Algorithm Libraries for Multi-Core Processors

zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften

der Fakultät für Informatik
des Karlsruher Instituts für Technologie

genehmigte
Dissertation

von
Dipl.-Inform. Johannes Singler
aus Offenburg

Karlsruhe, Juli 2010

Tag der mündlichen Prüfung:	8. Juli 2010
Referent:	Prof. Dr. Peter Sanders
Korreferent:	Prof. Dr. Ulrich Meyer

Beginning in 2005, the advent of microprocessors with multiple CPU cores on a single chip has lead to a paradigm shift in computer science. Since the clock rates stagnate, there is no automatic performance increase anymore when switching to a newer processor. Instead, in order to achieve an acceleration, a program has to be executed in parallel on the multiple cores. This fact poses an immense challenge to the development of high-performance applications. Not only are multi-core processors ubiquitous nowadays, application developers are also forced to utilize them to avoid stagnation of performance.

In this thesis, we examine an approach that facilitates the development of applications that exploit parallelism. We implement parallelized variants of established algorithm libraries, which provide the same functionality as their sequential versions but execute faster by utilizing the multiple cores. This allows developers to use parallelism implicitly, encapsulated by the unaltered interface of the algorithm.

For a start, we consider a library for basic algorithms for internal (main) memory, the C++ Standard Template Library (STL). We data-parallelize its algorithms in the shared memory model, including algorithms for specific data structures, building the Multi-Core Standard Template Library (MCSTL).

Geometric algorithms are addressed next, by parallelizing selected routines of the *Computational Geometry Algorithms Library*, facilitated by the usage of MCSTL routines.

We apply MCSTL also to speed up internal computation for basic external memory algorithms, which efficiently handle large data sets stored on disks. In combination with some task parallelism for algorithmic stages on a higher level, this enables the *Standard Template Library for XXL Data Sets* (STXXL) to exploit multi-core parallelism.

The libraries had already been widely used in their sequential versions, so applications that employ them can immediately benefit from the improved speed.

We carry out experiments on the implementations, and evaluate the performance in detail, also relating to hardware limitations like the memory bandwidth and the disk bandwidth. Case studies using MCSTL and STXXL demonstrate the benefit of the libraries to algorithmic applications.

Finally, as a starting point of a generalization to distributed memory, and as an advanced application of the MCSTL and the STXXL, we design distributed external memory sorting algorithms for very large inputs. The implementation of the more practical algorithm set two performance world records in 2009.

Deutsche Zusammenfassung

Beginnend mit dem Jahr 2005 hat das Aufkommen von Prozessoren mit mehreren Rechenkernen auf einem Chip zu einem Paradigmenwechsel in der Informatik geführt. Da gleichzeitig die Taktfrequenzen nicht weiter steigen, ergibt sich eine Leistungssteigerung nun nun nicht mehr automatisch beim Verwenden eines neuen Prozessors. Vielmehr muss durch die parallele Ausführung eines Programms auf mehreren Rechenkernen eine Beschleunigung erreicht werden. Diese Tatsache ist eine große Herausforderung für das Entwickeln leistungsfähiger Anwendungen. Mehrkern-Prozessoren sind inzwischen nicht nur allgegenwärtig, Software-Entwickler müssen deren Fähigkeiten auch ausschöpfen, um eine Stagnation der Rechenleistung zu verhindern.

In dieser Arbeit untersuchen wir einen Ansatz, um das Entwickeln von Anwendungen, die Parallelismus ausnutzen, zu erleichtern: Wir implementieren parallelisierte Varianten von Algorithmenbibliotheken. Diese bieten die gleiche Funktionalität wie ihre sequentiellen Versionen, ihre Routinen laufen jedoch beschleunigt, da sie mehrere Rechenkerne nutzen. Die Verwendung solcher Bibliotheken ermöglicht einem Software-Entwickler impliziten Parallelismus, gekapselt durch die gleichbleibende Schnittstelle des Algorithmus.

Wir konzentrieren uns auf etablierte C++-Bibliotheken, als Rahmenwerk zur Verwaltung von Threads wählen wir OpenMP.

Zunächst betrachten wir eine Bibliothek für grundlegende Algorithmen im Hauptspeicher, die Standard Template Library (STL), welche Teil der C++-Standardbibliothek ist. Ihre Algorithmen werden daten-parallelisiert und ergeben somit die Multi-Core Standard Template Library (MCSTL). Diese ist geeignet für Parallelrechner mit gemeinsamem Speicher, insbesondere also Mehrkern-Rechner. Sie enthält Routinen wie z. B. suchen, partitionieren, mischen, sortieren, und zufällig permutieren. Wir wählen geeignete parallele Algorithmen für die einzelnen Routinen aus, passend zu den Gegebenheiten.

Desweiteren entwerfen wir Algorithmen zur Manipulation spezieller Datenstrukturen. Für das Einfügen großer Datenmengen in ein assoziatives Feld bzw. dessen Konstruktion aus einer großen Eingabemenge betrachten wir parallele Algorithmen für Rot-Schwarz-Bäume. Um auch für Sequenzen, die nicht wahlfrei zugreifbar sind, Daten-Parallelisierung zu ermöglichen, entwerfen wir einen Algorithmus, der sie in nur einem sequentiell Durchlauf in ungefähr gleich große Teile zerteilt, ohne anfangs die Länge zu kennen.

Die Leistung aller Algorithmen und ihrer Implementierungen wird ausführlich mit Experimenten auf verschiedenen Rechnern evaluiert. Dabei werden auch Beschränkungen der Hardware berücksichtigt, z. B. die geteilte Speicherbandbreite. Es werden meist gute bis sehr

gute Beschleunigungswerte erreicht, oft auch schon für relative kleine Eingaben, ab einer sequentiellen Laufzeit von 100 Mikrosekunden.

Die softwaretechnischen Aspekten der Integration der parallelisierten Routinen in eine existierende Bibliothek werden ebenfalls diskutiert. Hier ist insbesondere die Erhaltung der Semantik ein Kernthema.

Zwei Fallstudien demonstrieren die Verwendbarkeit der MCSTL-Routinen als Unterprogramme in algorithmischen Anwendungen. Zuerst betrachten wir die Konstruktion eines Suffix Arrays, einer Indexdatenstruktur für Volltextsuche. Durch Verwendung der parallelisierten Routinen als Unterprogramme wird eine existierende Implementierung um Faktor 5 beschleunigt. Außerdem wird eine Variante des Kruskal-Algorithmus zum Berechnen des minimalen Spannbaums durch die Parallelisierung mittels der MCSTL so viel schneller, dass sie andere, nicht parallelisierbare Algorithmen hinter sich lässt.

Aus dem Gebiet der algorithmischen Geometrie werden einige ausgewählte Routinen der Computational Geometry Algorithms Library (CGAL) parallel implementiert, auch unter Zuhilfenahme von MCSTL-Routinen.

Das Sortieren entlang der raumfüllenden Hilbert-Kurve dient vor allem als Vorverarbeitungsschritt anderer Algorithmen, und benötigt die Parallelisierung, um die Skalierbarkeit jener Algorithmen nicht zu gefährden.

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 eigentlichen Objekte tatsächlich schneiden.

Die Konstruktion der Delaunay-Triangulierung in 3 Dimensionen ist Teil vieler Verfahren zur Manipulation von 3D-Objekten. Im Gegensatz zu den bisherigen Algorithmen verwenden wir hier feingranulare Sperren zur Synchronisation während des konkurrierenden Zugriffs auf die Datenstruktur. Während für die beiden ersten Probleme gute Beschleunigungen erzielt werden, sind diese für die Delaunay-Triangulierung sogar sehr nah am Optimum.

Im weiteren werden Externspeicher-Algorithmen betrachtet, welche große Datenmengen bearbeiten, die auf Grund ihrer Größe nicht mehr in den Hauptspeicher passen, und deshalb auf Festplatte(n) gespeichert sind. Die Routinen der MCSTL werden verwendet, um die interne Rechenarbeit zu beschleunigen. In Kombination mit einem Rahmenwerk zur parallelen Ausführung von Aufgaben auf einer höheren algorithmischen Ebene wird damit auch der Standard Template Library for XXL Data Sets (STXXL) Mehrkern-Parallelismus ermöglicht.

Damit können wir die Lücke zwischen immer größerer Bandbreite der Festplatten und stagnierender Taktfrequenz der Prozessoren verkleinern. Anwendungsbeispiel ist hier wiederum

die Konstruktion eines Suffix Array, dieses Mal für noch größere Eingaben. Wir können diverse Externspeicher-Algorithmen für dieses Problem signifikant beschleunigen.

Alle erwähnten Bibliotheken hatten in ihren sequentiellen Versionen bereits eine große Verbreitung gefunden. Außerdem sind diese inklusive der parallelisierten Varianten frei verfügbar. Somit können Anwendungen, die sie verwenden, direkt von der höheren Geschwindigkeit profitieren.

Schließlich entwerfen wir Algorithmen zum Sortieren sehr großer Datenmengen auf Rechnerbündeln mit verteiltem Speicher und mehreren Festplatten pro Rechenknoten. Dies dient einerseits als Startpunkt für eine Verallgemeinerung der bisherigen Arbeit auf Systeme mit verteiltem Speicher, andererseits als ein weiteres, komplexes Anwendungsbeispiel der MCSTL und der STXXL. Die Implementierung der praktischeren Algorithmen-Variante wird wiederum ausführlich experimentell evaluiert und analysiert. Im Vergleich mit Konkurrenz-Implementierungen schneidet sie exzellent ab, sie erzielte 2009 zwei Leistungs-Weltrekorde, für ca. 1 Terabyte und 100 Terabyte an Daten. Die Leistung relativ zum Hardwareaufwand lag dabei eine Größenordnung über konkurrierenden Ansätzen.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Historical Development

Until 2004, the clock rates of microprocessors were continuously rising. Application performance benefitted from this fact with hardly any effort from the software developers.

This development came to an end in 2005 [Ros08] for reasons stemming from semiconductor technology. Higher clock rates imply higher heat dissipation, and while the computing power grows at most linearly with the clock rate on a given design, the energy consumption grows superlinearly. This is caused by the higher voltage needed to make the transistors switching more quickly. Intel reported that underclocking a certain processor by 20% actually halves the energy consumption, but sacrifices only about 13% of performance [Ros08]. This correlation makes further clock rate increase very inefficient, also in terms of monetary cost, since more powerful cooling devices are more expensive, not to speak of the increased noise level for desktop computers. There is also the cost for the electrical energy itself, growing more important, in particular for the case of large data centers. For portable machines, a higher energy consumption lowers battery lifetime, which is unacceptable in many settings. In addition to the energy consumption problems, the speed of light limits the distance a signal can travel in a clock cycle to a few centimeters, making synchronization across the whole chip increasingly difficult.

Moore's law states that the number of transistors on a single chip doubles about every two years. *This* law does still hold true for modern semiconductor technology. Since the huge transistor budget cannot be sensibly utilized by adding functional units or caches, the major processor vendors have started to integrate multiple full processor designs on a single chip, calling them the multiple *cores* of a processor.

The first AMD dual-core processor appeared in early 2005 (Athlon 64 X2), Intel shipped its first dual-core processor around the same time, however based on the superseded and clock-rate-hungry Pentium 4 design. The first Intel processor based on the power-efficient Core micro architecture appeared in early 2006. This way, multi-core processors started to become mainstream. Before, such designs had existed only in niche markets (IBM POWER4 from 2001, Sun UltraSPARC IV from 2004). In late 2005, Sun launched the first 8-core processor (UltraSPARC T1), which featured particularly trimmed-down cores to allow for such a high number on a regular chip. These cores do not support out-of-order execution, and there is only a single floating point unit for all the cores. On the other hand, they can hide memory latency by utilizing 4 hardware threads per core, which access a comparatively powerful memory subsystem.

Nowadays, in 2010, a usual personal computer has a quad-core processor, high-end machines feature six or eight cores on a single chip, optionally supporting two hardware threads per core.

Since the clock rate is limited, the only way to increase computing power significantly is adding more and more independent cores. For CPUs with a large number of cores, the term *many-core* has been coined, although the border between multi-core and many-core is not clearly defined.

Modern graphics processing units (GPUs) could well be termed to be many-core, they contain up to several hundred primitive compute engines. However, those are not completely independent in terms of control flow, so we have a complicated programming model. Also, the amount of directly accessible memory is limited to a few gigabytes, and communication from and to the main memory is a bottleneck. GPUs can be used as offload engine for large tasks, but are inappropriate for simple tasks, they do not have direct access to peripherals, and support for multiple applications utilizing them concurrently is still very basic. So there is still a conceptual break between the work done by the general purpose processor and the GPU, which is also noticeable in programming them.

In this thesis, we focus on multi-core parallelism based on common general-purpose micro architectures.

1.2 Motivation

As described, multi-core processors provide increased computing power, and they are ubiquitous nowadays. However, the number of simultaneously running programs on a computer is usually limited to a very small number, in particular on non-servers machines, not able to load each core with an own process. Thus, to benefit from the increased processing power, multiple cores have to be utilized by a single program, processing its work in parallel. This has become mandatory not just for a selected number of specialized programs, but for all nontrivial applications, if they do not want to be left behind by the evolving technology. So far, most of the available computing resources are idle due to insufficiently parallelized software.

However, writing properly parallelized programs is hard. Automatic parallelization works only for simple programs, since it is too hard for a compiler to detect high-level, algorithmic parallelization opportunities. For manual parallelization, software developers not only have to think about enabling parallel execution of work, but also about how to actually parallelize the algorithms, and then do testing and debugging. This is beyond the abilities of many developers, and thus costly and error-prone.

This thesis examines a way to tackle the problem by providing software libraries of parallel algorithm implementations. While this approach has been successful in numerics for a long time, it has not yet made its way into the mainstream of non-numeric programming. Here, we consider mainly combinatorial algorithms.

But let us start from the very beginning. Users'[1] desire is not to run a program in parallel, it is to solve a task as quickly as possible, or to process as much data as possible in a given amount of time. This transfers one level down, to the algorithms a program consists of. The libraries presented here provide building blocks to the programmers, offering algorithms with the usual interface, but in a parallelized form, executing faster. Hence, instead of explicitly

[1] We will refer to a person utilizing the libraries in his programs as the *developer*. The *user*, in contrast, runs the final application program.

parallelizing the program itself, it only has to call the parallelized basic algorithms. Hence, this approach raises the task of parallelizing a program to a higher level. The parallel execution becomes implicit.

Algorithm libraries play an important role in the field of *Algorithm Engineering*. Algorithm engineering is a "general methodology for algorithmic research" [San09], whose main process is a cycle of consisting of algorithm design, analysis, implementation, and experimental evaluation. Proximity to applications is kept by using realistic machine models, and doing evaluation on real-world data sets. Algorithm engineering does not replace algorithm theory, but tries to narrow the gap between theory and practice, which is sometimes quite apparent. Algorithm libraries facilitate the implementation of algorithms because they provide easily utilizable subroutines. Just as for theory, where we just cite an algorithm that solves a subtask in a certain running time, we can do the same implementation-wise if an adequate library routine is available. This foster the algorithms that are efficient in practice. In this thesis, we do not provide new algorithmic functionality, but better exploitation of the multi-core hardware, i. e. adapting the implementations to a nowadays more realistic machine model.

1.2.1 Basic Approach

We base our work on already established and widely used algorithmic libraries. This ensures that we can benefit from existing code, reusing software that is widely accepted. It also establishes a performance baseline that is not made up out of thin air. Otherwise, it would be easy to achieve good speedups over slow sequential execution. Also, looking at established libraries helps the selection of algorithms worthwhile considering. Because of performance characteristics or complexity of implementation, the corpus of desired parallel functionality may shift slightly, but they are a good starting point anyway. Finally, established libraries provide an existing user base, yielding early feedback and inspiring the usage of parallelism in other software projects.

Generally, with our algorithms their implementations, we try to achieve the best possible in terms of performance, while maintaining generic usability and flexibility.

Apart from exceptions due to an increased amount of available cache, acceleration is limited by the number of processor cores used. However, the user might appreciate already speedups that are considerably below this limit. After all, the hardware is readily available, at least in terms of acquisition cost. In the worst case, the energy consumption is increased a little bit. On the other hand, utilizing multiple cores can improve the performance per Watt, namely when parallelism compensates for reduced clock speeds and less instruction level parallelism. At first sight, energy consumption is proportional to the clock rate, so increased running time due to a lower clock rate just compensate each other. But lower clock rates allow lower core voltage, and energy consumption is usually proportional to the voltage squared, so we can actually save energy. This extrapolates the approach that led to multi-core processors. Instead of leaving the cores as they are, and adding more and more, we go back to simpler cores, sometimes based on old designs, to allow for an even greater number of them.

Solving a problem in parallel instead of sequentially usually involves an overhead. There is additional communication, synchronization, non-locality. So it is particularly difficult to speed up work that takes little time already sequentially. Still, we will evaluate the behavior for

small problems as well, and identify minimal time spans where parallelization is still beneficial. Repeated executions of the small problems can make the difference still noticeable to the user, improving the interactivity of the system.

1.3 Parallelism Available in Hardware

Parallelism appears in computer hardware in very different ways, the most general definition being that multiple *processing elements (PE)* simultaneously process data. This can happen in a fine-grained to very coarse-grained way. The work can be divided by splitting the data (*data parallelism*) or by executing different processing steps on different PEs (*task parallelism*).

On the most fine-grained level, we have bit parallelism, a form of data parallelism. But computing with numbers of a limited bit size in one step would probably not be considered parallelism by most people, it is just to natural to handle something adequately similar to a natural number with a single step. The sequential *random access machine (RAM)* is the most common model used for analyzing sequential algorithms, and allows just that.

Next on the granularity level follows *instruction level parallelism*, which allows the processor to execute multiple machine instructions in parallel, may it be through its multiple *arithmetic logic units (ALUs)* for superscalarity, or the multiple *pipeline* stages. While pipelining clearly performs task parallelism, for superscalar execution, the classification on data or task parallelism depends on the actual instructions executed. Apart from that, modern processors provide instructions that in one call process multiple data words. After Flynn's actually more general taxonomy [Fly72], they are usually termed *SIMD instructions*.

Before the multi-core era, a processor usually was superscalar, i. e. it contained several ALUs, but those were controlled by a common control unit, coordinating a single stream of instructions. Multiple of such processors could be plugged into a mainboard, forming a *symmetric multiprocessing (SMP)* machine, where all processors access shared memory.

A multi-core processor combines multiple of the *cores* that were formerly called processors on a single chip that sits in a single socket. In either case, it is *thread-level parallelism* that exploits such an installation, again serving either data parallelism or thread parallelism.

A core can support running a limited number of threads quasi-simultaneously by providing several register files, enabling very fast switching between the threads. This method is called *Simultaneous Multithreading (SMT)* and allows for a better utilization of the arithmetic units, mostly by hiding memory latencies.

Scaling the number of sockets in a system is expensive, since this usually includes non-commodity hardware. So in the last two decades, *clusters* composed of many commodity computers became very popular. The *compute nodes* are interconnected by a network of a certain kind, having separate address spaces for the internal memories. Communication happends via *message passing*, we call such systems *distributed memory* computers. Even if a global address space is emulated and memory accesses are transparently rerouted, such machines cannot be programmed the same way as shared memory machines, since (implicit) communication is usually much slower and has a higher latency than for shared memory.

The lower levels of parallelism (bit parallelism, instruction-level parallelism) are orthogonal to the higher levels (thread parallelism, distributed memory parallelism), so they usually

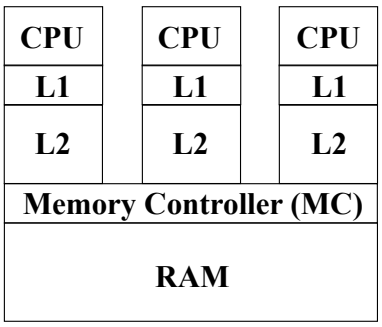


Figure 1.1: Schema for Uniform Memory Architecture.

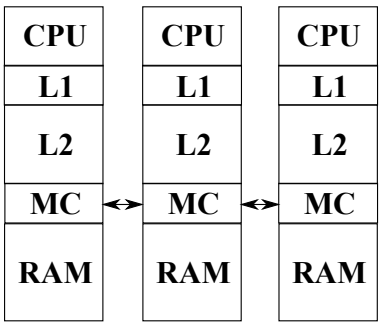


Figure 1.2: Schema for Non-Uniform Memory Architecture.

combine well. Depending on the context, the notion of a processing element can refer to anything from a switchting circuit to a fully autonomous computer. In this thesis, a PE usually relates to a thread running on a CPU core for shared memory, or a compute node for distributed memory.

On the shared memory level, to improve memory access bandwidth and latency, recent processors have an integrated memory controller. A multi-core processor typically has one to three memory controllers shared by all the contained cores. Formerly, the memory was accessed through the chipset, connected to the processor by a bus. When combining several processors with integrated memory controller on a mainboard, it is natural to associate memory with each of the sockets. To preserve the common address space and the shared view of the memory, the processors communicate data via integrated interconnects. These are *cache-coherent*, i. e. accesses to the cache always reflect a single global state of the memory. Fetching a memory word from the memory attached to a different socket requires communication, increasing the latency and lowering the bandwidth, the data possibly going through multiple hops. This non-uniform access costs lead to the notion of a *non-uniform memory architecture (NUMA)* in contrast to the formerly prevailing *uniform memory architecture (UMA)*, where the memory is accessed by one common memory controller. Schemas for UMA and NUMA are shown in Figures 1.1 and 1.2.

Since the communication including the routing is done transparently in hardware, we have a global address space, and the connecting bandwidths are very high. Thus, it would go too far to treat such a configuration as a distributed memory system with a certain interconnection topology. However, taking the properties into account qualitatively can improve performance in certain cases.

1.4 Models of Parallelism

Since the behavior of real hardware is usually very complex, computer scientists abstract away from too much details and use a *machine model*. A model is always a compromise between the accuracy of performance prediction, and the simplicity of the theoretical analysis.

The most basic and most established parallel model outfits the RAM with multiple ALUs, forming the *parallel random access machine (PRAM)* [JÁ92]. All ALUs access the shared memory clock-synchronously, four variants allow or forbid concurrent read and/or write access to the same memory cell. If concurrent write access is permitted, certain rules for achieving consensus are possible (e. g. arbitrary win, all must write same, operators combining all written values). A PRAM algorithm usually assumes that there are as many ALUs as input elements, which is unrealistic for large inputs. However, a PRAM with n ALUs can be emulated by a PRAM with p ALUs with a time dilatation factor of $O(n/p)$.

Only academic prototypes of PRAM machines have been built [BBF+96] to date, due to the hardware limitations that make clock-synchronous execution expensive and slow. Recently, the idea of constructing a PRAM has been revived by Vishkin [WV07], feasting on the high transistor density as well, putting a whole PRAM except the memory onto a single chip. Until competitive hardware is available, the absolute performance stays unclear, though. Emulating PRAM algorithms on available parallel hardware involves significant constant factors [Har94].

For message passing systems, many models haven been proposed that assume a specific connection network topology, e. g. a grid, a hypercube, or different variants of a tree. Others abstract away the interconnect, but introduce uniform communications costs between the PEs.

Valiant proposed the *Bulk Synchronous Parallel Model (BSP)* in 1990 [Val90]. The crucial concept it introduces is the *superstep*, an execution phase which consists of 1. concurrent computation, 2. communication, 3. barrier synchronization. Every BSP algorithm consists of subsequent supersteps, their number should be kept as small as possible. In the first part of each superstep, no data exchange whatsoever between PEs is allowed, the computations happen fully asynchronously. Several variables characterize the communication complexity of a superstep, h being the maximum number of messages a PE sends or receives. A communication is thus called an *h-relation*. The time necessary to deliver a fixed-size message is g. As the name says, the BSP model strives to have only few synchronization phases, which is supported by the fact that the barrier synchronization has cost l, penalizing fine-grained and individual synchronization. Thus, it fits distributed memory machines well.

A specialization of BSP is the *Coarse-Grained Computer (CGM)* model introduced by Dehne et al. in 1993 [DFRC93]. A coarse-grained computer consists of p processors, connected by an arbitrary connection network. Each PE has $O(n/p)$ local memory, i. e. the local memory is explicitly restricted, and a good balance of data must be kept across the PEs. Collective communication again happens through h-relations. Algorithms proposed for this model usually need $n/p > p$, which is a reasonable assumption for most distributed-memory machines, at which this model is aimed.

Recently, models have been proposed that model in particular the cache hierarchy of multi-core processors, establishing costs for the data transfer between hierarchy levels. Arge, Goodrich et al. propose the *Parallel External Memory (PEM)* model [AGNS08], while Valiant extends his BSP model to *Multi-BSP* [Val08].

As we want to exploit multi-core parallelism in this thesis, we use thread-level parallelism on shared memory machines for the libraries, extending it to distributed-memory parallelism for the very-large-scale sorting algorithms in Chapter 7. The parallelizations we do are usually data-parallel, some task-level parallelism is exploited for external memory in Chapter 6. Instruction level and SIMD parallelism are regarded orthogonal, and left to the compiler.

We analyze the shared memory algorithms in a PRAM-like model, but do not use clock synchronous operation or concurrent writes. Instead, many of our algorithms are inspired by the BSP model, avoiding many synchronizations, where this is too costly. This compromise leads to good performance in practice, while keeping performance guarantees under acceptable conditions.

1.5 Metrics

The performance gain from parallelization can be evaluated with different goals in mind, although the always include the relation between running time and input size. There are two ways this relation can be regarded:

1. Parallelization makes the algorithm process more data in the same amount of time.
2. Parallelization makes the algorithm process the same amount of data in less time.

For both ways, we can give a factor of improvement, i. e. how much more data was processed, and how much less time was used, respectively. This factor can then be related to the number of PEs used. Of course, under bad circumstances, parallelization can make the execution even slower, or process less data in the same amount of time, so this factor can be less than 1.

In this thesis, we usually focus on the second criterion, improving responsiveness of the system with respect to the user. Generally, we specify an input, and measure the running time for processing it. For the sequential algorithm, we get T_{seq}, and for the parallel algorithm $T_{\text{par}}(p)$, the parallel running time depending on the number of PEs used.

We define *absolute speedup* as $S(p) := T_{\text{seq}}/T_{\text{par}}(p)$, where T_{seq} is the running time of the sequential reference algorithm. In contrast, the *relative speedup* refers to $S_{\text{rel}}(p) := T_{\text{par}}(1)/T_{\text{par}}(p)$, comparing the running times of the parallel algorithms, once utilizing only one PE, once p PEs. The relative and the absolute speedup usually differ. The common case is that the parallel algorithm includes overheads for handling the general case of multiple PEs, which cannot be fully avoided for the $p = 1$ case, so $T_{\text{par}}(1) > T_{\text{seq}}$.

The speedup can be related to the number of PEs used, giving the *parallel efficiency* $E := S/p$. The *parallel work* is the parallel running time multiplied by the number of PEs, $W := pT_{\text{par}}$. When the parallel work is larger than the sequential work, this inevitably harms efficiency.

If the parallel algorithm is somehow more advanced than the sequential reference algorithm, there might be absolute speedup greater 1 for only one PE, and more than p for p PEs, resulting in $E > 1$. However, the comparison is not fully fair then, since one could simulate the p PEs on a single one in a time-slice manner, taking only $pT_{\text{par}}(p)$ time. In effect, this proposes a better sequential algorithm. Real super-linear speedup can only happen if not just the number of operations per second is increased by the parallelism, but also other resources are extended, such as an amount of memory with fast access (e. g. main memory instead of disk, cache instead of main memory).

Amdahl's law [Amd67] is a simple bound for the speedup of a parallel program, given that a parallelized algorithm still contains a sequential fraction s: $S \leq 1/(1-s)$, e. g. for an algorithm having a 10% sequential fraction, the speedup is limited to 9, no matter how many PEs are

p	Number of threads running in parallel, numbered 0 through $p-1$
$S[0],\ldots,S[n-1]$	Input sequence
n	Problem/input size (number of elements)
m	Secondary problem/input size (number of elements)

Table 1.1: Notation used in Chapters 2 to 6.

used. In practice, however, the sequential part is rarely a constant fraction of the overall time, but depends on the input size and the number of PEs, making this bound not that much useful for performance prediction. However, the theorem show us quite plainly that a sequential part should be reduced as much as possible, because it cannot just make speedup worse, but impose a hard limit.

All metrics described can be applied to both theoretical running time complexity and experimentally measured running time. Unless stated otherwise, we give experimental absolute speedups comparing to the sequential library algorithm we parallelize.

The term *scalability* appears often when discussing parallelism, it is not uniquely defined though, there are several ways to interpret it. “Good” scalability may mean a linear correlation between the number of PEs and the processable input size, for the first criterion, or a good parallel efficiency in general for a large number of PEs.

1.6 Thesis Overview

In Chapter 2, we will first recapitulate algorithms that appear multiple times subsequently. The programming language and the parallelism support libraries we base our work on will be described in Chapter 3. We will consider the first library to multi-core parallelize in Chapter 4, namely the Standard Template Library (STL). It provides basic algorithms for computations in main memory. We parallelize selected algorithms from the Computational Geometry Algorithms Library (CGAL) in Chapter 5. The usage of multi-core parallelism for external memory algorithms is described in Chapter 6, building on the Standard Template Library for XXL Data Sets (STXXL). Those algorithms handle data sets that do not fit into main memory any more, but have to be stored on disks. Table 1.1 states some notation for the Chapters 2 to 6 on the mentioned libraries to be parallelized for shared memory machines. Chapter 7 presents, as continuation and application of the parallelized STL and the parallelized STXXL, sorting algorithms that combine distributed and shared memory parallelism, as well as external memory on parallel disks, including an implementation. Finally, we draw the overall conclusions and indicate future work in Chapter 8.

2 Recurring Algorithms

In this chapter, we recapitulate algorithms that are used multiple times in this thesis.

Multiway mergesort is an I/O- and cache-efficient sorting algorithm. For parallelizing the merge step, the work has to be split up, helped by the so-called multiway partitioning.

2.1 Multiway Selection/Partitioning

Given k sorted sequences $S_1,\ldots,S_k$ and a *global rank* m, we are asked to find splitting positions $i_1,\ldots,i_k$ such that $i_1+\cdots+i_k=m$ and $\forall j,j' \in 1,\ldots,k : S_j[i_j-1] \leq S_{j'}[i_{j'}]$. This is similar to selection, although in the context of this thesis, we are usually not interested in the element of global rank m, but rather the splitting positions. We call finding the splitting positions *multiway partitioning*. Finding the respective element is trivial after multiway partitioning, just take the smallest element right of the split.

Our starting point is an asymptotically optimal algorithm by Varman et al. [VSIR91] for selecting the element of global rank m in a set of sorted sequences. It is fairly complicated, and to our knowledge, has been implemented before only for the case that the number of sequences is a power of two and all sequences have the same length $|S_j| = 2^k - 1$ for some integer k.

The problem is solved by partitioning a more and more refined sample of the sequences. Let $s = 2^i \geq \max_j |S_j|$. Initially, only every s^{th} element is taken into account, then every $(s/2)^{\text{th}}$, $(s/4)^{\text{th}}$ element, and so on. Each refinement doubles the number of considered elements, until all elements are considered for $(s/2^i) = 1$.

Therefore, after a logarithmic number of refinements, the algorithm terminates. In each refinement step, the algorithm must determine for only $\mathrm{O}(k)$ elements whether they belong to the left or the right side. There is an asymptotically efficient but complicated method that achieves this goal in $\mathrm{O}(k)$ time. We do this using a priority queue in (slightly suboptimal) time $\mathrm{O}(k \log k)$ per refinement step. With each refinement step, the number of regarded elements per sequence doubles, so the maximum number of refinements[1] is $\lceil \log \max_j |S_j| \rceil$. The total running time amounts to $\mathrm{O}\big(k \log k \cdot \log \max_j |S_j|\big)$.

Our implementation can handle any number of sequences of arbitrary length. Explicit care has been taken of the case of many equal elements surrounding the requested rank. To allow stable parallelized merging based on this partitioning, the splitter positions may not be in arbitrary positions in the equal subsequence. In fact, there must not be more than one sequence S_j having a splitter "inside" the equal subsequence. All S_i with $i < j$ must have the splitter at the end of it, all S_i with $i > j$ must have the splitter at its beginning.

[1] Unless stated otherwise, all logarithms have base 2.

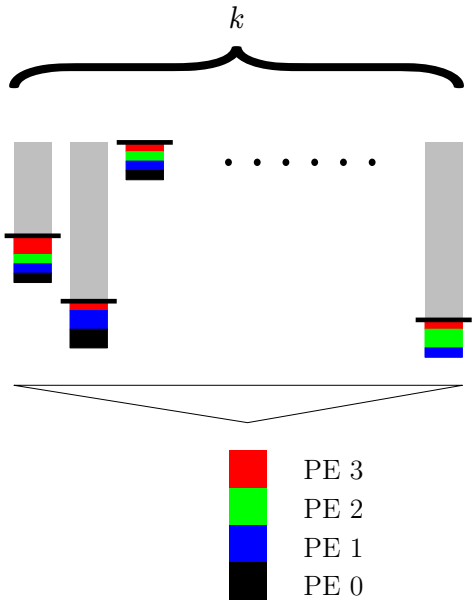


Figure 2.1: Schema of parallel multiway merging, in this case merging a specified number of elements only.

2.2 Multiway Merging

Merging two sorted sequences S_1 and S_2 is usually understood as efficiently coalescing them into one sorted sequence. Its generalization to multiple sequences $S_1,\ldots,S_k$ is called *multiway merging*.

It is often used (e. g., [San00, RKU00, DS03]) instead of repeated binary merging, since it is more I/O- and cache-efficient. The high "branching factor" allows to perform a considerable amount of work per item. This compensates for the costs of accessing the data on the (shared or external) memory.

We parallelize multiway merging using multiway partitioning. Let $n = \sum_{i=1}^{k} |S_i|$. In parallel, each processing element (PE) $i > 1$ does multiway partitioning on the set of sequences, requesting global rank $\lceil \frac{in}{p} \rceil$. Then, each PE i merges the part of the input that is determined by the locally computed splitting positions and by the splitting positions computed by PE $i+1$. For $i = 0$ we use splitting local positions $(0,\ldots,0)$, and for $i = p$, we use local splitting positions $(|S_1|,\ldots,|S_k|)$. The destination offset is also clear from the ranks. Figure 2.1 shows a schematic view of the algorithm.

Our implementation of sequential multiway merging is an adaptation of the implementation used for cache-efficient priority queues and external sorting in [San00, DS03]: For $k = 1$, we just copy, for $k = 2$, we use the folklore merge routine. For $3 \leq k \leq 4$, we use specialized routines that encode the relative ordering of the next elements of each sequence into the program counter, i. e. we execute a different code variant depending on the ordering, jumping around between these. For $k > 4$, a loser tree data structure [Knu98] keeps the next element of each sequence. The total execution time of our algorithm is $\mathrm{O}\left(\frac{n}{p}\log k + k\log k \cdot \log \max_i |S_i|\right)$. Note

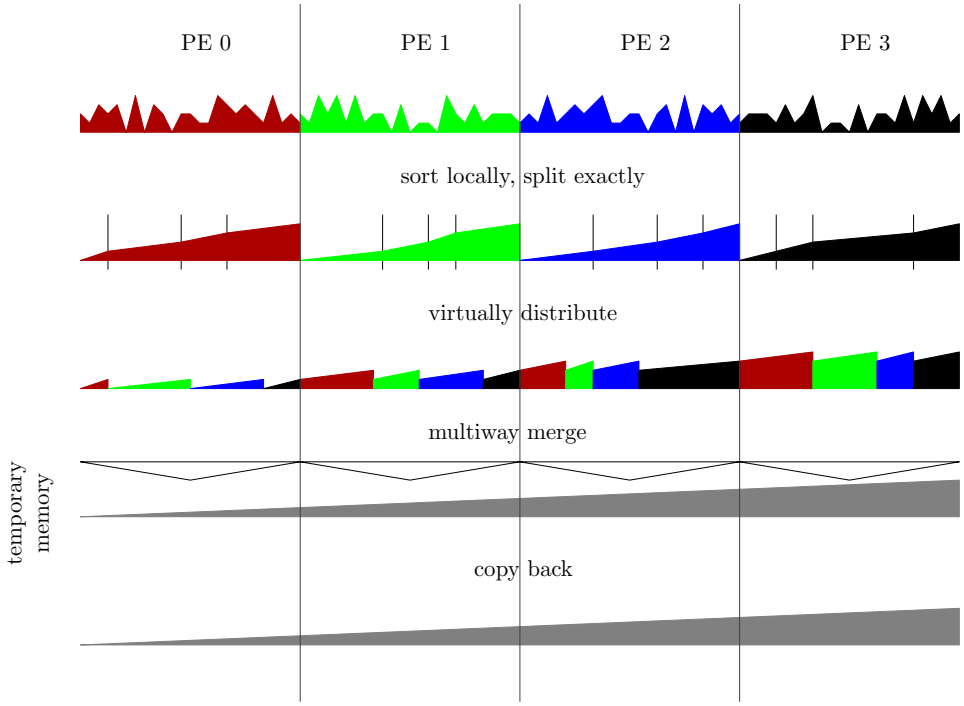


Figure 2.2: Schema of parallel multiway mergesort. For shared memory, the distribution depicted in the middle row takes place only virtually, i. e. the PEs read the data from where it is already, there is no copying. The final copying step is optional, only necessary if the result is expected in the same place as the input.

that the major first term is p times smaller than in sequential merging, the rest is lower-order in terms of the input size.

2.3 Multiway Mergesort

Based on the components explained so far, it is very easy to construct a parallel sorting algorithm. Each PE sorts about $\frac{n}{p}$ elements sequentially, plus minus 1 due to the rounding. The resulting sorted sequences are merged using parallel multiway merging afterwards. We assume the sequential algorithm to run in time $\mathrm{O}(n \log n)$, i. e. $\mathrm{O}\left(\frac{n}{p} \log \frac{n}{p}\right)$ per PE. Substituting $k = p$ and $S_i = \frac{n}{p}$ in the formula for multiway merging results in $\mathrm{O}\left(\frac{n}{p} \log p + p \log p \cdot \log \frac{n}{p}\right)$. The sequential sorting and the actual merging add up to $\mathrm{O}\left(\frac{n}{p} \log n\right)$, but there is still the partitioning, so we have in total $O(\frac{n}{p} \log n + p \log p \cdot \log \frac{n}{p})$. Figure 2.2 illustrates the algorithm.

3 Platform

3.1 C++

C++ is a well-established programming language with a rich syntax, supporting both object-oriented programming, and generic programming via templates. It comes with no inherent performance penalty, following the "zero overhead principle", which states that it does not incur overhead for features that are not used. Since it is compiled offline, elaborate optimization routines can be applied. For languages producing executables for a virtual machine, e. g. Java or C#, this is not so clear.

This makes C++ an ideal choice for high-performance applications. The standardization [C++03] of C++ has set a common ground for language and library implementations, which are widely available and of high quality. The C++ standard is also evolving, as can be seen by the proposed upcoming standard, termed C++0x [C++10]. The Standard Template Library (STL) [PSLM00] is part of the existing C++ library specification, and provides many useful generic data structures and algorithms to its user, its design serving as a blueprint for many algorithm libraries.

3.2 Platform Support for Threading

Any parallel program or library needs a foundation that enables concurrent execution. For our shared-memory parallel libraries, this foundation needs to be both efficient and platform-independent.

For the purely data-parallel libraries, for basic internal algorithms (Chapter 4) and geometric algorithms (Chapter 5), we have chosen to use *OpenMP* [Ope08]. OpenMP is a language extension supported by a run-time library. It provides a more elegant interface to parallel execution than library-style approaches like POSIX threads, and also the threading interface that will be part of the C++0x standard.

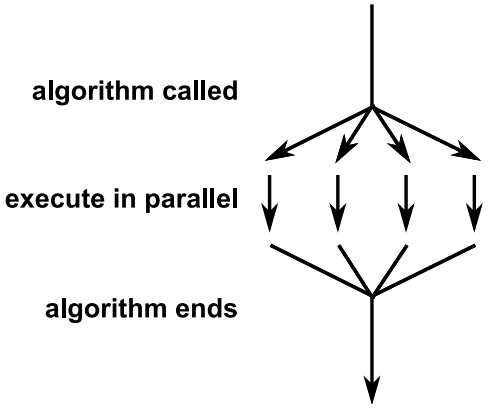


Figure 3.1: Schema of fork-join parallelism.

Its advantages include:

- OpenMP provides so-called "fork-join parallelism", which is exactly what we need for parallelizing data-parallel algorithms. The work is divided, the threads start working, they re-join, and the function call returns, as illustrated by Figure 3.1.

- OpenMP is orthogonal to the programming language, which is affirmed by a Fortran variant being available. Many programming languages with first class parallel programming support have been invented[1], however, those are usually decoupled from the other developments ongoing in language design. OpenMP makes a good compromise by extending C++ through the syntax that is intended for augmentation of the language, namely `pragma`.

- The `pragma` syntax allows to intertwine parallelism specifications and actual code closely, with no need to put functor classes somewhere far into the code, outside of the current function scope. This might be partly compensated by the closures from upcoming C++0x, though.

- Low-level issues like reusing threads from a pool, and getting information about the environment, e. g. the number of cores, can be left to the OpenMP runtime. A lot of research is happening in this field, whose results can be quickly brought in because we base on such an established interface. The user can configure the runtime environment using standard procedures.

- Since OpenMP is integrated into the compiler, very efficient code can be generated that uses platform-specific functionality like atomic operations provided by the hardware.

[1] In PPoPP 2007, a keynote speaker had counted all the parallel programming languages developed in academia over the recent years, some number $\ell > 100$, questioning this development. Immediately after the keynote, the next talk introduced parallel programming language number $\ell + 1$.

