fähigkeit des Betriebssystems können Prozesse bzw. Threads flexibel zwischen den CPUs hin und her geschoben werden.

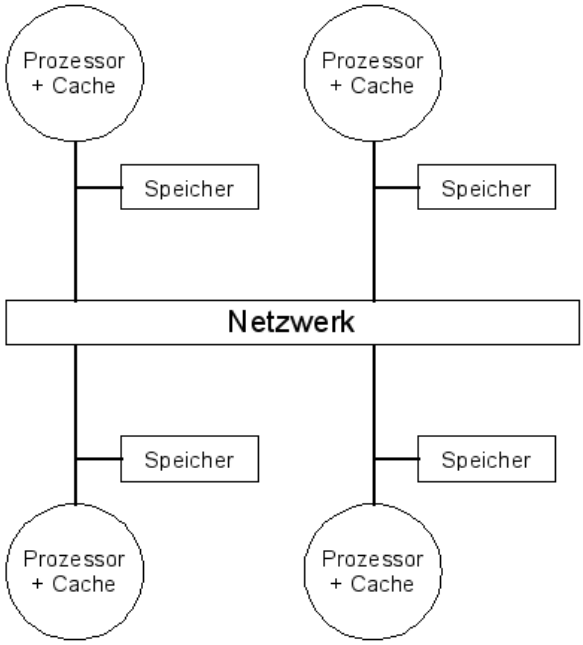


Abbildung 1: Distributed-Memory Architektur.

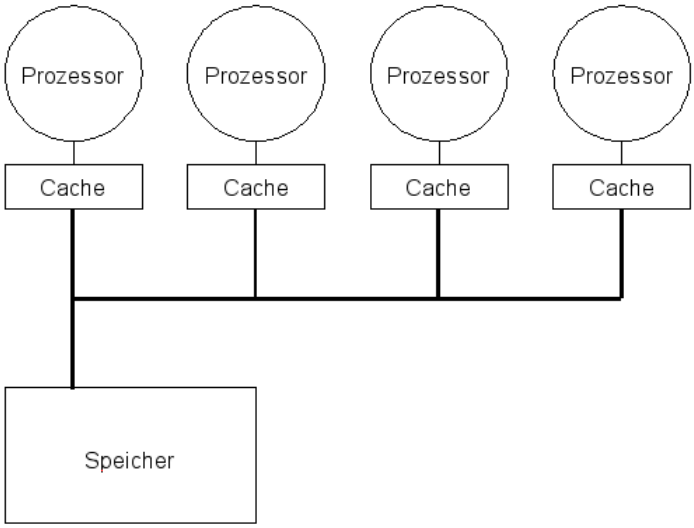


Abbildung 2: Shared-Memory Architektur.

Neben dem Vorteil der Flexibilität ist die Tatsache, dass moderne Prozessoren deutlich schneller sind als der Speicherzugriff, auf SMP-Systemen ein Nachteil. Im Allgemeinen müssen sich alle Prozessoren die Speicheranbindung teilen. Dies kann die Skalierung stark begrenzen.

Die im folgenden Abschnitt beschriebene Architektur gehört ebenfalls zur Klasse der Shared-Memory Systeme und versucht die Skalierung zu verbessern.

### 2.1.3 ccNUMA Systeme

Auf Systemen mit *NUMA* (*Non-Uniform Memory Architecture*) Architektur hängt die Speicherzugriffsgeschwindigkeit von der Position des Datums relativ zur Position des Prozessors ab. Auf einer NUMA-Architektur kann ein Prozessor also schneller auf seinen lokalen Speicher zugreifen, als auf entfernten (remote) Speicher.

Durch diese Architektur wird der oben beschriebene Engpass bei der Speicheranbindung in SMP-Systemen dadurch aufgehoben, dass jeder Prozessor mit voller Geschwindigkeit auf seinen eigenen Speicher zugreifen kann. Nach wie vor ist es aber möglich, dass ein Prozessor auf den Speicher eines anderen Prozessors zugreift. Dabei kann er allerdings eine spürbare Performanceverringerung erfahren. Bei entsprechender Programmierung, insbesondere bei optimaler Verteilung der Daten, kann eine NUMA-Architektur einen Geschwindigkeitsvorteil gegenüber SMP-Systemen vom Faktor der Anzahl der Prozessoren bieten.

Nahezu alle modernen Prozessordesigns setzen mittlerweile mindestens eine Cache-Ebene ein. Bei einer sog. *Cache-coherent NUMA*-Architektur (ccNUMA) wird der zu einem Speicherdatum gehörende Inhalt einer Cache-Line, falls es auf mehreren Prozessoren verwendet wird, kohärent gehalten. Das heißt, es werden durch ein gewähltes Verfahren Gültigkeitsprobleme vermieden, falls mehrere Prozessoren auf ein Speicherwort im Hauptspeicher zugreifen. Es ist aber durchaus eine Inkonsistenz zwischen Hauptspeicher und lokalen Caches erlaubt, solange die Zugriffe immer gültige Ergebnisse liefern. Diese Koheränzerhaltung führt zwar zu einem gewissen Aufwand in der Hardware-Implementierung, allerdings wäre auf Systemen ohne diese Cache-Kohärenz die Kompatibilität mit allgemeinen SMP-Systemen nicht gewährleistet.

## 2.2 Parallelisierung

In den folgenden drei Abschnitten sollen die wichtigsten Kriterien zur Beurteilung der Güte einer Parallelisierung vorgestellt werden und die generelle Art zur Programmierung von Distributed-Memory Systemen (Message Passing) sowie von Shared-Memory Systemen (Arbeitsverteilung) beschrieben werden.

### 2.2.1 Speedup und Effizienz

Die Größe *Speedup* $(S)$ berechnet sich als

$$S_p = \frac{T_1}{T_p}$$

und zeigt damit an, um wie viel eine parallele Ausführung eines Algorithmus schneller ist als die zugehörige serielle Ausführung, wenn $T_1$ die Zeit der Ausführung unter Benutzung einer CPU ist und $T_p$ die Zeit unter Benutzung von $p$ CPUs. Der ideale Speedup ist der *lineare* Speedup mit $S_p = p$. Im Fall eines linearen Speedups führt die Verdopplung der Anzahl der eingesetzten CPUs zu einer Halbierung der Laufzeit des Algorithmus.

Das Gesetz von Amdahl [Amdahl 67] (*Amdahl's Law*) besagt, dass der erreichbare Speedup nicht durch die Anzahl der verfügbaren Prozessoren bestimmt wird, sondern durch den Algorithmus selbst. Dies wird dadurch begründet, dass auch unter optimistischsten Annahmen für den Overhead, welcher durch die Parallelisierung eingefügt wird, irgendwann ein Punkt erreicht wird, an dem der Algorithmus nicht mehr weiter parallelisiert werden kann und ein serieller Teil verbleibt. Formal ausgedrückt ist dies

$$S_{max} = \frac{1}{F + \frac{(1-F)}{p}}$$

wobei $S_{max}$ der maximal erreichbare Speedup unter Einsatz von $p$ Prozessoren ist. $F$ ist dabei der Anteil des Algorithmus, der nicht parallelisiert werden kann und damit nicht schneller werden kann. $(1-F)$ ist der parallelisierte Anteil.

Die *Effizienz* $(E)$ ist als

$$E = \frac{S_p}{p}$$

definiert und hat einen Wert der zwischen 0 und 1 liegt. Die Effizenz gibt an, wie gut die eingesetzten Prozessoren beim Lauf des Algorithmus benutzt werden, in Bezug zum Aufwand, der für Kommunikation und Synchronisation eingesetzt werden muss. Wenn ein Algorithmus einen linearen Speedup hat, ergibt sich eine Effizienz von 1. Schwierig zu parallelisierende Probleme können z.B. eine Effizenz von $\frac{1}{\log p}$ haben, welche dann mit steigender Anzahl von eingesetzten CPUs gegen 0 strebt.

### 2.2.2 Message Passing

In einer *Message-Passing* Umgebung wird eine Menge von Funktionen zur Verfügung gestellt, mit denen ein Programm zur parallelen Ausführung aufgespalten werden kann. Die Daten werden zwischen den verschiedenen Prozessen verteilt und als Nachrichten unter den Prozessen ausgetauscht. Der Compiler ist hierbei nicht direkt in

die Parallelisierung einbezogen. Typischerweise werden Distributed-Memory Systeme nach diesem Ansatz programmiert, jedoch können auch Shared-Memory Systeme mittels Message Passing programmiert werden.

Die beiden bekanntesten Message-Passiung Umgebungen sind *PVM* [Sunderam 90] und *MPI* [Forum 95, Forum 03]. Beide Programmierkonzepte ähneln sich in den grundlegenden Bereichen des Message-Passing. Es existieren Befehle in beiden Bibliotheken, ab deren Aufruf auf den verschiedenen Knoten gleichzeitig eine Programmkopie gestartet wird (z.B. *MPI_Init*()), im Allgemeinen erfolgt der Aufruf dieser Befehle zu Beginn des Programms. Ebenfalls sind Befehle zum Beenden der Programmkopien auf allen Knoten sowie der Termination der Umgebung verfügbar (z.B. *MPI_Close*()). Somit besteht ein MPI-Programm aus mehreren Prozessen, die auf einem Parallelrechner, unterschiedlichen Computern und auch in heterogenen Umgebungen laufen können.

Notwendig sind ebenfalls Befehle zum eigentlichen Senden (z.B. *MPI_Send*()) und Empfangen (z.B. *MPI_Recv*()) der Daten. Hierbei können sowohl der Sender bzw. Empfänger als auch eine ID einer Nachricht angegeben werden. Auch das Senden einer Nachricht von einem Prozess an alle anderen Prozesse ist möglich (z.B. *MPI_Broadcast*()).

### 2.2.3 Arbeitsverteilung auf Shared-Memory Systemen

Zur Programmierung von Shared-Memory Systemen müssen die Daten nicht auf die Knoten verteilt werden, sondern verbleiben im gemeinsamen Hauptspeicher. Die Parallelisierung erfolgt dadurch, dass die parallel arbeitenden Programmströme die Arbeit mit den gemeinsamen Daten untereinander aufteilen. Die Art der Aufteilung kann durchaus sehr wichtig für die erreichbare Performance sein, insbesondere auf NUMA Architekturen.

Bei den meisten Konzepten zur Programmierung der Arbeitsverteilung, wie z.B. bei den Posix-Threads und bei OpenMP, werden Threads als Implementierung der parallelen Programmströme verwendet. Bei der Parallelisierungstechnik UPC wird ebenfalls der Begriff Thread verwendet, in Abhängigkeit des verwendeten Kommunikationssystems kann damit sowohl ein Prozess als auch ein Thread nach obiger Definition bezeichnet. werden. In dieser Diplomarbeit werden die drei wichtigsten Programmierkonzepte zur parallelen Programmierung von Shared-Memory Systemen in Kapitel 4 detailliert vorgestellt.

Bei der parallelen Programmierung ergeben sich neue Klassen von möglichen Fehlerquellen in einem Programm. Ein insbesonders in für Shared-Memory Systeme programmierten Programmen oft auftretender Fall ist die sog. *Race-Condition*. Generell wird mit einer *Race-Condition* ein unerwünschter Zustand im Programm bezeichnet, wenn zwei oder mehrere Operationen gleichzeitig durchgeführt werden, aber aus Gründen der Programmlogik oder der Implementierung der Operationen diese zur korrekten Durchführung in einer speziellen Reihenfolge ausgeführt werden müssen. Allerdings ist das Auftreten einer *Race-Condition* nicht immer deterministisch.

Dies soll am Beispiel eines *Lost-Update* demonstriert werden. Damit eine solche *Race-Condition* entsteht, müssen zwei gleichzeitig laufende Threads folgende Aktionen durchlaufen:

1. Wert *lesen*: der Wert wird aus dem externen Speicher in den internen Speicher gelesen.
2. Wert *ändern*: der Wert wird im internen Speicher geändert.
3. Wert *schreiben*: der Wert wird aus dem internen Speicher zurück in den externen Speicher geschrieben.

Bei einer entsprechend gewählten Reihenfolge, nämlich dass zuerst der erste Thread und anschließend der andere Thread die Aktionen ausführt, wird ein richtiges Ergebnis berechnet. Falls alle Aktion gleichzeitig ausgeführt werden, also falls beide Threads lesen bevor ein Thread schreibt, ist das Ergebnis bei entsprechender Änderungsaktion falsch.

## 2.3 Verwendete Hardware

Für die Entwicklung sowie die Messungen und Experimente wurden die Hochleistungsrechner am Rechenzentrum der RWTH Aachen benutzt. Sowohl die UltraSPARC-IV basierten Systeme als auch die Opteron basierten Systeme sind Shared-Memory Architekturen, wobei die Opteron Plattform wegen des HyperTransport-Busses recht deutlich ausgeprägte ccNUMA-Eigenschaften besitzt. Die beiden Gattungen von Systemen werden nun kurz beschrieben, detaillierte Informationen zur Benutzung der Systeme sind in [an Mey & Sarholz$^+$ 05] zu finden.

### 2.3.1 Sun Fire UltraSPARC-IV Server

Zwar stehen am Rechenzentrum mehrere UltraSPARC-IV basierte Systeme zur Verfügung, aber für Messungen auf Systemen mit einer flachen Speicherarchitektur wurden zur Gewährleistung der Vergleichbarkeit ausschließlich Sun Fire E6900 Server verwendet. Insgesamt sind 16 solcher Knoten installiert, wobei zweimal jeweils 8 Systeme durch ein Sun Fire Link Netzwerk zu einem Cluster verbunden sind.

Jedes Sun Fire E6900 System besitzt 24 UltraSPARC-IV Prozessoren, die mit 1,2 GHz getaktet sind, und 96 GByte Hauptspeicher. Da es sich bei den UltraSPARC-IV Prozessoren um *Dual-Core* CPUs handelt, stehen insgesamt 48 Prozessorkerne in einem System zur Verfügung, die vom Betriebssystem Solaris wie eigenständige Prozessoren behandelt werden.

Das Speichersystem ist wie folgt organisiert: Sechs CPU-Boards mit jeweils vier CPUs sind über eine *Crossbar* verbunden. Die gesamte Speicherbandbreite des Systems summiert sich zu $9,6$ GByte/s für alle 24 Prozessoren und ist wegen des eingesetzten Protokolls zur Erhaltung der Cache-Koheränz auf diesen Wert limitiert. Die

maximale lokale Bandbreite von einem Prozessor zum Speicher beträgt 2,4 GByte/s. Zwar ist der Zugriff auf lokalen Speicher auf einem CPU-Board schneller als auf entfernten Speicher, allerdings sind die Unterschiede so gering (die Latenz schwankt zwischen 230 und 280 ns), dass das Speichersystem als flach bezeichnet werden kann und Datenlokalität keinen dominierenden Einfluss besitzt.

Diese Systeme laufen unter dem Betriebssystem Solaris [Sun 05a] in Version 9. Als Compiler wurde in dieser Arbeit der Sun C++ Compiler in Version 10 [Sun 05b] verwendet. Solaris bietet viele moderne Funktionen wie z.B. *CPU-Binding*. Dabei werden Prozesse oder Threads an eine CPU gebunden auf der sie ausgeführt werden, um Laufzeitschwankungen durch Migration der Prozesse bzw. Threads zu vermeiden.

### 2.3.2 Sun Fire Opteron Server

Die am Rechenzentrum zur Verfügung stehenden Systeme mit AMD Opteron Prozessor sind in Sun Fire V40z Systemen verbaut. Insgesamt stehen 64 Knoten mit jeweils vier Opteron 848 Prozessoren und 8 GByte Speicher unter den Betriebssystemen Linux, Windows und Solaris zur Verfügung. Weiter stehen vier Knoten mit jeweils vier Opteron 875 Dual-Core Prozessoren und 16 GByte Speicher ausschließlich unter dem Betriebssystem Solaris zur Verfügung. Jeder Prozessor ist dabei mit 2,2 GHz getaktet.

Das Speichersystem ist wie folgt organisiert: Jedem Prozessor bzw. Prozessorkern stehen 2 GByte lokaler Speicher zur Verfügung, auf den mit 5,3 GByte/s auf den 848 Modellen und mit 6,4 GByte/s auf den 875 Modellen zugegriffen werden kann. Der Zugriff auf den entfernten Speicher erfolgt über den HyperTransport-Bus, der maximal 6,4 GByte/s auf den *Single-Core* und 8,0 GByte/s auf den *Dual-Core* Systemen übertragen kann. Hierbei wird deutlich, dass falls mehrere Prozessoren auf den Speicher eines einzelnen Prozessors zugreifen, die Performance absinkt.

Diese Systeme stehen am Rechenzentrum unter drei verschiedenen Betriebssystemen zur Verfügung:

1. Solaris 10: hier wurde der Sun C++ Compiler in Version 10 verwendet. Er unterstützt die Erstellung sowohl von 32bit als auch 64bit Programmen. Solaris bietet auch auf der Opteron Plattform moderne Fähigkeiten, welche in den anderen Betriebssystemen bisher nicht zur Verfügung stehen. Dazu zählt z.B. die Möglichkeit der Migration von Speicher.

2. Linux 2.6: hier wird die Distribution Fedora Core 4 verwendet. Als Compiler kommt der Intel C++ Compiler in Version 9 zum Einsatz. Ebenfalls unterstützt dieser die Erstellung sowohl von 32bit als auch von 64bit Programmen. Zwar unterstützt dieser Compiler die Opteron CPU nicht explizit, in mehreren Benchmark hat er allerdings die beste Performance unter den OpenMP-fähigen Compilern gezeigt. Eine Migration von Speicher wird nicht unterstützt, jedoch das *CPU-Binding*.

3. Windows 2003: hier wird der C++ Compiler aus Microsoft Visual Studio 2005 [Microsoft 05] verwendet. Seit dieser Version wird auch die Parallelisierung mit OpenMP unterstützt. Auf den Systemen ist bisher nur die Unterstützung zur Erzeugung von 32bit Programmen mit diesem Compiler installiert. Eine Migration von Speicher ist nicht möglich. Ein *CPU-Binding* muss manuell im Programm durch den Aufruf entsprechender Bibliotheksfunktionen aktiviert werden.

# 3 Vorstellung und Parallelisierung der untersuchten Programme

## 3.1 DROPS

Das Ziel des DROPS Softwarepakets [Gross & Peters$^+$ 02], das sich noch stark in der Entwicklung befindet, ist es, ein effizientes Softwaretool zur numerische Simulationen von dreidimensionalen inkompressiblen Mehrphasen-Flusssystemen zu erstellen. Zwar gibt es bereits einige Softwarepakete im CFD-Bereich, allerdings nicht als Black-Box Löser für komplexe Flussprobleme wie z.B. die physikalische Simulation des Verhaltens am Phaseninterface eines flüssigen Tropfens in einer weiteren Flüssigkeit. Dabei soll ebenfalls untersucht werden, wie moderne numerische Methoden, z.B. adaptive Gitter und iterative Löser, mit der für die zu untersuchenden physikalischen Phänomene notwendigen Flexibilität in einem C++ Code vereint werden können.

Die Hauptelemente des Lösungsverfahrens sind dabei:

- *Gitter*: es werden Multi-Level Gitter (Triangulierungshierarchien) verwendet. Zu jedem Level ist ein stabiles Tetraedergitter assoziiert, welches keine hängenden Knoten hat, so dass Multi-Grid Verfahren eingesetzt werden können.

- *Zeitdiskretisierung*: für eine stabile Zeitdiskretisierung der instationären Systeme wird ein implizites Teilschrittschema verwendet.

- *Ortsdiskretisierung*: es wird eine Finite Elemente Diskretisierung für Druck, Geschwindigkeit und Levelset eingesetzt. Dabei beschreibt das Levelset die Phasengrenze.

- *Iterative Lösungsverfahren*: das Navier-Stokes-Levelset System wird durch eine Fixpunktiteration entkoppelt. Ebenfalls durch eine Fixpunktiteration wird das Navier-Stokes System in ein Stokes Problem und ein nichtlineares Konvektion-Diffusion Problem gesplittet. Für die entstandenen Probleme können sowohl Krylov-Teilraumverfahren als auch Multi-Grid Verfahren verwendet werden, wobei in dem hier betrachteten Datensatz lediglich Krylov-Teilraumverfahren zum Einsatz kommen.

Die hier vorgestellte Aufgabenstellung von DROPS wird in Abbildung 3 dargestellt. Das Modell besteht aus 11000 Tetraedern, von denen sich 75% in der verfeinerten Region befinden. Es wird ein Tropfen Silikonöl in ein System mit schwerem Wasser (Deuteriumoxid, $D_2O$) gegeben. Der Fluss bzw. die Geschwindigkeit vor Hinzugabe des Tropfens und nach Hinzugabe und Berechung durch DROPS sind dargestellt.

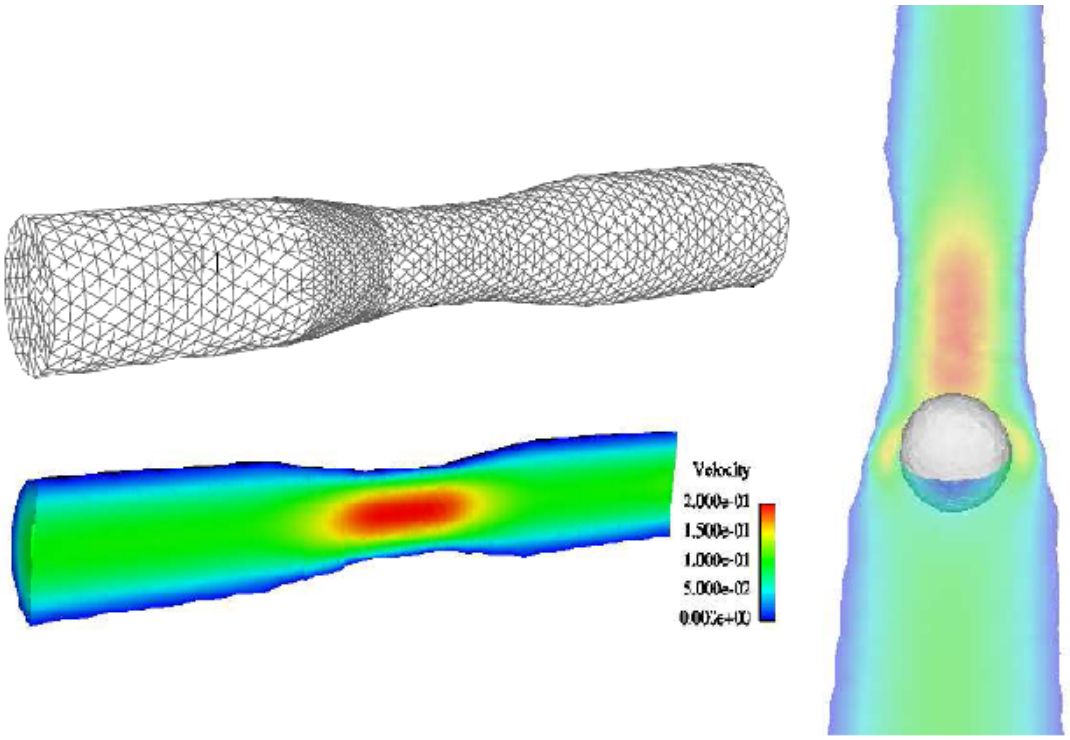


Abbildung 3: DROPS: Tropfen in Geometrie.

### 3.1.1 Parallelisierung

Im Rahmen der am Rechenzentrum durchgeführten Arbeiten sollten die laufzeitintensivsten Teile von DROPS parallelisiert werden, um den Entwicklungsprozess zu beschleunigen und erste Erfahrungen im Hinblick auf eine höhere Parallelisierung auf Ebene der Gittermodellierung zu sammeln. Dazu wurde zunächst ein detailliertes Laufzeitprofil erstellt. Die in Tabelle 1 gezeigten Zahlen wurden auf einem Sun Fire E6900 System gemessen.

Alle im Laufzeitprofil dargestellten Routinen wurden parallelisiert, so dass insgesamt 99% der Laufzeit von DROPS bearbeitet wurden. Auf den meisten betrachteten

Tabelle 1: DROPS: Laufzeitprofil auf Sun Fire E6900.

| **Klasse** | **% Laufzeit** | **Beschreibung** |
|---|---|---|
| SETUP | 52% | Vier Setup-Routinen zum Aufbau der Steifigkeitsmatrizen. |
| GMRES | 23% | PCG Lösungsverfahren inkl. dünnbesetzter Matrix-Vektor-Multiplikation. |
| PCG | 21% | GMRES Lösungsverfahren. |
| LINCOMB | 3% | Linearkombination zweier dünnbesetzter Matrizen. |
| Sonstige | 1% | Sonstige Routinen des DROPS Codes. |

Plattformen, dabei sind die am Rechenzentrum verwendeten Systeme eingeschlossen, hat die Parallelisierung aber nicht die in sie gesetzten Ziele bezüglich der Skalierung erreicht.

Im Rahmen dieser Diplomarbeit werden daher die gewählten Ansätze genauer betrachtet und es werden Verfahren zur Verbesserung der Performance sowohl mit Mitteln des Betriebssystems als auch mit Mitteln der Programmiersprache C++ vorgestellt, um die Skalierung zu verbessern. Im Folgenden wird die Parallelisierung kurz vorgestellt und die Performance auf den in dieser Diplomarbeit betrachteten Plattformen gezeigt. Weitere Informationen zur Parallelisierung sowie die Performance auf den restlichen Plattformen sind in [Terboven & Spiegel$^+$ 05a] zu finden.

Der Aufbau der Steifigkeitsmatrizen konnte vollständig parallelisiert werden. Da die entstehenden Matrizen dünnbesetzt sind, wird in DROPS das CRS-Format (*Compressed Row Storage*) zur Darstellung der Matrizen im Programm verwendet, wobei nur die Nichtnullelemente gespeichert werden. Die Matrixelemente werden dabei in einem Array *val* gespeichert, welches den Typ *std*::*valarray*<*double*> besitzt, weiter werden zwei Hilfsarrays vom Typ *std*::*valarray*<*size_t*> zur Darstellung der Position der Nichtnullelemente in der Matrix benötigt. Der Vorteil dieser Datenstruktur liegt darin, dass nur genau die benötigten Matrixelemente gespeichert werden, sowie die Berechnungen effizient implementiert werden können. Der Nachteil ist, dass das Einfügen von neuen Elementen in die Matrix aufwändig ist. Aus diesem Grund wird in den Setup-Routinen eine Technik zum Aufbau der Matrizen verwendet, die auf dem Datentyp *std*::*map* basiert. Dabei wird eine *std*::*map* für jede Zeile der Matrix verwendet und darin die Nichtnullemente mit ihrer Spaltenangabe gespeichert.

Aus dieser Parallelisierung erwuchsen zwei Probleme. Die Schleife über die Gitterelemente zur Berechnung der Matrizen ist nicht von der kanonischen Form, wie sie die OpenMP Spezifikation fordert, da die Iteratortechnik aus der STL eingesetzt wird. Die Parallelisierung solcher Schleifen mit OpenMP wird in Abschnitt 5.5 behandelt. Der Einsatz von *std*::*map* zur Zwischenspeicherung der Nichtnullelemente ist zwar sehr elegant, allerdings führt er unter den Betriebssystemen Solaris und Windows zu Problemen beim Speichermanagement. Methoden zur Verbesserung der Speicherverwaltung für diesen Anwendungsfall werden in Abschnitt 5.3 vorgestellt.

Es wurde auch eine Parallelisierung des PCG- und des GMRES-Lösungsverfahren durchgeführt. Die Skalierung auf den Sun Fire E6900 Systemen, die eine flache Speicherarchitektur haben, ist gut und nur dadurch begrenzt, dass das verwendete Vorkonditionierungsverfahren (Gauss-Seidel) nur teilweise parallelisiert werden kann. Deshalb wurde, wie in [Terboven & Spiegel$^+$ 05b] beschrieben, ein modifizierter Vorkonditionierer verwendet, der zwar das numerische Verhalten verändert, aber insgesamt zu einer besseren Performance bei mehr als vier Threads führt.

Ebenfalls gab es bei dieser Parallelisierung einige Punkte, die noch weiterer Untersuchung bedurften. Auf der NUMA-Architektur der Sun Fire V40z Systeme brach die Performance der Lösungsverfahren ein, da die darin dominierende Matrix-Vektor

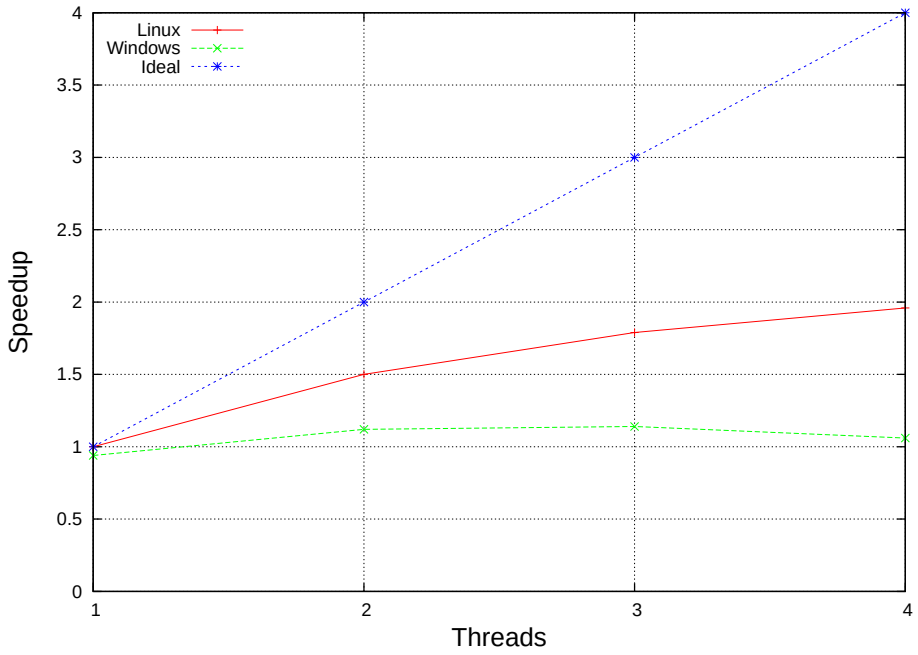


Abbildung 4: DROPS: Speedup auf NUMA-Architektur auf Sun Fire V40z.

Multiplikation der Routine *y_Ax* nicht skalierte. Ansätze dazu werden im Abschnitt 5.2 behandelt. Ebenfalls war zu untersuchen, welche Ansätze sich bei gegebenen objektorientierten C++ Codes zur Parallelisierung anbieten, dies wird in Abschnitt 5.4 diskutiert.

### 3.1.2 Performance

In der Grafik in Abbildung 4 ist der Speedup des DROPS Code auf einem Sun Fire V40z System unter Linux und Windows dargestellt. Auch mit vier Threads wird unter Linux kein Speedup von mehr als dem Faktor zwei erreicht. Das liegt vor allem daran, dass die NUMA-Architektur der Sun Fire V40z Systeme nicht berücksichtigt wird. Unter Windows wird die Performance bei mehr als einem Thread darüber hinaus noch dadurch verringert, dass die Setup-Routinen eine negative Skalierung aufweisen, also langsamer werden.

Mit den in dieser Arbeit vorgestellten Methoden wird die Skalierbarkeit von DROPS auf allen Plattformen verbessert.

## 3.2 FIRE

Das Image-Retrieval System FIRE [Deselaers 05] (*Flexible Image Retrieval Engine*) wurde im Rahmen der Diplomarbeit von Thomas Deselaers [Deselaers 03] und Daniel Keysers am Lehrstuhl für Informatik 6 der RWTH Aachen entwickelt. Es gibt zwar bereits mehrere inhaltsbasierte Image-Retrieval Systeme, das Ziel bei der Entwicklung von FIRE aber war es, eine Möglichkeit zum Vergleich von Bildmerkmalen (Features) und Bildvergleichsmaßen (Distanzen) zu schaffen. Weiter wird untersucht, wie verschiedene Features zueinander in Korrelation stehen.

Die Bandbreite der verschiedenen Features, die eingesetzt werden, ist groß. Im folgenden werden exemplarisch einige Beispiele genannt:

- *Pixelwerte*: Die Farbwerte der Pixel werden direkt als Bildmerkmale verwendet. Dazu werden die Bilder z.B. auf eine Größe von $32 \times 32$ Pixeln skaliert und pixelweise verglichen. Zum Vergleich dient entweder die recht einfache Euklidische Distanz oder das Image Distortion Modell, ein nicht-lineares Verformungsmodell, welches einen deutlich höheren Rechenaufwand verursacht als die Euklidische Distanz, dafür aber oftmals viel bessere Ergebnisse erzielt.

- *Farbhistogramme*: Die Verteilung von Farben im Bild wird betrachtet. Die Distanz kann dabei z.B. informationstheoretisch als Jeffrey-Divergenz berechnet werden. Es können aber auch aufwändigere Distanzen, wie z.B. die Earth-Movers Distanz, verwendet werden.

- *Lokale Merkmale*: Aus den Bildern werden sehr viele quadratische Teilbilder extrahiert und die Bilder werden dann anhand dieser Ausschnitte verglichen. Bei dieser Methode wächst zwar die zu verarbeitende Datenmenge und damit auch die notwendige Rechenzeit, aber es können sehr gute Ergebnisse erzielt werden.

- *Regionsbasierte Features*: Das Bild wird automatisch in Teilbereiche unterteilt, die grob den Objekten entsprechen. Bilder können dann anhand dieser Segmente verglichen werden.

FIRE wird als Forschungssystem entwickelt. Beim ImageCLEF (*Cross Language Image Retrieval Task*) werden Image-Retrieval Systeme anhand verschiedener Datenbanken verglichen. Zusätzlich zu den eigentlichen Bilddaten liegen hierbei noch weitere Informationen in Form von strukturierten oder unstrukturierten Textdokumenten vor. Bei ImageCLEF 2004 und 2005 wurden mit FIRE insbesondere bei der ausschließlich bildbasierten Suche sehr gute Ergebnisse erzielt. Bei ImageCLEF nehmen sowohl akademische als auch kommerzielle Gruppen teil.

Für die im Rahmen dieser Arbeit durchgeführte Parallelisierung war es dabei von sehr hoher Priorität, die Flexibilität z.B. beim Austausch von Distanzfunktionen und die Einfachheit der Implementierung von Erweiterungen nicht zu beschneiden. Die Flexibilität wird in großem Maße durch die objektorientierte Programmierung gewonnen, was auch bei der Parallelisierung von Vorteil ist, wie im nächsten Abschnitt gezeigt wird. FIRE unterstützt zwei Modi: den Batch-Modus, der Kommandos aus einer

Datei liest und diese abarbeitet, und einen Server-Modus, in dem FIRE die Kommandos über einen Netzwerksocket erhält und verarbeitet. Für den Servermodus stehen verschiedene Clients zur Verfgung. Einer davon ist ein interaktives Webinterface[1].

Da die verwendeten Bilddatenbanken immer größer werden und teilweise mehrere Gigabyte Speicher benötigen, ist eine Shared-Memory Parallelisierung notwendig, um mit FIRE Vorteile aus Mehrprozessorsystemen ziehen zu können. Der bisherige Ansatz, mehrere Instanzen von FIRE auf einem Rechner laufen zu lassen mit jeweils einem kleinen Teil der Anfragebilder, ist nicht mehr praktikabel, wenn eine Programminstanz bereits den Großteil des auf einem System (z.B. Sun Fire V40z) zur Verfügung stehenden Speichers benötigt. Die bei ImageCLEF 2005 verwendete Datenbank benötigt bereits ca. 5 GB Speicher zur Laufzeit, bei einer Größe von 50000 Bildern. Eine Vergrößerung der Datenbank für die Zukunft ist geplant.

### 3.2.1 Parallelisierung

Die Parallelisierung beruht auf dem formalen Modell der Image-Retrieval Aufgabe. Für eine gegebene Menge von positiven Anfragebildern $Q^+$ und einer möglicherweise leeren Menge von negativen Anfragebildern $Q^-$ wird ein Ähnlichkeitsmaß (*Score*) $S(Q^+,Q^-,X)$ für jedes Bild $X$ aus der Datenbank berechnet:

$$S(Q^+,Q^-,X) = \frac{1}{|Q^+|}\sum_{q\in Q^+} S(q,X) + \frac{1}{|Q^-|}\sum_{q\in Q^-} 1 - S(q,X)$$

$S(q,X)$ wird dabei als $e^{-D(q,X)}$ berechnet, wobei $D(q,X)$ eine gewichtete Summe von Distanzberechnungen ist:

$$D(q,X) = \sum_{m=1}^{M} w_m \cdot d_m(q_m,X_m)$$

$Q_m$ und $X_m$ bezeichnen jeweils das $m$-te Feature des entsprechenden Bildes, $w_m$ ist ein Gewichtungskoeffizient und $d_m$ schließlich ist ein Maß für die Distanz zweier Features. Für jedes $d_m$ wird $\sum_{X\in B} d_m(Q_m,X_m) = 1$ durch Renormierung erzwungen. Das Ergebnis wird nach dem *Nearest Neighbor* Prinzip ermittelt, d.h. für eine Anfrage $(Q^+,Q^-)$ werden die Bilder nach absteigendem Ähnlichkeitsmaß gelistet und es werden die $k$ Bilder $X$ mit dem höchsten Score $S(Q^+,Q^-,X)$ als Ergebnis geliefert.

Die Aufgabenstellung von FIRE wird in den beiden Abbildungen 5 und 6 dargestellt. In Abbildung 5 sind beispielhaft Bilder aus einer Bilddatenbank des IRMA (*Image Retrieval in Medical Applications*) Projektes[2] dargestellt. Durch Auswahl eines Bildes wird dieses als Anfrage an das System geschickt. Die Datenbank wird nach ähnlichen Bildern durchsucht und eine Vorgegebene Anzahl der ähnlichsten Bilder wird zurückgegeben. Das Ergebnis für das erste Bild als Anfrage ist in Abbildung 6 auszugsweise gezeigt.

[1] `http://www-i6.informatik.rwth-aachen.de/~deselaers/fire.html`

[2] `http://irma-project.org`

Abbildung 5: FIRE: Auszug von Bildklassen der IRMA Datenbank.

Abbildung 6: FIRE: Auszug des Ergebnisses für Bild 1 als Anfrage.

Für eine gegebene Menge von Anfragebildern ergeben sich direkt drei Ebenen zur Parallelisierung:

1. Die Berechnung für ein Anfragebild ist unabhängig von den Berechnungen der anderen Anfragebilder. Die Parallelisierung ergibt sich direkt, es muss allerdings darauf geachtet werden, dass bei Anfragen mit z.B. nur 32 Anfragebildern der Speedup, der mit dieser Parallelisierung erreicht werden kann, begrenzt ist. Diese Berechnung findet sich im Laufzeitprofil in der Routine $Server::batch(...)$ aus *server.cpp*. Sie ist allerdings nur im Batch-Modus verfügbar, nicht im Server-Modus, da nur dort gleichzeitig mehrere Anfragebilder gegeben sind.

