Parallelization of irregular parallel programs using Task Pools and OpenMP

Abschlussarbeit zum Diplom I
an der Universität Kassel

Offiziell abgegeben: 19.04.2006

Vorgelegt von
Alexander Wirz

Betreuer:
Dipl.-Inf. Michael Süß
Prof. Dr. Claudia Leopold
Prof. Dr. Kurt Geihs
Universität Kassel
Fachbereich 16 - Elektrotechnik/Informatik
Fachgebiet Programmiersprachen/-methodik
Wilhelmshöher Allee 73
34121 Kassel

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

Durch Nutzung von parallelen Rechensystemen können die Ausführungszeiten von rechenintensiven Programmen deutlich verkürzt werden. Um die Programmierung dieser Systeme zu erleichtern, wurden viele Programmiermodelle entwickelt. Eins davon ist OpenMP. OpenMP ist eine Spracherweiterung für C/C++ und FORTRAN. OpenMP versucht die Komplexität, die mit der Programmierung von parallelen Rechensystemen verbunden ist, zu minimieren, ohne dass die Performance dabei auf der Strecke bleibt.

OpenMP bietet viele mächtige Mechanismen, um reguläre Programme zu parallelisieren. Reguläre Programme führen gleiche, voneinander unabhängige Operationen auf verschiedenen Elementen einer regulären Datenstruktur, wie z.B. einem Array oder einer Matrix aus. Die Menge der Daten, die bearbeitet werden sollen, ändert sich nicht während der Ausführung des Programms. Die gesamte Menge an Berechnungen kann bei regulären Programmen vom Anfang an bestimmt werden. Das bedeutet, dass die Aufteilung der Arbeit zwischen einzelnen Threads oder Prozessen statisch, vor Beginn der eigentlichen Berechnung, erfolgen kann.

Eine andere Klasse von Programmen sind die so genannten irregulären Programme. Diese sind schwierig zu parallelisieren, da man bei dieser Art von Programmen die Menge an Arbeit nicht im Voraus abschätzen kann. Somit ist ein statisches Schedulingverfahren ungeeignet für diese Art von Programmen. Um eine effiziente Parallelisierung zu erreichen, muss die Arbeit während der Ausführung, dynamisch, zwischen den Prozessoren verteilt werden.

Ein Ansatz, um die dynamische Lastenbalancierung zu erreichen, ist die Verwendung von Taskpools. Taskpools sind Datenstrukturen, die eine dynamische Verteilung der Arbeit zwischen den Threads ermöglichen. Mit Hilfe

eines Taskpools können Arbeitseinheiten (Tasks) zwischengespeichert und bei Bedarf wieder angefordert werden.

Es gibt mehrere Möglichkeiten einen Taskpool zu implementieren. Einige davon werden in dem Artikel von Korch und Rauber [4] beschrieben. Diese Autoren haben die, in ihrem Artikel, beschriebenen Taskpoolvarianten mit Hilfe der Pthreads-Bibliothek in C und einige davon mit JavaThreads in Java implementiert. Das Ziel dieser Arbeit war, einige dieser Taskpools von der Pthreads Implementierung von Korch und Rauber nach OpenMP zu portieren und diese gegebenenfalls zu erweitern. Dabei sollten auch Probleme, die während der Portierung nach OpenMP auftreten, und die verwendeten Techniken beschrieben werden.

Um die Taskpools zu testen, habe ich drei Programme implementiert: Quicksort, das Labyrinth-Problem und die Cholesky-Faktorisierung von dünnbesetzten Matrizen. Quicksort ist ein bekannter und häufig verwendeter Sortieralgorithmus. Beim Labyrinth-Problem geht es darum, den kürzesten Weg durch ein Labyrinth zu finden. Die Cholesky-Faktorisierung zerlegt eine symmetrische positiv definite Matrix A in das Produkt aus einer unteren Dreiecksmatrix und deren Transponierten: $A = LL^T$. Alle drei Programme gehören zur Klasse der irregulären Algorithmen und sind daher schwer zu parallelisieren.

1.1 Struktur der Arbeit

Die Arbeit besteht aus sieben Kapiteln. Im Kapitel 2 werden Grundbegriffe geklärt, die für das Verständnis der Arbeit hilfreich sind. Dieses Kapitel enthält allgemeine Informationen zur Parallelverarbeitung und zum Taskpoolkonzept. Außerdem befindet sich in diesem Kapitel eine kurze Beschreibung des OpenMP Standards.

Kapitel 3 befasst sich mit Speichermanagern. Im Kapitel 4 werden die, während dieser Arbeit entstandenen Taskpoolvarianten beschrieben. Kapitel 5 enthält die Beschreibung der drei Programme, die implementiert wurden, um die Taskpools zu testen.

Im Kapitel 6 werden die Ergebnisse der Laufzeitmessungen vorgestellt. Kapitel 7 enthält eine Zusammenfassung dieser Arbeit.

Kapitel 2

Grundlagen

2.1 Paralleles Rechnen

Trotz der ständig steigenden Leistung der Mikroprozessoren reicht die Rechenleistung eines einzelnen Prozessors für viele Anwendungen mit hohem Berechnungsaufwand nicht aus, um diese in einer angemessenen Zeit ausführen zu können. Oft findet man solche Anwendungen im Bereich der Simulation von physikalischen Vorgängen. Der Einsatz von Parallelrechnern ist eine Möglichkeit, um eine höhere Rechenleistung für solche Anwendungen bereitzustellen. Ein Parallelrechner wird in [6] als

> *eine Ansammlung von Berechnungseinheiten (Prozessoren), die durch koordinierte Zusammenarbeit große Probleme schnell lösen können.*

definiert.

Beim Erstellen eines Programms für ein paralleles Rechensystem, versucht man die Berechnung in mehrere Teilaufgaben zu zerlegen, sodass diese unabhängig voneinander, gleichzeitig auf mehreren Prozessoren ausgeführt werden können. Damit versucht man, die Ausführungszeit eines Programms zu verkürzen. Um den Geschwindigkeitsgewinn eines parallelen Programms zu messen, verwendet man den Begriff *Speedup*. Der Speedup eines parallelen Programms mit der Laufzeit $T_p(n)$ wird in [6] definiert als:

> $S_p(n) = \frac{T^*(n)}{T_p(n)}$,
> *wobei p die Anzahl der Prozessoren zur Lösung des Problems der Größe n bezeichnet. $T^*(n)$ ist die Laufzeit einer optimalen sequenziellen Implementierung zur Lösung desselben Problems.*

Je schneller ein Programm mit mehreren Prozessoren ausgeführt werden kann, desto größer ist der Speedup. Im Idealfall entspricht der erzielte Speedup der Anzahl der verwendeten Prozessoren.

2.2 Taskpools

Für eine effiziente Parallelisierung ist es wichtig, die Teilaufgaben gleichmäßig auf alle beteiligten Prozessoren zu verteilen.

Bei einigen Applikationen kann diese Verteilung bereits vor dem Beginn der eigentlichen Berechnung erfolgen. So zum Beispiel, wenn das Programm gleiche und voneinander unabhängige Berechnungen auf unterschiedlichen Elementen einer Datenstruktur ausführt. In so einem Fall können die Elemente der Datenstruktur gleichmäßig auf alle beteiligten Prozessoren verteilt werden. Ein Verfahren, bei dem die Teilaufgaben beim Start des Programms verteilt werden, wird als *statisches Schedulingverfahren* bezeichnet.

Es gibt aber auch Programme, für die ein statisches Schedulingverfahren ungeeignet ist, weil ihr Berechnungsverhalten stark von der Eingabe abhängt. Man kann nicht vorhersagen, wie viel Zeit die Bearbeitung eines bestimmten Teils der Eingabe in Anspruch nehmen wird. Solche Applikationen werden als *irregulär* bezeichnet. Eine sinnvolle Verteilung der Teilaufgaben kann bei irregulären Applikationen erst während der Laufzeit erfolgen. Für irreguläre Applikationen ist daher ein *dynamisches Schedulingverfahren* nötig, um eine effiziente Parallelisierung zu erreichen.

Eine Methode, um dynamisches Scheduling zu realisieren, ist die Verwendung eines *Taskpools*. Dabei wird ein Programm in *Tasks* zerlegt. Tasks werden in [6] als „die kleinsten Einheiten der Parallelität“ definiert. Eine Task besteht aus einer Folge von Berechnungen und Daten, mit denen diese Berechnungen durchgeführt werden sollen. Ein Taskpool ist eine Datenstruktur, die diese Tasks verwaltet.

Ein Prozessor kann eine ausführbare Task im Taskpool ablegen und bei Bedarf Tasks aus dem Taskpool entnehmen. Der Taskpool regelt die Zuteilung der Tasks an die Prozessoren. Intern werden die Tasks in einer geeigneten Datenstruktur gespeichert. Dabei muss beachtet werden, dass die Zugriffsoperationen auf dieser Datenstruktur so einfach wie möglich sein sollen, um den zusätzlichen Aufwand, der bei der Verwendung eines Taskpools entsteht, klein zu halten. Solche Datenstrukturen sind zum Beispiel Warteschlangen (engl. Queues) oder Stacks.

Eine wesentliche Rolle spielt dabei die Organisation dieser Warteschlan-

gen. Ein Taskpool kann alle Tasks in einer zentralen Datenstruktur speichern, auf die alle Prozessoren zugreifen können. Diese Methode ist einfach zu implementieren und garantiert eine gleichmäßige Verteilung der Tasks zwischen den Prozessoren. Jeder Prozessor kann Tasks entnehmen, solange der Taskpool nicht leer ist. Ein Nachteil dieser Strategie ist jedoch, dass man einen Synchronisationsmechanismus benötigt, um den wechselseitigen Ausschluss beim Zugriff auf eine zentrale Warteschlange zu gewährleisten. Lock-Operationen sind jedoch teuer, was negative Auswirkungen auf die Performance des gesamten Programms hat. Bei einer steigenden Anzahl von Prozessoren kann sich eine zentrale Warteschlange zu einem Leistungsengpass entwickeln.

Eine andere Möglichkeit besteht in der Verwendung von dezentralen Warteschlangen. Dabei werden die Tasks auf mehrere Warteschlangen verteilt. Die Anzahl dieser Warteschlangen kann unterschiedlich sein. So kann man zum Beispiel jedem Prozessor seine eigene Warteschlange zuweisen, auf die nur der Besitzer zugreifen darf. Diese Variante entspricht aber einer statischen Lastenverteilung und ist somit für irreguläre Applikationen ungeeignet. Daher muss bei einer Verwendung von dezentralen Warteschlangen ein Transfer der Tasks zwischen den Prozessoren ermöglicht werden.

Bei der Realisierung des Transfers der Tasks zwischen den Warteschlangen kann man zwischen einer senderinitiierten und einer empfängerinitiierten Strategie wählen. Beim senderinitiierten Transfer werden die Tasks beim Einfügen in den Taskpool auf die Warteschlangen verteilt. Beim empfängerinitiierten Transfer versucht ein Prozessor, Tasks aus den Warteschlangen anderer Prozessoren zu entnehmen, wenn seine eigene Warteschlange leer ist (*task-stealing*). Beim empfängerinitiierten Tasktransfer ist die Anzahl der Verschiebeoperationen in der Regel kleiner als beim senderinitiierten Transfer, weil die Tasks nur dann von einer Warteschlange in die andere verschoben werden, wenn es wirklich notwendig ist. Einen ausführlichen Vergleich der beiden Strategien findet man in [3].

Die Taskpoolvarianten, die bei dieser Arbeit implementiert wurden, werden in 4.3 auf Seite 15 detailliert beschrieben.

2.3 OpenMP

Ein Ziel dieser Arbeit war es, bestehende Taskpool-Implementierungen nach OpenMP zu portieren. OpenMP ist ein von vielen führenden Software- und Hardware-Herstellern unterstützter Standard für die Programmierung von Parallelrechnern mit einem gemeinsamen Speicher. OpenMP ist keine eigenständige Programmiersprache, sondern nur eine Erweiterung der

Sprachen: C, C++ und FORTRAN. Der OpenMP-Standard spezifiziert eine Sammlung von Compilerdirektiven, Bibliotheksfunktionen und Umgebungsvariablen [1]. Die erste OpenMP-Spezifikation ist im Oktober 1998 erschienen. Die derzeit aktuelle Version ist 2.5 und wurde im Mai 2005 veröffentlicht.

OpenMP wurde entwickelt, mit dem Ziel, portabel zu sein. Für viele Prozessorarchitekturen und Betriebssysteme existieren bereits Compiler, die OpenMP unterstützen. Eine andere Stärke von OpenMP ist die Möglichkeit zur inkrementellen Parallelisierung. Mit OpenMP ist es möglich, bereits bestehende sequenzielle Programme beziehungsweise die rechenintensiven Teile dieser Programme schnell zu parallelisieren.

