Informatik Spektrum manuscript No.
(will be inserted by the editor)

Algorithmenbibliotheken

Peter Sanders

Received: date / Accepted: date

Zusammenfassung Schlüsselwörter Algorithmen · Datenstrukturen · Graphpartitionierung · Multicore · Softwarebibliotheken · Speicherhierarchien

Wir berichten über Erfahrungen mit Algorithmenbibliotheken im Rahmen des SPP 1307 Algorithm Engineering. Den Schwerpunkt bilden unsere Anpassungen der C++ STL für Sekundärspeicher und Multicore. Weitere Beispiele sind parallele Algorithmen für die Computational Geometry Algorithms Library CGAL sowie ein leistungsfähiger Graphpartitionierer. Als Anwendungsbeispiele diskutieren wir minimale Spannbäume und die Konstruktion von Suffix-Tabellen zur Volltextsuche.

Summary. We report on experiences with algorithm libraries in the context of the DFG priority program 1307 on algorithm engineering. Our focus is on adaptations of the C++ STL for secondary memory and multicore. Further examples are parallel algorithms for the Computational Geometry Algorithms Library CGAL and a powerful graph partitioner. As application examples we discuss minimum spanning trees and the construction of suffix arrays for full text search.

1 Einleitung

Implementierungen, die den Entwicklungszyklus des Algorithm Engineering vorantreiben, müssen bei begrenztem Budget schnell fertig werden (siehe auch

Teilweise unterstützt durch DFG SPP 1307 Algorithm Engineering SA 933/3-3

Karlsruher Institut für Technologie
Tel.: +49-721-608-47580
Fax: +49-721-608-43088
E-Mail: sanders@kit.edu

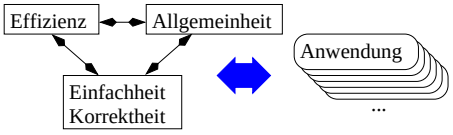

Abb. 1 Effizienz, Allgemeinheit und Einfachheit/Korrektheit von Algorithmenbibliotheken stehen in einem Spannungsverhältnis zueinander. Außerdem ist die unbegrenzte Zahl potenzieller Anwendungen eine Herausforderung.

das Schlagwort „Algorithm Engineering" in diesem Heft). Die so entstandenen Prototypen sind im Allgemeinen nicht direkt für Anwendungen einsetzbar. Selbst die Weiterentwicklung stößt bei fortgesetzter „Hackerei" an ihre Grenzen – spätestens wenn eine Master- oder Doktorarbeit von einer anderen Person weitergeführt werden soll. Deshalb sollte ein Algorithmeningenieur für gut befundene Algorithmen in Algorithmenbibliotheken konsolidieren, ggf. durch Neuimplementierung. Die Bibliotheken sollten effizient, leicht benutzbar, portabel und gut dokumentiert sein. Softwareengineering für Algorithmenbibliotheken ist eine besondere Herausforderung, weil es oft einen natürlichen Konflikt zwischen einfacher Benutzbarkeit, Allgemeinheit und Effizienz der Implementierungen gibt. Dazu kommt, dass zur Erstellungszeit der Bibliothek gar nicht alle Anwendungen bekannt sind. Besonders hoch sind auch die Anforderungen an die Zuverlässigkeit, weil Fehler in Bibliotheken durch den Benutzer schwer zu beheben sind und schnell zu einem Vertrauensverlust führen können. Abb. 1 fasst diese Herausforderungen zusammen.

Diesem hohen Aufwand gegenüber steht das große Potenzial von Algorithmenbibliotheken – eine Implementierung kann in vielen Anwendungen nützlich sein. Anwendungsprogrammierer können so auf einfache Weise das Know-How von Experten nutzen. Innerhalb des Algorithm Engineering erleichtert die Wiederverwendung von Code die Implementierung komplexer Algorithmen und die Kooperation mehrerer Arbeitsgruppen.

