Algorithmenbibliotheken für Mehrkern-Prozessoren

Johannes Singler
Institut für Theoretische Informatik
Karlsruher Institut für Technologie (KIT)
singler@kit.edu

Abstract: Allgegenwärtige Mehrkern-Prozessoren erfordern parallelisierte Programme, um die bereitgestellte Rechenleistung voll auszunutzen. In dieser Dissertation [Sin10] werden Bibliotheken parallelisierter Algorithmen implementiert, mit denen der Anwender auf einfache Weise, gekapselt und implizit, eine Anwendung parallelisieren kann. Betrachtet werden grundlegende Algorithmen für Hauptspeicher und Externspeicher sowie geometrische Algorithmen. Experimente, entweder mit den Algorithmen für sich oder in einer Anwendung, zeigen signifikante Beschleunigung. Als Generalisierung auf verteilten Speicher werden Sortieralgorithmen für Rechnerbündel entworfen, deren reale Leistung neue Weltrekorde aufgestellt haben.

1 Motivation und Ansatz

Die Geschwindigkeit sequentieller Prozessoren stieg früher von Jahr zu Jahr kontinuierlich an. Die Leistung von Softwareapplikationen erhöhte sich damit automatisch, und das praktisch ohne Aufwand von Seiten der Anwendungsentwickler. Diese Entwicklung kam 2005 zum Ende, bedingt vor allem durch die immer höhere Verlustleistung auf Grund der ansteigenden Taktraten. Das immer noch steigende Transistor-Budget eines Chips wird seitdem stattdessen in mehrere Prozessorkerne investiert.

Daher *müssen* Software-Entwickler inzwischen die Leistung der Mehrkern-Prozessoren durch parallele Ausführung des Programms ausnutzen, um eine Stagnation der Rechenleistung zu vermeiden. Dies ist eine große Herausforderung und hat zu einem Paradigmenwechsel in der Informatik geführt. Eine manuelle Parallelisierung ist aufwändig, und automatische Parallelisierung durch den Übersetzer meist nicht weitgehend genug.

In dieser Arbeit untersuchen wir den Ansatz, das Parallelisieren von Anwendungen durch parallelisierte Algorithmenbibliotheken zu erleichtern. Diese bieten die gleiche Funktionalität wie ihre sequentiellen Versionen, ihre Routinen laufen jedoch beschleunigt, da sie die mehreren Rechenkerne nutzen. Die Verwendung von Routinen solcher Bibliotheken ermöglicht dem Software-Entwickler auf einfache Weise impliziten Parallelismus, gekapselt durch die gleichbleibende Schnittstelle. Wir entwerfen bzw. wählen Algorithmen für die verschiedenen Probleme und implementieren sie.

Da das Ziel eine höhere Ausführungsgeschwindigkeit ist, gehen wir von einer Plattform aus, die das auch im sequentiellen Umfeld bereits anstrebt. Wir haben uns daher auf etablierte Bibliotheken für die Programmiersprache C++ konzentriert. Diese weit verbrei-

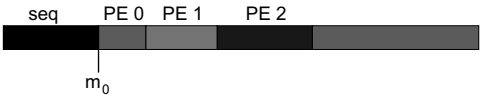

Abbildung 1: Schema des Algorithmus für `find`.

tete Sprache unterstützt gleich mehrere Programmierparadigmen, z. B. objekt-orientierte und generische Programmierung. Weiterhin folgt sie dem „Zero-Overhead“-Prinzip, d. h. Spracheigenschaften bringen nur Zusatzkosten, wenn sie wirklich verwendet werden, und dann auch nur im notwendigen Ausmaß. Letztlich ergibt sich daraus der Anspruch, dass man mit Hilfe eines gut optimierenden Übersetzers immer die Leistung (einer einzelnen) Recheneinheit ausreizen kann, und keinen Nachteil nur auf Grund der Sprache hat.

Bei numerischen Problemen ist das Bereitstellen leistungsfähiger Implementierungen in Form von Bibliotheken schon länger Tradition. Da numerische Operationen oft explizite Hardware-Unterstützung genießen, sind hier architekturspezifische Lösungen wichtiger und einträglicher. Damit einher ging schon früh der Ansatz zur Parallelisierung.

