Hochschule für Technik, Wirtschaft und Kultur Leipzig
Fakultät Informatik, Mathematik und Naturwissenschaften

Laufzeitmessung

Paralleles Mergesort mit Hilfe von OpenMP

B.Sc. Christof Pieloth

Matrikel-Nr.: 52674
Studiengang: Informatik
Modul: Algorithm Engineering

15. Februar 2010

Inhaltsverzeichnis

1 Einleitung **3**
1.1 Technische Entwicklung 3
1.2 OpenMP 3
1.3 Zielsetzung 3

2 Vorbetrachtung **5**
2.1 Testumgebung 5
2.2 Ablauf 5
2.3 Mergesort 6

3 Laufzeitmessung **7**
3.1 Implementierung ohne Schwellenwert 7
3.2 Implementierung mit Schwellenwert 9
3.3 Speedup und Effizienz 14
3.4 Paralleles Mergesort mit 16 Prozessorkernen 16

4 Fazit **19**

Literaturverzeichnis **20**

Anhang **21**

1 Einleitung

1.1 Technische Entwicklung

Vor nicht ganz zehn Jahren geriet das „Megahertz-Rennen" zwischen den Prozessorherstellern allmählich ins Stocken. Aufgrund der steigenden Taktfrequenz und der kleiner werdenden Strukturen schien vor allem die hohe Wärmeentwicklung die Leistungsgrenzen der damaligen Prozessoren aufzuzeigen. Als Lösung dieses Problems wurden Strategien zur Parallelverarbeitung entwickelt. Die Grundidee hierbei ist, dass zwei parallele Einheiten mit einer Frequenz f die selbe Arbeit leisten wie eine Einheit mit der Frequenz $2 * f$.

Im Jahr 2004 zeigte AMD den ersten Dual-Core-Prozessor mit x86-Befehlssatz [Hül08] und kurze Zeit später hielten erste Dual-Core-Prozessoren Einzug in den Desktop-Markt. Aktuell (Januar 2010) sind für den Desktop-Markt Multi-Core-Prozessoren mit bis zu vier physikalischen Kernen auf einem Chip erhältlich.

Die volle Leistung dieser Prozessoren kann mit den weit verbreiteten sequenziellen Algorithmen und Programmen jedoch nicht ausgeschöpft werden, da diese nur von einem Kern ausgeführt werden. In Programmiersprachen wie C oder Java verwendet der Softwareentwickler expliziten Parallelismus und nutzt die durch die Sprache gegebenen Möglichkeiten zur Arbeit mit Prozessen und Threads. Da sich nicht jeder Algorithmus effizient parallelisieren lässt, die gegebenen Möglichkeiten oft erheblichen Mehraufwand bei der Entwicklung verursachen und nicht zuletzt auch bis vor fünf Jahren auf Desktop-Systemen keinen Geschwindigkeitszuwachs bei der Ausführung der Programme möglich war, ist die Parallelisierung in der Anwendungsentwicklung nicht sehr weit verbreitet.

Um zukünftige Prozessoren besser Auslasten zu können, müssen Programmiersprachen und Bibliotheken entwickelt werden, die dem Entwickler die Parallelisierung von Programmen erleichtern.

1.2 OpenMP

OpenMP ist eine Programmierschnittstelle für die Entwicklung paralleler Programme in C, C++ und Fortran für Shared-Memory-Systeme [OMP]. Die Spezifikationen werden gemeinsam von vielen namhaften Hard- und Softwareherstellern definiert und dadurch von fast allen kommerziellen Compilern, sowie der GNU Compiler Collection, unterstützt.

OpenMP nutzt spezielle Compiler-Direktiven zur Parallelisierung. Dies hat den Vorteil, dass der Code in der Regel auch von Compilern übersetzt werden kann, die OpenMP nicht unterstützen. Zusätzlich ermöglicht dies ein schrittweises Parallelisieren, da der ursprüngliche Code nicht modifiziert werden muss, sondern nur die Compiler-Direktiven an den richtigen Stellen hinzugefügt werden müssen. Dies hat jedoch auch den Nachteil, dass nur sehr grob Parallelisiert werden kann.

1.3 Zielsetzung

In der Algorithmentheorie werden bevorzugt nur asymptotische Laufzeiten ohne genauere Betrachtung der Implementierung angegeben. Viele Konstanten und Faktoren werden

hierbei nicht genauer untersucht. Dadurch ist es möglich, dass ein asymptotisch langsamer Algorithmus im realen Einsatz für bestimmte Eingaben (z.B. sehr kleine) schneller ist als sein asymptotisch schnelleres Pendant.

Ziel dieser Arbeit ist die praktische Laufzeitmessung und dessen Auswertung einer sequenziellen und parallelen Implementierung des Mergesort-Algorithmus. Die sequenzielle Implementierung soll mit wenig Aufwand und mit Hilfe von OpenMP parallelisiert werden.

Ein Abriss der Möglichkeiten und des Funktionsumfanges von OpenMP ist nicht beabsichtigt.

2 Vorbetrachtung

2.1 Testumgebung

Als Testumgebung dient ein PC mit den folgenden Leistungsdaten:

CPU AMD Phenom(tm) II X4 955 (64 Bit, 4x 3.2 GHz)

RAM 4 GB DDR3 (1333 MHz)

Betriebssystem Windows 7 Professional 64 Bit

Auf diesem System ist mit Hilfe von Sun VirtualBox 3.1.2 eine virtuelle Maschine mit folgenden Daten eingerichtet:

Betriebssystem Ubuntu 9.10 64 Bit, Kernel 2.6.31-17

RAM 2 GB

Compiler GNU Compiler Collection 4.4.1

Das virtuelle System hat den Vorteil, dass die Anzahl der verfügbaren Prozessorkerne leicht verändert werden kann. Grundsätzlich kann die Anzahl der maximalen Threads in OpenMP durch `omp_set_num_threads(int num_threads)` begrenzt werden. Beim Vergleich der Programme mit 1, 2 oder 4 Threads auf einem System mit 4 aktivierten Prozessorkernen muss die Tatsache berücksichtigt werden, dass sich der Einfluss des Schedulings auf die Messergebnisse mit steigender Anzahl der Threads erhöht. Um einen möglichst konstanten Einfluss des Schedulings zu erreichen, werden die Threads anstatt mit der OpenMP-Methode durch die Anzahl der aktivierten Prozessorkerne der virtuellen Maschine begrenzt. Diese Lösung orientiert sich mehr an der Praxis, da Anwendungen auf Desktop-Systemen unabhängig vom genauen Prozessortyp eingesetzt werden und somit die genaue Anzahl der Kerne unbekannt ist. Andererseits muss berücksichtigt werden, dass der Aufwand des Schedulings aufgrund der zwei gestarteten Betriebssysteme etwas höher ist.

Die Nutzung von Linux mit dessen vielen Tools, wie zum Beispiel BASH-Skripte, GCC und Gnuplot, erleichtert den Ablauf der Tests.

Moderne Prozessoren wie der AMD Phenom(tm) II X4 haben komplexe Energiesparfunktionen. So muss das automatische Heruntertakten der Frequenz vor den Tests ausgeschaltet werden um mögliche Schwankungen der Laufzeit zu verringern.

2.2 Ablauf

Die Tests werden mit Hilfe des BASH-Skripts `runTests.sh` durchgeführt. Zu Beginn kompiliertert das Skript den Code mit aktivierten und deaktivierten OpenMP. Anschließend wird das Programm mit verschiedenen Problemgrößen gestartet. Für jede Problemgröße werden fünf Messungen vorgenommen und davon der Durchschnitt berechnet um Messungenauigkeiten durch das Scheduling zu verringern. Das Skript speichert die Testergebnisse in einer Datei im CSV-Format.

Bei den Tests wird die virtuelle Maschine mit 1, 2 oder 4 aktivierten Prozessorkernen gestartet.

2.3 Mergesort

Mergesort ist ein stabiler Sortieralgorithmus der nach dem „Divide-and-Conquer"-Prinzip arbeitet. Die asymptotische Laufzeit beträgt stets $\mathcal{O}(n \log n)$.

Mergesort arbeitet in zwei Phasen:

Aufteilen Das Feld wird solange geteilt bis ein Feld mit nur einem Element vorliegt.