2. Für ein Anfragebild ist der Vergleich mit einem Bild unabhängig von den Vergleichen mit anderen Bildern in der Bilddatenbank. Da ein Bild mit allen Bildern daraus verglichen wird, ergibt sich ein großes Potential für die Parallelisierung. Diese Berechnung findet sich in der Routine $Retriever::getScores(...)$ aus *retriever.cpp* im Laufzeitprofil. Sie ist unabhängig vom Ausführungsmodus von FIRE und somit auch im Server-Modus verfügbar, ebenso als zweite Ebene im Batch-Modus.

3. Die Abstandsberechnung $d_m$, welche je nach verwendeten Features aufwändig sein kann (z.B. Image Distortion Modell [Keysers & Gollan$^+$ 04]), kann für ein $d_i$ unabhängig von den anderen $d_j$ geschehen. Da FIRE aber stark in Entwicklung ist, wobei insbesondere mit verschiedenen Features gearbeitet wird, und die Parallelisierung die Entwicklung beschleunigen und nicht beeinträchtigen

Tabelle 2: FIRE: Laufzeitprofil auf Sun Fire E6900 für 16 Anfragebilder.

| **Routine** | **Wall exklusiv** | **Wall inklusiv** |
|---|---|---|
| ≪ Total ≫ | 6806,1 sec | 6806,1 sec |
| ImageDistortionModelDistance ::distance() | 4817,2 sec | 6760,7 sec |
| Retriever ::getScores() | 0,6 sec | 6762,4 sec |
| Server ::batch() | 0,2 sec | 6764,9 sec |

soll, wurde auf eine Parallelisierung dieser Ebene vorerst verzichtet. Diese Berechnung ist in der Routine *ImageDistortionModelDistance*::*distance*(...) aus *dist_idm.cpp* im Laufzeitprofil zu finden.

Die Tabelle 2 zeigt die drei angesprochenen Funktionen im Laufzeitprofil von FIRE auf einem Sun Fire E6900 System für 16 Anfragebilder. Vom Anteil der Laufzeit in den Routinen zur Gesamtlaufzeit von FIRE in der verwendeten Konfiguration bietet sich die Gelegenheit, sowohl auf der höheren Ebene (*Server*::*batch*()) als auch auf der tieferen Ebene (*Retriever*::*getScores*()) einen Anteil von 99,4% zu parallelisieren. Das serielle Gesamtprogramm erreicht ca. 90 MFLOPS, wobei die einzelnen Routinen, in denen die Berechnung der Distanz stattfindet, höhere Werte erreichen.

Zuerst soll die höhere Ebene der Parallelisierung von FIRE vorgestellt werden. Wie oben beschrieben, kann für ein Anfragebild unabhängig von den anderen Anfragebildern die Berechnung durchgeführt werden. Um die Interaktion mit anderen Programmen z.B. zur Nachbearbeitung nicht zu stören, muss die Ausgabe weiterhin über die Konsole erfolgen und darf nicht mit der Ausgabe von anderen, parallel bearbeiteten Anfragebildern vermischt ausgegeben werden. Der zu parallelisierende Code ist in Programmausschnitt 1 dargestellt. Für eine korrekte Ausgabe müssen immer die Paare der RECV-Ausgabe, in Zeile 11, und der SEND-Ausgabe, in Zeile 13, zusammengehören.

Da die `while`-Schleife in OpenMP nicht direkt zu parallelisieren ist und auch die korrekte Reihenfolge und Ordnung der Ausgabe sichergestellt sein soll, empfiehlt es sich, die Schleife aufzuspalten. Dies ist zusammen mit der Parallelisierung im Programmausschnitt 2 dargestellt. Innerhalb der originalen `while`-Schleife, also von Zeile 12 bis Zeile 18, werden in den Vektor `tasklist` die Aufgaben eingereiht, d.h. es wird ein Paar von zwei Strings abgespeichert, wovon der erste String die Anweisung enthält und der zweite String für das Ergebnis (Ausgabe) vorgesehen ist. Daran anschließend, von Zeile 21 bis Zeile 27, wird die gespeicherte Liste von Aufgaben parallel abgearbeitet. Abschließend, in Zeile 29 bis Zeile 33, wird die Aufgabenliste nochmals durchgegangen und jedes Paar von Anweisung und berechnetem Ergebnis wird ausgegeben.

Es wurde ein dynamischen Scheduling mit einer Chunkgröße von 1 festgelegt. Das

**Programmausschnitt 1** FIRE: Routine *batch*() aus server.cpp.

```
1 // batch mode processing
2 void Server::batch() {
3
4   [...]
5
6   if( commandfile.good())
7   {
8     getline(commandfile,cmdline);
9
10    while( not commandfile.eof()) {
11      cout << "RECV: " << cmdline << endl;
12      processCommand(cmdline,output,auth);
13      cout << "SEND: " << output << endl;
14      getline(commandfile,cmdline);
15    }
16  }
17 }
```

Scheduling von OpenMP wird in Abschnitt 4.1 beschrieben. Hiermit wird die Skalierung verbessert, wenn die Anzahl der auf der höheren Ebene verwendeten Threads die Anzahl der Anfragebilder nicht ohne Rest teilt.

In Zeile 21 wird die Anzahl der verwendeten Threads für diese parallele Region festgelegt. Dies wird im Zusammenhang mit der zweiten parallelen Region, die weiter unten vorgestellt wird, notwendig. Die Variable `iNumThreads1` kann von einer Umgebungsvariable durch den Benutzer gesteuert werden.

Nun soll die zweite Ebene der Parallelisierung von FIRE vorgestellt werden. Der Teil von *getScores*(), der relevant ist und die Rechenzeit in der Routine verbraucht, ist im Programmausschnitt 3 dargestellt. Zu parallelisieren sind die beiden `for`-Schleifen von Zeile 2 bis Zeile 8 und von Zeile 10 bis Zeile 20.

In der zweiten Schleife werden ausschließlich lokale Variablen bzw. Variablen aus dem Scope der Routine *getScores*() verwendet, so dass die Abhängigkeitsanalyse zur Einteilung der Variablen in *private* oder *shared* leicht fällt.

In der ersten Schleife ist die Zeile 4 kritisch bezüglich der Abhängigkeitsanalyse. Die Funktion *compare*() wird gleichzeitig von mehreren Threads aufgerufen. Es gibt zwei mögliche Szenarien, die zu einem Fehler führen würden:

1. Die Funktion oder weiterer von ihr benutzter Code enthält statische Daten. Falls auf diese Daten schreibend zugegriffen wird, kann bei entsprechender Änderung der Daten eine wie oben beschriebene *Race Condition* auftreten. In diesem Fall müsste der Zugriff auf die entsprechenden Daten geschützt werden, z.B. durch

**Programmausschnitt 2** FIRE: Parallelisierung von *batch*() aus server.cpp.

```
1 // batch mode processing
2 void Server::batch() {
3   typedef std::pair<std::string, std::string> rs_pair;
4   vector<rs_pair> tasklist;
5
6   [...]
7
8   if( commandfile.good())
9   {
10    getline(commandfile,cmdline);
11
12    while( not commandfile.eof()) {
13      rs_pair newtask;
14      newtask.first = cmdline;
15      newtask.second;
16      tasklist.push_back(newtask);
17      getline(commandfile,cmdline);
18    }
19  }
20
21 #pragma omp parallel num_threads(iNumThreads1)
22 {
23 #pragma omp for schedule(dynamic, 1) nowait
24   for (long l = 0; l < tasklist.size(); l++) {
25     processCommand(tasklist[l].first,tasklist[l].second,auth);
26   }
27 } // end omp parallel
28
29   vector<rs_pair>::iterator it;
30   for (it = tasklist.begin(); it != tasklist.end(); it++) {
31     cout << "RECV: " << it->first << endl;
32     cout << "SEND: " << it->second << endl;
33   }
34 }
```

eine kritische Region. Dies könnte z.B. elegant durch das objektorientierte Konzept von C++ geschehen.

2. Die Funktion benutzt die Variablen `q` (Anfragebild, Zeiger auf *ImageContainer*) und `database_` (Bilddatenbank, *Database*, was einem Vektor von Zeigern auf

**Programmausschnitt 3** FIRE: Routine *getScores*() aus retriever.cpp.

```
1  //get distance to each of the database images
2  for(long i=0;i<long(N);++i) {
3    vector<double>&d=distMatrix[i];
4    imgDists=imageComparator_.compare(q,database_[i]);
5    for(long j=0;j<long(M);++j) {
6      d[j]=imgDists[j];
7    }
8  }
9
10 for(long j=0;j<long(M);++j) {
11   double sum=0.0;
12   for (long i=0;i<long(N);++i) {sum+=distMatrix[i][j];}
13   sum/=double(N);
14   if(sum!=0.0) {
15     double tmp=1/sum;
16     for(long i=0;i<long(N);++i) {
17       distMatrix[i][j]*=tmp;
18     }
19   }
20 }
```

*ImageContainer* entspricht). Falls die Variable `q` geändert werden würde, könnte das zu einem Fehler führen. Wiederum ermöglicht die Programmiersprache C++ es dem Programmierer aber, die Abhängigkeitsanalyse schnell durchzuführen, da die Variable `q` als *const* Zeiger deklariert ist und somit nicht verändert werden darf. Die Variable `imgDists` muss als *private* deklariert werden.

Nach der durchgeführten Abhängigkeitsanalyse können die beiden Schleifen mit einem statischen Schedule direkt parallelisiert werden, wobei die Schleifen zu einer parallelen Region zusammengefasst wurden. Dies ist im Programmausschnitt 4 dargestellt.

Auch hier kann der Wert der Variablen `iNumThreads2` über eine entsprechende Umgebungsvariable vom Benutzer festgelegt werden. Damit kann die Anzahl der verwendeten Threads flexibel auf die beiden Ebenen der Parallelisierung verteilt werden.

### 3.2.2 Performance

Für die Performancemessungen wurde FIRE nur im Batch-Modus betrachtet. Da zwei Parallelisierungsebenen zur Verfügung stehen, können die Threads variabel auf den beiden Ebenen eingesetzt werden. Dies kann, wie oben beschrieben, vom Benutzer durch Umgebungsvariablen gesteuert werden.

**Programmausschnitt 4** FIRE: Parallelisierung von *getScores*() aus retriever.cpp.

```
1 #pragma omp parallel num_threads(iNumThreads2)
2 {
3 //get distance to each of the database images
4 #pragma omp for schedule(static) private(imgDists)
5 for(long i=0;i<long(N);++i) {
6    [...]
7 }
8
9 #pragma omp for schedule(static)
10 for(long j=0;j<long(M);++j) {
11    [...]
12 }
13 } // end omp parallel
```

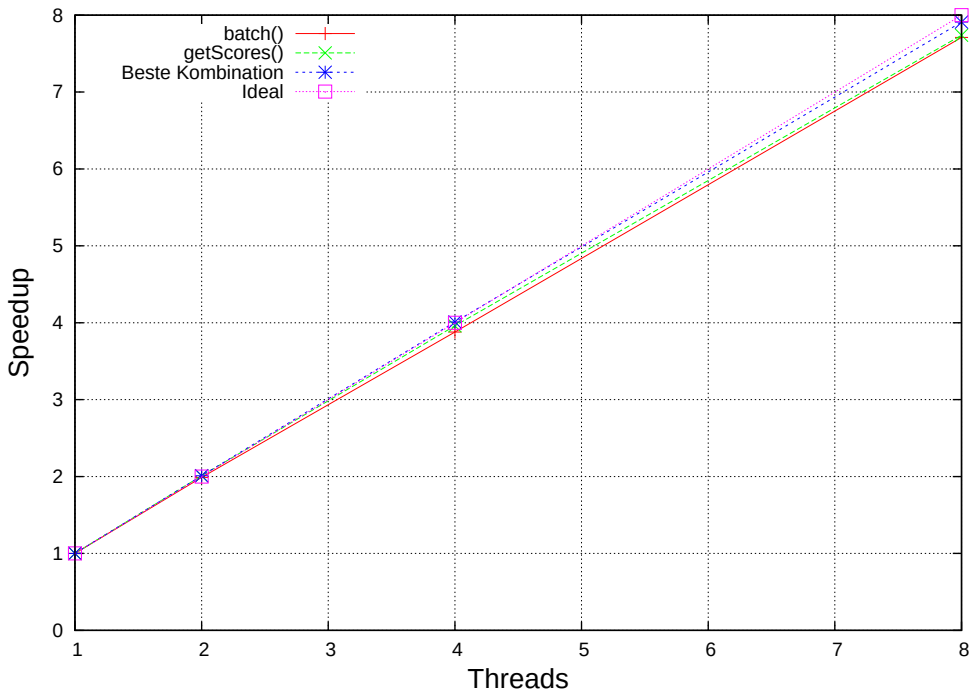


Abbildung 7: FIRE: Speedup mit 32 Anfragebildern auf Sun Fire v40z unter Solaris.

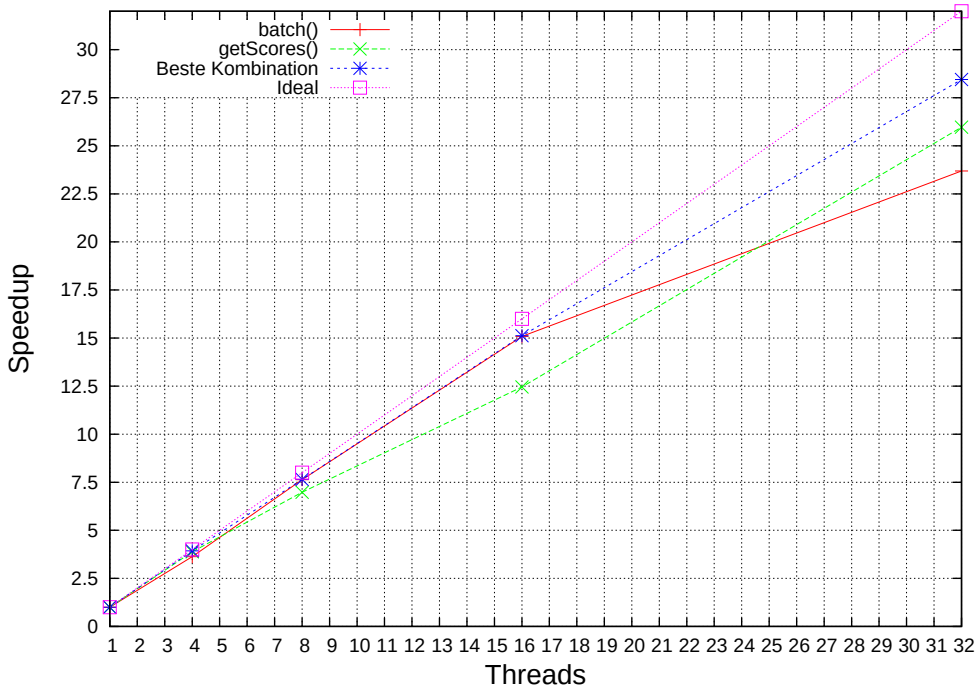


Abbildung 8: FIRE: Speedup mit 32 Anfragebildern auf Sun Fire E6900 unter Solaris.

Die Grafik in Abbildung 7 zeigt den Speedup von FIRE auf einem Sun Fire V40z System unter Solaris mit vier AMD Opteron Dual-Core Prozessoren. Die Grafik in Abbildung 8 zeigt den Speedup von FIRE auf einem Sun Fire E6900 System. Beide Messungen wurden mit 32 Anfragebildern durchgeführt. Um die beiden Ebenen der Parallelisierung direkt vergleichen zu können, wurden bis zu 32 Threads vollständig in den Ebenen, oder aber auf die Ebenen verteilt eingesetzt.

Die Datenreihe *batch*() bezeichnet die Messungen, bei denen alle verwendeten Threads in der höheren Ebene der Parallelisierung eingesetzt wurden und in der zweiten Ebene die parallele Region mit nur einem Thread ausgeführt wurde. Bei der Datenreihe *getScores*() wurden entsprechend alle Threads in der zweiten Ebene eingesetzt, während die erste Ebene mit nur einem Thread ausgeführt wurde. Bei der Datenreihe *beste Kombination* wird die beste mögliche Kombination der Verteilung von Threads für eine vorgegebene Anzahl von Threads auf die beiden Ebenen dargestellt.

Besonders auf dem Sun Fire v40z System liegen die Ergebnisse nahe beieinander und entsprechen weitgehend dem idealen (linearen) Speedup. Dies ist dadurch zu begründen, dass auf beiden Parallelisierungsebenen über 99% der Laufzeit des Programms parallel sind. Weiter befindet sich in der höheren parallelen Region kein und

in der tieferen parallelen Region nur ein Synchronisationspunkt. Da sowohl die Anzahl der Anfragebilder die Anzahl der verwendeten Threads teilt und die Anzahl der Bilder in der Datenbank um ein vielfaches größer ist als die Anzahl der Threads, ist auch die Lastverteilung optimal.

Der erreichte Speedup mit 8 Threads auf dem Sun Fire V40z System ist 7,91 für die beste Kombination, wobei dort auf der höheren Ebene 4 und auf der tieferen Ebene 2 Threads eingesetzt werden. Dies entspricht einer Effizienz von knapp 99%. Die *batch*()- Ebene erreicht einen Speedup von 7,75 und die *getScores*()-Ebene erreicht einen Speedup von 7,71, was beides einer Effizenz von 96% entspricht. Der Einsatz von geschachtelter Parallelisierung (*Nested Parallelisierung*) bringt hier zwar nur geringe, aber dennoch messbare Vorteile. Allerdings liegt der Speedup mit 7,71 bzw. 7,75 bei Verwendung aller Threads in einer Ebene bereits sehr nahe am Optimum, so dass kein Raum für Verbesserungen bleibt.

Etwas anders ist das Bild auf dem Sun Fire E6900 System. Auch hier liegen die Ergebnisse recht dicht beeinander, insgesamt ist aber ein größerer Unterschied zwischen den Versionen zu beobachten. Die beste Kombination erreicht mit 4 Threads auf der höheren und 8 Threads auf der tieferen Parallelisierungsebene einen Speedup von 28,45, was einer Effizienz von 89% entspricht. Während auf dem Opteron System die *batch*()-Ebene etwas schneller als die *getScores*()-Ebene ist, erreicht sie hier nur einen Speedup von 23,69 mit 32 Threads, was nur noch einer Effizienz von 74% entspricht. Die *getScores*()-Ebene erreicht mit 32 Threads einen Speedup von 25,97. Hier kann die geschachtelte Parallelisierung die Performance verbessern, indem der Speedup um ca. 2,5 bei 32 Threads angehoben wird.

Insgesamt ist die Parallelisierung von FIRE mit OpenMP sehr erfolgreich. In beiden Betriebsmodi wird ein deutlicher Vorteil beim Einsatz von Mehrprozessorsystemen erzielt. Der Einsatz geschachtelter Parallelisierung bringt auf beiden betrachteten Plattformen einen Performancegewinn. Durch die strukturierte Programmierung konnte die Parallelisierung gezielt an zwei Punkten durchgeführt werden, ohne dass größere Teile des Programms betroffen sind. Da der Einsatz von Parallelisierung mit OpenMP sehr flexibel ist und im Vergleich zur bisher durchgeführten Nutzung von Mehrprozessorsystemen die manuelle Zusammenfassung der Ergebnisse entfällt, ist die Parallelisierung bereits in den Hauptzweig der Entwicklung übernommen worden.

## 3.3 ADI

Der Poisson-Solver ADI wird am NeuroInformatics Center der Universität von Oregon von Adnan Salman entwickelt. Im Rahmen einer Kooperation mit weiteren Instituten wird die Leitfähigkeit des menschlichen Kopfes modelliert und simuliert. Ziel ist dabei die Abschätzung der elektrischen Leitfähigkeit eines Kopfes, dessen Modell aus einem computer-tomographischen (CT) Scan stammt.

Mit den Untersuchungen soll das Finden von Annomalien (z.B. Brüche oder Löcher) und das Lokalisieren der Quelle des elektrischen Potentials, das auf der Kopf-

Tabelle 3: Testfälle des STREAM-Benchmarks.

| Name | Kernel | Bytes/Iter. | FLOPS/Iter. |
|---|---|---|---|
| COPY | $a[i] = b[i]$ | 16 | 0 |
| SCALE | $a[i] = q \cdot b[i]$ | 16 | 1 |
| SUM | $a[i] = b[i] + c[i]$ | 24 | 1 |
| TRIAD | $a[i] = b[i] + q \cdot c[i]$ | 24 | 2 |

haut gemessen werden kann, verbessert werden. Bei den bisherigen Verfahren wird nicht berücksichtigt, dass die gemessenen elektrischen Signale durch die Hautschichten am Kopf verfälscht werden. Die Beeinflussung durch die Haut soll durch Simulation bekannt gemacht werden, so dass medizinische Behandlungen verbessert werden können[3].

Ausgehend von dem geometrischen Modell aus dem CT Scan werden die Grenzen der verschiedenen Hauttypen gesucht. Jedem Hauttyp wird dann ein durchschnittlicher Wert für die Leitfähigkeit zugewiesen. Damit wird dann durch Angabe der elektrischen Quellen im Gehirn das Potential an jedem Punkt im Kopf berechnet.

Das sich ergebende Poisson-Problem wird mit einem *Alternating Directions (ADI)* Algorithmus gelöst, wobei jeder Zeitschritt in drei Teilschritte (X-, Y- und Z-Richtung) zerlegt wird, in dem ein eindimensionales tridiagonales System gelöst wird. Die Berechnung zur Lösung des Gleichungssystems in jedem Teilschritt ist unabhängig von den Berechnungen in den anderen Teilschritten, so dass z.B. bei der X-Richtung diese über die Y- oder Z-Dimension parallelisiert werden kann.

Diese Parallelisierung dieses C++ Programms wurde in OpenMP durchgeführt und in [Salman & Turovets$^+$ 05] vorgestellt. Auf der NUMA-Architektur der Sun Fire V40z Systeme blieb die Skalierung aber hinter den Erwartungen zurück. Messungen unter Solaris haben gezeigt, dass keine Verteilung der Daten erfolgt, da das Gitter im seriellen Teil vom Master-Thread angelegt und initialisiert wird. Die in dieser Arbeit vorgestellten Methoden zur Beeinflussung der Datenplatzierung werden auch auf ADI erfolgreich angewendet. Dies wird in Abschnitt 5.2 beschrieben.

## 3.4 STREAM-Benchmark

Der STREAM-Benchmark [McCalpin 05] wurde von John McCalpin an der Universität von Delaware vorgestellt und ist mittlerweile ein weit verbreitetes Instrument zur Messung der Speicherbandbreite von Computersystemen. Hierbei wird die tatsächlich von Programmen erreichbare Bandbreite ermittelt, die durchaus von der theoretischen Bandbreite (*Peak*), die von den Hardwareherstellern angegeben wird, abweichen kann.

Dabei werden vier Testfälle durchgeführt, die in der Tabelle 3 dargestellt sind. Jeder Testfall wird mehrmals ausgeführt und das beste erreichte Ergebnis wird verwen-

[3] `http://duckhenge.uoregon.edu/hparchive/display.php?q=20.6.05-CerebralData.html`

**Programmausschnitt 5** Testfall des MAP-Benchmarks.

```
1 #pragma omp parallel
2 {
3   std::map<int, double> my_map;
4   int ognt = omp_get_num_threads();
5   int ogtn = omp_get_thread_num();
6
7   for (int i = 0 + ogtn; i < 999000; i += ognt)
8   {
9     my_map[i] = (double)i;
10  }
11 #pragma omp master
12 {
13    dRet = my_map[23];
14 }
15 } // end omp parallel
```

det. Die Angabe erfolgt dabei in der Einheit MB/sec. Da beim STREAM-Benchmark immer die gleichen Ansätze verwendet werden, sind die Ergebnisse von zwei unterschiedlichen Plattformen direkt vergleichbar.

Während die erste veröffentlichte Version des STREAM-Benchmarks nur für die Programmiersprache FORTRAN verfügbar war, steht nun auch eine Version in der Programmiersprache C zum Download zur Verfügung, die als Basis für die hier benutzten Implementierungen verwendet wurde. In der offiziellen Version werden die Datenarrays *a*, *b* und *c* als statische Felder angelegt. Da damit viele Experimente mit der Datenverteilung nicht möglich wären, wurde der Code so verändert, dass die drei Arrays dynamisch mittels $malloc()$ und $free()$ angelegt und freigegeben werden. Diese Version wird im restlichen Teil dieser Arbeit als „C “ Version bezeichnet. Ein Unterschied der „C “ Version zur offiziellen Implementierung war nicht messbar.

Die Version, die im Folgenden als „C++ “ Version bezeichnet wird, verwendet zur Darstellung der drei Arrays den Datentyp $std::valarray<double>$. Ein Unterschied in der seriellen Performance war bei den verwendeten Compilern ebenfalls vernachlässigbar.

Der STREAM-Benchmark wird in Abschnitt 4.4 und 4.6 zum Vergleich der Arbeitsweise der drei untersuchten Parallelisierungstechniken sowie in Abschnitt 5.2 zur Untersuchung der Verbesserungsansätze für NUMA-Architekturen eingesetzt. Der Übersicht halber werden hier nur die Ergebnisse für den SUMMING-Testfall betrachtet, die Aussagen gelten aber auch für die anderen Testfälle.

## 3.5 MAP-Benchmark

Der MAP-Benchmark ist kein offizieller Benchmark, sondern wurde während der Arbeit mit dem DROPS Paket zur Untersuchung der auftretenden Effekte bei den Setup-Routinen entwickelt. Er stellt dabei den Einsatz von *std*::*map* in der Parallelisierung dieser Routinen nach. Dies ist im Programmausschnitt 5 dargestellt.

Der gezeigte Programmkern wird dabei mehrere Male durchlaufen. Wie in den Setup-Routinen hat jeder eine eigene Instanz von *std*::*map*, in die er eine große Anzahl von Elementen einfügt. Das Einfügen geschieht dabei mit einer fixen Schrittweite im Index, wie in Zeile 7 und Zeile 9 zu sehen ist.

Zwar ist das Einfügen in die privaten Maps unabhängig voneinander, aber für jedes Element muss Speicher allokiert werden. Die benötigte Speichermenge ist dabei sehr gering, jeweils muss nur Speicher für eine *int* Variable und eine *double* Variable angefordert werden, zusätzlich dazu noch Platz in den Verwaltungsstrukturen der Map.

Der MAP-Benchmark wird in Abschnitt 5.3 zur Untersuchung der Verbesserungsvorschläge für die Setup-Routinen eingesetzt.

# 4 Vergleich von OpenMP, Posix-Threads und UPC

Besonders bei C++ Programmen ist der Einsatz der Posix-Threads oder die Anwendung anderer Parallelisierungsbibliotheken (noch) deutlich häufiger anzutreffen als OpenMP. Es lässt sich beobachten, dass beinahe jedes Parallelisierungsparadigma gut anzuwenden ist, wenn dies bereits bei der Codeentwicklung berücksichtigt wurde. Meistens ist es deutlich aufwändiger einen Code zu parallelisieren, wenn dieser rein seriell entworfen wurde.

In diesem Kapitel werden nach einer Vorstellung der drei betrachteten Parallelisierungstechniken diese hinsichtlich der Vorgehensweise bei der Parallelisierung (Programmierung der Arbeitsverteilung), der Unterstützung des Entwicklers durch Software Tools während der Parallelisierung und ihrer erreichbaren Performance verglichen. Der Vergleich erfolgt hauptsächlich anhand der extrahierten Programmkerne von DROPS sowie des STREAM-Benchmarks.

Auch mit MPI, welches vom Ansatz her für Systeme mit Distributed-Memory ausgelegt ist, kann auf Systemen mit Shared-Memory gearbeitet werden. Darüber hinaus bietet MPI-2 erweiterte Möglichkeiten zur Parallelisierung von Remote Shared-Memory Zugriffen. Dennoch ist in vielen Fällen die Vorgehensweise bei der Parallelisierung deutlich anders und somit soll MPI wegen der insgesamt anderen Ausrichtung auf Distributed-Memory Systeme in dieser Arbeit nicht betrachtet werden.

## 4.1 OpenMP

In diesem Abschnitt wird die direktivengesteuerte Parallelisierung mit OpenMP vorgestellt.

### 4.1.1 Übersicht

OpenMP hat sich in den letzten Jahren zum Standard für die direktivengesteuerte Parallelisierung von Shared-Memory Systemen entwickelt. Durch diesen Ansatz ist OpenMP generell von der Unterstützung in einem Compiler abhängig, die Anzahl der OpenMP-fähigen Compiler ist in den letzten Jahren aber stetig gewachsen. OpenMP bietet eine Spezifikation für die Programmiersprachen FORTRAN, C und C++. Die erste Version der Spezifikation wurde im Jahr 1997 für FORTRAN und im Jahr 1998 für C und C++ vorgestellt. Im Mai 2005 wurde die aktuelle Spezifikation in Version 2.5 vorgestellt, die nun alle drei Programmiersprachen in einem Dokument behandelt.

Parallele Programme in OpenMP setzen mehrere Threads ein. Dabei beginnt jedes Programm seine Ausführung mit nur einem Thread, dem sog. *Master-Thread*. Die Parallelisierung erfolgt explizit über vom Benutzer in den Code geschriebene Compileranweisungen (Direktiven), in C und C++ als Pragmas. In OpenMP wird das sog. *Fork-Join* Modell benutzt. Dabei wird beim *Fork* an einer parallelen Region ein Team von Threads erzeugt und beim *Join* am Ende einer parallelen Region werden die Threads, nach Beendigung ihrer Arbeit, synchronisiert und beendet (oder schlafen gelegt), so

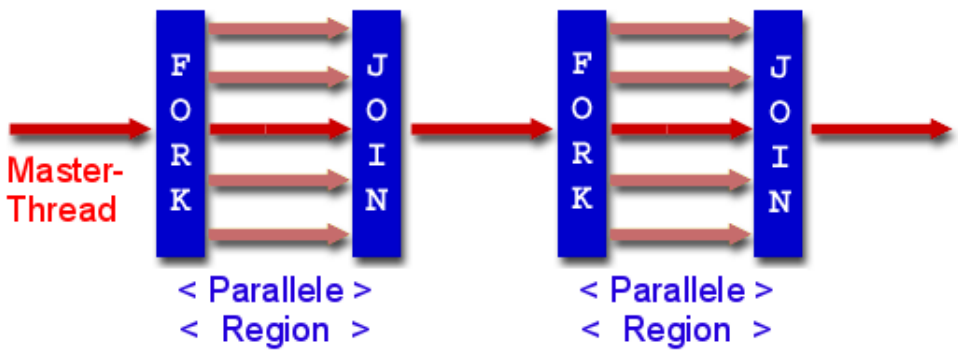


Abbildung 9: OpenMP: Ausführungsmodell.

dass die Programmausführung nur noch vom Master-Thread fortgeführt wird. Dieses Konzept ist in Abbildung 9 dargestellt.

Als *statische Abgrenzung* (*lexical extent*) wird der Text (Code) bezeichnet, der sich innerhalb eines auf eine Direktive folgenden strukturierten Blocks befindet. Dieser Wirkungsbereich einer Direktive darf nicht die Grenzen einer Funktion oder einer Quellcodedatei verlassen. Die *dynamische Abgrenzung* besteht aus dem Code der statischen Abgrenzung und den gegebenenfalls aufgerufenen Routinen, die auch in anderen Quellcodedateien liegen dürfen. Sie ergibt sich damit erst zur Laufzeit, da sie vom Programmfluss abhängig ist, und entspricht dem parallel ausgeführten Code. Hierin ist es erlaubt, dass Worksharingkonstrukte ohne direkte Bindung zur umschließenden Direktive erscheinen, diese heißen dann *orphaned* bzw. verweist.

Der Programmierer muss im Allgemeinen nur angeben, welche Arbeit verteilbar ist und wie das Datenzugriffsverhalten aussieht, wobei die Art der Arbeitsverteilung beeinflusst werden kann. OpenMP verfolgt damit einen High-Level Ansatz.

### 4.1.2 Parallelisierungsanweisungen

Die Parallelität eines Programms befindet sich in OpenMP innerhalb der parallelen Regionen. Eine parallele Region ist ein Codeblock, der von mehreren Threads parallel ausgeführt wird. Die Anzahl der Threads kann dabei dynamisch festgelegt werden, sofern die Implementierung dies nicht verbietet. Ebenfalls ist es bei entsprechender Implementierung möglich, dass parallele Regionen geschachtelt sind, dies wird *Nesting* genannt.

Die Verteilung der Arbeit in einer parallelen Region auf die Threads wird durch *Worksharingkonstrukte* festgelegt. Dabei wird die Ausführung des zugehörigen Codeblocks nach der vorgegebenen Art unter den Threads des Teams aufgeteilt. Neue Threads werden nicht erstellt. Beim Eintritt in ein Worksharingkonstrukt findet keine Synchronisation statt, dafür aber eine implizite Synchronisation beim Austritt, wobei diese weggelassen werden kann. In der aktuellen Spezifikation sieht OpenMP drei Arten der Arbeitsverteilung vor:

1. *for*-Worksharing: hierbei werden die Iterationen einer Schleife unter den Threads des Teams aufgeteilt (Datenparallelisierung). Es können nur *for*-Schleifen (keine *while*-Schleifen) parallelisiert werden, die von kanonischer Form sind und deren Iterationsanzahl sich vor Schleifenbeginn berechnen lässt (siehe auch 5.5). Die Art der Verteilung kann über die Angabe des Scheduling festgelegt werden. Hier liegt eine der Stärken von OpenMP, da durch die Angabe eines Parameters viel Einfluss auf die Parallelisierung genommen werden kann. So lassen sich z.B. in vielen Fällen Lastungleichheiten durch ein anderes Scheduling beheben. In den anderen beiden hier vorgestellten Parallelisierungstechniken und auch in MPI muss dies explizit programmiert werden. Allerdings muss das Scheduling fest vorgegeben sein und kann zur Laufzeit nicht vom Programm beeinflusst werden. Folgende Arten der Arbeitsverteilung sind in der aktuellen Spezifikation definiert:

   - *STATIC*: die gesamte Anzahl der Iterationen wird in Stücke einer optional anzugebenden Größe aufgeteilt und statisch den Threads zugewiesen. Falls die Größe nicht angegeben wird, werden die Iterationen, soweit möglich, gleichmäßig auf die Threads verteilt.
   - *DYNAMIC*: die gesamte Anzahl der Iterationen wird in Stücke einer optional anzugebenden Größe aufgeteilt und dynamisch den Threads zugewiesen. Das heißt, dass wenn ein Thread einen Block abgearbeitet hat, er den nächsten zu bearbeitenden Block zugewiesen bekommt.
   - *GUIDED*: die Größe der zu verteilenden Blöcke nimmt exponentiell ab. Dabei ist es möglich, eine Mindestgröße der Blöcke anzugeben.
   - *RUNTIME*: die Art der Arbeitsverteilung wird zur Laufzeit des Programms aus der Variable *OMP_SCHEDULE* gelesen.

2. *sections*-Worksharing: hierbei wird die Arbeit in anzugebende, abgeschlossene Bereiche aufgeteilt, die jeweils von einem Thread unabhängig von den anderen Bereichen bearbeitet werden (Aufgabenparallelisierung). Innerhalb des *sections*-Konstrukt werden die Bereich durch die *section*-Direktive angegeben und abgegrenzt. Jeder Bereich wird genau einmal durchlaufen, allerdings legt die Implementierung die Zuordnung von Threads zu Bereichen fest, wobei eine gleiche Zuordnung bei mehrmaligem Durchlaufen des Worksharingkonstruktes nicht gewährleistet ist. Es ist nicht erlaubt, innerhalb eines Bereiches zu verzweigen, oder aus ihm heraus.

3. *single*-Worksharing: hierbei wird der Codebereich serialisiert, d.h. von nur einem Thread ausgeführt. Die Implementierung entscheidet, welcher Thread im Team den Codebereich durchläuft, und diese Festlegung kann sich bei mehrmaligem Durchlaufen des Worksharingkonstruktes ändern. Die anderen Threads warten am Ende des Bereichs an einer impliziten Barriere.

In der OpenMP Spezifikation werden mehrere Bedingungen gestellt, damit ein Worksharingkonstrukt korrekt abgearbeitet wird. Viele Bedingungen können dabei vom Compiler während der Compilezeit festgestellt werden, einige aber nicht, da ihre Erfüllung erst zur Laufzeit überprüft werden kann, da sie z.B. an die dynamische Abgrenzung oder aber an den Programmfluss gebunden sind. Dazu zählen beispielsweise die Bedingung, dass ein Worksharingkonstrukt sich innerhalb der dynamischen Abgrenzung einer parallelen Region (bzw. der enstprechenden Direktive) befinden muss, oder dass ein Worksharingkonstrukt von entweder allen Threads eines Teams oder keinem Thread des Teams getroffen werden muss.

Neben den oben vorgestellten Worksharingkonstrukten gibt es noch fünf weitere Direktiven in OpenMP, die für die Synchronisation von Threads sowie die Sicherstellung des korrekten Datenaustausches notwendig sind:

1. *master*-Direktive: hierbei wird ein Codebereich angegeben, der ausschließlich vom Master-Thread ausgeführt werden soll. Die anderen Threads überspringen diesen Bereich, an dessen Ende sich keine implizite Barriere befindet.

2. *critical*-Direktive: hierbei wird ein Codebereich angegeben, der von nur einem Thread gleichzeitig durchlaufen werden kann. Threads, die auf die kritische Region stoßen, warten mit dem Eintritt, falls sich bereits ein anderer Thread innerhalb der kritischen Region befindet. Durch die Angabe eines Namens für eine kritische Region können mehrere Regionen innerhalb eines Programms unterschieden werden.

3. *barrier*-Direktive: an einer Barriere werden alle Threads eines Teams synchronisiert. Das bedeutet, dass wenn ein Thread eine Barriere erreicht, er solange dort wartet, bis alle Threads die Barriere erreicht haben.

4. *flush*-Direktive: nach dieser Direktive wird eine konsistente Darstellung des Speichers garantiert, d.h. das entsprechende Werte in den Hauptspeicher übertragen wurden. Es können auch einzelne Variablen angegeben werden. Ein Flush wird am Ende eines Worksharingkonstruktes implizit ausgeführt.

5. *atomic*-Direktive: hierbei wird angegeben, dass ein spezieller Speicherbereich ein atomares Update erfährt, also dass nicht mehrere Threads gleichzeitig dorthin schreiben dürfen.

Während bei Verwendung der Worksharingkonstrukte eine *Deadlocksituation* ausgeschlossen ist, kann diese bei Verwendung der oben aufgelisteten fünf Direktiven sowie bei Verwendung entsprechender Laufzeitfunktionen bei fehlerhafter Programmierung auftreten.