OpenMP verwendet Threads, um eine parallele Ausführung zu realisieren. Threads sind Kontrollflüsse eines Prozesses. Alle Threads eines Prozesses arbeiten im selben Adressraum. Anders als bei Low Level Thread-Bibliotheken (wie Pthreads) werden die Threads in einem OpenMP-Programm nicht explizit erzeugt oder beendet. Das Erzeugen und Beenden von Threads erfolgt implizit nach einem *fork-join* Prinzip. Ein OpenMP-Programm startet mit einem Master-Thread. Sobald ein, mit Hilfe der Compilerdirektiven definierter, paralleler Bereich erreicht wird, wird automatisch ein Team von Threads erzeugt, die zusammen mit dem Master-Thread den parallelen Abschnitt ausführen. Nach dem Ende des parallelen Abschnitts arbeitet der Master-Thread wieder allein weiter. Die Anzahl der Threads, die gemeinsam einen parallelen Bereich ausführen, kann entweder mit Bibliotheksfunktionen oder über Umgebungsvariablen festgelegt werden.

Ein paralleler Bereich wird im Programmcode mit einer *parallel*-Direktive gekennzeichnet. Auf diese Weise teilt man dem Compiler mit, dass der unmittelbar hinter der Direktive stehende Anweisungsblock parallel ausgeführt werden soll. Dabei führen alle Threads dieselben Berechnungen aus. Innerhalb eines parallelen Bereichs kann eine *single*-Direktive verwendet werden. Diese bewirkt, dass der mit dieser Direktive gekennzeichnete Anweisungsblock nur von einem Thread ausgeführt wird.

Zusätzlich bietet OpenMP die Möglichkeit, die Berechnungen, die innerhalb eines parallelen Bereichs ausgeführt werden sollen, auf die Threads so zu verteilen, dass jeder Thread nur einen Teil der Berechnungen ausführt. So lassen sich for-Schleifen, bei denen die Anzahl der Iterationen im voraus bestimmt werden kann, mit der *for*-Direktive parallelisieren. Eine andere Möglichkeit besteht in der Benutzung der *sections*-Direktive. Innerhalb eines *sections*-Blocks kann der Programmierer explizit angeben, welche Abschnitte innerhalb eines parallelen Bereichs, von welchen Threads ausgeführt werden können.

Um den Zugriff auf gemeinsame Variablen zu regeln, verfügt OpenMP über Lock-Variablen, die mit Hilfe der Bibliotheksfunktionen manipuliert werden können. Zusätzlich stehen dem Programmierer mehrere Compilerdirektiven für die Koordination von Threads zur Verfügung. So ermöglicht die *critical*-Direktive die Deklaration eines kritischen Abschnitts, der zu jedem Zeitpunkt nur von einem Thread ausgeführt werden darf. Die *barrier*-Direktive zwingt einen Thread solange zu warten, bis alle anderen Threads diese Barriere ebenfalls erreicht haben. Mit der *flush*-Direktive kann eine konsistente Sicht auf den gemeinsamen Speicher hergestellt werden. Eine *atomic*-Direktive bewirkt, dass die nachfolgende Zuweisung atomar ausgeführt wird, das heißt so, dass der Thread während der Ausführung dieser Zuweisung nicht unterbrochen wird.

Nähere Informationen zur OpenMP findet man in [2].

Kapitel 3

Speichermanager

Während der Abarbeitung eines Programm, welches Taskpools benutzt, müssen sehr oft neue Tasks erzeugt werden. Für diese muss Speicher reserviert und nach dem Ausführen der Task wieder freigegeben werden. Um Speicher dynamisch zu reservieren, muss ein Systemaufruf ausgeführt werden. Häufige Systemaufrufe können sich aber merkbar negativ auf die Performance eines Programms auswirken, insbesondere wenn das Betriebssystem zentrale Datenstrukturen zur Speicherverwaltung für alle Threads eines Programms benutzt.

Durch das Wiederverwenden von nicht mehr benötigten Speicherblöcken lässt sich der zusätzliche Aufwand, der beim Reservieren beziehungsweise Freigeben des Speichers entsteht, verringern. Das ist die Aufgabe eines Speichermanagers. Ein Speichermanager verwaltet mehrere Speicherblöcke einer gestimmten Größe.

Speichermanager bilden eine Zwischenschicht zwischen den Systemfunktionen für die Speicherverwaltung und einem Benutzerprogramm. Falls ein Programm Speicher dynamisch reservieren muss, sendet es eine Anfrage an den Speichermanager, anstatt an das Betriebssystem. Dazu ruft das Programm die Funktion mm_get() auf, die einen Zeiger auf einen freien Speicherblock liefert. Wird ein Speicherblock nicht mehr benötigt, kann er mit der Funktion mm_free() freigegeben werden.

Der Speichermanager reserviert bei der Initialisierung für jeden Thread eine bestimmte Anzahl an Speicherblöcken. Zusätzlich verwaltet der Speichermanager für jeden Thread eine Liste mit den vom Thread freigegebenen Speicherblöcken. Falls ein Speicherblock nicht mehr gebraucht wird, ruft ein Programm die entsprechende Funktion des Speichermanagers. Der Speichermanager gibt den Speicherblock nicht sofort frei, sondern speichert den Zei-

ger auf diesen Block in der Liste mit freigegebenen Speicherblöcken. Wenn ein Programm das nächste Mal einen Speicherblock benötigt, liefert der Speichermanager ein Element aus der Liste mit freien Speicherblöcken, anstatt neuen Speicher vom Betriebssystem anzufordern. Ist die Liste mit freigegebenen Speicherblöcken leer, liefert der Speichermanager einen während der Initialisierung allokierten Speicherblock. Wenn der bei Initialisierung des Speichermanagers reservierte Speicher nicht ausreicht, fordert der Speichermanager zusätzlichen Speicher von Betriebssystem an. Dabei werden immer mehrere Speicherblöcke auf einmal reserviert, um die Anzahl der Systemaufrufe zu verringern.

Da man immer dieselbe Menge an Speicherplatz benötigt, um eine Task abzuspeichern, reicht es aus, wenn ein Speichermanager nur Speicherblöcke einer bestimmten Größe verwaltet. Das erleichtert die Implementierung eines solchen Speichermanagers und ermöglicht schnellere Operationen zum Anfordern und Freigeben von Speicherblöcken.

In ihrem Artikel [4] beschreiben Korch und Rauber unter anderem mehrere Implementierungsmöglichkeiten für Speichermanager und untersuchen die Auswirkung der Speichermanager auf die Performance der Taskpools. Der von mir verwendete Speichermanager basiert auf dem Speichermanager *memdbk* aus der Pthreads-Implementierung von Korch und Rauber. Der Speichermanager *memdbk* hat allerdings den Nachteil, dass er nur die, während der Initialisierung allokierte Speichermenge, verwenden kann. Wenn dieser Speicher verbraucht ist, kann der Speichermanager keine Speicherblöcke mehr liefern. Daher habe ich den Speichermanager so erweitert, dass er zusätzlichen Speicher vom Betriebssystem anfordern kann, wenn die vorreservierte Speichermenge nicht ausreicht.

Um die Auswirkung der Speichermanager auf die Performance der Taskpools zu untersuchen, habe ich den von Korch und Rauber implementierten Speichermanager *membs* verwendet. Dieser kapselt die Systemaufrufe malloc() und free(). Wenn Speicher angefordert wird, allokiert mm_get() einen Speicherblock mit malloc(). mm_free() gibt den Speicher mit Hilfe von free() frei.

Kapitel 4

Taskpools

In diesem Kapitel beschreibe ich die Taskpoolvarianten, die ich während dieser Arbeit implementiert habe. Einige davon wurden von der Pthreads-Implementierung [4] portiert, andere entstanden durch Erweiterungen der portierten Taskpools. Im Abschnitt 4.1 wird die Struktur beschrieben, in der die internen Daten eines Taskpools gespeichert werden. Der Abschnitt 4.2 befasst sich mit der Programmierschnittstelle, die alle Taskpoolvarianten zur Verfügung stellen. Im Abschnitt 4.2.5 werden die Unterschiede zwischen der Pthreads-Implementierung und meinen Taskpool-Implementierungen erläutert. Der Abschnitt 4.3 enthält die Beschreibung der Taskpoolvarianten, die während dieser Arbeit untersucht wurden.

4.1 Taskpool Datenstruktur

Die Abbildung 4.1 zeigt die Typdefinition für die Struktur, in der die internen Daten eines Taskpools gespeichert werden. Das erste Element dieser Struktur enthält die Adresse des Speicherblocks, der die Daten enthält, die sich von Taskpoolvariante zur Taskpoolvariante unterscheiden. Das sind zum Beispiel die Zeiger auf die Taskqueues, die Lock-Variablen, usw. Da verschiedene Taskpoolvarianten unterschiedlich aufgebaut sind, weil sie z. B. unterschiedlich viele Queues zum Speichern der Tasks verwenden, können diese Daten nicht durch eine zentrale Datenstruktur, die für alle Taskpoolvarianten gleich ist, beschrieben werden. Das zweite Element von tpool_t ist ein Zeiger auf den Speichermanager, der den Speicher für die Taskqueue-Einträge verwaltet. Die Taskqueues sind in allen Taskpoolvarianten als einfach verkettete Listen implementiert. Einzelne Knoten dieser Listen enthalten Zeiger auf die Taskdaten. Dieser Speichermanager verwaltet Speicherblöcke, die zum Abspeichern der Knoten dieser Listen dienen. Das nächste Element

```
1 typedef struct
2 {
3     void *data;
4     mm_t *tl_item_mm;
5     mm_t *task_mm;
6     int num_threads;
7     int num_waiting_threads;
8 } tpool_t;
```

Abbildung 4.1: Inhalt der Taskpooldatenstruktur.

zeigt ebenfalls auf einen Speichermanager. Dieser kann von der Applikation, die diesen Taskpool verwendet, benutzt werden, um den Speicher für die Taskdaten zu verwalten. Das vierte Element gibt die Anzahl der Threads an, die diesen Taskpool verwenden und das letzte Element wird benutzt, um die Anzahl der wartenden Threads zu zählen.

4.2 Schnittstelle

Alle während dieser Arbeit entwickelten Taskpoolvarianten haben eine einheitliche Schnittstelle. Sie ermöglicht einen einfachen Austausch der verschiedenen Taskpoolvarianten, ohne dabei den Quellcode der Applikation, die einen Taskpool verwendet, ändern zu müssen. Es ist außerdem möglich, gleichzeitig mehrere Taskpools in einem Programm zu benutzen. Alle Taskpool-Implementierungen verfügen über Funktionen zum Initialisieren des Taskpools, Einfügen und Entnehmen der Tasks und zur Freigabe der reservierten Ressourcen.

4.2.1 Initialisierung

Damit ein Programm Taskpools nutzen kann, muss zuerst festgelegt werden, welche Programmabschnitte als Tasks aufgefasst werden können. Zusätzlich muss eine Datenstruktur definiert werden, die diese Tasks beschreibt. Diese Datenstruktur muss alle Daten beinhalten, die für die Ausführung einer Task gebraucht werden. Bei der Cholesky-Faktorisierung zum Beispiel berechnet eine Task eine Spalte der Matrix. Die Datenstruktur, die die Task beschreibt, muss daher den Index dieser Spalte enthalten. Die Größe dieser Datenstruktur muss während der Initialisierung bekannt sein, damit der Speichermanager mehrere Blöcke dieser Größe vorreservieren kann. Die Initialisierung eines Taskpools erfolgt durch den Aufruf der Funktion tpool_init(). Diese Funktion reserviert den benötigten Speicher und initialisiert die Taskpool-

daten mit Hilfe der übergebenen Parameter. tpool_init () liefert bei erfolgreicher Ausführung einen Zeiger auf eine Struktur, mit deren Hilfe man auf den Taskpool zugreifen kann. Diese Struktur wurde im Abschnitt 4.1 beschrieben. Die Abbildung 4.5 zeigt ein typisches Programm, das einen Taskpool verwendet.

Ein Beispiel für die Benutzung von tpool_init () ist in der Abbildung 4.2 gezeigt. Das erste Argument gibt die Anzahl der Threads an, die parallel auf den Taskpool zugreifen sollen. Bei Taskpools mit dezentralen Warteschlangen muss die Anzahl der Threads bereits bei der Initialisierung bekannt sein, um die Warteschlangen initialisieren zu können. Das zweite Argument gibt die Speichermenge an, die benötigt wird, um eine Task zu speichern. Diese Angabe wird für die Initialisierung des Speichermanagers, der den Speicher für die Tasks verwaltet, benötigt. Das dritte Argument bestimmt die Anzahl der Speicherblöcke, die der Speichermanager vorreservieren soll.

```
1 tpool_t *tp = tpool_init(4, sizeof(args_t), 1024);
```

Abbildung 4.2: Beispiel für einen Aufruf von tpool_init ()

4.2.2 Tasks einfügen

Die Abarbeitung eines Programms, welches Taskpools benutzt, kann in zwei Phasen unterteilt werden: die Initialisierungsphase und die Arbeitsphase. Während der Initialisierungsphase wird die eigentliche Berechnung vorbereitet. Es wird Speicher allokiert, Daten werden initialisiert und Tasks eingefügt, die die Berechnung starten sollen. Während der Arbeitsphase findet die eigentliche Berechnung statt. Die Tasks werden aus dem Taskpool entnommen und abgearbeitet. Für jede Phase bieten die Taskpools jeweils eine Funktion zum Einfügen der Tasks. Da während der Initialisierungsphase nur ein Thread arbeitet, werden keine Synchronisationsoperationen bei Einfügen der Tasks benötigt. Die entsprechende Funktion heißt tpool_initial_put (). Beim Einfügen einer Task während der Arbeitsphase muss der wechselseitige Ausschluss beim Zugriff auf gemeinsame Taskqueues sichergestellt werden, weil während dieser Phase mehrere Threads gleichzeitig auf einen Taskpool zugreifen können. Die Funktion zum Einfügen von Tasks während der Arbeitsphase heißt tpool_put(). Abbildung 4.3 zeigt ein Beispiel für den Aufruf von tpool_put().