Mischen Die Teilfelder werden in der gewünschten Reihenfolge (aufsteigend oder absteigend) gemischt.

Den geringsten Aufwand verursacht das Aufteilen. Hierbei wird bei einer rekursiven Umsetzung die Mergesort-Methode mit den neuen Grenzen immer wieder rekursiv aufgerufen, wodurch immer kleinere Teilfelder entstehen. Alle Teilfelder in der selben Rekursionstiefe können unabhängig voneinander bearbeitet werden. Ist die Bearbeitung zwei benachbarter Teilfelder in einer Rekursionstiefe abgeschlossen, können diese gemischt werden. Da beim Mischen die eigentliche Sortierung stattfindet, benötigt dieser Schritt den meisten Aufwand.

Das „Divide-and-Conquer"-Prinzip eignet sich aufgrund der unabhängigen Teilprobleme sehr gut für eine Parallelisierung. Jedes Teilproblem kann getrennt voneinander, demzufolge parallel, gelöst werden. Da das Mischen erst ausgeführt werden kann, wenn beide Teilprobleme fertig bearbeitet wurden, kann dieser Schritt nur sequenziell ausgeführt werden. In Abbildung 2.1 ist dieses Bearbeitungsschema grafisch dargestellt.

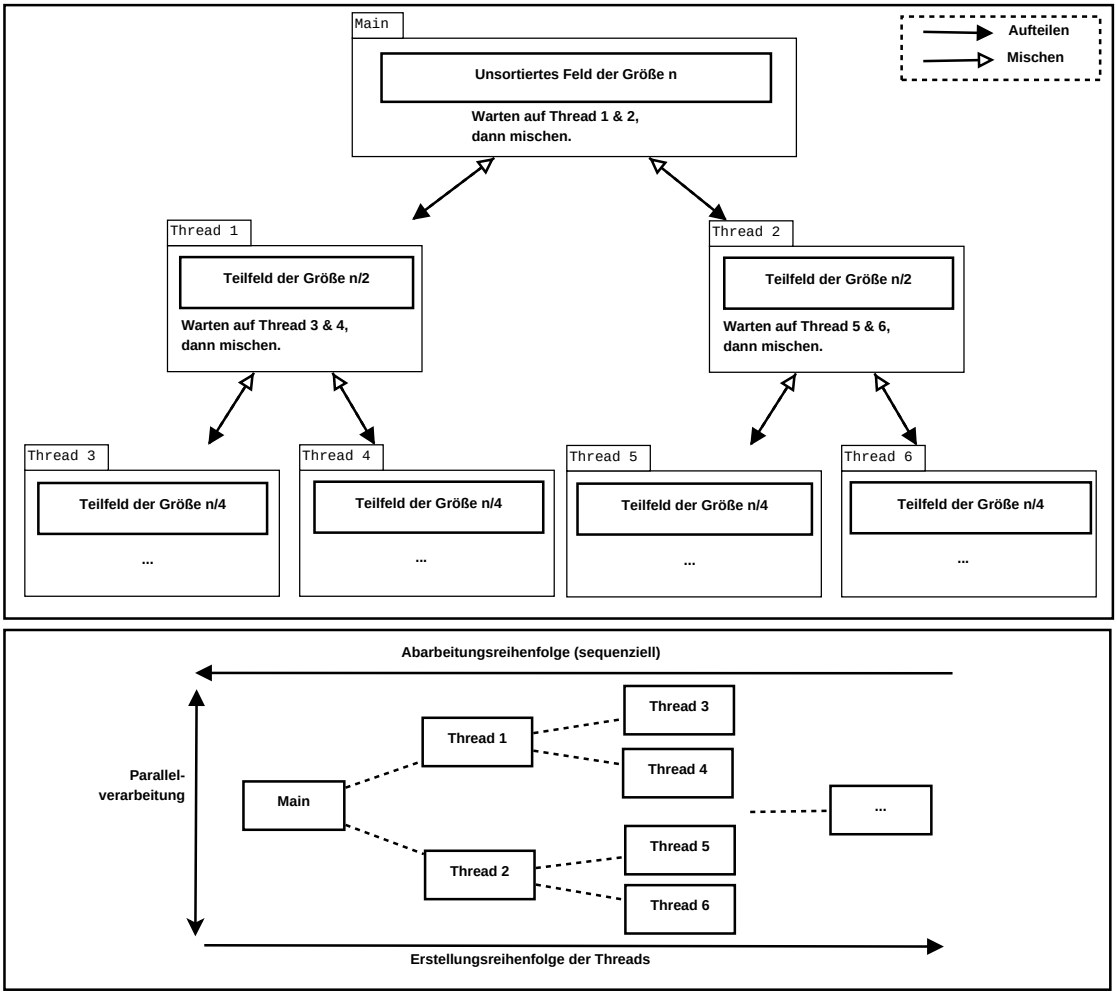

Abbildung 2.1: Bearbeitungsschema eines parallelen Mergesort.

3 Laufzeitmessung

3.1 Implementierung ohne Schwellenwert

Bei dieser Implementierung wird ohne weitere Überlegungen auf die Parallelisierungsmechanismen von OpenMP vertraut. Jedes der zwei Teilfelder in Mergesort kann unabhängig voneinander bearbeitet werden und lediglich das Mischen muss auf die fertig bearbeiteten Teilfelder warten. Folglich wird OpenMP angewiesen die zwei `mergeSort()`-Aufrufe der Teilfelder parallel auszuführen und anschließend auf deren Abarbeitung zu warten um `merge()` auszuführen.

```
void mergeSort(int* array, int low, int high) {
   #pragma omp parallel
   #pragma omp single
   {
   mergeSortParallel(array, low, high);
   }
}

void mergeSortParallel(int* array, int low, int high) {
   if (low < high) {
      int m = (low + high) / 2;
      #pragma omp task
      mergeSortParallel(array, low, m);
      mergeSortParallel(array, m + 1, high);
      #pragma omp taskwait
      merge(array, low, m, high);
   }
}
```

Die Ergebnisse sind in den Abbildungen 3.1 und 3.2 dargestellt. Entgegen den Erwartungen sind die Laufzeiten mit aktivierten OpenMP langsamer als die sequenzielle Abarbeitung. So ist Mergesort ohne OpenMP mit 10 Millionen Elementen fast 25% schneller als die parallelisierte Variante auf 2 Kernen und fast 60% schneller als das Pendant auf 4 Kernen.

Die Kosten für die Organisation der Threads sind hier als Ursache anzunehmen. Da bei jedem Aufruf von `mergeSortParallel()` 2 Threads erstellt werden, steigt die Anzahl der zu erstellenden Threads auf $2 * n - 2$!