Neben der Verteilung der Arbeit muss vom Programmierer bei OpenMP noch die Einteilung in *shared* und *private* für die Variablen getroffen werden. Da OpenMP das Shared-Memory Modell benutzt, sind alle globalen Variablen und alle Variablen ohne

Tabelle 4: OpenMP: Auszug der Laufzeitfunktionen und Umgebungsvariablen.

| | |
|---|---|
| *omp_get_num_threads*() | Gibt die Anzahl der Threads im Team der aktuellen parallelen Region zurück. |
| *omp_set_num_threads*(...) | Setzt die Anzahl der Threads, die in der nächsten parallelen Region verwendet werden sollen. |
| *omp_get_thread_num*() | Gibt die Thread-Nummer des aufrufenden Threads im aktuellen Team zurückt, der Master-Thread ist 0. |
| *omp_get_max_threads*() | Gibt den größten Wert zurück, der von *omp_get_num_threads*() geliefert werden kann. |
| *omp_get_num_procs*() | Gibt die Anzahl der Prozessoren zurück, die dem Programm zur Verfügung stehen. |
| *OMP_NUM_THREADS* | Umgebungsvariable; legt die maximale Anzahl der Threads fest, die vom Programm verwendet werden sollen. |

weitere Angabe *shared*, außer es handelt sich um lokale nicht-statische Stackvariablen in Routinen, die aus einer parallelen Region aufgerufen wurden.

Für eine parallele Region sowie beim $for$-Worksharingkonstrukt und auch beim *sections*-Worksharingkonstrukt können Variablen für die entprechenden Direktiven als *shared* oder *private* angegeben werden. Ist eine Variable als *private* deklariert, erhält jeder Thread eine uninitialisierte Instanz dieser Variablen. Ist eine Variable *shared*, greifen alle Threads auf die ursprüngliche Instanz dieser Variablen zu. Um globale Daten zu privatisieren, sind sie als *threadprivate* zu deklarieren. Die Verwendung von *threadprivate* in OpenMP entspricht dem Konzept der TSD Daten bei den Posix-Threads.

Für das $for$-Worksharingkonstrukt sowie das *sections*-Worksharingkonstrukt können mittels der *firstprivate* Klausel private Variablen mit dem Wert der Variablen vor Beginn der parallelen Region initialisiert werden. Mit der *lastprivate* Klausel können Variablen nach Ende der parallelen Region den Wert erhalten, den sie in der letzten Schleifeniteration bzw. letzten Sektion hatten. Variablen, die *threadprivate* deklariert wurden, können mit der *copyin* Klausel zu Beginn einer parallelen Region initialisiert werden.

### 4.1.3 Laufzeitfunktionen

Neben den bisher vorgestellen Direktiven zur Steuerung der Parallelisierung und Einteilung der Variablen schreibt die OpenMP Spezifikation auch eine Menge von Laufzeitfunktionen und Umgebungsvariablen vor, die eine konforme Implementierung anbieten bzw. berücksichtigen muss. Die Tabelle 4 zeigt die wichtigsten und am häufigsten verwendeten Funkionen und Variablen. Für die in dieser Diplomarbeit betrachteten und durchgeführten Implementierungen wurde nur eine Teilmenge daraus verwendet.

Tabelle 5: Posix-Threads: grundlegende Funktionalität.

| | |
|---|---|
| *pthread_create*(...) | Erzeugt einen neuen Thread, der seine Arbeit im Code einer Routine beginnt, welche als Argument übergeben wird. Ebenfalls wird ein Pointer als Argument an diese Routine übergeben, sowie weitere Verwaltungsinformationen. |
| *pthread_exit*(...) | Terminiert die Ausführung des aufrufenden Threads. Alle registrierten Aufräumfunktionen werden ausgeführt, ebenso werden alle Thread-privaten Daten freigegeben. |
| *pthread_join*(...) | Der aufrufende Thread wird solange schlafen gelegt, bis der als Argument angegebene Thread beendet ist (z.B. durch Aufruf von pthread_exit()). Der Thread muss *joinable* sein. |
| *pthread_detach*(...) | Setzt den aufrufenden Thread in den *detached* Zustand. In diesem Zustand kann er nicht mehr von anderen Threads ge-joint werden. Dieser Zustand kann auch direkt bei der Erzeugung angefordert werden. |
| *pthread_cancel*(...) | Sendet ein Signal zur sofortigen Terminierung an einen Thread. Der zu terminierende Thread kann sich aber in einem Zustand befinden, in dem ein Abbruch nicht möglich ist. Dies kann mittels *pthread_testcancel*() abgefragt werden. |

## 4.2 Posix-Threads

In diesem Abschnitt wird die Posix-Threads Bibliothek (pthreads) zur Programmierung von Shared-Memory Systemen vorgestellt.

### 4.2.1 Übersicht

Die Posix-Threads stellen POSIX-konforme Funktionalität zur Erzeugung und Verwaltung von Threads zur Verfügung. Sie sind für fast alle UNIX-Systeme, aber auch für Windows und andere Betriebssysteme verfügbar. Als Bibliothek sind sie generell unabhängig von der gewählten Programmiersprache. Zwar stellen sie keine objektorientierte Benutzerschnittstelle zur Verfügung, sie können aber ohne Einschränkungen mit C++ verwendet werden.

### 4.2.2 Laufzeitfunktionen

Die grundlegenste Funktionalität, welche die Posix-Threads bieten, sind Befehle zum Erzeugen, Zusammenführen und Beenden von Threads. Die Tabelle in 5 listet die wichtigsten Funktionen aus diesem Bereich auf und gibt eine kurze Beschreibung ihrer Funktionalität.

Ein *Mutex* (Mutual Exclusion), ermöglicht den gegenseitigen Ausschluss aus Programmsegmenten, also die Erstellung einer *Kritischen Region*. Ein Mutex kennt zwei

Tabelle 6: Posix-Threads: Arbeit mit Mutexen.

| | |
|---|---|
| *pthread_mutex_init*(...) | Initialisiert ein Mutex-Objekt und setzt, falls vorhanden, die Attribute. Der Zustand nach der Initialisierung ist *unlocked*. |
| *pthread_mutex_lock*(...) | Lockt ein Mutex für den aufrufenden Thread, falls es nocht nicht gelockt ist; andernfalls blockiert der aufrufende Thread solange, bis das Mutex frei wird und lockt es dann. |
| *pthread_mutex_trylock*(...) | Wie *pthread_mutex_lock*(), blockiert aber nicht, falls das Mutex schon gelockt ist, sondern gibt dann einen Fehlercode zurück. |
| *pthread_mutex_timedlock*(...) | Wie *pthread_mutex_trylock*(), wartet aber eine anzugebende Zeit auf das Freiwerden eines eventuell bereits gelockten Mutex. |
| *pthread_mutex_unlock*(...) | Gibt ein bereits gelocktes Mutex frei. |
| *pthread_mutex_destroy*(...) | Löscht ein Mutex-Objekt. |

Zustände: zum einen *unlocked* und zum anderen *locked* (der Lock gehört zu einem Thread). Ein Mutex kann nie von mehr als einem Thread gleichzeitig gelocked sein.

Ein Mutex wird in den meisten Fällen zum Schutz vor Inkonsistenz beim Update von globalen Daten eingesetzt. Die Posix-Threads bieten Funktionen zur Arbeit mit Mutexes wie in Tabelle 6 aufgelistet sowie weitere Verwaltungsfunktionalität an.

Eine weitere wichtige Funktionalität, welche die Posix-Threads zur Programmierung von Shared-Memory Systemen anbieten, sind die sog. *Condition* Variablen. Mit Hilfe dieser Variablen können ein oder auch mehrere Threads schlafen gelegt werden, bis ein Zustandswechsel in der Variable erfolgt. Der Zustandswechsel wird durch ein Signal herbeigeführt.

Eine Condition Variable muss immer im Zusammenspiel mit einem Mutex verwendet werden um eine *Race-Condition* zu vermeiden, falls ein Thread das Warten auf eine Variable vorbereitet während ein zweiter Thread der Variable ein Signal schickt, genau bevor sich der erste Thread schlafen legt. Die Condition Variablen können zum Beispiel verwendet werden um Barrieren zu implementieren. Die wichtigsten Funktionen zur Arbeit mit Condition Variablen aus der pthreads Bibliothek sind in der Tabelle 7 aufgelistet und kurz beschrieben.

Der Standard *1003.1j-2000* der Posix-Threads bietet eine direkt zu verwendende Implementierung von Barrieren, wie in der Tabelle in Abbildung 8 aufgelistet ist.

In einem Programm kann es notwendig sein, dass globale oder statische Daten für jeden Thread privat sein müssen, d.h. dass jeder Thread einen anderen Wert dieser Daten hat. Diese Daten sollen weiter für die gesamte Lebensdauer eines Threads erhalten bleiben. Die Funktionen der Posix-Threads hierfür sind in der Tabelle 9 aufgelistet. Jeder Thread besitzt einen privaten Speicherbereich, den TSD (thread-specific data) Bereich, in welchem mit den dargestellten Funktionen Schlüssel abgelegt werden kön-

Tabelle 7: Posix-Threads: Arbeit mit *Condition* Variablen.

| | |
|---|---|
| *pthread_cond_init*(...) | Initialisiert eine Condition Variable und setzt, falls vorhanden, die Attribute. |
| *pthread_cond_signal*(...) | Startet einen der Threads, die auf die Variable warten. Falls mehrere Threads warten, wird genau einer gestartet, welcher ist nicht definiert. Falls kein Thread wartet, passiert nichts. |
| *pthread_cond_broadcast*(...) | Startet alle Threads, die auf die Variable warten. Falls kein Thread wartet, passiert nichts. |
| *pthread_cond_destroy*(...) | Löscht eine Condition Variable. |

Tabelle 8: Posix-Threads: Verwendung von Barrieren.

| | |
|---|---|
| *pthread_barrier_init*(...) | Initialisiert eine Barriere und setzt, falls vorhanden, die Attribute. |
| *pthread_barrier_wait*(...) | Wartet an einer Barriere, bis alle geforderten Threads diese Routine aufgerufen haben. |
| *pthread_barrier_destroy*(...) | Löscht eine Barriere. Eine mehrfache Benutzung einer Barriere ohne neuen Aufruf von *pthread_barrier_init*() ist undefiniert. |

nen, über die auf die Daten zugegriffen wird. Die Größe der Daten ist nur durch den zur Verfügung stehenden Speicher begrenzt, allerdings ist die Anzahl der gleichzeitig verwendeten Schlüssel durch die Implementierung begrenzt.

Weitere Funktionen, die in größeren Programmen oft benötigt werden, sind in der Tabelle 10 aufgeführt. Die folgende Tabelle und auch die obigen Tabellen beschreiben nicht den gesamten Funktionsumfang der Posix-Threads nach dem aktuellen Standard, sondern stellen nur einen Auszug der wichtigsten Routinen dar.

Tabelle 9: Posix-Threads: Arbeit mit *thread-privaten* Daten.

| | |
|---|---|
| *pthread_key_create*(...) | Erzeugt einen neuen Schlüssel im TSD Bereich, optional kann eine Funktion zum Abräumen der Daten angegeben werden. |
| *pthread_setspecific*(...) | Setzt die hinter einem Schlüssel liegenden Daten. |
| *pthread_getspecific*(...) | Liest die hinter einem Schlüssel liegenden Daten. |
| *pthread_key_delete*(...) | Gibt einen Schlüssel wieder frei; falls bei der Initialisierung angegeben, wird die Funktion zum Abräumen der Daten aufgerufen. |

Tabelle 10: Posix-Threads: weitergehende Funktionalität.

| | |
|---|---|
| *pthread_attr_setstacksize*(...) | Setzt die minimal benötigte Stackgröße des zu erstellenden Threads. |
| *pthread_once*(...) | Die als Argument angegebene Funktion wird höchstens einmal aufgerufen (ausgeführt). |

## 4.3 UPC

In diesem Abschnitt wird die Parallelisierung mit UPC (Unified Parallel C) vorgestellt. UPC erweitert den ISO C99 Standard um ein paralleles Ausführungsmodell mit einem *shared* Adressraum sowie Primitiven zur Synchronisation und Speicherverwaltung. Dabei basiert es auf den Ansätzen *AC*, *Split-C* und *Parallel C Preprocessor (PCP)*, welche ebenfalls parallele Erweiterungen des C99 Standards sind.

Die erste UPC-Spezifikation V1.0 wurde im Februar 2001 veröffentlicht. Die aktuelle UPC-Spezifikation ist V1.2 und wurde im Mai 2003 vorgestellt. Eine Einführung in UPC ist in [Chauvin & Saha$^+$ 04] zu finden, detaillierte Informationen sind in der UPC-Spezifikation des UPC Konsortiums zu finden.

### 4.3.1 Übersicht

UPC verfolgt einen anderen Ansatz als die Posix-Threads und OpenMP bezüglich der Darstellung des Speichers. Ebenso wie bei den beiden bereits vorgestellten Konzepten wird in einem globalen Adressraum gearbeitet, dieser ist aber partitioniert. Man spricht hier vom Distributed-Shared-Memory Modell. Die Arbeitsverteilung erfolgt ähnlich wie bei OpenMP durch Worksharing-Konstrukte, bei der Deklaration von shared Speicher kann aber deutlich weitergehend Einfluss auf die Datenverteilung (d.h. Lokalität zu den Threads) genommen werden.

UPC bietet in seiner bisherigen Form nur Unterstützung der Programmiersprache C, es ist aber möglich UPC-Code in C++ Programme zu integrieren. Da die Verbindung von zwei oder mehreren Programmiersprachen immer von den verwendeten Compilern abhängt, ist dies auch hier der Fall. Die folgende Beschreibung gilt für den Berkeley UPC Compiler in Version 2.x.

Damit aus C++ auf Variablen und Funktionen aus UPC und andersherum zugegriffen werden kann, müssen diese zunächst als **extern** "C" deklariert sein. Falls die *main*()-Routine nicht im UPC-Code liegt, muss das Laufzeitsystem durch einen Aufruf von *bupc_init*() oder *bupc_init_reentrant*() initialisiert werden, wobei die zweite Routine bei Verwendung von Posix-Threads als Kommunikationssystem verwendet werden muss. Durch den Aufruf von *bupc_exit*() wird das Laufzeitsystem terminiert. Dabei sollten die Routinen als erste und letzte Anweisungen im Programm stehen. Die UPC Quellcodedateien werden mit dem UPC Compiler und die C++ Quellcodedateien mit dem C++ Compiler kompiliert. Der Linkeraufruf muss mit dem UPC Compiler gestartet werden, dem der C++ Linker über den Parameter $-link-with=$ mitgeteilt wird

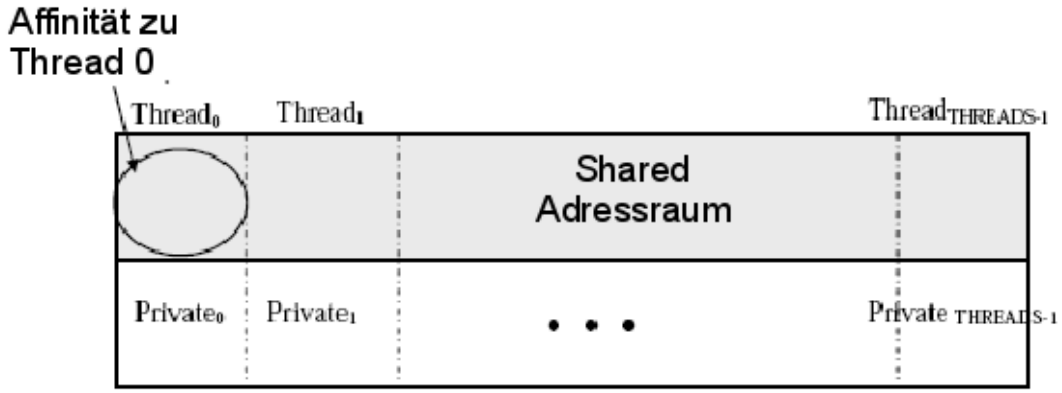


Abbildung 10: UPC: Distributed-Shared-Memory Modell.

und damit den letzten Linkschritt durchführt.

Dies impliziert, dass einige C++ Codes nicht direkt mit UPC parallelisiert werden können, z.B. eine Member-Funktion einer Klasse. Als Lösung muss der Code so umgeschrieben werden, dass die Berechnung in eine „C “ Funktion ausgelagert wird und diese dann parallelisiert wird. Dabei wird dann gegebenenfalls das Klassenkonzept gebrochen. Als Beispiel für eine solche Umstrukturierung kann die Parallelisierung des DROPS Programmkerne betrachtet werden, die im Anhang beschrieben ist.

Bei der Verwendung von Posix-Threads als Kommunikationssystem ist dazu noch eine Besonderheit zu beachten, da dadurch Unterschiede zwischen globalen C++ und UPC Variablen entstehen. Da jeder Thread eine eigene Instanz der globalen UPC Variablen erhält, kann aus dem C++ Code nicht direkt auf globale UPC Variablen zugegriffen werden. Die Privatisierung gilt natürlich nicht für die C++ Variablen, z.B. wäre die Standardbibliothek mit mehreren Instanzen von *cout* nicht benutzbar.

### 4.3.2 Speichermodell

Wie bereits erwähnt, kommt in UPC das sog. Distributed-Shared-Memory Modell zum Einsatz. Dieses Modell ist ähnlich zum Shared-Memory Modell, erweitert dieses aber um Möglichkeiten der Datenlokalität. Der Speicher ist in Partitionen unterteilt, so dass die *i*-te Partition zum *i*-ten Thread affin ist. Die Affinität gibt an, in welchem lokalen *shared* Speicherbereich eines Threads ein Datum liegt. Darüber hinaus hat jeder Thread noch einen lokalen Speicherbereich. Hieraus ergibt sich eine Speicherdarstellung wie in Abbildung 10 dargestellt.

Zur Benutzung dieses Speichermodells wurden die Pointer der Programmiersprache C erweitert. Ein *shared* Pointer kann alle Adressen im shared Speicherbereich referenzieren, während ein *local* Pointer nur Adressen im lokalen (privaten) Speicher sowie im lokalen Teil des *shared* Speichers halten kann. Für beide Arten von Speicher wird sowohl statische als auch dynamische Allokation unterstützt. Die Verteilung des Speichers bei UPC ergibt sich direkt aus der Deklaration eines statischen Datums bzw.

aus der Allokation eines dynamischen Datums.

Der Zugriff auf den privaten Speicherbereich erfolgt mittels der in der Programmiersprache C bereitgestellten Funktionalität zur Speicherverwaltung. Um auf den gemeinsamen Speicherbereich aller Threads zuzugreifen, wurde das Schlüsselwort *shared* in UPC eingeführt, welche die in Programmausschnitt 6 gezeigte Syntax besitzt.

**Programmausschnitt 6** UPC: *shared* Deklaration.

```
1 shared [block_size] type variablename;
```

Eine Deklaration mit einer Layoutangabe *block_size* bedeutet, dass das shared Objekt im gemeinsamen Speicher mit einen Blockgröße *block_size* pro Thread verteilt ist. Falls keine Blockgröße angegeben wird, wird als Standard eins verwendet.

Durch das Hinzufügen von *shared* zum Sprachstandard ergibt sich auch eine Erweiterung der Deklaration von Variablen und Zeigern, was im Programmausschnitt 7 beispielhaft erläutert wird.

**Programmausschnitt 7** UPC: Verwendung von *shared* Zeigern.

```
1 int v1;                          private (lokale) Variable
2 shared int v2;                   shared Variable
3 shared int v3[N];                shared Array
4 shared [N/THREADS] int v4[N];    shared Array mit Verteilung
5 shared int *p1;                  privater Pointer auf shared
6 shared int *shared p2;           shared Pointer auf shared
```

Im Zusammenhang mit *shared* Speicher muss auch die Konsistenz beachtet werden. UPC bietet hierzu zwei unterschiedliche Modi an:

1. *Strict Mode*: eine strikte Konsistenz wird erzwungen, d.h. vor jedem Zugriff auf shared Daten werden diese synchronisiert. Falls hierbei ein Thread auf Daten zugreifen will, die gerade von einem anderen Thread verändert werden, wird er vor dem Zugriff auf einen Synchronisationspunkt warten und erst dann den Zugriff durchführen. Darüber hinaus wird dem Compiler verboten, diverse Optimierungen durchzuführen, so darf hier beispielsweise die Reihenfolge von unabhängigen Zugriffen auf *shared* Speicher nicht verändert werden.

2. *Relaxed Mode*: hier wird den Threads erlaubt, frei auf *shared* Speicher zuzugreifen. Dieser Modus ist Standard und erlaubt dem Compiler viele Optimierungen und liefert im Allgemeinen eine bessere Performance.

Da der erste Modus signifikanten Overhead mit sich bringen kann, sollte er, sofern möglich, vermieden werden. Der Modus kann auf drei Ebenen im Programm ein-

gestellt werden, wobei die tiefste Ebene die jeweils höheren Ebenen in dem von ihr spezifizierten Bereich überschreibt:

- Global für eine Datei: dies wird durch Inkludieren von entweder *upc_strict.h* oder *upc_relaxed.h* entschieden.
- Für einen Codeblock: dies wird durch Angabe eines umgebenden Pragmas der Form *pragma upc* [*strict*|*relaxed*] entschieden.
- Für eine Variable: dies wird bei der Deklaration der Variable durch [*strict*|*relaxed*] *shared var* festgelegt.

Falls MPI als Kommunikationssystem eingesetzt wird, wird ein UPC-Executable mit mehreren Prozessen gestartet. Falls die Posix-Threads als Kommunikationssystem eingesetzt werden, wird ein UPC-Executable mit mehreren Threads gestartet. Eine Kombination der beiden Laufzeitsysteme ist ebenfalls möglich. Da sich diese Diplomarbeit aber auf die Betrachtung von Shared-Memory Systemen beschränkt, wurden nur die Posix-Threads als Kommunikationssystem verwendet, da es auf diesen Systemen naturgemäß eine bessere Performance bietet als der Einsatz von MPI als Kommunikationssystem.

Somit bezeichnet ein Thread im folgenden tatsächlich einen Thread aus Sicht des Betriebssystems, allerdings kann mit einem Thread in UPC bei Verwendung von MPI als Kommunikationssystem ansonsten auch ein Prozess bezeichnet werden.

### 4.3.3 Parallelisierungsanweisungen

Das UPC Schlüsselwort *THREADS* gibt die Anzahl der Threads an, mit welcher das Programm zur Zeit ausgeführt wird. Diese Anzahl kann entweder zur Compilezeit fest vorgegeben werden, oder aber zur Laufzeit als Parameter an das Programm gegeben werden.

Das UPC Schlüsselwort *MYTHREAD* gibt die Nummer des aktuell ausgeführten Threads an. Der erste Thread hat die Nummer 0.

Weiter bietet UPC ein Worksharingelement, nämlich *upc_forall*. Dieses hat die Syntax wie in Programmausschnitt 8 dargestellt. Es ist an die `for`-Schleife in C angelehnt mit dem Zusatz eines vierten Parameters der festlegt, welcher Thread die aktuelle Schleifeniteration ausführen soll.

**Programmausschnitt 8** UPC: *for_all* Worksharingkonstrukt.

```
1 upc_forall(expression; expression; expression; affinity)
```

Eine Bedingung des *upc_forall* Konstrukts ist, dass alle Iterationen unabhängig voneinander sein müssen. Der vierte Parameter, der auch Affinitätsfeld genannt wird, kann entweder einen Integer erhalten welcher zu (integer % *THREADS*) übersetzt wird, oder eine Adresse, welche den Thread festlegt, der dazu affin ist.

Tabelle 11: UPC: Auszug aus der Laufzeitbibliothek.

| | |
|---|---|
| $upc\_all\_alloc(...)$ | Kollektive Allokation von *shared* Speicher, Blockgröße zur Festlegung der Affinität kann angegeben werden. |
| $upc\_global\_alloc(...)$ | Allokation von *shared* Speicher von nur einem Thread, Blockgröße zur Festlegung der Affinität kann angegeben werden. |
| $upc\_alloc(...)$ | Allokation von *shared* Speicher mit Affinität zum aufrufenden Thread. |
| $upc\_free(...)$ | Freigabe von dynamisch allokiertem *shared* Speicher, bei kollektiver Allokation darf $upc\_free()$ nur einmal aufgerufen werden. |

Beim $upc\_forall$ Statement erfolgt keine Synchronisation der Threads. Hierzu gibt es das *upc_barrier* Statement, welches alle Threads synchronisiert, bevor sie weiterlaufen können. Das *upc_barrier* Statement ist blockierend. Mit den zwei Statements $upc\_notify$ und *upc_wait* steht ein gesplittetes Barrierenkonzept zur Verfügung.

### 4.3.4 Laufzeitfunktionen

Für den gegenseitigen Ausschluss aus gemeinsamen Programmsegmenten unterstützt UPC, ähnlich wie die Posix-Threads mit der Mutex-Funktionalität, die Programmierung einer kritischen Region. Nach der Deklaration einer Variable vom entsprechenden Typ und deren Allokation und Initialisierung durch $upc\_all\_lock\_alloc(...)$, stehen die Funktionen $upc\_lock(...)$ und $upc\_unlock(...)$ zur Verfügung. Der Aufruf von $upc\_lock(...)$ ist blockierend.

Eine weitere wichtige Gattung von Funktionen aus der UPC Laufzeitbibliothek sind die zur Allokation und Freigabe von dynamischem Speicher. Wegen des Distributed-Shared-Memory Modells bietet sie zusätzliche Funktionalität gegenüber der Standardbibliothek der Programmiersprache C. Die wichtigsten Funktionen sind in der Tabelle 11 aufgelistet.

## 4.4 Programmierung der Arbeitsverteilung

Die Vorgehensweise der drei Parallelisierungstechniken wird anhand einer nachträglichen Parallelisierung eines Programms untersucht, d.h. es wird davon ausgegangen, dass bereits ein serielles C++ Programm vorliegt, welches für ein Shared-Memory System parallelisiert werden soll. Dies war auch die Aufgabe bei der Parallelisierung der Softwarepakete DROPS, FIRE und ADI.

Es gilt weiter die Bedingung, dass der Code möglichst wenig umgestellt werden soll. Dies ermöglicht die Arbeit an einem Programmpaket in einer bestimmten Version, weitgehend unabhängig von der weiteren (seriellen) Entwicklung, und das Zusammenfügen zu einem späteren Zeitpunkt mit möglichst geringem Aufwand.

Die Grundlage einer solchen Parallelisierung sollte, unabhängig von der gewählten Parallelisierungstechnik, eine detaillierte Betrachtung des Laufzeitprofils des Programms auf einem representativen Datensatz (besser noch mehreren) auf allen relevanten Plattformen sein. Weiter empfiehlt es sich, während der Parallelisierung die Erfolge regelmäßig am Laufzeitprofil des Gesamtprogramms zu messen, da sich losgelöste Kernel unter gewissen Umständen anders verhalten, als beim Einsatz im Herkunftsprogramm. Bei der Erstellung eines representativen Laufzeitprofils ist folgendes zu beachten:

- Das Laufzeitprofil kann sich während der Laufzeit des Programms verändern, d.h. die Teile des Programms, in denen die meiste Rechenzeit verbraucht wird, verschieben sich. Dies ist auch bei DROPS der Fall, wo die ersten Zeitschritte deutlich mehr Rechenzeit benötigen als die späteren Zeitschritte.

- Das Laufzeitprofil kann auf verschiedenen Plattformen unterschiedlich ausfallen. Dies gilt sowohl für den seriellen als auch besonders für den parallelen Fall. Jede Plattform bietet mit ihrer Hardware, aber auch mit dem zum Einsatz kommenden Betriebssystem und Compilern, eine andere Charakteristik zum Beispiel bezüglich der Speicherhierarchie und der Stärke der Prozessoren bei bestimmten programmtypischen Operationen.

Mit der Fokussierung auf die Shared-Memory Parallelisierung erlauben es alle drei Parallelisierungstechniken, die laufzeitkritischen Stellen (Hotspots) eines Programms auszuwählen und unabhängig voneinander zu parallelisieren, soweit dies natürlich bei dem betrachteten Programm möglich ist. Anhand des einfachen Codes des STREAM-Benchmarks wird die Struktur der Parallelisierung mit den drei betrachteten Techniken dargestellt. Der Kern des STREAM-Benchmarks hat die Struktur wie im Programmausschnitt in Abbildung 9 dargestellt, wobei hier aus Gründen der Übersichtlichkeit nur der Summing-Testfall detailliert dargestellt wird, die Ergebnisse für die anderen STREAM-Testfälle und DROPS folgen im Verlauf des Kapitels.

Die äußere Schleife, im Programmausschnitt 9 von Zeile 2 bis Zeile 15, wiederholt die Testfälle $n$-mal, wobei $n$ durch die Makrodefinition $NTIMES$ vorgegeben wird. Für jeden Testfall wird die Zeit einzeln gemessen. Dies geschieht in Zeile 6 und Zeile 12. Die eigentliche Messung erfolgt von Zeile 8 bis Zeile 11: dem Array $c$ wird elementweise die Summe der Arrays $a$ und $b$ zugewiesen. Diese Schleife wird in den folgenden Abschnitten mit den drei betrachteten Parallelisierungstechniken parallelisiert. Der Übersicht halber werden die anderen drei Testfälle nicht dargestellt, wie in Zeile 4 und Zeile 14 angedeutet.

### 4.4.1 OpenMP

In OpenMP kann das Einfügen der Direktiven gezielt an den Hotspots erfolgen, während das restliche Programm unverändert bleibt, so dass die Codemodifikationen minimal sind. Da OpenMP als Compilererweiterung über Direktiven realisiert ist, kann bei

**Programmausschnitt 9** STREAM-Benchmark: Originalversion.

```
1 // MAIN LOOP --- repeat test case NTIMES times
2 for (k=0; k<NTIMES; k++)
3 {
4   [...]
5
6   times[k] = second();
7   // test case: Summing
8   for (j=0; j<N; j++)
9   {
10    c[j] = a[j] + b[j];
11  }
12  times[k] = second() - times[k];
13
14  [...]
15 }
```

Einhaltung der Konventionen (Korrektheit der Blöcke) sowie Verzicht auf die Benutzung von Laufzeitfunktionen ein OpenMP-Code auch von einem Compiler kompiliert werden, der OpenMP nicht unterstützt.

Die OpenMP Parallelisierung des obigen Testfalls des STREAM-Benchmarks ist im Programmausschnitt 10 dargestellt. Die parallele Region wurde um den gesamten Kern nach außen gezogen und erstreckt sich von Zeile 1 bis Zeile 27. Die drei Arrays *a*, *b* und *c* sowie das Array zur Zeitmessung *times* werden als *shared* deklariert, der Schleifenindex *k* muss privat sein. Die Zeitmessung wird nur vom Master-Thread durchgeführt. Da das *master*-Worksharingkonstrukt in OpenMP keine Barriere am Ende besitzt, muss ein Barriere folgen, wie in Zeile 12 dargestellt. Vor Zeile 20 muss sich keine Barriere befinden, da am Ende eines *for*-Worksharingkonstruktes eine implizite Barriere steht (Zeile 19).

Da die Umstrukturierung des Codes und das Einfügen von Anweisungen zur Erzeugung und Steuerung der Parallelisierung für den Benutzer unsichtbar vom Compiler durchgeführt wird, bleibt die Hauptarbeit bei der Shared-Memory Parallelisierung mit OpenMP das *Scoping*, das Einteilen von Variablen in *private* und *shared*. In dem hier dargestellten Code sind nur sehr wenige Variablen zu betrachten, dies ist aber bei anderen Programmen im Allgemeinen nicht der Fall.

Insgesamt mussten für den hier betrachteten Teil des STREAM-Benchmarks nur fünf Zeilen eingefügt werden, zusätzlich noch entsprechende Klammerung. Auf die Angabe eines Scheduling beim *for*-Worksharingkonstrukt könnte verzichtet werden, da auf den betrachteten Plattformen das statische Scheduling als Standard vom Compiler verwendet wird und dieses für diese Anwendung die beste Performance liefert.

**Programmausschnitt 10** STREAM-Benchmark: OpenMP Parallelisierung.

```
1 #pragma omp parallel shared(times,c,a,b) private(k)
2 {
3 // MAIN LOOP --- repeat test case NTIMES times
4 for (k=0; k<NTIMES; k++)
5 {
6   [...]
7
8 #pragma omp master
9 {
10   times[k] = second();
11 }
12 #pragma omp barrier
13
14   // test case: Summing
15 #pragma omp for schedule(static)
16   for (j=0; j<N; j++)
17   {
18     c[j] = a[j] + b[j];
19   }
20 #pragma omp master
21 {
22   times[k] = second() - times[k];
23 }
24
25   [...]
26 } // end for: k
27 } // end omp parallel
```

### 4.4.2 Posix-Threads

Bei der Parallelisierung mit den Posix-Threads ist der Programmierer für das Erzeugen und Zusammenführen der Threads selbst zuständig und muss dafür Code einfügen. Weiter muss er selbstständig zum einen das *Outlining* durchführen, zum anderen die Verteilung der Arbeit explizit programmieren. Mit *Outlining* wird das Erstellen einer Arbeitsroutine gemeint, die von jedem Thread mit entsprechenden Parametern zur Arbeitsverteilung aufgerufen wird. Diese Vorgehensweise wird auch von OpenMP-fähigen Compilern bei der Umsetzung der Direktiven in Binärcode angewendet, allerdings erfolgt sie dabei für den Benutzer unsichtbar.

Die Posix-Thread Parallelisierung des obigen Kernels des STREAM-Benchmarks teilt sich damit in zwei Teile. Bei dieser Implementierung wurde der Übersicht hal-

**Programmausschnitt 11** STREAM-Benchmark: Posix-Threads Parallelisierung (1).

```
1  // MAIN LOOP --- repeat test case NTIMES times
2  for (k=0; k<NTIMES; k++)
3  {
4    times[k] = second();
5    for (i = 0; i < NTHREADS; i++)
6    {
7      pthread_create(&threads[i], &attr, work_1,
8                     (void*) &i);
9    }
10   for (i = 0; i < NTHREADS; i++)
11   {
12     pthread_join(threads[i], NULL);
13   }
14   times[k] = second() - times[k];
15
16   [...]
17 }
```

ber die Anzahl der verwendeten Threads zur Compilezeit festgelegt, da bei dynamischen Datenstrukturen die Darstellung unübersichtlicher geworden wäre; natürlich lassen sich mit den Posix-Threads mit etwas mehr Aufwand beim Erzeugen und Beenden der Threads auch Programme schreiben, bei denen die Anzahl der Threads zur Laufzeit erst festgelegt wird.

Der Code im Programmausschnitt 11 zeigt die Programmierung des Kerns des STREAM-Benchmarks. Die äußere Schleife sowie die Zeitmessung bleiben unverändert. In Zeile 7 werden die Threads gestartet, als Argumente bekommen sie neben ihren Attributen ihre Thread-ID als Variable *i*. Anhand dieser Thread-ID wird später die statische Arbeitsverteilung berechnet. In Zeile 12 wird auf die Beendigung jedes Threads gewartet.

Die eigentliche Arbeit wurde aus dem Kern ausgelagert und in die Arbeitsroutine verschoben, welche für den betrachteten Testfall im Programmausschnitt 12 dargestellt ist. Von Zeile 4 bis Zeile 10 wird die statische Arbeitsverteilung berechnet. Die eigentliche Berechnung bleibt unverändert, bis auf die neu berechneten Grenzen der Schleife. Abschließend muss *pthread_exit*() aufgerufen werden, um den Kontrollfluss zu beenden, damit *pthread_join*() zurückkehrt.

### 4.4.3 UPC

Durch die Distributed-Shared-Memory Darstellung in UPC ist die Programmausführung etwas anders als bei den Posix-Threads oder OpenMP. Im Distributed-Shared-

**Programmausschnitt 12** STREAM-Benchmark: Posix-Threads Parallelisierung (2).

```
1 // Outlined worker function
2 void* work_1(void *i)
3 {
4   int iCount = nearbyint( (double)N / (double)NTHREADS );
5   int iStart = *(int*)i * iCount;
6   int iEnd, j;
7   if (*(int*)i == (NTHREADS - 1))
8     iEnd = N;
9   else
10    iEnd = ((*(int*)i+1) * iCount) - 1;
11  // test case: Summing
12  for (j=iStart; j<iEnd; j++)
13  {
14    c[j] = a[j] + b[j];
15  }
16  pthread_exit(NULL);
17 }
```

Memory Modell von UPC ist jede Variable lokal, falls nicht als *shared* deklariert, und jede Anweisung wird, falls nicht entsprechend gesteuert, von jedem Prozess bzw. Thread ausgeführt.

Wie bei den beiden anderen Parallelisierungstechniken ist es ebenfalls möglich, mit der Parallelisierung der Hotspots zu beginnen. Allerdings muss bei UPC darauf geachtet werden, dass dann die Ein- und Ausgabe nur von einem Prozess bzw. Thread durchgeführt wird und das Ergebnis, falls nötig, kommuniziert wird. In Programmausschnitt 13 wird die Ein- und Ausgabe vom Master-Thread durchgeführt.

Die Arrays *a*, *b* und *c* sind shared, das Array zur Zeitmessung *times* ist privat. Die Arbeitsverteilung erfolgt mittels *upc_forall*() in Zeile 16 und entsprechend der Affinität des zu bearbeitenden Elementes. Um eine korrekte Zeitmessung der gesamten Testfälle zu gewährleisten, müssen Barrieren eingefügt werden, wie z.B. in Zeile 14 und Zeile 20.

Die Abbildung zeigt die intuitive Implementierung des obigen Kernels des STREAM-Benchmarks in UPC, die damit erreichte Performance ist allerdings sehr enttäuschend im Vergleich zu den anderen Parallelisierungstechniken. Wie die Performance verbessert werden kann, wird in Abschnitt 4.6 beschrieben.

### 4.4.4 Ergebnis

Insgesamt lässt sich feststellen, dass die Parallelisierung mit OpenMP den meisten Komfort für den Programmierer bietet. Daneben ist in diesem Fall nur ein sehr gerin-

**Programmausschnitt 13** STREAM-Benchmark: UPC Parallelisierung.

```
1 shared double *a;
2 a = (shared double *) upc_all_alloc((N + OFFSET) *
3       sizeof(double), 1);
4 [...]
5
6 if (MYTHREAD == 0) {
7   [...] // IO
8 }
9
10 // MAIN LOOP --- repeat test case NTIMES times
11 for (k=0; k<NTIMES; k++)
12 {
13   times[k] = second();
14   upc_barrier;
15   // test case: Summing
16   upc_forall (j=0; j<N; j++; j)
17   {
18     c[j] = a[j] + b[j];
19   }
20   upc_barrier;
21   times[k] = second() - times[k];
22
23   [...]
24 }
```

ger Eingriff in den Code nötig. Weiter untersucht, aber aus Gründen der Übersichtlichkeit nicht im Text sondern im Anhang dargestellt, wurde die Parallelisierung der Rechenkerne aus DROPS, nämlich der Sparse-Matrix-Vektor-Multiplikation und des PCG-Lösers, welche den gleichen Schluss erlauben.

OpenMP bietet dem Programmierer somit einen High-Level Ansatz zur Parallelisierung, weitestgehend ohne Modifikation des Codes, was allerdings nicht immer möglich und auch nicht immer gewollt ist. Der Compiler setzt diese Parallelisierung letztlich in die Funktionalität um, welche von den Posix-Threads zur Verfügung gestellt wird. Diese kann der Programmierer auch selber benutzen, allerdings ist dies mit einem höheren Programmieraufwand verbunden als die Verwendung von OpenMP und bietet bei den betrachteten Programmen keinen Vorteil.