Das erste Argument ist der Zeiger auf die Datenstruktur des Taskpools, in dem die Task abgelegt werden soll. Dieser Zeiger wird von der Funktion tpool_init () geliefert. Das zweite Argument ist die Nummer des Threads, der

```
1 tpool_put(tp, my_number, &data);
```

Abbildung 4.3: Beispiel für einen Aufruf von tpool_put()

diese Funktion aufruft. Letztes Argument ist ein Zeiger auf die Taskdaten. Dieser Zeiger wird in einer Taskqueue gespeichert. Tasks können komplexe Strukturen sein, die mehrere Elemente enthalten. Damit diese Daten nicht jedes Mal kopiert werden müssen, verwalten die Taskpools Zeiger auf die Taskdaten.

4.2.3 Tasks entnehmen

Mit der Funktion tpool_get() können Tasks aus dem Taskpool entnommen werden. Falls zum Zeitpunkt des Aufrufs von tpool_get() keine Tasks im Taskpool sind, blockiert die Funktion und kehrt erst dann zurück, wenn entweder neue Tasks eingefügt wurden, oder der Taskpool leer ist. Um festzustellen, dass der Taskpool leer ist, reicht es nicht zu prüfen, ob die Taskqueues noch Tasks enthalten oder nicht. Auch wenn zu einem Zeitpunkt sich keine Tasks mehr im Taskpool befinden, ist es dennoch möglich, dass ein oder mehrere Threads weiterarbeiten und neue Tasks erzeugen, die sie später in den Taskpool einfügen. Erst wenn alle Threads mit ihrer Arbeit fertig sind und auf neue Tasks warten, ist ein Taskpool endgültig leer. Die Abbildung 4.4 zeigt ein Beispiel für den Aufruf von tpool_get(). Wie bei tpool_put() ist das erste Argument ein Zeiger auf die Datenstruktur des Taskpools und das zweite die Nummer des Threads, der die Funktion aufruft. Das letzte Argument ist ein Zeiger auf einen Speicherblock, in den der Zeiger auf die Taskdaten geschrieben werden soll. Falls es tpool_get() gelingt eine Task aus dem Taskpool zu entnehmen, wird der Zeiger auf die Taskdaten in den durch das letzte Argument angegebenen Speicherblock geschrieben und die Konstante TPOOL_OK zurückgegeben. Wenn der Taskpool leer ist, wird TPOOL_EMPTY zurückgegeben.

```
1 if(TPOOL_OK == tpool_get(tp, my_number, &args_buffer))
2     run_task(args_buffer);
```

Abbildung 4.4: Beispiel für einen Aufruf von tpool_get()

4.2.4 Andere Funktionen

Die Funktion tpool_reset() bringt einen Taskpool wieder in den Ausgangszustand, d.h. den Zustand, in dem der Taskpool, direkt nach der Initialisierung war. Mit tpool_destroy() wird der, für einen Taskpool, reservierter Speicher wieder freigegeben.

```
1  task_data_t *task_data;
2  tpool_t *pool = tpool_init(num_threads, sizeof(task_data_t));
3  task_data = generate_initial_task();
4  tpool_put(pool, 0, task_data);
5  #pragma omp parallel shared(pool)
6  {
7    task_data_t *my_task_data;
8    int me = omp_get_thread_num();
9    while(TPOOL_EMPTY != tpool_get(pool, me, &my_task_data);
10     do_work(my_task_data);
11 }
12 tpool_destroy(pool);
```

Abbildung 4.5: Beispiel für die Verwendung eines Taskpools

4.2.5 Unterschiede zur Pthreads-Implementierung

Obwohl ich mich bei der Portierung von Taskpools an der Pthreads-Implementierung von Korch und Rauber orientiert habe, gibt es Unterschiede zwischen der Pthreads-Implementierung und meinen Taskpool-Implementierungen.

So gibt es zum Beispiel Unterschiede in der Programmierschnittstelle, die die Taskpools zur Verfügung stellen. Zusätzlich zu den Funktionen zum Initialisieren des Taskpools, Einfügen und Entnehmen der Tasks verfügen die Taskpools aus der Pthreads-Implementierung über die Funktion tpool_run(), die nichts anderes tut, als Tasks aus dem Pool zu entnehmen und auszuführen, bis der Taskpool leer ist. Nach der Initialisierung des Taskpools und dem Einfügen erster Tasks rufen alle Threads tpool_run() auf. Nachdem diese Funktion ausgeführt wurde, ist der Taskpool leer und die Berechnung ist abgeschlossen. Bei der Portierung habe ich auf die tpool_run() Funktion verzichtet. Die Operationen, die während einer Task ausgeführt werden und die Art, wie die Threads die Tasks aus dem Pool entnehmen bestimmt das Anwendungsprogramm, das die Taskpools benutzt. Ein Taskpool dient lediglich als ein Behälter für die Taskdaten.

Um die Funktion tpool_run() zu realisieren, speichern die Taskpools aus der Pthreads-Implementierung für jede Task neben den Daten, die für die

Ausführung dieser Task benötigt werden, auch Zeiger auf die Funktion, die ausgeführt werden soll. Im Taskpool werden also sowohl die Informationen über die Operationen, die eine Task ausführen soll, als auch über die Daten, die benötigt werden, um diese Operationen auszuführen, abgelegt. Die von mir implementierten Taskpools speichern dagegen nur die Daten, die für die Ausführung einer Task benötigt werden, enthalten aber keine Informationen über die Operationen, die mit diesen Daten ausgeführt werden sollen.

Die Funktion tpool_reset () ist in der Pthreads-Implementierung nicht enthalten. Sie ermöglicht, dass eine Taskpoolinstanz in einem Programm mehrfach verwendet werden kann. Diese Funktionalität wird für das Labyrinth-Problem benötigt.

4.3 Taskpoolvarianten

Insgesamt wurden sieben Taskpoolvarianten untersucht. Die Taskpools sq1, sdq1 und dq8 habe ich von [4] übernommen und nach OpenMP portiert. Alle anderen Taskpoolvarianten entstanden durch Veränderungen und Erweiterungen dieser drei Taskpools. In diesem Abschnitt werden die Unterschiede zwischen den einzelnen Taskpoolvarianten beschrieben. Die Tabelle 4.1 gibt einen Überblick über die implementierten Taskpoolvarianten.

4.3.1 Taskpools mit zentralen Taskqueues

Es wurden zwei Varianten mit jeweils einer zentralen Taskqueue implementiert: *sq1* und *sq2*. Alle Tasks werden bei diesen beiden Taskpoolvarianten in einer zentralen Queue gespeichert. Da alle Threads auf die Taskqueue zugreifen dürfen, wird der Zugriff auf die Queue durch eine Lock-Variable geschützt. Jedes Mal, wenn eine Task in die Queue eingefügt wird oder aus der Queue entnommen wird, wird der Zugriff für alle anderen Threads gesperrt. Wie alle anderen Taskpoolvarianten verwenden *sq1* und *sq2* einfach verkettete Listen, um die Tasks zu speichern. Der Speicher für die Listeneinträge wird mit Hilfe eines Speichermanagers reserviert.

Der einzige Unterschied zwischen diesen beiden Implementierungen liegt in der Reihenfolge, in der die Tasks aus dem Pool entnommen werden. Bei der ersten Variante (*sq1*) werden die Tasks in der LIFO-Reihenfolge (*last-in, first-out*) entnommen. Das heißt, dass der Taskpool bei jedem Aufruf von tpool_get () die Task liefert, die zuletzt eingefügt wurde. Bei der zweiten Variante werden die Tasks in derselben Reihenfolge entnommen, in der sie eingefügt wurden (*first-in, first-out*).

4.3.2 Taskpools mit dezentralen Taskqueues

Alle Taskpoolvarianten mit dezentralen Taskqueues verwenden zwei Arten von Taskqueues: private und öffentliche. Für jeden Thread wird eine private Taskqueue verwaltet. Der Zugriff auf eine private Taskqueue ist ausschließlich dem Besitzer-Thread gestattet. Auf diese Weise greift nie mehr als ein Thread gleichzeitig auf eine private Queue, sodass kein wechselseitiger Ausschluss für diese Art von Taskqueues realisiert werden muss. Dadurch können Tasks schneller eingefügt oder entnommen werden, als bei einer zentralen Taskqueue.

Neben den privaten Queues benötigt ein Taskpool auch öffentliche Taskqueues, auf die alle Threads des Programms zugreifen dürfen. Über die öffentlichen Taskqueues können Tasks zwischen den Threads ausgetauscht werden. Wenn ein Thread viele Tasks in seiner privaten Taskqueue hat, verschiebt er einige davon in eine öffentliche Taskqueue. Andere Threads können diese Tasks aus einer öffentlichen Queue entnehmen, wenn ihre eigene private Taskqueue leer ist. Auf diese Weise wird die Arbeit auf mehrere Threads verteilt.

Um die Anzahl der Zugriffe auf die öffentlichen Queues zu reduzieren, ist es sinnvoll mehrere Tasks auf ein mal aus der öffentlichen in die private Taskqueue zu verschieben. Dazu werden mehrere Tasks in Gruppen zusammengefasst. Die Taskqueues speichern nicht einzelne Tasks, wie es bei den Taskpoolvarianten mit einer zentralen Queue der Fall war, sondern Taskgruppen, die aus mehreren Tasks bestehen.

Da mehrere Threads versuchen könnten, gleichzeitig auf eine öffentliche Queue zuzugreifen, muss der wechselseitige Ausschluss beim Zugriff auf eine öffentliche Taskqueue sichergestellt werden. Dazu werden, wie bei den Taskpoolvarianten mit einer zentralen Taskqueue, Lock-Variablen verwendet. Für jede öffentliche Queue verwaltet der Taskpool eine Lock-Variable. Diese bleibt gesperrt, solange ein Thread Änderungen an der Queue vornimmt.

sdq1

Die Taskpool-Variante *sdq1* verwaltet neben den privaten Taskqueues eine öffentliche Taskqueue. Da die Zugriffe auf die öffentliche Taskqueue synchronisiert werden müssen und deshalb nicht so effizient wie die Zugriffe auf eine private Queue sind, versucht *sdq1* die Anzahl Operationen auf der öffentlichen Taskqueue zu reduzieren. Falls eine neue Task in den Taskpool eingefügt werden muss, wird sie in der privaten Taskqueue des Threads, der

diese Task erzeugt hat, gespeichert. Um zu verhindern, dass einige Threads sehr viele Tasks in ihren privaten Queues sammeln, während andere Threads auf Arbeit warten müssen, wird die Größe der privaten Queues auf zwei Taskgruppen begrenzt. Ist die private Taskqueue voll, wird eine Taskgruppe aus der privaten in die öffentliche Taskqueue verschoben.

Wenn ein Thread eine Task anfordert, versucht *sdq1* eine Task aus der privaten Queue des aufrufenden Threads zu entnehmen. Ist die private Taskqueue des Threads leer, muss zunächst eine Taskgruppe aus der öffentlichen in die private Taskqueue verschoben werden. Falls auch die öffentliche Taskqueue leer ist, wartet der Thread, bis neue Tasks in die öffentliche Queue eingefügt werden.

dq8

Eine einzige öffentliche Taskqueue stellt bei einer steigenden Anzahl der Threads einen Leistungsengpass dar. Je höher die Anzahl der Threads, desto größer ist die Wahrscheinlichkeit, dass mehrere Threads gleichzeitig versuchen auf die öffentliche Taskqueue zuzugreifen. Die Verwendung von mehreren öffentlichen Taskqueues könnte daher die Performance eines Taskpools verbessern. Dazu verwaltet *dq8* für jeden Thread eine öffentliche Queue. Auf diese Weise verfügt jeder Thread über zwei Taskqueues: eine private, auf die nur er zugreifen darf und eine öffentliche, auf die alle Threads zugreifen dürfen. Wie bei sdq1 ist die Größe einer privaten Taskqueue begrenzt.