- OpenMP is platform-independent. The GNU compiler collection (GCC) supports OpenMP 2.5 from version 4.2, and OpenMP 3.0 from version 4.4. OpenMP is also supported by all major commercial C++ compilers, like the ones from Sun, Intel, and Microsoft.
- Augmented code also compiles and runs correctly (though not in parallel) with compilers not natively supporting OpenMP, adding an dummy runtime library.
- OpenMP is an open standard, supervised by the OpenMP Architecture Review Board.

It is a common misunderstanding that OpenMP is only good for parallelizing simple for-loops. This impression comes from the fact that most introductory examples present this simple case.

We use OpenMP to execute a code block by a certain number of threads (called a *team*) in parallel, in a so-called *parallel region*. Splitting the work is done manually, breaking symmetry by using the `omp_get_num_threads` and `omp_get_thread_num` function calls. Intermediate barriers and sections that are to be executed by one thread only, are achieved by the respective pragmas. Thus, a algorithm framework looks like this:

```
void algorithm(...)
{
  #pragma omp parallel
  {
    int num_threads = omp_get_num_threads(),
        iam         = omp_get_thread_num();

    #pragma single
    {
      //split up work sequentially
    }

    //do work in parallel for PE iam

    #pragma omp barrier  //synchronize for data exchange

    //go ahead in parallel
  }
}
```

A recent and very convenient addition (in version 3.0) is the `task` construct, which allows (nested) asynchronous processing of code blocks, scheduled by the OpenMP runtime. The tasks can be spawned at arbitrary places in the code, and the spawning thread can wait for their termination. The parallel region starts the threads, immediately suspending them to wait for tasks, as exemplified here.

```
void algorithm(...)
{
  #pragma omp parallel
  {
```

```
    #pragma single
    {
      //create tasks
      ... //recursive, iterative
        #pragma omp task
        {
          //spawn off as task
        }
      ...

      #pragma omp taskwait //wait for all task spawned in this thread/task
    }
  }
}
```

In addition to the actual thread control via OpenMP, we use a thin platform-specific layer that provide access to a small number of efficiently implemented primitive atomic operations like fetch-and-add and compare-and-swap. Atomic fetch-and-add increases an integer by a certain amount, with no interference from other threads possible during reading and writing the value. Since it returns the former value, it is very suitable to reserve a range for operation, namely the one between the former value and the increased value. Atomic compare-and-swap compares an integer to a given value, and if they are equal, exchanges it atomically by another given value. This function can be used to transfer one state safely to another.

The atomic operations are usually implemented in hardware, and allow for very efficient communication and synchronization in the simple cases where a single integer is sufficient. Soon, we will able to replace them by the primitives demanded by the C++0x standard, which are already available for some compilers. In case the hardware does not support atomic operations, software solutions can be used at the cost of a performance penalty.

An concurrently modifiable data structure that does not use mutual exclusion through a lock, but just atomic operations, is called *lock-free*.

A program using OpenMP can be configured by the user through different environment variables. Most important are `OMP_NUM_THREADS` and `OMP_DYNAMIC`, which influence the number of threads. Although an OpenMP `parallel` clause takes an argument on the desired number of threads, this value poses an upper limit only, and if no explicit value is given, `OMP_NUM_THREADS` is used. If the dynamic mode is switched on, the runtime can decide to schedule a smaller number of threads to the parallel region. By default, `OMP_NUM_THREADS` is usually set to the number of cores available in the system, `OMP_DYNAMIC` is switched off.

Overall, this means that the number of threads is clear only after the `parallel` statement, i. e. in the parallel region. The code must not make any assumptions about the number of threads beforehand. In many algorithms, this leads to a `single` block right after the beginning of the parallel region, in order to distribute the work appropriately. For best performance, the compiler should identify this case, and start the threads only after the `single` block, having fixed the number of threads in advance, and thus returning the right value from `omp_get_num_threads`.

The environment variable `OMP_SCHEDULE` only influences OpenMP `parallel for` loops, which are rarely used in this work.

Nested parallelism *is* used by some of the algorithms presented, e. g. quicksort. For those, to get best performance, `OMP_NESTED` should be set to `true`. We also configure the OpenMP library, *libgomp*, to provide better responsiveness by threads by lowering `GOMP_SPINCOUNT` to 10 000 in this case, to limit busy waiting and yield to the operating system scheduler more quickly.

In Chapter 6, we will use coarse-grained task-parallelism in addition. OpenMP is not well-suited for this case, because threads have to synchronize in producer-consumer scheme. Since the threads work on large data sets and therefore for long times, we can admit to use regular OS threading without pooling, using the *pthread* library. As we will see, pthread for the task parallelism and OpenMP for the data parallelism combine nicely.

We do *not* make use of transactional memory [HM93]. *Transactional memory* is a concepts that is supposed to simplify concurrent programming by allowing a group of reads and writes from and to memory, usually from a code block, to execute atomically. However, competitive hardware supporting this feature is not yet available. For the case that transactions are emulated in software, it is clear that a hand-programmed transaction protocol can always be at least as fast. In fact, we do use *atomic instructions*, which can be seen as transactional memory limited to a single data word. For this granularity, hardware support is available on multi-core processors, delivering very good performance.

Anyway, the libraries considered are supposed to keep their interface in the parallelized version. So it does not make a difference to the developer whether we use transactional memory internally or not. If high-performance implementations are available one day, we could utilize it internally, without the necessity of changes by the developer. Independent of that, memory transactions on the developer level could actually improve usability by ensuring atomicity of developer-defined functions, which might be called asynchronously by a library routine.

3.3 Platform Support for Distributed-Memory Computing

The algorithms presented in Chapter 6 are designed for distributed memory systems, but also use shared memory parallelism locally on each node. For communication between the nodes, we use the Message Passing Interface [MPI94], which is the standard for this kind of computer. It follows a *single-program multiple-data* approach, i. e. the same program runs on every node.

3.4 Machines for Experimental Evaluation

For evaluating the performance of the parallelized algorithms, we used three computers. Predominantely, we used a machine with two quad-core AMD Opteron processor, which is referred to as OPTERON. Secondly, we used a similarly structured system based on Intel Xeon processors, which is referred to as XEON. Their main difference is that the two sockets of XEON access the whole memory through the same controller (UMA), while OPTERON is a NUMA. For the OPTERON, all cores of a socket share the L3 cache, while for the XEON, only two cores share the L2 cache, which is then directly connected to the main memory. The third machine,

the SUN T1, was the first processor to combine eight cores on a single chip. In contrast to the Intel and AMD machines, the cores support simultaneous multi-threading with 4 threads per core. However, the performance of a single core is comparatively low, which might be partly due to the low clock rate. Furthermore, all the cores even have to share a single floating-point unit.

The detailed specifications of all machines can be seen in Table 3.1. They were running 64-bit Linux 2.6 kernels.

Name	OPTERON	XEON	SUN T1
CPU Model	AMD Opteron 2350	Intel Xeon 5345	Sun Ultrasparc T1
#Sockets	2	2	1
#Cores	8 (4 per socket)	8 (4 per socket)	8 (8 per socket)
#Threads	8 (1 per core)	8 (1 per core)	32 (4 per core)
Architecture	Barcelona	Woodcrest	Niagara
Clock Rate	2.0 GHz	2.33 GHz	1.0 GHz
L1 Cache L2 Cache L3 Cache	2x 4x 64 KiB 2x 2x 4 MiB (shared by 2 cores each) –	2x 4x 16 KiB 2x 4x 512 KiB 2x 2 MiB (shared)	 3 MiB shared
Memory	16 GiB	16 GiB	8 GiB
Memory Model	NUMA	UMA	UMA
Hard Disks	8x 320 GB	8x 500 GB	n. a.

Table 3.1: Technical Details of the Test Machines.

Since the memory bandwidth plays an important role for overall performance, we have measured it using the renowned Stream Benchmark [McC07]. In Table 3.2, we report the results[2] for the category which gave the highest values. The Stream Benchmark is multi-core-enabled, we notice that multiple threads increase the bandwidth. This is most obvious for the SUN T1, where the full bandwidth is only achieved when loading each core with 4 threads, for a total of 32. The OPTERON benefits from the NUMA architecture if the data is distributed across two sockets. However, when we restrict the data to one socket, a common case for input data which is allocated by a single allocation, the results are similar to the XEON, namely about 5–6 GiB/s of bandwidth.

The theoretical peak bandwidth for the XEON is about 21 GiB/s, so only a quarter of that is utilized in practice, using the Intel 5000P chipset. The OPTERON gains more performance from its integrated memory controller and the NUMA.

[2] 1 MB = 10^6 bytes, 1 GB = 10^9 bytes, 1 TB = 10^{12} bytes, and 1 KiB = 2^{10} bytes, 1 MiB = 2^{20} bytes, 1 GiB = 2^{30} bytes.

	Bandwidth in MiB/s			
Number	OPTERON		XEON	SUN T1
of threads	two sockets	one socket		
1	3611	3611	3138	370
2	6966	5462	5058	735
3	8077	5863	4387	1095
4	10581	5950	5209	823
5	9705	6093	5189	668
6	8962	6080	5198	602
7	10425	5972	5209	700
8	11737	5132	5191	798
16				1531
32				2848

Table 3.2: Memory bandwidths of the machines tested on, measured using the Stream benchmark.

4 Basic Internal Memory Algorithms

In this chapter, we consider the shared-memory parallelization of basic algorithms for internal (i. e. main) memory. These algorithms are most basic, but also most fundamental, including functionality like load balancing embarrassingly parallel work, sorting, merging two or more sorted sequences, random shuffling, partitioning a sequence by a pivot, finding specific elements in sequences, prefix summations, and set operations. We also present algorithms working on dictionaries represented as a balanced binary tree, and a method to process linked lists in parallel.

These basic algorithms are usually used as subtasks of the actual problem solving strategy. By using their parallelized variants, the developer can overall speed up the computation.

We describe the design and implementation of the *Multi-Core Standard Template Library (MCSTL)*, which provides efficient parallel implementations of the algorithms in the Standard Template Library (STL) [PSLM00]. Since the STL interface is long-established, well-understood, and widely used, it is a very good way to bring easy-to-use parallel algorithms to the developer. By utilizing parallelized STL algorithms as black boxes, the developer can completely abstract from parallel programming, but still benefit from multi-core power. Programs can profit from the implicit parallelization even by just recompiling them.

The MCSTL has mostly[1] been integrated into the STL implementation of the GNU C++ compiler, the *libstdc++*, as the so-called *parallel mode*. We will use MCSTL and libstdc++ parallel mode as synonyms, but we will refer to the latter in particular for the software engineering issues, since they are more concerned with the actual integration.

The MCSTL limits itself to shared memory systems and thus can offer features that would be difficult to implement efficiently on distributed memory systems, e. g. fine-grained dynamic load-balancing. This approach brings its own challenges. "Traditional" parallel computing works with many processors, specialized applications, and huge inputs. In contrast, the MCSTL should already yield noticeable speedup for as few as two cores, and also for relatively small inputs. In fact, the amount of work submitted to each call of one of the simple algorithms in STL may be fairly small.

We do not consider thread-safe versions of STL data structures, i. e. implementations that can be safely accessed by multiple threads concurrently.

Another issue is that the MCSTL has to coexist with other forms of parallelism. The operating system will use some of the computational resources, multiple processes (all of them possibly using the MCSTL) may execute on the same machine, and there might be some degree of high-level parallelization using multi-threading within the application. For many algorithms, the MCSTL provides dynamic load balancing to counter this problem.

[1] The data structure algorithms are still missing, but there is no principal obstacle in integrating it as well.

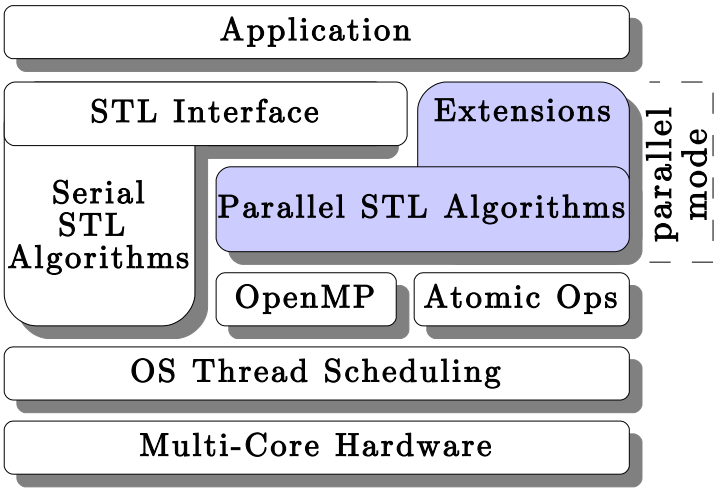


Figure 4.1: Abstraction layers of the MCSTL/libstdc++ parallel mode. The parallelized STL algorithms exploit the multi-core hardware, threading is enabled via OpenMP and the operating system scheduler. The application uses the parallelized and the remaining sequential algorithms using the original STL interface, plus some additional parallel algorithms and parallelization helper algorithms as STL extensions.

We base the MCSTL on OpenMP plus atomic instructions, as described in Chapter 3. The abstraction layers are visualized in Figure 4.1.

Most of the algorithms we present in Section 4.2 are previously known or can be considered folklore if we do not cite a specific reference. Our main contributions are selecting good starting points and engineering efficient implementations, and evaluating them in detail on current multi-core hardware. Interestingly, many of our algorithms were originally developed for distributed memory parallel computing. Very often, such algorithms naturally translate into shared memory algorithms with good cache locality and few costly synchronization operations. In contrast, the classical PRAM model for shared memory algorithms often yields algorithms that have too fine-grained parallelism and too large overheads for practical use. The key ideas behind our sorting algorithms are also not new [VSIR91, TZ03] but not widely known. It is somewhat astonishing that although there are virtually hundreds of papers on parallel sorting, so few notice that multiway merging can be done with exact splitting, and that the partition in quicksort can be parallelized without changing the inner loop. The algorithms for data structures in Sections 4.3 are new, to the best of our knowledge.

Overview

After mentioning generally related work in Section 4.2, we describe the basic algorithms, which usually work on random-accessible sequences. The dictionary algorithms specialized for dictionaries are treated in Section 4.3.

The issues of providing an STL-compatible interface are discussed in Section 4.4. One major point among that, enabling parallel execution for sequences with non-random-access iterators, is treated in Section 4.5.

In Section 4.6, we show two applications that benefit from easy parallelization.

All the sections also contain experimental performance evaluation. Conclusions and proposals for future on this part work can be found in Section 4.7.

4.1 Related Work

Parallelizing STL algorithms has been addressed before. STAPL [AJR+01, TTT+05] provides parallel container classes that allow writing parallel programs scalable to cluster . However, judged from publications, only few of the STL algorithms have been implemented, and those that have been implemented sometimes deviate from the STL semantics (e. g. `p_find` in [AJR+01]). Also, the focus is very much on distributed memory systems and large scale. Recently, the work also covered linear algebra [BST+08]. No code is publicly available.

The Intel Threading Building Blocks [Int] focus on the support of task based parallelism. They contain only few parallel algorithms, but in exchange, many thread-safe data structures.

MPTL [Bae06] is another shared memory STL library. It parallelizes many of the simple algorithms in STL using elegant abstractions. However, it does not implement the more complicated parallel algorithms `partition`, `nth_element`, `random_shuffle`, `partial_sum`, `merge`. MPTL has a "naïve" parallelization of quicksort using sequential partitioning. Similarly, there is only a "naïve" implementation of `find` that does not guarantee any speedup even if the position sought is far away from the starting point. MPTL offers a simple dynamic load balancer based on the master worker scheme and fixed chunk sizes.

Our early publications have inspired more work on the topic. The particular point of Mallach [Mal10] is performance improvement by using thread affinity, i. e. assigning threads to specific cores. However, such a fixed relation can lead to very bad performance when multiple programs do this at the same time. We prefer to leave such issues to the underlying infrastructure (i. e. OpenMP), but propose that the OS provides an interface to specify desired closeness of threads in terms of shared cache [Blo08]. Alternative parallel algorithms for the STL are proposed by Traore et al. [TRM+08]. However, as in [Mal10], the measured performance results for the MCSTL are usually worse than presented here, which might be due to the old versions of compiler, OpenMP library and MCSTL/libstdc++ parallel mode used. Our work on partition is continued by Frias in [FP08].

4.2 Algorithms

Table 4.1 summarizes the status of parallelization for the STL algorithms. The ones having a sublinear running time are not worthwhile parallelizing, due to the very short running time not compensating for the parallelization overhead.

Algorithm Class	Function Call(s)	Status	w/LB	w/oLB
Embarrassingly Parallel	`for_each`, `generate(_n)`, `fill(_n)`, `count(_if)`, `transform`	impl	yes	yes
Find	`find(_if)`, `search(_n)`, `mismatch`, `equal`, `adjacent_find`, `lexicographical_compare`	impl	yes	nww
Numerical Algorithms	`accumulate`, `partial_sum`, `inner_product`, `adjacent_difference`, `min_element`, `max_element`	impl	nww	yes
Partition	*`(stable_)`*`partition`,	impl	yes	nww
Merge	`merge`, `multiway_merge`, *`inplace_merge`*	impl	ww	yes
Partial Sort	`nth_element`, `partial_sort`*`(_copy)`*	impl	yes	ww
Sort	`sort`, `stable_sort`	impl	impl	yes
Shuffle	`random_shuffle`	impl	yes	nww
Set Operations	`unique_copy`, `replace(_copy)(_if)`, `set_union`, `set_intersection`, `set_(symmetric_)difference`	impl	nww	yes
Containers	`(multi_)map/set`	impl	yes	no
Vector	*`valarray` operations*	ww		
Heap	*`make_heap`*, *`sort_heap`*	ww		
Sublinear Functions	*`push_heap`*, *`pop_heap`*, *`min`*, *`max`*, *`next_permutation`*, *`prev_permutation`*, *`lower_bound`*, *`upper_bound`*, *`equal_range`*, *`binary_search`*	nww		

Table 4.1: Parallelization status of STL functions. **impl** = implemented, (except *italicized*), **ww** = worthwhile implementing, **nww** = not worthwhile implementing, **wLB** = with dynamic load-balancing, **w/oLB** = without dynamic load-balancing

Embarrassingly Parallel Computation

Several STL algorithms basically process n independent atomic jobs in parallel, as listed in the row "Embarrassingly Parallel" in Table 4.1. This looks quite easy on the first glance, we could simply assign n/p jobs[2] to each thread. Indeed, we provide such an implementation in order to scale down to very small, fine-grained, and uniform inputs on dedicated machines. However, in general, we cannot assume anything about the availability of cores or the running time of the jobs, which might reach from a few machine instructions to complex computations. Hence, we can neither afford dynamic load balancers that schedule each job individually nor should we a priori cluster chunks of jobs together. *Random Polling* or *randomized work stealing* is a way out of this dilemma. The method goes back at least to [FM87], using it for loop scheduling is proposed in [San98b]. An elegant analysis for shared memory that coined the term work stealing is in [BL99].

Initially, each thread gets n/p consecutive jobs defined by a pair of iterators. A busy thread processes one job after the other. When a thread is done, it looks for more work. If all jobs are already finished, it terminates and so does the whole algorithm. Otherwise, it tries to steal half of the jobs from a randomly chosen other thread. We implement this without intervention of the victim, using an atomic fetch-and-add instruction, which moves the victim's current position to the right. This way, the jobs will be dynamically partitioned into a small number of consecutive intervals, which important for (parallel) cache efficiency. Very large jobs will never irrevocably be clustered together, and threads without a processor core assigned to them will lose jobs to stealing (and thus active) threads. To achieve a good compromise between worst-case performance and overhead, the user can optionally devise the algorithm to process the jobs in indivisible chunks of a certain size. Using known techniques (e. g. [BL99, San02]), it can be shown that almost perfect load balancing can be achieved at the cost of only a logarithmic number of work stealing operations per thread.

The latest OpenMP standard specifies the `parallel for` loop to work on C++ random access iterators in addition to integer loop variables. This supplements the `for_each` functionality. One drawback is that no scheduling is required that provides a worst-case guarantee on running time.

Find

Function `find_if` determines the first position in a sequence that satisfies a certain predicate. Routines `find`, `adjacent_find`, `mismatch`, `equal`, and `lexicographical_compare` can be immediately reduced to `find_if`. On the first glance, this looks like just another embarrassingly parallel problem. However, a naïve parallelization may not yield any speedup. Assume the first matching element is at position m in a sequence of length n. The sequential algorithm needs time $O(m)$. A naïve parallelization that splits the input into p pieces of size n/p needs time $\Omega(n/p) = \Omega(m)$ if $m = (n/p) - 1$.

[2] If n is not divisible by p, we round up for the first $n \mod p$ threads, and round down for the others. We will refer to this method as "division into parts of almost equal size", since the part sizes differ by at most 1.

In practice, we expect slowdown also if m is so small that the overhead for starting the threads becomes overwhelming. Note that this is a quite common situation even if n is huge, and that the library has no way to predict m given n.

To amortize synchronization overhead, the threads work block-wise. Each thread reserves a block of a certain size by applying the fetch-and-add primitive to the current offset o, which starts at 0 relative to the sequence begin. If the search in this block is successful, the thread stores the resulting position, unless another thread has found an earlier hit in the meantime. Then, it reserves the rest of the sequence, making the the other threads terminate as soon as they try to reserve their next block.

A large block size is good for overhead amortization, but can make the performance for small m very bad, because every thread will finish scanning the current block, even if another thread has found the first match already (checking for that more often would increase overhead again). Therefore, to limit slowdown, the block size B should be chosen depending on m. Since m is unknown beforehand, we have an online problem.

The sequential algorithm needs m predicate calls (steps) to find the match, and the adversary can determine m. For the parallel algorithm, we upper bound the running time by $m/p+B$ steps. With a growing number of threads, one of them expectedly still has to scan an almost full block. If we aspire speedup at least S in terms of steps, we get: $\frac{m}{m/p+B} \geq S$, which solves to $B \leq (1/S-1/p)m$. Substituting the parallel efficiency $E = S/p$, we have $B \leq (1/E-1)m/p$. Intuitively, this is justified by the fact that the overhead B should be a constant fraction of the number of steps done so far, namely m/p. However, for full efficiency and linear speedup, this equation solves to 0, which is impossible. Thus, we have to content ourselves with an efficiency less than 1. The atomic operation overhead will increase with the number of threads, which is not so nice, and proved to perform worse in practice. Thus, we keep the dependency on m, but drop the dependency on p, leading to $B \leq (1/E-1)m$. Then, for 99% desired efficiency, the block size is limited by $B \leq 0.01m$.

A good and reasonably conservative estimation for m is the current offset o, although some threads might not be finished scanning blocks before o. Before allocating a new block, each thread looks up o and computes the next block size accordingly. Since the sequence length covered now grows exponentially with each block reservation, we have only $O(\log m)$ block reservations. The logarithm has a small base in practice, though.

In the very beginning, the estimated m is $o = 0$, so again, we have trouble finding an appropriate B. Hence, our algorithm starts with a *sequential* search for the first m_0 steps. The tuning parameter m_0 should take into account the practical atomic operation overhead in relation to the predicate computation time.

The find algorithm is visualized in Figure 4.2. In addition to the atomic operations, we need only a single synchronization, for termination.

Partial Sum

For a given sequence S, the partial sums form the sequence $S[0], S[0]+S[1], S[0]+S[1]+S[2], \ldots, \sum_{i=0}^{n-1} S[i]$. They are also known as prefix sums. For computing them, we synchronize only twice, instead of $\log p$ times, as done by typical textbook algorithms (e. g. [JÓ92]). After splitting the sequence into $p+1$ parts, the partial sums of part 0 and the total sums of parts

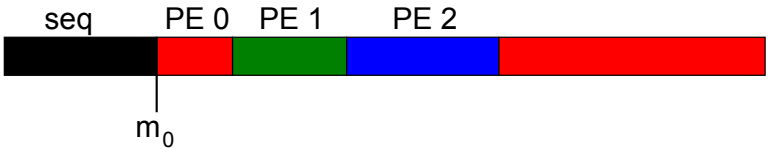


Figure 4.2: Visualization of the parallel find algorithm

$1,\ldots,p-1$ are computed in parallel. After prefix summing these intermediate results in a sequential step, the partial sums of parts $1,\ldots,p$ are calculated, now knowing all start values. The first thread possibly takes longer for the first step, since it has to write back many results. To compensate for that, we can specify a dilatation factor d. Part 0 will be made d times smaller than the other parts, so all threads have about the same running time for the first step. The sequential cost is then n operations comprised of summing up and writing back, proportional to dn summations. To balance work in the first step, $dm = (n-m)/p$ must hold, which solves to $m = n/(1+dp)$. The total parallel running time is $(1+d)(n-m)/p = (1+d)nd/(1+dp)$, the maximum speedup achievable is $S = (dp+1)/(1+d)$, i. e. at least $S = \frac{1}{2}p + \frac{1}{2}$ for $d = 1$, and at most $S = p$ for $d \to \infty$. For $d = 2$, the reasonable assumption of taking a read access as expensive as a write access, we have $S = \frac{2}{3}p + \frac{1}{3}$.

Partition

For `partition`, given a *pivot predicate* P, we are asked to permute $S[0],\ldots,S[n-1]$ such that we have $P(S[i])$ for $i < m$ and $\neg P(S[i])$ for $i \geq m$. This STL routine is the most important building block for quicksort, selection, etc. We use a parallel algorithm similar to the one by Tsigas and Zhang [TZ03], which has many advantages. Its inner loop is the same as in sequential quicksort, it works in-place, and it is dynamically load-balanced.

The normal sequential algorithm scans S from both ends until it finds two elements $S[i]$ and $S[j]$ that belong to the "other" side respectively. It swaps $S[i]$ and $S[j]$ and continues scanning, until the pointers overlap.

The parallel algorithm works similarly. However, each thread reserves two chunks of a certain size B from each end. It performs the partitioning of those two chunks, until one of them runs empty. If the left chunk runs empty, it reserves a succeeding block using a fetch-and-add primitive. Symmetrically, if the right size runs empty, it reserves a preceding block. This process terminates when there are less than B elements remaining between the left and the right boundary.

When all threads have noticed this condition, there is at most one chunk per thread that is partly unprocessed. The threads swap their respective unprocessed part to the "middle" of the sequence as follows. One thread sums up how many elements remain unprocessed on each side, defining the size of the middle zone. All threads reserve the space for the blocks that already lie in the middle zone, which saves unnecessary copies. The threads reserve space for the remaining blocks by atomically setting a value from 0 to 1 afterwards, before actually copying them there. The remaining elements in the middle are subsequently treated sequentially. It is not beneficial to try on in parallel for a limited number of threads. The running time of

this algorithm is bounded by $O(n/p + Bp)$, the second added takes the sequential cleanup into account.

m[th] Element /Partial Sort

The `nth_element` routine reorders the input sequence such that the element with rank[3] m is at position m, and all elements to the left are less equal, and all elements to the right are greater equal. Using the above parallel partitioning algorithm, it is easy to parallelize the well-known quickselect algorithm: partition around a pivot chosen as the median of three. If the left side has at least m elements, recurse on the left side. Otherwise recurse on the right side. We switch to a sequential algorithm when the subproblem size becomes smaller than size $2Bp$ where B is the block size used in `partition`. We get a total expected execution time $O\left(\frac{n}{p} + Bp \log p\right)$, since we have $O(\log p)$ `partition` calls.

By completing to sort the left subsequence $S[0], \ldots, S[m-1]$ with the parallel `sort` call, we can easily implement `partial_sort`, which requires all elements of up to rank m to be in sorted order, starting at the beginning of the sequence.

Merging

We implement `merge` as described in Section 2.2 using multiway partitioning, each thread serving as a PE. We also provide `multiway_merge` as an STL extension. Both use rely on `multiseq_partition` for splitting up the work. If required, our implementation outputs only the m smallest elements, e. g. for block-wise operation for external memory.

Sort

Using the infrastructure presented here and in Chapter 2, we can implement two different parallel algorithms for comparison-based sorting.

(Stable) Parallel Multiway Mergesort: We implement parallel multiway mergesort as described in Section 2.3, i. e. using the multiway merging as above. It does not support dynamic load balancing, but on the other hand, stable sorting.

The STL semantics require that the input sequence is sorted in-situ. Since the merging is hard to do in-place, especially in parallel, we have to copy all the elements sooner or later. For NUMA systems, it is beneficial to perform the bandwidth-intensive local sort in the close-by memory. Thus, each thread copies the input to temporary storage first, and sorts it there, using the usual `std::sort` routine, which uses *introspective sort*, a worst-case-proof variant of quicksort [Mus97]. Then, the threads merge the elements back to the original location.

Optionally, the sort is made stable by calling `stable_sort` locally, and using stable merging, which is a bit slower.

Load-Balanced Quicksort: Using the parallel partitioning algorithm from Section 4.2, we can obtain a parallel variant of quicksort, as described in [TZ03]. Although this algorithm is

[3]For consistency with our notation, where n is the input size, we use m for the requested rank, despite the name of the STL function.

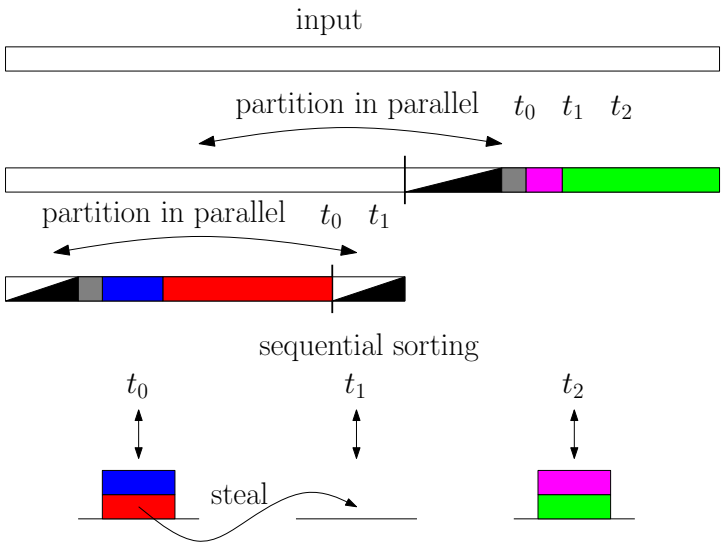


Figure 4.3: Schema of parallel balanced quicksort, using threads t_0, t_1, t_2. The ramped parts are already sorted, while the dark gray parts are currently being partitioned. The colored parts are remembered on the stack and wait for being (stolen and) processed.

likely to be somewhat slower than parallel multiway mergesort, it has the advantage to work in-place and to support dynamic load balancing.

The sequence is partitioned into two parts recursively, using the median-of-three as the pivot. The threads are split up according to the lengths of both parts. When there is only one thread left for a part, the thread sorts that range sequentially. However, the length of those ranges may differ strongly over the threads, resulting in poor overall performance. To overcome this problem, we augment the quicksort with work stealing. Lock-free double-ended queues replace the local call stacks. After partitioning a range, the longer part is pushed [4] onto the top end of the local queue, while the shorter part is sorted recursively. When the recursion returns, a range is popped from the *top* end of the local queue. If there is none available, the thread steals a subsequence from the queue of a random other thread. In fact, it pops a block from the *bottom* end of the queue of the victim. Since this part is relatively large, the overhead of this operation is compensated for quite well. If the length of the current range is smaller than some threshold, no pushing to the local queue is done any more, so the sort is finished completely by the owning thread. Figure 4.3 shows a possible state of operation.

The functionality required for the double-ended queue is quite restricted, as is its maximum size. Operations at the top end are performed by only one specific thread. However, there are concurrent operations by many threads at the bottom end. The push operations must always succeed, all pop operations may (explicitly) fail, and checking for emptiness is futile in this setting because the situation might have been modified before the caller is able to trigger a change itself. The length of each queue is restricted by $\lceil \log_2 n \rceil + p$ at any point in time, since only parts of one recursive descent can reside in one queue, plus less than p entries which were

[4] Only the corresponding iterators are pushed onto the stack, the sequence stays in place.

popped, but still must be read out from other threads. Thus, a circular buffer held in an array and atomic operations like fetch-and-add and compare-and-swap suffice to implement such a data structure, with all operations taking only constant time.

The termination of the algorithm is assured by counting the number of already sorted elements. The count is incremented atomically only when a thread runs out of work, for the sake of low overhead. If an attempt to steal a block from another thread is unsuccessful, the thread offers the operating system to yield control using the respective system call. This is necessary to avoid starvation of threads, in particular if there are less (available) processors than threads. There could still be work available albeit all queues are empty, since all busy threads might be in a high-level partitioning step.

The total expected running time is $\mathrm{O}\left(\frac{n \log n}{p} + Bp \log p\right)$, similar to partition, ignoring the load-balancing overhead.

Set Operations

The STL provides the natural boolean operations on sets, namely union, intersection, difference, and symmetric difference. The two input sets are expected as sorted sequences, equality is assumed when the compare functor returns false in either direction. The operations look similar to merging, however, the crucial difference is that the the output length is unknown beforehand. Thus, we combine merging and prefix sums here. After splitting the sequences using `multiseq_partition`, the first thread executes the actual operation, while the others only count the number of elements resulting from their parts. Sequentially, the target offsets are computed, before the remaining parts are finally processed. This results in a maximum speedup of $\frac{1}{2}p + \frac{1}{2}$.

Random Shuffle

We use a cache-efficient algorithm random permutation algorithm [San98a] which extends naturally to the parallel setting: Split the input into parts of almost equal size, one for each thread. In parallel, throw each element into one out of k random bins, and permute the resulting bins independently in parallel (using the standard algorithm). The parameter k is chosen as the smallest power of 2 such that a bin completely fits into the L2 cache for the second step. We use the Mersenne-Twister random number generator [MN98] which is known to have very good statistical properties even if random numbers are split into their constituent bits, as we do.

4.2.1 Experimental Results

Setup

In the following, we show how the running time relates to the original sequential algorithm provided by the corresponding STL, expressed as speedup. We also give the running time of the sequential algorithm, to allow for objective comparison by the reader.

For the experiments presented here, the parallel execution of all algorithms was forced, to also show the results for small inputs which would not have executed in parallel otherwise.

Usually, a threshold determines the minimum input size an algorithm is called for, to prevent excessive slowdown.

All test were run at least 10 times, for smaller ones more often, the running times were averaged. The speedup is calculated by running the sequential algorithm the same number of times, also averaging the running times. The running time was measured using the OpenMP `omp_get_wtime` call.

Unless stated otherwise, we used uniform random input data for our experiments. The input data varies with every run. The input sizes are rounded powers of $\sqrt{10}$, avoiding possible timing artifacts for powers of 2, and also to demonstrate the general applicability of our algorithms. As data type, we often use 32-bit integer, which is a very simple and small data type. Computations and comparisons on such numbers are very fast, but per byte of transferred data, they are still significant. To demonstrate the change in performance due to a different relation of computation and bandwidths needs, we either increase the size of the data type, or use computationally more expensive functors.

For each number of threads, the repetitions were performed right after the other, which allows for effective thread caching. This is a realistic setting because we expect to run one parallel algorithm after the other on data, usually using the same number of threads.

The input is generated sequentially by the main program thread, immediately before each repetition. Hence, it may (partly) reside in cache of one core, and for NUMA systems, it will reside in the RAM associated with the socket the main program runs on. This is a realistic setting which in fact favors the sequential version because it can access the data right away, while the many threads of the parallel implementation may have to communicate the data via the relatively slow main memory or the processor interconnects.

We used the libstdc++ parallel mode included with GCC 4.4.3, plus some patches[5], which will be integrated into GCC 4.6 the latest. Optimization was switched on using `-O3 -DNDEBUG`.

Embarrassingly Parallel Computations

We tested the algorithms for performing embarrassingly parallel computations by computing the Mandelbrot fractal. For each pixel, this means thousands of iterations of a computation involving complex numbers in double precision. Since the computation is interrupted as soon as the point known to be outside the Mandelbrot set, the computation time for the different pixels varies greatly. The performance for embarrassingly parallel computation depends very much on the actual operation executed. We do not use a particularly tuned version for the Mandelbrot calculation here, so this is only an proof of concept. In particular, the proportion between computation and memory access is crucial.

Speedup of up to 7.4 is achieved for $i = 1000$ iterations per pixel with the dynamical load balancing, whereas static load balancing gains a factor of only 4.1, using 8 cores on OPTERON, as shown in figure 4.4. This demonstrates the superiority of the dynamic load balancing for jobs with highly varying running time.

[5] `http://algo2.iti.kit.edu/singler/mcstl/dissertation_patches.tar.gz`

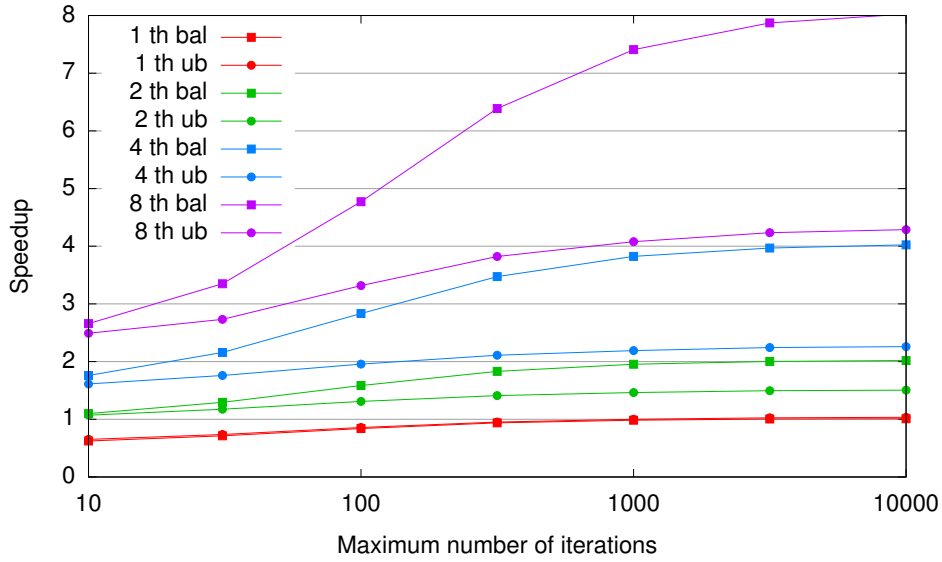


Figure 4.4: Speedup for computing the Mandelbrot fractal with 1 000 000 pixels on OPTERON. bal stands for balanced, ub for unbalanced.

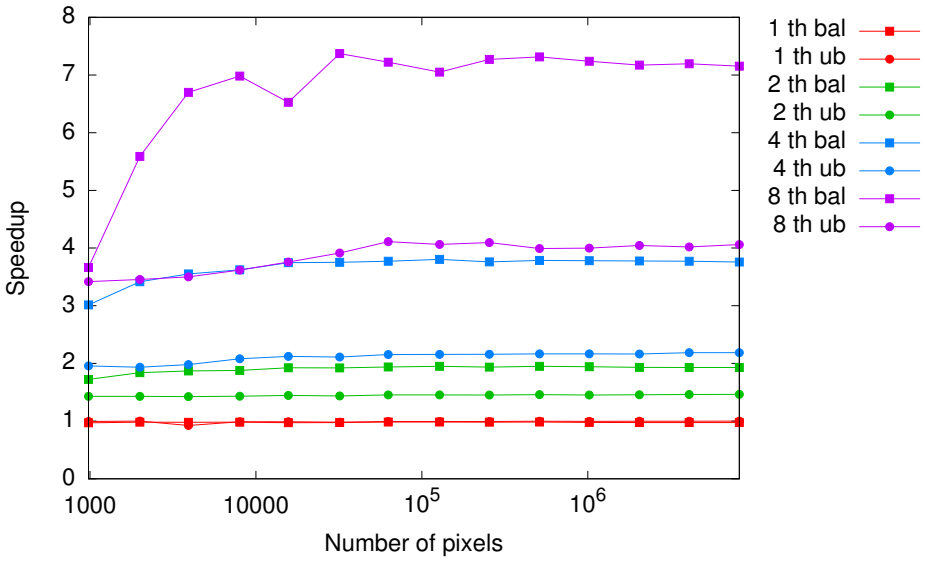


Figure 4.5: Speedup for computing the Mandelbrot fractal with a maximum of 1000 iterations on OPTERON. bal stands for balanced, ub for unbalanced.

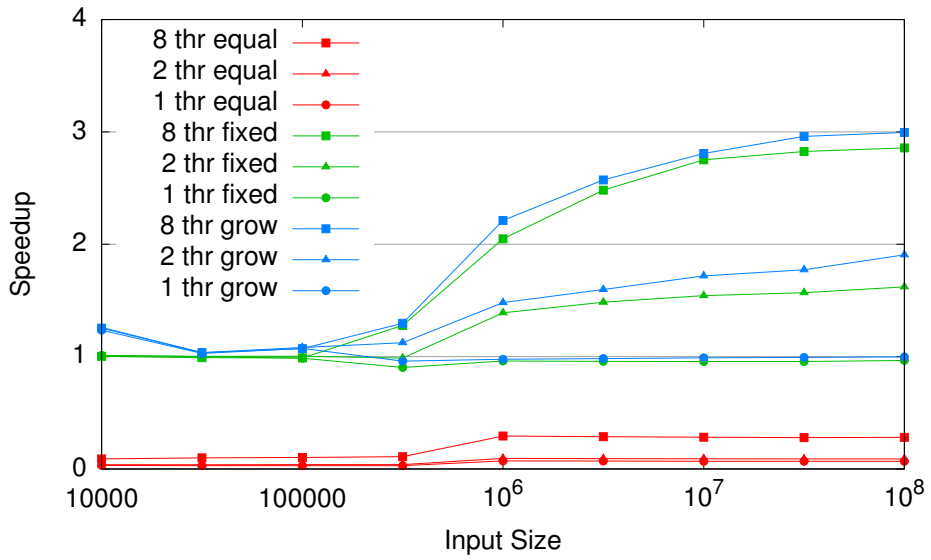


Figure 4.6: Speedup for finding a 32-bit integer at uniformly random position, using different algorithm variants: growing block size (grow), fixed-size blocks (fixed), and naive (equal).