UPC bietet ebenfalls eine hohe Abstraktion und besondere Möglichkeiten zur Beeinflussung der Verteilung der Daten im Speicher. Falls die Arbeitsverteilung unabhängig von der Affinität der Daten geschehen soll, muss dies vom Programmierer manuell programmiert werden. Dies ist aber nicht als Nachteil zu sehen, da UPC mit dem

Distributed-Shared-Memory Modell einen etwas anderen Ansatz verfolgt als OpenMP und die Posix-Threads.

Weiter ist zu bemerken, dass OpenMP und UPC wegen der geringen Codeänderungen einen strukturierteren parallelen Code erlauben als bei der Programmierung mit den Posix-Threads. Allerdings müssen bei der Verwendung von UPC mit C++ ein paar Einschränkungen in Kauf genommen werden, wie in Abschnitt 4.3 dargestellt.

## 4.5 Vergleich der Tool-Unterstützung

Eines der wichtigsten Hilfsmittel für den Programmierer während der Entwicklung eines Programms ist ein Debugger. Beim Debuggen eines parallelen Programms sind folgende zwei Dinge die wichtigsten Funktionen, die von einem parallelen Debugger unterstützt werden müssen:

- Möglichkeit zur Anzeige privater Daten: neben Darstellungsmöglichkeiten von globalen (*shared*) Daten muss es möglich sein, private Daten von Threads bzw. Prozessen zu betrachten, da diese gerade für jeden Thread bzw. Prozess unterschiedlich sein können.

- Kontrolle einzelner Threads: es muss möglich sein, einzelne Threads bzw. Prozesse anzuhalten bzw. generell von den anderen losgelöst zu steuern. Ebenso sollte es möglich sein, globale Barrieren einzufügen und dann alle Threads bzw. Prozesse einzeln untersuchen zu können.

Der Debugger TotalView [Etnus 05] der Firma Etnus gilt allgemein als führender Debugger für parallele Programme und steht auf allen in dieser Diplomarbeit betrachteten UNIX-Plattformen zur Verfügung. Prinzipiell unterstützt er alle drei Parallelisierungstechniken mit den oben geforderten Funktionen. Darüber hinaus bietet er noch einige weitere, teilweise sehr nützliche Funktionen. Insgesamt muss die Verwendungsmöglichkeit aber ein wenig eingeschränkt werden:

1. OpenMP: prinzipiell werden zwar die verwendeten Kombinationen von Hardware, Betriebssystem und Compiler unterstützt, allerdings sind beim Kompilieren spezielle Optionen zu verwenden, damit die Darstellung des Programmcodes während des Debugging korrekt funktioniert. Dies ist aber nicht immer in vollem Umfang möglich, insbesondere höhere C++ Sprachkonstrukte wie z.B. geschachtelte Templates bereiten dem Debugger teilweise Probleme. Generell funktioniert das Debuggen aber gut und die oben genannten wichtigen Funktionen stehen zur Verfügung.

2. Posix-Threads: die unterstützten Kombinationen von Hardware, Betriebssystem und Compiler sind für die Posix-Threads zwar etwas geringer, die verwendeten Kombinationen werden aber unterstützt. Besser als bei OpenMP ist hier die Darstellung des Codes, da bei Verwendung der Posix-Threads Bibliothek keinerlei Optimierungen notwendig sind, wie dies bei OpenMP durch Einschalten der OpenMP Unterstützung im Compiler oft erzwungen wird.

3. UPC: auch wenn UPC prinzipiell unterstützt wird, gilt dies nicht für das in dieser Diplomarbeit verwendete Kommunikationssystem der Posix-Threads. Möglich ist die Arbeit mit MPI als Kommunikationssystem, dann wird das UPC Programm wie ein natives MPI-Programm behandelt. Allerdings ist es nicht möglich, UPC-spezifische Fähigkeiten zu verwenden sowie auf UPC-Zeiger zuzugreifen.

Allgemein lässt sich feststellen, dass der hier betrachtete parallele Debugger TotalView dem Programmierer in vielen Situationen in parallelen Programmen helfen kann. Dies gilt insbesondere bei einfachen Codes und relativ kurzen parallelen Regionen. Wenn sich innerhalb der parallelen Regionen eines Programms kompliziertere Codestellen befinden, muss man aber mit gelegentlichen Fehlern (Abstürze) des Debuggers oder mit Versagen bei der Darstellung von Daten rechnen.

## 4.6 Untersuchung der Performance

In der Grafik in Abbildung 11 ist der erreichte Speedup der oben im Abschnitt 4.4 vorgestellten Parallelisierungen des STREAM Summing-Kernels auf einem Linux-System mit 4 AMD Opteron Prozessoren (Sun Fire V40z) dargestellt. Die Versionen sind als C-Versionen bezeichnet, da keine C++ Sprachelemente verwendet werden und sie von den C++-Versionen in Kapitel 5 unterschieden werden sollen.

Gemessen wird der Speedup gegenüber der seriellen Originalversion. Aus diesem Grund ist der Speedup für die Posix-Threads Version und besonders für die UPC Version kleiner als eins, da diese Versionen langsamer als die Originalversion sind. Die OpenMP Version ist von den drei Versionen die beste, da sie seriell mit der Originalversion gleich auf liegt. Dennoch liefert sie nur einen minimalen Speedup. Der Hauptgrund für die schlechte Skalierung, die deutlich hinter den Erwartungen zurück bleibt, ist der, dass die Opteron Architektur, wie in Abschnitt 2.3 beschrieben, deutliche NUMA-Eigenschaften aufweist. Auf solchen Architekturen ist die Datenlokalität sehr wichtig. Im Folgenden wird zunächst der Grund für die schlechte Skalierung detailliert beschrieben und in den anschließenden Abschnitten dargestellt, wie die Datenlokalität in den drei betrachteten Parallelisierungstechniken berücksichtigt werden kann.

Wenn ein Thread auf die Teile der Daten, die er bearbeitet, nicht lokal sondern remote zugreifen muss, dann geschieht das über den HyperTransport Bus. Bei dem STREAM-Benchmark, der die Speicherperformance einer Architektur misst, führt dies natürlich zu besonders negativem Verhalten: bei der OpenMP Version und der Posix-Threads Version werden die Daten im seriellen Teil des Programms allokiert und initialisiert und kommen damit auf dem Speicher der CPU zu liegen, auf der der Master-Thread ausgeführt wird. Dieses Verhalten, welches vom Betriebssystem vorgegeben wird, heißt *First-Touch*. Wenn nun mehrere Threads eingesetzt werden, müssen sie alle auf den Speicher einer einzelnen CPU zugreifen, was natürlich den erreichbaren

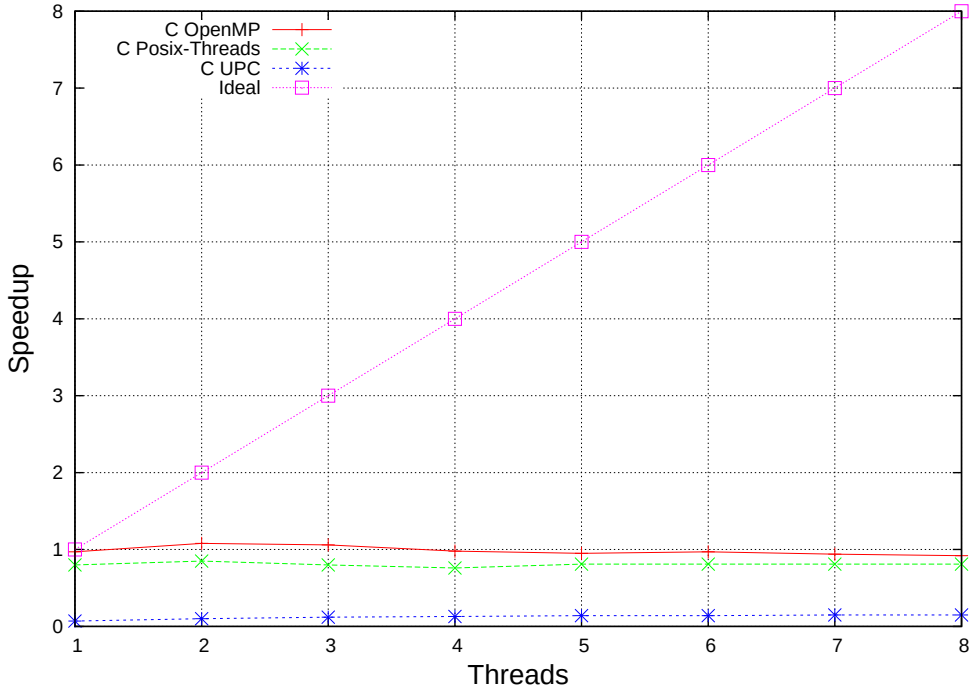


Abbildung 11: STREAM-Benchmark: Speedup auf Sun Fire V40z unter Solaris.

Speedup stark einschränkt.

Es fällt deutlich auf, dass die UPC Version seriell weit hinter den anderen Versionen liegt. Die Datenlokalität ist hier kein Problem, da die Arrays als shared Daten direkt verteilt angelegt, initialisiert und verwendet werden. UPC unterstützt dies dank des Distributed-Shared-Memory Modells. Der Grund für die sehr geringe serielle Performance und die kaum messbare Skalierung liegt im Compiler, der viele notwendige Optimierungen nicht selbstständig durchführt. Bevor die Verbesserung der Performance der UPC Version weiter unten in diesem Abschnitt beschrieben wird, wird zunächst die Skalierung der anderen beiden Versionen verbessert.

### 4.6.1 Tuning: OpenMP

Das Ziel ist es, die Daten so im Speicher abzulegen, dass die Verteilung genau dem Zugriff später im Programm entspricht. In OpenMP kann dies dadurch erreicht werden, dass die Initialisierung der Daten bereits parallel geschieht. Auf die Datenverteilung von C++ Programmen mit OpenMP wird detailliert im Abschnitt 5.2 eingegangen.

### 4.6.2 Tuning: Posix-Threads

Eine parallele Initialisierung soll auch bei den Posix-Threads die Datenlokalität verbessern. Hierzu ist eine weitere Arbeitsroutine zu erstellen und entsprechend in der Initialisierung parallel aufzurufen. Das Problem hierbei ist, dass die Zuordnung von Threads zu CPUs während der Initialisierung anders sein kann, als später im Programm bei der Berechnung. Wenn der zuerst erstellte Thread mit 0 bezeichnet wird und während der Initialisierung auf der CPU *i* ausgeführt wird, kann er später während der Arbeitsroutinen auf einer anderen CPU ausgeführt werden. Selbst ein Betriebssystem wie Solaris, das ein *CPU-Binding* unterstützt, kann hier keine Vorteile bringen, da die Threads ständig neu erstellt und wieder beendet werden. Prinzipiell ist die Erhaltung der Zuordnung auch in der OpenMP-Version nicht garantiert. Da die meisten OpenMP-Implementierungen aber mit Thread-Pools arbeiten, kann dann ein CPU-Binding greifen, da die Threads nach einer parallelen Region nicht beendet werden, sonst bei einer späteren parallelen Region wiederverwendet werden. Eine Änderung der Zuordnung ist bei den Posix-Threads somit deutlich wahrscheinlicher als bei OpenMP, da bei jeder Aufgabe die parallel zu bearbeiten ist, die Worker-Threads neu erzeugt und anschließend wieder terminiert werden. Die Performance nach den gerade beschriebenen Tuning-Maßnahmen, sowohl für die OpenMP Version als auch für die Posix-Threads, ist in der Grafik in Abbildung 12 dargestellt.

Der Grund für den Einbruch der OpenMP Version mit fünf Threads ist die Speicheranbindung, die dann von zwei Threads, die auf einer CPU laufen, geteilt werden muss. Diese können auf den Dual-Core Systemen zwar unabhängig voneinander arbeiten, allerdings hat ein physikalischer Prozessor nur eine Speicheranbindung. Mit vier bzw. acht Threads erreicht die Version mit ca. $11,8$ GB/s ungefähr den Maximalwert der Hardware.

Die schlechtere Performance der Posix-Threads Version, insbesondere der Einbruch der Performance mit vier Threads, ist hauptsächlich durch den oben beschriebenen Effekt der Änderung der Zuordnung von Threads auf CPUs zu erklären. Dazu kommt noch, dass das häufige Erstellen und Terminieren von Threads einen weiteren Aufwand gegenüber der OpenMP Version darstellt.

Unter Linux gibt es zur Zeit keine standardisierte API zur Abfrage der CPU-ID, unter Solaris ist dies möglich. Die CPU-ID könnte in diesem einfachen Fall bei der Berechnung der Arbeitsverteilung benutzt werden, um von der Zuordnung unabhängig zu sein.

Ebenfalls möglich ist ein CPU-Binding an eine CPU, die über die logische Thread-ID berechnet wird. Unter Linux ist dies mit der *numactl* [SuSE 05] Bibliothek möglich, Solaris unterstützt CPU-Binding ebenfalls. Weiter wurde ein einfacher Thread-Pool für die Posix-Threads implementiert, um den Aufwand des Erzeugens und Beendens von Threads zu verringern. Die Implementierung des Thread-Pools ist im Anhang C beschrieben. Die Performance, die unter Verwendung des Thread-Pools und CPU-Binding erreicht erreicht wird, ist in der Grafik in Abbildung 13 dargestellt.

Die Posix-Threads Version zeigt bei Verwendung des Thread-Pools mit Binding

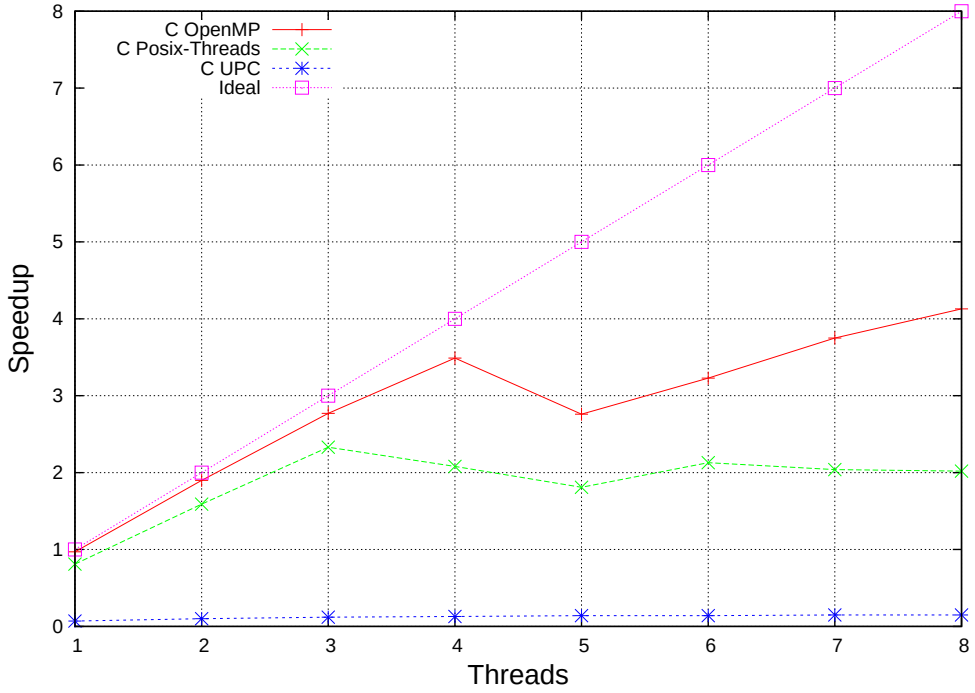


Abbildung 12: STREAM-Benchmark: Speedup auf Sun Fire V40z. Tuning: OpenMP + Posix-Threads.

ein anderes Verhalten als die OpenMP Version. Während die OpenMP Version mit vier Threads ein lokales Maximum erreicht, steigt der Speedup der Posix-Threads Version stetig bis auf das Maximum bei acht Threads an. Dies ist dadurch zu erklären, dass bei der OpenMP Version das Scheduling, also das Zuordnen von Threads auf CPUs, dem Betriebssystem überlassen wird und dabei zum einen freie CPUs für neue Threads bevorzugt werden, zum zweiten erst alle physikalischen CPUs belegt werden, bevor Threads auf die logischen CPUs verteilt werden. In der Posix-Threads Version ist im Thread-Pool ein einfaches CPU-Binding implementiert, bei dem der logische Thread $i$ (beginnend mit 0) auf die logische CPU $i$ gebunden wird. Dies bedeutet, dass bei der Messung mit zwei Threads beide Threads auf einer einzigen physikalische CPU gelaufen sind und sich somit die Speicheranbindung teilen mussten. Weiter werden bei vier Threads nur zwei physikalische CPUs benutzt und somit kann die maximale Bandbreite des Systems noch nicht erreicht werden.

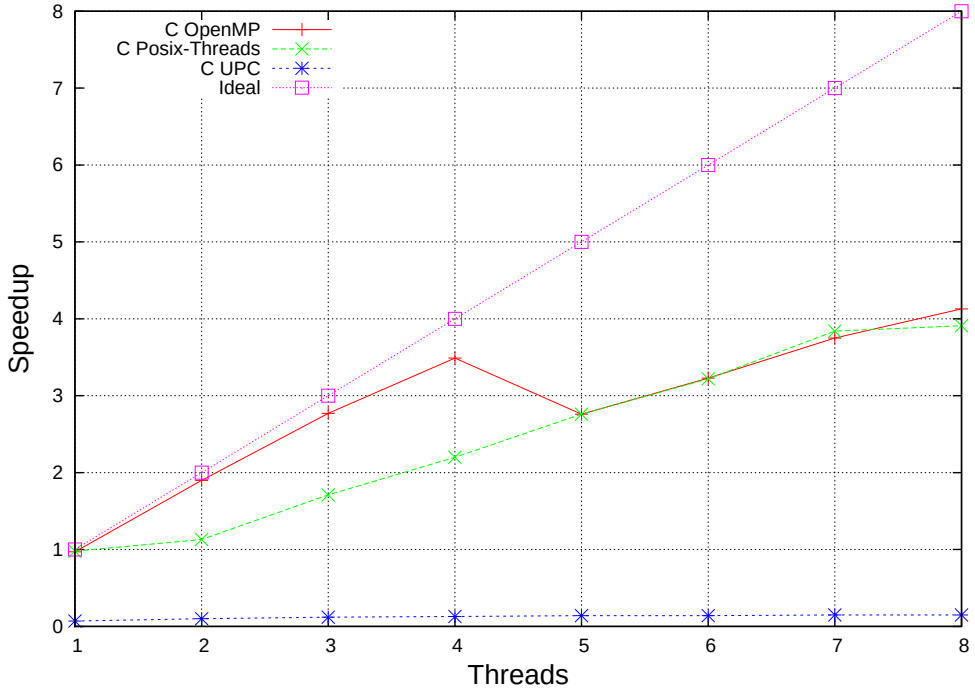


Abbildung 13: STREAM-Benchmark: Speedup auf Sun Fire V40z. Tuning: OpenMP + Posix-Threads mit Thread-Pool und Binding.

### 4.6.3 Tuning: UPC

Bei der UPC Version muss zunächst die serielle Performance deutlich verbessert werden. Dazu wurden drei Optimierungen durchgeführt, zwei davon manuell im Code und eine durch Verwendung anderer Parameter für den Compiler. Diese werden im Folgenden beschrieben.

Die erste manuelle Optimierung war es, die Variablen $a$, $b$ und $c$ vom globalen in den lokalen Scope zu verschieben, also in die Arbeitsroutine hinein. Der Grund dafür ist der, dass bei Verwendung von Posix-Threads als Kommunikationssystem eine Indirektionsebene beim Zugriff auf globale Variablen besteht, wenn diese nicht *shared* sind. Diese Verschiebung brachte einen Performancegewinn von ca. 10% bei der Ausführung mit einem Thread, die Skalierung wurde nicht verbessert.

Die nächste Optimierung war das Einschalten der Compileroptimierung durch den Parameter $-opt$. Diese Option besteht ab dem Berkeley UPC Release 2.2, befindet sich allerdings in einem als experimentell beschriebenen Zustand. Die Compileroptimierung (damit ist eine Optimierung des generierten C Codes, welcher später vom Systemcompiler übersetzt wird, gemeint) führt zu einer anderen Strategie bei der Co-

degenerierung für einfache, analysierbare `for`-Schleifen. Bei der einfachen Parallelisierung von for-Schleifen mit *upc_forall* führt jeder Thread in jeder Schleifeniteration den Schleifenkopf aus. Bei der optimierten Umsetzung führt nun jeder Thread nur die Schleifenköpfe aus, welche von ihm entsprechend der Affinität zu bearbeiten sind. Diese Optimierung hat die Performance nahezu verdoppelt und die Verbesserung skaliert mit einer steigenden Anzahl von Threads.

Die dritte Optimierung hat schließlich die Performance auf die der anderen Versionen angehoben. Sie wird als *Privatisierungsoptimierung* auch in [Chauvin & Saha+ 04] beschrieben. Das Ziel ist die Vermeidung von Overhead der *Pointer-to-Shared* Arithmetik in den inneren Schleifen. Hierzu werden lokale Variablen deklariert, welche auf den Teil der *shared* Variablen zeigen, der entsprechend der Affinität zu bearbeiten ist. Hierdurch wird erreicht, dass der generierte Code in den Schleifen gleich dem direkten Code ist, welcher auch in den anderen beiden Parallelisierungstechniken steht, und somit vom Plattformcompiler optimiert in Maschinencode umgesetzt werden kann. In Zukunft kann man hoffen, dass diese Optimierung vom Compiler selbstständig durchgeführt werden kann, zumindest in den Fällen, in denen die Datenverteilung statisch und damit analysierbar ist.

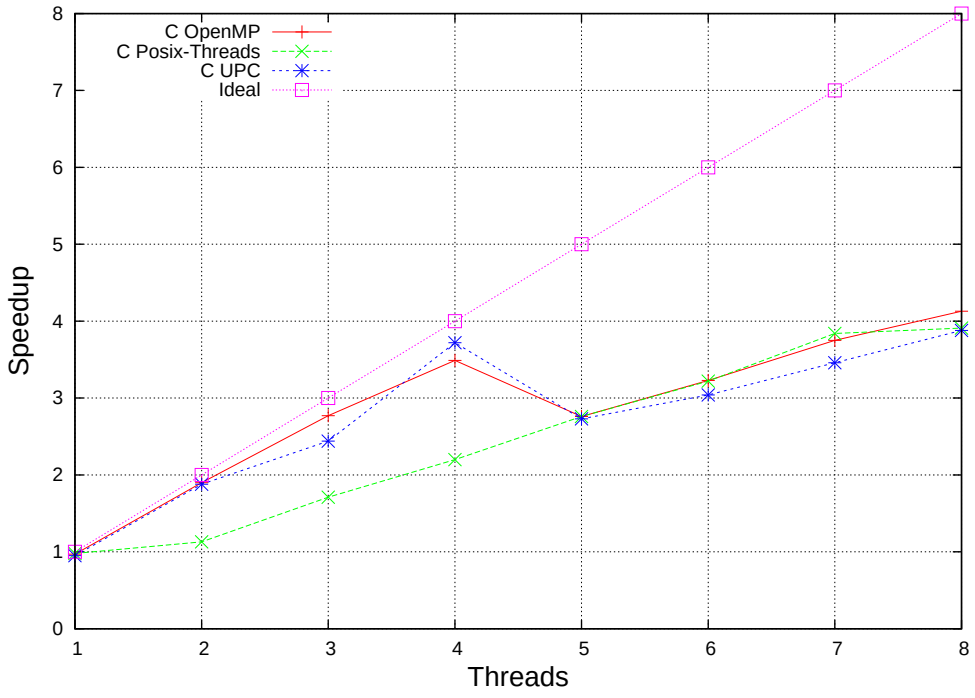


Abbildung 14: STREAM-Benchmark: Speedup auf Sun Fire V40z. Tuning: OpenMP + Posix-Threads mit Thread-Pool und Binding + UPC.

Nach der Anwendung aller Optimierungen ist die Performance des betrachteten Testfalls wie erwartet und in der Grafik in Abbildung 14 dargestellt. Mit allen drei Parallelisierungstechniken wird auf der NUMA-Architektur weitestgehend die maximale Bandbreite der Architektur erreicht.

### 4.6.4 Ergebnis

Die UPC-Version liegt nun noch ein kleines Stück vor der OpenMP-Version. UPC bringt direkt Unterstützung für die Verteilung der Daten mit, bei den beiden anderen Parallelisierungstechniken ist wie oben beschrieben darauf einzugehen, wobei dies bei den Posix-Threads nur bedingt erfolgreich ist. Insgesamt erreicht man mit der UPC-Version und der OpenMP-Version nun Grenzen dessen, was auf der betrachteten Hardware möglich ist, ebenfalls auch bei den Posix-Threads, wenn der Thread-Pool verwendet wird. Dies gilt beim STREAM-Benchmark neben dem betrachteten Testfall auch für die drei anderen Testfälle.

Der STREAM-Benchmark ist ein sehr einfacher Code, der zum Vergleich herangezogen wurde. Um in den Vergleich der drei Parallelisierungstechniken auch komplexere Codes aufzunehmen, wurden die bereits oben beschriebenen Programmkerne von DROPS untersucht, nämlich die dünnbesetzte Matrix-Vektor-Multiplikation der Routine *y_Ax* und das Lösungsverfahren mit Vorkonditionierer der Routine *PCG*. Es wurde nur das PCG-Verfahren und keines der anderen Lösungsverfahren betrachtet, da hierbei bei der Parallelisierung deutlich mehr Aufwand betrieben werden muss und es sich damit besser zum Vergleich der drei Techniken eignet.

Aus Gründen der Übersichtlichkeit soll hier nicht detailliert auf die Parallelisierung der Routinen eingegangen werden, da alle wesentlichen Konzepte bereits vorgestellt wurden. Die wichtigsten Codefragmente mit ein paar Erläuterungen finden sich im Anhang im Abschnitt A für die Routine *y_Ax* und im Abschnitt B für die Routine *PCG*. Hier soll nun aber auf die Performance der Routinen mit den drei Parallelisierungstechniken im Vergleich eingegangen werden.

In der Grafik in Abbildung 15 ist der Speedup der Routine *y_Ax* mit den drei Parallelisierungstechniken dargestellt. Die Messungen wurden auf einem Sun Fire V40z System unter Linux durchgeführt, da Linux die Produktionsplatform für DROPS ist. Insgesamt liegt die erreichte Performance aller drei Parallelisierungstechniken dicht beeinander. Dennoch wird auch beim Einsatz von vier Threads kein Speedup von mehr als 2,2 erreicht. Dies ist damit zu begründen, dass bei DROPS die Daten der benutzten Matrix und Vektoren im seriellen Teil des Programms angelegt und initialisiert werden und somit im Speicher eines Prozessors auf der NUMA-Architektur des Opteron zu liegen kommen. Da in DROPS eine nachträglich Ad-Hoc Parallelisierung durchgeführt wurde, wobei weitestgehend nur die rechenintensiven Routinen betrachtet wurden, kann mit den Posix-Threads und mit UPC kein Einfluss auf die Datenverteilung genommen werden. Wie dies mit wenig Aufwand mit OpenMP geschehen kann, wird am Ende von Abschnitt 5.2 beschrieben. Der begrenzte Speedup soll damit nicht als Nachteil der Parallelisierungstechniken ausgelegt werden.

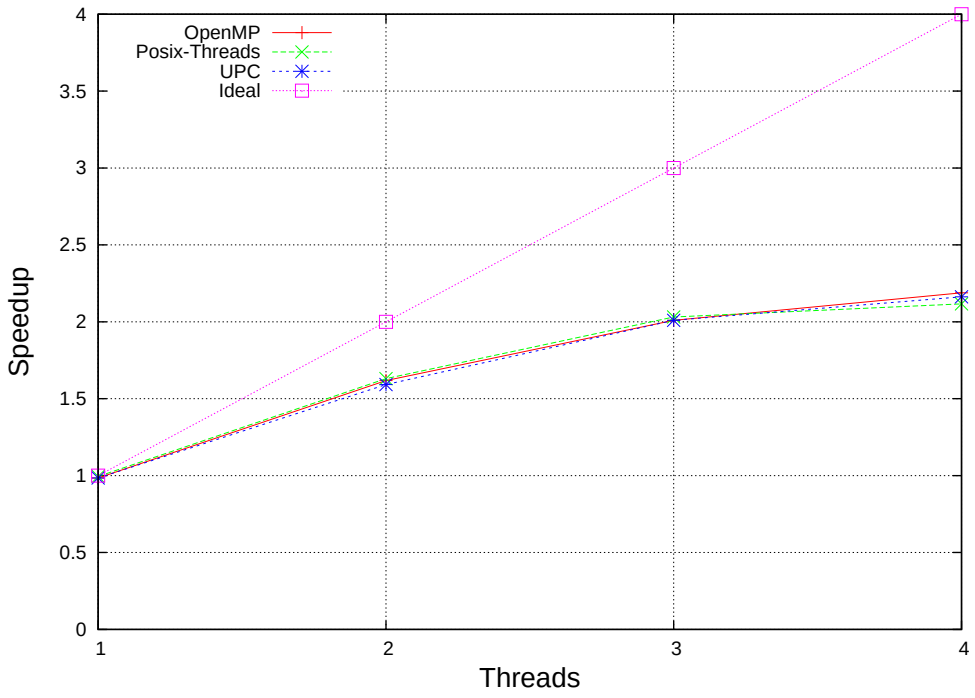


Abbildung 15: DROPS: Speedup *y_Ax* auf Sun Fire V40z unter Linux.

In der Grafik in Abbildung 16 ist der Speedup der Routine *PCG* mit den drei Parallelisierungstechniken auf einem Sun Fire E6900 System dargestellt. Die Performance der drei Techniken liegt wiederum dicht beeinander. Die Posix-Threads Version ohne Verwendung des Thread-Pools fällt aber deutlich hinter die anderen Versionen zurück, da in der Routine *PCG* noch deutlich mehr parallele Teile aufeinander folgen als in den bisher betrachteten Codes und der Aufwand zum Erstellen und Beenden von Threads entsprechend steigt. Insgesamt wird ein Speedup von über 3 erreicht. Die Begrenzung ist durch die Synchronisationspunkte innerhalb der PCG-Routine zu begründen.

## 4.7 Zusammenfassung

Es wurden die drei Parallelisierungstechniken Posix-Threads, OpenMP und UPC vorgestellt und miteinander verglichen. Grundsätzlich eignen sich alle drei Techniken für die nachträgliche Parallelisierung des DROPS C++ Codes. Neben der Untersuchung des Aufwandes wurde auch die Fähigkeit der einzelnen Techniken betrachtet, Einfluss auf die Platzierung der Daten auf einer NUMA-Architektur zu nehmen.

Bei OpenMP und bei UPC sind die notwendigen Codeänderungen im Vergleich zu

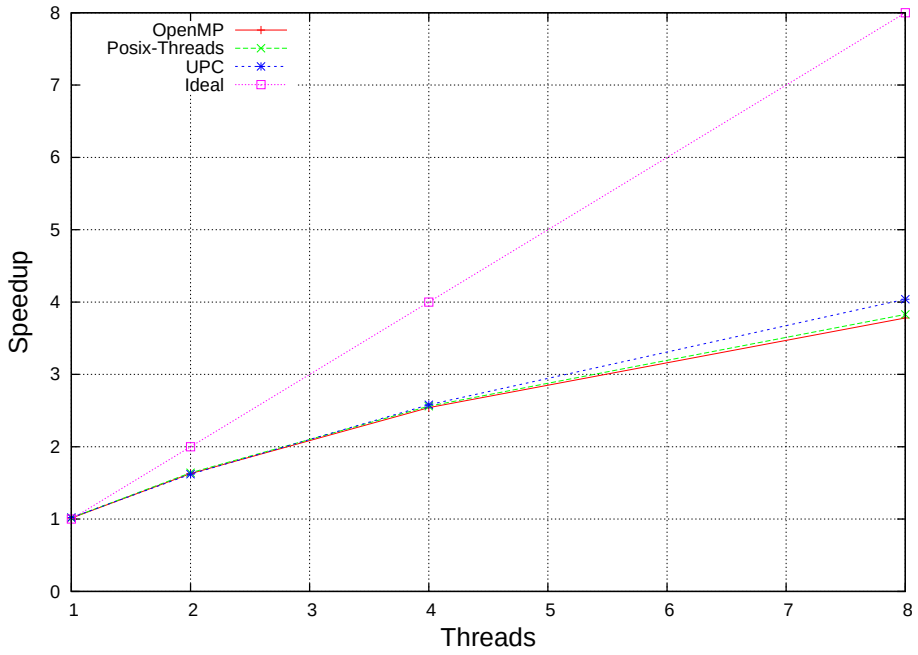


Abbildung 16: DROPS: Speedup *PCG* auf Sun Fire V40z unter Linux.

den Posix-Threads deutlich geringer. Durch das Einfügen oder Ändern weniger Zeilen war eine nachträgliche Parallelisierung sowohl des einfachen STREAM-Benchmarks als auch der umfangreicheren DROPS Routinen möglich. Mit dem anderen Ausführungs- und Speichermodell von UPC gegenüber OpenMP und den Posix-Threads ist die nachträgliche, vollständige Parallelisierung eines Programms mit UPC aber im Allgemeinen aufwändiger als mit OpenMP, da zumindest bei der Bearbeitung der Ein- und Ausgabe mehr Arbeit aufgewendet werden muss. Die Herangehensweise von Posix-Threads und OpenMP ähneln sich, bei den Posix-Threads ist aber zum Teil deutlich mehr manueller Aufwand notwendig, besonders wenn die Arbeitsverteilung nach einem dynamischen Schema geschehen soll. Dafür bieten sie auf der anderen Seite mehr Flexibilität bei der Parallelisierung von irregulären Codes, wie zum Beispiel in [Balart & Duran$^+$ 05] untersucht wurde. Solche Anforderungen sind im HPC-Umfeld aber eher selten.

Bei allen drei Parallelisierungstechniken wird der Programmierer durch die Möglichkeit des Einsatzes eines parallelen Debuggers unterstützt. Im Allgemeinen lassen sich Threads einzeln und unabhängig voneinander steuern und die privaten Variablen der Threads können sichtbar gemacht werden. Allerdings ist bei komplexeren Codes teilweise mit Instabilitäten der aktuellen Generation von Tools zu rechnen.

Mit den oben beschriebenen Maßnahmen lässt sich für alle drei Techniken die Per-

formance auf der NUMA-Architektur auf das erwartete Niveau heben. Mit Betrachtung der momentanen Entwicklung im Bereich der Hochleistungsrechner kann man feststellen, dass die Möglichkeiten zur Beeinflussung der Datenhaltung im Speichersystem für die erreichbare Performance einer Parallelisierungstechnik sehr wichtig sind. UPC hat auf diesem Gebiet wegen des anderen Speichermodells Vorteile. Es wurde aber auch gezeigt, dass bei entsprechender Programmierung mit den Posix-Threads und insbesondere mit OpenMP die Datenlokalität beachtet werden kann und damit die gleiche Performance erreicht werden kann.

# 5 C++ und OpenMP

Einer der Gründe, der den Einsatz von C++ [Stroustrup 00] im Hochleistungsrechnen attraktiv macht, ist die Unterstützung der objektorientierten Programmierung. Weiter erhöht der Einsatz von vordefinierten, leistungsstarken Typen aus der C++ Standard Template Library (STL) den Programmierkomfort und führt zu einer Verkürzung der Programmentwicklungszeit. Allerdings hat unter anderem auch die Parallelisierung von DROPS [Terboven & Spiegel$^+$ 05a] gezeigt, dass zum Erreichen einer zufriedenstellenden Performance bzw. Skalierbarkeit einige Dinge berücksichtigt werden müssen, welche im Folgenden detailliert betrachtet werden.

Zunächst erfolgt eine kurze Betrachtung der Thread-Safety in den verwendeten STL Implementierungen. In den Programmen DROPS und FIRE wird, teilweise recht intensiv, Gebrauch von STL-Datentypen gemacht. In diesem Zusammenhang ist es insbesondere bei DROPS zu Problemen bei der Skalierbarkeit gekommen, was in den Abschnitten über Datenlokalität und die Allokation kleiner Objekte betrachtet wird.

Daran anschließend wird untersucht, auf welche Weise die Parallelisierung eines High-Level objektorientierten Codes möglichst effizient erfolgen kann. Dies wird anhand des PCG-Lösers aus DROPS untersucht. Bei der Verwendung von Datentypen aus der STL werden gegebenenfalls Iterator-Schleifen programmiert, die nicht direkt in OpenMP parallelisiert werden können; Techniken zur Parallelisierung solcher Schleifen werden ebenfalls vorgestellt.

Das Kapitel schließt mit einer kritischen Betrachtung der aktuellen OpenMP Spezifikation im Hinblick auf die Unterstützung der C++ Programmiersprache. Es wird eine Liste von Beschränkungen durch OpenMP formuliert, welche die Arbeit an den in dieser Diplomarbeit beschriebenen Programmen aufwändiger gemacht haben.

## 5.1 Thread-Safety der STL

In den folgenden Abschnitten wird auf die parallele Programmierung mit High-Level C++ Konstrukten eingegangen. In vielen Fällen werden dabei Datentypen aus der *Standard Template Library* (STL) verwendet. Diese Datentypen verbergen hinter den von ihnen angebotenen Benutzerschnittstellen teilweise eine komplexe Funktionalität, so dass beim parallelen Zugriff auf sie einige Regeln zu beachten sind.

Ein Code ist *thread-safe*, wenn er bei gleichzeitiger Ausführung durch mehrere Threads korrekt ist. Der die STL beschreibende Standard macht keine Aussagen über die Sicherheit (Korrektheit) einer STL bei paralleler Verwendung, so dass diese je nach Implementierung unterschiedlich sein kann. Die hier beschriebenen Eigenschaften gelten für die STL Implementierungen auf den betrachteten UNIX-Systemen mit den zum Einsatz kommenden Compilern. Dies umfasst unter Solaris mit dem SUN C++ Compiler die SUN STL, welche auf der SGI STL basiert, und STLport, welche ebenfalls von der Implementierung der SGI STL abstammt. Unter Linux wird die GNU STL in Version 3 beschrieben, von der auch viele Elemente im Intel C++ Compiler verwendet

werden, welche sich im Bezug auf die Thread-Safety an die Vorgaben der SGI STL hält [FSF 05].

Um eine Aussage über die Korrektheit beim Zusammenspiel von Instanzen von STL-Datentypen mit mehreren Threads machen zu können, müssen zwei Fälle unterschieden werden:

1. Mehrere Threads greifen gleichzeitig auf eine einzige Instanz eines Datentypes zu.

2. *n* Threads greifen auf *n* Instanzen von STL-Datentypen zu, wobei nie mehr als ein Thread gleichzeitig auf eine Variable zugreift.

Zunächst wird Fall 1 betrachtet. Falls alle Threads ausschließlich lesend auf die Instanz zugreifen, können keine Fehler auftreten. Dies gilt auch, falls nur ein Thread schreibend auf die Instanz zugreift. Falls mehrere Threads gleichzeitig auf eine Instanz zugreifen, wobei mindestens ein Thread schreibend zugreift, könnte es zum einen zu internen Fehlern in der Datenstruktur kommen. Als Beispiel sei hier der Datentyp *std*::*map* genannt, der später noch verwendet wird. Die Einträge in einer *std*::*map* werden in einer Baumstruktur verwaltet, welche bei gleichzeitigen Veränderungen beschädigt werden kann. Zum zweiten könnte ein Update von einem oder mehreren Threads verloren gehen (*Lost Update*).