Stellt ein Thread beim Einfügen fest, dass die private Taskqueue voll ist, verschiebt er eine Taskgruppe aus der privaten in seine öffentliche Taskqueue und fügt anschließend die neue Task in die private Taskqueue ein. Wenn beim Entnehmen einer Task die private Queue leer ist, versucht der Thread eine Taskgruppe aus seiner eigenen öffentlichen Taskqueue zu entnehmen. Ist diese Queue ebenfalls leer, versucht der Thread eine Taskgruppe aus der öffentlichen Taskqueue eines anderen Threads zu 'stehlen'. Dazu untersucht tpool_get() nacheinander die öffentlichen Queues der anderen Threads, angefangen mit der Queue des Threads mit der nächsthöheren Nummer. Sobald eine nicht leere Taskqueue gefunden wird, wird aus dieser eine Taskgruppe in die private Queue des aufrufenden Threads verschoben. Dieses Vorgehen wird als *task stealing* bezeichnet.

dq9

Sowohl *sdq1* als auch *dq8* haben den Nachteil, dass die Taskgruppen öfter als nötig aus den privaten in die öffentlichen Taskqueues verschoben wer-

den. Beim Verschieben einer Taskgruppe müssen zwar nur wenige Zeiger verändert werden, was keine spürbare Auswirkung auf die Performance hat. Allerdings muss während des Verschiebens die entsprechende öffentliche Queue mit Hilfe der Lock-Variablen gesperrt werden. Dies könnte sich negativ auf die Performance des Programms auswirken, wenn Taskgruppen häufig verschoben werden müssen.

Bei *dq9* müssen die Tasks nur dann aus der privaten in die öffentliche Queue verschoben werden, wenn es tatsächlich nötig ist. Wie bei *dq8* hat jeder Thread jeweils eine öffentliche und eine private Taskqueue.

Auch bei *dq9* werden die Tasks immer zuerst in die private Taskqueue eingefügt. Im Gegensatz zu *dq8* ist die Größe der privaten Taskqueue bei *dq9* nicht auf zwei Taskgruppen beschränkt. Die private Taskqueue kann bei *dq9* beliebig viele Elemente enthalten. Ein Thread verschiebt eine Taskgruppe erst dann in die öffentliche Queue, wenn er beim Einfügen feststellt, dass es Threads gibt, die auf Arbeit warten und seine eigene öffentliche Queue leer ist.

Der Nachteil dieser Vorgehensweise ist, dass bei jedem tpool_put() Aufruf geprüft werden muss, ob es wartende Threads gibt und ob die öffentliche Queue leer ist oder nicht. Der zusätzliche Aufwand, der bei dieser Abfrage entsteht, ist aber gering. Dafür lassen sich unnötige Zugriffe auf öffentliche Queues vermeiden.

dq9-1

Wie *dq8* und *dq9* verwaltet die Taskpoolvariante *dq9-1* für jeden Thread jeweils eine private und eine öffentliche Queue. Wie *dq9* fügt *dq9-1* neue Tasks nur dann in eine öffentliche Queue, wenn diese leer ist und wenn es Threads gibt, die auf Tasks warten. Anders als *dq9* prüft *dq9-1* dabei auch, ob die private Queue noch Tasks enthält. Falls ja, wird die neue Task in die öffentliche Queue eingefügt. Wenn die private Queue jedoch leer ist, wird die neue Task in die private Queue eingefügt, unabhängig davon, ob es wartenden Threads gibt oder nicht.

Auch beim Entnehmen der Tasks gibt es Unterschiede zwischen *dq9-1* und den anderen Taskpoolvarianten. Wenn die private Queue eines Threads leer ist, muss er Tasks aus den öffentlichen Queues entnehmen. Wenn auch die öffentlichen Queues des Taskpools leer sind, muss der Thread warten, bis neue Tasks eingefügt werden. Um diese Situationen zu vermeiden, versucht die Taskpoolvariante *dq9-1* sicherzustellen, dass die private Queue nicht leer wird. Dazu wird für jeden Thread die Anzahl der Taskgruppen in seiner privaten Queue mitgezählt. Falls die Anzahl der Taskgruppen unter einen be-

Name	**Zugriff**	**Anzahl Queues (priv./öffentl.)**	**Größe der priv. Queue**	**Taskstealing**
sq1	LIFO	1 (0/1)	-	-
sq2	FIFO	1 (0/1)	-	-
sdq1	LIFO	n + 1 (n/1)	2 Taskgruppen	-
dq8	LIFO	2n (n/n)	2 Taskgruppen	priv. Queue leer
dq9	LIFO	2n (n/n)	unbegrenzt	priv. Queue leer
dq9-1	LIFO	2n (n/n)	unbegrenzt	priv. Queue fast leer
dq10	FIFO	2n (n/n)	unbegrenzt	priv. Queue leer

Tabelle 4.1: Überblick über die verschiedenen Taskpoolvarianten. (n steht für die Anzahl der verwendeten Threads)

stimmten Grenzwert sinkt, versucht der Thread, Tasks aus einer öffentlichen Queue zu "stehlen". Um zu vermeiden, dass ein Thread beim Versuch eine Task aus der öffentlichen Queue zu "stehlen" blockiert wird, weil die Lock-Variable, die diese Queue schützt, gerade durch einen anderen Thread belegt ist, wird hier die nicht blockierende OpenMP-Funktion omp_test_lock() verwendet. Wenn die Lock-Variable zur Zeit belegt ist, blockiert der Thread nicht, sondern versucht eine Task aus einer anderen öffentlichen Queue zu entnehmen. Findet der Thread eine freie öffentliche Queue, die noch Tasks enthält, verschiebt er eine Taskgruppe aus dieser Queue in seine private Queue. Wenn es dem Thread nicht gelingt auf diese Weise eine Taskgruppe aus einer öffentlichen Queue zu "stehlen", blockiert er aber nicht, sondern entnimmt weiterhin Tasks aus seiner privaten Taskqueue. Erst wenn alle öffentlichen Queues und die private Taskqueue leer sind, muss der Thread, warten, bis neue Tasks in den Pool eingefügt werden.

dq10

Alle bisher beschriebenen Taskpoolvarianten mit dezentralen Warteschlangen liefern die Tasks in der LIFO-Reihenfolge. dq10 ist eine Taskpool-Variante, die genau so aufgebaut ist, wie dq9, allerdings mit dem Unterschied, dass die Tasks in derselben Reihenfolge entnommen werden, in der sie eingefügt wurden.

Kapitel 5

Applikationen

In diesem Kapitel werden drei Applikationen beschrieben, die ich implementiert habe, um die im Kapitel 4 beschriebenen Taskpools zu testen. Alle drei gehören zur Klasse der irregulären Algorithmen und sind daher schwierig zu parallelisieren.

5.1 Quicksort

5.1.1 Problembeschreibung

Ein in der Informatik oft vorkommendes Problem ist das Sortieren. Die Aufgabe besteht darin, alle Elemente einer Folge in eine bestimmte Reihenfolge zu bringen. Quicksort ist einer der vielen Algorithmen, die entwickelt wurden, um dieses Problem zu lösen. Der Algorithmus wurde bereits 1962 von C.A.R. Hoare vorgestellt und wird wegen seiner Einfachheit und einer guten Performance häufig verwendet.

Quicksort basiert auf dem Prinzip "Teile und herrsche" (*divide and conquer*). Dabei wird ein Problem in mehrere kleinere Teilprobleme zerlegt, die einfacher zu handhaben sind. Die Lösungen der Teilprobleme werden verwendet, um die Lösung des Gesamtproblems zu finden. So wird bei Quicksort die zu sortierende Folge in zwei Teilfolgen aufgeteilt, die dann unabhängig voneinander mit dem selben Verfahren sortiert werden. Für das Aufteilen der Folgen wird zunächst ein Element aus der Folge gewählt (*Pivotelement*). Alle Elemente die kleiner sind als das Pivotelement bilden die erste Teilfolge. Die zweite Teilfolge besteht dann aus den Elementen, die größer sind als das Pivotelement. Diese Aufteilung wird auch *Partitionieren* genannt. Der Algorithmus bricht ab, wenn die Teilfolgen die Länge 0 haben. Meis-

tens werden aber die Teilfolgen, die nur aus wenigen Elementen bestehen, mit einem anderen Verfahren sortiert. Häufig wird Insertionsort verwendet. Die bei der Aufteilung entstehenden Teilfolgen sind unabhängig von einander und können somit parallel sortiert werden. Da die Teilfolgen aber erst zur Laufzeit erzeugt werden, benötigt man hier ein dynamisches Schedulingverfahren, um Quicksort effizient zu parallelisieren. Es bietet sich an, einen Taskpool zu verwenden, um Teile der Berechnung auf mehrere Threads zu verteilen.

5.1.2 Implementierung mit Taskpools

Die Programmstruktur entspricht im Wesentlichen einem sequenziellen rekursiven Quicksort. Anstelle der rekursiven Aufrufe werden neue Tasks erzeugt und in den Taskpool eingefügt. Eine Task entspricht somit dem Partitionieren einer Teilfolge. Die Berechnung beginnt mit einer Task, die die gesamte Zahlenfolge partitioniert. Diese wird in der Initialisierungsphase in den Taskpool eingefügt. In der Arbeitsphase entnimmt ein Thread diese Task und führt sie aus. Dabei entstehen zwei Teilfolgen, die zwei neue Tasks bilden. Eine Task wird sofort von demselben Thread ausgeführt. Die andere Task wird im Taskpool abgelegt, sodass ein anderer Thread diese Task entnehmen und bearbeiten kann. Ist die Länge einer Folge kleiner als ein bestimmter Grenzwert, wird diese Folge mit Insertionsort sortiert, sodass nach dem Bearbeiten dieser kleinen Teilfolgen keine neuen Tasks entstehen. Laufzeitmessungen haben ergeben, dass der Algorithmus die besten Ergebnisse erreicht, wenn die Folgen mit weniger als 100 Elementen mit Insertionsort sortiert werden. Nachdem alle Tasks abgearbeitet wurden und der Taskpool leer ist, ist die ursprüngliche Zahlenfolge sortiert. Bei der Implementierung habe ich bei der Wahl des Pivotelements die Median-von-drei Strategie verwendet. Dabei werden das erste, mittlere und das letzte Element der Eingabefolge betrachtet. Das Pivotelement ist das zweitgrößte Element dieser Menge.

5.2 Labyrinth

5.2.1 Problembeschreibung

Beim Labyrinth-Problem besteht die Aufgabe darin, den kürzesten Weg durch ein Labyrinth zu finden. Die klassische Strategie, um den Ausweg aus einem Labyrinth zu finden, entspricht einer Tiefensuche: Man verfolgt einen Pfad solange, bis man entweder den Ausweg findet, oder in eine Sackgasse läuft. Endet der Weg in einer Sackgasse, kehrt man zu einer Stelle

zurück, an der ein anderer Weg ausgewählt werden kann, und geht von dort aus weiter. Aber wenn man auf diese Weise einen Weg findet, hat man das Problem noch nicht gelöst, weil dieser Weg möglicherweise nicht der kürzeste ist. Um sicherzustellen, dass man die richtige Lösung gefunden hat, müssen alle Wege durch das Labyrinth ausprobiert werden. Je nach Größe und der Beschaffenheit des Labyrinths können sehr viele Wege zum Ausgang führen.

Daher habe ich mich bei der Implementierung für eine andere Methode entschieden: die Breitensuche. Dabei werden für jede besuchte Zelle als nächstes alle Nachbarzellen (Labyrinthzellen, die direkt mit der aktuellen Zelle verbunden sind) besucht. Sobald man das Ziel erreicht hat, hat man den kürzesten Weg gefunden.

Der Algorithmus startet vom Ausgang des Labyrinths. Für jede Zelle, die vom Ausgang aus erreichbar ist, wird die Entfernung dieser Zelle vom Ausgang in einer Matrix gespeichert, bis der Eingang in das Labyrinth erreicht wird. Wurde der Eingang erreicht, kann der kürzeste Pfad mit Hilfe der berechneten Entfernungen bestimmt werden. Dazu besucht man ausgehen vom Eingang in das Labyrinth als nächstes immer die Zelle, deren Entfernung vom Ausgang um eins kleiner ist, als die der aktuellen Zelle.

5.2.2 Implementierung mit Taskpools

Das Programm benutzt zwei Taskpools und eine zusätzliche Matrix, um die Entfernungen der einzelnen Zellen vom Ausgang zu speichern. Da die Berechnung der Entfernungsmatrix, der aufwendigste Teil des Programms ist, wird sie mit Hilfe der Taskpools parallelisiert.