Figure 4.5 shows that already for as little as 4000 pixels, very good speedups are achieved, converging to full speedup quickly for the balanced variants, so overhead is low. Computing about 1 000 000 pixels with at most 1000 iterations takes about 1.7 s sequentially.

The load balancing of the MCSTL `for_each` has also proven very good for parallelizing the preprocessing step of a route planning speedup technique [Hol08].

Find

The naïve parallel implementation of find (using equal splitting) performs very badly and is far from achieving any speedup, as shown in Figure 4.6.

For the block-fetching variants, we chose $m_0 = 100\,000, B = 1000$ as parameters, the latter being the block size for the fixed block size variant. Speedup starts when the input cannot be kept in cache any more, increasing the sequential running time by a factor of 10 from 316 227 to 10^6 elements. The growing block size variant outperforms the fixed-size block variant for all inputs.

The maximum speedup reached is 3, being memory bandwidth bound here. For 10^7 integers as input, in average, the first hit is after 5 000 000 elements. The sequential algorithm takes 0.01 s in this case, equivalent to a bandwidth of 2 GB/s. The parallel algorithms with 8 threads are bound to speedup of at most 3, which is consistent with the memory bandwidth of about 6 GB/s on one socket.

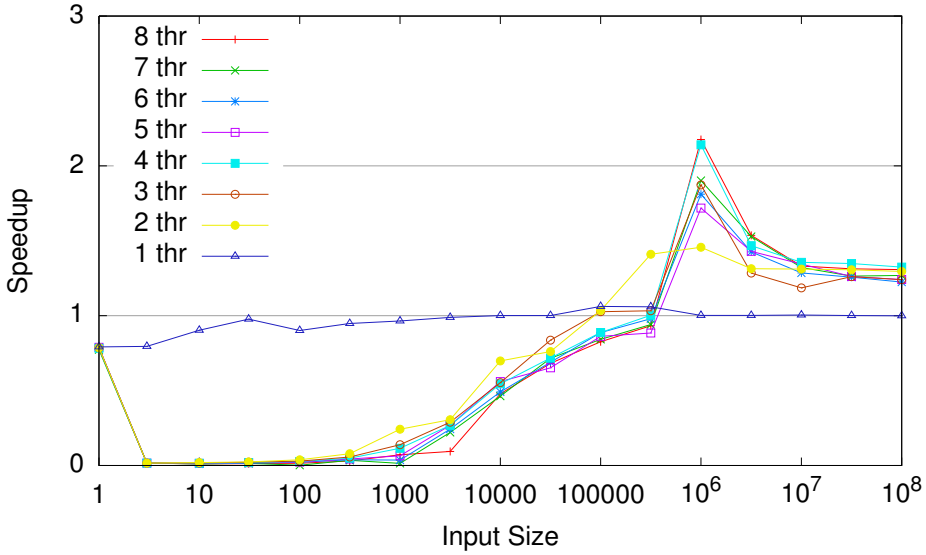


Figure 4.7: Speedup for computing partial sums of 32-bit integers on XEON.

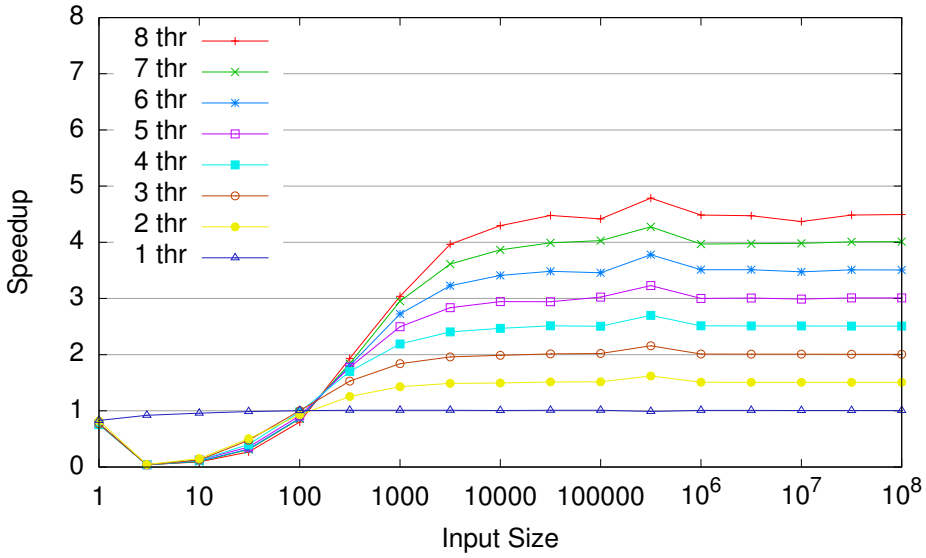


Figure 4.8: Speedup for effectively computing partial products of floats by adding up the logarithms on XEON.

Partial Sum

For `partial_sum`, the picture looks bad summing up just integers, as shown in Figure 4.7. Except for a peak at 10^6 elements, the speedup is about 1.3, no matter what number of threads $p > 1$ are used. Again, this is due to a limitation of the memory bandwidth. 10^7 32-bit integers are summed up in about 0.03 s sequentially. Since for the sequential algorithm, every input element and every result element must be accessed twice, this is equivalent to a bandwidth of 2.7 GB/s. The parallel algorithm needs up to 50% more bandwidth, since most of the input is read twice. Multiplying this with the speedup amounts to 5.2 GB/s, which is almost exactly the memory bandwidth given in Table 3.2 for XEON. The peak superseding in a speedup above 2 can be explained by cache effects. Including the result sequence, we have a total of 8 MB of data, still fitting into the overall cache of 16 MiB. This leads to temporal locality in addition to spatial locality between the first and the second summation step. For the next larger input, we have more than 25 MB, which is too large for the caches. For smaller inputs, the sequential algorithm benefits overproportionally, which prevents speedup in this case. The running time for 316 227 elements is less than a fifth as for 10^6 elements, although the size factor is only about 3. When setting the dilatation factor to 2, the speedup for 10^6 on two threads in fact increases from 1.46 to 1.62, the ideal speedups being 1.5 and 1.66. However, little changes for the all other cases.

The results for OPTERON are even worse, since the a single core can even better exploit the memory bandwidth, processing 10^6 elements in only 0.025 s, and without NUMA-distributed data, the bandwidth is not much higher.

However, if we have a costly operator, e. g. one that multiplies the values by adding up their logarithm, speedup is fully as expected (see Figure 4.8). The peak effect explained above shows up here in an alleviated form, for 316 227 elements of input.

Partition /m$^{\text{th}}$ Element /Partial Sort

Partitioning 32-bit integers gains very good speedup for up to 5 threads, for more threads, the speedup is limited by about 5 on 8 cores, as depicted in Figure 4.9. Again, this is due to bandwidth limitations. Partitioning 10^8 elements takes 0.53 s sequentially, each element is read once and swapped (and thus written) with a 50% chance, resulting in a bandwidth of 1.1 GB/s. Multiplied by the speedup in this case, we get 5.4 GB/s of bandwidth, which is close to the maximum again.

For `nth_element_sort`, we choose a uniformly random splitting position. The algorithm calls `partition` repeatedly for sequences of decreasing length, so its performance is bounded by the one of partition. Given the additional overhead, the speedup is limited by less than 4 on 8 threads. The absolute time for a 10^6 input is 0.011 s sequentially.

For `partial_sort`, we sort up to a uniform random position from 1 to n. The sequential algorithm in libstdc++ provides very poor performance for inputs larger than the cache, because of the bad cache-efficiency of the heapsort used. Thus, to avoid excessive numbers, we exceptionally give speedups relative to the 1-thread parallel version here, which can process 10^6 elements in 0.064 s on average. The original sequential version takes acceptable 0.17 s for 10^6 elements, but already excessive 21 s for only 31 times the number of elements.

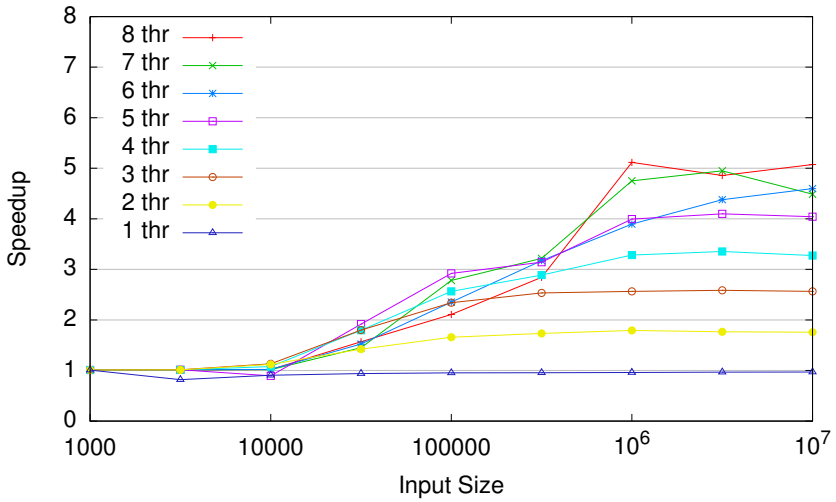


Figure 4.9: Speedup for partitioning 32-bit integers on OPTERON.

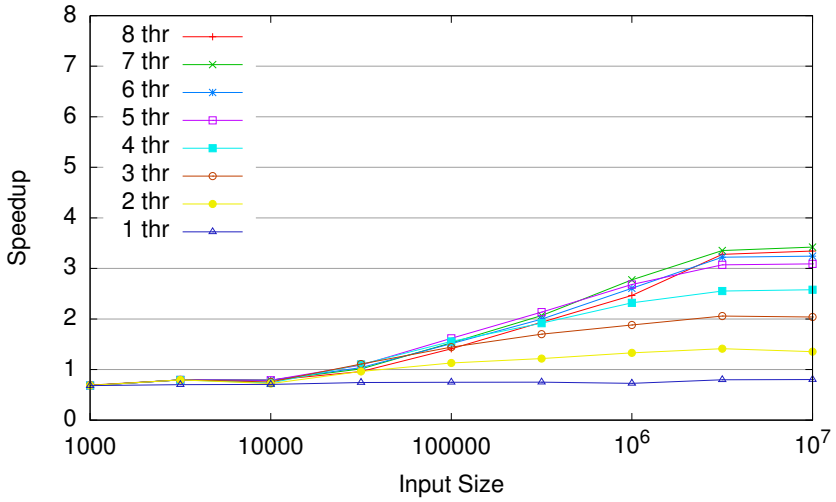


Figure 4.10: Speedup for `nth_element` for 32-bit integers on OPTERON.

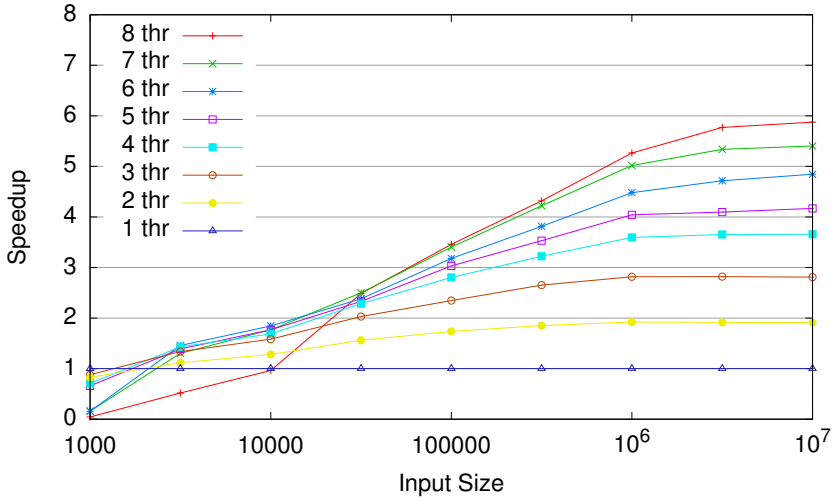


Figure 4.11: Relative speedup for `partial_sort` for 32-bit integers on OPTERON.

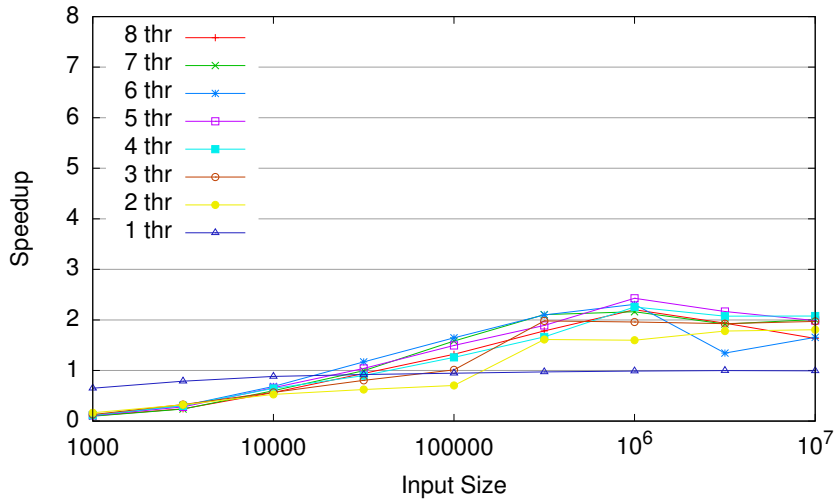


Figure 4.12: Relative speedup for 2-way merging 32-bit integers on OPTERON.

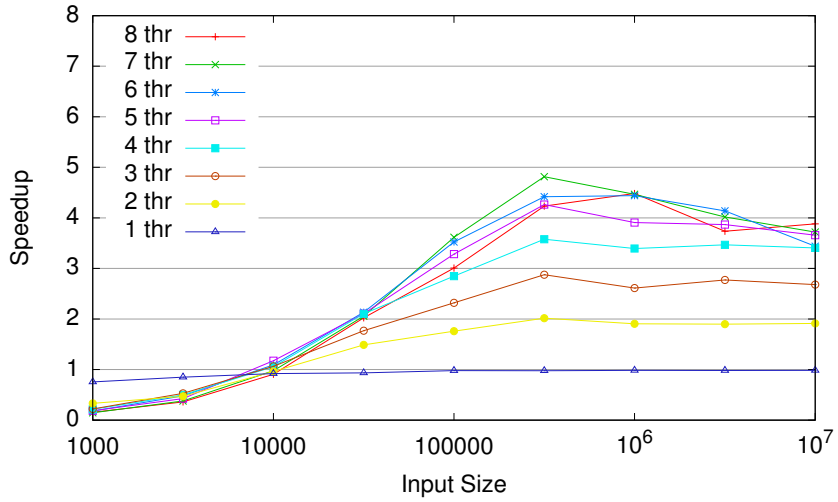


Figure 4.13: Relative speedup for 4-way merging 32-bit integers on OPTERON.

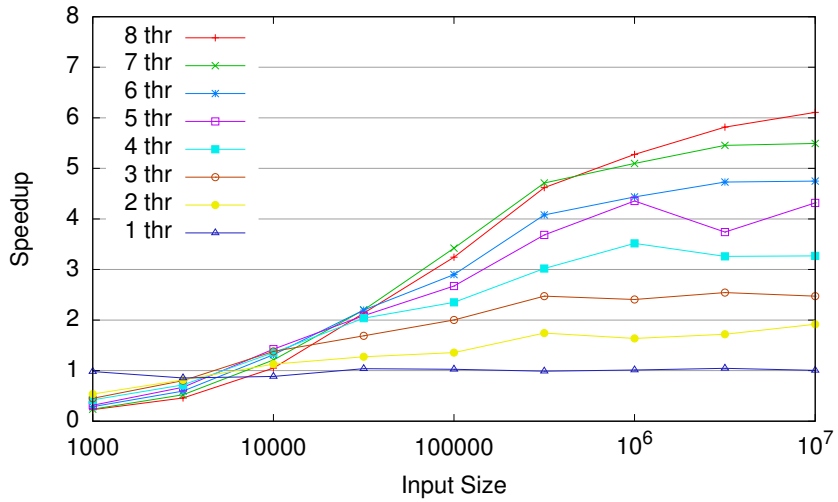


Figure 4.14: Relative speedup for 16-way merging 32-bit integers on OPTERON.

(Multiway) Merge

We have tested the multiway merging for different numbers of input sequences $k = 2, 4, 16$.

The sequential running time for 10^6 32-bit integers increases from 3.7 ms to 6.8 ms to 23 ms for the mentioned values of k. This grows much more steeply than the $O(\log k)$ asymptotic running time expected. For the smallest k for which we use a loser tree, $k = 5$, the execution takes 19 ms, so this where the sudden running time increase happens.

The speedup for parallelization behaves conversely. For $k = 2$, the sequential algorithm already uses a bandwidth of 2.1 GB/s for 10^6 elements. This bounds the speedup by 2.4 (see Figure 4.12, reaching the memory bandwidth wall. The speedups double for $k = 4$ (see Figure 4.13), which is not surprising given the sequential running time increase of about a factor of 2. For both cases, we see additional speedup for the cases where a considerable part of the input fits into cache, for up to 10^6 elements. This reduces bandwidth usage on the first socket, however, much smaller inputs still not compensate for the parallelization overhead. Finally, for $k = 16$, the computation dominates, resulting in speedups up to, as shown in Figure 4.14.

Sort

On OPTERON, speedups are excellent for 32-bit integers (Figure 4.15), for up to four threads, speedup is almost linear, going up to more than 6 for 8 threads. When the computation per element size (and therefore bandwidth) is less, as for the pair of 64-bit integers (Figure 4.16), taking the first value as the key, the speedup is worse. This again underlines that memory bandwidth limits processing speed here, although we profit from the NUMA architecture, because each thread copies the data to be locally sorted to a locally allcoated space first.

Speedup greater than 1 starts from 1000 elements, which executes sequentially in about 50 microseconds.

The detailed timing breakdown shown in Figures 4.17 and 4.18 reveals interesting insights. For 1000 elements, the exact splitting takes up to 40% for 8 threads. This looks like a lot, but we have to realize that this means only 125 elements per thread. Initializing the threads takes about 1 microsecond per thread, the local sort is actually still sped up despite the small size. For 100 000 elements, things look very different, the parallelization overheads (initializing the threads, splitting, waiting) become insignificant. However, on the first glance, it looks like there is hardly any speedup for the multiway merging step. We have to keep in mind though that the cost for merging increases with the number of sequences, i. e. the number of threads. In theory, this is only logarithmic, but in practice, the asymptotics are not precise for this small k. Due to switching to a loser tree, we have a large running time increase from 4 to 5 sequences, which also appears here. The parallelization of the merge can just compensate for the more difficult work, but not speed it up absolutely. On the other hand, not parallelizing the merging would slow down things significantly. This is consistent with the experiments on multiway merging alone as discussed before. In fact, we have a little running time decrease from 5 to 8 threads, which we expect to extrapolate to larger p, since the cost for merging will grow sublinearly. Here, we see the artifacts of particularly optimizing for up to 4 sequences, in our effort to provide best performance also on a small number of cores.

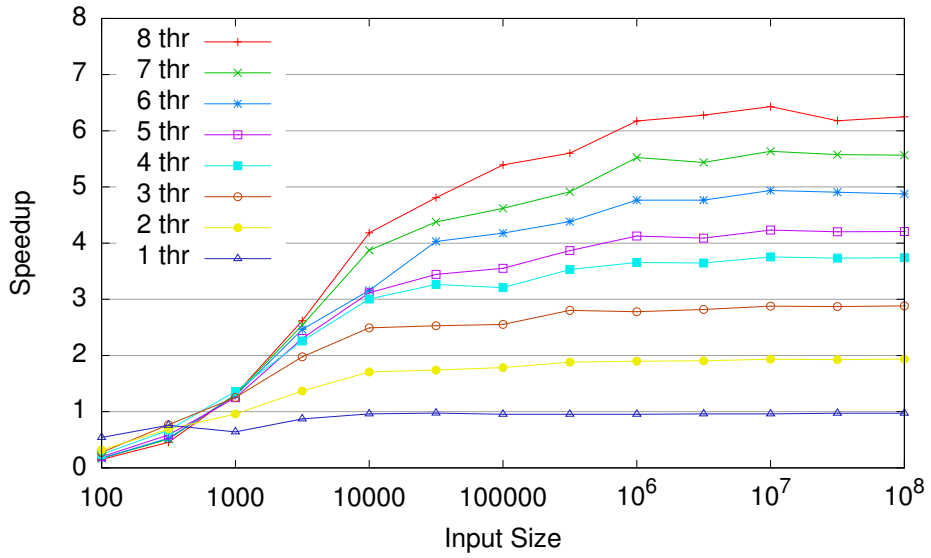


Figure 4.15: Speedup for multiway mergesort on 32-bit integers on OPTERON.

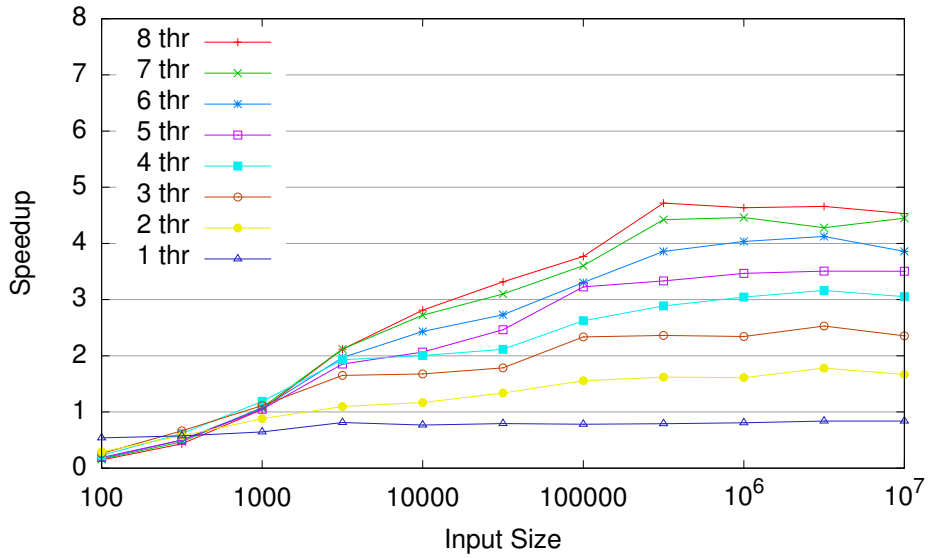


Figure 4.16: Speedup for sorting pairs of 64-bit integers on OPTERON.

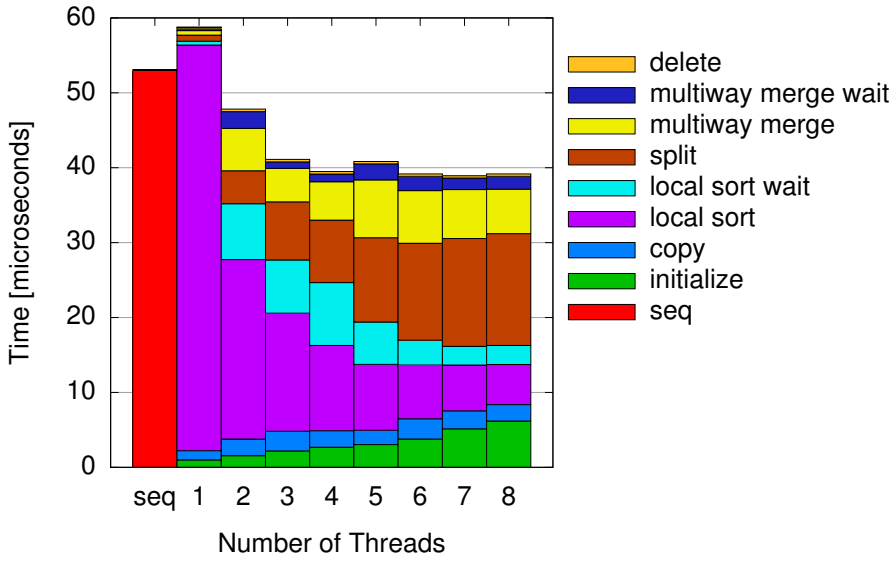


Figure 4.17: Detailed timing breakdown for for multiway mergesort on 1000 32-bit integers on OPTERON.

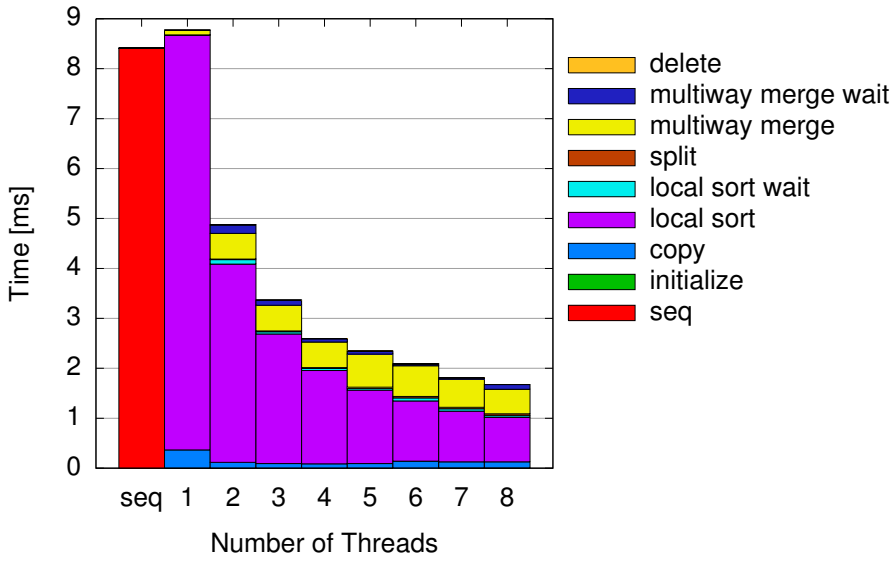


Figure 4.18: Detailed timing breakdown for for multiway mergesort on 100 000 32-bit integers on OPTERON.

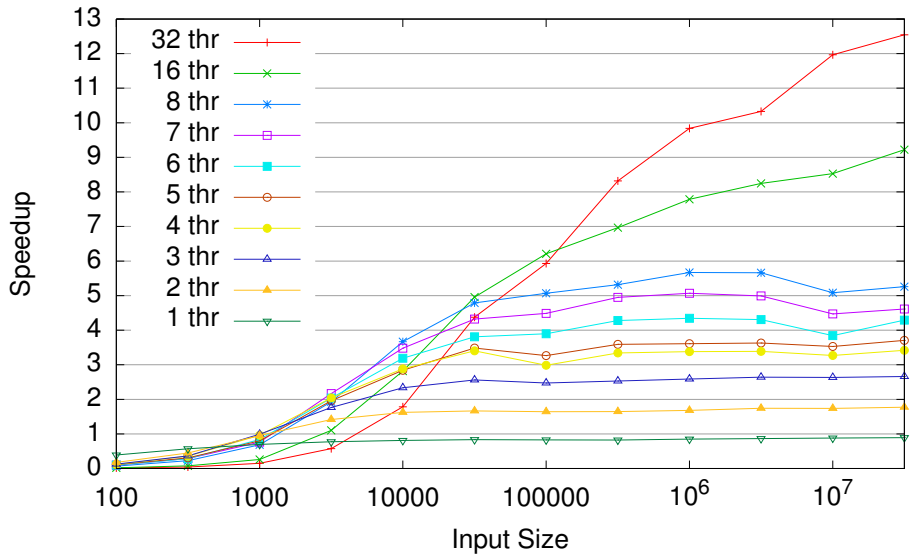


Figure 4.19: Speedup for sorting pairs of 64-bit integers on SUN T1.

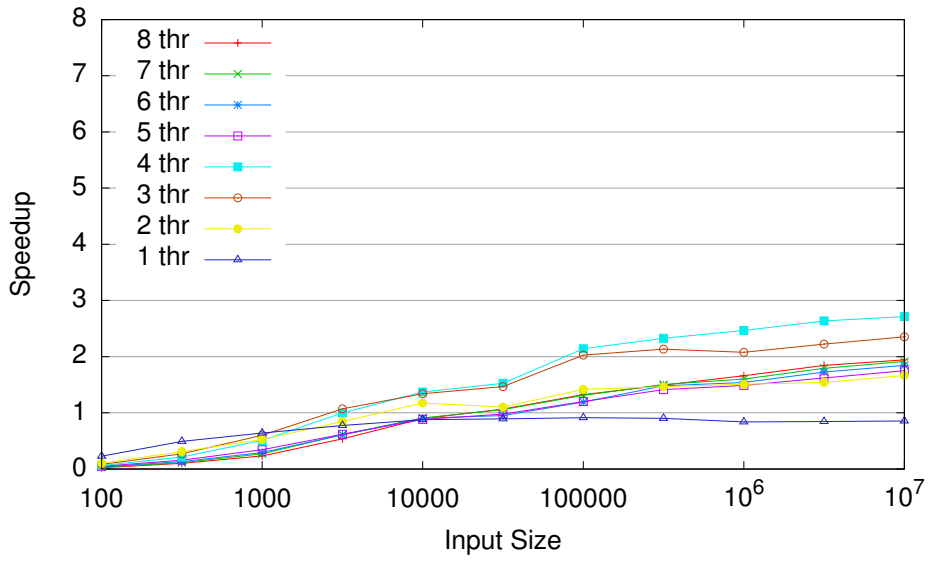


Figure 4.20: Speedup for multiway mergesort on 32-bit integers on OPTERON, the sequence containing all equal elements, and using sampling instead of exact splitting.

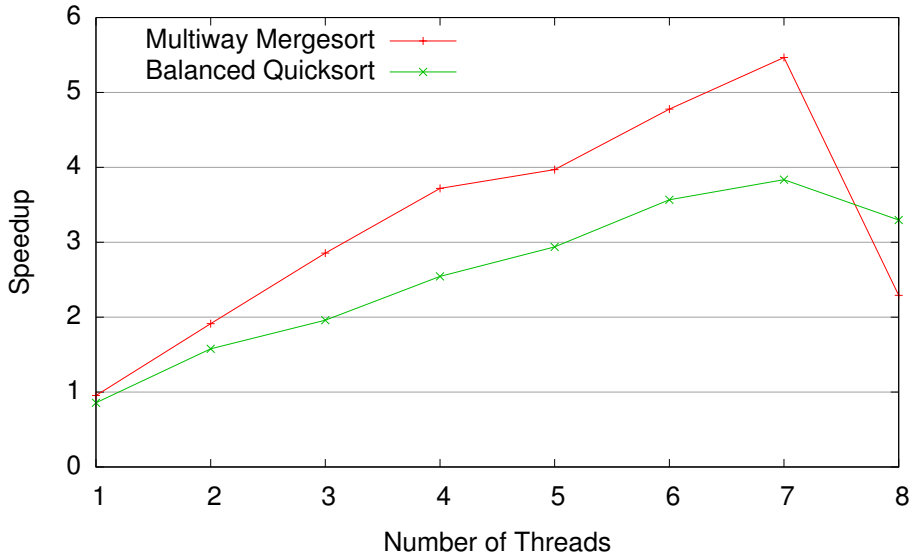


Figure 4.21: Speedup for sorting 10^7 32-bit integers with different algorithms, while one core is permanently blocked by another program.

Input sizes larger than 100 000 elements show very similar breakdowns, except that the local sorting takes more time relatively, so the overall speedup increases a little further.

Instead of exactly splitting the sequences for the merge step, we could also use sampling, i. e. select a sample of elements, sort them, and split the sequences by doing binary search. However, in the worst case where all elements are equal, the performance for splitting by samples is erratic, and the speedups for 8 threads shown in Figure 4.20 are much worse than 4.7 as for exact splitting (taking a much shorter sequential running time into account for both cases). This demonstrates the superior robustness of exact splitting. For large random data, random sampling is typically only 5% slower.

We also examine the usefulness of dynamic load balancing for sorting. We sort while another program blocks one core completely, running at a slightly higher priority. Figure 4.21 shows the performance of the two sort algorithms for a large input (10^7 integers). Balanced quicksort is worse than multiway mergesort for up to 7 threads. For the overloaded 8-thread case, however, the speedup for mergesort halves, due to one core having to effectively process two parts. The quicksorts remains almost as good as for 7 threads, proving the benefits of dynamic load-balancing.

On the SUN T1, the multiway mergesort achieves speedup of close to 13 on with 32 threads on 8 cores, for pairs of 64-bit integers (see Figure 4.19). This is quite impressive, the multi-threading is utilized extensively. With as many threads as cores, speedup up to 5.7 is possible. However, in absolute timings, SUN T1 is more than an order of magnitude slower than OPTERON, sorting 10^6 of these elements in 1.54 s instead of 0.12 s.

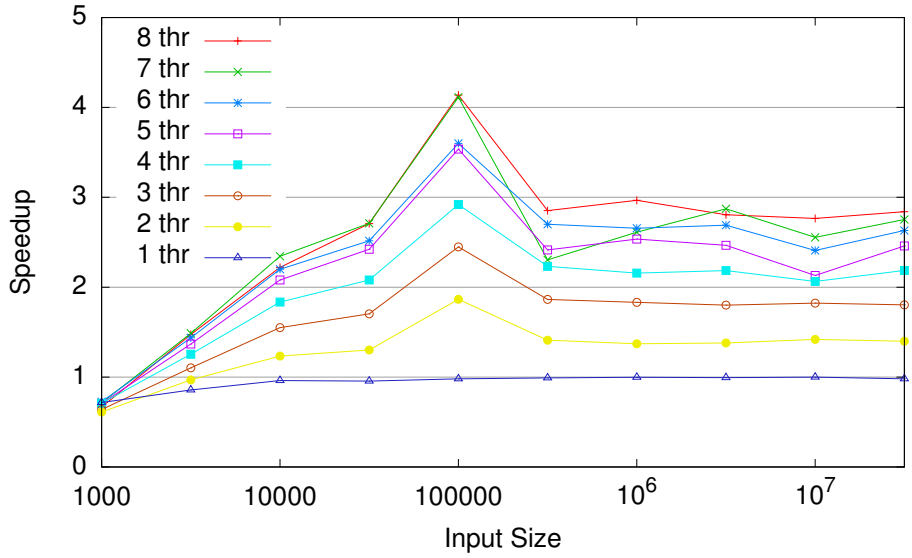


Figure 4.22: Speedup for set union on 32-bit integers on OPTERON. The input size refers to the length of each of the two sequences.

Set Operations

Exemplary for the set operations, the speedup for `set_union` is shown in Figure 4.22. Each element is read twice, except for the $n/(p+1)$ first ones, which are only read once. All elements are written once, except the duplicates. Since the integers are chosen randomly from $[0,\ldots,10n)$, we have about $1.9n$ elements in the output. So for $p = 8$, this results in about $3.79n$ memory accesses. Taking 4.2 ms for 10^6 32-bit integers per sequences, this is equivalent to an average bandwidth of 3.62 GB/s. Thus, we cannot fully saturate memory bandwidth here. A dilation factor as in `partial_sum` could help by reducing the imbalance, giving a smaller piece to the first thread in the first step, where it also has to *write* data.

Random Shuffle

The performance for `random_shuffle`, shown in Figure 4.23, profits from the cache-aware implementation that makes the sequential algorithm already more twice as fast as the standard one. The speedup continues to scale with the number of threads on for inputs exceeding the cache.

The sequential running time is significantly non-linear. 10^5 elements take 0.0067 s, rising to 21.6 s for 10^8 elements, which is a factor 3 232 for an input size increase factor of 1 000. Note that this is actually slower than sorting the same amount of data. With our algorithm, we can alleviate this bad behavior.

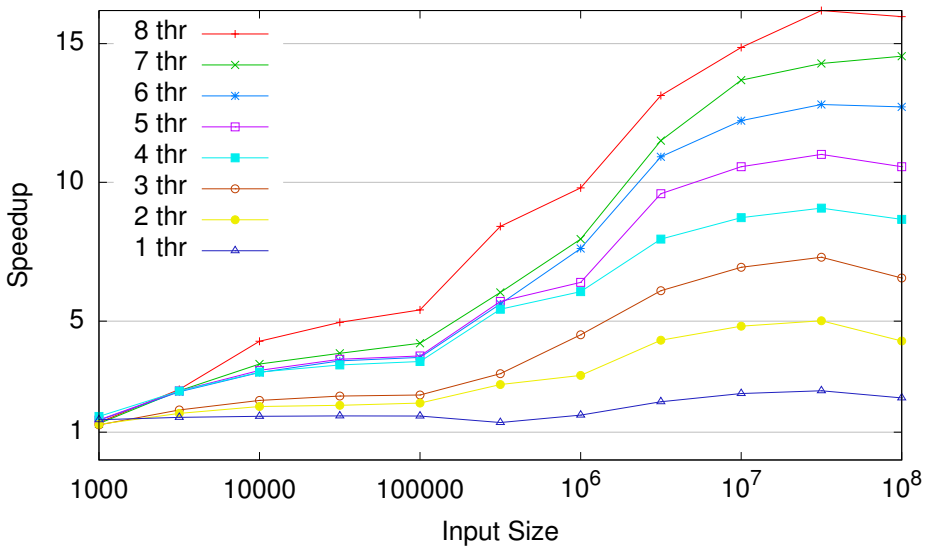


Figure 4.23: Speedup for Random shuffling 64-bit integers on OPTERON.

4.3 Dictionary Bulk Construction and Insertion

Data structures and algorithms are usually closely related. However, many algorithms rely only on very general properties of the data structure they are applied to. For example, most algorithms treated so far work on sequences of elements that support efficient random-access, i. e. as a constant time operation.

Data structures themselves cannot be parallelized in the sense of doing computational work in parallel, since a data structure itself is basically just a collection of objects. “Parallelizing a Data Structure” could be understood as making access to it thread-safe, but this is not a concern of this thesis.

Rather, we treat in this chapter algorithms that are used to manipulate a very specific data structure. Usually, these operations are described with the data structure and form the vital part of its invention. They also characterize the data structures in terms of running time efficiency.

Usually, data structures contain many objects of the same type, or a small number of types. Most basic operations process a single object in combination with the data structure, e. g. inserting it, removing it, finding an object with certain desired properties. To avoid quadratic processing time for n objects to process, the data structure operations must have a running time sublinear in n, i. e. usually constant or logarithmic. Since the constant factors are also quite low, obtaining speedup from parallelizing a single operation is hard. However, there are two ways to circumvent this problem. We focus on combining many such operations, and provide such a *bulk operation* as a new interface to the user.

The generic STL types `set` and `map` implement sorted sequences. This includes the functionality of a dictionary, either for self-contained elements, or for mappings of keys to values, also known as associative arrays. As motivated above, we consider here the parallelization of

two bulk operations, namely the construction from many elements, and the insertion of many elements in a single operation. Although the latter subsumes the former, as we can insert into an empty dictionary, particular optimizations are possible for construction. The routines are useful when bursts of data arrive every once in a while, or a dictionary has to be initialized from a large data set.

We also consider the variants `multiset` and `multimap`, which allow multiple equal elements to be contained, which introduces some subtle differences.

Algorithm

We implement our bulk operations on top of the existing data structure implementation of the libstdc++, which is based on red-black trees [GS78]. The nodes of a red-black tree are colored either read or black. Both children of a red node are black, and every path from a node to one of its descendant leaves contains the same number of black nodes. By maintaining these invariants, the maximum depth and therefore the running time of the basic operations can be kept logarithmic in the total number of contained elements.

Other operations than the bulk insertion and construction stay unaffected, as does the data structures itself. Our partitioning of the work is flexible, e. g. multiple threads can work on a relatively small part of the tree.

Instead of regarding bulk construction as a special case of bulk insertion, we do it the other way around. Bulk construction (of a subtree) is the base case for bulk insertion, i. e. when many elements have to be inserted between two already contained elements. This gives good performance even for degenerate inputs.

In any case, we stably sort the input elements first. For the unique dictionary variants, duplicates are removed. Both these steps are done in parallel.

For the construction of a new (sub)tree, the elements are divided into parts of almost equal size. Each thread allocates and initializes the nodes for its elements. Then, the color is set and the nodes are linked among each other. Since the nodes are referenced by an array of pointers, the tree is perfectly balanced (except the last level), all computation can happen independently in parallel.

Insertion into an existing tree is much more involved. First, the tree is split into p subtrees, the smallest elements of each thread acting as splitters. In the second phase, each thread inserts its elements into its subtree in parallel, using an advanced sequential bulk insertion algorithm. In the last phase, which overlaps with the second phase, each thread that is finished inserting, joins its new subtree with another subtree recursively, if that one is already finished as well.

We assume that the input is much larger than the number of cores, $n > p^2$. Then, the construction of a tree from k elements takes time $O(k/p)$, and the insertion of k elements takes time $O(k \log n/p)$, both of which is optimal. To also keep constant factors low, we dynamically balance the load using randomized work stealing, similar to the quicksort algorithm presented in Section 4.2.

Related Work. We are aware of parallel red-black tree algorithms only for the PRAM model [PP01]. They are highly theoretical, use fine-grained pipelining etc., and are thus not suitable for real-world machines.

