technische universität
dortmund

Diplomarbeit

Beschleunigung ausgewählter paralleler Standard Template Library Algorithmen

Sven Mallach

26. November 2008

Erst-Gutachterin: Prof. Dr. Petra Mutzel
Zweit-Gutachter: Dipl.-Inf. Carsten Gutwenger

Fakultät für Informatik
Algorithm Engineering (Ls11)
Technische Universität Dortmund
http://ls11-www.cs.uni-dortmund.de

Widmung

Diese Arbeit ist allen mir nahe stehenden Personen gewidmet, die in den letzten Jahren während meiner Prüfungsphasen, der Projektgruppe und der Anfertigung dieser Diplomarbeit mit den Folgen von Stress, Anspannung und Zeitmangel leben mussten und mir dennoch immer zur Seite standen. Besonderer Dank gilt meiner Mutter, die mir in dieser Zeit stets den Rücken freigehalten hat.

Danksagungen

Mein Dank gebührt meinem Betreuer Carsten Gutwenger für die permanente und wochentagunabhängige Unterstützung bei meiner Arbeit. Des Weiteren danke ich Danny van Dyk für rege Diskussionen über C++ und Programmierkonzepte, Dominik Göddeke für die Bereitstellung von Zugängen zu Testsystemen sowie Stephan Brabender, Markus Geveler, Dirk Ribbrock, Andreas Schmutzer und Bastian Steinbach für Korrekturen und Anmerkungen zum Text.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Multicore-Architekturen . . . 1

1.2 Standard Template Library . . . 4

2 Explizite Parallelisierung der STL **7**

2.1 Standard Template Adaptive Parallel Library . . . 7

2.1.1 Aufbau und API . . . 8

2.1.2 Realisierung paralleler Ausführung und Ergebnisse . . . 11

2.1.3 Fazit . . . 15

2.2 Multi-Processing Template Library . . . 16

2.2.1 Aufbau und API . . . 16

2.2.2 Realisierung paralleler Ausführung und Ergebnisse . . . 18

2.2.3 Fazit . . . 25

2.3 Multi-Core Standard Template Library . . . 26

2.3.1 Aufbau und API . . . 26

2.3.2 Realisierung paralleler Ausführung und Ergebnisse . . . 28

2.3.3 Fazit . . . 37

2.4 Vergleich der drei Parallelisierungsansätze . . . 39

3 Beschleunigung ausgewählter STL-Algorithmen **41**

3.1 Entwickeltes Multithreading-Konzept . . . 41

3.1.1 Begriffserklärung . . . 41

3.1.2 Konstruktion der Multithreading-Basis . . . 42

3.1.3 Wesentliche Designentscheidungen . . . 46

3.1.4 Thread Affinity . . . 48

3.1.5 Vermeidung häufiger Fehler . . . 52

3.2 Speedup-begrenzende Faktoren . . . 53

3.2.1 Architekturen und Einfluss des Speichersystems . . . 53

3.2.2 Experimentelle Untersuchungen . . . 55

3.2.3 Maßnahmen zur effektiven Ausnutzung der Speicherbandbreite . . . 59

3.2.4 Benchmarks zur Speicherbandbreite . . . 61
3.3 Auswahl der Algorithmen für die Experimente . . . 70
3.3.1 Nicht-mutierende Algorithmen . . . 70
3.3.2 Mutierende Algorithmen . . . 71
3.3.3 Numerische Algorithmen . . . 78
3.3.4 Sortierverfahren . . . 82

4 Ergebnisse 87
4.1 Testumgebung . . . 87
4.1.1 Systeme . . . 87
4.1.2 Betriebssysteme und Compiler . . . 87
4.1.3 Testbedingungen . . . 89
4.1.4 Zeitnahme . . . 91
4.2 Benchmarkergebnisse . . . 91
4.2.1 Nicht-mutierende Algorithmen . . . 91
4.2.2 Mutierende Algorithmen . . . 96
4.2.3 Numerische Algorithmen . . . 104
4.2.4 Sortieralgorithmen . . . 116

5 Evaluierung 125
5.1 Zusammenfassung und Diskussion der Ergebnisse . . . 125
5.2 Einschränkungen und Grenzen . . . 128
5.3 Optimierungsvorschläge . . . 130
5.4 Fazit und Ausblick . . . 131

Abbildungsverzeichnis 136

Literaturverzeichnis 141

Erklärung 143

Kapitel 1

Einleitung

1.1 Multicore-Architekturen

In der Vergangenheit wurden Steigerungen der Rechenleistung in erster Linie durch Erhöhung der Taktfrequenz oder das Verwenden mehrerer Einkern-Prozessoren auf einer Hauptplatine erreicht. Mit Taktfrequenzen um vier GHz stießen die Hersteller aber zunehmend an elektrotechnische und physikalische Grenzen, insbesondere weil sich die Wärmeverlustleistung und damit auch die Energieeffizienz in nicht mehr vertretbare Dimensionen entwickelte.

Nachdem zunächst nur einzelne Prozessorkomponenten wie etwa Registersätze dupliziert wurden, um zwei logische innerhalb eines physischen Prozessors zu ermöglichen (vgl. *Hyper-Threading*), gibt es bei der Entwicklung von Desktop-, Server- und High-Performance-Prozessoren, neben der ständigen Weiterentwicklung von Vektorbefehlssätzen wie z.B. SSE, in der jüngsten Zeit einen klar erkennbaren Trend zu Multicore-Architekturen.

Dabei werden mehrere vollständige Recheneinheiten, also Kerne (*Cores*) mit eigenen Rechenwerken, Caches und teilweise auch Speichercontrollern gemeinsam auf einem Prozessor-Die platziert. Theoretisch wird somit die Rechenleistung um die Anzahl der Kerne vervielfacht. Auf diese Weise können Taktfrequenzen moderat gehalten werden und durch die Möglichkeit, ein Vielfaches der Instruktionen gleichzeitig auszuführen, wird auch ein wesentlich besseres Verhältnis von Rechenleistung und Energieverbrauch erreicht.

In der Praxis skaliert der erreichbare Beschleunigungsfaktor (*Speedup*) jedoch selten mit der Anzahl der Kerne. Einerseits verhindern Hardware-Aspekte wie der Flaschenhals des Speicherdurchsatzes, Cache Misses oder TLB Misses, sowie Software-Einflüsse wie das Scheduling des Betriebssystems eine lineare Skalierung. Andererseits beschränken Datenabhängigkeiten innerhalb eines Programms seinen parallelisierbaren Anteil. Nach Amdahls Gesetz [17] lässt sich der Speedup S durch parallele Ausführung auf n Prozessoren gegenüber der sequentiellen Ausführung allgemein ausdrücken als:

$$S = \frac{T_{sequentiell}}{T_{parallel}} = (1-p) + \frac{p}{n} + O$$

Dabei wird mit p der parallelisierbare Anteil des Programms beziffert und mit O der Overhead, der durch die Partitionierung und das Anstoßen von Threads zu erwarten ist.

Die Prozessoren verschiedener Hersteller von Multicore-Architekturen unterscheiden sich wesentlich in ihren Merkmalen. Vor allem die Speicherhierarchien weisen bei den verschiedenen Repräsentanten sehr markante Unterschiede auf, wie ein Blick auf die neuesten Entwicklungen der beiden führenden Hersteller von Desktop- und Serverprozessoren zeigt. So haben Intels Dualcore Prozessoren auf Basis der „Core“-Architektur zwei Recheneinheiten mit jeweils eigenem L1-, aber geteiltem (*shared*) L2-Cache (siehe Abbildung 1.1). Intel führt dieses Schema bei Vierkernprozessoren fort, d.h. vier Recheneinheiten erhalten ihren eigenen L1-Cache, jedoch teilen sich zwei Kerne weiterhin einen L2-Cache. Die Anbindung zum Hauptspeicher und zu anderen Prozessoren wird über den *Front Side Bus* (FSB) realisiert, welcher die Verbindung zur *Northbridge* des Chipsatzes (bei Intel als *Memory Controller Hub* (MCH) bezeichnet) herstellt. In der Regel ist in einem System nur ein FSB vorhanden. In einigen Server-Chipsätzen für Zweiprozessorsysteme existiert jedoch für jede CPU eine Verbindung zum MCH. Dies wird von Intel heute als *Dual Independent Bus* vermarktet [12], während dieser Begriff früher das im Pentium II eingeführte System zweier getrennter Busse zur Verbindung der CPU mit dem Hauptspeicher und dem L2-Cache beschrieb.

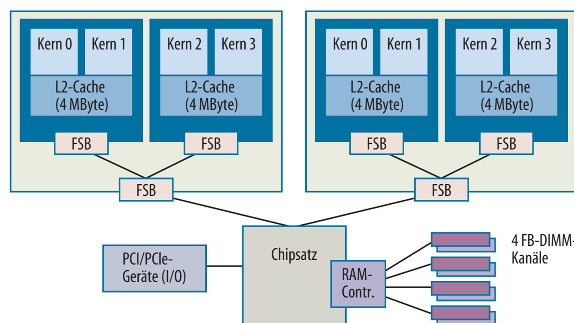

Abbildung 1.1: Vierkern-Konzept von Intel. Quelle: [52]

Konkurrent AMD verfolgt ein anderes Konzept. Seit mit der K9-Architektur erste Dualcore-Prozessoren eingeführt wurden, besitzen diese getrennte L1- und L2-Caches. Zur Abgrenzung gegenüber der Intel-Architektur, bei der immer zwei Kerne einen Verbund bilden, spricht man auch von *monolithischen* Kernen. Mit den aktuellen Vierkernprozessoren (K10-Generation, Codenamen „Barcelona“, „Agena“) hat auch AMD sein Paradigma weitergeführt, d.h. es existieren vier getrennte L2-Caches, die nun durch einen kohärenten shared L3-Cache ergänzt werden (siehe Abbildung 1.2). Die Verbindung mehrerer

Prozessoren untereinander, sowie zum Hauptspeicher, erfolgt über den *HyperTransport*-Bus [8] und einen *Crossbar-Switch*. Der Speichercontroller wurde bei AMD bereits in die CPU integriert und muss nicht mehr über die Northbridge angesprochen werden. Jeder Prozessor hat folglich eigene Speicherbänke, weshalb über den eigenen Speichercontroller allozierter Hauptspeicher schneller zugreifbar ist, als der Hauptspeicher anderer Controller. Man spricht deshalb von einer *Non Uniform Memory Architecture* (NUMA), präziser von einer *cache-coherent* NUMA (cc-NUMA). Das Ziel einer solchen Architektur ist es, in Multiprozessor-Systemen nicht mehr alle Speicheranfragen zentral über den Chipsatz abwickeln zu müssen, welcher sonst schnell einen Flaschenhals darstellen kann. Für die bei Abschluss dieser Arbeit veröffentlichte neue Generation der Intel Multicore-Prozessoren („Core i7", „Nehalem"-Architektur) ist daher ebenfalls ein integrierter Speichercontroller vorgesehen.

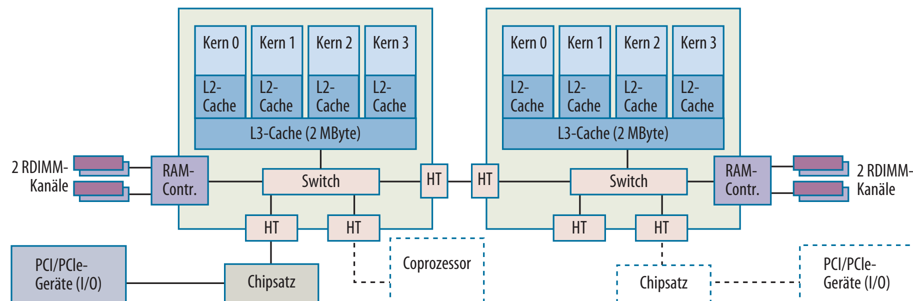

Abbildung 1.2: Vierkern-Konzept von AMD. Quelle: [52]

Ein weiteres wichtiges Unterscheidungsmerkmal aktueller CPUs ist die maximale Anzahl von *leichtgewichtigen* Prozessen, *Threads*, die simultan auf einem Kern ausführbar sind (*Simultaneous Multithreading, SMT*). Im Fall aktueller Prozessoren von AMD und Intel kann zu einem Zeitpunkt nur ein Thread auf einem Kern zur Ausführung kommen. Andere Architekturen ermöglichen hingegen die gleichzeitige Ausführung von zwei oder mehr Threads. Ein Sun UltraSPARC T1 Prozessor [9] mit acht Kernen kann beispielsweise 32 Threads parallel ausführen, sein Nachfolger T2 64 Threads auf 16 Kernen. Auch in diesem Bereich plant Intel, durch eine Weiterentwicklung des Hyper-Threading-Konzepts, mit der Nehalem-Architektur zwei Threads effizient parallel auf einem Kern ausführen zu können.

Auf Grund dieser und weiterer wesentlicher Unterschiede im Design von Multicore-Prozessoren ist es eine komplizierte Aufgabe, einen Algorithmus zu implementieren, dessen Laufzeit auf den verschiedenen Architekturen skaliert. Ähnlich wie bei den eingangs erwähnten Multiprozessorsystemen mit mehreren Einkern-Prozessoren, handelt es sich auch bei Multicore-Architekturen um *Symmetric Multiprocessing, SMP* Systeme. Ihre effiziente Nutzung ist nur durch geeignete Parallelisierungsmaßnahmen, insbesondere den angesprochenen Threads, möglich. Andernfalls obliegt es lediglich dem Betriebssystem schwerge-

wichtige Prozesse auf unterschiedliche Kerne zu verteilen. Die Last eines einzelnen besonders ressourcenbindenden schwergewichtigen Prozesses kann dann jedoch nicht verteilt werden. Auf diese Weise blieben in Zukunft zunehmend mehr Ressourcen ungenutzt, da die Hersteller immer mehr Kerne pro Prozessor und immer mehr mögliche Threads pro Kern vorsehen.

Eine weitere Herausforderung bei mehreren Kernen auf einem Chip ist die Frage der Speicherauslastung und der Erzeugung eines kontinuierlichen Datenflusses zu allen Kernen. Da zunehmend mehr Kerne eine Anbindung zu gemeinsamem Speicher besitzen, werden Verhältnisse wie „Speicherbandbreite pro Thread“ oder auch „Cachegröße pro Thread“ ein immer wichtigerer Aspekt bei der Programmierung von effizienter Software. Zusätzlich werden zukünftige Prozessoren mit z. B. 32 oder mehr Kernen keinesfalls so homogen bzw. monolithisch sein können wie AMDs Quadcores heute. Des Weiteren ermöglichen Multicore-Prozessoren, wie in den Abbildungen 1.1 und 1.2 angedeutet, den Aufbau von Multiprozessor-Systemen, deren CPUs selbst bereits SMP-Systeme sind. Zwei Kerne auf demselben Prozessor haben dabei eine andere Beziehung zu einander, als Kerne auf unterschiedlichen Prozessoren. Damit wird die Kommunikation zwischen Threads auf zwei verschiedenen Kernen unterschiedlich teuer. Auch der Zugriff auf Caches bestimmter Kerne sowie die Sicherstellung ihrer Kohärenz wird aufwändiger.

Es existieren zwar Ansätze zur automatischen Ausnutzung von Parallelität durch Compiler (implizite Parallelisierung), jedoch hängt der Effizienzgewinn sehr stark von den Eigenschaften des Programms und dem verwendeten Compiler ab und kann zum jetzigen Zeitpunkt handoptimiertes Multithreading (explizite Parallelisierung) nicht ersetzen. Die Aufgabe, Probleme zu partitionieren und an die Recheneinheiten zu verteilen, liegt also beim Programmierer. Er selbst muss parallel ausführbare Programmabschnitte finden und mit Hilfe von Threads dafür sorgen, dass die verfügbaren Kerne möglichst gut ausgelastet werden.

Die beschriebene Entwicklung und die realistische Erwartung, dass in naher Zukunft das Gros der verfügbaren Computersysteme mit Multicore-Prozessoren ausgestattet sein wird, hat neben dem Wunsch nach verbesserter impliziter Parallelisierung durch Compiler auch zu Überlegungen und Ansätzen geführt, von vielen Programmen durch Bibliotheken verwendete Grundbausteine effizient zu parallelisieren. Ein Beispiel für eine solche viel verwendete Basisbibliothek ist die Standard Template Library.

1.2 Standard Template Library

Die Standard Template Library (STL) ist eine C++ Bibliothek, deren anfängliche Entwicklung innerhalb der Firma Hewlett-Packard bereits in den 70er Jahren vorangetrieben wurde. Später wechselte einer der Entwickler, Alexander Stepanov, zur Firma Sillicon Graphics (SGI), wo er die Arbeiten an der STL fortsetzte und sie stark erweitert wurde. Aus

diesem Grund wird sie auch heute noch von SGI zur Verfügung gestellt [6]. Gegenwärtig ist die STL in ihrer bei Hewlett-Packard entwickelten Form weitgehend Bestandteil der C++ Standardbibliothek, die der Programmierer über den Namensraum `std::` verwenden kann.

Die STL ist eine hochgradig generische Bibliothek, basiert also in sehr hohem Maße auf *Templates*. Sie stellt in erster Linie Container-Klassen, grundlegende Algorithmen, Datenstrukturen und Iteratoren zur Verfügung, die auf Grund der Templates mit beliebigen Datentypen, also auch eigenen abstrakten Datentypen, verwendet werden können. Container wie `vector` und `list`, sowie Algorithmen wie `sort()`, `copy()`, `find()` oder `fill()` bieten von verschiedensten Programmen häufig genutzte grundlegende Funktionalität. Die Algorithmen der STL können dabei grob in vier Klassen eingeteilt werden: mutierende, nicht-mutierende, numerische und Sortier-Algorithmen. Mutierende Algorithmen verändern Werte von Containern. Ein Beispiel hierfür ist `fill()`. Bei nicht-mutierenden Algorithmen sind sämtliche Container invariant, etwa bei `find()`. Zu den numerischen Algorithmen gehören beispielsweise `inner_product()` oder `partial_sum()`. Eine vollständige Liste mit Erläuterungen zur Funktionalität findet sich in [6] und [32].

Ein weiterer essentieller Baustein der STL sind *Iteratoren*. Ähnlich wie Zeiger verweisen sie auf ein Element einer Datenstruktur, etwa einer Liste oder eines Vektors. Per Dereferenzierung kann auf den Wert des Elementes zugegriffen werden. Wie der Name bereits nahe legt, eignen sich Iteratoren insbesondere für das Durchlaufen von Containern oder Teilen davon. Damit bilden sie die Schnittstelle zu generischen Algorithmen, die statt Containern Iteratoren als Argumente erhalten. Dabei kann derselbe Iteratortyp für viele Container geeignet sein. Insgesamt unterscheidet die STL fünf Iteratortypen, die hierarchisch aufeinander aufgebaut sind, d.h. komplexere Typen erfüllen auch die Anforderungen einfacherer Typen. `Input`- und `OutputIterator` sind die einfachsten Vertreter. Sie garantieren lediglich einmaliges Lesen bzw. Schreiben und ermöglichen das Inkrementieren des Iterators. `ForwardIterator` erweitert die beiden vorherigen Konzepte um die Garantie, dass Werte mehrmals lesbar sind, also auch nach dem Inkrementieren ihre Gültigkeit behalten. Man kann sie also gefahrlos kopieren und mehrmals dereferenzieren. `BidirectionalIterator` erlaubt dies ebenfalls, bietet aber zusätzlich die Möglichkeit zum Dekrementieren des Iterators. Zuletzt repräsentiert `RandomAccessIterator` nahezu die vollständige Funktionalität von Zeigern, inklusive Arithmetik- und Vergleichsoperationen. Das Verschieben eines `RandomAccessIterator` um k Positionen ist daher in amortisiert konstanter Zeit möglich, während für alle anderen Iteratortypen Zeit $\mathcal{O}(k)$ veranschlagt wird.

Auf Grund der hohen Verbreitung der STL und verschiedenster Anforderungen hinsichtlich der Performance wurden bereits mehrere Versuche unternommen, die zur Verfügung gestellten Algorithmen und Datenstrukturen effizient zu parallelisieren.

Ziel dieser Arbeit wird es sein, ausgewählte STL-Algorithmen zu beschleunigen, für die insbesondere bei kleineren und mittleren Eingabegrößen mit den bisherigen Ansätzen

kein oder nur sehr wenig Speedup erzielt werden konnte. Dazu ist eine detaillierte Analyse der Implementierung der bisherigen Ansätze erforderlich, um Ergebnisse vergleichen zu können und an den richtigen Stellen für eine Optimierung anzusetzen. Die vorliegende Diplomarbeit ist daher wie folgt aufgebaut.

Kapitel 2 vollzieht die angesprochene detaillierte Analyse für die drei wichtigsten bekannten parallelen STL-Bibliotheken. In Kapitel 3 wird das im Rahmen der vorliegenden Arbeit entwickelte Multithreading-Konzept erläutert und die Auswahl der zu parallelisierenden STL-Algorithmen getroffen sowie begründet. Des Weiteren erfolgen experimentelle Untersuchungen bzgl. der den erzielbaren Speedup begrenzenden Faktoren. Die Ergebnisse der Arbeit werden in Kapitel 4 vorgestellt, Kapitel 5 schließt die Arbeit mit einem Fazit und einem Ausblick ab.

Kapitel 2

Explizite Parallelisierung der STL

In diesem Kapitel erfolgt der angekündigte Blick auf bereits existierende Ansätze zur expliziten Parallelisierung der STL. Erste Bemühungen in Richtung paralleler Iteratortypen und Algorithmen sind auf die im Jahre 1997 innerhalb des HPC++-Projekts erwähnte PSTL [33] zurückzuführen. Die letzte von den HPC++ Entwicklern veröffentlichte Version Ihres Projekts enthält jedoch keinen Quellcode zu dieser Bibliothek. Offenbar fand keine Weiterentwicklung statt, da die zugehörige Webseite seit 1999 nicht verändert wurde und keine aktuelle Literatur verfügbar ist. Die Entwickler der Multi-Core Standard Template Library (MCSTL) verweisen in [3] auf die Bibliothek STAPL als Pionier-Projekt dieser Idee und nennen die MPTL als weitere konkurrierende Bibliothek. Alle drei Bibliotheken wollen eine möglichst zur STL identische Programmierschnittstelle (*Application Programming Interface, API*) zur Verfügung stellen, um insbesondere eine gleichartige Nutzung zu ermöglichen und damit die Parallelisierung bereits existierender, auf der STL basierender, Programme zu erleichtern. Bei genauerer Untersuchung zeigt sich jedoch, dass sie sich deutlich von einander unterscheiden und für gewisse Anforderungen stärker geeignet bzw. optimiert sind, als für andere.

2.1 Standard Template Adaptive Parallel Library

STAPL, die Standard Template Adaptive Parallel Library [18], wird an der Texas A & M Universität entwickelt und ist der erste bekannt gewordene Ansatz, die Algorithmen und Datenstrukturen der STL vollständig zu parallelisieren. Ziel der Entwickler ist es, effektive Parallelisierung sowohl für *Shared Memory*, d.h. SMP-, NUMA- oder Multicore-Systeme, als auch für *Distributed Memory Systeme* (Cluster) zu erzielen. Die Bibliothek wird in einigen wissenschaftlichen Projekten der Universität eingesetzt, bei denen hohe arithmetische Rechenleistung bei sehr großen Eingaben erforderlich ist. Dazu gehören etwa die Zerlegung von Eiweißstoffen (protein folding) oder die Ermittlung von seismischen Lichtstrahlen (seismic ray tracing).

2.1.1 Aufbau und API

Das STAPL zugrundeliegende Programmiermodell wird als *SPMD (Single Program, Multiple Data)* bezeichnet. SPMD bedeutet, dass mehrere Ausführungseinheiten dasselbe Programm ausführen und dabei von einander unabhängige Daten manipulieren. Auf diese Weise werden Situationen gemieden, die gegenseitigen Ausschluss zwischen Prozessen erfordern und somit die Programmierung vereinfacht. Der Begriff lehnt sich an die Bezeichnung SIMD (Single Instruction, Multiple Data) an, da in Vektorprozessoren ebenfalls mehrere voneinander unabhängige Daten gleichzeitig manipuliert werden. Für die Parallelisierung der STL-Funktionen ist dieses Programmiermodell passend, da in den meisten Fällen lediglich die Eingabesequenz zu partitionieren und anschließend die entsprechende Funktion auf allen Teilintervallen auszuführen ist.

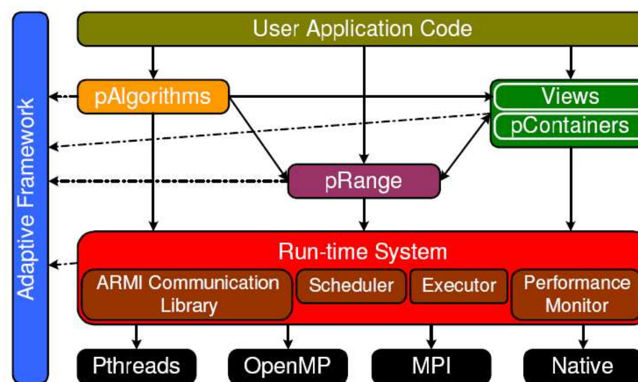

Abbildung 2.1: Schematischer Aufbau der STAPL Bibliothek (Quelle: [43])

Abbildung 2.1 zeigt den schematischen Aufbau der Bibliothek. Als Basis für die Realisierung der Parallelität können alle gängigen APIs verwendet werden. Namentlich sind dies die *Native POSIX Threads Library* (NPTL, „Pthreads“ [25]), *OpenMP* [16], das *Message Passing Interface* (MPI) [4], sowie native Multithreading-Konzepte wie z. B. *LinuxThreads*. Damit unterstützt STAPL sowohl Shared Memory Synchronisation zwischen lokalen Threads, als auch die Verwendung von Message Passing als Kommunikationsbasis zwischen mehreren Systemen. Um Portabilität zu erreichen und sämtliche Kommunikationsdetails zu verdecken, wird dieser Ebene eine eigene Kommunikationsschicht übergeordnet, welche auf der Verwendung von RMI [30] (Remote Method Invocation, verwandt mit Remote Procedure Call (RPC)) basiert. Da es sich um eine adaptive Bibliothek handelt, haben die Entwickler diese Schicht als ARMI (Adaptive RMI) bezeichnet [43].

ARMI ist für drei wesentliche Aufgaben innerhalb von STAPL zuständig:

- Objekt-Registrierung
- Prozess-Kommunikation

- Prozess-Synchronisation

Jedes Objekt, das von mehreren Prozessen oder Threads bearbeitet wird und somit Gegenstand von Kommunikation sein kann, muss zunächst in ARMI registriert werden. Jeder Thread besitzt einen lokalen Repräsentanten (z. B. ein Teilintervall) eines solchen Objekts und eine *RMI Registry*, welche in erster Linie ein Vektor von Zeigern auf lokale Repräsentanten ist [42]. Bei der Registrierung übergeben alle Threads ihren lokalen Zeiger auf das Objekt an eine Funktion `register_rmi_object()`, und erhalten identische `rmiHandle`, welche einen Index in den beschriebenen Vektor darstellen. Da es sich um SPMD-Programmierung handelt, erzeugen alle Threads ihre lokalen Repräsentanten in gleicher Reihenfolge, sodass die Indizes für alle Threads auf das gleiche gemeinsame Objekt verweisen [42]. Mit Hilfe des `rmiHandle`s und einer Thread-ID kann folglich auf lokale Repräsentanten eines anderen Threads zugegriffen werden. Eine anderweitige Manipulation globaler Variablen (wie z. B. auch statischer Klassenattribute) ist in ARMI nicht vorgesehen, weil diese in Shared Memory und Message Passing Umgebungen zur Laufzeit unterschiedlich behandelt würden. In ersteren existiert nur eine einzige Instanz einer solchen Variable, wodurch *Race Conditions* unvermeidbar wären, während jeder MPI-Thread eine lokale Kopie erhält und somit ein Thread die Änderungen anderer Threads nicht zur Kenntnis nehmen könnte [42].

RMI-Aufrufe sind ursprünglich blockierend, jedoch wurden im Bereich des High Performance Computings und innerhalb ARMI asynchrone Variationen entwickelt. Bei diesen Funktionen muss dann ein *Callback-Handler* verwendet werden, also ein Objekt mittels dessen die abgeschlossene Ausführung einer Funktion und etwaige Rückgabewerte überprüft werden können. Alternativ kann die Beendigung vorausgegangener RMI-Aufrufe durch Synchronisationsfunktionen wie `rmi_fence()` sichergestellt werden. Die nicht-blockierende Form der asynchronen RMI-Funktionen erlaubt darüber hinaus ihre Aggregation, also die temporäre Sammlung in Puffern bis zum späteren gebündelten Transfer. Auf diese Weise wird insbesondere bei Distributed Memory Systemen die Bandbreite der Verbindungsinfrastruktur besser genutzt und Overhead für viele kleine Nachrichten verhindert. ARMI implementiert beide Konzepte, synchrone (blockierende) Aufrufe für die einfache Übergabe von Rückgabewerten und asynchrone (nicht-blockierende) Aufrufe für hohe Leistungsanforderungen [43, 48, 42].

Die grundlegende Idee und der wesentliche Unterschied zu anderen Kommunikationsmodellen besteht darin, dass in ARMI nicht in erster Linie Daten zwischen Threads ausgetauscht werden, sondern die vollständige zu verrichtende Arbeit, inklusive der aufzurufenden Methode. Die Signaturen der folgenden Kommunikationsprimitiven enthalten daher alle einen Parameter `method`, welcher einem Zeiger auf eine Methode (Funktionszeiger) entspricht. Der Parameter `dest` repräsentiert jeweils die ID des Zielthreads.

- `void async_rmi(dest, rmiHandle, method, arg1...)`

- `returnType sync_rmi(dest, rmiHandle, method, arg1...)`
- `returnType sync_rmi(opaque, dest, rmiHandle, method, arg1...)`

Die letzte Funktion entspricht der nicht-blockierenden HPC-Variante der ursprünglich synchronen Funktion, weshalb sich die Signatur um den Callback-Handler `opaque` erweitert. Dieser stellt eine Methode `ready()` zur Verfügung, die verwendet werden kann, um die Verfügbarkeit des Rückgabewerts zu überprüfen und eine Funktion `value()`, um ihn auszulesen [48]. Es bestehen weitere Funktionen, etwa für die Kommunikation zu allen Threads (`broadcast_rmi()`) und zur Reduktion lokaler Teilergebnisse zu einem Gesamtergebnis (`reduce_rmi()`) [42].

Zur Veranschaulichung dient ein Beispiel eines bucketbasierten Sortieralgorithmus. Angenommen, ein Thread hat in seinem lokalen Intervall `input` ein Element an Position `i` gefunden, welches in den Bucket eines anderen Threads mit der ID `thread_id` gehört. Dann kann er diesen mit folgendem Funktionsaufruf anweisen, dieses Element in seinem lokalen Bucket (dessen Adresse in der RMI Registry an Position `handle_index` abgelegt wurde) zu speichern.

```
async_rmi(thread_id, handle_index, &pvector::push_back, input[i]);
```

Wird eine Methode von zwei oder mehr Prozessen gleichzeitig aufgerufen, stellen die entsprechenden RMI-Funktionen eine atomare Ausführung sicher und garantieren somit gegenseitigen Ausschluss [42]. Für das Warten auf einen eingehenden RMI-Aufruf steht die Funktion `rmi_wait()` zur Verfügung. Die bereits erwähnte Methode `rmi_fence()` kann zur Synchronisation aller Threads verwendet werden. Während auf den Moment der Synchronisation gewartet wird, überprüft `rmi_fence()` in regelmäßigen Abständen das Eingehen weiterer RMI-Anfragen und kann diese bereits an die wartenden Threads delegieren [42]. Bei asynchronen Funktionsaufrufen muss sichergestellt sein, dass adressierte Threads die Anfrage wahrnehmen. Mögliche Lösungsansätze für dieses Problem sind Interrupts, separate Kommunikationsthreads, welche allein die Kommunikation abwickeln und nicht an der Bearbeitung von Nutzdaten teilnehmen, sowie explizites Abfragen (*explicit polling*). ARMI verwendet laut [43, 48] und [42] explizites Abfragen zwischen Aufrufen von Kommunikations- und Synchronisationsfunktionen. Dies ist einerseits transparent für den Benutzer, hat jedoch hohe Reaktionszeiten zur Folge, falls über einen langen Zeitraum keine Kommunikationsfunktionen aufgerufen werden [42]. Im gegensätzlichen Fall sehr starker Kommunikation wird nur noch nach jedem k-ten Aufruf ein Poll gestartet, wobei k ein Tuningparameter ist, der durch den Benutzer manipuliert werden kann. Niedrige Werte haben eine schnellere Reaktion zur Folge, verlangsamen aber potenziell den Fortschritt der Berechnung. Hier bleibt Raum für Optimierung, die Entwickler forcierten bereits in [43] und [48] eine Variante mit Kommunikationsthreads, wodurch aber neue Probleme entstehen, wie etwa die effiziente Realisierung solcher Threads parallel zu den die eigentliche Arbeit verrichtenden Threads, sowie Abhängigkeiten vom Thread Scheduling.

Um die Parameter eines RMI-Aufrufs speichern, übermitteln und an anderer Stelle ausführen zu können, sowie eine Optimierung für jede als Basis zur Verfügung stehende Multithreading-API zu ermöglichen [42], erfolgte ihre Implementierung mit Hilfe von als `virtual` deklarierten Funktionsobjekten und Templates. Ein Funktionsobjekt ist ein Konzept der Programmiersprache C++. Es handelt sich dabei um eine Instanz einer Klasse, welche den Funktionsoperator `operator()` definiert und somit wie eine Funktion aufgerufen werden kann. Eine ebenfalls häufig verwendete Bezeichnung für ein solches Objekt ist *Funktor*. Dennoch wird durch das SPMD-Modell und die Datenstrukturen zur Adressübersetzung deutlich, dass STAPL vorwiegend auf die Nutzung in Distributed Memory Systemen spezialisiert ist. Denn in Shared Memory Systemen sind lokale Zeiger für alle Threads gültig, lokale Kopien bei Existenz entsprechender Funktionen für gegenseitigen Ausschluss nicht notwendig. Für die Verwendung in reinen Shared Memory Systemen offenbaren sich darüber hinaus noch weitere Nachteile. In [43] und [42] wurde auf einigen großen SMP-Architekturen und Clustern die Latenz untersucht, die durch die zusätzliche RMI-Ebene gegenüber einer direkten Shared Memory oder Message Passing Implementierung entsteht, indem Recheneinheiten Anfragen an eine andere Recheneinheit senden und die Ankunftszeit der Antwort gemessen wird („ping pong benchmark“). Dabei stellte sich heraus, dass insbesondere gegenüber der direkten Verwendung der NPTL ein signifikanter Overhead entsteht. Mit Hilfe der bereits angesprochenen Aggregation von asynchronen Aufrufen kann dieser Nachteil jedoch für größere Eingaben ausgeglichen, bzw. können sogar Vorteile erzielt werden. Auch die Ausführung einer Funktion dauert jedoch über RMI-Aufrufe wesentlich länger, als durch direkte Methodenaufrufe oder die Verwendung von einfachen Funktionszeigern, wie sie in Shared Memory Systemen ohne weiteres möglich sind. Weiterer Overhead entsteht durch die Notwendigkeit alle zu übergebenden Funktionsparameter zu kopieren, sowie durch zusätzliche Dereferenzierungen durch den Einsatz virtueller Methoden und der Adressübersetzung per `rmiHandle`. In der aktuell verfügbaren Literatur bietet die Masterarbeit von Steven Saunders [42] den besten Überblick für weitere Details zur ARMI-Bibliothek.

2.1.2 Realisierung paralleler Ausführung und Ergebnisse

Nachdem die Kommunikation zwischen parallel arbeitenden Threads bereits diskutiert wurde, folgt nun ein Blick auf die Realisierung der parallelen Ausführung selbst, sofern dies möglich ist, da STAPL noch nicht veröffentlicht ist und auch auf Anfrage weder Einblick in den Code gewährt, noch ausführbarer Objektcode zur Verfügung gestellt wurde. Leider spiegelt die verfügbare Literatur den aktuellen Stand der Bibliothek nicht mehr vollständig wieder, weshalb persönliche Kommunikation mit den leitenden Entwicklern, in erster Linie mit Prof. Dr. Lawrence Rauchwerger, erforderlich war, um sichere Aussagen über die verwendeten Konzepte treffen zu können.

STAPL bietet dem Nutzer einen *„shared object view“* auf die Daten, d.h. er kann Objekte so verwenden, als ob sie lokal gespeichert sind, obwohl sie über den gesamten adressierbaren Speicherbereich verteilt werden, welcher sowohl mehrere Rechner umfassen, als auch lediglich zwischen mehreren Prozessen geteilt sein kann. Die Bibliothek stellt parallelisierte Versionen der STL-Pendanten mit den Namen *pContainers* und *pAlgorithms* zur Verfügung. Die Daten eines pContainers werden von Beginn an auf den lokalen Adressraum mehrerer Threads verteilt [42] und sämtliche Berechnungen und Datenübertragungen sofern möglich parallelisiert, wie z. B. der Datenstrom beim Aufruf von Kopier-Konstruktoren [18]. Da viele STL-Container-Datenstrukturen auf Bäumen basieren, wurde auch eine parallelisierte Baum-Datenstruktur namens *pTree* entwickelt, die paralleles Einfügen und Entfernen unterstützt. Ein Prozessor verwaltet dann einen oder mehrere Teilbäume eines pTree, während auf oberster Ebene nur atomare Operationen ausgeführt werden, um die Konsistenz der Datenstruktur sicher zu stellen. Diese Konzepte erfordern laut [18] nur vernachlässigbar mehr Speicherplatz gegenüber den STL-Containern. Natürlich verursacht die Konsistenzerhaltung und Verwaltung der Bäume aber einen gewissen Mehraufwand, insbesondere da Prozessoren nicht nur Elemente ihrer eigenen Teilbäume manipulieren, sondern auch Austauschoperationen über temporäre Buckets vorgesehen sind [18]. In [46] werden einige Ergebnisse zum Overhead von asynchronen gegenüber synchronen Funktionsaufrufen für das Einfügen von Elementen in eine pMap und pHashMap für HPC-Cluster vorgestellt.

Als besondere und zentrale Entwicklung in STAPL stellt sich das Konzept der *pRange* heraus, welche ein Teilintervall eines Containers repräsentiert und die Rolle der Iteratoren `begin()` und `end()` einnimmt. Ähnlich wie die STL-Iteratoren bilden pRanges die Schnittstelle zwischen Containern und Algorithmen und werden ebenso ungültig, wenn Elemente im zugehörigen pContainer verändert werden [18]. Die Übergabe eines Intervalls als Parameter an einen Algorithmus erfolgt in STAPL folglich nicht mehr durch zwei Iteratoren, sondern durch eine pRange. Durch die Möglichkeit zur rekursiven Erzeugung von pRanges wird die Implementierung von *nested parallelism*, also Threads, die ihrerseits Threads erzeugen, unterstützt.

Ein pAlgorithm ist das parallele Gegenstück zu einem STL-Algorithmus, dessen Aufbau am Beispiel der Funktion `std::accumulate()` nachfolgend beispielhaft beschrieben wird. Zunächst wird ein Funktionsobjekt (siehe Quelltextauszug 2.1) definiert, welches die Funktion auf einer pRange ausführt und von der Basisklasse `stapl::work_function_base<T>` (in der älteren Literatur,z. B. [18], wird diese noch als `pFunction` bezeichnet) erbt. Die Hauptroutine des parallelen Algorithmus (Quelltextauszugs 2.2) instantiiert dieses Funktionsobjekt (Zeile 4) und erzeugt Tasks in Abhängigkeit der durch das pRange-Objekt definieren Intervalle (Zeile 5). Anschließend erfolgt die Allokation temporären Speicherplatzes für die Teilergebnisse der Threads und in Zeile 8 wird eine zugehörige Reduktionsklasse definiert. `p_for_all()` ist ein so genannter *„Parallel Region Manager“* (siehe Abbildung 2.2), d. h. ein für die jeweilige Art des Algorithmus angepasster Scheduler. Wie der Name

```
template<typename PRange, typename BinaryFunction>
class p_accumulate_w : public work_function_base<PRange>
{
   public:
   void operator()(typename PRange::subview_set_type& subrange_data)
   {
      this->m_subrange_result =
      std::accumulate(at<0>(subrange_data)->begin(),
      at<0>(subrange_data)->end(), ...);
   }
};
```

Quelltextauszug 2.1: Funktionsobjekt für `p_accumulate()`

```
template<typename PRange, typename T, typename BinaryFunction>
T p_accumulate(PRange& pr, T init, BinaryFunction binary_op)
{
    p_accumulate_w wf(pr, binary_op);
    pr.create_tasks(wf);
    tPRange::result_storage_type result_storage(pr);

    task_result_aggregator aggregator(result_storage.prange(), wf);
    p_for_all(pr, result_storage.prange(), wf);
    T result = binary_op(init, aggregator.result());

    return result;
}
```

Quelltextauszug 2.2: `p_accumulate()`

bereits suggeriert, handelt es sich im hier betrachteten Fall von `std::accumulate()` um eine linear auf dem Speicher arbeitende Funktion, die einer `for each`-Routine gleicht. Auch an dieser Stelle spielt das pRange-Konzept eine entscheidende Rolle. Der gerichtete Graph bestimmt im Fall von Datenabhängigkeiten die Bearbeitungsreihenfolge der Teilintervalle. Der Parallel Region Manager realisiert mit Hilfe des darauf beruhenden Schedulings und des übergebenen Funktionsobjekts die parallele Ausführung, d.h. die Erzeugung der Threads je nach verwendeter Multithreading-API [18, 39]. An dieser Stelle wird das SPMD-Modell erneut sichtbar, da der gleiche Funktionsoperator auf allen Teilintervallen ausgeführt wird.

Sowohl die Vergleichsergebnisse für verschiedene Sortierverfahren mit OpenMP und NPTL-Implementierungen in [43], als auch die Aussagen zu STL-Algorithmen in [18], deuten darauf hin, dass die Verwendung der NPTL als Multithreading-Basis in der Regel die schnellste (getestete) Variante ist. Da sich der Entwicklungsstand der Bibliothek je-

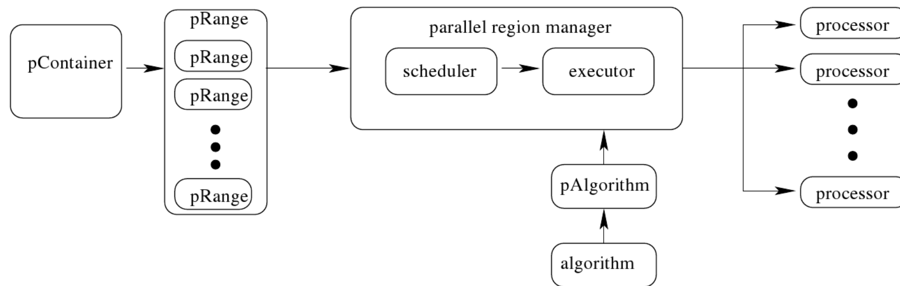

Abbildung 2.2: Zusammenhänge der STAPL-Komponenten bei der parallelen Ausführung (Quelle: [18])

doch seitdem maßgeblich verändert hat, können keine wissenschaftlich verwertbaren Rückschlüsse aus dieser Literatur gezogen werden. Zusätzlich ist auf Grund der verwendeten Architekturen ohnehin keine Vergleichbarkeit mit den im Rahmen der vorliegenden Arbeit erzeugten Ergebnissen gegeben.

Anders als bei den folgenden Bibliotheken MPTL und MCSTL wurde bei STAPL nicht nur Wert auf Leistung durch Parallelität gelegt, sondern als weiteres Merkmal die Adaptivität forciert. Adaptivität bedeutet, dass die Bibliothek in der Lage ist, zur Laufzeit den Geeignetesten aus einer Menge von zur Verfügung stehenden und semantisch äquivalenten Algorithmen auszuwählen. In der aktuelleren Literatur [40] wird dieser Teil von STAPL als eigenes Framework namens FAST (Framework for Algorithm Selection & Tuning) bezeichnet.

Um die korrekte Auswahl eines Algorithmus zu ermöglichen, betreibt STAPL einen hohen Aufwand zur Sammlung von Daten. Bei der Installation werden diverse Benchmarks ausgeführt, sowie z. B. die Anzahl der Recheneinheiten, die Größe des Hauptspeichers, die Anzahl und Größe der Caches usw. in einer SQL-Datenbank abgespeichert [18]. Unter anderem wird auf Basis dieser Daten und Containergrößen berechnet, welche Last durch einen Algorithmus in etwa zu erwarten ist und anhand dieser Daten entschieden, ob die sequentielle STL- oder welche parallele STAPL-Variante aufgerufen wird [18]. Selbst zur Laufzeit werden weitere Tests ausgeführt, die basierend auf einer Analyse der Datenbank ausgewählt werden, um die Auswahl des richtigen Algorithmus zu verbessern [49]. In der Tat zeigen Ergebnisse aus [49], dass auf unterschiedlichen Architekturen verschiedene semantisch äquivalente Algorithmen die schnellsten sein können. In dieser Hinsicht gibt es eine Legitimation für die adaptive Selektion von Algorithmen, auch unter Inkaufnahme eines gewissen Mehraufwandes, der immer noch kleiner sein kann, als die Differenz zu einem auf der jeweiligen Architektur langsamer ausgeführten Standardalgorithmus. Für typische Sortieralgorithmen zeigt [49], dass die Auswahl des korrekten Algorithmus in den meisten Fällen gelingt, während bei komplexeren Aufgaben mit sich syntaktisch deutlicher unterscheidenden Algorithmen, wie z.B. der Matrixmultiplikation, wesentlich häufiger Fehlauswahlen

auftreten. Des Weiteren basieren alle Ergebnisse aus [49] auf Clustern. Aus dem Blickwinkel dieser Arbeit wäre es interessanter zu sehen, wie stark die Verluste auf Multicore- oder einzelnen SMP-Systemen variieren. Aus diesen Gründen wird auf eine Darstellung der Ergebnisse an dieser Stelle verzichtet. Eine Einschränkung bezüglich der Adaptivität ist, dass die Selektion des richtigen Algorithmus erst nach sehr vielen „Trainingsläufen“ zuverlässig funktioniert. So wurden für Sortierverfahren auf einem Hochleistungscomputer mit Intel Itanium Prozessoren, sowie zwei Clustern, mindestens 100 Trainingsdurchläufe benötigt, um zu akzeptablen Ergebnissen zu gelangen. Für die Matrixmultiplikation konnten diese erst nach 200 Durchläufen erreicht werden [49]. Eine derart hohe Anzahl ist für die Entwicklung von Programmen im kleinen Rahmen, die auf verschiedenen Architekturen Nutzen aus mehreren Prozessoren ziehen sollen, vollkommen unbefriedigend.

2.1.3 Fazit

Auf Basis der in der Literatur vorliegenden Ergebnisse lässt sich feststellen, dass STAPL insbesondere für sehr große SMP-Systeme und Cluster ein interessantes Konzept darstellt. Hier können die Stärken ARMIs ausgenutzt werden, indem z.B. Shared Memory Synchronisation innerhalb eines Cluster-Knotens (Node) und Message Passing zwischen Nodes angewandt wird. Auch die hohe Anzahl notwendiger Testiterationen bis zur zufriedenstellenden Funktionsfähigkeit der Adaptivität stellt hier ein geringes Problem dar.

Auf Grund der beschriebenen Ausrichtung auf Distributed Memory Systeme, der hohen Abstraktion der Kommunikationsebene, der SPMD-Orientierung, der Verwaltung einer Datenbank zur Laufzeit und der weiteren beschriebenen Einschränkungen und Latenzen lässt sich jedoch vermuten, dass STAPL für die Domäne der Shared Memory Systeme und die im Rahmen dieser Arbeit fokussierten kleinen und mittleren Eingaben nicht optimal geeignet ist. In [48] wird darüber hinaus konstatiert, dass für komplexere und mehr Synchronisation erfordernde Aufgaben, wie etwa die LU-Dekomposition einer Matrix, die Notwendigkeit der Aggregation von asynchronen Funktionsaufrufen noch deutlicher wird.

Das Ziel einer zur STL identischen Schnittstelle wurde nur teilweise erreicht. Des Weiteren werden durch den jeweiligen Parallel Region Manager für jeden pAlgorithm-Aufruf neue Threads generiert, wodurch zusätzlicher und durch Verwendung eines Threadpools vermeidbarer Overhead entsteht. Auch die in Abschnitt 3.1.4 näher erläuterte Thread Affinity wird nicht realisiert.

In der zitierten Literatur finden sich viele Hinweise darauf, dass die NPTL die effizienteste Basis für die ARMI-Implementierung auf Shared Memory Systemen darstellte. Nicht zuletzt gibt die verfügbare Literatur keinen vollständigen Überblick über die Anzahl und Leistung der bisher realisierten parallelen STL-Funktionen. Es bleibt zu hoffen, dass STAPL in absehbarer Zeit unter einer freien Lizenz veröffentlicht wird und dementsprechend sinnvolle Leistungsvergleiche mit anderen Bibliotheken für STL-Algorithmen mög-

lich werden. Interessant wird dann auch die Leistungsfähigkeit einer eigens entwickelten Speicherverwaltung sein [40].

2.2 Multi-Processing Template Library

Die Multi-Processing Template Library (MPTL) wurde 2006 im Rahmen einer Masterarbeit an der Universität Genf entwickelt. Sie bietet eine sehr nutzerfreundliche Schnittstelle, da STL-Algorithmen einfach durch Aufrufe von MPTL-Algorithmen ersetzt und eigene parallele Algorithmen mit relativ wenig Aufwand und Multithreading-Wissen erzeugt werden können.

2.2.1 Aufbau und API

Die MPTL verzichtet auf eine eigene Kommunikationsschicht und setzt auf der Native POSIX Threads Library auf, ist also eine reine Shared Memory Bibliothek. Die nativen Funktionen der NPTL kommen direkt zum Einsatz und werden nicht durch eigene Funktionen oder Klassen gekapselt. Das zentrale Konzept der MPTL stellt ein Klassenrahmen dar, der als *„Classe de Spécifications Parallèles pour un Algorithme“* (CSPA) bezeichnet wird. Mit Hilfe einer CSPA kann ein durch die MPTL parallel auszuführender Algorithmus nahezu vollständig beschrieben werden. Am Beispiel für eine einfache fiktive Funktion `addValue()`, die einen konstanten Wert auf alle Elemente eines Containers addiert, werden im Folgenden die wichtigsten Bausteine einer CSPA beschrieben.

Wie die Klassendeklaration in Zeile 1 des Quelltextauszugs 2.3 zeigt, handelt es sich dabei um eine generische Klasse mit Templates für den Iteratortyp `Iterator` und den Datentyp `T`. Die zugehörigen privaten Variablen definieren den Beginn und das Ende des zu partitionierenden Intervalls und in diesem Fall den konstanten Wert für die Addition. Typisch für die MPTL ist die Verwendung von `typedef` zur Übersetzung von Eigenschaften auf innerhalb der Bibliothek standardisierte Namen. So wird in Zeile 8 der Iteratortyp auf den Namen `iterator1`, und in Zeile 9 die Anzahl der verwendeten Iteratoren auf den Namen `nIterators` übersetzt. Andere Instanzen der Bibliothek können diese Informationen durch dieses Konzept über eine einheitliche Schnittstelle, nämlich `CSPA::iterator1` und `CSPA::nIterators` abrufen. Die Anzahl der Iteratoren bestimmt gleichzeitig die Anzahl der für die Funktion verwendeten Intervalle, bzw. Eingabesequenzen. Ein Intervall wird durch zwei Iteratoren, zwei Intervalle durch drei Iteratoren und drei Intervalle durch vier Iteratoren definiert. Daher existiert zum Zweck dieser Übersetzung eine Struktur (`struct`) mit den Möglichkeiten `TwoIterators`, `ThreeIterators` und `FourIterators`. Der Konstruktor hat lediglich die Aufgabe die beschriebenen privaten Variablen zu initialisieren. Für jeden Iterator wird eine Methode bereitgestellt, um ihn nach Außen für andere Instanzen der Bibliothek zugreifbar zu machen, z.B. für die Partitionierung eines Intervalls.

```
template <typename Iterator, typename T> class MPTLAddValue
{
  private:
     Iterator first, last; // Intervall
     const T value; // Konstanter Wert fuer die Addition

  public:
     typedef Iterator iterator1;
     typedef TwoIterators nIterators;

     // Konstruktor mit Initialisierung
     MPTLAddValue(Iterator first_, Iterator last_, const T value_) :
         first(first_), last(last_), value(value_) { }

     // Methoden für den Zugriff auf die Intervalle von Außen
     Iterator getFirst1() { return first; }
     Iterator getLast1() { return last; }

     // Algorithmus auf dem (Teil-)Intervall [first_, last_)
     void applyAlgorithm(Iterator first_, Iterator last_)
     {
        for ( ; first_ != last_; ++first_)
           *first_ += value;
     }
};
```

Quelltextauszug 2.3: Beispiel einer CSPA

Abschließend erfolgt die Definition einer Methode mit dem Namen `applyAlgorithm()`, die den sequentiellen Algorithmus für ein (Teil-)Intervall enthält.

Für eine Funktion ohne Rückgabewert ist dies schon alles Erforderliche, um einen parallelen MPTL-Algorithmus zu definieren. Wird ein Rückgabewert benötigt, sind einige zusätzliche Schritte notwendig, u.a. ein weiteres `typedef`, welches den Typ des Rückgabewertes auf den durch die MPTL definierten Typ `resultType` übersetzt, z.B. `typedef int resultType`. Darüber hinaus wird eine Klasse benötigt, die spezifiziert, wie der Rückgabewert aus den ermittelten Teilergebnissen berechnet wird. Auch für diese Klasse wird ein `typedef` auf den standardisierten Namen `reduction` verwendet. Quelltextauszug 2.4 als Beispiel für eine solche Reduktionsklasse zeigt, dass dieser Schritt in der MPTL sequentiell geschieht und Threads nicht etwa parallel Teilergebnisse unter Verwendung von gegenseitigem Ausschluss zum Gesamtergebnis reduzieren.

```
1  class MPTLAccumulateReduction
2  {
3    public:
4      template <typename T, typename CSPA> static T
5          reduce(const std::vector<T>& results, CSPA algoSpec)
6      {
7         T full_result(0);
8         typename std::vector<T>::const_iterator it = results.begin();
9         for ( ; it != results.end(); ++it)
10           full_result += *it;
11
12        return full_result;
13     }
14 };
```

Quelltextauszug 2.4: MPTL Reduktionsklasse

Nachdem ein paralleler Algorithmus auf diese Weise definiert wurde, stellt sich die Frage, wie die internen Mechanismen der MPTL zusammenspielen, um aus den beschriebenen Komponenten eine parallele Ausführung zu ermöglichen. Der folgende Abschnitt 2.2.2 geht auf diese Fragestellung ein.

2.2.2 Realisierung paralleler Ausführung und Ergebnisse

Um einen Algorithmus parallel ausführen zu können, stellt die MPTL zwei Methoden namens `execute()` und `dynamic_execute()` zur Verfügung, die ähnlich wie der Parallel Region Manager in STAPL das vollständige Threadmanagement kapseln. Die erste Methode hat eine statische Partitionierung des durch die Iteratoren definierten Datenraumes zur Folge. Die Anzahl der Elemente wird durch die zuvor per `mptl::setNumThreads()` gewählte Anzahl der Threads dividiert. Ein möglicher Rest wird der Reihe nach auf alle Threads aufgeteilt [19]. Für einfache, linear auf dem Speicher arbeitende STL-Algorithmen ist dies die schnellste Vorgehensweise, weil sie insbesondere wenig Overhead erzeugt [19]. In einigen Fällen ist aber damit zu rechnen, dass gewisse Threads mehr Zeit benötigen als andere, z.B. wenn einzelne logische Ausführungseinheiten eine stärkere Auslastung aufweisen oder die Laufzeit vom Ort oder der Position der zu verarbeitenden Daten abhängt. In diesem Fall kann per `dynamic_execute()` die Lastverteilung (*Load Balancing*) verbessert werden, indem die Größe eines Teilstücks (chunk) selbst definiert wird. Der Hauptprozess erzeugt in diesem Fall die vorgegebene Anzahl von Threads und übergibt ihnen nach und nach die Iteratoren für die definierten Teilstücke. Die Wahl einer kleinen Größe kann auf diese Weise zu einer effizienteren Verteilung der Teilintervalle führen. Eine zu klein gewählte Größe zieht aber erhöhten Overhead durch die häufiger notwendigen Austauschoperatio-

nen nach sich. Es handelt sich hierbei also um ein sehr einfaches Load Balancing, welches dem Benutzer der Bibliothek überlassen wird und in der Praxis einige Tests erfordert, um eine möglichst gute Chunk-Größe zu finden.