Man könnte für jede Labyrinthzelle eine Task erzeugen. In diesem Fall wäre allerdings der Aufwand für das Erzeugen einer Task, das Einfügen dieser in den Taskpool und das Entnehmen dieser Task viel größer, als der Aufwand, der bei Ausführung dieser Task entsteht. Die Threads wären die meiste Zeit mit der Taskpoolverwaltung beschäftigt, während die eigentliche Berechnung nur einen kleinen Teil der Laufzeit ausmachen würde. Daher bearbeitet eine Task mehrere Labyrinthzellen auf ein mal. Dabei wird die Entfernung der Zellen vom Ausgang gespeichert und alle Nachbarzellen ermittelt. Diese Nachbarzellen, die in der nächsten Iteration abgearbeitet werden sollen, werden zunächst in einem Array zwischengespeichert. Ist das Array voll, wird für die im Array gespeicherten Labyrinth-Zellen eine neue Task erzeugt und in den Taskpool eingefügt. Für die Laufzeitmessungen wurden Arrays mit 100 Elementen verwendet, d.h., in jeder Task wurden maximal 100 Labyrinth-Zellen bearbeitet.

In der Initialisierungsphase wird eine Task mit den Koordinaten des

Ausgangs in den Taskpool eingefügt. Die Arbeitsphase wird in mehrere Iterationen aufgeteilt. In einer Iteration werden alle Zellen mit der Distanz d zum Ausgang bearbeitet, in der nächsten Iteration dann die alle Zellen mit der Distanz $d + 1$. Um sicherzustellen, dass in einer Iteration nur die Zellen mit der gleichen Entfernung vom Ausgang bearbeitet werden, werden zwei Taskpools benutzt: $tp1$ und $tp2$. Die Tasks, die in der aktuellen Iteration bearbeitet werden sollen, befinden sich in $tp1$. Der zweite Taskpool $tp2$ dient zum Sammeln der Tasks für die nächste Iteration. Nach jeder Iteration werden die Taskpools vertauscht, sodass $tp1$ alle Tasks enthält, die in der neuen Iteration ausgeführt werden müssen. $tp2$ ist nach dem Vertauschen der Taskpools leer und kann zum Sammeln neuer Tasks verwendet werden.

5.3 Cholesky-Faktorisierung

5.3.1 Problembeschreibung

Das dritte irreguläre Problem ist die Cholesky-Faktorisierung von dünnbesetzten Matrizen. Dieses Verfahren wird benutzt, um lineare Gleichungssysteme der Form $Ax = b$ zu lösen. Die Matrix A wird dabei in das Produkt aus einer unteren Dreiecksmatrix L und ihrer Transponierten L^T umgeformt: $A = LL^T$. Auf diese Weise entstehen zwei neue Gleichungssysteme: $Ly = b$ und $L^T x = y$. Diese können aufgrund ihrer speziellen Form einfacher gelöst werden als das ursprüngliche Gleichungssystem.

Die Cholesky-Faktorisierung lässt sich allerdings nur mit symmetrischen positiv-definiten Matrizen durchführen. Für A muss daher gelten: $A = A^T$ und $Ax > 0$ für alle $x \in \mathbb{R}^n$ mit $x \neq 0$.

Das gesamte Verfahren zur Lösung des Gleichungssystems besteht aus mehreren Schritten. Im ersten Schritt wird das Gleichungssystem so umgeformt, dass die *Fill-in-Effekte*, die bei der Faktorisierung entstehen, reduziert werden. Ein *Fill-in-Effekt* tritt auf, wenn die Matrix L Nichtnulleinträge an Stellen hat, an denen die Eingabematrix A Nulleinträge hatte. Im zweiten Schritt wird die Besetzungsstruktur der Matrix L bestimmt, d.h., es werden die Positionen der Nichtnulleinträge ermittelt. Zusätzlich wird für die Matrix L eine geeignete Datenstruktur initialisiert. Diese Datenstruktur wird im Abschnitt 5.3.3 beschrieben. Der dritte Schritt des Verfahrens ist die numerische Faktorisierung. Dabei wird die Matrix L berechnet. Im letzten Schritt werden die resultierenden Gleichungssysteme gelöst. Ich habe nur den dritten Schritt des Verfahrens implementiert: die numerische Faktorisierung. Dieser Schritt nimmt die meiste Rechenzeit in Anspruch und ist daher von größerem Interesse für die Parallelisierung als der Rest des Verfahrens.

Bei der numerischen Faktorisierung wird die untere Dreiecksmatrix L spaltenweise aus der Matrix A berechnet. Bei der Berechnung einer Spalte wird zuerst das Diagonalelement berechnet:

$$l_{jj} = \sqrt{a_{jj} - \sum_{k=0}^{j-1} l_{jk}^2} \tag{5.1}$$

Danach werden die restlichen Elemente der Spalte mit der Formel:

$$l_{ij} = \frac{1}{l_{jj}}(a_{ij} - \sum_{k=0}^{j-1} l_{jk} l_{ik}) \tag{5.2}$$

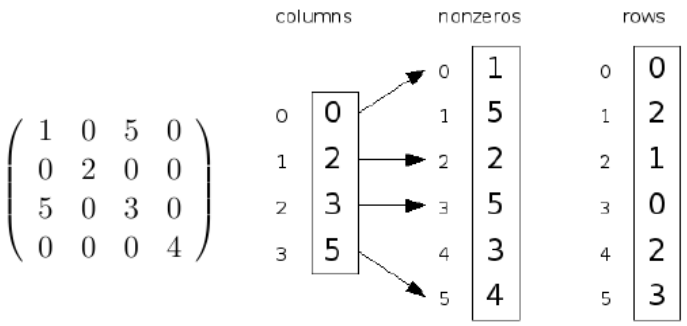
berechnet, wobei i der Zeilenindex des aktuellen Elements der Spalte ist.

Enthält eine Matrix nur wenige oder keine Nullelemente, kann eine Spalte erst dann berechnet werden, wenn alle Spalten links von der aktuellen Spalte berechnet worden sind. Die Faktorisierung kann in diesem Fall nur sequenziell ausgeführt werden. Anders ist es bei dünnbesetzten Matrizen. Hier können einzelne Spalten unabhängig voneinander berechnet werden, wenn es keine Abhängigkeiten zwischen diesen Spalten gibt. Eine Spalte j der Matrix L ist von einer Spalte i abhängig, wenn das Element l_{ij} ungleich 0 ist.

Für die numerische Faktorisierung gibt es hauptsächlich drei verschiedene Vorgehensweisen: Left-Looking-, Right-Looking- und Superknoten-Algorithmus. Alle drei Berechnungsschemata haben ihre Vor- und Nachteile. Alle drei Varianten erfordern einen zusätzlichen Aufwand für die Taskverwaltung. Die Left-Looking-Variante benötigt für jede Spalte der Matrix einen Stack. Bei den anderen Varianten wird für jede Spalte ein Zähler verwaltet. Die Zugriffe auf die Stacks bzw. die Zähler müssen durch Lock-Variablen geschützt werden. Da die Operationen auf einem Stack normalerweise aufwendiger sind als das Inkrementieren eines Zählers ist die Left-Looking-Variante aufwendiger als die anderen beiden. Die Superknoten-Variante erzeugt im Vergleich zu anderen zwei Varianten weniger Tasks, weil bei dieser Variante mehrere Spalten zu einer Superknotentask zusammengefasst werden. Alle drei Varianten werden in [6] beschrieben. Das Buch enthält auch einen Vergleich dieser drei Varianten. In meiner Implementierung habe ich den Right-Looking-Algorithmus verwendet.

5.3.2 Implementierung mit Taskpools

Die Elemente der Dreiecksmatrix L werden Spaltenweise berechnet. Eine Task berechnet dabei eine Spalte. Anfangs wird für jede Spalte, die von keiner anderen abhängig ist, eine Task erzeugt und in den Taskpool eingefügt.



$$\begin{pmatrix} 1 & 0 & 5 & 0 \\ 0 & 2 & 0 & 0 \\ 5 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

Abbildung 5.1: Abspeicherungsschema für eine dünnbesetzte Matrix

Bei der Ausführung einer Task werden zuerst alle Elemente der entsprechenden Spalte berechnet. Danach werden alle Spalten aktualisiert, die von der aktuellen Spalte abhängig sind. Bei der Aktualisierung wird das l_{jk}-fache der aktuellen Spalte von der abhängigen Spalte abgezogen, wobei j der Index der aktuellen Spalte und k der Index der abhängigen Spalte ist. Sobald alle Abhängigkeiten für eine Spalte erfüllt sind, wird diese in den Taskpool eingefügt.

Da eine Spalte von mehreren anderen abhängig sein kann, kann es vorkommen, dass mehrere Threads gleichzeitig versuchen die Aktualisierung dieser Spalte vorzunehmen. Um Race-Conditions zu vermeiden, wird daher jede Spalte durch eine OpenMP-Lock-Variable geschützt. Diese Lock-Variable wird jedes mal gesperrt, wenn eine Spalte aktualisiert wird.

5.3.3 Abspeicherungsschema für dünnbesetzte Matrizen

Da die meisten Elemente einer dünnbesetzten Matrix Nullen sind, wird für solche Matrizen oft ein spezielles Abspeicherungsschema verwendet, bei dem nur die Nichtnullelemente gespeichert werden. Zusätzlich werden die Zeilen- und Spaltenindizes dieser Elemente getrennt gespeichert. Ich habe ein Abspeicherungsschema verwendet, welches die Matrix in drei separaten Arrays speichert. Im ersten Array (`nonzeros`) werden nacheinander alle Nichtnullelemente der Matrix gespeichert. Ein anderes Array (`columns`) enthält für jede Spalte den Index des ersten Nichtnull-Elements dieser Spalte im Array `nonzeros`. Das dritte Array (`rows`) enthält die Zeilenindizes der entsprechenden Nichtnullelemente. Abbildung 5.1 zeigt eine Illustration dieses Abspeicherungsschemas.

Kapitel 6

Ergebnisse

In diesem Kapitel werden die Ergebnisse der Laufzeitmessungen der drei Testapplikationen vorgestellt. Die Laufzeitmessungen wurden jeweils auf zwei verschiedenen Rechnersystemen durchgeführt: einem Linux-System mit 4 Opteron 847 Prozessoren mit jeweils 2.2 GHz und einer Sun Fire E6900 (Solaris) mit 1,2 GHz SPARC Prozessoren. Auf den Opteron Systemen wurden die Programme mit dem Intel C++ Compiler 9.0 übersetzt. Auf dem Sun Fire E6900 System wurde der Guide Compiler von Kuck and Associates, Inc. verwendet. Alle Tabellen zeigen die Durchschnittwerte der Zeitmessungen aus jeweils drei Programmdurchläufen.

Im Folgenden werden die Ergebnisse für alle drei Testapplikationen einzeln vorgestellt.

6.1 Quicksort

Die Tabellen 6.1 und 6.2 zeigen die Ausführungszeiten von Quicksort mit verschiedenen Taskpoolvarianten. Als Eingabe für Quicksort wurde ein Array mit 100000000 Gleitkommazahlen auf dem Opteron-System und ein Array mit 10000000 Zahlen auf dem Sun Fire System.

Aus den Ergebnissen geht hervor, dass Quicksort mit Taskpoolvarianten, die private Queues benutzen, die beste Performance erreicht. Der Grund dafür ist die große Anzahl der Tasks die diese Applikation erzeugt. Für jede Teilfolge, die beim Partitionieren entsteht, wird eine neue Task erzeugt (vergleiche 5.1). Dies hat zu Folge, dass es während der gesamten Ausführung von Quicksort genügend Tasks im Taskpool bleiben. Die Abbildung 6.1 zeigt, wie sich die Anzahl der Tasks in einem Taskpool während der Ausführung

	Speichermanager			malloc()/free()		
Threads:	1	2	4	1	3	4
sq1	22,94	15,38	10,65	23,14	15,67	11,05
sq2	27,98	18,28	13,64	28,37	18,94	13,98
sdq1	22,45	14,41	9,68	22,56	14,48	9,79
dq8	22,49	12,67	7,77	22,62	12,43	7,53
dq9	22,23	12,53	8,06	22,32	12,38	7,63
dq9-1	22,21	12,32	7,42	22,33	12,21	7,27
dq10	27,4	15,99	10,77	27,56	16,13	11,08
Threads:	1		2		4	
Workqueue	24.23		13.94		8.04	

Tabelle 6.1: Ausführungszeiten von Quicksort in Sekunden mit verschiedenen Taskpoolvarianten auf dem Opteron 847 System.

	Speichermanager				malloc()/free()			
Threads:	1	2	4	8	1	2	4	8
sq1	5,54	3,91	3,08	2,38	5,41	4,16	3,86	5,2
sq2	13,81	8,45	4,6	3,41	20,3	9,77	6,36	5,5
sdq1	5,24	3,75	2,62	2,18	5,14	3,68	3,13	3,42
dq8	5,27	2,99	2,18	1,93	5,58	3,57	2,99	3,77
dq9	5,36	3,11	2,07	1,72	5,46	3,37	3,42	3,54
dq9-1	4,98	3,08	2,13	1,64	5,62	3,65	2,73	3,33
dq10	12,01	7,13	3,01	1,91	13,25	7,82	5,02	4,69
Threads:	1		2		4		8	
Workqueue	7.25		4.4		3.31		2.76	

Tabelle 6.2: Ausführungszeiten von Quicksort in Sekunden mit verschiedenen Taskpoolvarianten auf dem Sun Fire E6900 System.