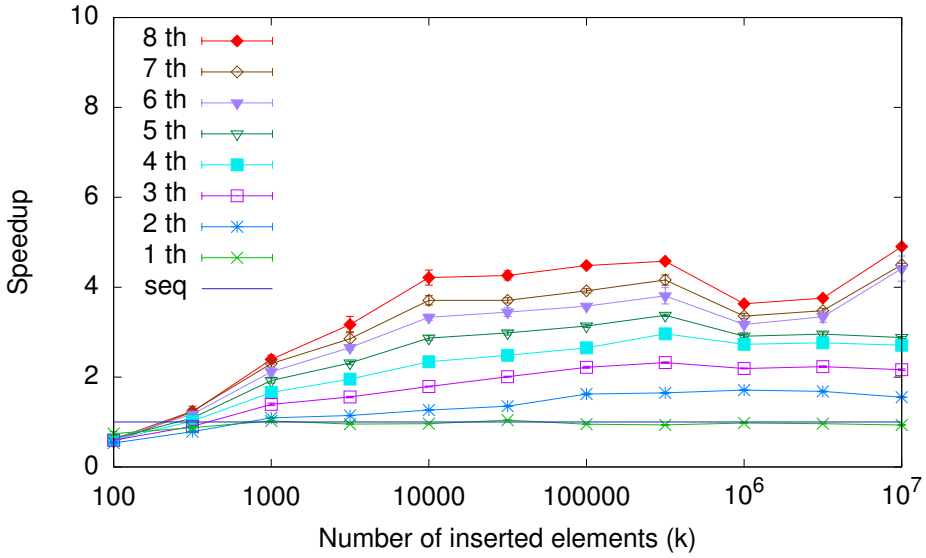


Figure 4.24: Speedup for constructing a set of integers on XEON.

The STAPL [AJR+01] library does provide a parallel tree implementation. However, its quite rigid partitioning of the tree for distributed memory can lead to all the elements being inserted by one processor, in the worst case.

Memory Management

In contrast to most of the algorithms described so far, the bulk dictionary routines have to actually allocate memory, namely the tree nodes. This cannot be done in a large chunk, since the nodes should be discardable individually. Thus, the threads have to allocate many nodes individually, and also concurrently. Allocation is handled by the C++ runtime library, and for bad implementations, the contention of the memory manager lock could inhibit speedup completely. However, for the tests we conducted, the performance penalty was very acceptable.

Experimental Results and Conclusion

This time, the tests were compiled using the Intel compiler in version 10.0.25. We ran at least 30 instances and present the average values in the plots, accompanied by the standard deviation (too small to be seen in most cases). We chose 32 bit integers as element type, and presorted the randomly generated input sequences, since parallel sorting is a separate issue. The parameter r represents the ratio between the number of already contained elements n and the number of elements to be inserted, $r = n/k$. Thus, $r = 0$ is equivalent to bulk construction.

The original bulk insertion routines just inserts one element after the other. At least, it performs a finger search using the position of the last inserted element as a hint. This helps when the input is presorted, as is the case here. Still, our specialized routine beats the original by almost a factor of two even when run sequentially (see Figures 4.24 and 4.25).

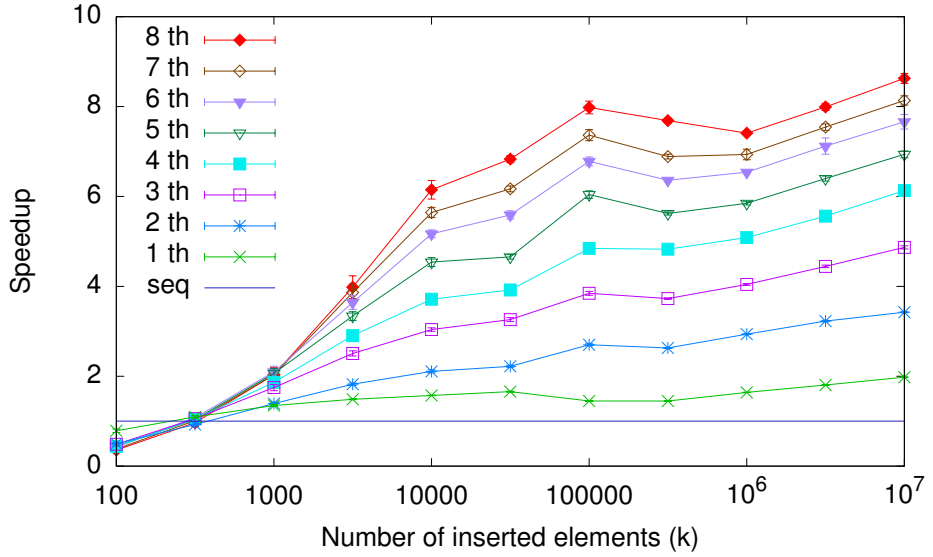


Figure 4.25: Speedup for inserting integers into a set ($r = 0.1$) on XEON.

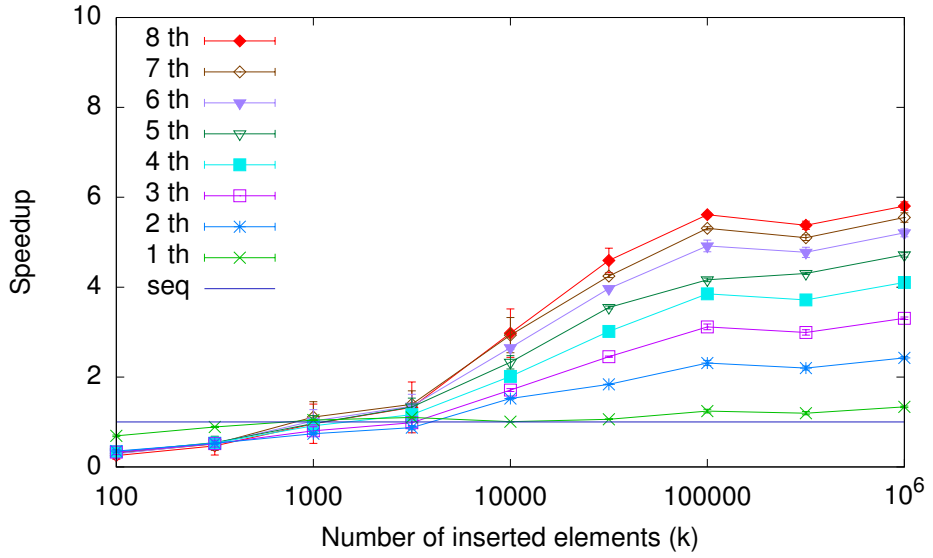


Figure 4.26: Speedup for inserting integers into a set ($r = 10$) on XEON.

Parallel execution, our original aim, brings further speedup. For construction, as seen in Figure 4.24, the speedup goes up to almost 5 on 8 cores. For insertion, our algorithm is most effective when the existing tree is smaller than the input data to insert, reaching an overall acceleration of more than 8 (please compare Figures 4.25 and 4.26). In all cases, speedups of at least 3 for 4 threads and 5 for 8 threads are achieved. The break-even is reached for as few as 1000 elements.

Conclusion

We have devised an effective multi-core parallelization for constructing and insertion into dictionaries, based on red-black trees. The ordinary sequential operations can be used as well, no change to the data structure itself was necessary.

To also speed up programs that usually do not insert in large chunks, we could use lazy updating: Collect insertions as long as there is no query, and insert them in a bulk fashion when a query arrives.

This content of this section was developed in close collaboration with Leonor Frias, and is also part of her PhD thesis. Therefore, this is just a short mentioning, details on all aspects can be found in [FS07].

4.4 Software Engineering

In this section, we describe the goals, the design choices and the software engineering issues discovered during design and implementation of the MCSTL. We address ease of use, standards compliance, algorithm and parameter choices at compile and run time, and the interplay between execution speed, executable size and compilation time. For some of the issues, the next version of the C++ standard, called C++0x, will improve the situation greatly. It is expected to finally appear in 2011, however, the current draft [C++10] is said to be very close to the final standard.

Major Design Goals

The major design goal of the MCSTL is to provide parallelism with as little effort as possible for the developer.

The interface should comply to the C++ Standard Template Library. This is easily fulfilled in terms of syntax. However, parallelism needs some additional requirements when it comes to semantics. For example, the user-defined functors called in algorithms like `for_each` must have no interfering side-effects. Also, the order of execution cannot be guaranteed any more for operations which have that property by the standard, e. g. `for_each`. This is because the standard was designed for serial implementation, without having parallel processing in mind at all.

The algorithm descriptions often do not specify only a input/output relation, but instead provides code examples for which the operation is “equivalent.” This implies the order in which the elements are processed (say in `for_each`), and thus ultimately forbids parallelization when taken strictly.

Furthermore, programs might rely on properties of algorithms that are not guaranteed by the standard. For example, for the usual sequential `merge` implementation, both input sequences can overlap without harming correctness. However, this is fatal for the parallel algorithm.

Ultimately, we have to strive for a pragmatic balance between adherence to the C++ standard and parallelization/software engineering benefits. We look at the functions names here, and provide semantics that are in the their spirit, omitting specifications that were introduced often just for "simplicity", like the ordering.

Further usability objectives for the MCSTL include:

1. Transparent integration of parallel algorithms, i. e. compile in normal mode or in parallel mode, without changes to the user code.

2. Possibility to tune algorithms and choose variants, and to adjust the degree of parallelism, in order to achieve best performance.

3. Limited increase in compilation time and executable size.

Portability. Many parallel algorithms that are published show excellent scalability results. However, the experiments are usually either limited to a platform, a specific data type as input, and/or assumptions on the input data type. The code is hard to re-use since it is overfitted to a specific machine and particular aims.

In contrast to this, a library implementation of a parallel algorithms must be generic. First of all, the algorithms need be parameterizable by input and output data type. The data type carries implicit semantics, e. g. the comparison operator might be redefined, or the assignment operator and the copy constructor. This latter prevents the use of `memcpy` for an optimized copying of data.

For some algorithms, executing user-defined functors is vital. The running time of those may be arbitrarily distributed. For some operations, e. g. prefix sum and accumulation, we assume the execution time to be uniform, while for `for_each`, we try to handle the most general case, i. e. no assumptions about running time.

A library should be mostly platform-independent. Thus, it can never be optimized as much as an implementation designed specifically for a certain platform. We rely on the compiler for efficient code, not using assembler or intrinsics, except the atomic operations.

Usage of the Parallel Mode

There are two options for enabling parallelism:

1. Activate the parallel mode by default, i. e. all STL algorithms that have parallel implementations available, will have parallel code included into the program.

2. Activate the parallel mode on request, i. e. specific algorithm calls can be annotated by the developer to call the parallel version of the algorithm.

Using the first option is extremely simple, just add a define to the compiler command line, and enable OpenMP support: `g++ -fopenmp -D_GLIBCXX_PARALLEL program.cpp`. The library will then use default values for the minimum input size to execute an algorithm in parallel. However, these defaults are chosen conservatively, assuming simple data types and cheap operations, thus biased toward avoiding slowdown instead of maximum speedup. If the user wants to customize the values for maximum performance in any setting, he can do so at little effort. Also, the number of threads used can be easily specified by standard OpenMP procedures. The algorithm alternatives and further tuning parameters are also accessible to the user via a global settings data structure. E. g. for the embarrassingly parallel functions, the user can deactivate dynamic load balancing for easily splittable problems on a dedicated machine.

For choosing the second option, the developer prepends the namespace qualifier `__gnu_parallel::` to his call, explicitly switching to the parallel version.

If an algorithm is to be executed sequentially in any case, even if parallelism is switched on by default, the developer specifies this by adding `__gnu_parallel::sequential_tag()` to the end of the call. The original STL version will then be called without any runtime overhead since the decision is made at compile-time through function overloading.

Also, the parallel mode provides a variable to globally force (at runtime) parallel and sequential execution respectively, independent of the problem size, but still respecting a `sequential_tag`.

Sequence Access through Iterators

Most STL algorithms take one or more sequences as their main argument(s). To abstract from the underlying representation, the iterator concept is used. A sequence is usually specified by passing an iterator referencing the first element `begin()`, and an iterator referencing a virtual element behind the last element `end()`. This allows a consistent treatment of arrays, regarding pointers as iterators, and using pointer arithmetic. Also, different iterator types or instantiations can access a single sequence in different ways, e. g. selecting a range, advancing with a stride, or going backwards. So on the one hand side, this adds a lot of flexibility.

On the other hand side, information gets lost when passing iterators. The best example for that is the length of a linked listed. A linked list container might implement the size query efficiently, just by keeping track of the number of inserted and deleted elements. When an algorithm is passed the corresponding `begin` and `end` iterators, however, this information is lost, since the iterators do not include a reference to the underlying container. Since the iterators are not random-access iterators, the distance between them cannot be computed efficiently. The only way to find out the length of the sequence is to traverse the whole sequence, taking sequential linear time, impossible to parallelize.

Since all parallel mode algorithms are *data-parallel*, the input must be split into parts, the splitting positions being scattered along the whole sequence. This can be done in sublinear time only if random access to the elements is granted. Hence, parallel execution seems to be limited to random-accessible sequences on the first glance. However, we will give a method on how to workaround this problem with using as little sequential computation as possible, in Section 4.5.

The decision on whether random access is possible happens at compile time, using partial template specialization. For each used iterator, an appropriate specialization of

`std::iterator_traits` must be available, as requested for all iterators by the standard [C++03, Section 24.3.3]. If no random access is possible, and the workaround is not implemented for this algorithm, the sequential version is called.

Code Reuse

Parallel algorithms are usually far more complicated than their sequential counterparts. For example, the one-liner describing `for_each` explodes into a parallel version of many hundred lines, implementing complex load balancing mechanisms like work-stealing.

On the one hand, this fact makes the usage of a library even more valuable, since the developer does not have to write such complicated code himself. On the other hand, the library itself is threatened by hard-to-maintain complexity. For this reason, the library developers must make efforts to keep the code as maintainable as possible, without diminishing performance. Two important aspects in that are *fostering code reuse*, and *avoiding code duplication*.

Code duplication is particularly tempting for the STL, since it contains many algorithm calls that act similarly. `for_each` and (unary) `transform`, for example, only differ in the fact that the functor passed to `transform` is allowed (even supposed) to modify its input in-place. There are several other similar calls being "embarrassingly parallel", traversing one or two input sequences. One could write the code for `for_each`, copy and paste it several times, and change some little details. This is feasible for the sequential versions, but unmaintainable for the parallel ones. A bug fix or an algorithmic improvement would have to be applied in many places, making the process extremely error-prone.

In particular the algorithmic portions must be reused already by the library itself. In fact, for a family of algorithms, a most general version of the parallel algorithm is factored out, and subsequently called by all members of the family. The surplus arguments must be filled with dummies, e. g. noop-functors and dummy iterators. Similar action is taken for default functors. For example, the sequential implementation provides two completely independent implementations for both versions of `accumulate`, one having explicitly specified the addition functor, the other taking the default `operator+` for the respective type. For the complex parallel version, the implicit version must be redirected to the explicit one by filling in an equivalent functor as early as possible (see Figure 4.28 for an example on the compare functor of `sort`). We rely on the ability of the compiler to inline simple operators, and to optimize away empty functions. Otherwise, the performance penalty will be considerable. Compiling without optimizations will therefore impede the parallel implementation to a greater extent than the sequential one.

Apart from consolidating the code for families of functions, there is also extensive reuse between substantially different algorithms, calling each other as subroutine. Figure 4.27 shows the dependencies.

Parallelism Helper Functions

There are some helper algorithms in the parallel mode that can be very useful for the developer in parallelizing routines himself, in the case that the library functions are really not sufficient.

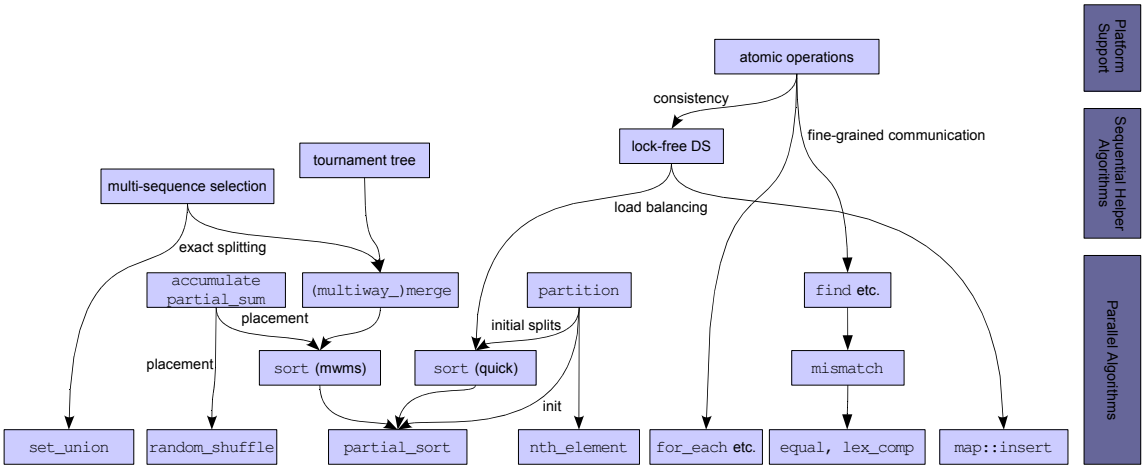


Figure 4.27: Schema of code reuse inside the parallel mode.

This includes the simple, but error-prone function for splitting a range into parts of almost equal size. Much more complex is the `multiseq_partition`. Both routines are used intensively by the parallel mode internals anyways: it is only natural to make them accessible to the developer.

Algorithm Variants

For some functions, the parallel mode offers multiple algorithmic variants, in order to allow best execution speed in a variety of circumstances. Particularly sophisticated are the three variants for sorting. This may sound like a lot, but note that the sequential implementation of `stable_sort` also has two completely independent versions.

The parallel mode features two kinds of parallel quicksort, unbalanced and balanced, as well as a the multiway mergesort. The quicksort variants work completely in-place, on the downside, they do not support stable sort.

Multiway mergesort does provide stability, and is very static in its usage of threads, minimizing overhead. Experiments show that it scales better down in terms of running time, i. e. for small inputs. However, it requires a temporary copy of the data.

Code and Binary Size, Compilation Time

Generic programming in C++ trades off a small executable for optimal adaptation to and integration of the specified data type, resulting in best performance. Instantiating an algorithm for each data type separately allows high-level programming without any run-time overhead, which would be implied by the use of virtual functions or similar run-time decision-making. If code size is the most important factor for a particular use-case, the developer can still derive all used data types from a base class with virtualized functions and instantiate the algorithm for the base class. So we apply the zero-overhead principle here ourselves.

```
#include <vector>
#include <algorithm>
int main() {
  std::vector<int> vi(100000000);
  std::sort(vi.begin(),vi.end() TAG); return 0;}

sort(begin,end)
 sort(begin,end,comp=std::less<int>())
  parallel_sort(begin,end,comp,stable=false)
   1. parallel_sort_mwms(...,stable);
   2. parallel_sort_qs(...);
   3. parallel_sort_qsb(...);
    respective OpenMP-parallelized code
```

Figure 4.28: Minimal sorting code example (above) and call stack (below) for "runtime variant choice", i. e. `TAG=""` with the three possible choices at runtime.

For the parallel mode implementation in particular, the algorithms are more complex and multiplied by variants, and thus the problem gets more important. As a result, the code size for the algorithms can grow up to a factor of 4 compared to the sequential code. This is shown in Table 4.2, for the program in Figure 4.28:

For `TAG`, the corresponding tags are inserted, in order to choose the algorithm variant. The code was compiled using the options `-O3 -g0 -fopenmp -D_GLIBCXX_PARALLEL -DNDEBUG` with a prerelease version of GCC 4.3.

The compilation time also differs, it approximately scales with the size of the resulting executable. This is mostly due to the optimization passes, not the parsing, as can be seen by comparing to the timings for optimization switched off.

As a conclusion from these measurements, it can be stated that including all variants of an algorithm for a run-time choice is best avoided. Instead, the developer should be able to choose a specific algorithm variant at compile time, and the default should include only the parallel algorithm forming the best choice.

This functionality is provided by the parallel mode through tag objects similar to `sequential_tag`.

Detailed C++ Standard Compliance

Functors. The library calls take many kinds of encapsulated user code as arguments.

- Explicit functors to process/transform an element, making up the main functionality of the algorithm. The functor object might contain "state" information. Examples: functors for `for_each`
- Implicit methods and functions, inherent to the value type. Examples: Overloaded (converting) constructors and assignment operators

Algorithm Variant(s)	Executable Size	Compile Time
Sequential	15 479	0.74
Quicksort	22 387	1.49
Balanced Quicksort	26 989	1.84
Multiway-Mergesort Sampling	36 002	3.49
Default (Multiway-Mergesort Exact)	41 229	4.68
Multiway-Mergesort Exact	41 237	4.78
Multiway-Mergesort (splitting choice at run-time)	46 003	5.48
All Parallel Variants (run-time choice)	61 543	6.50

Table 4.2: Executable sizes (in bytes) and compilation time (in seconds) for different algorithm variants.

- Implicit methods and functions, inherent to the iterator type. Examples: Increment operator (`operator++`), dereference operator (`operator*`), difference operator (`operator-`), default constructor
- Stateful helper functions. Example: random number generator for `random_shuffle`.
- (Hopefully) stateless helper functions. Example: Comparison functors, e. g. used by `sort`.

All of this user code might be able to interact with the environment in an non-thread-safe way through side effects. These are most legitimate for the first class of functors. However, for some routines, the standard states that they "shall not have any side effects", namely `transform` [C++03, Section 25.2.3] and the numerical algorithms `accumulate`, `inner_product`, `partial_sum`, and `adjacent_difference`.

For the other classes, side effects usually imply a bad programming style, or an abuse of functionality.

The extensive code re-use, justified in Section 4.4, yields another small problem. Functors have to be passed to subroutines (analogous to the call stack in Figure 4.28), and the question arises whether this should by done by value or by reference. The common prototypes of the algorithm calls take the functors by value. This prohibits returning any "state" to the caller, which is good news. However, in order to avoid copying the same thing too often , one should pass on *references* to subroutines.

Another problem is the use of user data types with a limited interface. Some algorithms copy elements from the input to some temporary storage. Unfortunately, the library must not assume that the given data type is default-constructable, i. e. it has a constructor without arguments producing a "vanilla" element. Thus, so the temporary storage cannot be easily allocated. The algorithms never produce "new" elements, so we can always circumvent this problem.

One option is to use pointers to the existing elements, the other is to reserve raw memory and to construct the element using *placement new*, calling the copy constructor with an existing element as argument.

Exception Safety. Two things that avoid each other like the plague are parallelism and exceptions. There is no theoretical reason for this, rather a lack of well-specified behavior in existing C++ implementations.

The problem surfaces when a routine in a thread throws an exception which is not caught inside the thread and propagates up the stack.

If the parallelization is done manually for a certain algorithm, this situation can be avoided by carefully specifying throw and catch regions within each thread of parallel execution, so that an exception never leaves the originating thread. The consequences to program execution can then be handled by checking for error conditions, e. g. forcing the other threads to terminate, or somehow sensibly resuming execution.

For a library which takes functors, such a specific solution is not applicable, and the search for a more robust solution is an area of active research. Some general restrictions should apply, as follows. The algorithms themselves should not throw any exceptions, as stated by the STL specifications. This is easy to achieve since malformed input is not possible in most cases, and the sequential algorithms do not throw exceptions either. Some algorithms rely on the given input sequences being sorted, however, even the sequential versions do not check for this precondition, and as a result may deliver wrong results, not terminate, or crash the whole program. For all algorithms taking a comparison functor, inconsistent return values (failing to be a strict weak ordering) can lead to undefined behavior, just as in the sequential case.

So exceptions have to be expected only from user-defined code, as listed in the paragraph above. But left as an open issue is the problem of a sensible reaction to an exception being thrown. Rethrowing the exception after the clean termination of the algorithm is a reasonable choice, as is stopping all involved threads as soon as possible.

A `catch` clause can either specify an exception type to catch, or catch all exceptions, as marked by an ellipsis (`...`). In the first case, the type of the exception is known, so it could be rethrown. In the second case, the exception object is unknown and one can only attempt to deduce it via nested try blocks containing rethrows. But of course, the set of all expectable exceptions is unknown to the library.

Luckily, C++0x comes at our help [C++10, Section 18.8.5] by introducing *exception propagation*. A pointer to the *current* exceptions can be obtained via `current_exception()`, and stored in a shared variable. After cleanly terminating all worker threads on the algorithm level, the exception could be handed over to the caller by calling `rethrow_exception()`.

This feature is not implemented yet in the parallel mode, though.

Tag Parameters. In order to allow compile time algorithm choices such as forcing sequential execution, the function prototypes must be augmented by additional “tag” parameters. This makes some of the standard compliance tests fail because of the augmented method signature. For user code, this should not make any practical difference, since all standard-compliant calls

will still succeed. However, danger could arise from a new standard adding further arguments to existing algorithm calls.

Conclusion on Standard Compliance. So far, the libstdc++ parallel mode works with "reasonably" programmed code. This includes assumptions as side-effect-safe and exception-free functors passed to the parallelized algorithms.

Others are inherent to parallel computing, like side-effect issues and exceptions. Those could be mitigated by providing thread-safe containers, as provided by the Intel Threading Building Blocks [Int].

Benefits from C++0x

In addition to the benefits listed for the specific issues, C++0x also provides improvements that are more generally helpful for using the parallel mode.

Lambda functions allow for more elegant definition of functors, thereby making the usage of STL algorithms easier and more intuitive.

In the MCSTL, we usually *copy* full objects, since this is the usual way the STL does things. Optimizations like taking the pointer could be done by the developer himself, by parametrizing the algorithm with a pointer instead of the full object, and adapting the functor. So-called *rvalue references*, however, can reduce cost of copying (potentially complex) objects. Now, they can be "moved" explicitly, implying that the data must not be accessed in the original location any longer. By providing a specific *move constructor*, the developer can thereby make moving much more efficient than the combination of copy construction/assignment and deletion. This helps the STL algorithms in general, because many of them actually only *move* data, but do not copy it, e. g. `sort` and `random_shuffle`. The parallel mode benefits in particular because copying data consumes memory bandwidth, a common bottleneck for parallel speedup, as described before.

4.5 Treating Sequences without Random Access

As pointed out before, iterators that do not allow constant time random access pose a problem to parallelization. First of all, a thread cannot "jump" to a certain starting point in sublinear time. Secondly, the length of the sequence might not be known to the to-be-parallelized algorithm.

More formally, we focus on the problem that a sequence of elements is given as a pair of *forward iterators*, which is the minimal requirement for many STL algorithms. The only operations a forward iterator supports by definition are dereferencing, advancing to the next element, and comparison for equality to another iterator. This property makes traversal inherently sequential. Determining the length n of such a sequences also takes linear time.

This problem should not be mistaken with the well-known list-ranking problem, where the pointers to all elements are known, just their order is to be determined. However, since splitting is a quite lightweight operation, its running time may be small compared to the executing time of the whole algorithm, which may allow for overall speedup. According to Amdahl's law, the sequential part should be finished as quickly as possible.

We need algorithms that partition a sequence of unknown lengths into p parts of similar length, bounded by iterators, for further processing in parallel.

A trivial algorithm would be to traverse the sequence twice. First to determine its length, then to collect the corresponding iterators. We call this algorithm TRAVERSETWICE. However, traversing the sequence can be costly, since the element can be spread in memory cache-unfriendly, and/or advancing to the next element could be a non-trivial operation. Thus, a second pass should be avoided.

Another trivial algorithm that needs only one pass is to copy all iterators into a unbounded array, effectively converting the sequence into a random-accessible one. However, this method called POINTERARRAY comes with a space penalty, and the unbounded array has high constant factors for internal reorganization.

We propose the algorithm SINGLEPASS, which divides a list into parts of similar length in only one traversal over the list. As quality measure, we take the ratio r between the length of the longest part and the length of the shortest part. Smaller r is better since this will result in better parallel load-balancing. A tuning parameter σ, the *oversampling factor* determines the accuracy.

4.5.1 Single-Pass List Partitioning Algorithm

Let L be a forward linearly traversable input sequence (e. g. a linked list). Our single-pass algorithm, denoted SINGLEPASS, keeps a sequence of boundaries S, where $[S[i], S[i+1])$ defines the i$^{\text{th}}$ subsequence of L. The basic approach is to add short lists to the end of the sequence, and to merge adjacent ones when those become too many. The pseudocode is given in Table 4.3.

This algorithms needs $\Theta(\sigma p)$ additional space to store S. Its time complexity is $O(n+\sigma p \log n)$, because we need to traverse the whole sequence, and step 3 visits $\Theta(\sigma p)$ elements in $\Theta(\log n)$ iterations. The quality metric r is bounded by $\frac{\sigma+1}{\sigma}$ in the worst case, where just one complete subsequence was appended after reducing the list. Overall, the algorithm guarantees that two parts differ at most in one complete subsequence, i. e. in at most k elements.

We can also give an upper bound to the expected ratio r, namely $r \leq 1 + \frac{1}{\sigma p}((p-1)\ln(2))$. E. g., for $\sigma = 10$ and $p = 32$, the longest subsequence is at most 10% longer than the shortest one, expectedly 7% longer.

A generalization of this algorithm executes steps 3a and 3b only every m^{th} loop iteration, where m is another tuning parameter. This way, the quality is improved, by spending more time and additional space, but sublinearly.

4.5.2 Experimental Results and Conclusion

We evaluated our program on the OPTERON system, running each test at least 10 times, and taking the average. We compare against the trivial algorithms TRAVERSETWICE and POINTERARRAY, the latter using an STL `vector`.

Concerning the running times of the pure partitioning, SINGLEPASS is faster than both competitors for all input sizes, and more than twice as fast for large inputs (see Figure 4.29). A sufficient quality (less than 10% difference) is achieved for an oversampling ratio of $\sigma = 10$, as

1. Let $k := 1$, $S := \{\}$.
2. Iteratively append to S the first $2\sigma p$ consecutive subsequences of length 1 from L.
   $S = \{0, 1, 2, \ldots, 2\sigma p\}$
3. While L has more elements do:

   Invariant: $|S| := 2\sigma p$, $S[i+1] - S[i] = k$

   a) Merge each two consecutive subsequences into one subsequence.
   $S[0, 1, 2, \ldots, \sigma p] := S[0, 2, 4, \ldots, 2\sigma p]$
   This results in σp subsequences of length $2k$.

   b) Let $k := 2k$.

   c) Iteratively append to S at most σp consecutive subsequences of length k from L.
   $S := \{0, k, \ldots \sigma pk, (\sigma p + 1)k, (\sigma p + 2)k, \ldots, l\}$,
   $\sigma pk < l \leq 2\sigma pk$
   If L runs empty prematurely, the last subsequence is shorter than k.
4. The $\sigma p \leq |S| \leq 2\sigma p$ subsequences are divided into p parts of similar lengths as follows. The $|S|$ mod p rightmost parts are formed by merging $\lceil |S|/p \rceil$ consecutive subsequences each, from the right end. The remaining $p - (|S| \bmod p)$ leftmost parts are formed by merging $\lfloor |S|/p \rfloor$ consecutive subsequences each, from the left end.

Table 4.3: Pseudocode for the SINGLEPASS list partitioning algorithm.

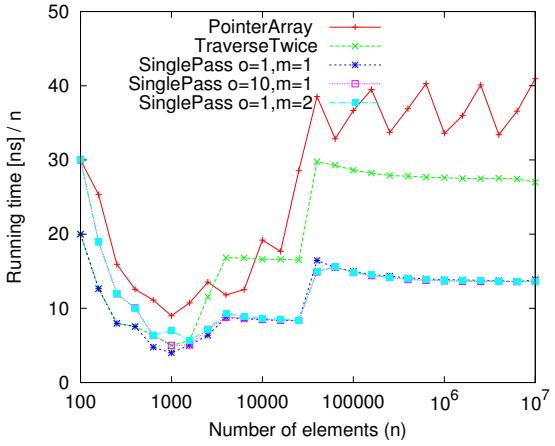


Figure 4.29: Running times of the list partitioning algorithms for $p = 4$.

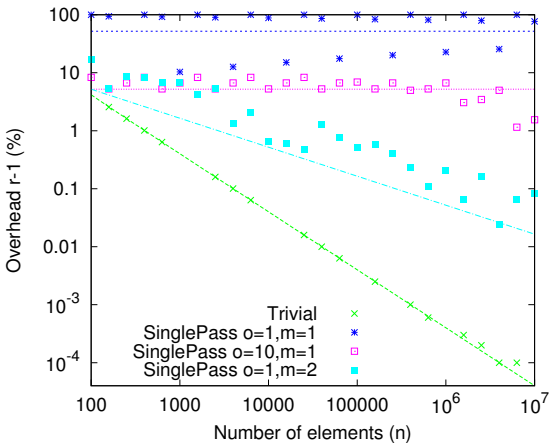


Figure 4.30: Quality of the list partitioning algorithms for $p = 4$. We show the worst-case overhead $r - 1$, as well as its expected value (dotted lines). The results are in percent. Note that the missing points are actually 0.

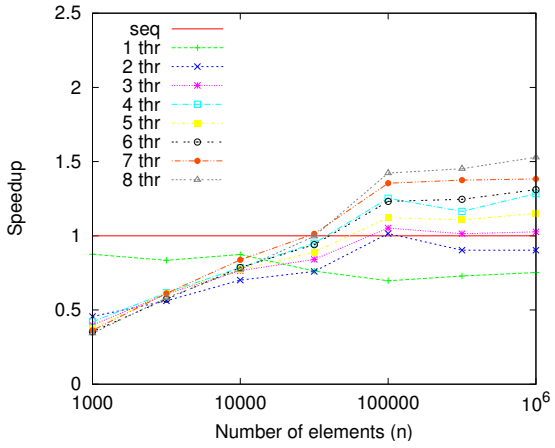


Figure 4.31: Speedup for STL list `sort` using TRAVERSETWICE partitioning.

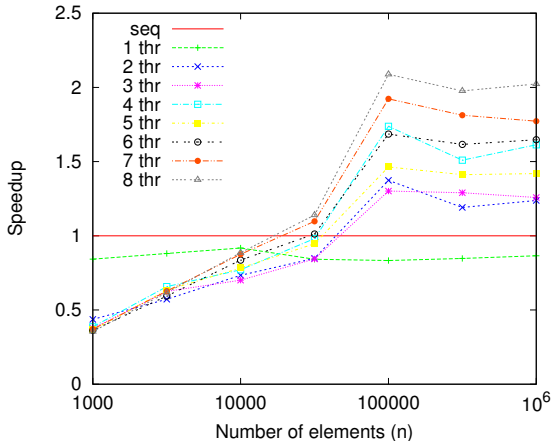


Figure 4.32: Speedup for STL list `sort` using SINGLEPASS partitioning.

expected. Please note that the result quality shown in Figure 4.30 is independent of the actual input size, it depends solely on the input *length*. Setting $m = 2$ improves quality dynamically for larger inputs, at no extra cost in running time, but using $\Theta(\sqrt{np})$ extra space. Of course, the two pass algorithms can deliver even better quality, but the improvement would not be significant for the running time of the overall parallel algorithm. Partitioning 10^6 elements takes about 14 ms with SINGLEPASS.

However, list partitioning is only a precomputation for the actual parallelized algorithm. Therefore, we have evaluated for sorting, using pairs of 8-byte integers as elements. We use mergesort to sort the partitioned subsequences by one thread each, which is implemented without splitting the sequence in the middle, using a bottom-up approach. Finally, the resulting sorted subsequences are 2-way merged recursively.

Figures 4.31 and 4.32 show that the SINGLEPASS algorithm with parameters $\sigma = 1$ and $m = 2$ enables much better speedups for sorting a linked list of pairs of 64-bit integers in parallel, than using the runner-up, TRAVERSETWICE.

Overall, the speedups are not particularly great, but we can show that sequences based on forward iterators do not deny parallelization categorically. With increasing complexity of the work, the overhead for sequential partitioning will drop, allowing better speedups.

Sequentially sorting a linked list of of pairs of 64-bit integers takes about 0.73 s, compared to 0.12 s for the random-access case.

This content of this section was developed in close collaboration with Leonor Frias, and is also part of her PhD thesis. Therefore, this is just a short mentioning, details on all aspects can be found in [FSS08a, FSS08b].

4.6 Case Studies

In order to show that a library of basic algorithms is indeed useful, we present two algorithmic applications. The first one stems from string processing, while the second one deals with graphs.

4.6.1 Suffix Array Construction

A *suffix array* states the sorted order of all suffixes of a string, usually respresented as a sequence of starting positions. It serves, for example, as an index data structure supporting efficient full-text search on strings. Further applications exist in string matching, genome analysis, and text compression.

There are several efficient algorithms known for constructing a suffix array. The DC3 algorithm [KSB06], based on so-called *difference covers* with length 3, runs in optimal linear time and is simple, it can actually be implemented in less than 100 lines of C++ code. Let T be the input string, consisting of characters $t_0 t_1 \dots t_n$, and S_i the suffix $t_i t_{i+1} \dots t_n$. The algorithm consists of four major steps:

Step 0: Construct a sample Construct a *sample* consisting of all suffixes with starting positions not divisible by 3. Of course, no data is duplicated, we store only offsets.

Step 1: Sort the sample suffixes For each sample suffix, take the three first characters to form a triple, and concatenate them to a string R of triples. In order to sort the suffixes of R, sort the triples themselves first and calculate their rank, where equal triples get equal ranks.

If the ranks are all unique, the order of the suffixes is already determined by the order of the triples.

Otherwise, replace each triple by its rank, and suffix sort the resulting string of ranks recursively, its length being two thirds of the original string. After that, assign to each sample suffix its rank.

Step 2: Sort the non-sample suffixes Sort the non-sample suffixes easily by taking the first character and the rank of the consecutive suffix, as computed in Step 1.

Step 3: Merge sample suffixes and non-sample suffixes Merge the sequence of non-sample suffixes and the sequence of sample suffixes using a regular comparison-based merge. To compare a sample suffix S_i with a non-sample suffix S_j, we distinguish two cases:

$$\begin{aligned} i \equiv 1 \pmod 3 : S_i < S_j \Leftrightarrow \quad & (t_i, rank(S_{i+1})) < (t_j, rank(S_{j+1})) \\ i \equiv 2 \pmod 3 : S_i < S_j \Leftrightarrow \quad & (t_i, t_{i+1}, rank(S_{i+1})) < (t_j, t_{i+j}, rank(S_{j+1})) \end{aligned}$$

As seen, the algorithm includes sorting tuples several times. Since those are tuples over a fixed-size alphabet and indices into the string, we can use integer sorting, which runs in linear

time, as do all other steps. Thus, the resulting recurrence $T(n) = T(2n/3) + O(n)$ solves to $T(n) = O(n)$.

A parallel version of the algorithm, designed for distributed-memory computers, has been studied extensively [KS07]. Simliar to our approach here, it parallelizes the steps of the algorithm, i. e. scanning and sorting long sequences of tuples. The implementation scales well to many nodes, however, its absolute performance is about an order or magnitude slower than the generic multi-core implementation presented here. Our implementation takes any sequence of elements as input, specified by begin and end random access iterators.

Parallelization efforts

To parallelize this algorithm, we have to lift some of the steps to the high level they deserve, enabling the use of STL functions.

On the first glance on the code, the first step is not so easy to parallelize.

```
for (int i = 0, j = 0; i < n + (n0-n1); i++)
    if (i % 3 != 0)
        s12[j++] = i;
```

However, we can get rid of the data dependency on `j` by recasting the assignment to `s12[j]=((3*j)>>1)+1` and changing the loop range accordingly. The resulting loop does not have any interdependencies and can be parallelized using the `omp parallel for` pragma. The multiplication and the shift should be even faster than the modulo calculation and a hard-to-predict branch.

Sorting the sample triples could be done very easily by calling the parallelized sorter with a respective comparison functor on the tuple. However, pre-tests showed that a comparison-based sorter is indeed considerably slower than the proposed integer sorting routine, which is also necessary to guarantee the linear running time. Therefore, we have implemented a parallelized version of the counting sort used in the original code. It parallelizes counting the occurrences and copying the elements into the respective buffer. Unfortunately, the STL does not include the interface for an integer sorter, but this would be a natural extension, and also an algorithm worth providing a parallel implementation for.

Determining the rank of the triples can be done by a customized `partial_sum`. The result sequence consists of pairs of the resulting rank and the last element. This way, in each "addition" operation, the current element can be compared with the former one. If and only if they differ, the rank is incremented.

Assigning the ranks to the suffixes, i. e. inverting the sample suffix array, is done by augmenting the respective simple for loops with the `omp parallel for` pragma as before. This is the easiest way of parallelizing a operation over a sequence that needs the verbatim index for calculation, and is not satisfied with a dereferenced iterators as argument.

Sorting the non-sample suffixes is done as before, using the parallelized counting sort on pairs.

Merging the sample suffixes with the non-sample suffixes is the most sophisticated and complicated step in the original code. In order to put all necessary information into the sequences, we actually merge triples again. Each triple stores

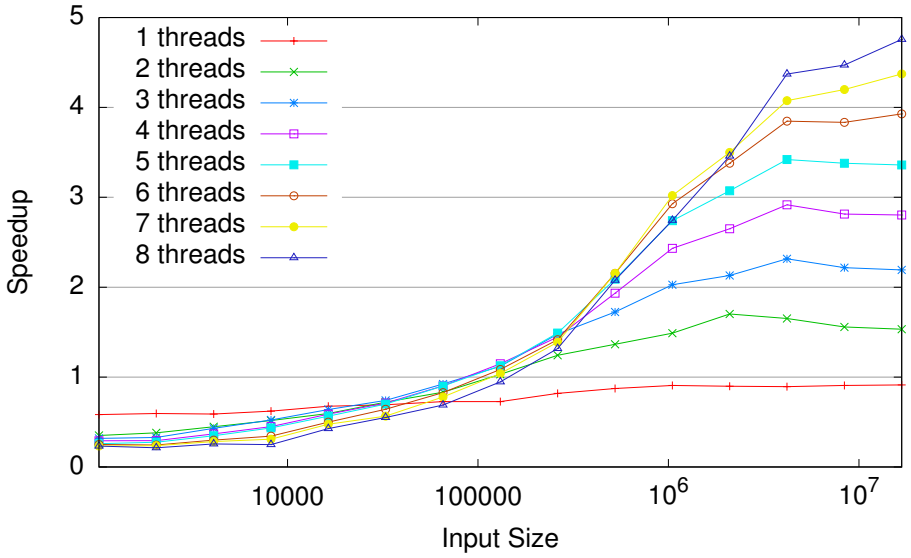


Figure 4.33: Speedup for suffix array construction on random strings on OPTERON.