Im folgenden liegt der Fokus jedoch auf kombinatorischen Algorithmen, und zwar für parallele Systeme mit gemeinsamem Speicher (vulgo Mehrkern-Rechner). Insgesamt betrachten wir Basisalgorithmen für Haupt- und Externspeicher, sowie (kombinatorische) geometrische Algorithmen. Dazu kommt der Ansatz zur Verallgemeinerung auf verteilten Speicher an Hand der Anwendung Sortieren größter Datenmengen, wo zusätzlich ebenfalls Mehrkern-Parallelismus genutzt wird.

2 Basis-Algorithmen im Hauptspeicher

Als erstes und grundlegendstes Fallbeispiel betrachten wir die Standard Template Library (STL), eine Bibliothek für die grundlegendsten Algorithmen, welche als Teil von C++ standardisiert ist. Ihr einfachster Algorithmus wendet eine benutzerdefinierte Operation auf jedes Element einer Sequenz an. Weitere dort bereitgestellte Routinen sind z. B. suchen, mischen, sortieren, zufällig permutieren und Präfixsummen berechnen, sowie partitionieren mittels eines Pivot-Elements. Die meisten STL-Algorithmen arbeiten auf einer oder mehreren Sequenzen, wobei der Datentyp eines einzelnen Elements generisch ist. Die Implementierungen sind typischerweise effizient, wenn wahlfreier Zugriff auf diese Elemente schnell ist. In der Praxis funktionieren Sie also nur gut für im Hauptspeicher liegende Daten.

Eine verwandte Arbeit ist STAPL [AJR+01], eine Bibliothek, die STL-ähnliche Container und Algorithmen für verteilten Speicher bietet. Die Intel Threading Building Blocks sind vor allem ein Rahmenwerk für Task-Parallelismus. Sie bieten nur wenige echte Algorithmen, dafür umso mehr thread-sichere Datenstrukturen. Letzteres steht jedoch nicht im Fokus dieser Arbeit.

Im Rahmen dieser Dissertation haben wie die Routinen, für die es sinnvoll war, daten-

parallel reimplementiert. Manche Operationen benötigen jedoch zu wenig Zeit (z. B. nur logarithmisch in der Eingabegröße), als dass sich eine Parallelisierung auf Grund des zu erwartenden Parallelismus-Overheads rechnen würde.

Selbst einfache Algorithmen zeigen in der parallelen Variante Kniffligkeiten, z. B. `find`, das erste Vorkommen eines bestimmten Elements in einer Sequenz zu ermitteln. Der sequentielle Algorithmus terminiert sofort, sobald das passende Element an Position m gefunden wird. Eine naive Parallelisierung, die die Sequenz in gleich langen Stücken pro Thread durchsucht, würde hier im schlimmsten Fall zu keiner Beschleunigung, sondern sogar zu einer Verlangsamung (bei Synchronisation erst nach kompletter Abarbeitung jedes Stücks) führen. Im letztlich implementierten Algorithmus arbeiten die Threads blockweise von Anfang weg (siehe Abbildung 1), wobei sie die Position des nächsten Blocks mittels atomarer Additionen auf einen Zähler synchronisieren. Da ein Block aber immer komplett durchsucht wird, ergibt sich ein Onlineproblem für die Blockgröße. Fordern wir konstante parallele Effizienz, so ergibt sich eine Proportionalität zu m. Der Algorithmus hält folglich die Blockgröße proportional zur aktuellen Startposition, der besten aktuell bekannten unteren Schranke für m.

Im Allgemeinen bevorzugen wir Algorithmen, die möglichst wenig Kommunikation bzw. Synchronisation benötigen, da dies auch auf Systemen mit gemeinsamem Speicher meist zu besserer Leistung führt. So basiert ein Sortieralgorithmus auf parallelem lokalem Sortieren mit folgendem Mehrwege-Mischen.

Algorithmen zur Manipulation von speziellen Datenstrukturen werden ebenfalls thematisiert. Für das Einfügen großer Datenmengen in eine sortierte Sequenz bzw. deren Konstruktion aus einer Eingabesequenz betrachten wir parallele Algorithmen für Rot-Schwarz-Bäume.