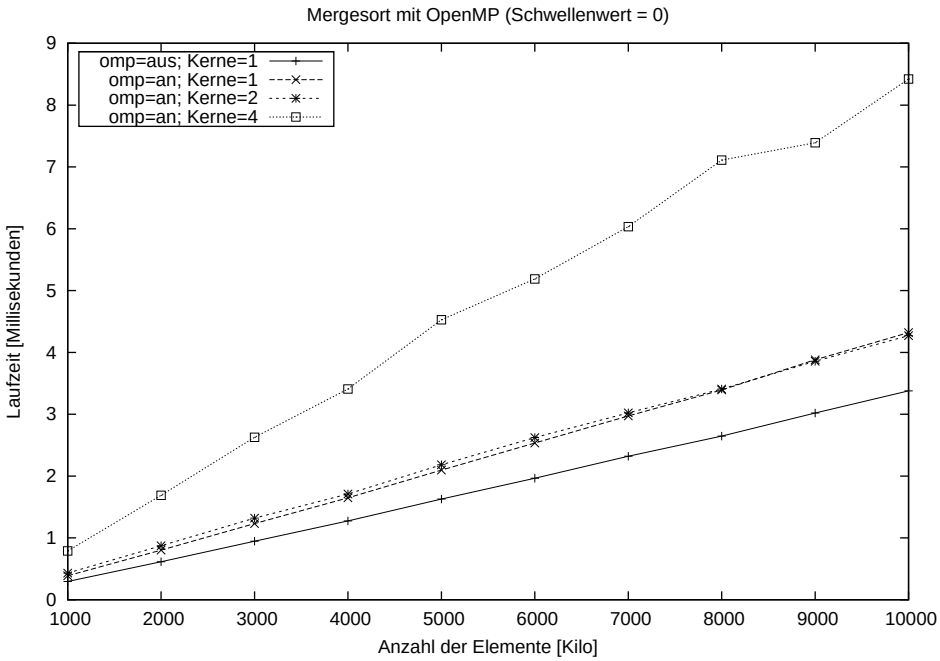

Abbildung 3.1: Laufzeiten von Mergesort ohne Schwellenwert

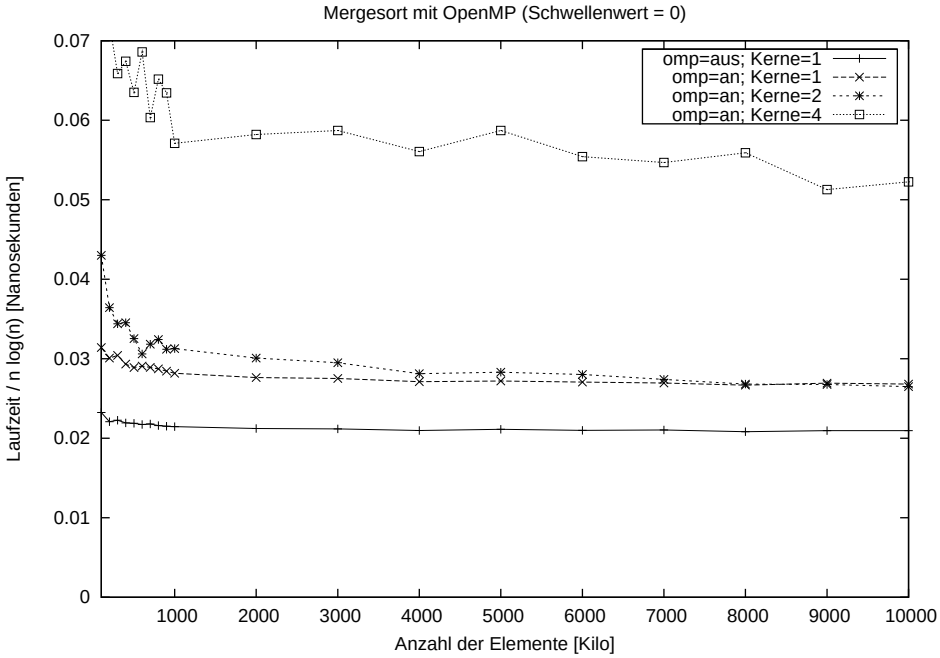

Abbildung 3.2: Laufzeiten von Mergesort ohne Schwellenwert.

3.2 Implementierung mit Schwellenwert

Um die Anzahl der zu erzeugenden Threads zu begrenzen und somit die Ausführungsgeschwindigkeit zu erhöhen, wurde der Algorithmus so erweitert, dass ab einer definierten Größe der Teilfelder keine neuen Threads mehr erzeugt werden.

```
void mergeSort(int* array, int low, int high) {
   int size = high - low + 1;
   // int threshold = ceil(sqrt(size) / 2);
   int threshold = size / _threads;
   // int threshold = THRESHOLD;
   #pragma omp parallel
   #pragma omp single
   {
   mergeSortParallel(array, low, high, threshold);
   }
}

void mergeSortParallel(int* array, int low, int high, ↩
   int threshold) {
   if (low < high) {
      int m = (low + high) / 2;
      if ((high - low + 1) <= threshold) {
         mergeSortSingle(array, low, m);
         mergeSortSingle(array, m + 1, high);
      } else {
         #pragma omp task
         mergeSortParallel(array, low, m, threshold);
         mergeSortParallel(array, m + 1, high, ↩
            threshold);
         #pragma omp taskwait
      }
      merge(array, low, m, high);
   }
}

void mergeSortSingle(int* array, int low, int high) {
   if (low < high) {
      int m = (low + high) / 2;
      mergeSortSingle(array, low, m);
      mergeSortSingle(array, m + 1, high);
      merge(array, low, m, high);
   }
}
```

Nun wurden zwei Konstanten und zwei Funktionen als Schwellenwerte für die Größe der Teilfelder getestet:

- 1000
- 100.000
- n/p; n $\widehat{=}$ Größe des zu sortierenden Feldes, p $\widehat{=}$ Anzahl der Prozessoren bzw. Kerne
- $\sqrt{n}/2$; n $\widehat{=}$ Größe des zu sortierenden Feldes

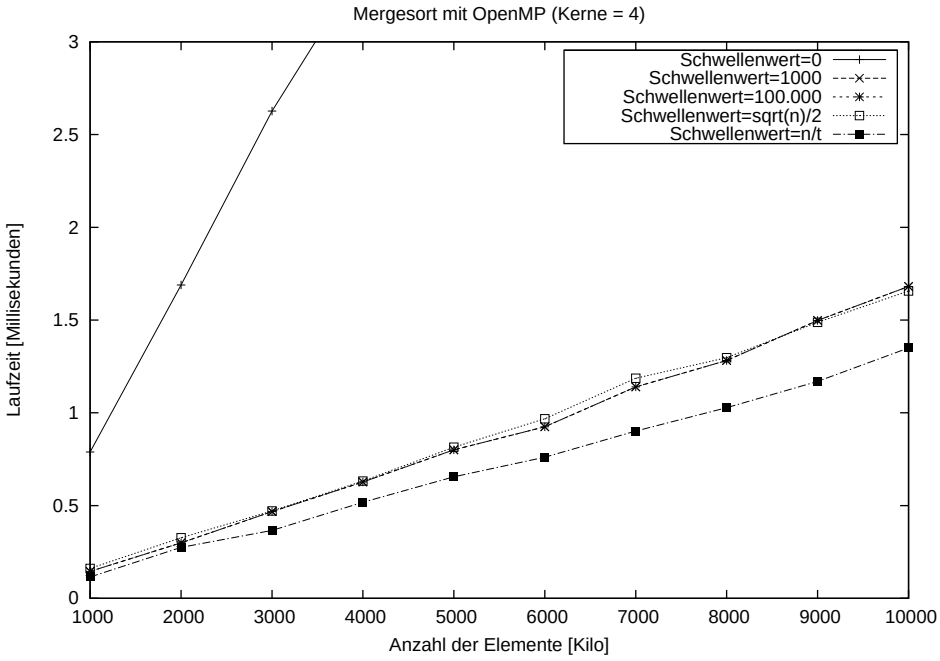

Abbildung 3.3: Laufzeit von Mergesort mit verschiedenen Schwellenwerten.

In Abbildung 3.3 sind die Laufzeiten der verschiedenen Schwellenwerten auf 4 Kernen dargestellt. Im Gegensatz zur Implementierung ohne Schwellenwerte sind alle Laufzeiten mit OpenMP schneller als der sequenzielle Algorithmus. Bis auf den Schwellenwert n/p liegen alle anderen Laufzeiten auf ungefähr den selben Niveau.

Der Schwellenwert n/p skaliert die Anzahl der Threads genauso, dass keine Threads mehr erstellt werden, wenn die möglichen parallelen Ausführungspfade die Anzahl der Prozessorkerne übersteigt. Dieser Wert ist optimal, da so möglichst wenige, aber große zu bearbeitende Aufgaben an die einzelnen Threads verteilt werden. Bei zwei Threads wird jeweils der `mergeSortParallel()`-Aufruf für das erste Teilfeld einem Thread zugeteilt und der Aufruf für das zweite Teilfeld dem anderen Thread zugeteilt. In der nächsten Rekursionstiefe der einzelnen Threads werden nun keine weiteren Threads erstellt. Somit muss nur vor dem `merge()` am Beginn des Rekursionsbaums auf die Abarbeitung der zwei Threads gewartet werden. Der Aufwand für die Organisation der Threads sinkt dadurch erheblich.