Größe der Taskgruppe:	1 Task			4 Tasks		
Threads:	1	2	4	1	2	4
sdq1	22,45	14,41	9,68	22,6	13,05	8,88
dq8	22,49	12,67	7,77	22,59	13,6	9,22
dq9	22,23	12,53	8,06	22,57	12,03	7,48
dq9-1	22,21	12,32	7,42	22,58	13,36	9,01
dq10	27,4	15,99	10,77	27,74	14,79	9,17

Tabelle 6.3: Ausführungszeiten von Quicksort mit verschiedenen Größen der Taskgruppen auf dem Opteron 847 System.

von Quicksort ändert. Ein Teil dieser Tasks kann in privaten Queues abgelegt werden, ohne dass die Lastenbalancierung beeinträchtigt wird. Es kommt nur selten vor, dass ein Thread auf Arbeit warten muss, obwohl der Taskpool noch Tasks enthält, diese aber alle in den privaten Queues anderer Threads gespeichert sind. Da die Zugriffe auf die privaten Queues wesentlich effizienter sind, weil dabei keine Synchronisation notwendig ist, steigt durch die Benutzung der privaten Queues die Performance des gesamten Programms.

Ein anderer Grund für das gute Abschneiden der Taskpoolvarianten mit privaten Taskqueues hängt mit der Zeit, die für die Ausführung einer Task benötigt wird, zusammen. Die Zeit, die ein Thread für die Ausführung einer Task benötigt, hängt von der Länge der zu sortierenden Teilfolge ab. Die meisten Tasks können schnell abgearbeitet werden, weil während des Programmablaufs immer kürzere Teilfolgen entstehen. Je schneller eine Task abgearbeitet werden kann, desto häufiger muss das Programm auf den Taskpool zugreifen. Durch die häufigen Zugriffe auf den Taskpool steigt die Wahrscheinlichkeit für Konflikte beim Zugriff auf eine gemeinsame Taskqueue. Beim Entnehmen der Tasks aus den privaten Queues entstehen solche Konflikte jedoch nicht.

Ein optimaler Speedup kann bei dem verwendeten Algorithmus auch mit privaten Queues nicht erreicht werden. Der Algorithmus startet mit nur einer Task, die auch die aufwendigste ist, weil die gesamte Zahlenfolge partitioniert werden muss. Diese Task kann nur von einem Thread ausgeführt werden, während alle anderen Threads warten müssen.

6.2 Labyrinth

Das Labyrinthprogramm erreicht von allen drei Benchmark-Applikationen die schlechtesten Ergebnisse. Die Tabellen 6.4, 6.5 und 6.6 zeigen die gemessenen Laufzeiten des Programms in Kombination mit verschiedenen Taskpoolvarianten. Als Eingabe wurde ein nicht perfektes Labyrinth mit 6000×6000 Zellen verwendet. In einem nicht perfekten Labyrinth gibt es im Gegensatz zu einem perfekten Labyrinth mehrere Wege, die zu einer bestimmten Labyrinthzelle führen. Mit einem perfekten Labyrinth als Eingabe erreicht keine der Taskpoolvarianten Speedups. Ein möglicher Grund dafür ist, dass bei einem perfekten Labyrinth weniger Tasks erzeugt werden. Die Abbildungen 6.2 und 6.3 zeigen, wie sich die Anzahl der Tasks während der Suche nach dem kürzesten Weg durch ein perfektes bzw. ein nicht perfektes Labyrinth ändert. Beide Labyrinthe sind 6000×6000 Zellen groß. Bei dem perfekten Labyrinth werden jedoch im Durchschnitt 10-mal weniger Tasks erzeugt.

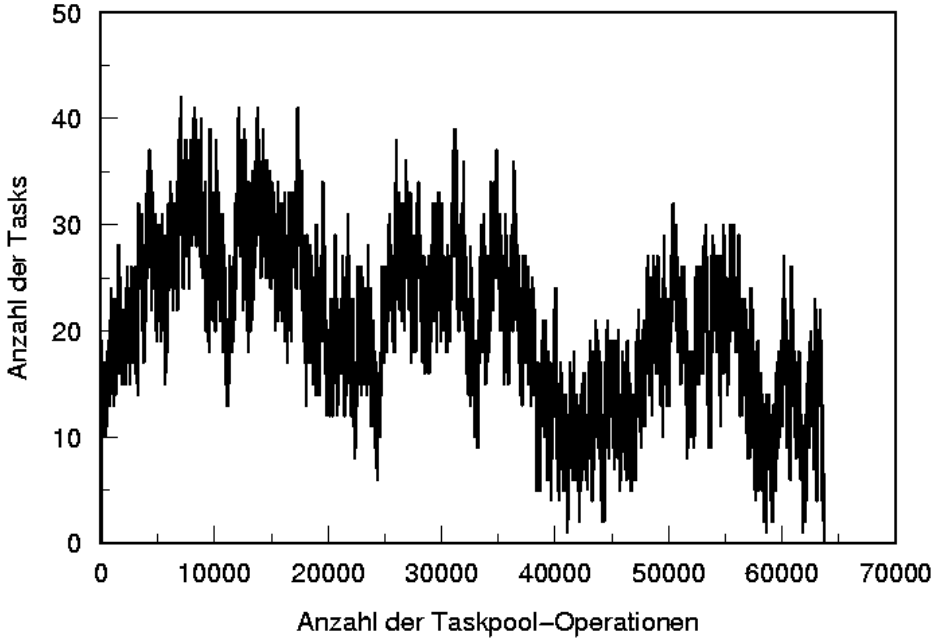


Abbildung 6.1: Anzahl der Tasks in einem Taskpool beim Sortieren eines Arrays mit 1000000 Elementen.

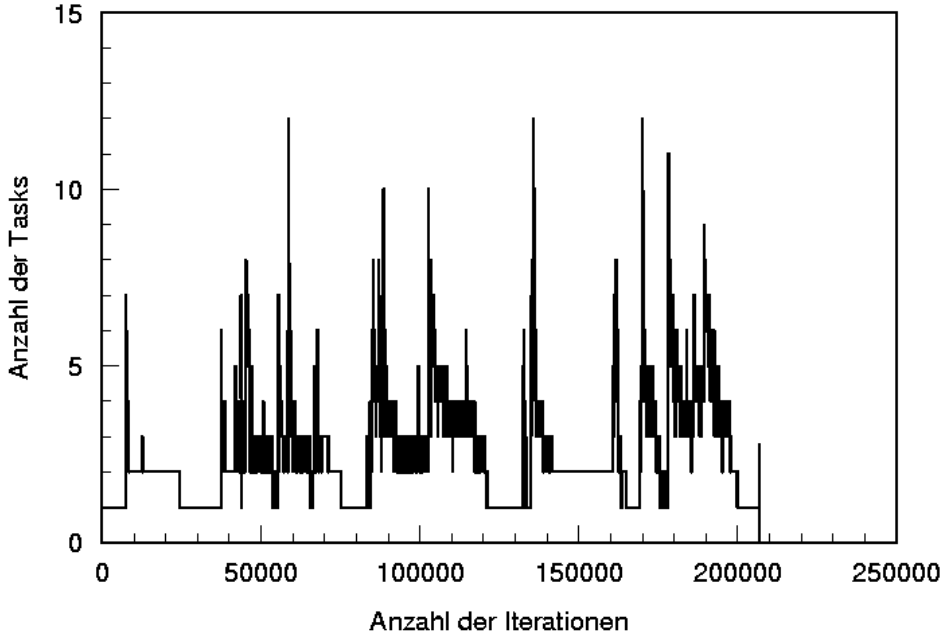


Abbildung 6.2: Anzahl der Tasks in einem Taskpool bei der Suche nach dem kürzesten Weg durch ein perfektes Labyrinth.

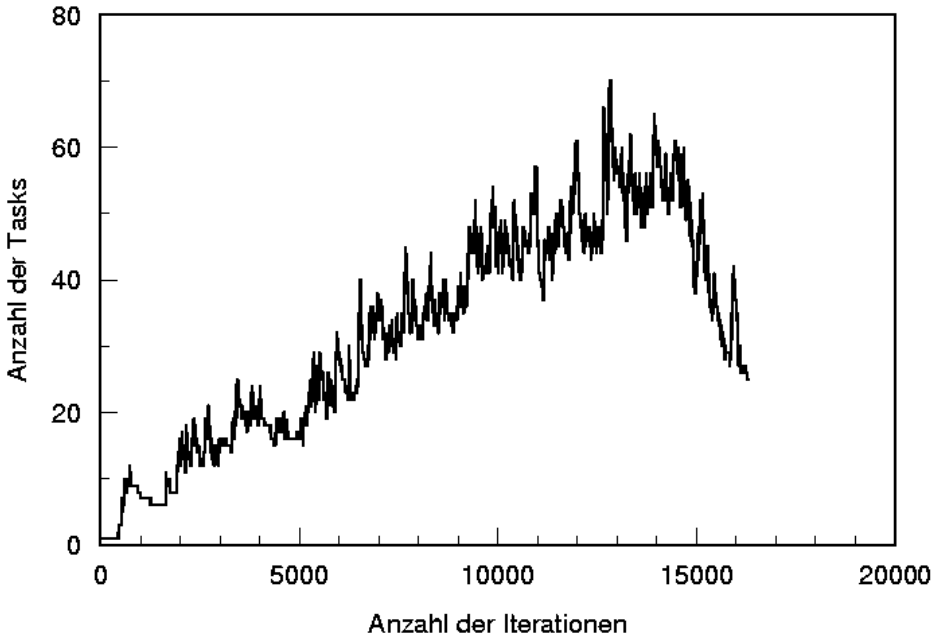


Abbildung 6.3: Anzahl der Tasks in einem Taskpool bei der Suche nach dem kürzesten Weg durch ein nicht perfektes Labyrinth.

	Speichermanager			malloc()/free()		
Threads:	1	2	4	1	3	4
sq1	2,42	2,55	2,28	2,49	2,72	2,59
sq2	2,56	2,83	2,62	2,5	2,64	2,38
sdq1	2,68	2,79	2,12	2,72	2,88	2,29
dq8	2,5	1,66	1,42	2,44	1,74	1,47
dq9	2,4	2,54	2,67	2,33	2,42	2,55
dq9-1	2,42	1,67	1,5	2,34	1,7	1,48
dq10	2,47	2,55	2,71	2,41	2,48	2,57

Tabelle 6.4: Ausführungszeiten des Labyrinthprogramms mit verschiedenen Taskpoolvarianten auf dem Opteron 847 System.

	Speichermanager				malloc()/free()			
Threads:	1	2	4	8	1	3	4	8
sq1	6,09	7,85	6,65	5,53	6,57	9,69	9,13	14,08
sq2	6,42	8,18	7,23	5,96	7,15	10,23	10,37	13,4
sdq1	6,11	7,45	5,92	4,74	6,62	8,37	9,26	12,39
dq8	6,12	4,58	3,69	3,17	6,51	6,61	8,89	11,75
dq9	5,83	6,04	6,14	6,48	6,47	6,87	6,72	11,32
dq9-1	5,88	4,5	3,6	3,28	6,41	6,82	8,25	12,48
dq10	6,28	6,49	6,77	7,15	6,79	7,02	7,25	10,8

Tabelle 6.5: Ausführungszeiten des Labyrinthprogramms mit verschiedenen Taskpoolvarianten auf dem Sun Fire E6900 System.

Größe der Taskgruppe:	1 Task			4 Tasks		
Threads:	1	2	4	1	2	4
sdq1	2,68	2,79	2,12	2,57	2,41	1,76
dq8	2,5	1,66	1,42	2,56	1,73	1,36
dq9	2,4	2,54	2,67	2,53	2,6	2,71
dq9-1	2,42	1,67	1,5	2,53	1,79	1,43
dq10	2,47	2,55	2,71	2,62	2,69	2,77

Tabelle 6.6: Ausführungszeiten des Labyrinthprogramms mit verschiedenen Größen der Taskgruppen auf dem Opteron 847 System.