Unsere parallelisierte Version der STL heißt *Multi-Core Standard Template Library (MCSTL)* [PSS07, SSP07]. Wie evaluieren die Leistung ihrer Algorithmen-Implementierungen ausführlich auf verschiedenen Mehrkern-Rechnern. Beispielhaft seien hier drei Resultate auf einer Maschine mit zwei Vierkern-Prozessoren vorgestellt. Sortieren von 32-Bit-Ganzzahlen (Abbildung 2) skaliert sehr gut, die Beschleunigung liegt über 6, und deutliche Geschwindigkeitsvorteile ergeben sich bereits für wenige tausend Elemente in der Eingabe. Zufälliges permutieren (`random_shuffle`, Abbildung 4) erreicht sogar superlineare Beschleunigung. Dies erklärt sich damit, dass der parallele Algorithmus hierarchisch arbeitet und damit cache-effizienter ist. Für Eingaben ab einer gewissen Größe ist er dadurch schon mit nur einem Thread zweimal so schnell wie die Originalimplementierung der STL, die außerhalb des Caches stark an Leistung verliert. Die absolute Beschleunigung bleibt also trotzdem unter der Anzahl Threads. Die Bandbreite des Systems, die wir unabhängig mit dem etablierten Stream Benchmark messen, limitiert `find` (siehe Abbildung 3). Der naive Algorithmus ist wie erwartet schlecht, wachsende Blockgrößen bringen tatsächlich einen Vorteil.

Die softwaretechnischen Aspekte der Integration der parallelisierten Routinen in eine existierende Bibliothek sind ebenfalls wichtig. Ziel ist zunächst eine möglichst einfache Verwendbarkeit. Der Nutzer hat die Möglichkeit, die parallele Varianten global einzuschalten. Damit kann sich durch reines Neukompileren für STL verwendende Programme eine Be-

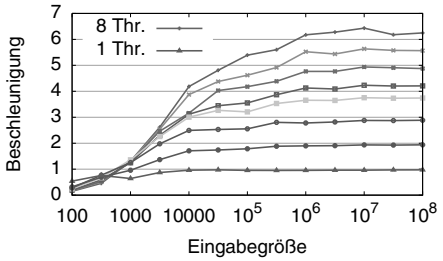

Abbildung 2: Beschleunigung von `sort`.

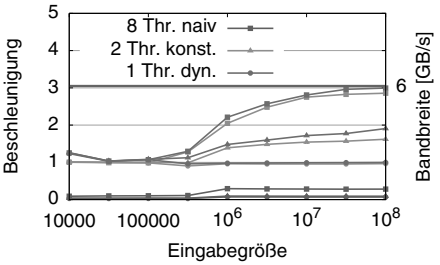

Abbildung 3: Beschleunigung von `find`. mit eingezeichneter Bandbreiten-Schranke.

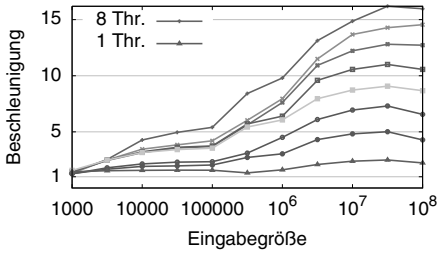

Abbildung 4: Beschleunigung von `random_shuffle`.

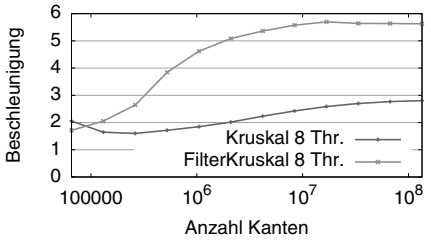

Abbildung 5: Beschleunigung der Berechnung des minimalen Spannbaums.

schleunigung ergeben. Eine selektive Aktivierung der parallelen Variante pro Aufruf ist aber ebenso möglich.