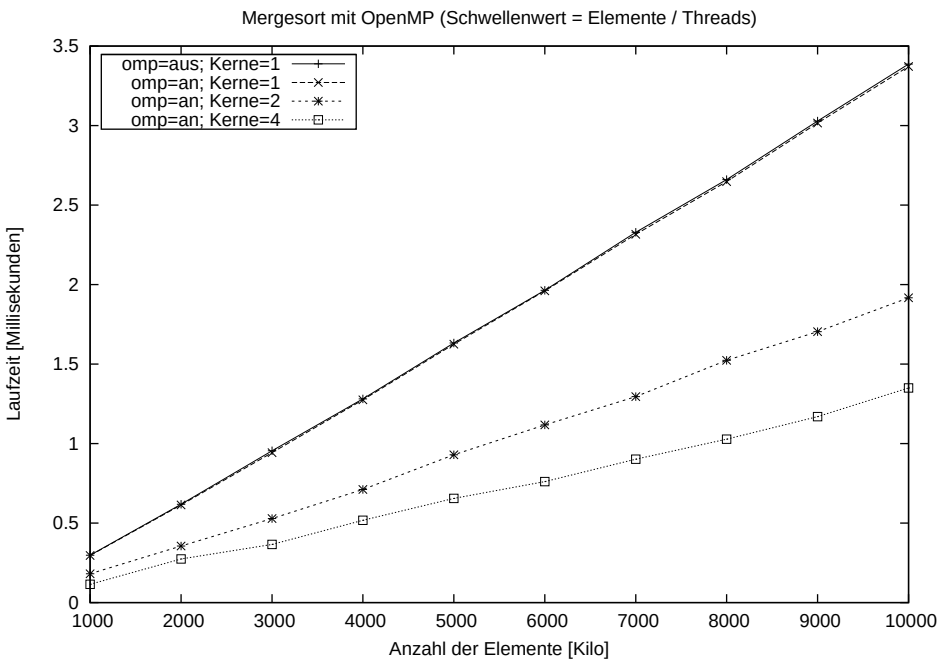

Abbildung 3.4: Laufzeit von Mergesort mit dem Schwellenwerten n/p.

Mit den folgenden Messungen kann der Schwellenwert n/p genauer untersucht werden. In Abbildung 3.4 sind die Laufzeiten für 1, 2 und 4 Kerne im Bereich von 1 bis 10 Millionen Elementen dargestellt. Das Diagramm zeigt deutlich wie die Laufzeit mit steigender Anzahl der Kernen sinkt.

Des Weiteren ist erkennbar, dass die Laufzeiten auf 1 Kern mit und ohne OpenMP nahezu identisch sind. Wird ein Programm mit aktivierten OpenMP kompiliert, aber dennoch nur auf einem Einprozessorsystem ausgeführt sind bei einer guten Implementierung keine höheren Laufzeiten zu erwarten.

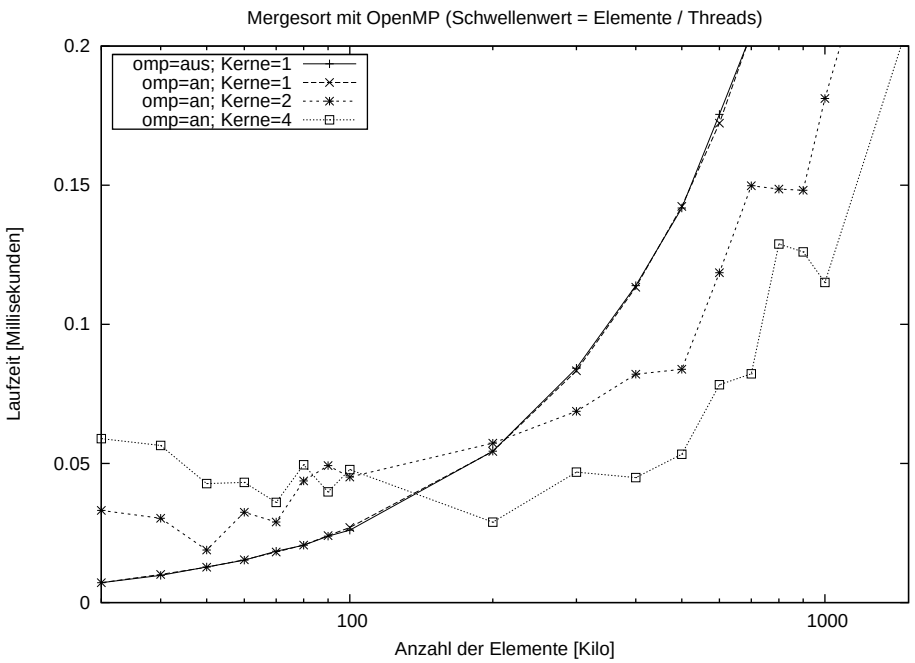

Abbildung 3.5: Laufzeit von Mergesort mit dem Schwellenwerten n/p.

Ein kleinerer Problembereich von 10 Tausend bis 1 Millionen Elementen ist in der Abbildung 3.5 dargestellt. Die x-Achse ist logarithmisch skaliert um einen großen Bereich übersichtlicher darstellen zu können. In diesem Diagramm ist sichtbar, dass alle parallelen Programme in kleineren Problembereichen langsamere Laufzeiten aufzeigen. Erst ab einem Bereich von ca. 100.000 bis 200.000 Elementen ist die Implementierung mit OpenMP auf mehreren Kernen schneller. Bei kleineren Problemgrößen beansprucht das Erstellen und Warten auf die Threads mehr Zeit als das sequenzielle Sortieren selbst.

Im Diagramm 3.6 sind alle Laufzeiten durch die asymptotische Laufzeit geteilt. Dadurch ist das Verhältnis zwischen Laufzeit und Anzahl der Kerne noch besser zu erkennen. Ab einer Problemgröße von ungefähr 1 Millionen Elementen bleibt der Abstand zwischen den Graphen gleich groß und somit ist ab diesen Problemgrößen ein konstanter Geschwindigkeitsgewinn zu erwarten.

Nun könnte jedoch angenommen werden, dass die Laufzeit nur aufgrund der steigenden Anzahl der Kerne verkürzt wird. Diese Vermutung kann genauer untersucht werden, indem die Anzahl Threads durch `omp_set_num_threads(int num_threads)` begrenzt wird und die Anzahl der Kerne auf 4 festgelegt ist. In Abbildung 3.7 sind diese Messungen dargestellt. Da in diesem Diagramm die Laufzeit mit steigender Anzahl der Threads, ähnlich wie im Diagramm 3.4, sinkt, ist bewiesen, dass die Laufzeit des Algorithmus tatsächlich durch die Anzahl der Threads gesenkt werden kann und somit eine echte Parallelverarbeitung stattfindet.

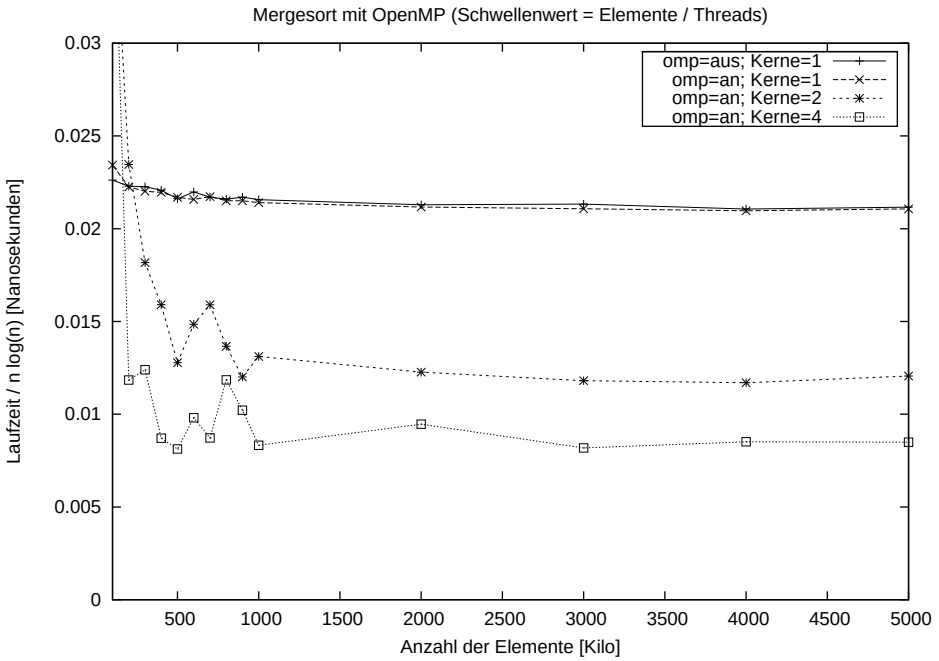

Abbildung 3.6: Laufzeit von Mergesort mit dem Schwellenwerten n/p.