Im folgenden geben wir Beispiele für Algorithmenbibliotheken und verweisen dabei auf detailliertere Literatur und weitere Abschnitte mit Beispielen aus unserer Arbeit.

Algorithmenbibliotheken haben eine lange Tradition bei numerischer Software. Gründe für diese Erfolgsgeschichte kann man in den klar definierten Schnittstellen und der breiten Anwendbarkeit numerischer Basisroutinen suchen. Die Standardbibliotheken von Programmiersprachen enthalten schon seit Langem ausgewählte nichtnumerische Routinen. Zum Beispiel gibt es in C Funktionen für Quicksort und binäre Suche. Neuere Programmiersprachen definieren zunehmend umfangreiche Standardbibliotheken, die auch algorithmische Komponenten enthalten. Zum Beispiel definiert die C++ Standardbibliothek STL vielfältige Containerklassen (Stapel, Warteschlangen, Hashtabellen, Prioritätslisten, Maps,...). Diese Funktionalität ist gut dokumentiert und breit einsetzbar und deshalb ein attraktiver Startpunkt für verbesserte oder erweiterte Implementierungen. In unserer Arbeitsgruppe haben wir uns deshalb

intensiv mit der STL beschäftigt, siehe Abschnitte 2 und 3. Die Boost Bibliotheken (`http://www.boost.org`) bringen nochmal zusätzliche Funktionalität als eine Art „erweiterte C++ Standardbibliothek“ einschließlich Graphenalgorithmen, String-Matching, Numerik und Ansätze einer Geometriebibliothek.

Selbst solche Erweiterungen gehen aus Sicht eines Algorithmikers kaum über die grundlegendsten Ansätze hinaus. Anders sieht es bei Bibliotheken aus, die in Algorithmikarbeitsgruppen entstanden sind und entscheidend zur Entwicklung des Algorithm Engineering beigetragen haben. An erster Stelle ist hier LEDA (Library of Efficient Data types and Algorithms) zu nennen [9], die seit 1988 am MPI Informatik entwickelt wurde und jetzt durch die Firma Algorithmic Solutions weiterentwickelt wird. LEDA implementiert eine Vielzahl fortgeschrittener Algorithmen (Graphen, Geometrie, Datenstrukturen,...) mit einem leicht zu benutzenden Interface und so dass die Algorithmen gut lesbar bleiben. Im Spezialgebiet algorithmische Geometrie geht CGAL (Computational Geometry Algorithms Library, `www.cgal.org`) noch weiter. Ursprünglich im Rahmen eines EU-Projekts entwickelt, implementiert CGAL ein breites Spektrum an Funktionen. Zum Beispiel hat das Handbuch gegenwärtig 81 Kapitel. Durch intensiven Einsatz von Templates wird hier ein teilweise guter Kompromiss aus Allgemeinheit und Effizienz erreicht, der aber fast zwangsläufig auf Kosten von einfacher Benutzbarkeit und Lesbarkeit geht. Besonderer Wert wird auf Algorithmen mit exakten Ergebnissen gelegt. Aus Platzgründen müssen wir hier auf weitere Beispiele erfolgreicher Algorithmenbibliotheken verzichten (Graphen zeichnen, Bioinformatik,...). Aus dem gleichen Grund konzentrieren wir uns auf C++-Bibliotheken, obwohl es natürlich auch wichtige Bibliotheken in Java oder anderen Programmiersprachen gibt.

Die meisten Softwarebibliotheken haben einen mehr oder weniger großen Teil von Funktionen hinter denen sich nichttriviale Algorithmen verbergen. Ein Beispiel beschreiben wir in Abschnitt 6. Ein Artikel über Algorithmenbibliotheken wurde auch im Rahmen eines GI-Dagstuhl Seminars über Algorithm Engineering geschrieben [7].