Um diese Probleme zu vermeiden, muss seitens der Applikation ein entsprechendes Locking, z.B. mittels kritischer Regionen, implementiert werden. Die folgenden Gründe, die in der SGI Implementierung der STL [SGI 05] aufgezählt werden, sprechen gegen eine Implementierung einer Locking-Strategie in der STL:

- In Fällen, in denen keine Parallelität zum Einsatz kommt, würde der Datentyp Performanceeinbußen hinnehmen müssen. Darüber hinaus kann die benötigte Granularität des Lockings bei verschiedenen Anwendungen differieren.

- Die Implementierung einer angepassten Locking-Strategie auf Seiten der Anwendung ist durch Ableitung vom STL-Datentyp flexibel und mit wenig Aufwand zu realisieren.

Allerdings führt auch der schreibende Zugriff mehrerer Threads auf eine einzelne Instanz nicht zwingend zu einem Fehler. Der Zugriff ist sicher, wenn die interne Representation der Daten nicht verändert wird. Als Beispiel sei hier der Datentyp *std*::*vector* genannt, der später noch verwendet wird. Falls der Vektor bereits angelegt ist, z.B. im seriellen Teil, und lediglich auf den Elementen des Vektors gearbeitet wird, wird die Darstellung nicht mehr verändert. Dennoch muss die Anwendung sicherstellen, dass keine *Race-Condition* für die Elemente des Vektors auftreten kann.

Nun wird Fall 2 betrachtet, in dem mehrere Instanzen eines Datentypes angelegt werden, aber auf eine Instanz immer nur von maximal einem Thread gleichzeitig zugegriffen wird. Damit dies zu keinem Fehler führt, müssen alle benutzten Funktionen

der Schnittstelle des entsprechenden Datentypes *reentrant* sein. Eine Funktion ist *reentrant*, wenn sie nur Variablen vom Stack benutzt, nur von den übergebenen Argumenten abhängt und alle von ihr aufgerufenen Funktionen ebenfalls diese Eigenschaften besitzen. Eine solche Funktion ist auch *thread-safe*. Dies impliziert, dass sich mehrere angelegte Instanzen eines Datentypes nicht beeinflussen können. Da auch keine globalen (z.B. statischen) Daten verwendet werden können, ist der Zugriff in diesem Fall sicher. Bei den betrachteten STL Implementierungen sind die von den Datentypen angebotenen Routinen alle *reentrant*. Bei der SGI STL und den darauf basierenden Implementierungen können lediglich die Allokatoren statische Daten enthalten, die allerdings durch ein Locking geschützt werden.
Als Ergebnis können folgende Aussagen zusammengefasst werden:

1. Ausschließlicher Lesezugriff von mehreren Threads auf eine oder mehrere Instanzen ist sicher.

2. Gleichzeitiger Zugriff von mehreren Threads auf mehrere unterschiedliche Instanzen ist sicher.

3. Falls ein oder mehrere Threads gleichzeitig eine Instanz in ihrer internen Darstellung verändern können, muss der Zugriff über eine Locking-Strategie seitens der Anwendung geschützt werden.

## 5.2 Datenlokalität

Wie schon in Kapitel 4 bei der Betrachtung der erreichbaren Performance anhand des STREAM-Benchmarks beschrieben, ist die Datenlokalität bei NUMA-Architekturen von zentraler Bedeutung zum Erreichen einer hohen Performance bzw. Skalierbarkeit. Leider ist dies beim Design von Klassenbibliotheken wie z.B. der STL nur begrenzt in Betracht gezogen worden. Gerade bei der High-Level Programmierung in C++ werden viele Details wie z.B. die Speicherverwaltung oft vor dem Programmierer verborgen, so dass bei der intuitiven Benutzung der angebotenen Datentypen oftmals nicht die optimale Performance erreicht wird.

Während man in C zur Darstellung von dynamischen Arrays direkt Zeiger verwendet, ist der STL-Datentyp *std*::*valarray*, wie in Abbildung 14 deklariert, der elegantere Weg, in C++ unter Verwendung von Templates schnelle Arrays insbesondere zur numerischen Berechnung zu verwenden.

**Programmausschnitt 14** STL-Datentyp *std*::*valarray*.

```
1 valarray<Data>
```

Der C++ Standard sagt aus, dass alle Elemente eines Valarrays nach dem Anlegen mit null initialisiert sind. Dies hat den Vorteil, dass die Initialisierung nicht manuell programmiert werden muss, aber auch einen Nachteil auf NUMA-Architekturen. Die

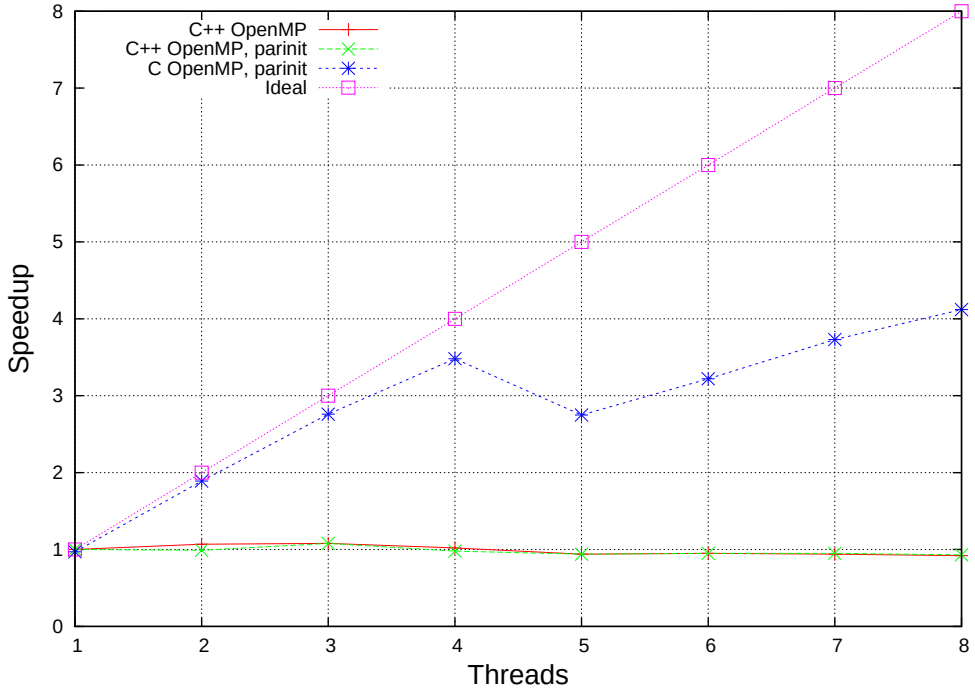


Abbildung 17: C++ STREAM-Benchmark: Speedup auf Sun Fire V40z unter Solaris.

Initialisierung mit null, also das erste Beschreiben vom allokierten Speicherbereich mit Werten, führt zur physikalischen Speicherzuweisung durch das Betriebssystem. Da in vielen Fällen das Anlegen der Daten im seriellen Teil eines Programms geschieht, wie auch im Fall von DROPS, oder aber durch den Master-Thread, werden die Daten im Speicher der entsprechenden CPU angelegt. Wenn im späteren parallelen Teil des Programms auf die Daten zugegriffen wird, erhält man den schon oben beschriebenen Engpass. Dies ist auch beim STREAM-Benchmark in der C++ Version so, wie die Grafik in Abbildung 17 zeigt. Hier wurde anstatt eines Zeigers ein Valarray zur Speicherung der Daten verwendet.

Die mit OpenMP parallelisierte C++ Version des STREAM-Benchmarks skaliert gar nicht. Der oben beschriebene Weg, die Initialisierung parallel durchzuführen, ist hier nicht erfolgreich. Da die Initialisierung bereits beim Anlegen des Valarrays erfolgt, sind die Daten danach physikalisch im Speicher angelegt und die parallele Initialisierung hat keinen Effekt mehr auf die Platzierung der Daten.

Um die Skalierbarkeit herzustellen, gibt es zwei Möglichkeiten: zum einen können Fähigkeiten des Betriebssystems zur Beeinflussung der Datenplatzierung benutzt werden, zum anderen kann durch Modifikation des Codes und Benutzung von C++

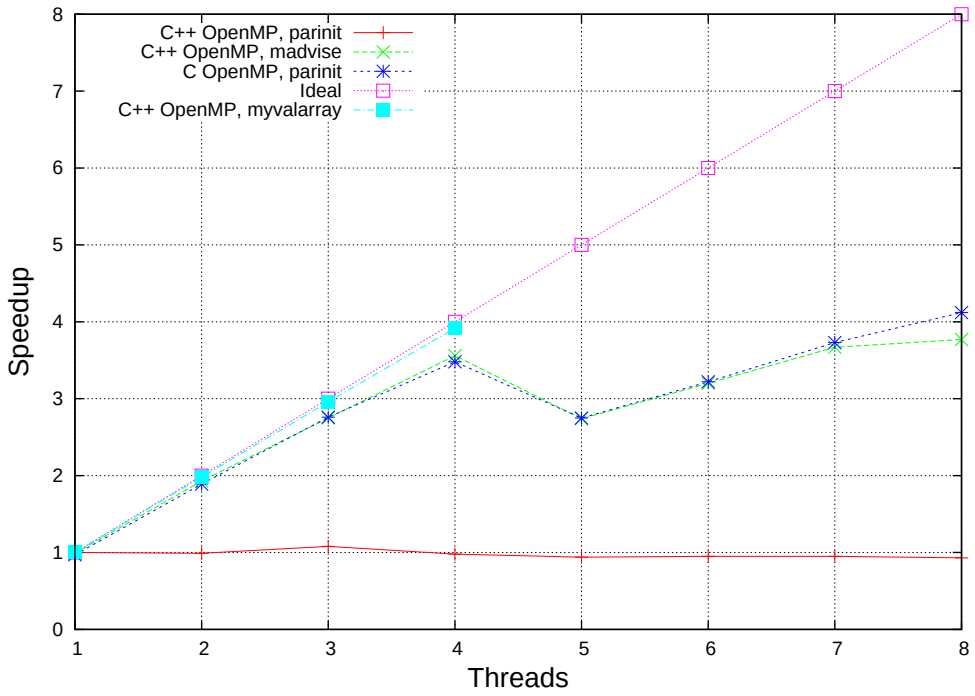


Abbildung 18: C++ STREAM-Benchmark: Speedup auf Sun Fire V40z unter Solaris bzw. Linux.

Sprachelementen Einfluss auf die Datenplatzierung genommen werden. OpenMP bietet in der aktuellen Version leider noch keine Mittel hierfür. Beide Wege werden im Folgenden vorgestellt und es wird sich zeigen, dass sie beide zu gleicher Performance führen. Der Schwerpunkt liegt aber auf der Benutzung von C++ Sprachelementen, da dieser Weg nicht vom eingesetzten Betriebssystem in einer speziellen Version abhängig ist, sondern portabel ist. Die dazu durchzuführenden Modifikationen am Code sind eher gering.

Zur Verbesserung der Skalierung ohne den Code zu verändern, muss auf Funktionalität vom Betriebssystem zurückgegriffen werden. Unter Solaris steht mit der Funktion *madvise*() die Möglichkeit bereit, bereits allokierten Speicher zu verschieben. Es gibt mehrere Arten der Migration. Die Strategie MADV_ACCESS_LWP führt dazu, dass beim nächsten Zugriff eines Threads auf ein Datum dieses zum Speicher der CPU migriert wird, auf der der zugreifende Thread läuft. Die Performance bei Anwendung dieser Methode beim STREAM-Benchmark ist in der Grafik in Abbildung 18 zu sehen.

Durch Benutzung von *madvise*() kann unter Solaris mit der C++ Version die gleiche Skalierung erreicht werden, wie mit der C Version und paralleler Initialisierung. Auch wenn die Initialisierung von nur einer CPU durchgeführt wird, kann also eine optimale Datenverteilung nachträglich hergestellt werden.

Unter Linux steht zur Zeit noch keine Funktionaliät zur Migration von Speicher zur Verfügung. Das bereits erwähnte *numactl* bietet dem Programmierer aber die Möglichkeit, Speicher verteilt zu allokieren. Dies kann, ohne grundlegende Änderungen am valarray Datentyp vorzunehmen, aber nicht eingesetzt werden. Im Konstruktor des *std*::*valarray* bzw. in der Routine *resize*() ist nur die Zielgröße bekannt. Um eine bessere Datenverteilung zu erzielen, wären weitere Informationen notwendig, wie zum Beispiel die Anzahl der Threads und die Abbildung von Thread-ID auf CPU-ID. Um diese Informationen zu erhalten, müsste das Interface des *std*::*valarray* Datentypes stark verändert werden.

Wenn man die Möglichkeit der Modifikation des Datentypes *std*::*valarray* zulässt, dann hat man mehrere Optionen. Das Überladen des Konstruktors und die entsprechende Verteilung der darin angeforderten Daten führt leider nicht zum Erfolg, da dort nur die Verwaltungsdaten angelegt werden. Zwar unterscheiden sich die Implementierungen von *std*::*valarray* in den verschiedenen STLs, aber generell werden die Daten in einem Array gehalten. Dieses wird mit dem *operator new* angelegt. Eine Möglichkeit, diesen für die STL zu überschreiben, ohne weitgehend nicht portable Modifikationen an der verwendeten STL vorzunehmen, ist mir nicht bekannt.

Als Experiment wurde ein eigener Datentyp *myvalarray* erstellt, indem die Implementierung von *std*::*valarray* aus der GNU STL übernommen wurde und lediglich die Speicherallokation verändert worden ist. In diesem Datenyp wird nach der Allokation des Speichers in einer mit OpenMP parallelisierten Schleife der Speicher verteilt mit null initialisiert. Da jeder Thread einen Bereich des Arrays initialisiert, werden diese Daten im Speicher der CPU angelegt, auf der der Thread ausgeführt wird. Dies ist nur erfolgreich, wenn sowohl *CPU-Binding* genutzt wird, als auch die Datenverteilung bei der Initialisierung der Datenverteilung in der späteren Rechnung entspricht. Die Skalierung dieses Ansatzes ist ebenfalls in der Grafik in Abbildung 18 zu sehen. Da diese Modifikation nur für die GNU STL der unter Linux eingesetzten Version durchgeführt wurde, ist dort nur der Speedup für bis zu vier Threads gezeigt.

Allgemein gibt es mehrere mögliche Arten, dynamisch Speicher anzulegen und zu verwalten. Daraus ergeben sich auch verschiedene Vorgehensweisen, um die Datenlokalität zu beeinflussen und zu optimieren. In den folgenden Abschnitten werden diese diskutiert, wobei nur Sprachelemente der Programmiersprache C++ und der dazugehörigen STL verwendet werden.

### 5.2.1 Zeiger

Insbesondere in C Programmen wird auf dynamische Daten wie z.B. Arrays mittels Zeigern zugegriffen. Diese Art der Programmierung ist natürlich auch in C++ möglich und auch weit verbreitet. Hier sieht der Programmierer bei Betrachtung des Codes

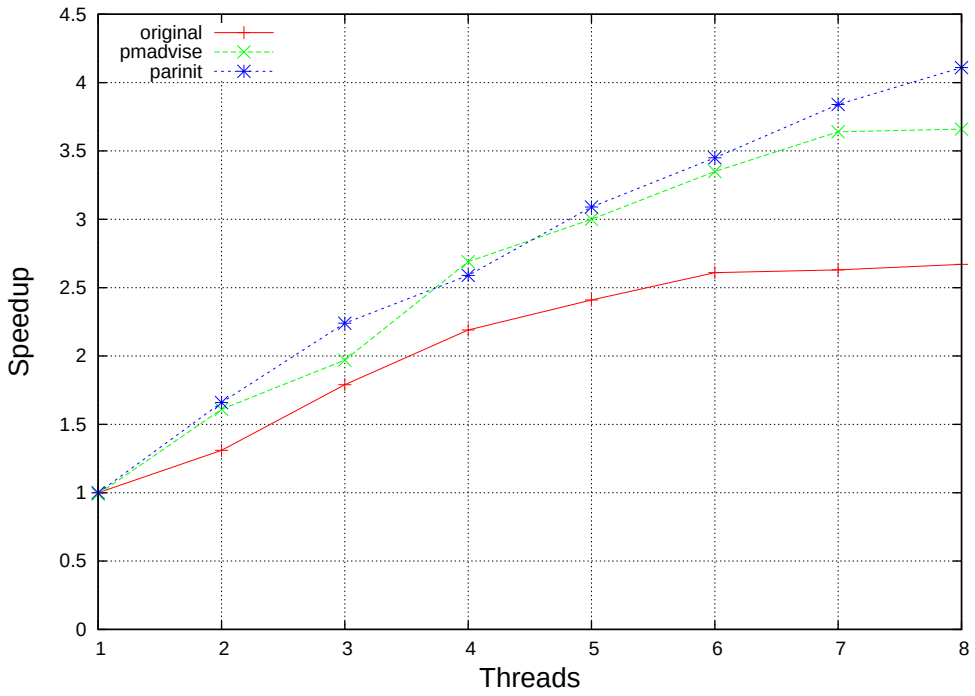


Abbildung 19: ADI: Speedup auf Sun Fire V40z unter Solaris.

direkt, wie die Daten angelegt werden und wie auf sie zugegriffen wird.

Ein Beispiel dafür ist ADI, welches in 3.3 beschrieben wurde. Die wichtigste Datenstruktur in ADI ist das Grid, welches wie im Programmausschnitt 15 gezeigt deklariert ist und angelegt und initialisiert wird. Da dies im seriellen Teil des Programms geschieht, werden alle Daten im Speicher einer CPU angelegt und im späteren parallelen Teil müssen die Threads auf anderen CPUs auf den Speicher der ersten CPU zugreifen.

Die Datenverteilung bei einem solchen Code zu beeinflussen, der ausschließlich Zeiger verwendet, ist in den meisten Fällen mit nur sehr wenigen Codeänderungen verbunden. In diesem Fall reicht es aus, die Initialisierungsschleife zu parallelisieren, was zu Code, wie im Programmausschnitt 16 dargestellt, führt.

Natürlich ist darauf zu achten, dass die Datenverteilung bei der Initialisierung auch der Datenverteilung während der Berechnung entspricht. Der hiermit erreichbare Speedup im Vergleich zum Code ohne Beachtung der Datenlokalität ist in Abbildung 19 gezeigt.

Diese Art der Programmierung der Datenverteilung hat den besonderen Vorteil, dass sie portabel ist, da keine Anforderungen an das Betriebssystem gestellt werden,

**Programmausschnitt 15** ADI: Definition, Deklaration und Allokation des Grids.

```
1 // Definition of gridPoint
2 class gridPoint {
3  public:
4    float P;
5    float P1;
6    float P2;
7    float P3;
8
9    float R1;
10   float R2;
11   float R3;
12
13   float source;
14   float * sigmap;
15
16   gridPoint() {
17     sigmap = new float;
18   }
19
20   [...]
21 };
22
23 [...]
24
25 // Declaration of variable grid
26 gridPoint ** grid;
27
28 [...]
29
30 // Allocation of variable grid
31 for (int i = 0; i < N; i++) {
32   grid[i] = new gridPoint;
33 }
```

sondern nur Elemente aus dem C++ Standard und dem OpenMP Standard eingesetzt werden. Falls die Plattform, auf der das Programm ausgeführt wird, jedoch kein Binding unterstützt, oder die Threads nach der Initialisierung auf andere CPUs migriert werden, ist der Performancevorteil verloren. Man kann dann sogar eine Verschlechterung der Performance erwarten.

Wie schon erwähnt bietet das Solaris Betriebssystem von SUN explizite Unter-

**Programmausschnitt 16** ADI: parallele Allokation des Grids.

```
1 // Allocation of variable grid
2 #pragma omp parallel for
3 for (int i = 0; i < N; i++) {
4   grid[i] = new gridPoint;
5 }
```

stützung zur Beeinflussung der Datenlokalität. Um die Performance der einfachen, manuellen Version mit der Performance unter Einsatz von Betriebssystemfunktionalität zu vergleichen, soll noch einmal kurz darauf eingegangen werden. Es gibt zwei Möglichkeiten:

1. Programmierung mit *madvise*(): hierbei wird für einen angegebenen Speicherbereich (auch der gesamte Speicherbereich des Programms ist möglich) die Speicherverwaltungsstrategie festgelegt. Für ADI vorteilhaft ist natürlich auch das oben schon erwähnte *MADV_ACCESS_LWP*, also die Migration von Speicher zu den CPUs auf denen die zugreifenden Threads laufen. Diese Strategie wird auch *Next-Touch* genannt.

2. Beeinflussung eines laufenden Programms mittels *pmadvise*: durch entsprechende Kommandozeilentools kann die Datenverteilung eines laufenden Programms verändert werden. Dies geschieht durch eine Modifikation der Speicherverwaltungsstrategie und hat einen Effekt wie ein Aufruf von *madvise*() aus dem Code selbst heraus. Die MPO-Tools sind bisher allerdings nicht öffentlich verfügbar.

In der Grafik in Abbildung 19 ist ebenfalls die Performance des Programms bei der Beeinflussung durch *pmadvise* dargestellt, wobei dies direkt nach Beginn des Programms, also während der ersten Iteration, geschah. Es ist zu sehen, dass die Version mit manueller Programmierung der Datenverteilung, ohne Benutzung der Fähigkeiten des Solaris Betriebssystems, die besten Ergebnisse liefert. Soweit dies möglich ist, ist diese Version somit den anderen Lösungen vorzuziehen, da sie portabel ist.

### 5.2.2 STL-Objekte

Viele Datentypen aus der STL bieten die Möglichkeit, einen anderen als den Standard-Allokator zu verwenden. Leider ist dies beim Array-Typ *std*::*valarray* nicht der Fall. Gerade dieser Typ, der intensiv im DROPS Code benutzt wird, hat bezüglich der Datenlokalität Probleme bereitet, wie weiter oben bereits beschrieben wurde. Die STL bietet noch einen zweiten Array-Typ an, und zwar *std*::*vector* vom Typ wie im Programmausschnitt 17 dargestellt.

Dieser Datentyp bietet die Möglichkeit, einen eigenen Allokator zu verwenden. Allerdings sind bei der Programmierung mit *std*::*vector* ein paar Punkte zu beachten, um die serielle Performance von *std*::*valarray* zu erreichen:

**Programmausschnitt 17** STL-Datentyp *std*::*vector*.

```
1 vector<Data, Alloc>
```

- Der Elementzugriff muss über *operator*[]() erfolgen, der Zugriff über *at*() führt zur Laufzeit eine Überprüfung des Zugriffsindex auf Gültigkeit durch. Dies ist für eine effizient Arbeit mit Elementen eines Vektors auf jeden Fall zu vermeiden.
- Die Größe von *std*:: *vector* muss entweder bei der Deklaration angegeben werden, oder dort weggelassen und im Code vor der ersten Benutzung mittels *resize*() angegeben werden. Weitere Größenänderungen sind zu vermeiden, da bei einer Vergrößerung ein *realloc*() (Implementierung der GNU STL v3.4) aufgerufen wird. Bei einer Verkleinerung wird der Speicher nicht freigegeben.
- Die Benutzung als Stack, d.h. die Verwendung von *push_back*() und ähnlichen Funktionen, ist zu vermeiden. Da der Implementierung seitens des Standards vorgeschrieben wird, möglichst wenig Wachstumsmehraufwand zu erzeugen, hat der Programmierer keine direkte Kontrolle über den tatsächlich verwendeten Speicher mehr.
- Der Datentyp *std*::*valarray* bietet einige mathematische Routinen an, die von *std*::*vector* nicht implementiert werden. Dies ist vor allem die elementweise Ausführung einer Operation auf alle Elemente. Benötigte Operationen müssen entsprechend implementiert werden.

Unter Beachtung dieser Vorgaben erreicht der STREAM-Benchmark unter Verwendung von *std*::*vector* anstelle von *std*::*valarray* bei Verwendung des Sun C++ Compilers die gleiche Performance. Das Interface ist für die beim STREAM-Benchmark und bei den DROPS Programmkernen benötigten Funktionen identisch.

Der Intel C++ Compiler bietet dennoch eine niedrigere Performance von ca. 10% bei Verwendung von *std*::*vector* im Vergleich zu einem *std*::*valarray* oder Zeiger. Dies liegt daran, dass bei *std*::*valarray* dem Compiler seitens des Standards viele Optimierungsmöglichkeiten zugestanden werden, die er ebenfalls bei Zeigern durchführen kann. Dazu hat der Intel C++ Compiler eine eigene Implementierung von *std*::*valarray*, während er sonst unter Linux die GNU STL (v3) benutzt (hier wurde der Intel C++ v9.0 Compiler betrachtet).

Die Idee eines Allokators, der bei einem Vektor die Datenlokalität verbessert, ist die, dass nach dem Aufruf von *malloc*() die Daten mit einem Wert (z.B. null) initialisiert werden und diese Initialisierung mit dem gleichen Datenzugriffsmuster erfolgt wie später die Berechnung. Dies entspricht der parallelen Initialisierung, wie sie in 4.6 gewinnbringend eingesetzt wurde. Ein solcher Allokator wurde implementiert und ist im Programmausschnitt 18 und Programmausschnitt 19 auszugsweise dargestellt.

**Programmausschnitt 18** Implementierung von *DistributedHeapMallocAllocator*.

```
1  // uses MallocHeapLayer for memory-allocation
2  // parallel initialization with 0 after memory allocation
3  template<typename _Tp, int OpenMPSchedule>
4  class DistributedHeapMallocAllocator
5  {
6   private:
7    DistributedHeapManager<MallocHeapLayer,
8                           OpenMPSchedule> HeapManager;
9
10   inline pointer allocate(size_type __n, const void* = 0)
11   {
12     pointer __ret = static_cast<_Tp*>
13                     (HeapManager.malloc(__n * sizeof(_Tp)));
14     return __ret;
15   }
16
17   struct rebind
18   {
19     typedef DistributedHeapMallocAllocator<_Tp,
20             OpenMPSchedule> other;
21   };
22
23   [...]
24
25 };
```

Die Klasse *DistributedHeapMallocAllocator* implementiert die Allokator-Schnittstelle, so wie sie von der STL vorgegeben ist und für jeden Typ bereits einen Standard-Allokator mit *std*::*allocator*<*typename*> vorhält. Die Klasse ist als Template realisiert und bekommt neben dem zu allokierenden Typ `_Tp` auch noch einen Parameter namens `OpenMPSchedule`, mit dem die Datenverteilung während der Initialisierung nach der Allokation gesteuert werden kann.

Die Routine *allocate*(...) aus der Allokator-Schnittstelle ist dafür verantwortlich, den benötigten Speicher zu allokieren. Als Parameter bekommt sie mit `__n` die Anzahl der angeforderten Elemente, der zweite Parameter wird nicht benutzt. Zurück gegeben wird ein Zeiger auf einen Speicherbereich ausreichender Größe, in der Allokation muss also die Größe der Elemente beachtet werden. `pointer` hat den Typ Zeiger auf `_Tp`.

Die eigentliche Allokation des Speichers vom Betriebssystem wird nicht in der Klasse *DistributedHeapMallocAllocator* durchgeführt, sondern in eine weitere Klas-

se ausgelagert. In Zeile 7 wird die Variable `HeapManager` deklariert, welche eine Instanz von *DistributedHeapMallocAllocator* ist. Innerhalb von *allocate*() wird in Zeile 13 die Routine *malloc*(...) aus `HeapManager` aufgerufen, welche dann für die tatsächliche Allokation verantwortlich ist. Die Auslagerung der eigentlichen Allokation hat keine technischen Gründe. Sie wird mit dem Ziel der Erstellung eines kleinen Frameworks zum Einsatz und Experimentieren mit verschiedenen Allokationstechniken begründet. Das Framework besteht aus drei Ebenen und ist an [Berger & Zorn$^+$ 01] angelehnt:

1. Allocator: Klassen, deren Namen auf Allocator enden, implementieren das Interface, welches die STL für einen Allokator vorgibt. In den Klassen selbst wird keine wirkliche Allokation durchgeführt, sondern sie besitzen als Member-Variable eine Instanz eines HeapManager, welcher die Routine *malloc*() anbietet.

2. HeapManager: Klassen, deren Namen auf HeapManager enden, bieten mindestens die Routinen *malloc*(...) und *free*(...) an. Diese Routinen erwarten als Parameter eine Anzahl von zu allokierenden Bytes, bzw. einen Pointer auf einen freizugebenden Speicherbereich. Auch hier wird noch keine eigentliche Allokation durchgeführt, sondern es wird eine Verwaltungslogik des allokierten Speichers implementiert.

3. HeapLayer: Klassen, deren Namen auf HeapLayer enden, bieten mindestens die Routinen *malloc*(...) und *free*(...) an. In diesen Routinen wird nun physikalischer Speicher angefordert. Diese Ebene war notwendig, um z.B. bei gleicher Verwaltungslogik verschiedene Systemaufrufe verwenden zu können.

Ziel war die verteilte Initialisierung des Speichers nach der Allokation, um die Datenplatzierung für die Anwendung zu verbessern. Diese Strategie implementiert die Klasse *DistributedHeapManager*, deren Code im Programmausschnitt 19 dargestellt ist. In Zeile 10 wird von `HeapLayer` Speicher vom Betriebssystem angefordert. Nach der Deklaration oben in *DistributedHeapManager* wird hier *MallocHeapLayer* verwendet, wo ein einfaches *malloc*() und *free*() aus der Standardbibliothek aufgerufen wird. Nach der Allokation wird von Zeile 11 bis Zeile 24 in Abhängigkeit des gewählten Scheduling der Speicher entsprechend parallel initialisiert. Da der Parameter `OpenMPSchedule` fest ist und schon zur Compilezeit feststeht, kann der Compiler entsprechenden Code generieren, so dass zur Laufzeit die `switch`-Unterscheidung nicht ausgeführt wird. Dies ist sehr wichtig für die Performance. Die Verwendung von OpenMP wird allerdings in den meisten Fällen das *Inlining* des Compilers verhindern.

Der nun ausführlich vorgestellte Allokator kann mit *std*::*vector* eingesetzt werden. Innerhalb von *std*::*vector* wird er dann für jegliche Art von Speicherallokation verwendet. Die `rebind`-Struktur, die in Abbildung 18 dargestellt ist, ermöglicht eine Konvertierung des Allokators für andere Typen innerhalb von *std*::*vector*. Jeglicher

**Programmausschnitt 19** Implementierung von *DistributedHeapManager*.

```
1  // uses user-provided heap-layer for memory-allocation
2  // arallel initialization with 0 after memory allocation
3  template<class HeapLayer, int OpenMPSchedule>
4  class DistributedHeapManager
5        : public HeapLayer
6  {
7   public:
8    void *malloc(size_t sz)
9    {
10     char *pRet = (char*) HeapLayer::malloc(sz);
11     switch (OpenMPSchedule)
12     {
13       case sched_static:
14 #pragma omp parallel for schedule(static)
15         for (long l = 0; l < sz; l++)
16           pRet[l] = 0;
17         break;
18       case sched_guided:
19 #pragma omp parallel for schedule(guided)
20         for (long l = 0; l < sz; l++)
21           pRet[l] = 0;
22         break;
23         [...]
24     };
25     return pRet;
26   }
27
28   [...]
29
30 };
```

Speicher, insbesondere der für ein angefordertes Feld, wird nun direkt nach dem Aufruf von *malloc*() innerhalb des Allokators verteilt initialisiert.

Das Ergebnis des C++ STREAM-Benchmarks mit *std*::*vector* und verteilter Initialisierung durch den beschriebenen Allokator ist in der Grafik in Abbildung 20 im Vergleich zu den anderen erfolgreichen C++ Techniken zur Beeinflussung der Datenplatzierung dargestellt.

Mit der vorgestellten Technik kann ebenfalls der DROPS Programmkern *y_Ax* modifiziert werden. Hierbei ermöglicht das Zusammenspiel von C++ und OpenMP das einfache Einbinden einer verteilten Initialisierung. Dies wäre auch mit den Posix-

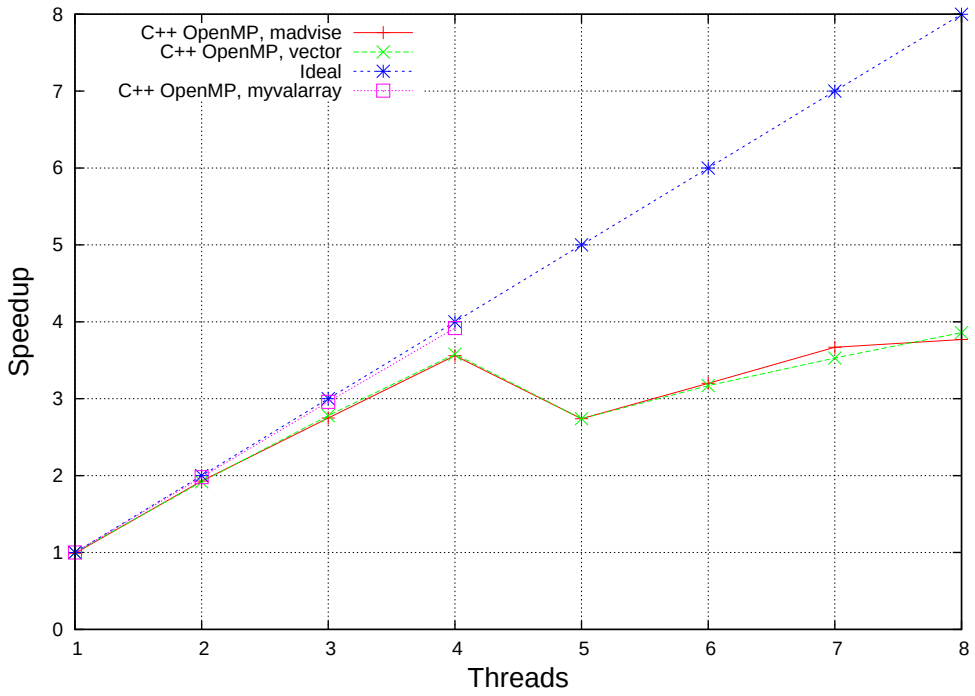


Abbildung 20: C++ STREAM-Benchmark: Speedup auf Sun Fire v40z unter Solaris.

Threads in ähnlicher Form möglich gewesen, allerdings zum einen aufwändiger in der Implementierung, zum anderen sind Bibliotheksaufrufe in der Allokation bezüglich der Performance des generierten Codes nicht optimal. Der Speedup im Vergleich zu den in 4.6 beschriebenen Implementierungen der drei Parallelisierungstechniken ist in der Grafik in Abbildung 21 gezeigt. Die Messung wurde auf einem Sun Fire V40z System unter Linux durchgeführt. Es zeigt sich, dass nun die erwartete Skalierung unter Berücksichtigung der NUMA-Architektur erreicht wird.

### 5.2.3 Allgemeine Klassen

Generell ist es möglich, bei allgemeinen Klassen in C++ mittels eines *Mixin* die Speicherdarstellung zu beeinflussen. Als *Mixin* [Bracha & Cook 90] werden Klassen bezeichnet, deren Superklasse verändert werden kann (die also eine neue Elternklasse erhalten können). In C++ werden *Mixins* als Templateklasse realisiert, die von der als Templateparameter angegeben Klasse abgeleitet werden, wie im Programmausschnitt 20 dargestellt ist. Mit allgemeinen Klassen sind diejenigen Klassen gemeint, die keine direkten Möglichkeiten zur Modifikation der Datenplatzierung, wie z.B. die Angabe eines Allokators, bieten.

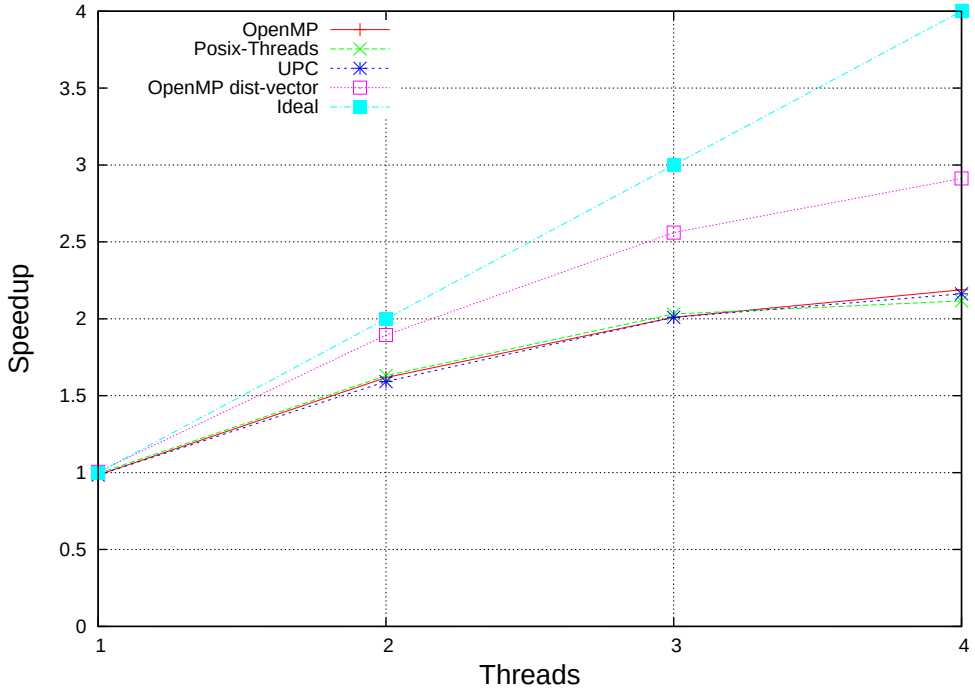


Abbildung 21: DROPS: Speedup *y_Ax* auf Sun Fire v40z unter Linux.

**Programmausschnitt 20** Mixin.

```
1 template <class Super>
2 class Mixin: public Super {};
```

Die Idee des Überschreibens des *new* und *delete* Operators einer Klasse durch ein Mixin zum Zweck des Eingriffs in die Speicherverwaltung wurde zusammen mit einigen spezialisierten Speicherverwaltungsklassen in [Berger & Zorn+ 01] vorgestellt. Die Klasse *PerClassHeapMixin*, die im Programmausschnitt 21 gezeigt ist, realisiert solch ein Mixin. Anstatt eine Instanz einer gegebenen Klasse anzulegen, wird eine Instanz der Klasse *PerClassHeapMixin* angelegt, wobei die gegebene Klasse der erste Templateparameter ist und ein Klasse vom Typ *HeapManager*, wie oben beschrieben, der zweite Templateparameter ist.

Bei Verwendung dieser Technik kann der von einer allgemeinen Klasse verwendete Speicher mittels einer eigenen Speicherverwaltung kontrolliert werden. Dies gilt allerdings nicht für Speicher, der in der Klasse dynamisch, das heißt durch den expliziten Aufruf von z.B. *malloc*() oder des *new* Operators, angefordert wird. Darauf kann von außen kein Einfluss genommen werden.