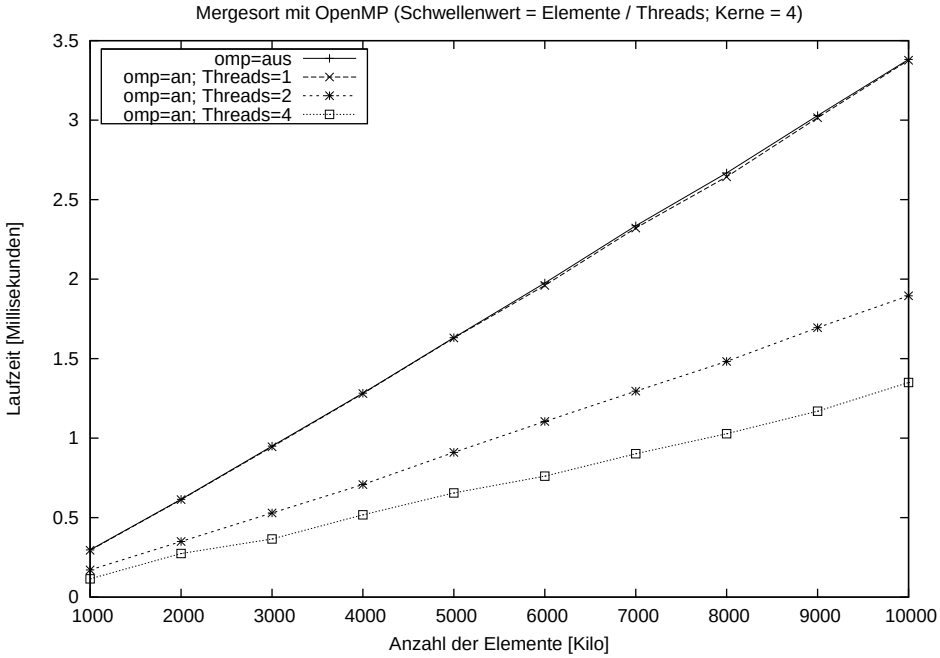

Abbildung 3.7: Laufzeit von Mergesort mit unterschiedlicher Threadanzahl auf vier Kernen.

3.3 Speedup und Effizienz

Der Speedup gibt an um welchen Faktor die Ausführung eines Programms beschleunigt werden kann, wenn dieses anstatt auf einem Einprozessorsystem, auf einem Mehrprozessorsystem ausgeführt wird. Im vorliegenden Fall berechnet sich dieser aus dem Quotienten der Gesamtlaufzeit des Algorithmus auf einem Prozessorkern und der Gesamtlaufzeit auf mehreren Kernen.

$$S(p) = \frac{T(1)}{T(p)} \quad \text{mit} \quad \begin{array}{l} \text{S(p)} \mathrel{\widehat{=}} \text{Speedup bei einem System mit p Prozessoren} \\ \text{T(1)} \mathrel{\widehat{=}} \text{Laufzeit auf einem Einprozessorsystem} \\ \text{T(p)} \mathrel{\widehat{=}} \text{Laufzeit auf einem System mit p Prozessoren} \end{array}$$

Weiterhin gilt im Allgemeinen für den Speedup $1 \leq S(p) \leq p$. Die untere Grenze von $S(p) = 1$ kann zum Beispiel bei schlechten Implementierung unterboten werden (siehe Diagramm 3.1).

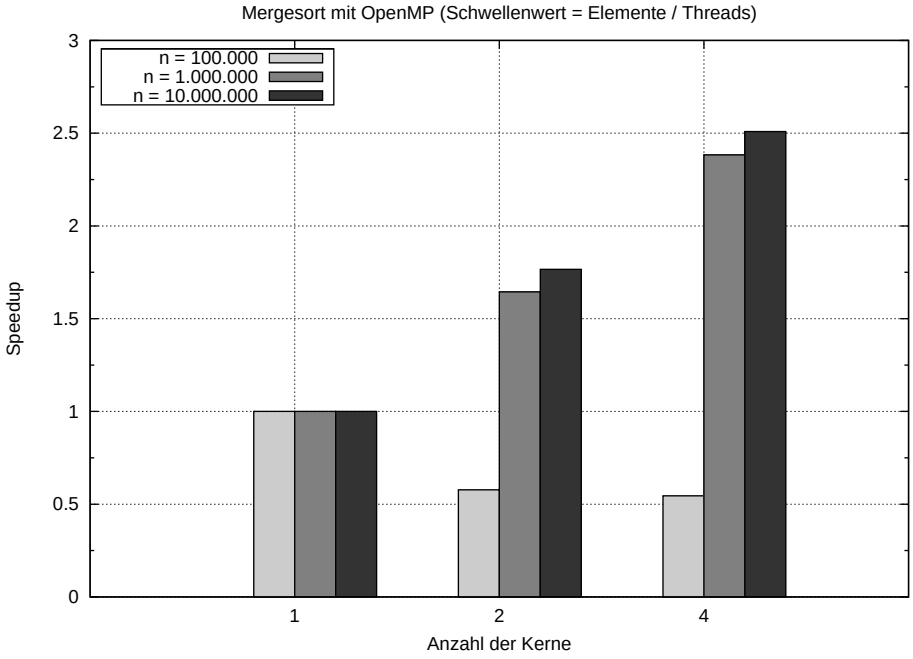

Abbildung 3.8: Speedup in Bezug auf Problemgröße und Anzahl der Kerne.

Der Speedup für verschiedene Problemgrößen in Bezug zur Anzahl der Kerne ist in Abbildung 3.8 dargestellt. Bei nur 100 Tausend Elementen ist der Speedup $S(p) < 1$. Dies liegt den Messungen aus dem Diagramm 3.5 zugrunde, aus denen hervor geht, dass erst ab ca. 100 Tausend Elementen ein Geschwindigkeitsgewinn zu verzeichnen ist. Ein Speedup $S(p) > 1$ und somit eine verkürzte Laufzeit ist bei den Problemgrößen von 1 und 10 Millionen zu erkennen. Der Speedup von $S(2) \approx 1,8$ ist dem optimalen Speedup von $S(2) = 2$ sehr nahe. Im Gegensatz dazu ist der Speedup von $S(4) \approx 2,5$ dem optimalen Speedup von $S(4) = 4$ recht weit entfernt. Daraus ist abzuleiten, dass der Speedup nicht linear steigt. Laut Abschätzung nach Giloi ist ein linearer Anstieg der Speedup-Funktion der günstigste Fall, da so die Ausführungszeit unter Zunahme weiterer Prozessoren immer merklich verkürzt werden kann. In diesem Fall liegt ein logarithmischer Anstieg vor. Somit nimmt der Geschwindigkeitsgewinn pro weiteren Prozessorkern ab und die Effizienz sinkt.

In der Parallelverarbeitung ist die Auslastung der einzelnen Recheneinheiten (in diesem Versuch die Prozessorkerne) eine obere Schranke für die Effizienz. Werden alle Prozessorkerne zu 100% während der gesamten Laufzeit ausgelastet, ist eine optimale Effi-