- whether the triple represents an entry of the sample or the nonsample suffix array
- the index of this entry in the corresponding suffix array
- the value that is to be written to the output

These triples are generated transparently from the input sequence with a custom iterator that iterates over the *indexes* of the sequences. Each index is passed through a functor which creates a triple from the index and the input sequence. The triples can then be compared (and thus merged) with an appropriate functor.

Experimental Results and Conclusions

We have evaluated the performance of the program on OPTERON, at first on random input data. As shown in Figure 4.33, we achieve speedups starting from an input size of about 100 000, approaching about 2.8 for 4 cores, and 4.7 for 8 cores. The overhead over the unchanged original code (shown as "speedup" for one thread) is also asymptotically small. The breakdown of timings for the different steps of the algorithm for a large input is given in Figure 4.34, including *relative* speedup. Obviously, all steps are sped up well, the non-parallelized parts ("other") do not take significant time at all, their share is invisible in the plots.

However, results for random inputs could be misleading for this kind of algorithm. Typically, the recursion depth is only 1, due to the random triples in Step 1 getting unique soon. Thus, we tried on 16 MiB prefixes of two real-world data sets: The input data consists of prefixes of the "Source" and the "Gutenberg" data set, as in [DKMS05]. The Gutenberg data sets (recursion depth 9) is comprised of natural language text, while the Source data set (recursion depth 8) contains program source code.

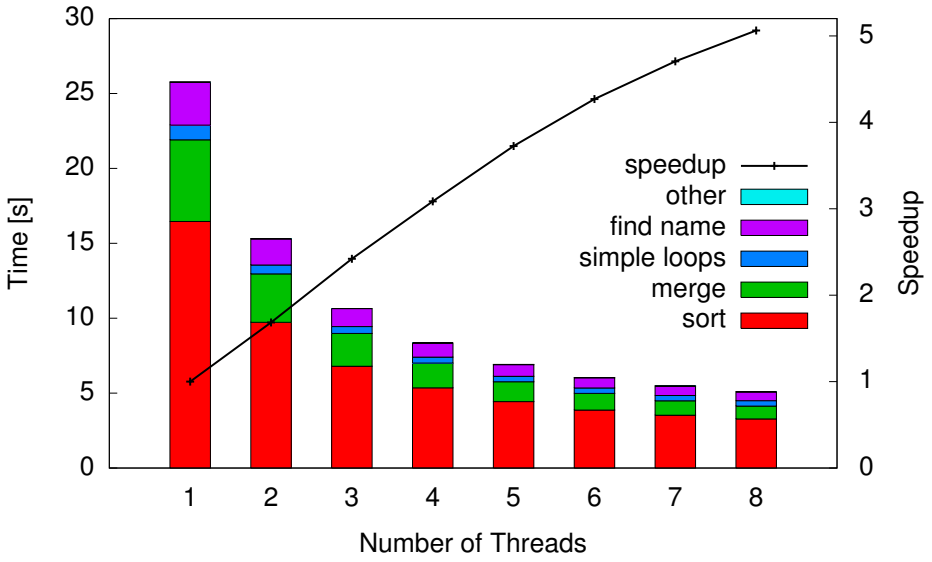


Figure 4.34: Detailed timings for suffix array construction on a random string of length 2^{24} on OPTERON.

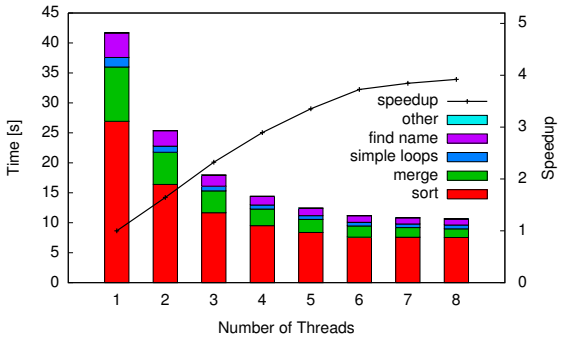


Figure 4.35: Speedup for suffix array construction on the Gutenberg data set on OPTERON.

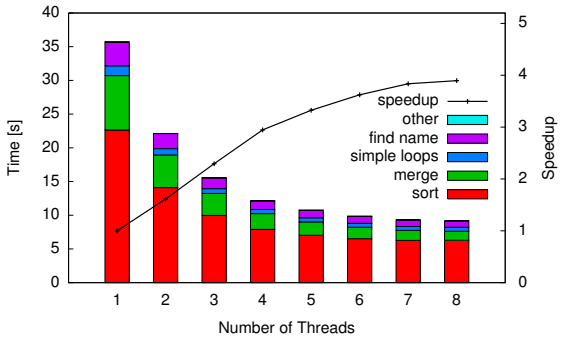


Figure 4.36: Speedup for suffix array construction on the Source data set on OPTERON.

Although the deeper recursion leads to shorter sequences, the relative speedup stays similar. We reach about 2.9 for 4 threads and 3.9 for 8 threads, as shown in Figures 4.35 and 4.36.

Better speedups for small inputs could have been achieved by switching off parallelization for less sped up steps in deep recursion levels, but this was not a primary goal here. We wanted to show that in fact an algorithmic application can be parallelized on a high level utilizing the basic algorithms provided by the MCSTL.

The DC3 algorithm is also regarded in its external memory variant in Chapter 6.

4.6.2 Minimum Spanning Trees

The minimum spanning tree of an undirected edge-weighted graph is the minimum total weight subset of edges that forms a spanning tree of the graph. Computing the minimum spanning tree is a classical problem from graph theory, and has many applications, the most obvious being the design of distribution networks.

One of the two canonical algorithms for this problem is *Kruskal's algorithm*. After sorting the edges by weight ascending, it iteratively inspects them in this order, starting with an empty tentative edge set T. If the inspected edge completes a cycle in T, it is discarded, otherwise, it is added. After all edges have been processed, the T is the sought MST.

The algorithm runs in $O((m+n)\log m)$ time, where n is the number of nodes and m is the number of edges. For dense graphs, the sorting step asymptotically dominates with running time $O(m \log m)$.

The other canonical algorithm, by Jarník-Prim, grows a tree starting from a certain node, using a priority queue for finding the leaving edge with the least weight leaving the node set visited so far. Using a sophisticated priority queue, the running time ca be made $O(m + n\log n)$.

In the following, we will present an algorithm-engineering-driven result on improving the performance on this problem. The algorithmic originality are not a contribution of this thesis, we concentrate on the parallelization. The experimental results show exemplarily how the simple usage of parallelized basic algorithms from the MCSTL can make a significant difference.

For the basic Kruskal algorithm, the natural thing to do is to parallelize the edge sorting step. However, the second step is inherently sequential, since its correctness depends on the order the edges are considered in. Still, the speedup achieved by just parallelizing the edge sorting is more than 5 on 8 cores for a common input (see Figure 4.38).

Parallelizing the Jarník-Prim algorithm is difficult for similar reasons. Each selection of an edge depends on the former selection because the priority queue holding the candidate edges is updated each time. So only very fine-grained parallelization would be possible, in the order of node degrees. And there is no preprocessing step that could be sped up as simply as for Kruskal's algorithm.

Let us have a look at an improved variant of Kruskal's algorithm. It improves Kruskal's performance on dense graphs, and also performs "bulky" operations for this input class, which allows for easy parallelization. The first key idea – lazy sorting – implies that the relative order of heavy edges is left open until it is evident that they are actually needed. The second key idea – filtering – exploits that all edges that connect nodes of the same component can be discarded early, before further sorting them.

```
 1: function FilterKruskal(E, T : Sequence of Edge, P : UnionFind)
 2: if m ≤ kruskalThreshold(n, |E|, |T|) then
 3:    Kruskal(E, T, P)
 4: else
 5:    pick a pivot p ∈ P
 6:    E_≤ := ⟨e ∈ E : e ≤ p⟩; E_> := ⟨e ∈ E : e > p⟩
 7:    FilterKruskal(E_≤, T, P)
 8:    E_> := filter(E_>, P)
 9:    FilterKruskal(E_<, T, P)
10: end if
11:
12: function filter(E : Sequence of Edge, P : UnionFind)
13: return ⟨(u, v) ∈ E : u, v are in different components of P⟩
```

Figure 4.37: Pseudocode for the Filter-Kruskal algorithm. Passing the parameters n and m in each recursive call is omitted for clarity of presentation.

For doing the sorting lazily, we integrate quicksort and the edge scanning part of Kruskal. As usual, the sequence of edges is partitioned with a randomly chosen pivot. Then, the algorithm calls itself with the lighter part to compute a partial MST. If the result is not yet a spanning tree, the algorithm continues. The heavier edges connecting nodes in the same component are removed by the filtering step, before calling the algorithm itself with the remainders. To counter overheads for small edge sets, a regular Kruskal algorithm is executed when the number of edges drops below a certain threshold. Figure 4.37 gives pseudocode for this so-called *Filter-Kruskal* algorithm. For graphs with random edge weights, Filter-Kruskal runs in $\mathrm{O}\left(m + n \log n \log \frac{m}{n}\right)$ time, i. e. linear for not too sparse graphs.

Parallelism can be incorporated into three steps of the algorithm. For the base-case Kruskal call, we use parallel `std::sort` for preprocessing. The partitioning in line 6 is done by `std::partition`. The *filter* subroutine can be implemented using `std::remove_if` and the appropriate predicate. Although the operations on the union-find data structure concurrently write elements due to *path compression*, no locking is needed as long as we consider pointer updates to be atomic, which is the case on current machines.

Just using the parallelized calls gives speedups up to about 3, despite this routine being not very coarse-grained. Thanks to this improvement, all other approaches are beaten by Filter-Kruskal for graphs of not too small sizes. Exemplary, we present this for random graphs in Figure 4.38.

A more elaborate study of the algorithms, a review of related work and more detailed results can be found in the respective publication [OSS09]. Although the Filter Kruskal algorithm itself is not a contribution of this thesis, the algorithm engineering work on MST algorithms was strongly driven by the fact that parallelized subroutines were easily available. Focussing on parallelizable algorithms justified experimentally.

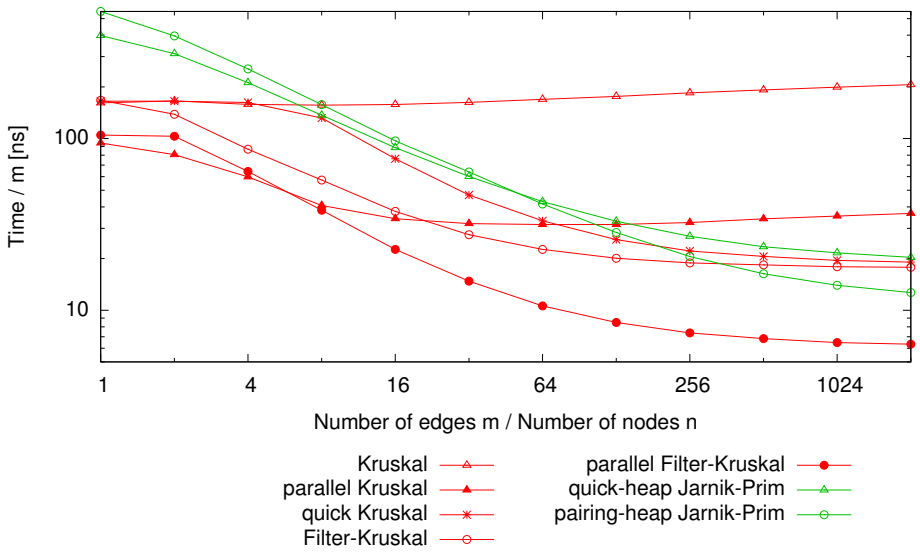


Figure 4.38: MST computation for random graphs with 2^{16} nodes on OPTERON. The filled points denote parallel versions. We compare against Jarník-Prim with two different priority queues, and to *quick* Kruskal, which is the Kruskal variant with lazy sorting, but without filtering.

4.7 Conclusions and Future Work

We have demonstrated that algorithms of the STL can be efficiently parallelized for multi-core processors. We parallelized algorithms of the library fully generically, including bulk operations for the dictionary data structures.

We have measured performance for representative algorithms, parameterized with selected data types and functors. The results have been analyzing in detail, taking theoretic complexities as well as hardware limitations such as memory bandwidth into account.

For large inputs, we usually have significant speedups. The Sun T1 processor even shows speedups far exceeding the number of cores when using symmetric multi-threading per core. When the speedup is less than linear, this can usually be lead back to a saturated memory connection in the examined cases. Better speedup is still possible for more costly functors.

For too small inputs, there is often no speedup, which is due to thread starting overhead and the advantage for the sequential algorithm of having the data in one of its close caches. However, generally, we have learned that speedup is regularly possible from 50–100 microseconds of sequential running time, which is quite short. Exceptions are algorithms where a parallel algorithm is called multiple times as subtask, such as for `nth_element`, where this value can go up to 300 microseconds.

We have discussed the software engineering issues that appeared to us in the implementation process. This includes standard compatibility, in particular with respect to semantics, as well as practical things like maintainability and executable size. For problems like partitioning linked

lists for data-parallelization, we give an algorithmic solution, namely a single-pass algorithm with a good quality tradeoff.

For two applications from different fields, we show that the MCSTL routines can significantly speed up higher level algorithms by accelerating the diverse subtasks. In the MST case, the easy availability of basic parallel algorithms shifted the focus to parallelizable algorithms.

We published first results on this topic [PSS07, SSP07] already in 2007. The work on the software engineering issues was published in 2008 [SK08].

In 2007, the MCSTL code was integrated into the STL implementation of the GNU C++ compiler, the libstdc++. The compiler is widely used, it provides state-of-the-art optimization, and is available for a large number of platforms, including x86, x86_64, IA-64, and Sparc. We consider the inclusion a great achievement, which shows the high interest in a parallelized STL implementation.

Future Work. Implementation of the MCSTL will continue with implementations of worthwhile functions that are not yet parallelized. Heap operations for the MCSTL have been considered in independent work [Rob08].

Of course, in an easy-to-use library we would like automatic support for selecting an implementation. Some work in this direction has been done in [TTT+05]. However, more work is needed because it is not sufficient to *select* an algorithm, we also have to use the most effective values for the tuning parameters, decide well when to skip the parallelization, and possibly to use just a part of the available cores. This leads us to the field of auto-tuning.

5 Geometric Algorithms

Computational Geometry is a field in algorithmics, which has many applications in practice, in particular in computer aided design, computer graphics, and geographic information systems [dBvKOS00].

Geometric algorithms are notoriously hard to implement, which is mostly due to arithmetic rounding errors [Sch00]. Theoretical geometric algorithms usually assume exact arithmetic, and exclude "unlikely" inputs like three collinear 2D points as special cases. However, real computers have only limited native accuracy, and real-world inputs are likely to contain degenerate cases. This can lead to contradicting decisions, and thus to crashes or endless loops.

The mentioned difficulties have made Computational Geometry particularly receptive for a joint implementation effort of the scientific community. In 1996, the Computational Geometry Algorithms Library (CGAL) project was started. CGAL [cgad] nowadays is a large C++ software library containing many data structures and algorithms from all areas of computational geometry. It is actively maintained by an international community and grows continuously. CGAL is open source, but also commercial licenses are sold to industry by a spin-off company.

The problem of rounding errors is solved elegantly in CGAL by introducing the notion of *geometric predicates*. A geometric predicate encapsulates a geometric test, e. g. an in-circle test, and guarantees the geometric consistency of all predicates for any input. Internally, the computations are based either on exact arithmetic (e. g. rational numbers) or floating-point arithmetic with arbitrary precision, guided by so-called *separation bounds*. Roughly speaking, the latter means that the tighter the decision is, the more exact the computation is done (based on the original input). This approach usually has only little computational overhead, e. g. about 25% for 3D Delaunay triangulation [CGAa]. This resilience to such problem makes CGAL easily usable in practice.

In general, users desire real-world entities to be modeled as closely as possible with geometry. This leads to large data sets, e. g. for modelling a 3-dimensional object as a point cloud. For this reason, higher computing performance is always very welcome, but now available only through multi-core support. To improve the situation in this respect, we investigate the parallelization of three selected algorithms.

The algorithms presented are 2-/3-dimensional spatial sorting of points (Section 5.1), typically used for preprocessing before incremental algorithms, d-dimensional axis-aligned box intersection (Section 5.2), and construction of a 3D Delaunay triangulation (Section 5.3). While the spatial sorting is a preprocessing step of many algorithms, the other two have been chosen because of expected high demand from users.

Another attempt to enable geometric algorithms for multi-core was recently made by Näher and Schmidt [NS09]. However, the only geometric algorithm presented there is a divide-and-conquer approach for convex hull computation. We mention further related work in the respective sections.

5.1 Spatial Sort

Many geometric algorithms implemented in CGAL are incremental, and their speed depends on the order of the insertion of the geometric primitives, depending on locality in geometric space and in memory. Often, space-filling curves like the Hilbert curve induce a good order on the primitives. Sorting the primitives along such a curve is called *spatial sorting*.

For cases where also some randomization is required for complexity reasons, the Biased Randomized Insertion Order method [ACR03] (BRIO) is an optimal compromise between randomization, and locality induced by the space-filling curve. Given n randomly shuffled points and a parameter α, BRIO recurses on the first $\lfloor \alpha n \rfloor$ points, and spatially sorts the remaining points. CGAL provides algorithms to sort points along a plain Hilbert curve as well as along a BRIO [Del08], in turn based on Hilbert sorting.

Spatial sorting is an important substep of several CGAL algorithms, in particular ones that process the input primitives one by one. The parallel scalability of these algorithms would be limited if the spatial sorting was computed sequentially, due to Amdahl's law. For the same reason, the random shuffling for BRIO should also be parallelized.

5.1.1 Parallel Algorithm

The sequential implementation uses a divide-and-conquer algorithm. It recursively partitions the set of points with respect to a certain dimension, taking the median point as pivot. The dimension is switched and the order is possibly reversed for each recursive call in a way such that the concatenation of the results leaves the points arranged along a virtual Hilbert curve.

Parallelizing this algorithm is straightforward. The division for Hilbert sorting is done by the parallelized `nth_element` function from the MCSTL, with the middle position as argument. The recursive subproblems are processed by newly spawned OpenMP tasks. Spawning is stopped as soon as the subproblem size gets below a configurable threshold size, in order to minimize parallelization overhead. The larger the threshold size, the better the parallelization overhead is amortized. The smaller the threshold size, the more fine-grained the load balancing becomes. When all spawned tasks are finished, the algorithm terminates.

For BRIO, the initial randomization is done using the parallelized `random_shuffle` from the MCSTL. Recursively, it calls the parallelized Hilbert sort as subroutine. Apart from that, there is only trivial splitting, so parallelization is complete.

5.1.2 Experimental results

We used GCC 4.4 for the experiments, the code was based on CGAL 3.4. The input points have double-precision float-point coordinates and are uniformly randomly distributed in the unit square and cube, respectively.

The speedups obtained for 2D Hilbert sorting on OPTERON are shown in Figure 5.1. For a small number of threads, there is good speedup for problem sizes greater than 1000 points, the efficiency is about 60% for 8 threads. This behavior resembles the usual quicksort. The later increase of speedup for 7 and 8 threads stems from the respective behavior of `partition` as

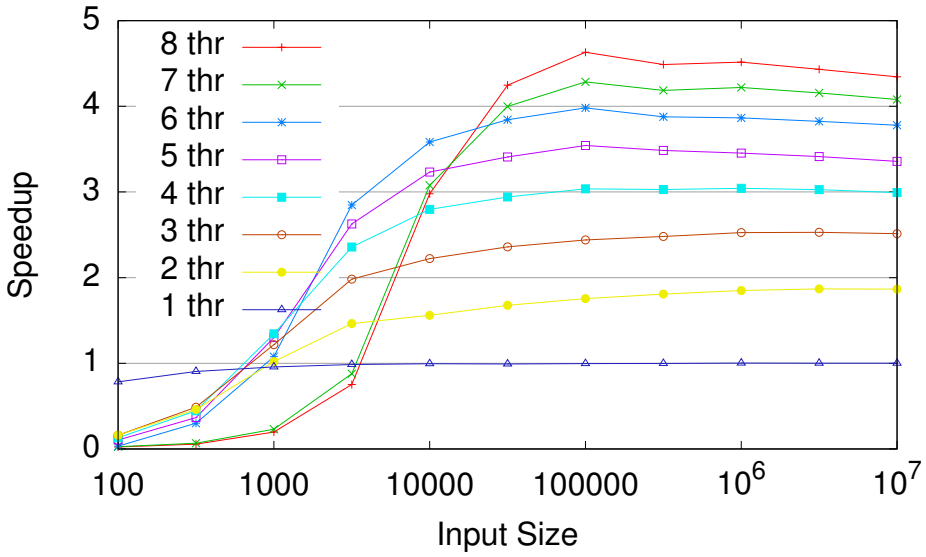


Figure 5.1: Speedup for 2D Hilbert sort on OPTERON.

shown in Figure 4.9. Note that, for reference, the sequential code sorts 10^6 random points in 0.39s.

The results for the 3D case (Figure 5.2) are very similar except that the speedup is 10–20% less for large inputs.

5.2 d-dimensional Box Intersection

As the second algorithm, we consider the problem of finding all intersections among axis-aligned d-dimensional boxes. This problem has applications in fields where complex geometric objects are approximated by their bounding box in order to filter them against some predicate, e. g. collision detection.

There are two flavors of the problem. In the *bipartite* case, we look for pairwise intersections between boxes belonging to two different sets A and B. In the *complete* case, we look for pairwise intersections between boxes belonging to a single set A with cardinality n.

5.2.1 Sequential Algorithm

We parallelize the algorithm proposed by Zomorodian and Edelsbrunner [ZE02]. A version generalized to d dimensions had already been used for the sequential implementation in CGAL [KMZ08], and proven to perform well in practice. We will first describe the sequential algorithm, following some general insights.

Axis-aligned boxes intersect if and only if their projected intervals intersect in all dimensions. Two intervals intersect if and only if one contains the lower endpoint of the other.

For a certain dimension, the algorithm takes the lower endpoints of the first set of intervals, and for each lower endpoint reports all intervals of the second set of intervals that contain

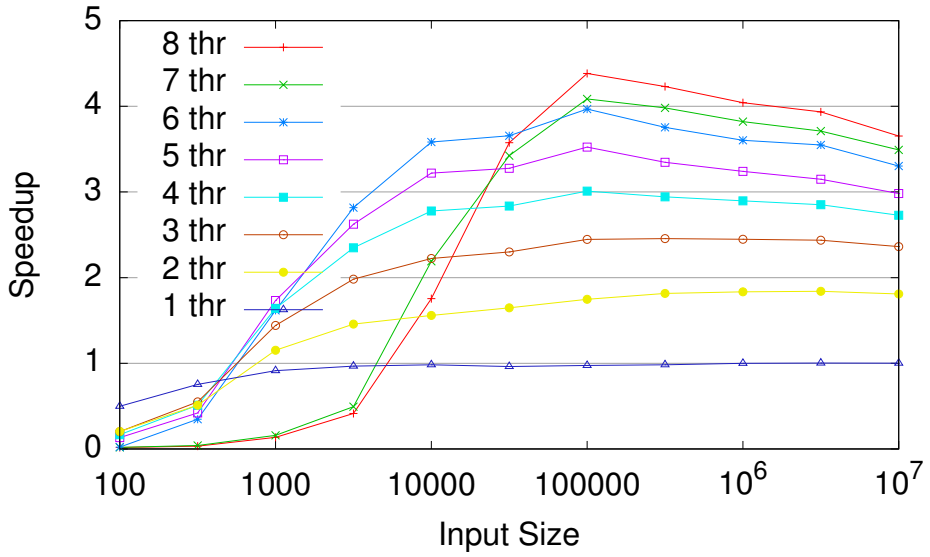


Figure 5.2: Speedup for 3D Hilbert sort on OPTERON.

it. This method is called *batched stabbing*. By doing so vice-versa, all pairs of intersecting intervals are found. Thus, we have a duality between the intervals and their endpoints.

To implement the batched stabbing efficiently over all dimensions, initially, a complex data structure is proposed. For the first dimension, it consists of a segment tree for A viewed as intervals, and a range tree[1] for A viewed as points. Both are balanced binary trees and have the property that each node contains all intervals and points respectively that are contained in the subtree rooted at it, including for each the reference to the originating box. This results in an $O(n \log n)$ space consumption for the first dimension, every point or interval is contained in each of the $O(\log n)$ levels.

For the bipartite case, the batched stabbing is implemented by querying the segment tree with B viewed as points, then querying the range tree with B viewed as intervals. For the complete case, we just query the segment tree with A viewed as points, thereby avoiding a doubled result. In the following, we will focus on the complete case for simplicity.

For query box b, the result for stabbing in the first dimension consists of a set of intersecting boxes in this first dimension. The algorithm still has to check for intersection in the remaining higher dimensions, i. e. perform batched stabbing again. For doing that efficiently, each node of the first-dimension trees includes both a segment and a range tree in turn. These trees contain all boxes contained this node, projected to the second dimension. The same happens up to the d^{th} dimension dimension, making the data structure a multi-level tree.

However, this complex data structure has an overall $O\left(n \log^d n\right)$ worst-case space consumption. Since this is unacceptably large, the trees cannot be stored in memory. Instead, we *stream* them, i. e. construct and traverse them on the fly using a divide-and-conquer algorithm. The division step corresponds the split between the left and the right child node. This approach needs only logarithmic extra memory (apart from the possibly quadratic output size). A further

[1] Segment and range trees are general geometric data structures and described e. g. in [dBvKOS00].

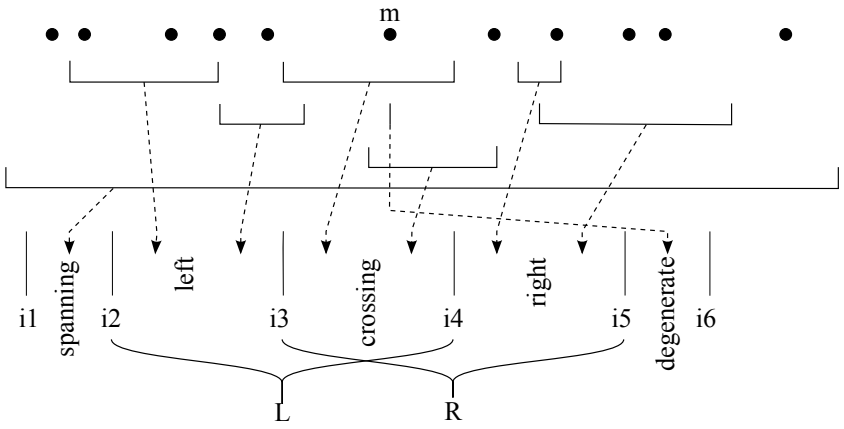


Figure 5.3: Partitioning the sequence of box intervals.

simplification is possible thanks to the duality of points and intervals. Instead of constructing a range tree from the points and querying it with the intervals, we can build a segment tree from the intervals and query it with the points. Thus, we end up with an algorithm just streaming segment trees, swapping points and intervals when appropriate.

For small subproblems below a certain cutoff size, a simpler base-case algorithm checks for intersections, sorting and then scanning the two sequences.

For each recursive call of the divide-and-conquer algorithm, the input consists of two sequences: points and intervals (lower endpoints of the boxes and the boxes themselves projected to this dimension, respectively). The goal is to determine for each point the intervals it stabs. The algorithm processes them as follows.

Degenerate (i. e. empty) intervals are discarded. Intervals spanning the full range are checked on whether they also intersect in higher dimensions, comparing them with the boxes corresponding to all the points. If there is no further dimension, all the boxes spanning the full ranges intersect with the boxes corresponding to all the points, so we output all pairs.

Then, following the divide-and-conquer approach, the sequence of points is partitioned using one of the points m as pivot, chosen such that a good balance is maintained. The sequence of intervals is also partitioned according to m, but in a more complex way. The subsequence L contains the intervals that have their left end point to the left of m, the subsequence R contains the intervals that have their right end point (strictly) to the right of m.[2] Both L and R are passed to the two recursive calls, accompanying the points that possibly stab them. L and R can overlap, common elements are exactly the ones crossing m. All the cases are illustrated in Figure 5.3. The sequential running time of the algorithm is $\mathrm{O}\left(n \log^d n + k\right)$, where k is the number of intersections found.

[2]Whether the comparisons are strict or not, depends on whether the boxes are open or closed. This does not change anything in principle. Here, we describe only the open case.

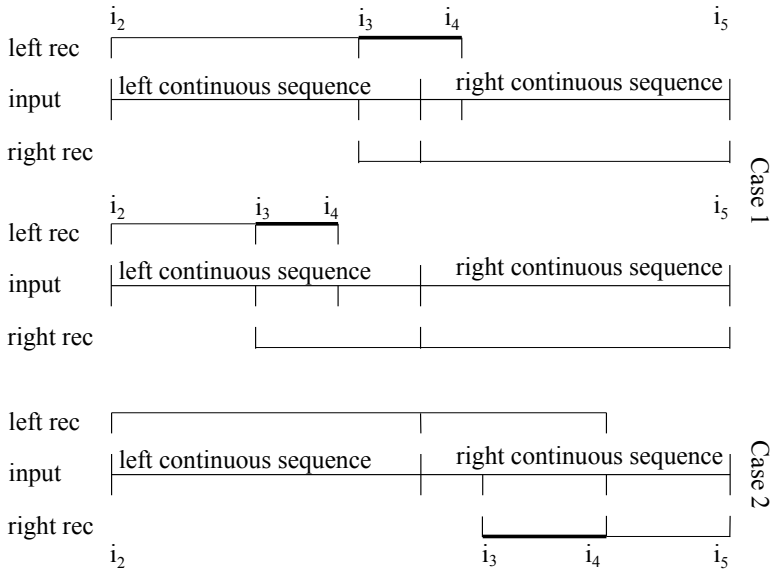


Figure 5.4: Two cases for treating the split sequence of intervals. The upper two situations apply to Case 1, the lower one applies to Case 2. Fat lines denote copied elements.

5.2.2 Parallelization

Again, the divide-and-conquer paradigm promises good parallelization opportunities. We can assign the two parts from the division to different groups of threads, since their computation is mostly independent.

However, as stated before, in general, L and R are not disjoint. Although the recursive calls do not modify the contained intervals, they may reorder them by partitioning, so concurrent access is forbidden, even if read-only. To enable parallelization, we have to *copy* intervals. Note that we could take pointers instead of full objects in all cases since they are only reordered. However, this saves only a constant factor and leads to cache inefficiency due to lacking locality (see the Section "Runtime Performance" in [KMZ08]).

Intelligent Copying. We can reorder the original sequence such that the intervals to the left are at the beginning, the intervals to the right are at the end, and the common intervals are placed in the middle. Intervals not contained in any part (degenerate to an empty interval in this dimension) can be moved behind the end. Now, we have five consecutive ranges in the complete sequence, delimited by iterators i_j, as shown in Figure 5.3. $[i_1, i_2)$ are the intervals spanning the whole region. $[i_2, i_3)$ and $[i_4, i_5)$ are the exclusive intervals for the left and right recursion steps, respectively. $[i_3, i_4)$ are the intervals for both the left and the right recursion steps. $[i_5, i_6)$ are the ignored degenerate intervals.

To summarize, we need $[i_2, i_4) = L$ for the left recursion, and $[i_3, i_5) = R$ for the right recursion, which do overlap. The easiest way to solve the problem is to make a copy of either $[i_2, i_4)$ or $[i_3, i_5)$. However, this is inefficient, since for well-shaped data sets (having a relatively small number of intersections), the middle part $[i_3, i_4)$, which is the only one we really need to duplicate, will be quite small. Thus, we will in fact copy only $[i_3, i_4)$ to a newly allocated

sequence $[i'_3, i'_4)$. Now we can pass $[i_2, i_4)$ to the left recursion, and the concatenation of $[i'_3, i'_4)$ and $[i_4, i_5)$ to the right recursion.

The concatenation must be made only conceptually, to avoid further copying. However, there is danger that passed sequences will consist of an increasing number of subsequences, leading to overhead in time and space for traversing them, counteracting the speedup gained from parallelism. We will now prove that this can always be avoided.

Let a *continuous sequence* be the original input or a copy of an arbitrary range. Let a *continuous range* be a range inside a continuous sequence. Then, the concatenation of *at most two* continuous ranges always suffices for passing a part to a recursive call.

Proof: We can ignore the ranges $[i_1, i_2)$ and $[i_5, i_6)$, since they do not take part in the overlapping recursion, so we have to deal with only $[i_2, i_3)$, $[i_3, i_4)$, and $[i_4, i_5)$.

Induction begin: The original input consists of one continuous range.

Induction hypothesis: $[i_1, i_6)$ consists of at most two continuous ranges.

Inductive step: $[i_1, i_6)$ is split into parts as described above. If i_3 is in the left range of $[i_1, i_6)$ (Case 1), we pass the concatenation of $[i_2, i_3)[i'_3, i'_4)$ (two continuous ranges) to the left recursion step, and $[i_3, i_5)$ to the right one. Since the latter is just a subpart of $[i_1, i_6)$, it consists of at most two continuous ranges.

If i_3 is in the right range of $[i_1, i_6)$ (Case 2), we pass the concatenation of $[i'_3, i'_4)[i_4, i_5)$ (two continuous ranges) to the right recursion step, and $[i_2, i_4)$ to the left one. Since the latter is just a subpart of $[i_1, i_6)$, it consists of at most two continuous ranges.

The two cases and their treatment are shown in Figure 5.4.

There is another problem with copying, though. Because boxes are permuted and copied, and copies are dropped eventually, boxes can be duplicated or removed from the input sequence (an example is shown in Figure 5.5). This destruction of the input sequence might be unexpected to the caller of the routine, but is particularly problematic for the bipartite case, since the second batched stabbing query is performed right after the first one, on the same data. But also, the CGAL specification states that the input is only permuted, but the elements themselves are unchanged. The only feasible way is to copy the data beforehand. However, doing so can cost up to 20% of overall performance, in particular for small inputs. On the other hand, this simplifies parallelization in the case when the sequences are passed using forward iterators.

Deciding whether to subtask. The general question is how many tasks to create, and when to create them. Having many tasks exploits parallelism better and improves load balancing. We assume that the runtime system serializes the task regions if all threads are currently busy, so the running time overhead for task creation stays on an acceptable level. On the other hand, the number of tasks T should be kept low in order to limit the memory overhead. In the worst case, all data must be copied for the recursive call, so the size of additional memory can grow with $O(T \cdot n)$. Generally speaking, only *concurrent* tasks introduce overhead, since the additional memory is deallocated immediately after having been used. So it is helpful to limit the degree of concurrency.

There are several criteria that should be taken into account when deciding whether to spawn a task.

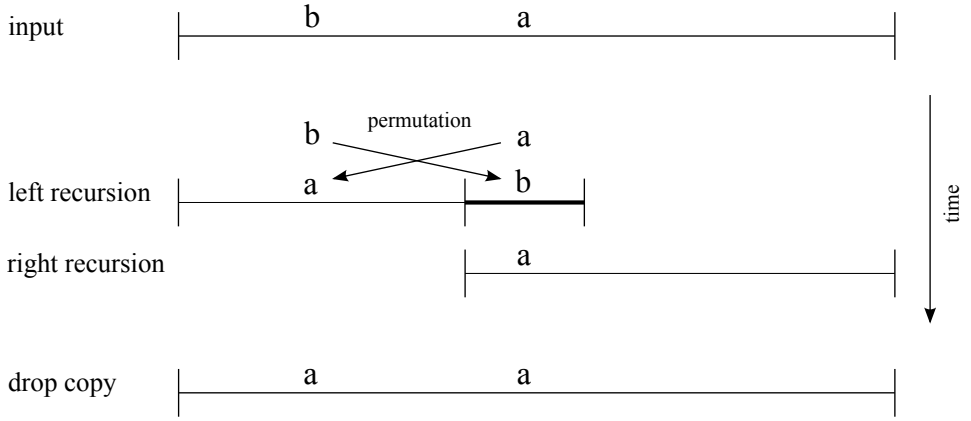


Figure 5.5: The problem in copying elements for box intersection recursion. Copying elements and dropping the copies later might lead to a sequence with duplicated and/or lost elements, because of permutation in the meantime. Here, box a is duplicated, box b gets lost.

- Spawn a new task if the problem to process is large enough (both the number of intervals and the number of points are beyond a certain threshold value c_{min}, which is a tuning parameter). However, in this setting, the running time overhead can be proportional to the problem size because of copying. In the worst case, a constant share of the data must be copied a logarithmic number of times, leading to excessive memory usage.

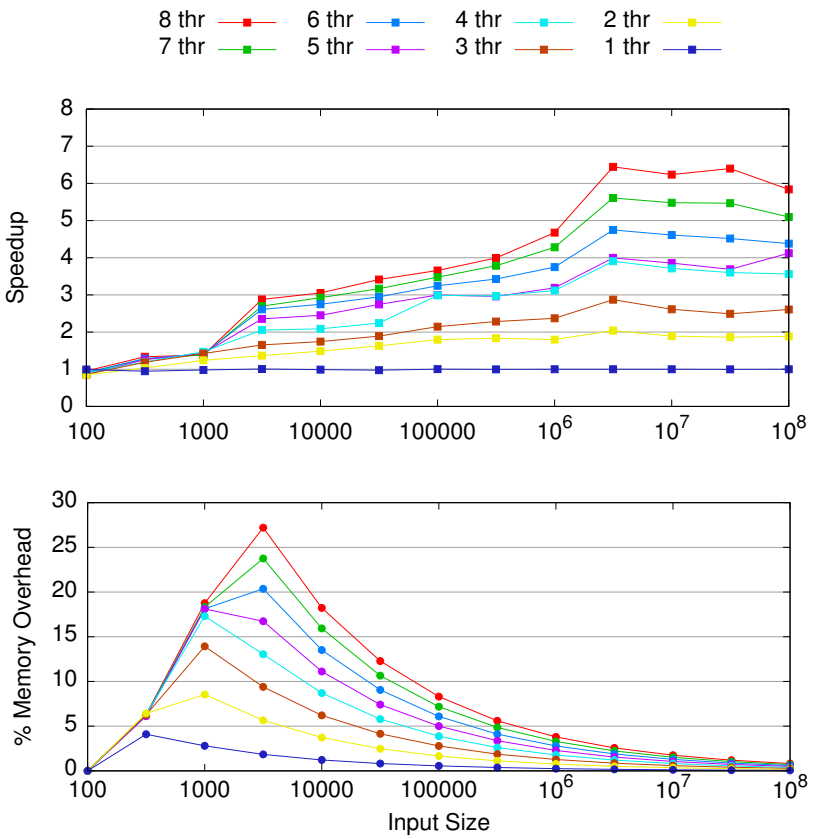
- Spawn a new task if there are less than a certain number of tasks t_{max} (tuning parameter) in the task queue. Since OpenMP does not allow to inspect its internal task queue, we have to count the number of currently active tasks manually, using atomic increment and decrement on a counter. This strategy can effectively limit the number of concurrently processed tasks, and thus, the memory consumption indirectly.

- Spawn a new task if there is memory left from a pool of size s (tuning parameter).

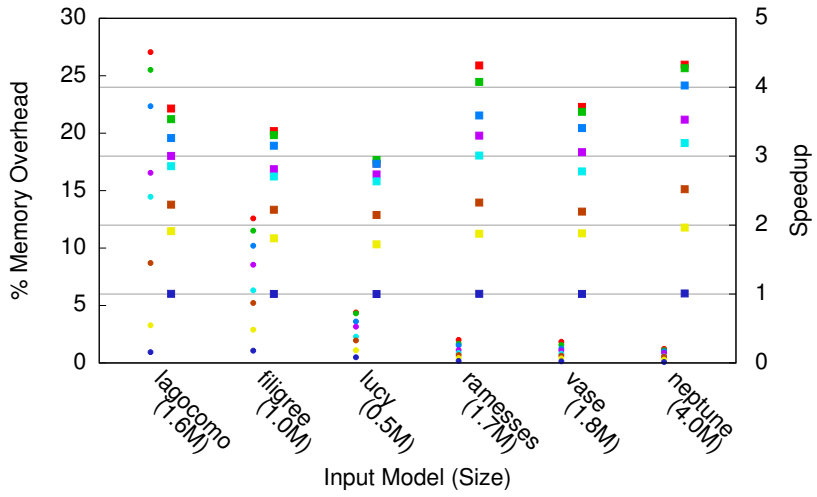
In fact, we combine the three criteria in a logical conjunction, i. e. all three conditions must be fulfilled to spawn a new task.

Related Work.

A theoretical algorithm for the 2-dimensional problem on the PRAM is given in [Cho80]. The duality of endpoints and intervals is not exploited there, the intersections are found in two different consecutive parts. The first part looks for rectangles with intersecting segments. In the second part, it is determined for each left bottom point of a rectangle whether it is contained by any other rectangle. All points are inserted into a range tree for the first dimension, containing lists of points ordered by the second dimension. Then, the rectangles are batch processed to find the contained points and output the resulting intersections. The parallelization is not explicitly done by divide-and-conquer. Instead, by assigning a processor to each node of the tree, all processors descend it in a synchronized fashion.



(a) Randomly generated integer coordinates



(b) Real world data sets

Figure 5.6: Speedup (squares) on OPTERON and relative memory overhead (circles) for intersecting boxes on OPTERON.

5.2.3 Experimental results

3-dimensional boxes with integer coordinates were randomly generated as in [ZE02], such that the expected number of intersections for n boxes is $n/2$. For each intersection found, a user-defined functor is called, which in this case contains just an atomic increment to count the number of intersections.