Beispiele für den Aufruf parallelisierter STL-Algorithmen für einen STL-Container namens `coll` können z. B. folgendermaßen formuliert werden.

```
// Statisch, keine Reduktion
mptl::execute(mptl::count(coll.begin(), coll.end(), 10));

// Statisch, mit Reduktion
int result(0);
mptl::execute(mptl::accumulate(coll.begin(), coll.end(), &result));

// Dynamisch, keine Reduktion
mptl::dynamic_execute(mptl::count(coll.begin(), coll.end(), 10, 5));
```

Für die genaue Untersuchung der Realisierung der parallelen Ausführung wird die Funktion `execute()` (Quelltextauszug 2.5) herangezogen. Die Schleife erzeugt für die konfigu-

```
1 for (unsigned int i = 0; i < getNumThreads(); ++i)
2 {
3    ThreadData<CSPA>* pData = new ThreadData<CSPA>(i, algoSpec);
4    pthread_create(&threads[i], NULL, functionWrapper<CSPA>,
5       (void *) pData);
6 }
```

Quelltextauszug 2.5: MPTL `execute()`

rierte Anzahl Threads zunächst jeweils ein Objekt vom Typ `ThreadData`. Dieses speichert die notwendigen Informationen für den jeweiligen Thread, nämlich seine ID `i` und die auszuführende CSPA (`algoSpec`). Durch `pthread_create()` wird ein neuer POSIX-Thread erzeugt und im Array `threads` abgespeichert. Wie ein normales C++-Programm, das eine `main()`-Hauptfunktion besitzen muss, erhält der Thread als solche die Funktion `functionWrapper()`. Die NPTL erlaubt lediglich einen `void`-Zeiger als Parameter dieser Funktion, weshalb der Zeiger auf das soeben erzeugte `ThreadData`-Objekt mit Hilfe eines Casts übergeben wird. Es existieren mehrere Varianten der Funktion `functionWrapper()`, abhängig von der Art des gewählten Load Balancings sowie einer möglichen Reduktion. Ihre einzige Aufgabe ist es, eine Instanz der Funktionsobjekte `ThreadObject` oder `ThreadObjectResult`, bzw. `ThreadObjectDynamic` oder `ThreadObjectDynamicResult` zu erzeugen und deren Funktionsoperator aufzurufen. Auch dieser Funktionsoperator hat lediglich den Zweck eines Aufrufs der Funktion `selectNumIterators()`, die ebenfalls abhängig von der Art des Load Balancings unterschiedlich implementiert ist, und der Übergabe der Anzahl der ver-

wendeten Iteratoren mittels `CSPA::nIterators()`. Quelltextauszug 2.6 zeigt ein Beispiel für den Fall zweier Iteratoren.

```
1 void selectNumIterators(TwoIterators)
2 {
3     typedef typename CSPA::iterator1 iterator;
4     iterator first = data.algoSpec.getFirst1(); // Intervall-Beginn
5     iterator last = data.algoSpec.getLast1(); // Intervall-Ende
6
7     int inf, sup;
8     iterator pos1, pos2;
9
10    calcInterval(data.tId, mptlNumThreads,
11        std::distance(first, last), inf, sup);
12
13    pos1 = pos2 = first;
14    std::advance(pos1, inf);
15    std::advance(pos2, sup);
16
17    exec(pos1, pos2); // ruft applyAlgorithm() auf
18 }
```

Quelltextauszug 2.6: Die Funktion `selectNumIterators()`

Durch Zugriff auf die Iteratoren der CSPA wird die Länge des Intervalls mittels `std::distance()` bestimmt und mit Hilfe der Thread-ID und der Anzahl der Threads die entsprechende Partition durch `calc_interval` berechnet. Die Partitionierung erfolgt also nicht a priori sequentiell durch den Hauptprozess, sondern ist Teil der parallelen Ausführung. Dieses Vorgehen ist effizient, auch wenn der Prozess der Partitionierung, bestehend aus einigen arithmetischen Operationen und dem Verschieben von Iteratoren, nach eigenen Messungen keinen signifikanten Overhead erzeugt. Nach entsprechender Anpassung der lokalen Iteratoren `pos1` und `pos2` wird mittels `exec` die Ausführung von `applyAlgorithm()` auf dem entsprechenden Intervall veranlasst. Für den Fall der dynamischen Lastverteilung wird in der Funktion `selectNumIterators()` ein Task-Manager verwendet, bei dem sich die Threads nach und nach entsprechende Teilintervalle zur Ausführung abholen. Im Gegensatz zur statischen Zuweisung von Tasks an Threads wird auf diese Weise Rücksicht auf die Auslastung von Kernen genommen. Ein Thread, der auf einem (z. B. durch andere Prozesse) stark ausgelasteten Kern läuft, wird also erst gar nicht dazu kommen, sich viele Tasks abzuholen und andere, weniger ausgelastete Kerne, werden diese Arbeit übernehmen.

Die aus der Masterarbeit übernommene und übersetzte Abbildung 2.3 fasst den typischen Ablauf eines MPTL-Programms nochmals schematisch zusammen, wobei Kompi-

lierzeit (Testen des Iteratortyps) und Laufzeit der Funktion `execute()` etwas verwirrend vermischt werden.

Nur etwa die Hälfte der STL-Algorithmen wurde von Didier Baertschiger parallelisiert. Ein Grund dafür ist, dass die MPTL-Threads nicht kommunizieren können, da weder eine Kommunikationsschicht noch andere geeignete Synchronisationsmechanismen implementiert wurden [19]. Dies bedeutet, dass beispielsweise Algorithmen, die Elemente austauschen müssen, nicht mit Hilfe der MPTL parallelisierbar sind. Es existiert daher z. B. nur eine simple parallele Variante von Quicksort, die bereits aus dem beschriebenen Ausführungsschema mit `execute()` herausfällt, und keine Implementierung eines anderen Sortieralgorithmus, obwohl in der Literatur vielfach in situ arbeitende und dennoch effizient parallelisierbare Verfahren beschrieben werden, vgl. z. B. [50]. Darüber hinaus werden nur Algorithmen unterstützt, die keine Input- und Output-Iteratoren verwenden, wie auch in Abbildung 2.3 angedeutet wird. Dies ist sinnvoll, da wie in Abschnitt 1.2 beschrieben, Input- und Output-Iteratoren kein mehrfaches Durchlaufen des jeweiligen Containers ermöglichen und das Verschieben solcher Iteratoren mittels der Funktion `std::advance()`, etwa zur Erzeugung von Partitionen, lineare Zeit benötigt. Die Parallelisierung ist also in ihrem Fall nicht erfolgversprechend. Es entsteht hierbei jedoch der Eindruck, als ob Didier Baertschiger in [19] mit diesem Argument auch begründen möchte, wieso Algorithmen wie etwa `inner_product()`, die lediglich die Anforderungen eines `InputIterator` an ihre Eingaben stellen, nicht im Rahmen der MPTL entwickelt wurden. Es handelt sich dabei jedoch nur um eine Mindestanforderung und somit wäre die Implementierung von Funktionen dieser Art insbesondere für Iteratoren vom Typ `RandomAccessIterator` durchaus möglich gewesen. Durch explizite Spezialisierung mit Hilfe von Templates und der Eigenschaft `iterator_category()` der STL [6, 32] ist eine Fallunterscheidung zur Kompilierzeit darüber hinaus einfach zu realisieren. Auf diese Weise verliert die MPTL einen wichtigen Kompatibilitätsaspekt zur STL, da viele Algorithmen lediglich `InputIterator` als Eingabe erfordern.

Ähnlich wie STAPL erlaubt die Bibliothek des Weiteren keinen schreibenden Zugriff auf statische Variablen, da sie keine internen Mechanismen bereitstellt, um gegenseitigen Ausschluss zu realisieren. Auf diese Weise, und durch die Tatsache, dass jeder Thread zunächst eine eigene Ergebnisvariable erhält, die dann in einer nachfolgenden sequentiellen `for`-Schleife zum Gesamtergebnis reduziert werden, wird sehr simple Thread-Sicherheit erreicht, allerdings auch Performance verschenkt. Des Weiteren werden für jeden Funktionsaufruf durch `execute()` oder `dynamic_execute()` neue Threads und damit auch jedes Mal neuer Overhead erzeugt. Auch Thread Affinity wird durch die MPTL nicht realisiert, sowie in [19] behauptet, dass der NPTL-Standard dazu keine Möglichkeit böte und es folglich unmöglich sei, dies in Verbindung mit POSIX Threads zu implementieren. Dies ist jedoch nicht korrekt, da sowohl die NPTL selbst Funktionen für diesen Zweck zur Verfügung stellt, als auch die Systemfunktionen gängiger Unix- und Windows-Betriebssysteme

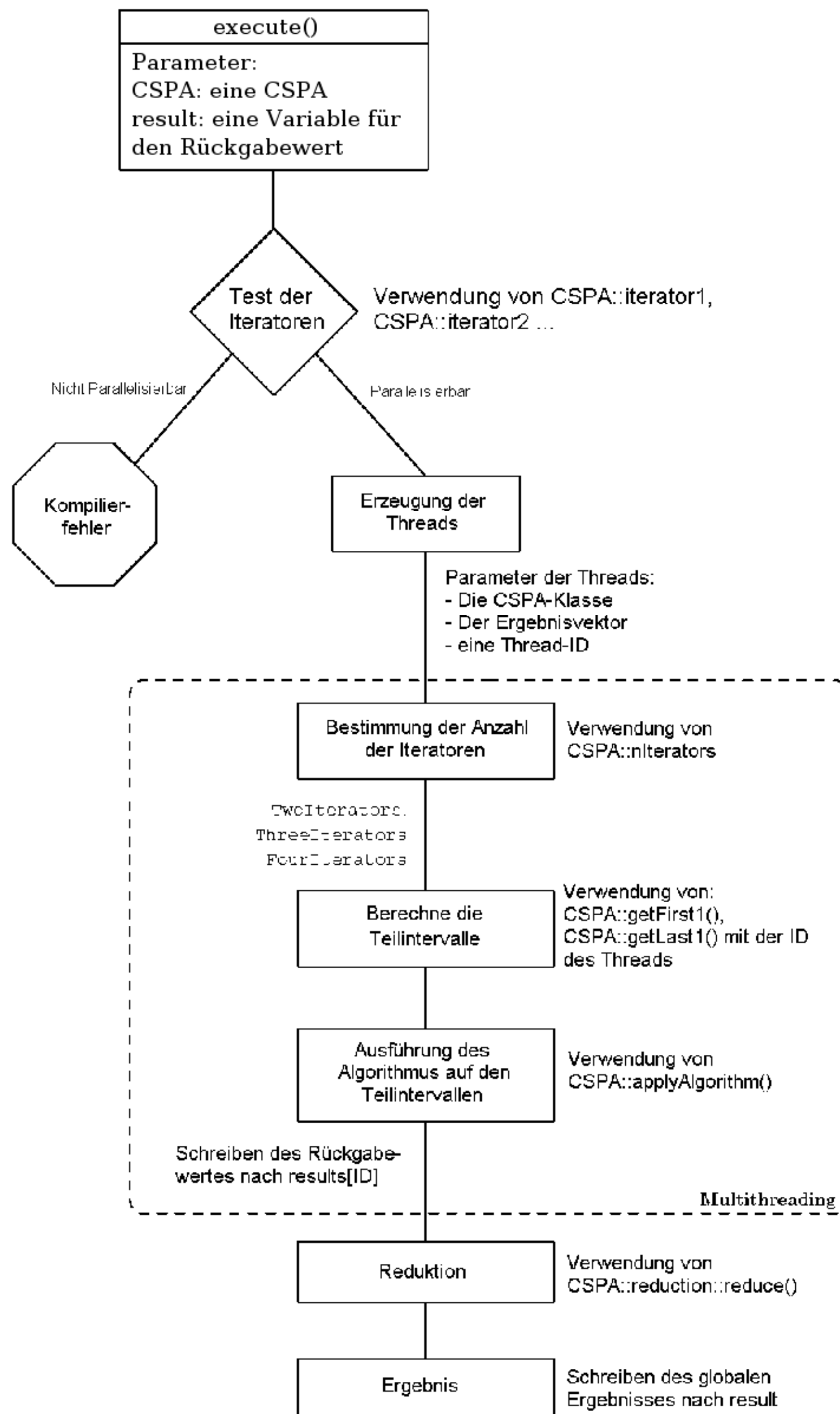

Abbildung 2.3: Ablauf eines MPTL-Programms (übersetzt aus [19])

[24, 35]. Die Nutzung dieser Schnittstellen wird darüber hinaus auch von Intel empfohlen [11].

In [19] wurden Benchmarkergebnisse für drei beispielhafte STL-Algorithmen `generate()`, `transform()` und `replace_copy_if()` veröffentlicht. Für die Benchmarks wurden folgende Systeme eingesetzt:

- Intel Xeon 2,2 GHz Quadcore mit Hyper-Threading und 2 GB Hauptspeicher
- Sun Enterprise 10000: 24 UltraSPARC Prozessoren mit 400 MHz und 20 GB Hauptspeicher
- Intel Pentium 4 3 GHz mit Hyper-Threading und 1 GB Hauptspeicher

Für `generate()` wurde eine Funktion übergeben, die immer den *konstanten* Wert des Terms $\frac{123.3}{23} \cdot \sin(32.2)$ in doppelter Genauigkeit zurückliefert. Der G++-Compiler optimiert die überflüssigen Aufrufe der Sinusfunktion in eigenen Tests auch bei höchster Optimierungsstufe nicht heraus, theoretisch könnte ein Compiler den n-maligen Aufruf aber durch einen einmaligen ersetzen. Die resultierende Funktion wäre dann zu `fill()` äquivalent und damit wesentlich trivialer als `generate()`. Leider macht der Autor keine Angaben über den verwendeten Compiler, sowie die verwendeten Parameter. Der Funktion `transform()` wird als Aufgabe die Division des jeweiligen Elementes durch die Konstante 23 übergeben. Für

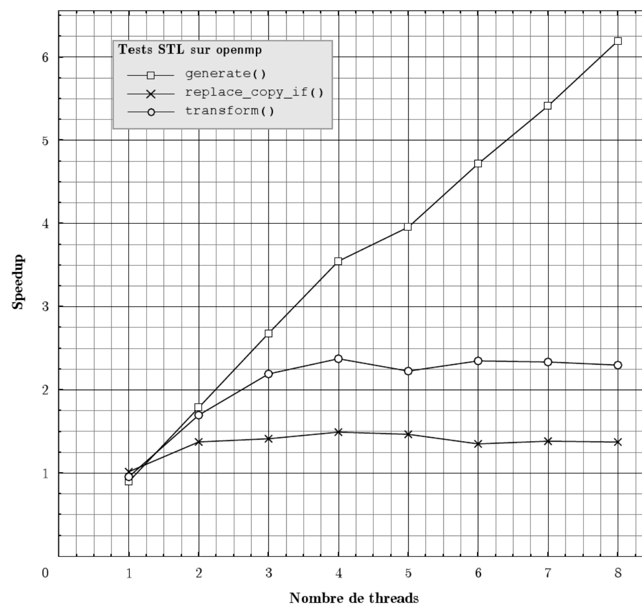

Abbildung 2.4: MPTL Speedup für `generate()`, `transform()` und `replace_copy_if()` (Quadcore Xeon) (Quelle: [19])

das Quadcore Xeon-System und Linux (Kernel 2.6) unterschieden sich die Ergebnisse für

die drei Funktionen sehr stark von einander, wie Abbildung 2.4 zeigt. Während die Laufzeit für `generate()` annähernd linear skaliert, konvergiert der Speedup bei den anderen beiden Funktionen nach einem flachen Anstieg bei Verwendung von bis zu vier Threads auf einem relativ niedrigen Niveau. Offenbar kann hier aus dem Hyper-Threading kein Nutzen mehr gezogen werden. Didier Baertschiger vermutet den elementweisen Schreibzugriff von `transform()` als Grund für die schwächeren Ergebnisse. Anders als von ihm beschrieben, führt `generate()` aber ebenfalls einen Schreibzugriff pro Element aus und `transform()` verursacht lediglich einen weiteren Lesezugriff. Während dies bei `generate()` wie beschrieben fraglich bleibt, muss darüber hinaus hier definitiv eine arithmetische Operation pro Element ausgeführt werden. Die MPTL-Implementierung des durch bedingte Sprünge dominierten `replace_copy_if()` scheint auf diesem System nahezu keinen Nutzen aus Multithreading ziehen zu können. Jede an dieser Stelle geäußerte Begründung bliebe jedoch rein spekulativ, da in [19] insbesondere Angaben zu den Eingabegrößen der Container fehlen. Auch die Tatsache, dass nur Ergebnisse für eine einzige Eingabegröße veröffentlicht wurden, führen zu einer geringen Aussagekraft.

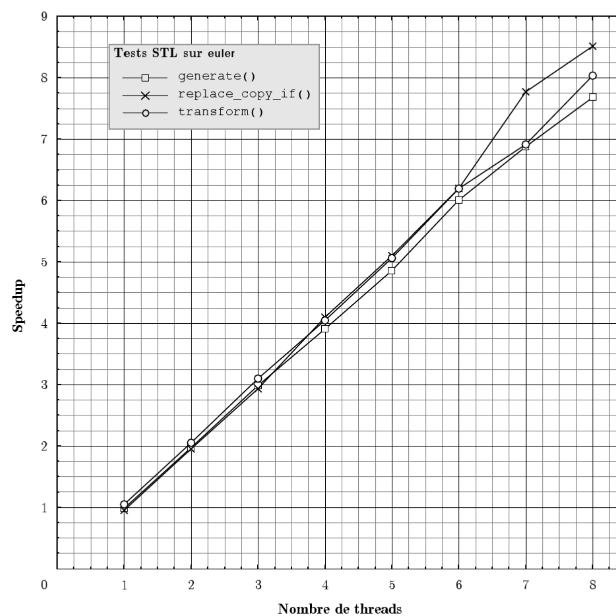

Abbildung 2.5: MPTL Speedup für `generate()`, `transform()` und `replace_copy_if()` (Sun Enterprise) (Quelle: [19])

Auf dem Sun SMP-System und SunOS 5.9 zeigt sich ein vollkommen anderes Bild. Es ist jedoch fraglich wie verlässlich die präsentierten Ergebnisse sind, da die Laufzeit in Abbildung 2.5 teilweise superlinear skaliert. Dies ist bei hinreichend genauer Messung nur in sehr seltenen Fällen möglich, z. B. wenn durch die Partitionierung aus einer Out-of-Cache Eingabegröße eine (partielle) In-Cache Größe wird oder die Partitionen threadlokal alloziert wurden. Letzteres wäre allerdings nicht realistisch im Sinne der Verwendung der

Bibliothek. Darüber hinaus wird das System nicht voll ausgereizt, da nur acht Threads bei 24 Prozessoren eingesetzt werden.

	generate()		transform()		replace_copy_if()	
	Zeit [s]	Speedup	Zeit [s]	Speedup	Zeit [s]	Speedup
Sequentiell	5.39	–	2.51	–	4.91	–
1 Thread	5.39	1	2.51	1	4.92	1
2 Threads	4.27	1.26	1.42	1.76	5.07	0.97

Abbildung 2.6: MPTL Speedup für `generate()`, `transform()` und `replace_copy_if()` (Pentium 4) (Quelle: [19])

Zuletzt noch ein Ergebnis (Abbildung 2.6) für das Einprozessorsystem mit Hyper-Threading unter Linux. Es zeigt sich, dass der Intel Prozessor hier durchaus Nutzen aus der Parallelisierung mit zwei Threads ziehen kann. Lediglich für `replace_copy_if()` wurde eine verlängerte Laufzeit gemessen, vermutlich auf Grund der hohen Anzahl bedingter Sprünge, denn bei Hyper-Threading werden funktionale Einheiten (wie etwa auch die arithmetisch-logische Einheit zur Auswertung von boolschen Bedingungen) nicht repliziert.

2.2.3 Fazit

Didier Baertschiger sieht in [19] eine gewisse Nähe zur OpenMP API, da die Ausführung der Algorithmen in der MPTL dem *fork-and-join*-Paradigma

```
# pragma START_PARALLEL_REGION
// Processing..
# pragma END_PARALLEL_REGION
```

folgt. Dennoch verwendet er die NPTL für die Realisierung seiner Bibliothek, ohne die damit verbundenen größeren Möglichkeiten zur Einflussnahme auf die parallele Ausführung und der Kommunikation zwischen Threads zu nutzen. Weitere Benchmark-Ergebnisse in [19] (z. B. für die Berechnung von Julia-Mengen) zeigen, dass für die dokumentierten Fälle die Leistung der MPTL mit der einer äquivalenten OpenMP-Implementierung mithalten kann, insbesondere wenn dynamisches Load Balancing verwendet wird.

Insgesamt bietet die MPTL eine sehr einheitliche und leicht zu erlernende Schnittstelle zur Entwicklung einfacher Parallelität ausnutzender Programme, die die Details der Realisierung gut verdeckt. Der Autor macht Gebrauch von Konzepten, die auch in der STL starken Einsatz finden, wie etwa generischer Programmierung oder dem Einsatz von Funktionsobjekten. Auf der anderen Seite haben einige Missverständnisse bezüglich der Iteratoreneigenschaften dazu geführt, dass viele potenziell parallelisierbare Algorithmen nicht implementiert wurden.

Die Performance, beurteilt nach den veröffentlichten Ergebnissen aus [19], variiert sehr stark bei den unterschiedlichen Funktionen und Architekturen. Ein Grund dafür ist mögli-

cherweise, dass es vermieden wurde, eine komplexere Basisstruktur zu entwickeln, um alle Features des Multithreadings auszunutzen. Insbesondere der Verzicht auf die Manipulation gemeinsamer (Synchonisations-)Variablen durch Methoden gegenseitigen Ausschlusses ist hier zu nennen. Bei nur auf komplexe Weise parallelisierbaren Algorithmen offenbaren sich die Schwächen und Einschränkungen der MPTL, welche auch durch das Fehlen eines effizienten Sortieralgorithmus sichtbar werden. Des Weiteren wurde keine Wiederverwendung von bereits erzeugten Threads (z. B. durch einen Threadpool) oder Thread Affinity implementiert.

Leider hat seit dem Jahr 2006 keine Weiterentwicklung der MPTL stattgefunden. Da die hier mit der NPTL erzeugten Ergebnisse auch in nur rudimentär handoptimierten Algorithmen nicht den Vergleich zu OpenMP-Implementationen scheuen müssten, wird die Hoffnung gestützt, dass unter Ausnutzung aller Features der expliziten Parallelisierung die Performance noch gesteigert werden kann. Durch ihre freie Verfügbarkeit kann die MPTL im Rahmen dieser Arbeit für Vergleiche herangezogen werden. Durch eine verbesserte Dokumentation der Testbedingungen können auf diese Weise neue Schlussfolgerungen bezüglich ihrer Leistung auf ausgewählten Systemen erfolgen.

2.3 Multi-Core Standard Template Library

Die Multi-Core Standard Template Library (MCSTL) ist der aktuellste und ambitionierteste Vertreter der hier behandelten Bibliotheken und wurde bereits in die Version 4.3 der *GNU Compiler Collection* (GCC) aufgenommen. Sie wird seitdem auch als *libstdc++ parallel mode* (siehe auch Abbildung 2.7) bezeichnet [44]. Zum Zeitpunkt der im Folgenden dokumentierten Untersuchungen und der in Kapitel 4 vorgestellten Benchmarks war die GCC-interne Version noch nicht hinreichend dokumentiert und die Konfiguration einiger für Vergleiche essentieller Parameter noch nicht funktionsfähig, weshalb die letzte MCSTL-Version (0.8.0-beta) zum Einsatz kam. Trotz des Beta-Status implementiert die MCSTL im Gegensatz zur MPTL bereits zu diesem Zeitpunkt auch komplexere STL-Funktionen. Lediglich einige wenige Funktionen, für die eine Parallelisierung wenig erfolgversprechend ist, wurden von vornherein vom Entwicklungsprozess ausgeschlossen. Einen vollständigen Überblick über die bereits implementierten Funktionen bieten die Webseite des Projekts [3] und der Hauptentwickler der MCSTL, Johannes Singler in [45]. Die in der Zwischenzeit erfolgte Online-Dokumentation des parallel mode ist in [2] zu finden.

2.3.1 Aufbau und API

Die MCSTL konzentriert sich ähnlich wie die MPTL auf Shared Memory Systeme und verzichtet auf Message Passing sowie Optimierungen für Cluster. Bezüglich der verwendeten Multithreading-API haben sich die Entwickler auf eine Einzellösung festgelegt. Die Wahl fiel auf OpenMP, obwohl Vorgänger und Konkurrenten z. T. Geschwindigkeitsvorteile bei

der Verwendung der NPTL dokumentiert haben. Auch die am selben Lehrstuhl der Universität Karlsruhe entwickelte *Standard Template Library for XXL Data Sets* (STXXL) [23], die auf die Beschleunigung von Externspeicher-Algorithmen durch parallele Verwendung mehrerer Festplatten spezialisiert ist, verwendet POSIX Threads für ihre asynchrone Kommunikationsschicht [7]. Als Beispielanwendungen der STXXL dienen insbesondere graphen- bzw. netzwerkbasierte Algorithmen, die auf sehr großen Datenmengen operieren und mehrere Terabytes zwischen Festplatten, Hauptspeicher und Prozessoren transferieren müssen. Da bisher keinerlei Mechanismen zur Ausnutzung von Parallelität auf Instruktionsebene bestehen, wird die STXXL nun stellenweise mit der MCSTL kombiniert.

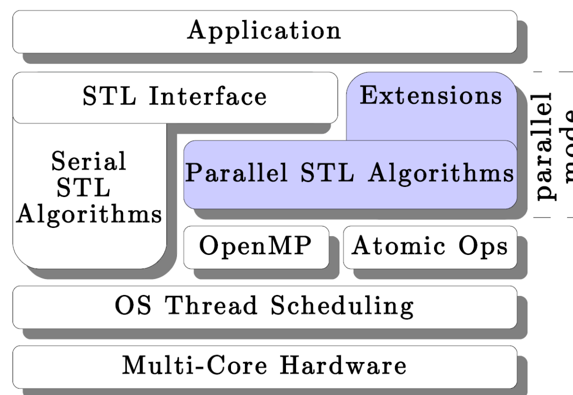

Abbildung 2.7: Eingliederung des libstdc++ parallel mode in die STL (Quelle: [44])

In [44] wird der Vorzug von OpenMP gegenüber der NPTL begründet. Einerseits liefert OpenMP eine „*fork-and-join*“-Parallelisierung, welche allerdings mit der NPTL genauso möglich ist, wie bereits die MPTL zeigte. Des Weiteren wird angeführt, dass man auf diese Weise die Aufgaben der niedrigen Abstraktionsebenen an OpenMP auslagern könne. Dies birgt einerseits die Gefahr auch Optimierungspotential auszugliedern, andererseits ist eine compiler- und damit z. T. auch architekturabhängige Optimierung (z. B. Intel C++ Compiler) OpenMPs möglich. Ein anderer Grund ist die höhere Plattform-Unabhängigkeit von OpenMP, das auch von Compilern unter Windows-Betriebssystemen unterstützt wird, während die NPTL auf Unix-Derivate beschränkt ist, obwohl es auch hier bereits Ansätze zur Portierung auf Windows gibt (vgl. [5]). In diesem Zusammenhang ist auch interessant, was geschieht, wenn ein System den entsprechenden Code nicht unterstützt. Im Fall von OpenMP handelt es sich lediglich um Compiler-Direktiven, die notfalls ignoriert werden können, wodurch eine sequentielle Ausführung dennoch ermöglicht wird. Auch dies ist, wenn auch mit erhöhtem Aufwand, bei Verwendung der NPTL ebenfalls durch Einsatz von Compiler-Flags zu gewährleisten.

Die MCSTL wurde so konzipiert, dass ihre Schnittstelle nach außen identisch zur STL ist. Denn das erklärte Ziel der Entwickler ist es, allein durch das Einbinden der MCSTL und anschließende Rekompilation eines bisher auf der STL basierenden Programms, ei-

ne Beschleunigung durch Ausnutzung von Parallelität zu erreichen [45]. Einzige Ausnahme bilden die nachfolgend beschriebenen Parameter `sequential_tag()` und `parallel_tag()`, die am Ende einer jeden Funktionsparameterliste hinzugefügt werden können. Die Möglichkeiten des Benutzers, auf die Art und Weise der Parallelisierung Einfluss zu nehmen, sind auf wenige Parameter beschränkt.

- `mcstl::SETTINGS::num_threads`: Erlaubt das Setzen der Anzahl der zu verwendenden Threads.
- `mcstl::sequential_tag()`: Erzwingt sequentielle Ausführung.
- `mcstl::parallel_tag()`: Weist die parallele Ausführung an.
- `mcstl::force_parallel()`: Erzwingt die parallele Ausführung, auch wenn dies auf Grund der jeweiligen Eingabegröße möglicherweise nicht sinnvoll ist.
- `mcstl::SETTINGS::[algorithm_name]_minimal_n`: Erlaubt die Definition einer minimalen Eingabegröße, ab der die parallele Ausführung verwendet werden soll.

Ähnlich wie STAPL mit `p_for_all()` fasst die MCSTL Algorithmen zusammen, die linear auf dem Speicher arbeiten, d. h. gewisse Operationen einmalig auf alle Elemente eines oder mehrerer Container anwenden und somit einer `for each` Routine ähneln. Für sie bieten sich die gleichen bibliotheksinternen Partitionierungs- und Dispatching-Strategien, auf die im folgenden Abschnitt genauer eingegangen und von denen die jeweils erfolgversprechendste ausgewählt wird. Für andere, komplexere Algorithmen wie Sortierverfahren oder beispielsweise `partial_sum()` gibt es hingegen eigene Strategien, da mehr Datenabhängigkeiten bestehen und sich das Auffinden parallel verarbeitbarer Abschnitte nicht auf das soeben beschriebene Schema reduzieren lässt.

2.3.2 Realisierung paralleler Ausführung und Ergebnisse

Zentrales Merkmal der parallelen Ausführung im Quellcode sind diverse Compiler-Direktiven, so genannte „pragmas". Ein Beispiel dafür ist etwa:

```
#pragma omp parallel shared(busy) num_threads(num_threads)
```

Hier wird ein Programmabschnitt definiert, der von `num_threads` Threads ausgeführt wird, welche alle Zugriff auf eine gemeinsame und daher zu schützende Variable `busy` erhalten. Auf diesen Befehl folgt dann nach der parallelen Ausführung die Direktive `#pragma omp flush(busy)`, welche sicherstellt, dass der manipulierte Wert der Variablen zurück in den Hauptspeicher geschrieben wird und somit globale Gültigkeit erhält. Manueller gegenseitiger Ausschluss wird in OpenMP durch `#pragma omp critical`, bzw. `#pragma omp`

`atomic` erreicht und `#pragma omp barrier` erlaubt die Synchronisation. Die Definition paralleler (und kritischer) Abschnitte erfolgt also ausschließlich per Anwendung von Compiler-Direktiven. Wie letztendlich Thread-Sicherheit erreicht wird und wie weite Teile der Threadverwaltung realisiert werden, liegt dadurch weder in der Hand des MCSTL-Entwicklers, noch in der des Anwenders, sondern hängt ausschließlich von der verwendeten OpenMP-Implementierung und -Version ab. Das Gleiche gilt für die konsequente Wiederverwendung von Threads durch ein Threadpool-Konzept, auf das sich die Autoren in [45] berufen. Denn die erst im Mai 2008 veröffentlichte Version 3.0 der Spezifikation OpenMPs [16] enthält lediglich die Definition so genannter *„Work Sharing Constructs“*. Diese ermöglichen die Nutzung einer Menge von Threads in mehreren Schleifen oder Ausführungsblöcken (sections), welche allerdings ihrerseits Teil desselben parallelen Abschnitts (eingeleitet durch `#pragma omp parallel`) sein müssen. Diese schwache Definition würde nur zu einer Wiederverwendung von Threads innerhalb eines Funktionsaufrufs führen, für jede MCSTL-Funktion müssten jedoch neue Threads erzeugt werden. Ein Blick in den frei verfügbaren Quelltext der Bibliothek `libgomp` [1], die OpenMP aufbauend auf POSIX Threads innerhalb der GCC realisiert, zeigt jedoch, dass einmal verwendete Threads angehalten und für spätere Aufrufe neu gestartet werden. Dies gilt nicht, wenn es sich um geschachtelte Aufrufe handelt, da dies zu Synchronisationsproblemen führen würde. Mitarbeiter des die MCSTL entwickelnden Lehrstuhls arbeiten an aktuellen Versionen dieser Bibliothek mit und haben nun auch einen echten Threadpool für geschachtelte Parallelität konzipiert. Auch der Intel Compiler sowie der OMPi [31] Compiler implementieren Threadpool-Konzepte. Dennoch erwähnt Intel in Kapitel 8 seines Architecture Optimization Manuals [11], dass die Implementierung im Intel C++ Compiler einen, wenn auch minimalen, Overhead gegenüber von Hand programmiertem Multithreading mit sich bringt. Für wichtige Funktionalität, wie das Einrichten kritischer Abschnitte, zeigt [31] darüber hinaus erhebliche Unterschiede verschiedener OpenMP-Implementierungen hinsichtlich ihrer Performance.

Ein typischer Programmablauf, etwa für die Operation `inner_product()`, läuft innerhalb der MCSTL wie folgt ab: Nach dem Funktionsaufruf durch den Anwender erzeugt die MCSTL intern daraus einen weiteren Aufruf von `inner_product_switch()`. Diese Funktion überprüft den übergebenen Iteratortyp und das die Form der Ausführung bestimmende Tag. Für alle Iteratortypen außer dem `RandomAccessIterator` wird die sequentielle Ausführung der Funktion angestoßen. Dies ist einerseits damit zu begründen, dass etwa für Input-Iteratoren die in Abschnitt 1.2 und 2.2.2 erwähnten Einschränkungen gelten und auch für `Forward`- oder `BidirectionalIterator` die Laufzeit für den Zugriff auf beliebige Elemente der Sequenz nicht als amortisiert $\mathcal{O}(1)$ angenommen werden kann, sondern in der Regel von der Anzahl der Elemente anhängt, um die verschoben wird [6, 32]. Dennoch wurden im Rahmen der MCSTL Ideen entwickelt, wie man auch für diese Iteratortypen Speedup durch parallele Ausführung erreichen kann [29]. Der entsprechende Code ist jedoch noch kein Bestandteil aktueller Veröffentlichungen. Für das hier vorgetragene Beispiel wird

im Folgenden ein `RandomAccessIterator` angenommen, für den die parallelisierte Variante ausgeführt werden soll.

Je nach Parallelisierungs-Tag wird die erfolgversprechendste interne Partitionierungs- und Dispatching-Funktion aufgerufen. Da es sich hier um eine Funktion handelt, die der bereits erwähnten `for each`-Routine entspricht, stehen folgende Varianten zur Verfügung:

- `for_each_template_random_access_ed`: statische Partitionierung im Fall des Tags `parallel_unbalanced`
- `for_each_template_random_access_omp_loop`: dynamische oder statische Partitionierung durch eine `#pragma omp for schedule`-Direktive im Fall des Tags `parallel_omp_loop` bzw. `parallel_omp_loop_static`
- `for_each_template_random_access_workstealing`: dynamische Partitionierung mit Work Stealing bei Einsatz des Tags `parallel_balanced`

Auf das Konzept des Work Stealings wird im Folgenden noch genauer eingegangen. Für einige Aufrufe der für das Dispatching zuständigen Funktionen müssen einige Parameter mit Platzhaltern aufgefüllt werden. Die Entwickler verlassen sich an dieser Stelle darauf, dass der verwendete Compiler etwa Funktionsobjekte mit leerem Rumpf aus dem Code herausoptimiert [44]. Die Standardeinstellung für das Beispiel ist eine statische Partitionierung, wie im folgenden Quelltextauszug 2.7 in Zeile 5 zu erkennen ist.

```
1  template <...> inline T inner_product(InputIterator1 first1,
2     InputIterator1 last1, InputIterator2 first2, T init,
3     BinaryFunction1 binary_op1, BinaryFunction2 binary_op2,
4     mcstl::PARALLELISM parallelism_tag =
5     mcstl::SETTINGS::parallel_unbalanced)
6  {
7     return inner_product_switch(first1, last1, first2, init,
8        binary_op1, binary_op2, typename std::iterator_traits
9        <InputIterator1>::iterator_category(), typename
10       std::iterator_traits<InputIterator2>::iterator_category(),
11       parallelism_tag);
12 }
```

Quelltextauszug 2.7: Frontend für `inner_product()`

Zunächst initialisiert `inner_product_switch()` das Ergebnis mit dem vom Aufrufer übergebenen Wert `init` und erzeugt anschließend ein Funktionsobjekt `my_selector` vom Typ `mcstl_inner_product_selector` (siehe Quelltextauszug 2.8) und übergibt es dem nachfolgend aufgerufenen statischen Parallelisierungsalgorithmus.

```
1  template<typename It, typename It2, typename T>
2  struct inner_product_selector : public generic_for_each_selector<It>
3  {
4    It begin1_iterator;
5    It2 begin2_iterator;
6
7    explicit inner_product_selector(It b1, It2 b2) :
8        begin1_iterator(b1), begin2_iterator(b2) {}
9
10   template<typename Op> inline T operator()(Op mult, It current)
11   {
12     typename std::iterator_traits<It>::difference_type
13     position = current - begin1_iterator;
14     return mult(*current, *(begin2_iterator + position));
15   }
16 };
```

Quelltextauszug 2.8: `inner_product_selector()`

Durch den Funktor `my_selector` kann der Parallelisierungsalgorithmus nun für jede Iteration die gewünschte Funktion auf den Iteratoren ausführen, indem diesem die Distanz des aktuell zu verwendenden Iterators zum Startiterator übergeben wird. Bei genauerer Betrachtung des Funktors fällt auf, dass aus diesem Grund für jede Iteration zwei arithmetische Operationen erforderlich werden, da die bei der Initialisierung verwendeten Startiteratoren nie direkt inkrementiert werden, sondern jedes Mal die Distanz des aktuell übergebenen Iterators `current` zum Startiterator berechnet und anschließend addiert werden muss (siehe Zeilen 12 bis 14 in Quelltextauszug 2.9). In STL-Funktionen wird die jeweilige Aufgabe in der Regel in einem einzigen Funktionsaufruf mittels einer Schleife für das vollständige, durch die Iteratoren definierte, Intervall ausgeführt. Die MCSTL hingegen erzeugt für jede Iteration einen eigenen Funktionsaufruf und verlässt sich auf Compiler-Optimierungen mittels Verwendung des Schlüsselwortes `inline`. Der Compiler entscheidet jedoch heuristisch, ob er eine `inline` Anweisung umsetzen soll. Erst durch Angabe von `__attribute__((always_inline))` wäre das Inlining sichergestellt, jedoch wird dieses Attribut nicht verwendet.

Die Integration des *Work Stealing*-Paradigmas in die MCSTL ist ein wichtiges Konzept für die Effizienz der Bibliothek. Work Stealing bedeutet, dass Threads, welche die ihnen zugeteilte Arbeit vollständig erledigt haben, anderen Threads, deren Jobliste noch nicht vollständig abgearbeitet wurde, einige ihrer Jobs „stehlen“. Auf diese Art der dynamischen Lastverteilung kann der Tatsache Rechnung getragen werden, dass zur Laufzeit keine Annahmen über die Verfügbarkeit und aktuelle Auslastung von CPUs und Kernen getroffen

werden können. Eine Erhöhung der Laufzeit durch einzelne besonders unter Last stehende Kerne kann also abgemildert werden.

Bezüglich dieses Vorgehens und der analytischen Auswertung der zu erwartenden Kosten berufen sich die Entwickler insbesondere auf [22] und [41]. Die in [22] beschriebene Form des Work Stealings zielt jedoch hauptsächlich auf ein hohes Maß an geschachtelter Parallelität ab und basiert auf einem „Busy-Leaves"-Algorithmus. Dies bedeutet, dass Threads, die ihrerseits weitere Threads erzeugen, sich anschließend „schlafen legen" und in einer Datenstruktur verwaltet werden. Aus dieser können dann inaktive Recheneinheiten Jobs herausnehmen, während sichergestellt ist, dass zu jedem Zeitpunkt jedes Blatt (leaf) des Threaderzeugungs-Baumes einen Prozessor zugewiesen hat, um Deadlocks zu vermeiden. In diesem Kontext wird in [22] auch das Konzept eines Threadpools nahegelegt. Der Algorithmus geht im Weiteren davon aus, dass die Threads gleich bei ihrer Erzeugung mehrere Aufgaben zugewiesen bekommen.

Im Fall der STL und aktuellen Mehrkernprozessoren macht dies und die geschachtelte Threaderzeugung allerdings wenig Sinn, da es kaum rechenintensive Unterprogramme gibt, deren aufrufende Threads sich schlafen legen, sondern in fast allen Funktionen, insbesondere den linearen und numerischen, nur ein und dieselbe Aufgabe auf vielen Teilproblemen auszuführen ist. Lediglich für Quicksort wird geschachtelte Parallelität verwendet [44]. Daher verfolgt die MCSTL eine eher an das in [41] beschriebene *receiver-initiated Load Balancing* angelehnte Implementierung. Dabei erfolgt zunächst eine statische Partitionierung mit einer gewissen Teilproblemgröße. Anschließend können bereits inaktive Threads mit Hilfe atomarer Funktionen Aufgaben von anderen noch arbeitenden Threads übernehmen, ohne diese dafür unterbrechen zu müssen. Wenn ein Thread also mit der ihm zugeteilten Arbeit fertig ist, führt er folgendes Programmfragment (Quelltextauszug 2.9) aus.

Falls ein Thread mit weiterer Arbeit gefunden werden konnte (`supposed_load` ist > 0), erfolgt der eigentliche „Stehlvorgang", (Quelltextauszug 2.10), dem einige Synchronisierungsaufrufe folgen. So lange ein Thread Arbeit hat, führt er die entsprechende Funktion auf den Iteratoren aus.

Wie bereits beschrieben, verwendet die MCSTL dieses Konzept aber nicht für alle Funktionen. Nach manueller Zählung auf Basis der Version 0.8.0-beta wird Work Stealing lediglich für die Algorithmen `adjacent_difference()`, `fill()`, `find()`, `balanced_quicksort()`, `for_each()`, `transform()`, `replace()` und `generate()` sowie Varianten dieser, wie etwa `replace_if()`, angewendet.

Beim Funktionsaufruf des Threads in Zeile 4 des Quelltextauszugs 2.11 wird die bereits aus der STL bekannte hohe Abstraktion und Funktionsaufruftiefe auch innerhalb der MCSTL deutlich. Die Instanz `op` ist ein Funktionsobjekt, welches die gewünschte STL-Funktion beinhaltet und dem im Beispiel zuvor beschriebenen `my_selector`-Objekt entspricht. Die Funktion `f` ist ebenfalls ein Funktor, dessen Funktionsoperator hier verwendet

```
DiffType supposed_first, supposed_last, supposed_load;
do
{
   yield(); // Gib aktuellen Zeitschlitz freiwillig frei

   // Globalisiere Informationen über Arbeitsstatus der Threads
   #pragma omp flush(busy)

   victim = rand_gen(); // Wähle zufälligen Opfer-Thread

   // Kopiere Informationen über dessen Job-Liste in thread-eigene Variablen
   supposed_first = job[victim * stride].first;
   supposed_last = job[victim * stride].last;
   supposed_load = job[victim * stride].load;
} while (busy > 0 && ((supposed_load <= 0) ||
      ((supposed_first + supposed_load - 1) != supposed_last)));
```

Quelltextauszug 2.9: Opfer-Thread Suche für Work Stealing

```
   // Anzahl der zu stehlenden Elemente (> 0)
   steal = (supposed_load < 2) ? 1 : supposed_load / 2;

   // Stehle den Startiterator und erhöhe den des Opfers (atomare Operation)
   DiffType stolen_first = fetch_and_add<DiffType>
        (&(job[victim * stride].first), steal);

   // Setze eigene Iteratoren
   my_job.first = stolen_first;
   my_job.last = std::min<DiffType>
       (stolen_first + steal - (DiffType)1, supposed_last);
   my_job.load = my_job.last - my_job.first + 1;
```

Quelltextauszug 2.10: Stehlvorgang

wird, um `op` den aktuell zu verarbeitenden Iterator (`begin + my_first`) zu übergeben. Für entsprechende Funktionen kommt zusätzlich ein Reduktionsfunktor `r` zum Einsatz.

Bei den Benchmarks in [45] wurde die Ausführung der parallelen Variante erzwungen, auch wenn auf Grund der Eingabegröße die sequentielle Ausführung schneller gewesen wäre. Abbildung 2.8 zeigt die Ergebnisse für den numerischen Algorithmus `partial_sum()`, der mit 32-bit Werten vom Typ `int` auf einem Sun UltraSPARC T1 Prozessor ausgeführt wurde. Die CPU stellt acht Kerne à 1 GHz, bis zu 32 simultan unterstützte Hardware-Threads und 3 MB shared L2-Cache zur Verfügung. Der parallele MCSTL-Algorithmus

```
1 if(my_job.first <= my_job.last)
2 {
3     DiffType my_first = my_job.first;
4     result = f(op, begin + my_first); // Funktionsaufruf
5     my_job.first++;
6     my_job.load--;
7 }
```

Quelltextauszug 2.11: Iterationsschritt der parallelen Funktionsausführung

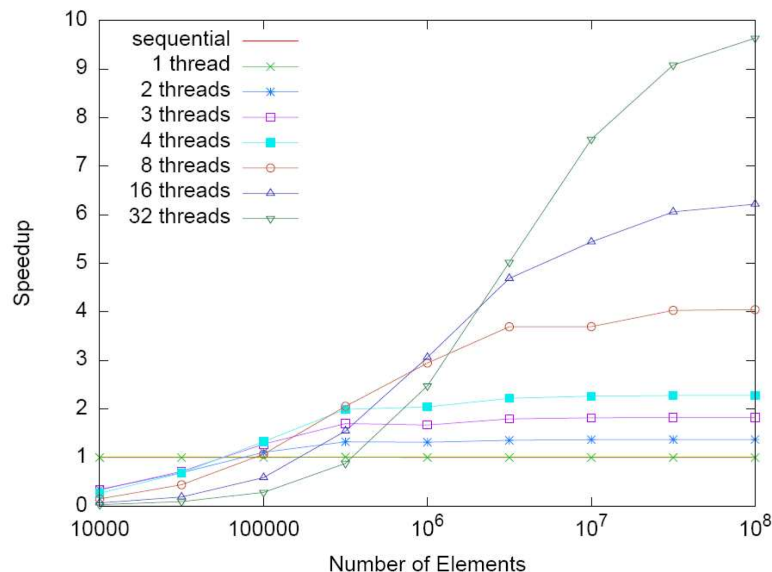

Abbildung 2.8: `partial_sum()` für 32-bit Integer auf einem Sun T1 System (Quelle: [45])

ist bei zwei verwendeten Threads ab einer Eingabegröße von 10^5 genauso schnell wie die STL selbst. Für bis zu vier Threads ist auch im Folgenden der Speedup eher mäßig, mit acht und mehr Threads hingegen kann ab 10^6 Elementen signifikanter Speedup erreicht werden. Die Ergebnisse werden damit erklärt, dass nahezu die doppelte Arbeit in der parallelisierten Variante zu erledigen sei [45]. Diese Begründung ist auch akzeptabel, da für alle Teilprobleme außer dem ersten in der Tat der jeweilige initiale Wert, auf den die weiteren Partialsummen addiert werden müssen, nicht zur Verfügung steht (und folglich durch einen zusätzlichen Durchlauf berechnet werden muss). Dieses Problem und die verwendeten Lösungsansätze werden in Abschnitt 3.3.3 detaillierter behandelt. Für vollständiges Sortieren (siehe Abbildung 2.9) auf einem Quadcore Xeon- und einem Dualcore Opteron-System zeichnet sich für größer werdende Eingaben eine hervorragende Skalierung und eine gewisse Überlegenheit des Intel Prozessors mit geteilten Caches ab.

Als weiteres Beispiel aus [45] wird die Funktion `find()` betrachtet, da die MCSTL hier den anderen vorgestellten Bibliotheken einen wesentlichen Schritt voraus hat. Naive

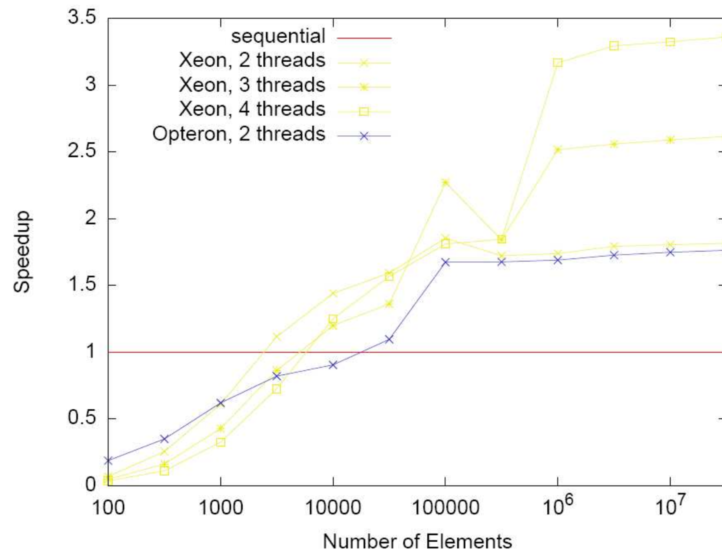

Abbildung 2.9: `sort()` für 32-bit Integer auf Xeon- und Opteron-Systemen (Quelle: [45])

Implementierungen müssen, wenn das entsprechende Element von einem der parallel arbeitenden Threads gefunden wurde, die Terminierung des Suchalgorithmus innerhalb der anderen Threads abwarten, bzw. können diesen nicht unterbrechen.

Sei nun ein Element an Stelle m in einem Container der Länge n positioniert. Dann benötigt der sequentielle Algorithmus Zeit $\mathcal{O}(m)$. Eine naive parallele Variante teilt den Container in p Teile der Größe $\frac{n}{p}$. Wenn auf die Terminierung aller Threads gewartet werden muss, ist die Laufzeit also in jedem Fall $\Theta(\frac{n}{p})$. Dies bedeutet für $m < \frac{n}{p}$ einen Speedup < 1 gegenüber dem sequentiellen Vorgehen. Um unverhältnismäßig hohen Overhead für ein sehr kleines m zu vermeiden, beginnt die MCSTL-Variante mit einer sequentiellen Suche innerhalb der ersten m_0 Elemente. Danach werden mehreren Threads nach verschiedenen Strategien Abschnitte einer bestimmten Größe zugewiesen. Die Größe kann entweder einheitlich sein (fixed-size block (fsb)) oder ansteigen (growing block (gb)). Findet ein Thread das gesuchte Element, stiehlt er alle Jobs der anderen Threads, sodass diese schnell terminieren können. Aus diesem Grund ist die Wahl der Blockgröße auch leistungsbestimmend. Wird sie klein gewählt, funktioniert die abschließende Terminierung der anderen Threads relativ schnell, aber andererseits sind dann mehr atomare *fetch_and_add* Operationen erforderlich, um den Threads Blöcke zuzuweisen [45]. Abbildung 2.10 zeigt, dass die gb-Variante die besten Ergebnisse auf dem Sun T1 System erreicht. Leider wird die Performance für Container mit Größen unterhalb 10^5 Elementen nicht dargestellt.

Den bisher vorgetragenen Ergebnissen ist gemein, dass sie eher die Performance spezieller, sich zum Teil auch deutlich von der STL unterscheidender Algorithmen beschreiben. So mussten `find()` und `partial_sum()` aus den beschriebenen Gründen neu implementiert werden und für die Sortierverfahren wird in [45] insbesondere das Konzept des *multiway*

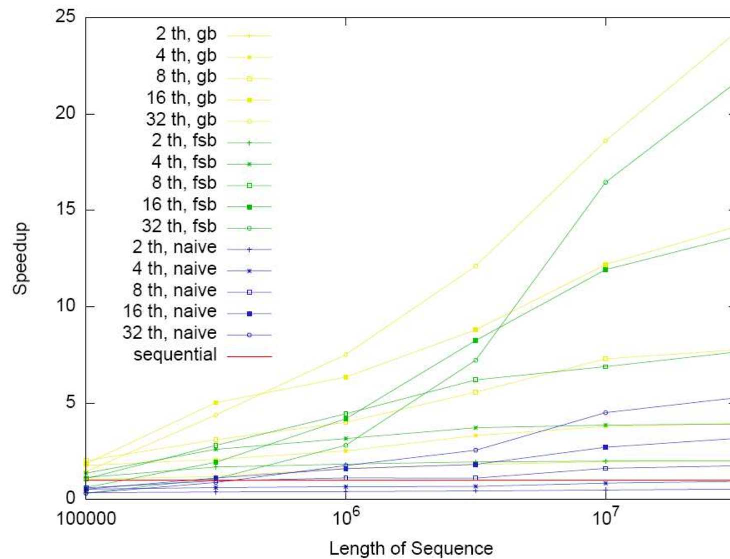

Abbildung 2.10: verschiedene Verfahren für `find()` und 32-bit Integer auf einem Sun T1 System (Quelle: [45])

merging beworben. Interessant sind jedoch auch Benchmarks für einfachere Algorithmen, die einen oder zwei Container linear durchlaufen und eine arithmetische Operation darauf ausführen. Denn insbesondere bei diesen trivial parallelisierbaren und dem bereits erwähnten `for each`-Schema entsprechenden Funktionen wird schnell erkennbar, wie hoch der Overhead der Partitionierung und des Multithreadings wirklich ist. Als Beispiel dient die numerische Funktion `accumulate()` in Abbildung 2.11. Erneut wird die Leistung des sequentiellen STL-Aufrufs bei ca. 50000 Elementen Eingabegröße erreicht. Für Eingabegrößen ab etwa einer halben Million Elemente skaliert der Algorithmus auf dem Sun T1 Prozessor sehr gut, für bis zu vier Threads wird der maximale Speedup erreicht. Werden mehr Threads eingesetzt, als logische Ausführungseinheiten vorhanden sind, sind die Ergebnisse ebenfalls der Architektur entsprechend erwartet gut. Es wird deutlich, dass selbst bei einem in der Laufzeit vom Speicherdurchsatz abhängigen Algorithmus eine enorme Anzahl Threads erforderlich ist, um die Speicherbandbreite des Systems zu saturieren. Dies ist ein wesentlicher Unterschied zu gängigen x86-Prozessoren, deren absolute sequentielle Performance bereits deutlich höher ist, als die eines T1-Kerns. Bei ersten eigenen Messungen der Leistung auf x86-Multicore-Prozessoren offenbarten sich daher deutlich größere Schwächen der MCSTL für kleinere Eingaben, als bei den vorgestellten Sun T1 Ergebnissen. Insgesamt ist die Schwelle, ab der mit Hilfe der Parallelität Speedup erzielt wird, bei vielen der Funktionen hoch. Dies wird von den Autoren in [45] weitgehend ignoriert, mit der Begründung, dass für solche Eingabegrößen der Algorithmus normalerweise sequentiell ausgeführt würde. Es ist jedoch äußerst fraglich und auch Gegenstand des experimentellen

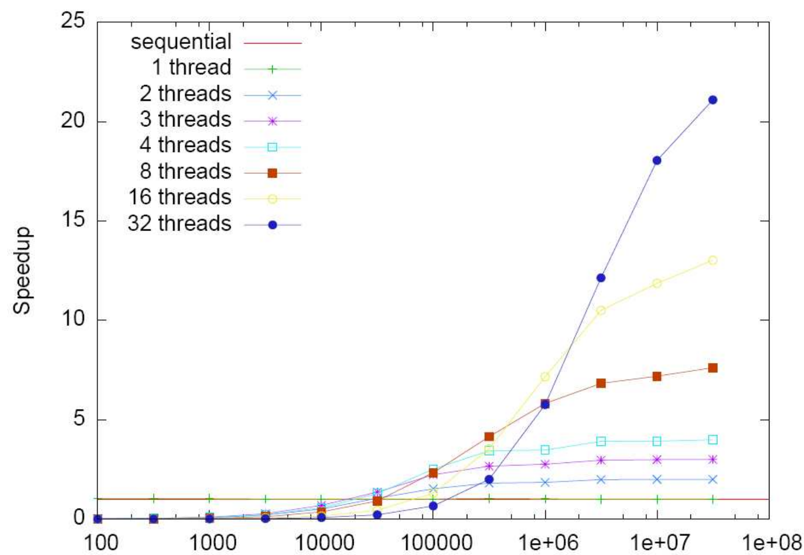

Abbildung 2.11: Speedup für `accumulate()` auf einem Sun T1 System (Quelle: [45])

Teils dieser Diplomarbeit, ob man in diesem Bereich nicht bereits einen Leistungsgewinn gegenüber der sequentiellen Ausführung erzielen kann.

Die MCSTL-Entwickler beschäftigen sich neben der Parallelisierung von Algorithmen auch mit parallelem Einfügen und Löschen für die Datenstrukturen `std::map` und `std::set`. Dabei wurden Strategien entwickelt, wie Baumstrukturen (die STL verwendet Rot-Schwarze Bäume) effizient von mehreren Threads manipuliert werden können. In [28] werden erste erfolgreiche Ergebnisse präsentiert, die zum Teil auch für kleinere Einfügesequenzen Speedup zeigen. In der Version 0.8.0-beta sind diese Erweiterungen jedoch noch nicht implementiert.

Zum Abschluss dieses Abschnitts sei erwähnt, dass auch die MCSTL ein wenig Adaptivität beinhaltet, da in der Datei mcstl_features.h einige Algorithmenvarianten (wie z.B. auch das soeben erwähnte multiway merging) je nach gesetztem Compiler-Flag zur möglichen Verwendung aktiviert werden oder nicht. Für die Wahl des entsprechenden Algorithmus und unterschiedlicher Strategien innerhalb der Algorithmen (etwa zur Partitionierung von Sequenzen), werden zur Laufzeit diverse Tags im Namensraum `MCSTL::SETTINGS` verwendet.

2.3.3 Fazit

Von den drei hier betrachteten Bibliotheken ist die MCSTL die ausgereifteste. Sie erreicht insbesondere das Ziel einer sehr einfachen Anwendung nahezu ausschließlich durch Rekompilation und bietet, soweit bekannt, die größte Anzahl parallelisierter STL-Funktionen.

Des Weiteren verfolgt sie die Konzepte der STL konsequent weiter, insbesondere durch die sehr starke Verwendung von generischer Programmierung und Iteratoren, sowie ihrer durch die STL definierten Eigenschaften (`iterator_traits`). Auch die Unterstützung der

Algorithmen für alle Iteratortypen wird auf diese Weise sichergestellt und stellt einen Vorteil gegenüber der MPTL dar, welche bestimmte Iteratorentypen vollständig (d.h. auch von einer sequentiellen Ausführung) ausschließt.

Die MCSTL-Entwickler haben für einige Algorithmen, wie etwa `random_shuffle()`, die explizite Berücksichtigung von Hardware-Faktoren wie Cache- und TLB-Puffergrößen implementiert. Auch hier sind sie der MPTL überlegen, während STAPL ebenfalls solche Werte für die adaptive Selektion von Algorithmen verwendet.

Mit dem Work Stealing Algorithmus wurde erstmals ein neuer Ansatz für dynamisches Load Balancing erfolgreich in die Praxis umgesetzt. Damit ist die MCSTL den anderen beiden hier betrachteten Bibliotheken einen weiteren Schritt voraus, wobei in Vortragsfolien zu STAPL aus dem Jahr 2007 ebenfalls von Work Stealing die Rede ist [40].