2 Sekundärspeicher – STXXL

Die Standard Template Library for Extra Large Data Sets (`http://stxxl.sourceforge.net/`) [5,3] implementiert eine Großteil der Funktionalität der STL für Sekundärspeicher, also für den Fall, dass so große Datenmengen zu verarbeiten sind, dass sie nicht in den Hauptspeicher sondern nur auf Festplatte oder SSD passen (siehe auch Abb. 2). Trotz des aus algorithmischer Sicht geringen Funktionsumfangs der STL ist das erstaunlich nützlich, da genau die Funktionen abgedeckt sind, die besonders häufig für Sekundärspeicherberechnungen benötigt werden. Neben einfachen Collection-Datentypen wie Stapel, Schlangen und Vektoren (mit Caching) sind dies vor allem Sortieralgorithmen, Prioritätslisten und Suchbäume (B-Bäume). Sortieren ist für den Sekundärspeicher noch grundlegender als in „normalen“ Algorithmen, weil es Objekte

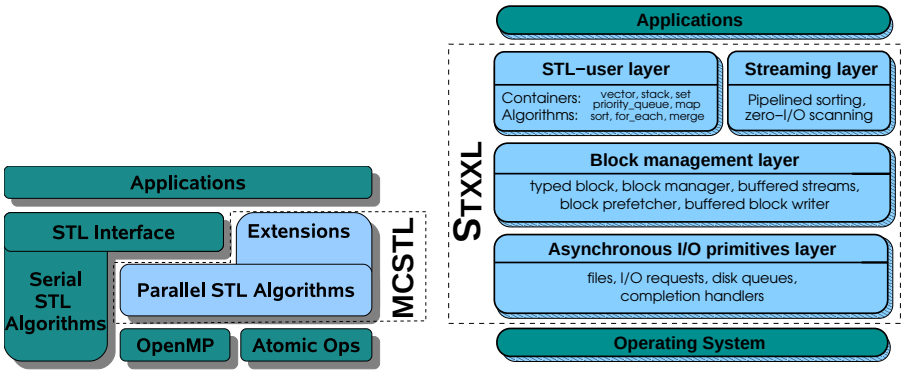

Abb. 2 Schichtenstruktur unserer STL-Bibliotheken.

zusammen bringt, die gemeinsam verarbeitet werden sollen und damit zu verbesserter Lokalität führt. Prioritätslisten unterstützen das wichtige Konzept des Time-Forward-Processing – eine Verallgemeinerung des „Zusammenbringprinzips“ von Sortieren – bei dem Objekte noch während der Berechnung erzeugt werden können. Suchbäume schließlich werden schon seit langem für Indexdatenstrukturen, z.B. in Datenbanken eingesetzt.

Mit der oben beschriebenen Funktionalität lassen sich viele Sekundärspeicheralgorithmen elegant implementieren. Jedoch ist uns aufgefallen, dass wir immer wieder einen Faktor ≥ 2 mehr Plattenzugriffe benötigen als eine optimierte Handimplementierung, weil gerade geschriebene Zwischenergebnisse gleich wieder gelesen werden müssen. Deshalb haben wir zusätzliche Funktionalität in Form von *Pipelining* eingeführt, die sich nahtlos in das Iteratorkonzept der STL einfügt. Eine Pipeline besteht aus Eingabeströmen von Tupeln, die durch eine Funktion weiterverarbeitet werden und in einen oder mehrere Ausgabeströme geleitet werden. Ein wichtiger Spezialfall sind Sortierer, deren Eingabe aus einer Pipeline kommen kann und die ihre Ausgabe in eine Pipeline weiterleiten. Ein einfaches Beispiel ist eine Funktion, die eine große Menge von Objekten erzeugt (z.B. Lösungen von Rubik's Cube) und die Aufgabe besteht darin die Anzahl der Objekte zu zählen ohne Duplikate mitzuzählen. Ein Pipeline-Sortierer empfängt die unsortierten Objekte, sortiert sie und leitet sie an eine Ausgabe-Pipeline weiter. Diese entfernt (nun beieinanderstehende) Duplikate und leitet das Ergebnis an einen Zähler weiter. Die einzigen hier auftretenden Plattenzugriffe passieren innerhalb des Sortierers. Dagegen würde eine Implementierung ohne Pipelining die Objekte zunächst auf Platte schreiben, dann den externen Sortierer aufrufen und schließlich das Sortierergebnis nochmal von der Platte lesen. Weitere Beispiele finden sich in den Abschnitten 7.1 und 7.2 sowie in Abb. 5.