Der Benutzer kann/muss die Algorithmen mit ausführbarem Code parametrisieren, der zum Teil sogar in überladenen Operatoren und Konstruktoren versteckt ist. Hier können Seiteneffekte die semantische Kompatibilität mit der Bibliotheksschnittstelle gefährden. Ein weiteres generelles Problem bei paralleler Ausführung sind Ausnahmen, die potentiell vom Benutzer-Code ausgelöst werden.

Inzwischen ist die MCSTL in die Standardbibliothek des GNU C++-Übersetzers eingegangen und bildet dort den so genannten *libstdc++ parallel mode*. Dies vereinfacht die Verwendung weiter, zwei Kommandozeilen-Optionen aktivieren die parallelisierten Implementierungen des ansonsten unveränderten Programms:

```
#include <algorithm>
std::vector<int> a(10000000);
int main()
{ std::sort(a.begin(), a.end()); }

g++-4.5 -D_GLIBCXX_PARALLEL -fopenmp sort.cpp
```

1: **function** FilterKruskal(E, T : Sequence of Edge, P : UnionFind)
2: **if** $m \leq \text{kruskalThreshold}(n, |E|, |T|)$ **then**
3: Kruskal(E, T, P)
4: **else**
5: pick a pivot $p \in P$
6: $E_{\leq} := \langle e \in E : e \leq p \rangle; E_{>} := \langle e \in E : e > p \rangle$
7: FilterKruskal$(E_{\leq}, T, P)$
8: $E_{>} := \text{filter}(E_{>}, P)$
9: FilterKruskal$(E_{<}, T, P)$
10: **end if**

Abbildung 6: Der Filter-Kruskal-Algorithmus.

2.1 Anwendungen

Eine Variante des Kruskal-Algorithmus [OSS09] (siehe Abbildung 6) berechnet den minimalen Spannbaum. Die Kanten werden aber nicht wie üblich zu Beginn komplett nach Gewicht sortiert, sondern nur nach Bedarf partitioniert, und Kanten, die sicher nicht mehr benötigt werden, früh ausgefiltert. Das Partitionieren (`partition`, Zeile 6), dass filtern (`remove_copy_if`, Zeile 8) und das Sortieren im Basisfall (Zeile 2) lassen sich mit Hilfe der MCSTL einfach parallelisieren. Sie wird dadurch so viel schneller (siehe Abbildung 5), dass sie andere Algorithmen, die schlecht parallelisierbar sind, hinter sich lässt.

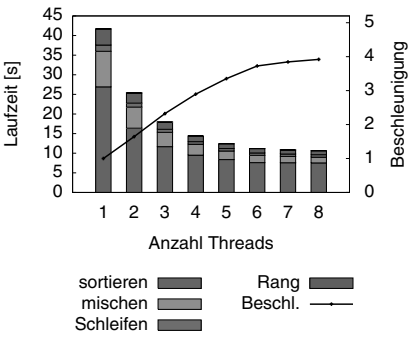

Abbildung 7: Beschleunigung der Suffix-Array-Konstruktion.

Suffix Arrays sind eine Indexdatenstruktur für Volltextsuche. Der DC3-Algorithmus zu deren Konstruktion läuft in Linearzeit und lässt sich gut parallelisieren, aus der MCSTL werden sortieren, Präfixsummen und mischen verwendet. Dadurch wird eine existierende Implementierung des DC3-Algorithmus um den Faktor 5 beschleunigt. Abbildung 7 zeigt weiterhin auf, dass alle Schritte des Algorithmus beschleunigt werden.

3 Geometrische Algorithmen im Hauptspeicher

Im Gebiet Algorithmische Geometrie wählen wir als Ansatzpunkt die dort wichtigste Algorithmenbibliothek CGAL (Computational Geometry Algorithms Library). Aus der Fülle

der dort implementierten Algorithmen wählen wir drei aus, die sich durch hohen Rechenaufwand bzw. häufige Verwendung auszeichnen [BMPS10].