Die hier und in der Literatur vorgestellten Ergebnisse können eine gewisse Schwäche der Algorithmen für kleinere Eingabegrößen jedoch nicht verdecken. Insgesamt verlässt sich die MCSTL in hohem Maße auf Compiler-Optimierungen, beispielsweise durch das Inlining vieler Funktionen und die Erwartung, keinen Overhead durch leere Funktionsaufrufe (Dummies) hinnehmen zu müssen. Durch die Verwendung von OpenMP wird die Leistung der Bibliothek von nicht beeinflussbaren Faktoren abhängig, wie etwa der jeweiligen Implementierung im Rahmen des verwendeten Compilers. Die in der Zukunft immer wichtiger werdende feste Zuweisung von Threads zu Prozessorkernen (Thread Affinity) wird ebenfalls erschwert, ist jedoch denkbar, wenn sie intern in der OpenMP-Implementierung erfolgt. Ein Gedankenaustausch mit Johannes Singler bei seiner Anwesenheit am Lehrstuhl XI der Fakultät Informatik führte zu einigen Ansätzen, wie dies in Zukunft innerhalb von libgomp verwirklicht werden könnte. Insgesamt lässt sich die Wahl OpenMPs erklären. Die notwendige Allgemeinheit und Plattform-Unabhängigkeit der Bibliothek wird auf diese Weise einfacher erreicht. Ebenso kann die Stabilität der Implementierung besser garantiert werden, indem auf eine bereits viel verwendete und bekannte Schnittstelle für das Multithreading zurückgegriffen wird. Verbesserungen und Erweiterungen OpenMPs, auch durch Entwickler der MCSTL, führen des Weiteren auch zu unmittelbaren Verbesserungen der gesamten Bibliothek. Nicht zuletzt ist die Verwendung von Direktiven aus Sicht einer Bibliothek eleganter, als das manuelle Management von Threads.

Im folgenden Kapitel ist es die Aufgabe, herauszustellen, ob die in der Literatur zu findende mäßige Skalierung der MCSTL-Implementierungen für kleine Eingabegrößen mit der Wahl dieses Programmierkonzepts zu begründen ist, oder ob die manuelle Ausnutzung sämtlicher Multithreading-Konzepte auch keine wesentlich besseren Ergebnisse hervorbringt. Dies wäre dann ein Hinweis darauf, dass die Erzeugung und Verwaltung von Threads für derart kleine Eingabegrößen in der Tat zu teuer ist und jeden Geschwindigkeitsgewinn durch parallele Ausführung mehr als aufwiegt, wie es die Autoren in [45] ohne weitere Analyse behaupten.

Interessant werden die Entwicklungen im Bereich der parallelen Container sein, die zum aktuellen Zeitpunkt noch nicht verfügbarer Bestandteil des GNU parallel mode sind. Mit ihrer Hilfe könnten gegebenenfalls auch einige Algorithmen, die viel auf Datenstrukturen arbeiten, nochmals beschleunigt werden. Hier wäre dann eine feste Bindung von Threads an Kerne jedoch noch bedeutender, da in diesem Fall Partitionen eines Containers auf unterschiedlichen Kernen erzeugt werden und die Daten daher auch in ihren Caches verbleiben.

2.4 Vergleich der drei Parallelisierungsansätze

Wie in diesem Kapitel ausführlich beschrieben, haben die behandelten Bibliotheken sowohl viele Gemeinsamkeiten, als auch Unterschiede. Tabelle 2.1 fasst die wichtigsten Eigenschaften von STAPL, MPTL und MCSTL nochmals zusammen.

Merkmal	**STAPL**	**MPTL**	**MCSTL**
Multithreading-API	NPTL, OpenMP, MPI	NPTL	OpenMP
Kommunikationslayer	ARMI	nein	nein
parallele Container	ja	nein	nein
Threadpool	nein	nein	OpenMP-intern
dynamisches Load Balancing	ja	ja	ja + WorkStealing
Thread Affinity	nein	nein	nein
Einbeziehung von Hardware-Eigenschaften	ja	nein	ja
Adaptivität	ja	nein	teilweise

Tabelle 2.1: Gegenüberstellung wichtiger Bibliothekseigenschaften

Sehr auffällig ist, wie bereits in den Fazits zu den einzelnen Bibliotheken herausgestellt, dass sie alle keinen Gebrauch des modernen Konzepts der Thread Affinity machen. Es stellt sich die Frage, ob durch ihren Einsatz in Kombination mit einem manuell verwalteten globalen Threadpool das Ziel einer besseren Leistung bei kleinen und mittleren Eingabegrößen erreicht werden kann. Deshalb werden sie in der im Rahmen dieser Diplomarbeit entwickelten Funktionen eine zentrale Rolle spielen. Das nachfolgende Kapitel beschreibt nun im Detail, mit welcher Strategie dieses Ziel verfolgt wird.

Kapitel 3

Beschleunigung ausgewählter STL-Algorithmen

3.1 Entwickeltes Multithreading-Konzept

Bevor der im Rahmen dieser Diplomarbeit entwickelte Multithreading-Ansatz detailliert beschrieben wird, erfolgt zunächst die Erläuterung einiger zentraler Begriffe, die häufig Verwendung finden und zum Teil auch als Entitäten im Quellcode existieren.

3.1.1 Begriffserklärung

- Mutex: Variable zur Realisierung gegenseitigen Ausschlusses (***mutual exclusion***). In der NPTL wird ein Mutex durch ein Objekt vom Typ `pthread_mutex_t` repräsentiert.

- Lock: Inanspruchnahme einer Mutex-Variablen durch einen Prozess oder Thread (Man spricht auch von „locken"). Ist sie bereits im „Besitz" eines anderen Prozesses oder Threads, muss der anfragende Thread auf die Freigabe (*Unlock*) warten, bevor er mit seinen Instruktionen fortfahren kann. Die zugehörigen Funktionen lauten `pthread_mutex_lock()` und `pthread_mutex_unlock()`.

- ConditionVariable: Dient der Synchronisation zwischen Prozessen und Threads. Ein Prozess oder Thread bewirkt mittels `pthread_cond_signal()` bzw. `pthread_cond_-broadcast()` eine Signalisierung, auf die ein weiterer Prozess oder Thread mit Hilfe der Funktion `pthread_cond_wait()` wartet.

- Task: Aufgabe, die von einem Thread ausgeführt werden soll und durch die Klasse `Task` repräsentiert wird.

- Barrier: Form der Synchronisation mehrerer (oder sämtlicher) Threads. Die Rückkehr der realisierenden Funktion erfolgt erst, wenn die Threads ihre aktuellen Tasks

beendet haben. Ein Barrier ist eine strengere Synchronisation als ein *Fence*, da die Threads bis zur Rückkehr der Funktion ebenfalls keine neuen Aufgaben verrichten dürfen. Für welche Threads und welche Tasks die Synchronisation erfolgt, ist dabei implementierungsabhängig. Die NPTL stellt ein Objekt namens `pthread_barrier_t` bereit, welches für die Implementierung dieser Funktionalität verwendet werden kann.

Im Folgenden werden einige dieser Begriffe sowohl im allgemeinen Sinne als Gegenstände verwendet, als auch als konkrete Objekte zur Laufzeit. Zweites wird durch die Verwendung von Schreibmaschinenschrift angedeutet. Dies bezieht sich insbesondere auch auf den Begriff Thread. Die Begriffe „logische Ausführungseinheit", „logischer Prozessor" und „Kern" werden im Folgenden wie Synonyme behandelt, auch wenn im Sinne von Simultaneous Multithreading kein vollständiger Kern erforderlich ist, um eine logische Ausführungseinheit bereitzustellen. Des Weiteren werden für experimentelle Untersuchungen häufig Systeme eingesetzt und im Text genannt, die auch für die Erzeugung der Ergebnisse in Kapitel 4 verwendet werden. Für Details zu ihrer Hardware sei auf Tabelle 4.1 verwiesen.

3.1.2 Konstruktion der Multithreading-Basis

In Kapitel 2 wurde auf Basis vieler Ergebnisse die These geäußert, dass die NPTL einen vergleichsweise geringen Overhead erzeugt. Da sie darüber hinaus wesentlich mehr Möglichkeiten zur Einflussnahme bietet als OpenMP, wurde diese Multithreading-API für das hier entwickelte Konzept gewählt. Ein White Paper ([25]) zeigt die Verbesserungen, die mit Hilfe der NPTL gegenüber älteren LinuxThreads-Varianten erreicht werden konnten und erläutert wesentliche Designentscheidungen, wie z. B. die Beziehung zwischen *User-Level-Threads* und *Kernel-Threads*.

Thread und Task. Grundlegender Baustein des hier gewählten Ansatzes ist die Klasse `Thread`, welche jeweils den Zugriff auf einen POSIX Thread steuert. Dem Programmierparadigma *Private Implementation Pattern* folgend wurde innerhalb der Klasse eine Struktur `Implementation` entwickelt, die alles Notwendige für die Ausführung von selbst definierten Funktionen innerhalb eines POSIX Threads beinhaltet, und die diesem bei der Erzeugung per Zeiger übergeben werden kann. Als Hauptfunktion erhält der Thread eine `while`-Schleife, die nur dann terminiert, wenn der Destruktor des Thread-Objekts aufgerufen wird. Stehen keine Aufgaben zur Ausführung an, betritt der Thread mit Hilfe einer `ConditionVariable` einen Wartezustand. Dies erlaubt dem Scheduler des Betriebssystems, den Thread zunächst von der Ausführung auszuschließen, bis eine entsprechende Signalisierung eintritt. Somit interferiert dieser nicht mit anderen Prozessen auf dem jeweiligen Kern und der threadeigene Mutex wird freigegeben.

Mit Hilfe der Methode `Thread::run()` kann dem Thread ein `Task` übergeben werden. Dieser ist im Wesentlichen ein Funktionsobjekt vom Typ `std::tr1::function`, welches

beliebige andere Funktionsobjekte mit gleicher Signatur aufnehmen kann. `Thread::run()` bewirkt eine Signalisierung der soeben beschriebenen `ConditionVariable`, in Folge dessen der Thread aufwacht und den Funktionsoperator des übergebenen Funktors aufruft, sodass die dort befindlichen Instruktionen ausgeführt werden. Anschließend prüft der Thread durch direkten Zugriff auf die Taskliste des im Folgenden beschriebenen Threadpools, ob weitere Arbeit verfügbar ist. Ist dies nicht der Fall, fällt er in den Wartezustand zurück, ansonsten entnimmt er der Taskliste ein weiteres Funktionsobjekt und fährt mit dessen Ausführung fort. Dieses Verfahren wird fortgesetzt, bis irgendwann der Destruktor des `Thread`-Objekts aufgerufen wird und der Thread terminiert.

Immer dann, wenn der Thread die im Funktionsobjekt gespeicherten Instruktionen vollständig ausgeführt hat, signalisiert er die `ConditionVariable` des `Task`-Objekts mittels derer der erzeugende Prozess auf die Fertigstellung wartet. Erfolgt die Signalisierung bereits bevor auf sie gewartet wird, geht ihr Effekt verloren. Deshalb sichert eine zusätzliche boolsche Variable ab, dass die bereits erfolgte Ausführung dennoch nachvollzogen werden kann und nicht unnötig gewartet wird.

Bei Konstruktion besteht die Möglichkeit, einem `Task`-Objekt die Anzahl der zu transferierenden Bytes zu übergeben, welche dann als Parameter `load` gespeichert wird. Ein Aufruf von `std::inner_product()` für zwei Vektoren der Größe 10^5 und dem Datentyp `double` erfordert beispielsweise Speichertransfers für $2 \cdot 8 \cdot 10^5\ B = 1,6\ MB$. Für einige Algorithmen kann diese Information im Zusammenspiel mit der Kenntnis von Cachegrößen von Nutzen sein. Wie genau diese Information für das Zuweisen von Tasks an Threads verwendet wird, behandelt Abschnitt 3.2.3.

ThreadPool. Wie das beschriebene Konzept bereits suggeriert, wurde für die effektive Wiederverwendung von Threads ein Threadpool konzipiert, welcher durch die *Singleton*-Klasse `ThreadPool` repräsentiert wird. Ein Singleton ist eine Klasse, die während der Laufzeit nur einmalig instantiiert werden kann. Sie erleichtert auf diese Weise auch die vollständige Übernahme der STL-Funktionssignaturen, da es mit diesem Konzept nicht notwendig ist, einen Zeiger auf ein `ThreadPool`-Objekt zu übergeben. Wie viele Objekte vom Typ `Thread` der Pool verwaltet, hängt von den beiden Parametern `num_threads` und `affinity` des Konstruktors ab. Wird für den zweiten Parameter der Wert `true` übergeben, wird der Betriebssystem-Scheduler mit entsprechenden C-Funktionen angewiesen, den jeweiligen Thread nur auf einem bestimmten, fest zugewiesenen Kern auszuführen. Die Zuweisung der Threads zu Kernen erfolgt nach dem round-robin Prinzip. Sei p die Anzahl der verfügbaren logischen Ausführungseinheiten, dann beträgt die Anzahl der erzeugten Threads $\max(\texttt{num_threads} - 1,\ p - 1)$, da aus den in Abschnitt 3.2.2 erläuterten Gründen der Hauptprozess in allen entwickelten Funktionen an der Ausführung dieser beteiligt wird. Stehen weniger Ausführungseinheiten zur Verfügung, als Threads erzeugt wurden, werden einigen von ihnen mehrere Threads zugewiesen. Der Grund dafür, dass bei Verwen-

dung von Thread Affinity mindestens $p-1$ `Thread`-Objekte erzeugt werden ist, dass wie ebenfalls in Abschnitt 3.2.2 beschrieben, die explizite Auswahl von Ausführungseinheiten für die Laufzeit der hier entwickelten Funktionen äußerst kritisch sein kann. Die genaue Zuordnung der Scheduler-Nummerierung zu der realen Zugehörigkeit von Kernen zu Prozessoren kann jedoch nicht vor oder während der Erzeugung der Threads ermittelt werden. Daher muss zunächst jedem Kern ein Thread zugewiesen werden, welcher im Fall von x86-Prozessoren anschließend mit Hilfe von Assemblerbefehlen die eindeutige *Local APIC ID* des jeweiligen Kerns bestimmt. Bei anderen Architekturen erfolgt die Zuweisung auf Basis der vom Betriebssystem vergebenen ID, welche in vielen Fällen mit der realen Anordnung der Kerne übereinstimmt. Danach kann auf Basis dieser Nummerierung eine Sortierung der `Thread`-Objekte erfolgen, um die gewünschte explizite Zuweisung vornehmen zu können. Bei ausgeschalteter Thread Affinity bleiben diese Maßnahmen aus und es werden exakt $num_threads - 1$ Threads erzeugt. Mehr zum Thema Thread Affinity folgt in Abschnitt 3.1.4. Mit der Methode `add_threads(num_threads)` können nachträglich Threads erzeugt und dem Pool hinzugefügt werden. Die Zuweisung von Threads an Kerne wird an der Stelle fortgesetzt, an der der Konstruktor, oder der letzte Aufruf dieser Funktion, aufgehört hat. Entsprechend können mit `delete_threads(num_threads)` Threads aus dem Pool entfernt werden.

Jede Zuweisung von, und jedes Warten auf `Task`s, geschieht indirekt über den Threadpool, sodass dieser zu jeder Zeit die Information verwaltet, welche Threads aktuell Aufgaben ausführen und welche für neue Anfragen verfügbar sind. Zu diesem Zweck werden zwei Listen namens `idle_threads` und `working_threads` eingesetzt. Für die Zuweisung von Tasks stehen mehrere Methoden zur Verfügung. Diese sind im Einzelnen:

- `ThreadPool::dispatch(Task)`: Übergibt einen `Task` an einen `Thread`, falls `idle_threads` nicht leer ist. Ansonsten wird der Task in eine weitere Liste namens `tasks` eingefügt. Diese Funktion eignet sich besonders dann, wenn die Summe der auszuführenden Tasks größer ist, als die Anzahl der zur Verfügung stehenden Threads. Wie bereits beschrieben, nehmen sich Threads nach Ausführung ihrer ursprünglichen Aufgabe ggf. vorhandene weitere Tasks selbsttätig aus der Liste heraus. Auf diese Weise wird der Overhead für die Zuweisung minimiert und der Hauptprozess muss zur Übergabe eines Tasks nicht auf die Fertigstellung anderer Tasks warten, sondern kann asynchron bereits weitere Instruktionen ausführen.

- `ThreadPool::wait_and_run(Task)`: Arbeitet wie `dispatch()`, jedoch wird im Fall, dass `idle_threads` leer ist gewartet, bis ein Thread seine aktuelle Aufgabe beendet hat und diesem anschließend der `Task` übergeben. Diese Methode hat als Rückgabetyp einen Zeiger auf den verwendeten Thread. Auf diese Weise wird es aufrufenden Funktionen im Zusammenspiel mit der nachfolgend beschriebenen Methode erlaubt, Threads für nachfolgende Tasks wiederzuverwenden. Für einige Algorithmen kann

dies von Vorteil sein, etwa um Cache-Effekte bei der Arbeit verschiedener Tasks auf den gleichen Daten auszunutzen.

- `ThreadPool::dispatch(Thread, Task)`: Hier wird dem übergebenen `Thread` der übergebene `Task` zugewiesen. Ist der Thread nicht inaktiv, wird zunächst auf die Fertigstellung des aktuellen Tasks gewartet.

- `ThreadPool::wait_for_task(Task)`: Wie bereits beschrieben, wird auch indirekt über Funktionen des Threadpools auf die Fertigstellung eines Tasks gewartet. Diese Methode realisiert die entsprechende Funktionalität, entnimmt den für die Ausführung verwendeten Thread der Liste `working_threads` und fügt ihn der Liste `idle_threads` hinzu, sodass er für die Wiederverwendung zur Verfügung steht.

- `ThreadPool::barrier(Tasks, num_tasks)`: Realisiert einen lokalen Barrier, d.h. bei Rückkehr dieser Methode ist sichergestellt, dass alle übergebenen Tasks fertiggestellt wurden. Zusätzlich werden die `Task`-Objekte zerstört und ihr Speicherplatz freigegeben. Sie findet daher vorwiegend zur Synchronisation nach einem parallelen Abschnitt Verwendung und aktualisiert ebenfalls die beiden Statuslisten.

- `ThreadPool::global_barrier()`: Bei Rückkehr dieser Methode ist sichergestellt, dass alle Threads inaktiv sind und sich kein Task mehr in der Taskliste des Threadpools befindet. Diese Funktionalität wird von den hier entwickelten parallelen Algorithmen nicht benötigt und ist daher nur der Vollständigkeit halber implementiert.

Für die zuletzt aufgeführte Methode und `ThreadPool::dispatch(Thread, Task)` ist es erforderlich, auf die Signalisierung der Inaktivität eines Threads, anstelle auf die Fertigstellung eines Tasks zu warten. Die Klasse `Thread` unterstützt beide Verfahren wie im nachfolgenden Abschnitt 3.1.3 erläutert wird.

Es ist Teil des Konzepts, dass nahezu alle beschriebenen Funktionen ohne *kritische Abschnitte* implementiert sind und demnach auch kein gegenseitiger Ausschluss erforderlich ist. Dies ist möglich, weil der `ThreadPool` die alleinige Kontrolle über alle seine Datenstrukturen und die Threads hat. Die einzige Ausnahme hiervon besteht darin, dass Threads auf die Taskliste des Threadpools zugreifen. Daher werden Zugriffe auf diese Liste mit einem `Mutex` geschützt, die internen Datenstrukturen wie `idle_threads` oder `working_threads` jedoch benötigen keinen gesonderten Schutz.

Neben den beschriebenen Synchronisationsaufgaben stellt die Klasse `ThreadPool` auch die zentrale Informationseinheit für alle implementierten Algorithmen dar. Bei ihrer Konstruktion werden mit Hilfe von Betriebssystemfunktionen und Assemblerbefehlen wie etwa `cpuid` [14] diverse Hardwareeigenschaften ausgelesen und in einer Struktur namens `Architecture` gespeichert. Beispiele hierfür sind die Größe von L1-, L2- und L3-Caches und ihrer Blöcke (*Cachelines*), sowie architekturbezogene Informationen wie die Anzahl der

physikalischen Prozessoren (*Physical Processor Packages*) und die Anzahl der logischen Ausführungseinheiten, die Teil eines solchen Prozessors sind. Mit diesen Daten und weiteren vom Benutzer übergebenen Compiler-Flags (wie z. B. `-DNUMA`), sowie dem erwähnten Parameter `load` der `Task`-Objekte, ist es möglich, die Ausführung der parallelen Algorithmen an die Hardware anzupassen. Abbildung 3.1 fasst den Aufbau der beschriebenen Komponenten nochmals schematisch zusammen.

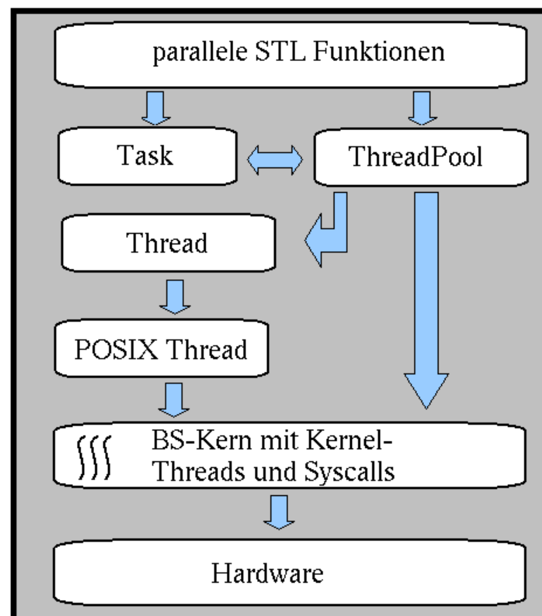

Abbildung 3.1: Schematischer Aufbau der Multithreading-Komponenten

3.1.3 Wesentliche Designentscheidungen

Grundsätzlich stellt sich bei einem Multithreading-Konzept die Frage, ob man einen Task oder einen Thread als zentrales Ausführungsobjekt ansehen möchte. Der wesentliche Unterschied besteht darin, ob Algorithmen Tasks definieren und auf die Signalisierung ihrer Fertigstellung warten (*task-based programming*), oder nach Zuweisung von Aufgaben an Threads darauf gewartet wird, dass diese ihre Inaktivität signalisieren und somit die Aufgaben erledigt wurden (*thread-based programming*). Wenn man neben der expliziten Zuweisung von Tasks einem Thread auch erlaubt, sich selbstständig Tasks aus der Liste des Threadpools zu holen, dann ist bei der zweiten Strategie nicht immer gegeben, dass das Warten auf die Signalisierung des Threads mit der Fertigstellung genau eines Tasks korreliert.

Konzeptuell spielt diese Fragestellung im Fall der Parallelisierung der meisten STL-Funktionen eine untergeordnete Rolle. Denn aufgrund des Typs des überwiegenden Teils dieser Funktionen besteht eine Parallelisierung selten aus Aufgaben, auf deren Fertigstellung zu verschiedenen Zeitpunkten gewartet werden muss. Das heißt, dass die Paralleli-

sierung in fast allen Fällen einem bereits zuvor auch als „fork-and-join“-Parallelisierung bezeichneten Schema genügt.
1. statische oder dynamische Partitionierung und Aufgabenverteilung
2. parallele Ausführung
3. Barrier (Warten auf alle Tasks / Threads)
4. Reduktion (optional)
Die Schritte 2 und 3 können dabei ggf. mehrfach wiederholt werden. Im Fall komplexerer Funktionen, bei denen es erforderlich ist, zu unterschiedlichen Zeitpunkten auf die Fertigstellung einer bestimmten Teilmenge der Tasks zu warten, ist die hier diskutierte Fragestellung aber von entscheidender Bedeutung. Denn dies ist bei threadbasierter Programmierung ohne komplexere Task-Objekte nicht möglich, wenn nicht anderweitig zu jedem Zeitpunkt bekannt ist, welcher Thread welches Funktionsobjekt ausführt. Auch ermöglicht die taskbasierte Programmierung bei gleichzeitiger Verwendung eines Threadpools erst eine Implementierung von geschachtelter Parallelität, da Deadlock-Situationen kaum zu vermeiden wären, wenn `Thread`-Instanzen gegenseitig aufeinander warten müssten.

Nachdem die Implementierung zunächst threadbasiert begonnen wurde, wurde sie im Laufe der Entwicklung aus den genannten Gründen und wegen der höheren Flexibilität in eine taskbasierte geändert. Es gelang dabei, trotz der nun für jedes `Task`-Objekt zusätzlich zu konstruierenden `Mutex`- und `ConditionVariable`-Objekte, keinen messbaren zusätzlichen Overhead hinnehmen zu müssen, obwohl die Thread-eigenen Objekte wesentlich effizienter im Cache gehalten werden können, da sie über die gesamte Laufzeit existieren. Der Grund liegt im geringeren Synchronisationsaufwand, denn im Gegensatz zu einem Thread, der während der gesamten Ausführung des Funktionsobjekts den Lock auf sein `Mutex`-Objekt inne haben muss, um zu verhindern, dass das auszuführende Funktionsobjekt ausgetauscht wird, wird der `Task`-eigene Mutex kaum benötigt und steht beim Aufruf von `wait()` in der Regel sofort zur Verfügung. Im Zuge der Implementierung haben sich drei mögliche Ansätze herauskristallisiert, um die Verteilung von Tasks und das Warten auf ihre Fertigstellung zu realisieren:

- Jeder Task wird per `Thread::run()` explizit einem Thread zugewiesen und anschließend wird auf ihn gewartet.
- Tasks werden nicht zugewiesen, sondern nur in der Taskliste abgelegt. Anschließend werden alle Threads aufgeweckt und entnehmen selbstständig Tasks aus der Taskliste.
- Tasks werden per `Thread::run()` zugewiesen oder in einer Taskliste abgelegt. Threads überprüfen nach Ende der Ausführung ihrer Aufgabe selbstständig ob weitere Arbeit verfügbar ist.

Wie die Auflistung der `ThreadPool`-Methoden bereits zeigte, ist sowohl das erste, als auch das dritte Verfahren implementiert. Das zweite Verfahren erlaubt sehr minimalisti-

sche Implementierungen von `Thread` und `ThreadPool`, wodurch für einzelne Funktionen und gewisse Eingaben geringe Geschwindigkeitsvorteile erzielt werden, die aber beachtlichen Verlusten in anderen Fällen gegenüberstehen. Denn dieses Verfahren ist wesentlich unflexibler und erlaubt eine deutlich geringere Anpassung an die jeweilige Architektur. Der Grund dafür ist, dass durch das Aufwecken aller Threads mittels der Funktion `pthread_cond_broadcast()` nicht mehr beeinflusst werden kann, welcher Thread bzw. Kern die Ausführung eines Tasks übernimmt. Dies ist auch ein Problem der aktuellen OpenMP-Implementierung, welche ebenfalls nur alle im Pool wartenden Threads aufwecken kann, nicht jedoch bestimmte einzelne [21]. Sollen z. B. nur drei von fünf existierenden Threads verwendet werden, müssten zunächst zwei aus dem Pool entfernt werden, d. h. terminieren. Mit einer solchen Strategie ist folglich die gezielte Wiederverwendung von Threads nicht realisierbar. Ein wesentlicher Teil der Vorteile, die im Rahmen dieser Diplomarbeit aus dem Konzept der Thread Affinity gezogen werden und die optimale Unterstützung verschiedener Algorithmen durch mehrere Varianten des Dispatchings und Wartens, wie in Abschnitt 3.1.2 beschrieben, wäre damit hinfällig. Das dritte Verfahren kombiniert eine zunächst statische Zuweisung von Tasks an Threads mit dem großen Vorteil des zweiten Ansatzes, dass durch selbstständiges Abholen weiterer Tasks eine Art automatisches Load Balancing stattfindet. Einfache, linear auf dem Speicher arbeitende Algorithmen können also weiterhin statisch partitioniert werden. Bei komplexeren Algorithmen, die mehr Tasks erzeugen als Threads vorhanden sind, werden diese jedoch sinnvoll dynamisch verteilt. Die Implementierung steht damit dem Work Stealing der MCSTL kaum nach, da diese ebenfalls zunächst statisch Aufgaben verteilt und lediglich anschließend weitere Tasks von zu stark belasteten Threads stiehlt, ohne aber einen einmal zur Ausführung gebrachten Task zu unterbrechen [45]. Bei statischer Lastverteilung erfolgt die Partitionierung anders als bei der MPTL und ähnlich wie bei der MCSTL a priori durch den Hauptprozess. Dies hatte in experimentellen Messungen, gegenüber einer Berechnung des jeweils zu bearbeitenden Teilintervalls durch die Threads im Zuge der parallelen Ausführung, keinen messbaren Overhead zur Folge.

3.1.4 Thread Affinity

Nicht nur für die effektive Ausnutzung von Caches und um sicherzustellen, dass wirklich alle (gewünschten) Kerne an der Ausführung beteiligt werden, spielt Thread Affinity eine Rolle, sondern bereits bei der Entscheidung, auf welchem Prozessorkern der Hauptprozess laufen soll. Nachdem der Hauptprozess seine Threads erzeugt hat, fährt er fort und beansprucht weiterhin CPU-Zeit, da etwa die Partitionierung und die gesamten Aufgaben des `ThreadPool`s von ihm übernommen werden. Es hat sich durch viele Experimente nachweislich und reproduzierbar herausgestellt, dass es ineffizient ist, wenn der Hauptprozess und ein Thread auf demselben Kern laufen.

Denn unabhängig davon, welche der drei beschriebenen Designentscheidungen zum Zuweisen von und Warten auf Tasks gewählt wird, muss es immer einen Zeitpunkt geben, an dem schlafende (wartende) Threads vom Hauptprozess aufgeweckt (signalisiert) werden. Erfolgt die Signalisierung, wacht der Thread auf und wird versuchen, den Mutex zu locken, während diesen aber noch der Hauptprozess inne hat. Erst nach dessen Freigabe kann der Thread fortfahren. Dieser Vorgang, nämlich das Zusammenspiel aus Signalisierung, Freigabe und erneutem Locken des `Mutex` ist signifikant teurer, wenn beide Prozesse auf demselben Kern laufen. Jeder von ihnen muss mindestens ein Mal vom Scheduler mit CPU-Zeit versehen werden, weshalb die Kosten eines Kontextwechsels anfallen. Wird der Hauptprozess darüber hinaus unterbrochen, bevor er den Mutex freigegeben hat, kann der Thread seine Zeitscheibe nicht nutzen. Während der Entwicklungsphase sind darüber hinaus noch weitere laufzeitkritische Probleme aufgetreten, deren anschließende Untersuchung an dieser Stelle erläutert wird. Im Fall des entwickelten Konzepts ist die Methode `Thread::run(Task)` diejenige, in der die beschiebenen Vorgänge erfolgen.

Nehmen wir nun also an, ein Thread befindet sich im wartenden Zustand. Dann hat er in jedem Fall folgende Instruktionen ausgeführt.

```
pthread_mutex_lock(mutex);
pthread_cond_wait(mutex);
```

Nachdem der Thread in den Wartezustand übergegangen ist, wird der im ersten Schritt erfolgte Lock des Mutex zur Inanspruchnahme eines weckenden Prozesses wieder freigegeben und der Thread zunächst nicht mehr vom Scheduler berücksichtigt. Sei nun der Hauptprozess in der Funktion `Thread::run(Task)`, deren Laufzeit in einer experimentellen Messung durch die Messpunkte a und b, wie in Quelltextauszug 3.1 angedeutet, ermittelt wird. Bevor er die ConditionVariable signalisiert, muss der Hauptprozess den Lock auf den Mutex besitzen, um einen reproduzierbaren korrekten Ablauf des Programms zu gewährleisten. Da dieser freigegeben ist, stellt dies zunächst noch kein Problem dar. Für den untersuchten Fall ist der weitere Synchronisationsprozess dennoch mit hohen Kosten verbunden, wie die folgende Abbildung 3.2 für die Laufzeit der Methode `Thread::run()` bei Übergabe eines Tasks mit der Funktion `std::accumulate()` für verschiedene Eingabegrößen und zwei Threads auf einem Intel Core 2 Duo T5450 Prozessor zeigt. Verglichen wird die Latenz für den Fall, dass der Hauptprozess und einer der beiden Threads auf demselben Kern laufen (hier bezeichnet als ineffektives Scheduling) mit der Latenz, wenn beide Threads auf unterschiedlichen Kernen laufen (hier bezeichnet als effektives Scheduling). Es ist ersichtlich, dass beim effektiven Scheduling die Latenz im Bereich von Messtoleranzen konstant bleibt und somit unabhängig von der Eingabegröße kaum Overhead verursacht. Im Fall des ineffektiven Schedulings hingegen steigt die Latenz mit zunehmender Eingabegröße stark an und führt vor allem bei wenigen verfügbaren Kernen zu geringerem oder ausbleibendem Speedup. Dass die Latenz mit der Eingabegröße korreliert, lässt

```
1  void Thread::run(Task * task)
2  {
3      // Messpunkt a
4      pthread_mutex_lock(mutex);
5      // Weitere Instruktionen
6      pthread_cond_signal(mutex);
7      // Weitere Instruktionen
8      pthread_mutex_unlock(mutex);
9      // Messpunkt b
10 }
```

Quelltextauszug 3.1: Zeitnahme für die Funktion `Thread::run()`

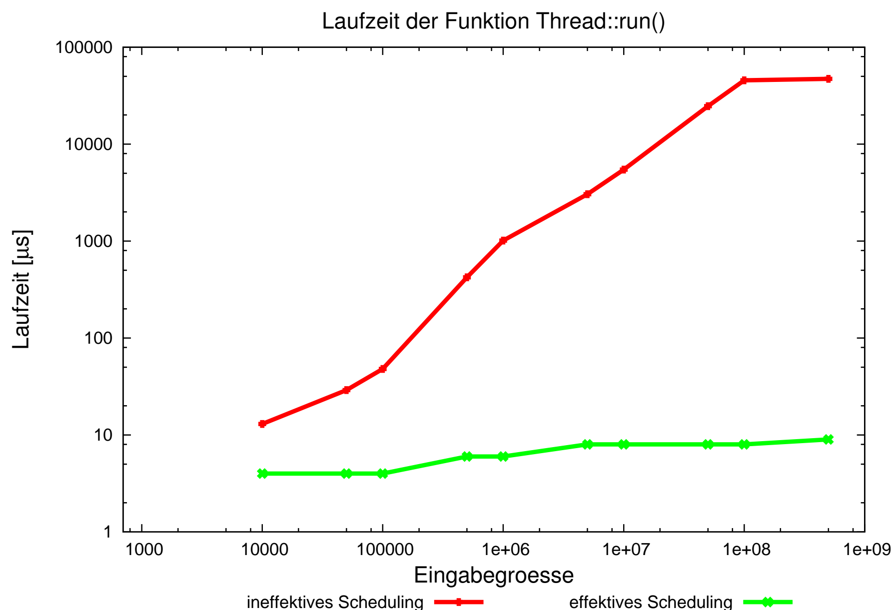

Abbildung 3.2: Overhead beim Aufwecken eines Threads (Intel Core 2 Duo T5450)

zunächst darauf schließen, dass neben Synchronisationseffekten auch mit dem Kontextwechsel verbundene Migrations- und Speichertransferoperationen eine Rolle spielen. Mit dem Hilfsprogramm *cachegrind* [1] konnten bei einer Eingabegröße von $5 \cdot 10^7$ Fließkommazahlen lediglich 240 (1,5 ‰) zusätzliche und damit keine signifikant erhöhte Anzahl *Cache Misses* nachgewiesen werden. Selbst bei konservativer Berechnung mit einem *Cache Miss Penalty* von 300 Taktzyklen führen 240 cache misses lediglich zu einer Verzögerung von $300 \cdot 240 = 72000$ Zyklen. Da die Taktfrequenz des Core 2 Duo Prozessors 1,66 GHz beträgt, werden in einer Mikrosekunde 1666 Zyklen bewältigt. Die Latenz beträgt also

[1] *cachegrind* ist ein Programm zur Messung von cache misses und Teil der freien Profiler-Software *valgrind*, beziehbar über `http://valgrind.org`

$\frac{72000}{1666} \simeq 43\ \mu s$. Die real gemessene Latenz zwischen beiden Fällen beträgt aber 5461 μs, wie Abbildung 3.2 zeigt. Demnach können Cache Misses nicht die Ursache für den gemessenen Overhead sein. Nachfolgende Experimente mit Hilfe von Breakpoints führten zu dem Ergebnis, dass die Freigabe des Mutex mittels `pthread_mutex_unlock()` der Schwachpunkt bei der Ausführung auf dem gleichen Kern ist. In vielen Fällen tritt die folgende problematische Situation ein. Die Funktion `pthread_cond_signal()` ist zurückgekehrt, der Thread wurde aufgeweckt und ihm die CPU zugeteilt, sodass er den durch das Funktionsobjekt übergebenen Tasks ausführt. Zu diesem Zeitpunkt hat der Hauptprozess aber die Funktion `pthread_mutex_unlock()` noch nicht beendet. Er konnte vor Ende seiner CPU-Zeit lediglich den Teil von `pthread_mutex_unlock()` ausführen, der ausreicht, damit der Mutex vom Thread wieder gelockt werden kann. Die Methode selbst wurde jedoch während der Ausführung weiterer interner NPTL-Funktionen, wie etwa `__lll_mutex_unlock_wake()`, `__pthread_mutex_unlock_usercnt()` und `__lll_unlock_wake()`, unterbrochen. Je nach Scheduling und Eingabegröße führt der Thread in diesem Fall das ihm zugewiesene Funktionsobjekt vollständig aus, bevor der Hauptprozess wieder die CPU erhält und die Methode `Thread::run()` beenden kann. Das Ergebnis ist folglich eine sequentielle Ausführung und dies ist der Grund für die in Abbildung 3.2 gezeigte Verzögerung. Die negativen Folgen sind noch größer, wenn der hier betrachtete Thread nicht der letzte mit Arbeit zu versehende ist. Wenn dies zutrifft und wie durch das nachfolgende Beispiel angedeutet, alle Threads durch eine Schleife mit Tasks versehen werden, kann dies für alle folgenden Threads erst geschehen, nachdem der betrachtete Thread bereits seine Aufgabe vollständig beendet hat.

```
1 for (unsigned i(0) ; i < num_threads ; ++i)
2 {
3     pool->wait_and_run(tasks[i]); // ruft Thread::run(tasks[i]) auf
4 }
```

Quelltextauszug 3.2: Iteratives Dispatching mittels der Klasse `ThreadPool`

Es ist also essentiell wichtig, dass möglichst zu keinem Zeitpunkt der Hauptprozess und einer seiner Threads auf demselben Kern zur Ausführung kommen. Dies ist ausschließlich mit Thread Affinity zuverlässig zu erreichen, denn experimentelle Versuche haben gezeigt, dass beispielsweise der Linux Scheduler ohne ihren Einsatz, für die hier hauptsächlich betrachteten kleineren Problemgrößen, häufig ein ineffektives Scheduling wählt, oder sogar alle Threads zunächst auf einem Kern zur Ausführung bringt. In Folge dessen macht es Sinn, bei der Partitionierung eines Problems für beispielsweise ein Vierkernsystem lediglich drei Threads mit Aufgaben zu versehen und den Hauptprozess mit dem vierten Teil zu beauftragen, ohne dafür einen Thread aufzuwecken. Die Aufgabe des Hauptprozesses sollte darüber hinaus die letzte Partition der Nutzdatensequenz sein, da für sie die Wahr-

scheinlichkeit, dass sie vollständig oder zum Teil im Cache verblieben ist, am Größten ist. Dies ist die Vorgehensweise für alle im Rahmen der vorliegenden Arbeit implementierten Algorithmen.

3.1.5 Vermeidung häufiger Fehler

Dieser Abschnitt behandelt typische Speedup einschränkende oder verhindernde Fehler, die häufig in Verbindung mit parallelen Algorithmen auftreten und erläutert, auf welche Art und Weise das im Rahmen dieser Arbeit entwickelte Konzept ihnen begegnet.

False sharing. Dieser Begriff beschreibt den Fall, dass zwei oder mehr Threads auf unterschiedliche Datenworte innerhalb einer Cacheline zugreifen. Dies führt bei jedem Zugriff zum Zurückschreiben ihres Inhaltes in den Hauptspeicher und zur nachfolgenden Aktualisierung der anderen Caches, die ebenfalls Einträge für diesen Hauptspeicher-Block besitzen, um die Kohärenz des Cache-Systems sicherzustellen.

Um diesen Effekt zu vermeiden, müssen von mehreren Prozessen manipulierte Daten an entsprechenden Cacheline-Grenzen ausgerichtet (*aligned*) werden. Dies geschieht im Rahmen des entwickelten Konzepts mit Hilfe der Funktion `posix_memalign()`. Im Fall von x86-Systemen beträgt die Cacheline-Größe in den meisten Fällen 64 Bytes. Sie wird bei Konstruktion der `ThreadPool`-Instanz mit Hilfe des Assemblerbefehls `cpuid` ausgelesen.

Zusätzlich zu zwischen Threads und Prozessen geteilten Objekten wie Mutexes, ConditionVariables oder Ergebnisvariablen, bezieht sich das Problem auch auf die Eingabe. Für diese kann jedoch nicht, oder nur praxisfern, sicher gestellt werden, dass sie an Cacheline-Grenzen ausgerichtet ist. Deshalb partitionieren die im Rahmen dieser Diplomarbeit entwickelten Algorithmen die Eingabe so, dass alle an Threads übergebenen Teilprobleme mit Ausnahme des ersten, an einer Cacheline-Grenze beginnen, also aligned sind. Aus diesem Grund können die Teilgrößen für die verschiedenen Threads auch bei statischer Partitionierung leicht voneinander abweichen. Da im Fall von 32- bzw. 64-bit Operanden aber lediglich 16 respektive 8 solche Werte in eine 64-Byte-Cacheline passen, ist diese Abweichung nicht signifikant mit Hinblick auf das Load Balancing und damit auch nicht in Bezug auf die Ausführungszeiten der verschiedenen Threads.

Hot and cold cache. Ein weiterer wichtiger Aspekt bei der Entwicklung von paralleler Software ist die Rücksichtnahme auf den Cache. Die effektive Wiederverwendung von Daten sollte durch das Scheduling der Threads unterstützt und nicht erschwert werden. Wird keine Thread Affinity verwendet, kann der Betriebssystem-Scheduler unterschiedliche Threads auf einem Kern zur Ausführung bringen. Arbeiten diese auf unterschiedlichen Daten, muss der jeweilige Thread seine eigenen Daten aus dem Hauptspeicher laden und damit die Daten des anderen überschreiben. Kommt dieser im Anschluss wieder zur Ausführung, tritt dasselbe Problem auf und jegliche Wiederverwendung von Daten wird unmöglich.

Insofern wirkt das Einschalten von Thread Affinity diesem Problem entgegen. Die Verwendung spezieller threadgebundener Dispatch- und Wartefunktionen (siehe Abschnitt 3.1.2) kann ebenfalls die geschickte Wiederverwendung von noch „heißen" Caches unterstützen. Dies wird auch am Beispiel des Algorithmus `partial_sum()` in Abschnitt 3.3.3 erläutert.

Time slicing and mutex locks. Eng verwandt mit dem vorherigen Problem ist auch das Problem des *time slicing*s. Wie bereits zuvor beschrieben, kann der Scheduler unterschiedliche Threads auf einem Kern zur Ausführung bringen, wenn Thread Affinity nicht verwendet wird. Jedoch kann dies auch bei eingeschalteter Affinität passieren, nämlich dann wenn mehr Threads erzeugt wurden, als Kerne existieren. Wie in Abschnitt 3.1.2 beschrieben, werden deshalb vom Threadpool nur so viele Threads erzeugt, wie logische Ausführungseinheiten vorhanden sind, falls kein anderer Parameter angegeben wird.

Ein anderes Problem im Zusammenhang mit dem Ende einer Zeitscheibe tritt auf, wenn ein Thread zu diesem Zeitpunkt noch einen Lock auf einen Mutex besitzt. In diesem Fall erfolgt keine Freigabe des Mutex, weshalb andere Threads möglicherweise ihren kritischen Abschnitt nicht betreten und somit die ihnen nun zugewiesenen Zeitscheiben nicht nutzen können. Es folgt unmittelbar, dass `Mutex`-Objekte so kurz wie möglich gelockt werden sollten.

3.2 Speedup-begrenzende Faktoren

In der Entwicklungsphase hat sich sehr schnell gezeigt, dass für wenige Threads in der Regel eine gute Skalierung erzielt werden konnte, mit mehr Threads der Laufzeitgewinn aber auch bei großen Eingaben stagniert. Deshalb wurde versucht, die Gründe für diesen Effekt zu ermitteln. Der Speedup durch parallele Programme kann durch viele Aspekte beschränkt werden. Einige mögliche Gründe, wie z.B. die Existenz von Datenabhängigkeiten oder der Einfluss von Cache-Misses und des Betriebssystem-Schedulers wurden bereits in Abschnitt 1.1 genannt. Andere typische und relativ einfach vermeidbare Fehler, die zu geringer Skalierung führen können, wurden im letzten Abschnitt 3.1.5 behandelt. In diesem Abschnitt erfolgt insbesondere eine experimentelle Untersuchung der Begrenzung des Speedups durch das Speichersystem.

3.2.1 Architekturen und Einfluss des Speichersystems

Es ist bekannt, dass sich die Geschwindigkeit der Speicheranbindung gegenüber der Taktfrequenz der Prozessoren in der Vergangenheit deutlich langsamer entwickelte. Die resultierende Lücke wird häufig als *Memory Gap* bezeichnet. Eine Diskussion dieses Phänomens aus technischer und industrieller Sicht bieten McKee in [37] und Wilkes in [51]. Um die hohen Latenzen eines Speicherzugriffs in möglichst vielen Fällen vermeiden zu können, ist

eine effektive Cache-Hierarchie erforderlich. Ein Cache Miss im letzten Cache-Level hat zur Folge, dass die Ausführungspipeline des Prozessors angehalten werden muss (*stall*), bis die nachfolgende Speicheranfrage bearbeitet wurde. Dieser Vorgang kann mehr als 200 CPU-Taktzyklen benötigen. Wenn asynchroner Zugriff auf den Speichercontroller besteht, dann ist es möglich, Phasen der Berechnung und das Laden der für den nachfolgenden Berechnungsschritt benötigten Daten zeitlich zu überlappen (*double buffering*). Während dies beispielsweise beim Cell Prozessor ([34]) der Fall ist, wird dem Programmierer gängiger PowerPC-, SPARC- oder x86-Architekturen dieser Zugriff verwehrt.

Die meisten Algorithmen der STL führen lediglich wenige arithmetische Operationen auf den zu bearbeitenden Daten aus. Sie sind als *memory bound* zu bezeichnen, da somit ihre Laufzeit im Wesentlichen von der Geschwindigkeit des Datentransfers abhängt. Im Gegensatz dazu bearbeiten *compute bound*-Algorithmen einmal geladene Daten so lange, dass die Laufzeit durch diese Operationen dominiert wird. Bei der üblichen Verwendung von sequentiell allozierten Containern liegt eine Eingabe für einen parallelen Algorithmus nur in den Caches des Prozessors oder Kerns vor, der den Hauptprozess ausführt. Handelt es sich dabei in keinem Fall um einen shared Cache, besteht für alle anderen Kerne, die nachfolgend an der parallelen Ausführung beteiligt sind, eine vollständige Out-Of-Cache Situation. Die Geschwindigkeit der Ausführung auf diesen Kernen hängt daher maßgeblich davon ab, wie schnell die Daten aus anderen Caches oder dem Hauptspeicher transferiert werden können. Dies wiederum wird von der Architektur des Speichersystems bestimmt. Die Laufzeit des langsamsten Kerns ist dabei eine triviale untere Schranke für die Laufzeit des gesamten parallelen Algorithmus.

Das linke Modell in Abbildung 3.3 zeigt den schematischen Aufbau einer üblichen Uniform Memory Architecture, wie sie in gängigen Single- und Multicore-Prozessoren Realität ist. Bei allen abgebildeten Modellen wird von der Verwendung der als *Dual Channel* vermarkteten Speicherkonfiguration ausgegangen. Dabei werden mehrere Speicherbänke besetzt und die Blöcke eines kontinuierlichen Datenstromes abwechselnd (*interleaved*) auf den entsprechenden RAM-Bausteinen gespeichert. Folglich können aufeinanderfolgende Daten anschließend simultan von mehreren Speicherbänken ausgelesen werden.

Die Verbindung eines oder mehrerer Prozessoren mit dem Hauptspeicher geschieht in den meisten Fällen über einen einzigen Bus, z. B. den bereits in Abschnitt 1.1 erwähnten Front Side Bus. Alle Speichertransfers müssen über diesen Bus und den Speichercontroller (MC bzw. MCH) abgewickelt werden. Es ist offensichtlich, dass dieses Konzept nicht für beliebig viele Prozessoren skaliert, da eine steigende Anzahl Busteilnehmer zu immer mehr simultan zu verarbeitenden Speichertransferbefehlen führt. Bei Existenz vieler Kerne auf einem Prozessor kann die Anbindung daher ebenfalls zum Flaschenhals werden.

Für Hochleistungsserver versucht beispielsweise Intel dieses Problem abzumildern, indem zwei Front Side Busse zur Verfügung gestellt werden. Abbildung 3.3 zeigt, wie ein solches System aufgebaut ist. Auch das im Rahmen dieser Arbeit verwendete Xeon-System

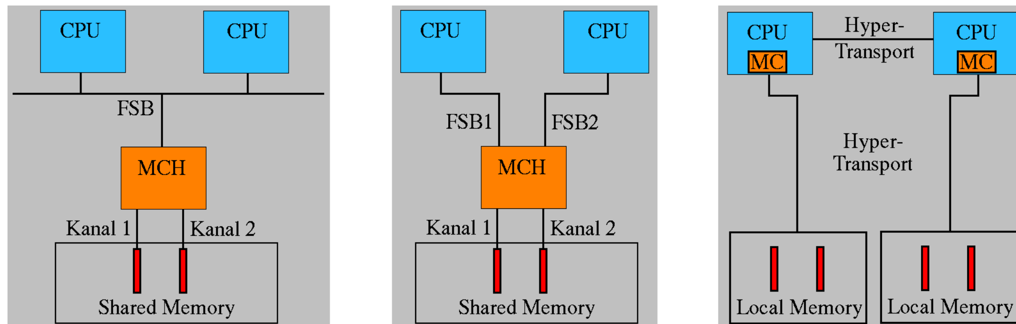

Abbildung 3.3: Schematischer Aufbau einer Uniform Memory Architecture (links), einer UMA im Fall eines Dual-Xeon Systems mit Dual Independent Bus (Mitte) und einer Non-Uniform Memory Architecture mit HyperTransport als Bussystem (rechts).

basiert auf dieser Architektur. Der zusätzliche Bus verdoppelt die theoretische Bandbreite, letztendlich bleibt es aber bei einem zentralen Speichercontroller, der alle ankommenden Anfragen bearbeiten muss. Auch deshalb handelt es sich nach wie vor um eine Uniform Memory Architecture, da die Adressierung des gesamten Speicherbereichs über beide Busse mit den gleichen Kosten verbunden ist.

Anders ist dies bei einer Non-Uniform Memory Architecture, wie im rechten Modell der Abbildung 3.3 gezeigt und durch das hier verwendete Opteron-System vertreten. Jeder Prozessor besitzt seinen eigenen Speichercontroller, welcher jeweils eigene Bandbreite und Bearbeitungskapazität bereitstellt. Folglich hat jeder Prozessor eine eigene Anbindung zum Hauptspeicher, weshalb dieses Architektur-Konzept auch auf sehr große Systeme skaliert, bei vielen Kernen auf einem Chip aber ebenfalls beschränkend wirken kann. Arbeitsspeicher, der von einem Speichercontroller alloziert wurde, muss auch über diesen von anderen Prozessoren adressiert werden. Aus diesem Grund ist der Zugriff auf „fremde“ Speicherbereiche teurer. Wie der nachfolgende Abschnitt zeigt, muss auf diese Architekturunterschiede reagiert werden, wenn parallele Algorithmen optimalen Nutzen aus der vorhandenen Hardware ziehen sollen. Insbesondere für eine Bibliothek ist dies daher von großer Bedeutung.

3.2.2 Experimentelle Untersuchungen

Wie bereits erwähnt, wurde in der Entwicklungsphase festgestellt, dass der Speedup in vielen Fällen auch bei großen Eingaben durch den Einsatz weiterer Threads nicht mehr gesteigert werden kann. Die entsprechenden Messungen um herauszufinden, weshalb die Laufzeit der Algorithmen auf diese Weise und die einhergehende stärkere Partitionierung nicht mehr verringert werden konnte, führten zu Ergebnissen, die nicht den Erwartungen entsprechen. Wenn die gleichen (Cache-)Voraussetzungen vorliegen und Seiteneffekte keine Rolle spielen, erwartet man bei linear auf dem Speicher arbeitenden Algorithmen für eine bestimmte Eingabegröße stabile Laufzeiten für die Ausführung der Funktion allein. Man

erwartet außerdem einen konstanten Overhead für die Thread-Synchronisation und die Problempartitionierung, der sich jedoch nicht in der Laufzeit der jeweiligen Funktion selbst niederschlägt. Genau eine solche Varianz in der Laufzeit ist aber ab einer gewissen Eingabegröße und für eine steigende Anzahl Threads auf den hier betrachteten x86-Systemen zu beobachten.

Um dies zu visualisieren wurde folgender Test mit der Funktion `std::accumulate()` und Eingabegrößen aus dem Intervall $[10000, 5 \cdot 10^6]$ konstruiert. Für jeden beteiligten Thread wird die Laufzeit der Funktion für die ihm zugewiesenen Partition ermittelt. Der Test wird für jede Eingabegröße 100 mal wiederholt und immer auf neu konstruierten Eingaben ausgeführt, um das realistische Szenario zu erhalten, dass die Eingabe lediglich einmalig durch ihre Erzeugung vollständig oder teilweise im Cache verbleibt. Als Referenz wird die benötigte Zeit für die sequentielle Ausführung der STL-Funktion bei ebenfalls 100-maliger Ausführung herangezogen. Abbildung 3.4 zeigt die Ergebnisse beispielhaft für den dritten Kern der ersten CPU des Xeon-Systems, während der Hauptprozess auf Kern 7 (zweite CPU) ausgeführt wird und somit Transferoperationen über den Front Side Bus erforderlich sind. Wenn in diesem Zusammenhang vereinfacht von der Laufzeit eines Prozesses, Threads, oder Kerns gesprochen wird, so ist damit die Dauer der entsprechenden Ausführung von `std::accumulate()` gemeint, da die Threads, wie in Kapitel 3.1 erläutert, nach Abschluss eines parallelen Abschnitts noch nicht terminieren.

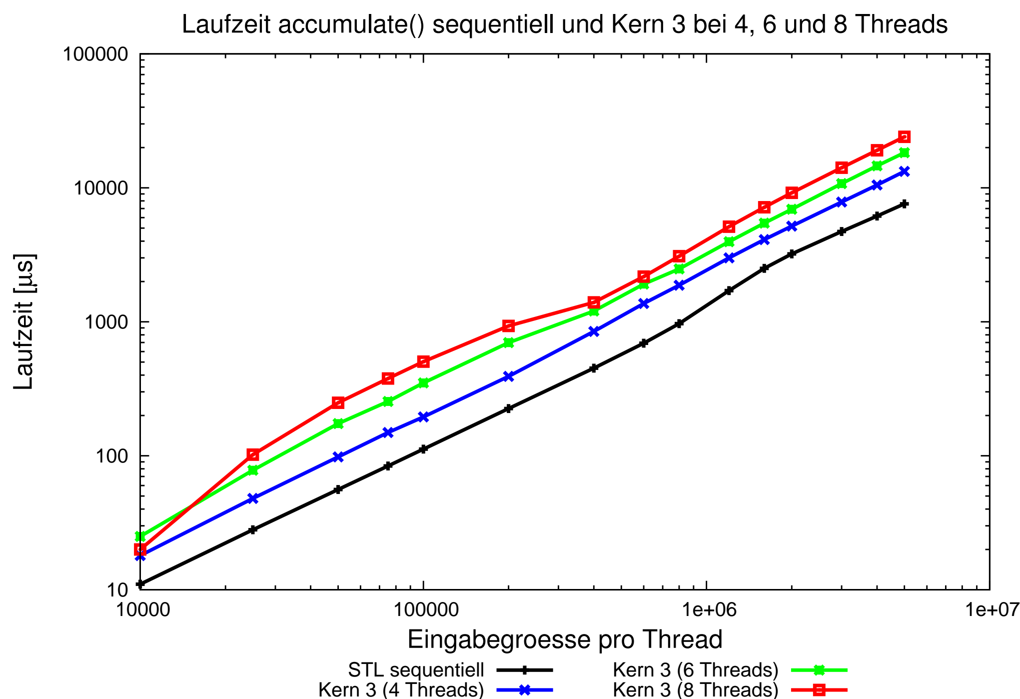

Abbildung 3.4: Xeon-System: Laufzeitunterschiede für Aufrufe von `std::accumulate()` auf Kern 3 bei insgesamt 4, 6 und 8 verwendeten Threads

Wie zu sehen ist, benötigt ein und derselbe Kern für den gleichen Aufruf von `std::accumulate()` umso länger, je mehr Threads aktiv sind. Die Laufzeit ist außerdem deutlich höher als bei rein sequentieller Ausführung. Ersteres ist dadurch zu erklären, dass die höhere Anzahl aktiver Threads, die alle gleichzeitig die zu verarbeitenden Daten anfordern, eine wesentlich höhere Aktivität auf dem Front Side Bus zur Folge hat. Der zweite Effekt korreliert mit dem ersten, denn im Gegensatz zur sequentiellen Ausführung liegt die Eingabe (oder ein Teil von ihr) für den hier betrachteten Kern 3 bei Beginn der Ausführung nicht bereits im Cache, wie es beim Hauptprozess (auf Kern 7) der Fall ist, da dort die Eingabe erzeugt wurde. In weiteren Abbildungen wird sich zeigen, dass der Hauptprozess für In-Cache Größen daher auch keinerlei Einbußen gegenüber der sequentiellen Laufzeit zu verzeichnen hat. Zunächst folgen jedoch die Ergebnisse für das Opteron-System (Abbildung 3.5). Solange sich die Ausführung auf eine CPU (Kern 2 und 3) beschränkt, ist

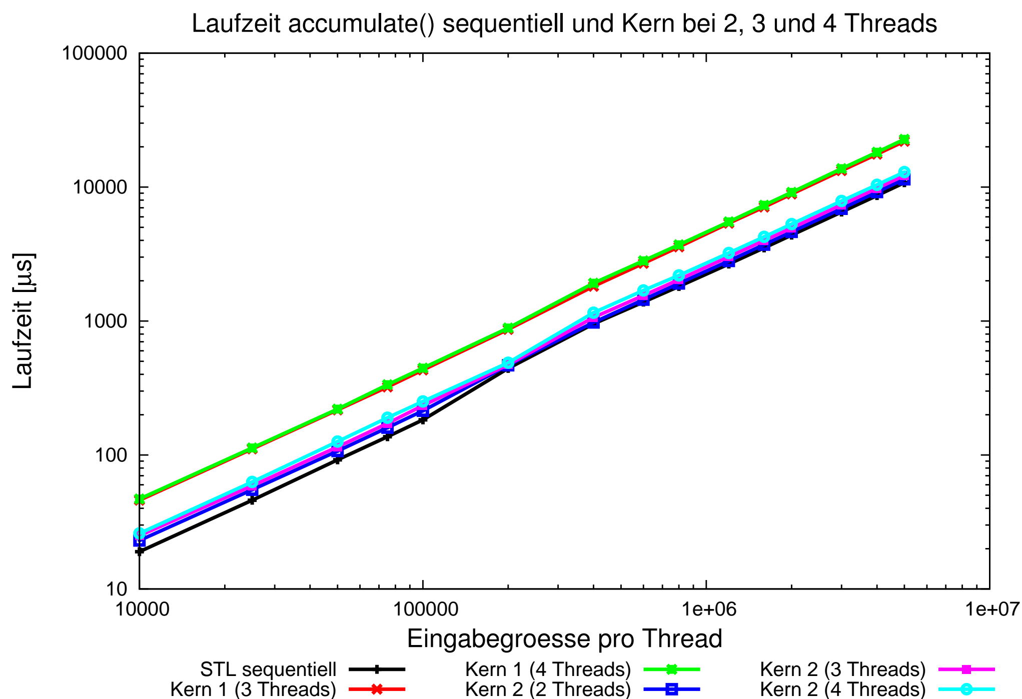

Abbildung 3.5: Opteron-System: Laufzeitunterschiede für Aufrufe von `std::accumulate()` bei insgesamt 2, 3 und 4 verwendeten Threads

kaum Overhead gegenüber der sequentiellen Ausführung zu verzeichnen. Wird die andere CPU (hier Kern 1) verwendet, treten deutlichere Unterschiede in der Laufzeit auf, da hier ebenfalls die Nutzdaten über das Bussystem übertragen werden müssen. Die Laufzeit des Threads auf Kern 2 ist darüber hinaus immer ein wenig höher als die des Hauptprozesses (auf Kern 3), da sie auf getrennte Caches zugreifen.