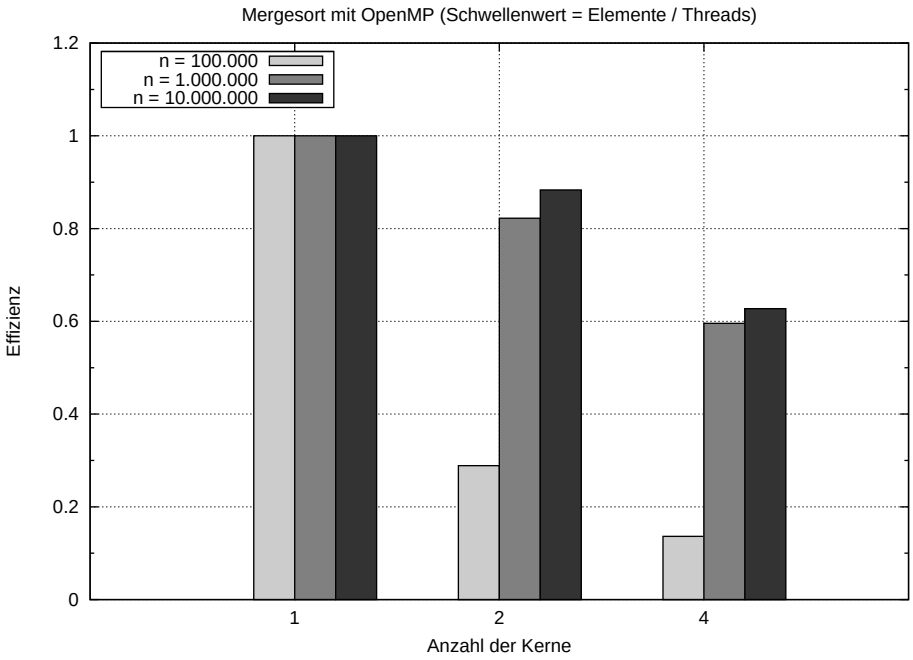

Abbildung 3.9: Effizienz in Bezug auf Problemgrößen und Anzahl der Kerne.

zienz vorhanden. In Diagramm 3.9 ist zu erkennen, dass die Effizienz mit zunehmender Anzahl der Kerne abnimmt. Des Weiteren ist auch hier wieder die schlechte Skalierung bei kleinen Problemgrößen sichtbar.

In Abbildung 3.10 ist die Prozessorauslastung bei Ausführung des parallelen Mergesort mit 1, 2 und 4 Threads auf 4 Kernen dargestellt. Auf diesem Bild sind sehr gut die verkürzte Laufzeit bei steigender Anzahl der Threads und die Auslastung der einzelnen Kerne zu erkennen. So wird in der Zeit zwischen 35 und 55 Sekunden nur ein Kern ausgelastet, da der Algorithmus mit nur 1 Thread gestartet wurde. Bei der Ausführung mit 4 Threads zwischen 5 und 12 Sekunden ist zu erkennen, dass weniger Zeit benötigt wird und die 4 Kerne unterschiedlich stark ausgelastet werden.

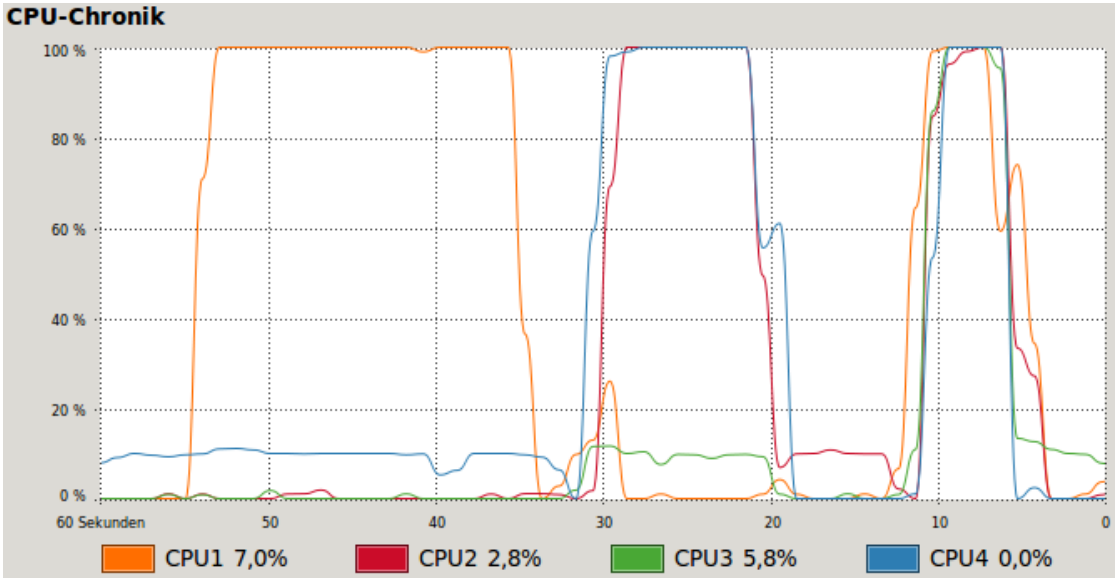

Abbildung 3.10: Prozessorauslastung bei Ausführung des parallelen Mergesort mit ein, zwei und vier Threads.

3.4 Paralleles Mergesort mit 16 Prozessorkernen

Mit freundlicher Unterstützung von Marcel Weiss vom *Max Planck Institute for Human Cognitive and Brain Sciences* wurde dieser Algorithmus auf folgendem Testsystem ausgeführt:

CPU 4x AMD Opteron(tm) 8382 (64 Bit, 4x 2.6 GHz)

RAM 32 x 4 GB (ECC, 533 MHz)

Somit wurde ein System mit insgesamt 16 Prozessorkernen und 128 GB Speicher genutzt. Besonderen Einfluss auf die Messwerte hat vor allem die Anzahl der verfügbaren Prozessorkerne. Die Größe des verfügbaren Speichers ist aufgrund der maximal Problemgröße von 10 Millionen Elementen, welche sich im Bereich von einigen zehn Megabyte befindet, eher vernachlässigbar.

Die Testläufe wurden wieder mit Hilfe des BASH-Skripts `runTests.sh` ausgeführt. Da immer alle 16 Prozessorkerne aktiviert waren, wurden die maximalen Threads in OpenMP durch `omp_set_num_threads(int num_threads)` begrenzt. Diese Tests sollen hauptsächlich die Skalierung des Algorithmus bei einer größeren Anzahl von Prozessorkernen zeigen. Da die zuvor getroffenen Erläuterungen auch auf diese Messergebnisse zutreffen, wird nur kurz auf Besonderheiten eingegangen.

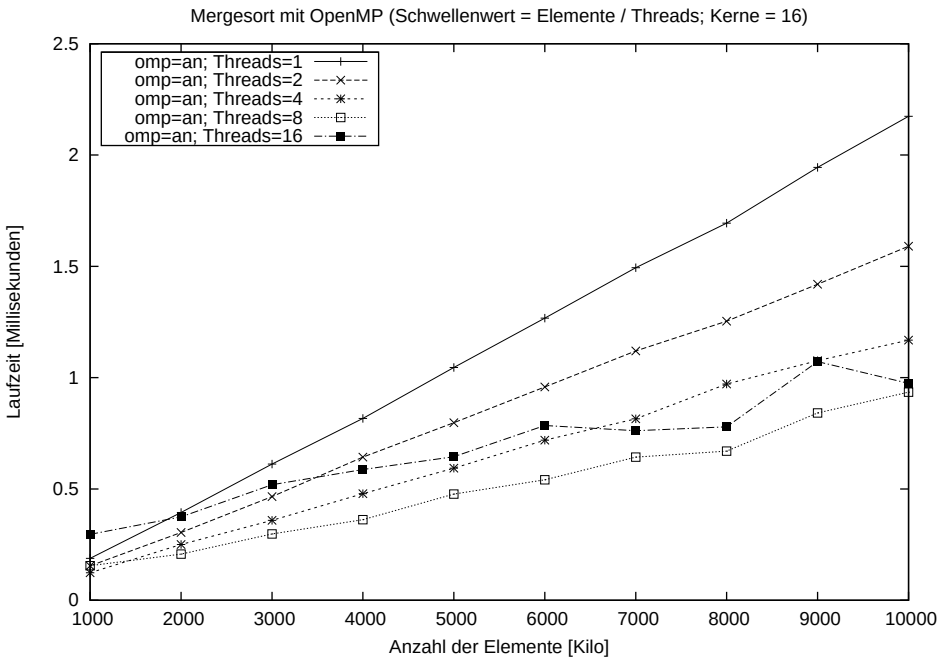

Abbildung 3.11: Laufzeit von Mergesort mit 16 Prozessorkernen.

In Abbildung 3.11 sind die Laufzeiten bei verschiedener Anzahl der Threads dargestellt. Wie im Diagramm 3.4 sinkt auch hier die Laufzeit mit steigender Anzahl der Threads. Lediglich die Laufzeit bei 16 Threads ist in dem dargestellten Bereich langsamer als die bei 8 Threads. Da der Anstieg des Graphen für 16 Threads geringer ist als der Anstieg der anderen Graphen ist anzunehmen, dass die Problemgröße für die Anzahl der Threads zu gering ist. Ab einer gewissen Anzahl von Elementen sollte sich die Laufzeit bei 16 Threads in Bezug auf 8 Threads verkürzen.