**Programmausschnitt 21** Implementierung von *PerClassHeapMixin*.

```
1  // C++ mix-in based on BZM01
2  template<class Object, class HeapManager>
3  class PerClassHeapMixin
4        : public Object
5  {
6   public:
7    inline void * operator new (size_t sz) {
8      return getHeap().malloc (sz);
9    }
10
11   inline void operator delete (void *ptr) {
12     getHeap().free (ptr);
13   }
14
15  private:
16   static HeapManager & getHeap () {
17     static HeapManager theHeap;
18     return theHeap;
19   }
20 };
```

### 5.2.4 Ergebnis

Um auf NUMA-Architekturen einen hohen Speedup zu erreichen, muss bei der parallelen Programmierung der Verteilung der Daten eine hohe Priorität zugewiesen werden. Dabei erfolgt die Verteilung im Idealfall genau so, wie der Datenzugriff in der späteren Rechnung. Die Programmiersprache C++ sieht keine direkten Möglichkeiten zur Beeinflussung der Datenplatzierung im Speicher vor. Darüber hinaus war es in den betrachteten Codes so, dass die gewählten C++ Datenstrukturen einige ungünstige Eigenschaften für den Einsatz zur Parallelisierung auf NUMA-Architekturen mitbringen. Um die Datenverteilung eines für Shared-Memory Systeme programmierten C++ Programms zu beeinflussen, wurden zwei Ansätze untersucht.

Zum einen kann auf Funktionalität des Betriebssystems zurückgegriffen werden. Hier wurden die Fähigkeiten des Betriebssystems Solaris 10 von SUN dargestellt. Sowohl durch den Aufruf von entsprechenden Bibliotheksroutinen aus dem Programm heraus, als auch durch die Verwendung von Kommandozeilenprogrammen während der Laufzeit, lässt sich die Datenverteilung sowohl darstellen als auch beeinflussen. Der Vorteil liegt hierbei darin, dass nur wenige oder gar keine Änderungen am Programm notwendig sind. Der Nachteil ist allerdings, dass beim Einsatz des Programms unter einem anderen Betriebssystem (durchaus auf der gleichen Hardwareplattform)

die Fähigkeiten gegebenenfalls nicht zur Verfügung stehen und die Performance somit nicht portabel ist.

Im zweiten Ansatz wurde ausschließlich auf Mittel der Programmiersprache C++ zurückgegriffen. Die Möglichkeit der Modifikation des Daten *std*::*valarray* wurde vorgestellt und damit wird die gewünschte Performance erreicht, aber die Bindung an einen Compiler verhindert in den meisten Fällen die Portabilität der Performance. Beim Austausch von *std*::*valarray* durch *std*::*vector* sind zwar ein paar Vorschriften zur Erhaltung der Performance zu beachten, allerdings bietet sich damit die Möglichkeit der Verwendung eines eigenen Allokators. Im vorgestellten Allokator wird der Speicher dem späteren Zugriffsmuster entsprechend verteilt und damit wird ebenfalls die gewünschte Performance erreicht. Zwar ist bei diesem Ansatz die Umstellung im Programm aufwändiger als bei der Benutzung von Mitteln des Betriebssystems, allerdings ist die erreichte Performance auf die meisten Plattformen portabel.

## 5.3 Allokation kleiner Objekte

Der STL-Datentyp *std*::*map* ist vom Typ wie im Programmausschnitt 22 angegeben. Er ist ein sortierter Container, der Objekte vom Typ `Key` mit Objekten vom Typ `Data` assoziiert. Dies bedeutet, dass der Wert eines *std*::*map* Containers damit den Typ *pair*<*const Key*,*Data*> hat. Es gilt, dass nicht zwei Elemente einer Map den gleichen Schlüssel haben können.

**Programmausschnitt 22** STL-Datentyp *std* :: *map*.

```
1 map<Key, Data, Compare, Alloc>
```

Die Einsatzmöglichkeiten von Maps als partielle Abbildungen sind vielfältig. Im Softwarepaket DROPS werden sie zum Aufbau der Steifigkeitsmatrizen in den Setup-Routinen verwendet. Die Matrizen sollen im CRS-Format erstellt werden, wobei zu Beginn der Routinen die Anzahl der Nichtnullelemente der Matrizen noch nicht bekannt ist. Hierzu wird für jede Zeile der zu erstellenden Matrix eine *std*::*map* angelegt, in welcher die Elemente einzeln abgelegt werden, wobei der Schlüssel die Spaltennummer ist. Dieses Verfahren ist deutlich speichereffizienter als z.B. das Vorhalten eines vollen Arrays für jede Zeile.

In einer Map werden die Elemente sortiert nach den Schlüsselwerten (also z.B. baumartig) gespeichert, aber natürlich nicht zusammenhängend. Das bedeutet, dass für jedes Einfügen in eine Map Speicher angelegt werden muss, ebenso für jedes Entfernen eines Elementes wiederum der Speicher freigegeben wird, durch entsprechende Bibliotheksaufrufe. Bei der Parallelisierung der Setup-Routinen in DROPS wurde festgestellt, dass dies zu Skalierbarkeitsproblemen führen kann. Zur genaueren Untersuchung des Verhaltens des Datenyps *std*::*map* wurde der MAP-Benchmark erstellt.

Die Grafik in Abbildung 22 zeigt die Laufzeit des MAP-Benchmark mit dem Intel C++ 9.0 Compiler unter Linux, dem SUN Studio10 C++ Compiler unter Solaris und

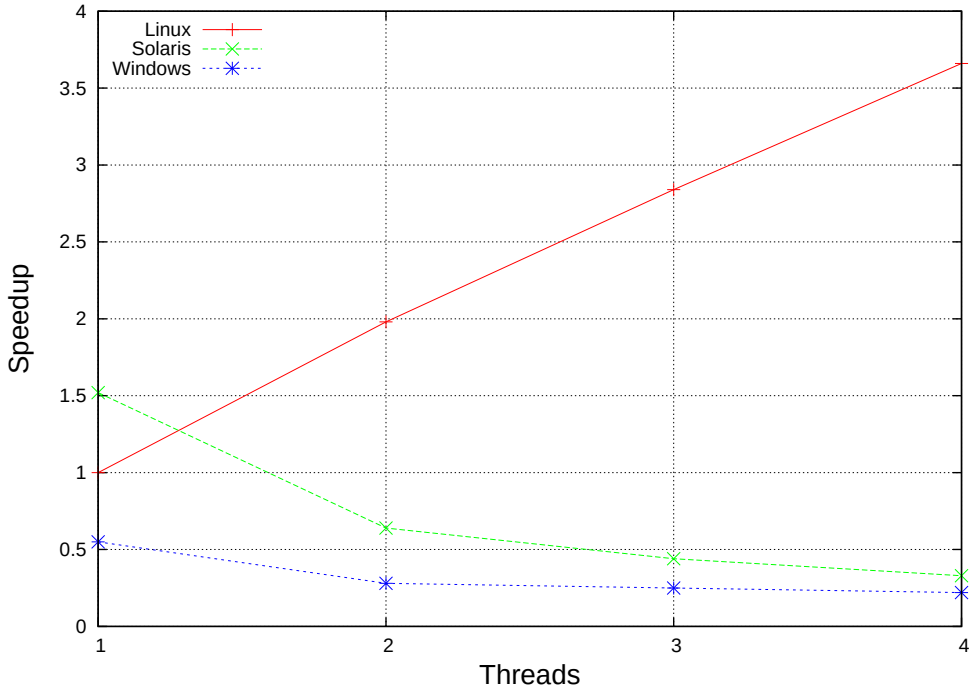


Abbildung 22: MAP-Benchmark: Speedup auf Sun Fire V40z.

dem Microsoft Visual Studio 2005 C++ Compiler auf dem Opteron: Ausgehend von einer seriellen Laufzeit von 7,47 Sekunden wird mit vier Threads ein Speedup von 3,61 mit dem Intel C++ Compiler erzielt.

Die Performance der beiden anderen Kombinationen aus Betriebssystem und Compiler ist aber enttäuschend. Beide zeigen einen Speedup kleiner als eins, d.h. dass das Programm mit zunehmender Anzahl von Threads langsamer wird. Da es im hier betrachteten MAP-Benchmark keine Synchronisationspunkte gibt und jeder Thread Elemente in eine private Map einfügt, was unabhängig vom Einfügen von Elementen in eine andere Map eines anderen Threads ist, wird die negative Skalierung durch die Speicherallokation beim Einfügen von Elementen hervorgerufen.

In den folgenden beiden Abschnitten wird untersucht, wie auf den beiden Kombinationen von Betriebssystem und Compiler die Skalierung des MAP-Benchmarks hergestellt werden kann. Zuerst erfolgt eine Konzentration auf Mittel des Betriebssystems, anschließend wird eine weitere Verbesserung mit C++ Sprachmitteln vorgestellt.

### 5.3.1 SUN Solaris

Auf den Solaris-Plattformen steht die *mtmalloc*-Bibliothek zur Verfügung, welche eine skalierbare *malloc*()-Implementierung anbietet. Im Allgemeinen genügt es, beim Linken die Bibliothek über $-lmtmalloc$ anzugeben, um ihre Version von *malloc*() und $free()$ zu verwenden.

Die beim Sun C++ Compiler zum Einsatz kommende STL-Bibliothek verwendet jedoch im Zusammenhang mit Maps einen speziellen Allokator zum Anlegen kleiner Objekte auf dem Heap, den sog. *node_allocator*. Ebenfalls unter Solaris, aber nur auf den UltraSPARC Servern, kommt der KAI Guide C++ Compiler zum Einsatz, welcher ähnlich dem SUN C++ Compiler einen eigenen Allokator zum Anlegen kleiner Objekte auf dem Heap verwendet, den sog. *fast_allocator*. Beide Allokatoren implementieren dieselbe Idee: Anstatt für jedes anzulegende Element den entsprechenden kleinen Speicherplatz vom Betriebssystem anzufordern, werden größere Blöcke allokiert und aus diesen die Anforderungen der Anwendung bedient.

Dies führt zu einer höheren seriellen Performance, da Verwaltungsaufwand eingespart wird. Aus diesem Grund ist der Speedup für den Sun C++ Compiler unter Solaris in Abbildung 22 auch größer als eins für einen Thread. Ein weiterer Vorteil ist eine Verringerung der Fragmentierung des Speichers bei vielen (nachfolgenden) Allokationen. Ein Vergleich verschiedener Allokationstechniken ist in [Attardi 03] zu finden.

Der Grund für die schlechte Skalierbarkeit sowohl des $node\_allocator$ als auch des $fast\_allocator$ ist der, dass die Bearbeitung der Anfragen aus der Anwendung (also das Verteilen von kleinen Speicherstücken aus den vom Betriebssystem angeforderten Blöcken) *thread-safe* sein muss. In diesen Allokatoren finden sich die einzigen Punkte in den betrachteten STL Implementierungen, an denen statische Daten verwendet werden. Um eine korrekte Implementierung zu gewährleisten, wurde die Blockaufteilung als eine kritische Region implementiert. Da somit bei jedem Einfügen eines Elementes in eine Map, auch wenn sie privat für jeden Thread ist, eine Synchronisation erfolgt, wird eine Skalierung der parallelen Region verhindert.

Bei der Verwendung von Maps aus den STL Implementierungen im SUN C++ Compiler und im KAI Guide C++ Compiler führt eine Anbindung der $libmtmalloc$ nicht direkt zum Erfolg. Da beim Anlegen eines neuen Elementes in einer Map kein Aufruf an *malloc*() stattfindet, sondern an den jeweiligen spezialisierten Allokator, hat die Ersetzung von *malloc*() in der Anwendung keinen Effekt. Nach dem Binden mit $libmtmalloc$ ergibt sich eine Performance, welche sich kaum von der Situation vorher unterscheidet, da die kritischen Aufrufe nicht ersetzt wurden.

Damit die $libmtmalloc$ greifen kann, muss die Verwendung von *malloc*() und $free()$ erzwungen werden. Hier bieten, wie oben schon dargestellt, viele Datentypen aus der STL die Angabe eines eigenen Allokators. Ein Allokator, welcher zum Anfordern von Speicher vom Betriebssystem *malloc*() und zur Freigabe von Speicher $free()$ verwendet, kann wie, im Programmausschnitt 23 dargestellt, implementiert werden.

Die Grundidee der Implementierung der $libmtmalloc$ ist die, dass sowohl ein globaler Heap für die Anwendung, als auch mehrere private Heaps für die Threads ver-

**Programmausschnitt 23** Implementierung von *MallocHeapAllocator*.

```
1  // uses OS-provided malloc and free for memory-allocation
2 template<typename _Tp>
3 class MallocHeapAllocator {
4  public:
5   typedef size_t      size_type;
6   typedef _Tp*        pointer;
7
8   template<typename _Tp1>
9   struct rebind {
10    typedef MallocHeapAllocator<_Tp1> other;
11  };
12
13  [...]
14
15  pointer allocate(size_type __n, const void* = 0) {
16    pointer __ret = static_cast<_Tp*>(malloc(__n *
17                    sizeof(_Tp)));
18    return __ret;
19  }
20
21  void deallocate(pointer __p, size_type) {
22    free(static_cast<void*>(__p));
23  }
24 };
```

waltet werden. Somit kann die Speicherallokation ohne Synchronisation stattfinden. Eine detailliertere Erläuterung der Funktionsweise von skalierbaren Allokatoren ist anhand des HOARD-Allokators in [Berger & McKinley$^+$ 00] zu finden. Die Performance unter Solaris unter Verwendung der *libmtmalloc* ist in der Grafik in Abbildung 23 dargestellt.

### 5.3.2 Microsoft Windows

Die Performance beim MAP-Benchmark ist unter Windows mit dem Microsoft C++ Compiler aus Visual Studio 2005 noch niedriger als unter Solaris. Hier wird in der STL zwar kein spezialisierter Allokator mit interner Synchronisation verwendet, aber der Zugriff auf den Heap ist in der derzeitigen Version zur Sicherstellung der Thread-Safety mit einer kritischen Region seitens des Betriebssystems versehen und somit synchronisiert. Dies betrifft alle Aufrufe von *malloc*().

Auch für Windows stehen mehrere Bibliotheken zur Verfügung, welche eine eigene

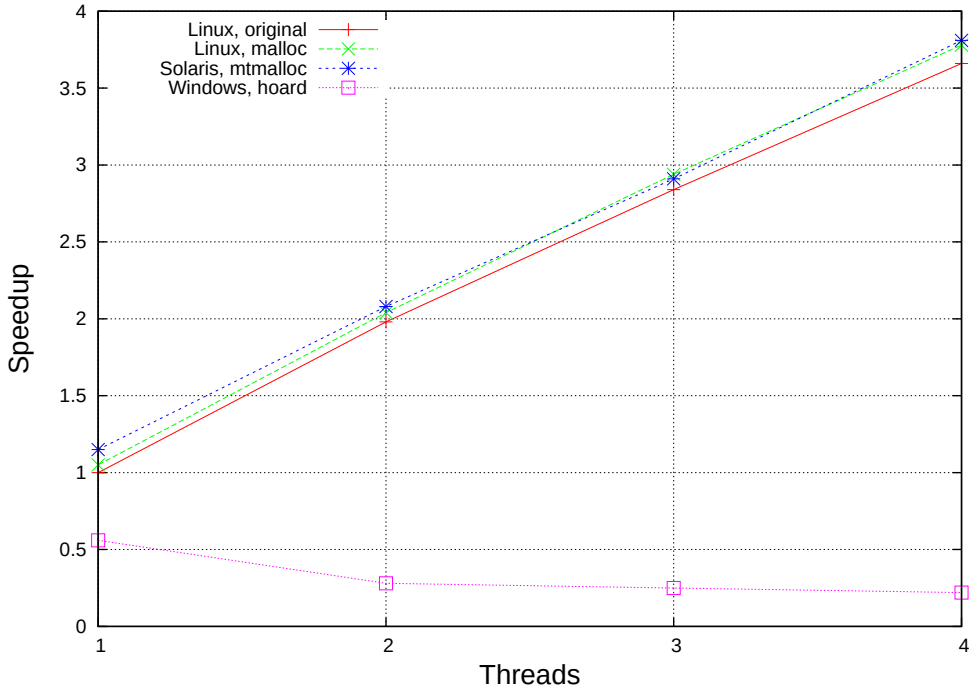


Abbildung 23: MAP-Benchmark: Speedup auf Sun Fire V40z. Tuning: *libmtmalloc* + *libhoard*

Heap-Verwaltung implementieren. Die *libmtmalloc* ist zwar nicht verfügbar, jedoch die *libhoard*, welche wie in [Attardi 03] untersucht in vielen Fällen der *libmtmalloc* im Bezug auf die Performance gleicht und sogar noch eine bessere Ausnutzung des Speichers bietet.

Die Performance des Benchmarks unter Verwendung von *libhoard* ist in der Grafik in Abbildung 23 dargestellt. Es kann keine serielle Verbesserung erzielt und auch keine positive Skalierung erreicht werden, jedoch kann eine Verschlechterung mit mehr als zwei Threads verhindert werden.

### 5.3.3 C++ Sprachmittel: Chunk-Allokator

Wie schon in 5.2 und weiter oben in diesem Abschnitt beschrieben, bietet der STL-Datentyp *std*::*map* die Möglichkeit der Angabe eines eigenen Allokators. Während oben ein einfacher Allokator benutzt wurde um den Aufruf von $malloc()$ und $free()$ sicherzustellen, wird nun eine eigene Speicherverwaltung in einem Allokator implementiert.

Der *Chunk-Allokator*, der ebenfalls nach dem oben vorgestellen Framework imple-

**Programmausschnitt 24** Implementierung von *ChunkHeapManager*.

```
1  template<class HeapLayer>
2  class ChunkHeapManager
3        : public HeapLayer
4  {
5   public:
6    [...]
7
8    ~ChunkHeapManager() {
9      std::list<char*>::iterator it;
10     for(it = lChunkList.begin(); it != lChunkList.end(); it++)
11       {
12         HeapLayer::free(*it);
13       }
14   }
15
16   void *malloc(size_t sz) {
17     if (sz <= stFreeBytesOnChunk)
18     {
19       char* vpPtr = vpNextPtr;
20       stFreeBytesOnChunk -= sz;
21       vpNextPtr = (char *)vpNextPtr + sz;
22       return vpPtr;
23     }
24     else
25     {
26       char* vpPtr = (char*)HeapLayer::malloc(ChunkSize);
27       lChunkList.push_back(vpPtr);
28       stFreeBytesOnChunk = ChunkSize - sz;
29       vpNextPtr = (char *)vpPtr + sz;
30       return vpPtr;
31     }
32   }
33 };
```

mentiert ist und dessen Implementierung des *HeapManager* im Programmausschnitt 24 dargestellt ist, wendet folgende Idee an:

- Bei der ersten Anforderung der Anwendung oder falls nicht genügend Platz auf dem aktuellen Chunk verfügbar ist: reserviere einen Chunk Speicher fester Größe vom Betriebssystem, füge ihn in die Liste der bereits reservierten Chunks ein, bediene die Anforderung der Anwendung davon und berechne den verbleiben-

den Speicherplatz auf dem Chunk.

- Falls noch genügend Platz auf dem aktuellen Chunk verfügbar ist: bediene die Anforderung der Anwendung und berechne den verbleibenden Speicherplatz auf dem aktuellen Chunk.
- Falls Speicher freigegeben werden soll: ignoriere diese Anforderung.
- Bei der Destruktion des Allokators gebe den gesamten allokierten Speicher auf einmal frei.

Durch diese Strategie wird, in Abhängigkeit der in der Implementierung festgelegten Größe des Chunks, die Anzahl der $malloc()$-Aufrufe an die Laufzeitbibliothek und damit vor allem die Anzahl der möglichen Synchronisationspunkte (unter Solaris und unter Windows) deutlich reduziert. Ein weiterer Vorteil, der sich bei entsprechender Größe der Chunks ergibt, ist eine Verringerung der Fragmentierung des Speichers. Zur Evaluation der Performance werden sowohl der MAP-Benchmark als auch die Betrachtung der Setup-Routinen aus DROPS herangezogen.

Vor der Betrachtung der Performance sind aber noch ein paar Einschränkungen der oben gezeigten Implementierung anzugeben, da der $Chunk-Allokator$ speziell für den Einsatz in den Setup-Routinen von DROPS entworfen wurde. Eine Behandlung der aufgeführten Probleme würde zwar eine weitergehende (allgemeine) Anwendbarkeit ermöglichen, aber in vielen Fällen eine Verringerung der erreichbaren Performance bedeuten.

1. Einmal angeforderter Speicher wird erst bei der Destruktion des Allokatorobjektes freigegeben. Dies würde sich verbieten, falls eine Map über die gesamte Lebenszeit der Anwendung benutzt werden würde. Der Einsatz in DROPS ist aber auf die Setup-Routinen beschränkt und die verwendeten Maps (und damit ihre Allokatoren) werden beim Verlassen der Routinen zerstört.

2. Es findet keine Fehlerüberprüfung statt. Falls kein Speicher mehr angefordert werden kann oder falls die von der Anwendung angeforderte Menge größer als die Größe eines Chunks ist, wird mit einem Nullzeiger weitergearbeitet. Letzteres ist beim DROPS Programm ausgeschlossen, für die erste Fehlerquelle gibt es in DROPS aber keinen gültigen Zustand mehr, so dass die Termination des Programms mit entsprechender Fehlermeldung eine sinnvolle Aktion ist.

3. Falls eine Anforderung vom aktuellen Chunk nicht mehr bedienbar ist, aber noch freier Speicher auf dem aktuellen Chunk zur Verfügung steht, wird dieser nicht mehr verwendet. Dies kann den Speicherbedarf des Programms deutlich erhöhen.

Ziel dieser Implementierung ist es nicht, Bibliotheken wie *libmtmalloc* zu ersetzen. Diese wenden neben der Block-Allokation noch weitere Techniken zur Erhöhung

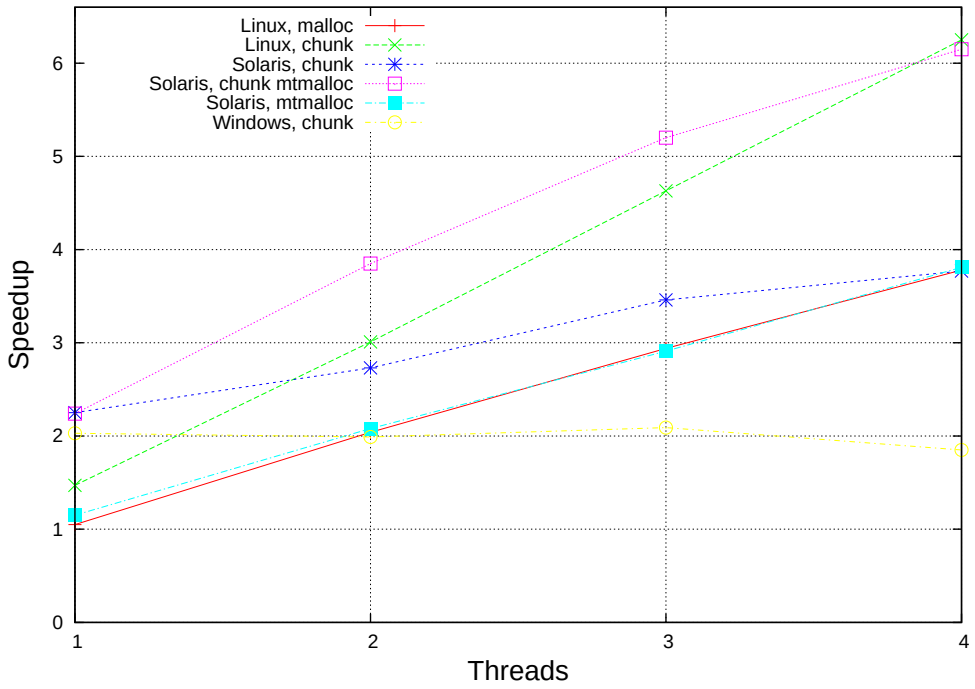


Abbildung 24: MAP-Benchmark: Speedup auf Sun Fire V40z mit *Chunk−Allokator*.

der Performance von parallelen Programmen an, z.B. werden mehrere Heaps verwendet und *False-Sharing* vermieden. *False-Sharing* tritt dann auf, wenn zwei oder mehrere Threads Daten im *shared* Speicher verändern und diese Daten so dicht beeinander liegen, dass sie auf dieselbe Cache-Line abgebildet werden. Auf ccNUMA-Architekturen wird eine Cache-Line bei Modifikation des zugehörigen Speichers als ungültig markiert, die später zugreifenden Threads müssen somit vor dem eigentlichen Update erst den Speicherbereich aktualisieren. Dies kann auf dem Verbindungssystem der Prozessoren zu einem hohen Verkehrsaufwand führen und damit die Skalierbarkeit begrenzen.

Die Implementierung von parallelen Heaps oder der Vermeidung von *False-Sharing* erfordert mehr Aufwand, als im Rahmen dieser Diplomarbeit möglich war. Zudem stehen bereits mehrere Bibliotheken mit dieser Funktionalität zur Verfügung. Das Hauptziel des *Chunk−Allokators* ist es also, die Anzahl der *malloc*()-Aufrufe zu reduzieren und er wird, soweit möglich, zusammen mit einer Bibliothek zur parallelen Speicherverwaltung verwendet.

Die Grafik in Abbildung 24 zeigt den erreichten Speedup des MAP-Benchmarks bei Verwendung des *Chunk−Allokators* gegenüber dem *Malloc−Allokator* unter Li-

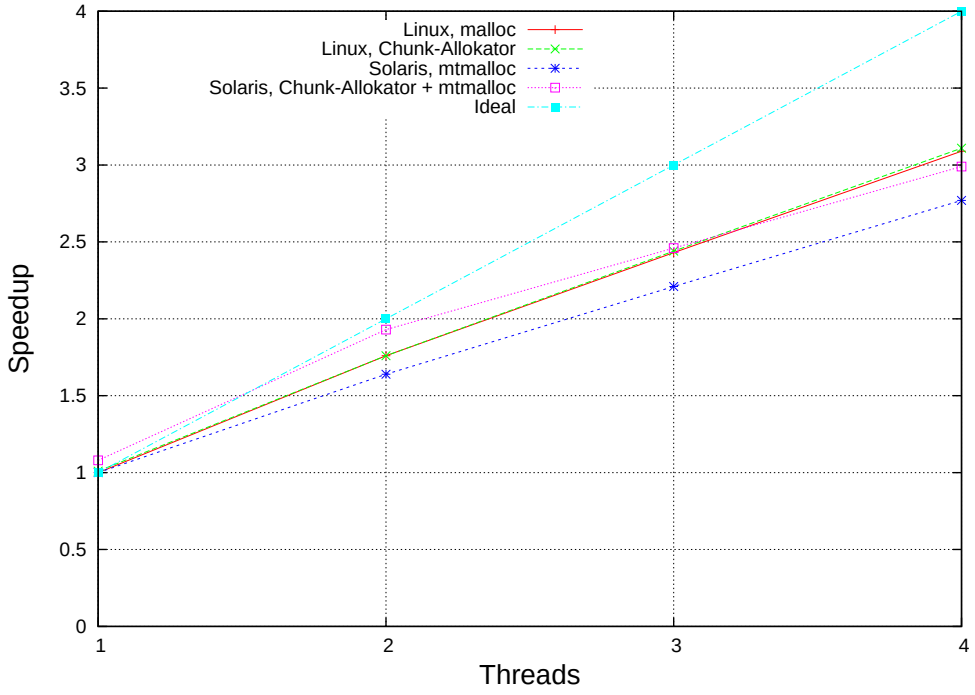


Abbildung 25: DROPS: Speedup *SetupSystem*1 auf Sun Fire V40z mit *Chunk−Allokator*.

nux. Unter Solaris wird der Speedup sowohl mit als auch ohne Benutzung der Bibliothek *libmtmalloc* dargestellt. Man kann erkennen, dass der *Chunk−Allokator* bei einer Chunk-Größe von 1 KB eine ähnlich gute Performance aufweist wie die Verwendung der *libmtmalloc* unter Solaris. Bei Anbindung der *libmtmalloc* ist noch eine weitere Steigerung des Speedups möglich. Dies ist damit zu begründen, dass in der parallelen Region im MAP-Benchmark ausschließlich Elemente in eine Map eingefügt werden, was nun weitestgehend ohne Synchronisation erfolgen kann.

Die Grafik in Abbildung 25 zeigt den erreichten Speedup der Setup-Routinen des DROPS Programms bei Verwendung von performancesteigernden Bibliotheken und mit und ohne Anwendung des *Chunk−Allokators*. Der Vorteil des *Chunk−Allokators* ist unter Linux sehr gering. Unter Solaris sieht man eine Steigerung der Performance. In den Setup-Routinen (hier: *SetupSystem*1) wird ein Array von Maps angelegt, und zwar eine *std*::*map* für jede Zeile in der zu erstellenden Matrix. Bei der Verfeinerungsstufe des Datensatzes, welcher bei den Zeitmessungen verwendet wurde, befinden sich in jeder Zeile der dünnbesetzten Matrix im Mittel nur ca. 30 Elemente. Der Einfügeaufwand in die Maps ist damit im Verleich zum sonstigen Rechenaufwand in den Routinen nicht dominant.

### 5.3.4 Paralleler Einsatz des Chunk-Allokators

Beim Einsatz in DROPS in obiger Implementierung wird für jede Map ein eigener Allokator angelegt. Um den Speicher besser auszunutzen und durch größere Chunks noch weniger mögliche Synchronisationspunkte zu haben, wäre es wünschenswert, nur einen Allokator zu verwenden. Eine Möglichkeit wäre es, die Variable *HeapManager* statisch zu deklarieren. Dadurch würde jede Instanz der Allokatorklasse auf eine einzige Instanz der HeapManager Klasse zugreifen und man könnte die Größe der Chunks entsprechend erhöhen.

Dies würde aber beim parallelen Einsatz zu Problemen führen. Wenn jede Map, somit auch z.B. zwei Maps von zwei unterschiedlichen Threads, auf dieselbe einzige Instanz von `HeapManager` zugreifen, kann das bei der Allokation von Speicher zu einer *Race-Condition* führen. Man könnte die entsprechenden Stellen in der Allokatorklasse durch kritische Regionen schützen, würde aber dadurch Synchronisationspunkte einführen, die gerade zu vermeiden sind.

Eine bessere Lösung ist es, die Variable *HeapManager* für jeden Thread zu privatisieren. Wie im Abschnitt 5.6 beschrieben wird, gibt es bei der Privatisierung von Klassen-Membern und bei der Privatisierung von Klassen ein paar Unzulänglichkeiten. Für die Privatisierung von *HeapManager* treten beide Fälle ein. Ein gangbarer Weg, der natürlich sowohl vom C++ Standard als auch vom OpenMP Standard her korrekt ist und auch mit den meisten Compilern und ihren OpenMP-Implementierungen funktioniert, wird nun beschrieben.

Die Variable *HeapManager* ist im File-Scope außerhalb der Klasse zu deklarieren. Falls die Headerdatei mit der Definition der Allokatorklasse nur von einer Quellcodedatei eingebunden wird (wie im Fall von DROPS), kann *HeapManager* auch innerhalb der Headerdatei deklariert werden. Falls die Headerdatei von mehreren Quellcodedateien eingebunden wird, muss sie in einer Quellcodedatei deklariert werden und in der Headerdatei als extern deklariert werden. Die nun entsprechend deklarierte Variable kann mit OpenMP als *threadprivate* deklariert werden, so dass jeder Thread eine eigene Instanz der Variable (Klasse) hat.

Variablen, die *threadprivate* deklariert wurden, werden wie folgt angelegt:

- Für den Master-Thread wird die Instanz beim Programmbeginn, vor Eintritt in die *main*-Funktion, angelegt. Diese Instanz wird im gesamten seriellen Teil des Programms benutzt und vom Master-Thread in den parallelen Regionen verwendet.

- Zu Beginn einer parallelen Region mit $n$ Threads werden $n-1$ Instanzen der Variable angelegt. Diese Instanzen bleiben auch nach dem Verlassen der parallelen Region über die gesamte Laufzeit des Programms und damit der Threads bestehen. Nach dem Verlassen der ersten Region werden beim Eintritt in die zweite parallele Region natürlich keine neuen Instanzen mehr angelegt.

Leider ist die OpenMP-Implementierung bei vielen Compilern in diesem Punkt nicht korrekt. Während der Intel C++ Compiler in Version 9 korrekt nach obiger Be-

schreibung arbeitet, wird beim Sun C++ Compiler in Version 11 nur für eine einzige Instanz der Konstruktor aufgerufen, alle anderen Instanzen bleiben uninitialisiert. Da dies für den Allokator kein gültiger Zustand ist, würde DROPS nicht korrekt arbeiten. Eine Lösung ist, die Variable *HeapManager* als Zeiger zu deklarieren und in einer parallelen Region zu Beginn des Programms jeden Thread diesen Zeiger auf eine neu erstellte Instanz der Speicherverwaltung (*HeapManager*) zeigen zu lassen. Dieser Weg wurde bei den DROPS Routinen gewählt.

Vor der Betrachtung der Performance müssen noch die Nachteile dieser Implementierung diskutiert werden.

1. Da die Variable *HeapManager* nicht mehr in der Klasse deklariert ist, wird das Interface verletzt. Damit ist gemeint, dass nun aus jedem Teil des Programms auf die Variable zugegriffen werden kann. Neben der Möglichkeit von Namenskonflikten besteht die Gefahr, dass (absichtlich) die Variable außerhalb der Maps verwendet wird. Dies ist in DROPS natürlich nicht der Fall, außer bei der Initialisierung, wo dies für die Verwendung mit verschiedenen Compilern notwendig ist.

2. Da die Instanzen von *HeapManager* jedes Threads über die gesamte Laufzeit des Programms bestehen bleiben, wird der Speicher nicht mehr freigegeben. Bei wiederholten Aufrufen der Setup-Routinen würde dies zu einer beträchtlichen Menge von allokiertem und nicht mehr verwendetem Speicher führen. In [Terboven & Spiegel$^+$ 05a] wurde beim seriellen Tuning die Reuse-Technik vorgeschlagen und auch in DROPS implementiert. Dadurch werden die Setup-Routinen nur dann genutzt, wenn sich die Struktur der Matrix verändert hat und ein vollständiger Neuaufbau notwendig ist. Hiermit werden die Maps nur wenige Male im Programm verwendet. Bei den zur Zeit verwendeten Datensätzen und Verfeinerungsstufen steht auf den in Frage kommenden Maschinen für Produktionsläufe genügend Speicher zur Verfügung, um diesen Kompromiss hinsichtlich der besseren Performance in Kauf nehmen zu können.

Die Performance des MAP-Benchmarks unter Verwendung dieser parallelen Version des *Chunk−Allokators* ist in der Grafik in Abbildung 26 dargestellt. Es wird die Laufzeit des MAP-Benchmarks unter Linux und Solaris für den *Chunk−Allokator* und für den parallelen *Chunk−Allokator* (*PChunk−Allokator*) dargestellt.

Unter Linux ist bei der Laufzeit für einen Thread noch ein deutlicher Vorteil zu sehen, der aber mit steigender Anzahl der Threads abnimmt. Dies liegt daran, dass für den *PChunk−Allokator*, der bei der Ausführung mit nur einem Thread äquivalent zu einer statischen Variable deklariert ist, nur eine einzige Konstruktion notwendig ist. Für den vorher betrachteten *Chunk−Allokator* ist bei der Änderung der internen Darstellung von *std*::*map*, falls dafür Speicher notwendig ist, eine Konstruktion über die `rebind`-Struktur notwendig.

Unter Solaris ist die Performance des PChunk-Allokators dem Chunk-Allokator unterlegen. Da nur eine einzige Konstruktion des Allokators stattfindet, und zwar im

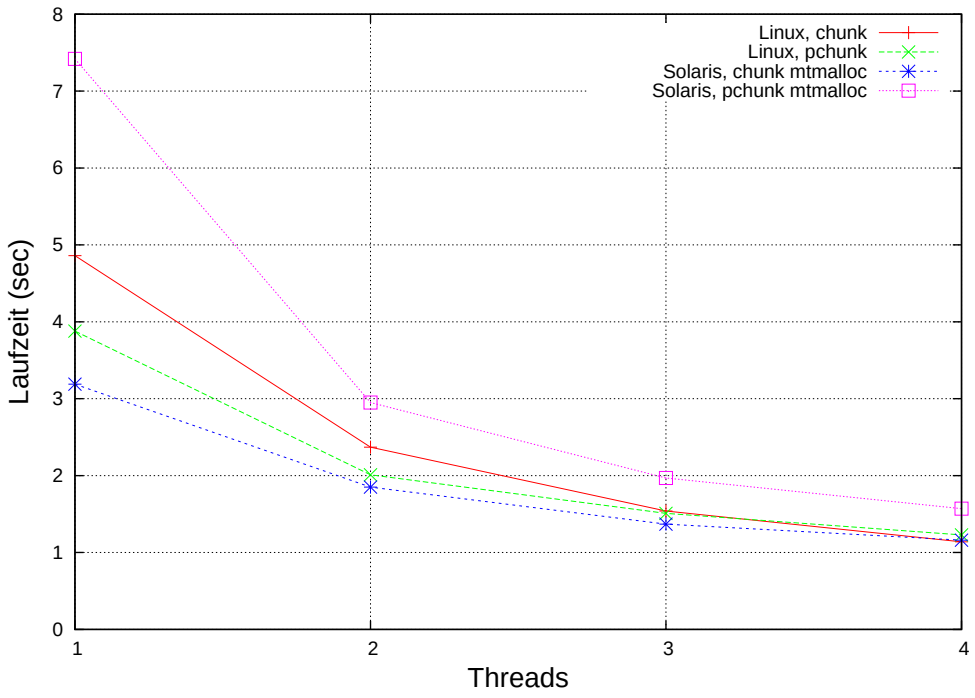


Abbildung 26: MAP-Benchmark: Laufzeit auf Sun Fire V40z mit *PChunk−Allokator*.

seriellen Teil des Programms vom Master-Thread, wird bei Benutzung der *libmtmalloc* nur ein Heap verwendet und die Vorteile der *libmtmalloc* gehen verloren.

Bei der Verwendung von vier Threads sind alle Versionen nahezu gleich schnell.

### 5.3.5 Ergebnis

Die Problematik der Allokation kleiner Objekte ist nicht ein speziell bei der C++ Programmiersprache auftretendes Problem, sondern ist z.B. auch in FORTRAN beim Anlegen automatischer Variablen in ähnlichem Umfang zu finden. Da die Verwaltung des physikalischen Speichers nicht von der Anwendung sondern vom Betriebssystem erledigt wird, sind Wege zur Verbesserung der Performance zuerst außerhalb der Anwendung zu suchen.

Der unter Linux zum Einsatz kommende Allokator ist bereits für die parallele Allokation von Speicher gut geeignet. Das Betriebssystem Solaris bietet mehrere Wege zur Allokation von Speicher an. Besonders für parallele Anwendungen ist die Bibliothek *libmtmalloc* gedacht, durch deren Einsatz die Skalierung auf das erwartete Niveau gehoben wurde. Die Speicherallokation unter Windows fällt im Vergleich zu den anderen beiden Betriebssystemen deutlich zurück. Auch die Verwendung eines anderen Heap-

managements und der *libhoard* Bibliothek konnte die Skalierung nicht verbessern.