Ein weiteres Ergebnis dieser Versuche ist, dass sich auch die Laufzeit der Threads untereinander z. T. deutlich unterscheidet, obwohl die Eingabegröße für alle identisch ist.

Abbildung 3.6 zeigt diesen Zusammenhang beispielhaft für den Fall der Ausführung mit acht Threads auf dem Xeon-System.

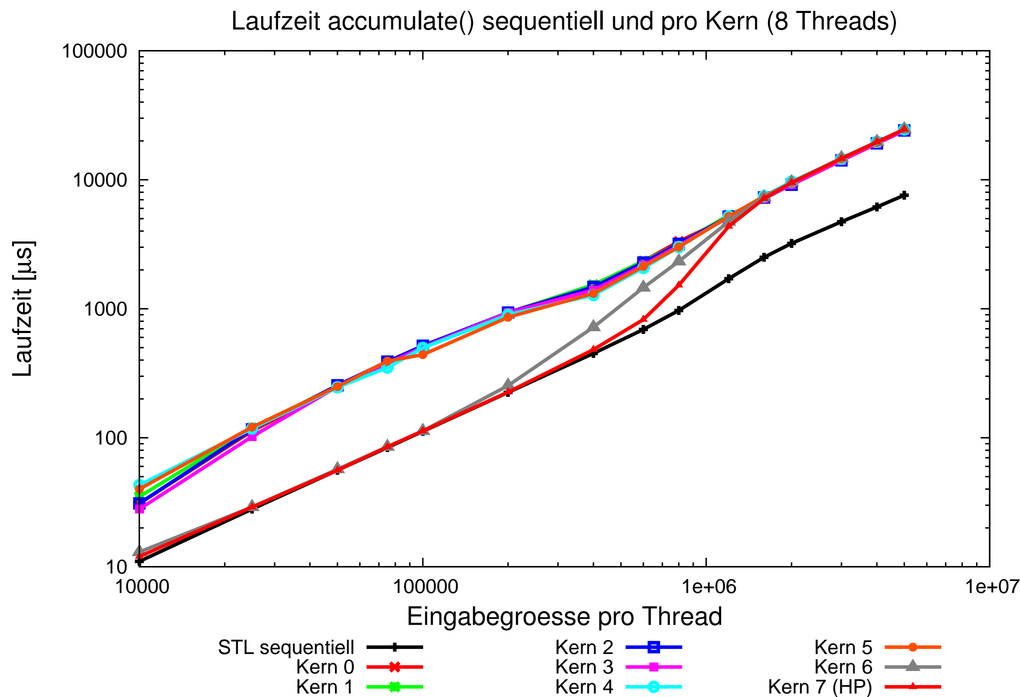

Abbildung 3.6: Xeon-System: Laufzeitunterschiede unterschiedlicher Threads für Aufrufe von `std::accumulate()` bei insgesamt 8 Threads

Es ist deutlich zu erkennen, dass wie oben bereits vorweg genommen, der Hauptprozess für In-Cache Größen keinerlei Laufzeiteinbußen gegenüber der rein sequentiellen Ausführung verzeichnet. Auch Kern 6, der sich mit dem Hauptprozess den Cache teilt, kann lange Zeit davon profitieren, dass die Eingabe für ihn sofort zugreifbar ist. Beiden Kernen stehen theoretisch 6 MB L2-Cache zur Verfügung, es könnten also bis zu $1,5 \cdot 10^6$ Fließkommazahlen einfacher Genauigkeit (d.h. vom Typ `float`) dort abgelegt werden. Da jeweils zwei Threads auf einen shared Cache zugreifen, liegt die Schwelle also bei $7,5 \cdot 10^5$ Elementen. Zusätzlich werden auch andere Nutzdaten und Instruktionen benötigt, so dass der in Abbildung 3.6 sichtbare Bereich, in dem sich die Laufzeiten von Kern 6 und 7 von der sequentiellen Laufzeit abheben, tatsächlich mit der Cachegröße korreliert. Die anderen Threads haben aus den soeben beschriebenen Gründen grundsätzlich eine höhere Laufzeit. Ab einer Eingabegröße von ca. $1,5 \cdot 10^6$ besteht für die Kerne 6 und 7 dann die gleiche vollständige Out-Of-Cache Situation wie für die anderen Kerne, sodass sich die Laufzeiten angleichen.

Für das Opteron-System zeigt sich diesmal ein sehr ähnliches Bild, mit dem einzigen Unterschied, dass die Laufzeiten der Kerne 2 und 3 von Beginn an nicht identisch sind, da sie keinen shared Cache besitzen. Die Kosten des chip-internen Kopierens sind jedoch deut-

lich geringer, als der Datentransport über das Bussystem zur anderen CPU, wie Abbildung 3.7 verdeutlicht.

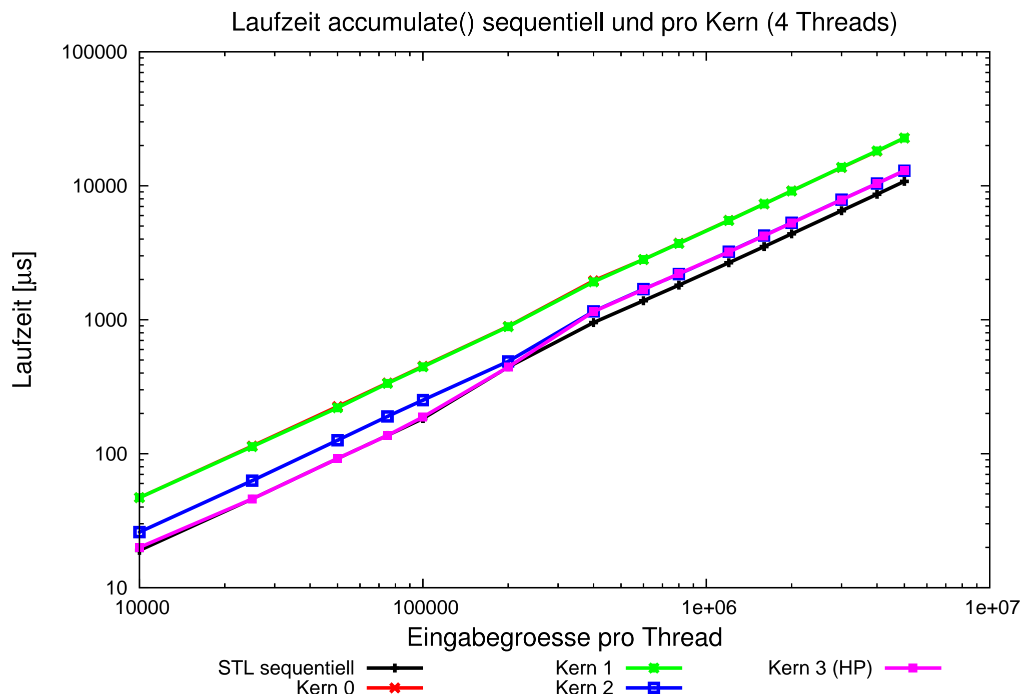

Abbildung 3.7: Opteron-System: Laufzeitunterschiede unterschiedlicher Threads für Aufrufe von `std::accumulate()` bei insgesamt 4 Threads

Die beschriebenen Effekte zeigen also, dass die mit steigender Anzahl Threads größer werdenden Latenzen durch die stärkere Auslastung des Speichersystems zu begründen sind. Dass ein identischer Aufruf einer numerischen STL-Funktion länger dauert, kann nur dadurch erklärt werden, dass für den Prozessor höhere Wartezeiten auf Speichertransfers bestehen. Mit steigender Anzahl an aktiven Bus-Teilnehmern, die alle simultan ihre Partition der zu bearbeitenden Daten anfordern, begrenzt der Flaschenhals Speichersystem folglich den erzielbaren Speedup. Gegenüber der sequentiellen Ausführung können also auch mit vielen Threads noch Gewinne erreicht werden. Eine lineare Skalierung ist jedoch bei memory bound Algorithmen, bei denen die Threads simultan die gleiche Menge an Daten anfordern, nicht zu erwarten, da die Zeit, die der Hauptprozess auf die anderen Threads nach Fertigstellung seiner eigenen Aufgabe noch warten muss, mit der Anzahl der verwendeten Threads ansteigt.

3.2.3 Maßnahmen zur effektiven Ausnutzung der Speicherbandbreite

Die im vorherigen Abschnitt dokumentierte Problematik lässt sich auch für die anderen, hier betrachteten, Bibliotheken MPTL und MCSTL feststellen und ist gleichermaßen eine Erklärung für das Phänomen, dass mit den für die Benchmarks ausgewählten x86-

Prozessoren für viele Threads keine gute Skalierung mehr erreicht wurde. Dies lässt sich offensichtlich nicht verhindern, dennoch kann der Effekt mit einigen Maßnahmen abgemildert werden. Zentrale Instrumente hierzu sind die Einbeziehung von Informationen über die Cache-Hierarchie und Thread Affinity. Für In-Cache Größen muss bei Kernen mit shared Cache der schnellere Zugriff auf die Daten ausgenutzt werden. Bei Out-Of-Cache Größen jedoch hängt die optimale Strategie von der Architektur ab. Das hier verwendete Xeon-System ist beispielsweise zwar eine Uniform Memory Architecture, besitzt aber, wie in Abschnitt 1.1 beschrieben, zwei Front Side Busse, also einen pro CPU. Es ist dann, ggf. auch bei nur zwei verwendeten Threads sinnvoll, diese auf den unterschiedlichen Prozessoren zur Ausführung zu bringen, da dann beide die volle Bandbreite des jeweiligen FSB erhalten. Anders ist es bei einer Non-Uniform Memory Architecture, wie dem verwendeten Opteron-System. Die Abbildung 3.7 zeigte bereits, dass wenn die Eingabe mit dem Speichercontroller auf CPU 1 erzeugt wurde, für Threads auf CPU 0 eine signifikante Latenz durch den notwendigen Speichertransfer über den HyperTransport-Link entsteht. Deshalb sollten bei NUMA-Systemen auch für Out-Of-Cache Größen zunächst möglichst viele Kerne auf einer CPU verwendet werden.

Der `ThreadPool` nimmt Rücksicht auf diese Unterschiede, indem er bei seiner Konstruktion überprüft, wie viele logische Prozessoreinheiten bei wie vielen physikalischen Prozessoren zur Verfügung stehen. Zusätzlich kann bei der Kompilation das im Rahmen des Quelltexts definierte Flag `-DNUMA` verwendet werden, um die soeben beschriebene Strategie des Schedulings für NUMA-Systeme zu erreichen. Durch die ebenfalls ausgelesenen Informationen über die Cache-Hierarchie kann für UMA-Systeme ein Schwellenwert berechnet werden, der als Entscheidungskriterium dient, ob Threads den Kernen einer CPU zugewiesen oder die Speicherbandbreite verschiedener CPUs ausgenutzt werden sollte. Der Schwellenwert wird bezogen auf die letzte Cache-Stufe vor dem Hauptspeicher nach folgender Formel berechnet:

$$threshold = 1,5 \cdot \frac{cache_size}{\min(cores_per_cache,\ num_threads) + 1}$$

Teilen sich weniger als die insgesamt zur Verfügung stehenden Kerne einen Cache, dann muss die entsprechende Anzahl (cores_per_cache) für die Entscheidungsfindung herangezogen werden. Ist der Cache zwischen allen Ausführungseinheiten geteilt, kann die Größe des Caches entsprechend der aktuellen Nutzer (num_threads) aufgeteilt werden. Der Nenner wird zusätzlich um Eins erhöht, da außer den reinen Nutzdaten für die Funktion auch noch weitere Daten und Instruktionen im Cache gehalten werden. Intel empfiehlt in [11], bei Prozessoren mit Hyper-Threading zwischen einem Viertel und der Hälfte des Caches für jeden Thread zu beanspruchen. Das hier für zwei logische Prozessoren gewählte Drittel liegt in diesem Bereich und hat sich als guter heuristischer Wert herausgestellt. Die Multiplikation mit $1,5$ erfolgt, weil es sich erst dann lohnt, auf die Ausnutzung des Caches zu verzichten, wenn weniger als die Hälfte der Eingabe dort verfügbar ist. Ein optimaler Wert

für alle Architekturen und Algorithmen ist darüber hinaus unmöglich zu finden. Deshalb wird bei den Laufzeitergebnissen zu erwarten sein, dass es rund um diesen Schwellenwert gelegentlich Ausreißer geben wird. Beim Versuch diese zu verhindern traten in aller Regel Ausreißer an anderen Stellen auf. In fast allen anderen Fällen zeigt sich jedoch, insbesondere auch für zwei Threads, eine deutliche Verbesserung der Laufzeiten gegenüber der sonst zufälligen Zuteilung von Tasks an Threads.

Die hier beschriebenen Effekte zeigen, wie wichtig das Konzept der Thread Affinity ist, um auf verschiedene Architekturen und mit ihnen verbundene Problemstellungen effektiv reagieren zu können und unnötigen Overhead zu vermeiden. Zusätzlich sind alle diese Experimente und die dadurch neu gewonnenen Erkenntnisse erst mit Hilfe von Thread Affinity zugänglich geworden, da nur auf diese Weise explizit gemessen werden kann, welcher Prozessorkern welche Laufzeiten erreicht.

3.2.4 Benchmarks zur Speicherbandbreite

In diesem Abschnitt geht es um die Unterschiede zwischen Theorie und Praxis bezüglich der Speicherbandbreite, das Ermitteln praktischer oberer Schranken für den Speicherdurchsatz und die Frage, ob die hier entwickelte Implementierung diese Schranke erreicht und folglich der Speedup tatsächlich durch das Speichersystem beschränkt wird. Die entsprechenden Benchmarks werden auf den Systemen ausgeführt, die in Abschnitt 4.1.1 als Testsysteme für die Ergebnisse dieser Arbeit vorgestellt werden. Den maximalen realen Durchsatz zu ermitteln ist deshalb schwierig, weil eine Verfälschung der Ergebnisse durch den Cache schwer auszuschließen ist. Dabei ist die Assoziativität der Caches zu bedenken, auf Grund derer das Überschreiben vormals in den Cache geladener Daten kaum zu steuern ist. Aktuelle x86-Prozessoren bieten für die Aufgabe der vollständigen Invalidierung des Caches die Assemblerbefehle `invd`, bzw. `wbinvd`, deren Nutzung aber privilegierter Software, wie z. B. dem Betriebssystemkern, vorbehalten ist. Des Weiteren existiert die Instruktion `clflush`, die es erlaubt, eine einzelne Cacheline für ungültig zu erklären. Mit ihrer Hilfe konnte jedoch keine realistische Out-Of-Cache Simulation erzeugt werden, obwohl der Befehl für jede Speicheradresse angewendet wurde, die ein Element der Eingabe enthält. Die Laufzeiten, und damit die Speicherdurchsätze, blieben nahezu unverändert gegenüber der normalen Ausführung. Es ist daher zu vermuten, dass der Hardware-Prefetcher das gewünschte Szenario verhindert. Die realistischsten Ergebnisse wurden erreicht, indem eine wesentlich größere Eingabe in Form von Objekten des Typs `std::vector` erzeugt wurde, als für die jeweilige Messgröße *size* erforderlich. Anschließend kann die Funktion auf den ersten *size* Elementen, die mit hoher Wahrscheinlichkeit bereits ersetzt wurden, ausgeführt werden. Andere Konfigurationen, die insbesondere eine höhere Gefahr von Umordnungen des Codes durch den Compiler beinhalteten, führten zu stark unterschiedlichen Ergebnissen auf den verschiedenen verwendeten Systemen. Das beschriebene Verfahren kommt daher im Rahmen des

Benchmarks `bench_throughput()` für die eigens für diesen Zweck entwickelten parallelen Funktionen für `std::copy()` und die C-Funktion `memcpy()` zum Einsatz. Der Unterschied zwischen den beiden Kopierfunktionen ist, dass `memcpy()` direkt auf dem Speicher agiert und nicht den Zuweisungsoperator des jeweiligen Datentyps verwendet. Zusätzlich wird der erreichte Durchsatz von `inner_product()` gemessen. Die ermittelten Werte können dann mit ausführlichen Messungen über einen größeren Eingabegrößen-Bereich für `inner_product()` und der in Abschnitt 3.3 erläuterten zugehörigen SSE-Version in Beziehung gesetzt werden. Zusätzlich zu den eigenen Benchmarks kommt der *STREAM*-Benchmark [36] zum Einsatz. Das C-Programm misst die beste erreichte Transferrate mehrerer Iterationen für die folgenden in Tabelle 3.1 aufgeführten Operationen:

Funktion	**arithmetische Operation**	**Lese- / Schreibe-Operationen**
Copy	$b[i] = a[i]$	n/n
Scale	$b[i] = x \cdot a[i]$	n/n
Add	$c[i] = a[i] + b[i]$	$2n/n$
Triad	$c[i] = a[i] + x \cdot b[i]$	$2n/n$

Tabelle 3.1: STREAM-Benchmark-Operationen im Überblick

Die Größe der zu verwendenden Arrays, der Offset zwischen ihnen und die Anzahl der Iterationen sind dabei durch den Benutzer zu manipulierende Parameter. Es besteht die Möglichkeit OpenMP-Threads zu verwenden, indem ein entsprechendes Compiler-Flag übergeben wird. Die Berechnung des Durchsatzes geschieht in allen hier verwendeten Benchmarks auf dieselbe Art und Weise.

$$Durchsatz\ [MB/s] = \frac{transferierte\ Daten \cdot (\frac{1024}{1000})^2\ [B]}{benötigte\ Zeit\ [\mu s]}$$

Der konstante Faktor bei der Umrechnung von $B/\mu s$ in MB/s ist deshalb erforderlich, weil $1\ MB = 1024 \cdot 1024\ B$, aber $1\ s = 1000 \cdot 1000\ \mu s$ gilt.

Xeon-System. Für das Xeon-System mit Dual Independent Bus wird in [12] eine theoretische Lesebandbreite von $21,3\ GB/s$ und eine Schreibbandbreite von $10,7\ GB/s$ für jeden der beiden Front Side Busse angegeben. Zunächst erfolgte die Messung mit dem STREAM-Benchmark. Die Kompilation erfolgte mit dem GNU C Compiler, bei wie vom Autor empfohlen höchstem Optimierungslevel (`-O3`) und einer hinreichenden Eingabegröße von $5 \cdot 10^7$ Elementen (Typ `double`). Offset und Anzahl der Iterationen wurden bei den Standardwerten 128 und 10 belassen. Tabelle 3.2 zeigt die Ergebnisse für die sequentielle Ausführung und bei Einsatz von acht OpenMP-Threads.

Mit acht Threads kann im Fall dieses Benchmarks lediglich eine Verdopplung des Speicherdurchsatzes gegenüber der sequentiellen Ausführung erreicht werden. Die Ergebnisse

Funktion	**Durchsatz [MB/s] sequentiell**	**Durchsatz [MB/s] 8 Threads**
Copy	$2821,6397$	$5796,0896$
Scale	$2853,4427$	$5755,0626$
Add	$2973,9630$	$6109,1142$
Triad	$2960,2951$	$6106,9942$

Tabelle 3.2: STREAM-Benchmark: Ermittelter Speicherdurchsatz auf dem Xeon-System

für die arithmetischen Operationen, die zwei Operanden lesen müssen und bei denen zusätzlich zu den Speichertransferoperationen auch Zeit für die Berechnung benötigt wird, sind etwas besser, als für ausschließliches Kopieren. Der Grund dafür liegt in der eingangs erwähnten höheren zur Verfügung stehenden Lesebandbreite im Vergleich zur Speicherbandbreite. Insgesamt bleiben die Ergebnisse deutlich unterhalb den theoretischen Angaben. Experimente mit der Funktion `bench_throughput()` (Tabelle 3.3) zeigen, dass das im Rahmen dieser Arbeit entwickelte `copy()`, welches wegen der Verwendung des `operator=` weitgehend identisch zum Copy des STREAM-Benchmarks ist, in etwa dieselben Werte erzielt. Mit `inner_product()` werden die Werte für die arithmetischen Operationen des STREAM-Benchmarks noch leicht übertroffen, da die Funktion zwar auch $2n$ Leseoperationen, aber nur eine statt n Speicheroperationen durchführen muss. Lediglich mit der Funktion `memcpy()`, welche direkte Operationen auf dem Speicher ohne Zuweisungsoperator und den Overhead von Containern dynamischer Größe realisiert, kann ein noch höherer Durchsatz gemessen werden.

Funktion	**Durchsatz [MB/s] 8 Threads float**	**Durchsatz [MB/s] 8 Threads double**
`copy()`	$5894,36$	$5899,64$
`inner_product()`	$6708,40$	$6706,67$
`memcpy()`	$8375,24$	$8370,64$

Tabelle 3.3: Auf dem Xeon-System mittels `bench_throughput()` ermittelter Speicherdurchsatz

Abbildung 3.8 setzt den Durchsatz des entwickelten parallelen `inner_product()` mit einer Out-Of-Cache Simulation in Beziehung. Die Simulation ist gelungen, da alle Kurven für die gleiche Anzahl Threads zum Schluss zusammenlaufen. In diesem Fall ist auch bei normaler Ausführung die im Cache liegende Partition der Daten so klein, dass die Laufzeit nicht mehr maßgeblich von ihr abhängt. Zunächst werden im Bereich von In-Cache Größen entsprechend hohe Durchsätze mit zwei und vier Threads erreicht. Bei Verwendung der zweiten CPU (mit sechs und acht Threads) sind die Kosten für die Speichertransfers noch zu hoch. Ihr Einsatz lohnt sich erst nach Verlassen der In-Cache Größen, der durch

einen deutlichen Knick erkennbar ist. Bei $2 \cdot 10^6$ und $3 \cdot 10^6$ Elementen wird der maximale Speicherdurchsatz erreicht, ehe er anschließend auf das reale Out-Of-Cache Niveau sinkt. Bei den größten Eingaben ist erkennbar, dass mit zwei Threads der Durchsatz gegenüber der sequentiellen Ausführung noch verdoppelt werden kann, aber anschließend mit bis zu acht Threads insgesamt nur noch 1 GB/s mehr erreichbar ist. Folglich begrenzt der Speicherdurchsatz den möglichen Beschleunigungsfaktor enorm.

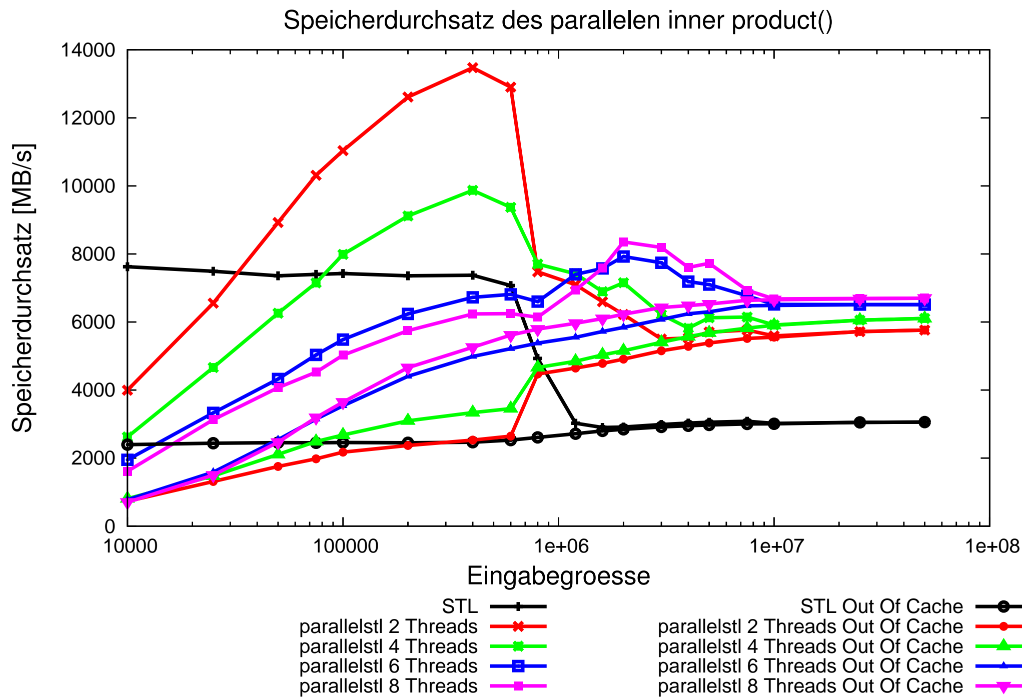

Abbildung 3.8: Speicherdurchsatz des parallelen `inner_product()`

An zwei Stellen wird deutlich, dass das Vorhandensein von benötigten Daten ausschließlich im Cache des den Hauptprozess ausführenden Kerns, auch die Laufzeiten aller anderen Kerne begünstigt, obwohl deren Caches zunächst keine dieser Daten enthalten. Das erste Merkmal ist der Durchsatz mit vier Threads - die Gesamtlaufzeit ist deutlich besser als mit sechs oder acht Threads, obwohl die Daten lediglich in einem der beiden vorhandenen Caches dieser CPU vorliegen. Das zweite Merkmal ist der Bereich des Durchsatzmaximums. In weiteren Messungen zeigte sich, dass die benötigte Laufzeit der vier Kerne auf der CPU mit dem Hauptprozess deutlich geringer war, als die der anderen vier. Und auch die vier langsameren Kerne waren in diesen Versuchen immer noch deutlich schneller, als bei der vollständigen Out-Of-Cache Variante. Dies lässt darauf schließen, dass sowohl On-Chip- als auch Off-Chip-Kopieroperationen zu einer besseren Leistung führen, als wenn alle acht Kerne gleichermaßen und simultan Daten vom Hauptspeicher anfordern müssen.

Für die parallele und vektorisierte Version von `inner_product()` (Abbildung 3.9) zeigt sich dasselbe Bild, mit dem Unterschied, dass während der In-Cache Größen der Nutzen

aus dem schnellen Zugriff auf die Daten noch wesentlich höher ist. Außerdem kann mit vier Threads in diesem Bereich die sequentielle Laufzeit nicht erreicht werden. Werden die In-Cache Größen verlassen, wird deutlich, dass auch mit Hilfe der Vektorisierung keine höhere Bandbreite mehr erreicht werden kann, als die ca. 6700 MB/s, die bereits ohne Vektorisierung erzielt wurden.

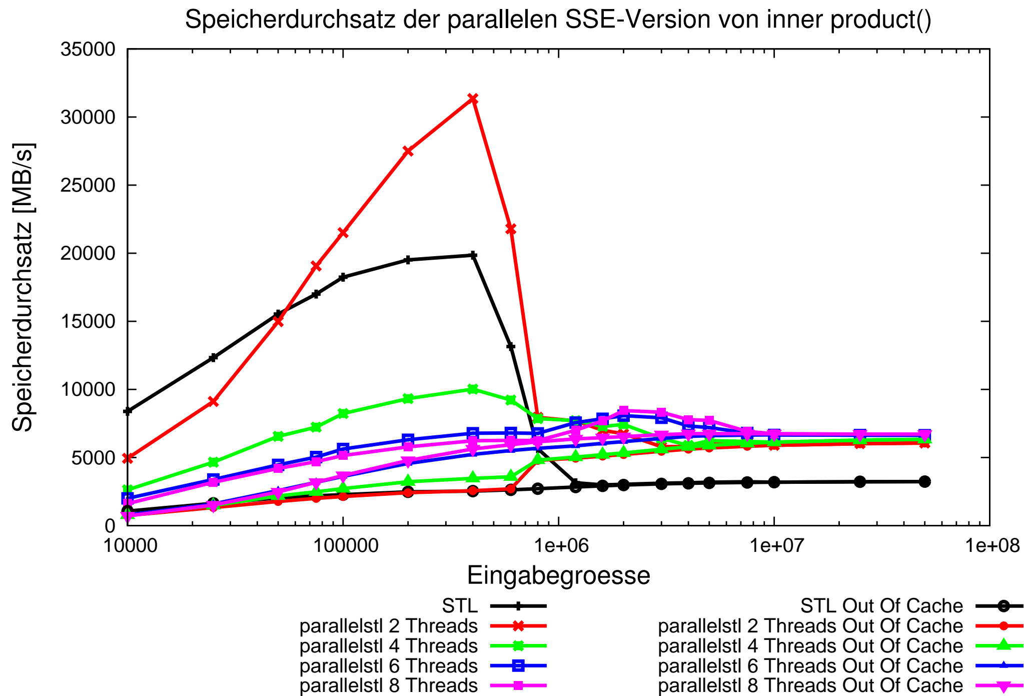

Abbildung 3.9: Speicherdurchsatz der parallelen SSE-Version von `inner_product()`

Opteron-System Der HyperTransport-Bus des Opteron-Systems hat eine Taktfrequenz von 1000 MHz. Die resultierende theoretische Speicherbandbreite jeder CPU zu von ihr alloziertem Speicher beträgt im Dual Channel Betrieb folglich 16 GB/s. Zunächst wurde erneut der STREAM-Benchmark ausgeführt, dessen erreichte Durchsätze zunächst überraschend hoch sind. Der Vergleich der parallelen mit den sequentiellen Ergebnissen zeigt einen Faktor von etwa drei, der mit vier Threads erreicht werden kann. Bei den Ergebnissen, die mittels des Intel C Compilers ermittelt wurden, hat der Compiler alle Schleifen und OpenMP-`parallel for`-Routinen zusätzlich vektorisiert. Das Szenario, bei dem die hohen Werte in Tabelle 3.4 erreicht wurden, sind jedoch für die Verwendung der hier untersuchten parallelen STL-Algorithmen unrealistisch. Sie resultieren aus der Tatsache, dass die Eingabe im STREAM-Benchmark nicht nur parallel verarbeitet, sondern auch parallel initialisiert wird, wovon in typischen Anwendungsfällen paralleler STL-Algorithmen nicht ausgegangen werden kann. Im Fall von NUMA-Systemen führt dies zu einem wesentlich höheren Durchsatz, weil Betriebssysteme bei der Allokation des zugehörigen Speichers ei-

ne *first-touch Strategie* [10] verwenden. Das heißt, dass dem jeweiligen Array zwar bereits zur Kompilierzeit ein virtueller Adressraum, jedoch noch keine physikalische Adresse zugewiesen wurde. Dies geschieht erst zum Zeitpunkt der ersten Verwendung des Arrays oder eines Teiles davon, wenn auf Grund des resultierenden *page fault* die Kontrolle an das Betriebssystem übergeben wird und dieses versucht, eine möglichst zu dem zugreifenden Thread lokale Adresse zu verwenden. Man spricht dabei auch von *lazy instantiation*. Die Allokation durch den zuerst zugreifenden Thread geschieht daher mittels des jeweiligen Speichercontrollers des ausführenden Kerns, und nicht durch den Speichercontroller des den Hauptprozess ausführenden Kerns, wie es im typischen Anwendungsfall wäre. Folglich sind die anschließenden Zugriffe bei der parallelen Verarbeitung wesentlich billiger und die resultierenden Durchsätze höher. Selbst bei diesem Szenario zeigt sich jedoch erneut, dass alle real erreichbaren Werte deutlich hinter den theoretischen Angaben zurückbleiben.

Funktion	**Durchsatz [MB/s] sequentiell**	**Durchsatz [MB/s] 4 Threads**	**Durchsatz [MB/s] 4 Threads ICC**
Copy	2737, 0996	7865, 8071	11494, 9049
Scale	2661, 7172	7899, 0449	11445, 1106
Add	2872, 6896	8633, 5268	11529, 8071
Triad	2860, 4939	8619, 4513	11476, 1257

Tabelle 3.4: STREAM-Benchmark: ermittelter Speicherdurchsatz auf dem Opteron-System

Wird der STREAM-Benchmark so modifiziert, dass die Initialisierung der Eingabe wie üblich sequentiell durch den Hauptprozess erfolgt, werden lediglich noch die in Tabelle 3.5 gelisteten Werte erreicht.

Funktion	**Durchsatz [MB/s] 4 Threads**	**Durchsatz [MB/s] 4 Threads ICC**
Copy	3885, 8366	5966, 4946
Scale	3878, 2015	3826, 5051
Add	4040, 3206	4020, 7741
Triad	4125, 9913	4192, 1381

Tabelle 3.5: STREAM-Benchmark: ermittelter Speicherdurchsatz auf dem Opteron-System ohne parallele Initialisierung der Eingabe

Im Vergleich mit diesen Werten zeigen die mit `bench_throughput()` erzeugten Messergebnisse, dass sich das im Rahmen dieser Diplomarbeit entwickelte Konzept für die beiden Kopieroperationen und `inner_product()` im Bereich des maximal erreichbaren Speicherdurchsatzes bewegt.

Funktion	**Durchsatz [MB/s] 4 Threads float**	**Durchsatz [MB/s] 4 Threads double**
`copy()`	3469,09	3464,86
`inner_product()`	4613,55	4782,62
`memcpy()`	5417,99	5400,95

Tabelle 3.6: Auf dem Opteron-System mittels `bench_throughput()` ermittelter Speicherdurchsatz

Für das parallele `inner_product()` wird der mit vier Threads gemessene Durchsatz bei lediglich zwei verwendeten Threads noch übertroffen, wie Abbildung 3.10 zeigt. Die Kosten für Zugriffe über den zweiten Speichercontroller sind so hoch, dass bei dieser Funktion, deren Laufzeit vom Speicherdurchsatz dominiert wird, die annähernd lineare Beschleunigung mit zwei Threads eines Dualcore-Chips auch mit drei und vier Threads nicht mehr überboten werden kann. Daher zeigt sich auch mit drei Threads ein durchgängig geringerer Speedup als mit vier, weil bei gleichen Kosten für den entfernten Speicherzugriff die Problemgröße pro Thread (und damit die Laufzeit) größer ist. Ähnlich wie beim Xeon-System zeigt sich zunächst der erwartete Vorteil im Bereich von In-Cache Größen und eine Konvergenz des erreichbaren Out-Of-Cache Durchsatzes.

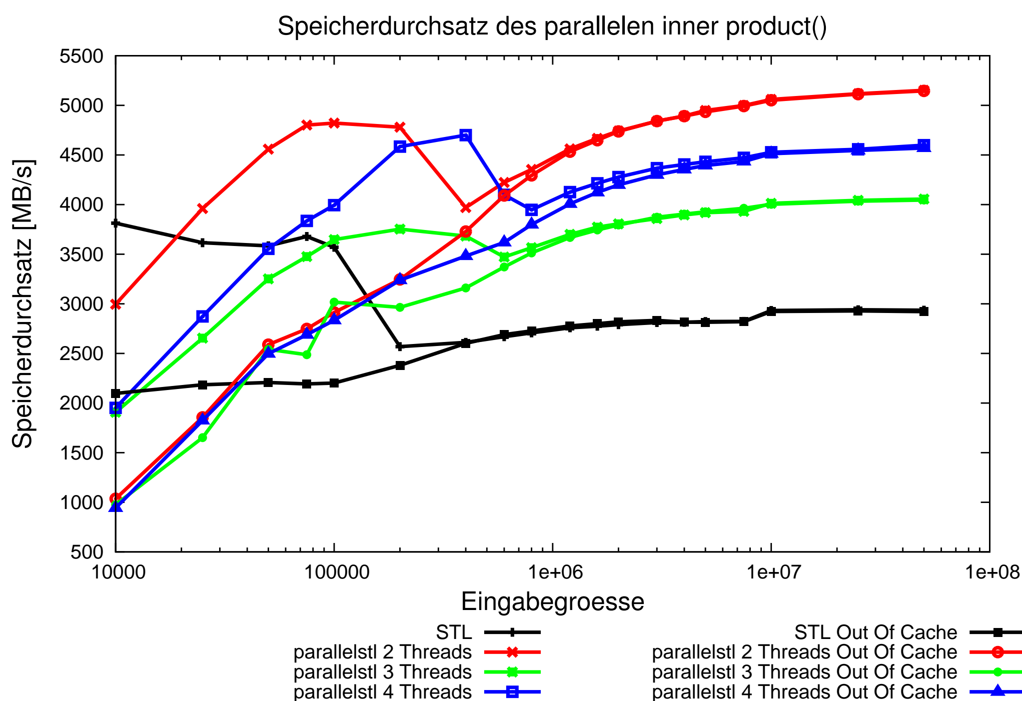

Abbildung 3.10: Speicherdurchsatz des parallelen `inner_product()`

Für die SSE-Variante (Abbildung 3.11) besteht die gleiche Situation. Der Nutzen aus dem Cache ist bei sequentieller Ausführung wie erwartet erneut wesentlich höher und die Abstände zwischen den Out-Of-Cache Grenzwerten sind geringer geworden, da die Verarbeitung bereits geladener Daten entsprechend doppelt so schnell abläuft. Der erreichte maximale Durchsatz bei der nicht-vektorisierten Ausführung von 5150 MB/s kann jedoch auch mit SSE nur um ca $0,4$ MB/s übertroffen werden.

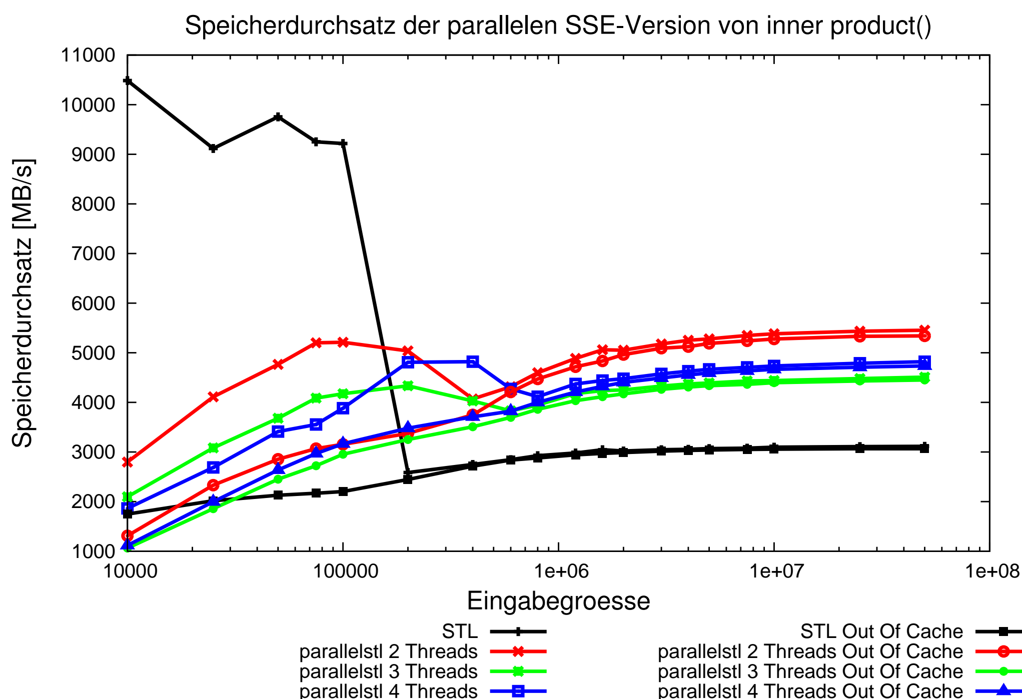

Abbildung 3.11: Speicherdurchsatz der parallelen SSE-Version von `inner_product()`

Cell Blade Beim Cell Blade wird deutlich, dass die beiden Power Processing Units die zur Verfügung stehende Bandbreite des Speichersystems nicht saturieren können. Dies ist allerdings auch nicht ihre Aufgabe, da sie ursprünglich nur für die Steuerung von und Aufgabenverteilung an jeweils acht SIMD-Prozessoren (SPEs) vorgesehen sind. Aus diesem Grund führt die Verwendung des `-DPPU`-Flags für den Compiler auch nicht zu einem NUMA-speziellen Dispatching der Tasks, wie es bei AMD-Systemen der Fall ist. Durch Übergabe von `-DNUMA` kann dies aber dennoch vom Benutzer erzwungen werden. Wie der folgende STREAM-Benchmark (Tabelle 3.7) zeigt, liegen die Durchsätze auch im Vergleich zu den zuvor behandelten x86-Prozessoren auf Grund der geringeren Rechenleistung auf einem deutlich niedrigeren Niveau. Jedoch ist teilweise eine lineare Skalierung des Durchsatzes mit vier Threads zu beobachten, obwohl es sich jeweils um Simultaneous Multithreading mit zwei Threads handelt, und nicht um vollständige Cores auf den beiden Prozessoren.

Funktion	**Durchsatz [MB/s] sequentiell**	**Durchsatz [MB/s] 4 Threads**
Copy	820, 7552	2844, 8387
Scale	782, 7889	2679, 9102
Add	1161, 7084	3955, 7150
Triad	1161, 2188	3992, 3810

Tabelle 3.7: STREAM-Benchmark: ermittelter Speicherdurchsatz auf dem Cell Blade

Die mittels `bench_throughput()` ermittelten Speicherdurchsätze für die Kopieroperationen bestätigten die Ergebnisse des STREAM-Benchmarks, während mit `memcpy()` kein deutlich verbesserter Durchsatz mehr erreicht werden kann. Im Gegensatz zu den beiden x86-Systemen liegt die Leistung von `inner_product()` deutlich unter der der anderen. Dies ist dadurch zu erklären, dass auf diesem System die Speicherbandbreite nicht saturiert ist. Auf den x86-Systemen erreichen die Kopieroperationen auf Grund ihrer Speicheroperationen bereits die Grenze des Durchsatzes, während sie auf dem Cell Blade dadurch nicht beschränkt werden. Da aus diesem Grund auch keine langen Wartezeiten für die CPUs entstehen, sind die Multiplikation und die Addition, die `inner_product()` im Vergleich zu den beiden Kopierfunktionen zusätzlich für jedes Element ausführen muss ein realer Nachteil in der Laufzeit und damit im erreichten Speicherdurchsatz.

Funktion	**Durchsatz [MB/s] 4 Threads float**	**Durchsatz [MB/s] 4 Threads double**
`copy()`	3129, 35	3148, 52
`inner_product()`	1526, 54	1651, 71
`memcpy()`	3135, 64	3142, 27

Tabelle 3.8: Auf dem Cell Blade mittels `bench_throughput()` ermittelter Speicherdurchsatz

Die Abbildung 3.12 des erreichten Speicherdurchsatzes für `inner_product()` mit bis zu vier Threads unterstreicht die zuvor erfolgten Äußerungen. Die lineare Skalierung des Durchsatzes zeigt erneut, dass für jeden zusätzlichen Thread genügend Speicherbandbreite vorhanden ist und trotz der NUMA-Architektur auch der Zugriff über den zweiten Speichercontroller für eine rechtzeitige Versorgung der CPUs mit Daten sorgt. Dies wird durch die Kombination einer breitbandigen Anbindung der Prozessoren untereinander (FlexIO-Interface mit 20 GB/s theoretischem Durchsatz) und einer im Vergleich zu den x86-Prozessoren verhältnismäßig geringen Ausführungsgeschwindigkeit erreicht. In allen anderen Punkten gleicht das Diagramm denen zu den x86-Systemen, lediglich sieht man ein deutlich früheres Zusammenlaufen mit den Out-of-Cache Durchsätzen, da der Cache der PPU mit 512 KB kleiner bemessen ist.

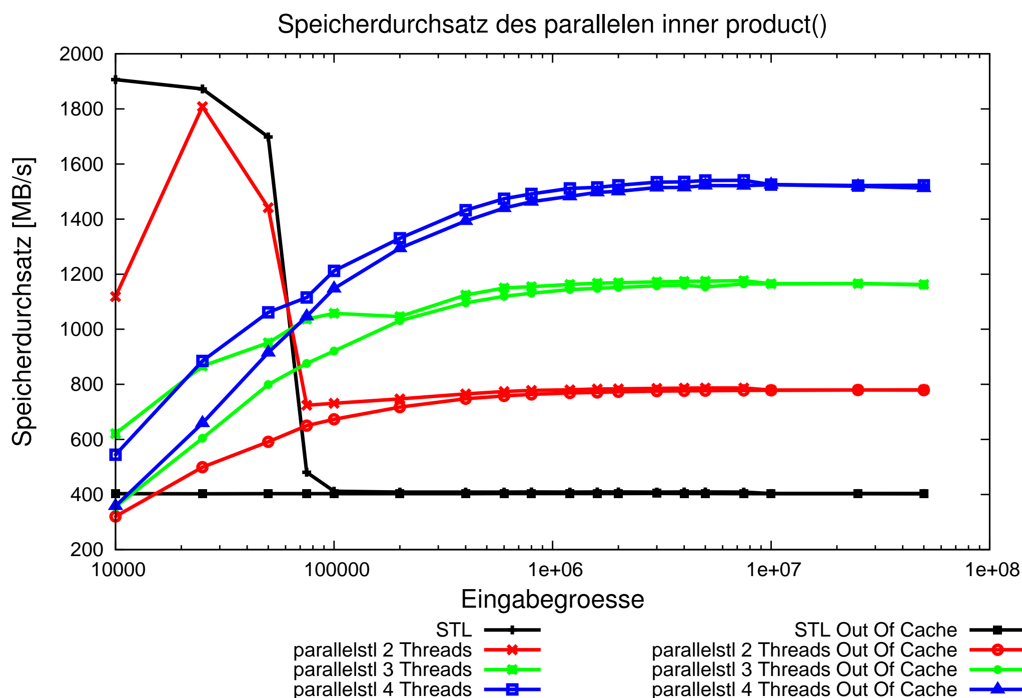

Abbildung 3.12: Speicherdurchsatz des parallelen `inner_product()`

3.3 Auswahl der Algorithmen für die Experimente

In diesem Abschnitt erfolgt die Auswahl der STL-Algorithmen, für die im Anschluss angestrebt wird, bereits für kleinere Eingabegrößen einen Leistungsgewinn gegenüber der sequentiellen Ausführung zu erzielen. Die Struktur beruht auf der in Abschnitt 1.2 beschriebenen groben Klassifikation von Algorithmentypen. Es wird mindestens ein Algorithmus pro Typklasse betrachtet. Neben der Begründung der Auswahl werden an dieser Stelle auch strategische Details der Implementierung erläutert, sowie welche Entscheidungen für die Laufzeit besonders wichtig sind.

3.3.1 Nicht-mutierende Algorithmen

In dieser Algorithmenklasse ist `find()` die interessanteste Aufgabenstellung, insbesondere weil hier durch die MCSTL bereits gute Ergebnisse erzielt wurden, da man im Gegensatz zu anderen Bibliotheken eine Möglichkeit gefunden hat, andere parallel arbeitende Threads ihre Suche abbrechen zu lassen, wenn das gesuchte Element gefunden wurde. Wie in Abschnitt 2.3.2 beschrieben, kann nur auf diese Art und Weise in bestimmten Fällen eine Laufzeit von $o(\frac{n}{p})$ erreicht werden.

Ein weiteres Argument für diesen Algorithmus ist, dass Vergleichbarkeit auch mit der MPTL hergestellt werden kann, da er auch in dieser Bibliothek implementiert wurde.

find. Bei der Entwicklung eines parallelen `find()`-Algorithmus gibt es zwei wesentliche Probleme. Zunächst muss, wie bereits angedeutet, ein Weg gefunden werden, die Suche aller Threads zu beenden, sobald das korrekte Element gefunden wurde. Um dies zu bewerkstelligen, muss es aber auch während der parallelen Phase möglich sein herauszufinden, ob ein gefundenes Element das erste passende Element in der Sequenz ist. Zu diesem Zweck müssen alle Suchvorgänge vor dem gefundenen Element abgeschlossen sein. Es liegt daher grundsätzlich nahe, in der in Abschnitt 2.3 beschriebenen Weise mit Blöcken ansteigender Größe (growing block) vorzugehen. Auch die dort abgebildeten Ergebnisse der MCSTL auf einem Sun UltraSPARC T1 Prozessor vermitteln dies. Wenn der Overhead für den Austausch eines durch einen Thread zu durchsuchenden Blocks niedrig ist, kann aber auch eine Variante mit konstant kleiner Blockgröße sehr effizient sein.

Die hier entwickelte Implementierung eines parallelen `find()` verwendet konstante Blockgrößen und dynamisches Load Balancing. Ein zentraler `BlockDonator` dient als Abholort für Blöcke, deren Größe jeweils der Anzahl der maximal im L1-Cache zu speichernden Elemente $l1_elements$ entspricht. Diese Wahl sorgt für eine sehr schnelle Terminierung, falls das erste Element in der Sequenz gefunden wurde. Die Idee für diese Strategie kommt von dem im noch folgenden Abschnitt zu `partition()` erläuterten Verfahren. Dort kommt ebenfalls ein `BlockDonator` zum Einsatz, der allerdings zur Abholung von Blöcken beider Enden der Sequenz dient. Die Threads durchsuchen die Blöcke nach dem gesuchten Element und prüfen nach jeweils vier Elementen, ob bereits ein Element gefunden wurde, welches vor ihrer aktuellen Position liegt. In diesem Fall terminiert die Ausführung. Andernfalls wird die Suche bis zum Ende des Blocks fortgesetzt und gegebenenfalls ein neuer Block angefordert. Da prinzipiell mehrere Elemente mit dem gesuchten Wert gefunden werden können, bevor klar ist, welches das erste in der Sequenz ist, können auch mehrere Threads gleichzeitig Schreibzugriffe auf die entsprechende Ergebnisvariable durchführen. Es ist daher notwendig, diese Zugriffe mit einem Mutex zu schützen, um gegenseitigen Ausschluss sicher zu stellen. Der entsprechende Overhead fällt aber nur an, wenn zwei Threads gleichzeitig ein Element mit gesuchtem Wert finden. Lesezugriffe müssen nicht geschützt werden, die schlimmste Folge beim Lesen eines grade in der Aktualisierung befindlichen Ergebnisses ist eine weitere Iteration der Suchschleife, die vier zusätzliche Vergleiche zur Folge hat. Ähnlich wie das Verfahren der MCSTL, geht der Algorithmus zunächst einige wenige Elemente sequentiell vor, nämlich genau $n \mod l1_elements$. Falls das gesuchte Element am Anfang der Sequenz positioniert ist, ist so eine bessere Leistung zu erwarten.

3.3.2 Mutierende Algorithmen

Bei den mutierenden Algorithmen fällt die Wahl auf die Funktionen `partition()` und `replace_copy_if()`. Der Partitionsalgorithmus ist als Basisfunktion für viele weitere Algorithmen wie `nth_element()` und diverse Sortierverfahren von großer Bedeutung. Des

Weiteren erlauben viele unterschiedliche theoretische Ansätze hier auch neue Ergebnisse in der Praxis. Die höhere Komplexität gegenüber einfach linear arbeitenden Algorithmen wie etwa `fill()`, machen diese Funktion zu einer interessanten Problemstellung.

`replace_copy_if()` hingegen ist ein linear arbeitender Algorithmus, der abhängig von einem Prädikat entweder Elemente von einer Sequenz in eine andere kopiert, oder sie durch ein als Parameter übergebenes Element ersetzt. Er stellt aber insbesondere auf Grund seiner hohen Anzahl an bedingten Sprüngen ein interessantes Problem dar, da sich diese auf verschiedenen Architekturen durchaus unterschiedlich in der Laufzeit auswirken können. Des Weiteren ist diese Funktion ebenfalls Bestandteil der MPTL und kann so zu Vergleichen herangezogen werden. Interessanterweise scheint sie jedoch in der Version 0.8.0-beta der MCSTL vergessen worden zu sein, denn die STL-Funktion wird zwar per `#define replace_copy_if __mcstl_sequential_replace_copy_if` umbenannt, jedoch tritt in der Datei mcstl_algo.h keine parallelisierte Funktion an ihre Stelle. Ein Aufruf von `std::replace_copy_if()` führt also zu einem Kompilierfehler mit der Meldung, dass die Funktion kein Teil des Namensraumes `std::` sei. Aus diesem Grund wurde zusätzlich `replace_if()` implementiert, welches sich von `replace_copy_if()` darin unterscheidet, dass es in situ arbeitet, die Ersetzungen also ggf. auf dem Eingabevektor vorgenommen werden und keine Kopieroperationen stattfinden.

Als vierte Funktion aus dieser Typenklasse wurde die Funktion `copy()` parallelisiert, um die Benchmarks zur Speicherbandbreite in Abschnitt 3.2.4 zu ermöglichen.

partition. Die entwickelte Implementierung von `partition()` ist eine Verfeinerung des Verfahrens von Tsigas et al. [50]. Grundlegende Idee des Algorithmus ist die Zerlegung der Sequenz in eine Menge von Blöcken. Die Autoren empfehlen die Wahl der Blockgröße B so, dass jeweils zwei Blöcke gleichzeitig in den L1-Cache passen. Jedoch können auch andere Konfigurationen sehr effizient sein [27]. Der Ansatz lässt sich grob in eine parallele und zwei sequentielle Phasen unterteilen.

Jeder Thread bearbeitet einen Block von der linken und einen von der rechten Seite und *neutralisiert* diese, d.h. führt falls erforderlich Austauschoperationen zwischen beiden Blöcken aus, sodass im rechten Block kein Element gleichwertig oder kleiner und im linken Block kein Element größer als das Pivotelement ist. Entweder werden beide Blöcke zufällig gleichzeitig neutralisiert, dann fordert der Thread zwei neue Blöcke an, oder bei einem Block müssen weniger Austauschoperationen ausgeführt werden, dann wird zunächst nur dieser ersetzt. In zwei Variablen `LN` und `RN` wird die Anzahl der Elemente summiert, die Teil von links oder rechts neutralisierten Blöcken sind. Diese Information ist hilfreich, um zum Schluss berechnen zu können, welches Intervall in der sequentiellen Phase noch betrachtet werden muss. Pseudocode 3.1 beschreibt diesen Prozess, den jeder Thread in der parallelen Phase ausführt, bis keinerlei Blöcke mehr zur Verfügung stehen. Die Werte `LN`

und `RN` haben zu Beginn den Wert Null.

Pseudocode 3.1 Sukzessive Block-Neutralisierung durch die Threads

```
Block left = get_left_block();
Block right = get_right_block():
repeat
  result = neutralize(left, right);
  if (result == BOTH) then
    left = get_left_block();
    right = get_right_block():
    LN += BLOCK_SIZE;
    RN += BLOCK_SIZE;
  else if (result == LEFT) then
    left = get_left_block();
    LN += BLOCK_SIZE;
  else
    right = get_right_block():
    RN += BLOCK_SIZE;
  end if
until no block is left
```

Ziel ist es, lediglich das Intervall $[LN, N - RN]$ weiterhin partitionieren zu müssen, wobei N die Anzahl der Elemente ist. Zum Ende der parallelen Phase verbleibt jedoch bei jedem Thread höchstens ein Block, falls das letzte Blockpaar nicht zufällig vollständig neutralisiert werden konnte. Diese Blöcke können an beliebigen Stellen in der Sequenz liegen, insbesondere auch im Bereich $[0, LN]$, sowie $[N - RN, N - 1]$. Sie werden in einer Datenstruktur gesammelt, aufsteigend nach Position sortiert und anschließend erneut sequentiell neutralisiert. Nach dieser sequentiellen Neutralisierungsphase ist das Ziel möglicherweise noch immer nicht erreicht, denn einige Blöcke sind zwar neutralisiert, aber nicht korrekt platziert. Beispielsweise ist es möglich, dass nach der parallelen Phase ausschließlich Blöcke übrig bleiben, die dort als „rechte“ Blöcke betrachtet wurden und nun in der sequentiellen Phase als „linke“ Blöcke in die Neutralisierung eingehen. Liegt also ein als linker (rechter) betrachteter Block in der sequentiellen Phase nicht im Bereich $[0, LN-1]$ ($[N-RN, N-1]$), dann wird er nach der Neutralisierung nicht zu LN (RN) hinzugerechnet und auch nicht aus der Menge der verbleibenden Blöcke gelöscht. Nun machen sich die Autoren zu nutze, dass wenn es einen verbleibenden Block im Intervall $[0, LN - 1]$ gibt, es einen vollständig neutralisierten Block rechts von LN, also im Intervall $[LN, N - RN]$ geben muss. Deshalb werden die verbleibenden Blöcke mit neutralisierten Blöcken im Bereich $[LN, N-RN]$ ausgetauscht, um definitiv korrekt platzierte Elemente links von LN und rechts von $N - RN$

zu erhalten. Damit ist das Ziel erreicht und die Funktion kann mit einem sequentiellen Aufruf von `partition()` auf dem Intervall $[LN, N - RN]$ abgeschlossen werden. Abbildung 3.13 veranschaulicht die einzelnen Phasen des Algorithmus.