For the results in Figure 5.6a, we used $c_{min} = 100$, $t_{max} = 2 \cdot t$, and $s = 1 \cdot n$, where t is the number of threads. The maximum memory overhead allowed for copies of the data is limited hard to 100%, but as we can see, the relative memory overhead is much lower in practice, below 27% for all inputs. The speedups are quite good, reaching more than 6 for 8 cores, and being just below 4 for 4 threads. For reference, the sequential code performs the intersection of 10^6 boxes in 1.86s.

Figure 5.6b shows the results for real-world data. We tested 3-dimensional models for self-intersection by approximating each triangle with its bounding box, which is a common application. The memory overhead stays reasonable as before. The speedups are a bit worse than for the random input of the equivalent size. This could be due to the much higher number of intersections found (about $7n$).

5.3 3D Delaunay Triangulation

Given a set S of n points in $\mathbb{R}^d$, a *triangulation* of S partitions the convex hull of S into simplices (cells) with vertices from S. The *Delaunay* triangulation $\mathcal{DT}(S)$ is characterized by the empty sphere property stating that the circumsphere of any cell does not contain any other point of S in its interior. The Delaunay triangulation of a point set is used as base for many other geometric methods because of its nice properties. In particular, the Delaunay triangulation avoids sharp angles.

In CGAL, a Delaunay triangulation is provided as a data structure that supports adding and removing points from the set. The triangulation is kept up-to-date and takes queries like point location.

Parallelizing the insertion of a single point is not worthwhile, so we focus on bulk insertions of points into the set. When starting from the empty set, this is equivalent to constructing the triangulation from scratch.

The existing sequential 2D and 3D implementation in CGAL [BDP+02] are based on this *incremental construction* paradigm, so it is a natural starting point for parallelization. In incremental construction, the input points are inserted into an initially empty partition one by one, following a BRIO ordering.

A point q is said to be *in conflict* with a cell if it belongs to the interior of the circumsphere of that cell, and the *conflict region* of q is defined as the set of all such cells. The conflict region is non-empty since it must contain at least the cell the point lies in, and is known to be connected. For the case where the new point lies outside the convex hull of the already inserted points, there is a virtual cell for each face of the convex hull, connected to a fourth vertex at conceptual infinity.

Each insertion consists of two major steps, *locate* and *update*. In the locate step, the algorithm determines the cell that contains the point to be inserted. This is done by using a walk over the cells, starting at at a cell incident to the vertex that was inserted right before. It proceeds through the cells using orientation tests.

In the update step, following the Bowyer-Watson approach [Bow81, Wat81], the *conflict region* of the point is computed. It is then updated by replacing all edges by a "star" of new edges spanning the conflict region in a way that is consistent with the Delaunay criterion.

In contrast to all other algorithms presented in this theses, we do not split the work into mostly independent parts, but perform concurrent manipulation (in this case insertion) on a shared data structure. Atomicity of operation is assured by locking.

In the following, we focus on the 3D problem, the proposed methods could be carried over to two dimensions as well.

5.3.1 Parallel Algorithm

Bulk insertion of a point set S is parallelized by several threads inserting points into the same triangulation concurrently.

Initially, a comparatively small random subset $S_0 \subset S$ is inserted sequentially in the so-called *bootstrap phase*. This avoids imminent lock conflicts in the beginning, which would be too time-consuming. The size of S_0 is a tuning parameter.

Before the actual parallel insertion phase, the remaining points are Hilbert-sorted, using the parallel routine described in Section 5.1. The points are then distributed to the threads equally. Each thread inserts its points into the triangulation, using basically the sequential algorithm (location and update steps), augmented with locking that protects against concurrent modification of the same region of the triangulation. This protection is achieved using fine-grained locks stored in the cells. We will elaborate on the details in the following paragraphs.

Data Structure. The CGAL data structure used for storing the Delaunay triangulation is made up of vertices and cells. Each vertex stores its coordinates and a pointer to one of the coincident cells. Each cell stores pointers to its (four) vertices and the corresponding (opposite) neighbors cells. An additional data field keeps the status of a cell during an update step, we will give a detailed description of that later. The position of a vertex never changes, and removal of vertices from the triangulation cannot happen during insertion. So after creation, a vertex is always accessible, whereas cells can change or vanish completely. The only member that has to be changed sometimes is the pointer to an incident cell.

Memory Allocation. The incremental constructions allocates and deallocates cells and allocates vertices very often. Thus, we use a thread-safe and thread-local variant of the `Compact_container` class available in CGAL.

The `Compact_container` [HKPW09] consists of a list of blocks with linearly growing size, which allows for amortized constant time addition and removal of elements, very low asymptotic space overhead, and good memory locality.

The version used here [BMPS09, BMPS10] is particularly well-suited for concurrent access by having thread-local blocks. With this optimization, threads have to synchronize only if allocating new blocks, which happens only $O(\sqrt{n})$ times for n contained elements.

Locking and Retreating. The threads only *read* the data structures during the locate step, and they modify it (locally) during the update step. To avoid race conditions, some cells are locked in both steps.

When a thread tries to acquire a lock that is already taken, we have a *lock conflict*. This can lead to a deadlock, since the cells and therefore the locks are geometrically related, and the threads cannot agree on a fixed order. To solve this problem, we assign a unique *priority* to each thread. In case a thread has higher priority, it waits for the lock to be released. Otherwise, it *retreats*, i. e. it releases all locks and restarts the insertion, possibly with another input point. This method avoids deadlocks, since there can be no circular dependencies. A long-lasting livelock is also very unlikely, since retreating takes some time, while the higher priority thread eagerly tries to get the lock. In practice, we have not seen any livelocks.

Locking Details. The algorithm locks multiple cells for a certain amount of time, but it locks vertices only for an instant, and only one at a time.

For the lock, we abuse an integer value contained by each cell object, which normally takes the values 0, 1, and 2, depending on the status of the cell during conflict search. This has the advantage that the triangulation data structure stays unchanged. A value of 0 means that the cell is currently not being used (by any thread). Instead of 1 and 2, we set the value to $1+2\cdot t$ and $2+2\cdot t$ respectively, where t is the thread number, starting from 0. This implies a lock for the specified thread.

During point location, we lock the current cell, test the orientation predicates, lock the next cell, make the next cell the current cell, and unlock the last one again. For the update step, we lock all cells that are in conflict or adjacent to the conflict boundary.

We have to be particularly cautious about locking the first cell. The algorithm must start at a vertex since a reference to a cell might always be outdated. Reading the pointer to an adjacent cell and locking this cell must be performed without a thread interfering by destroying the cell or altering the pointer of the vertex. We achieve this by atomically swapping the pointer with the null pointer and locking all vertices of a to-be-destroyed cell in the same way.

The papers [BMPS09, BMPS10] publishing this work describe three other locking strategies in addition. Instead of having locks for the cells, they lock all or a certain number of vertices for mutual exclusive access to a cell. Since in 3D, the number of vertices is much lower than the number of cells, this could save space for the locks. The vertex-locking variants show worse performance, though, in particular for inputs like the ellipsoid. Also, the cell locking strategy described here has the major software engineering advantage that the data structures can be left completely unchanged. By piggy-backing the lock into an already existing data member, we avoid any space overhead. Thus, the only possible advantage of locking vertices is also void.

Interleaving. After a thread has retreated, it should not retry inserting the same point immediately, because it is likely to fail again in locking the same or a close-by cell. In fact, it should

insert a point that is far away, in order to avoid a lock conflict with the higher priority thread. Taking a random point is not a good option since a close-by vertex is not known but needed for efficient location. So we implement a compromise between locality of reference and conflict avoidance, which we call *interleaving*.

Each thread splits the input points assigned to it into several parts, the number of parts being a tuning parameter, the *interleaving degree*. For each part, it remembers the cell into which the last point was inserted. This hint is used to initialize the locate step for the next point that is inserted from the same part. When a thread has to retreat, it tries to insert a point from the next part, in a round-robin fashion. Because the parts consist of continuous ranges in the Hilbert-sorted input sequence, points taken from different parts are unlikely to be geometrically close, lowering the lock contention.

Load Balancing. As the time for inserting a point may vary greatly depending on its location, and the work load of threads has geometric locality, some threads may take a much longer time to finish their work share. To counter the effects of such bad load balancing, we make it dynamic, using work stealing again. A thread which is out of work steals half of the remaining points from one part of a random other thread. Each part of the stealing thread is refilled this way, so interleaving stays functional. Again, we use only atomic operations, with no explicit communication with the victim thread while stealing. To lower the overhead of the atomic operations while inserting the points, we reserve and steal chunks of points, e. g. 100. Unfortunately, usage of the `for_each` routine from the MCSTL (see Section 4.2) is impossible because of the interleaving and the state shared by multiple insertions, namely the handle of the most recently inserted vertex of the thread.

Related Work

There has been quite some work in parallelizing 3D Delaunay triangulation.

A natural idea is to apply parallel divide-and-conquer, but the merging step is usually very complicated and also hinders good performance [CMPS93]. One can also go the other way around and move the complexity to a partitioning step [LPP01]. However, the partitioning alone takes 75% of the whole processing time.

Results that are related to Delaunay triangulation were obtained in the field of *mesh generation* [CN03, OP97], which usually includes Delaunay triangulation. However, it is unclear how the obtained speedups of up to 2.8 on 8 processors extends to just triangulation.

Parallelization of the incremental construction is intensively studied by Kohout et al. in [KKv05]. For half a million 3D points, the speedup reaches 3.6 on 4 Intel Itanium processor running at 800 MHz. We observed, however, that the absolute speed is about an order of magnitude lower than the CGAL implementation, relativizing the speedup results.

Recently, Blandford et al. [BBK06] published work on concurrent insertion, where the input points are distributed to the processors using geometric relations to tetrahedra of a sample triangulation. They estimate a speedup of 46 on 64 Alpha processors at 1.15 GHz, on very large data sets in the order of 10^9 points. However, the absolute performance is again about an order of magnitude slower than for CGAL.

Conflicting results exist for the usage of transactional memory for Delaunay triangulation. While [KCP06] sees parallelization prevented almost completely, the results of [SSDM07] scale quite well. However, absolute performance of the latter is hard to compare, the authors state results only for 2 dimensions.

5.3.2 Experimental Results

We have implemented the described algorithm based on the existing CGAL code for 3D Delaunay triangulation, and evaluated its performance on three synthetic and three real-world inputs.

The synthetic data sets are varying numbers of uniformly distributed points in the unit cube, and 10^6 points on the surface of an ellipsoid of axis lengths 1, 2 and 3. In addition, we have 10^3 points equidistant on two non-parallel lines, which results in a triangulation of quadratic complexity. For the real-world instances, we took points from the surfaces of a molecule model (525 K points), a Buddha statue (543 K points), and a dryer handle (50 K points). To avoid rounding problems, we use the CGAL geometry kernel that provides exact predicates. For all results presented, we used a bootstrap size equal to $100p$ and interleaving degree 2, which had proved to work well in pretests.

For the random inputs, we reach excellent speedups of more than 7 on eight cores, with a very smooth and concise behavior over the input size, see Figure 5.7a. There is speedup even for as little as 1000 input points, very good speedup for 10 000, and full speedup starting from 100 000 input points. The original sequential code computes a triangulation of 10^6 random points in 15.84 s, showing that this is a very compute-intensive task in general.

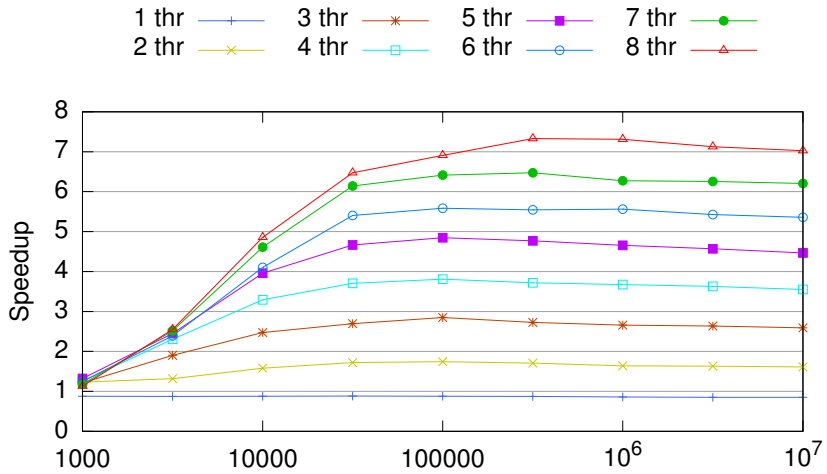
For the real-world inputs, the speedups are a little worse, ranging from 5 to 6.5, as shown in Figure 5.7b. The ellipsoid is still sped up linearly in the number of threads, but with less efficiency. The reason for this behavior is that needle-shaped cells connect far-away vertices, leading to less spatial locality and thus lock contention. The extreme case is respresented by the two-line input. Since every vertex is connected to every other, concurrency is impossible, and thus, there is no speedup. However, this kind of input is unlikely to occur in real applications, it is unclear how to deal with the resulting vast triangulation anyway.

Figure 5.8 gives details on the time spent in the various steps of the algorithm: spatial sort, bootstrap, locate, and update, as well as their subroutines. We see that the speedup is quite consistent over all steps, also affirming the acceleration for spatial sort once more.

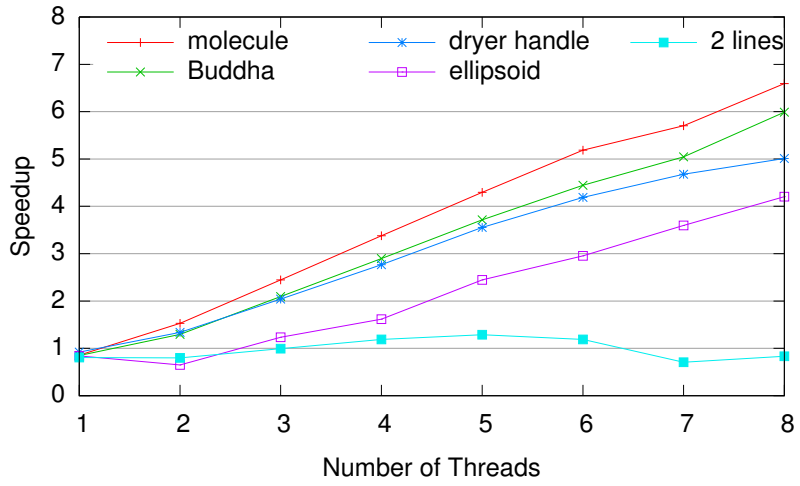
5.4 Conclusions and Future Work

We have presented parallelized algorithms and implementations for important geometric problems, focused on multi-core parallelism in a library setting. The speedups for box intersection and 3D Delaunay triangulation are very good to excellent, which we have also confirmed on real-world inputs in both cases.

The results of this section were published in [BMPS09] and [BMPS10].



(a) Random points.



(b) Real-world data sets and specific synthetic data sets.

Figure 5.7: Speedups for 3D Delaunay triangulation on OPTERON.

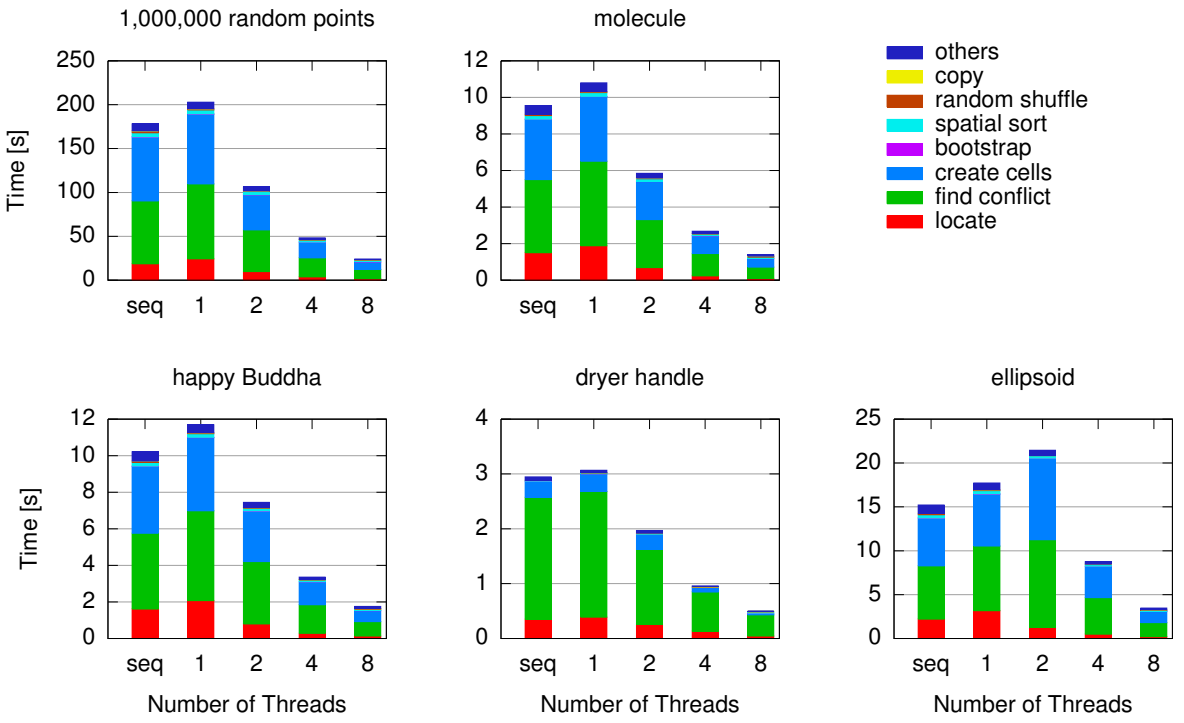


Figure 5.8: Breakdown of the running times for the sequential (seq) and parallel 3D Delaunay construction for 1, 2, 4 and 8 threads.

Future Work. The implementations will be incorporated into future versions of CGAL. Usage of the Delaunay triangulation may extend to the mesh generation algorithms in CGAL. Many other algorithms in the library await parallelization, we have just presented a starting point. The integration of the algorithms developed so far will also demand some software engineering choices on how to call the parallel variants. We can certainly learn from the libstdc++ parallel mode here.

There might be potential to benefit from parallelism on a lower level, too. For some CGAL packages, algebraic computations are needed, which can be quite costly. This is also the case when floating-point filters fail and the decisions get more complex. Parallelizing the predicates could lead to a significantly better performance, especially for slow cases, i. e. degenerate input.

6 Basic External Memory Algorithms

External memory algorithms process large amounts of data that do not fit into the main memory of a computer, but must be kept in *external memory*, i. e. on hard disks usually.

Several models of computation have been proposed for the external memory setting, which is also called *out-of-core* computing. The most applicable here being the parallel disk model introduced by Vitter and Shriver [VS94]. Many algorithms have been designed for this model [ABT04, VS94, DS03], which features the number of I/O operations as its major complexity metric.

However, implementing those algorithms is hard, involving a lot of low-level and platform-dependent details. Thus, at least the basic algorithms that are used as parts of more complex ones, should be provided in a library. The *Standard Template Library for XXL Data Sets (STXXL)* introduced by Dementiev et al. [DKS08] does just that. It allows easy implementation of algorithms based on the parallel disk model, providing data structures and algorithms, including a priority queue and a sorter. Its most basic data structure is the generic STXXL vector, a resizable random-accessible sequence of elements. Similarly to the MCSTL, it is also inspired by the C++ STL, however, the interface of the STXXL is not always fully identical, it is just similar to STL.

The number of disks cheaply attachable to a system and the bandwidth of each disk grows steadily. However, as stated before, the clock rate of microprocessors stagnates. On modern machines with multiple disks and multiple processor cores, the combination of the practical I/O performance of the STXXL and the I/O efficiency of the algorithms lead to a paradoxical situation. Many external memory algorithms are not I/O-bound, but actually *compute*-bound. The disks are idle and the CPU is fully loaded, while external memory algorithms were actually invented to overcome the problem of the CPU waiting for the slow disks.

Examples for this behavior include sorting, priority queues, breadth-first search, and suffix array construction [Dem06, pp. 48, 111, 163]. This situation was in desperate need for improvement. To overcome this steadily worsening limitation, we enable the use of multi-core parallelism for the STXXL. Our contribution is twofold.

Firstly, we consider the encapsulated STXXL algorithms and data structures in Section 6.2. By using the Multi-Core Standard Template Library (MCSTL), we achieve parallelization of the internal memory work of the algorithms. The same happens for the internal computation of the priority queue, which is a vital data structure for many external memory algorithms.

Secondly, we extend the pipelining framework of the STXXL in Section 6.3. The original purpose of pipelining in the STXXL was to save I/Os by passing data between algorithmic components without writing them to disk in-between. However, it also serves very well for modeling the data flow of external memory algorithms in general. By using our extensions to the framework, the developer can automatically exploit the task parallelism inherent in the algorithm pipeline.

Using synthetic and practical use cases, we show that this increases performance greatly.

6.1 Related work

Parallel external algorithms can be implemented by simulating parallel distributed memory algorithms. The number of PEs is virtually increased such that it fits into the memory associated with them. Since this amount does not fit into the actual available RAM, the states of the virtual PEs are saved to disk before switching processing to the next one. The results for sorting presented in [Rob10] can only compete for 8 cores versus the sequential STXXL.

The basic concept of pipelining is considered folklore in parallel programming. However, using it in this coarse-grained manner is not that common. There exist frameworks to support asynchronous pipelining, e. g. FG [CD04, DC05] and the Intel Threading Building Blocks [Int]. FG is targeted to distributed memory systems, though. The pipeline nodes process fixed-length blocks only, which is inflexible. Building non-linear graph structures is not the primary goal of FG, so it is quite inelegant. The Intel Threading Building Blocks library also supports task parallelism, but is not particularly well-suited for continuously streaming data, i. e. pipelining. The tasks are rather intended to calculate and return a single result.

As a conclusion, none of the frameworks mentioned fully satisfies our requirements.

6.2 Parallelizing STXXL algorithms using the MCSTL

The STXXL provides full genericity through the C++ template mechanism and adheres to concepts of the STL like iterators, functors, etc. Since the MCSTL and the STXXL exploit orthogonal characteristics of the hardware, namely parallel disks and parallel cores, they complement each other very nicely.

The STXXL provides several external memory data structures, the priority queue being the only one that is compute-intensive. In terms of algorithms, there is comparison-based sorting, integer sorting, random shuffling, and scanning (`for_each()`/`generate()`, and `find()`). Parallelizing these algorithms and data structures was done in two steps:

1. use STL routines instead of custom-written code, where not yet done
2. compile the code with the MCSTL

We describe the details for the comparison-based sort and the priority queue. Sorting is the most important nontrivial operation needed for processing huge data sets, and a major step of most external memory algorithms. For example, sorting can be used to build index data structures. A priority queue is also frequently used in this setting.

Efficient external sorting usually consists of two major steps: run creation and run merging. First, the data is sorted internally and written back to disk, run after run (one run filling the whole internal memory). Second, the runs are merged from disk to disk.

The run creation is done using `std::sort()`, which immediately enables parallel execution when compiled using the MCSTL. Choosing the in-place parallel quicksort variant instead of parallel mergesort avoids using additional internal memory.

The run merging step was revised to use `multiway_merge()` instead of a hand-coded loser tree structure. However, this was not trivial since I/O and computation should be overlapped. Disk access is strictly blockwise and prefetching is obligatory. Parallelization, however, requires a partitioning of the data, so that all input processed in one call to `multiway_merge()` must be available from the beginning, and no sequence may run empty. This leads to the following sub algorithm, which is executed as long as there are elements left.

1: **while** not all sequences empty **do**
2: find the minimum element *m* over all sequence maxima (regarding only the respective current block), let *s* be the sequence *m* is contained in
3: rank *m* in all sequences except *s* using binary search and calculate the number of elements *n* which can be processed before *s* runs empty
4: multiway-merge *n* elements
5: swap in the block that has been fetched as next block of *s* in the meanwhile and start prefetching the next block for *s*
6: **end while**

In a similar way, parallel versions of `sort()` and `multiway_merge()` were integrated into the priority queue implementation, which is based on sequence heaps [San00].

The part of our work described in this Section 6.2 is referred to as *STXXL Parallel Algorithms*. Its implementation has added only less than 10% in lines of code to the STXXL.

6.3 Task parallelism through asynchronous pipelining

In the following, we first describe the kind of pipelining that was supported by the STXXL previously (conventional pipelining). After that follow our augmentations that allow parallel execution based on task parallelism.

6.3.1 STXXL Conventional Pipelining

The STXXL supports passing data between algorithmic components without writing them back to disk in-between. This technique is referred to as *pipelining*. Pipelining is programmed by setting up a directed acyclic *flow graph*.

Nodes of the flow graph represent components that process data. The edges denote the flow of data, indicating its direction. Its structure is usually fixed in the source code, the type dependencies being resolved at *compile time*.

A node may either be a *scanning* node (drawn as square), a *sorting* node (drawn as funnel), or a *file* node (drawn as circle). They process *elements* of possibly different type and number. Scanning operations consume one element after the other, and emit zero or more elements (typically one per element read), but do all work *in-order*. Their actual functionality is defined by the developer, e. g. applying some kind of reduction on the elements, filtering elements out, or changing the elements themselves. The sorting nodes sort the elements with respect to some total order, storing them to disk temporarily. File nodes either read elements from disk and provide them to their respective successor, or consume elements from their predecessor and store them onto disk.

With these three kinds of nodes, most efficient external memory algorithms can be represented. To avoid random access, which is very costly, they rather sort and scan the elements in large chunks, which is I/O-efficient.

In the flow graph, elements can be *pulled* (drawn as solid arrows) or *pushed* (drawn as dashed arrows). For that reason, we have two interfaces for nodes. When its `operator*()` is called, a node implementing the *pull interface* (shortly called *pull node*) reads data from its predecessor(s), processes it, and returns the result to the caller. Checking whether the next element exists is done via `empty()`, advancing via `operator++()`. This is similar to the input iterator concept of the STL. Symmetrically, an element is passed to a node implementing the *push interface* (shortly called *push node*) via `push()`. It is then processed, and the result is usually pushed to its successor(s). A counterpart to `empty()` is not necessary, since a push node always has to accept more data. Conversely, we have to provide a call to notify the node of the end of data, namely `push_stop()`.

Branching a data stream can be achieved by a node pushing into additional successor nodes while being pulled. Joining is achieved by making a node pull multiple predecessors, before returning the (somehow combined) result.

A flow graph needs to have a *primary sink*, usually a file node. Execution of the pipeline is triggered by starting to fill the primary sink using the `materialize()` function call. In order to get data, the primary sink then pulls data from its predecessor in the data flow graph. The predecessor of the primary sink recursively pulls from its predecessor(s), and pushes to other successors, putting the whole pipeline into action, up to the source node(s). This traversal of the graph is basically a depth-first search.

To ensure that all parts of the flow graph are active, all nodes must be reachable from the primary sink by traversing pull edges backwards and push edges forwards. This implies (each node can have at most one pulling out edge) that there is exactly one primary sink in a valid flow graph.

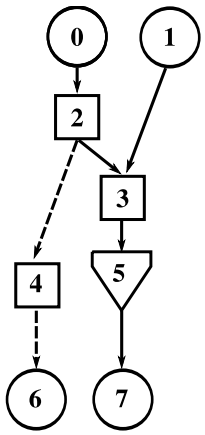


Figure 6.1: Basic example for STXXL pipelining.

We subsume all the introduced terminology using the example from Figure 6.1. Nodes 0 and 1 are (source) file nodes, while nodes 6 and 7 are (sink) file nodes. Node 7 is the primary

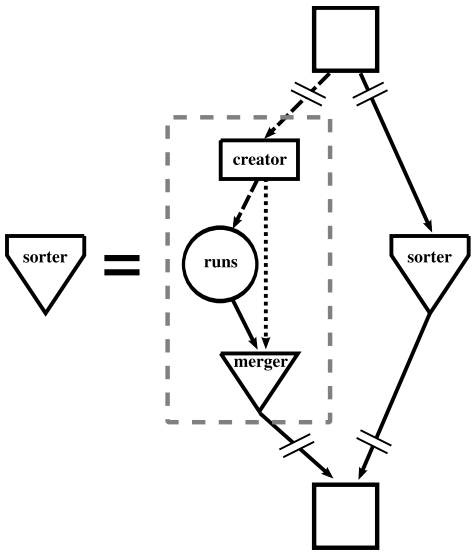


Figure 6.2: Split-up sorting node in context.

sink, while node 6 is not, since not all nodes can be reached starting from node 6 (in fact, no node at all can be reached). Nodes 2, 3 and 4 are scanning nodes. When node 2 is pulled by node 3, it pushes side-ways to node 4, which in turn pushes to file node 6. Node 3 pulls from both node 2 and node 1, combining the received data into a sequence that is sorted subsequently by node 5.

The glue code for setting up this example pipeline is given in Table 6.2, showing the difference to the original code in Table 6.1.

Sorting nodes (actually their run creation part) also form sinks, implying a `materialize()` step at node construction time. Therefore, in the conventional STXXL pipelining, the sorting nodes had to be split into a (pushed) run creator node and a (pulled) run merger node if there were multiple sorters fed from a common source. This construct is called a *split-up sorting node*, see Figure 6.2 for an illustration. Otherwise, the first sorting node constructed would have started pulling data before the other nodes were even constructed. In fact, this is already impossible to compile because it leads to a circular type definition.

6.3.2 STXXL Asynchronous Pipelining

Many algorithmic problems [Dem06] have been solved successfully using the pipelining approach. Therefore, it is promising to speed up things on a conceptual level.

There exist two opportunities for task parallelism in the context of pipelining. For a more intuitive understanding, assume that the nodes are ordered topologically from top to bottom, the source(s) at the very top, the sink(s) at the very bottom, data always streaming downwards.

Horizontal task parallelism: Nodes that are not connected (respecting the normal direction of the edges) do not have any data dependency. Thus, they can be executed in parallel.

Vertical task parallelism: Let scanning node N_1 be an (indirect) predecessor of scanning node N_2 with respect to the flow graph (respecting the normal direction of the edges). Then, there exists a data dependency between the two nodes. However, since the elements are accessed in order, both computations can be overlapped and thereby parallelized. Each node fetches a part of the input from its predecessor(s), processes it, and passes it on to its successor.

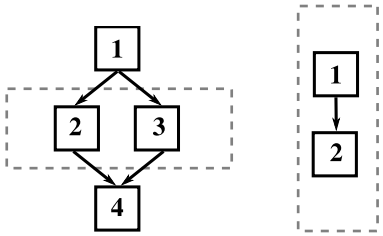


Figure 6.3: Horizontal (left) and vertical (right) task parallelism for nodes 1 and 2.

Figure 6.3 shows the two situations described.

We now supplement the STXXL pipelining framework with so-called *asynchronous* nodes, which can be used by the developer to decouple two algorithmic nodes by inserting an asynchronous node in-between. The asynchronous nodes process data by spawning a worker thread, communicating with their predecessors and successors to control the data flow. This automatically enables parallel execution in both cases stated above.

The asynchronous nodes come in three flavors:

- An *async pull node* pulls data from its predecessor(s) and processes it asynchronously in a worker thread, storing the resulting elements for availability to its successor.
- Symmetrically, an *async push node* gets elements pushed into. It processes and stores the resulting elements. Asynchronously, resulting elements are pushed into the successor.
- A *push-pull node*, finally, does not do work actively, but just synchronizes two nodes, one pushing into it, the other pulling from it.

To make this work sharing efficient, each asynchronous node needs two element buffers, an producer buffer and a consumer buffer. The producer buffer absorbs resulting elements (or just input elements for the push-pull node), while the consumer buffer is used for serving the successor node. Routines may block when there is no data readily available. When the producer buffer is full and the consumer buffer is empty, the two buffers are swapped, and operation can continue (see Figure 6.4). Waiting for these two events is done using OS-supported synchronization, so no CPU power is wasted. The buffer is needed to amortize the overhead for synchronization of the threads. Synchronizing for every single element would be far too time-consuming. Finally, the buffer size is a tuning parameter, blending these two extremes one element and all elements. Larger buffers amortize the synchronization overhead better, but decrease parallelism because of the implied latency.

```
// input from external vector, implicit source file node
typedef typeof(streamify(input.begin(), input.end())) input_node_type;
input_node_type input_node = streamify(input.begin(), input.end());

// compute checksum over all elements: scanning node
typedef checksummer<input_node_type> checksum_node_type;
checksum_node_type checksum_node(input_node);

// double the value of all elements: scanning node
typedef doubler<checksum_node_type> doubler_node_type;
doubler_node_type doubler_node(checksum_node);

// connection to sorting node
typedef sort<doubler_node_type, comparator_type> sort_node_type;
sort_node_type sort_node(doubler_node, comparator, run_size);

// output to external vector, implicit sink file node
materialize(sort_node, output.begin(), output.end());
```

Table 6.1: Source code for a simple linear STXXL pipeline.

```
// input from external vector, implicit source file node
typedef typeof(streamify(input.begin(), input.end())) input_node_type;
input_node_type input_node = streamify(input.begin(), input.end());

// compute checksum over all elements: scanning node
typedef checksummer<input_node_type> checksum_node_type;
checksum_node_type checksum_node(input_node);

// introduced async pull node
typedef pull<checksum_node_type> pull_node_type;
pull_node_type pull_node(buffer_size, checksum_node);

// double the value of all elements: scanning node
typedef doubler<pull_node_type> doubler_node_type;
doubler_node_type doubler_node(pull_node);

// connection to sorting node
typedef sort<doubler_node_type, comparator_type> sort_node_type;
sort_node_type sort_node(doubler_node, comparator, run_size);

// output to external vector, implicit sink file node
materialize(sort_node, output.begin(), output.end());
```

Table 6.2: Pipeline as above, with an async pull node added between the two scanning nodes. The differences are printed in bold-face.

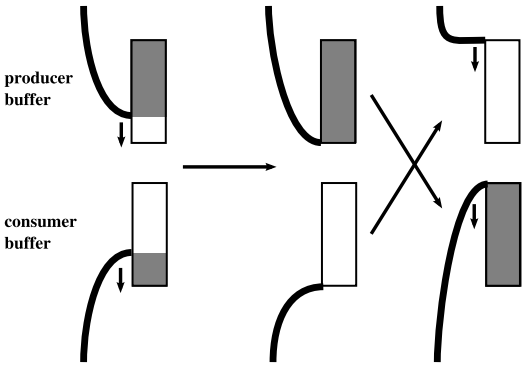


Figure 6.4: Swapping pipelining buffers.

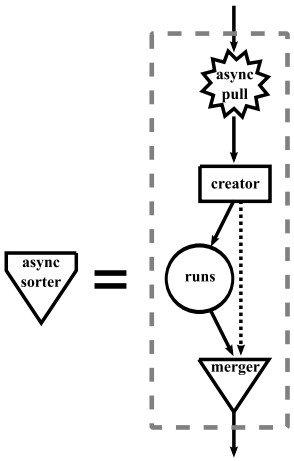


Figure 6.5: Equivalent for asynchronous sorting node.

Unfortunately, the insertion of asynchronous nodes cannot happen without changes to the source code of an already implemented algorithm. This is because of the very tight coupling of nodes at compile time, which is necessary for good performance[1]. Routines like `operator*()` immediately call the same routine of the predecessors, so it is impossible to insert asynchronous nodes automatically.

In addition to the asynchronous pipeline nodes, we also added overlapping of reading the input and sorting the runs for the pipelined sorter, which was a shortcoming of the existing version. This effectively adds an async pull node to each sorter, located right in front of the run creator (see Figure 6.5). Differently to a usual async pull node, this one's buffer is necessarily as large as a whole run size, since the run creation needs a complete run of data before it can start. By default, the merging is not done asynchronously, because this relatively cheap task combines well with subsequent scanning nodes. If the developer desires otherwise, he can just add an async pull node right after the sorting node.

[1]The compiler is able to fold several stages into one call using inlining, which is crucial for performance.

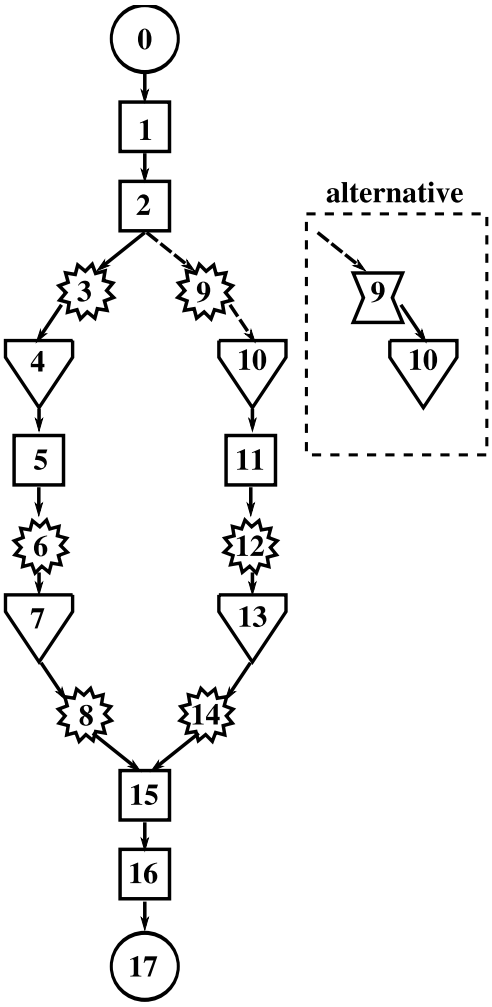


Figure 6.6: Diamond flow graph.

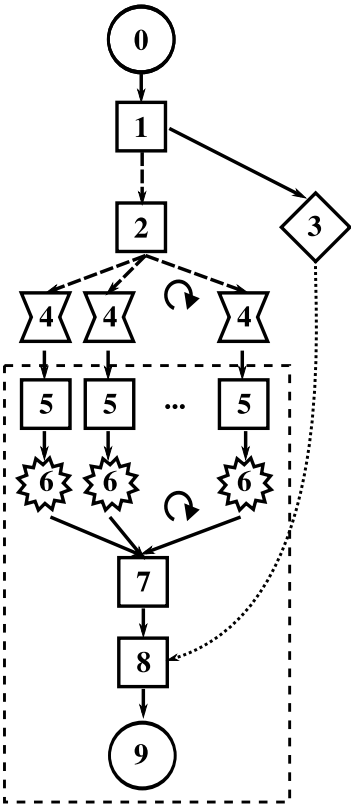
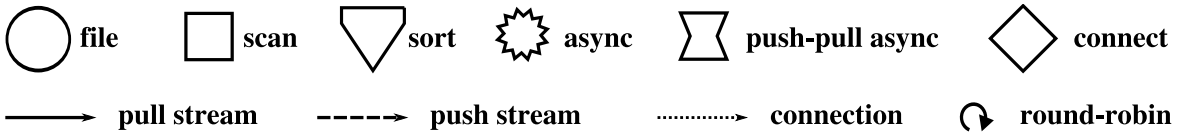


Figure 6.7: Distribute-Collect flow graph.



For an usage example see the *Diamond flow graph* in Figure 6.6, disregarding the asynchronous nodes (drawn as gear wheels) at first. This flow graph is an example for an external memory algorithm that can well exploit task parallelism. File node 0 contains a sequence of pairs. In node 2, the stream is doubled, after calculating a checksum in node 1. Nodes 4 and 10 sort by the second value of the pair, ascending and descending, respectively. Sorting by the first value of the pair is done in nodes 7 and 13, after taking the checksums in nodes 5 and 11, again. Node 15 rejoins the data into pairs of pairs, another checksum is taken in node 16. The Diamond flow graph is quite typical for a data flow graph since the sorting nodes, consuming most of the computational effort, are connected with each other by relatively cheap scanning nodes.

The conventional pipelining executes the paths between nodes 4 and 7 and between the nodes 10 and 13 *non-concurrently*, one after the other. To enable task-parallel execution of the program, the async pull nodes 3, 6, 8, 12, and 14, and the async push node 9 are introduced. In fact, the nodes 3, 6, 9, and 12 are integrated into their successive sorting nodes, by using async sorting nodes. Adding the others is very easily programmed, adding two lines of code per asynchronous node, and changing two more.

The following sets of nodes will run in parallel, in consecutive phases: *1st phase:* 0, 1, 2, 3, run creation part of 4, 9, run creation part of 10; *2nd phase:* run merging part of 4, 5, 6, run creation part of 7, run merging part of 10, 11, 12, run creation part of 13; *3rd phase:* run merging part of 7, run merging part of 13, 15, 16, 17. The number of phases is defined by the maximum number of sorting nodes on a path from a source to a sink (two in this case), plus one. All inherent task parallelism is automatically exploited.

More elegant flow graph design. As mentioned before, the conventional STXXL pipelining starts running a node already at its construction time. However, this can happen at a point in time where the flow graph is still incomplete. This is undesirable, since it inhibits horizontal parallelism, forces the usage of split-up sorting nodes, and also forbids analysis of the flow graph as a whole. Therefore, we introduce a `start()` call, which triggers processing, i. e. thread creation for the asynchronous nodes. It can be called after construction the whole flow graph.

The sorting nodes can be used symmetrically now, all being of the same generic type. Starting the nodes is done again in a depth-first manner starting at the primary sink, but only after the flow graph has been fully constructed.

This raises the question, whether in the course of adding asynchronous pipelining, we can completely abandon to *push* elements. Multiple consumers nodes could *pull* from an emitting node, thereby implicitly splitting the stream. However, this is ineffective for several reasons. First of all, the data emitted to different successors could be different in type, size, and/or volume. This would force the respective node to emit tuple types being the union of all requested outputs, each successor selecting its respective component. Hence, the required internal memory bandwidth would be unnecessarily high. Secondly, advancing the emitting node to the next element would have to be synchronized among all consuming nodes. Doing this for every single element imposes a huge system overhead, which renders asynchrony useless. Even synchronizing for a large batch of elements, as described in Section 6.3.3, would

not help without interconnecting a buffer. As a consequence, we have to keep pushing elements, at least into a push-pull node.