Um eine gute Leistung insbesondere bei parallelen Programmen zu erreichen, muss die Anwendung entsprechend ausgelegt und ihre Speicherverwaltung entsprechend optimiert sein. In einigen Fällen bietet hier die Programmiersprache C++ Vorteile, wie am Beispiel das Daten *std*::*map* gezeigt wurde. Durch die Verwendung eines eigenen Allokators konnte die Anzahl der Speicheranforderungen an Bibliothek oder Betriebssystem verringert werden und die parallele Performance noch einmal erhöht werden.

## 5.4 Parallelisierung und Objektorientierung

In vielen C++ Programmen, in denen die Objektorientierung intensiv genutzt wird, liegt ein großer Teil der Berechnung innerhalb von Member-Funktionen von Objekttypen. Bei der Parallelisierung solcher Codes ergeben sich drei Möglichkeiten:

- Interne Parallelisierung: die parallele Region befindet sich komplett innerhalb einer Member-Funktion.
- Externe Parallelisierung: innerhalb der Member-Funktion wird mit *Orphaning* gearbeitet, die parallele Region befindet sich außerhalb des Objektes, es können also aufeinanderfolgende Aufrufe der Member-Funktion innerhalb einer parallelen Region vorkommen.
- Keine Parallelisierung: innerhalb der Member-Funktion befinden sich keine Konstrukte der Parallelisierungstechnik. Falls eine solche Member-Funktion in einer parallelen Region aufgerufen wird, muss sie *reentrant* oder anderweitig *threadsafe* sein.

Anhand des im Programmausschnitt 25 dargestellten Auszugs des PCG-Lösers aus DROPS werden die beiden Ansätze interne und externe Parallelisierung verglichen: bei der internen Parallelisierung ist die gesamte OpenMP Parallelisierung in den Operatoren verborgen, bei der externen Parallelisierung ist die OpenMP Parallelisierung in den Originalcode eingefügt. Die Vergleichskriterien sind zum einen der Aufwand bei der Parallelisierung, zum anderen die erreichbare Performance. Um diese zu messen, wurde zum einen die Dimension ($n$) der Vektoren groß gewählt, zum zweiten die Routine mehrere Male hintereinander ausgeführt.

Zur Demonstration sollen die Zeilen 8 bis 11 näher betrachtet werden. Die Variablen $x$, $r$, $p$ und $q$ sind Vektoren vom Typ *Vec* und haben die Dimension $n$, $alpha$ und *roh* sind skalare Variablen vom Typ *double* die vorher berechnet wurden. Der Operator $*$ liegt in zwei Versionen vor. Zum einen als Skalarprodukt von zwei Vektoren (*Vec*), zum zweiten als elementweise Skalierung eines Vektors. Die Operatoren $+=$ und $-=$ sind in der Klasse *Vec* definiert und führen eine elementweise Addition bzw. Subtraktion des rechten Operanden auf den linken Operanden durch.

Vor der Betrachtung der Parallelisierung ist noch auf die serielle Performance dieses Codes einzugehen. Für die Programmierung von objektorientierten Codes, die vom

**Programmausschnitt 25** PCG-Kernel.

```
1 PCG(const Mat& A, Vec& x,const Vec& b, ...)
2 {
3     Vec p(n), z(n), q(n), r(n);
4     [...]
5     for (int i = 1; i <= max_iter; ++i)
6     {
7         [...]
8         q = A * p;
9         double alpha = rho / (p * q);
10        x += alpha * p;
11        r -= alpha * q;
12        [...]
13    }
14    [...]
15 }
```

Compiler performant umgesetzt werden sollen, ist vom C++ Programmierer einige Erfahrung notwendig. Die Lernkurve ist in diesem Bezug in C++ vielleicht steiler als bei anderen Programmiersprachen, obwohl zur Kombination von flexiblen und eleganten Codes und einer hohen Performance sicherlich immer ein gewisser Aufwand betrieben werden muss. Um komplexe Ausdrücke mit hohem Abstraktionsniveau, wie dies im obigen Code durch die Überladung von Operatoren und die Verwendung mit Matrizen und Vektoren der Fall ist, effizient umzusetzen, wird in entsprechend ausgelegten Klassenbibliotheken mit sog. *Template Expressions* gearbeitet. Diese wurden in [Veldhuizen 95] als *Expression Templates* vorgestellt. Dadurch wird es möglich, komplexe Ausdrücke auf einfache Schleifen abzubilden, welche die Operationen direkt implementieren und somit die Erzeugung von temporären Variablen minimieren. Das Anlegen von temporären Objekten vom Typ $Vec(n)$ ist bei großem $n$ bereits eine teure Operation.

Während aktuelle Versionen der STL bereits mit der Technik der *Template Expressions* implementiert wurden und darum im Vergleich zu direkten, effizienten Implementierungen in C der Datentyp *std*::*valarray* keine Nachteile in der Performance in Kauf nehmen muss, ist bei vom Benutzer implementierten Datentypen einiger Aufwand nötig. Da eine Behandlung von Optimierungstechniken von High-Level Code in C++ im Rahmen dieser Diplomarbeit nicht vorgesehen war, soll die Wichtigkeit nur an einem kurzen Beispiel dargestellt werden. Im Programmausschnitt 26 sind zwei Implementierungen des $+=$ Operators für *Vec* dargestellt.

In beiden Implementierungen wurde darauf geachtet, dass die Argumente nicht unnötig kopiert werden müssen und es wird mit Referenzen gearbeitet. Bei einer Referenz wird ein Zeiger auf das Objekt übergeben, mit dem aber wie mit einer echten

**Programmausschnitt 26** Operator + = der Klasse *Vec*.

```
1  // temporary vector "r" used
2  S_Vector<T> operator+=(const S_Vector<T>& op1,
3                         const S_Vector<T>& op2)
4  {
5    const size_t rows = op1.num_rows();
6    S_Vector<T> r(rows);
7  #pragma omp parallel for
8    for (size_t row = 0; row < rows; row++)
9    {
10     r.element(row) = op1.element(row) + op2.element(row);
11   }
12   return r;
13 }
14
15 // no temporary vector
16 S_Vector<T>& operator+=(const S_Vector<T>& op1)
17 {
18   const size_t rows = op1.num_rows();
19 #pragma omp parallel for
20   for (size_t row = 0; row < rows; row++)
21   {
22     _data[row] += op1.element(row);
23   }
24   return *this;
25 }
```

Kopie des Objektes gearbeitet werden kann. Da die Argumente aber nicht verändert werden und auch nicht verändert werden dürfen, sind die Referenzen zusätzlich noch als *const* deklariert, so dass der Compiler Änderungsversuche schon zur Compilezeit nicht akzeptiert.

Die erste Implementierung, von Zeile 2 bis Zeile 13, verwendet die temporäre Variable *r* vom Typ *Vec* mit der vollen Dimension, um das Ergebnis zu berechnen und zurückzugeben. Diese Art der Implementierung ist in älteren C++ Bücher häufig zu finden. Der Aufwand des Anlegens eines temporären Vektors und das Kopieren des Ergebnisses ist nicht notwendig und bei einer solchen Implementierung kann nur gehofft werden, dass dies vom Compiler erkannt und vermieden wird. In modernen Compilern ist das bereits teilweise der Fall. Die zweite Implementierung, von Zeile 16 bis Zeile 25, kommt ohne temporäre Variable aus. Sie ist als Klassen-Member implementiert und benutzt auch für das Ergebnis eine Referenz, die natürlich nicht als *const* deklariert ist, weil sie verändert wird. Es sind keine Kopieroperationen notwendig, da direkt mit den Eingabedaten und dem Ergebnis über den Zeiger der Referenz gearbeitet wird.

### 5.4.1 Interne Parallelisierung

Wie bereits oben beschrieben, befindet sich bei der internen Parallelisierung eine abgeschlossene parallele Region innerhalb der Memberfunktionen von Objekten. Für den Operator $+=$ wurde dies bereits in Abbildung 26 gezeigt, die parallele Region beginnt in Zeile 19 und endet in Zeile 23.

Der Vorteil dieser Implementierung liegt vor allem darin, dass die Parallelität innerhalb des Objektes abgeschlossen ist und die Benutzerschnittstelle sich gegenüber einem seriellen Objekt nicht verändert hat. Es wäre darüber hinaus einfach möglich, das Interface dahingehend zu erweitern, dass die Angabe der Anzahl von zu benutzenden Threads optional möglich ist.

Der Nachteil dieser Art der Implementierung liegt allerdings ebenfalls in der Abgeschlossenheit. Zu Beginn jeder parallelen Region werden die Threads erzeugt und am Ende jeder parallelen Region terminiert. Auch wenn die aktuellen Implementierungen von OpenMP dies über Thread-Pools effizient verwalten können und die Threads nur schlafen gelegt werden, stellt es dennoch einen gewissen Aufwand dar. Falls mehrere Aufrufe von parallelen Memberfunktionen aufeinander folgen, wie dies im betrachteten Beispielcode der Fall ist, kann dieser Aufwand durch eine andere Implementierung verringert werden.

In der Literatur sind mehrere Ansätze zur Datenparallelisierung in der STL zu finden, die auch zur Kategorie der internen Parallelisierung zu zählen sind. Als Beispiele dazu können [An & Jula$^+$ 01] und [Fletcher & Sankaran 03] genannt werden. In den in diesen beiden Papern vorgestellten Ansätzen wird die STL so erweitert, dass die Nutzung von Parallelität für den Benutzer nahezu keinen Mehraufwand bedeutet.

Dabei wird der Ansatz des generischen Programmierens ausgenutzt, dass auch Algorithmen eine natürliche Abstraktion besitzen und für verschiedene Datentypen nur eine Implementierung erstellt werden muss. Die STL bietet bereits eine Menge solcher Algorithmen an, z.B. zum Sortieren von Mengen oder zur elementweisen Anwendung einer Operation auf eine Menge.

Bei Verwendung der vorgestellten Ansätze kann nun anstelle der von der STL bereitgestellten Implementierungen der Algorithmen eine parallele Implementierung verwendet werden. Die Umsetzung der Parallelisierung erfolgt in weiten Teilen für den Benutzer unsichtbar und basiert auf einem Laufzeitsystem, das in den meisten Fällen sowohl Distributed-Memory Systeme als auch Shared-Memory Systeme unterstützen kann.

Der Aufwand für die Verwendung dieser Systeme ist gering. Durch einen bereitgestellten Präprozessor oder durch die manuelle Auswahl entsprechender Container und Algorithmen wird die parallele STL Implementierung verwendet. Da diese eine kompatible Benutzerschnittstelle anbietet, muss dabei der restliche Code im Allgemeinen nicht angepasst werden. Durch die Konzentration auf die STL und die darin angebotenen Datentypen sind die Einsatzmöglichkeiten aber stark begrenzt. Die im HPC-Umfeld zu findenden C++ Programme verwenden zwar oft recht intensiv die in der

STL angebotenen Container, allerdings ist der Laufzeitanteil, der in generischen Algorithmen der STL liegt, meist vernachlässigbar. Daher bieten die beiden vorgestellten Ansätze keine Möglichkeiten für die Parallelisierung der hier betrachteten Programme DROPS, FIRE und ADI.

### 5.4.2 Externe Parallelisierung

Bei der externen Parallelisierung befindet sich die parallele Region außerhalb der Memberfunktionen der Objekte. Für den Fall des PCG-Lösers aus Programmausschnitt 25 bedeutet dies, dass die parallele Region mindestens den Bereich von Zeile 6 bis Zeile 13 umfasst. Die parallele Region kann sogar noch hinter die `for`-Schleife in Zeile 5 gezogen werden, allerdings müssen die Iterationen natürlich aufeinanderfolgen, so dass dort kein $for$-Worksharingkonstrukt verwendet werden darf.

Innerhalb der Memberfunktionen, z.B. des $+=$ Operators, werden nun keine parallelen Regionen eröffnet, sondern es wird mit *Orphaning* gearbeitet. Dazu ist die Zeile 19 aus Programmausschnitt 25 durch **#pragma** omp **for** zu ersetzen. Der Vorteil hiervon ist, dass die Threads über einen längeren Zeitraum bestehen bleiben und damit eine bessere Skalierbarkeit erreicht wird.

Auf der anderen Seiten gibt es aber einen Nachteil dieser Implementierung bei falscher Verwendung. Auch wenn die Memberfunktionen mit den Direktiven aus einem seriellen Programm aufgerufen werden, ist nach der OpenMP Spezifikation eine korrekte Berechnung gesichert. Falls die Memberfunktionen mit den verweisten Direktiven aber aus einem *single*-Worksharingkonstrukt aufgerufen werden, wird in den betrachteten Implementierungen nur ein Teil der verteilten Arbeit der `for`-Schleife berechnet, nämlich genau die Arbeit von nur einem Thread. Die OpenMP Spezifikation bietet keine Möglichkeit, einen solchen Aufruf innerhalb der Memberfunktionen festzustellen, da es sich bei *single* um ein Worksharingkonstrukt handelt und z.B. $omp\_get\_num\_threads()$ die Anzahl der Threads des Teams zurückgibt und nicht eins. Ebenso wird angezeigt, dass man sich in einer parallelen Region befindet. Bei einer solchen Implementierung muss der Programmierer sicherstellen, dass der Code korrekt aufgerufen wird.

Neben dem vorgestellten Ansatz gibt es noch eine weitere Möglichkeit, die auch zur externen Parallelisierung gezählt werden kann. Dabei wird die parallele Region, wie bereits beschrieben, möglichst weit nach außen gezogen. Die Parallelisierung findet nun aber nicht in den Memberfunktionen statt, sondern der Code der Memberfunktionen wird in die `for`-Schleife gestellt. Danach befinden sich alle Worksharingkonstrukte in der `for`-Schleife. Der Vorteil hiervon ist, dass dadurch parallele Bereiche zusammengefasst werden können und damit Synchronisationspunkte entfallen. Am Beispiel des PCG-Lösers aus Abbildung 25 können die Zeilen 10 und 11 so zusammengefasst werden, dass sie in einem $for$-Worksharingkonstrukt bearbeitet werden, da sie über die gleichen Indizes laufen.

Die Nachteile sind die, dass einerseits zum einen einige Eingriffe in den Code

Tabelle 12: DROPS: Performance von *PCG* auf Sun Fire V40z unter Linux.

| **Version** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|
| Intern | 219 | 108 | 83,2 | 65,5 |
| Intern (temp. Vektor) | 218 | 108 | 82,5 | 65,1 |
| Extern | 216 | 107 | 82,3 | 65 |
| C-Code | 216 | 107 | 82,3 | 65 |

Tabelle 13: DROPS: Performance von *PCG* aud Sun Fire E6900 unter Solaris.

| **Version** | **1** | **2** | **3** | **4** | **6** | **8** |
|---|---|---|---|---|---|---|
| Intern | 375 | 185 | 156 | 112 | 68 | 42 |
| Intern (temp. Vektor) | 427 | 211 | 178 | 128 | 78 | 47 |
| Extern | 320 | 161 | 141 | 97 | 61 | 37 |
| C-Code | 315 | 157 | 138 | 95 | 57 | 35 |

notwendig sind, zum zweiten das objektorientierte Konzept gebrochen wird. Dadurch geht viel Flexibilität verloren und Code muss an mehreren Stellen gepflegt werden. Dennoch wird mit diesem Ansatz die beste Performance erreicht.

### 5.4.3 Ergebnis

Das Einsparen der Synchronisationspunkte aus der zuletzt vorgestellten Version durch das Aufbrechen des objektorientierten Konzeptes wäre auch in der ersten Version der externen Parallelisierung teilweise dadurch möglich gewesen, dass die verweisten Direktiven den Zusatz *nowait* erhalten. Dabei hätte der Programmierer aber sicherstellen müssen, dass an den notwendigen Stellen Barrieren eingefügt werden. Dennoch ist dabei das Zusammenfassen von Schleifen nicht möglich. In der aktuellen OpenMP Spezifikation ist darüber hinaus nicht sichergestellt, dass bei einem statischen Schedule die Threads die gleichen Abschnitte bezüglich der Indizes in zwei aufeinanderfolgenden Schleifen erhalten, so dass die Korrektheit von der Implementierung abhängig wäre.

Es muss also in vielen Fällen eine Wahl zwischen erreichbarer Performance und Beibehaltung der Objektorientierung getroffen werden. Die Programmierung wie in der Programmiersprache C hat bei diesem Code zur besten Performance geführt und wurde deshalb in DROPS angewandt. Dies liegt zum Teil auch daran, dass die Optimierer in den Compilern diese Art der Programmierung besser umsetzen können. Allerdings zeigen die Messungen mit dem Intel C++ Compiler unter Linux, dass die Compiler bereits sehr gut gewisse High-Level Codes umsetzen können. Im Fall von DROPS, wo noch weitere Sprachelemente und Parameter verwendet werden, war dies allerdings noch nicht immer erfolgreich.

## 5.5 Parallelisierung von nicht OpenMP konformen Schleifen

In C++ Programmen, insbesondere bei der Verwendung von Datentypen aus der STL oder anderen objektorientierten Bibliotheken, sind Schleifen in vielen Fällen nicht als `for`-Schleife über eine Integer Variable implementiert. Damit eine Schleife in OpenMP parallelisiert werden kann, stellt die OpenMP Spezifikation [Board 05] folgende Ansprüche an eine Schleife:

1. Sie muss von folgender kanonischer Form sein:
   **for** ( initExpr ; var relationalOp b; incrExpr ) statement
   wobei *initExpr* entweder eine Zuweisung an die Schleifenvariable oder eine Deklaration der Schleifenvariable mit Zuweisung ist, *relationalOp* eine relationale Operation und *incrExpr* eine Addition oder Subtraktion eines schleifenunabhängigen Wertes ist.

2. Die Variable *var* ist eine vorzeichenbehaftete Integer-Variable wie in der Programmiersprache definiert und wird implizit privatisiert. Diese Variable darf in der Schleife nicht außerhalb *incrExpr* verändert werden.

3. Die Schleife darf nicht durch *break* verlassen werden.

4. Falls die relationale Operation $<$ oder $<=$ ist, muss die Schleifenvariable in jeder Iteration erhöht werden. Entsprechend muss sie erniedrigt werden, wenn die relationale Operation umgekehrt ist.

Die geforderten Bedingungen garantieren, dass die Anzahl der Iterationen zu Beginn der Schleife berechnet werden können. Nicht-Integer Schleifen, die obige Bedingungen in mindestens einem Punkt verletzen, resultieren in der Programmiersprache C recht häufig aus der Zeigerarithmetik. In reinen C++ Programmen wird diese Technik eher selten angewandt. Allerdings hat die STL mit dem Konzept der Iteratoren eine Abstraktion eines Zeigers auf ein Element einer Sequenz eingeführt. Die Iteratoren implementieren die notwendige Funktionalität, um in einer Schleife verwendet werden zu können.

Zwar bieten Iteratoren über die Funktion *distance*() eine Möglichkeit zur Berechnung der Anzahl der Iterationen vor der Schleife (außer bei Output-Iteratoren). Dies wird von der aktuellen OpenMP Spezifikation aber nicht unterstützt. Dazu ist noch anzumerken, dass der Aufruf der Funktion *distance*() je nach Iteratortyp einen aufwändigen Durchlauf der gesamten Sequenz zur Berechnung der Distanz zur Folge haben kann.

Ein Beispiel für eine Vorkommen einer Schleife mit Iteratoren sind die Setup-Routinen in DROPS. Der Schleifenrumpf ist im Programmausschnitt 27 dargestellt.

Die Schleifenvariable ist hier *sit*, sie wird in Zeile 2 initialisiert und in Zeile 4 befinden sich Abbruchbedingung und Erhöhungsanweisung, die am Ende einer jeden Schleifeniteration ausgeführt werden. Die Variable *sit* erfüllt nicht die oben beschriebenen Bedingungen, so dass eine direkte Parallelisierung mit OpenMP nicht möglich ist. Zur Parallelisierung einer solchen Schleife wurden drei Möglichkeiten betrachtet:

**Programmausschnitt 27** DROPS: Iterator-Schleife aus *SetupSystem*1.

```
1 for (MultiGridCL::const_TriangTetraIteratorCL
2       sit = _MG.GetTriangTetraBegin(lvl),
3       send = _MG.GetTriangTetraEnd(lvl);
4       sit != send; ++sit)
5 {
6   [...]
7 }
```

**Programmausschnitt 28** ITERLOOP: Experiment 1.

```
1 long l = 0, lSize = 0;
2 for (it = list1.begin(); it != list1.end(); it++)
3   lSize++;
4 valarray<CComputeItem*> items(lSize);
5 for (it = list1.begin(); it != list1.end(); it++) {
6   items[l] = &(*it); l++;
7 }
8 #pragma omp parallel for default(shared)
9 for (long l = 0; l < lSize; l++) {
10  items[l]->compute();
11 }
```

1. Die Zeiger der Iteratoren werden in einer zusätzlichen Schleife mit den Grenzen der Originalschleife in einem Array gespeichert. Damit kann eine einfache Schleife über dieses Array mit dem OpenMP *for*-Worksharingkonstrukt parallelisiert werden. Dies ist im Programmausschnitt 28 dargestellt.

2. Intel hat in [Su & Tian+ 02] das sog. *Workqueuing*-Modell als Erweiterung von OpenMP vorgeschlagen und in [Shah & Haab+ 99] die Implementierung im Intel C++ und Guide C++ Compiler beschrieben. Für jeden Wert der Schleifenvariable wird hierbei der Schleifenrumpf in die Arbeitsschlange eingefügt. Die Arbeitsschlange wird von allen Threads abgearbeitet. Diese Implementierung ist im Programmausschnitt 29 dargestellt.

3. Die `for`-Schleife wird in eine parallele Region eingebettet, wobei der Schleifenrumpf in einem *single*-Worksharingkonstrukt steht, dessen implizite Barriere durch Angabe von *nowait* ausgelassen wird. Dies ist im Programmausschnitt 30 dargestellt.

Während auf den Sun Fire V40z Systemen unter Linux mit dem Intel C++ Compiler alle drei Möglichkeiten umsetzbar sind, unterstützt der Sun C++ Compiler auf

**Programmausschnitt 29** ITERLOOP: Experiment 2.

```
1 #pragma intel omp parallel taskq
2 {
3 for (it = list2.begin(); it != list2.end(); it++) {
4 #pragma intel omp task
5 {
6   it->compute();
7 }
8 } // end for
9 } // end omp parallel
```

**Programmausschnitt 30** ITERLOOP: Experiment 3.

```
1 #pragma omp parallel private(it)
2 {
3 for (it = list3.begin(); it != list3.end(); it++) {
4 #pragma omp single nowait
5 {
6   it->compute();
7 }
8 } // end for
9 } // end omp parallel
```

den Sun Fire E6900 Systemen das *Workqueueing* nicht. Es soll nun untersucht werden, welches der drei Verfahren zu bevorzugen ist. Dazu dienen zum einen die Setup-Routinen von DROPS, zum anderen aber der synthetische Benchmark *ITERLOOP*. Als Experimente werden darin die drei betrachteten Möglichkeiten der Parallelisierung von nicht OpenMP konformen Schleifen durchgeführt, deren Implementierung bereits dargestellt wurde.

In diesem Benchmark kann zum einen durch den Parameter *DIMENSION* die Anzahl der Listeneinträge und damit die Schleifenlänge gesteuert werden, zum zweiten kann durch den Parameter *ITERATIONS* der Aufwand der Berechnung für jede Schleifeniteration gesteuert werden. Hiermit wird untersucht, unter welchen Bedingungen sich welche Implementierung anbietet.

Zunächst wird die Sun Fire V40z Architektur unter Linux mit dem Intel C++ 9.0 Compiler betrachtet, wo alle drei Techniken unterstützt werden. Generell ist festzustellen, dass sich hier die drei Arten der Schleifenparallelisierung nur in geringem Maße bezüglich der Performance unterscheiden. Als erstes wird der Parameter *ITERATIONS* auf 1000 gesetzt, was einem Rechenaufwand von ca. 1 Sekunde für jede Schleifeniteration entspricht, und mit dem Parameter *DIMENSION* wird die Anzahl der Schleifendurchläufe von 5 bis 320 variiert. Wie in der Grafik in Abbildung 27 zu sehen ist, liefert Methode 1, also das Speichern der Zeiger der Iteratoren in einem Feld, die besten Ergebnisse und ist ungefähr 3% schneller als Methode 2, das *Workqueueing*. Methode 3, die *Single-Nowait*-Technik, liegt von der Performance her zwischen den beiden anderen Techniken. Wird der Parameter *ITERATIONS* auf 1 gesetzt, so dass nahezu keine Arbeit innerhalb der Schleife verrichtet werden muss, und der Parameter *DIMENSION* von 10 bis 1280 variiert, dann ergeben sich keine Unterschiede.

Die gezeigten Ergebnisse entsprechen den Erwartungen. Der Ansatz mit Verwendung der *Single-Nowait*-Technik hat im Vergleich zu den anderen beiden Methoden einen höheren Aufwand, der durch die Abstimmung der Threads am *SINGLE*-Worksharingkonstrukt entsteht. Dadurch fällt diese Version in Bezug auf die Performance leicht zurück. Ebenfalls etwas langsamer ist die *Workqueuing*-Technik, da auch hier ein administrativer Aufwand notwendig ist. Eine Verbesserung für diese Version würden Möglichkeiten zur Angabe eines Scheduling bzw. die Anzahl der von einem Thread zu bearbeitenden Aufgaben bringen.

Auf der Sun Fire E6900 Architektur ergibt sich ein ähnliches Bild, wenn sich innerhalb der Schleife Arbeit befindet (Parameter *ITERATIONS* = 1000). Die hier zur Verfügung stehenden Varianten 1 und 3 liegen aber dichter beeinander als beim Intel Compiler. Ein deutlicher Unterschied ergibt sich allerdings, wenn sich innerhalb der Schleife kaum Arbeit befindet und die Anzahl der Schleifeniterationen sehr hoch ist. Dies ist auch in Abbildung 27 dargestellt. Hier fällt die *Single-Nowait*-Technik wegen ihres Aufwands am *SINGLE*-Worksharingkonstrukt um einen Faktor von ungefähr 10 hinter dem Speichern der Zeiger der Iteratoren zurück.

Als Ergebnis lässt sich Methode 1, das Speichern der Zeiger der Iteratoren in einem Array, welches anschließend parallel abgearbeitet wird, empfehlen. Sie liefert auf

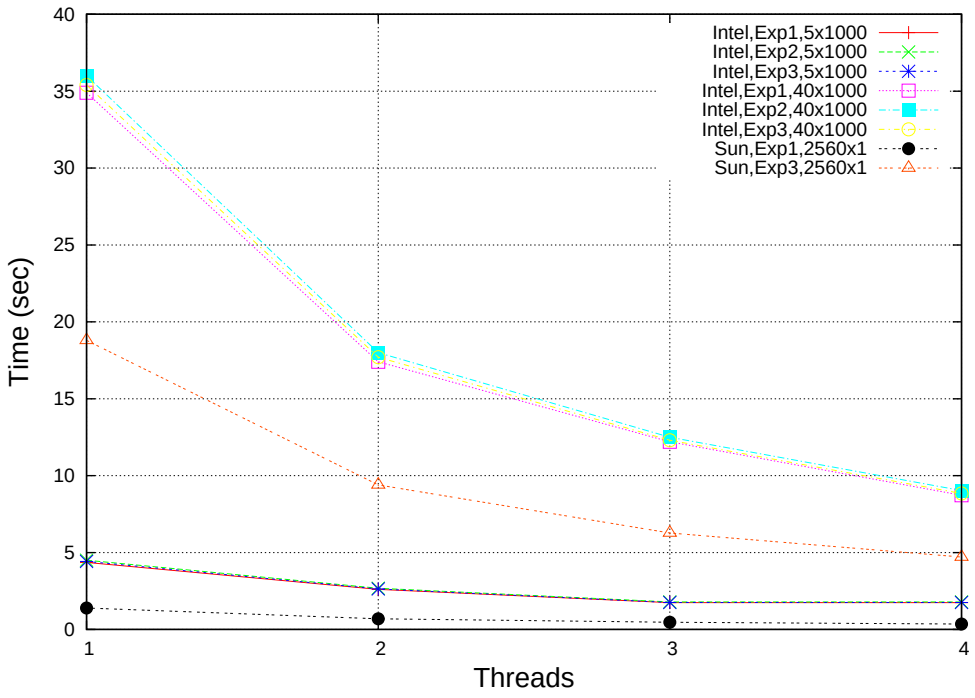


Abbildung 27: ITERLOOP: Laufzeit auf Sun Fire V40z und Sun Fire E6900.

den betrachteten Plattformen die besten Ergebnisse. Als Nachteil ist hierbei aber zu sehen, dass Modifikationen im Code notwendig werden, die vielleicht nicht als elegant zu bezeichnen sind. Die von Intel vorgestellte *Workqueuing*-Technik mit erweiterten Möglichkeiten zur Angabe des Scheduling würde diesbezüglich Abhilfe schaffen und die gleiche Performance erreichen. Zwar liegen dem OpenMP Architecture Review Board entsprechende Vorschläge zur Standardisierung vor, allerdings ist eine Aufnahme in die nächste OpenMP Spezifikation zum jetzigen Zeitpunkt noch nicht sicher.

## 5.6 Kritische Betrachtung der OpenMP Spezifikation

Insgesamt wurde festgestellt, dass OpenMP leistungsfähige Mittel zur Parallelisierung bereitstellt, sowohl zur Parallelisierung auf hoher Ebene in FIRE, als auch auf tieferer Ebene und Schleifenebene in DROPS. Zu den Zielen von OpenMP gehört, dass bereits mit einigen wenigen Direktiven die Parallelisierung erfolgen kann und es dem Benutzer freigestellt ist, z.B. durch die Angabe von Parametern für das Scheduling die Skalierbarkeit zu verbessern. Dieser eher minimalistische Ansatz führt auch dazu, dass neue Fähigkeiten nur langsam in die Spezifikation aufgenommen werden. Dies

hat in der Vergangenheit dazu geführt, dass neue Funktionen schon sehr bald nach der Veröffentlichung der Spezifikation bereits in die Compiler integriert wurden.

Im vorigen Abschnitt wurde bereits diskutiert, wie Schleifen, die über Zeiger oder Iteratoren laufen, in OpenMP parallelisiert werden können. Aber auch einfache Schleifen über Integer Variablen entsprechen nicht der geforderten Form, falls die Schleifenvariable z.B. kein Vorzeichen besitzt. In C und C++ Programmen wird in vielen Fällen der Datentyp *size_t* zur Zählung von Elementen eingesetzt, deren Anzahl größer oder gleich null ist. Aus diesem Grund besitzt der Datentyp *size_t* kein Vorzeichen, um eine größere Anzahl von Elementen darstellen zu können. Zum Beispiel in FIRE ist die Anzahl der Bilder in der Datenbank immer positiv und darum ist die Schleifenvariable in *retriever.cpp* vom Typ *size_t*. Bei der Parallelisierung darüber melden sowohl der Sun C++ Compiler als auch der Intel C++ Compiler einen Fehler, dass die Schleife nicht der kanonischen Form entspricht und akzeptieren den Code nicht. Aus diesem Grund mussten die Schleifenvariablen in *long* geändert werden. In dem hier betrachteten Fall stellte dies keinen nennenswerten Aufwand dar, allerdings sind generell drei Punkte dazu anzumerken:

- In vielen Stellen in DROPS wurde ebenfalls der Typ *size_t* verwendet und nach *long* geändert. Dort waren insbesondere in den Lösern viele Stellen betroffen. Dazu wurden ebenfalls die oberen Schleifengrenzen als Wert vom Typ *size_t* von Memberfunktionen von STL-Datentypen zurückgegeben, so dass Casting-Operatoren eingefügt werden mussten.

- Insbesondere auf 64bit Architekturen ist es unwahrscheinlich, dass bei der Umstellung von *size_t* auf *long* durch den verkleinerten Darstellungsbereich Fehler im Code entstehen. Dennoch muss der Code auf mögliche Probleme und Überläufe untersucht werden.

- Der PGI C++ Compiler akzeptiert nur Schleifen mit Indexvariablen vom Typ *int* zur Parallelisierung. Aus diesem Grund kann damit der DROPS Code nicht parallel kompiliert werden.

Die Gründe für die Aufnahme der Vorzeichenbedingung in die OpenMP Spezifikation sind nicht dokumentiert. Es kann vermutet werden, dass mit vorzeichenbehafteten Indexvariablen die Berechnung der Schleifeniterationen für jeden Thread etwas einfacher zu implementieren ist. Allerdings würde eine vorzeichenlose Variable keine Verletzung der Forderung darstellen, die Anzahl der Schleifeniterationen vorab berechnen zu können. Die Änderung der Implementierung in den Compilern sollte nur von geringem Aufwand sein, während ein Verzicht auf die Vorzeichenbedingung den Programmierer in einigen Fällen entlastet und zu mehr Komfort führt.

In der OpenMP Spezifikation wird leider nicht detailliert auf die Privatisierung von Klassenvariablen eingegangen. Folgende Aussage ist zu finden: *Data-sharing attribute clauses apply only to variables whose names are visible in the construct on which the*

*clause appears, [...]*. Die betrachteten Compiler unterstützen eine Privatisierung von Membervariablen in C++ nicht, wobei die Fehlermeldungen teilweise den Grund als in der Spezifikation begründet beschreiben, teilweise der Benutzer aber erst einen Fehler beim Linken erhält. Dies ist für mit OpenMP noch unerfahrene Programmierer nicht immer hilfreich.

Es gibt aber Fälle, in denen die Privatisierung von Klassenvariablen sinnvoll ist. Bei der Parallelisierung der in dieser Arbeit betrachteten Programme wurde gegen die Parallelisierung auf tieferen Ebenen, z.B. in von STL-Datentypen abgeleiteten Klassen, entschieden. Dort wäre aber eine solche Privatisierung notwendig geworden, ebenfalls auch im $Chunk-Allokator$. Da die Compiler dies zur Zeit nicht unterstützen, muss mit lokalen Variablen (Kopien) gearbeitet werden die gegebenenfalls zu initialisieren sind, oder aber das Klassendesign muss verändert werden.

In der OpenMP Spezifikation gibt es keine Möglichkeit zu erkennen, ob eine Routine aus einem Worksharingkonstrukt aufgerufen wird. Falls eine Routine weitere Worksharingkonstrukte enthält, z.B. ein $for$-Worksharingkonstrukt, wird die Arbeit nicht korrekt verteilt, falls diese Routine z.B. aus einem *single*-Konstrukt aufgerufen wurde. In den betrachteten Implementierungen wird dann bei einem statischen Scheduling genau nur der Teil der Arbeit berechnet, der dem Anteil eines Threads entspricht.

Der Aufruf von $omp\_get\_num\_threads()$ liefert die Anzahl der Threads im Team der umgebenden parallelen Region und nicht z.B. eins. Ebenfalls liefert der Aufruf von $omp\_get\_parallel()$ den Wert $true$ zurück. Dies erschwert die Implementierung von Bibliotheken, da durch *Orphaning* der Aufwand der Erzeugung von neuen Threads und ihre anschließende Beendigung erspart werden kann, wenn sich die parallele Region über einen größeren Bereich erstreckt. Ein Beispiel für diese Problematik wurde in Abschnitt 5.4 bei der externen Parallelisierung gegeben. In diesem Fall muss der Programmierer sicherstellen, dass der Code nur unter korrekten Vorbedingungen aufgerufen wird. Allerdings verbietet die OpenMP Spezifikation das Schachteln von Worksharingkonstrukten in einer parallelen Region.

Die OpenMP Spezifikation sagt, dass eine Variable, die als $private$ in einer parallelen Region oder in einem Worksharingkonstrukt deklariert ist, einen uninitialisierten Zustand in der parallelen Region hat. Jeder Thread erhält dabei eine neue Instanz der Variablen, wobei die vorherige Instanz nicht innerhalb der parallelen Region referenziert werden darf und nach Beendigung der parallelen Region in einem uninitialisierten Zustand ist.

Darüber hinaus wird folgende Aussage für die Programmiersprache C++ gemacht: *The new list item is initialized, or has an undefined initial vaue, as if it had been locally declared without an initializer. The order in which any default constructors for different private objects are called is unspecified*. Da OpenMP in vielen Fällen die serielle Äquivalenz zu erhalten versucht, wird für ein konformes Programm weiter gefordert, dass der Aufruf von Konstruktoren bzw. Destruktoren keine Nebeneffekte haben darf, welche die serielle Äquivalenz zerstören.

Ein Beispiel für die Privatisierung von Objekten ist die oben beschriebene Implementierung des *Chunk−Allokators* für den parallelen Einsatz. Hierbei soll die Variable *HeapManager* privatisiert (*threadprivate*) werden. Diese Variable ist ein Template vom Typ *ChunkHeapManager*. Der C++ Standard legt fest, dass wenn eine instanzierbare Klassenvariable angelegt wird, auch ohne Initialisierer, der Standardkonstruktor aufgerufen wird. Nach den obigen Aussagen aus der OpenMP Spezifikation muss dies auch für *HeapManager* geschehen. Dies ist für eine korrekte Funktion notwendig, da die Liste zur Verwaltung der Speicherblöcke angelegt werden muss sowie interne Kontrollvariablen gesetzt werden müssen.

Leider wurde festgestellt, dass sich nicht alle Compiler entsprechend der Spezifikation verhalten. Während der Intel C++ Compiler für jede Instanz von *HeapManager*, also für jeden Thread, den Standardkonstruktor aufruft, geschieht dies beim Sun C++ Compiler nicht. Beim Zugriff auf diese Variablen wird dann zur Laufzeit eine *Null Pointer Exception* geworfen. Dieses Verhalten ist nicht korrekt und wurde leider bei weiteren Compilern ebenfalls festgestellt.

Um eine korrekte Arbeit des Codes zu ermöglich, muss dieses Problem vom Programmierer durch eine andere Implementierung umgangen werden. Dazu bietet es sich an, Variablen vom Typ Zeiger auf *HeapManager* zu privatisieren. Zu Beginn des Programms, zumindest vor der ersten Benutzung, kann dann in einer parallelen Region ohne Worksharing jeder Thread seinen Zeiger auf *HeapManager* durch eine entsprechende Konstruktion einer Instanz initialisieren. Dieser Weg wurde auch bei der Implementierung des parallelen *Chunk−Allokators* gewählt.

Bei den ersten Untersuchungen von DROPS im Hinblick auf eine Parallelisierung, die in [Terboven & Spiegel$^+$ 05a] beschrieben sind, wurde die Lauffähigkeit der ersten Version auf verschiedenen verfügbaren Plattformen untersucht. Die serielle Version von DROPS wird am IGPM mit dem GNU C++ Compiler unter Linux entwickelt. Dieser Compiler unterstützt den C++ Standard recht weitgehend und in DROPS werden keine compilerspezifischen Erweiterungen der Sprache C++ verwendet.