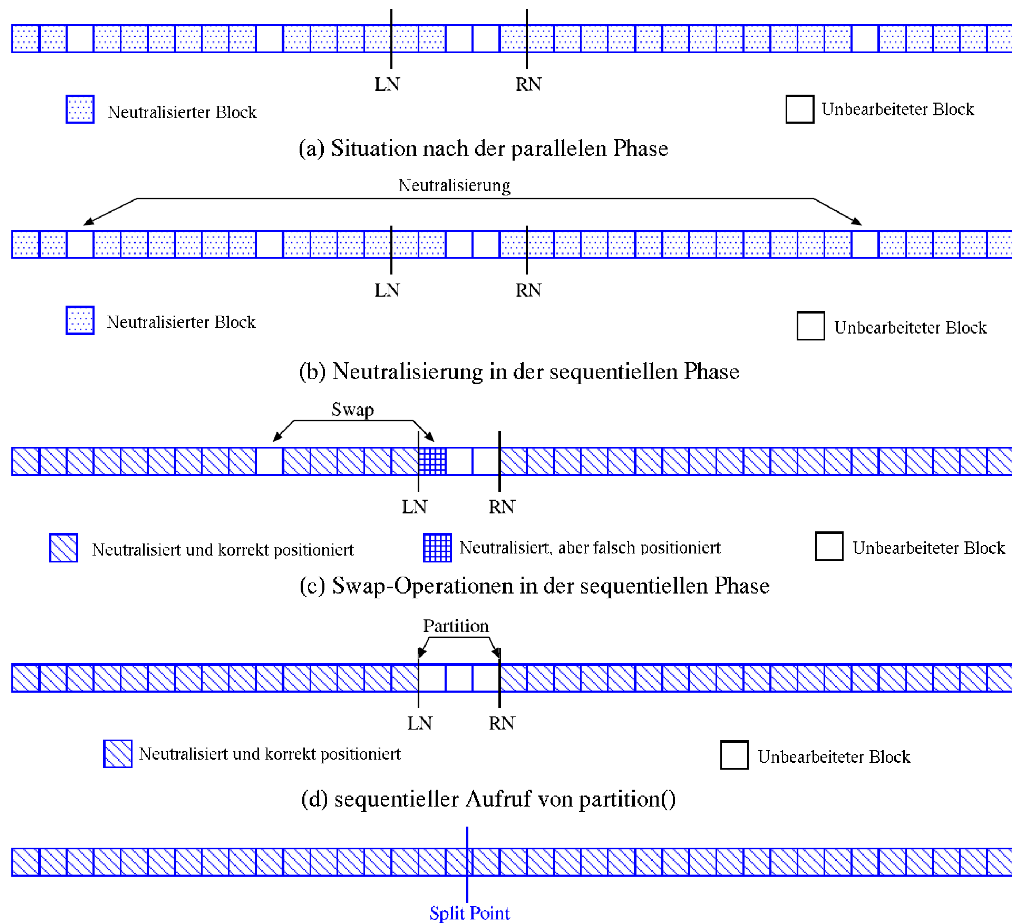

Abbildung 3.13: Schematische Darstellung des Verfahrens von Tsigas et al. Übersetzt aus [50]

Die Implementierung der MCSTL beruht ebenfalls grundlegend auf diesem Verfahren. Frias et al. beschreiben die Tatsache, dass der Algorithmus Nutzinformation, die in der parallelen Phase bereits berechnet wurde, in den sequentiellen Phasen ignoriert und damit teilweise redundante Vergleichsoperationen ausführt [27]. Als Lösung wurde eine baumartige Datenstruktur vorgeschlagen, die es auf Basis der in der parallelen Phase berechneten Information ermöglicht, mit zwei Threads weitere simultane Austauschoperationen vorzunehmen, bevor der sequentielle Aufruf von `std::partition()` für ein Restintervall erfolgt. In der Tat geht zwischen der parallelen und sequentiellen Phase Information verloren. In letzterer ist nicht mehr bekannt,

- an welcher Stelle innerhalb eines Blocks die Neutralisierung abgebrochen wurde
- ob der jeweilige Block als linker oder als rechter Block betrachtet wurde

- bis zu welchem Punkt Blöcke als „linke“, bzw. ab welchem Punkt Blöcke als „rechte“ Blöcke betrachtet werden.

Insbesondere im Fall der Swap-Operationen mit Blöcken im Bereich $[LN, N - RN]$ (siehe (c) in Abbildung 3.13), werden auf Grund dessen im schlimmsten Fall bis zu B unnötige Vergleiche und Austauschoperationen für jeden verbleibenden Block durchgeführt. Dennoch ist nach eigenen Messungen der Anteil der sequentiellen Phase an der Gesamtlaufzeit des Verfahrens mit zunehmender Eingabegröße vernachlässigbar. Dies ist dadurch zu erklären, dass die sequentielle Phase unabhängig von der Eingabegröße immer auf $\leq num_threads$ Blöcken fixer Größe arbeitet. Dennoch kann bei geschickter Implementierung ein geringer, aber messbarer Performancegewinn gegenüber der ursprünglichen sequentiellen Phase erzielt werden. Obwohl auch ohne zusätzliche Maßnahmen bereits ein Geschwindigkeitsvorteil gegenüber der MCSTL und der Implementierung von Frias et al. vorlag, wurde die grundlegende Idee, die hinter dem Vorschlag von Frias et al. steckt, für eine weitere Optimierung des im Rahmen dieser Diplomarbeit entwickelten Algorithmus verwendet.

Zunächst werden die oben aufgeführten verlorenen Informationen aus der parallelen Phase gespeichert. Die Position nach dem Ende des letzten Blockes, der als „linker“ Block betrachtet wurde, wird dabei hier als β bezeichnet. Wie in [27] nur schwer nachvollziehbar und erst durch Einblick in den Quellcode verständlich, geht der dort vorgestellte Algorithmus dann wie im Folgenden beschrieben vor. Man betrachte dazu das Intervall $[0, \beta)$, für die rechte Seite (d.h. $[\beta, N - 1]$) verfährt der Algorithmus analog.

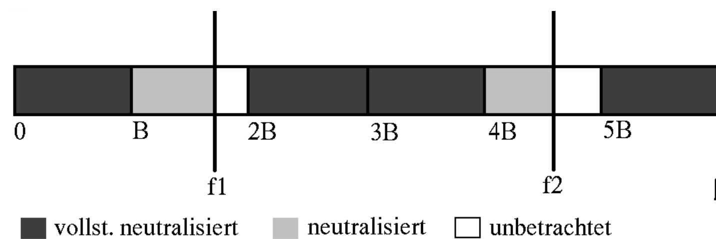

Abbildung 3.14: Beispiel für die Situation nach der parallelen Neutralisierungsphase

Abbildung 3.14 zeigt eine mögliche Situation nach der parallelen Phase mit zwei verbleibenden linken Blöcken, deren aktueller Stand der Neutralisierung durch die Stellen $f1$ und $f2$ markiert ist. Sei nun $left_elements$ die Summe aller Elemente zwischen dem Beginn des ersten nicht vollständig neutralisierten Blockes (Position B im Beispiel) und β. Sei außerdem $left_processed$ die Summe aller *neutralisierten* Elemente zwischen B und β. Folglich ist $left_unprocessed = left_elements - left_processed$ die Summe aller nicht neutralisierten Elemente im Intervall $[0, \beta)$. Die Position $l\beta = \beta - left_unprocessed$ begrenzt schließlich das Intervall, bzw. die Summe der Elemente, für die anhand der parallelen Phase definitiv sichergestellt ist, dass sie links vom Pivotelement stehen müssen. Die

Differenz $\beta - l\beta$ entspricht außerdem exakt der Summe der nicht betrachteten Elemente $left_unprocessed$. Die wesentliche Erkenntnis ist nun, dass folglich die nicht neutralisierten Bereiche genau mit den in Abbildung 3.15 rot markierten (neutralisierten) Elementen ausgetauscht werden können. Frias et al. sprechen in [27] von einer weiteren parallelen Austauschphase, da dasselbe datenunabhängig auf der rechten Seite für ein Intervall $[\beta, r\beta]$ geschehen kann. Anschließend verbleibt die Aufgabe, sequentiell `std::partition()` für das Intervall $[l\beta, r\beta]$ aufzurufen.

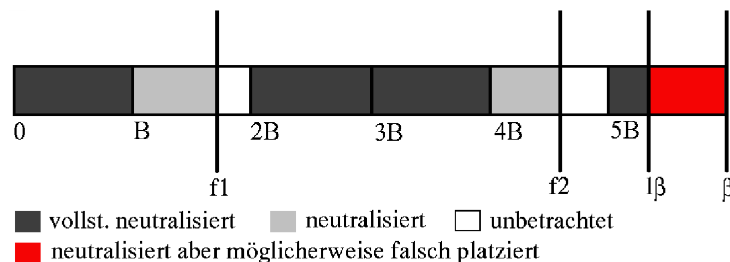

Abbildung 3.15: Visualisierung des Vorgehens nach der parallelen Neutralisierungsphase

Die baumartige Datenstruktur mit zusätzlichen Knoten, die keinerlei neue Nutzinformation speichern und die Art und Weise, wie die Informationen der Knoten im Quellcode zu [27] berechnet werden, erscheint dabei unnötig aufwändig. Die im Rahmen dieser Diplomarbeit entwickelte Variante verwendet daher lediglich einfache doppelt-verkettete Listen, um die Teilintervalle der Blöcke, die noch nicht betrachtet wurden, sowie die neutralisierten Blöcke zwischen den nicht-neutralisierten, zu speichern. Alle anderen Informationen lassen sich durch Summen- und Differenzbildungen einfach daraus herleiten. Zusätzlich werden die entsprechenden Intervalle abhängig davon, ob sie sich links oder rechts von der Position $l\beta$ (bzw. $r\beta$) befinden, sofort in unterschiedliche Listen sortiert, während der Ansatz von Frias et al. zunächst alle Intervalle gleichartig behandelt und daher eine Hilfsmethode benötigt, um die austauschbaren Teilintervalle nachträglich zu berechnen.

Auf Grund der beschriebenen geringen Signifikanz der sequentiellen Phase für die Laufzeit, hat sich in Messungen herausgestellt, dass je nach Anzahl der nach der (ersten) parallelen Phase verbleibenden Blöcke, eine rein sequentielle Ausführung dieses Verfahrens zu schnelleren Ergebnissen führen kann, als das realisierte Anstoßen von zwei Threads. In jedem Fall ist der entwickelte Ansatz (ob sequentiell oder parallel ausgeführt) effizienter, als die ursprüngliche sequentielle Phase von Tsigas et al.

Dennoch vermeidet auch dieses verbesserte Verfahren nicht alle unnötigen Austauschoperationen. Die in Abbildung 3.14 und 3.15 weiß markierten Flächen enthalten möglicherweise korrekt platzierte Elemente, welche aber in jedem Fall, ob notwendig oder nicht, mit Elementen aus dem rot markierten Bereich ausgetauscht werden. Zusätzlich werden eben diese Elemente, die nach diesem Prozess im rot markierten Bereich stehen, vom nachfolgenden sequentiellen Aufruf von `std::partition()` ein zweites Mal durchlaufen.

replace_copy_if / replace_if. Am Beispiel `replace_copy_if()` wird nun einmalig die triviale Parallelisierung eines linear auf dem Speicher arbeitenden Algorithmus beschrieben. Dies gilt in gleichem Maße für alle anderen Funktionen dieses Typs, und wird daher auch im folgenden Abschnitt für die Funktionen `inner_product()` und `accumulate()` nicht wiederholt. Ebenso erfolgt die Parallelisierung der Funktionen `copy()` und `replace_if()` nach demselben Schema.

Zu Beginn des Algorithmus wird eine Anfrage an den ThreadPool gestellt, wie viele Threads zur Verfügung stehen. Anschließend wird ein Feld für Zeiger auf entsprechend viele `Task`-Objekte erzeugt, wobei der Hauptprozess wie in Abschnitt 3.1.4 erläutert auch eine Teilaufgabe übernimmt.

```
const unsigned parts(pool->get_num_threads());
Task * tasks[parts - 1];
```

Mit Hilfe der Funktion `get_cache_line_size()` wird darüber hinaus die Größe einer Cacheline miteinbezogen, um in der folgenden Partitionierung false sharing zu vermeiden. `ValueType` bezeichnet hierbei den Datentyp der im jeweiligen Container vorhandenen Elemente, `DiffType` den Datentyp mit dem die Distanz zwischen zwei Iteratoren beschrieben werden kann (in der Regel `unsigned int`, bei Zeigern `ptrdiff_t`) und `Pointer` entspricht einem Zeiger auf `ValueType`.

```
1  const unsigned elements_per_line =
2     pool->get_cache_line_size() / sizeof(ValueType);
3
4  const Pointer a = &(*a_first); // Adresse des 1. Elementes der Sequenz
5  const unsigned long long not_aligned_elements(((unsigned long long)a
6     & (l2_cache_line - 1)) / sizeof(ValueType));
7  DiffType size(dist / parts);
8  // Verwende nun immer Vielfache von elements_per_line
9  size -= (size % elements_per_line);
10 const DiffType first_size(size - not_aligned_elements);
11 const DiffType last_size(dist -((parts - 2) * size) - first_size);
```

Quelltextauszug 3.3: Vorarbeiten für die Partitionierung

Bei der Partitionierung wird für das erste `Task`-Objekt ein Teilintervall der Größe `first_size` verwendet. Anschließend werden *parts* − 2 weitere `Task`-Objekte für die jeweiligen Teilintervalle der Größe `size` des Algorithmus erzeugt, im lokalen Feld abgespeichert und dem ThreadPool zur Ausführung übergeben, wie Quelltextauszug 3.4 zeigt. Das zweite Argument für den `Task`-Konstruktor ist der bereits in Abschnitt 3.1 erläuterte `load`-Parameter. Zusätzlich werden die Iteratoren in jeder Schleifeniteration mittels

`std::advance()` um die Größe der jeweiligen Partition verschoben (hier nicht dargestellt).

```
1 ReplaceCopyIfWorker<DT_, InputIterator, OutputIterator, Predicate>
2    w(a_first, a_end, b_first, pred, value);
3 tasks[i] = new Task(&w, 2 * size * sizeof(ValueType));
4 pool->dispatch(tasks[i]);
```

Quelltextauszug 3.4: Erzeugung der Funktionsobjekte und Dispatching

Der Hauptprozess, bei eingeschalteter Thread Affinity auf dem verbleibenden Kern laufend, führt wie nachfolgend dargestellt den letzten Teil direkt aus, d.h. ohne Einsatz eines Funktionsobjekts und Threads. Anschließend wartet er indirekt über den `ThreadPool` auf die Fertigstellung aller anderen Tasks und gibt den dynamisch allozierten Speicherplatz wieder frei. Das `return`-Statement beendet die Ausführung.

```
1 std::replace_copy_if(a_first, a_end, b_first, pred, value);
2 pool->barrier(tasks, parts - 1);
3 return b_first + last_size;
```

Quelltextauszug 3.5: Funktionsaufruf des Hauptprozesses und Barrier

3.3.3 Numerische Algorithmen

Auf Grund der Wichtigkeit numerischer Algorithmen in der Praxis und ihres häufigen Einsatzes als Evaluatoren der Leistungsfähigkeit diverser Prozessoren wurden alle Algorithmen dieser Klasse reimplementiert. Besonderes Augenmerk wird auf Grund der in Abschnitt 2.3.2 beschriebenen Eigenheiten der Implementierung, neben den eher trivial zu parallelisierenden Algorithmen `inner_product()` und `accumulate()`, auf `partial_sum()` gelegt.

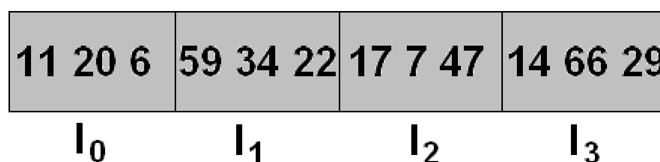

Abbildung 3.16: Beispielhafte Partitionierung eines 12-elementigen Intervalls in vier Teilintervalle

partial_sum. Die Schwierigkeit in der Parallelisierung von `partial_sum()` liegt darin begründet, dass die „Startwerte“ aller Teilintervalle mit Ausnahme des ersten, unbekannt sind. Die Summe über alle Elemente des Intervalls I_0 in Abbildung 3.16 wird benötigt, um

die Werte für das Intervall I_1 und damit auch alle Folgeintervalle korrekt berechnen zu können. Grundsätzlich sind mehrere Ansätze denkbar, wie man diesem Problem begegnet.

Die erste Möglichkeit ist, zunächst überall Null als Startwert zu verwenden. Nach der parallelen Ausführung steht mit dem letzten Element des Intervalls I_0 der Startwert für Intervall I_1 fest, der Startwert für Intervall I_2 ist die Summe der letzten Elemente aus I_0 und I_1, usw. Ein sequentieller Aufruf von `partial_sum()` für diese jeweils letzten Elemente der Intervalle $I_0 \dots I_2$ würde also alle gesuchten Startwerte liefern. Ein paralleler Durchlauf der Intervalle $I_1 \dots I_3$, bei dem elementweise der jeweilige Startwert auf die bereits vorberechneten Summen addiert wird, führt anschließend zum gewünschten Ergebnis.

Ein weiterer Ansatz geht genau umgekehrt vor. Während für das Intervall I_0 die Partialsummen seiner Elemente bereits berechnet werden können (der Startwert ist hier immer Null), wird auf allen anderen Intervallen, mit Ausnahme des letzten, die Funktion `std::accumulate()` ausgeführt, um die Summe der Elemente zu bestimmen. Der Startwert für das Intervall I_1 kann direkt per Dereferenzierung des Iterators auf das letzte Element aus I_0 übernommen werden. Anschließend können aus den berechneten Summen per sequentiellem Aufruf von `partial_sum()` die Startwerte für alle anderen Intervalle berechnet und die parallele Ausführung von `partial_sum()` auf den Intervallen I_1 bis I_3 angestoßen werden.

Der zweite Ansatz ist in der Praxis besser als der erste, denn das vorherige (parallele) Aufsummieren per `std::accumulate()` ist effizienter als das nachträgliche elementweise Addieren eines Startwertes auf vorberechnete Werte. Denn die Funktion `accumulate()` führt n Leseoperationen, n Additionen und eine Speicheroperation für ein Intervall der Länge n aus. Beim ersten Ansatz müssen aber statt einer n Schreiboperationen ausgeführt werden.

An dieser Stelle ist anzumerken, dass auch die Funktionen `accumulate()` und `partial_sum()` sich nur in einer zusätzlichen Schreiboperation pro Element unterscheiden und ansonsten äquivalent sind. Deshalb sind auch die Laufzeiten für In-Cache Größen identisch, gehen für Out-Of-Cache Größen aber zunehmend stark auseinander, wie die Abbildung 3.17 für die sequentielle Ausführung zeigt. Dem einzig aktiven Kern stehen dabei 6 MB (Xeon-System) bzw. 1 MB (Opteron-System) L2-Cache zur Verfügung. Es ist daher, wenn man von äußeren Einflüssen möglicher anderer Prozesse auf die verfügbaren Kerne abstrahiert, sichergestellt, dass die parallelen `accumulate()`-Aufrufe terminieren können, bis der Hauptprozess die Funktion `partial_sum()` auf dem Intervall I_0 ausgeführt hat. Somit stehen ohne signifikanten Overhead im Anschluss alle benötigten Informationen für die parallele Ausführung auf den übrigen Intervallen zur Verfügung.

Während der sequentielle Algorithmus jedes der n Elemente genau einmal betrachtet, müssen parallele Verfahren Mehrarbeit leisten. Betrachten wir nun also den zweiten zuvor beschriebenen Ansatz etwas genauer:

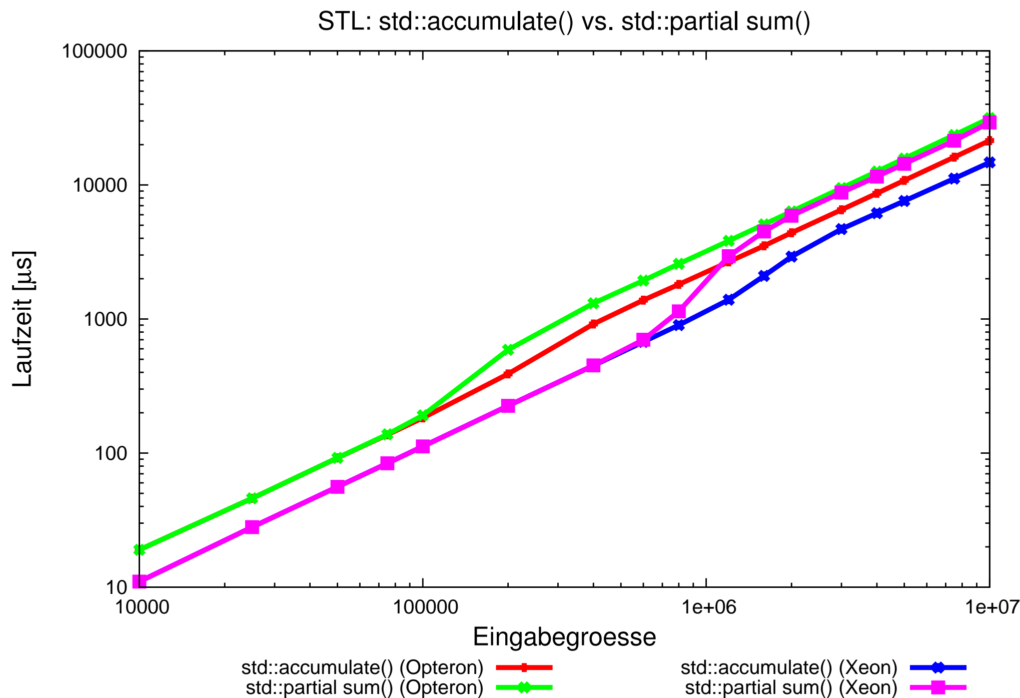

Abbildung 3.17: Sequentielle Laufzeiten für `std::accumulate()` und `std::partial_sum()` auf dem Xeon- und dem Opteron-System

Sei p die Anzahl der Partitionen, dann sind insgesamt p Ausführungen von `partial_-sum()` auf einem Intervall der Länge $\mathcal{O}(\frac{n}{p})$ erforderlich. Zunächst eine Ausführung parallel zu $p-2$ Aufrufen von `std::accumulate()` und danach $p-1$ simultane Ausführungen. Zusätzlich wird ein sequentielles `partial_sum()` für $p-1$ Startwerte benötigt.

Zusammengefasst ergeben sich

$$(p \cdot \frac{n}{p}) + (1 \cdot p - 1) + ((p-2) \cdot \frac{n}{p}) = n + (p-1) + (n - 2 \cdot \frac{n}{p}) = 2n - \frac{2n}{p} + (p-1)$$

Betrachtungen und damit nahezu doppelt so viele wie beim sequentiellen Vorgehen. Des Weiteren macht eine Partitionierung in zwei Teile wenig Sinn, denn während der Berechnung der Partialsummen auf I_0 kann für I_1 keinerlei Nutzinformation berechnet werden. Auch im umgekehrten Fall, bei der Berechnung der Summe des Intervalls I_0 bei paralleler Berechnung der Partialsummen auf I_1 ohne Startwert, müsste ein weiterer Durchlauf folgen. Selbst wenn man, wie im Folgenden beschrieben, noch etwas geschickter vorgeht, ist zu erwarten, dass mit zwei Threads auch im besten Fall nur ein geringer Speedup erreicht werden kann.

Der im Rahmen der Diplomarbeit entwickelte Algorithmus partitioniert das Problem immer in genau einen Teil mehr, als Threads zur Verfügung stehen. Dies ist deshalb sinnvoll, weil in beiden Phasen des Algorithmus jeweils ein Intervall missachtet werden kann. In der ersten Phase handelt es sich dabei um das letzte Intervall, welches nicht mehr durch-

laufen werden muss, um einen Startwert für ein Folgeintervall zu berechnen. In der zweiten Phase kann das erste Intervall unangetastet bleiben, weil dort die Partialsummen bereits in der ersten Phase berechnet wurden. Alle anderen Intervalle werden vom Algorithmus jedoch genau zweimal betrachtet. Tabelle 3.9 veranschaulicht dieses auch von der MCSTL weitgehend verwendete Vorgehen für den Fall drei verfügbarer Threads.

Intervall	I_0	I_1	I_2	I_3
Phase 1	`partial_sum()`	`accumulate()`	`accumulate()`	-
	Hauptprozess	Thread 0	Thread 1	-
Phase 2	-	`partial_sum()`	`partial_sum()`	`partial_sum()`
		Thread 0	Thread 1	Hauptprozess

Tabelle 3.9: Verbesserte Dispatch-Strategie für 3 Threads bei `partial_sum()`

Als weitere Optimierung kann man daher den Algorithmus so konstruieren, dass mit Hilfe der Thread Affinity beide Durchläufe eines Intervalls auf dem gleichen Kern stattfinden, um Cache-Vorteile insbesondere bei kleinen Eingaben auszunutzen. In der ersten Phase verwendet der Algorithmus zwei Threads für die Kerne 1 und 2, die auf den Intervallen I_1 und I_2 `accumulate()` ausführen. Der Hauptprozess selbst läuft auf dem dritten Kern und führt `partial_sum()` auf I_0 aus. Nach einem Barrier und dem sequentiellen `partial_sum()` auf den berechneten Startwerten, arbeiten die beiden Threads wieder auf den Intervallen I_1 und I_2 und führen dort diesmal `partial_sum()` aus. Da für I_0 bereits alle Arbeit verrichtet ist, kann der Hauptprozess nun den verbleibenden Aufruf von `partial_sum()` auf Intervall I_3 übernehmen. Für In-Cache Größen ist dieses Vorgehen effizient, für Out-Of-Cache Größen lassen sich die Ergebnisse jedoch noch verbessern. Denn in Phase 1 agiert der Hauptprozess ausgerechnet auf Intervall I_0, dessen Elemente als erstes aus dem Cache ausgetauscht werden und daher wie in Abbildung 3.17 gezeigt, für `partial_sum()` eine wesentlich höhere Laufzeit benötigt wird. Das Verhältnis zu der parallelen Laufzeit von `accumulate()` stimmt für diesen Fall nicht mehr, die Threads bleiben ohne Arbeit bis der Hauptprozess die Funktion beendet hat. Dies lässt sich vermeiden, indem ein Thread das initiale `partial_sum()` für Intervall I_0 übernimmt, und der Hauptprozess stattdessen die Ausführung von `accumulate()` auf Intervall I_2 als Aufgabe erhält. Das resultierende Verfahren bildet Tabelle 3.10 ab. Zusätzlich konnten mit dieser Strategie diverse Schleifenoptimierungen erfolgen.

SSE-Version von inner_product. Auf Grund des beschriebenen Stellenwertes numerischer Funktionen im Bereich des High Performance Computings und als interessante Zugabe zu den normalen skalaren Algorithmen, wurde die parallele Variante der Funktion `std::inner_product()` zusätzlich vektorisiert. Die hier entwickelte Version nutzt die SSE-Einheiten (*Streaming SIMD Extensions*) aktueller x86-Prozessoren und basiert auf der

Intervall	I_0	I_1	I_2	I_3
Phase 1	`partial_sum()`	`accumulate()`	`accumulate()`	-
	Thread 0	Thread 1	Hauptprozess	-
Phase 2	-	`partial_sum()`	`partial_sum()`	`partial_sum()`
		Thread 1	Hauptprozess	Thread 0

Tabelle 3.10: Dispatch-Strategie für 3 Threads bei `partial_sum()`

Implementierung der Funktion `dot_product()` aus dem HONEI-Projekt [26]. Es erfolgte eine explizite Spezialisierung der mit Hilfe von Templates entwickelten Funktoren für die Datentypen `float` und `double`. Dieses Beispiel soll zeigen, welche Geschwindigkeitsvorteile für konkrete Architekturen möglich sind, wenn auf eine iteratorenbasierte skalare Ausführung zu Gunsten der Nutzung von Vektoreinheiten verzichtet wird. Mit dieser Entscheidung verlässt man allerdings auch die grundlegenden Paradigmen der STL.

3.3.4 Sortierverfahren

Aus dem Bereich der Sortieralgorithmen wurden im Rahmen dieser Diplomarbeit parallele Algorithmen für `std::nth_element()`, `std::sort()` und `std::stable_sort()` entwickelt.

Die Funktion `nth_element()` ist von praktischem Nutzen, wenn die Information ausreichend ist, welches Element an Position n stünde, wenn die gesamte Sequenz sortiert wäre. Diese ist mit Hilfe dieser Funktion in Linearzeit (average case) zu berechnen. Ein schneller paralleler Sortieralgorithmus muss an dieser Stelle nicht motiviert werden, sein Nutzen ist offensichtlich. Wie in Abschnitt 2.3.2 beschrieben, haben die Entwickler der MCSTL in diesem Bereich sehr große Anstrengungen unternommen, um zu guten Ergebnissen zu kommen, daher ist auch hier ein Vergleich äußerst interessant.

Die Sortierverfahren der STL basierten früher hauptsächlich auf einer Quicksort-Routine. Während Quicksort als bestes average-case Sortierverfahren bekannt ist, hat es, anders als beispielsweise Heapsort, eine worst-case Laufzeit von $\mathcal{O}(n^2)$. 1997 zeigte David R. Musser, für welche Art von Eingaben diese Laufzeit erreicht wird und bezeichnete diese als *median-of-three-killer* [38], in Anlehnung an die am häufigsten verwendete und effektivste Art das Pivotelement auszuwählen. Des Weiteren entwickelte er in der Folge ein Verfahren namens *introspective sort*, kurz *Introsort* [38].

Bei Introsort wird, wie bei Quicksort, zunächst bis zu einer bestimmten Rekursionstiefe ein Partitionsalgorithmus aufgerufen. Als guter Schwellenwert für die Tiefe hat sich der Wert $s = \log(2n)$ erwiesen. Wird s erreicht, wechselt der Algorithmus zu Heapsort. Liegt eine für den Quicksort-Algorithmus gute Eingabe vor, dann sind die verbleibenden Teilintervalle nun klein genug, um auch mit Heapsort effizient sortiert werden zu können. Liegen hingegen schlechte Eingaben oder median-of-three-killer Sequenzen vor, dann verhindert

der Wechsel auf Heapsort immerhin eine quadratische Laufzeit. Auch deshalb verwendet `std::sort()` heute dieses Verfahren.

Für sehr kleine ($\leq$ 16 Elemente) oder schon gut vorsortierte Sequenzen lohnt sich in der Regel sogar ein Wechsel zu Insertionsort, während dieser Algorithmus ansonsten auf Grund der quadratischen Laufzeit nicht in Frage kommt. Abbildung 3.18 setzt verschiedene sequentielle und auf Basis der STL implementierte Sortierverfahren für randomisierte Eingaben nochmals visuell in Beziehung.

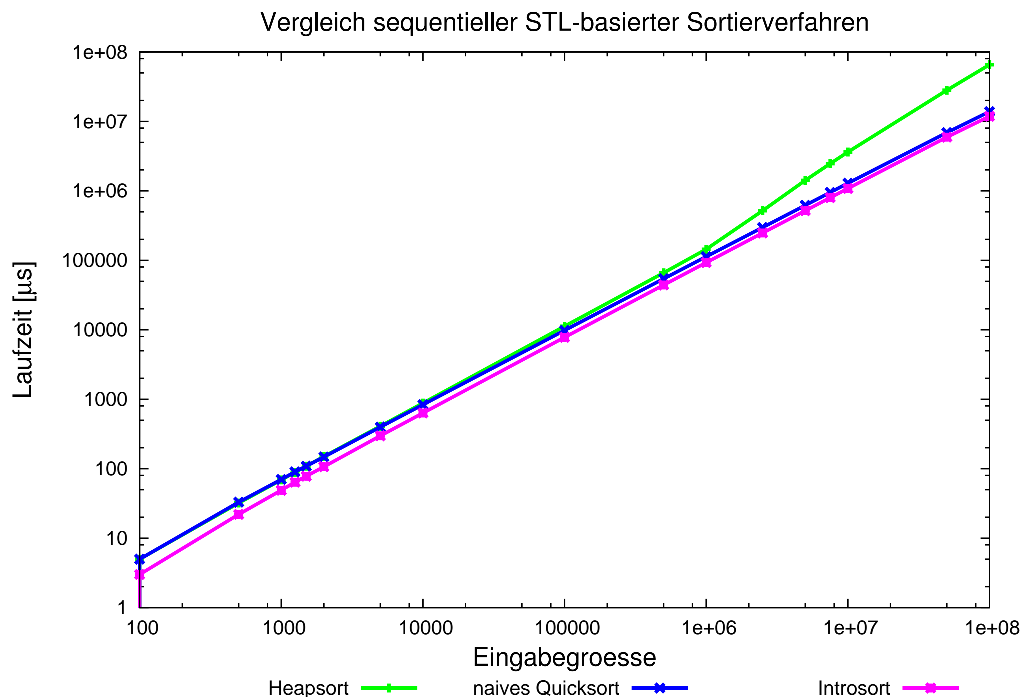

Abbildung 3.18: Laufzeitverhältnisse sequentieller STL-Sortierverfahren (gemessen auf dem Xeon-System)

sort und stable_sort. Während die MCSTL-Entwickler neben einer balancierten und einer unbalancierten Quicksort-Variante ein paralleles *multiway mergesort* (im Bereich der Externspeicheralgorithmen auch bekannt als *k-way mergesort*) implementiert haben, wurde zunächst versucht, den Introsort-Algorithmus auf Grund dessen deutlicher Überlegenheit auch als Basis der im Rahmen der vorliegenden Arbeit entwickelten parallelen Sortierfunktionen zu verwenden. Mit dem resultierenden Verfahren war es möglich, schneller als die Quicksort-Varianten der MCSTL zu sein, jedoch war es der hervorragenden Skalierung des parallelen multiway mergesort klar unterlegen. Aus diesem Grund wurde auch im Rahmen dieser Diplomarbeit ein multiway mergesort Algorithmus implementiert.

Das Verfahren besteht im Wesentlichen aus einer lokalen Sortierphase und einer anschließenden Mergephase mit Intervallen aus der lokalen Sortierphase anderer Threads.

Innerhalb der MCSTL existieren zwei Varianten des Load Balancings, die jeweilige Wahl hängt von den angesprochenen adaptiven Laufzeiteinstellungen (`MCSTL::SETTINGS`) ab. Bei beiden Verfahren erhält jeder der p Threads zunächst eine Partition der Länge $\mathcal{O}(\frac{n}{p})$ einer zu sortierenden Sequenz der Länge n und sortiert sie mit Hilfe von `std::sort()`. Das erste Verfahren verwendet anschließend, ähnlich wie *sample sort* (vgl. z. B. [20]), globale *Splitter* als Schwellenwerte, um die Teilintervalle erneut für die Mergephase zu partitionieren. Auf diese Weise wird sichergestellt, dass Thread i nur Elemente zugewiesen bekommt, die kleiner sind als alle Elemente für Thread $i+1$. Die Größe der resultierenden zweiten Partitionen hängt somit von der (zufälligen) Wahl der Splitter ab und kann zu großen Unterschieden führen, die eine schlechte Lastverteilung nach sich ziehen. Abbildung 3.19 zeigt ein Beispiel für unterschiedlich lange Merge-Intervalle.

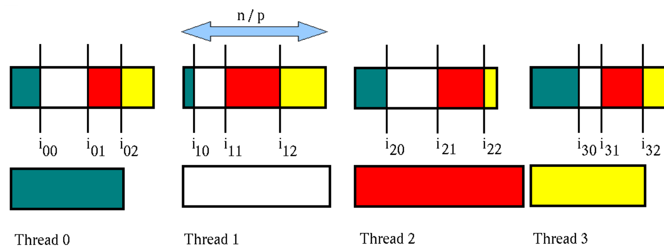

Abbildung 3.19: Multiway mergesort mit Splittern

Wie bereits erwähnt, haben die MCSTL-Autoren sehr viel Energie in die Problematik des parallelen Sortierens investiert. So wurde eine Funktion namens `multiseq_partition()` entwickelt, die genau an diesem Problem ansetzt und erreicht, dass die Merge-Intervalle die gleiche Länge haben, wie die ursprünglichen Intervalle der lokalen Sortierphase. Allgemein berechnet die Funktion für k sortierte Sequenzen $S_0, \ldots, S_{k-1}$ jeweils Trennpositionen $i_0, \ldots, i_{k-1}$ mit $i_j \in S_j$, sodass $\sum_{j=0}^{k-1} i_j = m$. Dabei wird m als *globaler Rang* bezeichnet. Die ersten $p-1$ Threads rufen `multiseq_partition()` mit $m_j = \frac{jn}{p}$ für $j = 1, \ldots, p-1$ auf. Für den letzten Thread bilden immer die Enden der jeweiligen Ausgangspartitionen die entsprechenden Trennpositionen. Somit sind alle Merge-Intervalle eindeutig definiert, gleich lang und somit optimales Load Balancing gegeben. Die Funktion `multiseq_partition()` benötigt eine Laufzeit von $\mathcal{O}(k \log k \cdot \log \max_j |S_j|)$ [45]. Des Weiteren wurden im Rahmen der MCSTL optimierte Funktionen für das multiway merging für die Fälle $k \in [2, 4]$ entwickelt.

Der im Rahmen dieser Diplomarbeit entwickelte Algorithmus adaptiert die Funktion `multiseq_partition()` weitgehend. Lediglich geringfügige Schleifenoptimierungen und Anpassungen an den Hauptalgorithmus sind erfolgt. Dieser unterscheidet sich stärker von dem der MCSTL, einerseits auf Grund des zugrundeliegenden Multithreading-Konzepts und andererseits durch die Vermeidung von mehreren Datenstrukturen (Vektoren, Arrays), die im Wesentlichen die gleichen Nutzinformationen enthalten. Nicht zuletzt wird auch der

Austausch von Informationen zwischen Threads anders realisiert, da im hier entwickelten Algorithmus die Threads mit Ausnahme der Datenstruktur der Teilintervalle (die für `multiseq_partition()` benötigt wird) ausschließlich auf lokalen Arrays arbeiten, während in der MCSTL-Variante mehrere globale Arrays verwendet werden.

Für die `std::merge()`-Aufrufe wird neuer Speicherplatz alloziert, um die ursprünglichen Teilintervalle der Threads zunächst nicht zu überschreiben, da diese von jeweils allen anderen Threads für ihre Merge-Operationen verwendet werden. Der Algorithmus arbeitet folglich nicht *in situ* und benötigt einen Speicherplatz von $\mathcal{O}(n + p^2)$, da jeder Thread zusätzlich jeweils ein Array der Länge p für die Speicherung der Merge-Intervalle benötigt. Nach einem Barrier kann dann die vollständig sortierte Teilsequenz an den Ort der zugewiesenen Partition kopiert werden.

Ein weiterer wesentlicher Vorteil der Verwendung des parallelen multiway mergesort neben der sehr guten Skalierung, ist die einfache Überführung des Algorithmus in einen stabiles Sortierverfahren. Da `std::merge()` stabil arbeitet, kann bereits durch einfaches Ersetzen der `std::sort()`-Aufrufe der Threads durch `std::stable_sort()`-Aufrufe ein stabiler und schneller Sortieralgorithmus realisiert werden.

nth_element. Wie `std::sort()` beruht auch `std::nth_element()` auf dem in [38] entwickelten Verfahren, in diesem Fall auf *Introselect*. Dabei wird, mit dem gleichen Schwellenwert für die Rekursionstiefe wie bei Introsort, zunächst mit `partition()` der Kreis der möglichen Zielelemente für die n-te Position in der Sequenz eingegrenzt. Ist die Position `split`, also das Ergebnis des jeweiligen Aufrufes von `partition()`, kleiner als die durch den Iterator `nth` definierte, dann erfolgt der nächste Aufruf für das Intervall $[middle, end]$ und ansonsten für das Intervall $[first, middle]$. Sinkt die Größe des betrachteten Intervalls während dieses Vorgangs unterhalb vier Elemente, werden diese sortiert und der Algorithmus terminiert. Ansonsten wird im Fall des Erreichens der maximalen Rekursionstiefe ein Heap für das Intervall $[first, nth]$ erzeugt und mittels des STL-internen Heapselect das korrekte Element für die n-te Position bestimmt.

Die hier entwickelte Implementierung ist an das ursprüngliche Introselect angelehnt und parallelisiert die Aufrufe von `partition()`. Dies führt für zunehmende Eingabegrößen bereits zu akzeptablem Speedup und zu guten Ergebnissen im Vergleich zur Implementierung der MCSTL. Eine Verfeinerung des sequentiellen heapbasierten Teils oder ein vollständig anderer Ansatz kann weitere Laufzeitgewinne ermöglichen.

Kapitel 4

Ergebnisse

4.1 Testumgebung

4.1.1 Systeme

Als Plattformen wurden in erster Linie die an den Fakultäten Informatik und Mathematik verfügbaren Intel Xeon und AMD Opteron Systeme verwendet. Zusätzlich, um Messdaten auf einem Nicht-x86-System zu erhalten, kamen die beiden Power Processing Units (PPU) eines IBM Cell Blade QS22 zum Einsatz. Es handelt sich dabei um dual-threaded 64-bit CPUs der fünften PowerPC-Generation, die in Architektur und Befehlsatz vom PowerPC 970 Prozessor abgeleitet wurden. Jedoch ist die Anbindung der Prozessoren zum Speicher anders realisiert, nämlich über den Element Interconnect Bus der Cell Architektur [34]. Die Prozessoren untereinander sind über fünf FlexIO-Kanäle verbunden, die eine Gesamtbandbreite von 20 GB/s zwischen zwei Prozessoren zur Verfügung stellen. Einen kurzen Überblick über die Architektur bietet [13]. Die wichtigsten Daten dieser Systeme werden in Tabelle 4.1 zusammengefasst. Dabei beziehen sich die Angaben zu L1-Caches immer auf einen einzelnen Kern und Angaben zu L2-Caches immer auf eine CPU. Die Zeile „Threads / Kern" beschreibt die Anzahl der hardwareseitig simultan unterstützten Anzahl von Threads (SMT).

4.1.2 Betriebssysteme und Compiler

Die zentralen Kenndaten der auf den Testsystemen installierten Betriebssysteme und Compiler, sowie die verwendeten Compilerflags fassen Tabelle 4.2 und 4.3 zusammen. Für die Messungen zur MCSTL wurde die Version 0.8.0-beta verwendet, da die Einflussnahme auf für die Vergleichbarkeit der Ergebnisse essentielle Parameter, wie z. B. die Anzahl der eingesetzten Threads innerhalb des GNU parallel mode, zu Beginn der Ergebnismessungen noch nicht hinreichend dokumentiert war. Kurz vor Abgabe der vorliegenden Arbeit durchgeführte Tests mit entsprechender Manipulation dieser Parameter (streng nach Ka-

Eigenschaft	Xeon-System	Opteron-System	Cell Blade QS22
Hersteller	Intel	AMD	IBM (STI)
Prozessor-Typ	Xeon E5430	Opteron 2214	PowerXCell 8i (PPU)
Taktfrequenz	2,66 GHz	2,2 GHz	3,2 GHz
CPUs	2	2	2
Kerne / CPU	4	2	1
Threads / Kern	1	1	2
L1 Daten (Kern)	32 KB	16 KB	32 KB
L1 Instr. (Kern)	32 KB	16 KB	32 KB
L2 (CPU)	2x 6 MB	2x 1 MB	512 KB
Cacheline	64 B	64 B	128 B
Hauptspeicher	8 GB	8 GB	8 GB

Tabelle 4.1: Für die Benchmarks verwendete Testsysteme

pitel 31 der aktuellen Manual-Beschreibung mit `omp_set_num_threads()` und `__gnu_parallel::_Settings`) konnten auf dem Opteron-System bei Einsatz der GCC-Version 4.3.2 darüber hinaus zu keinen zuverlässigen bzw. realistischen Ergebnissen führen.

Eigenschaft	Xeon-System	Opteron-System	Cell Blade QS22
Betriebssystem	Ubuntu Linux	Suse Linux 10.2	Yellow Dog Linux
Adresslänge	64-bit	64-bit	64 bit
Kernel	2.6.24-16-generic	2.6.16.21-smp	2.6-18-92.e15
GNU Compiler	g++ 4.2.3	g++ 4.2.2	g++ 4.2.4
Intel Compiler	-	icc 10.1.015	-

Tabelle 4.2: Auf den Testsystemen installierte Betriebssysteme und Compiler

Ein vollständiger Befehl zur Kompilierung der Benchmarks für den entwickelten Code lautet beispielhaft für `accumulate()` und den GNU Compiler:

```
g++ accumulate_benchmark.cc mutex.cc lock.cc synchronisation.cc thread.cc
exception.cc -O2 -march=ARCH_TAG -CPU_TAG -pthread
```

Das Architekturtag `ARCH_TAG` ist dabei von System und Compiler-Version abhängig. Bei anderen als x86-Prozessoren muss statt `-march` oft `-mcpu`, bzw. `-mtune` verwendet werden. Für das `CPU_TAG` ist neben den in Tabelle 4.3 aufgeführten noch das Tag `DT1` für den Sun UltraSPARC T1 Prozessor möglich, für den das Einschalten der Thread Affinity im Fall eines Solaris-Betriebssystems allerdings wirkungslos bleibt, so lange das Programm nicht mit Superuser-Rechten kompiliert und ausgeführt wird. Um für diese Systeme eine optimale Leistung zu erzielen, sollten darüber hinaus ggf. einige Werte in der Struktur `Architecture` angepasst werden. Der Grund für die Unterscheidung zwischen AMD und

System	MPTL / eigener Code	MCSTL
Xeon-System	`-pthread -O2 -march=nocona -DINTEL`	`-fopenmp -O3 -march=nocona -Imcstl/c++`
Opteron-System	`-pthread -O2 -march=opteron -DAMD`	`-fopenmp -O3 -march=opteron -Imcstl/c++`
Cell Blade	`-pthread -O2 -mcpu=power5+ -DPPU`	`-fopenmp -O3 -mcpu=power5+ -Imcstl/c++`

Tabelle 4.3: Verwendete Compiler-Attribute

Intel ist eine Differenz in der Bereitstellung von L1-Cache-Informationen mittels des `cpuid`-Befehls.

4.1.3 Testbedingungen

Für die jeweiligen Benchmarks wurden die Eingaben auf folgende Art und Weise erzeugt: Für jede getestete Eingabegröße und jede Iteration werden neue Container vom Typ `std::vector<float>` konstruiert, per Pseudozufallsgenerator initialisiert und der Algorithmus auf dieser Eingabe ausgeführt. Wird mehr als ein Container benötigt, so erfolgt die Initialisierung abwechselnd, d.h. bei Out-Of-Cache Größen sind die Enden der Container noch im Cache und nicht nur ein (größeres) Teilstück eines Containers. Dieser Prozess wird abhängig davon, wie viele Iterationen notwendig sind, um durch Berechnung des arithmetischen Mittels zu stabilen Messwerten zu kommen, mindestens 100 Mal wiederholt. Dieses in Pseudocode 4.1 nochmals zusammengefasste Vorgehen entspricht einem realistischen Szenario. Die Nutzdaten werden erzeugt, und verbleiben anschließend (zum Teil) im Cache des Kerns, auf dem der Hauptprozess ausgeführt wird. Die häufige Wiederholung führt zu stabileren Ergebnissen, die den Einfluss von Seiteneffekten minimiert. Eine mehrmalige Ausführung auf ein und derselben Eingabe wird nicht als realistisch bewertet, da ab der zweiten Iteration die Daten nicht nur im Cache des den Hauptprozess ausführenden Kerns, sondern in allen Caches bereit stünden. Die Laufzeiten würden somit gegenüber der realen einmaligen Ausführung einer Funktion zu gering ausfallen und die im Rahmen dieser Arbeit entwickelte Implementierung könnte auf Grund der Thread Affinity einen noch wesentlich höheren Nutzen aus der Cache-Wiederverwendung ziehen, als die zum Vergleich herangezogenen Bibliotheken. Der Speedup wird gegenüber der sequentiellen STL-Funktion beziffert.

Da die MCSTL-Funktionen mit einem höheren Optimierungslevel ausgeführt werden, wurde bei jeder Funktion genau beobachtet, ob gleiche Voraussetzungen für alle Konkurrenten vorliegen. Bei den im Rahmen dieser Arbeit entwickelten parallelen Funktionen wird auch der Hauptprozess auf einen bestimmten Kern gelegt, weshalb die STL-Zeiten

Pseudocode 4.1 Benchmarkverfahren

```
for all sizes do
  for all number of threads do
    TimeStamp a, b;
    time = 0;
    for number of iterations do
      std::vector<float> v1(size);
      std::vector<float> v2(size);
      for all x = v1[i], y = v2[i] do
        x = float(rand());
        y = float(rand());
      end for
      if (num_threads == 1) then
        a.take();
        std::_function_(v1.begin(), v1.end(), v2.begin(), ...):
        b.take();
      else
        a.take();
        parallel_function_(v1.begin, v1.end(), v2.begin(), ...);
        b.take();
      end if
      time += difference(a, b);
    end for
    print(time / iterations);
  end for
end for
```

nochmals durch Benchmarks ohne nachfolgende parallele Ausführung auf ihre Richtigkeit hin überprüft wurden. Wichen sie bei `-O2` und `-O3` voneinander ab (vorwiegend der Fall bei komplexeren Funktionen wie `nth_element()`, `sort()` oder `partition()`), so wurden die jeweiligen unterschiedlichen Messwerte für die Berechnung des Speedups zugrundegelegt. Dies bedeutet, dass in diesen Fällen die MCSTL mit den `O3`-Werten ins Verhältnis gesetzt wurden, die MPTL und der hier entwickelte Code mit den `O2`-Werten. Liegt kein Unterschied vor (dies ist bei den meisten linear auf dem Speicher arbeitenden Algorithmen der Fall), wurde darauf geachtet, dass alle Speedupberechnungen auf identischen und stabilen Zeiten basieren. Bei den genannten komplexeren Funktionen ist darüber hinaus die Eingabe für die Laufzeit entscheidend, da sie z.B. bei `partition()` über die Anzahl der notwendigen Swap-Operationen entscheidet. Aus diesem Grund wurden bei allen diesen Funktionen *Seeds* benutzt, damit der Pseudozufallsgenerator die gleichen Eingaben für

alle zu vergleichenden Implementierungen erzeugt. Die Wahl der Eingabegrößen für die im Folgenden dargestellten Diagramme fiel auf das Intervall $[10000, 10^7]$. Bei dieser großen Spannweite bietet sich eine logarithmische Skalierung der x-Achse an, da sonst der hier mit besonderem Augenmerk versehene Bereich der kleinen Eingabegrößen zu klein dargestellt würde. Auf diese Weise kann man beispielsweise besser erkennen, wann die Linie der sequentiellen Laufzeit überschritten wird.

4.1.4 Zeitnahme

Gemessen wird die absolute Laufzeit des vollständigen Funktionsaufrufs. Es findet kein nachträglicher Abzug, etwa von Latenzen durch interne Funktionsaufrufe, statt, um die Testbedingungen möglichst nah an der realen Situation eines Anwenders auszurichten.

Zur Messung der Laufzeiten wurden Aufrufe der C-Funktion `gettimeofday()` verwendet. Diese ist der Standard zur Zeitnahme im Bereich von Mikrosekunden und wird beispielsweise auch von der MCSTL und intern bei den *Intel Threading Building Blocks* [15] verwendet. Die Aufrufe dieser Funktion wurden zur einfacheren Handhabung in einer Klasse `TimeStamp` gekapselt. Alternative Messungen mit `clock_gettime()`, deren verwendete Struktur auch die Ausgabe von Nanosekunden erlaubt, führten zu denselben Ergebnissen.

4.2 Benchmarkergebnisse

In diesem Abschnitt erfolgt die Darstellung der Ergebnisse des implementierten Konzeptes im Vergleich mit der Leistung der MCSTL und der MPTL für die in Abschnitt 3.3 ausgewählten Algorithmen. Die Reihenfolge der Betrachtung entspricht dabei ebenfalls der dieses Abschnitts. Der im Rahmen dieser Arbeit entwickelte Programmcode wird dem verwendeten Namensraum entsprechend im Folgenden als ParallelSTL bezeichnet.

4.2.1 Nicht-mutierende Algorithmen

find. Beim `find()`-Algorithmus hängt die Laufzeit maßgeblich von der Position des gesuchten Elements ab. Ergebnisse für randomisierte Eingaben sind daher nicht aussagekräftig genug. In den Benchmarks zu den folgenden Abbildungen wurde die Eingabegröße fest auf 10^7 Elemente vom Typ `float` gesetzt und die Laufzeit für unterschiedliche Positionen des gesuchten Elements ermittelt, welche im Unterschied zu allen anderen Funktionen daher anstelle der Eingabegröße auf der x-Achse der folgenden Diagramme abgetragen ist.

Die MPTL erzielt selbst bei dieser verhältnismäßig groß gewählten Eingabe unabhängig von der Position des gesuchten Elements keinen signifikanten Speedup auf dem Xeon-System. Wie Abbildung 4.1 zeigt, zeichnet sich erst im Fall der Positionierung ganz am Ende des Testvektors ein steiler Anstieg der Leistung ab, sodass die sequentielle Laufzeit geringfügig unterboten werden kann. Es wird deutlich, dass der naive Ansatz ohne

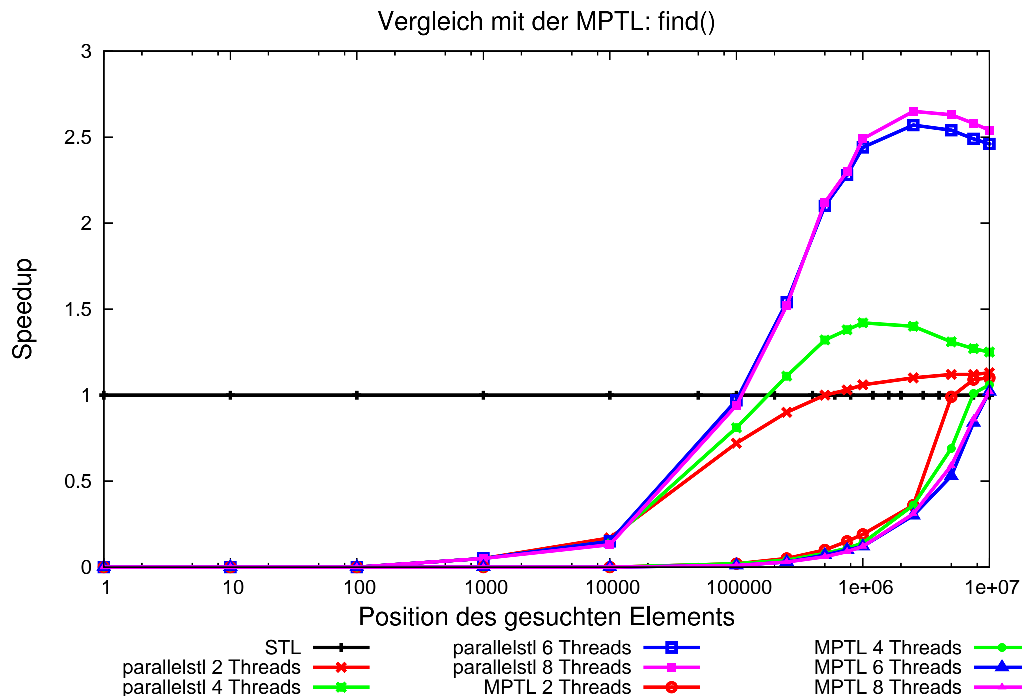

Abbildung 4.1: Xeon-System: Vergleich mit der MPTL für die Funktion `find()` abhängig von der Position des gesuchten Elements

die Möglichkeit der vorzeitigen Unterbrechung parallel arbeitender Threads, wie in Abschnitt 3.3 beschrieben, ineffizient ist. Der Speedup für zwei Threads ist zwar auch für die ParallelSTL-Variante nicht höher, jedoch wird die sequentielle Laufzeit deutlich früher erreicht. Mit vier, sechs und acht Threads geschieht dies auf Grund der stärkeren Partitionierung bereits bei der Position 10^5, bei einem steilen Anstieg für noch spätere Positionen und einem maximalen Speedup von $2,63$.

Für sehr frühe Positionen macht sich das zunächst sequentielle Vorgehen des MCSTL-Algorithmus bezahlt, der aber in der Folge nur sehr geringen Speedup auf diesem System erzielen kann (siehe Abbildung 4.2). Auch der im Rahmen dieser Arbeit entwickelte `find()`-Algorithmus geht für einige Elemente zunächst sequentiell vor, jedoch ist der Overhead bis zum internen STL-Aufruf verglichen mit der MCSTL offenbar zu hoch. Während die STL das gesuchte Element in weniger als einer Mikrosekunde gefunden hat, benötigt der parallele Algorithmus ca. 40 μs.

Der angesprochene Overhead fällt auf dem Opteron-System nicht an, wie die Abbildungen 4.3 und 4.4 zeigen. Erneut übertrifft die Leistung der MPTL erst für die letzten Positionen die Leistung des sequentiellen Algorithmus. Das MCSTL-Verfahren und der ParallelSTL-Algorithmus erreichen mit zwei Threads bereits etwa bei Position 10^5 Speedup, die MCSTL bleibt in der Folge allerdings unabhängig von der Anzahl verwendeter Threads unterlegen. Der maximale erreichte Speedup beträgt $2,54$ mit vier Threads.