Das Diagramm 3.12 zeigt den Speedup im Bezug auf die Problemgröße und Anzahl der Threads. Im Gegensatz zum Speedup mit 4 Kernen fällt auf, dass die Abhängigkeit zwischen Speedup und Problemgröße mit steigender Anzahl der Threads zunimmt. Da die Größe der Teilfelder mit steigender Anzahl der Threads geringer wird, nimmt der Aufwand für die Organisation der Threads im Verhältnis zur eigentlichen Sortierzeit zu. Um einen konstanten Speedup des Algorithmus sicherzustellen, müsste die Anzahl der Threads in Abhängigkeit zur Problemgröße und den verfügbaren Prozessorkernen beschränkt werden.

Der logarithmische Anstieg des Speedup ist im Diagramm 3.12 noch stärker zu erkennen als im Diagramm 3.8. Im Zusammenhang mit der Effizienz in Abbildung 3.13 kann die Aussage getroffen werden, dass die Zunahme von weiteren Prozessoren mit steigender Anzahl immer weniger Nutzen bringt.

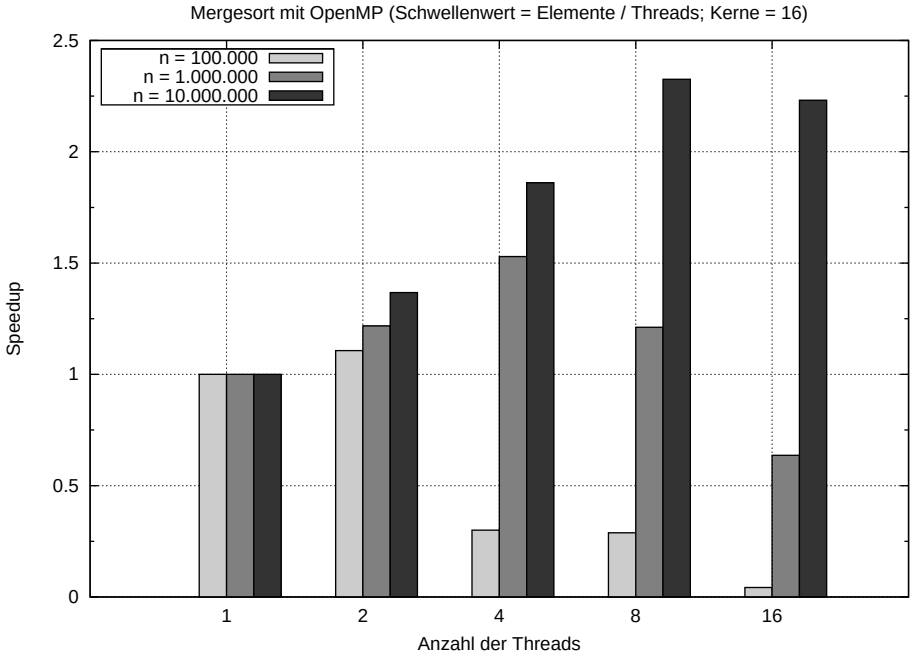

Abbildung 3.12: Speedup mit 16 Prozessorkernen.

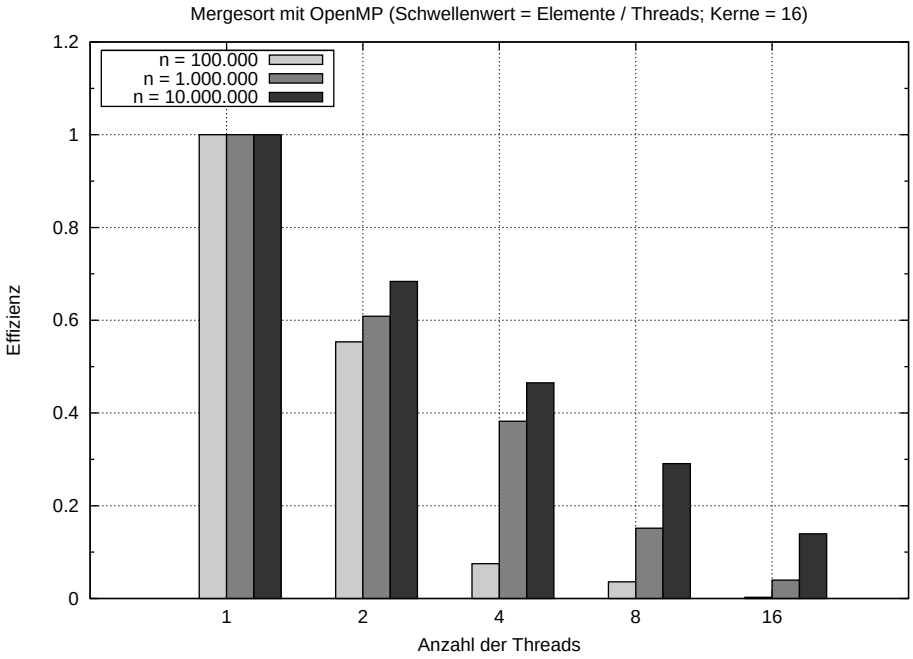

Abbildung 3.13: Effizienz mit 16 Prozessorkernen.

4 Fazit

Ziel dieser Arbeit war die Portierung eines sequenziellen Mergesort zu einem parallelen Mergesort mit Hilfe von OpenMP, sowie die Messung und Auswertung der praktischen Laufzeit.

Die Implementierung eines parallelen Mergesort ist mit sehr wenig Aufwand gelungen. Mit den wenigen Compiler-Direktiven

- `#pragma omp parallel`
- `#pragma omp single`
- `#pragma omp task`
- `#pragma omp taskwait`

konnte eine Parallelverarbeitung erzielt werden. Die Messergebnisse zeigen jedoch, dass zusätzlich zu OpenMP weitere Eingriffe nötig sind, um auch den erhofften Geschwindigkeitszuwachs zu erhalten.

Aufgrund der rekursiven Implementierung von Mergesort werden bei jeder Rekursionstiefe Threads erstellt. Da die Organisation von Threads einen zusätzlichen Aufwand in sich birgt, muss ein möglichst optimales Verhältnis zwischen der Anzahl der Threads und der Größe deren Arbeitspakete gefunden werden. Dies kann nicht durch OpenMP gelöst werden, sondern muss für jeden Algorithmus separat berechnet werden.

Im vorliegenden Fall wurde dies durch Schwellenwerte umgesetzt. Ab einer bestimmten Größe der Teilfelder sollen keine neuen Threads mehr angelegt werden. Für Mergesort ist ein Schwellenwert von n/p mit n Elementen und p Prozessoren optimal, da keine Threads mehr erstellt werden, wenn genau p Teilfelder bearbeitet werden. Durch diese Beschränkung der Threads konnte die Laufzeit mit steigender Anzahl der Prozessorkerne gesenkt werden.

Doch diese Beobachtung tritt nicht bei allen Problemgrößen auf. Da der Aufwand für die Organisation der Threads konstant ist, müssen diese Kosten durch eine schnellere Bearbeitung ausgeglichen werden. Bei kleinen Problemgrößen beansprucht die Organisation der Threads jedoch mehr Zeit als das eigentliche Sortieren und somit ist das sequenzielle Pendant schneller. Schlussfolgernd steigt die minimale Anzahl der Elemente, ab der ein Geschwindigkeitsgewinn zu verzeichnen ist, mit der steigenden Anzahl der Prozessorkerne an.

Für den praktischen Einsatz, bei dem die Anzahl der Elemente und der vorhandenen Prozessorkerne oft variabel und unbekannt sind, müssen diese Schwellenwerte für die Parallelverarbeitung, ab wann wieviele Threads genutzt werden sollen, zusätzlich in der Implementierung berücksichtigt werden. Ähnliche Verfahren werden auch in Standardbibliotheken angewendet. So sortiert zum Beispiel in Java 6 die Klasse `java.utils.Arrays` kleinere Felder mit Insertionsort und größere Felder in Quicksort [JAP].