Schon die Portierung der seriellen Version erforderte einigen Aufwand. Dies lag nicht an benutzten Bibliotheken oder Abhängigkeiten vom Betriebssystem Linux, sondern lediglich an der unterschiedlichen oder nicht vollständigen Umsetzung des C++ Standards in den verschiedenen Compilern. Für jeden Compiler mussten teilweise recht umfangreiche Modifikationen am Code vorgenommen werden.

Noch schwieriger gestaltete sich die Portierung der unter Linux mit dem Intel C++ Compiler erstellten Parallelisierung mit OpenMP. Die Probleme lassen sich in folgende Kategorien einteilen:

1. Fehlende Unterstützung: einige Compiler unterstützen noch nicht die aktuelle Version 2.5 der OpenMP Spezifikation. Werden darin beschriebene Funktionen verwendet, ist eine Portierung nicht möglich, ohne den Code teilweise deutlich zu verändern. Darüber hinaus gibt es Compiler, die OpenMP nicht im Zusammenspiel mit einigen C++ Konstrukten wie z.B. Templates oder Exceptions un-

terstützen. Eine Anpassung des Codes für diese Compiler war nicht möglich.

2. Fehlerhafte Codeerzeugung: einige Compiler erzeugen einen fehlerhaften Code bei der Verwendung von C++ Klassen oder anderen Konstrukten mit OpenMP. Bei den entsprechenden Compilern wurden Fehlerberichte an die Hersteller geschickt. Die Suche solcher Fehler ist sehr aufwändig und hat, z.B. im Fall der fehlenden Aufrufe von Konstruktoren bei Verwendung des Sun C++ Compilers, einige Zeit in Anspruch genommen.

3. Auslegung der Spezifikation: nicht alle Hersteller interpretieren die OpenMP Spezifikation in gleicher Weise. Dies hat sich besonders bei der Parallelisierung von Schleifen gezeigt, wie oben beschrieben wurde.

Wie bereits in den vorherigen Abschnitten behandelt, ist die Performance einer OpenMP Parallelisierung von C++ nicht unbedingt portabel. Während einige Probleme z.B. beim Wechsel von einer flachen Speicherhierarchie auf eine NUMA-Architektur nicht in der Programmiersprache oder OpenMP begründet sind, können verschiedene STL Implementierungen durchaus deutliche Unterschiede in der Performance auf gleichen Hardwareplattformen begründen.

## 5.7 Zusammenfassung

Es wurden die Fähigkeiten von OpenMP zur Parallelisierung von C++ Codes, die ausgiebigen Gebrauch von komplexen Datentypen z.B. aus der STL machen, untersucht. Die auf den betrachteten Systemen zum Einsatz kommenden modernen Implementierungen der STL sind bereits gut für den Einsatz in parallelen Programmen vorbereitet, dennoch darf nicht uneingeschränkt von mehreren Threads auf Instanzen von STL-Datentypen zugegriffen werden.

Am Beispiel von *std*::*valarray* wurde bei der Parallelisierung des DROPS Codes festgestellt, dass eine komfortable und performante serielle Implementierung Nachteile bei der Parallelisierung mit sich bringen kann. Die implizite Initialisierung mit Null führt auf NUMA-Architekturen zu einem deutlichen Verlust von Skalierbarkeit. Die Template-Technik in Verbindung mit der Objektorientierung erlaubt einen einfachen Austausch des verwendeten Datentypes. Durch das flexible Design der STL mit der Möglichkeit der Verwendung eines eigenen Allokators bei *std*::*vector* wurde eine elegante und portable Möglichkeit ausschließlich unter Verwendung von C++ Sprachmitteln vorgestellt, mit der eine optimale Performance auf der NUMA-Architektur des Sun Fire V40z Systems erreicht wurde. Es wurde gezeigt, wie auch bei anderen Arten der Speicherallokation Einfluss auf die Datenplatzierung genommen werden kann, zum einen mit Mitteln der C++ Programmiersprache, zum zweiten mit Fähigkeiten des Solaris Betriebssystems.

Hinter einer einfachen Benutzerschnittstelle von komplexen Datentypen kann eine aufwändige Implementierung verborgen werden. Der Datentyp *std*::*map* ist zwar auf

allen betrachteten Plattformen effizient implementiert, allerdings ist die Implementierung unter Solaris und unter Windows nicht für die Parallelisierung ausgelegt oder geeignet. Es wurde gezeigt, dass C++ wiederum einfache Möglichkeiten zur Beeinflussung der Speicherverwaltung bietet, mit denen die Skalierbarkeit hergestellt werden kann, ohne größere Änderungen am Benutzercode durchführen zu müssen. Da die Speicherverwaltung aber Aufgabe vom Betriebssystem ist, ist hier eine Zusammenarbeit z.B. über die vorgestellten Bibliotheken notwendig.

Bei der Parallelisierung von objektorientierten Codes gibt es mehrere Ebenen, auf denen die Parallelisierung durchgeführt werden kann. Zwar ist es möglich, die benutzten Bibliotheken wie z.B. die STL in einer parallelen Version zur Verfügung zu stellen, aber im HPC-Umfeld ist der darin verbrauchte Anteil der Rechenzeit in den meisten Fällen eher gering, so dass die Parallelisierung allenfalls als zusätzliche Ebene (*Nesting*) eingesetzt werden könnte. Bei den betrachteten Programmen würde eine solche Parallelisierung nur äußerst geringe Skalierbarkeit bieten. Um einen guten Speedup zu erreichen, muss also auf möglichst hoher Ebene im Programm angesetzt werden. Dazu ist es, wie untersucht wurde, teilweise notwendig, den C++ Code aufzubrechen und Schleifen auszuprogrammieren, um z.B. Barrieren einsparen zu können. Wie im Fall von FIRE festgestellt wurde, kann ein objektorientiertes Design des Programms mit der dadurch erreichten Datenkapselung Vorteile bei der Abhängigkeitsanalyse bieten und die Parallelisierung erleichtern.

In der OpenMP Spezifikation wird als Vorrausetzung für eine zu parallelisierende Schleife die kanonische Form gefordert. In C++ Codes ist dies aber nicht immer der Fall, insbesondere bei der Iteration über die Elemente von STL-Datentypen. Es wurden drei Ansätze zur Parallelisierung von nicht konformen Schleifen diskutiert. Dabei wurde festgestellt, dass die von Intel vorgeschlagene Erweiterung der *Taskqueues* zusammen mit Parametern zur Steuerung des Scheduling eine elegante und performante Erweiterung der aktuellen OpenMP Spezifikation darstellen würde. Der bisher effizienteste Weg ist das Speichern der Zeiger in einem Array und das anschließende parallele Bearbeiten dieses Arrays. Ebenfalls wurden Einschränkungen in der OpenMP Spezifikation diskutiert, welche die Arbeit an den hier vorgestellten Codes aufwändiger gemacht haben.

# 6 Ergebnisse und Ausblick

In dieser Diplomarbeit wurden verschiedene Aspekte der Parallelisierung von C++ Programmen für Shared-Memory Systeme betrachtet. Der Schwerpunkt lag dabei auf Techniken zur Parallelisierung der Programme DROPS und FIRE sowie der Verbesserung der Skalierung von ADI auf NUMA-Architekturen. Um ausgewählte Effekte besser demonstrieren zu können, wurden synthetische Benchmarks erstellt und untersucht.

Zur Parallelisierung von C++ Programmen für Shared-Memory Systeme bieten sich drei Parallelisierungstechniken an: die Posix-Threads Bibliothek, die direktivengesteuerte Parallelisierung mit OpenMP und die Programmiersprache UPC. Diese drei Techniken wurden vorgestellt und ihre wesentlichen Unterschiede dargelegt. Dabei unterschieden sich die Posix-Threads und OpenMP hauptsächlich in ihrer Abstraktionsebene für den Programmierer, während UPC mit dem Distributed-Shared-Memory Speichermodell einen etwas anderen Ansatz verfolgt. Es wurde gezeigt, dass sich alle drei Techniken zur Parallelisierung des einfachen STREAM-Benchmarks eignen, aber auch zur nachträglichen Parallelisierung von DROPS. Die wichtigsten Vergleichskriterien waren dabei der Aufwand bei der Programmierung der Arbeitsverteilung und die auf der NUMA-Architektur der Sun Fire V40z Systeme erreichbare Performance. Während OpenMP mit den wenigsten Codeänderungen auskommt und die einfachste Herangehensweise bietet, ist es auf der NUMA-Architektur UPC bezüglich der Performance unterlegen. Um dort gleichzuziehen, sind etwas aufwändigere Maßnahmen notwendig, die diskutiert wurden. Die Posix-Threads bieten zwar die größte Flexibilität, allerdings ist diese bei den betrachteten Codes nicht notwendig und zum Erreichen der Performance der anderen beiden Techniken muss einiger Aufwand betrieben werden.

Die Parallelisierung von DROPS mit OpenMP, welche in weiten Teilen bereits im Vorfeld durchgeführt wurde, war grundsätzlich erfolgreich. Allerdings wurde festgestellt, dass die Skalierbarkeit nicht uneingeschränkt zwischen Betriebssystemen und Hardwareplattformen portabel ist. Der Schwerpunkt lag dabei auf den im Rechenzentrum der RWTH Aachen in großem Umfang eingesetzten UNIX-Systemen, nämlich der Opteron-Architektur mit den Sun Fire V40z Systemen und der SPARC-Architektur mit den Sun Fire E6900 Systemen. Die Hauptgründe für die Begrenzung des Speedups lagen zum einen darin, dass die NUMA-Architektur der Sun Fire V40z Systeme bislang nicht berücksichtigt wurde, zum anderen in einem nicht optimalen Speichermanagement von STL-Datentypen. In dieser Arbeit wurden erfolgreiche Ansätze zur Verbesserung der Skalierbarkeit vorgestellt, die sowohl mit Mitteln der Programmiersprache C++ als auch mit Mitteln des Betriebssystems implementiert wurden. In einigen Aspekten wurde auch das Betriebssystem Windows auf den Sun Fire V40z Systemen betrachtet, allerdings waren die Ansätze dort nur bedingt erfolgreich. Im Bezug auf die Programmierung von HPC-Anwendungen für Windows müssen noch Arbeit und Untersuchungen investiert werden.

Auf der NUMA-Architektur der Sun Fire V40z Systeme hat sich die Verwendung

von *std*::*valarray* in DROPS als problematisch erwiesen. Zwar ist die Umsetzung ebenso effizient wie ein Zeiger in C und die automatische Initialisierung bei der Programmentwicklung ein komfortables Feature, allerdings sorgt die Initialisierung für eine fixe Datenverteilung, die seitens der Anwendung nicht mehr nachträglich beeinflusst werden kann. Zu diesem Problem wurden Lösungen sowohl unter Zuhilfenahme von Fähigkeiten des Betriebssystems als auch ausschließlich mit Mitteln der C++ Programmiersprache vorgestellt. Während der Rückgriff auf das Betriebssystem Solaris zwar auf der entsprechenden Plattform die gewünschte Performance bringt, ist dieser Ansatz nicht portabel. Bei Beachtung von angegebenen Regeln ist die Verwendung von *std*::*vector* ebenso effizient, darüber hinaus ist die Angabe eines Allokators möglich. Es wurde ein Framework vorgestellt, mit dem sowohl ein Allokator zur verteilten Allokation von STL Datentypen erstellt werden kann, als auch der angeforderte Speicher von allgemeinen Klassen und Objekten nach Vorgabe verteilt werden kann. Die Art der Verteilung kann dabei entsprechend dem Scheduling in OpenMP angepasst werden.

Bei den aktuellen STL Implementierungen wurde Wert auf eine hohe serielle Performance gelegt, so verwenden z.B. die auf der SGI STL aufbauenden STL Implementierungen einen speziellen Allokator für kleine Objekte. Da dieser Allokator aber statische Informationen enthält, führt er ein internes Locking durch, welches eine Skalierung in den Setup-Routinen von DROPS und im MAP-Benchmark verhindert. Da die Verteilung des Speichers an Anwendungen vom Betriebssystem erfolgt und das Betriebssystem Solaris bereits eine Bibliothek mit einer besseren Speicherallokation für parallele Programme mitbringt, ist dies der einfachste Weg, die Skalierbarkeit zu erhöhen. Der Datentyp *std*::*map* erlaubt aber die Angabe eines eigenen Allokators, so dass durch eine Block-Allokationsstrategie die Performance beim MAP-Benchmark noch einmal deutlich erhöht werden konnte. Während die Implementierung der Allokation unter Linux bereits gut für parallele Programme geeignet ist, wird unter Solaris erst durch die Verwendung von *libmtmalloc* der erwartete Speedup erreicht. Die Verwendung der Block-Allokationsstrategie brachte bei den Setup-Routinen von DROPS nur noch einen geringen Performancegewinn. Hier könnte durch Integration der Fähigkeiten von parallelen Allokationsbibliotheken, z.B. die Verwaltung von mehreren Heaps, in einen entsprechend auf die Anwendung zugeschnittenen Allokator noch ein weiterer Gewinn erzielt werden.

Es wurde dargelegt, dass ein objektorientiertes Design die Parallelisierung erleichtern kann. Zum einen wird die Abhängigkeitsanalyse, die in den meisten Fällen vom Programmierer durchgeführt werden muss, erleichtert. Darüber hinaus sorgt eine gut durchgeführte Kapselung der Daten dafür, dass der Aufwand bei der Parallelisierung verringert wird, wie z.B. bei FIRE. Bei der Parallelisierung von objektorientiertem Code gibt es meist mehrere Ebenen, auf denen die Parallelisierung möglich ist. Es wurden Ansätze aus der Literatur vorgestellt, die Parallelisierung bereits in Bibliotheken wie der STL anzubieten. Bei den betrachteten Programmen und den meisten Anwendungen im HPC-Umfeld bietet eine solche Parallelisierung aber nur eine geringe Skalierbarkeit. Bei den betrachteten Programmen hat es sich empfohlen, an einigen Stellen

den Code so umzuschreiben, dass z.B. Schleifen zusammengefasst werden können um Barrieren zu eliminieren, auch wenn dies eventuell zu Lasten der Eleganz gehen mag.

Im Rahmen dieser Diplomarbeit wurde eine Parallelisierung von FIRE mit OpenMP durchgeführt, wobei sich dieses Programm für eine geschachtelte Parallelisierung anbietet und somit zwei Ebenen mit Nested OpenMP erstellt wurden. Die Parallelisierung war sehr erfolgreich. Der Einsatz von Nesting brachte den Vorteil eines höheren Speedups bei steigender Anzahl von eingesetzten Prozessoren im Vergleich zur Verwendung aller Prozessoren in nur einer Ebene. Daneben kann damit die Parallelisierung flexibel an die Anzahl der zur Verfügung stehenden Prozessoren sowie der Anzahl der Anfragebilder und der Größe der Bilddatenbank angepasst werden. Da weiterhin einer serielle Kompilation auch mit nicht OpenMP-fähigen Compilern möglich ist und die Änderungen am Code minimal sind, wurde die Parallelisierung bereits in die weitere Entwicklung von FIRE übernommen.

Bei der Parallelisierung von C++ Programmen wie z.B. von DROPS, die Gebrauch von komplexen Datentypen aus Bibliotheken wie der STL machen, sind einige Regeln zum Erreichen einer guten Skalierbarkeit und korrekt arbeitender paralleler Programme zu beachten. Um die Performance nicht zu behindern, sind die Datentypen aus der STL nicht intern durch Locks vor parallelem Zugriff geschützt. Falls mehrere Threads gleichzeitig auf eine Instanz eines STL-Daten zugreifen und dabei die interne Representation der Instanz ändern können, muss ein Locking zur Sicherstellung der Korrektheit auf Seiten der Anwendung implementiert werden. Für den Fall, dass höchstens ein Thread auf eine Instanz zugreift, wie z.B. in DROPS und FIRE, sind die Datentypen der aktuellen STL Implementierungen *thread-safe*, da alle ihre Funktionen *reentrant* sind.

Bei den in dieser Arbeit untersuchten Programmen waren die zu parallelisierenden Schleifen nicht immer von der kanonischen Form, wie sie die OpenMP Spezifikation vorschreibt. Es wurden drei Ansätze diskutiert. Auf allen Plattformen war der Ansatz möglich, eine Iterator-Schleife ein erstes Mal seriell zu durchlaufen und dabei die Zeiger in einem Array zu speichern und anschließend in einem zweiten Durchlauf parallel das Array abzuarbeiten. Eine etwas elegantere Technik hat Intel mit dem *Taskqueuing* vorgestellt. Hier fehlt zwar noch die Möglichkeit der Beeinflussung des Scheduling, allerdings wäre dies eine elegante Technik zur Parallelisierung von Schleifen mit Zeigern oder Iteratoren, welche OpenMP für C++ Programme noch attraktiver machen würde. Darüber hinaus wurden einige Einschränkungen in der aktuellen Spezifikation und Fehler in aktuellen Compilern diskutiert, welche die Parallelisierung der hier betrachteten Codes haben aufwändiger werden lassen.

Insgesamt lässt sich feststellen, dass OpenMP gut zur Parallelisierung von C++ Programmen eingesetzt werden kann. Im Vergleich zu den anderen betrachteten Parallelisierungstechniken bietet es den meisten Komfort und kann am flexibelsten eingesetzt werden. Die Grundlage der Parallelisierung sollte ein bereits seriell performantes Programm sein. Bei der Verwendung von dynamischen Daten sind zur Erstellung eines effizienten Programms in C++ bereits einige Erfahrungen vom Programmierer gefor-

dert, im Vergleich zu anderen Programmiersprachen ist die Lernkurve in einigen Fällen steiler.

Bei der Parallelisierung der betrachteten Programme wurde die Skalierbarkeit zunächst zwar von einigen Problemen begrenzt, allerdings waren diese nicht ausschließlich durch den Einsatz der Programmiersprache C++ begründet. Die vorgestellten Lösungen, die in vielen Fällen ausschließlich mit Sprachmitteln erstellt wurden, sind flexibel und lassen sich auf andere Programme übertragen. Dabei haben gerade C++ typische Sprachkonstrukte die Implementierung erleichtert. Zwar wurde OpenMP vor allem in Hinblick auf FORTRAN und C spezifiziert und es lassen sich einige Wünsche zur Erleichterung der Parallelisierung mit C++ formulieren, diese Unzulänglichkeiten sind in den meisten Fällen aber mit nur geringem manuellen Aufwand zu umgehen.

# A Parallelisierung der Matrix-Vektor-Multiplikation

Der Code der Routine *y_Ax*, der sparse Matrix-Vektor-Multiplikation aus DROPS, ist im Programmausschnitt 31 dargestellt. Das Schlüsselwort *__restrict* ist zwar im aktuellen Sprachumfang der Programmiersprache C++ nicht deklariert, allerdings wird es bereits von einigen Compilern akzeptiert, da es Bestandteil der Spezifikation C99 ist. Für Compiler wo dies nicht gilt, muss es z.B. über ein Makro ausgeschaltet werden. Seine Semantik ist, dass der mit *__restrict* bezeichnete Zeiger mit anderen Zeigern überschneidungsfrei ist, was dem Compiler gewisse Optimierungen ermöglicht.

**Programmausschnitt 31** DROPS: Routine *y_Ax*, Originalversion.

```
1  void y_Ax(double* __restrict y,
2            size_t num_rows,
3            const double* __restrict Aval,
4            const size_t* __restrict Arow,
5            const size_t* __restrict Acol,
6            const double* __restrict x)
7  {
8    double sum;
9    long rowend;
10   long rowbeg;
11   for (int i = 0; i < num_rows; i++) {
12     sum = 0.0;
13     rowend = Arow[i+1];
14     rowbeg = Arow[i];
15     for (long nz=rowbeg; nz<rowend; ++nz) {
16       sum+= Aval[nz]*x[Acol[nz]];
17     }
18     y[i] = sum;
19   }
20 }
```

Die Rückgabe des Ergebnisses erfolgt in die Variable `y`. Die Variablen `Aval`, `Arow` und `Acol` bezeichnen die drei Arrays aus der CRS-Darstellung der Matrix `A` und `x` den zu multiplizierenden Vektor. Die Multiplikation und Addition erfolgt von Zeile 15 bis Zeile 17, die Indexgrenzen werden direkt davor berechnet. Wegen der CRS-Darstellung ist ein indirekter Zugriff auf den Vektor `x` notwendig.

Bei der Parallelisierung mit OpenMP wird die `for`-Schleife von Zeile 11 bis Zeile 17 parallelisiert, wie im Programmausschnitt 32 dargestellt. Die Berechnung wird zeilenweise aufgeteilt. Die drei Variablen, welche die Matrix representieren, sowie der Vektor *x* sind natürlich *shared*, ebenso der zu berechnende Vektor *y*. Die Variablen zur Berechnung der Elemente einer Zeile sowie zur Summenbildung müssen privatisiert

**Programmausschnitt 32** DROPS: *y_Ax*, OpenMP Parallelisierung.

```
1 #pragma omp parallel shared(num_rows,Arow,Aval,x,Acol,y)
2                      private(i, rowend, rowbeg, sum)
3 {
4 #pragma omp for
5   for (i = 0; i < num_rows; i++) {
6     sum = 0.0;
7     rowend = Arow[i+1];
8     rowbeg = Arow[i];
9     for (long nz=rowbeg; nz<rowend; ++nz) {
10      sum += Aval[nz]*x[Acol[nz]];
11    }
12    y[i] = sum;
13  }
14 } // end omp parallel
```

werden, könnten aber auch lokal in der *for*-Schleife deklariert werden.

Die Parallelisierung mit den Posix-Threads ist im Programmausschnitt 33 dargestellt. Hier muss die Übergabe der benötigten Daten manuell programmiert werden.

Die in der Arbeitsroutine benötigten Daten sind nicht mehr global im Programm verfügbar, wie dies beim STREAM-Benchmark der Fall war. Dazu wurde die Struktur *sThreadData* deklariert, die neben den benötigten Daten für die eigentliche Berechnung aus der Struktur *sExperimentData* auch die logische Thread-ID enthält. In der Arbeitsroutine werden zunächst lokale Zeiger auf die Daten aus der übergebenen Struktur deklariert, damit der darauf folgende Code nur wenig geändert werden muss, wie in Zeile 13 angedeutet. Da ein *void*-Zeiger übergeben wird, ist eine entsprechende Typkonvertierung notwendig. Danach wird anhand der Thread-ID die statische Arbeitsverteilung berechnet, also unterschiedliche Bereiche des Laufindex *i* für alle Threads. Der Schleifeninhalt muss nicht verändert werden und ist deshalb nicht abgebildet.

Die UPC Version ist im Programmausschnitt 34 dargestellt. Wie in 4.3 beschrieben wurde, ist ein wenig Aufwand notwendig, um einen mit UPC parallelisierten Programmkern mit einem C++ Programm zu verbinden.

Aus dem C++ Programm wird nicht mehr die Funktion *y_Ax* aufgerufen, sondern die Funktion *upckernel*, die in eine eigene Quellcodedatei ausgelagert ist. Um in UPC die Arbeit zu verteilen, müssen die Daten im globalen Speicher liegen. Da die Daten im C++ Teil des Programms angelegt wurden, ist dies nicht der Fall. Somit werden Kopien der Daten angelegt und entsprechend verfügbar gemacht, wie in Zeile 6, 7 und 14 dargestellt ist. Daran anschließend wird die Routine *y_Ax* aufgerufen (Zeile 20), wobei nun jeder Thread nur auf seinem Teil der *shared* Daten arbeitet.

Das Ergebnis muss anschließend auch aus dem *shared* Speicher wieder zurückko-

**Programmausschnitt 33** DROPS, *y_Ax*, Posix-Threads Parallelisierung.

```
1  struct sExperimentData {
2    size_t    num_nonzeros;
3    [...]
4  } expData;
5
6  struct sThreadData {
7    sExperimentData         *pExpData;
8    int                     iThreadNum;
9  } threadData[NTHREADS];
10
11 void* worker(void *pThreadData)
12 {
13   double* y = ((sThreadData*)pThreadData)->pExpData->y;
14   [...]
15 }
```

**Programmausschnitt 34** DROPS: *y_Ax*, UPC Parallelisierung.

```
1  void upckernel(double* y, size_t num_rows, double* Aval,
2                 size_t* Arow, size_t* Acol, double* x,
3                 size_t num_nonzeros)
4  {
5    // allocate shared memory
6    shared double* __Aval;
7    __Aval = (shared double*) upc_all_alloc(num_nonzeros,
8                                            sizeof(double));
9    [...]
10
11    // put master's local data into shared memory
12   if (MYTHREAD == 0)
13   {
14     upc_memput((shared void*)__Aval, (void*) Aval,
15                num_nonzeros * sizeof(double));
16     [...]
17   }
18   upc_barrier(123);
19
20   y_Ax(__y, num_rows, __Aval, __Arow, __Acol, __x);
21 }
```

**Programmausschnitt 35** DROPS: Routine *PCG*, OpenMP Parallelisierung.

```
1  bool PCG(const SparseDistributedMatCL<double>& A, [...])
2  {
3    [...]
4  #pragma omp parallel
5  {
6    for (int i = 1; i <= max_iter; i++ )
7    {
8      [...]
9      y_Ax_par(&q.raw()[0], A.num_rows(),     // q = A * p;
10               A.raw_val(), A.raw_row(),
11               A.raw_col(), Addr( p.raw()));
12 #pragma omp barrier
13     [...]
14                          // x += alpha * q; r -= alpha * q;
15 #pragma omp for
16     for (long j=0; j<n; j++)
17     {
18       x[j] += alpha * p[j];
19       r[j] -= alpha * q[j];
20     }
21     [...]
22 } // end parallel
23   [...]
24 }
```

piert werden. Zwar nimmt das Kopieren eine gewisse Zeit in Anspruch, jedoch ist es bei der Kombination von C++ und UPC notwendig, wenn die Daten, auf denen parallel gearbeitet werden soll, im seriellen Teil angelegt werden. Bei der Routine *y_Ax* ist der Aufwand des Kopierens im Vergleich zur Berechnung so gering, dass sich die Parallelisierung lohnt.

# B Parallelisierung des PCG-Verfahrens

In Abschnitt 5.4 wurde bereits der Kern der Routine *PCG* dargestellt. Im Programmausschnitt 35 wird die Parallelisierung der Routine mit OpenMP gezeigt.

Anstatt der Matrix-Vektor-Multiplikation wird die Routine *y_Ax* aufgerufen, deren Parallelisierung bereits oben beschrieben wurde und *y_Ax_par* entspricht der OpenMP Version. Ebenso ist die Parallelisierung der Skalierungsoperationen der Vektoren $q$ und $p$ gezeigt. Da beiden Vektoren die gleiche Dimension besitzen, konnten die Schleifen

in einem *for*-Worksharingskonstrukt zusammengefasst werden.

Die Parallelisierung in den beiden anderen Parallelisierungstechniken erfolgt nach dem selben Prinzip. Es wird die entsprechende Version der Matrix-Vektor-Multiplikation aufgerufen, ebenfalls können wieder die Skalierungsoperationen zusammengefasst werden. Die Routine *PCG* erhält als Parameter alle benötigten Daten, diese müssen in UPC wiederum als *shared* Daten zur Verfügung gestellt werden. Bei den Posix-Threads müssen entsprechende Strukturen zur Übergabe der benötigten Daten für die Arbeitsroutinen erstellt werden. Darüber hinaus ist der Einsatz des Thread-Pools bei den Posix-Threads wichtig zum Erreichen einer guten Skalierung.

# C Implementierung des Thread-Pools

Der hier implementierte Thread-Pool stellt die Funktionen zur Verfügung, die im Programmausschnitt 36 dargestellt sind.

**Programmausschnitt 36** Thread-Pool: Interface.

```
1 void tp_start(size_t);
2 void tp_stop();
3 void tp_create(size_t, void* (*)(void*), void *);
4 void tp_join(size_t, void **);
```

Das Ziel dieser Implementierung war nicht, einen allgemein einsetzbaren Thread-Pool für die Posix-Threads zu entwickeln, sondern bei den in dieser Diplomarbeit durchgeführten Experimenten die Vorteile eines Thread-Pools zu evaluieren und die Performance dieser Parallelisierungstechnik auf das Niveau von OpenMP und UPC zu heben. Ein allgemeiner Thread-Pool mit Schwerpunkt für irreguläre Parallelisierungen wie z.B. in Web-Server wird in [Butenhof 97] vorgestellt. Das hier vorgestellte Interface wurde darauf ausgelegt, den Thread-Pool ohne größere Änderungen am Code in den Programmen verwenden zu können.

Vor der Verwendung des Thread-Pools ist die Routine *tp_start*() aufzurufen. Als Argument erwartet sie die Anzahl der zu erzeugenden Threads. Die Routine *tp_stop*() sorgt dafür, dass alle Threads terminiert werden und muss somit nach der Verwendung der Parallelität aufgerufen werden. Falls Threads noch mit Arbeit beschäftig sind, wird auf die Beendigung der Arbeit gewartet.

Die Routinen *tp_create*() und *tp_join*() werden als Ersatz für *pthread_create*() und *pthread_join*() aufgerufen. Beim Aufruf von *tp_create*() wird kein neuer Thread erzeugt, aber das zweite Argument (der Funktionspointer) wird dem Thread mit der Nummer, die im ersten Argument angegeben wird, zugewiesen. Das dritte Argument kann einen Zeiger auf einen Bereich für die Argumente aufnehmen.

Die Routine *tp_join*() wartet, bis der im ersten Argument angegebene Thread seine Arbeit beendet hat. Im zweiten Argument kann ein Zeiger auf einen Bereich für

das Ergebnis angegeben werden. Damit entspricht das Interface dieser beiden Routinen in weiten Teilen dem Interface der entsprechenden Routine aus der Posix-Threads Bibliothek. Lediglich die Angabe der Threads über ihre logische Nummerierung ist geändert.

**Programmausschnitt 37** Thread-Pool: Routine *tp_activewaiting*.

```
1  void * tp_activewaiting(void * pThreadNum) {
2    size_t ThreadNum = *((size_t*)pThreadNum);
3
4    bool bHaveWork = false; bool bKeepRunning = true;
5    while(bKeepRunning)
6    {
7      pthread_mutex_lock(&(m_vMutexes[ThreadNum]));
8      if (m_vStatus[ThreadNum] == STAT_TERM)
9      {
10       bKeepRunning = false;
11     }
12     if (m_vStatus[ThreadNum] == STAT_WORK)
13     {
14       bHaveWork = true;
15     }
16     pthread_mutex_unlock(&(m_vMutexes[ThreadNum]));
17     if (bHaveWork)
18     {
19       m_vReturnValues[ThreadNum] = m_vRoutines[ThreadNum]
20                                    (m_vArguments[ThreadNum]);
21       m_vStatus[ThreadNum] = STAT_WAIT;
22       pthread_cond_signal(&(m_vConditions[ThreadNum]));
23       bHaveWork = false;
24     }
25   }
26   pthread_exit(NULL);
27   return NULL;
28 }
```

Nach dem Start und notwendigen Initialisierungen rufen die Threads die Routine *tp_activewaiting*() auf, die im Programmausschnitt 37 dargestellt ist. Darin warten die Threads in einer *while*-Schleife von Zeile 5 bis Zeile 24 darauf, dass ihnen Arbeit zugewiesen wird (Test in Zeile 12) oder auf einen Hinweis zum Beenden (Test in Zeile 8). In Zeile 19 wird die eigentliche Arbeitsroutine aufgerufen. Das Signal in Zeile 22 ist notwendig, um dem eventuell in *tp_join*() wartenden Master-Threads mitzuteilen, dass die Arbeit erledigt ist. Dieser Code ist im Programmausschnitt 38 dargestellt. In

Zeile 8 wird darauf gewartet, dass der Thread für Arbeit frei ist.

**Programmausschnitt 38** Thread-Pool: Routine *tp_join*.

```
1  void tp_join(size_t ThreadNum, void **work_return) {
2    bool b = true;
3    while (b)
4    {
5      pthread_mutex_lock(&(m_vMutexes[ThreadNum]));
6      while (m_vStatus[ThreadNum] != STAT_WAIT)
7      {
8        pthread_cond_wait(&(m_vConditions[ThreadNum]),
9                          &(m_vMutexes[ThreadNum]));
10     }
11     if (work_return != NULL)
12       *work_return = m_vReturnValues[ThreadNum];
13     b = false;
14     pthread_mutex_unlock(&(m_vMutexes[ThreadNum]));
15   }
16   return;
17 }
```

# Abbildungsverzeichnis



# Tabellenverzeichnis

# Verzeichnis der Programmausschnitte

# Literatur

[Amdahl 67] G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. Proc. *AFIPS Conference Proceedings*, pp. 483–485, 1967.

[An & Jula$^+$ 01] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, L. Rauchwerger. STAPL: An Adaptive, Generic Parallel C++ Library. Proc. *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pp. 193–208, 2001.

[an Mey & Sarholz$^+$ 05] D. an Mey, S. Sarholz, A. Spiegel, C. Terboven, R. van der Pas, E. Loh. *The RWTH Sun Fire SMP-Cluster User Guide*. Center for Computing and Communication, RWTH Aachen University, 2005.

[Attardi 03] J. Attardi. A Comparison of Memory Allocators in Multiprocessors. `http://developers.sun.com/solaris/articles/multiproc/multiproc.html`, 2003. Zuletzt besucht am 28.01.2006.

[Balart & Duran$^+$ 05] J. Balart, A. Duran, M. Gonzalez, X. Martorell, E. Ayguade, J. Labarta. Experiences parallelizing a Web Server with OpenMP. Proc. *IWOMP '05: Proceedings of the First International Workshop on OpenMP*, 2005.

[Berger & McKinley$^+$ 00] E. D. Berger, K. S. McKinley, R. D. Blumofe, P. R. Wilson. Hoard: a scalable Memory Allocator for Multithreaded Applications. Proc. *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pp. 117–128, 2000.

[Berger & Zorn$^+$ 01] E. D. Berger, B. G. Zorn, K. S. McKinley. Composing high-performance Memory Allocators. Proc. *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pp. 114–124, 2001.

[Board 05] O. A. R. Board. *OpenMP Application Program Interface*. OpenMP Architecture Review Board, 2005.

[Bracha & Cook 90] G. Bracha, W. Cook. Mixin-based inheritance. Proc. N. Meyrowitz, editor, *Proceedings of the Conference on Objekt-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pp. 303–311, 1990.

[Butenhof 97] D. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[Chauvin & Saha$^+$ 04] S. Chauvin, P. Saha, F. Cantonnet, S. Annareddy, T. El-Ghazawi. *UPC Manual*. George Washington University, 2004.

[Deselaers 03] T. Deselaers. Features for Image Retrieval. Master's thesis, RWTH Aachen University, 2003.

[Deselaers 05] T. Deselaers. FIRE - Flexible Image Retrieval Engine. http://www-i6.informatik.rwth-aachen.de/~deselaers/fire.html, 2005. Zuletzt besucht am 28.01.2006.

[Dowd & Severance 98] K. Dowd, C. Severance. *High Performance Computing*. O'Reilly and Associates Inc., 1998.

[Etnus 05] Etnus. The TotalView Debugger. http://www.etnus.com/TotalView/index.html, 2005. Zuletzt besucht am 28.01.2006.

[Fletcher & Sankaran 03] G. Fletcher, S. Sankaran. Approaches to Parallel Generic Programming in the STL Framework. 2003.

[Flynn 66] M. J. Flynn. Very High-Speed Computing Systems. Proc. *Proceedings of the IEEE*, Vol. 54, pp. 1901–1909, 1966.

[Forum 95] M. Forum. *MPI: a Message-Passing Interface Standard*. MPI Forum, 1995.

[Forum 03] M. Forum. *MPI-2: Extensions to the Message-Passing Interface*. MPI Forum, 2003.

[FSF 05] FSF. *GCC online documentation*. Free Software Foundation, 2005. http://gcc.gnu.org/onlinedocs/. Zuletzt besucht am 28.01.2006.

[Gross & Peters$^+$ 02] S. Gross, J. Peters, V. Reichelt, A. Reusken. The DROPS Package for Numerical Simulations of Incompressible Flows using Parallel Adaptive Multigrid Techniques. Technical Report 211, IGPM, RWTH Aachen University, 2002.

[Hennessy & Patterson 03] J. L. Hennessy, D. A. Patterson. *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann Publishers, 2003.

[Keysers & Gollan$^+$ 04] D. Keysers, C. Gollan, H. Ney. Classification of Medical Images using Non-linear Distortion Models. Proc. *BVM 2004, Bildverarbeitung fuer die Medizin 2004*, pp. 366–370, 2004.

[McCalpin 05] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/, 2005. Zuletzt besucht am 28.01.2006.

[Microsoft 05] Microsoft. *Visual Studio 2005*. Microsoft, 2005. http://msdn.microsoft.com/vstudio. Zuletzt besucht am 28.01.2006.

[Salman & Turovets+ 05] A. Salman, S. Turovets, A. D. Malony, J. Eriksen, D. M. Tucker. Computational Modeling of Human Head Conductivity. Proc. *International Conference on Computational Science*, pp. 631–638, 2005.

[SGI 05] SGI. *Standard Template Library Programmers Guide*. Silicon Graphics, 2005. `http://www.sgi.com/tech/stl/`. Zuletzt besucht am 28.01.2006.

[Shah & Haab+ 99] S. Shah, G. Haab, P. Petersen, J. Throop. Flexible Control Structures for Parallelism in OpenMP. 1999.

[Stroustrup 00] B. Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, 2000.

[Su & Tian+ 02] E. Su, X. Tian, M. Girkar, G. Haab, S. Shah, P. Petersen. Compiler Support of the Workqueuing Execution Model for Intel SMP Architectures. Proc. *EWOMP '02: Proceedings of the Fourth European Workshop on OpenMP*, 2002.

[Sun 05a] Sun. *Solaris Enterprise System*. Sun Microsystems, 2005. `http://www.sun.com/software/solaris/index.jsp`. Zuletzt besucht am 28.01.2006.

[Sun 05b] Sun. *Sun Studio 10 for Solaris Platforms*. Sun Microsystems, 2005. `http://developers.sun.com/prodtech/cc/documentation/ss10\_docs/index.ht%ml`. Zuletzt besucht am 28.01.2006.

[Sunderam 90] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. Proc. *Concurrency: Practice and Experience*, pp. 315–339, 1990.

[SuSE 05] SuSE. *freshmeat: numactl 0.8 for Linux*. SuSE, 2005. `ftp://ftp.suse.com/pub/people/ak/numa/numactl-0.8.tar.gz`. Zuletzt besucht am 28.01.2006.

[Terboven & Spiegel+ 05a] C. Terboven, A. Spiegel, D. an Mey, S. Gross, V. Reichelt. Experiences with the OpenMP Parallelization of DROPS, a Navier-Stokes Solver written in C++. Proc. *IWOMP '05: Proceedings of the First International Workshop on OpenMP*, 2005.

[Terboven & Spiegel+ 05b] C. Terboven, A. Spiegel, D. an Mey, S. Gross, V. Reichelt. Parallelization of the C++ Navier-Stokes Solver DROPS with OpenMP. Proc. *PARCO '05: Proceedings of Parallel Computing 2005*, 2005.

[Veldhuizen 95] T. L. Veldhuizen. Expression templates. Proc. *C++ Report*, Vol. 7, pp. 26–31, 1995.