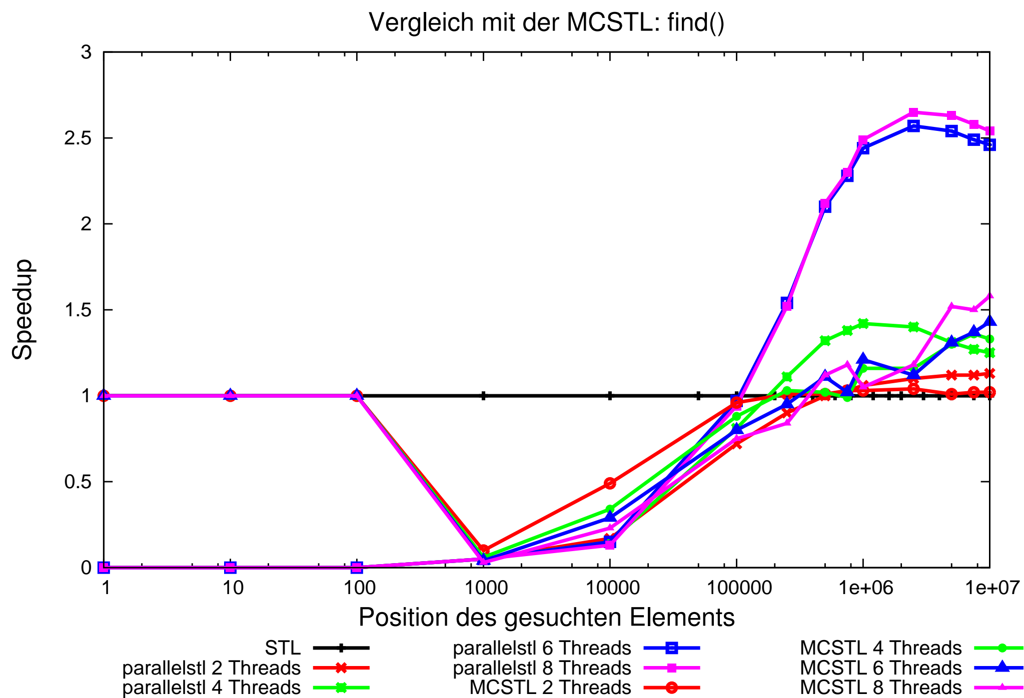

Abbildung 4.2: Xeon-System: Vergleich mit der MCSTL für die Funktion `find()` abhängig von der Position des gesuchten Elements

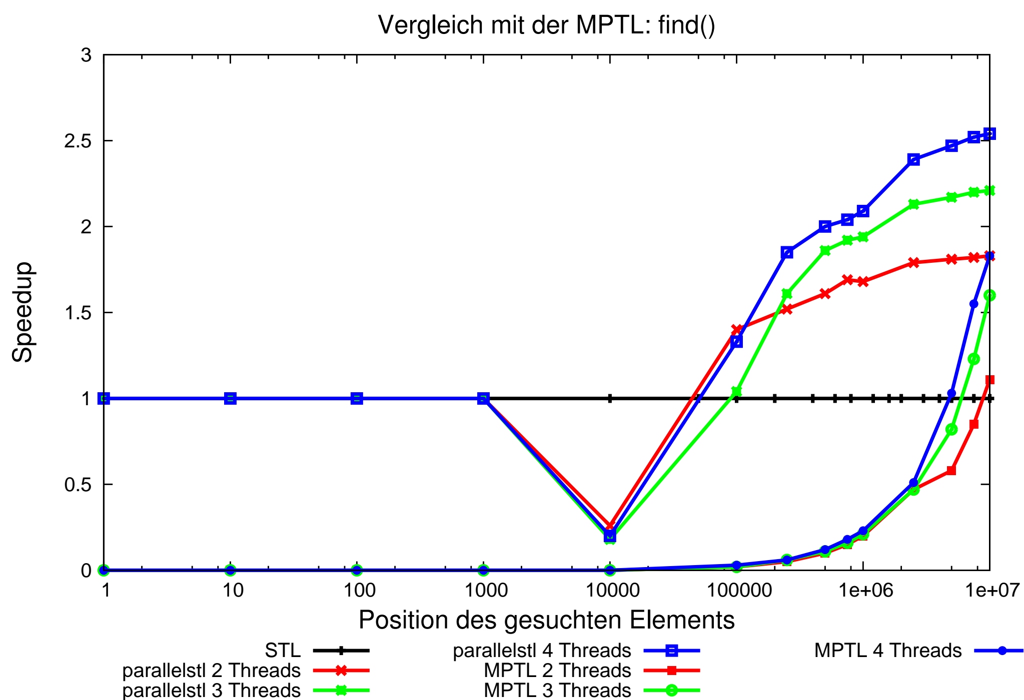

Abbildung 4.3: Opteron-System: Vergleich mit der MPTL für die Funktion `find()` abhängig von der Position des gesuchten Elements

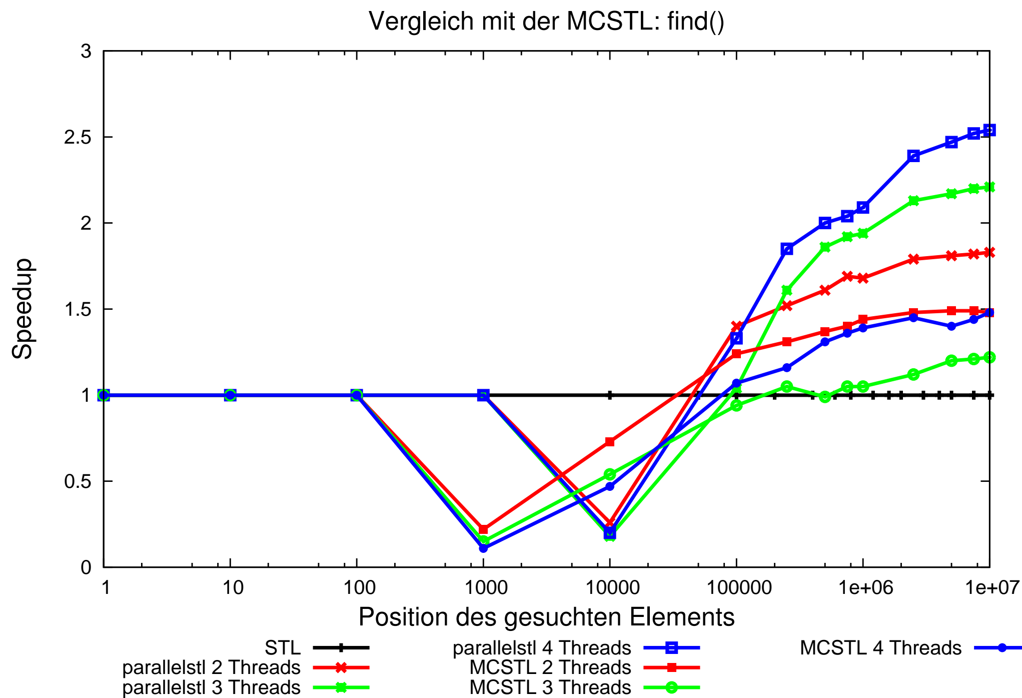

Abbildung 4.4: Opteron-System: Vergleich mit der MCSTL für die Funktion `find()` abhängig von der Position des gesuchten Elements

Auf dem Cell Blade zeigt sich in Abbildung 4.5 dieselbe Situation für die MPTL, wie bei den beiden x86-Systemen, allerdings mit deutlich steilerem Anstieg der Leistung für die Positionen am Ende des Vektors. Bereits bei Stelle 50000 erreicht das ParallelSTL-Verfahren Speedup, der im Anschluss zunächst steil ansteigt und für die letzten Positionen eine Tendenz zur Stagnation zeigt. Der MCSTL-Algorithmus erzielt auf diesem System bei etwas früheren Positionen Speedup, erreicht seine maximale Leistung aber bereits bei $4 \cdot 10^5$ Elementen und wird in der Folge durch das entwickelte Verfahren noch übertroffen (siehe Abbildung 4.6). Der maximale Speedup auf dem Cell Blade beträgt $2,96$.

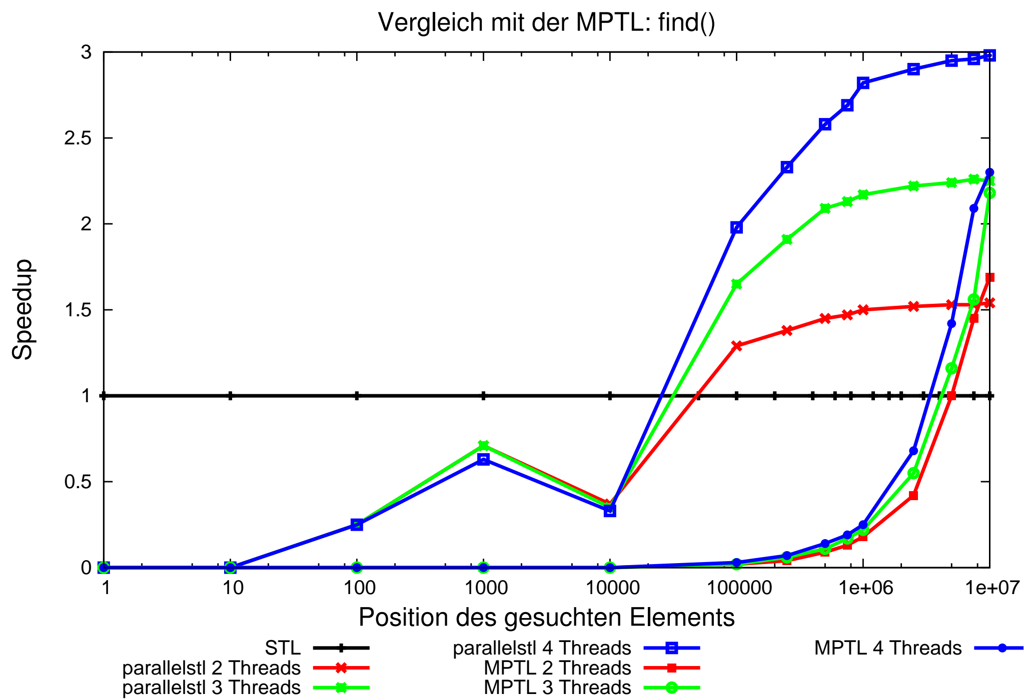

Abbildung 4.5: Cell Blade: Vergleich mit der MPTL für die Funktion `find()` abhängig von der Position des gesuchten Elements

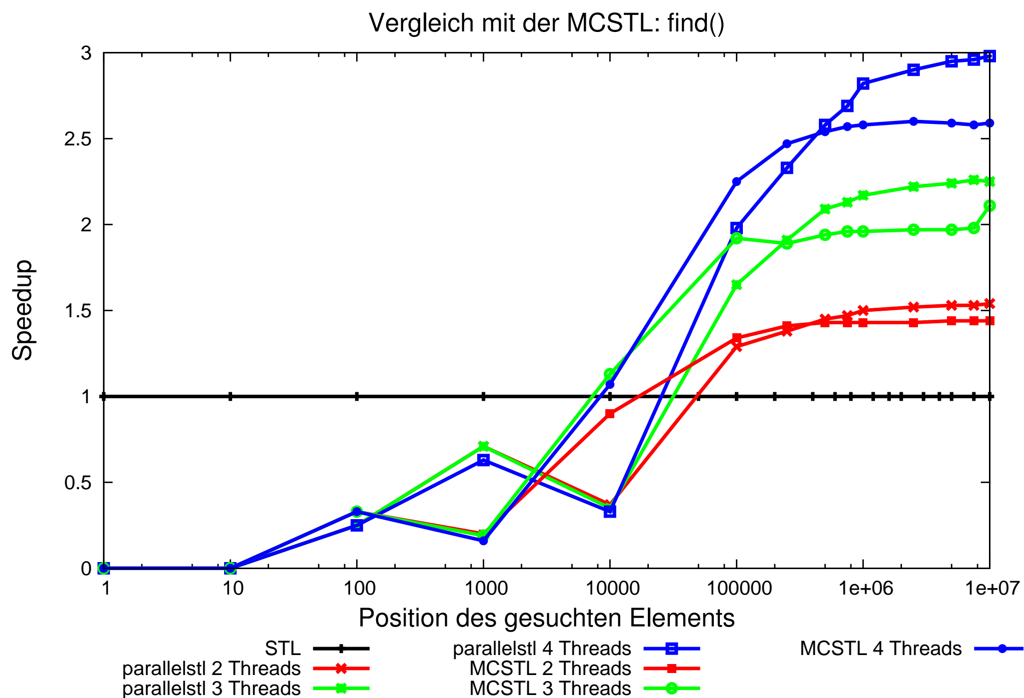

Abbildung 4.6: Cell Blade: Vergleich mit der MCSTL für die Funktion `find()` abhängig von der Position des gesuchten Elements

4.2.2 Mutierende Algorithmen

replace_copy_if / replace_if. Die Benchmarks zu den beiden Algorithmen `replace_if()` und `replace_copy_if()` wurden mit Hilfe von Seeds erzeugt, um jederzeit reproduzierbare und einheitliche Eingaben für alle Messungen zu erhalten. Dies ist wichtig, da die Laufzeit maßgeblich davon beeinflusst wird, für wie viele Elemente das übergebene Prädikat zutrifft und ob die Sprungvorhersage ein mögliches Muster erkennen kann. Die Initialisierung erfolgte zudem abweichend von Pseudocode 4.1 bei `replace_copy_if()` nur für den ersten Eingabevektor, da die typische reale Anwendung der Funktion das Kopieren einer Sequenz in eine andere bei Ersetzung bestimmter Werte ist. Es würde daher keinen Sinn machen, die Zielsequenz zuvor mit anderen Werten zu initialisieren. Das für die Ersetzungen zu verwendende Element kann im Cache gehalten werden und verursacht im Fall von `replace_copy_if()` kaum höhere Kosten, da prinzipiell pro Iterationsschritt eine Schreiboperation anfällt. Bei `replace_if()` hingegen ist erst im Fall einer solchen Ersetzung überhaupt eine Schreiboperation erforderlich, während andernfalls die Eingabesequenz invariant bleibt. Durch die zusätzliche Schreiboperation ist `replace_copy_if()` daher wesentlich stärker vom Speicherdurchsatz abhängig, wie Abschnitt 3.2.4 zeigte.

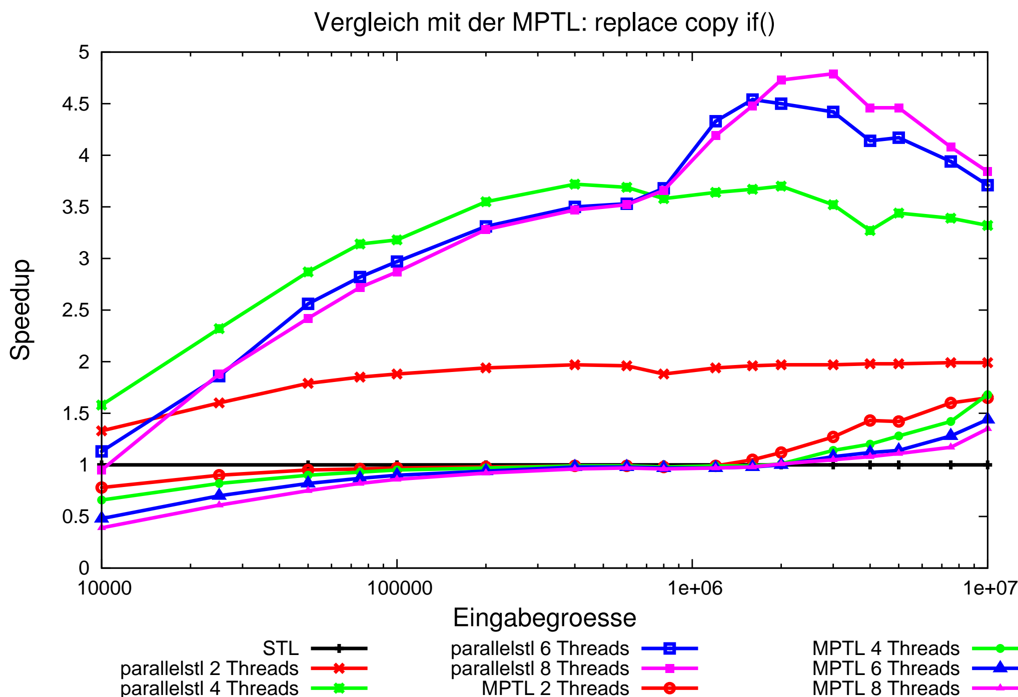

Abbildung 4.7: Xeon-System: Vergleich mit der MPTL für die Funktion `replace_copy_if()`

Da sowohl für die sequentielle, als auch für die parallele Funktion die Elemente der Zielsequenz nicht im Cache vorliegen, kann bereits für kleinste Eingabegrößen Speedup erzeugt werden, wie Abbildung 4.7 zeigt. Liegt ab ca. $1,2 \cdot 10^6$ Elementen auch bezüglich der

Quellsequenz eine zunehmende Out-of-Cache Situation für den sequentiellen Algorithmus vor, kann die Leistung der parallelen Ausführung nochmals verbessert werden. Während die Beschleunigung bei zwei und vier Threads phasenweise annähernd linear ist, wird mit sechs und acht Threads nur noch geringfügig höherer Speedup erzielt, da die Speicherbandbreite begrenzend wirkt. Nachdem mit partiell im Cache verbleibenden Daten ein gewisses Maximum (hier ein Speedup von $4,76$) erreicht ist, sinkt der Speedup leicht in Richtung seines Niveaus bei vollständigen Out-Of-Cache Situationen, so wie es bei den Benchmarks zur Speicherbandbreite in Abschnitt 3.2.4 zu sehen war. Dieses Verhalten bei den größten gemessenen Eingaben wird sich in der Folge auch bei allen anderen memory bound Algorithmen zeigen. Die MPTL erreicht auf dem Xeon-System für Out-of-Cache Größen nur minimalen Speedup und kaum Skalierung bei Einsatz mehrerer Threads. Ihre Leistung auf dem Opteron-System in Abbildung 4.8 ist jedoch wesentlich besser. Sie bleibt lediglich bei zwei verwendeten Threads ohne Speedup, bei drei und vier Threads lässt sie aber zeitweise den ParallelSTL-Algorithmus mit entsprechend gleich vielen Threads knapp hinter sich. An dieser Stelle zeigt sich das ebenfalls bereits in Abschnitt 3.2.4 dokumentierte Problem für NUMA-Systeme ohne L3-Cache und mit geringer realer Bandbreite zwischen den Prozessoren, welches in gleicher Form bei allen anderen memory bound Algorithmen auftritt. Der Speedup ist für zwei Threads bereits annähernd linear und kann durch mehr Threads nicht mehr übertroffen werden, da die andere CPU zum Einsatz kommt und somit die Speichertransfers wesentlich teurer werden.

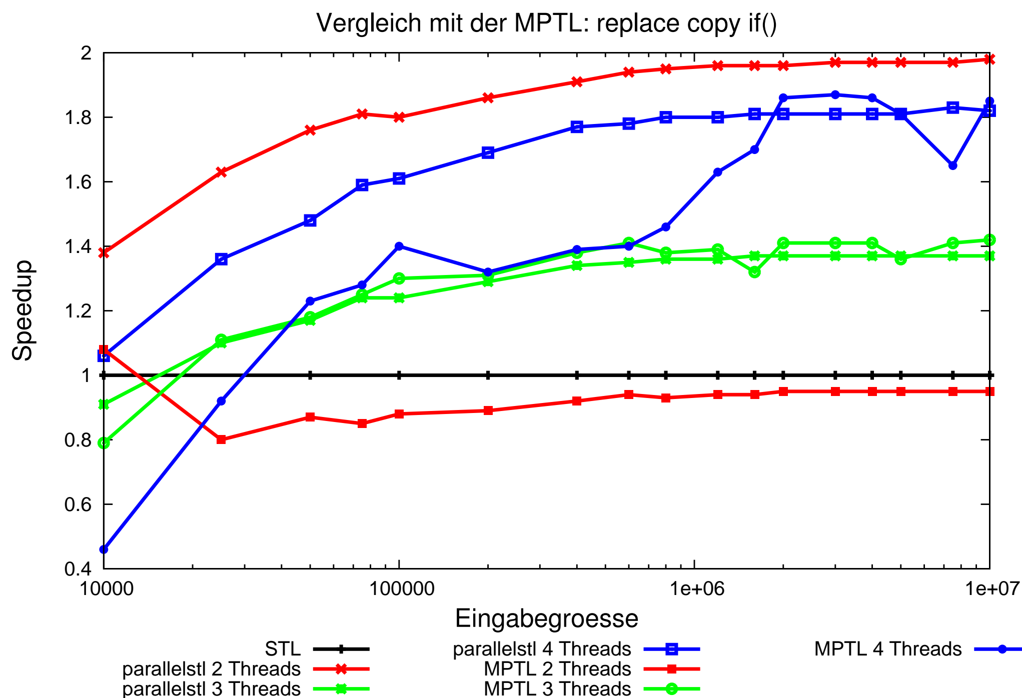

Abbildung 4.8: Opteron-System: Vergleich mit der MPTL für die Funktion `replace_copy_if()`

Auf dem Cell Blade kann die MPTL für größere Eingaben und zwei Threads exakt die Leistung des ParallelSTL-Ansatzes erreichen, für drei und vier Threads ist sie jedoch deutlich unterlegen. In Abbildung 4.9 ist eine wesentlich bessere Skalierung durch Einsatz mehrerer Threads zu erkennen als zuvor, da es sich beim Cell Blade zwar wie beim Opteron-System um ein NUMA-System handelt, der maximale Durchsatz aber, wie in Abschnitt 3.2.4 gezeigt, durch die beiden PPUs allein nicht ausgereizt werden kann. Ebenfalls erkennbar ist der etwas frühere Anstieg des Speedups durch das Verlassen der In-Cache Größen, die im Fall der PPUs nur halb so groß ausfallen, wie bei den Opteron-Prozessoren.

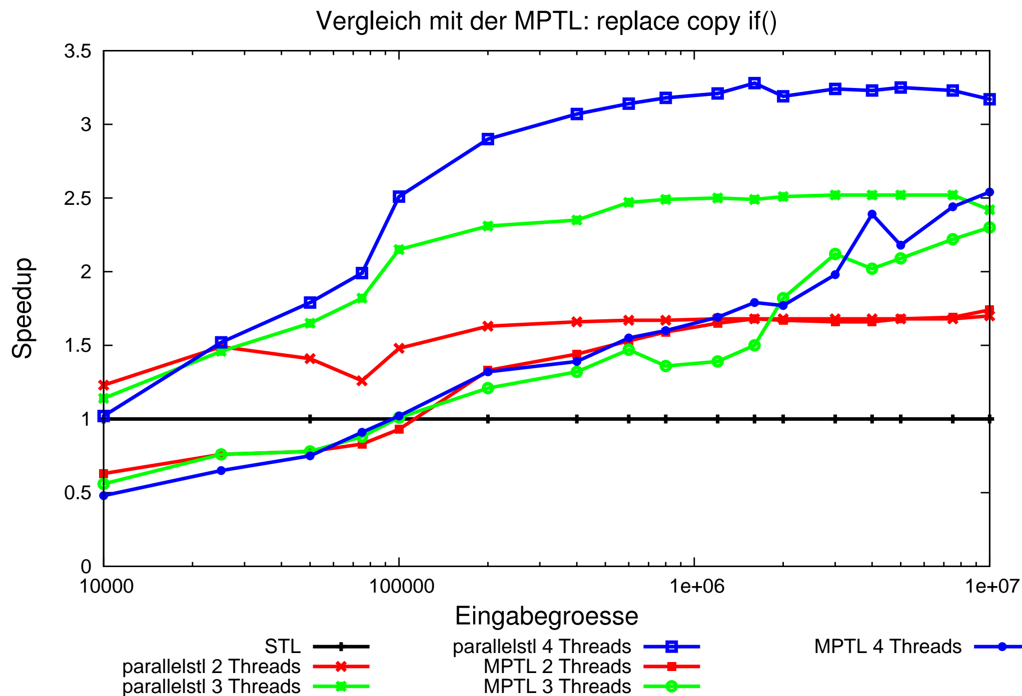

Abbildung 4.9: Cell Blade: Vergleich mit der MPTL für die Funktion `replace_copy_if()`

Für den Vergleich mit der MCSTL wurde, wie in Abschnitt 3.3 erläutert, die Funktion `replace_if()` herangezogen, welche in situ arbeitet und deren Laufzeit dadurch etwas weniger vom Speicherdurchsatz abhängt. Die möglichen Speedups sind daher deutlich höher, wie Abbildung 4.10 für das Xeon-System zeigt. Eine Beschleunigung wird im Fall des ParallelSTL-Konzepts erneut bereits für die kleinsten Eingabegrößen erreicht, fällt aber zunächst etwas geringer aus, als bei `replace_copy_if()`, da der sequentielle Algorithmus durch den Cache und die ausschließlichen Leseoperationen im Vorteil ist. Für zwei und vier Threads ist das im Rahmen dieser Arbeit entwickelte Verfahren über alle gemessenen Eingabegrößen überlegen, für sechs und acht Threads ist die MCSTL jedoch ab ca. $4 \cdot 10^5$ Elementen schneller. Ab $5 \cdot 10^6$ Elementen dreht sich dieses Verhältnis um, da der Speedup des ParallelSTL-Ansatzes im Out-Of-Cache Bereich einen starken Anstieg mit Tendenz

zu linearem Speedup erfährt und den flachen Verlauf der MCSTL-Kurven letztendlich übertrifft.

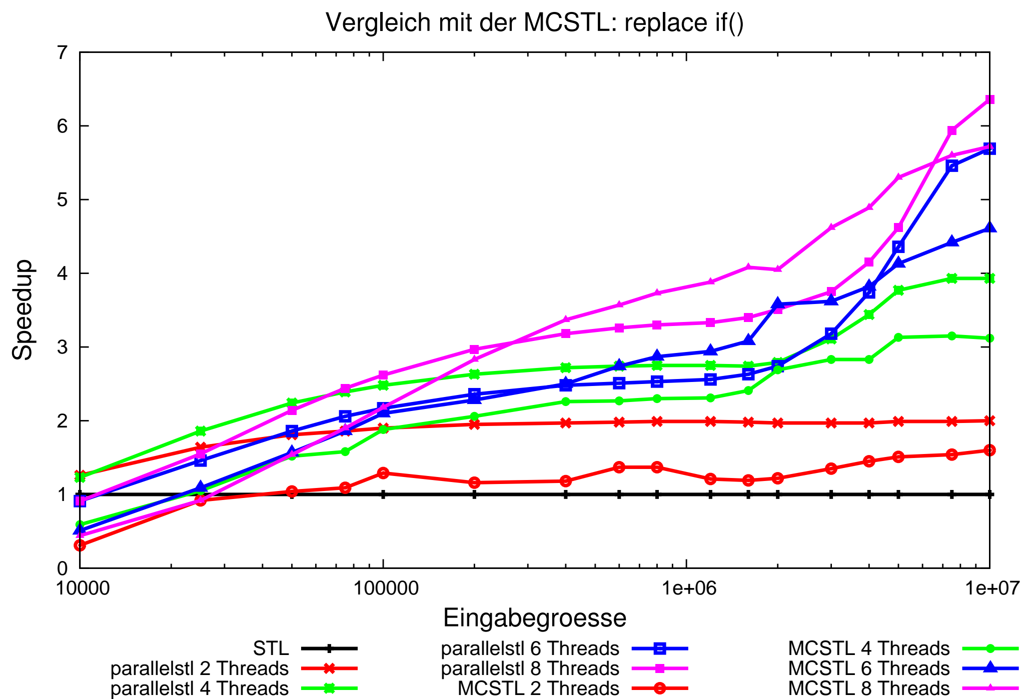

Abbildung 4.10: Xeon-System: Vergleich mit der MCSTL für die Funktion `replace_if()`

Auf dem Opteron-System (Abbildung 4.11) kann hingegen kein deutlich verbesserter Speedup gegenüber `replace_copy_if()` gemessen werden, da er für zwei Threads bereits linear ist und, wie beschrieben, auch mit zusätzlichen Threads nicht mehr übertroffen werden kann. Die Situation für kleine Eingaben ist jedoch identisch, lediglich führt der Cache-Vorteil des sequentiellen Algorithmus hier zu einem etwas späteren Erreichen erstmaligen Speedups. Der MCSTL gelingt mit vier Threads ein geringfügig höherer Beschleunigungsfaktor von $2,09$, sie ist mit zwei Threads jedoch klar unterlegen, während die Leistungen mit drei Threads für Out-of-Cache Größen ebenbürtig sind. Insgesamt benötigt die MCSTL erneut eine deutlich größere Eingabe, um Speedup zu erzielen.

Anders als auf dem Opteron- und wesentlich deutlicher als auf dem Xeon-System kann das ParallelSTL-Verfahren auf dem Cell Blade die Leistung des MCSTL-Algorithmus übertreffen. Während die MCSTL mit drei und vier Threads die sequentielle Laufzeit nur geringfügig unterbietet, wird bereits für die kleinsten gemessenen Eingaben eine Beschleunigung erreicht, ehe der Speedup bei $1,71$, bzw. $2,52$ und $3,31$ konvergiert (Abbildung 4.12).

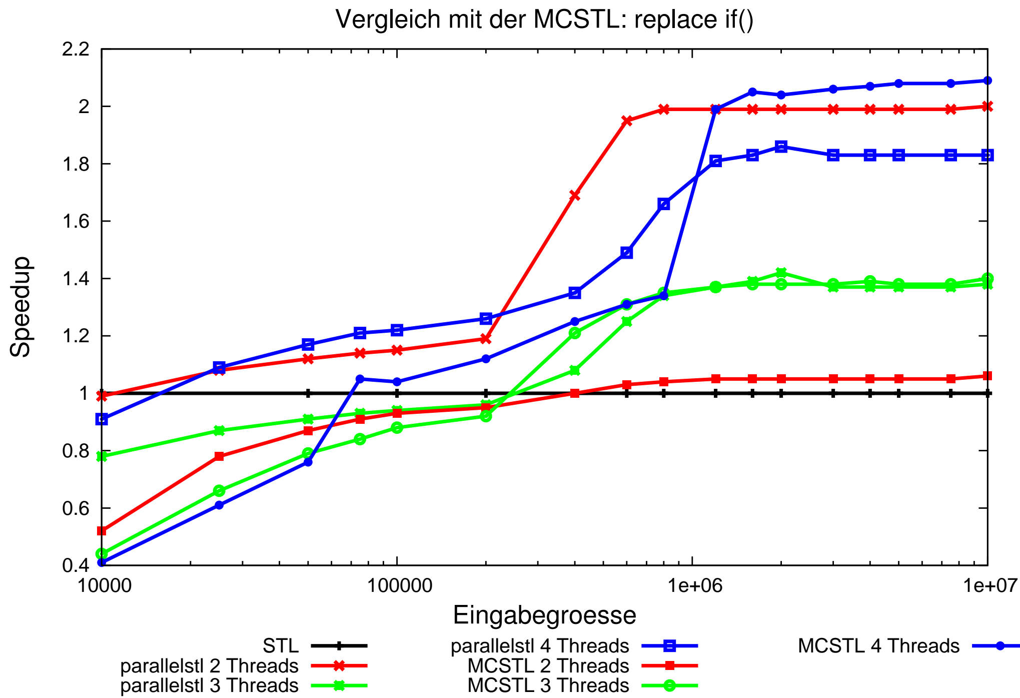

Abbildung 4.11: Opteron-System: Vergleich mit der MCSTL für die Funktion `replace_if()`

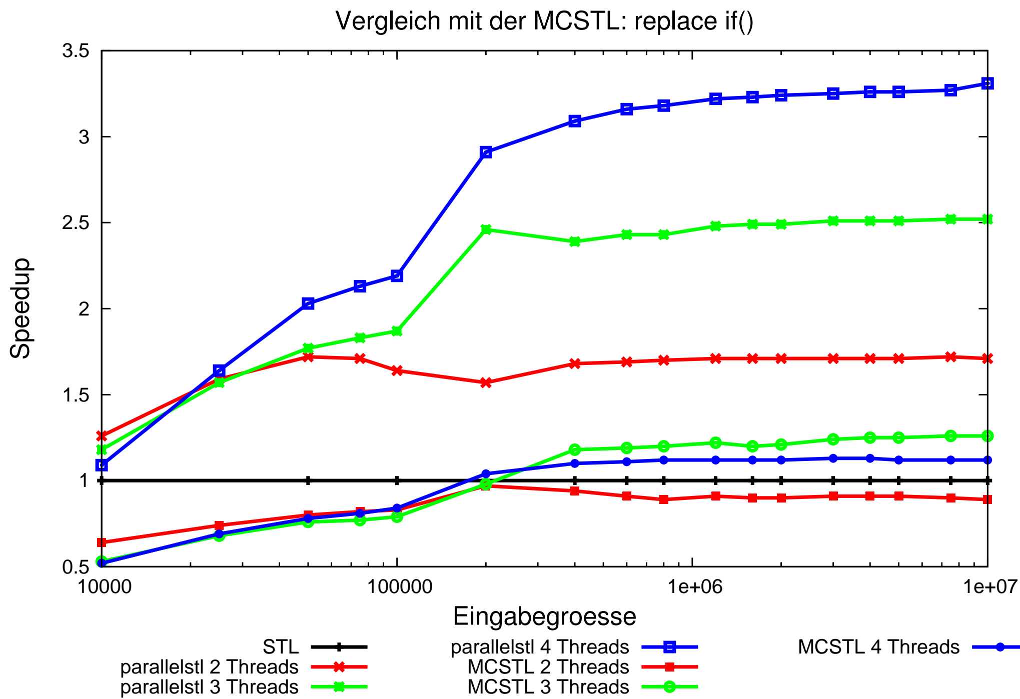

Abbildung 4.12: Cell Blade: Vergleich mit der MCSTL für die Funktion `replace_if()`

partition. Für die Funktion `partition()` werden die MCSTL und das in Abschnitt 3.3 beschriebene Verfahren von Frias et al. zu Vergleichen herangezogen. Sowohl auf dem

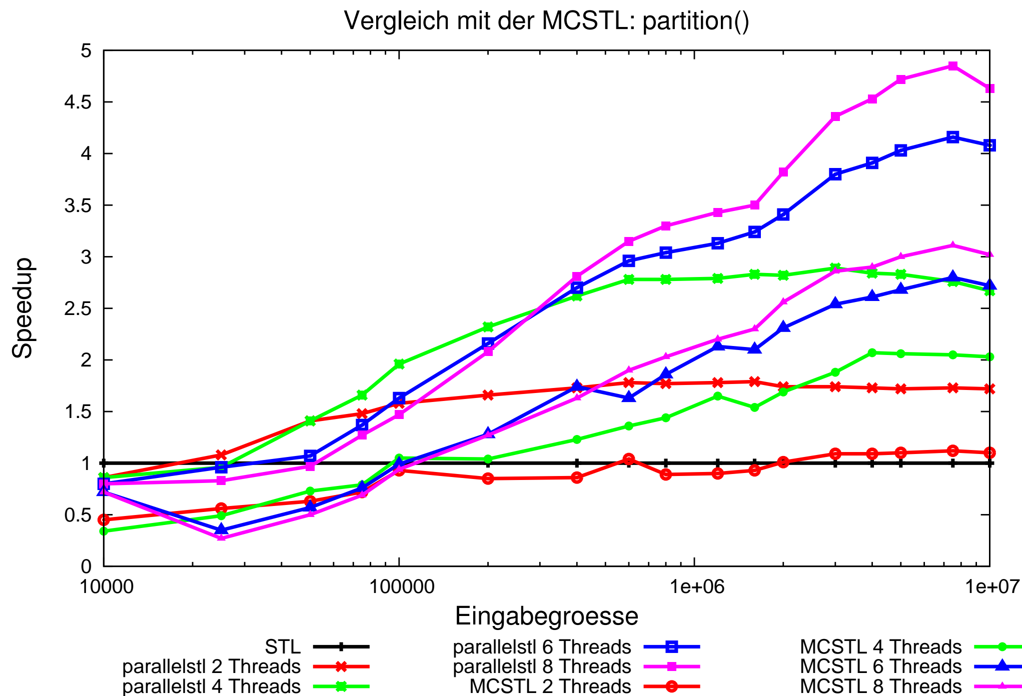

Abbildung 4.13: Xeon-System: Vergleich mit der MCSTL für die Funktion `partition()`

Xeon-System (Abbildungen 4.13 und 4.14), als auch auf dem Opteron-System (Abbildungen 4.15 und 4.16) ist eine deutliche Überlegenheit des PARALLELSTL-Algorithmus zu erkennen. Für sehr kleine Eingaben wird so lange die sequentielle STL-Variante aufgerufen, bis mindestens zwei zu neutralisierende Blöcke in den L1-Cache passen. Sobald dies der Fall ist, wird Speedup auf beiden Systemen erreicht, während die MCSTL und das Verfahren von Frias et al. noch unterhalb der Linie bleiben. Der Algorithmus skaliert bis zu einem Speedup von 5 auf dem Xeon-System und $2,5$ auf dem Opteron-System. Auf Letzterem wird ebenso deutlich, dass bei diesem nicht in erster Linie vom Speichertransfer abhängigen Algorithmus die Laufzeiten für drei und vier Threads besser sind, als für zwei Threads. Dies untermauert erneut die Erläuterungen zu der Problematik mit NUMA-Systemen bei memory bound Algorithmen.

Anders als auf den x86-Systemen ist die Überlegenheit des entwickelten Verfahrens auf dem Cell Blade nicht ganz so deutlich, aber dennoch stetig. Erneut wird bereits für wesentlich kleinere Eingabegrößen Speedup erreicht, welcher dann bis zu einem stabilen Maximum von etwa $1,7$, $2,5$ und $3,2$ ansteigt. Ähnlich wie auf dem Opteron-System zeichnet sich für die größten Eingaben eine Stagnation des Speedups ab.

Der Vergleich mit der MCSTL zeigt darüber hinaus für die hier betrachteten Eingabegrößen keine Vorteile für das neuere Verfahren von Frias et al. Zumindest auf den

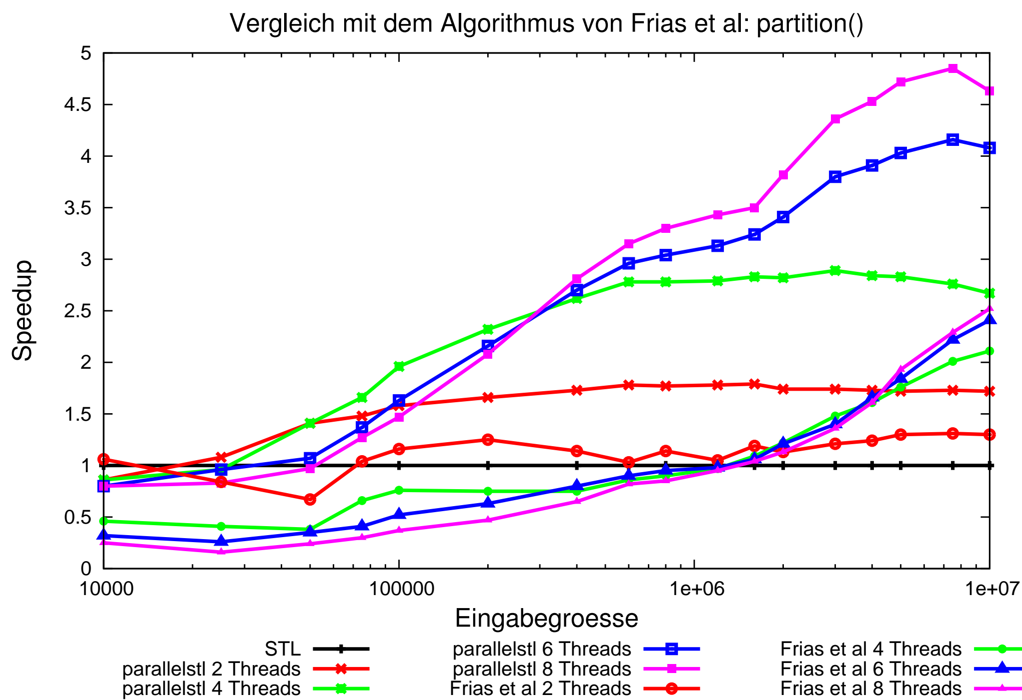

Abbildung 4.14: Xeon-System: Vergleich mit dem Algorithmus von Frias et al für die Funktion `partition()`

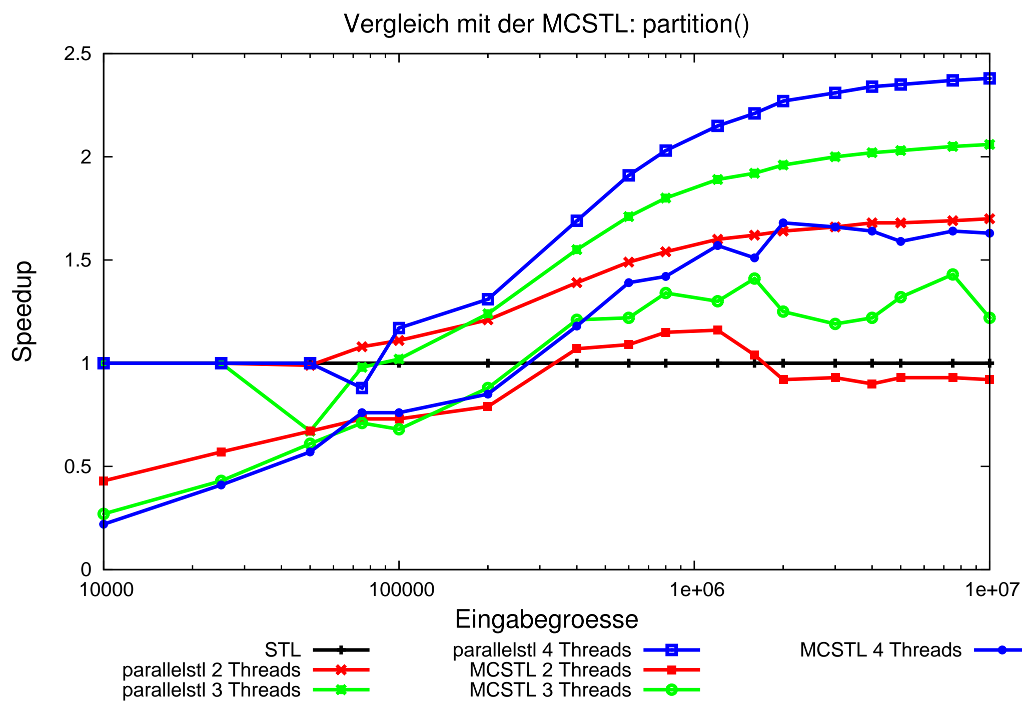

Abbildung 4.15: Opteron-System: Vergleich mit der MCSTL für die Funktion `partition()`

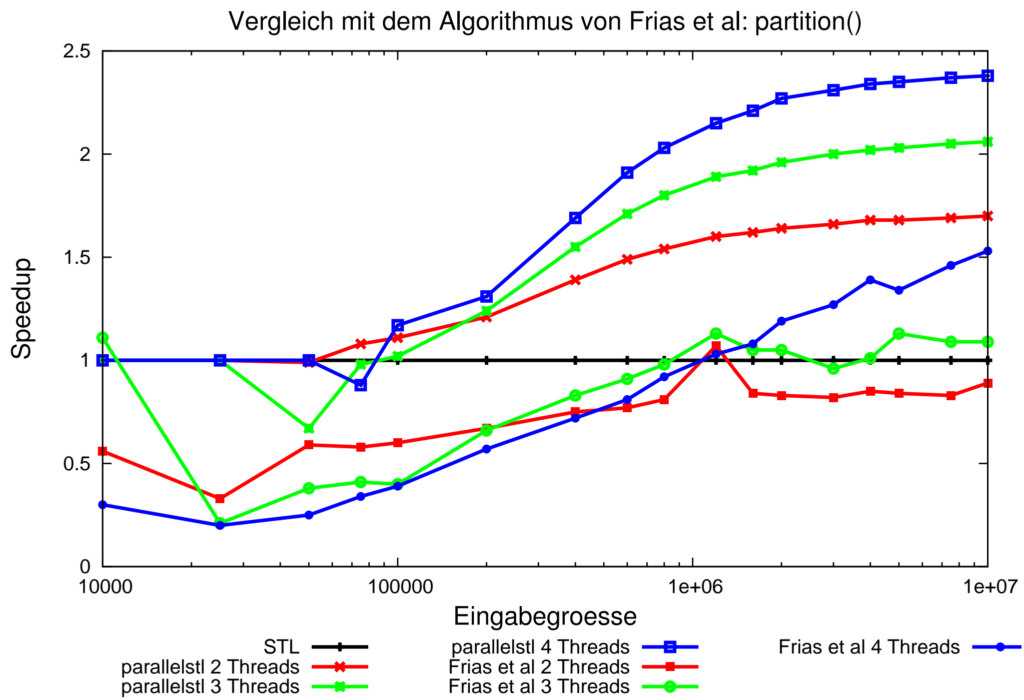

Abbildung 4.16: Opteron-System: Vergleich mit dem Algorithmus von Frias et al für die Funktion `partition()`

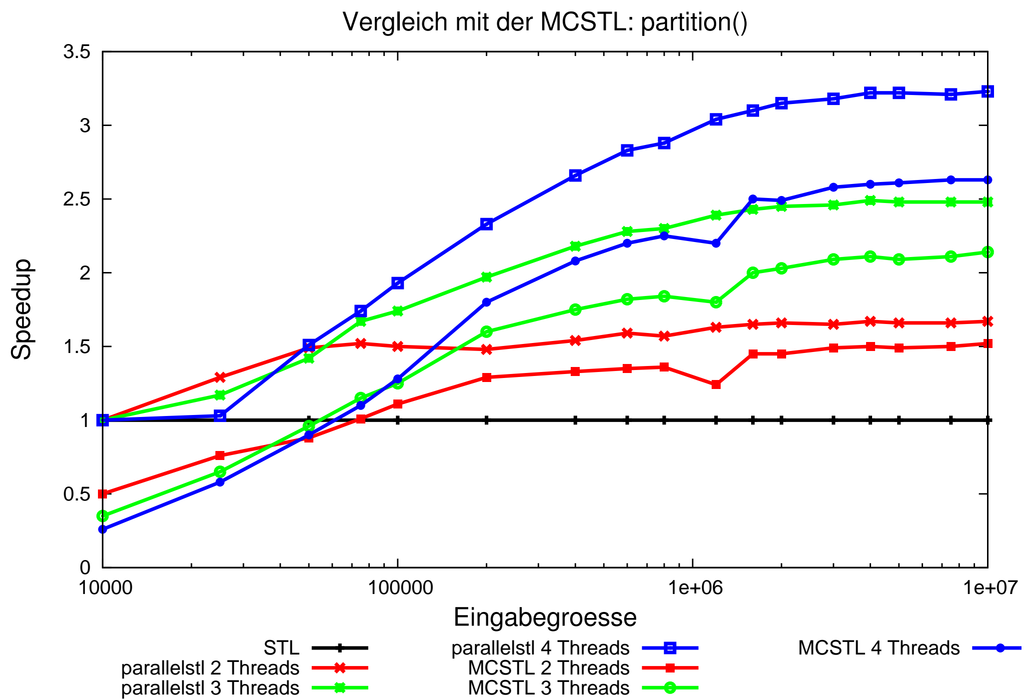

Abbildung 4.17: Cell Blade: Vergleich mit der MCSTL für die Funktion `partition()`

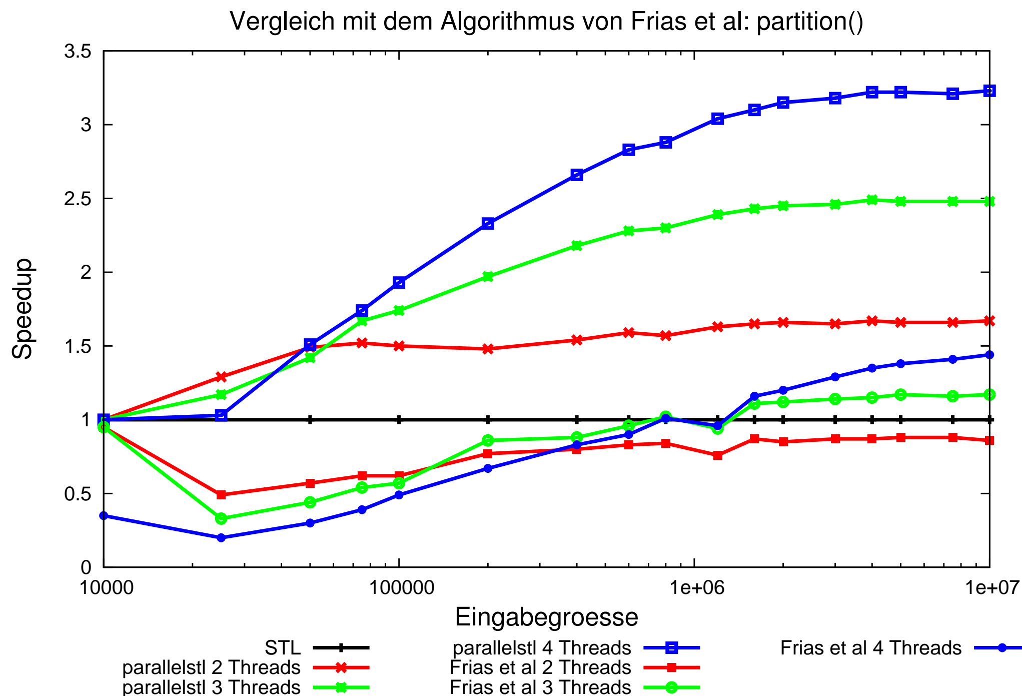

Abbildung 4.18: Cell Blade: Vergleich mit dem Algorithmus von Frias et al für die Funktion `partition()`

x86-Systemen zeigt sich jedoch für die größten gemessenen Eingaben ein deutlich steilerer Anstieg des Speedups als beim MCSTL-Algorithmus, weshalb dessen Leistung für noch größere Eingaben durchaus übertroffen werden könnte.

4.2.3 Numerische Algorithmen

accumulate. Abbildung 4.19 zeigt erneut ernorme Probleme der MPTL bei memory bound Algorithmen, in diesem Fall für `accumulate()`. In Korrektheitstests wurde darüber hinaus deutlich, dass die parallele MPTL-Variante falsche Ergebnisse liefert, da der vom Benutzer zu übergebene Startwert `init` nicht sinngemäß behandelt wird. Für die ParallelSTL-Implementierung ist auf dem Xeon-System das bereits bekannte Schema für vom Speicherdurchsatz abhängige Funktionen zu sehen. Der Speedup steigt im In-Cache Bereich zunächst flach an und es lohnen sich höchstens vier Threads, da ansonsten die andere CPU in Anspruch genommen wird und dadurch Speichertransferoperationen teurer werden. Wird der In-Cache Bereich verlassen, steigt der Speedup steil bis zu seinem Höhepunkt bei einem Faktor von 3 an, und fällt anschließend zurück auf ein reines Out-Of-Cache Niveau.

Beim Vergleich mit der MCSTL in Abbildung 4.20 gibt es auf dem Xeon-System keinen klaren Sieger. Bei zwei und vier Threads erreicht der entwickelte Code mehr Speedup, bei sechs und acht zunächst die MCSTL. Dieses Verhältnis setzt sich im Bereich der In-Cache

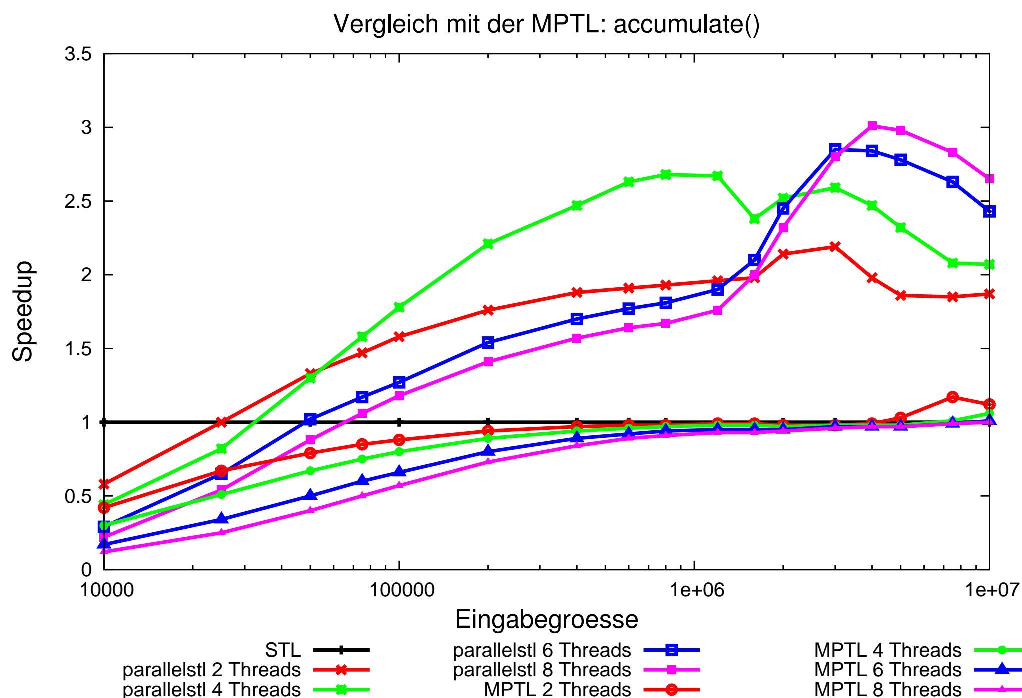

Abbildung 4.19: Xeon-System: Vergleich mit der MPTL für die Funktion `accumulate()`

Größen fort, beim Wechsel auf Out-Of-Cache Größen erhält die Leistung der MCSTL jedoch einen Knick und wird erneut übertroffen.

Auf dem Opteron-System (Abbildungen 4.21 und 4.22) ist die Situation eindeutig, da der PARALLELSTL-Algorithmus die besten Ergebnisse erzielt. Wieder zeigt sich, dass die nah am linearen Speedup liegende Leistung mit zwei Threads auch mit vier Threads bestenfalls geringfügig überboten werden kann, während mit drei Threads Speedup verloren geht. Die MPTL erreicht anders als auf dem Xeon-System Speedup, bleibt aber hinter den beiden anderen Implementierungen zurück.

Anders ist dies auf dem Cell Blade. Beim Vergleich der Abbildungen 4.23 und 4.24 fällt auf, dass bei zwei verwendeten Threads alle Verfahren den gleichen maximalen Beschleunigungsfaktor von $1,4$ vorweisen können. Bezüglich des Unterbietens der sequentiellen Laufzeit sind die MCSTL und das PARALLELSTL-Verfahren hier für drei und vier Threads weitgehend gleich auf, während das entwickelte Verfahren bei zwei Threads dieses Ziel bereits ab 50000 Elementen erreicht. Der MPTL gelingt dies erst ab $6 \cdot 10^6$ Elementen und sie skaliert auch in der Folge nur mäßig. Die Leistung der MCSTL bei vier Threads schwankte stark, auch die nach vielen Durchläufen besten, hier abgebildeten, Werte blieben jedoch abgesehen von den größten Eingaben unterhalb der Kurven des PARALLELSTL-Ansatzes. Der insgesamt höchste erreichte Wert liegt bei einem Beschleunigungsfaktor von $2,76$.