Der Schnitt achsenparalleler (Hyper-)Quader wird oft als Heuristik in der Kollisionserkennung verwendet. Nur falls sich die einhüllenden achsenparallelen Quader zweier Objekte schneiden, muss detailliert untersucht werden, ob sich die exakten Objekte tatsächlich schneiden. Als Teile-und-Herrsche-Algorithmus ist eine Parallelisierung mittels des OpenMP-Task-Konstrukts sowie des MCSTL-Partitionierers relativ einfach möglich. Die Tücke lauert jedoch im Detail: Zwei parallelen rekursiven Aufrufen müssen zwei Teilsequenzen übergeben werden, die sich erstens überlappen und zweitens von den rekursiven Aufrufen permutiert werden. Kopiert man nur den überlappenden Teil, ergibt sich aber das Problem, dass die Teilsequenzen mit der Tiefe der Rekursion immer zerstückelter werden, was wahlfreien Zugriff immer langsamer macht. Letztlich lässt sich jedoch beweisen, dass zwei Teilstücke immer ausreichen, wenn man je nach Pivotposition geschickt dupliziert.

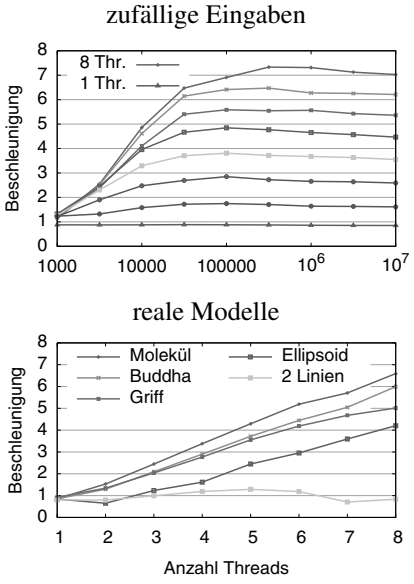

Abbildung 8: Beschleunigung der 3D-Delaunay-Triangulierung.

Für die 3D-Delaunay-Triangulierung nutzt der sequentielle CGAL-Algorithmus randomisierte inkrementelle Konstruktion. Die Eingabepunkte werden als Vorverarbeitung entlang der raumfüllenden Hilbert-Kurve sortiert. Diese wichtige Subroutine wurde im Übrigen auch parallelisiert. Da dieser Algorithmus einen sehr hohen Rechenaufwand hat, ist hier konkurrierender Zugriff der Threads auf die Datenstruktur praktikabel, feingranular geschützt durch eine sorgfältig unter mehreren Alternativen ausgewählte Locking-Strategie. Hinzu kommt eine ausgefeilte Speicherverwaltung. Algorithmen, die wenig Kommunikation garantieren, sind sehr kompliziert und lassen hohe konstante Faktoren befürchten. Wie in Abbildung 8 gezeigt, lassen sich fast perfekte Beschleunigungen erzielen, auch für reale Modelle. Damit sind wir schneller als konkurrierende Ansätze [KKv05], bei denen zudem die numerische Robustheit unklar ist, welche CGAL gewährleistet.

4 Basis-Algorithmen im Externspeicher

Bei Datenmengen größer als der Hauptspeicher greift man auf Externspeicher zurück, d. h. üblicherweise Festplatten. Auf diesen externen Daten arbeiten RAM-Algorithmen typischerweise nicht mehr effizient, da wahlfreier Zugriff einzelner Elemente auf Grund der hohen Latenzzeiten sehr teuer ist. Das etablierte Externspeicher-Modell mit paralle-

len Festplatten [VS94] zählt daher blockweise Zugriffe, die in der Praxis jeweils in der Größenordnung von Megabytes liegen.

Die Standard Template Library for XXL Data Sets (STXXL) [DKS08] implementiert Algorithmen und Datenstrukturen, die in diesem Modell effizient arbeiten, d. h. möglichst wenige Blockzugriffe tätigen. Dies funktioniert in der Praxis so gut, dass beim Einsatz mehrerer Festplatten jetzt überraschenderweise der Prozessor den Engpass bildet.

Wir setzen MCSTL-Routinen ein, um die interne Rechenarbeit zu beschleunigen. Sortieren beispielsweise, als Basisoperation gerade für Externspeicheralgorithmen noch wichtiger, wird durch diese Maßnahme bis an die Ein-/Ausgabe-Grenze beschleunigt (siehe Abbildung 9).