Wie Quicksort profitiert auch das Labyrinthprogramm von der Nutzung von privaten Taskqueues. Mit Ausnahme von *dq9* und *dq10* erreichen alle Taskpools, die private Queues verwenden, Speedups. Der Grund für das schlechte Abschneiden von *dq9* und *dq10* hängt mir der Art, wie diese Taskpoolvarianten die Tasks zwischen der privaten und der öffentlichen Queue eines Threads verteilen, zusammen. *dq9* und *dq10* versuchen möglichst viele Tasks in der privaten Queue zu halten, um teure Synchronisationsoperationen zu vermeiden. Eine neue Task wird daher nur dann in die öffentliche Queue eines Threads eingefügt, wenn diese Queue leer ist und es Threads gibt, die auf Arbeit warten. Das Labyrinthprogramm verwendet aber zwei Taskpools: einen zum Ablegen neu erzeugter Tasks und einen zum Entnehmen der Tasks. Aus dem Taskpool, in den neue Tasks eingefügt werden, werden gleichzeitig keine Tasks entnommen, sodass die Anzahl der wartenden Threads für diesen Taskpool immer null bleibt. *dq9* und *dq10* fügen alle Tasks in die privaten Queues ein. Ein Austausch der Tasks zwischen den Threads ist daher nicht möglich. Hinzu kommt, dass der Algorithmus mit nur einer Task startet. Diese wird in der privaten Queue eines Threads gespeichert. Alle weiteren Tasks, die während der Berechnung erzeugt werden, bleiben ebenfalls in der privaten Queue dieses Threads, sodass nur ein Thread arbeiten kann, obwohl es möglicherweise genügend Tasks

	Speichermanager			malloc()/free()		
Threads:	1	2	4	1	3	4
sq1	7,03	3,65	1,96	7,03	3,64	1,97
sq2	7,04	3,64	1,88	7,04	3,64	1,93
sdq1	7,03	7,04	7,08	7,03	7,04	7,03
dq8	7,03	7,03	7,02	7,03	7,04	7,02
dq9	7,03	3,69	1,99	7,03	3,71	1,97
dq9-1	7,04	7,04	7,06	7,04	7,03	7,02
dq10	7,04	3,72	1,98	7,03	3,71	1,98
Threads:	1		2		4	
Workqueue	9.42		4.84		2.51	

Tabelle 6.7: Ausführungszeiten der Cholesky-Faktorisierung mit verschiedenen Taskpoolvarianten auf dem Opteron 847 System.

für alle Threads gibt.

6.3 Cholesky-Faktorisierung

Die Cholesky-Faktorisierung erzeugt relativ wenige Tasks. Daher ist eine gute Lastenverteilung entscheidend. Die Tabellen 6.7 und 6.8 zeigen die Ausführungszeiten der Cholesky-Faktorisierung mit verschiedenen Taskpoolvarianten. Als Eingabe wurde eine 500×500 Matrix mit 196306 Nichtnullelementen verwendet. Die Taskpoolvarianten mit einer zentralen Taskqueue erreichen hier die besten Speedups. sq1 und sq2 ermöglichen beide eine gute Lastenverteilung, da alle Tasks in einer zentralen Queue gespeichert sind, so dass jeder Thread auf diese Tasks zugreifen kann. Durch die geringe Anzahl der Tasks führt das Programm nur wenige Operationen auf dem Taskpool aus. Damit sinkt auch die Anzahl der Konflikte beim Zugriff auf die zentrale Queue.

Private Queues verhindern in diesem Fall eine optimale Lastenverteilung. Die Abbildung 6.4 zeigt, wie sich die Anzahl der Tasks während der Cholesky-Faktorisierung ändert. Die wenigen Tasks, die der Taskpool enthält, bleiben alle in den privaten Queues einzelner Threads, sodass nur die Besitzer dieser Queues auf die Tasks zugreifen können. Alle anderen Threads, deren Queues leer sind, warten, obwohl es noch Tasks im Taskpool gibt.

Die Taskpoolvarianten *dq9* und *dq10* sind die einzigen Varianten mit privaten Taskqueues, die Speedups erreichen. Die Ursache dafür liegt in der Art, wie diese Taskpoolvarianten die Tasks zwischen den privaten und den

	Speichermanager				malloc()/free()			
Threads:	1	2	4	8	1	3	4	8
sq1	21,16	11,15	5,78	3,05	21,18	11,17	5,77	3,09
sq2	21,16	11,19	5,79	3,04	21,17	11,17	5,82	3,05
sdq1	21,15	21,23	21,21	21,43	21,24	21,2	21,22	21,3
dq8	21,18	21,17	21,18	21,36	21,17	21,18	21,28	21,34
dq9	21,19	11,3	5,87	3,22	21,19	11,26	5,86	3,24
dq9-1	21,21	21,18	21,18	21,35	21,2	21,22	21,22	21,41
dq10	21,15	11,29	5,92	3,31	21,21	11,35	5,96	3,38
Threads:	1		2		4		8	
Workqueue	17.41		9.3		4.92		2.71	

Tabelle 6.8: Ausführungszeiten der Cholesky-Faktorisierung mit verschiedenen Taskpoolvarianten auf dem Sun Fire E6900 System.

Größe der Taskgruppe:	1 Task			4 Tasks		
Threads:	1	2	4	1	2	4
sdq1	7,03	7,04	7,08	7,02	7,02	7,03
dq8	7,03	7,03	7,02	7,02	7,03	7,03
dq9	7,03	3,69	1,99	7,02	3,69	1,96
dq9-1	7,04	7,04	7,06	7,02	7,02	7,03
dq10	7,04	3,72	1,98	7,02	3,71	1,99

Tabelle 6.9: Ausführungszeiten der Cholesky-Faktorisierung mit verschiedenen Größen der Taskgruppen auf dem Opteron 847 System.

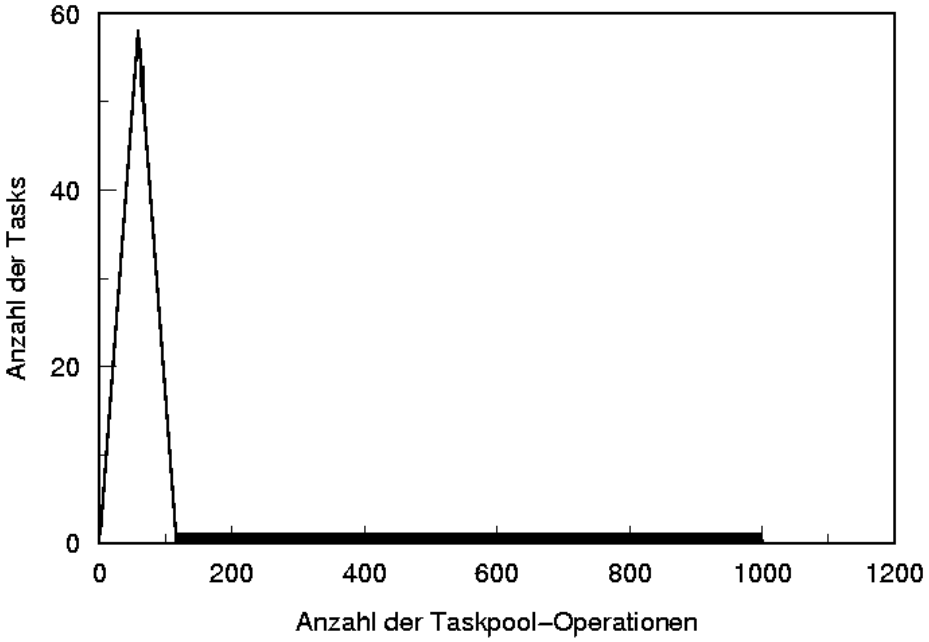


Abbildung 6.4: Anzahl der Tasks in einem Taskpool bei der Berechnung der Cholesky Zerlegung eine 500×500 Matrix mit 196306 Nichtnullelementen.

öffentlichen Queues verteilen. Bei *dq9* hängt die Entscheidung, ob eine neue Task in die private oder öffentliche Queue eingefügt wird, von dem Zustand der öffentlichen Queue und der Anzahl der wartenden Threads ab: Ist die öffentliche Queue leer und gibt es Threads, die auf Tasks warten, dann wird die neue Task in die öffentliche Queue eingefügt. Auf diese Weise ermöglicht *dq9* eine bessere Lastenverteilung als alle anderen Taskpoolvarianten mit privaten Queues.

Wie die Abbildung 6.4 zeigt, bleibt der Taskpool bei der Cholesky Faktorisierung die meiste Zeit fast leer. Nachdem anfangs alle unabhängigen Spalten berechnet worden sind, enthält der Taskpool im Durchschnitt nur eine Task. Dass die Applikation trotzdem gute Speedups erreicht, liegt unteranderem auch daran, dass neue Tasks anders als z.B. bei Quicksort nicht erst am Ende der Ausführung einer Task erzeugt werden. Das bedeutet, dass ein Thread anders als bei Quicksort nach dem Erzeugen einer neuen Task weiter arbeitet, während die neue Task gleichzeitig von einem anderen Thread ausgeführt werden kann.

6.4 Größe der Taskgruppen

Wie die Tabelle 6.3 zeigt, verbessert sich die Performance der Taskpoolvarianten mit dezentralen Taskqueues bei Quicksort, wenn man mehrere Tasks zu einer Gruppe zusammenfasst. Durch die Verwendung von Taskgruppen lassen sich mehrere Tasks auf einmal aus einer öffentlichen Queue entnehmen. Damit sinkt die Anzahl der aufwendigen Zugriffe auf die öffentlichen Queues. Die Taskpool-Variante *sdq1*, die eine zentrale gemeinsame Taskqueue verwaltet, profitiert am meisten von größeren Taskgruppen. Nur die Taskpoolvarianten *dq8* und *dq9-1* werden durch die Verwendung von Taskgruppen verlangsamt. Diese beiden Taskpoolvarianten fügen neue Tasks immer zuerst in die privaten Queues ein, bis diese voll sind (zwei Taskgruppen enthalten). Mit der steigenden Anzahl der Tasks in einer Taskgruppe steigt auch die maximale Anzahl der Tasks in einer privaten Queue. Bei Quicksort startet die Berechnung mit nur einer Task, sodass anfangs nur ein Thread arbeiten kann. Wenn die private Queue nun größer ist, müssen andere Threads länger auf Tasks warten. Da bei Quicksort die ersten Tasks die aufwendigsten sind, weil größere Teilfolgen partitioniert werden müssen, wirken sich größere Taskgruppen negativ auf die Performance der Taskpoolvarianten *dq8* und *dq9-1* aus.

Die Tabelle 6.6 zeigt die Ausführungszeiten des Labyrinth-Programms mit Taskpoolvarianten mit dezentralen Taskqueues. Die einzige Taskpool-Variante, die beim Labyrinthprogramm wirklich von größeren Taskgruppen profitiert, ist *sdq1*.

Bei der Cholesky-Faktorisierung hat die Größe der Taskgruppen anscheinend keinen Einfluss auf die Performance des Programms. Da bei der Cholesky-Faktorisierung im Vergleich zu den anderen Testapplikationen nur wenige Tasks erzeugt werden, ist eine gute Lastenverteilung entscheidend. Das Zusammenfassen von mehreren Tasks zu einer Gruppe ist nur dann sinnvoll, wenn relativ viele Tasks erzeugt werden. Die Ergebnisse der Laufzeitmessungen mit Taskgruppen verschiedener Größen befinden sich in der Tabelle 6.9.

6.5 FIFO- und LIFO-Queues

Es zeigt sich außerdem, dass die Reihenfolge, in der die Tasks aus dem Pool entnommen werden, eine spürbare Auswirkung auf die Performance von Quicksort hat. Die Taskpoolvarianten, die die Tasks in der FIFO-Reihenfolge einfügen und entnehmen (*sq2* und *dq10*), schneiden deutlich schlechter ab, als vergleichbare Taskpoolvarianten mit LIFO-Queues (*sq1* und *dq9*). Der

Grund dafür ist, dass Quicksort in Kombination mit Taskpoolvarianten, die LIFO-Queues benutzen, eine höhere *Lokalität der Speicherzugriffe* aufweist, als mit Taskpoolvarianten, die FIFO-Queues verwenden. Beim Partitionieren einer Folge entstehen zwei neue Tasks, die mit denselben Daten arbeiten, wie die ursprüngliche Task. Bei Verwenden der Taskpools mit LIFO-Queues werden diese neuen Tasks sofort nach der Task, die sie erzeugt hat, ausgeführt. Dabei ist die Wahrscheinlichkeit größer, dass die benötigten Daten sich noch im Cache befinden, weil die Task, die davor ausgeführt wurde, mit diesen Daten gearbeitet hat. Das Nachladen neuer Speicherblocks in den Cache wird vermieden, was die Performance des Programms verbessert.

Beim Labyrinth-Problem ist der Performance-Unterschied zwischen LIFO- und FIFO-Queues nicht so deutlich wie bei Quicksort. Aber auch hier sind die Taskpoolvarianten mit LIFO-Queues auf der SPARC-Architektur schneller als die entsprechenden Taskpools mit FIFO-Queues.

Während bei Quicksort die Taskpoolvarianten mit LIFO-Queues deutlich besser abschneiden, als Taskpools, die FIFO-Queues benutzen, erzielen bei der Cholesky-Faktorisierung beide Taskpool-Arten vergleichbare Resultate. *sq2* erreicht von allen sieben Taskpoolvarianten sogar die besten Speedups. Anders als bei Quicksort arbeitet bei der Cholesky-Faktorisierung jede Task mit anderen Daten. Damit hat die Reihenfolge der ausgeführten Tasks keinen Einfluss auf die Lokalität der Speicherzugriffe.