3 Multicore – MCSTL

Das Programmiermodell unserer Multicore-Implementierung der STL (MCSTL, siehe auch Abb. 2) sieht einen einzelnen Benutzer-Aktivitätsfaden vor, dessen Aufrufe von STL Funktionen parallel ausgeführt werden [17,15]. Wenn der Benutzer bereit ist auch einfache Schleifen mit Hilfe von STL-Algorithmen zu implementieren, ergibt sich dadurch eine automatische Parallelisierung. Häufiger wird ein Aufruf paralleler STL-Algorithmen als Teil einer manuell parallelisierten Anwendung sein. Ein Beispiel findet sich in Abschnitt 7.1. Die MCSTL parallelisiert alle STL-Methoden, bei denen eine Parallelisierung sinnvoll erscheint: Sortieren und verwandte Funktionen wie Mischen, Partitionieren und Auswählen (Mediane und Quantile); Zufallspermutationen; Präfixsummen etc. Auch scheinbar triviale Algorithmen wie for_each liefern hier einen Mehrwert, weil sie dank eines work-stealing Lastverteilers auch unregelmäßig verteilte Arbeit gut über die Prozessoren verteilen. Einzelne Zugriffe auf Datenstrukturen brauchen i. Allg. zu wenig Zeit für eine sinnvolle Parallelisierung, so dass nur Bulk-Operationen implementiert sind [8]. Bei der Implementierung wurde erheblicher softwaretechnischer Aufwand getrieben um wirklich volle Kompatibilität zur sequenziellen STL zu erreichen. Dadurch war es möglich, die MCSTL als Teil des GNU C++ Systems zu platzieren [16]. Interessante Beispiele für Wiederverwendung gibt es bereits innerhalb der MCSTL. Zum Beispiel ergibt sich automatisch eine effiziente parallele Selektion sobald die Partitionierungsoperation parallelisiert ist, die ja ohnehin für Quicksort benötigt wird.

4 Multicore-Algorithmen für algorithmische Geometrie – CGAL

Für eine erste Fallstudie zur Parallelisierung von CGAL haben wir uns drei häufig benutzte und zeitintensive Funktionen ausgesucht [1]: Den Aufbau von KD-Bäumen – einer wichtigen Datenstruktur zur Verwaltung von Punktmengen. Die Schnittberechnung von achsenparallelen d-dimensionalen Rechtecken, die zum Beispiel für die Kollisionserkennung zwischen sich bewegenden Objekten genutzt wird. 3D Delaunay-Triangulationen, die z.B. für das wissenschaftliche Rechnen und für Oberflächenrekonstruktionen wichtig sind. Obwohl das recht gut funktioniert hat bleibt die Aufgabe, eine so große Bibliothek zu parallelisieren, natürlich schwierig. Allerdings ist zu hoffen, dass es genügt, eine kritische Masse an von Experten programmierter Basisfunktionalität zur Verfügung zu stellen, auf die Andere dann aufbauen können ohne sich mit hardwarenahen Details der Parallelisierung beschäftigen zu müssen. Schon in [1] haben sich mehrere solche Basisfunktionen herausgestellt: Sortieren und Partitionieren aus der MCSTL, KD-Bäume und eine kompakte thread-safe Container-Datenstruktur.