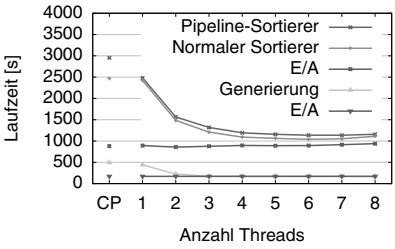

Abbildung 9: Laufzeit-Entwicklung für externes Sortieren von 100 GiB an Daten.

Dieser Datenparallelismus ist aber nicht der einzige Ansatzpunkt. Um temporäres Schreiben/Lesen bei der Weitergabe zwischen algorithmischen Komponenten zu vermeiden, unterstützt die STXXL Pipelining zwischen den Komponenten. Der sich aus diesen Verbindungen ergebende Datenflussgraph bietet Potential für weitere Parallelisierung. Durch Einfügen asynchroner Pufferkomponenten können Teilpfade unabhängig voneinander und damit parallel ausgeführt werden, es ergibt sich zusätzlich Task-Parallelismus. Während die sequentielle Ausführung des Datenflussgraphen (Abbildung 12) 11 124 s benötigt, sind es mit 3 Threads daten-parallel 6 098 s, mit Task-Parallelismus 4 629 s, und mit beiden zusammen nur noch 3 057 s. Dabei stellen die Zahnräder die asynchronen Pufferkomponenten dar. Möglichkeit für Task-Parallelismus besteht auf den Teilströmen 4–7 und 10–13.

Beide Ansätze kombinieren also ihr Beschleunigungspotential [BDS09]. Konkretes Anwendungsbeispiel ist hier wiederum die Konstruktion eines Suffix Array, dieses Mal für noch größere Eingaben.

5 Verteiltes Externes Sortieren

Bei extrem großen Datenmengen reichen auch die Festplatten eines einzelnen Rechners nicht mehr aus, es werden daher viele Rechner mittels eines Netzwerks verknüpft. Wir entwerfen Algorithmen zum Sortieren sehr großer Datenmengen auf solchen Rechnerbündeln mit verteiltem Speicher, und mehreren Festplatten pro Rechenknoten [RSS10]. Dies dient einerseits als Startpunkt für eine Verallgemeinerung der bisherigen Arbeit auf Systeme mit verteiltem Speicher, andererseits als ein weiteres Anwendungsbeispiel von MCSTL und STXXL, die in hierarchischer Weise auf den einzelnen Knoten parallel arbeiten.

Ein eher theoretischer Algorithmus verteilt die Daten randomisiert über alle Rechenknoten, und kommt dann für einen großen Eingabegrößenbereich mit 4 Platten-Transfers pro Block Elemente (zweimal lesen, zweimal schreiben) aus, was optimal ist. Allerdings

1. *globale* Teilsequenzen (Größe M) sortieren — $1\times$Komm.$+2\times$ E/A
   - Auswahl *(blockweise randomisiert)*
   - internes Sortieren *selbstähnlich*
2. *exakte* Verteilung über Knoten bestimmen
   - mittels Selektion über mehrere Sequenzen — $o(\mathrm{sort}(N))$
3. Daten *umverteilen* — 0..1 (ϵ)$\times$($1\times$Komm.$+2\times$ E/A)
   - wahrscheinlich nur wenig Daten am falschen Ort — ϵN
4. Teilsequenzen *lokal mehrwege-mischen* — $2\times$ E/A

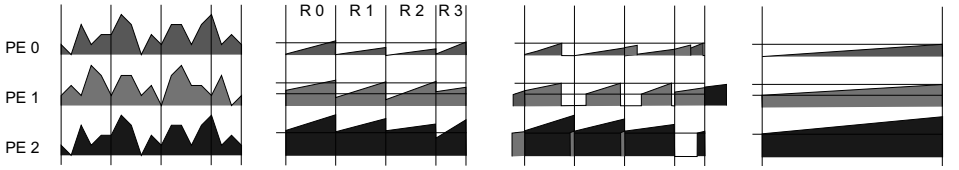

Abbildung 10: Schritte und Gesamtschema des CANONICALMERGESORT-Algorithmus.

müssen die Daten auch vier- bis fünfmal kommuniziert werden, was bei einem relativ zu den Platten langsamen Netzwerk zu teuer käme.