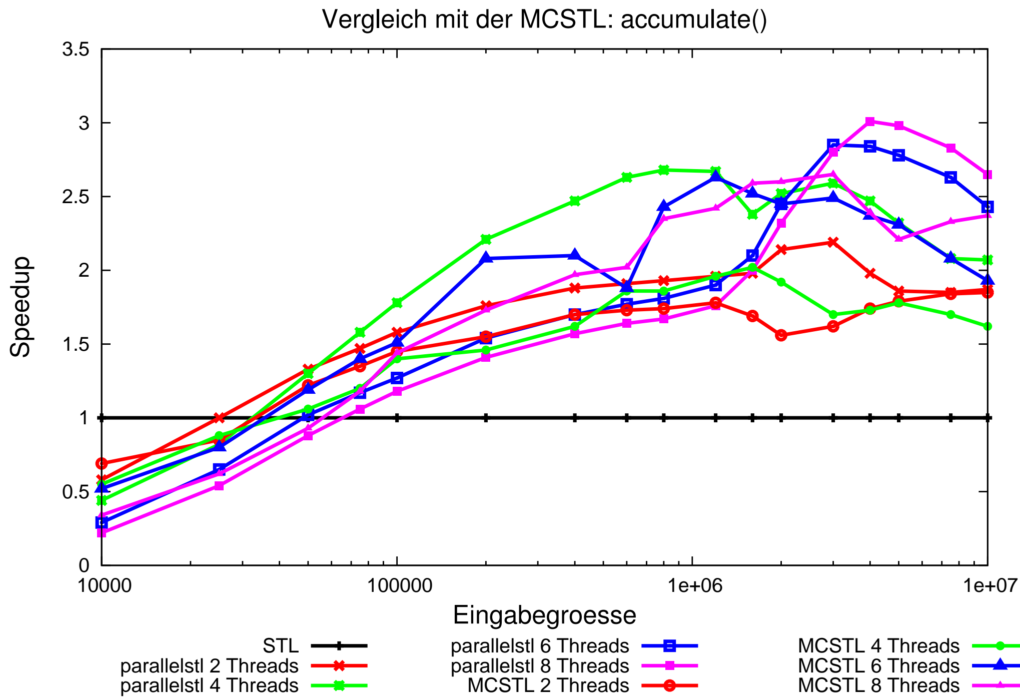

Abbildung 4.20: Xeon-System: Vergleich mit der MCSTL für die Funktion `accumulate()`

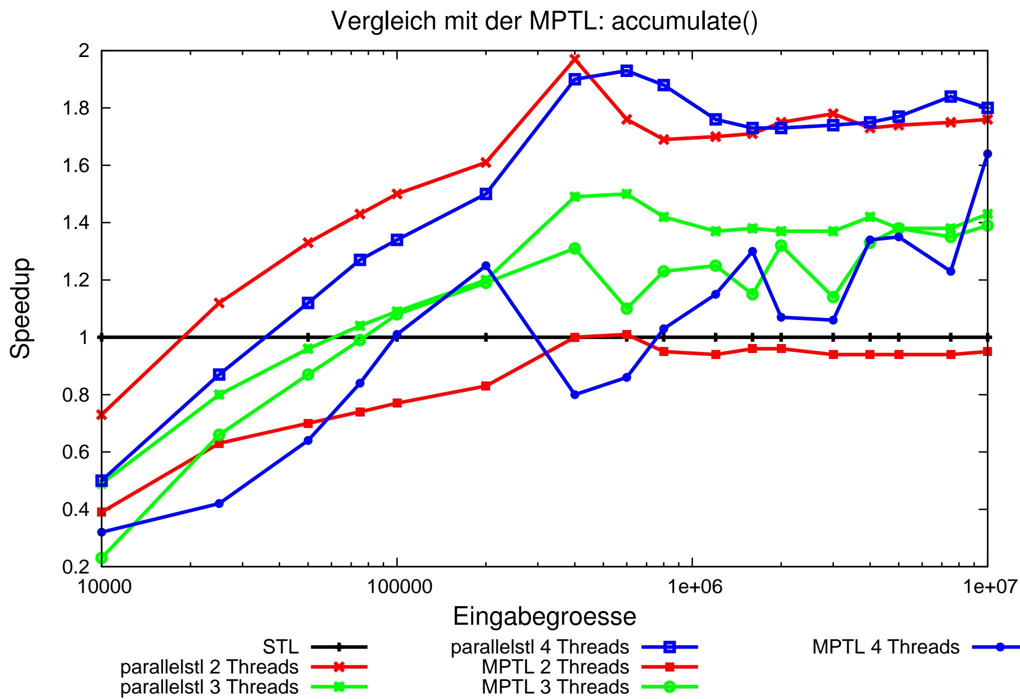

Abbildung 4.21: Opteron-System: Vergleich mit der MPTL für die Funktion `accumulate()`

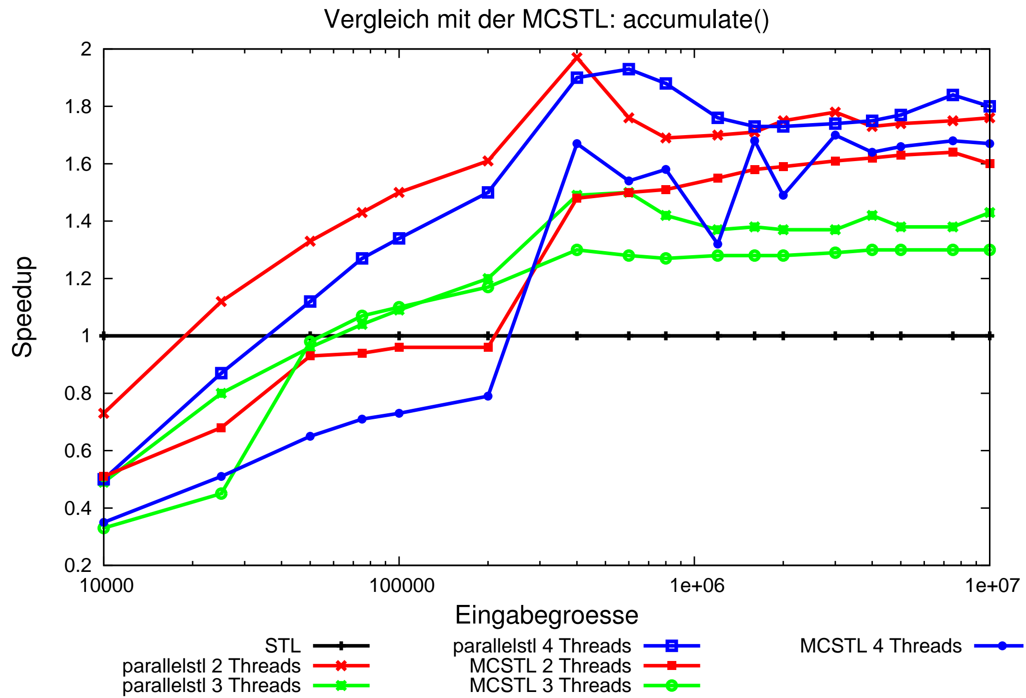

Abbildung 4.22: Opteron-System: Vergleich mit der MCSTL für die Funktion `accumulate()`

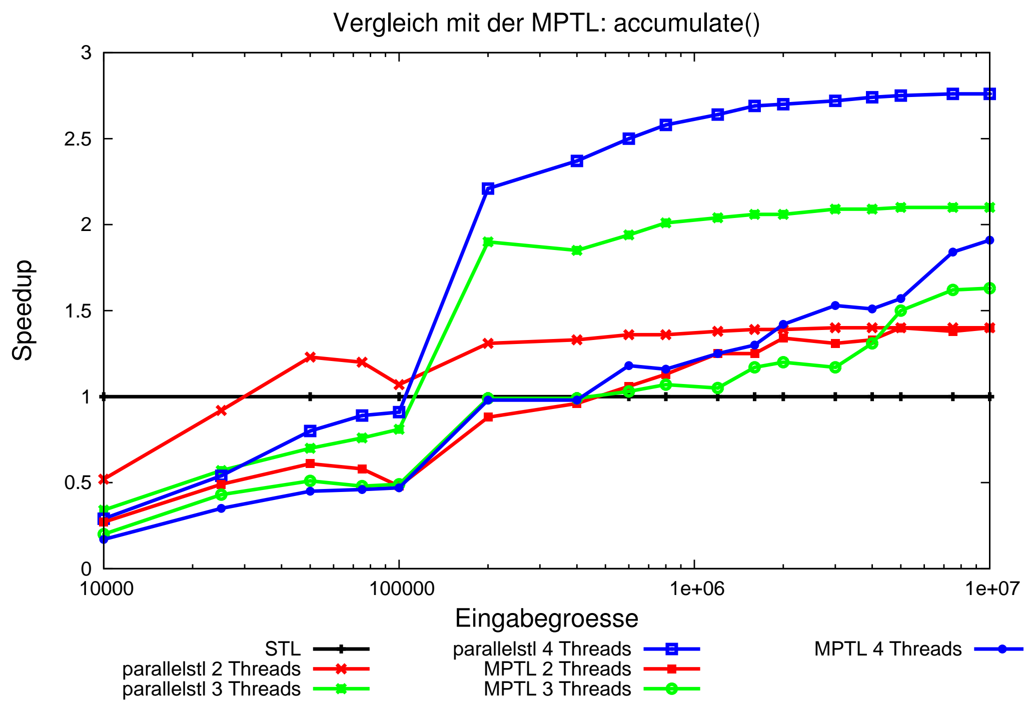

Abbildung 4.23: Cell Blade: Vergleich mit der MPTL für die Funktion `accumulate()`

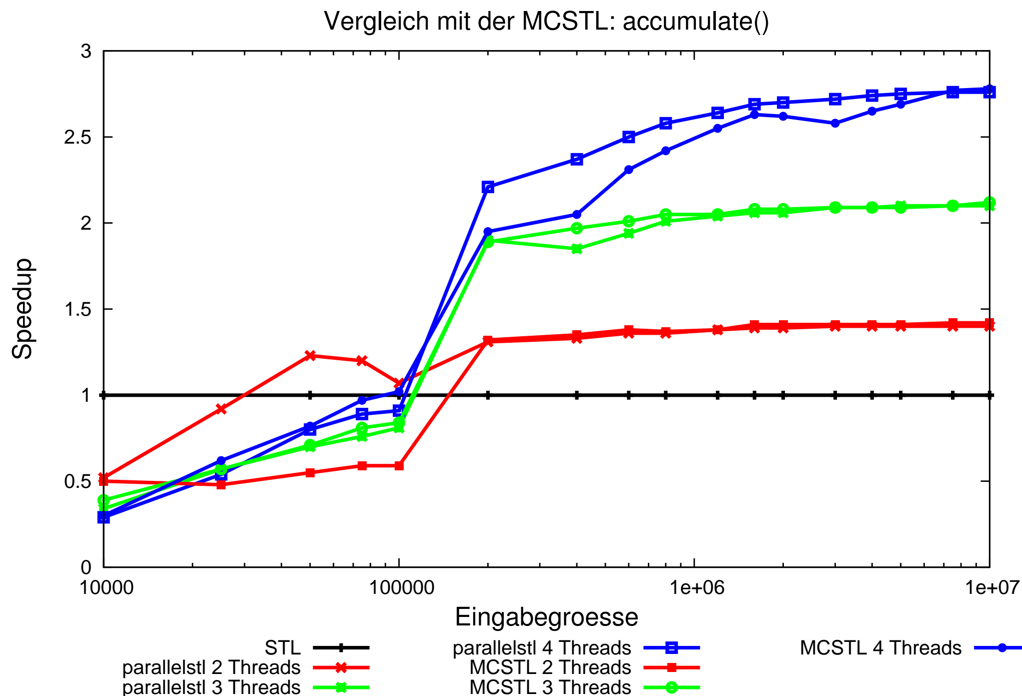

Abbildung 4.24: Cell Blade: Vergleich mit der MCSTL für die Funktion `accumulate()`

inner_product. Auch bei der Funktion `inner_product()`, bei der im Vergleich zu `accumulate()` lediglich eine zweite Eingabesequenz und eine Multiplikation pro Elementpaar hinzu kommt, gibt es auf dem Xeon-System (Abbildung 4.25) keinen klaren Sieger. Bereits bei `accumulate()` gab es im Bereich des Wechsels von In-Cache auf Out-of-Cache Größen und des damit verbundenen Strategiewechsels beim Dispatching (erläutert in Abschnitt 3.2.3) eine kleine superlineare Phase, die nun durch die zweite Eingabesequenz etwas früher sichtbar wird und stärker ausgeprägt ist. Ein Blick auf Abbildung 3.8 in Abschnitt 3.2.4 hilft, dies zu erklären. Bei einer Eingabegröße von $1,2 \cdot 10^6$ liegt etwa die Hälfte der Eingabe nicht mehr im Cache. Bei der parallelen Ausführung mit zwei Threads ist diese Hälfte genau die Partition, die der Hauptprozess bearbeitet. Für den anderen Thread besteht zwar eine Out-of-Cache Situation, er hat aber die volle Bandbreite des Speichersystems zur Verfügung. Bei sequentieller Ausführung jedoch beginnt der Algorithmus am Anfang der Sequenz, die nicht mehr im Cache liegt. Das Laden dieser Elemente überschreibt andere Einträge im Cache, wodurch beim Erreichen der zweiten Hälfte diese ebenfalls nicht mehr zur Verfügung stehen. Deshalb ist der Durchsatz des STL-Algorithmus in Abbildung 3.8 bereits in etwa bei seinem vollständigen Out-of-Cache Niveau angekommen, während dies mit zwei Threads noch nicht der Fall ist. Die STL benötigt 3349 μs und der parallele Algorithmus 880 μs für den In-Cache Teil des Hauptprozesses, sowie 1320 μs für Kern 0, der den Out-of Cache Teil ausführt. Addiert mit dem Overhead ergibt sich eine Gesamtlaufzeit von 1404 μs, und damit ein Speedup > 2.

Ebenfalls wie bei `accumulate()` übertrifft der ParallelSTL-Algorithmus mit sechs und acht verwendeten Threads erst bei Out-of Cache Größen die Leistung der MCSTL. Für die größten Eingaben laufen die Kurven beider Verfahren zusammen, insgesamt bleibt die erreichte Beschleunigung stets unterhalb eines Faktors von 3. Die Verhältnisse auf dem

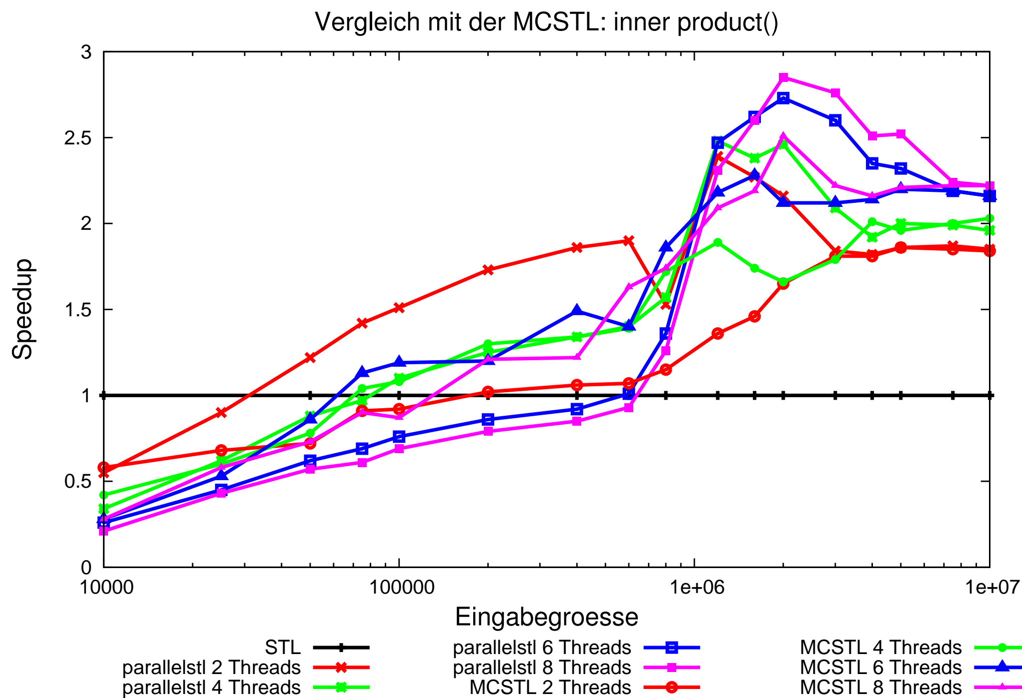

Abbildung 4.25: Xeon-System: Vergleich mit der MCSTL für die Funktion `inner_product()`

Opteron-System (Abbildung 4.26) zeigen Vorteile für den ParallelSTL-Algorithmus, welcher für deutlich kleinere Eingaben Speedup und auch anschließend die höchsten Werte erreicht. Der maximal erzielte Speedup von $1,76$ ist auf einem insgesamt mäßigen Niveau. Für die dargestellten Ergebnisse der MCSTL waren sehr viele Messiterationen erforderlich. In einzelnen Messungen kam es sogar vor, dass die sequentielle Laufzeit mit zwei oder mehr Threads nicht unterboten werden konnte.

Auf dem Cell Blade zeigt sich ebenfalls in etwa die gleiche Situation wie bei `accumulate()`, auch bei zwei verwendeten Threads sind die erzielten Speedups nun weitgehend identisch. Erneut schwankte die Leistung der MCSTL für vier Threads enorm und erforderte sehr viele Durchläufe, um zu stabilen Werten zu gelangen. Sie bleibt dennoch leicht hinter dem Speedup des ParallelSTL-Algorithmus zurück, welcher mit einem Faktor von $2,6$ den höchsten erreichten Speedup auf diesem System erzielt.

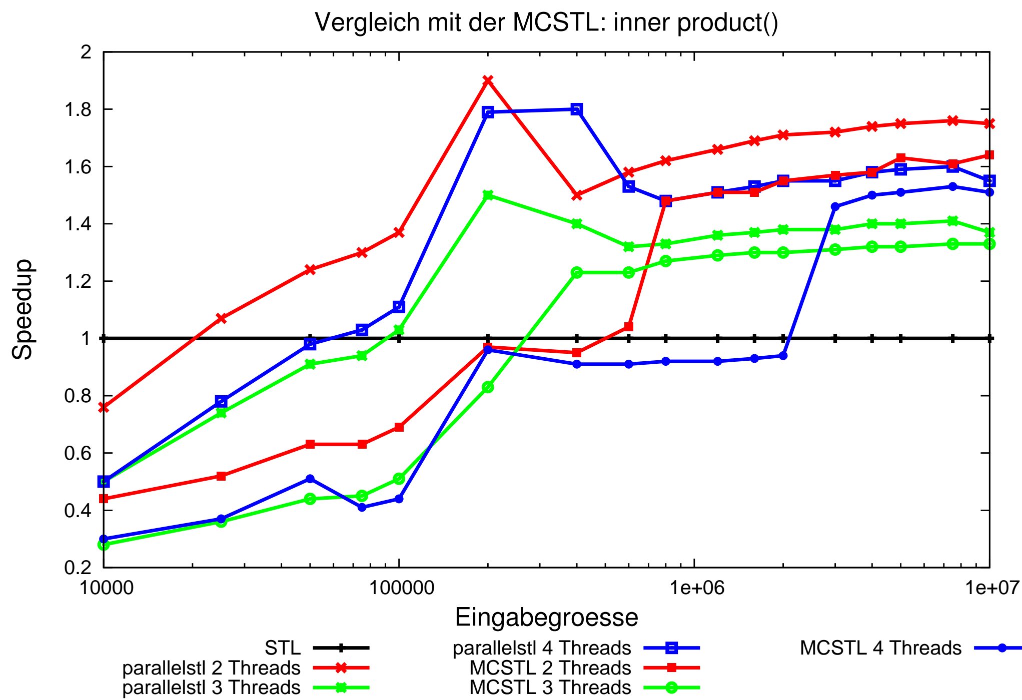

Abbildung 4.26: Opteron-System: Vergleich mit der MCSTL für die Funktion `inner_product()`

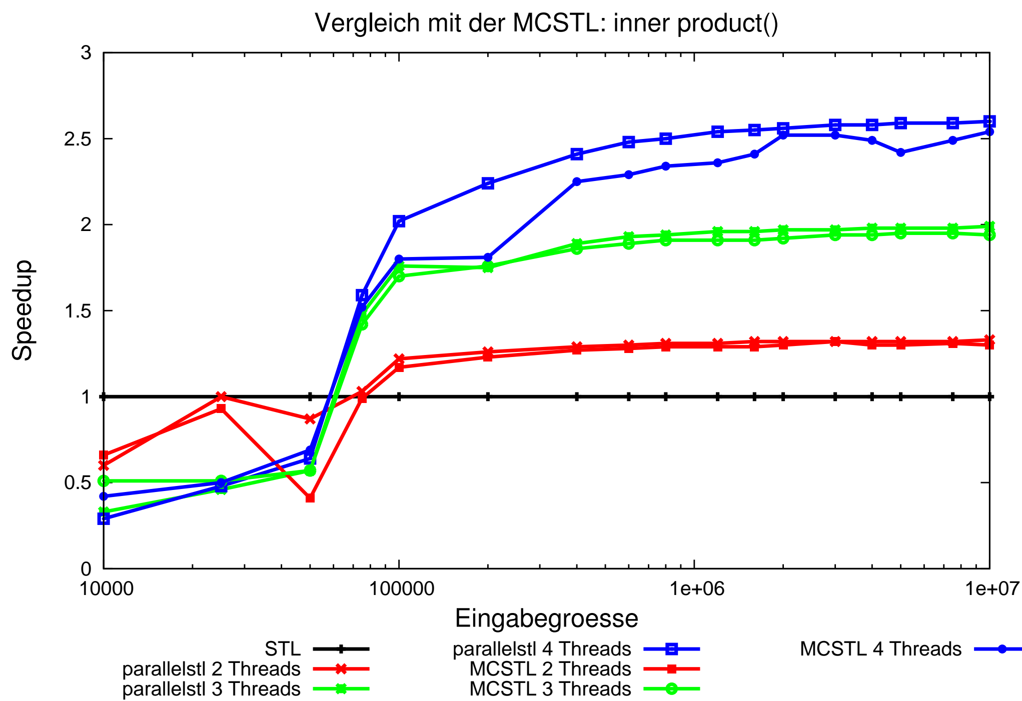

Abbildung 4.27: Cell Blade: Vergleich mit der MCSTL für die Funktion `inner_product()`

inner_product SSE Version Es folgen die Ergebnisse zu der vektorisierten Variante von `inner_product()`. Zunächst erfolgt ein Vergleich der parallelen vektorisierten mit der parallelen skalaren Funktion, in Abhängigkeit von der Anzahl der verwendeten Threads. Der theoretische maximale Speedup beträgt vier, da bis zu vier Werte vom Typ `float` gleichzeitig in einem SSE-Register verarbeitet werden können.

Die Abbildungen 4.28 und 4.29 verdeutlichen, dass Speedup grundsätzlich nur für In-Cache Größen erreicht wird. Im Out-Of-Cache Fall ist wie gesehen bereits die skalare Variante memory bound. Diese Situation ändert sich auch durch die Vektorisierung nicht, weshalb keine zusätzliche Beschleunigung erfolgt. Aus diesem Grund kann auf dem Xeon-System wegen des shared Cache mit bis zu zwei Threads, auf dem Opteron-System nur bei sequentieller Ausführung Nutzen aus der Vektorisierung gezogen werden.

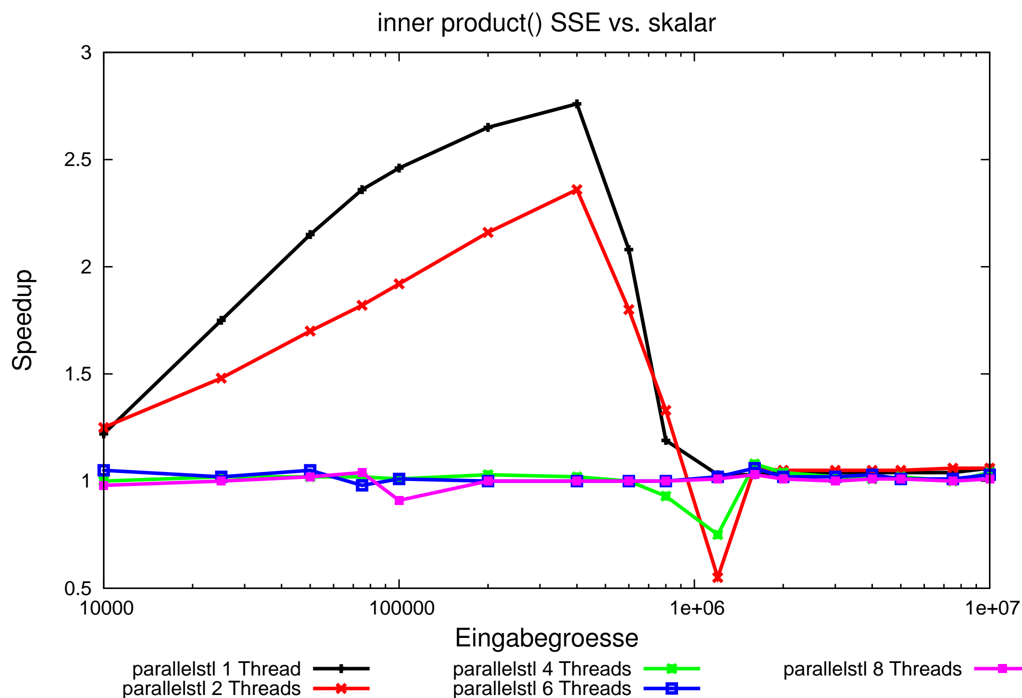

Abbildung 4.28: Xeon-System: Speedup der SSE-Variante über die skalare Funktion `inner_product()`

Als Nächstes folgt mit den Abbildungen 4.30 und 4.31 der Vergleich der parallelen vektorisierten Ausführung mit der sequentiellen vektorisierten Ausführung. Wie nach den zuvor vorgestellten Ergebnissen zu erwarten ist, kann auf dem Xeon-System nur mit zwei Threads ein Speedup für In-Cache Größen erreicht werden. Bei mehr Threads sind die Kosten der erforderlichen Speichertransfers höher als der Gewinn durch die Parallelisierung. Für Out-Of-Cache Größen kehrt sich das Kostenverhältnis um, mit acht Threads wird der höchste Speedup von ca. $2,75$ erzielt.

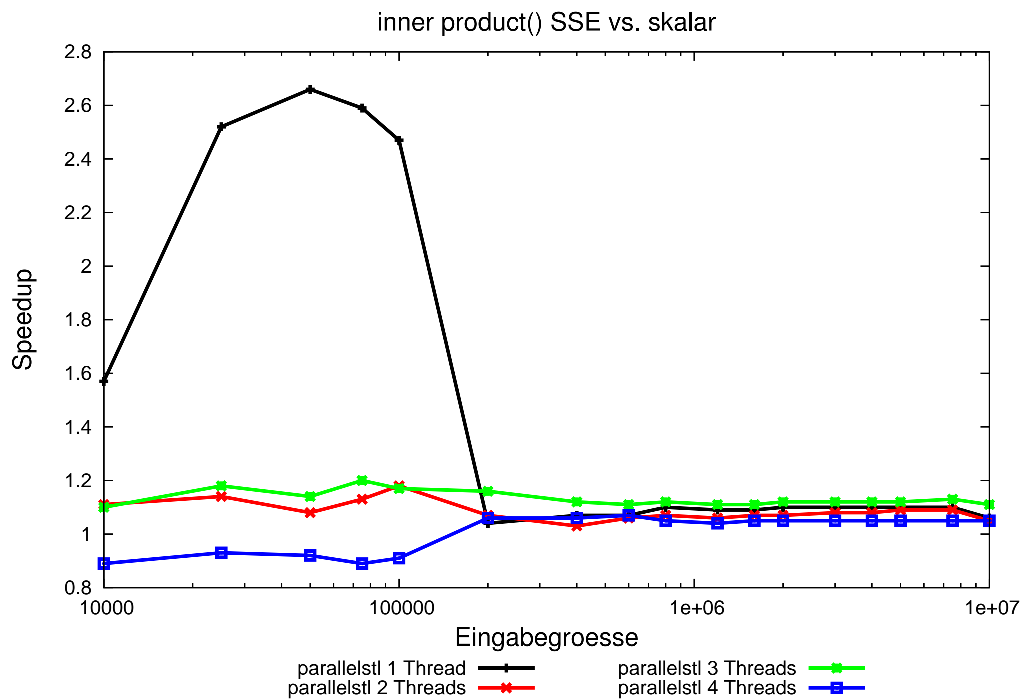

Abbildung 4.29: Opteron-System: Speedup der SSE-Variante über die skalare Funktion `inner_product()`

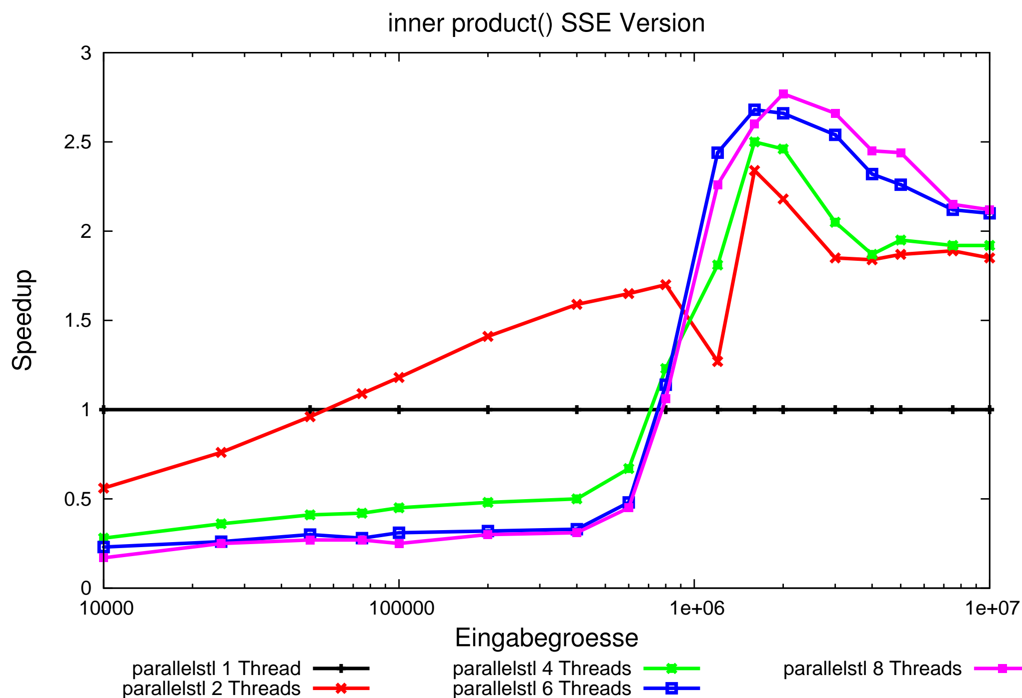

Abbildung 4.30: Xeon-System: Speedup der parallelen Ausführung der SSE-Variante von `inner_product()`

Da keine shared Caches vorliegen, wird wie erwartet auch mit zwei Threads zunächst kein Speedup auf dem Opteron-System erreicht. Auch hier ändert sich die Situation für Out-Of-Cache Größen, jedoch zeigt sich erneut das Problem für memory bound Algorithmen und die Laufzeit für zwei Threads kann auch mit drei und vier Threads nicht mehr unterboten werden.

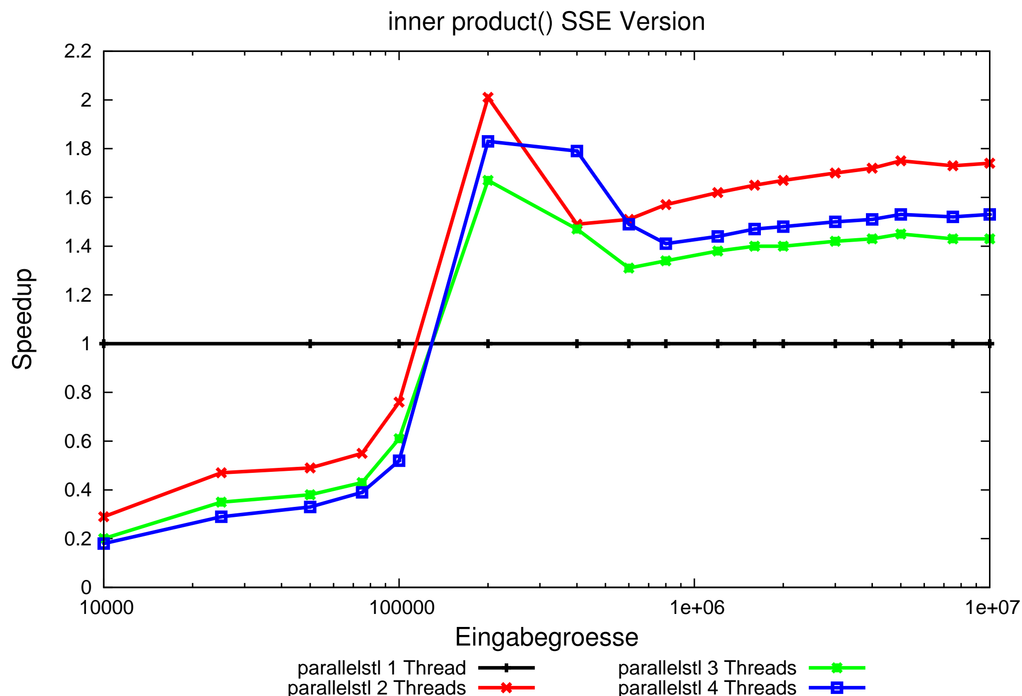

Abbildung 4.31: Opteron-System: Speedup der parallelen Ausführung der SSE-Variante von `inner_product()`

partial_sum. Trotz des in Abschnitt 3.3 beschriebenen Mehraufwandes kann die entwickelte Variante von `partial_sum()` bereits ab 75000 Elementen mit zwei Threads die sequentielle Laufzeit und nachfolgend Speedup erreichen (siehe Abbildung 4.32). Die Skalierung ist insgesamt mäßig, auch mit acht Threads wird nicht mehr als ein Faktor von 2, 31 erreicht. Abgesehen von einigen Eingabegrößen, bei denen mit sechs und acht Threads noch kein Speedup erzielt wird, ist der ParallelSTL-Ansatz auf dem Xeon-System durchgehend schneller als die MCSTL. Die MCSTL-Ergebnisse schwanken darüber hinaus z. T. stark, insbesondere bei zwei eingesetzten Threads.

Auf dem Opteron-System erreicht der entwickelte Algorithmus die sequentielle Laufzeit bei einer Eingabegröße von 10^5 und damit deutlich früher als die MCSTL. Für Eingabegrößen bis $1,6 \cdot 10^5$ kann ein Vorsprung gehalten werden, ab dann liegen die Leistungen auf demselben Niveau, wie Abbildung 4.33 zeigt. Auch hier ist der maximale Speedup mit 1, 56 insgesamt sehr gering. Sehr deutlich zu sehen ist eine bessere Leistung mit drei und

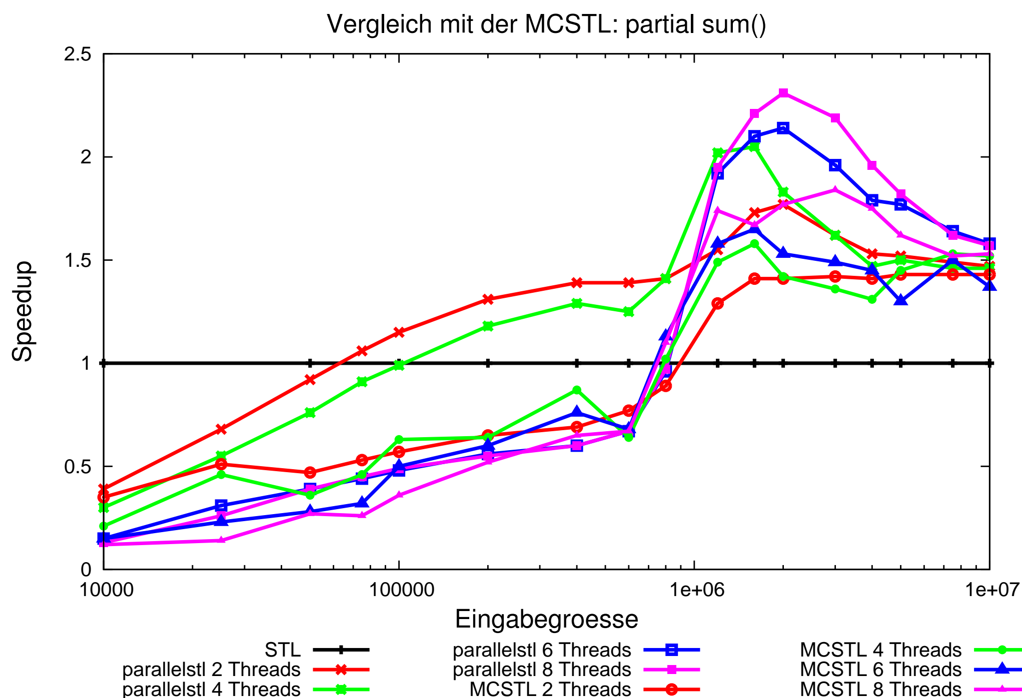

Abbildung 4.32: Xeon-System: Vergleich mit der MCSTL für die Funktion `partial_sum()`

vier Threads für mittlere Eingaben, obwohl es sich um einen memory bound Algorithmus handelt. Dies ist durch die Wiederverwendung der Kerne (und damit ihrer Caches) im zweiten Teil des Algorithmus zu erklären. Bei den Out-of-Cache Größen wird anschließend wieder die für memory bound Algorithmen übliche Reihenfolge eingenommen.

Anders als auf den x86-Systemen sind die Leistungen des entwickelten `partial_sum()`-Algorithmus und des MCSTL-Verfahren auf dem Cell Blade unabhängig von der Anzahl der eingesetzten Threads ebenbürtig. Beide erreichen mit zwei Threads jedoch nur geringen Speedup, während die insgesamt bessere Skalierung zu einem maximal erreichen Speedup von $1,71$ führt (Abbildung 4.34).

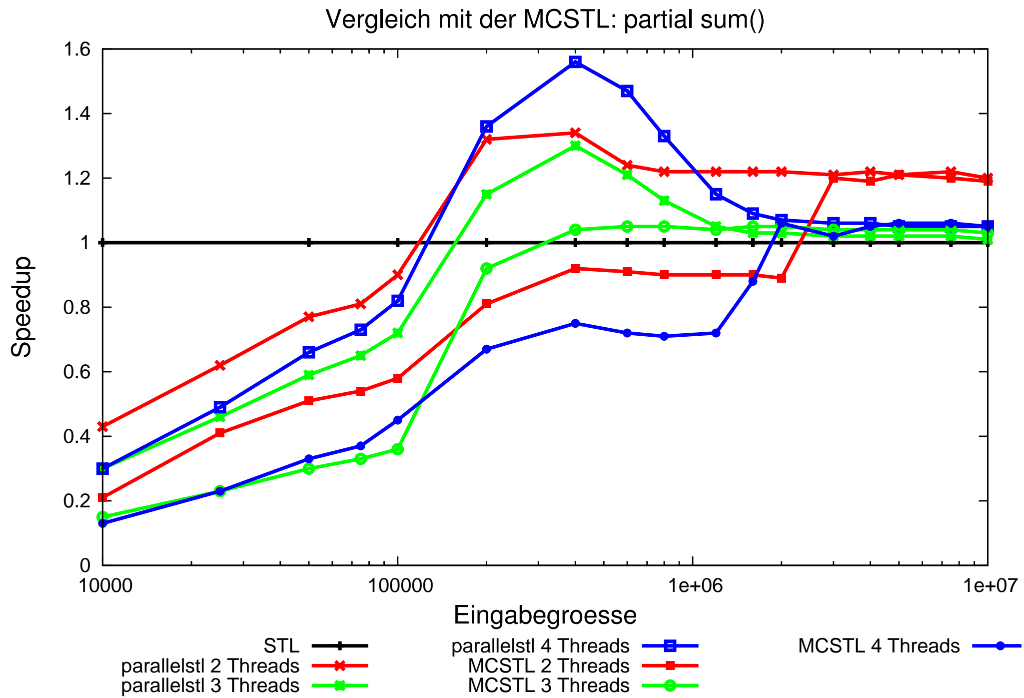

Abbildung 4.33: Opteron-System: Vergleich mit der MCSTL für die Funktion `partial_sum()`

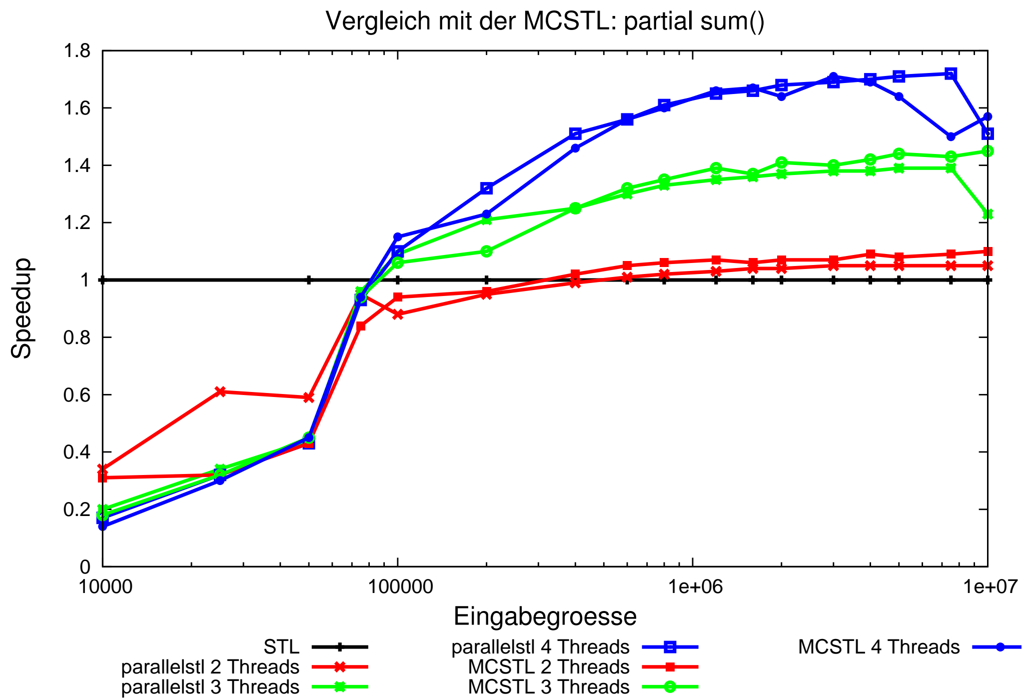

Abbildung 4.34: Cell Blade: Vergleich mit der MCSTL für die Funktion `partial_sum()`

4.2.4 Sortieralgorithmen

nth_element. Für die Funktion `nth_element()` kann das in Abschnitt 3.3 erläuterte Verfahren gute Ergebnisse erzielen. Auf dem Xeon-System in Abbildung 4.35 wird bereits für deutlich kleinere Eingaben Speedup erzielt als durch die MCSTL. Die Leistung ist im direkten Vergleich über alle gemessenen Eingabegrößen überlegen. Es ist ersichtlich, wie sich die Ausführung mit sechs und acht Threads erst später lohnt, als mit zwei oder vier Threads. Der steile Anstieg der Skalierung deutet auf ein starkes Potential nach oben für noch größere Eingaben hin. Das Gesagte gilt ebenfalls für die Ergebnisse auf dem

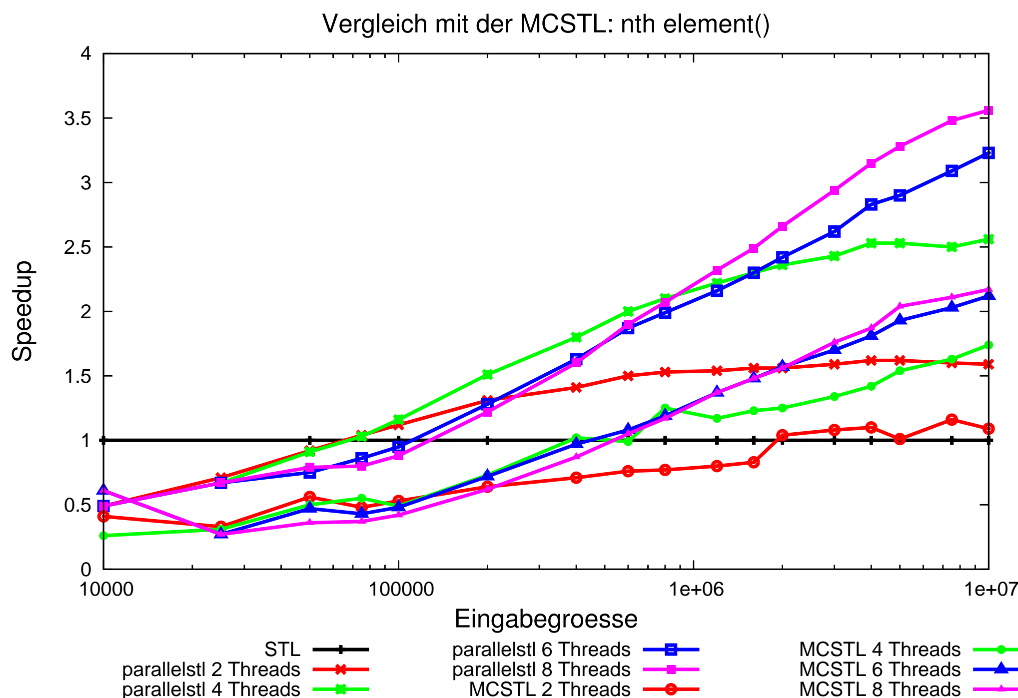

Abbildung 4.35: Xeon-System: Vergleich mit der MCSTL für die Funktion `nth_element()`

Opteron-System (siehe Abbildung 4.36). Einzig die Skalierung ist etwas schwächer, man erkennt bei den größeren Eingaben bereits eine Tendenz zur Konvergenz des Speedups, der ein Maximum von $2,13$ nicht übersteigt. Das Problem bezüglich der Kosten bei Einbindung der anderen CPU tritt hier wie erwartet nicht auf, die Leistung mit drei und vier Threads übertrifft die Leistung mit zwei Threads deutlich.

Auch auf dem Cell Blade sind die Verhältnisse zwischen den verschiedenen Ansätzen identisch, obwohl die MCSTL bereits mit zwei Threads etwas mehr Speedup erreicht, als auf den anderen Systemen. Der ParallelSTL-Ansatz erzielt mit einem Faktor von $3,11$ eine bessere Skalierung als auf der AMD-Architektur, wie Abbildung 4.37 zeigt.

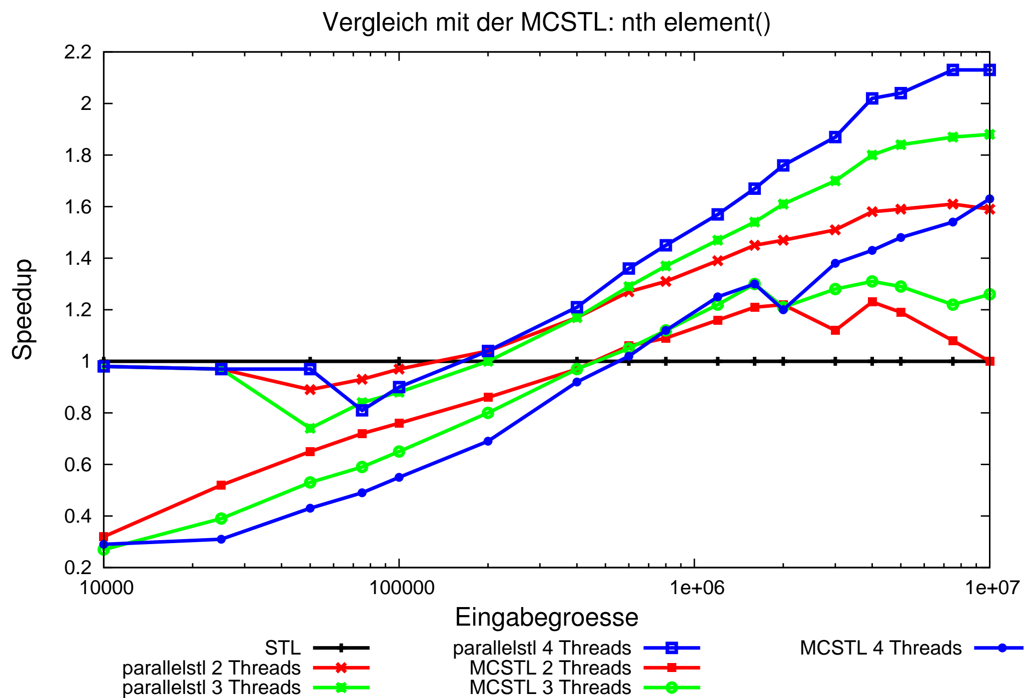

Abbildung 4.36: Opteron-System: Vergleich mit der MCSTL für die Funktion `nth_element()`

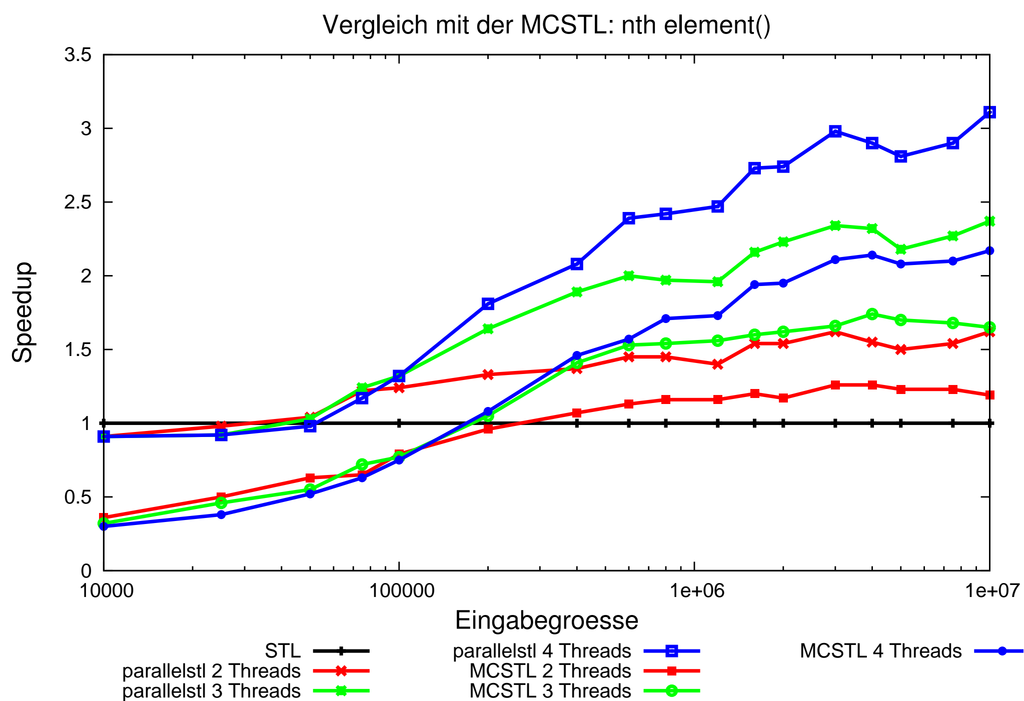

Abbildung 4.37: Cell Blade: Vergleich mit der MCSTL für die Funktion `nth_element()`

sort. Bei der Disziplin des parallelen Sortierens zeigt sich wie erwartet eine umkämpfte Situation zwischen den weitgehend identischen PARALLELSTL- und MCSTL-Algorithmen. Die Ergebnisse bestätigen auch die in Abschnitt 2.3 gezeigten Werte für die größten Eingaben, bei denen die MCSTL immer ein wenig schneller ist. Abbildung 4.38 zeigt eine Pari-Situation für zwei und vier Threads auf dem Xeon-System und zunächst deutliche Überlegenheit bei sechs und acht Threads für das im Rahmen dieser Arbeit entwickelte Verfahren, bevor die Leistung durch die MCSTL übertroffen wird. Da die MCSTL-Entwickler optimierte Varianten für zwei bis vier Threads entwickelt und viel Aufwand in ihre Algorithmen investiert haben, ist dieses Ergebnis für ein allgemeines Verfahren durchaus ein Erfolg. Für kleinere Eingaben schwankt der MCSTL-Ansatz unabhängig von der Anzahl eingesetzter Threads relativ stark, sodass erneut viele Iterationen notwendig waren, um stabil skalierende Ergebnisse präsentieren zu können. Die MPTL kann auf allen

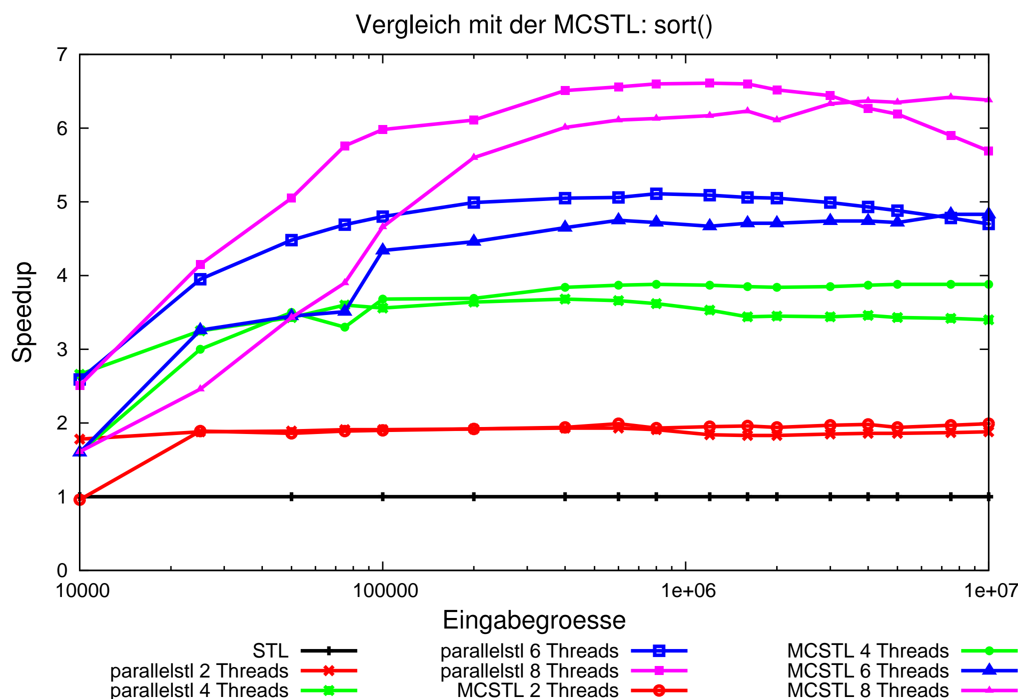

Abbildung 4.38: Xeon-System: Vergleich mit der MCSTL für die Funktion `sort()`

Testsystemen (Abbildungen 4.39, 4.41 und 4.43) zunächst keinen Speedup erzielen, ehe für größere Eingaben ein steiler Anstieg der Beschleunigung mit hohem Potential erfolgt. Auf dem AMD-System erzielt die MCSTL, erneut abgesehen von den kleinsten gemessenen Eingabegrößen, einen höheren Nutzen aus der Parallelität, als der PARALLELSTL-Code und tendiert für größere Eingaben noch deutlicher zu einer linearen Skalierung (Abbildung 4.40). Die Überlegenheit des MCSTL-Verfahrens wird auch auf dem Cell Blade deutlich. Bei zunächst geringen Vorteilen des PARALLELSTL-Verfahrens für kleine Eingaben, dreht

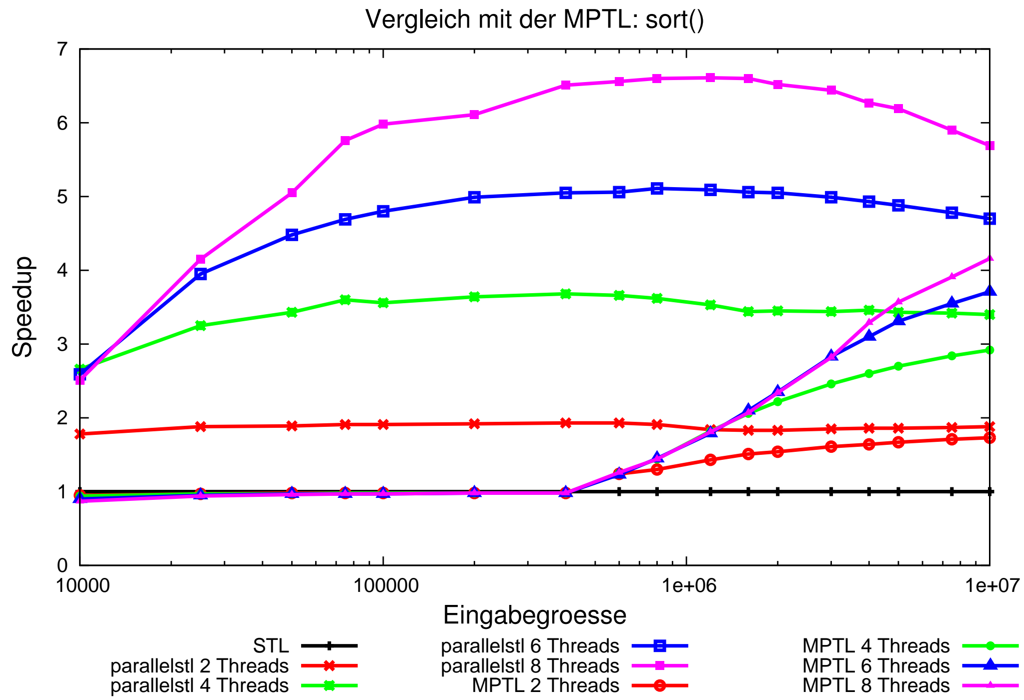

Abbildung 4.39: Xeon-System: Vergleich mit der MPTL für die Funktion `sort()`

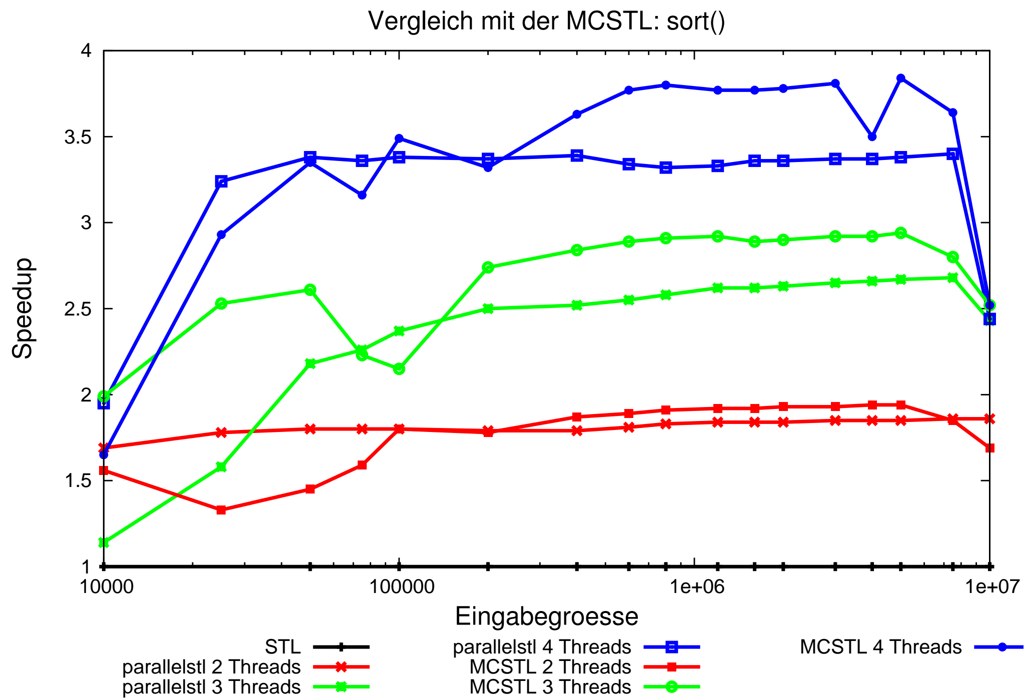

Abbildung 4.40: Opteron-System: Vergleich mit der MCSTL für die Funktion `sort()`

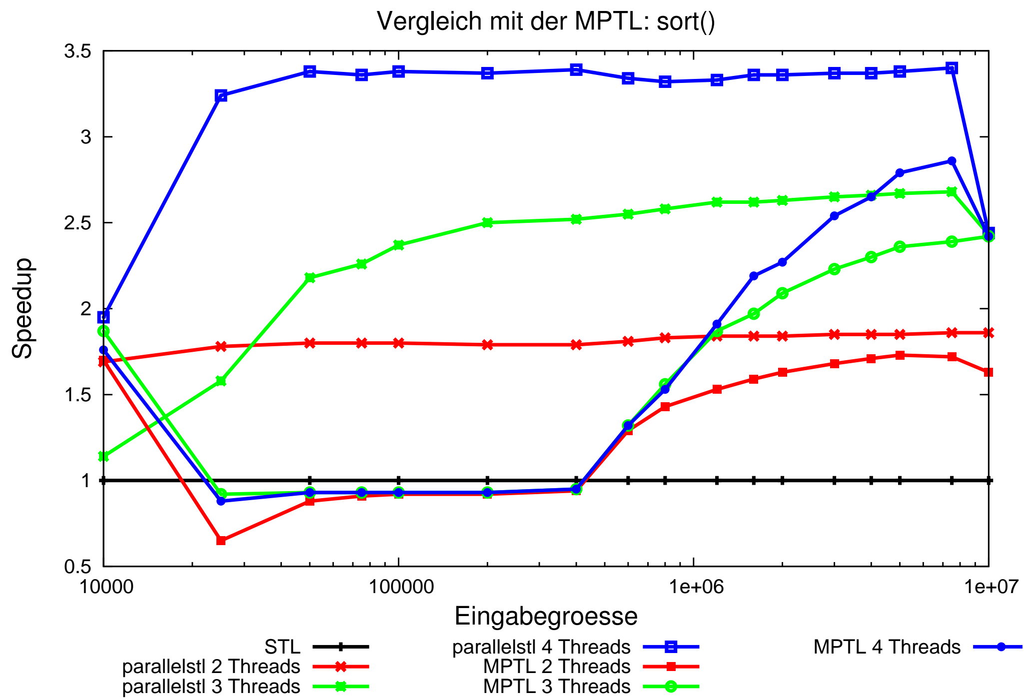

Abbildung 4.41: Opteron-System: Vergleich mit der MPTL für die Funktion `sort()`

sich das Verhältnis beim Verlassen der In-Cache Größen um. Die MCSTL ist dann sowohl mit zwei, als auch mit drei und vier Threads um einen konstanten Faktor schneller.