5 Graphpartitionierung – KaFFPaE

Bei der (balancierten) Graphpartitionierung geht es darum, einen Graphen in eine vorgegebene Anzahl „ungefähr“ gleich großer Teile zu zerlegen und dabei möglichst wenig Kanten zu entfernen. Graphpartitionierung hat viele Anwendungen zum Beispiel bei der Parallelisierung von Simulationen auf unregelmäßig verfeinerten Gittern (finite Elemente,...) und bei der Routenplanung. Obwohl das Problem NP-hart ist, gibt es sogenannte Multilevelverfahren, die das Problem für viele praktische Instanzen schnell und relativ gut lösen. Da es sich aber um komplexe Algorithmen mit einer 5-stelligen Anzahl Codezeilen handelt, ist Wiederverwendung in Form von Algorithmenbibliotheken hier der Standard. Im Extremfall enthält eine bereits nützliche Graphpartitionierungsbibliothek nur eine einzige Funktion, so dass man diskutieren kann ob es sich wirklich um eine Bibliothek handelt. Andererseits ist ein gesonderter Begriff für diesen Spezialfall wenig sinnvoll. Wir entwickeln gerade eine Graphpartitionierungsbibliothek, die noch im Prototypenstadium ist aber in Benchmarks sehr gut abschneidet (z.B. Abb. 3). Insbesondere berechnet sie in vielen Fällen die besten bekannten Partitionen [10].

6 Message Passing Interface

MPI ist eine Softwarebibliothek zum effizienten Nachrichtenaustausch zwischen Computern und ein Quasistandard beim Hochleistungsrechnen. MPI dient in erster Linie der Abstraktion von Hardwaredetails. Dieser Aspekt hat wenig mit Algorithmen zu tun. Allerdings gibt es eine reichhaltige Auswahl an Operationen für kollektive Kommunikation, deren effiziente Implementierung algorithmisch sehr interessant ist: Broadcast (siehe auch Abb. 4), Reduktion

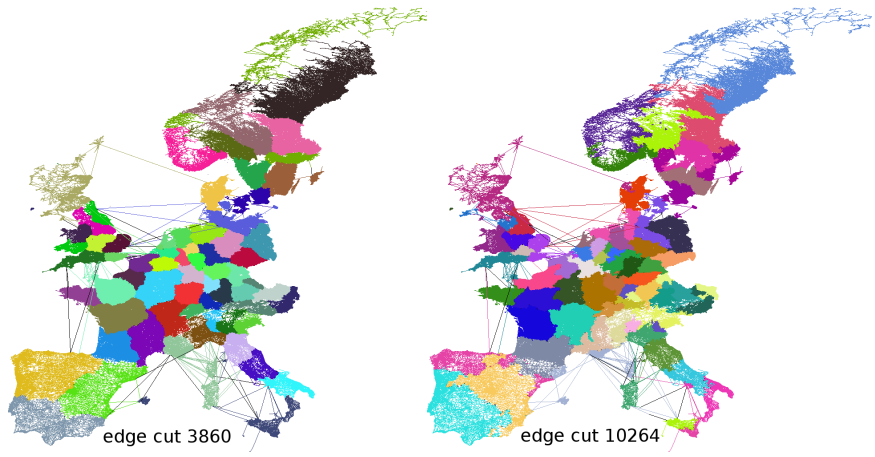

Abb. 3 Unterschiedliche Qualität von Multilevel-Graphpartitionierern.

(berechne $\bigotimes_{0\le i<p} x_i$ für eine assoziative Operation $\otimes$ und Werte x_i auf p Prozessoren), Präfixsumme (berechne $\bigotimes_{0\le i<j} x_i$ für alle $j < p$.), einsammeln, verteilen, an alle einsammeln, jeder an jeden,.... Diese Operationen sind die Grundlage vieler skalierbarer paralleler Algorithmen und somit Teil der Basic Toolbox der Parallelverarbeitung. In Kooperation mit NEC haben wir verbesserte Algorithmen für viele dieser Funktionen entwickelt [13,14,12].