6.6 Einfluss der Speichermanager auf die Performance der Taskpools

Wie die Tabellen 6.2 und 6.5 zeigen erreichen Quicksort und Labyrinth unter Solaris deutlich bessere Ergebnisse, wenn ein Speichermanager verwendet wird. In manchen Fällen sind diese beiden Testapplikationen mehr als doppelt so schnell, wenn der Speicher mit Hilfe des Speichermanagers allokiert wird. Das Labyrinthprogramm erzielt Speedups nur, wenn ein Speichermanager verwendet wird. Quicksort wird ohne Speichermanager mit acht Threads wieder langsamer als mit vier. Mit einem Speichermanager erzielt das Programm dagegen auch mit acht Threads Speedups.

Bei Linuxsystemen hat der Speichermanager dagegen keinen Einfluss auf die Performance der beiden Programme. Anscheinend ist die Implementierung von malloc() von Linux besser für multithreaded Programme geeignet als die Solaris-Implementierung. Das geht auch aus dem Bericht von Chuck Lever und David Boreham [5] hervor.

Für die Performance der Cholesky-Faktorisierung spielt die Tatsache, ob

ein Speichermanager verwendet wird auf beidem Plattformen keine Rolle. Die Ursache dafür liegt in der geringen Anzahl der Tasks, die diese Test-Applikation erzeugt. Die Tasks werden selten erzeugt, was die Wahrscheinlichkeit für Konflikte beim dynamischen Reservieren von Speicherblöcken zwischen den Threads verringert.

6.7 Vergleich der Taskpoolvarianten

6.7.1 Taskpools mit einer zentralen Taskqueue

Wie erwartet schneiden die Taskpool-Varianten mit nur einer gemeinsamen Taskqueue bei Applikationen, die sehr viele Tasks erzeugen, schlecht ab. Die zentrale Taskqueue erweist sich als Leistungsengpass. Für Applikationen, die nur wenige Tasks erzeugen und selten auf den Taskpool zugreifen sind die Taskpools mit einer zentralen Queue gut geeignet. Bei der Cholesky-Faktorisierung erreichen *sq1* und *sq2* von allen Taskpoolvarianten sogar die besten Ergebnisse.

6.7.2 Taskpools mit dezentralen Taskqueues

Applikationen, die viele Tasks erzeugen und dementsprechend oft auf den Taskpool zugreifen müssen, profitieren von dezentralen Taskqueues. Taskpools mit dezentralen Taskqueues haben nicht den Nachteil einer zentralen Engstelle und erreichen bei Quicksort und Labyrinth deshalb bessere Ergebnisse, als Taskpools mit einer zentralen Taskqueue. Man kann auch davon ausgehen, dass auch die Skalierbarkeit von Taskpools mit dezentralen Taskqueues besser ist, als die von Taskpools mit einer zentralen Queue, weil mit der steigenden Anzahl der Threads auch die Wahrscheinlichkeit für Konflikte beim Zugriff auf eine zentrale Queue steigt.

Die Nutzung von privaten Queues ermöglicht einen zusätzlichen Geschwindigkeitsvorteil bei Applikationen, die viele Tasks erzeugen, weil beim Zugriff auf private Taskqueues teure Synchronisationsoperationen (wie z.B. das Sperren von Lock-Variablen) vermieden werden können. Die Taskpools mit privaten Queues haben allerdings den Nachteil, dass sie die Tasks nicht so gleichmäßig zwischen den einzelnen Threads verteilen, wie es bei Taskpools mit einer zentralen Taskqueue der Fall ist. Bei Applikationen, die nur wenige Tasks erzeugen (wie z.B. die Cholesky-Faktorisierung) ist eine gleichmäßige Verteilung der Tasks entscheidend für die Performance des Programms. Hier schneiden die meisten Taskpoolvarianten, die private Taskqueues verwenden, unerwartet schlecht ab.

6.8 Implementierung mit OpenMP

6.8.1 Aktives Warten

Insgesamt verlief die Implementierung mit OpenMP weitgehend ohne Hindernisse, da OpenMP einfache, aber dennoch mächtige Mechanismen für die Parallelisierung von Programmen bereitstellt. Aber für ein Problem, das während der Implementierung aufgetaucht ist, bietet OpenMP keine geeignete Lösung: Versucht ein Thread eine Task aus dem Pool zu entnehmen, stellt aber fest, dass alle Queues, auf die er zugreifen darf, leer sind, muss er warten, bis eine neue Task von einem anderen Thread eingefügt wird. Korch und Rauber [4] benutzten Bedingungsvariablen in ihrer Pthreads-Implementierung bzw. das wait()−notify()-Konstrukt in Java. Bedingungsvariablen (engl. *condition variables*) ermöglichen es, einen Thread zu blockieren, bis eine bestimmte Bedingung erfüllt ist. Während der Thread auf die Erfüllung dieser Bedingung wartet, verbraucht er keine CPU-Zeit. Sobald die Bedingung erfüllt ist, wird der Thread wieder aufgeweckt und kann weiter rechnen. Analog gibt es in Java die Möglichkeit, mit dem Aufruf von wait() auf einem Objekt, einen Thread anzuhalten. Mit notify() kann ein blockierter Thread wieder aufgeweckt werden. Leider gibt es in OpenMP keine Möglichkeit, einen Thread anzuhalten und auf die Erfüllung einer Bedingung warten zu lassen. Dem Programmierer bleibt nichts anderes übrig, als den Thread dauernd die Bedingung auswerten zu lassen, bis diese erfüllt ist. Dieses Verfahren wird *aktives Warten* (engl. *busy waiting*) genannt. Der Nachteil dieser Lösung ist, dass der Thread, während er die Bedingung auswertet, CPU-Zeit verbraucht und damit möglicherweise andere Threads, die denselben Prozessor nutzen, ausbremst.

6.8.2 Das Workqueueing-Modell

Das Workqueueing-Modell ist eine OpenMP-Erweiterung, die entwickelt wurde mit dem Ziel, die Parallelisierungen von irregulären Algorithmen mit OpenMP zu erleichtern. Diese Erweiterung wurde von Shah et. al. [7] vorgestellt und erlaubt es dem Programmierer, Tasks zu definieren, die dynamisch zwischen den Threads des Programms verteilt werden können. Dazu wird mit der taskq-Direktive eine Umgebung angegeben, in der die Tasks erzeugt werden können. Der Code innerhalb dieser Umgebung wird von einem Thread ausgeführt. Dieser Thread erstellt eine Taskqueue und fügt die, mit Hilfe der task-Direktive erstellten, Tasks in diese Queue ein. Alle anderen Threads warten, bis Tasks in die Queue eingefügt werden und führen diese dann aus. Auf diese Weise ermöglicht das Workqueueing-Modell eine dynamische Verteilung der Arbeit zwischen den Threads eines Programms.

Leider ist diese Erweiterung nicht im OpenMP-Standard enthalten. Da aber sowohl der Intel C++ Compiler, als auch der Guide Compiler das Workqueueing-Modell bereits unterstützen, habe ich jeweils eine Variante von dem Quicksort-Algorithmus und der Cholesky-Faktorisierung implementiert, die anstelle der Taskpools diese OpenMP-Erweiterung nutzen. Bei beiden Testapplikationen erreicht die Workqueueing-Variante eine gute Performance. Die Tabellen 6.1 und 6.2 enthalten neben den gemessenen Ausführungszeiten der verschiedenen Taskpoolvarianten auch die Laufzeiten der Workqueueing-Variante von Quicksort. Auf dem Opteron-System ist die Workqueueing-Variante von Quicksort schneller als die Taskpools mit zentralen Queues. Nur die Taskpools mit privaten Taskqueues erreichen hier bessere Ergebnisse. Auf dem Sun Fire E6900 System hat der Speichermanager eine große Auswirkung auf die Performance der Taskpools. Hier sind alle Taskpools mit Ausnahme von *sq2* schneller als die Workqueueing-Variante, wenn ein Speichermanager verwendet wird. Ohne Speichermanager schneiden die Taskpools schlechter als die Workqueueing-Variante ab. Bei der Cholesky-Faktorisierung erreicht die Workqueueing-Variante auf dem Sun Fire System bessere Performance, als alle Taskpoolvarianten. Die Tabellen 6.7 und 6.8 zeigen die gemessenen Ausführungszeiten der Workqueueingvariante und der verschiedenen Taskpoolvarianten der Cholesky-Faktorisierung.

Für das Labyrinth-Problem habe ich keine Workqueueing-Variante implementiert, da ich keinen Weg gefunden habe zwei separate Taskqueues zu verwenden und dabei sicherzustellen, dass erst alle Tasks aus einer Queue abgearbeitet werden, bevor die Tasks aus der anderen Queue entnommen werden. Der in der Taskpool-Implementierung verwendete Algorithmus lässt sich nicht direkt auf das Workqueueing Modell übertragen.

Kapitel 7

Zusammenfassung und Ausblick

In dieser Arbeit wurden sieben Taskpoolvarianten beschrieben, die ich mit OpenMP implementiert habe. Einige davon wurden von einer bestehenden Pthreads Implementierung portiert, andere entstanden durch Erweiterungen der portierten Taskpools. Eine Schwäche von OpenMP, die während der Implementierung der Taskpools sichtbar wurde, ist das Fehlen eines Mechanismus, der es erlaubt einen Thread anzuhalten, bis eine Bedingung erfüllt ist. Wenn ein Thread bei Entnehmen einer Task einen leeren Taskpool vorfindet, muss er warten, bis neue Tasks in den Pool eingefügt werden. Die einzige Lösung für dieses Problem ist aktives Warten. Der Thread prüft dauernd, ob eine Bedingung erfüllt ist und verschwendet dabei die Prozessorleistung.

Um die Taskpools zu testen, habe ich drei irreguläre Programme implementiert, die diese Taskpools benutzten, um die dynamische Lastenbalancierung zu erreichen: Quicksort, das Labyrinth-Problem und die Cholesky-Faktorisierung von dünnbesetzten Matrizen. Für alle drei Testapplikationen wurden Laufzeitmessungen auf zwei verschiedenen Hardwareplattformen durchgeführt. Die Ergebnisse zeigen, dass es keinen Sieger unter den sieben Taskpoolvarianten gibt. Die Effizienz einer Taskpoolvariante hängt entscheidend von der Art der Applikation ab, die diesen Taskpool verwendet. Während Programme, die sehr viele Tasks erzeugen und häufig auf den Taskpool zugreifen müssen, von Taskpools mit dezentralen und privaten Taskqueues profitieren, benötigen Programme, die wenige Tasks erzeugen, eine gute Lastenbalancierung, sodass eine Taskpoolvariante mit einer zentralen Queue hier die bessere Wahl ist. Um eine gute Performance zu erzielen, sollte der Taskpool an die Eigenschaften der Applikation angepasst werden.

Aber nicht nur die Art und die Anzahl der verwendeten Taskqueues sind

für die Performance der Taskpools von Bedeutung. Auch die Reihenfolge, in der neue Tasks in die Queues eingefügt werden, kann sich auf die Performance auswirken. So ermöglichen die Taskpools mit LIFO-Queues eine bessere Ausnutzung der Lokalität der Speicherzugriffe. Auch hier hängt es von der Applikation ab, ob LIFO-Queues eine bessere Performance erreichen oder nicht.

Zusätzlich kann auch die Speicherverwaltung Auswirkungen auf die Performance eines Programms haben. Wenn ein Programm viele Tasks erzeugt, muss es häufig Speicher allokieren. Je nach Betriebssystem kann sich das negativ auf die Performance auswirken. In so einem Fall ist die Verwendung eines Speichermanagers sinnvoll.

In dieser Arbeit wurden nur einige Taskpoolvarianten untersucht. Es gibt jedoch viele andere Möglichkeiten einen Taskpool zu implementieren. Einige davon könnten möglicherweise bessere Performance erzielen, als die in dieser Arbeit beschriebenen Taskpoolvarianten. Zusätzlich können auch andere Verfahren zur dynamischen Lastenbalancierung untersucht werden.

Literaturverzeichnis

[1] *OpenMP Application Programm Interface, Version 2.5*, May 2005.

[2] Rohit Chandra, Leonardo Dagum, and Dave Kohr. *Parallel Programming in OpenMP*. Morgen Kaufman, October 2000.

[3] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Perform. Eval.*, 6(1):53–68, 1986.

[4] Matthias Korch and Thomas Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms. *Concurrency and Computation: Practice and Experience*, 16(1):1–47, 2004.

[5] Chuck Lever and David Boreham. malloc() performance in a multithreaded linux environment. Technical report, May 2000.

[6] Thomas Rauber and Gudula Rünger. *Parallele und verteilte Programmierung*. Springer-Verlag Berlin Heidelberg, 2000.

[7] Sanjiv Shah, Grant Haab, Paul Petersen, and Joe Throop. Flexible control structures for parallelism in OpenMP. In *Proceedings of the First European Workshop on OpenMP - EWOMP'99*, September 1999.