For the Diamond flow graph, a push-pull node (drawn as waisted square) could be inserted at position 9, while changing node 10 from having a pushed run creator to a usual sorting node. There are only negligible differences in execution between both alternatives. However, concerning programming, while the first one stays backward-compatible, i. e. the asynchronous nodes can be taken out without losing liveness, the second alternative is more symmetric and thus more elegant.

Round-robin task parallelism. Generating random input for the tests arose to be a bottleneck.

There, the computations can happen completely independently, the results feeding further nodes in arbitrary order, which can also happen in other circumstances (e. g. processing input from independent internal-memory sources, where intermixing of the produced elements does not harm). To enable parallelization in such a case, we added *distribute* and *collect* node types. While the former pushes elements to several nodes of the same type in a round-robin fashion, the latter collects them in the very same way, keeping the overall order. Executing each of the intermediate nodes by a separate thread parallelizes the computation.

This number of intermediate nodes can be configured at runtime, they will be replicated as necessary, making this approach easily scalable.

To make every node reachable from the primary sink with the edge directions stated above, a special *connect node* has to be inserted, see Figure 6.7 for an example (connect node 3 between nodes 1 and 8). The connect node has an additional outgoing edge to another node (drawn as dotted line) that starts the connect node before its actual predecessor(s). However, no data is transferred across the edge ever, it is only good for signalling. Instead of the distribute-collect construct, we also could have used an input node with a parallelized `generate()` call, but this would request a natively parallel random number generator.

The work described in this Section 6.3.2 is referred to as *STXXL Asynchronous Pipelining*.

6.3.3 Implementation details

Threading support. While the MCSTL nicely benefits from OpenMP for its data-parallel routines, fork-join parallelism is not sufficient for the pipelining's task parallelism. Here, we use POSIX threads.

Low-level issues concerning the pipelining layer. The pipelining approach can lead to constant-factor computation overhead due to high-level programming, i. e. passing only one element at a time from one node to the next. For small types, this streaming bandwidth can be even lower than the I/O bandwidth, which is unacceptable here because it contradicts speeding up internal computation. Therefore, we had to think about modifying the existing code in order to provide small-overhead streaming, while keeping the interface mostly backward-compatible.

As mentioned above, the conventional STXXL pipeline interface (for pulling) consists of the three methods `empty()`, `operator*()`, and `operator++()`, resembling an input

iterator interface. We extended this interface by a batch-wise access method. The functions `batch_length()`, `batch_begin()`/`operator[]()` and `operator+=()` do the same things as their counterparts, but for a chunk of elements at a time. `batch_begin()` returns a random-access iterator (a typed pointer in the best case), that allows efficient processing of many elements without calling back the predecessor node explicitly, in particular without calling `empty()` every single time. For backward compatibility to nodes that support only emitting one element at a time, the developer just inserts an asynchronous node which pulls individual elements, but allows batch access to the successor node. Symmetric methods exist for push nodes.

The asynchronous nodes support the batch methods particularly well using arrays as buffers.

6.4 Experiments

To evaluate the practical performance of our implementation, we conducted synthetic tests to measure the principal performance, as well as an algorithmic application of the library, to show the effect in a use-case.

We tested our programs on the XEON with 8 hard disks (about 72 MiB/s I/O each).

At the time of testing, GCC did not yet support nesting OpenMP parallelism in multiple OS threads[2]. Therefore, we used the Intel C++ Compiler 10.0.026 and 10.1.011 with optimizations switched on (-O3), accessing the GCC 4.2 STL implementation.

Synthetic tests.

We tested comparison-based sorting and priority queues, as well as the Diamond flow graph, comparing the conventional version of the STXXL against the incrementally improved ones. We took 100 GiB of input data (pairs of 64-bit unsigned integers), stored in an STXXL vector, and 1 GiB of internal memory for the run creation and run merge nodes. The asynchronous node buffers were equipped with 64 MiB each. This makes the asynchronous pipelining version take more memory than the conventional one, but allowing more memory would not speed the conventional pipelining up, since the run merging steps are non-recursive in either case.

Plain Sorting. As shown in Figure 6.8, the improvements in running time for sorting are decent. While a single thread takes 2 423 s to sort the data, the 4-thread version needs only 1 092 s. There is no further speedup for more threads, since the parallel sorting is in fact memory-bandwidth-bound from that point on, for this data type. Still, we get pretty close to the I/O-limited minimum running time of about 880 s, almost fully loading the machine. The speedups for smaller or harder to compare elements would be even better, since the bandwidths are the limiting factors.

The differences between a monolithic sort call and a sort call embedded in a pipelined flow graph are now reduced to few percents. The conventional pipelined sorter, which does not support overlapping of reading and sorting in the run creation step, clearly performs worse (2 953 s).

[2]This is supported as of version 4.4.

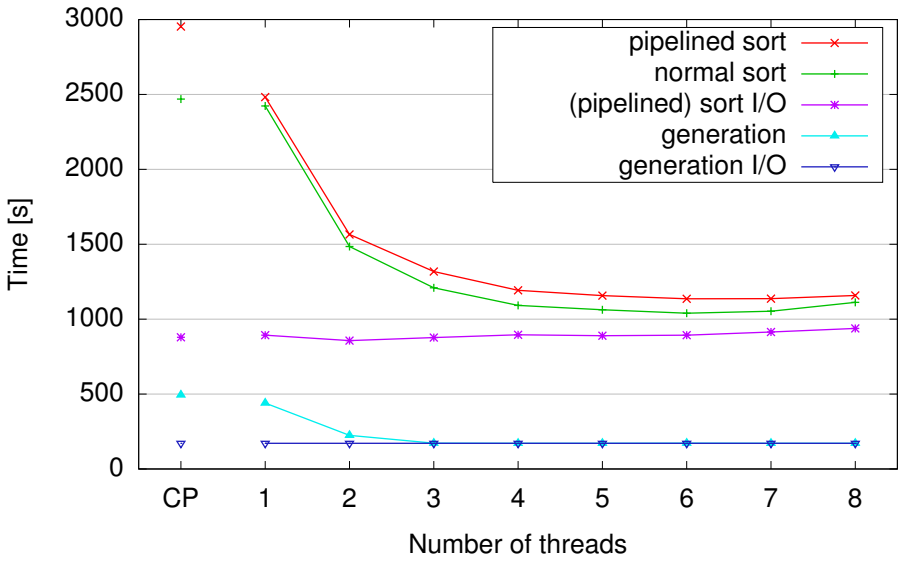


Figure 6.8: The I/O lines shows for how much time the I/O was active during execution.

Priority Queue. Priority queues are the essential part of external memory algorithms that use *time forward processing* [CGG+95, Arg95], often dominating processing time. Time forward processing uses a priority queue to gather data elements that are required for a future processing step by using an step identifier as key.

Our test case first increases the number of contained elements by calling (`insert()`, `delete-min()`, `insert()`) in a loop, until the priority queue size of 100 GiB is reached. Then, the queue is emptied again by calling (`delete-min()`, `insert()`, `delete-min()`), symmetrically. The inserted elements of size 8 byte are chosen to be a random amount larger than the element deleted last, i. e. we have a *monotone* priority queue, as is the case for time forward processing. We set an internal memory limit of 4 GiB.

Threads	1	2	4	8
Filling Phase	9 031	6 615	5 564	5 099
Removal Phase	5 260	3 921	3 291	3 068
Total	14 291	10 536	8 855	8 167
(speedup)		(1.36)	(1.61)	(1.75)

Table 6.3: Processing a priority queue with 8 byte elements, of maximum size 100 GiB. Timings are in seconds.

The result are stated in Table 6.3. Using only 2 threads, the total running time decreases from 14 291 s to 10 536 s, being equivalent to a 36% relative speedup. The speedup increases to 61% and 75% for 4 and 8 threads, respectively. This is quite impressive for a data structure handling such fine-grained requests.

input data	Source								Gutenberg							
MEM	1 024 MiB								2 048 MiB							
algorithm	doubling				DC3				quadrupling				DC3			
input size	2^{28}		2^{29}		2^{28}		2^{29}		2^{29}		2^{30}		2^{29}		2^{30}	
	[s]	S	[s]	S	[s]	S	[s]	S	[s]	S	[s]	S	[s]	S	[s]	S
CP	2 462	1	5 087	1	864	1	1 748	1	3 587	1	7 546	1	1 820	1	3 664	1
P	1 097	2.24	2 357	2.16	589	1.47	1 217	1.44	1 981	1.81	4 333	1.74	1 227	1.48	2 525	1.45
A	1 946	1.26	3 775	1.35	776	1.11	1 505	1.16	2 680	1.34	5 206	1.45	1 646	1.11	3 171	1.16
A+M	1 924	1.28	3 751	1.36	676	1.28	1 277	1.37	–	–	–	–	1 485	1.23	2 774	1.32
A+P	999	2.46	2 023	2.51	572	1.51	1 144	1.53	1 911	1.88	4 266	1.77	1 217	1.50	2 367	1.55
A+M+P	998	2.47	2 036	2.50	558	1.55	1 123	1.56	–	–	–	–	1 201	1.52	2 249	1.63

Table 6.4: Suffix array creation using different algorithm and library features. MEM gives the maximum amount of internal memory the algorithms were allowed to use for sorting and pipeline buffering, S is the speedup. This amount is shared by all concurrent sorters and asynchronous pipelines.

Generating random input by using distribute/collect. We consider the generation of random input data first. For each thread, there is an asynchronous node, pulling random numbers from another node. Those are collected in a buffer-wise round-robin fashion, similar as shown in Figure 6.7 (only the part inside the dashed rectangle is needed here). The running time decreases from 494 s of the conventional pipelining implementation (CP), to 171 s with 3 or more threads, being strictly I/O-bound from that point on.

Threads	1	3
CP	11 124	6 098
(speedup)		(1.82)
BP	10 944	5 965
(speedup)	(1.02)	(1.86)
PP	4 629	3 057
(speedup)	(2.40)	(3.64)

Table 6.5: Executing the Diamond flow graph with different STXXL and machine configurations. Input data size 100 GiB. Timings are in seconds.

Diamond flow graph. To evaluate how effectively data parallelism and task parallelism can join forces, we have run tests with the conceptual pipelined algorithm represented by the Diamond flow graph from Figure 6.6.

Executing the Diamond flow graph with CP, processing 100 GiB of data takes three hours on our machine (see Table 6.5). If we allow 3 threads for each sorting node (run creator) using the parallelized sort algorithm, the running time almost halves. Adding the batching (BP) improves the performance a bit more, for both the sequential and the parallel case. However,

when we add all the asynchronous nodes to gain full parallel pipelining (PP), many nodes can work simultaneously, dropping the running time to 4 629 s for one thread per sorting node, and finally to less than one hour (3 057 s) with three threads per sorting node. This corresponds to a speedup of more than 3.6 overall. The maximum number of *concurrent* threads is actually 8, two teams of 3 sorting the current runs, two more filling the upcoming runs. Since the very minimum I/O time is about 2 500 s, most of the potential speedup is attained.

6.4.1 Application test

We also evaluated our work with an algorithmic application, again suffix array construction, this time in the external memory setting.

Several variants had already been implemented using STXXL pipelining [DKMS05], namely doubling, quadrupling, and the DC3 algorithm as basically described in Section 4.6.1. We basically used the same input data sets, just longer prefixes of them.

The reference is the conventional pipelining (CP) of the STXXL. We added STXXL Parallel Algorithms (P), sorters with asynchronous reading (A), and their combination (A+P). Finally, we manually inserted additional asynchronous nodes into the algorithm and tested with and without STXXL Parallel Algorithms (A+M/A+M+P). The numbers of threads were chosen such that all cores were loaded. The results are stated in Table 6.4.

For the *doubling* algorithm, speedups up to 2.51 could be achieved by combining STXXL Parallel Algorithms and sorters with asynchronous reading. No additional improvements are gained by manually adding asynchronous nodes because of the simple structure (see Figure 6.9) of the algorithm. One iteration of the loop consists of a single straight forward path of a runs merger, full sorter and runs creator with two scanning nodes in-between. Therefore, the asynchronous nodes in the run formation part of the sorters already gain a maximum of task parallelism. The *quadrupling* algorithm is a variant of the doubling algorithm that processes larger elements in the same data flow graph (Figure 6.9). The scanning and sorting nodes require more internal work per element, but the number of iterations and I/Os needed to compute the suffix array is reduced, resulting in a faster algorithm [DKMS05]. The speedup achieved by using STXXL Parallel Algorithms and sorters with asynchronous reading is 1.88. Since we expected no improvements from it, manually adding asynchronous nodes has not been tested on the quadrupling algorithm.

In the *difference cover* (DC3) algorithm (see Figure 6.10), the data flows through three parallel paths between nodes 8 and 10, which contain sorters and benefit from running asynchronously. Additional asynchronous nodes are inserted after these sorters (node 9), running the merging steps in parallel, too, before the paths are joined in node 10. The pipeline shown here is actually instantiated recursively, reducing the number of elements by one third each time. The total speedup of DC3 is limited to 1.63 by the sequential scanning nodes, requiring more explicit parallelization. Overall, the DC3 algorithm is still the fastest, and also sped up significantly.

We also note that the combined strategies achieve better speedups for bigger inputs. In detail, the results show that each of the improvements to the STXXL is worthwhile, as it improves performance.

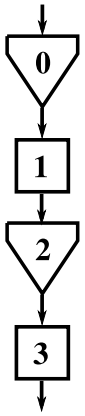


Figure 6.9: Doubling/ quadrupling flow graph.

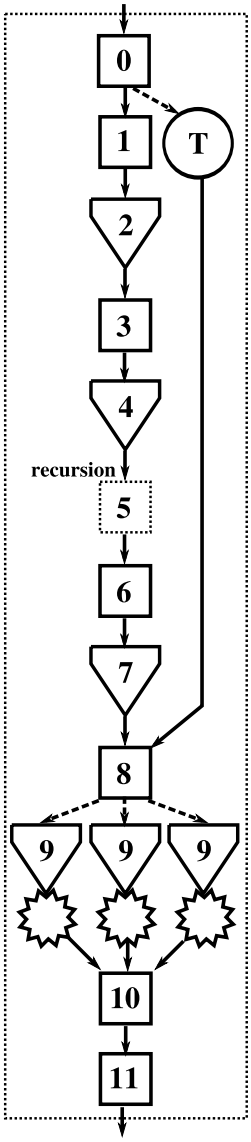
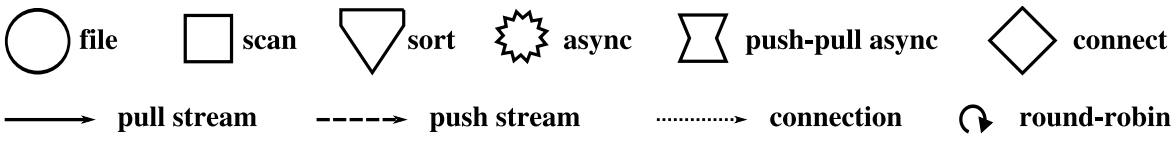


Figure 6.10: DC3 flow graph.



6.5 Conclusions and Future Work

We have shown how to catch up the margin that disks have gained over single-core processing power, by combining two libraries and adding task-parallel pipelining. STXXL Parallel Algorithms and STXXL Asynchronous Pipelining combine to *STXXL Parallel Pipelining*, enabling both data parallelism and task parallelism automatically. To our knowledge, there is no other library providing integrated support for parallel external computation. Our experiments, both synthetic and applied, have proven the potential of our approach, achieving considerable speedup in both synthetic and application tests. This is despite the fact that only little had to be changed in the source codes. Library users can benefit from the improvements very easily, even for existing code.

The results on the parallelization of the STXXL were published [BDS09] in 2009.

Future work. The memory assigned to each node (both working and asynchronous nodes) must still be manually configured by the developer. This could be done automatically with respect to a global memory limit. The nodes should be queried by a to-be-defined interface for their memory requirements and desires. Then, the available memory could be split in a fair way, using some optimization strategy. For this purpose, the flow graph would have to be segmented into the phases that have to run consecutively. The optimizer could also decide to run less nodes than possible in a phase, in order to have more memory available for critical nodes (e. g. in order to save a recursive merge pass for sorting).

Similarly, the allotments of threads to nodes could be handled more intelligently. So far, all parallel algorithm nodes are executed with a fixed specified number of threads. Again, by analyzing the graph and requesting information from the nodes about the effectiveness of more parallelism, the available processor cores could be used with best effect.

Although we discussed directed *acyclic* pipelining graphs here, with asynchronous nodes, having cyclic graphs is not off-limits any longer. A loop could be driven by an asynchronous node, without the need for a sink. Termination would be induced by a node emitting no further elements.

7 Distributed External Memory Sorting

There are currently two main ways to handle huge inputs in a cost-efficient manner: keeping most data externally on low cost hard disks, and clustering many inexpensive machines. We have considered the single machine case in Chapter 6. Combining *many* machines allows a relatively cheap cluster to handle huge inputs that would otherwise require high-end, power-hungry super computers with lots of internal memory. In this chapter, we propose new sorting algorithms for clusters where each node has multiple disks.

These algorithms are included in this thesis for two reasons. Firstly, they are a starting point for a generalization of the work presented so far to distributed memory. In fact, the implementation of the more practical algorithm enables the collective sorting of many STXXL vectors, each node storing one. Secondly, the implementation demonstrates once more the utilization of two of the considered libraries, namely the MCSTL and the STXXL.

Fundamental lower bounds [AV88] basically tell us that in order to sort inputs significantly larger than the *cumulative* main memory size[1] M, at least two passes[2] over the data are needed. For a node, we use the same external memory model as in Chapter 6 [VS94]. Then, up to M^2/B elements can be processed in two passes (refer to Table 7.1 for explanations of the symbols P, M, D, B, N, R used in this chapter).

Although there is a lot of previous work on parallel external sorting, the problem was not yet solved. In particular, algorithms used in practice can have very bad behavior for worst-case inputs, whereas all previous theoretical results lead to algorithms that need more than two passes even for easy inputs. Section 7.1 gives more details on the related work.

The two algorithms proposed here are based on the multiway merging paradigm again. In Section 7.2, we outline a conceptually simple variant of multiway mergesort that needs two passes even for inputs whose size is close to the theoretical limit for being sorted with two passes. However, this algorithm has a relatively large communication overhead, and it outputs the data in a globally striped fashion, i. e. subsequent blocks of output are allocated on subsequent PEs[3]. Therefore, in Section 7.3, we change the algorithm so that it needs very little communication, and outputs the data in a format more conventional in parallel computing, and more convenient for further processing: PE i gets the elements of ranks $(i-1)N/P+1,\ldots,iN/P$, i. e. the smaller the PE number, the smaller the elements. At least on the average, and up to usually small "clean

[1] To avoid cumbersome notation, we will also use M to denote the size of a run in external mergesort algorithms. Depending on details of the implementation, the actual run size might be a factor of up to four smaller[Knu98]. However, this difference has little effect on the overall performance.

[2] One pass comprises reading and writing the data once.

[3] In this chapter, a PE corresponds to a node of the cluster, which communicates via message passing. Shared-memory parallelism is exploited on a different level, using encapsulated algorithms from the MCSTL.

up" costs, this CANONICALMERGESORT algorithm needs only two passes and communicates elements only once, even for very large inputs. For $P = 1$, both algorithms are equivalent to the usual sorting algorithm as in Chapter 6. Section 7.5 gives experimental results on a careful implementation, which we describe in Section 7.4. These experiments show that the algorithm performs very well in practice. We summarize the results and outline possible future work in Section 7.6.

7.1 Related Work

Since sorting is an essential ingredient of most external memory algorithms, considerable work has been invested in finding I/O-optimal parallel disk sorting algorithms (e. g., [VS94, NV95, BGV97, HSV05]) that approach the lower bound of $2N/DB(1 + \lceil \log_{M/B} N/M \rceil)$ I/O operations for sorting N elements on a machine with D disks, fast memory size[4] M and block size B. The challenge is to avoid getting a base M/DB for the logarithm that spoils performance for very large systems. An important motivation for this work is the observation that a large D only makes sense in a system with many PEs. Although [VS94, NV93, AP94] develop sophisticated asymptotically optimal parallel algorithms, these algorithms imply considerable constant factors of overhead with respect to both I/Os and communication compared to the best randomized sequential algorithms [BGV97, HSV05, DS03].

Many external memory algorithms for distributed-memory machines have been proposed (e. g., see the references in [Ric94]). One of the most successful ones is NOW-Sort [ADADC+97] which is somewhat similar to our algorithm CANONICALMERGESORT, i. e. it sorts up to $M^2/(PB)$ elements in two passes. However, it only works efficiently for random inputs. In the worst case, it deteriorates to a sequential algorithm since all the data ends up in a single PE. This problem can be repaired by finding appropriate splitter keys in a preprocessing step [MRL98]. However, this costs an additional scan of the data and still does not result in exact partitioning.

[4] All sizes are given in number of elements.

Table 7.1: Symbols used in this chapter. We generally omit trivial rounding issues when dividing these quantities.

Resource/Number	Symbol
#PEs (#nodes)	P
internal memory (in #elements)	M
#disks overall	D
block size (in #elements) in the EM model	B
#elements	N
#runs	R

In [Raj04], a merge-based parallel external sorting algorithm is proposed that is inspired by parallel mesh algorithms. This algorithm needs at least four passes over the data.[5]

In [CC06] an algorithm based on column-sort is proposed that sorts up to $(M/P)^{3/2}/\sqrt{2}$ elements using three passes over the data. Using one additional pass, the input size can be increased to $\max\left(\mathrm{O}\left(M^{3/2}, (M/P)^{5/3}\right)\right)$ elements. It is instructive to use some realistic numbers. On current machines it is quite realistic to assume about 2 GiB of RAM per core. Using this number, the amount of data that can be sorted with the three pass algorithm is limited to inputs of size around $2^{31^{3/2}}/\sqrt{2} = 2^{46}$, i. e. about 64 TiB, regardless of the number of available PEs.

In [DDHM02], a general emulation technique for emulating parallel algorithms on a parallel external memory machine is developed. It is proposed to apply this technique to a variant of sample sort. This results in an algorithm that needs five passes over the data for sorting $\mathrm{O}\left(M^2/(PB)\right)$ elements. An implementation is given in [Rob10], but as explained in Section 6.1, it cannot compete in practice.

7.2 Mergesort with Global Striping

Since multiway mergesort proved a good algorithm for parallel disk external sorting and parallel internal sorting, it is a natural idea to use it also for parallel external sorting. Here we outline how to do this in a scalable way: The first phase is *run formation* where initial runs of size M are loaded into the *cumulative* memory of the parallel machine, sorted *in parallel*, and written back to disk.

Next follow one or more merging phases[6] where up to $k = \mathrm{O}(M/B)$ sorted runs are merged in a single pass. The challenge is that we are only allowed a constant number of buffer blocks in internal memory for each run. In particular, we may not be able to afford k buffer blocks on *each* PE. We solve this by fetching a batch of $\Theta(M/B)$ blocks at a time into the internal memory (those blocks that will be needed next in the merging process), extracting the $\Theta(M)$ smallest unmerged elements using internal parallel merging, and writing them to the disks. Fetched elements that are larger than the smallest unfetched elements are kept in internal memory until the next batch. Note that this is possible since by definition of the blocks to be fetched, for each run, at most B elements remain unmerged. Note that we could even afford to replace batch merging by fully-fledged parallel sorting of batches without performing more work than during run formation.

The difficult part is how to do the disk accesses efficiently. However, this can be done in an analogous fashion to previous (sequential) parallel disk sorting algorithms [BGV97, DS03, HSV05]. The runs and the final output are *striped* over all disks, i. e. subsequent blocks are allocated on subsequent disks. This way, writing becomes easy: We maintain D buffer blocks. Whenever they are full, we output them to the disks in parallel. Efficiently fetching the data is more complicated. A *prediction sequence* consisting of the smallest element in each data block

[5] This bound is derived from the bounds in the paper assuming that logarithms with fractional values have to be rounded up.

[6] In general we need $\lceil \log_{\Theta(M/B)} \frac{N}{M} \rceil$ merging phases.

can be used to predict in which order the data blocks are needed during merging [Knu98, DS03]. Using randomization, some buffer space, and appropriate prefetching algorithms, it is then possible to make good use of all disks at once. The only part of this algorithm that is not straight-forward to parallelize is the prefetching algorithm. In the following Section 7.2.1, we outline an efficient prefetching algorithm for the cases $B = \Omega(\log P)$ and $M = \Omega(DB \log D)$.

We believe that the above algorithm could be implemented efficiently. However, it requires a substantial amount of communication: During run formation, all the data has to be communicated in the parallel sorting routine, and again for writing it in a striped fashion. Similarly, during a merging pass, the data has to be communicated during internal memory multiway merging and for outputting it in a striped fashion. Thus, we need 4 communications for two passes of sorting. In Section 7.3, we want to bring this down to a single communication, at least in the best case.

7.2.1 Prefetching Details

Perhaps the easiest way to do prefetching is to simply use the order in which the blocks are needed for merging, which is determined by the order of the smallest[7] keys in the data blocks. Unfortunately, it is open whether this leads to optimal I/O rates unless $\Omega(D \log D)$ prefetch buffer blocks are available [BGV97]. However, in [DS03], very good performance is observed for random inputs. In [DS03, HSV05], an optimal prefetching algorithm is used that is efficient already for $\Omega(D)$ buffer blocks. This algorithm is based on simulating a buffered writing process which iteratively fills a shared write buffer, and then simulates the output of one block on each disk whose queue contains a write request. We can use the sequential algorithm from [HSV05] as long as $B = \Omega(P)$ since we only perform constant work for each data block. Parallelizing this algorithm is possible but requires relatively fine-grained coordination: Allocating $O(D)$ blocks to disk queues can be implemented using plain message passing since randomization ensures that the number of blocks per queue is at most $O(\log D / \log \log D)$. Simulating the outputs can be done locally on each PE. But then we have to count how many write queues are nonempty, which requires a global sum-reduction. This is possible in time $O(\log P)$ on distributed-memory parallel machines.

7.3 CANONICALMERGESORT

In the following, we describe a variant of parallel external mergesort that produces its output in way more canonical for parallel processing – PE i gets the elements of ranks $(i-1)N/P + 1, \ldots, iN/P$, and this data is striped over the local disks. This is not only more useful for applications but it also reduces the amount of communication to a minimum, at the price of possibly some additional I/Os.

In the first phase, $R = N/M$ global runs of size M are created, the last run possibly being smaller (Section 7.3.1 gives more details on this phase). This is similar to the algorithm of

[7] In [Knu98] and a lot of subsequent work, the *largest* key in the *previous block* is used. Here we use the approach from [DS03] which is arguably more elegant and allows the merger to proceed for slightly longer without having to wait for a block from disk.

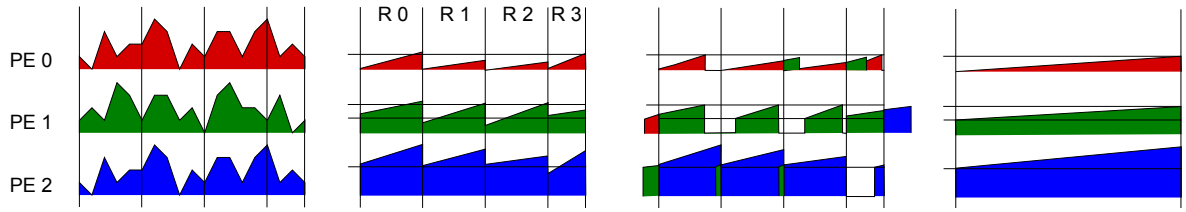


Figure 7.1: Schema of CANONICALMERGESORT: On the very left the initial input situation, going to the right the results of the three phases: run formation, redistribution, local merging. The y-axis denotes the PEs and the element rank, the horizontal lines illustrate the global splitters.

Section 7.2, but now, the output is not striped globally over the disks, but striped over the local disks only, saving communication. Moreover, if all runs have a similar input distribution, most elements will already end up on the PE where they are supposed to be for the globally sorted final output. In order to make this assumption approximately true even for bad inputs, each PE chooses the participating blocks for each run randomly. This is implemented by randomly shuffling the IDs of the local input blocks in a preprocessing step. For each run, we need $O(M/P \log M)$ internal computation, $O(N/P \log M)$ in total.

In the second phase, distributed multiway partition operations (as defined in Section 2.1) are performed, where each distributed run is regarded as a sorted sequence (see Section 7.3.2 for the algorithm). Each PE $i \neq 0$ selects for each run the first rank it is supposed to contain in the final result, resulting in $P-1$ splitter positions per run. After communicating the splitter positions to PEs i and $i-1$, every PE knows the elements it has to receive. The partitioning algorithm performs only negligible I/O.

The data is then redistributed accordingly, using a global external all-to-all operation as described in Section 7.3.3. As argued before, most of the data usually is on the right node already, so the all-to-all operation is expected to take only little time. Nevertheless, a detailed analysis on the amount of data to be transferred is given in Section 7.3.5.

In the third phase, the data is merged node-locally. Each element is read and written once, no communication is involved in this phase. The internal computation amounts to $O(N/P \log R) = O(N/P \log N/M)$.

Overall, we need $O(N/P \log N)$ internal computation, with a very low constant factor. For not too large data sets, CANONICALMERGESORT needs 2–3 passes over the data, depending on the severity of the redistribution in the second phase. An illustration of CANONICALMERGESORT is shown in Figure 7.1. The following subsections will explain the involved details.

7.3.1 Internal Memory Parallel Sorting

In the first phase, we sort in parallel on distributed memory, but only data located in the collective internal memory. We use a distributed-memory implementation of the parallel multiway merging approach described Section 2.3. The multiway partitioning needed for that in the distributed memory setting is described in the next subsection, the I/O problems not applying here. An all-to-all communication is used to move the pieces to the right PE before

merging them there. Note that in the best case, this phase is the only time when the whole data is actually communicated in CANONICALMERGESORT.

The approach "sort locally, multiway partition and redistribute, merge" resembles the overall external memory sorting algorithm, so we have a self-similarity here.

7.3.2 Multiway Partitioning

We defined multiway partition in Section 2.1. Here, we apply it to the R runs as the sorted sequences. Since each of the runs is distributed over all nodes, accessing an element can be quite expensive. A request is sent to the containing node, this node reads the data from disk if it is not cached, and sends it back. Because of this skewed relation between element access and internal computation, we use a slightly different algorithm for multiway partitioning than before.

Let the length of the sequences M be a power of two, round up and (conceptually) fill up with ∞ otherwise. We maintain approximate splitter positions that are moved in steps of size s. The basic algorithm uses initial splitter positions 0 and step size $s = M$. Within a *round*, the splitter corresponding to the smallest element is increased by s until the number of elements to the left of the splitters becomes larger than the desired rank r. Then, s is halved and the splitters corresponding to the largest element are decreased by s while the number of elements to the left of the splitters is still larger than r. This process is repeated until $s = 1$. After at most $\lceil \log_2 M \rceil$ rounds, the process terminates. Since in each half round every splitter is touched at most once, the overall number of sequence elements touched is $O(R \log M)$ and the total execution time is $O(R \log R \log M)$ using a priority queue for identifying the sequences to be touched.

In phase two of CANONICALMERGESORT, PE i runs multiway partitioning for rank $r = iN/P$. Although these operations can run in parallel, they have to request data from remote disks and thus the worst case number of I/O steps is $O(RP \log M)$, when a constant fraction of request is directed to a single disk.

However, this bottleneck for large P is greatly reduced by the randomization used during run formation. Furthermore, during run formation, we store every K-th element of the sorted run as a sample (for some parameter K) in a reserved share of the internal memory. During multiway partitioning, this sample is used to find initial values for the approximate splitters. As a third optimization, we cache the most recently accessed disk blocks to eliminate the $R \log B$ last disk accesses. In our practical experiments, the resulting algorithm takes only negligible time.

In [RSS09, Appendix B], we analyze a slightly more complicated variant that provably scales to very large machines, still using only very little time.

7.3.3 External All-to-All

In an *all-to-all* operation, each PE sends and receives different amounts of data to/from all other PEs. Compared to the ordinary all-to-all operation provided we are facing two problems.

First, each PE might in general communicate more data than fits into its local memory. We solve this problem by splitting the external all-to-all into k internal memory suboperations by logically splitting the data sent to a receiver into k (almost) equally-sized parts. The choice of k depends on the available internal memory but will be at most $O(R)$.

The second problem is that the data to be sent from each node to another node has to be collected from R different run parts. We therefore assemble the submessages by consuming all the participating data of run i before switching to run $i+1$. This way, each PE j needs only a single buffer block for each PE that it sends data to. Note that due to randomization, the number P' of required buffer blocks will usually grow much more slowly than the worst case of $P-1$ communication partners.

7.3.4 Analysis Summary

The easiest summary of the analysis is that CANONICALMERGESORT needs I/O volume $4N + o(N)$, communication volume $N + o(N)$, and local work similar to a fast sequential internal algorithm. Here, the "$o(\cdot)$"-notation expresses that the overheads are independent of the input size N or only grow sublinearly. A little more care must be taken however, since these bounds only hold under a number of assumptions on the values of the machine parameters, namely P, M, B, and D. To simplify matters a little bit, we assume that $D = \Theta(P)$. We also introduce the shorthand m for the local memory size M/P. For example, we have a machine with $P \in 1..200$ nodes (with 8 cores each), $D = 4P$, $m = 2^{34}$ byte, and $B = 2^{23}$ byte.

The theoretically most important restriction is that the maximal amount of data that can be sorted is $O\left(M \cdot \frac{M}{PB}\right) = O\left(P\frac{m^2}{B}\right)$. This is a factor $\Theta(P)$ less than for the globally striped algorithm from Section 7.2, since *every* PE must be able to hold one buffer block from each run in the merging phase. However, $P\frac{m^2}{B}$ is P times the amount that can be sorted by a single PE, which sounds very reasonable. In particular, any single PE equipped with a *reasonable* amount of RAM and disks can sort the complete content of these disks in two passes since for technological reasons, the price ratio between one byte of disk space and one byte of RAM has always been bounded by a few hundred. In this sense, the CANONICALMERGESORT is sufficiently scalable.

The second most important restriction is that even the randomized algorithm cannot move all the data to the right PE already during run formation. In Section 7.3.5, we show that this amount of data remains asymptotically small if $m \gg PB \log P$ (and the factor $\log P$ may be an artifact of the analysis), i. e. each PE must be able to store some number of blocks for each other PE. This assumption is reasonable for the medium-sized machine we have used for experiments, and for average case inputs, the B disappears from the restriction, leading to an algorithm that scales even to very large machines with many thousands of PEs. For very large machines and worst case inputs, our algorithm degrades to a three-pass algorithm which is still a good result.

A similar restriction on the local memory size applies to the external all-to-all algorithm from Section 7.3.3 – each local memory must be able to hold a constant number of blocks for each other PE. However, randomization will mitigate this problem so that this part of the algorithm will scale to very large machines.

Another similar restriction applies to the multiway selection algorithm described in Section 7.3.2 whose naïve implementation is only efficient if $m \gg PB \log M$. There, our more clever implementation with sampling and caching basically eliminates the problem.

7.3.5 Analysis of Data Redistribution for CANONICALMERGESORT

We analyze the amount of data that is moved in the second phase of CANONICALMERGESORT, the all-to-all. The general rule is that the more different the runs look, the more data must be moved. Firstly, we give asymptotic bounds which are unlikely to be exceeded. Secondly, we estimate expected values, including constant factors, and compare to experiments.

Asymptotic Bounds

We use the bounded difference inequality [McD89] which bounds how any function f of independent random variables is concentrated around its mean, provided that changing a single variable does not change the function value by too much. In our case, we have M/B random variables that are used to determine the blocks sorted by a particular run j.[8] The function f we consider is the global rank of the smallest element of run j that is stored on PE i. The expectation of f is iM/P. The deviation from this rank is proportional to the amount of data from run j to be moved to PE i. By changing a single random variable, the value of f changes by at most the block size B. We get

$$\mathbb{P}[f - \mathbb{E}[f] \geq t] \leq \exp\left(-\frac{t^2}{2\frac{M}{B}B^2}\right) = \exp\left(-\frac{t^2}{2MB}\right), \tag{7.1}$$

i. e. it is unlikely that the more than $O\left(\sqrt{MB}\right)$ elements have to be moved per run. However, we have to be a bit careful since the running time of a parallel algorithm depends on the PE where things are worst. Equation (7.1) also shows that it is unlikely that *any* PE has to move more than $O\left(\sqrt{MB\log P}\right)$ elements for run j. Since we are *really* interested in the worst sum of data movements over all runs, the truth lies somewhere in the middle. Anyway, $O\left(R\sqrt{MB\log P}\right)$ is an upper bound for the expected amount of data movement to/from any PE. This is small compared to the total data movement per PE of N/P if $M/P \gg PB\log P$, i. e. each PE must be able to store $\Omega(P\log P)$ data blocks (and the $\log P$ factor is probably overly conservative). This is a reasonable assumption for small and medium P but does not hold for very large machines.[9] We also see that the reorganization overhead grows with the square-root of B. Figure 7.3 supports this claim, where a quadrupling of the block size leads to about a doubling of the I/O volume. Hence, on large machines, it might pay to use a smaller block size for reading blocks during run formation. Note that the smaller block size affects less than one fourth of the I/Os, namely the ones reading in the merge phase, which are truly random. During run formation, we can use offline disk scheduling techniques to reduce seek times and rotational delays.

Average Case. By setting $B = 1$, we get a bound on the data movement for random inputs, i. e. for a random permutation of distinct elements. For low data movement, we need the condition that $M/P \gg P\log P$, i. e. every PE must be able to hold a logarithmic amount of data

[8] Note that these block indices are *not independent*. However, if we determine the blocks to be used by generating local random permutations the standard algorithm for determining random permutations uses M/B independent random values for determining the blocks used in a single run.

[9] For example, our nodes can hold only about 2 000 of the very large blocks we are using.

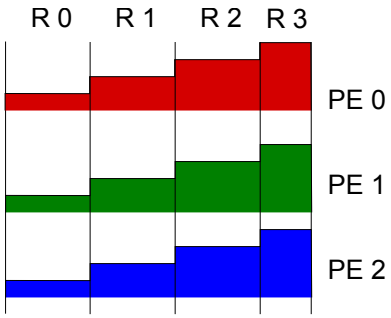


Figure 7.2: Bad input for CANONICALMERGESORT.

for every other PE in its local memory. This is a very mild condition. Indeed, internal memory parallel sorting algorithms that work with a single communication of the data have a similar condition.

Practical Estimations

Without randomization in the first phase, the following input distribution is a bad input with respect to the data movement in the second phase. Let the elements be identical to their global rank, i. e. we sort the elements $1,\ldots,N$. Each PE i hosts the N/P elements $i, i+P, i+2P, \ldots, i+(N/P-1)P$. Then, the run j contains elements from only $[jM,(j+1)M)$, but its elements are equally distributed over all PEs, instead of being concentrated on PEs $[\lfloor jP/R\rfloor, \lceil (j+1)P/R\rceil]$. So almost all data contained in this run, $((P-1)/P)M$ elements, must be moved, which is $((P-1)/P)N$ in total, quickly converging to all elements for larger P. To give an intuition, we show a simplification of this input in Figure 7.2, all elements of a run being equal.

Let us see how this behavior is diminished by choosing the blocks in a randomized order in the first phase. We make a number of simplifying assumptions in order to arrive at an estimate of the behavior of the algorithm.

First, we do the calculation for individual elements. Later, an element will be replaced by a block containing a consecutive interval of equal elements with the corresponding blown-up ranks. This then directly states the result in terms of blocks, which is our aim.

Let $m = M/P$ be the number of elements per run per PE, again. We regard one run at a time, since after an element is sorted in a certain run, it is clear whether it is in the right place or not. Since choosing the blocks is not independent among the runs, this is a simplification, but acceptable for large R.

An element of run-local rank $e \in \{1,\ldots,M\}$ is in the right place after the first phase if its position is in-between the bounds $\ell = \lfloor e/m\rfloor m$ and $u = \lceil e/m\rceil m$ (i. e. round e with granularity m). Its expected rank is e, and the distribution is binomial, since the rank is the same as the number of elements chosen that are smaller than e. The probability of being smaller than e is just e/M, so we have the binomial distribution $B(M, e/M)$. Since M is large, we can approximate the binomial distribution by the normal distribution $\mathcal{N}(e, M\cdot e/M\cdot(1-e/M)) = N(e, e\cdot(1-e/M))$, which is symmetric.

From now on, as further simplification, we regard the middle PE with number $\lfloor P/2 \rfloor$, for which the expected data movement is maximal. This can be realized by multiplying $e \cdot (1 - e/M)$ by the "constant" M, which shows that the variance is maximal around $e = M/2$. The standard deviation σ of the above normal distribution does not vary much for the middle PE, since e is in the range $[\ell, u]$, which is small (m) compared to the absolute value (the relative deviance is $1/P$), and also, there is a square root involved. Thus, we take the same $\sigma = \sqrt{e' \cdot (1 - e'/M)}$ for all elements, where $e' = (\ell + u)/2$ is the average position. This has the advantage that all the distributions have the same shape now, but are just shifted according to e.

The probability that e is close enough to its genuine rank is

$$\sum_{i=\ell}^{u} \mathcal{N}(e, \sigma)(i - e). \tag{7.2}$$

Summing up over all elements that should be placed the middle PE, this amounts to

$$\sum_{e=\ell}^{u} \sum_{i=\ell}^{u} \mathcal{N}(e, \sigma)(i - e) = (m+1)\mathcal{N}_0(0) + 2 \sum_{k=\ell}^{u} (m + 1 - k)\mathcal{N}_0(k) \tag{7.3}$$

elements being in the wrong place, where $\mathcal{N}_0 := \mathcal{N}(0, \sigma)$ and $m = u - \ell$. This formula is sufficiently simple to evaluate it numerically.