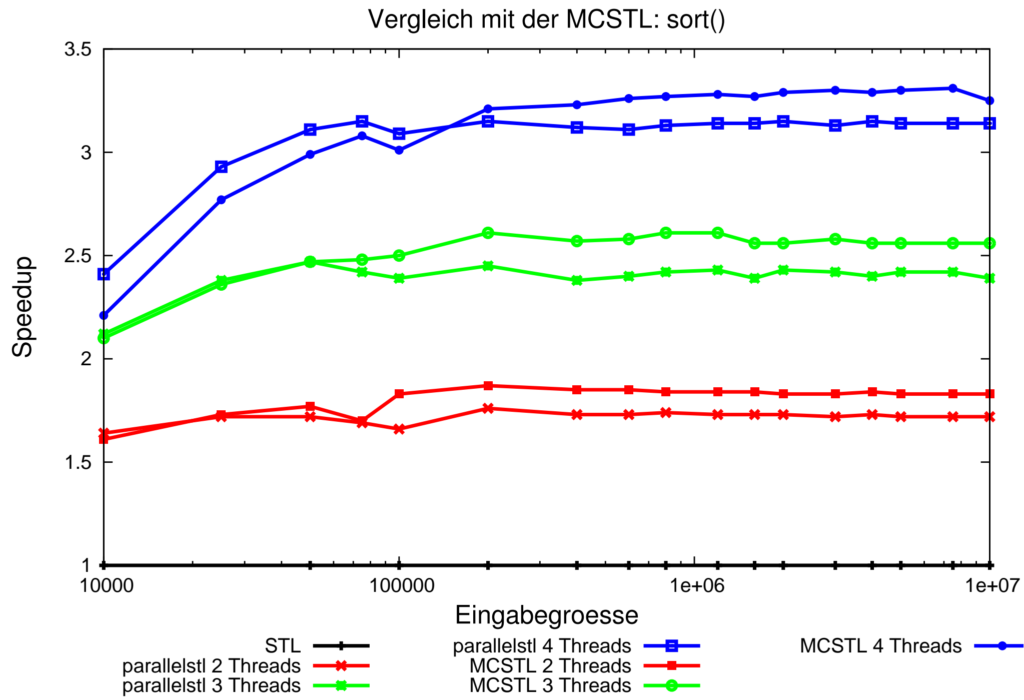

Abbildung 4.42: Cell Blade: Vergleich mit der MCSTL für die Funktion `sort()`

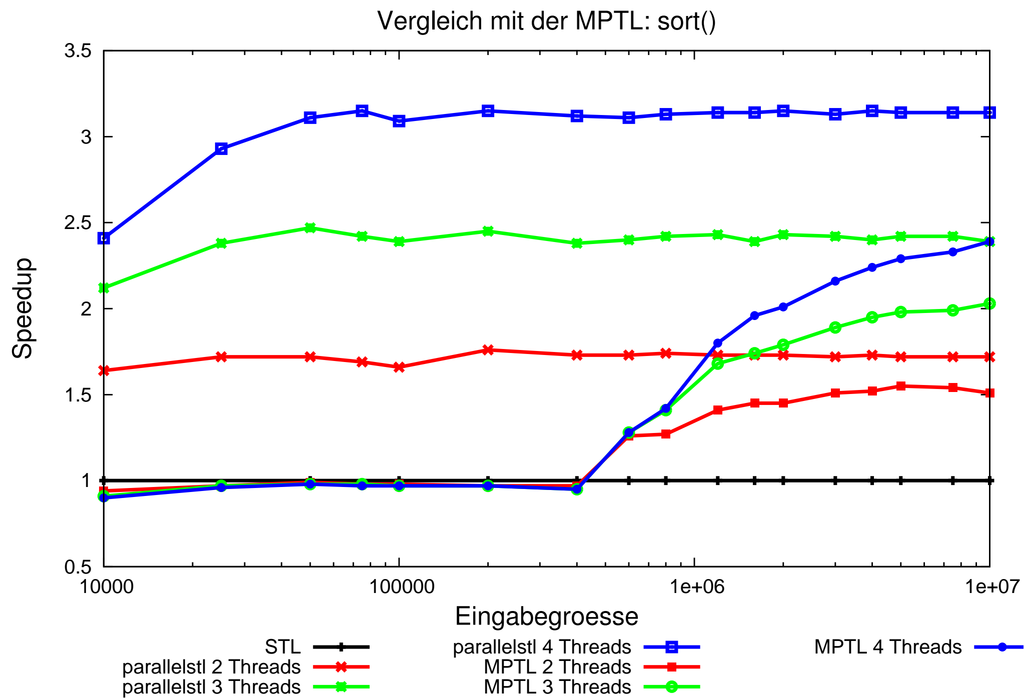

Abbildung 4.43: Cell Blade: Vergleich mit der MPTL für die Funktion `sort()`

stable_sort. Die Ergebnisse für `sort` werden auch für das entsprechende stabile Sortierverfahren bestätigt. Der Vergleich erfolgt hier ausschließlich mit der MCSTL, da die MPTL keinen `stable_sort`-Algorithmus implementiert. Auf dem Xeon-System in Abbildung 4.44 ist die Leistung im Vergleich für zwei Threads erneut weitgehend identisch, für vier Threads ist die MCSTL über weite Phasen überlegen. Für sechs und acht Threads skaliert der entwickelte Code über nahezu das gesamte Messintervall besser, bis ähnlich wie bei `sort` bei den größten Eingaben die Plätze getauscht werden.

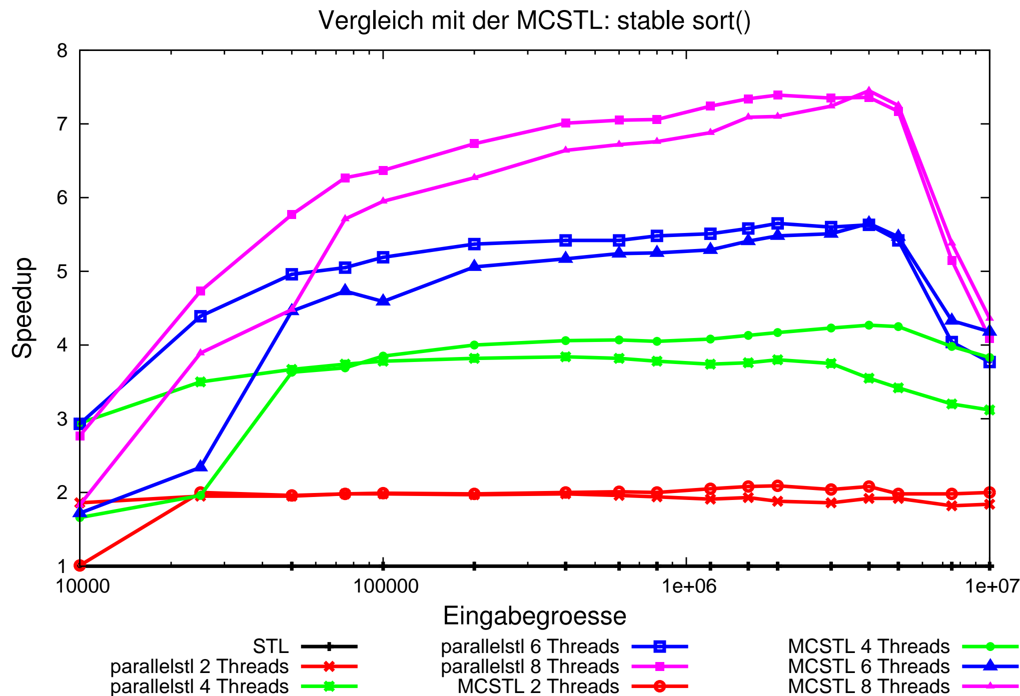

Abbildung 4.44: Xeon-System: Vergleich mit der MCSTL für die Funktion `stable_sort()`

Auf dem Opteron-System (Abbildung 4.45) zeigt sich erneut eine Überlegenheit des MCSTL-Codes, mit der Ausnahme kleiner Eingabegrößen, während auf dem Cell Blade (Abbildung 4.46) nach ebenfalls zunächst leichten Vorteilen für den ParallelSTL-Ansatz für In-Cache Größen, die Leistungen der beiden Konzepte ebenbürtig sind. Der bei beiden Ansätzen und jeder Anzahl Threads erkennbare Knick bei $2 \cdot 10^5$ Elementen ließ sich auch durch viele Wiederholungen nicht als Messfehler herausstellen und hängt mit einer verhältnismäßig hohen Steigerung der absoluten sequentiellen Laufzeit beim Verlassen der In-Cache Größen zusammen.

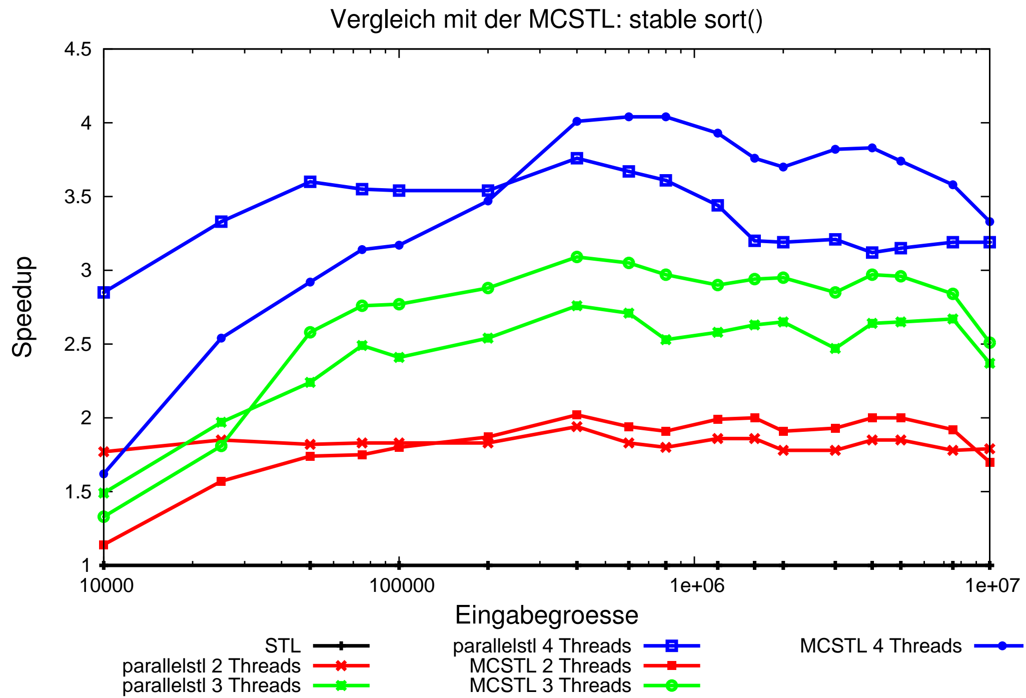

Abbildung 4.45: Opteron-System: Vergleich mit der MCSTL für die Funktion `stable_sort()`

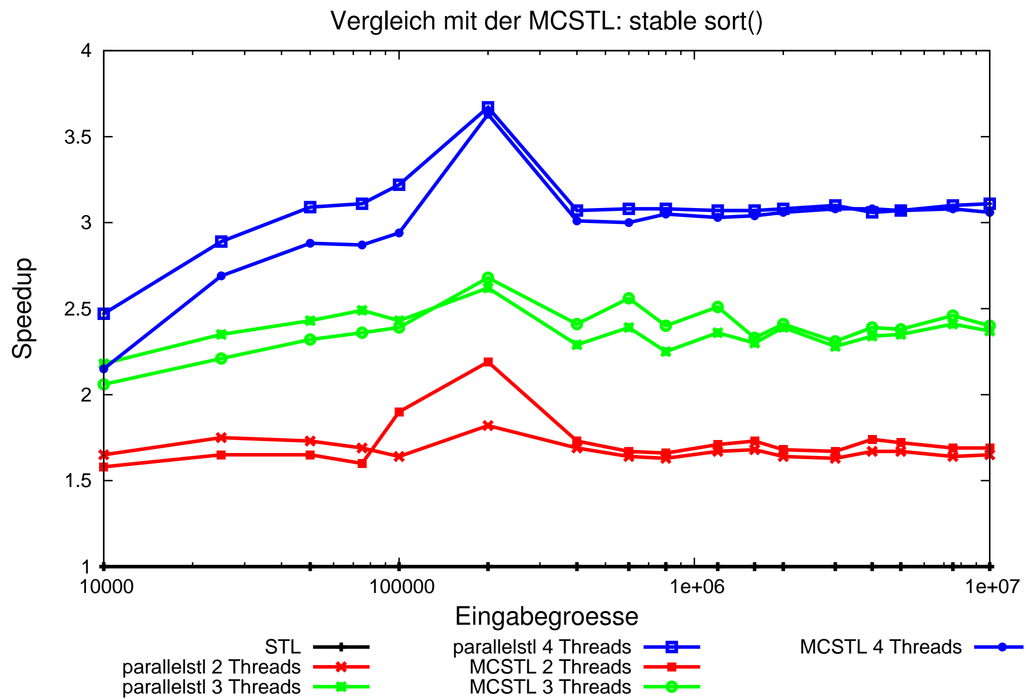

Abbildung 4.46: Cell Blade: Vergleich mit der MCSTL für die Funktion `stable_sort()`

Kapitel 5

Evaluierung

5.1 Zusammenfassung und Diskussion der Ergebnisse

Bereits in Kapitel 2 wurde deutlich, wie groß die Unterschiede in der Realisierung des Multithreadings bei verschiedenen Bibliotheken sein können und Kapitel 4 dokumentiert die Auswirkungen dieser Unterschiede in der Praxis. Dabei hat sich bei einigen Funktionen gezeigt, dass die Ergebnisse auf unterschiedlichen Systemen durchaus voneinander abweichen können. So erreichte die MPTL beispielsweise für `replace_copy_if()` auf dem Opteron-System akzeptablen Speedup, während die sequentielle Laufzeit auf dem Xeon-System nie unterboten wurde. Auch für die MCSTL zeigten sich ähnliche Differenzen, etwa bei `replace_if()`, für das sie auf dem Cell Blade kaum Speedup erzielte, auf dem Xeon-System aber einen Faktor von $6,5$ bei acht eingesetzten Threads. Auch innerhalb der Messwerte auf einem System sind insbesondere bei der MCSTL z. T. sehr hohe Schwankungen zu beobachten. Häufig war hier das Zusammensetzen der Ergebnisse aus vielen Messfolgen erforderlich, um eine nachvollziehbare und gute Skalierung zu erreichen. In einigen Fällen gelang dies jedoch selbst auf diese Art und Weise nicht. Bezüglich dieser Form der Stabilität zeigte das im Rahmen dieser Arbeit entwickelte Konzept deutliche Vorteile. Die Ergebnisse unterliegen bei allen Benchmarks wesentlich geringeren Schwankungen als bei den Vergleichsbibliotheken ohne Thread Affinity. Erreicht wird dies durch eine an Systemparametern optimierte Auswahl der Threads für die Zuweisung von Tasks, die erst durch die Verbindung von Thread Affinity und dem Threadpool möglich wird. Die ausführenden Kerne sind für ein bestimmtes Problem immer dieselben, während bei Verzicht auf diese Konzepte die Auswahl der Threads allein vom Scheduler abhängt und die Laufzeiten daher gewissen Schwankungen unterliegen. Dass Thread Affinity, wie in Abschnitt 3.2.2 erläutert, unerlässlich ist, um eine gute Skalierung auf Multicore- und insbesondere NUMA-Architekturen zu erreichen, unterstreichen Terboven et al. in [47] und empfehlen ihre Verwendung auch in OpenMP-Programmen. Die Optimierung der Auswahl des Kerns für einen auszuführenden Task hat im Laufe der Entwicklung zu wesentlich stärkeren Lei-

stungsverbesserungen geführt, als jede Form der Mikrooptimierung, wie beispielsweise das Alignment von Daten, das Inlining von Funktionen und die Verwendung höherer Optimierungslevel des Compilers.

Die vorliegende Arbeit zeigt, dass insbesondere bei stark vom Speicherdurchsatz abhängigen Funktionen ein globales Threadpool-Konzept effektiv ist. Denn in diesem Fall ermöglicht die mit der soeben beschriebenen Kombination von Threadpool und Thread Affinity einhergehende Sortierung der Threads das Erreichen eines Speedups bereits für kleinere Eingabegrößen. Dass sich die entsprechende Auswahl nach den Gesichtspunkten der Speicherarchitektur lohnt, zeigten insbesondere die wesentlich besseren Ergebnisse für kleine Eingabegrößen und wenig eingesetzte Threads zu `replace_if()`, `replace_copy_if()`, `accumulate()` und `inner_product()`. Eine derartige Sortierung und optimierte Zuweisung von Threads ist mit OpenMP, abgesehen von einer zukünftig möglichen internen Lösung, nicht effizient realisierbar. Für die genannten Funktionen, sowie `partial_sum()`, konnte in Kapitel 4 (in Verbindung mit den Ergebnissen zur Speicherbandbreite aus Abschnitt 3.2.4 und zur SSE-Variante von `inner_product()`) gezeigt werden, dass sie memory bound sind. Für das verwendete Opteron-Testsystem zeichnete sich dabei ab, dass eine Skalierung nur mit so vielen Threads erreicht wird, wie logische Ausführungseinheiten auf einem der eingesetzten Prozessoren vorhanden sind, wenn alle Speichertransfers nur über einen Speichercontroller abgewickelt werden. Bei Prozessoren der neueren Generation mit shared L3-Cache ist eine deutliche Abmilderung dieses Effekts zu erwarten.

Beim `find()`-Algorithmus wurde erkenntlich, dass eine triviale Parallelisierung ohne die Möglichkeit der Unterbrechung parallel arbeitender Threads im Fall eines gefundenen Elements nicht erfolgversprechend ist. Ebenso ist eine dynamische Lastverteilung für gute Ergebnisse unerlässlich. Das entwickelte Verfahren mit konstanten Blockgrößen konnte sich auf den verwendeten Testsystemen auch gegenüber der growing-block Strategie der MCSTL als überlegen erweisen.

Für die nicht trivial parallelisierbaren Funktionen, wie `partition()`, `nth_element()`, `sort()` und `stable_sort()` wurden ebenfalls vollständig neue Ansätze entwickelt, die alle zu zufriedenstellenden Ergebnissen führten. Für die Sortierverfahren konnte annähernd der maximal mögliche Speedup auf den Testsystemen erreicht werden. Während für kleine Eingaben die im Rahmen dieser Arbeit entwickelten Funktionen häufig schneller waren, zeigte die MCSTL für größere Eingaben ihre Überlegenheit auf Grund hochspezialisierter Merge-Unterprogramme und eines sehr effizienten Verfahrens. Bei den anderen genannten Funktionen zeigte sich jedoch auch gegenüber der MCSTL ein wesentlich höherer Speedup und eine bessere Skalierung. Insbesondere für `partition()` und `nth_element()`, aber auch für `replace_copy_if()` lag der erzielte Speedup zwischen 4 und 5 auf dem Xeon-System, für `replace_if()` sogar bei knapp 7. Auch auf dem Opteron-System wurden für diese Problemstellungen Speedups > 2 erreicht und eine Skalierung bei Einsatz mehrerer Threads. Nur beim Sortieren konnte jedoch annähernd der bestmögliche Speedup erzielt werden.

Wenn wie auf dem Cell Blade die Speicherbandbreite nicht beschränkend wirkte, konnte die MCSTL im Vergleich zu den x86-Systemen in aller Regel etwas besser abschneiden. Dennoch lohnen sich die Konzepte Thread Affinity und Threadpool auch hier und führen bei den genannten nicht trivialen Funktionen zu einem früheren Erreichen und einer besseren Skalierung des Speedups.

Das ausgegebene Ziel, Speedup bereits für deutlich kleinere Eingabegrößen als bei den in Kapitel 2 vorgestellten Bibliotheken zu erzielen, konnte also für das Gros der untersuchten Funktionen realisiert werden. Häufig wurde der in diesem Bereich erreichte Vorsprung auch für steigende Problemgrößen gehalten, bei keiner einzigen Funktion war die Leistung im Vergleich zur Leistung der MCSTL und MPTL hingegen abgeschlagen. Während die MPTL kaum Möglichkeiten zum Vergleich bei den komplexeren STL-Funktionen bot, zeigten die Vergleiche mit der MCSTL sehr deutlich, dass die jeweils erreichte Skalierung neben den beschriebenen Faktoren auch maßgeblich von der Komplexität und Beschaffenheit des Algorithmus abhängt. Bei durch den Speicherdurchsatz in der Laufzeit dominierten Algorithmen ist die Skalierung in der Regel für alle betrachteten Verfahren mäßig, Unterschiede und damit auch Verbesserungen konnten in erster Linie im Bereich der kleinen In-Cache Eingabegrößen realisiert werden. Umso länger die Bearbeitungszeit zwischen Laden und Speichern eines Elements und umso geringer die Datenabhängigkeiten zwischen den Elementen, desto höheres Potential besteht für eine gute Skalierung durch Multithreading.

Neben den Benchmarkergebnissen konnten mit Hilfe der Thread Affinity sonst unzugängliche Ursachen gefunden und tiefgründig untersucht werden, die das Erreichen von Speedup erschweren oder sogar verhindern. Insbesondere für memory bound Algorithmen wurde herausgearbeitet, weshalb eine lineare Skalierung vor allem auf aktuellen x86-Systemen nicht zu erwarten ist. Gleichzeitig zeigen die Ergebnisse auf dem Cell Blade, wie nah die Ergebnisse an das Ideal herankommen können, wenn der Speicherdurchsatz nicht die primäre Beschränkung darstellt. Die Ausführungen zu diesen Ursachen erlauben Entwicklern Rücksicht auf derartige Einschränkungen zu nehmen und motivieren ein sehr handwerkliches, intensives Auswerten der Geschehnisse zur Laufzeit parallel arbeitender Threads.

Die entwickelte Multithreading-Basis selbst erzeugt nur sehr geringen Overhead und ist in hohem Maße flexibel. Durch die klare, einheitliche Struktur und das Threadpool-Konzept ist es sehr einfach um zusätzliche Funktionen erweiterbar und an neue architekturielle Herausforderungen anzupassen. Trotz der Notwendigkeit der Existenz eines `ThreadPool`-Objekts zur Laufzeit gelang es, die Funktionsköpfe und damit die Schnittstellen der Algorithmen vollkommen identisch zur STL zu gestalten. Die Struktur auf Basis der Funktionsobjekte ist noch geradliniger als das CSPA-Konzept der MPTL. Es kann in noch stärkerem Maße auch zur Entwicklung von parallelen Algorithmen verwendet werden, die nicht Teil der STL sind, da bei der MPTL z. B. das in Abschnitt 2.2 beschriebene `typedef`-Konzept

zu einer Einschränkung wird, wenn mehr als vier Iteratoren benötigt werden oder der Algorithmus erst gar nicht auf Iteratoren basiert.

Bei der Bewertung der Ergebnisse, insbesondere im Vergleich zur MCSTL, mag angeführt werden, dass ein derart für verschiedene Systeme optimierter Ansatz zwangsläufig zu besseren Ergebnissen führen muss, als eine generalisierte Bibliothek, die auf allen Plattformen Speedup erreichen soll. Es handelt sich jedoch bei dem hier entwickelten Ansatz nicht um eine auf eine Plattform spezialisierte, sondern um eine an die am häufigsten real existierenden Speicherarchitekturen angepasste Implementierung. Ohne Einbezug des unterliegenden Speichersystems ist kein optimaler Nutzen aus mehreren zur Verfügung stehenden Recheneinheiten möglich. Gerade eine Bibliothek mit dem Ziel auf verschiedensten Architekturen zu skalieren und auch für lediglich zwei Threads bereits Speedup zu erreichen, muss also zumindest eine grobe Klassifikation des Systems vornehmen, um auf die Unterschiede reagieren zu können, solange Scheduler und Compiler nicht in der Lage sind, diese Aufgabe in optimalem Maße zu übernehmen.

5.2 Einschränkungen und Grenzen

Obwohl versucht wurde, auf NUMA-Systeme mit speziellem Dispatching Rücksicht zu nehmen, konnte bei trivialen, linear auf dem Speicher operierenden Algorithmen mit mehr als zwei Threads in der Regel keine zusätzliche Skalierung mehr auf dem für die Benchmarks herangezogenen Opteron-System mit zwei Dualcore-Prozessoren erreicht werden. In vielen Fällen erging es jedoch den anderen Bibliotheken genauso, bzw. hatten diese z. T. noch größere Skalierungsprobleme. Die Ursache liegt daher insgesamt eher darin, dass es bei derart vom Speicherdurchsatz abhängigen Algorithmen keine Möglichkeit gibt, die höheren Latenzen für den Zugriff über den anderen Speichercontroller zu verstecken, als an einer mangelnden Anpassung der entwickelten Software. Dennoch ist die Anpassung an NUMA-Systeme noch nicht optimal. Die in Abschnitt 3.2.3 beschriebene Strategie, auch für Out-of-Cache Größen zunächst alle Kerne einer CPU für die Zuteilung von Aufgaben zu verwenden, bevor teure Speicherzugriffe durch weitere Speichercontroller in Kauf genommen werden, ist nur effizient, so lange die Bandbreite des einen Speichercontrollers ausreicht um alle Anfragen zu bedienen. Andernfalls können die Kosten dieser Zugriffe niedriger sein, als die der Wartezeit auf Grund mangelnder Bandbreite. Gerade mit Blick auf eine wachsende Anzahl der Kerne pro CPU muss hier noch differenzierter vorgegangen werden.

Im Gegensatz zum STAPL-Konzept (durch pRange) und der MCSTL (durch OpenMP) unterstützt das entwickelte Multithreading-Konzept geschachtelte Parallelität nur partiell. Wie bereits in Abschnitt 3.1.3 erläutert, wird dies für die effiziente Parallelisierung von STL-Funktionen allerdings auch nicht benötigt, bzw. ist dadurch keine bessere Skalierung o.ä. zu erwarten. Im Fall des im Rahmen der Arbeit entwickelten Codes ist es möglich,

innerhalb eines parallel ausgeführten Funktionsobjekts manuell neue Threads zu erzeugen und anschließend auf ihre Fertigstellung zu warten. Diese Threads können jedoch nicht zentral vom Threadpool, sondern müssen von der erzeugenden Funktion selbst verwaltet werden. Für die vollständige Unterstützung geschachtelter Parallelität wären tiefgreifende, aber durchführbare Änderungen des Threadpools erforderlich. Beispielsweise müsste der Zugriff auf die Threadlisten durch einen Mutex geschützt werden, da es nun mehrere Vaterprozesse gäbe. Auch ein spezielleres Task- und Dispatching-Konzept wäre notwendig, um unter allen Umständen Deadlock-Situationen auszuschließen und den Fortschritt immer mindestens eines Blattprozesses im Erzeugerbaum zu gewährleisten.

In bestimmten Szenarien kann selbst der Einsatz des Hauptkonzepts dieser Arbeit, der Thread Affinity, problematisch sein. In der implementierten Art ist eine optimale Leistung bei Funktionsaufrufen, deren Laufzeit geringer ist, als die Dauer einer Zeitscheibe, nur zu gewährleisten, wenn keine anderen rechenintensiven Prozesse einen oder mehrere der gebundenen Kerne auslasten. Ein einfaches Beispiel für einen Dualcore-Prozessor in Abbildung 5.1 veranschaulicht dies. Angenommen, zum Zeitpunkt T_{start}, an dem das parallele Programm beginnt, läuft bereits ein anderer Prozess P_{fremd} auf Kern 0. Da der Thread, der den Programmteil P_0 ausführt, fest an Kern 0 gebunden ist, muss gewartet werden, bis der Scheduler dem Thread eine Zeitscheibe zuweist. Bei Abstraktion von den Kosten eines Kontextwechsels beträgt die Laufzeit folglich $T_{slice} + T_{func} - T_{start}$. Könnte die Bindung in einem solchen Fall gelöst oder das Scheduling flexibler gehandelt werden, wie in Teil (b) der Abbildung angedeutet, müsste allerdings nur eine verdoppelte Laufzeit von $2 \cdot T_{func}$ hingenommen werden. Im Fall von zwei Kernen entspricht dies in etwa der sequentiellen Laufzeit und bei mehr Kernen würde immer noch Speedup erreicht, während beim ersten Szenario negativer Speedup gegenüber der sequentiellen Ausführung auf Kern 1 zu erwarten ist. Ein möglicher Lösungsansatz ist es, bei der für die Thread Affinity verwendeten Maske mehr als einen Kern anzugeben. Dies kann aber offensichtlich in unproblematischen Szenarien zu einem nicht optimalen Scheduling führen. Eine andere Variante ist es, die Maske erst zu verändern, wenn festgestellt wurde, dass ein Thread keinen Fortschritt erzielt. Dies ist aber mit einem Overhead verbunden. Experimentelle Versuche, wie man diesen Overhead möglichst klein halten kann, werden im Anschluss an diese Diplomarbeit in Kooperation mit dem MCSTL-Entwickler Johannes Singler erfolgen.

Eine weitere Einschränkung, die aber auch alle anderen bekannten parallelen Ansätze gleichermaßen betrifft, ist eine numerische Abweichung der parallelisierten Funktionen gegenüber der sequentiellen Ausführung bei Fließkommaberechnungen. Dies ist außer durch unverhältnismäßig hohen Aufwand unvermeidbar. Deutlich wird das Problem an einer einfachen numerischen Funktion wie `accumulate()`. Die Funktion bildet die Summe über alle Elemente a_i einer Sequenz, also $\sum_{i=1}^{n} a_i$. Sind die Werte entsprechend groß, geht auch bei der sequentiellen Ausführung auf Grund der beschränkten Anzahl zur Verfügung stehender Bits Genauigkeit verloren. Sind mehrere Threads im Einsatz, bilden diese zunächst

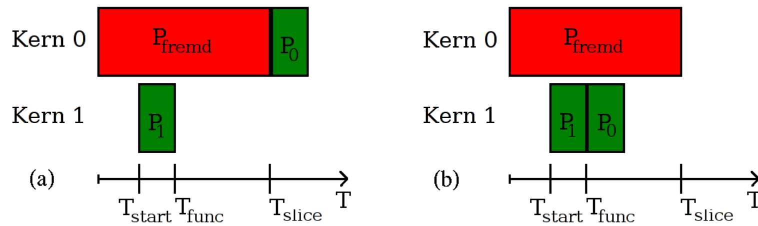

Abbildung 5.1: Worst-Case Szenario bei Einsatz von Thread Affinity (a) und Situation bei flexiblerem Scheduling (b)

lokale Summen, die zum Schluss zum Gesamtergebnis reduziert werden. Dies ist entspricht dem Term $\sum_{j=1}^{p}\left(\sum_{i=1}^{\frac{n}{p}} a_i\right)$. Die Teilsummen sind zunächst kleiner und möglicherweise genauer als das Ergebnis der sequentiellen Ausführung. Die nachträgliche Summation der unterschiedlich genauen Teilergebnisse führt dann jedoch zu weiteren Ungenauigkeiten und ggf. zu einem etwas anderen Ergebnis als im sequentiellen Fall. Bezüglich der Korrektheit der entwickelten Algorithmen wurde für jede Funktion ein Test implementiert, der in der Regel einen Vergleich mit der entsprechenden Lösung der STL vornimmt. Im Fall von Sortierverfahren oder numerisch abweichenden Funktionen erfolgten speziell angepasste Tests.

Zuletzt muss angeführt werden, dass die Erzeugung des `ThreadPool`-Objekts relativ teuer ist, insbesondere weil bei seiner Konstruktion sämtliche Architekturinformationen ausgelesen werden. Die notwendige Information und alle nachfolgend einsatzfähigen Threads werden jedoch auf diese Weise nur einmal erzeugt und daher fallen auch die Kosten nur einmalig an, und nicht bei jedem Funktionsaufruf erneut. In realistischen Szenarien ist davon auszugehen, dass nicht nur eine einzelne STL-Funktion auf den Nutzdaten ausgeführt wird und somit die einmaligen Kosten sehr schnell amortisiert werden. Selbst bei nur einmaliger Verwendung einer parallelen STL-Funktion ist bei hinreichend großer Eingabegröße der Overhead der Erzeugung vielfach geringer, als die Differenz zu den Laufzeiten der zum Vergleich herangezogenen Bibliotheken.

5.3 Optimierungsvorschläge

Um insbesondere auf NUMA-Systemen zu noch besseren Ergebnissen zu kommen, ist vor allem die Entwicklung eines eigenen Speichermanagements, sowie diverser Allokatoren für (parallele) Container zu empfehlen. Erfolgt die Erzeugung der entsprechenden Teile eines Containers von Beginn an auf den Kernen (und wird somit dort lokal Speicher für sie alloziert), auf dem später auch Berechnungen auf ihnen ausgeführt werden, liegen die Daten in den jeweiligen Caches vor oder es kann zumindest die teure Latenz für den Zugriff auf die Daten durch andere Speichercontroller verhindert werden.

Ein Speichermanager kann auch das Alignment von Daten noch wesentlich verbessern. Streng genommen sind die Maßnahmen, die im Hinblick auf false sharing getroffen wur-

den, noch nicht ausreichend. Das Alignment an einer Cacheline-Grenze ist nur der erste Schritt, gleichzeitig müsste (z. B. durch Padding) sichergestellt werden, dass auch hinter den erzeugten Daten keine anderen Objekte innerhalb der Cacheline mehr abgelegt werden. Andernfalls tritt dasselbe Problem auf, dass durch Schreibvorgänge in diesen Objekten die Aktualisierung der entsprechenden Cacheline in allen Caches erforderlich wird. Zusätzlich muss man beachten, dass STL-Funktionen alle ihre Argumente kopieren. Dies hat zur Folge, dass ursprünglich am Speicher ausgerichtete Objekte an Stellen kopiert werden, die möglicherweise nicht ausgerichtet sind. Dies betrifft insbesondere auch die durch die Threads ausgeführten Funktionsobjekte innerhalb der Klasse `Task`, da sie vom Typ `std::tr1::function` sind und der übergebene Funktor somit ebenfalls kopiert wird.

Die STL-Container und Allokatoren sind nur auf sequentielle Ausführung ausgelegt. Die Entwicklung effizient parallel verwendbarer Varianten von z. B. `std::vector` würde die Implementierung eines zentralen Multithreading-Managers, wie dem `ThreadPool`, wesentlich vereinfachen.

5.4 Fazit und Ausblick

Im Rahmen dieser Diplomarbeit konnte eine effiziente Multithreading-Basis mit geringem Overhead implementiert und neue Erkenntnisse bezüglich der den Speedup einschränkenden Faktoren dokumentiert werden. In vielen Fällen wurde im Vergleich zur MCSTL und MPTL die beste Skalierung für an der realen Nutzung ausgerichtete und damit realistische Szenarien erzielt. Bei Verwirklichung einiger Ideen zu Optimierungsvorschlägen sind durchaus noch leicht verbesserte Ergebnisse möglich. Auf Grund der einheitlichen Struktur und der fortgeschrittenen Basis ist das entwickelte Konzept grundsätzlich zu einer vollständigen Bibliothek nach Form der MCSTL erweiterbar.

Ein derartiger Vergleich der Konzepte, so wie in dieser Arbeit dokumentiert, ist nach aktuellem Kenntnisstand bisher einmalig und kann sowohl den Entwicklern der vorgestellten Bibliotheken, als auch zukünftigen Neuentwicklungen einige Ansatzpunkte für die Implementierung geben. Eine Veröffentlichung STAPLs unter einer freien Lizenz könnte neue interessante Vergleichsmöglichkeiten erschließen und neue Ansätze publik machen. Die MCSTL hat sich als die ausgereifteste vollständige Bibliothek herausgestellt und bleibt mit Hinblick auf die Vielzahl implementierter STL-Funktionen bisher ohne gleichwertige Konkurrenz. Mit ihrer Aufnahme in die GCC konnte sie bereits einen sehr großen Erfolg verbuchen und somit heute bereits vielfach zur schnellen und effektiven Parallelisierung existierenden Codes eingesetzt werden. Als am kontinuierlichsten entwickelte parallele Bibliothek bieten sich daher auch mit Blick auf die Effizienz große Chancen noch Fortschritte zu erzielen, insbesondere weil die Entwickler auch an der Implementierung der libgomp beteiligt sind und versuchen den Overhead diverser Primitiven zu verkleinern (vgl. z. B. [21]). Wie bereits im Abschnitt zu den Einschränkungen erwähnt, konnten in kontroversen

Diskussionen mit Johannes Singler zum Abschluss der vorliegenden Arbeit Rahmenbedingungen erörtert werden, wie Thread Affinity den GNU parallel mode noch beschleunigen könnte. Entsprechende gemeinsame Experimente in Form konkreten Quelltexts sind geplant.

In hohem Maße bessere Skalierungen, insbesondere lineare Skalierungen wie sie häufig für trivial parallelisierbare Algorithmen erwartet werden, werden nur möglich, wenn das Problem des Memory Gap überwunden wird und die Geschwindigkeit der Speicheranbindung gegenüber der Taktfrequenz der Prozessoren aufholt. Verbesserte und gänzlich neue Architekturen, wie z.B. die heterogene Struktur des Cell Prozessors mit der Möglichkeit asynchroner Speichertransfers, sind erste Ansätze in diese Richtung. Solche Prozessoren sind jedoch nach aktuellem Stand noch nicht in der gewohnten Art und Weise zu programmieren und erfordern erweiterte Hardware-Kenntnisse, die im Bereich der Software-Entwicklung nicht immer vorhanden sind. Letztendlich muss auch die Diskrepanz zwischen theoretischer und praktisch erreichbarer Speichertransferrate überwunden werden. Dazu erfordert es mehr, als die Speicheranbindung marketingträchtig als *double-* oder *quad-pumped* anzupreisen, während sich dies nur auf die Datenleitung bezieht und nur für aufeinanderfolgende Daten (*Burst*) funktioniert, während andere wichtige Anbindungen wie der Adressbus des FSB weiterhin nur *single-* oder *double-pumped* sind.

Die Tatsache, dass immer mehr Rechner mit Multicore-Prozessoren ausgestattet werden, wird neue Lösungen und auch neue Software-Ansätze mit sich bringen. An dem Streben Intels, mit den *Intel Threading Building Blocks* eine einfach zu programmierende Schnittstelle für parallelen Code zu etablieren, zeigt sich, wie wichtig auch den CPU-Herstellern (nicht zuletzt aus monetär motivierten Gründen) das Bewusstsein für parallele Software ist. Es ist jedoch zu erwarten, dass die Entwicklung der Prozessoren und die Anzahl logischer Ausführungseinheiten pro CPU wesentlich schneller fortschreiten wird, als die Etablierung neuer Software-Lösungen. Umso wichtiger ist das Thema und Ziel dieser Arbeit, effiziente Bausteine für parallelen Quelltext voranzutreiben, die sich auch den zukünftig eingesetzten Systemen möglichst gut anpassen können.

Abbildungsverzeichnis

1.1 Vierkern-Konzept von Intel. Quelle: [52] 2
1.2 Vierkern-Konzept von AMD. Quelle: [52] 3

2.1 Schematischer Aufbau der STAPL Bibliothek (Quelle: [43]) 8
2.2 Zusammenhänge der STAPL-Komponenten bei der parallelen Ausführung (Quelle: [18]) 14
2.3 Ablauf eines MPTL-Programms (übersetzt aus [19]) 22
2.4 MPTL Speedup für `generate()`, `transform()` und `replace_copy_if()` (Quad-core Xeon) (Quelle: [19]) 23
2.5 MPTL Speedup für `generate()`, `transform()` und `replace_copy_if()` (Sun Enterprise) (Quelle: [19]) 24
2.6 MPTL Speedup für `generate()`, `transform()` und `replace_copy_if()` (Pentium 4) (Quelle: [19]) 25
2.7 Eingliederung des libstdc++ parallel mode in die STL (Quelle: [44]) 27
2.8 `partial_sum()` für 32-bit Integer auf einem Sun T1 System (Quelle: [45]) . 34
2.9 `sort()` für 32-bit Integer auf Xeon- und Opteron-Systemen (Quelle: [45]) . . 35
2.10 verschiedene Verfahren für `find()` und 32-bit Integer auf einem Sun T1 System (Quelle: [45]) 36
2.11 Speedup für `accumulate()` auf einem Sun T1 System (Quelle: [45]) 37

3.1 Schematischer Aufbau der Multithreading-Komponenten 46
3.2 Overhead beim Aufwecken eines Threads (Intel Core 2 Duo T5450) 50
3.3 Schematischer Aufbau einer Uniform Memory Architecture (links), einer UMA im Fall eines Dual-Xeon Systems mit Dual Independent Bus (Mitte) und einer Non-Uniform Memory Architecture mit HyperTransport als Bussystem (rechts). 55
3.4 Xeon-System: Laufzeitunterschiede für Aufrufe von `std::accumulate()` auf Kern 3 bei insgesamt 4, 6 und 8 verwendeten Threads 56
3.5 Opteron-System: Laufzeitunterschiede für Aufrufe von `std::accumulate()` bei insgesamt 2, 3 und 4 verwendeten Threads 57

3.6 Xeon-System: Laufzeitunterschiede unterschiedlicher Threads für Aufrufe von `std::accumulate()` bei insgesamt 8 Threads . . . 58
3.7 Opteron-System: Laufzeitunterschiede unterschiedlicher Threads für Aufrufe von `std::accumulate()` bei insgesamt 4 Threads . . . 59
3.8 Speicherdurchsatz des parallelen `inner_product()` . . . 64
3.9 Speicherdurchsatz der parallelen SSE-Version von `inner_product()` . . . 65
3.10 Speicherdurchsatz des parallelen `inner_product()` . . . 67
3.11 Speicherdurchsatz der parallelen SSE-Version von `inner_product()` . . . 68
3.12 Speicherdurchsatz des parallelen `inner_product()` . . . 70
3.13 Schematische Darstellung des Verfahrens von Tsigas et al. Übersetzt aus [50] 74
3.14 Beispiel für die Situation nach der parallelen Neutralisierungsphase . . . 75
3.15 Visualisierung des Vorgehens nach der parallelen Neutralisierungsphase . . . 76
3.16 Beispielhafte Partitionierung eines 12-elementigen Intervalls in vier Teilintervalle . . . 78
3.17 Sequentielle Laufzeiten für `std::accumulate()` und `std::partial_sum()` auf dem Xeon- und dem Opteron-System . . . 80
3.18 Laufzeitverhältnisse sequentieller STL-Sortierverfahren (gemessen auf dem Xeon-System) . . . 83
3.19 Multiway mergesort mit Splittern . . . 84

4.1 Xeon-System: Vergleich mit der MPTL für die Funktion `find()` abhängig von der Position des gesuchten Elements . . . 92
4.2 Xeon-System: Vergleich mit der MCSTL für die Funktion `find()` abhängig von der Position des gesuchten Elements . . . 93
4.3 Opteron-System: Vergleich mit der MPTL für die Funktion `find()` abhängig von der Position des gesuchten Elements . . . 93
4.4 Opteron-System: Vergleich mit der MCSTL für die Funktion `find()` abhängig von der Position des gesuchten Elements . . . 94
4.5 Cell Blade: Vergleich mit der MPTL für die Funktion `find()` abhängig von der Position des gesuchten Elements . . . 95
4.6 Cell Blade: Vergleich mit der MCSTL für die Funktion `find()` abhängig von der Position des gesuchten Elements . . . 95
4.7 Xeon-System: Vergleich mit der MPTL für die Funktion `replace_copy_if()` 96
4.8 Opteron-System: Vergleich mit der MPTL für die Funktion `replace_copy_if()` . . . 97
4.9 Cell Blade: Vergleich mit der MPTL für die Funktion `replace_copy_if()` . 98
4.10 Xeon-System: Vergleich mit der MCSTL für die Funktion `replace_if()` . . 99
4.11 Opteron-System: Vergleich mit der MCSTL für die Funktion `replace_if()` 100
4.12 Cell Blade: Vergleich mit der MCSTL für die Funktion `replace_if()` . . . 100

4.13 Xeon-System: Vergleich mit der MCSTL für die Funktion `partition()` . . . 101
4.14 Xeon-System: Vergleich mit dem Algorithmus von Frias et al für die Funktion `partition()` . . . 102
4.15 Opteron-System: Vergleich mit der MCSTL für die Funktion `partition()` . 102
4.16 Opteron-System: Vergleich mit dem Algorithmus von Frias et al für die Funktion `partition()` . . . 103
4.17 Cell Blade: Vergleich mit der MCSTL für die Funktion `partition()` 103
4.18 Cell Blade: Vergleich mit dem Algorithmus von Frias et al für die Funktion `partition()` . . . 104
4.19 Xeon-System: Vergleich mit der MPTL für die Funktion `accumulate()` . . . 105
4.20 Xeon-System: Vergleich mit der MCSTL für die Funktion `accumulate()` . . 106
4.21 Opteron-System: Vergleich mit der MPTL für die Funktion `accumulate()` . 106
4.22 Opteron-System: Vergleich mit der MCSTL für die Funktion `accumulate()` 107
4.23 Cell Blade: Vergleich mit der MPTL für die Funktion `accumulate()` 107
4.24 Cell Blade: Vergleich mit der MCSTL für die Funktion `accumulate()` . . . 108
4.25 Xeon-System: Vergleich mit der MCSTL für die Funktion `inner_product()` 109
4.26 Opteron-System: Vergleich mit der MCSTL für die Funktion `inner_product()` 110
4.27 Cell Blade: Vergleich mit der MCSTL für die Funktion `inner_product()` . 110
4.28 Xeon-System: Speedup der SSE-Variante über die skalare Funktion `inner_-product()` . . . 111
4.29 Opteron-System: Speedup der SSE-Variante über die skalare Funktion `in-ner_product()` . . . 112
4.30 Xeon-System: Speedup der parallelen Ausführung der SSE-Variante von `in-ner_product()` . . . 112
4.31 Opteron-System: Speedup der parallelen Ausführung der SSE-Variante von `inner_product()` . . . 113
4.32 Xeon-System: Vergleich mit der MCSTL für die Funktion `partial_sum()` . 114
4.33 Opteron-System: Vergleich mit der MCSTL für die Funktion `partial_sum()` 115
4.34 Cell Blade: Vergleich mit der MCSTL für die Funktion `partial_sum()` . . . 115
4.35 Xeon-System: Vergleich mit der MCSTL für die Funktion `nth_element()` . 116
4.36 Opteron-System: Vergleich mit der MCSTL für die Funktion `nth_element()` 117
4.37 Cell Blade: Vergleich mit der MCSTL für die Funktion `nth_element()` . . . 117
4.38 Xeon-System: Vergleich mit der MCSTL für die Funktion `sort()` 118
4.39 Xeon-System: Vergleich mit der MPTL für die Funktion `sort()` 119
4.40 Opteron-System: Vergleich mit der MCSTL für die Funktion `sort()` 119
4.41 Opteron-System: Vergleich mit der MPTL für die Funktion `sort()` 120
4.42 Cell Blade: Vergleich mit der MCSTL für die Funktion `sort()` 121
4.43 Cell Blade: Vergleich mit der MPTL für die Funktion `sort()` 121
4.44 Xeon-System: Vergleich mit der MCSTL für die Funktion `stable_sort()` . 122

4.45 Opteron-System: Vergleich mit der MCSTL für die Funktion `stable_sort()` 123
4.46 Cell Blade: Vergleich mit der MCSTL für die Funktion `stable_sort()` . . . 123

5.1 Worst-Case Szenario bei Einsatz von Thread Affinity (a) und Situation bei flexiblerem Scheduling (b) 130

Literaturverzeichnis

[1] *GCC libgomp Online Dokumentation.* http://gcc.gnu.org/onlinedocs/libgomp/index.html.

[2] *GNU libstdc++ parallel mode Online Dokumentation.* http://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html.

[3] *MCSTL Webseite.* http://algo2.iti.uni-karlsruhe.de/singler/mcstl.

[4] *MPI Webseite.* http://www-unix.mcs.anl.gov/mpi/.

[5] *Pthreads Win32 Webseite.* http://sourceware.org/pthreads-win32/.

[6] *Sillicon Graphics STL Webseite.* http://www.sgi.com/tech/stl/.

[7] *STXXL Webseite.* http://stxxl.sourceforge.net/.

[8] *HyperTransport I/O Technology Overview - An Optimized, Low-Latency Board-Level Architecture.* White Paper, The HyperTransport Consortium, Juni 2004.

[9] *OpenSPARC T1 Microarchitecture Specification.* Technischer Report, Sun Microsystems Inc., August 2006.

[10] *Performance Guidelines for AMD Athlon 64 and AMD Opteron ccNUMA Multiprocessor Systems.* Application Note, Advanced Micro Devices, Inc., Juni 2006.

[11] *Intel 64 and IA-32 Architectures Optimization Reference Manual.* Technischer Report, Intel Corporation, November 2007.

[12] *Intel® 5400 Chipset Memory Controller Hub (MCH).* Datasheet, Intel Corporation, November 2007.

[13] *An enhanced Cell Broadband Engine processor with improved double-precision floating-point performance.* Technische Spezifikationen, IBM Corporation, Mai 2008.

[14] *Intel 64 and IA-32 Architectures Software Developer's Manual.* Technischer Report, Intel Corporation, April 2008.

[15] *Intel Threading Building Blocks Tutorial.* Technischer Report, Intel Corporation, März 2008.

[16] *OpenMP Application Program Interface.* Technischer Report, OpenMP Architecture Review Board, Mai 2008.

[17] Amdahl, Gene: *Validity of the single-processor approach to achieving large scale computing capabilities.* In: *Proceedings of the AFIPS Conference*, Band 30, Seiten 483–485. AFIPS Press, April 1967.

[18] An, Ping, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy M. Amato und Lawrence Rauchwerger: *STAPL: A Standard Template Adaptive Parallel C++ Library.* In: *Proceedings of the International Workshop on Advanced Compiler Technology for High Performance and Embedded Processors (IWACT)*, Seite 10. IEEE Press, 2001.

[19] Baertschiger, Didier: *Multi-Processing Template Library.* Masterarbeit, Université de Genève, 2006.

[20] Blelloch, Guy E., Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith und Marco Zagha: *A comparison of sorting algorithms for the connection machine CM-2.* In: *SPAA '91: Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, Seiten 3–16. ACM Press, 1991.

[21] Blomer, Jakub: *Effizientes Starten und Verteilen von Threads auf Mehrkernprozessoren.* Diplomarbeit, Universität Karlsruhe (TH), Juni 2008.

[22] Blumofe, Robert D. und Charles E. Leiserson: *Scheduling multithreaded computations by work stealing.* Journal of the ACM, 46(5):720–748, 1999.

[23] Dementiev, Roman, Lutz Kettner und Peter Sanders: *STXXL: Standard Template Library for XXL Data Sets.* In: *Proceedings of the 13th European Symposium on Algorithms*, Band 3669 der Reihe *LNCS*, Seiten 640–651. Springer, 2005.

[24] Dow, Eli: *Take charge of processor affinity.* `http://www.ibm.com/developerworks/linux/library/l-affinity.html`, 2005.

[25] Drepper, Ulrich und Ingo Molnar: *The Native POSIX Thread Library for Linux.* White Paper, Red Hat, Inc., Februar 2005.

[26] Dyk, Danny v., Markus Geveler, Sven Mallach und Dirk Ribbrock: *The HONEI Project - Hardware Oriented Numerics Efficiently Implemented.* `http://honei.org`.

[27] FRIAS, LEONOR und JORDI PETIT: *Parallel Partition Revisited.* In: *Proceedings of 7th International Workshop on Experimental Algorithms (WEA)*, LNCS, Seiten 142–153. Springer, 2008.

[28] FRIAS, LEONOR und JOHANNES SINGLER: *Parallelization of Bulk Operations for STL Dictionaries.* In: *Euro-Par 2007 Parallel Processing*, Band 4854 der Reihe *LNCS*, Seiten 49–59. Springer, 2007.

[29] FRIAS, LEONOR, JOHANNES SINGLER und PETER SANDERS: *Single-Pass List Partitioning.* In: *Recent Developments in Multi-Core Computing Systems*, Band 9 der Reihe *SCPE*, Seiten 179–184, September 2008.

[30] GOVINDARAJU, MADHUSUDHAN, ALEKSANDER SLOMINSKI, VENKATESH CHOPPELLA, RANDALL BRAMLEY und DENNIS GANNON: *Requirements for and evaluation of RMI protocols for scientific computing.* In: *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, Seite 61. IEEE Computer Society, 2000.

[31] HADJIDOUKAS, PANAGIOTIS E. und VASSILIOS V. DIMAKOPOULOS. In: *Euro-Par 2007 Parallel Processing*, LNCS, Seiten 662–671. Springer, 2007.

[32] ISO, INTERNATIONAL ORGANIZATION FOR STANDARDS: *Programming languages - C++*, Band 14882:2003. ANSI, 2003.

[33] JOHNSON, ELIZABETH und DENNIS GANNON: *HPC++: Experiments with the Parallel Standard Template Library.* In: *Proceedings of the 11th International Conference on Supercomputing (ICS)*, Seiten 124–131. ACM Press, 1997.

[34] KISTLER, MICHAEL, MICHAEL PERRONE und FABRIZIO PETRINI: *Cell Multiprocessor Communication Network: Built for Speed.* IEEE Micro, 26(3):10–23, 2006.

[35] KLEEN, ANDREAS: *A NUMA API for Linux.* White Paper, Novell Inc., Suse Linux Products GmbH, April 2005.

[36] MCCALPIN, JOHN D.: *Memory Bandwidth and Machine Balance in Current High Performance Computers.* IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, Dezember 1995.

[37] MCKEE, SALLY A.: *Reflections on the memory wall.* In: *Proceedings of the 1st Conference on Computing Frontiers (CF)*, Seite 162. ACM Press, 2004.

[38] MUSSER, DAVID R.: *Introspective Sorting and Selection Algorithms.* Software Practice and Experience, 27(8):983–993, 1997.

[39] RAUCHWERGER, LAWRENCE: *STAPL: The Standard Template Adaptive Parallel Library.* `http://www.cs.unm.edu/~fastos/06meeting/FASTOS-STAPL_2006.pdf`, Mai 2006.

[40] Rauchwerger, Lawrence: *SmartApps: Application Centric Computing with STAPL.* http://www.cs.unm.edu/~fastos/07meeting/FASTOS07-Rauchwerger.ppt, Juni 2007.

[41] Sanders, Peter: *Randomized Receiver Initiated Load-balancing Algorithms for Tree-shaped Computations.* Computer Journal, 45(5):561–573, 2002.

[42] Saunders, Steven: *Object-Oriented Abstractions for Communication in Parallel Programs.* Masterarbeit, Texas A & M University, Mai 2003.

[43] Saunders, Steven und Lawrence Rauchwerger: *A Parallel Communication Infrastructure For STAPL.* In: *Proceedings of the Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL)*, 2002.

[44] Singler, Johannes und Benjamin Konsik: *The GNU libstdc++ parallel mode: software engineering considerations.* In: *Proceedings of the 1st International Workshop on Multicore Software Engineering (IWMSE)*, Seiten 15–22. ACM Press, 2008.

[45] Singler, Johannes, Peter Sanders und Felix Putze: *MCSTL: The Multi-core Standard Template Library.* In: *Euro-Par 2007 Parallel Processing*, LNCS, Seiten 682–694. Springer, 2007.

[46] Tanase, Gabriel, Chids Raman, Mauro Bianco, Nancy M. Amato und Lawrence Rauchwerger: *Associative Parallel Containers In STAPL.* In: *Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Seiten 156–171, 2007.

[47] Terboven, Christian, Dieter an Mey, Dirk Schmidl, Henry Jin und Thomas Reichstein: *Data and Thread Affinity In OpenMP Programs.* In: *Proceedings of the workshop on Memory access on future processors (MAW)*, Seiten 377–384. ACM Press, 2008.

[48] Thomas, Nathan, Steven Saunders, Tim Smith, Gabriel Tanase und Lawrence Rauchwerger: *ARMI: A High Level Communication Library for STAPL.* In: *Parallel Processing Letters*, Band 16(2), Seiten 261–280, 2006.

[49] Thomas, Nathan, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato und Lawrence Rauchwerger: *A Framework For Adaptive Algorithm Selection In STAPL.* In: *Proceedings of the 10th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, Seiten 277–288. ACM Press, 2005.

[50] Tsigas, Philippas und Yi Zhang: *A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000.* In: *Proceedings of*

the 11th Euromicro Conference on Parallel Distributed and Network based Processing (PDP), Seite 372. IEEE Press, 2003.

[51] WILKES, MAURICE V.: *The Memory Gap (Keynote)*. Proceedings of the Workshop on Solving the Memory Wall Problem at the 27th International Symposium on Computer Architecture, Juni 2000.

[52] WINDECK, CHRISTOPH: *Vier gegen Vier*. c't Magazin für Computertechnik, 20/2007:164–169, September 2007.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 26. November 2008

Sven Mallach