7 Anwendungen

7.1 Minimale Spannbäume

Beim MST-Problem geht es darum, in einem Graphen mit positiven Kantengewichten eine Teilmenge von Kanten auszuwählen, die alle Knoten miteinander verbinden und minimales Gesamtgewicht haben. MSTs sind ein fundamentales Netzwerkentwurfsproblem und werden auch als Clusteringverfahren eingesetzt. Entsprechend gehören MST Algorithmen zur Informatikgrundausbildung. MSTs sind auch ein sehr schönes Beispiel für den erfolgreichen Einsatz unserer STL Implementierungen.

In [6] beschreiben wir die erste Implementierung eines nichttrivialen Graphenalgorithmus überhaupt, der Graphen mit vielen Milliarden Kanten verarbeitet. Eine bereits recht brauchbare Basisimplementierung ist mit Hilfe der STXXL sehr einfach. Zum Beispiel umfasst das komplizierteste Unterprogramm dann nur 17 Zeilen Code [3].

In [11] entwickeln wir eine Verbesserung von Kruskal's Algorithmus, die „im Mittel" weniger Arbeit leistet und außerdem parallelisierbar ist. Diese Parallelisierung funktioniert vollständig mit MCSTL durch Einsatz von Sortieren, Partitionieren und der „trivialen" Funktion `remove_if`.

7.2 Suffix-Tabellen

Die sortierte Anordnung der Suffixe einer Zeichenkette (z.B. `banana`→`a`, `ana`, `anana`, `banana`, `na`, `nana`) liefert nützliche Information z.B. für Volltextsuche, Datenkompression oder Anwendungen in der Bioinformatik. In [4] entwickeln

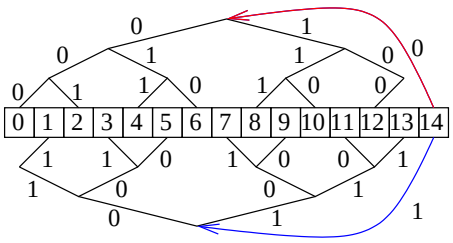

Abb. 4 Schema für pipelined Broadcast bei dem Daten auf zwei Binärbäumen gleichzeitig verteilt werden.

wir die bisher besten Konstruktionsalgorithmen für Suffix-Tabellen, die nicht in den Hauptspeicher passen. Insbesondere können wir damit deutlich größere Eingaben unterstützen als selbst die intensiv untersuchten Ansätze mit ausgefeilter Datenkompression. Alle betrachteten Algorithmen lassen sich durch eine Kombination von Sortieren und Pipelining sehr gut in STXXL ausdrücken (siehe auch Abb. 5). Die MCSTL beschleunigt die interne Suffix-Array Konstruktion. In [2] zeigen wir, dass eine Kombination von MCSTL, STXXL und einer Parallelisierung des Pipelining weitere Verbesserungen ermöglicht.

8 Ausblick

Aus unserer Sicht kommen Algorithmenbibliotheken eine Schlüsselrolle im Algorithm Engineering zu, weil sie einen Hauptmechanismus für den Transfer von algorithmischem Know-How in Anwendungen darstellen. Hier gibt es noch viel zu tun, z.B. weil der Aufwand für gute Algorithmenbibliotheken erheblich ist und weil sich softwaretechnische Herausforderungen ergeben, die jenseits der Kernkompetenz einer Algorithmenarbeitsgruppe liegen. Es muss sich außerdem ein Mechanismus finden, wie die Entwicklung aufwendiger Algorithmenbibliotheken unterstützt werden kann. Zum Beispiel sollten signifikante Beiträge zu Algorithmenbibliotheken für einen akademischen Lebenslauf eine ähnliche Rolle spielen wie gute Veröffentlichungen. Eine andere Möglichkeit ist die Open-Source-Community in die Weiterentwicklung von Algorithmenbibliotheken einzubeziehen.