Der in Abbildung 10 beschriebene praktischere Algorithmus verteilt das Resultat kanonisch: Die kleinsten Elemente liegen auf dem ersten Knoten, die größten auf dem letzten. Im besten Fall benötigt er nur noch 4 Platten-Transfers und eine Kommunikation. Diese Werte können ansteigen, wenn sich die Teilsequenzen aus Schritt 1 sehr unähnlich sind, und deshalb in Schritt 3 substantiell viele Daten umverteilt werden müssen. Diese Datenmenge ist erwartet jedoch klein, wie wir ausrechnen können.

Bisherige Algorithmen (z. B. [ADADC+97]) benötigen deutlich mehr Platten-Transfers, oder haben keine Laufzeitgarantien, und können z. B. für eine schlechte Eingabe zu einem sequentiellen Algorithmus degenerieren.

Die Implementierung der praktischeren Variante wird wiederum ausführlich experimentell evaluiert und analysiert. Die schwache Skalierung für zufällige Eingabedaten ist sehr gut (Abbildung 11). Jedoch gibt es einen schlechten Fall, der im dritten Schritt eine Übertragung fast aller Daten nötig macht, und damit die Laufzeit um ca 50% erhöht. Indem jedoch die Teilsequenzen im ersten Schritt blockweise randomisiert ausgewählt werden, erreicht der Algorithmus wieder fast die volle Leistung.

Zum Vergleich mit Konkurrenz-Implementierungen haben wir Ergebnisse beim renommierten internationalen Sort Benchmark eingereicht, und erzielten 2009 zwei Leistungs-Weltrekorde. Für 100 Terabyte erreichte Konkurrent Yahoo eine um 2% bessere Zeit, benötigte aber die 17fache Anzahl Knoten und Festplatten. Für 1 Terabyte konnten wir

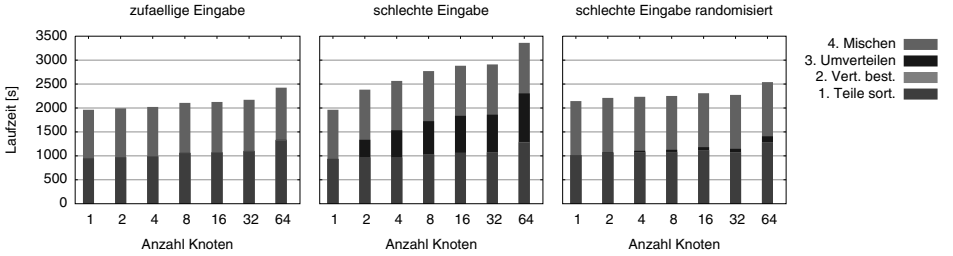

Abbildung 11: Schwache Skalierung von CANONICALMERGESORT, 100 GiB pro Knoten.

Yahoo um Faktor 2 schlagen, trotz deutlich geringerem Hardware-Einsatz.

Weitere Arbeiten [BMSS10] auf speziell ausgesuchter Hardware zeigen, dass sich mit Hilfe dieses Algorithmus auch sehr energiesparend sortieren lässt.

6 Fazit

Wir haben in dieser Arbeit demonstriert, dass sich Anwendungen mittels parallelisierter Algorithmenbibliotheken auf Mehrkern-Rechnern deutlich beschleunigen lassen. Alle erwähnten Bibliotheken hatten in ihren sequentiellen Versionen bereits eine große Verbreitung gefunden, die Anwendung ist sehr einfach. Auch dass die vorgestellten Bibliotheken sich teilweise bereits gegenseitig verwenden, zeigt deren Signifikanz. Mit Fallstudien demonstrieren wir die Verwendbarkeit der MCSTL-Routinen als Unterprogramme in komplexeren algorithmischen Anwendungen.

Insgesamt wurde der gesamte Weg vom theoretischen Algorithmus zur Implementierung, von der Benutzerfreundlichkeit zu softwaretechnischen Aspekten, betrachtet. Dies ist konsistent mit dem im *Algorithm Engineering* betrachteten Zyklus aus Theorie, Implementierung, Experiment und Rückkopplung, bei dem ja auch Bibliotheken eine wichtig Rolle spielen.