Literaturverzeichnis

[GS10] Gleim, Urs ; Stal, Dr. M.: Parallel Programming. In: *iX Special* (1/2010), S. 105 – 110

[Hül08] Hülsbömer, Simon: Der x86-Prozessor wird 30 - wie Intel dank IBM alle Gipfel stürmte. In: *Computerwoche* (06/2008). `http://www.computerwoche.de/hardware/notebook-pc/1866928/index8.html`. – Abruf: 22.01.2010

[JAP] *Java(tm) Platform, Standard Edition 6, API Specification.* `http://java.sun.com/javase/6/docs/api/`. – Abruf: 06.02.2010

[Lau09] Lau, Oliver: Programme parallelisieren mit OpenMP. In: *c't extra* (02/2009), S. 50 – 55

[OMP] *The OpenMP API specification for parallel programming.* `http://www.openmp.org`. – Abruf: 10.01.2010

Anhang

BASH-Skript `runTests.sh` **zum automatisierten Ablauf der Tests:**

```
#!/bin/bash

# Variables
printf "Initializing ..."
PRG="sort"
DATA="data"
SUFFIX=".csv"
PRGOMP="sortOpenMP"
START=100
END=10000000
ROUNDS=5
printf "done!\n"

# Functions
RETURN=0
getTotalSum () {
   input=$1
   array=`echo $input | sed ↩
      's/\([a-z]*=[0-9]*\);/\1\n/g' | grep '^total' | ↩
      tr '=' ' ' | awk '{ print $2 }'`
   sum=0
   for k in $array ; do
      let "sum += $k"
   done
   RETURN=$sum
}

doTests() {
   PROG=$1
   FILE=$2
   THREADS=$3
   FILE=$FILE$THREADS$SUFFIX
   echo "Program: "$PROG" Datafile: "$FILE" Threads: ↩
      "$THREADS
   # Overwrite old file
   printf "" > $FILE
   # Run test from start to end
   STEP=$START
   for((i=$START; i<=$END; i=i+$STEP)) do
      OUTPUT=""
      # Do for each n <£ROUNDS> tests
      for((j=0; j<$ROUNDS; j++)) do
         OUTPUT=$OUTPUT`./$PROG $i 0 1`
         printf "."
      done
      # Calculate average
      getTotalSum $OUTPUT
      SUM=$RETURN
      let "SUM =$SUM / $ROUNDS"
      # Write to file
```

```
      printf $i"\t"$SUM"\n" >> $FILE
      printf "\n"
      # Increase steps for each power
      let "tmp=$i/$STEP"
      if [ $tmp -ge "10" ] ; then
      let "STEP *= 10"
      fi
      # Restrict steps (optional)
      # if [ £STEP -ge "1000000" ] ; then
      # let "STEP = 100000"
      # fi
   done
}

# Compile
printf "Compiling ..."
gcc -o $PRG $PRGOMP".c" -lm
gcc -fopenmp -o $PRGOMP $PRGOMP".c" -lm
printf "done!\n"

# Run
printf "Testing ...\n"
doTests $PRG $DATA "0"
doTests $PRGOMP $DATA "1"
doTests $PRGOMP $DATA "2"
doTests $PRGOMP $DATA "4"
printf " done!\n"

# Plot
printf "Plot ..."
gnuplot "graph.gnu"
printf " done!\n"
```

Mergesort mit OpenMP `sortOpenMP.c`**:**

```
/*
 * File:   sortOpenMP.c
 * Author: Christof Pieloth
 * Date: January 2010
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/times.h>
#include <math.h>

#ifdef _OPENMP
#include <omp.h>
#endif

#define TIME 1
#define CHECK 2
#define ARRAY 3
#define TIMEFACTOR 1000000
#define THRESHOLD 100000

void mergeSort(int* array, int low, int high); void ↵
   mergeSortSingle(int* array, int low, int high);
```

```
void mergeSortParallel(int* array, int low, int high, ↩
   int threshold);
void merge(int* array, int low, int mid, int high);
void createRandomArray(int* array, int size);
void printArray(int* array, int size);
void isSorted(int* array, int size);
long int getNano(struct timespec tv);

short _threads = 0;

int main(int argc, char** argv) {
   struct timespec start, end;
   int n;
   short debug = 0;

   // Check arguments
   if (argc < 4) {
      printf("Usage: sortOpenMP <elements> <threads> ↩
         <output level>\n");                ↩
         exit(EXIT_FAILURE);
   } else {
      n = atoi(argv[1]);
      #ifdef _OPENMP
      _threads = atoi(argv[2]);
      if (_threads > 0)
         omp_set_num_threads(_threads);
      else
         _threads = omp_get_num_procs();
      #endif
      _threads = _threads == 0 ? 1 : _threads;
      debug = atoi(argv[3]);
   }

   int* array = malloc(n * sizeof (int));
   if (array == NULL) {
      exit(EXIT_FAILURE);
   }
   createRandomArray(array, n);
   if (debug >= ARRAY)
      printArray(array, n);

   gettimeofday(&start, NULL);
   mergeSort(array, 0, n - 1);
   gettimeofday(&end, NULL);

   if (debug >= ARRAY)
      printArray(array, n);
   if (debug >= CHECK)
      isSorted(array, n);
   if (debug >= TIME)
      printf("total=%ld;\n", getNano(end) - ↩
         getNano(start));

   free(array);
   exit(EXIT_SUCCESS);
}

/**
  * Sorts the array.
```

```
  * NOTE: high = array.size - 1
  */
void mergeSort(int* array, int low, int high) {
   int size = high - low + 1;
   // int threshold = ceil(sqrt(size) / 2);
   int threshold = size / _threads;
   // int threshold = THRESHOLD;
   #pragma omp parallel
   #pragma omp single
   {
   mergeSortParallel(array, low, high, threshold);
   }
}

/**
  * Sorts the array using OpenMP.
  */
void mergeSortParallel(int* array, int low, int high, ↩
   int threshold) {
   if (low < high) {
      int m = (low + high) / 2;
      if ((high - low + 1) <= threshold) {
         mergeSortSingle(array, low, m);
         mergeSortSingle(array, m + 1, high);
      } else {
         #pragma omp task
         mergeSortParallel(array, low, m, threshold);
         mergeSortParallel(array, m + 1, high, ↩
            threshold);
         #pragma omp taskwait
      }
      merge(array, low, m, high);
   }
}

/**
  * Sorts the array.
  */
void mergeSortSingle(int* array, int low, int high) {
   if (low < high) {
      int m = (low + high) / 2;
      mergeSortSingle(array, low, m);
      mergeSortSingle(array, m + 1, high);
      merge(array, low, m, high);
   }
}

/**
  * Merges the array.
  */
void merge(int* array, int low, int mid, int high) {
   int i, j, k;
   int *help = malloc((mid - low + 1) * sizeof (int));
   if(help == NULL) {
      exit(EXIT_FAILURE);
   }
    i = 0;
   j = low;
   // Copy first half of the array into help
```

```
   while (j <= mid)
      help[i++] = array[j++];
    i = 0;
   k = low;
   // Copy back the next greater elements
   while (k < j && j <= high)
      if (help[i] <= array[j])
         array[k++] = help[i++];
      else
         array[k++] = array[j++];
   // Copy back the rest of help
   while (k < j)
      array[k++] = help[i++];
   free(help);
}

/**
  * Fills the array with random numbers.
  */
void createRandomArray(int* array, int size) {
   int i = 0;
   srand(time(NULL));
   for (i = 0; i < size; i++)
      array[i] = (int) rand();
}

/**
  * Prints the array.
  */
void printArray(int* array, int size) {
   char sep = '>';
   int i;
   for (i = 0; i < size; i++) {
      printf("%c %d", sep, array[i]);
      sep = sep == '>' ? ',' : sep;
   }
   printf("\n");
}

/**
  * Counts the elements which are in wrong order.
  */
void isSorted(int* array, int size) {
   int errors = 0;
   int i;
   for (i = 1; i < size; i++) {
      if (array[i - 1] > array[i])
         errors++;
   printf("%d elements are in wrong order!\n", errors);
}

/**
  * Returns the time in nanoseconds.
  */
long int getNano(struct timespec tv) {
   return tv.tv_sec * TIMEFACTOR + tv.tv_nsec;
}
```