Literatur

1. Batista, V.H.F., Millman, D.L., Pion, S., Singler, J.: Parallel geometric algorithms for multi-core computers. Comput. Geom. **43**(8), 663–677 (2010)
2. Beckmann, A., Dementiev, R., Singler, J.: Building a parallel pipelined external memory algorithm library. In: 23rd IEEE International Symposium on Parallel and Distributed Processing, pp. 1–10 (2009)
3. Dementiev, R.: Algorithm engineering for large data sets (2006). Doktorarbeit, Universität des Saarlandes

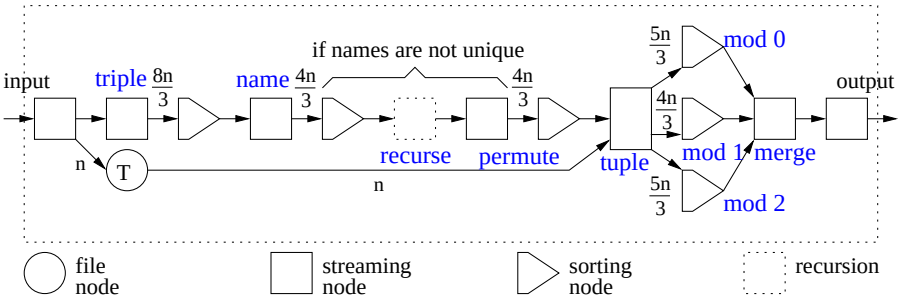

Abb. 5 Pipelining-Schema für einen I/O-optimalen externen Algorithmus zur Konstruktion von Suffix-Tabellen.

4. Dementiev, R., Kärkkäinen, J., Mehnert, J., Sanders, P.: Better external memory suffix array construction. ACM Journal of Experimental Algorithmics **12** (2008). Special issue on Alenex 2005
5. Dementiev, R., Kettner, L., Sanders, P.: STXXL: Standard Template Library for XXL data sets. Software Practice & Experience **38**(6), 589–637 (2008)
6. Dementiev, R., Sanders, P., Schultes, D., Sibeyn, J.: Engineering an external memory minimum spanning tree algorithm. In: IFIP TCS, pp. 195–208. Toulouse (2004)
7. Dementiev, R., Singler, J.: Libraries. In: M. Müller-Hannemann, S. Schirra (eds.) Algorithm Engineering, *LNCS*, vol. 5971, pp. 290–324. Springer (2010)
8. Frias, L., Singler, J.: Parallelization of bulk operations for STL dictionaries. In: Euro-Par Workshops, *LNCS*, vol. 4854, pp. 49–58. Springer (2007)
9. Mehlhorn, K., Näher, S.: The LEDA Platform of Combinatorial and Geometric Computing. Cambridge University Press (1999)
10. Osipov, V., Sanders, P., Schulz, C.: Engineering graph partitioning algorithms. In: 11th International Symposium on Experimental Algorithms (SEA), *LNCS*, vol. 7276, pp. 18–26. Springer (2012)
11. Osipov, V., Sanders, P., Singler, J.: The filter-Kruskal minimum spanning tree algorithm. In: 10th Workshop on Algorithm Engineering and Experiments (ALENEX), pp. 52–61 (2009)
12. Sanders, P., Speck, J., Träff, J.L.: Two-tree algorithms for full bandwidth broadcast, reduction and scan. Parallel Computing **35**(12), 581–594 (2009)
13. Sanders, P., Träff, J.L.: The factor algorithm for regular all-to-all communication on clusters of SMP nodes. In: 8th Euro-Par, no. 2400 in LNCS, pp. 799–803. Springer (2002)
14. Sanders, P., Träff, J.L.: Parallel prefix (scan) algorithms for MPI. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting, *LNCS*, vol. 4192, pp. 49–57. Springer (2006)
15. Singler, J.: Algorithm libraries for multi-core processors (2010). Doktorarbeit, Universität Karlsruhe
16. Singler, J., Kosnik, B.: The libstdc++ parallel mode: Software engineering considerations. In: International Workshop on Multicore Software Engineering (IWMSE) (2008)
17. Singler, J., Sanders, P., Putze, F.: MCSTL: The multi-core standard template library. In: 13th International Euro-Par Conference, *LNCS*, vol. 4641, pp. 682–694. Springer (2007)