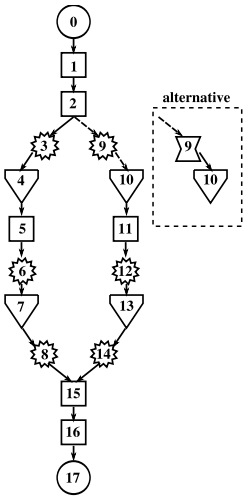

Abbildung 12: Datenflussgraph.

Literatur

[ADADC+97] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David P. Patterson. High-Performance Sorting on Networks of Workstations. In *ACM SIGMOD Conference*, pages 243–254, 1997.

[AJR+01] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. M. Amato,

and L. Rauchwerger. STAPL: An Adaptive, Generic Parallel C++ Library. In *LCPC*, pages 193–208, 2001.

[BDS09] Andreas Beckmann, Roman Dementiev, and Johannes Singler. Building A Parallel Pipelined External Memory Algorithm Library. In *23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2009.

[BMPS10] Vicente H. F. Batista, David L. Millman, Sylvain Pion, and Johannes Singler. Parallel Geometric Algorithms for Multi-Core Computers. *Computational Geometry – Theory and Applications*, 2010. http://dx.doi.org/10.1016/j.comgeo.2010.04.008.

[BMSS10] Andreas Beckmann, Ulrich Meyer, Peter Sanders, and Johannes Singler. Energy-Efficient Sorting using Solid State Disks. In *International Green Computing Conference*, 2010.

[DKS08] Roman Dementiev, Lutz Kettner, and Peter Sanders. STXXL: standard template library for XXL data sets. *Software: Practice and Experience*, 38(6):589–637, 2008.

[KKv05] Josef Kohout, Ivana Kolingerová, and Jiří Žára. Parallel Delaunay triangulation in E^2 and E^3 for computers with shared memory. *Parallel Computing*, 31(5):491–522, 2005.

[OSS09] Vitaly Osipov, Peter Sanders, and Johannes Singler. The Filter-Kruskal Minimum Spanning Tree Algorithm. In *Workshop on Algorithm Engineering & Experiments (ALENEX)*, pages 52–61, 2009.

[PSS07] Felix Putze, Peter Sanders, and Johannes Singler. The Multi-Core Standard Template Library (poster and extended abstract). In *12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 144–145, 2007.

[RSS10] Mirko Rahn, Peter Sanders, and Johannes Singler. Scalable Distributed-Memory External Sorting. In *26th IEEE International Conference on Data Engineering (ICDE)*, pages 685–688, 2010.

[Sin10] Johannes Singler. *Algorithm Libraries for Multi-Core Processors*. PhD thesis, Department of Informatics, Karlsruhe Institute of Technology, 2010.

[SSP07] Johannes Singler, Peter Sanders, and Felix Putze. The Multi-Core Standard Template Library. In *Euro-Par 2007: Parallel Processing*, volume 4641 of *LNCS*, pages 682–694. Springer-Verlag, 2007.

[VS94] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I/II. *Algorithmica*, 12(2/3):110–169, 1994.

Johannes Singler wurde 1980 in Offenburg geboren. Nach dem Abitur 1999 in Oberkirch leistete er zunächst seinen Zivildienst an der Schule für Körperbehinderte in Offenburg ab. Im September 1999 wurde er beim 17. Bundeswettbewerb Informatik zum Bundessieger ausgezeichnet. Seitdem ist er Mitglied der GI. Ab 2000 studierte er mit einem Stipendium der Studienstiftung des Deutschen Volkes Informatik an der Universtität Karlsruhe und der Carleton University, Ottawa, Kanada, und schloss 2005 als Diplom-Informatiker ab. Seit 2006 arbeitete er als wissenschaftlicher Mitarbeiter am Lehrstuhl Prof. Sanders am jetzigen Karlsruher Institut für Technologie (KIT). Die Promotion „summa cum laude“ zu vorliegendem Thema erfolgte im Juli 2010.