The reader can assure himself of the good coincidence of theory and practice — despite the numerous approximations involved — in Figure 7.4. We have an overestimation for small P, since PE number $\lfloor P/2 \rfloor$ is "far" from the middle. For larger P however, the measured values spread around the expected value. Decreasing the block size always improves the overhead, as predicted.

The I/O volume *averaged over all PEs* is indeed smaller, which is shown in Figure 7.3. However, the overall running time is dominated by the slowest node, i. e. the one doing the most I/O.

For random input data, there is no problem anyway, the overhead is less than one percent, being equivalent to a block size of $B = 1$.

7.3.6 Practical Aspects

Hierarchical Parallelism. In this chapter, a PE is a node of the cluster, i. e. defined with respect to communication. In practice, a node has multiple processors/cores and multiple disks. We exploit this *local* parallelism, too. The blocks on a PE are striped over the local disks. For complex operations like internal sorting and merging, the shared-memory parallel algorithms from the MCSTL are used. Those are integrated similarly as in Chapter 6, except that they have to be further intertwined with communication, making a direct call of the parallelized STXXL impossible.

Therefore, we exploit *hierarchical* parallelism. Taking each processor core as a PE would lead to a larger number P, while keepping M and D the same, negatively influencing some of the stated properties of the algorithm. In other terms, we benefit from the fact that communication

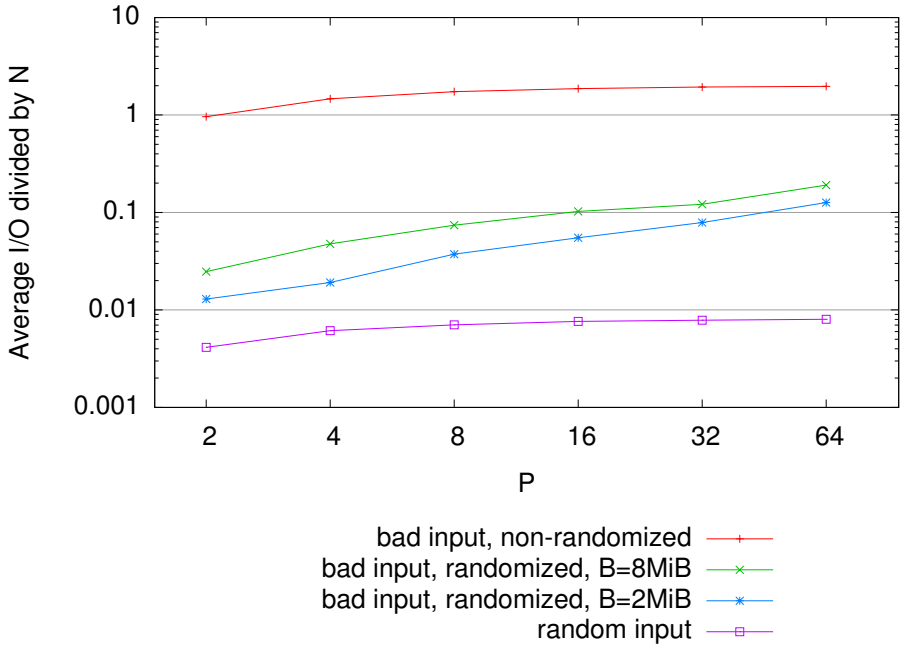


Figure 7.3: I/O volume for the all-to-all phase for different inputs with/without randomization, average over all nodes.

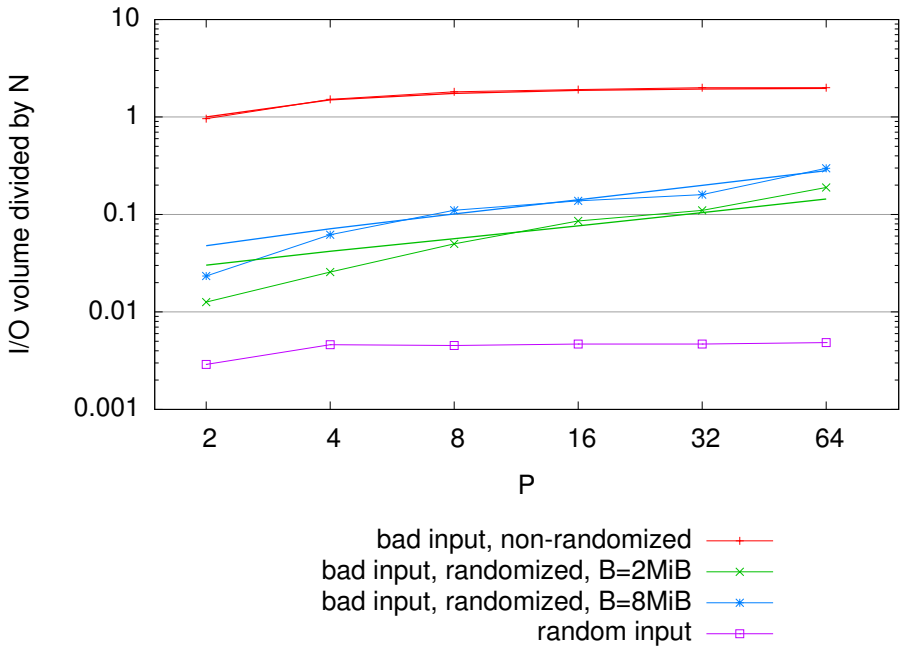


Figure 7.4: I/O volume for the all-to-all phase for different inputs with/without randomization, for the likely most affected middle node. The solid lines give the theoretical estimations.

from any disk to any core of the machine is equally fast, since it passes the shared memory anyway.

(Nearly) In-Place Operation. The run formation in phase 1 can be done in-place easily. The data is written back to the blocks where it was read from.

In phase 2, the multiway selection does not need considerable extra space. The subsequent external all-to-all operation has a certain overhead since in each round it receives P' pieces of data, which may lead to partially filled blocks. Since there is not sufficient internal memory to buffer all this data, these partially filled blocks have to be written out to disk. Also, the in-place global external all-to-all needs $P+1$ more blocks, leading to a total temporary overhead of RP' blocks per PE. However, this is usually proportional to N by a small constant factor.

For local merging in phase 3, blocks that are read to internal buffers are deallocated from disk immediately, so there are always blocks available for writing the output.

7.4 Implementation

We have implemented CANONICALMERGESORT in C++ on top of the STXXL. The program is called *DEMSort*, meaning *Distributed External Memory Sort*. Each node passes an STXXL vector as input, which after sorting contains the respective part of the globally sorted sequence. In fact, the parts on the nodes can be of different length, allowing for static load balancing, e. g. due to differing disk capacity or computational power. In the end, each node receives as many elements as it has provided in the beginning.

We use the STXXL also for asynchronous block-wise I/O to the multiple disks. To sort and to merge data internally we used the parallel mode of GCC 4.3.1. Communication between nodes is done using the message passing interface MPI [MPI94], we used MVAPICH 1.1 here. Unfortunately, in MPI, data volumes are specified using 32-bit signed integers. This means that no data volume greater than 2 GiB can be passed to MPI routines. We have re-implemented `MPI_Alltoallv` to break this barrier.

The program was compiled using GCC 4.3.1 including the respective parallel mode.

Overlapping. For run formation, we overlap internal computation and communication with I/O. While run i is globally sorted internally (including communication), the nodes first write the (already sorted) run $i-1$ before fetching the data for run $i+1$.

As a special optimization for inputs that do fit into the global internal memory, i. e. $N \leq M$, we also overlap in this single-run case: Immediately after a block is read from disk, it is sorted, while the disk is busy with subsequent blocks. When all blocks are read and sorted, the algorithm only has to merge the blocks instead of still sorting them entirely, which is faster.

We could also use overlapping of internal computation and communication in the internal global sort, splitting up the internal sort into three phases: local internal sort, global distribution, local internal merge. However, our current implementation does not yet support this. However, performance gains might be limited in practice, since all three operations still share the memory bandwidth.

7.5 Experimental Results

We have performed experiments on a 200-node cluster running Linux kernel 2.6.22. Each node consists of two Quad-Core Intel Xeon X5355 processors clocked at 2 667 MHz with 16 GiB main memory and 2×4 MiB cache. Apart from a slightly higher clock rate, the nodes are very similar to XEON. The nodes are connected through a 288-port InfiniBand 4xDDR switch, the resulting point-to-point peak bandwidth between two nodes is more than 1 300 MB/s full-duplex. However, this value decreases when all nodes are involved, because the fabric gets overloaded (we have measured bandwidths as low as 400 MB/s). On every compute node, the 4 disks were configured as RAID-0 (striping)[10]. Each node contains 4 Seagate Barracuda 7200.10 hard drives with a capacity of 250 GB each. We have measured peak I/O rates between 60 and 71 MiB/s, in average 67 MiB/s, on an XFS file system. If not stated otherwise, we used a block size of 8 MiB.

We tested scalability by sorting 100 GiB of data per PE, with the number of PEs increasing up to 64. The element size is (only) 16 bytes with 64-bit keys. This makes internal computation efficiency as important as high I/O throughput. As shown in Figure 7.5, the scalability is very good for random input data. For the bad input, a penalty of up to 50% in running time can appear (Figure 7.6), as expected by the additional I/O performed by the all-to-all phase. This overhead can be diminished by using randomization (Figure 7.7), which reduces the I/O penalty greatly. A smaller block size of 2 MiB can further improve the effect of randomization, but at the cost of a little worse I/O performance in practice.

As expected, run formation takes about the same time as the final merging. We are almost I/O-bound, thanks to the shared-memory parallel sorter from MCSTL. The average I/O bandwidth per disk is about 44 MiB/s, which is more than two thirds of the maximum. The reasons for the overhead are worse performance of tracks close to the center of a disk (when disks fill up), file system overhead, natural spreading of disk performance, and startup/finalization overhead. Multiway selection in fact takes only negligible time.

Figure 7.8 shows the time consumption across the nodes for a 32-node run. The work is very well balanced, but there is some variance in disk speed.

We have not compared DEMSort to implementations of other algorithms directly. However, we have made experiments on the well-established Sort Benchmark, which was initiated by Jim Gray in 1984 and is continuously adapted to the changing circumstances [Sor]. This setting considers 100-byte elements with a 10-byte key. Our results [RSSK09] using 195 nodes show that we can sort 1 TB in less than 64 seconds, which is about a third of the time needed by the 2007 winner TokuSampleSort. This is despite the fact the we use the same number of cores[11], but only a third of the hard disks. We also slightly improve on a recent result for the Terabyte category published informally by Google [Cza], where 12 000 disks[12] were used instead of 780 as in our case. This shows the superiority of our approach compared to the MapReduce framework [DG08].

[10] Parallel disks are also directly supported by the program and could lead to even better timings, but we could not configure the machine accordingly at the time of testing.

[11] For such large elements, our program is not compute-bound when using all cores.

[12] Google used 3-fold redundancy, but still, at least the performance of 4 000 disks could be achieved. Also, for a machine like ours, redundancy is not that desperately needed.

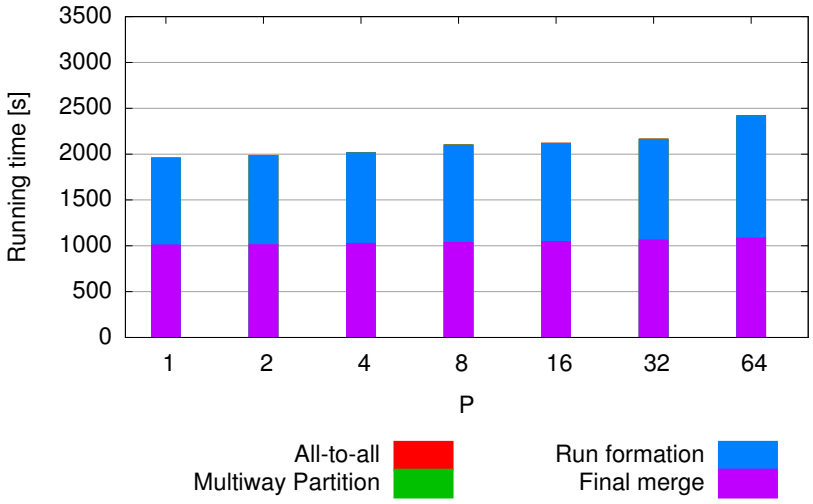


Figure 7.5: Running times for random input, split up by the phases of the algorithm. Please note that the order of the phases is different from the order in the algorithm, to allow for better visual comparison.

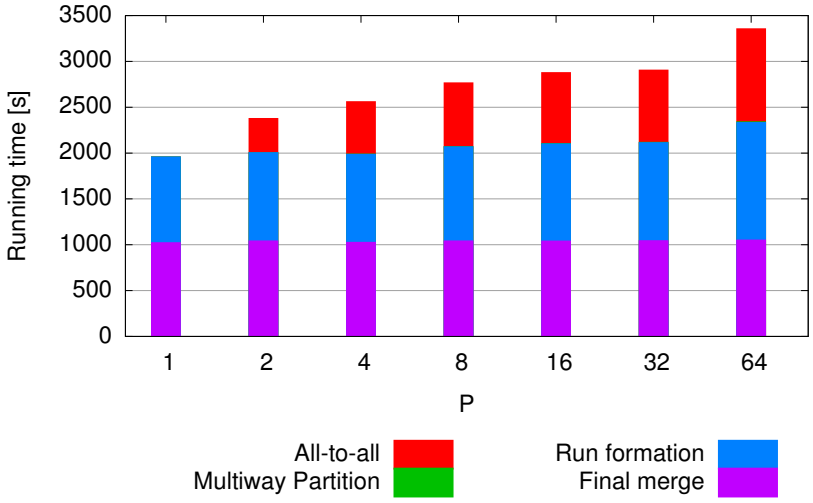


Figure 7.6: Running times for worst-case input to CANONICALMERGESORT *without* randomization applied.

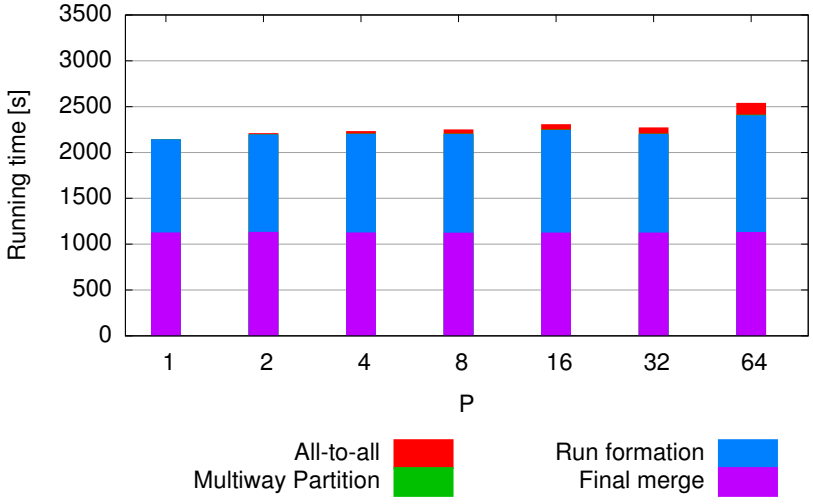


Figure 7.7: Running times for worst-case input to CANONICALMERGESORT *with* randomization applied.

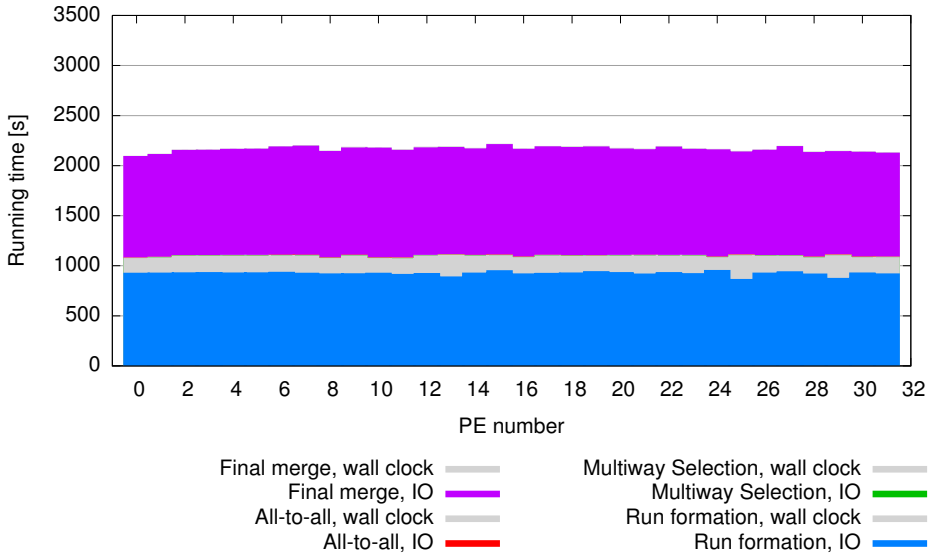


Figure 7.8: Running times of the different phases on 32 nodes for random input. For each phase, both the I/O time and the wall clock time are shown. If there is a gray gap, the phase is not fully I/O-bound, as is the case for the run formation.

In the MinuteSort category, a time limit of one minute is given, the processed amount of the data is the metric. We have beaten the former record of TokuSampleSort by a factor of 3.6, processing 955 GB of data, which made the committee retire the TerabyteSort category. Yahoo achieved a result half as high using the Hadoop framework [OM, Had], but with a machine 7 times as large.

However, for the results mentioned so far, $N < M$, so the sort is merely internal and only 2 I/Os per block of elements are needed.

Concerning the newly established GraySort category, we can sort 100 TB in slightly less than three hours on 195 nodes, resulting in 564 GB Bytes of sorted data per minute. The Google program in this case takes only twice the time for ten times the amount of data, but they use an even larger machine than before, featuring 48 000 disks, which is a factor of 61 larger. The better performance of a factor of 5 is thus reduced to less than 0.1 in terms of relative efficiency. Yahoo's official result of 578 GB/min is only 2.5% faster than us, but its efficiency is even worse, since they used 17 times the number of nodes. Those nodes were very similar to the ones used by us, except having only half the memory. Their worse communication bandwidth would not have been a limiting factor for our algorithm.

In 2010, we also proved the energy efficiency of CANONICALMERGESORT, competing with a variant of DEMSort [BMSS10a, BMSS10b] in JouleSort, which measures the number of sorted records per energy. For the JouleSort 10^{10} record category of the Sort Benchmark, we used only a single, but highly energy-efficient machine, so the distributed memory capabilities remained unused. However, we utilized the unique in-place sorting capability of DEMSort to sort 1 TB with on 1024 GB of hard disk space. The program sorts 1 TB of data using only a

fifth of the energy of the former record holder, processing about 17 500 records per Joule. The results on the cluster make us presume that combining many of the energy-efficient machines would scale well, while increasing the energy per sorted record by at most a factor of two. This would still be almost an order of magnitude better than on the machine used for 100 TB, which we estimate to sort about 1000 records per Joule.

7.6 Conclusions and Future Work

We explored some of the design space for merging-based parallel external sorting, proposing two new algorithms based on multiway mergesort. Our globally striped algorithm minimizes the required I/Os. Our CANONICALMERGESORT algorithm is theoretically a bit less scalable but has close to minimal communication overhead, and a more useful output format. Moreover, it sorts any technologically reasonable inputs in two to three passes. For medium-sized machines or average case inputs, the I/O requirement remains closer to two passes.

As of 2009, the implementation of CANONICALMERGESORT, DEMSort, holds several world records in sorting huge data sets, being more efficient than the competitors by an order of magnitude. A recent paper [AT10] actually complains about the bad I/O efficiency of current *data intensive scalable computing (DISC)* system, DEMSort being the notable exception.

As mentioned, using the implementation described, we can sort a huge sequence of data stored in one STXXL vector per node. Simply add a distributed external memory scanning operation, and we have the major building blocks for many external memory algorithms, which could then easily be transferred to distributed memory as well. This is thus a first step towards a distributed memory version of the STXXL.

Our first idea for an algorithm was actually to create node-local runs first, and then to merge them in parallel. However, with the bad input mentioned for CANONICALMERGESORT, very strong I/O imbalance might have occurred, e. g. all PEs reading data from the disks of a single PE in a period of time.

Run formation could perhaps be improved to allow longer runs [Knu98, Section 5.4.1]. The main effect is that by decreasing the number of runs, we can further increase the block size. For the very largest inputs this could yield a slight improvement in performance. On large machines that have considerably higher communication bandwidth than I/O bandwidth, the globally striped algorithm could indeed be faster. However, there is still the uncommon distribution of the result. This algorithm could also be useful for pipelined sorting where the run formation does not *fetch* the data but obtains it from some data generator (no randomization possible for CANONICALMERGESORT) and where the output is not *written* to disk but fed into a postprocessor that requires its input in sorted order (e. g., variants of Kruskal's algorithm as regarded in Section 4.6.2). When scaling to very large machines, fault tolerance will play a bigger role. An interesting question is whether this can be achieved with lower overhead than in [Cza].

This work was initially published in [RSS10] and [RSS09].

8 Conclusions

In this thesis, we have investigated and executed the parallelization of established algorithmic C++ libraries, for the sake of easy exploitation of multiple CPU cores. This includes the STL, the STXXL, and CGAL. We have selected and engineered the parallel algorithms to be included, incorporating the specific properties of the library setting.

The achieved acceleration varies, depending on the algorithm, the input size, the data type, and developer-defined functors. In some cases, we have demonstrated that the performance is limited by the available memory bandwidth. For the other cases, the parallel efficiency is often very good for up to 4 cores, which is a common multi-core configuration for desktop machines nowadays. For 5 to 8 cores, the efficiency usually declines, but execution still gets faster.

For the external memory setting, we have reduced the gap between the increasing I/O bandwidth and the stagnating CPU clock rates. We have also made use of the libraries in case studies like minimum spanning tree and suffix array construction. These examples demonstrate the effectiveness of exploiting multi-core processors by using parallelized algorithm libraries.

Furthermore, we have developed new sorting algorithms for huge data sets in the multi-terabyte range, aimed for distributed memory machines with very many hard disks. An implementation utilizing both the MCSTL and the STXXL has beaten the former world records, and is an order of magnitude more efficient than the runner-up. This shows that absolute performance is also very competitive.

8.1 Impact

The results of this thesis have been published in major conferences (Euro-Par, IPDPS, SoCG, ICDE) and a journal (Computational Geometry – Theory and Applications).

The first publications, about the MCSTL, have lead to the cooperations with the Goethe University Frankfurt am Main (STXXL), the Universitat Politècnica de Catalunya (MCSTL data structures) and the INRIA Sophia Antipolis (CGAL). A student from Carleton University Ottawa wrote a thesis about heap operations for the MCSTL.

The developed software is already in use in both industry and academia. If you use a recent version of Linux, it is actually quite likely that the parallel mode of libstdc++, based on the MCSTL, is already installed on your computer as part of the GNU C++ compiler. The fact alone that it was included into this very important compiler shows general interest in this functionality. Also, the MCSTL has been mentioned by Germany's most renowned IT magazine [Lau08]. We have received support requests on the parallel mode from institutions like the Department of Mathematics at Texas A&M University, the Berlin-Brandenburgische Akademie der Wissenschaften, which maintains the digital dictionary of the German language of the 20th century, and IBM Dublin Software Labs. The Broad Institute at the Massachusetts

Institute of Technology, which is concerned with bioinformatics, has successfully used the parallel mode on a machine with 32 sockets and 512 GB of RAM.

The STXXL had a quite strong acadmic user base already before [Dem06, p. 12], and we know of at least the Interactive Systems Lab at the Karlsruhe Institute of Technology that uses the parallelized version.

CGAL has a large user base in both academia [CGAc] and industry [CGAb]. The 3D Delaunay package is particularly popular. Industry customers include the telecom and the oil business. The parallelized CGAL routines will become part of a future CGAL release.

8.2 Future Work

Some of the algorithms presented contain tuning parameters like block sizes. A parameter which is general to all algorithms is the minimum input that triggers parallel execution. Overestimation is conservative, since it prevents slowdown (which can be dramatic), but also limits parallelization, of course. Estimated values based on experience are helpful here, but usually limited to a usual case. By using custom data types, custom functors, and operator overloading, users can change the circumstances dramatically.

Thus, more automatic tuning is desirable. Crucial for that is some kind of performance prediction, which has to rely on running time measurement full algorithm executions. Evaluation of single calls of a user-defined functor are difficult due to lacking data, possible side effects, or

As stated in the introduction, thread-safety of containers is not our concern here. However, some of the algorithms that are closely tied to a container type, usually as a member function, have not been treated. The work could thus be extended in this direction.

One of the most urgent remaining problems is the decision of when to switch to parallel execution. This is crucial for the resulting program performance in the worst case, since the parallelization overhead cancels out speedups for “to easy” problems, i. e. algorithm calls that have to little or too local data for communication of the data being worthwhile. By testing and profiling, the application developer can discover break-even points and tuning parameters for a specific algorithm on a specific data type, a specific functors, and so on. However, this is tedious and does not lend to deployment on several platforms at the same time with the same code. As a result, developing methods for auto-tuning the algorithms are underway. Even with the noted deficiencies, the parallel mode user can already work at a higher level of abstraction and performance.

8.3 Acknowledgments

I thank my advisor Peter Sanders for support of this thesis, and his high availability to discussion and questions.

I also owe a lot to the others collaborators on the papers this work is based on. Felix Putze helped implementing the first versions of the MCSTL. In close cooperation with Leonor Frias, the dictionary bulk construction and the list partitioning projects were tackled. The MCSTL integration into GCC was made possible by Benjamin Kosnik, he participated in

solving the software engineering issues. Ulrich Drepper, Paolo Carlini, Jakub Jelinek, and Wolfgang Bangerth also supported the GCC integration, I thank them for valuable discussion and pointing out specific problems. Jakob Blomer improved the OpenMP implementation of GCC in his diploma thesis. Vitaly Osipov developed the minimum spanning trees algorithm utilizing MCSTL, while Manuel Holtgrewe wrote the internal memory suffix array program, and evaluated the MCSTL load balancing in the context of route planning in his Studienarbeit. Andreas Beckmann and Roman Dementiev were involved in the STXXL work. Vicente Helano, David Millman, Sylvain Pion belonged to the team for parallel geometry. Mirko Rahn was heavily involved in the distributed external memory sort project, whose software was in turn based on the Studienarbeit by Tim Kieritz. The participants of the lab course on MCSTL wrote initial versions of the `unique_copy` routine. I thank Frederik Transier for the introduction to the graduating process, and Ali Jannesari for giving us the opportunity to test programs on the Sun T1.

My gratitude extends to all current and former colleagues of the group "Algorithmik II" of the Institute for Theoretical Computer Science.

This work was partially supported by the DFG priority programme 1307 "Algorithm Engineering", project "Algorithm Engineering for the Basic Toolbox", DFG grants 933/3-1 and 933/3-2.

Bibliography

[ABT04] Lars Arge, Gerth S. Brodal, and Laura Toma. On external-memory MST, SSSP and multi-way planar graph separation. *Journal of Algorithms*, 53, 2004.

[ACR03] Nina Amenta, Sunghee Choi, and Günter Rote. Incremental constructions con brio. In *19th Symposium on Computational Geometry (SoCG)*, pages 211–219, 2003.

[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.

[AGNS08] Lars Arge, Michael T. Goodrich, Michael J. Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 197–206, 2008.

[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. `http://parasol.tamu.edu/groups/rwergergroup/research/stapl/`.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Spring Joint Computer Conference (SJCC)*, 1967.

[AP94] A. Aggarwal and C. G. Plaxton. Optimal parallel sorting in multi-level storage. In *5th ACM-SIAM Symposium on Discrete Algorithms*, pages 659–667, 1994.

[Arg95] Lars Arge. The Buffer Tree: A New Technique for Optimal I/O-Algorithms. In *4th Workshop on Algorithms and Data Structures (WADS)*, pages 334–345, 1995.

[AT10] Eric Anderson and Joseph Tucek. Efficiency matters! *ACM SIGOPS Operating Systems Review*, 44(1):40–45, 2010.

[AV88] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[Bae06] D. Baertschiger. Multi-processing template library. Master thesis, Université de Genève (in French), `http://spc.unige.ch/mptl`, 2006.

[BBF+96] P. Bach, M. Braun, A. Formella, J. Friedrich, Th. Grün, H. Leister, C. Lichtenau, Th. Walle, Jörg Friedrich, Thomas Grun, and Holger Leister. Building the 4 processor SB-PRAM prototype. In *Hawaii 30th International Symposium on System Science (HICSS)*, pages 14–23. IEEE Computer Society Press, 1996.

[BBK06] Daniel K. Blandford, Guy E. Blelloch, and Clemens Kadow. Engineering a compact parallel Delaunay algorithm in 3D. In *22nd Symposium on Computational Geometry (SoCG)*, pages 292–300, 2006.

[BDP+02] Jean-Daniel Boissonnat, Olivier Devillers, Sylvain Pion, Monique Teillaud, and Mariette Yvinec. Triangulations in CGAL. *Compututational Geometry – Theory and Applications*, 22:5–19, 2002.

[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.

[BGV97] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):601–631, 1997.

[BL99] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[Blo08] Jakob Blomer. Effizientes Starten und Verteilen von Threads auf Mehrkern-Prozessoren. Diplomarbeit, Universität Karlsruhe, 2008. `http://algo2.iti.kit.edu/documents/EffizientesStartenVerteilenThreadsBlomer2008.pdf`.

[BMPS09] Vicente H. F. Batista, David L. Millman, Sylvain Pion, and Johannes Singler. Parallel geometric algorithms for multi-core computers. In *25th Annual Symposium on Computational Geometry (SoCG)*, pages 217–226, 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.

[BMSS10a] Andreas Beckmann, Ulrich Meyer, Peter Sanders, and Johannes Singler. EcoSort, January 2010. http://sortbenchmark.org/ecosort_2010_Jan_01.pdf.

[BMSS10b] Andreas Beckmann, Ulrich Meyer, Peter Sanders, and Johannes Singler. Energy-efficient sorting using solid state disks. In *International Green Computing Conference*, 2010. submitted.

[Bow81] A. Bowyer. Computing dirichlet tessellations. *The Computer Journal*, 24(2):162, 1981.

[BST+08] Antal A. Buss, Timmie G. Smith, Gabriel Tanase, Nathan L. Thomas, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. Design for interoperability in stapl: pmatrices and linear algebra algorithms. In *21th International Workshops on Languages and Compilers for Parallel Computing (LCPC)*, pages 304–315, 2008.

[C++03] *The C++ Standard (ISO 14882:2003)*. 2003.

[C++10] Working draft, standard for programming language C++, March 2010. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf`.

[CC06] G. Chaudhry and T. H. Cormen. Slabpose columnsort: A new oblivious algorithm for out-of-core sorting on distributed-memory clusters. *Algorithmica*, 45:483–508, 2006.

[CD04] Thomas H. Cormen and Elena Riccio Davidson. FG: A framework generator for hiding latency in parallel programs running on clusters. In *17th International Conference on Parallel and Distributed Computing Systems (ISCA PDCS)*, pages 137–144, 2004.

[CGAa] The CGAL philosophy. `http://www.cgal.org/philosophy.html`.

[CGAb] GeometryFactory customers. `http://geometryfactory.com/aboutus.html`.

[CGAc] Projects using CGAL. `http://www.cgal.org/projects.html`.

[cgad] CGAL, Computational Geometry Algorithms Library. `http://www.cgal.org/`.

[CGG+95] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamasia, D. E. Vengroff, and J. S. Vitter. External memory graph algorithms. In *6th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 139–149, 1995.

[Cho80] Anita Liu Chow. *Parallel algorithms for geometric problems*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1980.

[CMPS93] P. Cignoni, C. Montani, R. Perego, and R. Scopigno. Parallel 3D Delaunay triangulation. *Computer Graphics Forum*, 12(3):129–142, 1993.

[CN03] Nikos Chrisochoides and Démian Nave. Parallel Delaunay mesh generation kernel. *International Journal for Numerical Methods in Engineering*, 58(2):161–176, 2003.

[Cza] Grzegorz Czajkowski. Sorting 1PB with MapReduce. http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html.

[dBvKOS00] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2000.

[DC05] Elena Riccio Davidson and Thomas H. Cormen. Building on a framework: Using FG for more flexibility and improved performance in parallel programs. In *19th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2005.

[DDHM02] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Bulk synchronous parallel algorithms for the external memory model. *Theory of Computing Systems*, 35:567–597, 2002.

[Del08] Christophe Delage. Spatial sorting. In CGAL Editorial Board, editor, *CGAL Manual*. 3.4 edition, 2008. http://www.cgal.org/Manual/3.4/doc_html/cgal_manual/packages.html#Pkg:SpatialSorting.

[Dem06] Roman Dementiev. *Algorithm Engineering for Large Data Sets*. PhD thesis, Universität des Saarlandes, 2006.

[DFRC93] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. In *9th Symposium on Computational Geometry (SoCG)*, pages 298–307, 1993.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DKMS05] Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, and Peter Sanders. Better external memory suffix array construction. In *7th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 86–97, 2005.

[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.

[DS03] R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *15th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 138–148, 2003.

[Fly72] Micahel Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972.

[FM87] R. Finkel and U. Manber. DIB – A distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, 1987.

[FP08] Leonor Frias and Jordi Petit. Parallel partition revisited. In *7th International Workshop on Experimental Algorithms (WEA)*, pages 142–153, 2008.

[FS07] Leonor Frias and Johannes Singler. Parallelization of bulk operations for STL dictionaries. In *Workshop on Highly Parallel Processing on a Chip (HPPC)*, number 4854 in LNCS, pages 49–59, 2007.

[FSS08a] Leonor Frias, Johannes Singler, and Peter Sanders. Single-pass list partitioning. In *International Workshop on Multi-Core Computing Systems (MuCoCoS)*, 2008.

[FSS08b] Leonor Frias, Johannes Singler, and Peter Sanders. Single-pass list partitioning. *Scalable Computing: Practice and Experience*, 9(3):179–184, 2008.

[GS78] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 8–21, 1978.

[Had] The Hadoop project website. `http://hadoop.apache.org/`.

[Har94] Tim J. Harris. A survey of PRAM simulation techniques. *ACM Computing Surveys*, 26(2):187–206, 1994.

[HKPW09] Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Ron Wein. STL extensions for CGAL. In *CGAL Manual*. 2009. `http://www.cgal.org/Manual/3.4/doc_html/cgal_manual/packages.html#Pkg:StlExtension`.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[Hol08] Manuel Holtgrewe. Parallel highway-node routing. Studienarbeit, Universität Karlsruhe, 2008. `http://algo2.iti.kit.edu/924.php`.

[HSV05] D. A. Hutchinson, P. Sanders, and J. S. Vitter. Duality between prefetching and queued writing with parallel disks. *SIAM Journal on Computing*, 34(6):1443–1463, 2005.

[Int] Intel Threading Building Blocks website. `http://osstbb.intel.com/`.

[JÓ2] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[KCP06] Milind Kulkarni, L. Paul Chew, and Keshav Pingali. Using transactions in Delaunay mesh generation. In *1st Workshop on Transactional Memory Workloads*, 2006.

[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.

[KMZ08] Lutz Kettner, Andreas Meyer, and Afra Zomorodian. Intersecting sequences of dD iso-oriented boxes. In CGAL Editorial Board, editor, *CGAL Manual*. 3.4 edition, 2008. `http://www.cgal.org/Manual/3.4/doc_html/cgal_manual/packages.html#Pkg:BoxIntersectionD`.

[Knu98] D. E. Knuth. *The Art of Computer Programming—Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.

[KS07] Fabian Kulla and Peter Sanders. Scalable parallel suffix array construction. *Parallel Computing*, 33(9):605–612, 2007.

[KSB06] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006.

[Lau08] Oliver Lau. GCC wird paralleler. *c't: Magazin für Computer Technik*, 5:69, 2008.

[LPP01] Sangyoon Lee, Chan-Ik Park, and Chan-Mo Park. An improved parallel algorithm for Delaunay triangulation on distributed memory parallel computers. *Parallel Processing Letters*, 11(2/3):341–352, 2001.

[Mal10] Sven Mallach. Beschleunigung ausgewählter paralleler standard template library algorithmen. Diplomarbeit, Technische Universität Dortmund, 2010. `http://www.zaik.uni-koeln.de/%7Epaper/preprints.html?show=zaik2010-599`.

[McC07] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[McD89] C. J. H. McDiarmid. On the method of bounded differences. *Surveys in Combinatorics*, pages 148–188, 1989.

[MN98] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8:3–30, 1998.

[MPI94] MPI Forum. MPI: A message-passing interface standard. Technical report, University of Tennessee, 1994.

[MRL98] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. *SIGMOD Records*, 27(2):426–435, 1998.

[Mus97] David R. Musser. Introspective sorting and selection algorithms. *Software—Practice and Experience*, 27(8):983–993, 1997.

[NS09] Stefan Näher and Daniel Schmitt. Multi-core implementations of geometric algorithms. In *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of his 60th Birthday*, pages 261–274, 2009.

[NV93] M. H. Nodine and J. S. Vitter. Deterministic Distribution Sort in Shared and Distributed Memory Multiprocessors. In *5th ACM Symposium on Parallel Algorithms and Architectures*, pages 120–129, Velen, Germany, 1993.

[NV95] M. H. Nodine and J. S. Vitter. Greed sort: An optimal sorting algorithm for multiple disks. *Journal of the ACM*, 42(4):919–933, 1995.

[OM] Owen O'Malley and Arun C. Murthy. Winning a 60 second dash with a yellow elephant. `http://sortbenchmark.org/Yahoo2009.pdf`.

[OP97] T. Okusanya and J. Peraire. 3D parallel unstructured mesh generation. In *Trends in Unstructured Mesh Generation*, volume 220 of *Applied Mechanics Division*, pages 109–115. 1997.

[Ope08] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*, May 2008. `http://www.openmp.org/mp-documents/spec30.pdf`.

[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.

[PP01] Heejin Park and Kunsoo Park. Parallel algorithms for red-black trees. *Theoretical Computer Science*, 262:415–435, 2001.

[PSLM00] P. J. Plauger, Alexander A. Stepanov, Meng Lee, and David R. Musser. *The C++ Standard Template Library*. Prentice-Hall, 2000.

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

[Raj04] Sanguthevar Rajasekaran. Out-of-core computing on mesh connected computers. *Journal of Parallel and Distributed Computing*, 64(11):1311–1317, November 2004.

[Ric94] Ivan L. M. Ricarte. *Performance and Scalability of Parallel Database Systems*. PhD thesis, University of Maryland College Park, College Park, MD, USA, 1994.

[RKU00] A. Ranade, S. Kothari, and R. Udupa. Register Efficient Mergesorting. In *High Performance Computing*, pages 96–103, 2000.

[Rob08] Dave E. Robillard. Parallel C++ STL heap building, 2008. `http://drobilla.net/comp5704/Final_Paper.pdf`.

[Rob10] David E. Robillard. Practical parallel external memory algorithms via simulation of parallel algorithms. *CoRR*, abs/1001.3364, 2010.

[Ros08] P. E. Ross. Why CPU frequency stalled. *IEEE Spectrum*, 45(4):72 –72, 2008.

[RSS09] Mirko Rahn, Peter Sanders, and Johannes Singler. Scalable distributed-memory external sorting. *CoRR*, abs/0910.2582, 2009.

[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.

[RSSK09] Mirko Rahn, Peter Sanders, Johannes Singler, and Tim Kieritz. DEMSort – distributed external memory sort, 2009. http://sortbenchmark.org/demsort.pdf.

[San98a] P. Sanders. Random permutations on distributed, external and hierarchical memory. *Information Processing Letters*, 67(6):305–310, 1998.

[San98b] P. Sanders. Tree shaped computations as a model for parallel applications. In *Workshop on Application Based Load Balancing (ALV)*, 1998.

[San00] Peter Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.

[San02] Peter Sanders. Randomized Receiver Initiated Load Balancing Algorithms for Tree Shaped Computations. *The Computer Journal*, 45(5):561–573, 2002.

[San09] Peter Sanders. Algorithm engineering – an attempt at a definition. In *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, pages 321–340, 2009.

[Sch00] Stefan Schirra. Robustness and Precision Issues in Geometric Computation. In *Handbook of Computational Geometry*, pages 597–632. Elsevier, 2000.

[SK08] Johannes Singler and Benjamin Kosnik. The libstdc++ parallel mode: Software engineering considerations. In *International Workshop on Multicore Software Engineering (IWMSE)*, 2008.

[Sor] Sort Benchmark home page. `http://sortbenchmark.org/`.

[SSDM07] Michael L. Scott, Michael F. Spear, Luke Dalessandro, and Virendra J. Marathe. Delaunay triangulation with transactions and barriers. In *10th International Symposium on Workload Characterization*, pages 107–113, 2007.

[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.

[TRM+08] Daouda Traore, Jean-Louis Roch, Nicolas Maillard, Thierry Gautier, and Julien Bernard. Deque-free work-optimal parallel STL algorithms. In Emilio Luque, Tomas Margalef, and Domingo Benitez, editors, *14th International Euro-Par Conference*, pages 887–897, 2008.

[TTT+05] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 277–288, 2005.

[TZ03] Philippas Tsigas and Yi Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000. In *11th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, page 372, 2003.

[Val90] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

[Val08] Leslie G. Valiant. A bridging model for multi-core computing. In *16th European Symposium on Algorithms (ESA)*, pages 13–28, 2008.

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

[VSIR91] Peter J. Varman, Scott D. Scheufler, Balakrishna R. Iyer, and Gary R. Ricard. Merging Multiple Lists on Hierarchical-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):171–177, 1991.

[Wat81] D. F. Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *Computer Journal*, 24(2):167–172, 1981.

[WV07] Xingzhi Wen and Uzi Vishkin. PRAM-on-chip: First commitment to silicon. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2007.

[ZE02] Afra Zomorodian and Herbert Edelsbrunner. Fast software for box intersections. *International Journal of Computational Geometry Applications*, 12(1-2):143–172, 2002.